Efficient finite-state algorithms for the application of local grammars

Javier Miguel Sastre Martinez

Ph.D. Thesis
Efficient finite-state algorithms for the application of local grammars

Thèse de Doctorat
Algorithmes performants à états finis pour l'application de grammaires locales

Tesis Doctoral
Algoritmos eficientes de estados finitos para la aplicación de gramáticas locales

Javier M. Sastre-Martínez

Programme de Doctorat en
Linguistique-Informatique

Programa de Doctorado en
Aplicaciones de la Informática

UNIVERSITÉ —— PARIS-EST

Universitat d'Alacant
Universidad de Alicante

Ph.D. Thesis

presented by
Javier M. Sastre-Martínez

to obtain the degree of
Ph.D. in Computational Linguistics

about the subject
Efficient finite-state algorithms for the application of local grammars

Ph.D. advisors
Mikel L. Forcada
Éric Laporte

defended on 11th July 2011 upon the tribunal composed by:
Olivier Carton (reviewer)
Mikel L. Forcada (advisor)
Éric Laporte (advisor)
Joan A. Sánchez (reviewer)

a Anna y a mis padres

Acknowledgements

I thank Prof. Mikel L. Forcada and Prof. Éric Laporte for giving me the chance to realize this doctoral work and for their supervision. In particular, I thank Prof. Mikel L. Forcada for his discussions on finite-state machines and algorithms, and Prof. Éric Laporte for his teachings on the local grammar approach. I thank Prof. Olivier Carton and Dr. Joan A. Sánchez for having accepted to be members of the examination tribunal of this dissertation. I thank Prof. Antal van den Bosch and Prof. Zygmunt Vetulani for having accepted to review this dissertation. I thank the Unitex and Outilex authors, Dr. Sébastien Paumier and Dr. Olivier Blanc, for solving my doubts about their respective systems. I also thank Telefónica's personnel, Joaquín M. López and Javier García, for giving me the chance to apply the algorithms I have developed throughout this work to their commercial chatterbot, the MovistarBot, and for their collaborative work. I thank Dr. Jorge Sastre for introducing me to Telefónica I+D, for supervising the tasks concerning the application of this work to the MovistarBot, and for his constant support, encouragement and optimism.

I thank Teresa Gómez and Eun-Jin Jung for the so many pleasant mo-

ments I have spent with them during my stay in France, and for their constant support and encouragement.

To my family, and specially to my wife.

Abstract

This work focuses on the research and development of efficient algorithms of application of local grammars (Gross, 1997), taking as reference those of the currently existent open-source systems: Unitex's top-down parser (Paumier et al., 2009) and Outilex's Earley-like parser (Blanc and Constant, 2006a).

Local grammars are a finite-state based formalism for the representation of natural language grammars. Moreover, local grammars are a model for the construction of fully scaled and accurate descriptions of the syntax of natural languages by means of systematic observation and methodical accumulation of data. The adequacy of local grammars for this task has been proved by multiple works (Roche and Schabes, 1997; Català and Baptista, 2007; Martineau et al., 2007; Laporte et al., 2008a,b).

Due to the ambiguous nature of natural languages, and the particular properties of local grammars, classic parsing algorithms such as LR (Knuth, 1965), CYK's (Cocke and Schwartz, 1970; Younger, 1967; Kasami, 1965) and Tomita's (1987) are either not viable in the context of this work or require non-trivial adaptations. Top-down and Earley parsers are possible alternatives, though they have an exponential worst-case cost for the case of local grammars.

We have first conceived an algorithm of application of local grammars having a polynomial worst-case cost (Sastre, 2009). Furthermore, we have conceived other optimizations which increase the efficiency of the algorithm for general cases, namely the efficient management of sets of elements and sequences. We have implemented our algorithm and those of the Unitex and Outilex systems with the same tools in order to test them under the same conditions. Moreover, we have implemented different versions of each algorithm, using either our custom set data structures or those included in GNU's implementation of the C++ Standard Template Library (STL).[1] We

---

[1] A detailed description of the STL can be found in Josuttis (1999). GNU's implemen-

iv

have compared the performance of the different algorithms and algorithm versions in the context of an industrial natural language application provided by the enterprise Telefónica I+D:[2] extending the understanding capabilities of a chatterbot that provides mobile services, such as sending SMSs to mobile phones as well as games and other digital contents (Sastre et al., 2009). Conversation with the chatterbot is held in Spanish by means of Microsoft's Windows Live Messenger.[3] In spite of the limited domain and the simplicity of the applied grammars, execution times of our parsing algorithm coupled with our custom implementation of sets were lower. Thanks to the improved asymptotic cost of our algorithm, execution times for the case of complex and large coverage grammars can be expected to be considerably lower than those of the Unitex and Outilex algorithms.

---

tation of this library is being distributed along with GNU's Compiler Collection: http://gcc.gnu.org/

[2]http://www.tid.es/en

[3]http://www.msn.com

Résumé

Notre travail porte sur le développement d'algorithmes performants d'application de grammaires locales (Gross, 1997), en prenant comme référence ceux des logiciels libres existants: l'analyseur syntaxique descendant d'Unitex (Paumier et al., 2009) et l'analyseur syntaxique à la Earley d'Outilex (Blanc and Constant, 2006a).

Les grammaires locales sont un formalisme de représentation de la syntaxe des langues naturelles basé sur les automates finis. Les grammaires locales sont un modèle de construction de descriptions précises et à grande échelle de la syntaxe des langues naturelles par le biais de l'observation systématique et l'accumulation méthodique de données. L'adéquation des grammaires locales pour cette tâche a été testé à l'occasion de nombreux travaux (Roche and Schabes, 1997; Català and Baptista, 2007; Martineau et al., 2007; Laporte et al., 2008a,b).

À cause de la nature ambiguë des langues naturelles et des propriétés des grammaires locales, les algorithmes classiques d'analyse syntaxique tels que LR (Knuth, 1965), CYK (Cocke and Schwartz, 1970; Younger, 1967; Kasami, 1965) et Tomita (1987) ne peuvent pas être utilisés dans le contexte de ce travail ou ont besoin d'adaptations non triviaux. Les analyseurs *top-down* et Earley sont des alternatives possibles ; cependant, ils ont des coûts asymptotiques exponentiels pour le cas des grammaires locales.

Nous avons d'abord conçu un algorithme d'application de grammaires locales avec un coût polynomial dans le pire des cas (Sastre, 2009). Ensuite, nous avons conçu des structures de donnés performantes pour la représentation d'ensembles d'éléments et de séquences. Elles ont permis d'améliorer la vitesse de notre algorithme dans le cas général. Nous avons mis en œuvre notre algorithme et ceux des systèmes Unitex et Outilex avec les mêmes outils afin de les tester dans les mêmes conditions. En outre, nous avons mis en œuvre différentes versions de chaque algorithme en utilisant nos structures

de données et algorithmes pour la représentation d'ensembles et ceux fournis par la *Standard Template Library* (STL) de GNU.[4] Nous avons comparé les performances des différents algorithmes et de leurs variantes dans le cadre d'un projet industriel proposé par l'entreprise Telefónica I+D : [5] augmenter la capacité de compréhension d'un agent conversationnel qui fournit des services en ligne, voire l'envoi de SMS à des téléphones portables ainsi que des jeux et d'autres contenus numériques (Sastre et al., 2009). Les conversations avec l'agent sont en espagnol et passent par Windows Live Messenger.[6] En dépit du domaine limité et de la simplicité des grammaires appliquées, les temps d'exécution de notre algorithme, couplé avec nos structures de données et algorithmes pour la représentation d'ensembles, ont été plus courts. Grâce au coût asymptotique amélioré, on peut s'attendre à des temps d'exécution significativement inférieurs par rapport aux algorithmes utilisés dans les systèmes Unitex et Outilex, pour le cas des grammaires complexes et à large couverture.

---

[4]Voire Josuttis (1999) pour une description détaillée de la STL. La STL de GNU fait partie de la *GNU's Compiler Collection*: http://gcc.gnu.org/

[5]http://www.tid.es/en

[6]http://www.msn.com

Resumen

Este trabajo se centra en la investigación y el desarrollo de algoritmos eficientes de aplicación de gramáticas locales (Gross, 1997), tomando como referencia aquellos que están siendo usados en sistemas *open-source*, a saber: el analizador sintáctico *top-down* de Unitex (Paumier et al., 2009) y el analizador sintáctico *à la Earley* de Outilex (Blanc and Constant, 2006a).

Las gramáticas locales son un formalismo basado en autómatas finitos para la representación de la sintaxis de los lenguajes naturales. Las gramáticas locales son un modelo de construcción de descripciones precisas y a gran escala de la sintaxis de los lenguajes naturales mediante la observación sistemática y la acumulación metodológica de información. La idoneidad de las gramáticas locales para esta tarea ha sido demostrada por múltiples trabajos (Roche and Schabes, 1997; Català and Baptista, 2007; Martineau et al., 2007; Laporte et al., 2008a,b).

Debido a la naturaleza ambigua de la lengua, y a las propiedades de las gramáticas locales, los analizadores sintácticos clásicos tales como LR (Knuth, 1965), el de CYK (Cocke and Schwartz, 1970; Younger, 1967; Kasami, 1965) y el de Tomita (1987) no son viables en el contexto de este trabajo o requieren adaptaciones no triviales. Los analizadores sintácticos *top-down* y de Earley son posibles alternativas, aunque tienen un coste asintótico exponencial en el caso de las gramáticas locales.

En primer lugar, hemos desarrollado un algoritmo de aplicación de gramáticas locales con un coste asintótico polinomial (Sastre, 2009). A continuación, hemos desarrollado estructuras de datos eficientes para la gestión de conjuntos de elementos y de secuencias. Estas estructuras han permitido mejorar la eficiencia de nuestro algoritmo en condiciones generales. Hemos implementado dicho algoritmo y los algoritmos de Unitex y Outilex con las mismas herramientas con el fin de compararlos bajo las mismas condiciones. Hemos implementado distintas versiones de cada algoritmo usando nuestras

estructuras de datos de tipo conjunto y aquellas incluidas en la implementación de GNU de la librería estándar de plantillas (*Standard Template Library* o STL).[7] Hemos comparado el rendimiento de los distintos algoritmos y de sus distintas versiones en el contexto de una aplicación industrial propuesta por la empresa Telefónica I+D: [8] aumentar la capacidad de comprensión de un robot conversacional capaz de suministrar servicios en línea, tales como el envío de SMS a teléfonos móviles así como de juegos y de otros contenidos digitales (Sastre et al., 2009). La comunicación con el robot se realiza en español a través de Windows Live Messenger de Microsoft.[9] A pesar del dominio restringido y de la simplicidad de las gramáticas aplicadas, los tiempos de ejecución fueron menores para nuestro algoritmo y nuestras estructuras de datos de tipo conjunto. Gracias al coste asintótico mejorado de nuestro algoritmo, son de esperar tiempos de ejecución significativamente inferiores a los de los algoritmos empleados por los sistemas Unitex y Outilex para el caso de gramáticas complejas y de gran cobertura.

---

[7]Una descripción detallada de la STL puede encontrarse en Josuttis, 1999. La implementación de GNU de la STL está siendo distribuida junto con la colección de compiladores de GNU: http://gcc.gnu.org/

[8]http://www.tid.es/en

[9]http://www.msn.com

Preface

In the last decades, our world's societies have been shifting an important part of their resources towards the production, distribution and use of information, earning the surname of Information Societies (Machlup, 1962, but see Crawford, 1983). Moreover, information has become a key factor in every aspect of our lives, from economy and politics to culture. Accordingly, computer science and technology has evolved in order to cope with the increasing demand for the management of information: nowadays computers are no longer mere programmable calculators, as Charles Babbage first conceived them in 1837,[10] but are able to process multiple kinds of data and present them in multiple formats. Pythagoras' claim '*the whole thing is a number*' is being exemplified each time a physical phenomenon is encoded into binary digits and processed by computers, from the so common JPEG images (ITU, 1992; ISO/IEC, 1994), MP3 tunes (ISO/IEC, 1993) and DivX®videos[11] to the particle collisions that take place at CERN's Large Hadron Collider (see Lefevre, 2009); upon these collisions, data bursts of 700 megabytes per second are streamed towards an array of data servers for its storage and distribution to computers around the world (Shiers, 2007), which will analyse this information —a total of 15 petabytes a year[12]— in order to learn about the nature of matter and of the universe itself.

Indeed, computer science and technology has not only provided the tools for data processing but also for its distribution around the world, starting with the creation of the Internet (see Leiner et al., 2009). Apart from pushing the boundaries of computer networking (Newman et al., 2010), CERN has also given rise to the most popular system for sharing information over the Internet: the World Wide Web (Berners-Lee et al., 1992), also called the

---

[10]Visit http://www.fourmilab.ch/babbage/

[11]Visit http://www.divx.com

[12]1 petabyte = 10^6 gigabytes.

WWW. The WWW has provided a universal mean for accessing and linking related digital contents over the Internet: anyone can write a text document with information that he or she considers of interest, make it available around the world by means of a hyperlink and extend it with hyperlinks to other related information, mimicking human's associative memory. Indeed, computers have not only assisted us in the management of numeric data, but in the management of text documents. Combined with search engines, the WWW has become the paramount expression of exploitable collective knowledge, mostly as text documents written in some human language (or natural language). The 15 petabytes of data produced at CERN each year amounts to nothing in comparison with the 20 petabytes of data processed by Google's clusters each day (Dean and Ghemawat, 2008) in order to index a fraction of the one trillion web pages that form the World Wide Web.[13]

Thanks to the appearance of Internet broadband connections, we rather spend our time searching for the information we need at a given moment than on downloading it: addresses and schedules, products and services, news on recent events along with comments from other people, solutions to a great variety of problems and a huge amount of digital content, starting with scientific productions. Indeed, most of the more than 300 papers cited in this dissertation have been searched and downloaded from the WWW, as for any other present scientific production. Actually, we can say that we are "flooded" with information. Efficient natural language processing (NLP) tools for information extraction, filtering and sorting are an obvious need. Moreover, machine translation tools are necessary for the exploitation of text documents written in languages that we do not master. Additionally, machine translation can help us to preserve the world's multi-linguistic culture which globalization is currently threatening, starting with the dominance of English in scientific literature (Enrique Hamel, 2007; Clavero, 2010). According to UNESCO, there are around 6,700 different languages spoken around the world of which about half of them are in danger of disappearing before the century ends (Moseley, 2010).

But the application of NLP technologies is not limited to web content: natural languages are our most common medium for the exchange and support of information. Apart from machine translators and the other mentioned applications, spell and grammar checkers help us to write well-formed

---

[13]Google's estimation as for 2008; visit http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html

texts, and conversational agents provide a more natural mean for human-machine interaction. Yet users of these technologies still complain about ridiculous translation errors, communication deadlocks with automated call center agents and incorrect warnings of their grammar checkers (Vetulani and Uszkoreit, 2009, Pref., p. VI).

Though babies usually start talking when they are two years old, the fact is that human language is very complex. Indeed, humans have two remarkable capabilities which are very hard to mechanize:

- the capability to reason, converse and make rational decisions in an environment of imprecision, uncertainty, incompleteness of information, partiality of truth and possibility, and

- the capability to perform a wide variety of physical and mental tasks without any measurements and any computations (Zadeh, 2009).

Qualitative spatial reasoning is a good example making use of both capabilities (see Freksa, 1991); for instance, we are able to give directions without knowing our environment's exact distribution and without using exact measures of distance or direction but qualitative ones (a little farther, to your left, etc.). As could be expected, natural languages are heavily influenced by our way of thinking. An intriguing example is the fact that all known languages draw heavily on spatial metaphors (Marcus, 2004); for instance, we say a happy person is *on top* of the world while a sad person is *down* in the dumps. But the most salient properties of natural languages —in contrast with formal languages— are their richness, ambiguity and irregularity. While such properties do not prevent us from learning them and communicating between us, the construction of formal descriptions of natural languages for their automatic treatment is no less than challenging. Though doubt has been thrown on the suitability of the current computational paradigm in order to achieve human-level machine intelligence (Zadeh, 2009), multiple natural language models and techniques have been proposed.[14] General human-level language understanding is far to be achieved yet, but more concrete and modest problems have been treated with more or less success: the nowadays existent natural language applications are the proof of this fact,

---

[14]We briefly describe the most popular computable language models in section 1.3, and the most popular parsing algorithms that may apply to our use case in section 1.4.

from Google's linguistic-aware searches to Google's machine translator.[15,16]

Most natural language techniques use statistical methods in order to automatically build some language model by observation of large annotated corpora.[17] The purpose of such techniques range from part-of-speech tagging and disambiguation (Church, 1988; Schmid, 1994; Brill, 1995) to the automated construction of lexicons (see Sun et al., 2008) and grammars (see Klein and Manning, 2005). These techniques avoid the cost of manually building large linguistic databases, which is very convenient for the industrial sector. However, to which extent can a computer capture linguistic information without the assistance of human experts but rather just by searching for coincidences? No magic can extract an information from a data set which does not contain it, either explicitly or not. In our case, we humans do not learn languages from examples alone but coupled with contexts of use —as for the case of our mother tongues— or with language rules made explicit —as for the additional languages we may have learnt from a teacher and/or a textbook. While the use of statistics has given positive results, better quality results can be obtained by using handcrafted linguistic resources. Despite this fact, defenders of the statistical approach have frequently criticized the handcrafted approach as inefficient, subjective, tedious, time-consuming or even boring. However, it appears that such criticisms are rather founded on personal preferences than on convincing evidence: tedious, laborious and boring are an assesment of how much fun researchers find in their work, which is rather a question of personal taste than a valid scientific point (Laporte, 2009). Many other researchers agree that the statistical approach will reach its limits soon, and that handcrafted linguistic resources will be then necessary in order to overcome such limits (Gross and Senellart, 1998; Abeillé and Blache, 2000). The future will tell.

---

[15]Google's search engine no longer searches for an exact list of words, but also searches for the inflected forms of the given words while tolerating spell errors.

[16]Visit http://translate.google.com

[17]An annotated corpus is a machine readable text extended with some linguistic metadata; for instance, the Brown Corpus (Francis and Kučera, 1982) and the Penn Treebank (Marcus et al., 1993) are two large annotated corpora of English: both comprise morphosyntactic annotations of words, while the latter also includes syntactic annotation of sentences (hence the name of 'Treebank').

Contents

II   Finite-state machines   119

IV Appendices   **401**

List of Figures

List of Abbreviations

BSP	blackboard set processing
BST	binary-search tree
CFG	context-free grammar
ECFG	extended CFG
ES	execution state
FPRTN	filtered-popping recursive transition network
FSA	finite-state automata
FSM	finite-state machine
FSMCO	FSM with composite output
FST	finite-state transducer
FSTBO	FST with blackboard output
FSTSO	FST with string output
iff	if and only if
NLP	natural language processing
O-FPRTN	output FPRTN
OS	output state
RFPRTN	reverse FPRTN
RTN	recursive transition network
RTNBO	RTN with string output
RTNSO	RTN with blackboard output
SB	set of blackboards
SES	set of execution states
SOS	set of output states
SS	set of states
UFSM	unification FSM
URTN	unification RTN
WFSM	weighted FSM
WO-FPRTN	weighted output FPRTN
w.r.t.	with respect to
WRTN	weighted RTN

Part I

Preliminaries

Chapter 1

Introduction

Parsing natural-language text with local grammars is one of the ways of locating meaningful sequences in texts. Local grammars are language resources describing sets of meaningful sequences in a language (e.g.: named entities, measurement phrases, etc.). When compared to statistical methods, the use of local grammars provides more control on the results. Current open-source systems for parsing text with local grammars, namely Unitex (Paumier et al., 2009) and Outilex (Blanc and Constant, 2006a), make use of various algorithms depending on the features of the grammars. In this dissertation we propose faster algorithms.

1.1 Local grammars

Formally, local grammars (Gross, 1997) are recursive transition networks (RTNs, Woods, 1970) with output defined on an alphabet of lexical masks. Lexical masks are powerful linguistic operators which ease the definition of natural language grammars: they allow for the representation of large sets of words by means of simple expressions specifying a set of morphosyntactic and/or semantic properties to comply with (e.g.: human noun singular, such as student, lover, fireman, etc.). Numerous studies have shown the adequacy of automata for linguistic problems at all descriptive levels, from morphology and syntax to phonetic issues (Roche and Schabes, 1997; Català and Baptista, 2007; Martineau et al., 2007; Laporte et al., 2008b,a). In particular, the suitability of local grammars for the description of multiple natural language microstructures has been attested by multiple works:

- named entities in Korean (Nam and Choi, 1997), French (Friburger, 2002; Friburger and Maurel, 2002, 2004; Martineau et al., 2007), Arabic (Mesfar, 2007; Traboulsi, 2009), etc.,

- nominal determiners in French (Gross, 2001; Silberztein, 2003a),

- expressions of measure and location adverbs in French (Constant, 2003b),

- date and duration adverbs in Korean (Jung, 2005),

- date adverbs in Greek (Voyatzi, 2006),

- measurement phrases in French (Constant, 2009),

- French determiners (Laporte, 2007),

- coordinated noun phrases in Serbo-Croatian (Nenadić, 2000),

- noun phrases and other clause elements in English (Mason, 2004),

- noun phrases with predicative head in French (Laporte et al., 2008c)),

- complex predicates in English (Gross, 1999) and Portuguese (Ranchhod et al., 2004),

- etc.[1]

Local grammars have also been used in pre-treatment stages facilitating further parsing, such as

- chunking (Poibeau, 2006),

- super-chunking (Blanc et al., 2007),

- annotating compound *da*-conjunctions in Bulgarian (Venkova, 2000),

- annotating French expletive pronouns (Danlos, 2005),

- etc.

---

[1]An extensive list of works using or citing Unitex —therefore likely to be based on local grammars— can be found at http://igm.univ-mlv.fr/~unitex/index.php?page=12

Local grammars have also been used for parsing French simple sentences, either belonging to a particular domain (Fairon and Paumier, 2005) or not (Paumier, 2003). Finally, local grammars have been extended with feature structures and unification processes in order to parse French complex sentences (Blanc and Constant, 2005; Blanc, 2006). The resulting formalism can be seen as an alternative version of lexical-functional grammars where rewrite rules are coded as finite-state automata instead of context-free rules (Blanc, 2006, p. 140).

Local grammars for natural language parsing can be semi-automatically built from lexicon-grammar tables (Roche, 1993; Constant, 2003a). A lexicon-grammar table constitutes a class of predicative elements which depends on the similarity of the sentence structures in which the predicative elements may appear (Leclère, 2002). Following the lexicon-grammar model of syntax (Gross, 1996), a large set of lexicon-grammar tables for French has been constructed since 1968. These tables constitute a very rich linguistic resource describing exhaustively the syntactic and distributional properties of 72000 predicative elements,[2] including

- verbs (Gross, 1975; Boons et al., 1976a,b, 1992),

- predicative nouns (Gross, 1989; Giry-Schneider and Balibar-Mrabti, 1993; Giry-Schneider, 1978, 1987),

- idiomatic expressions (Gross, 1982a, 1985, 1986b, 1988b,a, 1993; Giry-Schneider, 1987), and

- adverbs (Gross, 1982b, 1986a; Molinier and Levrier, 2000).

However, these tables were essentially the result of a linguistic approach with no intention to build a tool for computational applications (Leclère, 2003). Though they have been successfully exploited for the automatic treatment of French, to some extent (Paumier, 2003; Blanc, 2006), converting them into some exploitable format is a non-negligible task (see Hathout and Namer, 1998; Gardent et al., 2005, 2006; Constant and Tolone, 2008; Sagot and Tolone, 2009); indeed, large parts of the informations they contain are neither explicit nor represented in a uniform manner. As work on lexicon-grammar tables advances, we have compared our algorithms of application of local

---

[2]Accessible through the HOOP interface (Sastre, 2006b,a) at `http://hoop.univ-mlv.fr/licenseAgreement.html`

grammars with those of the Unitex and Outilex systems within a simpler use case we describe below. We have not only achieved lower execution times but also a lower asymptotic cost; therefore, even better results can be expected for the case of larger and more complex grammars such as the ones that could be semi-automatically built from the French lexicon-grammar tables.

1.2   The MovistarBot

In collaboration with the enterprise Telefónica I+D,[3] we have built a human-machine interface based on short text messages and local grammars (Sastre et al., 2009).[4] This interface makes use of the different algorithms we present in this dissertation (Sastre and Forcada, 2009; Sastre, 2009), and has served as an evaluation framework. The interface is aimed at extending the understanding capabilities of a chatterbot based on AIML (Wallace, 2004).[5,6] Conversation with the chatterbot is performed by means of short text messages sent through the Internet using one of the most popular instant messaging clients: Microsoft's Windows Live Messenger, commonly known as MSN or Messenger.[7]

As well as providing some general conversation, the chatterbot is aimed at providing mobile services (e.g.: sending SMSs), either requested in Spanish (e.g.: *envía Feliz Navidad al móvil 555-555-555*, which means *send Merry*

---

[3]Telefónica I+D is a research and development enterprise and member of the *Telefónica* group, leader of the telecommunications market in Spain and Latin America and which also enjoys a significant footprint in Europe.

[4]In particular, we have used weighted RTNs with output

[5]A chatterbot is a computer program designed to simulate an intelligent conversation with one or more human users.

[6]AIML is a language for the specification of chatterbot conversation rules based on XML. These rules are mainly composed of recognition patterns —less powerful than regular expressions— coupled with an output which may include input fragments. Conversation contexts may be defined so different sets of rules can be applied depending on the current context, allowing the chatterbot to follow the human into particular domains of conversation.

[7]Communication between MSN clients and Microsoft's servers is performed by means of the Microsoft Notification Protocol (MSNP). Currently, Microsoft servers recognize only MSNP version 8 or higher, but Microsoft published only the specifications of version 2 (Movva and Lai, 1999). However, the open-source community has been reversely engineering newer MSNP versions, and open-source compatible clients are currently available (e.g.: Empathy, Kopete and Pidgin).

Christmas to the cell phone 555-555-555) or by means of commands (e.g. *sms Feliz Navidad 555555555*). The interface we have built specifically targets sentences requesting the services the chatterbot can provide. We have built a grammar for this specific domain and implemented a natural language processing engine making use of the different algorithms presented in this dissertation, including those of the Unitex and Outilex systems in order to compare their performances. The engine has been packed as a Tomcat servlet so that it can provide its services through the Internet to several users concurrently.[8] Each time a message is to be sent to the chatterbot, the message is first sent to the engine for preprocessing. In case the message corresponds to a request of an available service, the engine extracts the specified arguments (e.g.: the message to send and the target phone number) and translates the message into the corresponding command, extending the chatterbot's understanding capabilities to a greater variety of natural language sentences. If the message is not recognized as a service request, it is returned as is to the chatterbot with a special code so that the general AIML conversational rules are applied. Without the engine, the robot mainly searches for keywords (e.g.: sms) and shows the precise syntax to be used in order to launch the presupposed service, obligating the user to retype any arguments that were already provided. In case the arguments are partially provided, the chatterbot is now able to ask only for the missing ones; for instance, upon sentence '*quiero enviar un SMS al 555 555 555*' (I want to send an SMS to the 555 555 555) the chatterbot will only ask for the message to send.

In order to compare the performances of the different algorithms, we have built a corpus of 168 possible sentences,[9] most of them service requests but also other sentences in order to control over-recognition.[10] Service requests are formed by a few compounds that may permute; for instance '*Envía el*

---

[8]Information on Tomcat can be found in the Apache Tomcat homepage http://tomcat.apache.org and in Brittain and Darwin (2007); information and tutorials on servlets can be found in http://java.sun.com

[9]The original corpus contains 334 sentences, though we have only taken into account the sentences that are accurately described by our grammars. Due to the time constraints of the MovistarBot project, accurate grammars for every service the MovistarBot had to support could not be built but simple keyword-matching rules were used instead, which are by far simpler than the grammar rules we can expect in a natural language parsing use case.

[10]By over-recognition we mean to recognize sentences that actually correspond to a service request plus others that do not, usually because of small differences that are not modelled in the grammar.

mensaje hola al móvil 555-555-555' (send the message hello to the mobile 555-555-555) could also be written as '*Envía al móvil 555-555-555 el mensaje hola*' (send to the mobile 555-555-555 the message hello), where each compound has a finite variability (e.g. 'al móvil 555-555-555', 'al 555555555', etc.). The corpus considers every possible permutation, using different compound variants in each one instead of considering every possible combination, thus the corpus is representative in spite of its size. We have aligned the corpus with the answers the system should return and implemented a tool for verifying the answer returned by the system for every sentence in the corpus. Execution times have been measured under the same conditions for each algorithm, namely the same linguistic resources and their corresponding data structures.

1.3   Popular computable grammar formalisms

1.3.1   Lexicon grammar

Lexicon grammar is a methodology for the empirical study of the syntax of natural languages created by Gross (1996), starting with his book '*Méthodes en syntaxe*' (Gross, 1975). Lexicon grammar is based on Harris's (1965) transformational theory of language; in this theory, the analysis of a sentence consists in applying some series of transformations (hence the name) to one or more elementary sentences: for instance, sentence 'The ball was hit by Mary' is analysed as the result of transforming elementary sentence 'Mary hit the ball' into passive form. Harris's aim was to constitute linguistics as a product of mathematical analysis of the data of language, taking elementary sentences as objects on which operators could be applied (Harris, 1968, 1991). Apart from Harris's theory, language models based on Harris's transformational theory of language are called transformational grammars, and they include lexicon grammar.

Lexicon grammar considers that general grammar rules cannot give accurate linguistic descriptions to the irregularities of natural languages; moreover, the presence of specific words within the sentences may condition the sentence structures (Gross, 1997). Though this idea was not original (see Harris, 1951 or Chomsky, 1965), Gross was the first one to shift from such theoretical observation to the empirical description of language, including its lexis (Laporte, 2005). As for Harris's theory, sentences are classified in

lexicon grammar depending on their syntax, and each class is associated to an elementary syntactic structure of sentence. In order to take into account the language irregularities, tables of predicative elements for each class are built; these tables are called lexicon-grammar tables (Leclère, 2002). Each table entry is completed with the data representing the syntactic particularities of the corresponding predicative element w.r.t. the other ones in the class. Parametrized local grammars can then be built in order to represent the elementary and non-elementary syntactic structures for each table. Local grammars for each entry are automatically built by instantiating the parameters of such parametrized grammars, for each entry of the corresponding table. The control exerted by the parameters ranges from the predicative element to appear in the sentences to the prepositions that introduce the sentence arguments, or even the substructures of the parametrized grammar that are to be kept or to be removed, depending on whether they apply or not to the concerned predicative element.

1.3.2 Context-free grammars

Context-free grammars (CFGs, initially called *phrase structure grammars*) were first proposed by Chomsky (1956) as a description method for natural languages. A similar idea was used shortly thereafter to describe computer languages: Fortran by Backus (1959) and Algol by Naur et al. (1960). The resulting Backus-Naur form (BNF) can be seen as an alternate notation for CFGs. Chomsky redefined Harris's transformations as operations mapping sets of deep structures (the syntax trees) to surface structures (the sequence of words that compose the sentences). CFGs mainly consist in a set of terminal symbols, a set of non-terminal symbols and a set of rewrite rules, where

- non-terminal symbols are labels of syntax tree structures (e.g.: 'NP' for noun phrase, 'S' for sentence, etc.),

- terminal symbols are the words of the language, and

- rewrite rules indicate possible replacements of non-terminal symbols within sequences of terminal and non-terminal symbols by other sequences of terminal and non-terminal symbols; for instance, rewrite rule 'NP → DET NOUN' indicates that a noun phrase can be com-

posed by a determiner followed by a noun, and 'DET → the' indicates that 'the' is a determiner.

Analyzing a sentence consists in generating it by transforming sequence 'S' (the non-terminal representing any sentence) into the sequence of words that form the sentence by performing some series rewrites (or transformations). CFGs and any other grammar following this methodology, including local grammars, are said to be generative grammars.

CFGs allow for a structured representation of languages by means of linguistic blocks which can be reused in the description of other blocks (e.g.: one can define what a noun phrase is, then define a prepositional phrase as a preposition followed by a noun phrase). CFGs are said to be *context free* since any rewrite rule for a given non-terminal symbol may apply independently of the context in which that non-terminal symbol may appear, that is, non-terminal symbol definitions do not take into account the non-terminal context.

ECFGs are CFGs where regular expressions can be used within the right part of rewrite rules in order to avoid repetition (but not for augmenting the generative power of the grammar formalism). As for CFGs, there exists an alternate notation for ECFGs based on BNF: extended BNF or EBNF. EBNF is widely used for the description of computer languages and other formal languages, such as XML DTDs (Albert et al., 1998). Indeed, the International Organization for Standardization has adopted an EBNF standard (ISO/IEC, 1996).

CFGs, ECFGs, RTNs and pushdown automata (Oettinger, 1961; Schützenberger, 1963; Evey, 1963) are equivalent formalisms, but CFGs and ECFGs are based on a set of rewrite rules while RTNs and pushdown automata are based on finite-state automata. Finite-state automata allow for a more compact and efficient representation than rewrite rules (Woods, 1969, sec. 1.7.3, p. 40) and can be graphically represented for better readability. More details on CFGs and ECFGs are given in appendix B.

1.3.3   Attribute grammars

Attribute grammars (AGs, Knuth, 1968) are CFGs extended with attributes. These attributes are given values as the grammar productions are applied. The attributes are divided into two groups: synthesized attributes and inherited attributes. The synthesized attributes are the result of the attribute

evaluation rules, and may also use the values of the inherited attributes. The inherited attributes are passed down from parent to children nodes, or from elder brothers to younger brothers. In some approaches, synthesized attributes are used to pass semantic information up the parse tree, while inherited attributes help pass semantic information down and across it; for instance, language translation tools (e.g.: compilers) may use attributes in order to assign semantic values to syntactic constructions. Additionally, it is possible to validate semantic checks that are not explicitly imparted by the syntax definition. AGs can also be seen as an extension of CFGs for output generation; see, for instance, syntax-directed translation based on AGs in Aho et al. (1986, chap. 5, p. 279). As for CFGs, RTNs have also been extended with attributes; the resulting finite-state machines have been called augmented transition networks (ATNs, Woods, 1969).

1.3.4 Probabilistic context-free grammars

CFGs generating ambiguous sentences associate several parse trees to the same ambiguous sentence, one per each interpretation. In order to choose one of the possible parse trees, rewrite rules can be associated to weights or scores, hence associating to each parse tree an overall weight which is the combination (e.g.: addition or multiplication) of the different weights of the successive rewrite rules that led to such parse tree. Such CFGs are called weighted CFGs (WCFGs). A special case of WCFGs, first proposed by Booth (1969), are probabilistic (or stochastic) CFGs (PCFGs): in these grammars, weights are probabilities which define a distribution over the different parse trees the grammar represents; details on how this can be accomplished can be found in Booth and Thompson (1973). WCFGs and PCFGs can be seen as an extension of CFGs for the generation of a particular kind of output (weights or probabilities), as for AGs. Probabilities can be computed by observation of large corpora as usually done in statistical approaches. As could be expected, RTNs can be extended with weights and probabilities (see, for instance, Blanc 2006, sec. 3.3, p. 85). More information on PCFGs can be found in Jurafsky and Martin (2008, chap. 12, p. 444).

1.3.5 Lexical-functional grammars

Lexical-functional grammars (LFGs, Kaplan and Bresnan, 1982) are basically CFGs extended with feature structures and unification processes. LFGs are

composed by two fundamental levels of syntactic representation: the constituent structure (c-structure) and the functional structure (f-structure); c-structures have the form of CFGs, and f-structures are sets of attribute/value pairs (the called feature structures). Attributes may be features (e.g.: tense, gender, etc.) or functions (e.g.: subject, object, etc.). The name of the theory emphasizes an important difference between LFG and the Chomskyan tradition from which it evolved: many phenomena are thought to be more naturally analysed in terms of grammatical functions as represented in the lexicon or in f-structure, rather than on the level of phrase structure. An example is the alternation between active and passive, which rather than being treated as a transformation, is handled in the lexicon. Grammatical functions are not derived from phrase structure configurations, but are represented at the parallel level of functional structure.

As stated before, local grammars have also been extended with feature structures and unification processes for parsing French complex sentences (Blanc and Constant, 2005; Blanc, 2006). Details on unification and how to extend CFGs with feature structures and unification processes can be found in Jurafsky and Martin (2008, chap. 11, p. 391).

1.3.6   Tree-adjoining grammars

Tree-adjoining grammars (TAGs, Joshi et al., 1975) are somewhat similar to CFGs, but the elementary unit of rewriting is the tree rather than the symbol: whereas CFGs have rules for rewriting symbols as strings of other symbols, tree-adjoining grammars have rules for rewriting the nodes of trees as other trees. A TAG consists of a number of elementary trees, which can be combined with a substitution and an adjunction operation in order to obtain derived trees. Interior tree nodes are non-terminals, and frontier tree nodes may either be terminals or non-terminals. Substitution replaces a frontier non-terminal by a tree having the same non-terminal as root. Adjunction is more complex; summarizing, it consists in inserting a tree within another tree, either recursively or not. Because of the formal properties of adjunction, the formalism is more powerful than CFGs, but only mildly so (Joshi, 1985).

Lexicalized TAGs (LTAGs, Abeille, 1988) are a variant of TAGs where each elementary tree contains at least one frontier node labelled with a terminal symbol. Thus each elementary tree is associated with at least one lexical element. Finally, TAGs have also been extended with probabilities (Schabes, 1992) and with feature-structures and unification processes (Vijay-Shanker

and Joshi, 1988; Vijay-Shanker, 1992).

1.4 Parsing algorithms

1.4.1 Top-down

Usually, computable language models are defined having in mind a particular procedure for their application to language utterances; for the case of CFGs, defining a top-down parser is quite straightforward: rewrite rules are applied in order to successively transform the sentence non-terminal into the sequence of words that form the sentence to analyse. The RTN case is analogous: starting from the initial state, outgoing arrows allowing to consume the next sentence word are followed until there are no words left. Due to the ambiguity of the language, or simply due to the grammar structure itself, multiple rewrite sequences (or paths within the RTN) for a given sentence may be possible. Variants of the top-down parser can be defined depending on the order in which rewrite rules (or paths) are explored:

- the depth-first variant (recursive descent, see Aho et al., 1986, sec. 4.4, p. 181) explores one rewrite sequence (or path) at a time, coming back to the last intersection when reaching a dead-end, and

- the breadth-first variant advances the exploration of every possible rewrite sequence (or path) as input words are read.

Top-down parsers may fall into an infinite loop when applying left-recursive grammars, and hence they do not support them. Both CFGs and RTNs can be transformed into some equivalent non-left-recursive grammar, though such transformations have some undesired side effects: the resulting parse trees no longer correspond to the original grammar and contain artificial non-terminals which obfuscate them. Obviously, an alternative solution is to avoid left-recursive structures when building the grammars. Top-down parsers are the simplest and easiest to implement, though they have an exponential worst-case cost. We will present both depth-first and breadth-first variants of top-down parsers in detail for the case of finite-state machines, including RTNs with and without output.

1.4.2   Bottom-up

Though CFGs (and RTNs) seem conceived for being applied by means of top-down parsers, other procedures are possible; for instance, bottom-up parsing with CFGs can be performed by reversely appling the rewrite rules to the sentence words in order to "undo" the rewrites, obtaining sequences that contain non-terminals which are to be searched in other rewrite rules to undo, and so on until obtaining the sentence non-terminal (see Aho et al., 1986, sec. 4.5, p. 195). Usually, whether an algorithm is more efficient than another one depends on the sentence to analyse and the grammar. Top-down parsers may blindly explore multiple rewrite rules until actually reaching some "bottom-level" rewrite rules which require for the presence of certain sentence words; a greater proportion of reachable bottom-level rules not complying with the sentence words will result in a greater percentage of wasted computational time. One may think that a bottom-up parse could solve this problem, since it actually starts from the specific sentence words instead of from a non-terminal symbol representing any sentence. However, bottom-up parsers may also waste time by undoing rewrite rules that do not lead to the sentence non-terminal (see Jurafsky and Martin, 2008, sec. 10.1, p. 355 for a comparative overview on top-down and bottom-up CFG parsers). Unfruitful grammar explorations may be reduced by using smarter algorithms, but one cannot expect to completely avoid them since the whole grammar cannot be applied to the whole sentence in a single operation: information units within both the grammar and the sentence are to be successively examined, and subsets of the computed partial parses may become inconsistent as additional data is taken into account.

1.4.3   LR-parsers

LR-parsers (Knuth, 1965, but see Aho et al., 1986, sec. 4.7, p. 215) are a very efficient class of top-down parsers, though they only support a subset of CFGs. In the name, 'L' stands for 'left-to-right input scanning' and 'R' for 'rightmost derivation'. LR-parsers are mainly based on a table of input symbols $\times$ grammar states $\rightarrow$ action to perform, which is to be constructed for each grammar.[11] Thanks to this table, the grammar can be efficiently applied by systematically executing the action inside the cell indexed by the

---

[11]Apart from the mentioned table, a table action $\times$ state $\rightarrow$ state is also to be built; see Aho et al., 1986, sec. 4.7, p. 215 for more details.

current state and input symbol. However, since only a single action can be defined for a given symbol and state, grammars must be deterministic and non-ambiguous, which is not the case of natural language grammars. Moreover, building an LR-table with a big alphabet will be inefficient or even impractical; this is the case of local grammars since they are defined on the alphabet of the words of the language rather than on the alphabet of letters and symbols.

1.4.4   Tomita

Tomita's (1987) parser is an extended version of LR-parsers which supports non-deterministic and/or ambiguous grammars: upon multiple actions, the parsing process is simply forked in order to execute all of them. However, building LR-tables for local grammars will still be inefficient or impractical due to the alphabet sizes. Efficient data structures for the representation of sparse tables could be used, though we have not studied this possibility.

1.4.5   CYK

CYK (Cocke and Schwartz, 1970; Younger, 1967; Kasami, 1965) is one of the first parsers supporting natural languages —namely ambiguous grammars— and CFGs having a polynomial worst-case cost (n^3). It is a bottom-up parser which makes use of dynamic programming: the parsing problem is broken down into simpler subproblems, and partial solutions are stored in order to be later reused so that no subproblem is solved twice. As drawback, the CYK parser requires to first transform the grammar into Chomsky's normal form (Chomsky, 1959).[12] Though this implies an additional operation to perform, one may only have to compute it once as long as the grammar is not to be modified. However, the grammar size will be considerably increased, its original structure modified, and many artificial non-terminal symbols will be introduced; as for the removal of left recursivity, the resulting parse trees will differ from those obtained from the original grammar, and artificial non-terminals will obfuscate them.

---

[12]Alternative descriptions of Chomsky's normal form can be found in Autebert et al. (1997, sec. 3.1), Hopcroft et al. (2000, sec. 7.1.5) and Sipser (2006, p. 106–109)

1.4.6   Earley

Earley's (1970) CFG parser is another algorithm able to parse natural language grammars.[13] It is a top-down breadth-first algorithm which also makes use of dynamic programming, as for CYK. It has the same worst-case cost than CYK, but does not require to transform the grammar, namely to put it in some normal form (as for CYK's), to determinize it (as for LR-parsers) or to remove left recursion (as for top-down parsers). Contrary to LR and Tomita parsers, it can equally treat grammars defined on either small or big alphabets. Thanks to its efficiency and flexibility, Earley's parser has become a classic natural language parsing algorithm; indeed, it has been adapted to several other grammar formalisms such as

- RTNs (Woods, 1969),

- attribute grammars (Correa, 1991),

- PCFGs (Stolcke, 1995)

- grammars making use of feature structures and unification processes, in general, such as LFG (Shieber, 1985; see, for instance, Jurafsky and Martin, 2008, sec. 11.5, p. 423 for the CFG case),

- tree adjoining grammars (Schabes and Joshi, 1988), and

- weighted RTNs extended with feature structures and unification processes (Blanc, 2006).

The algorithms we propose are mainly inspired in Earley's parser.

1.5   Existing software based on local grammars

1.5.1   Intex

Intex (Silberztein, 1993, 1994, 1998, 2004) was the first corpus processing system based on local grammars.[14] Intex is composed by a Windows graphical interface written in C++ and a set of command line programs written in

---

[13]We briefly describe Earley's CFG parser in appendix C, p. 411; see first appendix B, p. 405, for a description of the CFG notation.

[14]Intex homepage: http://intex.univ-fcomte.fr

C; the command line programs are either called by the graphical interface or through the command line in order to perform the different treatments available.[15] Though Intex is mainly a Windows application, Intex can be run on Macintosh platforms thanks to tools such as Virtual PC,[16] and Unix/Linux binary versions of the command line programs are also provided with Intex (Silberztein, 2004, sec. 17.1, p. 192). However, Intex is not open-source and cannot be freely downloaded except for academic purposes, and a license number is to be requested by email each time it is installed in a new hard drive;[17] moreover, details on the implemented algorithm of application of local grammars are not given in the documentation; hence, they remain obscure. In the academic sense, Intex is a tool for the study of natural languages but not for the study of parsing algorithms. Even in the former case, the Intex author may refuse to provide license numbers at will.[18]

1.5.2   NooJ

Intex development has been discontinued in favor of its successor: NooJ (Silberztein, 2003b, 2005a, 2007).[19] The author decided to reimplement the whole system from the scratch with a new set of programming tools from Microsoft: the C# programming language, the .NET framework and the Visual Studio software development environment.[20,21,22] Though there exist alternative implementations of C# and .NET compatible with non-Windows platforms, namely Mono and DotGNU,[23,24] they do not fully support NooJ yet (Silberztein, 2003b, p. 9). NooJ's author mentions two main reasons for choosing the new set of programming tools:[25] the benefits of a component

---

[15]According to paragraph entitled "*Chapitre* 1" of http://mshe.univ-fcomte.fr/intex/Unitex.htm

[16]According to last paragraph of http://mshe.univ-fcomte.fr/intex/Unitex.htm

[17]Intex will run in "demo mode" if a license number is not provided.

[18]Actually, the author of this dissertation experienced this situation when trying to install Intex in several machines in the context of academic project DRUID (Laforest and Badr, 2003).

[19]NooJ homepage: http://www.nooj4nlp.net/pages/nooj.html

[20]See, for instance, (Albahari and Albahari, 2010) for more information on C# programming language.

[21].NET homepage: http://www.microsoft.com/net

[22]Visual Studio homepage: http://www.microsoft.com/visualstudio

[23]Mono homepage: http://www.mono-project.com

[24]DotGNU homepage: http://www.gnu.org/software/dotgnu

[25]According to http://www.nooj4nlp.net/pages/links.html

programming methodology (in contrast with ANSI C) and a free automatic
memory management. While such features facilitate software development,
control is lost on certain implementation details which have an impact on the
efficiency of the parsing algorithms; for instance, the algorithms we propose
in this dissertation use complex data structures which are certainly more
expensive to delete once they are no longer needed. If we are to reduce
consecutive parsing times,[26] deletion of data structures cannot be left to a
garbage collector but has to be optimized as well.

1.5.3   Unitex

Unitex (Paumier, 2003, 2006, 2008; Paumier et al., 2009) has been the first
open-source alternative to Intex:[27] it is distributed under the GNU LGPL li-
cense and the linguistic resources it includes are distributed under the LPGL-
LR license.[28] The advantages of open-source development are multiple and
have been widely recognized (Raymond, 1999; Davis et al., 2000; Raymond,
2001; Graham, 2001; Ambati and Kishore, 2004; Forcada, 2006; von Krogh
and von Hippel, 2006; von Krogh and Spaeth, 2007; Paumier et al., 2009;
Scacchi, 2010), starting with the simple intention of letting others study
one's work in order to reuse or even to improve it. Unitex uses a top-down
depth-first algorithm of application of RTNs with string output, where out-
put strings may contain copies of input segments. In our case, we have reused
the Unicode library included in Unitex but have reimplemented its parsing
algorithm in order to test the different algorithms under the same conditions;
Unitex linguistic programs are mainly implemented in ANSI C while we have
preferred to take advantage of the C++ object oriented and generic program-
ming as well as of the new functionalities provided by the Standard Template
Library.[29]

---

[26]Recall that we are to analyse sentences requesting for online services as they are
received through the Internet from multiple users.

[27]Unitex homepage: http://igm.univ-mlv.fr/~unitex

[28]The terms and conditions of the LGPL-LR and GNU's LGPL licenses can be found at
http://igm.univ-mlv.fr/~unitex/lgpllr.html and http://www.gnu.org/licenses/
gpl.html, respectively.

[29]See Josuttis (1999, chap. 2, p. 13) for a good introduction on the new functionalities
added to C++, including the Standard Template Library.

1.5.4 Outilex

Outilex (Blanc et al., 2006; Blanc and Constant, 2006a; Blanc, 2006; Blanc and Constant, 2006b)[30] is another open-source platform for corpus processing (LGPL licensed), based on RTNs with a more complex output than Unitex's RTNs: weights combined with feature structures built by means of unification processes. Outilex uses an Earley-like parser equivalent to that presented in Sastre and Forcada (2009). Though the original Earley parser has a polynomial worst-case cost (n^3), extending it for output generation results in an exponential worst-case cost due to grammars generating an exponential number of outputs w.r.t. the length of certain inputs (Sastre and Forcada, 2009). Such cases occur in natural language grammars; for instance, if the grammar outputs represent sentence parses,[31] the number of possible sentence parses increases exponentially w.r.t. the number of unresolved prepositional phrase attachments it contains:

- in sentence 'the girl saw the monkey with the telescope', it is unknown whether the girl used the telescope or the monkey was holding it (2^1 interpretations),

- sentence 'the girl saw the monkey with the telescope in the garden', it is also unknown whether the monkey was in the garden or the action took place in the garden (2^2 interpretations),

- in sentence 'the girl saw the monkey with the telescope in the garden under the tree', it is unknown as well whether the monkey was under the tree or the action took place under the tree (2^3 interpretations),

- etc.[32]

In the MovistarBot use case, we have used string output combined with weights: output strings are tags which identify the requested service and the arguments provided (e.g.: to send an SMS to a given phone number), and weights are used in order to choose one interpretation among those of

---

[30]Outilex homepage: http://igm.univ-mlv.fr/~mconstant/outilex

[31]Outputs can be XML tags (Bray et al., 2008) that are inserted in certain sentence locations in order to identify and delimit the different sentence constituents, extending the original sentences with their parse trees.

[32]Example borrowed from (Butt, 2002). More information on this problem, along with a solution based on statistics can be found in Ratnaparkhi (1998).

ambiguous sentences (the one with the "highest score"). The parsing algorithm we propose is able to compute the highest-ranked output while keeping Earley's original worst-case cost.

Though active development on Outilex has been discontinued, its source code may be integrated into the Unitex system in the future.

1.6 Other finite-state software

1.6.1 Apertium

Apertium (Armentano-Oller et al., 2007; Forcada et al., 2009, 2010) is an open-source machine translation platform which is being distributed under the GNU GPL license.[33],[34] Apertium uses finite-state transducers for lexical processing, hidden Markov models for part-of-speech tagging, and multi-stage finite-state chunking for structural transfer. Apertium was initially designed to treat pairs of closely related languages spoken in Spain and Portugal, but it is nowadays able to treat other less related language pairs such as Spanish and French. Many of the breadth-first and minizimation strategies in Apertium have inspired this thesis.

1.6.2 SisHiTra

SisHiTra (*sistema híbrido de traducción* or hybrid translation system, Navarro et al., 2004) is another machine translation system making use of finite-state technology and statistical methods, as Apertium, but is restricted to Spanish and Catalan. It can be used online at http://sishitra.iti.upv.es/

1.6.3 FSA Utilities

FSA Utilities toolbox (van Noord, 1997) is a collection of utilities to manipulate finite-state automata and finite-state transducers.[35] Manipulations include determinization (both for finite-state acceptors and finite-state transducers), minimization, composition, complementation, intersection, Kleene

---

[33]The terms and conditions of GNU's GPL license can be found at http://www.gnu.org/licenses/gpl.html

[34]Apertium homepage: http://www.apertium.org

[35]FSA Utilities homepage: http://www.let.rug.nl/˜vannoord/Fsa/

closure, etc. Furthermore, various visualization tools are available to browse finite-state automata. The toolbox is implemented in SICStus Prolog and is being distributed under GNU's GPL license.[36]

1.6.4 XFST & Vi-xfst

XFST (Xerox finite-state tool, Karttunen et al., 1997) is a non-free general-purpose utility for computing with finite-state networks. It enables the user to create simple automata and transducers from text and binary files, regular expressions and other networks by a variety of operations. The user can display, examine and modify the structure and the content of the networks.

Vi-xfst (Oflazer and Yılmaz, 2004a,b) is a front-end for XFST which provides a visual interface and a development environment for the construction of finite-state language processing applications. Complex regular expressions can be built via drag-and-drop, treating simpler regular expressions as construction blocks.

More information on both tools can be found at the homepage of the 'Finite-State Morphology' book (Beesley and Karttunen, 2003): `http://www.fsmbook.com`

1.6.5 AT&T FSM library^{TM}

The AT&T FSM library^{TM} (Mohri et al., 1998) is a set of general-purpose software tools available for Unix. It allows for building, combining, optimizing, and searching weighted finite-state acceptors and transducers.[37] The original goal of the library was to provide algorithms and representations for phonetic, lexical, and language-modeling components of large-vocabulary speech recognition systems. The library is available under non-commercial (binary only) and commercial licenses from AT&T Labs.

1.6.6 OpenFST

OpenFst (Allauzen et al., 2007) is a library for constructing, combining, optimizing, and searching weighted finite-state transducers.[38] OpenFst con-

---

[36]SICStus Prolog homepage: `http://www.sics.se/sicstus/`

[37]AT&T FSM library^{TM} homepage: `http://www2.research.att.com/~fsmtools/fsm/`

[38]OpenFST homepage: `http://www.openfst.org/twiki/bin/view/FST/WebHome`

sists of a C++ template library with efficient WFST representations and over 25 operations for constructing, combining, optimizing, and searching them. OpenFst is an open source project and is being distributed under the Apache license.[39]

1.6.7   HFST

The Helsinki Finite-State Transducer software (HFST, Lindén et al., 2009) is intended for the implementation of morphological analysers and other tools which are based on weighted and unweighted finite-state transducer technology.[40] HFST is compatible with XFST, and is being distributed under GNU's LGPL license.

1.6.8   Foma

Foma (Hulden, 2009) is a compiler, programming language, and C library for constructing finite-state automata and transducers for various uses. It has specific support for many natural language processing applications such as producing morphological analysers.[41] Foma is compatible with XFST, and is being distributed under GNU's GPL license.

1.7   Structure of this work

The different elements that we expose in this dissertation are heavily interrelated, which makes difficult to describe them in some sequence without referring to future material; for instance, the optimization of set data structures for boosting the different parsing algorithms strongly depends on the particular requirements of the different parsing algorithms. Conversely, some implementation details of set data structures must be taken into account when constructing the parsing algorithms. We have chosen to follow a "weak" bottom-up approach: objects that are either components or simpler

---

[39]The terms and conditions of the Apache license can be found at http://www.apache.org/licenses/LICENSE-2.0

[40]HFST homepage: http://www.ling.helsinki.fi/kieliteknologia/tutkimus/hfst/

[41]Foma homepage: http://foma.sourceforge.net

cases of other objects are described first, but relevant properties of future objects are briefly described in advance when needed.

As we have seen, different authors define different kinds of finite-state machines depending on their needs, though the basic structures remain unchanged:

- Unitex uses RTNs with string output,

- Outilex uses RTNs outputting weights and feature structures, and

- we have used —in the MovistarBot use case— RTNs outputting weighted tables associating the identifiers of the detected service and arguments to the corresponding input intervals where they have been located; for instance, the following output table is generated (among others) for the case of sentence '*envía hola al 555-555-555*' (send hello to the 555-555-555):

$$
\begin{aligned}
\text{sms} &\rightarrow (1,1] \\
\text{message} &\rightarrow (1,2] \\
\text{phone} &\rightarrow (3,14]
\end{aligned}
$$

  This table is a representation of the following sentence bracketing, assuming that the first token position is 1 and that tokens are either words or digits: '*envía*`<sms/>` `<message>`*hola*`</message>` *al* `<phone>`555-555-555`</phone>`'.

Additionally, different kinds of finite-state machines are used depending on the data to represent:

- tries and other acyclic finite-state automata in order to represent dictionaries and other sets of sequences,

- different kinds of RTNs in order to represent grammars, and

- filtered-popping RTNs (Sastre, 2009) —a new kind of machine we also present in this dissertation— in order to serve as a compact representation of the result of applying a RTN with output.

Rather than being completely different objects, those machines having more complex features can be seen as extended versions of simpler ones, and are

indeed easier to describe by incrementally refining the simplest case. Because
of these reasons, we have chosen to build a hierarchy of finite-state machines
for any kind of input and output, along with the corresponding algorithms
of treatment. This hierarchy has served as a theoretical basis for the im-
plementation of a C++ library of finite-state machines and the corresponding
generic algorithms of application. The library has been adapted to the Mo-
vistarBot use case and is meant to be easily extended in order to consider
RTNs with different kinds of output, such as those of the Unitex and Out-
ilex systems. As a final remark, object oriented and generic programming
has not only allowed us to factor out common parts of the source code but
also to ensure that different performances are exclusively due to the different
strategies followed by the different parsing algorithms.

This dissertation is mainly divided in 4 parts as we describe below.

1.7.1   Part I: Preliminaries

Part I includes this introduction and the description of some objects that are
used by our implementation of finite-state machines and their algorithms of
treatment, namely:

- Chapter 2 describes efficient implementations of set and map data
  structures. Most of the parsing algorithms we describe in this dis-
  sertation make an intensive use of set and map data structures, hence
  the need for such efficient implementations.

- Chapter 3 describes some implementation concerns around character
  representation. The texts to analyse are basically sequences of charac-
  ters.

- Chapter 4 describes the dictionaries we have used in order to store
  morphosyntactic and semantic data for each word of the language, as
  well as some implementation details.

- Chapter 5 describes tokens, the minimal input unit our machines take
  into account, and how character sequences are segmented into tokens.

- Chapter 6 describes lexical masks and other predicates that we have
  used as input labels of the local grammar transitions in order to repre-
  sent sets of tokens, and to detect whether tokens are blank-separated or

not. Token sets are usually defined as the dictionary words complying with a set of morphosyntactic properties.

1.7.2 Part II: Finite-state machines

This part describes the hierarchy of machines we have defined. It comprises definitions and properties of finite-state machines, algorithms of application and other algorithms that optimize the machines and ensure the absence of "offending" machine substructures (substructures that may lead to infinite loops upon the application of the machines). Moreover, we will show that those machines having unavoidable offending structures make no sense as natural language grammars. Contrary to CYK's parser, these machine optimizations do not introduce artificial non-terminal symbols. Each chapter corresponds to a type of finite-state machine, namely:

- Finite-state machines or FSMs (chapter 7), the base class for every kind of finite-state machine. This class does not really define a specific machine but gives definitions, properties and algorithms common to all the machines; in the context of object oriented programming it would be an abstract class defining pure virtual methods.

- Finite-state automata or FSAs (chapter 8), sequence acceptors representing regular languages. Deterministic and non-deterministic FSAs (DFAs & NFAs) are subcategories of this class. In particular, we have used acyclic DFAs for the representation of electronic dictionaries.

- Tries (chapter 9), a particular case of DFAs used here for the optimization of sequence copies and comparisons as well as for the representation of electronic dictionaries 6.1.4 (apart from acyclic DFAs).

- Finite-state transducers with blackboard output or FSTBOs (chapter 10), a generic extension of FSAs for the generation of any kind of output. Blackboards are either simple or structured data types, and output symbols are functions on blackboards. Apart from generating output, blackboards may also be used in order to further restrict the language recognized by the original FSA. In particular, machines extended with feature structures and unification reject input interpretations that involve the generation of inconsistent feature structures.

- Finite-state transducers with string output or FSTSOs (chapter 11), letter transducers described as a special case of blackboard output where blackboards are strings and functions on blackboars append output symbols. String output may be used to enrich texts with meta-information, for instance tags indicating the syntactic structure of the sentences, or simply marking input segments containing relevant information to be extracted.

- Recursive transition networks or RTNs (chapter 12), recursive sequence acceptors equivalent to CFGs and pushdown automata (Oettinger, 1961; Schützenberger, 1963; Evey, 1963), hence having a greater generative power than FSAs; as CFGs, RTNs allow for structured definitions of grammars where subgrammars can be reused in the definition of higher level grammars by means of call transitions.

- Recursive transition networks with blackboard output or RTNBOs (chapter 13), a kind of machine combining recursive calls and blackboard output.

- Recursive transition networks with string output or RTNSOs (chapter 14), RTNBOs where blackboards are strings and functions on blackboards append output symbols (as for FSTSOs).

- Filtered-popping recursive transition networks or FPRTNs (chapter 15), RTNs where returning from a call is only possible under certain conditions (return or 'pop' transitions are filtered, hence the name). We also call FPRTNs filtered-popping networks or FPNs, though we rather use here acronym FPRTN since acronym FPN is already in use for fuzzy Petri nets (see, for instance, Aziz et al., 2010). We use here FPRTNs as a compact representation of the set of outputs generated by a RTNBO for a given input string. We present in this chapter an algorithm that computes such FPRTN-compacted outputs in time n^3, even for cases in which the number of outputs to generate increases exponentially w.r.t. the input length (an example of this situation with natural language grammars has been given in section 1.5.4, p. 19).

- Reversed FPRTNs or RFPRTNs (section 15.5, p. 306), these machines recognize the reverse of the languages accepted by some FPRTN. Reversing a FPRTN requires to filter pushing transitions —the call initializers— instead of popping transitions. RFPRTNs may also be referred

to as filtered-pushing RTNs or filtered-pushing networks, though we use acronyms RFPN or RFPRTN in order to avoid ambiguity (both words 'pushing' and 'popping' start with the same letter).

- Output FPRTNs or O-FPRTNs (chapter 16): the subclass of FPRTNs serving as a compact representation of a set of outputs. We study here the properties of O-FPRTNs and set the bases for further post-processings, mainly the efficient generation of the language of outputs represented by an O-FPRTN.

A schema of this machine hierarchy is given in figure ??.

The last 3 chapters of this part give the guidelines for constructing machines with other kinds of output as particular cases of blackboard output, namely:

- Finite-state machines with composite output or FSMCOs (chapter 17): FSMs generating multiple outputs, either of different types or not. FSMCOs equivalent to Turing machines (Turing, 1936, but see Hopcroft et al., 2000, sec. 8.2, p. 319) can be seen as machines with multiple output tapes.

- Weighted finite-state machines or WFSMs (chapter 18): FSMs with blackboard output where blackboards are weights and functions on blackboards may increase or decrease them. This kind of output serves as a non-arbitrary mechanism for the selection of a unique output upon ambiguous sentences, which is to be used by end-user applications such as chatterbots and machine translators. In this chapter, we present an algorithm able to generate only the top-ranked output represented by an O-FPRTN in time n^3, even for cases in which the O-FPRTN represents an exponential set of outputs.

- Unification finite-state machines or UFSMs (chapter 19): a kind of FSMs with blackboard output where blackboards are feature structures and transitions may define functions that unify pairs of blackboards, as for the case of Outilex's local grammars. Unification allows for a compact representation of grammatical phenomena such as agreement and subcategorization. Unification introduces the possibility of generating killing blackboards, which in this case are inconsistent feature structures; input sequences that involve to generate killing blackboards are

to be rejected. The efficient computation of the non-killing top-ranked output is a more complex problem that is left open here for a future work.

1.7.3 Part III: Results and conclusions

This is the concluding part and comprises two chapters:

- chapter 20 presents an empirical comparison of the performances of the different algorithms of application of local grammars in the context of the MovistarBot project, and also compares their performance drops with an artificial minimal grammar generating an exponential number of outputs w.r.t. an input increasing in length, and

- chapter 21 summarizes our contributions and gives a list of further improvements that could be applied to this work in the future.

1.7.4 Part IV: Appendices

Finally, in this part we briefly describe the basic algorithms —and the objects on which they operate— which have inspired some of the algorithms presented in this dissertation, namely

- appendix A summarizes the set of lexical masks and predicates we have used as input labels of the local grammar transitions,

- appendix B gives a brief overview of CFGs and presents the notation we have followed for representing them,

- appendix C briefly describes the original Earley parser for CFGs, and

- appendix D briefly describes PERT networks and Kahn's algorithm for computing a possible topological sort a PERT network,[42]

Last but not least, an index of terms can be found at the end of this manuscript, right after the bibliography. This index includes the different abbreviations, algorithms, functions, and variable identifiers. A list of the most common abbreviations used throughout this dissertation has been given in page xxiv.

---

[42]PERT stands for 'program evaluation and review technique'.
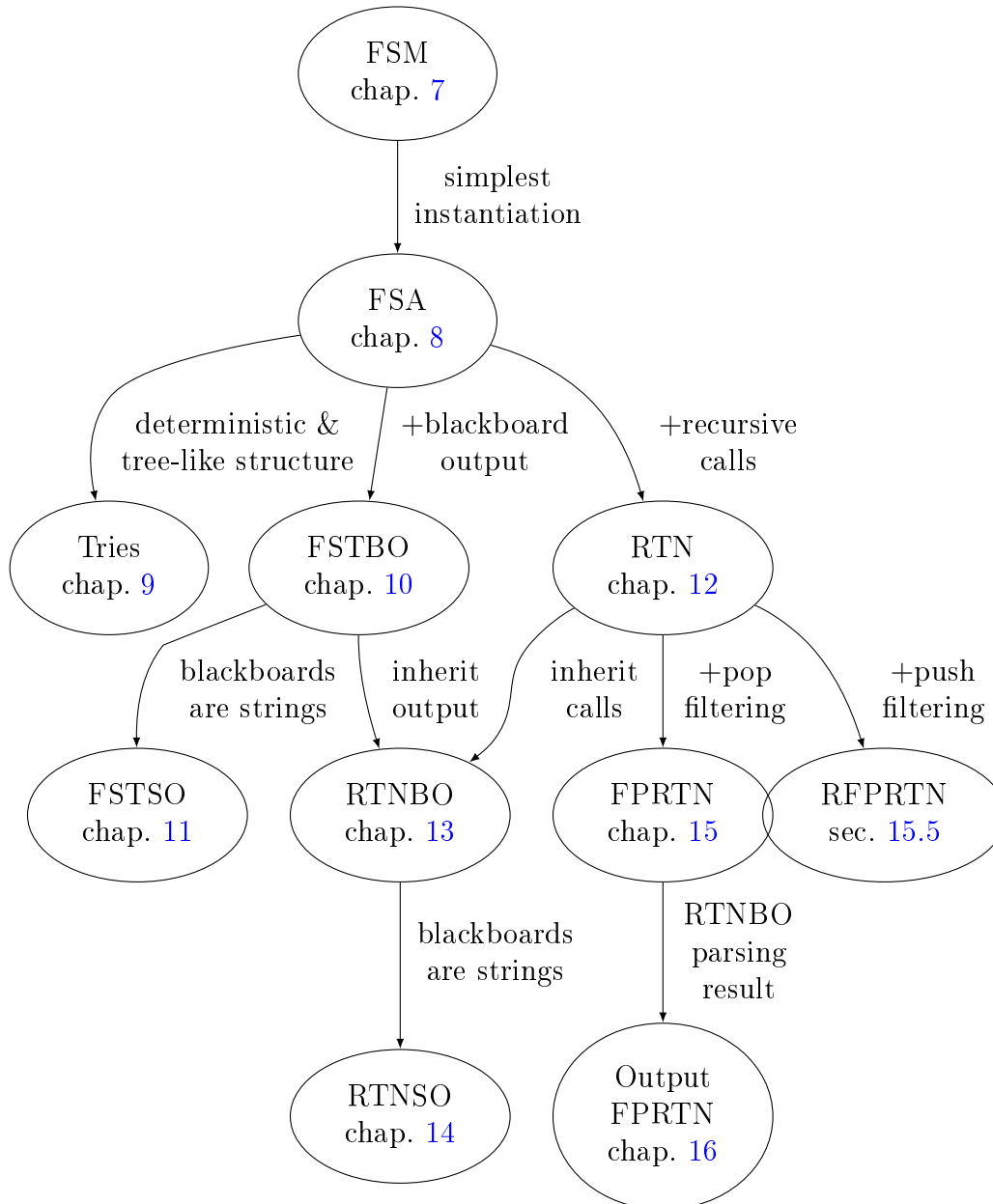
Figure 1.1: Hierarchy of the finite-state machines.

Chapter 2

Sets and maps

Most of the parsing algorithms that we will present in this dissertation make an intensive use of data objects representing sets, that is, collections of unique elements: upon adding an element to a set structure, the element must be first searched within the structure so that it is not added twice.[1] Other more sophisticated parsing algorithms also make use of maps.[2] However, set and map representation and management is, in essence, the same problem since maps can be regarded as sets of key/value pairs where keys are the only distinctive trait between pairs: upon adding an element (k, v_2) to a map that already contains an element (k, v_1), (k, v_2) will not be added (nor replace the former pair) since it is considered to be already present within the data structure.[3] At first, we simply used the set and map implementation provided by GNU's implementation of the C++ Standard Template Library (STL). This implementation is expected to be efficient in most cases. However, our experience has proved that other implementations perform better —

---

[1]We do not intend here to give a mathematical definition of set but rather treat them as a container class of an object oriented programming language, namely C++ coupled with the Standard Template Library (see for instance Josuttis, 1999, sec. 5.2, p. 70). Introductory material on set theory can be found in Devlin (1993, chap. 1).

[2]Likewise, we treat here maps as a class of associative container. To put them in mathematical terms, a map M is a binary relation between keys in K and values in V such that $k \, M \, v$ and $k \, M \, v'$ iff $v = v'$, and we say M maps k to v or associates key k with value v; in other words, M may either map a key to a single value or leave it unmapped, and no restriction applies on the amount of keys associated with the same value.

[3]Indeed, GNU's implementation of the STL uses the same data structures for the representation of sets and maps, but the stored elements are keys in the former case and key/value pairs in the latter one.

depending on the algorithm and the use of the structures— and are even
mandatory if we are to implement faster parsing algorithms than the ones
used in the Unitex and Outilex systems.

Most of the algorithms we propose use a dynamic programming approach
(Bellman, 1957): the parsing problem is broken down into simpler subprob-
lems, which are to be solved only once. Some data structure is used in
order to represent the subproblems along with their computed solutions (the
partial parses). The algorithms build either sets or maps of such data struc-
tures in order to ensure that the same pair subproblem/solution is not added
twice, hence avoiding the repeated computation of any further subproblems
that would follow the ones already solved. Since natural language sentences
can have multiple interpretations —and indeed they usually do— multiple
parses are possible. Once every subproblem is solved, the set of possible
parses is built by combining the different subproblem solutions, avoiding re-
peated parses thanks to the use of a set data structure. Last but not least,
the algorithms perform sequential traversals of the sets and maps in order to
execute at least one of the following operations:

- search and remove every useless partial parse due to sentence misinter-
  pretations,[4]

- apply some post-processing to each element of the set of total parses,
  and

- delete the sets and maps once they are no longer needed, which implies
  to first remove every set or map element one by one.

Whether more sophisticated algorithms will be faster than simpler ones will
strongly depend on the use of set and map implementations providing effi-
cient addition, removal and sequential traversal methods. We simultaneously
discuss these problems for both set and map structures by presenting solu-
tions to the efficient management of sets of key/value pairs (for the case of
sets, assume that values are empty).

The problem of efficient set management is ubiquitous. As could be ex-
pected, the solutions that have been proposed are numerous. In this chapter

---

[4]Note that, due to local ambiguities, parsing algorithms may not realize of a sentence
misinterpretation until reading enough sentence words; for instance, in sentence 'the man
whistling tunes pianos', one does not realize that 'tunes' is the sentence verb —rather than
a part of the subject— until reading 'pianos'.

we first introduce the problem of set management by presenting a trivial solution based on arrays (section 2.1). We further refine this solution by means of double-linked lists (section 2.2) and, furthermore, with binary-search trees (section 2.3); many of the solutions proposed in the literature, including the one of GNU's implementation of the STL, are based on some kind of binary-search tree. In the section, we describe some GNU implementation choices along with some alternative algorithms and optimizations. In section 2.4 we enumerate and summarize the advantages and drawbacks of different kinds of self-balanced binary-search trees, a further refined kind of binary-search trees. In section 2.5 we focus on red-black trees, the particular kind of self-balancing trees chosen for GNU's implementation. In section 2.6 we present our solution: a hybrid structure combining a double-linked list with a red-black tree. In section 2.7 we briefly describe other structures that could be used instead of those based on red-black trees; some of them —perhaps combined as well with a double-linked list— are worth to be considered in future works. In section 2.8 we discuss how to efficiently implement maps of keys to sets of values. Finally, we give in section 2.9 the guidelines for adapting the previously presented set and map structures for the representation of multisets and multimaps; these guidelines are to be followed in order to reimplement every set and map structure provided by the STL.

2.1 Arrays

In spite of the simplicity of the concept of set, the efficient implementation of set data structures is a rather complex problem. Sets are to be stored in a computer's memory, which in turn is an array of bytes. As stated before, emulating a set with an array requires to first search the array for elements having the same key than the ones to be added before actually adding them. While adding an element to an unordered sequence requires only a constant time (e.g.: to append it to the end of the sequence), searching for an element with a specific key requires an average time proportional to the array size since the element's key is to be compared one by one with the ones of the elements previously added to the array. In order to reduce this time, a total order is to be defined over the set of keys —say $k_i \prec k_j$ for every pair of keys (k_i, k_j) such that $i < j$— and the array is to be kept sorted w.r.t. this order, at least until element searches will no longer be required. A binary search can then be performed, which has a logarithmic worst-case cost w.r.t. the

array size instead of proportional.

Algorithm 2.1 *sorted_array_add* adds a key/value pair (k, v) to a set represented as a sorted array $a_0 \ldots a_{n-1}$. It first performs a binary search for the position where to insert the element, then inserts it in that position if it is not already occupied by an element having k as key. The algorithm returns a Boolean indicating whether the element was inserted or not. The binary search is based on the one performed by algorithm B in Knuth (1998, p. 410). During the whole algorithm execution, variables i and j represent the bounds of the search interval, starting with $[0, n)$, the range covering the whole array. As long as the interval is not empty ($i \leq j$), it first sets m to the middle position of the interval. In case the interval contains an even amount of elements, the greater of the two middle positions is chosen. If the key of the element to search is less than the one of the middle element, the search continues with interval $[i, m)$, the inferior half of the current interval without the middle element. If it is greater, it proceeds with $[m + 1, j)$, the superior half of the current interval without the middle element. If it is neither less or greater, the algorithm returns $((k_m, v_m), \text{false})$ without inserting the element, where (k_m, v_m) is the element in A such that $k_m = k$. If the array does not contain an element having k as key, the interval will be successively divided by 2 up to obtaining an empty interval $[i, j)$ with both i and j pointing to the element having the least key greater than k. In that case, the element is inserted at position i and $((k_i, v_i), \text{true})$ is returned. The insertion operation first requires to shift elements $(k_i, v_i) \ldots (k_{n-1}, v_{n-1})$ one position to the right in order to make room for the new element. Note that this algorithm does not require to define operators $=$ or $\succ$ (reverse total order operator) but only $\prec$, without loss of efficiency. Indeed, STL sets and maps require to define only one comparator operator. More information on binary searches can be found in Knuth (1998, chap. 6.2.1).

While the binary search has a logarithmic worst-case cost (the search space is divided by 2 at each unfruitful iteration), the insertion operation still has a worst-case cost proportional to the array size due to the shifting operation. Moreover, if there is no free memory right after the last element in order to allocate one more element, the whole array must be copied into a big-enough free memory segment. If the maximum amount of elements to be added is known before creating the set, enough free memory can be reserved in order to avoid this situation, but that will not be the case for the parsing algorithms presented in this dissertation and, anyway, we would still be facing the shifting problem. Deleting an element from the array will not

---

Algorithm 2.1 sorted_array_add($A, (k, v)$)

Input: $A = (k_0, v_0) \ldots (k_{n-1}, v_{n-1})$, a sorted array of n key/value pairs
(k, v), the key/value pair to add to the array

Output: A after inserting (k, v) at a position i, if there is no $k_j = k$ in A,
or A unmodified if there is a $k_m = k$ in A
returns $((k_i, v_i), \text{true})$ in the former case, and $((k_m, v_m), \text{false})$ in
the latter one

1: $i \leftarrow 0$
2: $j \leftarrow n$
3: **while** $i \neq j$ **do**
4:      $m \leftarrow i + \text{integer\_division}(i - j, 2)$
5:      **if** $k \prec a_m$ **then**
6:          $j = m$
7:      **else if** $a_m \prec k$ **then**
8:          $i = m + 1$
9:      **else**
10:         $b \leftarrow \text{false}$
11:         **return** $((k_m, v_m), \text{false})$
12:      **end if**
13: **end while**
14: insert$(A, i, (k, v))$
15: **return** $((k_i, v_i), \text{true})$

---

require to reallocate the whole array, but the elements at greater positions than the deleted one will still have to be shifted left. The operations in which arrays excel are both the sequential and random access of its elements (hence the name of random access memory or RAM).

2.2   Double linked lists

Double-linked lists are data structures having both efficient insertion and removal methods as well as a sequential traversal method. List elements do not necessarily lie on consecutive memory positions but in arbitrary ones. In order to enable both the forward and reverse traversal of the list, the list structure contains a pointer towards the first element and another towards the last one, and each list element contains a pointer towards its previous neighbour and another towards its next one. Both inserting and removing an element x consists in redirecting x's neighbouring pointers as well as the ones of its neighbours, hence saving the hassle of shifting every element with a key greater than the one of x (see Cormen et al., 2001, chap. 10 for more details). However, direct access to elements at random positions is no longer possible since they no longer lie at consecutive memory positions. In order to compute the middle element between two elements a_i and a_j, the list must be walked from a_i towards a_j and from a_j towards a_i, element by element in both directions, until both walks reach the same element. Hence, adding an element to a sorted list will still have an average cost proportional to the list size.

2.3   Binary search trees

Binary search trees (BSTs) are a straightforward representation of every possible binary search that can be performed on a sorted sequence. Like double-linked lists, they also augment each element data structure with two pointers, though their structure is not sequential but hierarchical: tree data objects contain a pointer towards the top element of the hierarchy, the *root* of the tree, and the two pointers of each element reference the root of their respective left and right subtrees. Subtree roots y and z of an element x are called the *children* of x and, conversely, x is called the *parent* of both y and z. We use symbol $\perp$ in order to represent the absence of element, namely

- root$(T) = \perp$ (T has no root),[5]

- left$(y) = \perp$ (y has no left child) and

- right$(z) = \perp$ (z has no right child).

In practice, the corresponding pointers are given a null value.

In the ideal case, the root of the tree is the middle element, its left and right subtrees contain, respectively, the inferior and superior halves of the tree minus the root, the subtrees of the root's children contain the quarters minus the tree and subtree roots, and so on until reaching a bottom hierarchy level whose elements are either missing or have no children (see figure 2.1(a)).[6] In other words, the number of hierarchy levels —the *height* of the tree— is minimal. Such trees are said to be *balanced*. Searching for an element with a key k inside a BST consists in traversing the tree downwards from the root, either stepping towards the left or right child of each element x if k is less or greater than the key of x, or stopping at x if its key is equal to k. Adding an element to a BST will finally have a logarithmic worst-case cost w.r.t. the tree size rather than proportional, provided that the tree is balanced. In return, the sequential access to the tree elements is more complex and expensive than with arrays or double-linked lists. We will first study the sequential traversal since it introduces some modifications to be done on the tree structure which the other algorithms must maintain.

2.3.1 Recursive traversal

Algorithm 2.2 *bst_process_in_order* performs an in-order walk of the tree having x as root in order to perform some processing to every element of the tree, in direct order.   Note that, in a balanced tree, the element that follows another one that is placed at the bottom hierarchy level belongs to an upper hierarchy level, and vice-versa (see figure 2.1(a)). The algorithm first navigates the tree all the way down from the root up to the tree's bottom-left corner in order to access and process the first element, then directly accesses the second one at an upper hierarchy level by returning from one recursive call and processes it, then navigates again towards the bottom of

---

[5]Note that a tree without root is an empty tree.

[6]We assume that every element is accessed with the same frequency. Taking into account that access frequencies differ from element to element is a more complex problem which is not relevant to our use case.
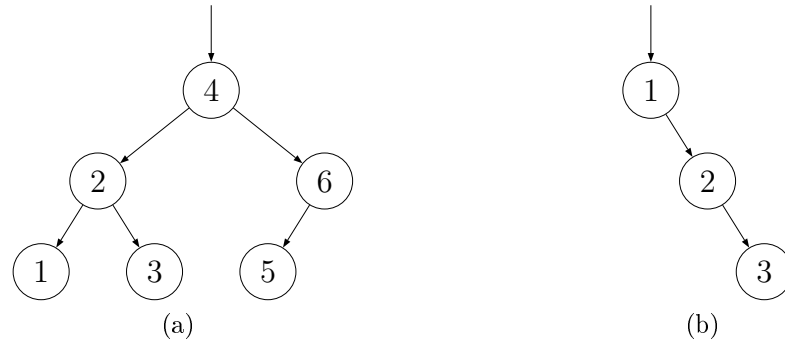
Figure 2.1: At the left, a balanced binary search tree and, at the right, a binary search tree having a sequential structure.

Algorithm 2.2 bst_process_in_order(x)

Require: x to be a BST element
Ensure: every element of the tree or subtree having x as root gets deleted
  1: **if** $x \neq \bot$ **then**
  2:      bst_process_in_order(left(x))
  3:      $y \leftarrow$ right(x)
  4:      process(x)
  5:      bst_process_in_order(right(y))
  6: **end if**

---

the tree and so on up to reaching and processing the last element at the tree's bottom-right corner. In particular, GNU's implementation of the STL uses this algorithm for deleting every element of a set or map.

2.3.2   Iterative traversal

The STL abstracts the actual structures representing sets and maps, thus raw access to the tree elements is not possible and therefore neither it is to perform a recursive traversal. On the contrary, the STL provides a universal mechanism for traversing any kind of container by means of *iterators*.[7] Such iterators are equivalent to pointers towards the container elements.[8] Containers provide methods *begin* and *end* which return, respectively, the iterator towards the first and *past-the-end* elements of the container. The latter element lies after the last element —out of the container— and its sole purpose is to serve as an end-of-sequence mark; therefore this element stores no key/value pair. *Forward* iterators provide post- and pre-increment operators which redirect them towards the next or the previous element, respectively, following the total order implemented by the chosen function object (e.g.: `less<key_type>`, see Josuttis, 1999, sec. 5.9, p. 114). Container elements are commonly accessed in direct order by means of a loop incrementing the iterator returned by the *begin* method up to equaling the one returned by the *end* one. Reverse versions of these methods and iterators are provided for reverse traversals. More information on STL iterators can be found in Josuttis (1999, chaps. 7, p. 220).

In order to efficiently compute either the next or the previous element of another one, the tree structure is modified as follows:

- each tree element is extended with a third pointer towards its parent,

- the pointer to the root of the tree within the tree structure is replaced with a pointer towards the past-the-end element,

- the parent, left and right pointers of the past-the-end element are directed towards the root, first and last elements of the tree, respectively, and

---

[7]Vectors, deques and lists are other containers provided by the STL. More information on STL containers can be found in Josuttis (1999, chap. 6, p. 129).

[8]Indeed, iterators are usually implemented as pointers with custom increment, decrement and dereference operators.

- the parent pointer of the root element is directed towards the past-the-
  end element.

The first modification is needed in order to navigate the tree upwards. The
second and third ones allow for direct access to the past-the-end, root, first
and last elements. The last one is needed for dealing with boundary condi-
tions (e.g.: computing the element after the last one, which results in the
past-the-end element).

  Algorithm 2.3 *bst_ next_ elem* is the one used by GNU's implementation
of the STL in order to find the next element of a BST element x.[9] The
algorithm first verifies whether x has a right child or not; if it does, the
element at the bottom-left corner of x's right subtree is returned as x's next
element. Note that this is true in any case where x has a non-empty right
subtree:

- if x is the root, or is the right child of the root, or can be reached from
  the root by always descending towards the right, elements in the right
  subtree of x are all those elements of the tree whose keys are greater
  than the one of x, and the element at the bottom-left corner of this
  subtree is the one having the least key amongst all of them,

- if x is the left child of an element p, elements in the right subtree of x
  are the ones whose keys are greater than the one of x but less than the
  one of p, the element at the bottom-left corner of this subtree being as
  well the one having the least key amongst them, and

- finally, if p is the left child of an element y and x is the right child of
  p, or x can be reached from p by always descending towards the right,
  the elements at the right subtree of x are those whose keys are greater
  than the one of x but less than the one of y, also being the one of the
  bottom-left element the least one amongst them.

  A simplified version of the last case applies when x has no right subtree
and is not the last element of the tree: since there are no elements between x
and y, y is the next element of x; in other words, the next element of x is the

---

[9]The original GNU C++ code corresponds to method `_Rb_tree_increment` in file
`tree.cc` of the `libstdc++-v3` library. This file can be downloaded from http://gcc.
gnu.org/viewcvs/trunk/libstdc%2B%2B-v3/src/tree.cc?view=co. The terms of use
of this file can be found in the own file header.

lowest ancestor of x in the tree hierarchy such that x lies at the ancestor's left subtree rather than at the right one. The algorithm iteratively navigates the tree upwards as follows: y is initialized as the parent of x and, at each iteration, x is set to y and y to its parent as long as x is a right child of y. Once the process stops, y will be the next element, as long as there is one.

If x is the last element of the tree, the past-the-end element is to be returned. An extra condition could be added before the loop that navigates the tree upwards so that the past-the-end element is returned if x is the left child of the past-the-end element. However, the past-the-end element cannot be accessed from the iterators, since they contain only the pointer towards a tree element but not towards the tree structure: the past-the-end pointer of the tree structure can only be accessed by methods that take the whole tree structure as argument rather than tree elements, such as methods *begin* and *end*. Algorithm *TREE-SUCCESSOR* in Cormen et al. (2001, p. 259) solves the problem as follows: the parent pointer of the root is null and the upwards navigation is stopped once a null parent or the next element of x is reached. In this case, the extra condition (null parent) is placed within the loop performing the upwards navigation which, in turn, would be nested in the loop traversing the whole tree. Rather than evaluating the extra condition a single time, it would be evaluated one or more times for each element of the tree. GNU's implementation of the STL solves this problem by navigating the tree upwards in order to reach the past-the-end element, if necessary, and by placing the extra condition *after* the loop. This condition verifies whether y is the right child of x. Since x will be the left child of y, this will only be possible for the past-the-end and root elements since those are the only ones that are the parents of each other. If x is not the last element of the tree, the next element of x will be reached before reaching the past-the-end element and, therefore, y will not be the right child of its own left child. Otherwise, two situations are possible depending on the presence or absence of a last element other than the root. These situations are illustrated in figure 2.2 by means of a minimal tree for each one, coupled with an execution trace under each tree. The traces contain the values of the relevant variables during the last execution stages, namely:

- x, y and their right children right before the first iteration having x and y as the root and the past-the-end elements, respectively,

- the same variables right before the following iteration, if any, and

- the variable whose value is to be returned at the end of execution.

As we can see, the value returned by the algorithm as the next of the last element is the past-the-end element, in both situations.

Algorithm 2.3 bst_next_elem(T, x)

Input: T, a binary search tree
$\quad\quad\quad$ x, an element of T
Output: returns the next element of x, or past_the_end(T) if $x = $ last(T)
 1: **if** right$(x) \neq \perp$ **then**
 2: $\quad\quad$ $x \leftarrow$ right(x)
 3: $\quad\quad$ **while** left$(x) \neq \perp$ **do**
 4: $\quad\quad\quad$ $x \leftarrow$ left(x)
 5: $\quad\quad$ **end while**
 6: $\quad\quad$ **return** x
 7: **end if**
 8: $y \leftarrow$ parent(x)
 9: **while** right$(y) = x$ **do**
10: $\quad\quad$ $x \leftarrow y$
11: $\quad\quad$ $y \leftarrow$ parent(y)
12: **end while**
13: **if** right$(x) \neq y$ **then**
14: $\quad\quad$ **return** y
15: **end if**
16: **return** x

The iterative traversal has a slight advantage and an important drawback w.r.t. the recursive traversal:

- the first element is directly accessed while the recursive traversal navigates the tree all the way down from the root up to the bottom-left corner, but

- the tree is navigated both upwards and downwards in order to search for the next elements while the recursive traversal stacks the elements at higher hierarchy levels during the downwards traversal so that they can simply be popped out when needed.

Instead of using a past-the-end element as the root's parent, Wein (2005) proposes to use a *before-the-begin* element as the left child of the first element
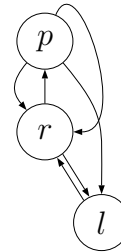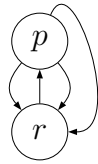
Status at first iteration with $x = r$	
$y = p$	$y = p$
$x = r$	$x = r$
$\text{right}(y) = \text{right}(p) = r = x$	$\text{right}(y) = \text{right}(p) = l \neq x$
	$\text{right}(x) = \text{right}(r) = l \neq y$
Status at next iteration, if any	
$y = r$	
$x = p$	
$\text{right}(y) = \text{right}(r) = \bot \neq x$	
$\text{right}(x) = \text{right}(p) = r = y$	
Returned value	
$x = p$	$y = p$

Figure 2.2: The two boundary cases of algorithm 2.3 *bst_ next_ elem* and their corresponding execution traces once the root is first reached up to the end of execution; p, r, and l stand for past-the-end, root and last, respectively.

as well as a past-the-end element as the right child of the last one. This
modification makes the boundary cases to behave exactly as the other ones;
therefore, it is no longer required to navigate the tree up to the root's parent
in order to retrieve the past-the-end element.

2.3.3   Reverse iterative traversal

As the iterative traversal can be performed by successively searching for the
next element of the current one, the reverse traversal can be done by succes-
sively searching for the previous one. Since BSTs are symmetric, algorithm
bst_previous_elem can be obtained by simply replacing right with left and
vice-versa in algorithm 2.3. In order to study the new boundary cases, the
same replacement is to be done to figure 2.2 as well as replacing the last ele-
ment by the first one. There is only one case that is not present in the normal
traversal and which requires a special treatment: computing the predecessor
of the past-the-end element. Let x be the element whose predecessor element
is to be computed, the algorithm verifies whether x has a left child or not
and, if so, returns the element at the bottom-right corner of the left subtree
of x. The past-the-end element does have a left child, but it is defined as
the first element of the tree. Hence, the algorithm would return either this
element or its the rightmost descendant, if any, instead of the last one. In
order to deal with this situation, GNU's implementation of the STL simply
verifies first whether x is the past-the-end element in order to return the last
element or to proceed as usual. The modification proposed by Wein (2005)
(first mentioned at the end of the previous section) also deals with this par-
ticular case as for any other case: since the past-the-end element is the right
child of the last element and has no children, its previous element is precisely
the last element, the closest ancestor such that the past-the-end element lies
at its right subtree.

2.3.4   Unrolled iterative traversal

Algorithm 2.4 *bst_unrolled_next_elem* is an optimized version of algorithm 2.3
bst_next_elem in which the trivial assignment $x \leftarrow y$ of the last loop has
been removed by *unrolling* the loop. The loop content is doubled so that an
iteration of the unrolled loop performs two iterations of the original one. In-
stead of performing the trivial assignment, the roles of variables x and y are
exchanged during the first half of the loop, and exchanged back during the

second one. Between the two halves, an extra stop condition is inserted which includes the post-processing to be done after the loop but with the exchanged roles of x and y. This kind of optimization is likely to be automatically done by modern compilers depending on the kind of optimization requested (e.g.: we use flag -O3 with the g++ compiler in order to obtain faster code in spite of the increase in size). One can manually code unrolled loops in order to ensure that this optimization is included, but by relying on flags one can generate different executables with the same source code depending on the characteristics and limitations of the targeted platform: while loop unrolling can accelerate the program execution in a desktop computer, the increase in size might not be an option for an embedded device with limited resources. GNU's implementation of the STL does not manually code this loop, thus we have relied on the optimization capabilities of the g++ compiler. We leave manual loop unrolling for a future work. More information on the removal of trivial assignments can be found in Mont-Reynaud (1976), and other examples of application of such technique can be found in Bentley (1982, p. 59). Apart from the removal of trivial assignments, other benefits as well as drawbacks of loop unrolling are discussed in Dongarra and Hinds (1979) and Sarkar (2001). The unrolled version of algorithm *bst_ previous_ elem* can be obtained by following the same procedure, or by simply replacing right with left and vice-versa in algorithm 2.4 *bst_ unrolled_ next_ elem*.

2.3.5 Addition with Knuth's algorithm

Algorithm 2.5 *bst_ knuth_ add* adds a key/value pair (k, v) to a set backed by a BST. This algorithm is a modified version of the one given in Knuth (1998, chap. 6.2.2) which also keeps trace of the root, first and last elements of the tree. The algorithm first checks whether the tree has a root or not, storing the root in variable x. If not, it calls algorithm 2.7 *bst_ add_ root* in order to add (k, v) as the tree's root, then returns pair $(\mathrm{root}(T), \mathrm{true})$. Otherwise, it performs a binary search in a similar fashion than algorithm 2.1 *sorted_ array_ add*. The tree is navigated downwards from the root by using two variables, x and y, storing the current tree element and the previous one, respectively. A third variable c stores the result of the last comparison $k \prec \mathrm{key}(x)$. The current element for the next iteration is either the left or right child of x depending on whether k is less or greater than x's key, respectively. The loop ends once reaching an x having k as key or an y lacking the child that would be the next element. In the former case, the

---

Algorithm 2.4 bst_unrolled_next_elem(T, x)

Input: T, a binary search tree
          x, an element of T
Output: returns the next element of x, or past_the_end(T) if $x = \text{last}(T)$
 1: **if** right(x) $\neq \perp$ **then**
 2:      $x \leftarrow$ right(x)
 3:      **while** left(x) $\neq \perp$ **do**
 4:          $x = \text{left}(x)$
 5:      **end while**
 6:      **return** x
 7: **end if**
 8: $y \leftarrow$ parent(x)
 9: **while** right(y) $= x$ **do**
10:      $x \leftarrow$ parent(y)
11:      **if** right(x) $\neq y$ **then**
12:          **if** right(y) $\neq x$ **then**
13:              **return** x
14:          **end if**
15:          **return** y
16:      **end if**
17:      $y \leftarrow$ parent(x)
18: **end while**
19: **if** right(x) $\neq y$ **then**
20:      **return** y
21: **end if**
22: **return** x

---

algorithm returns pair (x, false) from within the loop, and (k, v) is not added to the tree. In the latter one, algorithm 2.6 *bst_knuth_add_post* is called in order to perform the post-processing after the loop. This post-processing calls either algorithm 2.8 *bst_add_left* or algorithm 2.9 *bst_add_right* in order to create the lacking child of y with (k, v) as key/value pair. The value of variable c determines whether the new element is to be a left or right child since y's key cannot be equal to k at this point.

Algorithms 2.7 *bst_add_root*, 2.8 *bst_add_left* and 2.9 *bst_add_right* first call algorithm 2.10 *bst_create_elem* in order to create the new tree element. This last subroutine simply initializes the key, value and pointer fields to the passed values. Once the element is created, they update the pointers to the root, first and last elements of the tree, whenever necessary:

- a new element added as the tree's root becomes the first and last element as well as the new tree's root,

- a new element added as the left child of the first element becomes the new first element, and

- a new element added as the right child of the last element becomes the new last element.

Provided that the tree is balanced, algorithm 2.5 *bst_knuth_add* has a logarithmic worst-case cost w.r.t. the tree size. However, this algorithm does not ensure that the tree will still be balanced once a new element is added. Indeed, if the tree elements are added in either direct or reverse order then the resulting tree will resemble a double-linked list; for instance, the tree of figure 2.1(b) can be built by adding elements 1, 2 and 3 to the empty tree in that order. Therefore, the worst-case cost of adding an element to a BST will still be proportional to the tree size depending on the order in which the elements are added.

If one was to build a static set or map in order to be just searched rather than modified —for instance, whenever using a dictionary rather than building it— a balanced tree might not be the best option. Depending on the frequency in which the different keys are to be searched, some tree elements should appear at upper hierarchy levels rather than at lower ones (e.g.: the language's most frequent words). An algorithm for the construction of such trees in time n^2 is given in Knuth (1998, p. 436). For the case of our parsing algorithms, sets and maps are built rather than just searched, and key

---

Algorithm 2.5 bst_knuth_add($T, (k, v)$)

Input: T, a binary search tree
        (k, v), the key/value pair to add to the tree
Output: T after adding an element z with key$(z) = k$ and value$(z) = v$, if
    there is no element z' in T having k as key, or T unmodified otherwise
        returns (z, true) in the former case, and (z', false) in the latter one
1: **if** $(x \leftarrow \text{root}(T)) = \perp$ **then**
2:     **return** $(\text{bst\_add\_root}(T, (k, v), \text{true})$
3: **end if**
4: **repeat**
5:     $y \leftarrow x$
6:     **if** $c \leftarrow (k \prec \text{key}(x))$ **then**
7:         $x \leftarrow \text{left}(x)$
8:     **else if** $\text{key}(x) \prec k$ **then**
9:         $x \leftarrow \text{right}(x)$
10:     **else return** (x, false)
11:     **end if**
12: **until** $x = \perp$
13: **return** bst_knuth_add_post($T, (k, v), y, c$)

Algorithm 2.6 bst_knuth_add_post($T, (k, v), y, c$)

Input: T, a binary search tree
        (k, v), the key/value pair to add to the tree
        y, the parent of the new tree element
        c, a Boolean equal to $k \prec \text{key}(y)$
Output: T after adding the new element holding (k, v)
        returns the added element
1: **if** c **then**
2:     **return** $(\text{bst\_add\_left}(T, (k, v), y), \text{true})$
3: **else return** $(\text{bst\_add\_right}(T, (k, v), y), \text{true})$
4: **end if**

Algorithm 2.7 bst_add_root$(T, (k, v))$

Input: T, an empty binary search tree
       (k, v), a key/value pair
Output: T after creating its root
        z, the new tree's root holding (k, v)
 1: $z \leftarrow$ bst_create_elem$((k, v), \text{past\_the\_end}(T), \bot, \bot)$
 2: root$(T) \leftarrow$ first$(T) \leftarrow$ last$(T) \leftarrow z$
 3: first$(T) \leftarrow z$
 4: last$(T) \leftarrow z$

Algorithm 2.8 bst_add_left$(T, (k, v), y)$

Input: T, a binary search tree
       (k, v), a key/value pair
       y, an element of T
Output: T after adding a new element z as left child of y holding (k, v)
        z, the new tree element
 1: $z \leftarrow$ bst_create_elem$((k, v), y, \bot, \bot)$
 2: left$(y) \leftarrow z$
 3: **if** first$(T) = y$ **then**
 4:    first$(T) \leftarrow z$
 5: **end if**

Algorithm 2.9 bst_add_right$(T, (k, v), y)$

Input: T, a binary search tree
       (k, v), a key/value pair
       y, an element of T
Output: T after adding a new element z as right child of y holding (k, v)
        z, the new tree element
 1: $z \leftarrow$ bst_create_elem$((k, v), y, \bot, \bot)$
 2: right$(y) \leftarrow z$
 3: **if** last$(T) = y$ **then**
 4:    last$(T) \leftarrow z$
 5: **end if**

Algorithm 2.10 bst_create_elem($(k, v), p, l, r$)

Input: (k, v), the key/value pair of the new tree element
$\qquad$ p, the parent of the new tree element
$\qquad$ l, the left child of the new tree element
$\qquad$ r, the right child of the new tree element
Output: z, the new tree element
1: key(z) $\leftarrow k$
2: value(z) $\leftarrow v$
3: left(z) $\leftarrow l$
4: right(z) $\leftarrow r$

---

frequencies are unknown; hence, we will consider balanced trees as the ideal case.

2.3.6   Addition with Cormen's algorithm

Algorithm 2.11 *bst_cormen_add* is another algorithm for adding a key/value pair (k, v) to a set backed by a BST, based on algorithm *TREE-INSERT* described in Cormen et al. (2001, p. 261). Cormen's algorithm is conceived for adding an element whose key is new to the set; it is equal to Knuth's algorithm (Knuth, 1998, chap. 6.2.2) without the second test within the binary search loop. However, algorithm *bst_cormen_add* does take into account that k may not be new to the set. Rather than omitting the equality test, it transfers it to the post-processing after the loop, which is performed by algorithm 2.12 *bst_cormen_add_post*. This post-processing is divided into two main cases depending on whether the key of the last tree element stored in y is less than k or not. If it is less, we have the following subcases:

- k is less than any other key within the tree, thus the tree has always been navigated towards the left from the root up to the bottom-left corner. This case is recognized by verifying whether y is the first element of the tree. In this case, a new first element is added as left child of y by means of algorithm 2.13 *bst_add_first*.

- k is less than y's key, but not less than any other key within the tree. If there is an element y' having k as key, the tree will be navigated downwards up to such element. Then, the right child of y' will be

chosen and, since every key within the right subtree of y' will be greater than k, the tree will be navigated downwards up to the bottom by always turning left. Hence, y' is the lowest ancestor of y such that y lies on its right subtree rather than on its left one. Note that this corresponds to the reverse of one of the cases for the computation of the next element of another one. Hence, y' can be retrieved by means of the counterpart of algorithm 2.3 *bst_ next_ elem*, *bst_ previous_ elem* (section 2.3.3, p. 44). Once retrieved, the equality test is finally performed. If the keys are equal, the algorithm returns pair (y', false). If they are not, algorithm 2.14 *bst_ add_ left_ no_ first* is called in order to create a new element z as left child of y, and (z, true) is returned. *bst_ add_ left_ no_ first* is equal to *bst_ add_ first* without verifying whether the new element is the first one or not, since that corresponds to the previous case.

- k is greater or equal than y's key. In this case, the algorithm simply performs the equality test between k and y's key, and either adds or not the new element as the right child of y, depending on the result. Opposite to the previous case, if a new element is added then it must be verified whether it is to become the new last element or not.

This algorithm is the one used by GNU's implementation of the STL, without some minor code factoring in the post-processing part that we have omitted in favor of a more readable code.[10]

Summarizing, this algorithm has one advantage and one drawback w.r.t. algorithm 2.5 *bst_ knuth_ add*:

- the equality test is performed after the binary search loop a single time rather than one time per iteration, but

- when the key to add is already in the set, the algorithm does not stop at the corresponding tree element y' but navigates up to the tree bottom, then comes back to y' in order to perform the equality test.

---

[10]The original C++ code is splitted into methods `_M_insert_unique` and `_M_insert_` of file `stl_tree.h` and the first part of method `_Rb_tree_insert_and_rebalance` in file `tree.cc`. Both files belong to the `libstdc++-v3` library and can be downloaded from http://gcc.gnu.org/viewcvs/trunk/libstdc%2B%2B-v3/src/tree.cc?view=co and http://gcc.gnu.org/viewcvs/trunk/libstdc%2B%2B-v3/include/bits/stl_tree.h?view=co, respectively. The terms of use of these files can be found in their respective headers.

Algorithm 2.11 bst_cormen_add$(T, (k, v))$

Input: T, a binary search tree
         (k, v), the key/value pair to add to the tree
Output: T after adding an element z with key$(z) = k$ and value$(z) = v$, if
    there is no element z' in T having k as key, or T unmodified otherwise
             returns (z, true) in the former case, and (z', false) in the latter one

 1: **if** $(x \leftarrow \text{root}(T)) = \bot$ **then**
 2:     **return** (bst_add_root$(T, (k, v), \text{true})$
 3: **end if**
 4: **repeat**
 5:     $y \leftarrow x$
 6:     **if** $c \leftarrow (k \prec \text{key}(x))$ **then**
 7:        $x \leftarrow \text{left}(x)$
 8:     **else** $x \leftarrow \text{right}(x)$
 9:     **end if**
10: **until** $x = \bot$
11: **return** bst_cormen_add_post$(T, (k, v), y, c)$

We expect *bst_ cormen_ add* to be faster than *bst_ knuth_ add*, in general,
since

- the delayed conditional jump is one of the most expensive operations
  within the binary search loop,

- the search for y' is to be done in less than half of the cases, on the
  average, and

- this extra search will simply add an extra loop with a single conditional
  jump rather than 3 (less/greater/no further children) with a logarith-
  mic cost in the worst case, provided that the tree is balanced. Indeed,
  the average case will have an even smaller cost since the worst case
  will only take place when navigating backwards from the bottom-left
  corner of the root's right subtree up to the root.

2.3.7   Addition with Andersson's algorithm

Andersson (1991) gives an algorithm for searching for an element inside a
BST, rather than for element addition. This algorithm is almost the same

---

Algorithm 2.12 bst_cormen_add_post$(T, (k, v), y, c)$

Input: T, a binary search tree
  (k, v), the key/value pair to add to the tree
  y, the parent of the new tree element
  c, a Boolean equal to $k \prec \mathrm{key}(y)$
Output: T after adding an element z with $\mathrm{key}(z) = k$ and $\mathrm{value}(z) = v$, if
  there is no element z' in T having k as key, or T unmodified otherwise
    returns (z, true) in the former case, and (z', false) in the latter one
 1: **if** c **then**
 2:   **if** $y = \mathrm{first}(T)$ **then**
 3:     **return** $(\mathrm{bst\_add\_first}((k, v), y), \mathrm{true})$
 4:   **else**
 5:     $y' \leftarrow \mathrm{bst\_previous\_elem}(y)$
 6:     **if** $\mathrm{key}(y') \prec k$ **then**
 7:       **return** $(\mathrm{bst\_add\_left\_no\_first}((k, v), y), \mathrm{true})$
 8:     **else return** (y', false)
 9:     **end if**
10:   **end if**
11: **else if** $\mathrm{key}(y) \prec k$ **then**
12:   **return** $\mathrm{bst\_add\_right}((k, v), y)$
13: **else return** (y, false)
14: **end if**

Algorithm 2.13 bst_add_first$(T, (k, v), y)$

Input: T, a binary search tree
  (k, v), a key/value pair
  y, an element of T
Output: T after adding (k, v) as left child of y and first of T
    z, the new tree element holding (k, v)
 1: $z \leftarrow \mathrm{bst\_create\_elem}((k, v), y, \bot, \bot)$
 2: $\mathrm{left}(y) \leftarrow z$
 3: $\mathrm{first}(T) \leftarrow z$

Algorithm 2.14 bst_add_left_no_first$(T, (k, v), y)$

Input: T, a binary search tree
$\qquad$ (k, v), a key/value pair
$\qquad$ y, an element of T
Output: T after adding (k, v) as left child of y but not first of T
$\qquad\quad$ z, the new tree element holding (k, v)
 1: $z \leftarrow$ bst_create_elem$((k, v), y, \bot, \bot)$
 2: left$(y) \leftarrow z$

---

than algorithm 2.11 *bst_cormen_add*: both of them perform a binary search similar to the one performed by algorithm 2.5 *bst_knuth_add*, but omitting the equality test until the search loop is over. Apart from being a pure searcher, the difference consists in the way in which y' is retrieved: while algorithm *bst_cormen_add* walks the tree back in order to retrieve the previous element of y, Andersson's algorithm performs assignment $y' \leftarrow x$ inside the binary search loop each time the key of x is found to be less than the searched key, that is, it keeps track of the last explored element whose key might be equal to the searched one. Once the binary search ends, the algorithm simply uses the value of the precomputed y' instead of searching for the previous element of y. An addition version of Andersson's algorithm can be easily built by performing these modifications to algorithm *bst_cormen_add*.

Due to the differences between Knuth's and Andersson's algorithm, the discussion given in Knuth (1998, p. 436) on the construction of optimal BSTs, taking into account key frequencies, does not apply for the case of Andersson's algorithm. Spuler (1993) discusses the optimal BST for Andersson's algorithm, and gives an algorithm for the construction of such trees in time $n \log n$ rather than n^2. Due to the resemblance between Cormen's and Andersson's algorithms, this work is likely to apply as well to Cormen's algorithm.

Since we have already obtained faster parsing algorithms than those implemented in the Unitex and Outilex systems by using Knuth's and Cormen's addition algorithms, we have not tested the corresponding Andersson's addition algorithm; moreover, we present in section 2.6 another possible optimization of the tree structure which makes unnecessary to compute y'.

2.3.8 Addition with unrolled loops

Trivial assignment $y \leftarrow x$ in all the previous addition algorithms can be removed by unrolling their binary search loops, as shown for the case of algorithm 2.3 *bst_ next_ elem* in section 2.3.4 (p. 44).

Spuler (1992) gives another version of Andersson's binary search algorithm in which the search loop has been unrolled in order to remove the trivial assignment $y' \leftarrow x$. The procedure is similar to the one followed for the construction of algorithm 2.4 *bst_ unrolled_ next_ elem*, but with a small difference: a new loop is embedded inside the original one which has the roles of y' and y exchanged, and a conditional jump redirects the execution flow to the outer loop when the roles are exchanged back.

The addition version of Andersson's algorithm has two trivial assignments that could be removed: $y \leftarrow x$ and $y' \leftarrow x$. Rather than simply exchanging the roles of two variables, multiple combinations of exchanges between the 3 roles $(x, y$ and $y')$ are possible. This problem is quite more complex, requiring a quite greater number of loop versions than simply two, as well as a more complex network of execution flow deviations between the different loops. As stated in the previous section, we present in section 2.6 another optimization of the tree structure which no longer requires to compute y' and, therefore, to perform the second trivial assignment $y' \leftarrow x$.

2.3.9 Addition with a 3-way comparator

Algorithm 2.5 *bst_ knuth_ add* evaluates at each iteration of its binary search loop whether the searched key is less, equal or greater than the key of the current tree element. This operation is called a *3-way comparison*. This operation is emulated by means of two applications of the 'less than' comparator. However, some programming languages provide such operator (e.g.: Perl, Ruby, etc.). Let $<=>$ represent such operator, $a <=> b$ returns a negative value if $a < b$, 0 if $a = b$, and a value greater than 0 if $a > b$. The STL does not provide a generic version of this operator, but it is quite easy to implement. Moreover, if a and b are signed numbers, one can simply compute $a - b$. Anyway, we would still need to verify whether the result is either negative, positive, or null, in order to choose between navigating the tree left, right, or stopping the algorithm execution. This operator becomes of interest when keys are sequences of values to be lexicographically compared, rather than simple values.

In order to compare both the use of the less and 3-way comparators, we first recall how to extend the less comparator for being applied to sequences of elements. Algorithm 2.15 *array_ compare_ less* returns a Boolean indicating whether an array A is lexicographically less than an array B. For the sake of simplicity, the algorithm supposes that B is either shorter or has the same number of elements than A. A previous conditional instruction would be required in order to call the algorithm with B as the shorter array, if necessary. The algorithm emulates the 3-way comparison with two 'less than' comparisons, as for algorithm 2.5 *bst_ knuth_ add*. For each b_i in B, it verifies whether b_i is greater, less, or equal to the corresponding a_i. If it is greater, then the algorithm is to return true, if it is less then it is to return false, and if it is equal then the same procedure is to be performed on the next pair of array elements as long as B has elements left. If B runs out of elements, then either both arrays are equal or A is greater than B, depending on whether A has the same amount of elements than B or not. In either case, A is not less than B, thus false is to be returned.

Algorithm 2.15 array_compare_less(A, B)

Input: $A = a_0 \ldots a_{m-1}$, an array of m elements
$\qquad\quad$ $B = b_0 \ldots b_{n-1}$, an array of n elements such that $n \leq m$
Output: returns a boolean indicating whether A is lexicographically less
$\qquad$ than B

$\quad$ 1: $i \leftarrow 0$
$\quad$ 2: **while** $i \neq n$ **do**
$\quad$ 3: $\quad$ **if** $a_i < b_i$ **then**
$\quad$ 4: $\quad\quad$ **return** true $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad$ $\triangleright\, A < B$
$\quad$ 5: $\quad$ **else if** $b_i < a_i$ **then**
$\quad$ 6: $\quad\quad$ **return** false $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad$ $\triangleright\, A > B$
$\quad$ 7: $\quad$ **end if**
$\quad$ 8: $\quad$ $i \leftarrow i + 1$
$\quad$ 9: **end while**
10: **return** false $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\triangleright\, A \geq B$

Let us suppose that we are to use a 'less than' operator with algorithm 2.5 *bst_ knuth_ add*, and that we are to compare two arrays A and B having a common prefix of length l. We first verify whether A is less than B or not, which requires to perform $2l$ 'less than' comparisons before reaching the pair of elements that differ. If B is not a prefix of A, a last comparison for the

differing elements is needed. In case A is not less than B, we verify whether B is less than A, which doubles the amount of comparisons.

Algorithm 2.16 *array_compare_3w* is the corresponding extension of the 3-way comparator for sequences of signed numbers. At each iteration, it only needs to verify whether the current pair of elements are the ones that differ or not. If so, the result of the subtraction is returned and, if not, the same procedure is repeated for the next pair of elements. Once every element of B has been compared, A will be greater than B if it still has elements left, and equal if not. Subtraction $m - n$ will return a number greater than 0 in the former case, and 0 in the latter one. Note that this algorithm performs a single `if/then` instruction at each iteration instead of two, thanks to the use of the subtraction in order to emulate the 3-way comparison. Assignment $c \leftarrow a_i - b_i$ is not really performed since c is supposed to be the own processor register in which the subtraction result is returned, thus the result is not really copied into a memory position. Once the 3-way comparison is finished, algorithm 2.5 *bst_knuth_add* still requires to verify whether $A \prec B$, $B \prec A$ or $A = B$ in order to either navigate the tree left, right or stop the algorithm execution. However, the compiler is likely to embed the code of the 3-way comparison algorithm within algorithm 2.5 *bst_knuth_add* rather than performing a call; in this case, the `if/then` instruction of the 3-way comparison algorithm will also serve as the equality test of algorithm 2.5 *bst_knuth_add* (whether $\text{key}(x) = \text{key}(y)$ or not), hence algorithm 2.5 *bst_knuth_add* will only require to perform one additional `if/then` instruction after the equality test in order to verify whether $\text{key}(x) \prec \text{key}(y)$ or $\text{key}(y) \prec \text{key}(x)$.

Summarizing, we can expect a performance boost in Knuth's algorithm proportional to the average length of the common prefixes of the compared sequences. Since both Cormen's and Andersson's algorithms perform the equality test a single time after the binary search loop, they may only take advantage of the 3-way comparator a single time instead of once for each loop iteration. We have implemented an equivalent version of the STL sets and maps that use Knuth's algorithm, with and without the 3-way comparator, and compared it with GNU's implementation, which uses the modified version of Cormen's algorithm presented in section 2.3.6 (p. 50). Wein (2005) presents another implementation of the STL sets and maps using the 3-way comparator; however, details on the algorithm used for either searching or adding elements to the trees are not given.

Algorithm 2.16 array_compare_3w(A, B)

Input: $A = a_0 \ldots a_{m-1}$, an array of m signed numbers
          $B = b_0 \ldots b_{n-1}$, an array of n signed numbers such that $n \leq m$
Output: returns a number less than, equal to or greater than zero depending
    on whether A is lexigraphically less, equal or greater than B, respectively
 1: $i \leftarrow 0$
 2: **while** $i \neq n - 1$ **do**
 3:     **if** $(c \leftarrow a_i - b_i) \neq 0$ **then**
 4:         **return** c $\triangleright A \neq B$
 5:     **end if**
 6:     $i \leftarrow i + 1$
 7: **end while**
 8: **return** $m - n$ $\triangleright A \geq B$

2.3.10   Removal

As shown in section 2.3.1, the removal of every element of a BST can be performed by means of an in-order walk. In order to maintain the extra pointers added for the efficient iterative traversal of the tree (section 2.3.2, p. 39), the root pointer is to be given a null value and the pointers to the first and last elements are to be redirected towards the past-the-end element, once the in-order walk is finished. Apart from that, it will not be necessary to verify whether the resulting tree respects or not the well-formedness rules since it it will be empty. Summarizing, we say a BST is well-formed if and only if

- no tree element has more than two children,

- if the tree is not empty, only the root of the tree has no parent (or has the past-the-end as parent, in order to deal with the boundary cases of algorithm 2.3 *bst_ next_ elem* in section 2.3.2),

- the keys of every element at the left subtree of an element x are all less than the key of x, and

- the keys of every element at the right subtree of an element x are all greater than the key of x.

It is only necessary to ensure that, after deleting each element, the remaining ones are still accessible from the resulting tree structure so that every element

can be deleted. Since the in-order walk deletes the tree from the bottom to the top, this restriction is respected, and the deletion of every element can be efficiently performed even if the tree elements have no pointers towards their parents. However, removing a single element from an arbitrary position requires some further processing in order to obtain a well-formed BST. Let z be the element to remove:

- if z has no children, it is only necessary to nullify the corresponding child pointer of z's parent,

- if z has a unique child x, x is to take z's place within the tree structure, and

- if z has two children, a more complex processing is necessary since z's parent cannot adopt both z's children as its own left or right children (one left and one right children are possible, but not two left or two right children).

The first case is trivial. The second one is solved as stated, since:

- if x is a left child of z, the keys of x and every element under x will be less than the key of z and, consequently, less than the key of the parent of z, and

- the same reasoning applies if x is a right child of z but with greater keys instead of lesser ones.

For the third case, it is necessary to search for an alternative element y having at most one child, so that it can be put in the place of z within the tree structure. This element can be, for instance, the next element of z: as stated in section 2.3.2 (p. 39), the next element of an element z having a right child is the element at the bottom-left corner of z's right subtree. Obviously, this element y will have no left child but may have a right child x. Instead of popping z out of the tree structure, we pop y out by following either the first or the second case depending on whether y has a right child or not. Once this has been done, it will sure that y has no children, thus it will be possible to transfer z's parent and children to y. Moreover, the key of y will be less than the keys within the right subtree of z, and greater than the keys within the left subtree of z. Alternatively, the previous element of z can also be put in the place of z. Once the tree structure is rearranged, z can be deleted.

The actual algorithm can be found in Cormen et al. (2001, p. 262). This algorithm has a slight difference w.r.t. the third case of the previous explanation: instead of replacing element z by element y and then deleting z, z is given y's key and y is deleted instead. Note that, if keys are simple values, it will be faster to copy a single key than several pointers. However, this optimization is not compatible with the STL specification since set and map iterators must remain valid until the element they point to is deleted; summarizing, y cannot be deleted instead of z since iterators pointing at y would no longer point to an existent element.

GNU's implementation of the removal operation is based on Cormen's algorithm but replacing z by y instead of simply copying y's key into z. Moreover, the first and last elements of the tree must be updated as follows:

- if z is the root and has no children, the past-the-end element is to become the new first and last element,

- if z is the root and has only a left or a right child, this left or right child is to become either the new last or first element, respectively, and

- if z is not the root and has no left or right child, the parent of z is to become the new first or last element if z is the first or last element, respectively.

We will present in section 2.6 a quite simpler removal method based on the use of BSTs combined with double-linked lists; therefore we will not go deeper into the details of GNU's implementation of the removal operation.[11]

2.4   Self-balancing binary search trees

There exist several variations of BSTs whose addition and removal operations perform some series of rotations on the tree elements so that the tree is also kept more or less balanced; the most popular ones are: AVL trees (Adel'son-Vel'skiĭ and Landis, 1962, named after their inventors), symmetric binary B-trees (Bayer, 1972, rebaptized as red-black trees after Guibas and Sedgewick,

---

[11]The actual GNU's C++ code for element removal is a part of method `_Rb_tree_rebalance_for_erase` defined in file `tree.cc` of the `libstdc++-v3` library. This file can be downloaded from http://gcc.gnu.org/viewcvs/trunk/libstdc%2B %2B-v3/src/tree.cc?view=co. The terms of use of this file can be found in the own file header.

1978), AA trees (Arne Andersson, 1993, named after their inventor) and scapegoat trees (Galperin and Rivest, 1993). Except for the last one, all of them require extending the tree elements with some extra data in order to keep trace of the tree balance status. Pointers to the parent elements are also required in order to navigate the tree upwards, as well as to perform the element rotations. AVL trees extend their elements with the difference between the heights of their left and right subtrees, and rebalance the tree after each element addition or removal so that this difference is always kept between -1 and 1. This ensures that the tree will always be balanced; therefore AVL trees provide the best average search times (assuming that every element within the tree is searched with the same frequency). However, they also perform a greater number of tree restructurings, which may result in worst overall execution times. While adding elements to a BST in direct or reverse order results in sequential trees, adding them in random order tend to result in balanced trees, thus relaxing the rebalance constraints will accelerate the addition operation with slight or no penalization on the search times as long as elements are added in random sequences. Rather than ensuring a minimal tree height at all times, red-black trees ensure that, for every tree element x, the longest sequence of descendants of x is at most twice as long as the shortest one. Rather than a more efficient alternative, AA trees are a simplified version of red-black trees. Rather than having a fixed balance factor, scapegoat trees allow for choosing an unbalance tolerance index α, ranging from sequential trees ($\alpha = 1$) up to fully balanced trees ($\alpha = 0$). Instead of having to add some extra information to each tree element, only 2 integer numbers are to be added to the whole tree structure. In the case of scapegoat trees, the element that is not α-weight-balanced is searched in order to perform the rebalance operation on this element. The element is called the scapegoat, hence the name of the trees. Scapegoat trees with different unbalance tolerance indexes might be an option to consider for a future work. We focus here on red-black trees since those are the ones used by GNU's implementation of sets and maps in the STL. AVL trees do not seem a competitive option; indeed, Lynge (2004) has already tested them against the GNU's implementation, obtaining worst results for every operation.

2.5   Red-black trees

As stated in the previous section, red-black trees are self-balancing trees
which allow for some unbalancing degree. This degree is determined by 3
axioms that hold for every red-black tree:

- every tree element is either red or black (hence the name),

- if y is a child of a red element x, then y's color is black, and

- for every sequence of elements from a given element towards any de-
  scendant having an empty left or right subtree, the number of black
  elements is the same.

In the extreme cases, these axioms ensure that the shortest path from a given
element x towards a descendant having an empty left or right subtree will
be formed by a sequence of n red nodes, and the longest one by an alternate
sequence of n red and n black elements. Therefore, the length of the shortest
path cannot be less than half of the length of the longest one, hence ensuring
a "half" balance factor.

The red-black tree of figure 2.3(a) is a balanced BST equivalent to that
of figure 2.1(a) which can be obtained without performing any rebalance by
executing the following sequence of operations:

- add 4 to the empty tree,

- add 2 and 6 in any order, and

- add 1, 3 and 5 in any order.

The red-black tree of figure 2.3(b) can be obtained by adding the same el-
ements but in ascending order; note that the resulting tree is a partially
balanced tree which respects the 3 previously stated axioms.

Since red-black trees are themselves BSTs, the former algorithms on BSTs
also apply to red-black trees. However, after either adding or removing and
element, it is necessary to check whether the red-black tree axioms are still
respected and, if not, the tree is to be restructured. A detailed explanation
on how to perform these operations as well as the corresponding algorithms
is given in Cormen et al. (2001, chap. 13). For the case of red-black trees,
Cormen et al. make use of a *sentinel* element: instead of using null pointers
in order to represent a missing left or right child, a special element called the

(a)

MediumPicture

(b)
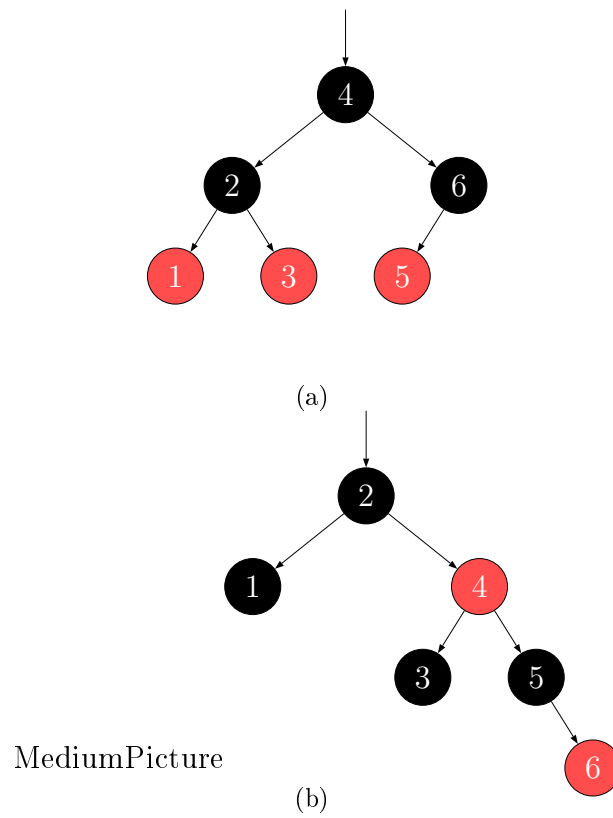
Figure 2.3: At the left, a balanced red-black tree and, at the right, a partially-balanced red-black tree.

sentinel is used in order to explicitly represent the empty subtrees. Since
the key of such element is of no use, elements having one or two empty
subtrees are all linked to the same sentinel element instead of having a dif-
ferent sentinel for each empty subtree. The sentinel allows for dealing with
boundary cases as one would with normal ones, since it is no longer required
to verify whether the child of an element exists or not before accessing it.
The tree structure must be augmented with such an extra element, which
in fact can perfectly be the past-the-end element, and a pointer towards the
sentinel must be stored inside the tree structure so that the algorithms can
verify whether an element's child is the sentinel or not. However, the imple-
mentation of the STL iterators conflicts with the use of sentinels: algorithms
operating on iterators rather than on whole tree structures do not have access
to the sentinel's pointer since iterators are implemented as pointers towards
tree elements rather than tree structures; therefore it is necessary to use a
fixed value such as the null pointer in order to represent the empty subtrees.

2.6   Double-linked red-black trees

As stated at the beginning of this chapter, some of the parsing algorithms that
we will present in this dissertation search and remove every useless partial
parse before starting to combine them in order to build the set of parses.
At a first implementation version, we first built a simplified version of the
sets of partial parses in the form of a single double-linked list, then threw
away the previous sets of partial parses and finally performed the removal
of useless partial parses from the list. Since no more elements were to be
added to the set of partial parses, and this set structure already ensured
that every contained partial parse was unique, the double-linked lists were
preferred due to their more efficient removal operation. Although appending
an element to a double-linked lists is a fast operation, this procedure wasted
time due to the duplicated construction and destruction of each partial parse.
In order to avoid this situation, we decided to perform all the treatment on
the original set structures. The execution times were lowered, but not as
much as expected: the removal operation was not only restructuring the tree
in order to respect the BST axioms, but it was also rebalancing the tree. This
rebalancing was unnecessary since the remaining operations to perform on
the trees involved only iterative traversals, but no search for specific elements.
Finally, we considered an even more radical solution: we extended each tree

element with two pointers linking the element with its previous and next elements (in the set order), obtaining a hybrid red-black tree and double-linked list structure, and then removed the useless elements by treating the structure as a simple double-linked list, without caring whether the BST axioms were still respected or not. The resulting structure still allows for its traversal, both for building the set of total parses and, afterwards, for completely deleting the set of partial parses.

Mixing both red-black tree and double-linked list structures had already been proposed by Das et al. (2008), but for a different purpose: accelerating the applications making an intensive use of set and map iterators. In fact, we had not considered the possible speedup due to the faster access to the next and previous elements of another one; this speedup has not only improved the algorithms removing useless partial parses but every algorithm building sets or maps of partial parses.

The added pointers towards the neighbours of each tree element not only allow for a faster implementation of the sets and maps, but for several simplifications of the different algorithms, namely:

- algorithm 2.3 *bst_next_elem* is reduced to simply accessing the pointer towards the next element and, conversely, the corresponding algorithm *bst_previous_elem* just requires to access the pointer towards the previous element,

- the implementation of the STL iterators no longer conflicts with the use of a sentinel element, since the retrieval of the previous or next element of another one no longer requires to navigate the tree down up to reaching the sentinel but to simply follow the pointer towards the previous or the next element,

- rather than using the left and right pointers of the past-the-end element in order to have direct access to the first and last elements of the tree, its previous and next pointers are used as for any other element of the tree; hence, when adding or removing an element so that the first or last elements are to be updated, no special operation is to be performed since the new first or last element will be simply relinked with the past-the-end element, in the same manner that it would happen when adding or removing elements in the middle of the list,

- algorithm 2.11 *bst_cormen_add* no longer requires to navigate the tree upwards in order to find the previous element of the last visited one for

performing the equality test, since the pointer to the previous element can simply be followed,

- Andersson's algorithm no longer requires to keep track of the last visited element whose key may be equal to the searched one, since the pointer towards the previous element can also be followed as for the previous case; indeed, both algorithms *bst_ cormen_ add* and the addition version of Andersson's algorithm become the same algorithm after this simplification,

- this simplified version of Andersson's algorithm no longer performs two trivial assignments but only one, thus unrolling its binary search loop in order to remove the remaining trivial assignment is as easy as for the other algorithms and, finally,

- the removal of an element no longer requires to navigate the tree down in order to find the next element of the one to remove, but to follow the pointer towards the next element.

We have implemented a set/map library equivalent to the one provided by the GNU's STL implementation —in order to facilitate their comparison— but using double-linked red-black trees with algorithm 2.5 *bst_ knuth_ add* instead of standard red-black trees with algorithm 2.11 *bst_ cormen_ add*. This alternate implementation has finally allowed us to make competitive algorithms out of the parsers we propose in this dissertation. We expect to improve them further with algorithm 2.11 *bst_ cormen_ add* in a future work.

2.7   Other structures

We briefly present here other structures for the implementation of sets and maps other than BSTs or BSTs ensuring some balance factor. Some of these structures may be worth to be tested in future implementations.

2.7.1   Treaps

Treaps (Seidel and Aragon, 1996, contraction of tree and heap) are BSTs which exploit the fact that randomly adding elements to a tree tends to result in balanced trees, rather than forcefully ensuring some balance factor. As elements are added to the treap, they are given a random *priority*. BST's

addition and removal algorithms are modified so that, once concluded, the priority of every element within the treap is greater than the one of its children. This results in exactly the same tree that would have been obtained by adding the elements in their priority order. As for red-black trees, a double-linked version of this structure might also perform well.

2.7.2 Splay trees

Rather than trying to accelerate the retrieval of arbitrary tree elements, splay trees (Sleator and Tarjan, 1985) restructure the tree so that the most frequently accessed elements are located at higher hierarchy levels. All normal operations on a BST are combined with one basic operation called splaying (hence the name): splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. Compared with the other self-balancing structures, they have two main drawbacks: they allow for sequential trees, and search operations are more expensive since they involve restructuring the tree in order to raise the most frequently searched elements within the tree hierarchy. Since our parsing algorithms add new elements to the sets and maps rather than repeatedly accessing some of the already added ones, we do not consider splay trees appropriate for our use case.

2.7.3 2-3 trees

2-3 trees are another kind of non-binary self-balancing search tree: they allow for elements having either zero, two or three children, hence the name. Elements having no children —also called *leaves*— may have one or two keys. From the point of view of BSTs, leaves having two keys are equivalent to two BST leaves which lie attached at the same hierarchy level, where the key of the left element is less than the one of the right one. Elements having two children are structured in the same manner than the ones of BSTs: they have a single key which must be greater than the key of their left children and less than the one of their right ones. Elements with 3 children have two attached keys k and k' rather than simply one. Let l, m and r be the keys of the left, middle and right child, respectively, such elements must hold that

$$l < k \leq m \leq k' < r. \tag{2.1}$$

After each addition and removal, every 2-3 tree leaf lies at the tree's bottom hierarchy level. This implies a "perfect" balance independently of the number of elements within the tree, which is possible thanks to the use of the different kind of tree elements.[12] 2-3 trees were introduced by Hopcroft in 1970, though not published (Cormen et al., 2001, p. 300). A description of 2-3 trees as well as of its management algorithms can be found in Aho et al. (1974, sec. 5.4).

2.7.4   2-3-4 trees

2-3-4 trees, also called 2-4 trees, are a slightly more complex version of 2-3 trees where elements having 4 children and 3 keys are also possible. Indeed, 2-4 and red-black trees are isometric structures as for 2-3 and AA trees.[13] Since we have already tested red-black trees and either 2-3, 2-4 and AA trees are structures analogous to red-black trees, we rather consider other alternative structures for a future work.

2.7.5   B-trees

B-trees (Bayer, 1972) are a generalization of 2-3 and 2-3-4 trees: leaves may have one, two or more keys, and non-leaves may have $n > 0$ keys $k_1 \ldots k_n$, in which case they have $n + 1$ children $x_1 \ldots x_n + 1$ such that

$$\text{key}(x_1) < k_1 < \text{key}(x_2) < k_2 < \ldots < \text{key}(x_n) < k_n < \text{key}(x_{n+1}). \qquad (2.2)$$

Note that the tree height decreases when using elements with a greater number of children and keys. B-trees were conceived for minimizing the number of input/output accesses to secondary storage devices (e.g.: hard disks) and have been widely used in database systems. Extensive information on B-trees can be found in Cormen et al. (2001, chap. 18). B-trees have already been used in an implementation of STL sets and maps (Hansen and Henriksen, 2001); the performances obtained w.r.t. red-black trees were better for searches, similar for additions and considerably worse for removals. A B-tree version combined with a double-linked list might perform better than double-linked red-black trees, since element removal would rather use the double-linked list structure rather than the B-tree one.

---

[12]Note that BSTs can only be perfectly balanced when having either 0 or 2^n elements, with $n \geq 0$.

[13]Recall that AA trees are a simplified version of red-black trees.

2.7.6 Hash tables

Hash tables are an array-based alternative to search trees. Let us suppose that the number of every possible key identifying the elements to store within the set is rather big in comparison with the number of elements that will be added to the hash table. We allocate an array bigger than needed for storing all the elements to add, and use a *hashing function* in order to compute the array position where to add each element. This function usually consists in some short sequence of cheap arithmetic/bitwise operations on the element's key. Efficiency highly depends on whether the hash function provides a uniform distribution of the elements along the array in order to minimize *element collisions*; such collisions take place each time a new element to add is given an already occupied array position. In such cases, multiple techniques might be used in order to solve the collision (e.g.: using a second hashing function). This implies that, whenever retrieving a previously added element, it does not suffice to compute the element's position, but also to verify that both the searched key and the one of the retrieved element match. Of course, by using bigger arrays than necessary one can minimize the number of collisions, though more memory will be wasted. As drawback, elements within hash tables are not necessarily kept in order, which prevents the use of further optimizations such as the one that we will present in section 2.8. More information on hash tables, including some historical background, can be found in Knuth (1998, sec. 6.4).

2.7.7 Skip lists

Skip lists (Pugh, 1990) are sorted linked lists where each element may not only be linked to its next element but to other elements several positions ahead. These pointers allow for skipping some elements during the traversal of the list, hence the name. The number of ahead pointers for each element to insert is randomly chosen such that the number of elements having i pointers decreases exponentially w.r.t. i. Searches are performed in a similar manner than for BSTs, though starting from the first element instead of the "middle" one (the root of the tree): for each visited element, a proper ahead pointer is followed so that the search space is reduced to the element subsequence between the last visited element and the next element to visit. The average search times are kept logarithmic, as for BSTs, while the total number of pointers increases proportionally to the list size. Opposite to double-linked

lists, no backwards pointers are stored, so only forward traversal is possible. Since they are already lists, there is no need to add extra pointers as done with red-black trees in order to combine both linked-list and tree structures. However, backward pointers are used by some of the parsing algorithms, apart from being useful for the optimization of some of the BST algorithms; whether their use could be avoided or adding such pointers to the skip lists would not result in an important loss of efficiency will require a further study. Seidel and Aragon (1996) state that both skip lists and treaps have similar performances; therefore both of them are interesting candidates for the optimization of our parsing algorithms.

2.7.8   Concurrent access structures

Personal computer technology has lately focused on increasing the number of processing cores —up to 6 cores nowadays— attached within a single chip rather than on augmenting their clock frequencies. A performance gain could be achieved by concurrently executing the independent subtasks that compose our parsing algorithms, taking advantage of multiple cores. However, parallel algorithms are more complex than their non-parallel counterparts, which we have not fully exploited yet. Moreover, it must be taken into account that the speedup factor that can be achieved by means of parallel computing is less than the available number of processors or processing cores. Hence, parallel computing will become an interesting option once the remaining alternatives will not be able to yield greater speedup factors. Be that as it may, a straightforward way of making parallel versions out of our parsing algorithms could consist in concurrently exploring several grammar rules in parallel, whenever multiple grammar rules apply to the same sentence fragment.[14] Since the exploration of such rules results in the addition of partial parses to some series of sets and maps, the main problem would consist in figuring out how to efficiently integrate concurrent set and map structures into our parsing algorithms. The list of works on concurrent versions of the set and map structures discussed in this chapter is quite extensive; we conclude this section with a small selection among them: Larsen et al., 2001 (self-balancing BSTs in general), Bronson et al., 2010 (AVL trees), Paul et al., 1983 (2-3 trees), Bender et al., 2005 (B-trees), Hanke, 1999 (red-black trees), Herlihy et al., 2006 (skip lists) and Triplett et al., 2010 (hash tables).

---

[14]This is the usual situation due to the ambiguous nature of natural languages.

2.8 Maps of keys to sets

Let us suppose that we are to implement a map M of keys in K to sets of keys in K'. One could simply use a BST structure with key/value pairs in $K \times \mathcal{P}(K')$, where elements in $\mathcal{P}(K')$ are also represented by other BST structures with keys in K'. Let (k, S) be a key/value pair where S is a set of keys in K'; if we are to add an element k' to S, we are to perform a binary search on M in order to retrieve S, then another one in S and finally rebalance S. If the map did not exist, we are to add a map k to a newly created empty set S and rebalance M and, finally, add k' to S. Moreover, in order to delete M's data structure once it is no longer needed, we are to traverse M and, for each pair (k, S) found, we are to traverse S as well in order to delete the whole structure. Indeed, the reverse procedure is performed in order to build the map, though decomposed into several element additions. It is possible to avoid the overhead caused by the use of several BST structures by using a single BST structure representing a set in $K \times K'$ rather than a map $K \to \mathcal{P}(K')$. For instance, if we are to represent map

$$\{a\, M\, \{x, y\}, b\, M\, \{z, t\}\}, \tag{2.3}$$

we rather emulate such structure by means of set

$$\{(a, x), (a, y), (b, z), (b, t)\}. \tag{2.4}$$

We redefine the key comparator in order to lexicographically compare pairs of keys in (K, K') as two-letter sequences having keys in K as first letter. In order to iterate over the elements of a set mapped by key k, we search for the first element with key k, then sequentially access the next elements until reaching the past-the-end element or an element with a key other than k. In contrast with the first solution, keys in K are repeated and some extra key comparisons are also to be performed. However, we have obtained shorter execution times in every parsing algorithm using maps of keys to sets implemented as sets of key/value pairs.

In general, we have found that minimizing the number of requests for free memory segments (dynamic memory allocations) has given better results in spite of the added repetition: rather than creating STL containers of pointers to other dynamically allocated structures, it is usually more efficient to let the container own its own elements.[15] Note that memory allocations involve

---

[15] By owning an element we mean to have the responsibility of allocating and deallocating

to search for a big-enough memory segment, which becomes more and more expensive as memory gets fragmented, and both memory allocations and deallocations involve to update the memory allocation table. A container of pointers will require two memory allocations for each contained element: one for the own element's structure and another for the pointer to add to the container. Containers of pointers are to be used whenever the element structures are to be shared among several containers; anyway, it is better to let one container have the ownership, and the others to take care only of the pointers but not of the pointed memory segments. Containers of Boost's shared smart pointers have also been tested,[16] obtaining quite worst results: such pointers involve to manage a counter of pointers to the same element so that the element is deallocated upon the deletion of the last remaining pointer.

Why would anyone use containers of pointers rather than containers of actual elements? The original STL containers use copy semantics, meaning that whenever adding an element to a container, a copy of such element is added rather than granting the container the ownership of the element to add. Containers of complex structures may require to perform expensive copies of elements that may just have been created for being added to the container, thus being destroyed right after the copy is added to the container. One could use some special pointer containers that become responsible for the deallocation of the memory pointed by the added pointers (e.g.: Boost's pointer containers). However, this involves two memory allocations and deallocations for each added element, as stated before. Another solution provided by the new C++ standard, C++0x,[17] is the use of *move semantics*: instead of copying the whole complex structure, only the main pointers governed by the structure are copied, and the original structure is left in an "empty" state for its deletion. For instance, an STL vector is basically a pointer to an array plus an element counter; rather than copying the whole array, only

---

the memory occupied by the element; properly defining each element's ownership is vital for avoiding both segmentation fault errors as well as memory leaks: the former take place whenever following a pointer whose pointed memory has not been allocated first, and the latter whenever the pointers used to access some memory segment are deleted before deallocating the pointed memory.

[16]Search for `shared_ptr` at the homepage of the Boost C++ libraries: http://www.boost.org

[17]Visit http://www2.research.att.com/ bs/C++0xFAQ.html for an overview of the new features offered by C++0x.

the pointer and the counter are copied and both of them are nullified in the original structure. Anyway, we have not needed to use such complex structures for the implementation of the algorithms presented in this dissertation, and we have preferred not to use the new C++0x features until its definitive establishment.[18]

Another optimization to take into account is the so called *RVO* or *return value optimization*: whenever assigning to a newly created object A the result B of a function F, the compiler is allowed to use B as A rather than copying B into A and then deleting B. Moreover, if B is to be passed to a function G as parameter C, the compiler is allowed directly use B as if it was C. Indeed, the g++ compiler actually performs these optimizations, ignoring any extra code that might be included within the copy constructor apart from the own object copying (e.g.: `printf` instructions). However, elements are yet systematically copied whenever added to STL containers, even when the added elements are temporal objects. The use of move semantics is the standard procedure to avoid such copies, performing shallow copies instead.

2.9 Multisets and multimaps

Multisets and multimaps are sets and maps that may contain several elements having the same key. The same data structures and algorithms presented here for the representation and management of sets and maps can be used for the case of multisets and multimaps, but with a slight difference: the equality test in the addition algorithms is to be removed so that, whenever the key to add is equal to some key k' within the tree, it is treated as if it was greater than k'; this implies that

- adding a key/value pair (k, v) always results in increasing the tree size by one element,

- tree elements having the same key appear at consecutive positions, from the sequential point of view, and

- whenever adding a key/value pair (k, v) such that k is already within the tree, the new element is added as the next element of the last one having k as key.

---

[18]Up to now, support for the new C++0x features provided by the g++ compiler is yet considered experimental; visit http://gcc.gnu.org/projects/cxx0x.html for a list of the currently supported features.

Wein (2005) gives an alternative implementation of the STL multisets and multimaps, also based in red-black trees, which supports two additional operations: split and catenation; splitting a set S by a key k results in two disjoint sets S_1 and S_2 such that S_1 contains all the elements of S whose keys are less than k, and S_2 contains all the other elements, and the catenation is the inverse operation.

Chapter 3

Character treatment

From the point of view of a computer, a text is a sequence of binary digits representing a sequence of characters, where the correspondence between digits and characters is given by a *character encoding system*. Text processing would be greatly simplified if there would be a unique and universal character encoding system; though work on such a universal character encoding is already quite advanced (The Unicode Consortium, 2007), there exist many other encoding systems which are not intended to be universal. Computer users are usually unaware of the existence of character encodings, since the default character encoding given by the operating system is transparently used. Problems arise when trying to open a text file created in a different computer with a different character encoding system. For the case of web pages, the recommendation is to include some meta-information specifying the employed character encoding; however, this recommendation is not always followed. In that case, there is no choice but to try different character encodings until the characters are properly rendered. If our character encoding is not completely different than the appropriate one, one may still be able to recognize the language in which the text was written and limit the search to the character encodings for that particular language. Be as it may be, the automatic processing of such texts will still be obstructed, if not prevented.

Character encodings are not only used for representing characters inside text files, but also inside the computer's memory as part of the data processed by a computer program. Depending on the language or languages to treat, and whether the encoding is to be used for data storage or for text processing, certain character encodings will be more appropriate than others. In our case, Telefónica imposed a character encoding system for data transmission

over the Internet, but we are to choose a character encoding system for both storing and processing linguistic data. We briefly present here the most relevant, namely Unicode and those from which it evolved, and some implementation details surrounding our choice.

3.1   ASCII

The American Standard Code for Information Interchange (ASCII, Gorn et al., 1963) is one of the first widely used character encoding systems, and the basis of many others; it is restricted to English and encodes each character as a sequence of 7 bits plus one bit, either used for data transmission control (parity bit) or simply left unused. Apart from the uppercase and lowercase English letters, and some other symbols (e.g.: numbers, arithmetic operators, punctuation symbols, etc.), ASCII includes 32 characters that do not represent printable information but text formatting marks and/or actions to be executed by devices such as printers or screens; for instance, the bell character (number 7) was used for alerting operators of an incoming message by means of an audible sound. Most of those characters are no longer used, but we still keep many of its text formatting marks, also called blank or white characters, namely: the white space (number 32) for separating words, the horizontal tabulation (code 9) for separating columns, and the line feed (number 10) for marking the end-of-line. Some systems (e.g.: Windows platforms) precede the line feed by a carriage return (number 13), and others (e.g. Unix and Linux platforms) use the line feed alone.[1] More details on ASCII, along with an evolving view of character encoding systems up to ASCII, can be found in Fischer (2000).

3.2   ISO-8859-x

ISO-8859-x is nowadays a family of 16 character encodings extending ASCII (x is a number between 1 and 16): the eighth bit is used in order to represent additional characters not present in English; for instance, ISO-8859-1 encodes

---

[1]The carriage return is the mechanism in a typewriter that pushes the cylinder on which the paper is held towards one side in order to start a new line; on a computer text interface, the carriage return moves the cursor towards the beginning of the current line, and can be used without a line feed in order to rewrite the current line of text, for instance for creating some text-based animation such as a progress bar.

additional characters for Western European languages such Spanish (ñ, á), French (ç, è), German (ß, ö), etc. ISO-8859-1 (ISO/IEC, 1998), also called *Latin 1*, was the default character encoding for the web.

3.3 Unicode

With the coming of globalization, Unicode (The Unicode Consortium, 2007) became the new web character encoding standard, though it still coexists with many others. Unicode covers most of the existing writing systems, including symbols of a great variety of domains such as mathematics ($\sum$, ∞), economy (€, £), culture (✝, ☯) and many others. Generically, Unicode maps integers between 0 and 1,114,111 to characters. Each integer is called a *code point*, and code points are grouped into 17 *planes* of 2^{16} code points each one. Plane 0 is called the *basic multilingual plane* or BMP: it comprises characters from most modern languages as well as a large number of special characters, including ISO-8859-1 as the first 256 code points. Unicode is constantly growing: around 100,000 characters have already been mapped, having occupied only the 10% of the available space. Though intended to be universal, there is not a unique Unicode character encoding form but several. They either use a fixed or variable amount of bytes in order to represent each code point, where the minimum amount of bytes is called a *code unit*. We briefly describe below the main Unicode character encoding forms.

3.3.1 UCS-2

UCS-2 uses 2 bytes per code unit, and each code unit corresponds to a code point. It encodes only the basic multilingual plane. UCS-2 files start with code point 0xFEFF, the so-called *byte order mark* or BOM. The file endianness can be deduced by verifying whether the BOM bytes are transposed (little endianness) or not (big endianness).[2]

---

[2]The terms big- and little-endian refer to two possible ways of laying out bytes in memory or transmitting them through a serial connection: starting from the most significant byte (big-endian) or from the least one (little-endian). These terms were introduced by Cohen (1981), who borrowed them from the satyric novel 'Gulliver's Travels' (Swift, 1726); in the novel, Lilliputians are divided into two religious factions: those who prefer cracking open their soft-boiled eggs from the little end, and those who prefer the big end (the Big-endian heretics).

3.3.2   UTF-16

UTF-16-LE & UTF-16-BE are 2 extension of UCS-2 also using 2 bytes per code unit, but either 1 or 2 code units per code point. UTF-16-LE imposes little endianness, while UTF-16-BE imposes big endianness; hence, the BOM is optional, though recommended for backwards compatibility. Code points of the basic multilingual plane are serialized as for UCS-2. Unused UCS-2 code units are used in pairs, forming the so-called *surrogate pairs*, in order to represent characters beyond plane 0. UCS-2 is usually mistaken for UTF-16 since they do not differ as long as surrogates are not required, and UCS-2 has been abandoned in favor of UTF-16.

3.3.3   UTF-32

UTF-32-LE & UTF-32-BE are the 32 bit versions of UTF-16-LE and UTF-16-BE. These are the only Unicode schemes representing every Unicode code point with a fixed amount of bytes, 4 to be exact. However, they usually imply a memory waste since the basic multilingual plane is enough in most cases. Fixed-length encoding forms are useful for randomly accessing single characters within strings, since a simple addition an multiplication is enough for computing their position. However, one can better use the UTF-16 encoding forms without loss of efficiency as long as surrogates are not required.

3.3.4   UTF-8

UTF-8 is a variable-length character encoding form, using 8 bits per code unit and 1 to 4 code units per code point. UTF-8 not only covers every Unicode code point but is also the only Unicode scheme fully compatible with ASCII, since ASCII characters are represented with a single byte. This implies that both encodings will yield the same byte sequences as long as no characters outside ASCII are used. The byte order mark is not needed here since code units are one-byte long. For the case of English and Western European languages, UTF-8 is preferred for data storage an transfer rather than for text processing: common characters take less space but accessing a character within a string requires to sequentially compute and add the lengths of every preceding character (unless it is previously known that every string character is coded with the same amount of bytes, such as the ones in the ASCII subset). Indeed, UTF-8 is the current character encoding standard

for the web. An important property of UTF-8 is that multi-byte code points may not contain other code points. Moreover, a null byte may also be used for terminating UTF-8 strings, such as in C and C++. Hence, UTF-8 strings may be treated as normal 1-byte char strings whenever comparing them for equality. Furthermore, if one is to build a set of UTF-8 strings with some implementation requiring an arbitrary total order (e.g.: binary search trees), a lexicographical per-byte order can be efficiently applied. Linguistically sorting a set of strings is a more complex problem: since not every language defines the same sorting rules, a universal character ordering is not possible. For instance, the letter after e in English (and ASCII) is f, but letter $é$ is placed between e and f in Spanish; to be exact, letters with and without acute accents are considered equal, unless the acute accents are the only distinctive traits between two words (e.g.: *ame* and *amé*).[3,4] Thus, *ame* comes before *amé*, but *amé* comes before *amerizar* (to land on the sea).

3.4 Implementation

Unicode aims to facilitate the exchange of text data by homogenizing character representation; however, the presence of multiple Unicode encoding forms has allowed for a heterogeneity of implementations between different platforms and programming languages. It is not surprising to find titles in the literature such as "*Unicode encoding forms: A devil in disguise?*" (Biswas, 2003). Among the different interoperability issues mentioned in the paper, we are mainly concerned by the C++ standard directives on Unicode support: a wide char type —as well as wide string and stream types— are to be used in order to represent Unicode characters longer than one byte, but each C++ compiler is free to assume a different Unicode encoding form. As a matter of fact, the g++ compiler on a Linux platform uses 4 bytes in order to represent

---

[3]Indeed, a vowel with an acute accent is still the same vowel: the acute accent simply marks the word's stressed syllable. That is not the tilde's case: n and $ñ$ are considered different letters and have a different pronunciation.

[4]*ame* and *amé* are inflected forms of verb *amar* or "to love"; opposite to conventional dictionaries, electronic dictionaries are not only to contain the infinitive forms but every inflected form.

wide chars, while the MinGW port of g++ to Windows uses only 2 bytes.[5,6]
If one is to write portable C++ code using Unicode, third party libraries are
to be used; the IBM ICU library is an open-source example of portable Uni-
code library, both available for C/C++ and Java.[7] The Outilex system uses
UTF-8 and the ICU library in order to represent and compare strings. In
our case, we have decided to reuse Unitex's Unicode libraries, both for com-
patibility with the Unitex system as well as for focusing on parsing rather
than on character encoding issues. These libraries mainly use UTF-16LE,
both for string processing as well as for textual data storage, and have been
tested in different platforms and computer architectures, including little- and
big-endian's. Both grammar and dictionary files provided with the Unitex
system are encoded with UTF16-LE, though they can be easily re-encoded if
necessary, for instance by using the GNU `iconv` tool.[8] Since we are mainly
to work on Spanish and other Western European languages, we simply ignore
the existence of surrogate pairs an treat every character as a single two-byte
code unit.

3.4.1   Exchanging characters between Java and C++

Another interoperability issue described in Biswas (2003) that concerns us
is the exchange of text data between Java and C++ programs: one of the
requirements given by Telefónica for the use of our NLP engine was that it
should be accessible through the Internet as a Java servlet inside a Tomcat
servlet container,[9,10] and characters should be received and transmitted as
UTF-8 streams. Servlets are Java programs that answer HTTP requests,
usually by returning a dynamically built web page. In our case, the servlet
receives UTF-8 streams corresponding to request sentences, and returns a
plain UTF-8 text containing the result returned by the NLP engine. An
immediate solution would have been to implement our NLP engine in Java,

---

[5]g++ is one of the compilers of the GNU Compiler Collection (GCC). More information
can be found in the GCC homepage: http://gcc.gnu.org/

[6]More information on MinGW can be found in its official homepage: http//www.
mingw.org

[7]More information on the ICU library can be found in the ICU project homepage
http://site.icu-project.org/.

[8]http://www.gnu.org/software/libiconv/documentation/libiconv/iconv.1.html

[9]More information and tutorials on servlets can be found in http://java.sun.com

[10]More information on Tomcat can be found in the Apache Tomcat homepage http://
tomcat.apache.org and in Brittain and Darwin (2007)

since Java provides a good Unicode support and it is easier not to mix different programming languages; however, this would have had an impact in performance, since Java code is to be executed in a virtual machine rather than compiled into the native code of the machine where it is to be executed. Moreover, we are also interested in the precise manner in which our objects are deallocated from the computer memory rather than on delegating this task to the Java garbage collector: our parsing time measures include the deallocation costs, since more complex parsers use more complex data structures which require more expensive deallocation methods. We have programmed a small Java servlet which simply interfaces Telefónica's MovistarBot with our C++ NLP engine. The servlet invokes the NLP engine through the Java Native Interface (JNI),[11] and performs the necessary character encoding translations from UTF-8 to UTF-16LE and vice-versa.

3.4.2 Character normalization

Finally, a last requirement given by Telefónica was to ignore every diacritic mark, except for the tilde in *ñ*, as well as to make no distinction between uppercase and lowercase letters. Omission of the diacritic marks is one of the most common orthographic mistakes in written Spanish. Moreover, diacritic marks are usually omitted for convenience when communicating by means of sort text messages: Spanish keyboards do not have separate keys for letters with diacritic marks (except for letter ñ) but an extra key must be pressed in order to add the diacritic mark. As usual, the same applies to uppercase letters: two keys must be combined in order to obtain the uppercase version of a letter. Since surrogate pairs are not needed for the representation of Spanish characters, we have built a look-up table mapping every single UTF-16LE code unit with its corresponding normalized version, that is, the corresponding lowercase letter without diacritic marks or the same code unit, if already normalized.[12] Rather than creating a normalized copy of each user sentence, we apply the look-up table on-the-fly during the grammar application, since user sentences are to be discarded once treated. Moreover, user sentences are kept unmodified so that original sentence fragments can be returned if necessary; for instance, in "*envía hola Paco al 555*" (send hello Paco to the

---

[11]See Liang (1999) for a comprehensive book on JNI.

[12]The correspondence between Unicode lowercase and uppercase letters, as well as letters with and without diacritic marks, can be extracted from http://www.unicode.org/Public/UNIDATA/CaseFolding.txt

555) the user is requesting to send the SMS "hola Paco" to phone number
555. Normalized copies of grammar and dictionary files are constructed since
they are to be applied to each user sentence.

Chapter 4

DELAF dictionaries

Monolingual lexicons (or vocabularies) for language processing are one of the ways of automatically annotating words with formalized linguistic information. As compared to statistic methods, the use of lexicons provides more control on the results. As compared to other language processing lexicons, those with the DELA format offer convenient functionality for update thanks to an automatic inflection mechanism.[1]

DELAF dictionaries (Silberztein, 1993) are a kind of electronic dictionaries whose purpose is to provide a set of unambiguous identifiers for each use of each simple word of a natural language, as well as to provide information inherent to each one. These properties implicitly define classes of words (e.g.: verbs, nouns, adjectives, etc.).[2] Grammar development can be greatly simplified by making reference to these classes instead of explicitly stating the corresponding list of words (the exact mechanism will be explained in chapter 6). Moreover, separating the information inherent to lexical units from the grammar rules results in a better structured approach. DELAF dictionaries for several languages are freely distributed with the Unitex platform under the LGPL-LR license.[3] We have adopted the DELAF formalism in order to keep the compatibility with the Unitex system, and used the Spanish DELAF (Blanco, 2000) freely distributed with Unitex for the MovistarBot

---

[1]DELA stands for *Dictionnaires Életroniques du LADL* or LADL's Electronic Dictionaries for inflected forms, where LADL is the *Laboratoire d'Automatique Documentaire et Linguistique*.

[2]'F' in DELAF stands for *formes fléchies* or inflected forms.

[3]The terms and conditions of the LGPL-LR license can be found in http://igm.univ-mlv.fr/~unitex/lgpllr.html

use case (section 1.2, p. 6). In particular, this dictionary describes more than 775.000 lexical units.

We first give a set of relevant definitions in section 4.1, and then describe the DELAF formalism in section 4.2. Next, we present our DELAF implementations in section 4.3. We describe the modifications we have introduced in the dictionaries for adapting them to the MovistarBot use case in section 4.4, and present the tools we have developed for automating some processes on DELAF dictionaries in section 4.5. Finally, we mention other electronic dictionaries in section 4.6.

4.1   Definitions

Definition 1 (Use of a word). *We call* use of a word *a context of utilization of the word. Such contexts vary depending on the different meanings of the word.*

Definition 2 (Surface form). *The surface form of a word is the exact sequence of characters that form it.*

Definition 3 (Semantic class). *A word semantic class is a set of words defined by semantic criteria which uniquely apply to the properties inherent to each word, for instance the set of 'human' words (e.g.: student, friend, inhabitant, etc).*

Definition 4 (Semantic feature). *A word semantic feature is a word property that determines whether the word belongs to a semantic class or not, for instance feature 'human'.*

Definition 5 (Part-of-speech). *The part-of-speech of a word, also called lexical or word category, is the semantic feature that determines the syntactic role the word plays inside a sentence (e.g.: verb, noun, adjective, etc.).*

Definition 6 (Inflected form). *A word's inflected form is the particular modification the word has undergone in order to express a particular case, gender, number, tense, person, mood and/or voice.*

Definition 7 (Inflectional feature). *A word's inflectional feature is a particular characteristic expressed by an inflected form.*

The set of inflectional features of a given word depends on the language and the word's part-of-speech; for instance, in Spanish, verb inflection comprises tense but not gender, adjective inflection comprises gender but not tense, and prepositions have no inflectional features (they are invariant). Opposite to Spanish, some Polish verb tenses also comprise gender (e.g.: '*był*', '*była*' and '*było*' correspond to 'he was', 'she was' and 'it was', respectively).

Definition 8 (Lemma). *The lemma, canonical form or dictionary form of a word is the inflected form of the word that is used in order to refer to the whole set of possible inflected forms.*

For instance, verb lemmas in English, French and Spanish are the infinitive forms, in Basque are the participles and in Bulgarian and Latin are the first person singular of the indicative present. Lemmas are also called the dictionary forms since conventional dictionaries include only such forms. For the case of electronic dictionaries, such as the DELAFs, every possible inflected form is to be included.

Definition 9 (Inflectional paradigm). *The inflectional paradigm of a given word is its set of every possible inflected form.*

For instance, 'color' and 'colors' form the inflectional paradigm of word 'color', with 'color' as the lemma, and 'texture' and 'textures' form the inflectional paradigm of word 'texture', with 'texture' as the lemma.

Definition 10 (Inflectional model). *An inflectional model is a set of rules or procedures common to a set of lemmas describing how to construct their inflectional paradigms.*

For instance, the inflectional model of lemmas 'color' and 'texture' states that the singular form is the lemma and that the plural form is built by appending 's' to the lemma ('colors' and 'textures'). In Spanish, '*color*' and '*textura*' do not share the same the same inflectional model: while the plural of '*textura*' is built as in English ('*texturas*'), the plural of '*color*' is built by appending 'es' to the lemma ('*colores*').

Definition 11 (Inflectional class). *An inflectional class is composed by the set of lemmas sharing the same inflectional model.*

For instance, Spanish first-conjugation regular verbs form an inflectional class. Inflectional classes containing a single lemma are also possible: for instance, irregular verb 'to go' in either English, French (*aller*) or Spanish (*ir*).

Definition 12 (Lexical unit). *A lexical unit is a surface form coupled with a lemma and a set of semantic and inflectional features which unambiguously identify a particular use and inflected form of the surface form.*

Definition 13 (Ambiguous word). *We say a word is ambiguous iff its surface form is shared among several lexical units, that is, the word has either multiple uses or meanings or corresponds to multiple inflected forms.*

The main purpose of lexical units is to unambiguously identify the words of the language. The set of properties added to the surface form depends on the ones taken into account by the dictionary. We have defined here lexical units given by DELAF dictionaries, but other dictionaries may give other sets of properties.

4.2   Description

DELAF dictionaries are text files listing a set of lexical units, arranged into lines. Opposite to conventional dictionaries, DELAF dictionaries do not include word definitions but describe every possible inflected form of the words. Dictionary lines, or entries, look as follows:

```
envía,enviar.V+Trans_msg:P3s
envía,enviar.V+Trans_msg:Y2s
```

Each entry is composed by the following data:

- surface form (‘*envía*’ = ‘he sends’, present indicative, or ‘send, you’, imperative) terminated by a comma,

- lemma (‘*enviar*’ = ‘to send’) terminated by a period,

- one or more semantic feature identifiers separated by plus symbols, where the first semantic feature corresponds to the part-of-speech (`V` = verb, `Trans_msg` = synonyms of ‘to send’ in the context of sending an SMS, for instance ‘to transmit’) and the last identifier is followed by a colon, and

- a sequence of characters identifying the inflected form of the word (`P3s` = present indicative, third person, singular; `Y2s` = imperative, second person, singular).

In order to avoid redundancy, lexical units sharing all properties except their inflectional features are compressed into a single line: one or more sequences of inflectional features may be specified, each one separated from the previous one by a colon; for instance, the two given entries for surface form *envía* are rather described in a single line as follows:

```
envía,enviar.V+Trans_msg:P3s:Y2s
```

As well, the lemma is omitted when it is equal to the surface form; for instance, the entry for the infinitive form of '*envía*' looks as follows (W = infinitive):

```
enviar,.V+Trans_msg:W
```

Both semantic and inflectional feature identifiers are case sensitive. Semantic feature identifiers are composed by an uppercase letter followed by zero, one or more letters, digits or underscores. Inflectional feature identifiers are composed by a single uppercase letter, lowercase letter or digit ('P' means 'present indicative tense', while 'p' means 'plural').

Both parts-of-speech and inflectional features are more or less fixed for each particular language; most common parts-of-speech considered in the Spanish DELAF, along with their identifiers, are: verb (V), noun (N), pronoun (PRON), determiner (DET), adjective (ADJ), adverb (ADV), preposition (PREP), conjunction (CONJ) and interjection (INTJ). The inflectional features, along with their identifiers, are:

- indicative tenses: present (P), imperfect (I), preterit (J), future (F) and conditional (C),

- subjunctive tenses: present (S), imperfect with -ra ending (T), imperfect with -se ending (Q) and future (R),

- other verbal forms: infinitive (W), gerund (G), past participle (K) and imperative (Y),

- genders: masculine (m), feminine (f), neutral (n),

- persons: first (1), second (2), third (3), and

- numbers: singular (s), plural (p).

4.3   Implementation

The main operation performed on DELAF dictionaries consists in searching for a particular surface form in order to retrieve the corresponding lexical units. Hence, we implement these dictionaries as maps of surface forms to sets of lexical units. Some data structures for the representation of maps where already presented in chapter 2. However, other data structures are more appropriate for mapping sequences rather than simple data.

4.3.1   Tries

In a first implementation version, we reused a trie C++ class we had already programmed for optimizing the representation and management of sequences (this optimization, along with a formal definition of tries, will be given in chapter 9). Briefly, tries (Fredkin, 1960) are a kind of search trees where each trie element corresponds to a unique prefix within the set of prefixes of the represented set of sequences. This correspondence is as follows:

- the root represents the empty prefix, and

- the children of an element representing a prefix α represent prefixes $\alpha\sigma_1$, $\alpha\sigma_2$, etc., where σ_i is a letter that is unique for each child.

Additionally, each trie element is extended with a pointer towards the corresponding set of lexical units. In case the trie element does not correspond to a complete word, the pointer is null.

This implementation not only allows for searching for surface forms and their corresponding lexical units, but also for programmatically adding, removing and/or modifying DELAF entries, as well as for saving the changes in DELAF text format. Tries allow for an efficient retrieval of the properties associated to a given surface form. However, DELAF text files are large (e.g.: 32.6 MB for the case of the Spanish DELAF dictionary), and loading them into a trie data structure takes a few seconds. In the MovistarBot use case (section 1.2, 6), the dictionary and grammar are loaded upon the reception of the first user sentence. Once the sentence analysis is finished, these data structures are kept in memory in order to be reused by later analyses. Hence, only the first analysis will be delayed due to data loading. However, while modifying and testing the system, we are required to reload the dictionaries many times, adding up those few seconds each time. Moreover, tries are not

the most compact representation of electronic dictionaries, though current average computers have memory sizes in the order of gigabytes rather than megabytes.

4.3.2 Minimal acyclic automata

Dominique Revuz studied during its PhD thesis (Revuz, 1991) compression techniques for DELAF dictionaries which would allow to load entire DELAF dictionaries in a personal computer's RAM, while keeping short searching times. Note that average computers of that moment had around 8 MB of RAM, against the more than 30 MB of DELAF text files. As result, he proposed to represent dictionaries as minimal acyclic automata and presented an efficient algorithm for the minimization of these machines (Revuz, 1992). While tries factor out common prefixes, minimal acyclic automata also factor out common suffixes. Hence, states within the automata may not only correspond to a unique surface form, such as within tries, but to multiple surface forms. If the automaton were to be fully minimized, searching a surface form within them may not only lead to their corresponding lexical units, but also to the lexical units corresponding to other surface forms sharing any non-empty suffix. In order to solve this situation, surface forms mapped to different sets of lexical units are regarded as having different endings, hence their suffixes will not be mixed together. Up to here, the resulting automata are not different than tries. However, the surface form is not stored within the lexical units, and the lemma is not fully stored: the lemma is replaced by a code indicating how to modify the ending of the surface form in order to obtain the lemma (e.g.: code '2in' for surface form 'begun' indicates that the last two characters should be replaced by 'in', obtaining the lemma 'begin'). As result, suffixes of surface forms belonging to the same inflectional class, though having different lemmas, may be factored out; for instance, surface forms 'loves', 'comes' and 'stores' are all mapped to the same lexical unit, `1.V:P3s`, and their suffix 'es' is factored out. This technique allows for a greater compression ratio than with tries for the case of inflectional languages, such as English, French, and specially Spanish. However, the addition and subtraction of dictionary words is no longer straightforward.

The Unitex platform provides a tool for the conversion of DELAF text files into a compressed format, based on these minimal acyclic automata, and uses the compressed format for retrieving the lexical units corresponding to the surface forms in the texts to analyze. This format is described in Paumier

(2008, sec. 12.8, p. 262). Following this description, we have programmed
a C++ compressed dictionary class which is able to interpret this format as
an automata for its application. The compressed dictionary file is loaded
as is, without requiring any reformatting as for the case of the text format
(e.g.: building a trie). As result, the time required for loading a compressed
dictionary is virtually imperceptible. As drawback, this format does not
support modifications on the dictionaries but only for the retrieval of lexical
units. Modifications are to be done on the text DELAF, which must be then
compressed again for its use.

The Unitex platform contains a set of DELAF dictionaries in compressed
format for several languages —freely distributed under the LGPL-LR license—
and a tool for converting DELAF text files into compressed ones.[4] However,
Unitex did not have a tool for reverting these dictionaries back to text for-
mat, hence these dictionaries could not be modified.[5] Text dictionaries were
to be either downloaded, if available, or requested to the Unitex author. Fol-
lowing the routine for the serialization of a trie into a DELAF text file, we
also implemented a routine for the serialization of minimal acyclic automata,
that is, reverting compressed DELAFs back to the text format. In our case,
we primarily developed such routine for verifying the correctness of our com-
pressed dictionary implementation: compressing and decompressing back a
text dictionary should not introduce any changes.

4.3.3   Alternative implementations

Ciura and Deorowicz (2001) give an alternative algorithm for the optimal
construction of minimal acyclic finite-state automata. Daciuk et al. (2000)
give algorithms for the management of minimal acyclic finite-state automata
so that words can be added and subtracted directly on this compressed rep-
resentation; Carrasco and Forcada (2002) extend these algorithms in order
to support minimal finite-state automata with cycles. Daciuk et al. (2005)
implement dynamic perfect hashing with finite-state automata in order to
allow for a full minimisation of the dictionary, as well as to add new entries
to the compressed dictionary without having to decompress and compress it
again. Hash tables have been briefly discussed in section 2.7.6, p. 69. Dy-
namic perfect hashing is a particular kind of hashing which ensures that every

---

[4]The terms and conditions of the LGPL-LR license can be found in http://www.gnu.
org/licenses/gpl.html

[5]Unitex's Uncompress tool is available since version 2.1

indexed element is given a unique index, and that updates can be performed efficiently. Moreover, minimal perfect hash functions ensure that indexes are consecutive integer numbers (e.g.: each word of a dictionary considering n words is mapped to a unique integer number between 0 and $n - 1$).

4.4 DELAF extensions

As stated before, parts-of-speech and inflectional features are more or less fixed, depending on the language. We say 'more or less fixed' since one may still add new parts-of-speech and inflectional features in order to deal with special cases; for instance, one may consider the contracted form 'al' (to the) as two lexical units, preposition 'a' (to) and determiner 'el' (the), or consider it as a single lexical unit with a special part-of-speech 'preposition-determiner' (PREPDET in the Spanish DELAF). Be as it may be, lexical units must be unique: the lemma coupled with the set of semantic features identify the use of the surface form, and together with the surface form and inflectional features the lexical unit is uniquely identified; for instance, the following extract of the English DELAF dictionary (Courtois, 2004) consists in four entries describing four possible uses of word 'lay', and a total of 14 lexical units:

```
lay,.A
lay,.N:s
lay,.V:W:1Ps:2Ps:1Pp:2Pp:3Pp
lay,lie.V+I:1Is:2Is:3Is:1Ip:2Ip:3Ip
```

The uses are:

- adjective (A) 'lay': non-specialist,

- noun (N) 'lay': romance,

- verb (V) 'lay', either infinitive (W) or present tense (P) and, in the latter case, either in first (1) or second (2) person, singular (s), or in first (1), second (2) or third (3), plural (p): place, put, prepare, etc.,

- intransitive (I) verb (V) 'lie', simple past tense (I), any person, any number: to be or to stay at rest in a horizontal position.

Whether to write one or more entries for the same surface form depends on the level of granularity we want to achieve; if a particular application requires to distinguish among nt meanings of a single entry, it suffices to create the semantic classes that identify each meaning, and write a separate entry with different semantic codes.

In contrast with the set of parts-of-speech, the set of semantic classes of a DELAF dictionary is open: one is expected to add the semantic classes that ease the definition of grammars for a particular application; for instance, we have added semantic classes for pronouns and determiners depending on the distance of the referred object:

- close to the speaker (`D1`): *este*, *esta*, *estos*, *estas*, etc. (this, these),

- close to the listener (`D2`): *ese*, *esa*, *esos*, *esas*, etc. (that, those), and

- far from both the speaker and the listener (`D3`): *aquel*, *aquella*, *aquellos*, *aquellas*, etc. (also translated as 'that' and 'those').

We have also defined the class of verbs for the *transmission of messages*, `Trans_msg` (*enviar*, *mandar*, *transmitir*, *comunicar*, etc). When describing the sentences requesting to send an SMS, it is only necessary to specify this semantic class rather than the full verb list. Due to the amount of different inflected forms of verbs in Spanish, this semantic class contains hundreds of words.

It has to be noted that the first DELAF dictionary was written for French (Courtois, 1990); writing DELAF dictionaries for other languages may require to adapt the DELAF formalism in order to cover other lexical phenomena not present in French, such as enclitic pronouns in Spanish. These pronouns appear attached to the end of verbs; for instance, '*léemelo*' is composed by verb '*lee*' (read) and pronouns '*me*' (to me) and '*lo*' (it). We have omitted every information concerning enclitic pronouns and treated the Spanish DELAF as the French one since the coverage on enclitic pronouns was quite incomplete. It has to be noted that enclitic pronouns are commonly used in Spanish when formulating requests, for instance '*muéstrame los juegos que tienes*' (show me the games you have). Treatment of these forms is discussed in the next chapter.

4.5 DELAF tools

In order to ease the analysis and extension of DELAF dictionaries, as well as for normalizing the dictionary characters, we have developed a set of 3 DELAF tools we describe below.

4.5.1 Analysis

The first tool extracts every dictionary entry matching at least one dictionary-based lexical mask of a given set of masks. These lexical masks are predicates that apply on the properties of the lexical units, and will be the object of chapter 6; for instance, lexical mask `DET+D1:m` can be used with this tool in order to extract any determiner with distance `D1` and gender masculine. We have used the tool for verifying the dictionary coverage on subsets of the language; for instance, we extracted the list of determiners and found that the poetic forms were missing ('*aqueste*', '*aquestos*', *aquese*, etc). Conversely, these tools can be used for verifying whether a lexical mask matches the expected entries or not.

4.5.2 Extension

The second tool is a modified version of the first one: instead of extracting the entries, it alters the set of semantic features of the matched entries. We have used this tool for adding new semantic classes to the Spanish DELAF. For instance, in order to add the semantic class '`Trans_msg`' we have first built a set of lexical masks matching every inflected form of the verbs for the transmission of messages (587 entries, the ones corresponding to verbs with a particular lemma); then, we have used this tool for adding the semantic feature '`Trans_msg`' to the matched entries. Special attention must be paid here since an important amount of dictionary entries could be modified by mistake. The first tool can be used here for verifying the lexical mask correctness. In case of mistake, this second tool can be also used for reverting the changes, since it allows for both removing and adding semantic features.

4.5.3 Normalization

The third tool normalizes the dictionary characters, as explained in section 3.4.2, p. 81. This tool also merges sets of dictionary entries when their

normalization results in the same entry, except for their inflectional features; for instance, entries for 'I love' and 'he/she loved'

$$\begin{array}{l} \texttt{amo,amar.V:P1s} \\ \texttt{amó,amar.V:J3s} \end{array} \quad \text{become} \quad \texttt{amo,amar.V:P1s:J3s,}$$

and so on for the same inflected forms of every regular verb of the first conjugation.

4.6   Other electronic dictionaries

DELAS dictionaries are similar to DELAF ones, but rather than explicitly specifying each inflected form and their corresponding lists of inflectional features, only lemmas coupled with their semantic features are specified, and the first semantic feature identifies both the part-of-speech and the inflectional class the lemma belongs to (e.g.: N4 corresponds to the fourth inflectional class of nouns).[6] The DELAS format is conceived for the construction and maintenance of electronic dictionaries, and the DELAF for their use by computer programs. Unitex is able to build the corresponding DELAF from a DELAS dictionary and a formal description of the different inflectional models. Probably, it will be more efficient to add new semantic classes within the DELAS dictionary and then generate the corresponding DELAF than directly modify the DELAF with the tool we have presented. However, open-source Spanish DELAS dictionaries are not available and, though possible, it will be more difficult to implement a tool for reversing the DELAF-to-DELAS transformation than implementing the tool we have proposed for altering the DELAF dictionaries directly, which has proved to be enough for our use case.

DELAC and DELACF dictionaries are the equivalent to DELAS and DE-LAF dictionaries but for compound words.[7] Currently, Unitex makes no distinction between simple and compound DELA formats, allowing to mix both kind of forms in a single dictionary. References to the DELAS and DELAF formalisms within the Unitex manual subsume the corresponding DELAC and DELACF ones. In our case, we have strictly supported the original DE-LAF format since the Spanish DELAF contains no compound words and,

---

[6]DELAS stands for (*DELA de formes simples* or simple forms)

[7]Letter 'C' in DELAC and DELACF stands for 'composées' or 'compound'.

anyway, our use case comprises only a few compound forms which we have simply coded within the grammar rules (e.g.: '*teléfono móvil*' or 'mobile phone'). DELAF and DELAS descriptions including compound words can be found in Paumier (2008, chap. 3, p. 49).

Though work on the Spanish DELA dictionaries has continued, newer versions have been distributed only in binary format along with the Intex system (Silberztein, 2004), and under a restrictive license forbidding their free use by either private or public organizations (without the author's consent). Currently, Intex development has been discontinued in favor of NooJ (Silberztein, 2003b), an evolved version of Intex. This evolution has also affected the different DELA dictionary formats, which are now all integrated within a single format (Silberztein, 2005b). Newer versions of the Spanish DELA are being distributed in NooJ's binary format and under the same terms than with the Intex system.

Apart from compound words, there exist other linguistic objects formed by multiple words and susceptible to be regarded as lexical units, such as complex terms and named entities. These linguistic objects, along with compound words, are all referred under the term *multi-word lexical unit* (MWLU), and may present and important degree of flexibility broader than simple inflection (e.g.: birth date/date of birth, hereditary disease/genetic disease, etc.). Rather than giving support to compound words only, we may rather consider more general frameworks supporting any kind of multi-word units. Currently, there exist a multiplicity of such frameworks (Savary, 2008); in particular, Multiflex is a formalism and a tool that copes with the flexibility and idiosyncrasy of multi-word units (Savary, 2009). Unitex includes a Multiflex version (see Paumier, 2008, chap. 10, p. 193), which is being distributed along with Unitex under GNU's LGPL license.[8]

Other Spanish dictionaries are being freely distributed under GNU's GPL license, along with the Apertium system.[9] These dictionaries follow the XML format described in Forcada et al. (2010, sec. 3.1.2, p. 20), and can be easily reformatted in order to convey the DELAF specification. Apart from monolingual dictionaries, Apertium includes other kind of dictionaries for the automatic translation between language pairs.

---

[8]The terms and conditions of GNU's LGPL license can be found in `http://www.gnu.org/copyleft/lesser.html`

[9]The terms and conditions of GNU's GPL license can be found in `http://www.gnu.org/licenses/gpl.html`

Chapter 5

Tokenization

Tokenization, or text segmentation, is the process by which character sequences are split into tokens or information units, which must correspond in some way with the words described in the lexicon (the DELAF dictionaries, in our case). In natural language processing (NLP), it makes more sense to consider words and symbols as information units rather than characters alone. Tokenization is the first stage of our NLP engine, and usually of any natural language processor. Since we are rather interested in the optimization of the algorithms of application of local grammars, the tokenization process we have implemented is very simple, yet enough for our use case. This process is a simplified version of the one followed by the Unitex (Paumier, 2008, sec. 2.5.4, p. 43) and Outilex (Blanc and Constant, 2006b, sec. 6.1, p. 23) systems, with some particular implementation features for better accommodating our use case. We describe the basic tokenization rules in 5.1, and its implementation in section 5.2. We briefly discuss the problem of lexical ambiguity (as for definition 13, p. 86), and how we have treated it in section 5.3. Furthermore, we shortly describe in section 5.3.2 the mechanism followed by the Unitex and Outilex systems for the representation and resolution of lexical ambiguity.

As stated before, this chapter does not intend to give a comprehensive view on lexical processing (indeed, tokenization is only a part of lexical processing). An alternative open-source lexical processor to the ones found in the Unitex and Outilex systems can be found in the Apertium system (see Forcada et al., 2010, chap. 3, p. 17). Opposite to Unitex, Apertium includes a particular treatment of Spanish enclitic pronouns; it must be noted that, while the first treated languages by the Unitex system were French and

English, the first ones treated in Apertium were Spanish and several other languages that are also spoken in Spain, such as Catalan. As well, while Unitex's approach is rather conservative (upon unresolved ambiguity, every interpretation is presumably accepted), Apertium's approach is rather selective: the most likely translation between pairs of languages is to be returned instead of the set of every possible translation; as result, Apertium includes a part-of-speech tagger based on statistical data, which is neither present in Unitex. It must be noted that, while Apertium's objective is the automatic translation between pairs of languages, Unitex is a tool for language analysis and information extraction.

Treatment of non-catenative (e.g. Arabic) and agglutinative languages (e.g. Basque, Korean, etc.) is more complex and beyond the scope of this work. Up to now, Apertium's treatment of such languages is a known limitation (Forcada et al., 2009). As a particular case, Unitex includes a tokenizer an a tagger for Korean texts (see Paumier, 2008, sec. 7.7, p. 174) which were conceived and developed by Huh (2005).

5.1   Description

The tokenization rules we have followed are similar to the ones of the Unitex and Outilex systems:

- we distinguish between word tokens and symbol tokens, where word tokens are sequences of letter characters, and symbol tokens are single symbol characters;

- word tokens are separated from adjacent word tokens by one or more blank characters,[1]

- symbol tokens may or may not be blank-separated from other tokens, and

- blanks are not tokens but token separators.

Following these rules, multi-words such as '*teórico-práctico*' (theoretical/practical) are segmented into multiple tokens, and attached words such as '*envíamelo*' are segmented as single tokens (*envía* + *me* + *lo* = send + to/for

---

[1]Blank characters have been described in section 3.1, p. 76

me + it). Hence, tokens correspond to surface forms of lexical units of simple words only. Blank characters are not considered tokens but token separators, and symbols are segmented as single tokens.

5.2 Implementation

We build a linked list of token structures, where each structure is composed by two pointers and an integer; the pointers delimit the token within the input character sequence, an the integer identifies the token type. As for Intex, Unitex and Outilex, we identify the following token types, based on their character types:

- symbol,

- digit symbol,

- punctuation symbol,

- neither digit or punctuation symbol (defined implicitly),

- word,

- uppercase word,

- lowercase word,

- proper noun word (first letter uppercase, the others lowercase), and

- neither uppercase, lowercase or proper noun word (defined implicitly).

Note that some token types subsume others (e.g.: 'symbol' subsumes all the other symbol types), while others are mutually exclusive (e.g.: 'symbol' and 'word'). Bitwise identifiers allow for an efficient representation and comparison of token type identifiers: one bit codes whether the type is 'symbol' or 'word', and two other bits code the 'symbol' and 'word' subtypes.

During the tokenization process, we do not only check whether the characters are either letters, symbols or blanks, but also whether they are digit, punctuation or other kind of symbols, and whether they are uppercase or lowercase letters. The character type is determined by means of a look-up table. This way, we efficiently compute the type of the next input token while

searching for its rightmost character. Higher levels of treatment deal with the token sequence rather than with raw characters. Since blanks are not tokens, blanks are implicitly omitted while iterating over the token sequence; the presence of blanks between two tokens can yet be detected by checking whether the tokens share one bound or not.

The Unitex and Outilex platforms perform some text normalization before tokenizing it, which includes replacing blank sequences by single blanks. The tokenization result is written into a file in some text format, then this file is read by higher levels of treatment. This procedure is appropriate for the linguistic study of texts, where different grammars are to be applied to the same text and partial results are to be examined. In our case, the same grammar is to be concurrently applied a single time to each user sentence as they are received, and only the final result is to be returned; hence, tokenization and any text normalization is to be performed *on the fly*.

Since our NLP engine is to be invoked for the analysis of single sentences (instant messaging communication is sentence-based), we do not implement any procedure for the segmentation of texts into sentences. Intex, Unitex and Outilex insert sentence delimiter tags by applying a RTN with output. Since our NLP engine also supports such kind of RTNs, the same sentence segmentation procedure could be easily implemented, if required.

More information on the particular text preprocessing, tokenization and segmentation into sentences carried out by Unitex can be found in Paumier (2008, sec. 2.5, p. 37), by Outilex in Blanc and Constant (2006b, chap. 6, p. 23), and by Intex in Silberztein (2004, chap. 10, p. 97).

5.3   Treating lexical ambiguity

Whether the same token may have different interpretations is trivially taken into account while applying the grammar: if the grammar requires the next token to be a verb, and one of the token interpretations corresponds to a verb, the token is assumed to be a verb; moreover, multiple grammar rules to be applied on the same token but having incompatible requirements will all be followed as long as the token has at least the same amount of interpretations, each one complying with the requirements of one of the rules.

Lexical ambiguity is rather treated at sentence level than at token level: every possible sentence interpretation is efficiently computed in some compressed format, factoring out common parts, then only the one derived from

the application of the most precise grammar rules is decompressed and returned. The exact procedure is based on weighted RTNs and FPRTNs, and will be explained in chapter 18. This procedure is new to Intex, Unitex and Outilex systems: Intex and Unitex do not support weighted RTNs, and Outilex's algorithm of application of weighted RTNs has an exponential worst-case cost rather than polynomial, as we have managed thanks to the use of FPRTNs; moreover, Outilex does not provide a mechanism for the automatic definition of grammar weights, though the one we have used is not hard to implement.

5.3.1 Multiple segmentations

Multiwords may give rise to multiple segmentations: for instance, '*cinturón negro*' (black belt) may correspond to either two lexical units (a belt that is black) or to a single one (a master degree in martial arts). Though Spanish is not an agglutinative language, enclitic pronouns attached to verbs are very common. Such forms also give rise to multiple segmentation possibilities, though this is not very frequent: for instance, '*dáte + lo*' (date it) and '*dá + te + lo*' (give it to yourself), '*correos*' (post office) and '*corre + os*' (move over [you all]), '*pésame*' (condolences) and '*pésa + me*' (weight [the oranges] for me), etc. More common cases appear when omitting diacritic marks, such as '*tomate*' (tomato) and '*tóma + te*' (have yourself [a drink/some vacation/etc.]), and '*leales*' (plural of loyal) and '*léa + les*' (read them [their rights]). As in French, there exist a few contracted words, but they are not ambiguous: for instance, '*al = a + el*' (to the, '*au = à + le*' in French) and '*del = de + el*' (of the, '*des = de + les*' in French). Since our use case comprises only a few multiwords, attached words and contractions, we have simply coded them inside the grammar rules: multiwords are coded as sequences of lexical units, and contractions and attached words are treated as single lexical units.

5.3.2 Text automata and ELAG grammars

Intex, Unitex and Outilex platforms perform a second tokenization level based on the application of electronic dictionaries. This procedure could also be applied in our case in order to build a structure of lexical units rather than a simple token sequence. They build what they call a *text automaton*: an acyclic FSA recognizing every possible sequence of interpretations of the
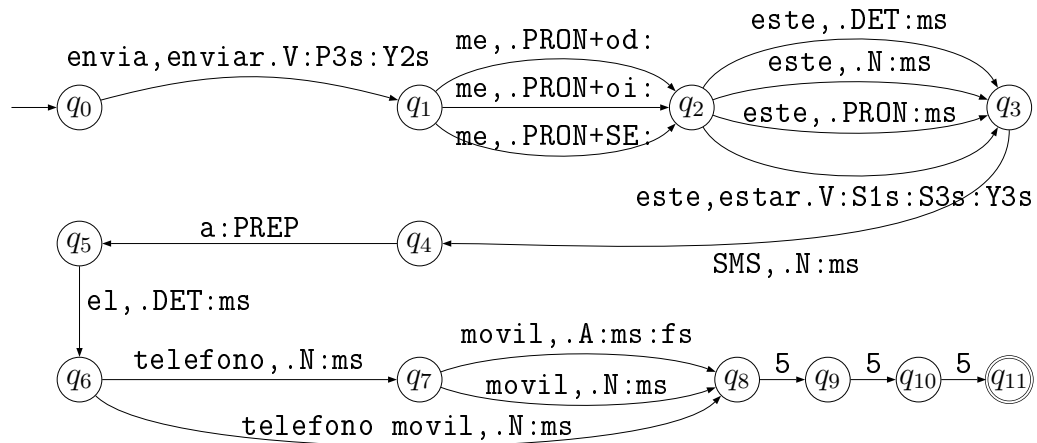
Figure 5.1: Text automaton illustrating the lexical ambiguities of sentence "*en-víame este SMS al teléfono móvil 555*" (send [for me] this SMS to the mobile phone 555), with diacritic marks removed from both the sentence and the dictionary.

token sequence, taking into account multiple segmentation possibilities (see figure 5.1 for an example). Though possible, attached words are not fragmented but unambiguous contractions are (*au* becomes *a le*), as part of a text-preprocessing. As we stated before, these systems where first focused on the treatment of French and English, which are even less agglutinative than Spanish.

Unitex and Outilex systems can partially disambiguate the text automaton by the application of ELAG grammars (Elimination of Lexical Ambiguities by Grammars, Laporte and Monceaux, 2000). These grammars are to be built manually by human experts rather than by some automated method, and mainly consist in associations of input fragments with possible tagging scenarios. Their application results in the removal of paths within the text automaton that do not correspond to any tagging scenario; for instance, some common words in both French and Spanish can be used both as determiners and as pronouns, and their use can be distinguished by their right contexts: they are determiners when preceding nouns, and pronouns when preceding verbs (e.g.: in '*la pintura, la ví*', meaning 'the painting, I saw it', the first '*la*' is a determiner and the second one is a pronoun). Though the Apertium system includes a part-of-speech tagger, it also allows to use simple 'forbid' rules for 2 part-of-speech sequences in order to improve the tagger results (Forcada et al., 2010, sec. 3.2.1, p. 56).

Unitex's implementation of text automata and ELAG grammars is described in Paumier (2008, sec. 7, p. 147), Outilex's implementation in Blanc (2006, sec. 2.7, p. 44) and Intex's implementation in Silberztein (2004, sec. 14.1, p. 153).

Chapter 6

Predicates and lexical masks

If one would define the machines representing the grammar of a natural language on the alphabet of tokens of that language, grammar rules for specific token sequences should be defined instead of being able to define general rules accepting tokens complying with a set of lexical properties (e.g.: verbs, nouns, etc.). Instead, we define an alphabet of predicates on the tokens so that sets of tokens having common properties can be represented by means of simple expressions. Specifying concrete tokens is still possible, allowing for a wide range between general and particular grammar rules. This idea was already proposed by van Noord and Gerdemann (2001) for the case of finite-state machines. We limit ourselves to describe here the set of predicates we have considered along with the codes used for representing them. These predicates are a subset of the ones supported by Unitex (Paumier, 2008, sec. 4.3, p. 72), and their codes are based on Unitex's FST2 format (Paumier, 2008, sec. 12.3.2, p. 252). Some of them constitute very powerful linguistic operators, allowing for referencing word classes depending on the properties described in DELAF dictionaries (e.g.: any word belonging to a set of semantic classes and having a set of inflectional features). A schematic list of the predicates along with their codes can be found in appendix A (p. 403).

6.1 Lexical masks

Definition 14 (Lexical mask). *A lexical mask is a predicate applicable to the tokens of a natural language, defining a subset of zero, one or more tokens by*

means of lexical criteria, that is, properties which only depend on the tokens themselves. We say a lexical mask matches a token iff it is true for that token; otherwise, we say it does not match the token.

Notice that there are predicates that are not lexical masks; for instance, the ε-predicates that will be presented in section 6.2 do not apply to tokens but depend on the presence or absence of blanks between them, which are not considered tokens.

6.1.1   Literal masks

Definition 15 (Literal mask)**.** *A literal mask is a lexical mask matching a unique token, where character case may or may not be restricted.*

Definition 16 (Lexicalization level)**.** *The lexicalization level of a grammar is a measure of dependence of grammar rules on specific tokens.*

Usually, the lexicalization level is expressed as a qualitative measure (e.g.: highly or strongly lexicalized). In our case, the lexicalization level of a grammar depends on the proportion of literal masks w.r.t. other kind of predicates.

Due to the irregularities of natural languages, natural language grammars cannot be defined as a set of general rules (Gross, 1997). Indeed, syntax is usually conditioned upon the presence of particular tokens (Harris, 1951; Chomsky, 1965; Gross, 1996). Applications requiring the recognition of properly written sentences, such as syntax validators and correctors, are to use highly lexicalized grammars. The objective of the grammars we have constructed for the MovistarBot use case is to detect the service the user is requesting for (e.g.: to send an SMS, to download a game, etc.), and to delimit the sentence segments corresponding to service arguments (e.g.: the text of the SMS to send, the title of the game to download, etc.). These grammars do not need to be strict and, indeed, we have tolerated some syntax errors in benefit of grammar simplicity; for instance, we have defined a grammar 'catalog' which recognizes several synonyms of catalog (e.g.: *catálogo*, *lista*, *repertorio*, etc.), where some of them are masculine and others are feminine, and we state that such words could be preceded by a determiner but do not specify the proper determiner's gender for each case (e.g.: we do not only accept '*el catálogo*' and '*la lista*' but also '**la catálogo*' and '**el lista*'). However, we still require some level of lexicalization since, usually, keywords determine what the user is talking about or asking for (e.g.: SMS,

message, send). Moreover, contexts of unrestricted arguments also require some lexicalization level (e.g.: the text of the SMS to send, or the title of the game to download). The only way to properly delimit such arguments is by elimination: once their left and right contexts are recognized, the argument is what is left; for instance, in '*envía el mensaje llego en 5 min al 555*' (send the message I arrive in 5 min to the 555), '*envía el mensaje*' indicates that a message is to be sent, and '*al móvil 555*' specifies the destination phone number, thus '*llego en 5 min*' is the message content. Fine descriptions of such contexts are to be given so that they are not considered as part of the unrestricted arguments; in the last example, segment '*el mensaje*' is optional (one could simply write '*envía llego en 5 min al 555*') and must not be recognized as part of the message to send.

Literal word masks

Definition 17 (Literal word mask). *A literal word mask is a literal mask which matches a unique word token and, in some cases, their case variations.*

Definition 18 (Case-sensitive word mask). *A case-sensitive word mask is a literal word mask strictly matching the specified word token, without case variations. We represent them as a '@' symbol followed by the sequence of characters that compose the word (e.g.: mask '@WORD' matches token 'WORD' but not tokens 'word' or 'Word)'.*

Definition 19 (Case-insensitive word mask). *A case-insensitive word mask is a literal word mask which also accepts case variations, and we represent them as a '%' symbol followed by the sequence of either uppercase or lowercase characters that compose the word (e.g.: mask '%wOrD' matches both tokens 'word' and 'WORD').*

As a convention, every predicate code starts with either '%' or '@'. For the case of literal word masks, the use of one or other symbol determines case sensitivity, and for the other cases there is no difference.

In most cases, the use of lowercase or uppercase letters does not alter the meaning but is just a matter of formatting; for instance, the MovistarBot considers equivalent the sentences 'envía llego en 5 min al 555' and 'ENVÍA LLEGO EN 5 MIN AL 555'. Proper nouns and abbreviations are an exception; for instance, 'RAM' is the random access memory of a computer, 'Ram' is the constellation or sign of Aries, and 'ram' has several other meanings, such as the male of the sheep.

Since our use case also requires the omission of diacritic marks, our case insensitive word masks are also insensitive to diacritic marks. As stated in section 3.4.2 (p. 81), we use a look-up table in order to retrieve the normalized version of the token characters as they are compared to the case-insensitive word masks, and these masks are normalized only once and then applied to every received user sentence.

Literal symbol masks

Definition 20 (Literal symbol mask). *A literal symbol mask is a literal mask which only matches a specific symbol token. We represent literal-symbol masks as a symbol '%' or '@' followed by the symbol to match (e.g.: '@@' matches token '@').*

6.1.2   Token mask

Definition 21 (Token mask). *We define the token mask as the lexical mask matching any token, and we represent it as '%<TOKEN>' or '@<TO-KEN>'.*

Note that the code representing the token mask is composed by uppercase letters. As a general rule, lexical mask codes are case sensitive, that is, *%<TOKEN>*' and *%<token>*' do not represent the same lexical mask.

The purpose of the token mask is to match unrestricted or unknown tokens and token sequences. We have used it for matching unrestricted arguments, and it can also be used for the complementary case: the extraction of well-known arguments from unrestricted texts; for instance, Laforest and Badr (2003) use Intex-based automata for extracting very specific data from a set of medical prescriptions written in natural language, namely the illness, drug, dose, dosage frequency and treatment duration (e.g.: influenza A, Tamiflu, 75 mg, twice daily, 5 days). This information is latter input into a structured database.[1]

---

[1]The author of this dissertation participated in the conception of the language of specification of extraction rules used in Laforest and Badr (2003) and in the implementation of a translator of such rules to Intex automata.

6.1.3 Character-class masks

Definition 22 (Character-class mask). *A character-class mask is a lexical mask which restricts only the type of the characters that compose the token.*

Definition 23 (Word mask). *We define the word mask as the character-class mask that matches any word token (e.g.: aBc), and we represent it as '%<MOT>' or '@<MOT>'.*[2]

Definition 24 (Symbol mask). *We define the symbol mask as the character-class mask matching any symbol token (e.g.: +), and we represent it as '%<!MOT>' or '@<!MOT>'.*

Note that the code representing the symbol mask is the same than for the word mask but inserting the '!' symbol right after '<'. As a general rule, by inserting the '!' symbol we negate the lexical mask. However, for different classes of lexical masks the negation is defined in a particular way. Note that the negation of the token mask would match no token, hence it makes no sense to define it: we simply do not define grammar rules matching grammatically incorrect sequences.

Definition 25 (Digit mask). *We define the digit mask as the character-class mask that matches any digit (e.g.: 7), and we represent it as '%<NB>' or '@<NB>',*[3] *and its negation as the mask matching any non-digit token (e.g.: aBc, +, ?, etc.), and we represent it as '%<!NB>' or '@<!NB>'.*

Note that digit masks do not match numbers composed by several digits or containing a decimal dot; each digit is considered a token, as well as the decimal dot, and each application of a lexical mask matches a single token.

Definition 26 (Punctuation-symbol mask). *We define the punctuation-symbol mask as the character-class mask matching every punctuation symbol (e.g.: ?), and we represent it as '%<PNC>',*[4] *and its negation as the mask matching any non-punctuation-symbol token (e.g.: +), and we represent it as '%<!PNC>' or '@<!PNC>'.*

---

[2]From French '*mot*', which means 'word'.

[3]From French '*nombre*', which means 'number'

[4]From French '*ponctuation*', which means 'punctuation'.

Case-dependent word masks

Definition 27 (Case-dependent word mask). *A case-dependent word mask is a character-class mask whose restrictions are uniquely based on the case of the characters that compose the tokens.*

Definition 28 (Negation of a case-dependent word masks). *We define the negation of a case-dependent mask m as the mask matching every word token that m does not match.*

Note that the negation of a case-dependent word mask does not match symbol tokens.

Definition 29 (Uppercase-word mask). *We define the uppercase-word mask as the case-dependent word mask matching every word token whose letters are all uppercase (e.g.: SMS), and we represent it as '%<MAJ>' or '@<MAJ>' and its negation as %<!MAJ>' or '@<!MAJ>'.[5]*

Definition 30 (Lowercase-word mask). *We define the lowercase-word mask as the case-dependent word mask matching every word token whose letters are all lowercase (e.g.: message), and we represent it as '%<MIN>' or '@<MIN>' and its negation as %<!MIN>' or '@<!MIN>'.[6]*

Definition 31 (Proper-noun mask). *We define the proper-noun mask as the case-dependent word mask matching every word whose first letter is uppercase and the others lowercase (e.g.: Chomsky), and we represent it as '%<PRE>' or '@<PRE>' and its negation as %<!PRE>' or '@<!PRE>'.[7]*

As stated in section 5.2 (p. 99), token types are computed during the tokenization process. The evaluation of character class masks is reduced to a bitwise comparison between the identifiers of the required token type and the next token type.

6.1.4   Dictionary-based masks

We have also added support for Unitex's dictionary-based lexical masks, which in turn are the same than the ones of the Intex system. These masks

---

[5]From French '*majuscule*', which means 'uppercase'.
[6]From French '*minuscule*', which means 'lowercase'.
[7]From French '*prénom*', which means 'proper noun' or 'first name'.

define subsets of the words of a DELAF dictionary —except the unknown-word mask, which we define below— usually depending on the properties considered in the dictionary.[8]

Definition 32 (Dictionary-word mask). *A dictionary-word mask is a lexical mask that matches word tokens depending on criteria based on the data contained in a dictionary.*

Definition 33 (Known-word mask). *We define the known-word mask as the dictionary-word mask matching every word that belongs to the dictionary, and we represent it as '%<DIC>' or '@<DIC>'.[9]*

Definition 34 (Unknown-word mask). *We define the unknown-word mask as the dictionary-word mask matching every word that does not belong to the dictionary, and we represent it as '%<!DIC>' or '@<!DIC>'.*

Note that the unknown-word mask does not match symbol tokens. This mask may be used for testing the dictionary coverage and for searching for new dictionary-word candidates.

Definition 35 (Constrained dictionary-word mask). *A constrained dictionary-word mask is dictionary-word mask matching the subset of dictionary words holding a given set of properties, which are to be specified in the dictionary (e.g.: being a verb in present tense).*

Definition 36 (Lemma mask). *A lemma mask is a constrained dictionary-word mask matching every dictionary word having the specified word as lemma, and we represent it as either '%<lemma>' or '@<lemma>', where 'lemma' is the lemma in lowercase letters.*

As for the case-insensitive masks, every grammar's lemma mask is normalized before applying the grammar in order to match the normalized lemmas within the dictionary. In general, any lemma specified within any kind of dictionary-word mask is normalized.

Definition 37 (Semantic-feature mask). *A semantic-feature mask is a constrained dictionary-word mask matching every dictionary word belonging to a set of mandatory semantic classes and not belonging to a set of forbidden semantic classes, and we represent them as '[%@] <[+−]? Sem_1 [+−]Sem_2*

---

[8]DELAF dictionaries have been described in chapter 4, p. 83.

[9]From French '*dictionnaire*', which means 'dictionary'.

$[+-]\ldots[+-]$ Sem$_n>$', where $[\%@]$ stands for either symbol '%' or '@', 'Sem$_1\ldots$Sem$_n$' are a sequence of $n \geq 1$ semantic codes, '$[+-]$' stands for either a '+' or '−' symbol indicating whether the following semantic code refers to a mandatory or forbidden semantic class, respectively, and '$[+-]?$' stands for an optional specification of this symbol, by default '+'; for instance, masks '<N+Hum>', '<−Hum+N>' and '<−N−Hum>' match every human noun, every non-human noun and every non-noun non-human word, respectively.

Definition 38 (Lemma and semantic-feature mask). *A lemma and semantic-feature mask is a constrained dictionary-word mask both restricting the inflectional and semantic classes of dictionary words, and we represent them as '$[\%@]$<lemma.<$[+-]?$ Sem$_1$ $[+-]$ Sem$_2$ $[+-]\ldots[+-]$ Sem$_n>$', that is, as for both the lemma and semantic-feature masks but first specifying the lemma, then a dot, then the sequence of either mandatory or forbidden semantic classes.*

Note that '<hum>' denotes a lemma mask while '<Hum>' denotes a semantic-feature mask. Note also that lexical units belong to a unique part-of-speech class (part-of-speech classes are disjoint), hence it only makes sense to either specify a unique mandatory part-of-speech or one or more forbidden parts-of-speech.

Definition 39 (Possible-inflectional-features mask). *A possible-inflectional-features mask is a constrained dictionary-word mask matching every dictionary word having all of the inflectional features represented by at least one of the specified sequences of inflectional codes; we represent the possible inflectional features as a colon-separated list of sequences of inflectional codes (e.g.: 'ms:mp' for masculine singular or masculine plural, which would be equivalent to 'm' since every number is accepted). Since inflection depends on the part-of-speech, masks restricting the inflected form alone are not to be defined but both restricting the part-of-speech and the inflected form, and possibly other semantic classes as well as the inflection class, and we represent them as either the semantic-features mask or the lemma and semantic-features mask but inserting a colon after the semantic-features specification followed by the list of possible inflectional-features list.*

For instance, mask '%<V+Trans_ msg:Y2:Y3>' matches 40 words that can be used for ordering the MovistarBot (Y = imperative) to send an SMS

(e.g.: *envía, envíe, enviad, envíen, manda, mande, mandad, manden,* etc).[10].
Mask '*%<enviar.V+Trans_ msg:Y2:Y3>*' matches the subset corresponding
to inflected forms of verb '*enviar*'.

Definition 40 (Negation of constrained dictionary-word masks). *The ne-
gated form of a constrained dictionary-word mask matches every dictionary
word not matched by the original mask; we negate a dictionary mask by
inserting symbol '!' after symbol '<' (e.g.: '%<!V>' matches any dictionary
word that is not a verb).*

Note that the negation of a constrained dictionary-word mask does not
match unknown words.

The implementation of such masks consists in searching the next input
token within the DELAF, either for verifying its presence or absence (case of
known and unknown word masks) or for verifying the presence of a use of the
token whose properties comply with a set of restrictions (case of constrained
dictionary-word masks).

6.2   ε-predicates

Definition 41 (ε-predicate). *An ε-predicate is a predicate that applies to the
space between two tokens.*

Some grammar rules are to be applied right before the next token to be
analysed. In other words, such rules apply on the emptiness between the last
analysed token and the next one. This emptyness is usually represented by
the empty symbol, ε. Usually, such emptiness has no associated properties
which could be evaluated in order to decide whether the rules are to accept
or reject it, hence the emptiness is always accepted. In our case, the space
between consecutive word tokens must contain at least one blank character,
and the space before and after symbol tokens may be either empty or contain
one or more blank characters. In some exceptional cases, the presence or
absence of blank characters is to be taken into account; for instance, in
French, sequence '1,2' without blanks represents a real number, while '1, 2'
with a blank is a sequence composed by two numbers. In the MovistarBot

---

[10]In Spanish, we address somebody using the third person instead of the second one in
order to show respect, courtesy or distance.

use case (section 1.2, p. 6), request sentences may also follow a command-like syntax, that is, a word command followed by a blank-separated list of arguments where arguments are not to contain blanks in some cases; for instance, one can request to send an SMS as follows:

`sms phone message`

where at least one blank character must appear before and after `phone`, `phone` is a sequence of digit symbols without blanks between them, and `message` is any sequence of tokens, either blank-separated or not. This situation leads to the definition of different kinds of ε-predicates depending on whether tokens are blank-separated or not.

Definition 42 (Blank-insensitive ε-predicate). *We define the blank-insensitive ε-predicate as the ε-predicate that always evaluates true, independently of the presence of absence of blanks between the last analysed token and the next one, and we represent it with code '%<E>' or '@<E>'.*

Definition 43 (Blank-sensitive ε-predicates). *Blank-sensitive ε-predicates are those who may or may not be true depending on the presence or absence of blanks between the last analysed token and the next one.*

Definition 44 (Mandatory-blank ε-predicate). *We define the mandatory-blank ε-predicate as the blank-sensitive ε-predicate that evaluates true iff the next token is blank-separated, and we represent it with code '%\⊔' or '@\⊔', where '⊔' represents a white space.*

Definition 45 (Forbidden-blank ε-predicate). *We define the forbidden-blank ε-predicate as the blank-sensitive ε-predicate that evaluates true iff the next token is not blank-separated, and we represent it with code '%#' or '@#'.*

As stated in section 5.2 (p. 99), the evaluation of these predicates consists in verifying whether the pointer to the right bound of the last analysed token and the pointer to the left bound of the next input token are equal or not.

These ε-predicates are usually enough for natural language processing. If necessary, one may define more fine-grained ε-predicates which would, for instance, take into account the amount and type of blank characters between tokens. For instance, programming languages usually mark the start and end of instruction blocks with start/end pairs of keywords (open/closed curly brackets in C, C++ and Java, pairs if/fi, do/done, case/esac in Unix/Linux shell scripts, etc), but in Python the start and end of a block is given by the length of instruction indentation (number of consecutive blanks at the beginning of the instruction line).

6.3 Supporting predicates

A grammar's application usually consists in keeping trace of a set of *live* grammar rules (some representation of grammar rules that have being partially applied to an input segment, depending on the algorithm of application of the grammar); the grammar must define an initial set of rules, which is used for building the initial set of live rules, and for each input token a new set is computed depending on the previous set and the next input token. Each live grammar rule imposes some restrictions on the next input token, which are expressed as lexical masks and ε-predicates in our case. In order to compute the next set of live grammar rules, the set of next lexical masks by the current set of live rules must be searched for the ones matching the next input token (in our case, to search for the transitions outgoing from the current states and whose labels match the next input token). In case the grammars were defined on an alphabet of letters rather than of predicates, the set of live grammar rules is to be searched in order to find the ones whose next letters are equal to the next input letter. The next-letter sets could be stored in some binary-search structure, such as the ones presented in chapter 2, so that the ones equal to the next input letter could be efficiently found. However, it is not possible to define a lexical mask ordering in order to guide a binary search depending on the next input token and the result of the last mask evaluation; for instance, let $\%{<}A{>}$ (is adjective), $\%{<}N{>}$ (is noun) and $\%{<}V{>}$ (is verb) be the current candidate predicates, and predicate $\%{<}N{>}$ be the first one to be evaluated, whether the next token is a noun word or not does imply whether the same token may or may not be a verb or an adjective (e.g.: 'love' can either be a noun or a verb, while 'orange' can either be a noun or an adjective); hence, all the candidate lexical masks must be systematically evaluated. While a binary search has a logarithmic cost, the systematic search has a cost proportional to the number of lexical masks to evaluate. However, if such number is small, the efficiency loss is insignificant; this is the case of the MovistarBot grammar, with an average of 2.3 outgoing transitions per state.

6.4 Assigning priorities to lexical masks

During the first stages of a grammar's development, one may define general rules (in our case, transitions labeled with lexical masks) in order to cover a

great number of cases, and later add more specific rules for well-known cases which may overlap with the general cases. Rule overlapping leads to multiple sentence interpretations. A good heuristic for choosing a single one is the overall specificity level of the sequence of rules that led to each interpretation (in our case, the sequence of transitions whose lexical masks recognized the entire sentence).

We have shortened lexical masks depending on the cardinality of the set of tokens matched by each one, from the most general (the token mask) to the most specific (case insensitive masks), and assigned a default weight for each case we have distinguished (see table 6.1). We have implemented a routine that automatically assigns the corresponding weights to grammar rules (transitions of recursive transition networks with weight and string output) depending on the specificity of their lexical masks. Rules that are not labeled with lexical masks (ε-predicate rules) are given a zero weight by default. We interpret weights as scores: the highest the specificity, the highest the score. Note that mask $\%<TOKEN>$ is given a null score: recognizing tokens without any restrictions does not increase or decrease the interpretation score.

The main achievement of this procedure has been to successfully deal with ambiguous sentences due to unrestricted arguments in the middle, such the text of the message that is requested to be sent by sentence '*envía el SMS Feliz Navidad al móvil 555*' (send the SMS Merry Christmas to the mobile 555). One of the general grammar rules allows for simply writing 'envía Feliz Navidad' in order to ask for sending the SMS 'Feliz Navidad', without either specifying any phone number (the MovistarBot would then ask for it) or the fact that what we want to send is an SMS. Consequently, this rule recognizes '*el SMS*' and '*al móvil 555*' as part of the message to send. However, the general rule will use the token mask in order to recognize those sentence segments, while the more specific rule will use literal masks, which are given higher scores. Other rule recognizes the case in which the user delimits the text of the message by means of quotes, in which case the quotes are neither interpreted as part of the message since they are also recognized by means of literal masks.

Grammar rules that are already given a weight are not touched by the weight assignment procedure, hence it is possible to define custom weights for specific grammar rules by hand. At a certain time, Telefónica requested a fast implementation of a grammar for the recognition of sentences requesting to send an MMS, based on the mere detection of keyword 'MMS'. However, we had already defined other more specific grammars that interpreted MMS

Mask	Cardinality	Weight
%<TOKEN>	∞	0
%<!NB>	∞ but subset of %<TO-KEN>	1
%<!PNC>	∞ but less than %<!NB>	2
%<MOT>	∞ but subset of %<!NB> and %<!PNC>	3
%<!DIC>	∞ but subset of %<MOT>	4
%<!PRE>	∞ but subset of %<MOT> and less than %<!DIC>	5
%<!MAJ>	∞ but less than %<!PRE>	6
%<!MIN>	∞ but uppercase is less frequent than lowercase	7
%<MIN>	∞ but subset of %<!MAJ> and less than %<!MIN>	8
%<MAJ>	∞ but subset of %<!MIN> and uppercase is less frequent than lowercase	9
%<PRE>	∞ but less than %<MAJ> and %<MIN>	10
%<DIC>	equal to the dictionary size	11
constrained dictionary-word mask	less than or equal to the dictionary size	11
%<!MOT>	equal to the cardinality of the set of symbol tokens, less than %<DIC> for natural languages	12
%<NB>	10 (decimal system)	13
case-insensitive word mask	$2^{|w|}$ for word w, though less than %<NB> in practice	14
literal symbol mask	1 among the set of symbol tokens	15
case-sensitive word mask	1 among the set of word tokens, which is greater than the set of symbol tokens	16

Figure 6.1: Default weights assigned to lexical masks.

requests as other kind of requests, hence the MMS requests were misinter-
preted in some cases (e.g.: '*quiero enviar un MMS al 555*' was recognized as
'I want to send the SMS *un MMS* to the phone number 555'). This situation
was solved by assigning by hand to the MMS grammar a weight higher than
those automatically assigned to the other grammars.

Part II

Finite-state machines

Chapter 7

Finite-state machines

We give in this section the common definitions, properties and algorithms for every machine used along this dissertation. The definition of FSM is too general to be directly applied as a machine, but is intended to be further refined in order to briefly define the concrete machines in the subsequent chapters. This definition is very similar to the usual definition of non-deterministic FSA (NFA) but leaving undefined the set of transition labels rather than being equal to the set of input symbols Σ plus the empty symbol, and defining both the input alphabet and the set of transition labels as either finite or potentially infinite; the latter is required in order to consider machines whose transition labels are taken from a potentially infinite alphabet of words or predicates on words, such as the set of lexical masks presented in chapter 6.[1] As stated in van Noord and Gerdemann (2001), predicate alphabets do not need to be explicitly defined —the rules for the construction of predicate expressions are to be explicitly defined instead— and, consequently, such alphabets do not need to be finite. Moreover, van Noord and Gerdemann (2001, sec. 1.1, p. 2) explicitly state that "robust syntactic parsing requires an infinite alphabet".

Definition 46 (Finite-state machine). *In general, finite-state machines (FSMs) are structures composed of, at least, the following 6 elements:*

- $Q = \{q_0, q_1, \ldots, q_{|Q|-1}\}$, *a finite set of states (SS),*

---

[1] We assume that a word is any sequence of letters, though in practice grammars are to consider only a finite set of words since grammars are to be finite descriptions of language structures (hence the expression '*potentially* infinite').

- $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{|\Sigma|-1}\}$, *an either finite or* potentially *infinite input alphabet,*

- $\Xi = \{\xi_0, \xi_1, \ldots, \xi_{|\Xi|-1}\}$, *an either finite or* potentially *infinite set of transition labels,*

- $\delta : Q \times \Xi \rightarrow \mathcal{P}(Q)$ *a finite and partial transition function where $\mathcal{P}(\cdot)$ represents the set of all subsets of a given set,*

- $Q_I \subseteq Q$ *is the set of initial states and*

- $F \subseteq Q$ *the set of final or acceptance states.*

The particular definition of the set of transition labels depends on the kind of particular machine.

Definition 47 (Letter FSM). *A letter FSM is a FSM whose alphabet is a finite set of letters, words or symbols rather than predicates, and whose transition labels are taken from such alphabet, except for the empty symbol.*

This definition corresponds to a generalization of the definition of letter transducer given by Roche and Schabes (1997, p. 14).

For the simplest incarnation of FSM, letter FSAs, transitions labeled with an input symbol can be traversed by consuming the next input symbol when both symbols are equal, and transitions labeled with the empty symbol can be traversed without input consumption. We will present FSAs as a particular case of this FSM definition in chapter 8 (p. 161).

Definition 48 (Lexical FSM). *A lexical FSM is a FSM whose transition labels are lexical masks and ε-predicates with, possibly, other extensions.*[2].

This definition is a generalization of the definitions of lexical automaton and decorated lexical RTN given in Blanc (2006, chaps. 2 & 4, pp. 13 & 112). Such RTNs are extended with unification processes and their labels are decorated with equations on feature structures.

Most of the theory on FSMs we will present does not depend on the exact mechanism for the evaluation of transition labels, but on whether transitions are taken or not and, therefore, the same theory applies to both letter or lexical FSMs. For the sake of simplicity, we will present this theory for

---

[2]Lexical masks and ε-predicates have been described in chapter 6, p. 105

the case of letter FSMs, and discuss the differences w.r.t. the case of lexical FSMs whenever they are not obvious. The grammars we have built for the MovistarBot use case (sec. 1.2, p. 6) are a kind of lexical RTNs.

Definition 49 (Null element). *In general, we explicitly represent an illegal, invalid or undefined result of an operation as* $\perp$*, the* null *element.*

For instance, given a state q of a FSM and a transition label ξ, we explicitly represent the lack of transitions from q, with label ξ, as $\delta(q, \xi) = \perp$. Notice that there is a difference between the null element and the neutral or identity element: given a binary operator $\cdot$ and two operands a and b, if b is the identity element of $\cdot$ then $a \cdot b = a$, but if b is the null element then the result is $\perp$, that is, undefined.

7.1 Transitions

Definition 50 (Transition). *We represent a transition of a FSM as a triplet* $(q_s, \xi, q_t) \in (Q \times \Xi \times Q)$ *where* $q_t \in \delta(q_s, \xi)$*. We call* q_s *and* q_t *the* source *and* target *states of the transition, respectively, and* ξ *the transition label. Transitions, also called moves or jumps, represent the possibility of changing the state of the machine from a source state to a target state depending on the predicate or condition expressed by the* ξ *label and the current context of execution of the machine, and to perform some other arbitrary actions which will further modify the current context of execution.*

The context of execution of a machine is a generalization of the state in which a machine can be during its application; for instance, the context of execution of an augmented transition network, or ATN (Woods, 1969), includes a set of registers which may be modified by additional actions associated to transitions, and whose value may condition the transition traversal. A formal definition of execution context or *execution state* will be given in section 7.6.

Definition 51 (Consuming transition). *A consuming transition is a transition conditioned upon the current input symbol and which triggers the* consumption *of the symbol (to advance the input pointer up to the next symbol) whenever the transition is traversed.*

Definition 52 (Pure consuming transition). *A pure consuming transition is a consuming transition not associating any other condition or action to its traversal than the sole consumption of an input symbol.*

A non-pure consume transition would be, for instance, a consuming transition that also generates output.

Definition 53 (ε-transition). *An ε-transition is a transition whose predicate requires no symbol to be consumed. We explicitly represent the absence of symbol or* empty *symbol as ε.*[3]

Transitions whose predicates hold independently of the current context effectively represent the possibility of being in several states at the same time. This is the case of ε-transitions for several machines, for instance FSAs, FSTs and RTNs. Other machines may consider other conditions to be taken into account which prevent the transition from being traversed for some execution contexts, even when no input symbol is required to be consumed; for instance, ε-transitions of machines extended with unification may require to unify two feature structures, and the traversal of such transition will not be possible if the feature structures to unify are incompatible. Unification machines will be the object of chapter 19.

Definition 54 (Pure ε-transition). *A pure ε-transition is an ε-transition not associating any condition or action to its traversal.*

Definition 55 (Outgoing and incoming transitions). *Given a transition $t = (q_s, \xi, q_t)$, we say that t is an* outgoing *transition from state q_s and an* incoming *transition into state q_t.*

7.2   Graphical representation

The classic representation of FSMs consists in a set of labeled circles and labeled arrows between the circles, the former representing states and the latter transitions (see figure 7.1(b)). Double-border circles represent acceptance states (e.g.: state q_8 of figure 7.1(b)), and initial states are pointed to by an arrow coming from nowhere (e.g.: state q_0 of figure 7.1(b)).

---

[3]Some authors use different symbols in order to represent the empty symbol and the empty string; for instance, Ortiz-Rojas et al. (2005) represent the former as θ and the latter as ε. We choose here to make no distinction in order to alleviate the notation.

Intex, Unitex and Outilex use another kind of representation they call *graph*, and which was conceived for facilitating the manual construction and maintenance of local grammars.[4] This is the representation we have used for the construction of the MovistarBot grammars.[5] Graphs basically consist in a set of linked boxes, where boxes correspond to transitions and links correspond to states. Each box represents a set of transitions, one transition per box line, which is shared among every pair of states represented by an incoming and an outgoing link (e.g.: box '<TOKEN>' of figure 7.1(a) corresponds to the 4 '%<TOKEN>' transitions of figure 7.1(b)). Start symbols '@' and '%' of lexical masks within boxes are not specified: case-sensitive masks are to be quoted, and case-insensitive masks are not. Moreover, a box entry may contain a sequence of lexical masks rather than a single one, in which case represents an alternating sequence of transitions and states rather than a simple transition (e.g.: a box entry '"Feliz Navidad"' represents two case-sensitive masks which are to be applied in that order). The direction of the transitions represented by a box is given by a triangular arrowhead attached to one side of the box. Graphs are meant to be read in the text sense, which depends on the language (e.g.: left-to-right in English, right-to-left in Arabic, etc.), thus the arrowhead is always attached to the same side of the boxes. Links between boxes do not carry state labels: state labels are not to be explicitly defined since they have no impact in the represented grammar. Graphs define a unique initial state represented by a link connected to a single box. To make the initial state more explicit, an empty box (a box having only the arrowhead) is inserted right after the link, though this is not necessary; indeed, boxes having a single entry '<E>' (the blank-insensitive ε-predicate) are drawn as a triangle alone. Links representing acceptance states are those connected to the circle with a square inside (see the rightmost box of figure 7.1(a)). Finally, graphs can be commented as for the case of source code (e.g.: see greyed text of figure 7.1(a)); labels of unlinked boxes are treated as comments, and their frames are not drawn. More information on Intex graphs can be found in Silberztein (2004, chap. 5–8), on Unitex graphs in Paumier (2008, chap. 5–6), and on Outilex graphs in Blanc and Constant (2006b, chap. 4).

---

[4]In the context of mathematics, a graph is a set of elements (vertices) coupled with a set of edges which connect pairs of elements; though the graphs presented here are very similar to these graphs, they are not the same kind of object.

[5]Indeed, we have used the Unitex graph editor.

Figure 7.1: (a) Unitex graph recognizing SMS command requests, (b) equivalent lexical FSM obtained by replacing boxes and links by trasitions and states, respectively, and (c) equivalent pseudo-minimal lexical FSM. Comments in (a) appear in greyed fonts and have been used here to make state labels explicit and to indicate what each graph fragment is supposed to recognize. Pseudo-minimization will be explained in section 8.6.

7.3 Sequences of transitions

Definition 56 (Connected transitions). *We say transition t is connected to transition t' iff the target state of t is the source state of t'.*

Definition 57 (Path or transition concatenation). *We define a* path *within a FSM as the concatenation of a sequence of $n > 0$ successively connected transitions*

$$(q_{s_0}, \xi_0, q_{t_0}) \dots (q_{s_{n-1}}, \xi_{n-1}, q_{t_{n-1}}) \quad such\ that \quad q_{t_i} = q_{s_{i+1}}, \quad for\ i = 0 \dots n-2. \tag{7.1}$$

and we represent it as

$$(q_{s_0}, \xi_0, q_{t_0}) \cdot (q_{s_1}, \xi_1, q_{t_1}) \cdot \dots \cdot (q_{s_{n-1}}, \xi_{n-1}, q_{t_{n-1}}), \tag{7.2}$$

or simply as

$$(q_{s_0}, \xi_0, q_{t_0})(q_{s_1}, \xi_1, q_{t_1}) \dots (q_{s_{n-1}}, \xi_{n-1}, q_{t_{n-1}}), \tag{7.3}$$

or even simpler as

$$q_{s_0} \xrightarrow{\xi_0} q_{s_1} \xrightarrow{\xi_1} \dots \xrightarrow{\xi_{n-2}} q_{s_{n-1}} \xrightarrow{\xi_{n-1}} q_{t_{n-1}}. \tag{7.4}$$

Definition 58 ($|\cdot|$). *We define $|\cdot|$ as the length or number of elements of a sequence.*

We use $|\cdot|$ in order to represent the length of a path inside a FSM as well as the length of a string.

Definition 59 (Start and end states of a path). *The* start *state of a path is the source state of its first transition, and the* end *state of a path is the target state of its last transition.*

Definition 60 (Connected paths). *We say path p is connected to path p' iff the end state of p is equal to the start state of p'.*

Definition 61 (Concatenation of paths). *We define the concatenation of a path p connected to a path p' as the path composed by the sequence of transitions of p followed by the sequence of transitions of p', and we represent it as $p \cdot p'$ or simply as pp'.*

Definition 62 (Cycle). *A cycle is a closed or self-connected path, that is, a path whose start and end states are the same. Cycles are also called* loops *or* closed paths.

Since cycles are to be described as transition sequences, one of the cycle states is to start and end the sequence. Cycles have no start or end states in the sense of whether the sequence of transitions can be followed, either in direct or reverse order, until there are no more transitions belonging to the cycle, since the first transition follows the last one. However, when regarding cycle descriptions as itineraries (a sequence of states and transitions to visit in the order specified), the choice of the start and end state is not arbitrary: traversing n times a cycle starting at a state q ends up at q and not at any other state.

Definition 63 (Self-concatenation of a cycle). *Given a cycle p and an integer $n \geq 0$, we define p^n as the result of concatenating path p with itself n times.*

Definition 64 (Empty path). *Given any closed or unclosed path p, we define p^0 as the* empty path *or* zero-length path.

Definition 65 (Self-concatenation of the empty path). *Given a path p, we define the special cases of concatenations of paths involving the empty path as follows:*

$$pp^0 = p$$
$$p^0 p = p$$

The empty path is the neutral element of the concatenation of paths.

Corollary 1 (Concatenation with the empty path). *Concatenating the empty path to itself any number of times results in the empty path, that is,*

$$p^0 p^0 \ldots p^0 = p^0 \tag{7.5}$$

Definition 66 (Subpath). *We say p_b is a subpath of p iff there exist two paths p_a and p_b such that $p_a p_b p_c = p$.*

Definition 67 (Consuming path). *A consuming path is a path having at least one consuming transition.*

Definition 68 (Consuming cycle). *A consuming cycle is a closed consuming path.*

Definition 69 (ε-path). *An ε-path is a path whose transitions are all ε-transitions.*

Definition 70 (ε-cycle). *An ε-cycle is a closed ε-path.*

Definition 71 (Reachable). *We say that a state q_t is reachable or derivable from a state q_s iff there exists at least one path p whose start and end states are q_s and q_t, respectively. We say q_t is reachable or derivable from q_s through path p.*

Definition 72 (Directly reachable). *We say that a state q_t is directly reachable from a state q_s iff q_t is reachable from q_s through a path p such that $|p| = 1$.*

We will first define the function computing sets of directly-reachable states from other states, and then define the function computing the set of reachable states by one or more applications of the former function, hence the distinction between reachable and directly reachable.

Definition 73 (ε-reachable). *We say that a state q_t is ε-reachable from a state q_s iff it is reachable through an ε-path.*

Definition 74 (Directly ε-reachable). *We say that a state q_t is directly ε-reachable from a state q_s iff q_t is directly reachable from q_s through an ε-path.*

7.4 Structures

Definition 75 (Empty machine). *We say a machine is empty iff it contains no states.*

Corollary 2 (Transitions of empty machines). *Empty machines have no transitions since no transitions can be defined without source and target states.*

Definition 76 (Acyclic machine). *We say a machine is acyclic iff it contains no cycles.*

Tries are an example of acyclic machines. These machines will be the object of chapter 9.

Definition 77 (Linear machine). *We say a machine is linear or has a linear structure iff the machine is acyclic, has at most one initial state and every state has at most one outgoing transition.*

A sequence of symbols w can be represented by a linear FSA having a unique acceptance state and a path consuming w from the initial state up to the acceptance state.

7.5   Substructures

Definition 78 (Machine substructure). *Given a machine with a set of states Q and a partial transition function δ, (Q', δ') identifies a machine substructure iff $Q' \subseteq Q$ and δ' is a partial transition function such that every transition defined by δ' is also defined by δ and the source and target state of every transition in δ' belongs to Q'.*

Definition 79 (Disjoint machine substructures). *We say two machine substructures are disjoint iff they do not share any states and/or transitions.*

Definition 80 (Properties of relations). *A relation R on a set A is*

- reflexive *iff $a\,R\,a$, for all a in A,*

- irreflexive *iff $\neg(a\,R\,a)$, for all a in A,*

- antisymmetric *iff $a\,R\,b$ and $b\,R\,a$ imply $a = b$, for all a, b in A, and*

- transitive *iff $a\,R\,b$ and $b\,R\,c$ imply $a\,R\,c$, for all a, b, c in A,*

Definition 81 (Topological sort). *Let (Q', δ') be a machine substructure and R a relation on Q' such that $q_s\,R\,q_t$ iff q_t is reachable from q_s; we say a sequence of states in Q' is a topological sort of (Q', δ') iff the following conditions hold:*

- *R is irreflexive, antisymmetric and transitive,*

- *the sequence contains every state in Q', and*

- *the sequence is compatible with R, that is, for every pair of states $q_i, q_j \in Q'$, if q_i appears before q_j within the sequence then either $q_i\,R\,q_j$ or q_i and q_j are not related.*

The problem of finding a topological sort for a graph was first studied for the case of PERT networks (see appendix D, page 419). Since vertices of a PERT network represent points in time and edges represent activities between two points in time, cycles of length 1 make no sense: activities require a positive amount of time in order to be performed. However, it makes sense that every temporal point is reachable from itself by performing no activity. The original definition of topological sort requires R to be a (non-strict) partial order, that is, it must be reflexive rather than irreflexive; since temporal points are considered to be reachable from themselves, R is also reflexive. The topological sort is defined as a (non-strict) total order compatible with R, that is, an extension of R such that antisymmetry and transitivity is kept while relating every pair of vertices of the network. Totality implies reflexivity since it also requires every vertex to be related to itself. Since FSMs may contain cycles of length 1, we must distinguish between just being at a state and reaching it from itself, hence we do not consider that a state is reachable from itself unless it has a both outgoing and incoming transition. Consequently, our definition of topological sort does neither require R or the topological sort to be reflexive and, indeed, it forbids it; in our case, the topological sort of a FSM represents a linear ordering of the states of a machine such that each state can only be reached from zero, one or more of the states preceding them in the topological sort, but not from themselves or from the states following them. This ordering will be used for optimizing the application of machines with blackboard output: we will see that it is possible to process each transition a single time as long as we follow the ordering given by a topological sort.

Lemma 1 (Existence of a topological sort). *At least one topological sort exists for a given machine substructure iff the substructure contains no cycles.*

Proof. Let (Q', δ') be a machine substructure, R be a relation on Q' such that $q_s \, R \, q_t$ iff q_t is reachable from q_s, and

$$p = q_0 \xrightarrow{\xi_0} q_1 \xrightarrow{\xi_1} \ldots \xrightarrow{\xi_{l-2}} q_{l-1} \tag{7.6}$$

be a path within the substructure. If p is a cycle of length 1, then $q_0 = q_{l-1}$ and q_0 is reachable from itself, which would not allow for R to be irreflexive. If p is a path of length greater than 1 having no cyclic subpath, then there is at least one state q_i between q_0 and q_{l-1} which is not equal to q_0 or q_{l-1}. If p is itself a cycle, then both holds that q_i is reachable from q_0 and q_0 is

reachable from q_i, which would not allow for R to be antisymmetric. Every path respects transitivity, since every state within a path is reachable from all those preceding them. Hence, R can only be irreflexive, antisymmetric and transitive iff the substructure contains no cycles. The existence of R implies the existence of at least one sequential ordering of the states in Q' compatible with R. $\qquad\square$

7.6   Behaviour

We give here the general definitions and equations that describe the language represented by a FSM.

Definition 82 (Execution state). *The execution states (ESs) of a given algorithm of application of a machine are structures composed by, at least, a machine state plus, possibly, other additional data which depend on the algorithm. These ESs represent execution contexts or partial computations that are performed in some order up to obtaining the final result. If the algorithm does not require any additional information, ESs are simply states of the machine rather than structures.*

Definition 83 (X). *We define X as the set of all possible ESs of an algorithm of application of a machine.*

For instance, ESs of an algorithm of application of FSTSOs may not only include a reached state q but also the partial output that has been generated from an initial state up to reaching q. Algorithms of application of FSAs require no further information, so $X = Q$ in those cases. Every machine can be reduced to a either finite or infinite-state automaton; for instance, we replace each state q of a FSTSO by its execution states (q, z), for every partial output z that can be generated from an initial state up to reaching q, and we copy each transition incoming to or outgoing from q but without the output, and having the corresponding ESs as source and target states. In other words, outputs are coded within the states of the machine rather than within the transitions. Since ESs having different partial outputs will no longer be the same ES, independently of whether they share the same state or not, the resulting machine will have an infinite number of states if an infinite number of partial outputs is possible; hence, this sort of machine is not to be generated for practical NLP, but only the necessary "real" states of

the machine —its ESs— are to be produced for each input, and the algorithm execution will finish as long as the number of ESs to produce for each input is finite. We will give an exact definition of ES for each algorithm and machine in their corresponding sections.

Definition 84 (Multiple ES). *A multiple ES is a set of execution states (SES) composed by every ES that an algorithm of execution of a machine can generate for a given input sequence; since machines may define several paths leading to different ESs for the same input sequence, we consider that the machine is able to be taken to a multiplicity of ESs at a given execution time. Throughout this dissertation, V and W will be used as SES identifiers.*

For instance, processing an ambiguous sentence will lead to multiple ESs at some execution time, one for each possible sentence interpretation considered in the grammar.

Definition 85 (Illegal ES). *Illegal ESs are a special kind of ESs which avoid the traversal of any transition that ends at them.*

For instance, machines extended with unification processes define as illegal every ES containing the null feature structure, that is, the result of unifying two incompatible feature structures; transitions that result in such illegal ESs cannot be traversed.

Definition 86 (Realizable transition). *We say a transition t within a FSM is realizable from a given legal (source) ES x_s iff, the machine being in ES x_s, the transition t can be traversed for some input symbol; in other words, t is an outgoing transition of the state associated to the ES x_s and the machine is taken to a legal (target) ES x_t from ES x_s by traversing t, and consecuently by executing the actions associated to t, if any (e.g.: consuming the current input symbol). In general, we say a transition is either realizable or not depending on the existence of some input sequence which allows for producing a legal ES x_s from where t can be realized.*

Following the former example, a transition requiring to unify two incompatible feature structures is not realizable even if the transition consumes the current input symbol.

Definition 87 (Realization of pure ε-transitions). *Given a legal ES $x_s = (q_s, a_0 \ldots a_{n-1})$, where $a_0 \ldots a_{n-1}$ is the additional data produced by the algorithm*

of application of the corresponding machine, a pure ε-*transition* $t = (q_s, \xi, q_t)$ *is realizable by bringing the machine to ES* $x_t = (q_t, a_0 \ldots a_{n-1})$. *Since pure* ε-*transitions do not impose any restriction to their traversal, every* ε-*transition is realizable, in general.*

Note that pure ε-transitions do not associate any action to their traversal, hence they do not modify the current context of execution except for the machine state.

Definition 88 (Realization of pure consuming transitions). *Given a legal ES* $x_s = (q_s, a_0 \ldots a_{n-1})$, *where* $a_0 \ldots a_{n-1}$ *is the additional data imposed by the algorithm of application of a machine, a pure consuming transition* $t = (q_s, \xi, q_t)$ *is realizable by consuming the input symbol specified in* ξ *and by bringing the machine to ES* $x_t = (q_t, a_0 \ldots a_{n-1})$. *Since pure consuming transitions do not impose any other restriction to their traversal than the presence of some input symbol at the current input point, every pure consuming transition is realizable, in general.*

As for the case of pure ε-transitions, since pure consuming transitions do not associate any action to their traversal other than the consumption of the current input symbol, the additional data of the ES is not modified.

Definition 89 (Realizable path). *A path within a FSM is realizable from a given legal ES* x_s *iff, the machine being in ES* x_s, *every transition within the path is consecutively realizable.*

Definition 90 (Execution path). *Given a realizable path*

$$p = t_0 t_1 \ldots t_n = q_0 \xrightarrow{\xi_0} q_1 \xrightarrow{\xi_1} q_2 \xrightarrow{\xi_2} \ldots \xrightarrow{\xi_{n-1}} q_n \xrightarrow{\xi_n} q_{n+1}$$

from an ES x_0, *we define its execution path from* x_0, $\mathcal{X}(p, x_0)$, *as*

$$\mathcal{X}(p, x_0) = x_0 \xrightarrow{\xi_0} x_1 \xrightarrow{\xi_1} x_2 \xrightarrow{\xi_2} \ldots \xrightarrow{\xi_{n-1}} x_n \xrightarrow{\xi_n} x_{n+1}$$

where x_1 *is the ES the machine is taken to by transition* t_0 *from ES* x_0, x_2 *the ES the machine is taken to by transition* t_1 *from ES* x_1, *and so on. We also call* $\mathcal{X}(p, x_0)$ *the realization of path p from ES* x_0. *If the ESs of a machine are states in* Q, *there is no difference between paths and their realizations.*

Definition 91 (Reachable ES). *Given two legal ESs x_t and x_s, we say x_t is reachable or derivable from x_s iff there exists at least one realizable path p whose execution from ES x_s brings the machine to ES x_t. We say x_t is reachable or derivable from x_s through path p.*

Definition 92 (Deterministic machine). *A machine is deterministic iff it holds the following properties:*

- *to have at most one initial state,*

- *not to have transitions that can be realized independently of the execution context,[6] and*

- *for every possible execution context of the machine there is at most one realizable transition.[7]*

Definition 93 (Equivalent machines). *In general, we say two machines are equivalent iff for every possible input they yield the same output, no matter how the machines are structured.*

Depending on the type of output generated by a class of machines, a particular definition of equivalence will be given (considering that a Boolean indicating whether an input sequence belongs to a language or not is already an output).

Definition 94 (Minimal machine). *We say a machine is minimal iff there exists no other equivalent machine having a smaller number of states.*

Machines of different types may be yet equivalent if they produce the same kind of output; for instance, both FSAs and RTNs return a Boolean value. However, some machine types may allow for more compact structures; for instance, FSAs allow for factoring out common prefixes and suffixes of the represented sequences, while RTNs can also factor out common infixes. In general, when we speak about minimal machines we are restricting the type of the equivalent machine to the one of the original machine.

---

[6]For instance, ε-transitions in FSAs. Recall that, apart from the input, execution contexts may comprise other data (e.g.: the output generated up to reaching a certain ES).

[7]For instance, deterministic FSAs have no pair of transitions outgoing from the same source state so that both are labeled with the same input symbol; otherwise both transitions might be realizable under the same execution context, that is, being at the source state of these transitions.

Definition 95 (Derivation rule)**.** *The set of derivation rules or derivation mechanisms of a machine describe the ES directly reachable from any given ES of the machine. For the case of FSMs, each transition class is associated to a derivation rule and each transition represents a particular case of such a rule for a particular pair of source and target states and transition condition (expressed by the transition's label). The realization of a transition consists in applying the derivation rule corresponding to the transition type.*

For instance, definitions 87 and 88 describe the derivation rules associated to pure ε-transitions and pure consuming transitions, respectively.

Definition 96 (Δ)**.** *We define function*

$$\Delta : \mathcal{P}(X) \times \Sigma \to \mathcal{P}(X) \tag{7.7}$$

as the extension of the consuming cases of transition function δ to source and target SESs instead of simple source and target states of the machine, that is, $\Delta(V, \sigma)$ is equal to the set of directly reachable ESs from any ES of V through transitions that consume the next input symbol σ. The exact behaviour of function Δ depends on the type of machine and algorithm followed.

Definition 97 (D)**.** *Analogous to function Δ, we define function*

$$D : \mathcal{P}(X) \to \mathcal{P}(X) \tag{7.8}$$

as the extension of the non-consuming cases of transition function δ to source and target SESs, that is, $D(V)$ is equal to the set of directly ε-reachable ESs from any ES of V. Let A be a machine with n ε-transition types,[8] we define D as

$$D : \bigcup_{i=0}^{n} D_i, \tag{7.9}$$

where $D_i(V)$ represents a particular derivation rule of $D(V)$ for a type of ε-transition. The exact behaviour of each D_i function depends on the type of machine and algorithm followed. For the sake of simplicity, if $i = 1$ we do not define a D_1 function but we define D itself as the direct-derivation function on SESs.

---

[8]for instance, RTNs have three different kinds of ε-transitions: explicit ε-transitions, push transitions and pop transitions; RTNs will be the object of chapter 12.

Definition 98 (Simple direct-derivation function on SESs). *A simple direct-derivation function on SESs is an extension of a particular derivation case of transition function δ to source and target SESs, namely function Δ and the D_i functions composing function D (or D for machines with a unique type of ε-transition).*

In general, simple direct-derivation functions on SESs are all defined by an expression of the form

$$\mathcal{F}(V,\sigma) = \{x_t : d \wedge x_s \in V\}, \tag{7.10}$$

where V is the source SES, σ is the current input symbol (for the case of Δ) or is omitted (for the case of D_i functions or function D), d is a predicate that depends on the followed derivation rule, and x_t is the target ES derived from source ES x_s if d holds. For instance, for the case of FSAs, $\Delta(V,\sigma)$ is defined as follows:

$$\Delta(V,\sigma) = \{q_t : (q_t \in \delta(\sigma)) \wedge (q_s \in V)\}, \tag{7.11}$$

where q_t and q_s correspond to x_t and x_s (for the case of FSAs, ESs are simple FSA states), and '$q_t \in \delta(\sigma)$' is the *derivation predicate*. In order to avoid repetition, we will define simple direct-derivation functions on SESs for each algorithm and machine by specifying x_s, x_t and d. This generalization will also be used for studying some properties common to every simple direct-derivation function on SESs.

Definition 99 (*i*-recursive function application). *Let $\mathcal{F}$ be a function of a set A into itself, that is, $\mathcal{F} : A \to A$, we define $\mathcal{F}^i$, the i-recursive application of $\mathcal{F}$, as*

- *the composition of $\mathcal{F}$ with itself $i - 1$ times, for $i > 1$*

- *the function itself, for $i = 1$, and*

- *id_A, the identity function of A, for $i = 0$.*

For instance, let f be a function of $\mathbb{N}$ into itself such that $f(x) = x + 1$,

the following equations hold:

$$
\begin{array}{rclclcl}
f^0(0) & = & \mathrm{id}(0) & = & 0 & & \\
f^1(0) & = & f(0) & = & 0+1 & = & 1 \\
f^2(0) & = & f(f(0)) & = & 0+1+1 & = & 2 \\
f^3(0) & = & f(f^2(0)) & = & 2+1 & = & 3 \\
& & \vdots & & & & \\
f^n(0) & = & f(f^{n-1}(0)) & = & n-1+1 & = & n
\end{array}
\tag{7.12}
$$

Definition 100 (ε-closure). *We define the ε-closure of a SES V as the SES containing V and every ε-reachable ES from any ES of V:*

$$
C_\varepsilon : \mathcal{P}(X) \to \mathcal{P}(X)
$$

$$
C_\varepsilon(V) = \bigcup_{i=0}^{m} D^i(V),
\tag{7.13}
$$

that is, the ESs of V plus the ESs reachable from any ES of V through one up to m ε-transitions, where m is the smallest k such that

$$
\bigcup_{i=0}^{k} D^i(V) = \bigcup_{i=0}^{k+1} D^i(V),
\tag{7.14}
$$

if such k exists, and undefined otherwise.

Indeed, there are machines having SESs for which such k does not exist and, hence, the ε-closure is not computable. For each kind of machine, we will identify the substructures allowing for such SESs, if any, in order to avoid them.

Definition 101 (Delayability of union). *Given n subsets V_i of a set X and a unary function $\mathcal{F} : \mathcal{P}(X) \to \mathcal{P}(X)$, we say that the union of sets is delayable w.r.t. $\mathcal{F}$ iff*

$$
\mathcal{F}(\bigcup_i V_i) = \bigcup_i \mathcal{F}(V_i).
\tag{7.15}
$$

Since the union of sets is associative, if the previous equality holds for the union of two sets then it holds for the union of two or more sets.

Lemma 2 (FSM D-union). *The union of sets is delayable w.r.t. function D if it is delayable for each D_i function composing D.*

Proof. Let D be the union of two functions D_1 and D_2 such that the union of sets is delayable w.r.t. both of them, and let V and V' be two SESs, then it holds that

$$
\begin{aligned}
D(V \cup V') &= D_1(V \cup V') \cup D_2(V \cup V') & (7.16)\\
&= D_1(V) \cup D_1(V') \cup D_2(V) \cup D_2(V') & (7.17)\\
&= D_1(V) \cup D_2(V) \cup D_1(V') \cup D_2(V') & (7.18)\\
&= D(V) \cup D(V'). & (7.19)
\end{aligned}
$$

$\square$

Theorem 1 (Union of direct-derivation functions on SESs). *The union of sets is delayable w.r.t. direct-derivation functions on SESs.*

Proof. Let $\mathcal{F}$ be a direct-derivation function on SESs, that is, a function of the form

$$
\mathcal{F}(V, \sigma) = \{x_t : d \wedge x_s \in V\}, \tag{7.20}
$$

where d is the derivation predicate, the following equations hold:

$$
\begin{aligned}
\mathcal{F}(V \cup V') &= \{x_t : d \wedge x_s \in V \cup V'\}\\
&= \{x_t : d \wedge (x_s \in V \vee x_s \in V')\}\\
&= \{x_t : d \wedge x_s \in V \vee d \wedge x_s \in V'\}\\
&= \{x_t : d \wedge x_s \in V)\} \cup \{x_t : d \wedge x_s \in V')\}\\
&= \mathcal{F}(V) \cup \mathcal{F}(V').
\end{aligned}
$$

$\square$

Theorem 2 (D-union). *Given lemma 2 and theorem 1, the union of sets is delayable w.r.t. the D function, in general.*

Lemma 3 (ε-closure-union). *Given a FSM, if the union of sets is delayable w.r.t. its D function then it is delayable as well w.r.t. its ε-closure function:*

$$
D(\bigcup_i V_i) = \bigcup_i D(V_i) \Longrightarrow C_\varepsilon(\bigcup_i V_i) = \bigcup_i C_\varepsilon(V_i). \tag{7.21}
$$

Proof. Let V and V' be two SESs, it holds that

$$
\begin{aligned}
C_\varepsilon(V \cup V') &= \bigcup_{j=0}^{m} D^j(V \cup V') \\
&= \bigcup_{j=0}^{m} D^j(V) \cup \bigcup_{j=0}^{m} D^j(V') \\
&= C_\varepsilon(V) \cup C_\varepsilon(V'),
\end{aligned}
$$

where m is the smallest k such that

$$
\bigcup_{i=0}^{k} D^i(V) = \bigcup_{i=0}^{k+1} D^i(V).
$$

$\square$

Theorem 3 (ε-closure-union). *Given lemma 2 and lemma 3, the union of sets is delayable w.r.t. the ε-closure, in general.*

Lemma 4 (Iterative ε-closure). *Since the union of sets is delayable w.r.t. the D function, the following is an equivalent definition of ε-closure, based on iterative computation:*

$$
C_\varepsilon(V_0) = V_m \quad with \quad V_{i+1} = V_i \cup D(V_i), \quad i > 0, \tag{7.22}
$$

and m is the smallest k such that $V_{k+1} = V_k$.

Proof. By computing the different V_i's we obtain:

$$
\begin{aligned}
V_1 &= V_0 \cup D(V_0) = \bigcup_{j=0}^{1} D^j(V_0) \\
V_2 &= V_0 \cup D(V_0) \cup D(V_0 \cup D(V_0)) \\
&= V_0 \cup D(V_0) \cup D(V_0) \cup D(D(V_0)) \\
&= V \cup D(V_0) \cup D(D(V_0)) = \bigcup_{j=0}^{2} D^j(V_0) \\
&\vdots \\
V_i &= \bigcup_{j=0}^{i} D^j(V_0)
\end{aligned}
$$

From the development of term V_m we obtain the first definition of ε-closure.

□

Lemma 5 (Finite ε-closure). *Given the iterative definition of ε-closure of lemma 4, if there exists a natural number $k \geq 0$ such that $V_k = V_{k+1}$ then $V_k = V_l$, for $l \geq k$, that is, once V_k is computed, computing further V_i's will not add anything to the ε-closure and, hence, the ε-closure will be finite.*

Proof. Let k be the smallest number such that $V_k = V_{k+1} = D(V_k)$, and l be a number greater than k, then

$$
\begin{aligned}
V_l &= V_k \cup D(V_k) \cup D(D(V_k)) \cup \ldots \cup D^{l-k}(V_k) \\
&= V_k \cup V_k \cup D(V_k) \cup \ldots \cup D^{i-k-1}(V_k) \\
&\vdots \\
&= V_k \cup V_k \cup V_k \cup \ldots \cup V_k \\
&= V_k
\end{aligned}
$$

and therefore computing further V_i's after V_k will not add anything to $C_\varepsilon(V_0)$.

□

Definition 102 (Δ^*). *We recursively define Δ^*, the extension of the transition function over SESs Δ for an input sequence $w \in \Sigma^*$, as follows:*

$$\Delta^* : \mathcal{P}(X) \times \Sigma^* \to \mathcal{P}(X)$$

$$
\begin{aligned}
\Delta^*(V, \varepsilon) &= C_\varepsilon(V) & (7.23) \\
\Delta^*(V, w\sigma) &= C_\varepsilon(\Delta(\Delta^*(V, w), \sigma)) & (7.24)
\end{aligned}
$$

This definition is analogous to that of $\hat{\delta}$ for NFAs (non-deterministic FSAs) given in Hopcroft et al. (2000, sec. 2.3.3, p. 58), though Δ^* is defined on SES and $\hat{\delta}$ is defined on NFA states.

Definition 103 (Initial and acceptance SESs). *We call X_I the initial SES of a FSM and X_F its acceptance or final SES. ESs in X_I or X_F are structures containing a state in Q_I or F, respectively, plus any additional information required to represent the initial or acceptance ES depending on the type of machine and algorithm followed. When no additional information is required, as in FSAs, $X_I = Q_I$ and $X_F = F$.*

Definition 104 (Deterministic execution of a FSM). *Let A be a FSM with an input alphabet Σ and a set of initial ESs X_I, we say A is deterministic —in terms of execution— iff it holds that*

$$|\Delta^*(X_I, w)| \leq 1, \quad for \ all \quad w \in \Sigma^*, \tag{7.25}$$

that is, A has a unique initial ES x_0 and the number of reachable ESs from x_0 by consuming any input sequence is at most one.

Definition 105 (Execution machine). *Given a FSM A with Σ as input alphabet, X as its ES domain, X_I as initial SES and X_F as acceptance SES, we define $\mathcal{X}(A)$, the execution machine of A, as an either finite- or infinite-state machine (depending on each case) having at least the following elements*

- *X as set of states, either finite or infinite,*

- *X_I as set of initial states, finite,*

- *X_F as set of acceptance states, either finite or infinite,*

- *Σ as input alphabet, either finite (for letter machines) or infinite (for machines on an alphabet of words or predicates), and*

- *δ' as partial transition function, either finite or infinite, such that*

  - *$x_t \in \delta'(x_s, \sigma) \iff$ the execution of A can directly derive x_t from x_s by consuming σ,*

  - *$x_t \in \delta'(x_s, \varepsilon) \iff$ the execution of A can derive x_t from x_s without input consumption, and*

  - *possibly other transitions depending on the type of machine and algorithm of application.*

Execution machines can be seen as execution traces of the algorithm of application of the machines for every possible input. The exact definition of execution machine depends on the kind of machine and execution method.

Note that for FSMs having an infinite number of ESs (e.g.: FSTs representing an infinite number of translations), the number of transitions linking the states of their corresponding execution machines will also be infinite. As stated before, execution machines are not to be entirely computed but only the necessary substructures for the application of FSMs to specific input

sequences. The concept of execution machine will be used for the case of machines with output generation in order to study the possibility of computing the different ESs in some specific order that will accelerate the machine application: the order given by a topological sort of the execution machine substructures produced by derivation functions on SESs. In particular, we will study the necessary conditions for the existence of such topological sorts.

Definition 106 ($\mathcal{F}$-substructure). *Given a derivation function $\mathcal{F}$ on SES of a machine A, we define the $\mathcal{F}(V)$-substructure of $\mathcal{X}(A)$ as the substructure of $\mathcal{X}(A)$ composed by every ES in V and every reached ES and traversed transition during the computation of $\mathcal{F}(V)$.*

Definition 107 (L). *We define $L(A)$, the language of a FSM A, as the set of sequences $w \in \Sigma^*$ recognized by A, that is, the set of sequences whose whole consumption reaches at least one acceptance ES from at least one initial ES:*

$$L(A) = \{w \in \Sigma^* : \Delta^*(X_I, w) \cap X_F \neq \emptyset\}. \tag{7.26}$$

This formula is similar to that of Hopcroft et al. (2000, sec. 2.3.4, p. 59), though using function Δ^* instead of $\hat{\delta}$, an initial SES X_I instead of a single initial state q_0, and an acceptance SESs X_F instead of a set of acceptance states F.

We say that a word w is accepted or recognized by a FSM A iff w belongs to the language of A; otherwise we say that w is rejected or not recognized by the FSM.

Definition 108 ($L_R(x)$). *Let x be an ES of a FSM, we define $L_R(x)$, the right language of x, as*

$$L_R(x) = \{w \in \Sigma^* : \Delta^*(\{x\}, w) \cap X_F \neq \emptyset\}. \tag{7.27}$$

Definition 109 (Acceptor machine). *We say that a machine or an algorithm of application of a machine is a pure acceptor iff its sole purpose is to compute the acceptance/rejection of an input sequence.*

In definition 93 (p. 135) we introduced the concept of equivalence between machines. Once defined what a pure acceptor machine is, we can give a more concrete definition of equivalence between pure acceptor machines:

Definition 110 (Equivalent pure acceptor machines). *We say two pure acceptor machines A and A' are equivalent iff $L(A) = L(A')$.*

Definition 111 (Interpretation). *Given a word w and a machine A, we say a path p within A is an interpretation of w iff $\mathcal{X}(p, x_s)$ reaches an acceptance ES from an initial ES x_s by consuming w.*

Definition 112 (Ambiguous word). *We say a word w is ambiguous for a given machine A iff there exist several interpretations within A that consume w.*

Definition 113 (Ambiguous machine and language). *Machines recognizing at least one word through several interpretations are said to be ambiguous, and so are their accepted languages.*

Definition 114 (Useful state). *We say a state is useful iff there exists at least one interpretation traversing the state; otherwise we say it is useless.*

Definition 115 (Useful transition). *We say a transition is useful iff it is a part of at least one interpretation; otherwise we say it is useless.*

Definition 116 (Useful path). *We say a path is useful iff it is a subpath of at least one interpretation; otherwise we say it is useless.*

Definition 117 (Useful machine substructure). *We say a machine substructure (Q', δ') is useful iff it contains at least one useful state. Conversely, we say a machine substructure is useless iff it contains no useful states.*

Corollary 3 (Transitions of machine subtructures and usefulness). *A machine substructure that contains useful transitions is also useful, since the usefulness of a transition implies the usefulness of its source and target states, which also belong to the machine substructure. Conversely, machine substructures containing no useful states neither contain useful transitions.*

One may be tempted to say that a machine is useless iff it contains no interpretation. However, such machines may have a purpose analogous to that of ε (to represent the empty symbol or the empty string), $\emptyset$ (to represent the empty set), or 0 (to represent the null quantity).

Definition 118 (Trimmed FSM). *We say a FSM is trimmed iff it contains no useless states or transitions.*

Definition 119 (w-usefulness). *We say a path, a transition or a state is useful for a given input sequence w, or simply w-useful, iff there exists an interpretation of w traversing such path, transition or state.*

Definition 120 (Pruning). *We call pruning the process of removing every useless substructure of a machine.*

Corollary 4 (Result of pruning). *The result of pruning a machine is either the empty machine or a machine without useless states or transitions.*

Some of the algorithms we will present in this dissertation require the generation of the language of a new kind of machine we have called filtered-popping recursive transition network (FPRTN). We will first study the cases in which such language is finite, in order to guarantee that the execution of such algorithms will finish. FPRTNs can be seen as RTNs whose popping transitions are not always realizable, and RTNs can be seen as FSAs extended with call, push and pop transitions. Apart from these transitions, the remaining transitions are the same and, hence, the corresponding substructures will have the same behaviour. Though it may seem obvious which kind of FSA and RTN substructures lead to infinite languages, that is not the case for FPRTNs. We will present the FPRTN case by extending the simpler cases, starting here with the general condition for any FSM.

Theorem 4 (Cardinality of the interpretation set). *The number of interpretations of a FSM without useful cycles is finite; otherwise, the number of interpretations is infinite iff the machine allows for the realization of an infinite number of self-concatenations of at least one useful cycle, and an infinite subset of the realizable self-concatenations is useful.*

Proof. Let it be a FSM A either without cycles or with useless cycles; since for any interpretation p the same state cannot be traversed twice, every interpretation must be formed by a sequence of connected transitions traversing a sequence of states without repetitions. The number of subsets of states of a machine is equal to

$$|\mathcal{P}(Q)| = 2^{|Q|}, \tag{7.28}$$

which is finite since Q is finite. For every subset of states $Q_i \subseteq Q$, the number of permutations without repetitions of the states in Q_i is $|Q_i|!$, giving a total of

$$\sum_{i=0}^{|\mathcal{P}(Q)|} |Q_i|! \tag{7.29}$$

possible sequences of states without repetitions, which is also a finite number since Q_i is finite for $i = 0 \ldots |\mathcal{P}(Q)|$. Moreover, not every sequence of states

will be possible for a machine without useful cycles since two interpretations without cycles may allow for the existence of a third interpretation with cycles (e.g: interpretations with sequences of states $q_0 q_1$ and $q_1 q_0$ may allow for an interpretation with state sequence $q_0 q_1 q_0$). Finally, for each sequence to be an interpretation, at least one realizable transition (q_j, ξ_j, q_{j+1}) must exist for each two consecutive states q_j and q_{j+1} within the sequence, each additional realizable transition allowing for an additional interpretation. However, since the number of transitions is finite, the total number of interpretations is also finite.

Otherwise, if A contains a useful cycle p_b for a given interpretation $p = p_a p_b p_c$, then A contains the infinite family of paths $p_i = p_a p_b^i p_c$ for $i \geq 0$; if an infinite subset of this family is realizable, then the number of interpretations of the machine is infinite iff p_i is an interpretation for some infinite set of values of i, that is, an infinite set of useful self-concatenations of a cycle is required for a machine to have an infinite set of interpretations. □

The proof's last paragraph may remind the reader of the pumping lemma for regular expressions (see Hopcroft et al., 2000, sec. 4.1.1, p. 126 or Sipser, 2006, sec. 1.4, p. 77) but from a more general perspective: the pumping lemma states that regular languages may not be composed by an infinite number of random sequences but, at most, by an infinite number of words that are built by the repeated self-concatenation of some finite set of random subsequences, and our proof states that cycles within FSMs may allow for an infinite number of interpretations by consecutively repeating the processing associated to the cycles, whatever the processing may consist in. Indeed, some of the machines we will present in this dissertation are equivalent to Turing machines and, therefore, go beyond regular and context-free languages.

Theorem 5 (Cardinality of the language)**.** *The language of a machine is infinite iff it contains at least one useful consuming cycle p and an infinite set of self-concatenations of p is useful.*

7.7 Reverse FSM

Some of the algorithms presented in this dissertation require to reversely traverse a FSM, namely: a general minimization algorithm (sec. 8.6, p. 174), a FPRTN pruning algorithm (sec. 16.1, p. 325) and an algorithm extracting

the top-ranked output represented by a FPRTN (alg. 18.2, p. 346). As for the cardinality of the language of FPRTNs, we will define the canonical reverse of a FPRTN as the extension of simpler cases, starting here with the general definition of reverse FSM.

Definition 121 (Reverse transition)**.** *Let t be a transition (q_s, ξ, q_t), we define t^R, the reverse of t, as (q_t, ξ, q_s)*

Definition 122 (Reverse sequence)**.** *Let A be a set of elements, a be an element of A, α and β be two sequences of zero, one or more elements of A and ε be the empty sequence, we define α^R, the reverse of α, as*

$$\alpha^R = \left\{ \begin{array}{ll} \varepsilon, & \alpha = \varepsilon \\ a\beta^R, & \alpha = \beta a \end{array} \right. \tag{7.30}$$

Definition 123 (Reverse path)**.** *We define p^R, the reverse of a path p, as the result of reversing the sequence of transitions forming p and then replacing each transition by its reverse, that is,*

$$p = t_0 t_1 \ldots t_n \quad \text{iff} \quad p^R = t_n^R \ldots t_1^R t_0^R. \tag{7.31}$$

Definition 124 (L^R)**.** *Let $L(A) = \{w_0, \ldots, w_{n-1}\}$ be the language of a machine A, we define $L^R(A)$, the reverse language of A, as $\{w_0^R, \ldots, w_{n-1}^R\}$, that is, reversed word $w_i^R \in L^R(A)$ iff $w_i \in L(A)$, for $i = 0 \ldots n-1$.*

Corollary 5 (Cardinality of L^R)**.** *Let A be a FSM, the cardinality of $L^R(A)$ is equal to the cardinality of $L(A)$*

Definition 125 (Reverse machine)**.** *We say a machine B is a reverse of a machine A iff $L(A) = L^R(B)$.*

Definition 126 (Canonical reverse machine)**.** *Let A be a FSM, we define A^R, the canonical reverse of A, as the result of reversing machine A by means of a particular procedure which is to be defined for each particular kind of machine.*

For all the machines presented here, their canonical reverses are machines of the same kind, except for FPRTNs: as we will see in chapter 15, reversing a filtered-*popping* recursive transition network results in a filtered-*pushing* recursive transition network and vice-versa.

7.8   Efficient computation of the ε-closure

In this section we derive from the iterative definition of ε-closure (lemma 4, p. 140) an equivalent and more efficient definition so that at each iteration we only consider the ESs from where new ESs may be ε-derived, and reuse previous computations as far as possible. We will then give an algorithm for the computation of the ε-closure of any kind of FSM, based on this definition.

Definition 127 (ε-expansion). *We define $E(V)$, the ε-expansion of a SES V, as the set of directly ε-reachable ESs from any ES of V that is not already present in V:*

$$E : \mathcal{P}(X) \to \mathcal{P}(X)$$

$$E(V) = D(V) - V \tag{7.32}$$

Lemma 6 (ε-expansion-based ε-closure). *Since the union of sets is delayable w.r.t. the D function (theorem 2, p. 139), the following is an equivalent and more efficient definition of ε-closure, based on successive ε-expansions:*

$$C_\varepsilon(V_0) = C'_\varepsilon(V_0, E(V_0)) \tag{7.33}$$

where C'_ε is an auxiliary function which is recursively defined as follows:

$$C'_\varepsilon : \mathcal{P}(X) \times \mathcal{P}(X) \to \mathcal{P}(X)$$

$$C'_\varepsilon(V_i, E_i) = \left\{ \begin{array}{ll} V_i, & E_i = \emptyset \\ C'_\varepsilon(V_i \cup E_i, D(E_i) - (V_i \cup E_i)), & E_i \neq \emptyset, \end{array} \right. \tag{7.34}$$

being V_i the SES resulting from the i-recursive call to function C'_ε and E_i the ε-expansion of V_i.

As we will see, E_i is eventually to be empty for the case of finite ε-closures.

Proof. Following the iterative definition of ε-closure of lemma 4, we compute the ε-closure of a SES V_0 by generating the successive V_i such that each one contains V_0 plus the ε-reachable states through i ε-transitions, that is, we increment V_0 with the reachable ESs through one ε-transition, two ε-transitions and so on. The new ESs appearing at a SES V_{i+2} will come from the ESs that were not formerly considered during the computation of $D(V_i)$;

rather than computing the new ε-reachable ESs at V_{i+2} from every state of V_{i+1}, we only consider the states $V_{i+1} - V_i$, that is, the ε-expansion of V_i:

$$
\begin{aligned}
V_{i+1} - V_i &= (V_i \cup D(V_i)) - V_i \\
&= (V_i - V_i) \cup (D(V_i) - V_i) \\
&= D(V_i) - V_i \\
&= E(V_i).
\end{aligned}
$$

Hence, we can reformulate the ε-closure definition as

$$
C_\varepsilon(V_0) = V_m \quad \text{such that} \quad V_{i+1} = V_i \cup E(V_i) \tag{7.35}
$$

and m is the smallest k such that $V_{i+1} = V_i$. If such k exists, then the ε-closure is finite (see lemma 5) and its computation will finish once the first empty ε-expansion is reached:

$$
\begin{aligned}
C_\varepsilon(V) = V_k : V_k = V_{k+1} &\iff V_k = V_k \cup D(V_k) \\
&\iff D(V_k) \subseteq V_k \\
&\iff D(V_k) - V_k = \emptyset \\
&\iff E(V_k) = \emptyset
\end{aligned}
$$

As well, we do not require to compute at each iteration i the ε-expansion from the whole set V_i but from the previous ε-expansion: let $E_i = E(V_i)$ for $i \geq 0$, it holds that

$$
\begin{aligned}
E_{i+1} &= E(V_{i+1}) \\
&= D(V_{i+1}) - V_{i+1} \\
&= D(V_i \cup E_i) - V_{i+1} \\
&= (D(V_i) \cup D(E_i)) - V_{i+1} \\
&= (D(V_i) - V_{i+1}) \cup (D(E_i) - V_{i+1}) \\
&= (D(V_i) - (V_i \cup D(V_i)) \cup (D(E_i) - V_{i+1}) \\
&= D(E_i) - V_{i+1} \\
&= D(E_i) - (V_i \cup E_i).
\end{aligned}
$$

Therefore, we can efficiently compute the ε-closure of V by following the iterative procedure below:

$$
\begin{aligned}
V_0 &= V \\
V_{i+1} &= V_i \cup E_i
\end{aligned}
$$

where

$$
\begin{aligned}
E_0 &= E(V_0) \\
E_{i+1} &= D(E_i) - V_{i+1}
\end{aligned}
$$

until reaching an $E_k = \emptyset$; by developing the equations of lemma 6 we obtain this pattern. □

Algorithm 7.1 *fsm_eexpansion_eclosure* is an implementation of the ε-closure based on ε-expansions (lemma 6), which uses algorithm 7.2 *fsm_eexpansion* in order to compute the successive ε-expansions. The implementation of function D, which is used for the computation of the ε-expansion, depends on the type of machine.

Algorithm 7.1 fsm_eexpansion_eclosure(V) $\triangleright C_\varepsilon(V)$, lem. 6

Input: V, the SES whose ε-closure is to be computed
Output: V after computing its ε-closure
 1: $E = \text{fsm\_eexpansion}(V)$
 2: **while** $E \neq \emptyset$ **do**
 3:     $V \leftarrow V \cup E$
 4:     $E \leftarrow \text{fsm\_eexpansion}(E)$
 5: **end while**

Algorithm 7.2 fsm_eexpansion(V) $\triangleright E(V)$, def. 127

Input: V, the SES whose ε-expansion is to be computed
Output: E, the ε-expansion of V
 1: **for each** $x_t \in D(V)$ **do**
 2:     **if** $x_t \notin V$ **then**
 3:         $E \leftarrow E \cup x_t$
 4:     **end if**
 5: **end for**

Finally, algorithm 7.3 *fsm_interlaced_eclosure* is a more efficient procedure for the computation of the ε-closure, also based on ε-expansions. Instead of computing the whole ε-expansion at each iteration and then adding it to V, it adds new states to V as they are found and it keeps a queue E of unexplored states that grows with each new ES found and decreases each time one

of its ESs is explored; the construction of the different ε-expansions is interlaced with the construction of the ε-closure. Algorithm 7.4 *add_enqueue_es* is a small routine used for adding ε-derived ESs to the SES whose ε-closure is to be computed. This routine is further used in the algorithm computing the Δ function of V (algorithm 7.6 *fsm_recognize_symbol*) in order to add ESs reached through the consumption of an input symbol.

Algorithm 7.3 fsm_interlaced_eclosure(V, E) $\qquad\qquad \triangleright C_\varepsilon(V, E)$, lem. 6

Input: V, the SES whose ε-closure is to be computed
$\qquad\quad E$, the queue of unexplored ESs containing every ES in V
Output: V after computing its ε-closure
$\qquad\qquad E$ after emptying it
 1: **while** $E \neq \emptyset$ **do**
 2: $\quad$ $x_s \leftarrow$ dequeue(E)
 3: $\quad$ **for each** $x_t \in D(x_s)$ **do**
 4: $\quad\quad$ add_enqueue_es(V, E, x_t)
 5: $\quad$ **end for**
 6: **end while**

Algorithm 7.4 add_enqueue_es(V, E, x_t)

Input: V, the SES where the ES is added
$\qquad\quad E$, the queue of unexplored ESs
$\qquad\quad x_t$, the target ES to add to V
Output: V after adding the ES
$\qquad\qquad E$ after enqueuing the ES, if new
 1: **if** add(V, x_t) **then**
 2: $\quad$ enqueue(E, x_t)
 3: **end if**

Notice that the addition of x_t to V by means of function *add* in algorithm 7.4 *add_enqueue_es* requires to check whether the ES already belongs to the SES or not so that the underlying data structure does not represent twice the same element. Thus, we are to use set data structures providing efficient search operations, such as the ones presented in chapter 2. However, if we can ensure that the same element is not going to be added twice during the whole life of the set, and the order in which the elements are to

be retrieved is not important, it is preferable to use a data structure that
sacrifices duplicity checking, element ordering and, consequently, ordered el-
ement retrieval but provides faster add and retrieval operations: for instance
queues and stacks. That is the case of the set of unexplored ESs E and
operation *enqueue* in the same algorithm: we want to add to E the elements
that are new to V, so they are going to be new as well to E, and the order
in which unexplored ESs in E are explored does not modify the algorithm
result. Function *add* is to return a Boolean indicating whether the element
has been added or not to the set so that we can safely enqueue elements in
E. As well, algorithm 7.3 *fsm_interlaced_eclosure* is not to initialize E with
every ES in V but E is to be built along with V by the algorithm computing
$\Delta(V)$ (algorithm 7.6 *fsm_recognize_symbol* in the next section), so that the
latter algorithm can also benefit from the *add*-*enqueue* mechanism.

Algorithm 7.3 *fsm_interlaced_eclosure* can be seen as a generalization
of the algorithm presented in van Noord (2000, sec. 3.2): while van Noord's
algorithm computes the ε-closure of a set of states of a FSA, our algorithm
computes the ε-closure of a SES of any kind of FSM. Side by side, the dif-
ferences between both algorithms are:

- van Noord's algorithm *marks* the states that have been explored, while
  our algorithm explicitly uses a queue E of unexplored ESs,

- van Noord's algorithm first unmarks every state of the set whose ε-
  closure is to be added, while our algorithm expects the initial E to be
  passed as argument (it will be constructed along with the set whose
  ε-closure is to be computed),

- van Noord's algorithm adds to the ε-closure every state q_t such that
  $q_t \in \delta(q_s, \varepsilon)$, with q_s the unmarked states already in the ε-closure, while
  we add the ESs x_t such that $x_t \in D(x_s)$, with x_s the ESs in E.

The equations given in this section, along with the equations given in sec-
tion 7.6 (p. 132) relative to the ε-closure, verify the algorithm correctness for
every kind of ES and D function.

7.9   Recognizing a string

Based on the previous definitions, algorithm 7.5 *fsm_recognize_string* is
a generic breadth-first algorithm which computes the acceptance/rejection

of a given word for a FSM (definition 46, p. 121). This algorithm uses algorithms 7.3 *fsm_interlaced_eclosure* and 7.6 *fsm_recognize_symbol* for the computation of the ε-closure and the Δ function of a SES, respectively. Two small routines are used in order to add an ES to a SES: algorithm 7.4, the same used for the computation of the ε-closure, and algorithm 7.7, a version of the former algorithm which unconditionally adds the ES to the SES. The latter algorithm is to be used whenever it is sure the ES is new so we can omit the conditional jump.

Algorithm 7.5 fsm_recognize_string$(\sigma_1 \ldots \sigma_l)$ $\triangleright \sigma_1 \ldots \sigma_l \in L$, def. (107)

Input: $\sigma_1 \ldots \sigma_l$, an input string of length l
Output: r, a Boolean indicating whether the input string belongs to L
 1: $V \leftarrow \emptyset$
 2: $E \leftarrow \emptyset$
 3: **for each** $x_s \in X_I$ **do**
 4:     unconditionally_add_enqueue_es(V, E, x_s)
 5: **end for**
 6: fsm_interlaced_eclosure(V, E)
 7: $i \leftarrow 0$
 8: **while** $V \neq \emptyset \wedge i < l$ **do**
 9:     $V \leftarrow$ fsm_recognize_symbol(V, E, σ_{i+1})
10:     $i \leftarrow i + 1$
11:     fsm_interlaced_eclosure(V, E)
12: **end while**
13: $r \leftarrow$ false
14: **for each** $q \in V$ **do**
15:     $r \leftarrow r \vee q \in X_F$
16: **end for**

This algorithm is a generalization of the breadth-first translator algorithm for RTNs presented in Sastre and Forcada (2007, 2009), which in turn is based on the algorithm presented in Garrido-Alenda et al. (2002) for the application of deterministic augmented letter transducers. It iteratively computes the Δ^* function: it first initializes V as the initial SES of the machine and marks every initial ES as unexplored for the ε-closure computation, then adds to V its ε-closure and afterwards performs a sequence of iterations so that for each one reinitializes V as the set of reachable ESs from the previous V by consuming the next input symbol and adds its ε-closure. Each time an

---

Algorithm 7.6 fsm_recognize_symbol(V, E, σ) $\triangleright \Delta(V, \sigma)$, def. (96)

Input: V, a SES

        E, the empty queue of unexplored ESs

        σ, the input symbol to recognize

Output: W, the set of reachable ESs from V by consuming σ

        E after enqueuing the ESs of W

  1: $W \leftarrow \emptyset$

  2: **for each** $x_t \in \Delta(V, \sigma)$ **do**

  3:     add_enqueue_es(W, E, x_t)

  4: **end for**

Algorithm 7.7 unconditionally_add_enqueue_es(V, E, x_t)

Input: V, the SES where the ES is added

        E, the queue of unexplored ESs

        x_t, the target ES to add to V

Output: V after adding the ES

        E after enqueuing the ES, if new

  1: add(V, x_t)

  2: enqueue(E, x_t)

ES is to be added, it verifies if the ES is new and, if so, it marks it as unexplored for the computation of the ε-closure by enqueuing it into E, the queue of unexplored ESs. The only exception is when building the initial SES: since every initial ES is unique, all of them are unconditionally marked as unexplored. Iterations proceed until every symbol has been consumed or an empty SES V is reached. After the computation of the ε-closure, an empty queue is returned which is refilled again when consuming the next input symbol. The algorithm accepts the string if the last computed V contains at least one acceptance ES. It is not necessary to explicitly check whether the whole input has been consumed or not: in case an input symbol cannot be consumed, the iterative process will be interrupted after building a last empty SES. Since this algorithm only computes acceptability, it could be further optimized by having a special last iteration which would immediately return true once the first acceptance ES is found, avoiding the construction of the whole last SES. The algorithm only requires to store two SESs: the SESs of the current and the next iteration, the latter stored as a local variable during the evaluation of the expression

$$V \leftarrow \text{fsm\_recognize\_symbol}(V, E, \sigma_{i+1}).$$

Since the algorithm performs a breadth-first exploration of the machine, parallel explorations of the machine will be joined together if they reach the same ES, avoiding the repeated exploration of common paths as happens with depth-first algorithms. Of course, if the machine is determinized (determinization is described in the next section) then there will be a unique path to be explored for every input string, thus there will not be parallel paths to be joined; a simplest algorithm just seeking for the consecutive ESs of the path execution would be more efficient. However, not every machine can be determinized.

In order to adapt these algorithms for any kind of FSM, we need to specify the following particularities of the machine:

- the initial SES X_I and how to build it,

- the Δ function or how to traverse a consuming transition,

- the D function or how to traverse an ε-transition and

- the acceptance SES X_F and how to evaluate if an ES x belongs to it.[9]

It must be taken into account that, since the algorithm is based in the computation of the ε-closure, the execution of this algorithm might fall into an infinite loop for the case of machines allowing for infinite ε-closures; for each kind of machine, we will study such cases in order to avoid them.

7.9.1   From breadth-first to depth-first

The same ESs and derivation functions used for the breadth-first application of a machine can be used for its depth-first application; the difference lies in the order in which the different ESs are computed:

- An arbitrary initial ES x is chosen, then the first successively realizable transitions, starting with the one outgoing from x, are followed until either an acceptor ES is reached after consuming the whole input or until reaching an ES from where no more successively realizable transitions are found.

    - In the former case, the input is to be accepted.
    - In the latter case, the traversed path is to be walked back until the last reached ES having some additional realizable outgoing transition that has not been explored yet, and the same process is to be repeated from that ES and transition and remaining input suffix from that ES.

- If no interpretation starting at x is found, the process repeats for the next unexplored initial ESs until either finding an input interpretation or until no more unexplored initial ESs are left, in which case the input sequence is to be rejected.

As we can see, the depth-first exploration stops at the first interpretation found while the breadth-first approach explores every realizable path starting at an initial ES by consuming some input prefix; the depth-first approach will only explore all those paths for inputs that are to be rejected. However, the exploration of each path is performed independently of the others, thus common subpaths of paths that join at some point will be explored several

---

[9]X_F might be infinite; however, the algorithm only requires to implement predicate $x \in X_F$ rather than constructing X_F.

times. In practice, the depth-first acceptor algorithm has been the fastest one. However, we are not only interested in recognition but in computing every possible translation of a given input sequence, in which case every input interpretation is to be explored. Depth-first translation will be discussed in section 10.6.1.

Algorithm 7.8 *fsm_depth_first_recognize_string* is a possible implementation of the depth-first application of a FSM. This algorithm simply initializes the explorations starting from each initial ES by calling algorithm 7.9 *fsm_depth_first_recognize_suffix*. This latter algorithm recursively performs the search for the first realizable path starting from a given ES x_s and accepting a given input suffix $\sigma_i \ldots \sigma_l$, the first calls taking an initial ES and the whole input. If the suffix is empty and x_s is an acceptor ES, the algorithm simply returns true. If the suffix is not empty, the algorithm calls itself for input suffix $\sigma_{i+1} \ldots \sigma_l$ and for each target ES reachable from x_s by consuming σ_i, until finding the first x_s whose right language (definition 108, p. 143) includes $\sigma_{i+1} \ldots \sigma_l$. If such ES is found, the algorithm returns true. Otherwise, the same process repeats for the ε-reachable ESs from x_s and the same input suffix $\sigma_i \ldots \sigma_l$. If neither here such ES is found, the algorithm finally returns false.

Algorithm 7.8 fsm_depth_first_recognize_string$(\sigma_1 \ldots \sigma_l)$ ▷
$\sigma_1 \ldots \sigma_l \in L$, def. 107

Input: $\sigma_1 \ldots \sigma_l$, an input string of length l
Output: a Boolean indicating whether the input string belongs to L
 1: **for each** $x \in X_I$ **do**
 2:     **if** fsm_depth_first_recognize_suffix$(\sigma_1 \ldots \sigma_l, 1, x)$ **then**
 3:         **return** true
 4:     **end if**
 5: **end for**
 6: **return** false

7.10 Determinization

In general, determinizing a FSM consists in finding an equivalent (definition 93, p. 135) but deterministic FSM (definition 92, p. 135). Determinizing

---

Algorithm 7.9 fsm_depth_first_recognize_suffix$(\sigma_1 \ldots \sigma_l, i, x_s)$ ▷
$\sigma_i \ldots \sigma_l \in L_R(x_s)$, def. 108

Input: $\sigma_1 \ldots \sigma_l$, an input string of length l
$\quad\quad\quad$ i, the index of the first suffix symbol
$\quad\quad\quad$ x_s, an ES with q_s as reached state
Output: a Boolean indicating whether suffix $\sigma_i \ldots \sigma_l$ can be recognized from
$\quad\quad$ x_s or not

1: **if** $i > l \wedge x_s \in X_F$ **then**
2: $\quad\quad$ **return** true
3: **end if**
4: **if** $i \leq l$ **then**
5: $\quad\quad$ **for each** $x_t \in \Delta(\{x_s\}, \sigma_i)$ **do**
6: $\quad\quad\quad$ **if** fsm_depth_first_recognize_suffix$(\sigma_1 \ldots \sigma_l, i + 1, x_t)$ **then**
7: $\quad\quad\quad\quad$ **return** true
8: $\quad\quad\quad$ **end if**
9: $\quad\quad$ **end for**
10: **end if**
11: **for each** $x_t \in D(\{x_s\})$ **do**
12: $\quad\quad$ **if** fsm_depth_first_translate_suffix$(\sigma_1 \ldots \sigma_l, i, x_t)$ **then**
13: $\quad\quad\quad$ **return** true
14: $\quad\quad$ **end if**
15: **end for**
16: **return** false

---

a machine before its application reduces the cost of application of the machine: instead of having to maintain a SES, only a single ES is to be computed for each input symbol. However, the resulting machine may be larger than the original one. Defining a generic determinization algorithm for any kind of FSM is not feasible since the definition of equivalence depends on the purpose of the machine, which may differ from machine to machine; for instance, sequence acceptors and sequence translators do not share the same definition of equivalence (definitions 109 and 163, pp. 143 and 193, respectively). Even for machines having the same purpose, the same determinization algorithm may involve to compute an infinite machine. We will first study FSA's and then give a generic algorithm that computes an equivalent FSA for some acceptor machine. The determinization issues for each kind of machine will be discussed in their respective chapters. In particular, a pseudo-determinization algorithm will be given for cases in which "full" determinization is not possible (e.g.: due to output generation, which will be discussed in section 10.7). Apart from performing a partial determinization, this algorithm removes certain kinds of ε-moves and, hence, avoids the possibility of falling into infinite loops due to ε-cycles.

7.11   Minimization

As for determinization, minimizing a FSM consists in finding an equivalent but minimal FSM (definition 94, p. 135). Once a machine is determinized, minimizing it will not accelerate the machine application —the size of the SESs to compute during the machine application will not be further reduced— but may considerably reduce the size of the machine; hence, minimization may reduce both the time and amount of memory required to load the machine. We will give in section 8.6 (p.174) a minimization algorithm which can be seen as an extension of the determinization algorithm, based on van de Snepscheut's (1985) minimization algorithm. Since grammars are to be minimized only once and then applied several times, we will rather focus on the optimization of the algorithms of application of the machines rather than on the optimization of their determinization and minimization algorithms.

Chapter 8

Finite-state automata

FSAs are equivalent to regular expressions,[1] that is, for any FSA there exists a regular expression representing the same (regular) language and vice-versa (Kleene, 1956). FSAs and regular expressions are or have been used for building lexical analysers as well as for describing search patterns and token sets (Hopcroft et al., 2000, secs. 2.4 & 3.3, pp. 68 & 108; Revuz, 1992; Daciuk et al., 2000; Carrasco and Forcada, 2002; Daciuk et al., 2005); they not only allow for describing finite sets of words, but also some infinite sets of sequences such as integer numbers and email addresses (see figure 8.1). Regular expressions are more convenient for describing simple patterns; the manual construction of FSAs is usually more cumbersome, either when using some graphical interface for drawing them, such as the ones included in the Intex (Silberztein, 2004, chap. 5, p. 49) and Unitex (Paumier, 2008, sec. 5.2, p. 90) systems, or by describing them in some text format, such as with Graphviz's dot format (Gansner and North, 2000) or with the V$\overline{\text{AU}}$C$\underline{\text{ANS}}\overline{\text{ON}}$-G LaTeX package (Lombardy and Sakarovitch, 2002).[2] Additionally, applying a FSA is more straightforward than applying its equivalent regular expression, hence regular expressions are usually transformed into their equivalent FSAs for their application: FSAs are procedural while regular expressions are declarative.

We present here letter FSAs as the simplest case of FSM. We give the basis for the definitions, properties and proofs of the machines presented in the following chapters. More extensive material on letter FSAs and regular

---

[1] A description of regular expressions can be found in Hopcroft et al. (2000, sec. 3, p. 83) or in Sipser (2006, sec. 1.3, p. 63).

[2] Graphviz homepage: http://www.graphviz.org

```
[_A-Za-z0-9-]+(\.[_A-Za-z0-9-]+)*@
          [A-Za-z0-9]+(\.[A-Za-z0-9]+)*\.[A-Za-z][A-Za-z]
```

(a)

(b)

Figure 8.1: (a) Unix regular expression (see Hopcroft et al., 2000, sec. 3.3, p. 108) and (b) FSA matching any email address; additional transitions for the same source an target states have been omitted, leaving a set of stacked labels, and labels of the form 'X-Y' represent any character between X and Y, both included.

expressions can be found in Hopcroft et al. (2000, chap. 2–3) and Sipser (2006, chap. 1).

Definition 128 (FSA). *A FSA $(Q, \Sigma, \delta, Q_I, F)$ is a special type of FSM (definition 46, p. 121) whose set of labels Ξ takes its elements from $\Sigma \cup \{\varepsilon\}$, where Σ is a finite input alphabet and ε is the empty symbol.*

8.1   Transitions

Definition 129 (Consuming transition). *Following definitions 51 and 52, transitions in $Q \times \Sigma \times Q$, that is, which consume and input symbol, are called* pure consuming *transitions or simply* consuming *transitions.*

Definition 130 (ε-transition). *Following definitions 53 and 54, transitions in $Q \times \{\varepsilon\} \times Q$, that is, which do not consume input, are called* pure ε-transitions *or simply* ε-transitions.

8.2   Behaviour

Definition 131 (Execution state). *The ESs of a FSA are states in Q.*

The realization of FSA transitions falls into the FSM general categories of pure consuming transitions and pure ε-transitions (see definitions 87 and 88, pp. 133 and 134, respectively).

Definition 132 (Δ). *The Δ function for FSAs is a simple direct-derivation function on SESs (definition 98, p. 137) with*

- $x_s = q_s,$

- $x_t = q_t,$ *and*

- $d = q_t \in \delta(q_s, \sigma).$

Definition 133 (D). *The D function for FSAs is a simple direct-derivation function on SESs (definition 98, p. 137) with*

- $x_s = q_s,$

- $x_t = q_t,$ *and*

- $d = q_t \in \delta(q_s, \varepsilon).$

Lemma 7 (Finite ε-closure). *Given the iterative definition of ε-closure of lemma 4 (p. 140) adapted for FSAs, there exists a finite number $n \leq |Q|$ so that $V_n = V_{n+1}$. Therefore, the ε-closure computation can be reduced to a finite union of sets*

$$C_\varepsilon(V_0) = V_n \quad such \; that \quad V_{i+1} = V_i \cup D(V_i), \quad i = 0 \ldots n, \qquad (8.1)$$

following lemma 5 (p. 141).

Proof. Let us suppose that $V_{i+1} \neq V_i$ for all $i \geq 0$, that is, there is no finite number n so that $V_n = V_{n+1}$. Since $V_{i+1} = V_i \cup D(V_i)$, it holds that $V_i \subset V_{i+1}$ and that $V_{i+1} - V_i \neq \emptyset$; therefore $|V_0| - |V_1| \geq 1, |V_0| - |V_2| \geq 2, \ldots, |V_0| - |V_\infty| = \infty$. However, since FSAs have a finite number of states and $V_i \subseteq Q$ for $i \geq 0$, V_i must be also finite for $i > 0$. Consequently, there must be a finite number $n \leq |Q|$ such that $V_n = V_{n+1}$, where V_n contains at most every state of Q. $\qquad\square$

Definition 134 (Initial and acceptance SESs). *The initial and acceptance SESs of a FSA are its initial and acceptance sets of states Q_I and F, respectively.*

Definition 135 (Execution machine). *The execution machine of a FSA A is defined as for the generic execution machine (definition 105, p. 142) without any other kind of transitions than pure consuming transitions and pure ε-transitions; since ESs of a FSA are FSA states, the execution machine of A is A itself.*

Definition 136 (L). *Following definition 107 (p. 143), we define L(A), the language accepted by a FSA A, as*

$$L(A) = \{w \in \Sigma^* : \Delta^*(Q_I, w) \cap F \neq \emptyset\}. \tag{8.2}$$

Lemma 8 (Self-concatenation usefulness). *If a cycle p inside a FSA is useful, then p^n is also useful for $n \geq 0$.*

Proof. Given an interpretation $p = p_a p_b p_c$ such that p_b is a cycle, $p_i = p_a p_b^i p_c$ for $i \geq 0$ is an infinite family of interpretations since p_i is a path within the FSA (p_a is connected to p_c and to p_b^i for $i > 0$, and p_b^i is connected to p_c for $i > 0$), every FSA path is realizable and the start and end states of p^i are initial and final, respectively, as for path p. □

As for the case of FSMs, this proof is related to the pumping lemma for regular expressions (see paragraph after proof of theorem 6, p. 164).

Theorem 6 (Cardinality of the interpretation set). *Given theorem 4 (p. 145) and the previous lemma, the number of interpretations within a FSA is infinite iff it contains at least one useful cycle.*

Theorem 7 (Cardinality of the language). *Given lemma 8 and theorem 5 (p. 146), the language of a FSA is infinite iff it contains at least one useful consuming cycle.*

8.3 Reverse FSA

Definition 137 (Reverse FSA). *Let A be a FSA $(Q, \Sigma, \delta, Q_I, F)$, we define A^R, the canonical reverse of A, as the FSA $(Q, \Sigma, \delta', Q_I', F')$ with*

- *$q_t \in \delta'(q_s, \sigma)$ iff $q_s \in \delta(q_t, \sigma)$*

- *$q_t \in \delta'(q_s, \varepsilon)$ iff $q_s \in \delta(q_t, \varepsilon)$*

- *$Q_I' = F$, and*

- $F' = Q_I$.

Our definition of reverse FSA is similar to that in Hopcroft et al. (2000, sec. 4.2, p. 137); as main difference, Hopcroft et al. require to add an additional state to the reverse machine that will serve as initial state. Additionally, a set of ε-transitions outgoing from that state towards the acceptor states of the original machine is to be added in order to simulate multiple initial states. We do not need such additional state and transitions since our definition of FSA is symmetric: we simply allow for multiple initial states as well as for multiple acceptor states, hence we only require to swap the sets of initial and acceptor states of the original machine.

Lemma 9 (Reverse FSA). *Let A be a FSA, A^R is a reverse of A.*

Proof. If $w = \sigma_1 \sigma_2 \ldots \sigma_l \in L(A)$ then there exists at least one finite path

$$p = p_0 \ (q_1, \sigma_1, q_1') \ p_1 \ (q_2, \sigma_2, q_2') \ p_2 \ \ldots \ (q_l, \sigma_l, q_l') \ p_l \tag{8.3}$$

within A that is an interpretation of w, where

- q_j and q_j' are states in Q, for $j = 1 \ldots l$,

- p_0 is a finite and non-empty ε-path having q_1 as end state, or is the empty path and q_1 is the start state of p,

- p_l is a finite and non-empty ε-path having q_l' as start state, or is the empty path and q_l' is the end state of p,

- p_j is a finite and non-empty ε-path having q_j' and q_{j+1} as start and end states, respectively, or is the empty path and $q_j' = q_{j+1}$, for $j = 1 \ldots l - 1$,

- the start state of p belongs to Q_I, and

- the end state of p belongs to F.

Consequently, the finite path

$$p^R = p_l^R \ (q_l', \sigma_l, q_l) \ \ldots \ p_2^R \ (q_2', \sigma_2, q_2) \ p_1^R \ (q_1', \sigma_1, q_1) \ p_0^R \tag{8.4}$$

belongs to A^R, where

- p_l^R is a finite and non-empty ε-path having q_l' as end state, or is the empty path and q_l' is the start state of p^R,

- p_0^R is a finite and non-empty ε-path having q_1 as start state, or is the empty path and q_1 is the end state of p^R,

- p_j^R is a finite and non-empty ε-path having q_{j+1} and q_j' as start and end states, respectively, or is the empty path and $q_{j+1} = q_j'$, for $j = 1 \ldots l - 1$,

- the start state of p^R belongs to Q_I', and

- the end state of p^R belongs to F'.

Therefore, p^R is an interpretation of w^R within A^R and $w^R \in L(A^R)$. If $w \notin L(A)$ then there is no such path p within A, hence neither there is a path p^R within A^R and, consequently, $w^R \notin L(A^R)$. $\qquad\square$

8.4   Recognizing a string

The base breadth-first and depth-first acceptor algorithms 7.5 (p. 153) and 7.8 (p. 157) for FSMs can be straightforwardly adapted for FSAs as explained in section 7.9 (p. 152).

8.5   Determinization of acceptors into FSAs

We present here a generic algorithm for the determinization of any kind of acceptor FSM which tries to compute an equivalent but deterministic FSA, whenever possible. This algorithm is a generalization of the FSA determinization described in Hopcroft et al. (2000, sec. 2.3.5, p. 60, FSAs without ε-moves) or in Sipser (2006, p. 54, FSAs with or without ε-moves), and will be the base of the determinization algorithms for all the machines presented in this dissertation.

Determinization (into FSAs) and application of sequence acceptors are similar problems, though determinization is more complex. Computing an equivalent and deterministic FSA can be viewed as applying a machine not just for a single input but for every input sequence the machine can consume —leading to the exploration of every realizable path within the machine—

and building an equivalent FSA so that for each set of reachable ESs by consuming a given input the new machine defines a unique state; that is, multiple ESs for every possible input sequence the machine can consume are precomputed and replaced by single FSA states. Once the machine is determinized, recognizing an input is reduced to searching for a single target state for each input symbol to consume rather than maintaining several parallel searches. The computation of the ε-closure is no longer required since ε-moves are removed during the determinization process. Application algorithms can be highly simplified for the case of deterministic machines, though we still need the original algorithms for the case of non-determinizable machines.

Theorem 8 (Equivalent deterministic acceptor). *Given a non-deterministic pure acceptor A having*

- Σ *as its input alphabet,*

- X *as domain of its ESs,*

- $X_I \in X$ *as initial SES,*

- $X_F \in X$ *as acceptor SES,*

- $\Delta : \mathcal{P}(X) \times \Sigma \to \mathcal{P}(X)$ *as its consuming transition function on SESs and*

- $C_\varepsilon : \mathcal{P}(X) \to \mathcal{P}(X)$ *as its ε-closure transition function on SESs,*[3]

the following is a description of an equivalent and deterministic finite- or infinite-state automata $A' = (Q', \Sigma, \delta', Q'_I, F')$, depending on whether the execution of A for any input sequence produces finite or infinite sets of ESs and transitions:

- $Q' \subseteq \mathcal{P}(X)$,[4]

- $V_0 = C_\varepsilon(X_I)$,

---

[3] Examples of machines following this description are letter FSAs in this chapter, letter RTNs in chapter 12 and letter FPRTNs in chapter 15.

[4] As stated in Hopcroft et al. (2000, p. 61, 3rd prg.), not every multiple ES in $\mathcal{P}(X)$ may be reachable from $\{X_I\}$, hence Q' does not need to contain every multiple ES in $\mathcal{P}(X)$. Indeed, the determinization algorithm must discard those unreachable states if it is to be used as part of the minimization algorithm *à la* van de Snepscheut (1985).

- $Q_I' = \begin{cases} \{V_0\} & V_0 \neq \emptyset \\ \emptyset & V_0 = \emptyset \end{cases}$

- *if $V_0 \neq \emptyset$ then $V_0 \in Q'$ (by definition of FSM, def. 46, p. 121),*

- *if $V_s \in Q'$ and $V_t = C_\varepsilon(\Delta(V_s, \sigma)) \neq \emptyset$ then $V_t \in Q'$ and $\delta'(V_s, \sigma) = V_t$, and*

- $F' = \{V_f \in Q' : V_f \cap X_F \neq \emptyset\}.$

Proof. Let A and A' be two acceptors as the ones described in the theorem and $w = \sigma_1 \ldots \sigma_l$ be a sequence such that there exists at least one path within A consuming it. The application of A to this sequence yields the following sequence of non-empty SESs:

$$V_0 = C_\varepsilon(X_I) \tag{8.5}$$
$$V_1 = C_\varepsilon(\Delta(V_0, \sigma_1)) \tag{8.6}$$
$$V_2 = C_\varepsilon(\Delta(V_1, \sigma_2)) \tag{8.7}$$
$$\vdots \tag{8.8}$$
$$V_l = C_\varepsilon(\Delta(V_{l-1}, \sigma_l)) \tag{8.9}$$

By construction, A' contains a path

$$p = V_0 \xrightarrow{\sigma_1} V_1 \xrightarrow{\sigma_2} V_2 \ldots \xrightarrow{\sigma_1} V_l. \tag{8.10}$$

If A accepts w then V_l contains at least one ES in X_F and, therefore, $V_l \in F'$. Since V_0 is the initial state of A' and p is composed only by pure consuming transitions, p is realizable and is an interpretation of w within A'. If $w \notin L(A)$ then V_l contains no acceptance ES and, therefore, path p exists within A' but is not an interpretation.

Up to here, we have proved that $w \in L(A)$ implies $w \in L(A')$ and that for every family of paths within A consuming a sequence w there exists a unique path within A' consuming it. Let us suppose that A' accepts some additional sequence not in A. If so, there must exist some interpretation of such sequence within A'. Since paths are added to A' by computing the SESs reachable from V_0, this interpretation must be equal to the concatenation of some sequence of subpaths of some of the added paths that share one or more SES. Let w and w' be two sequences of the form

$$w = \sigma_1 \ldots \sigma_j \sigma_{j+1} \ldots \sigma_l \quad \text{and} \tag{8.11}$$
$$w' = \sigma_1' \ldots \sigma_k' \sigma_{k+1}' \ldots \sigma_m'. \tag{8.12}$$

If both sequences are accepted by A, then they are also accepted by A' and the application of A to these sequences generates the following sequence of SESs:

$$
\begin{aligned}
V_0 &= C_\varepsilon(X_I) & V_0' &= C_\varepsilon(X_I) \\
V_1 &= C_\varepsilon(\Delta(V_0), \sigma_1) & V_1' &= C_\varepsilon(\Delta(V_0'), \sigma_1') \\
&\vdots & &\vdots \\
V_j &= C_\varepsilon(\Delta(V_{j-1}, \sigma_j)) & V_k' &= C_\varepsilon(\Delta(V_{k-1}', \sigma_k')) & (8.13) \\
V_{j+1} &= C_\varepsilon(\Delta(V_j, \sigma_{j+1})) & V_{k+1}' &= C_\varepsilon(\Delta(V_k', \sigma_{k+1}')) \\
&\vdots & &\vdots \\
V_l &= C_\varepsilon(\Delta(V_{l-1}, \sigma_l)) & V_m' &= C_\varepsilon(\Delta(V_{m-1}', \sigma_m')).
\end{aligned}
$$

Let $V_j = V_k'$, then the following sequences of SESs are also possible:

$$
\begin{aligned}
V_0 &= C_\varepsilon(X_I) & V_0' &= C_\varepsilon(X_I) \\
V_1 &= C_\varepsilon(\Delta(V_0), \sigma_1) & V_1' &= C_\varepsilon(\Delta(V_0'), \sigma_1') \\
&\vdots & &\vdots \\
V_j &= C_\varepsilon(\Delta(V_{j-1}, \sigma_j)) & V_k' &= C_\varepsilon(\Delta(V_{k-1}', \sigma_k')) & (8.14) \\
V_{k+1}' &= C_\varepsilon(\Delta(V_j, \sigma_{k+1}')) & V_{j+1} &= C_\varepsilon(\Delta(V_k', \sigma_{j+1})) \\
&\vdots & &\vdots \\
V_m' &= C_\varepsilon(\Delta(V_{m-1}', \sigma_m')) & V_l &= C_\varepsilon(\Delta(V_{l-1}, \sigma_l)).
\end{aligned}
$$

Therefore, sequences

$$\sigma_1 \ldots \sigma_j' \sigma_{k+1}' \ldots \sigma_m' \quad \text{and} \tag{8.15}$$

$$\sigma_1' \ldots \sigma_k' \sigma_{j+1} \ldots \sigma_l \tag{8.16}$$

are also accepted by both A and A'. The same reasoning can be extended for sequences whose interpretations are composed by more than two fragments of subpaths of two or more interpretations.

Since the languages of A and A' are equal, and every path within A consuming a sequence w is condensed into a unique path within A', A' is a deterministic machine equivalent to A. □

Following this description, algorithm 8.1 *fsm_determinize* is a generic determinization algorithm for acceptor machines. The algorithm can be adapted for any kind of acceptor machine by replacing X, X_I, X_F, Δ and C_ε by their particular definitions. The algorithm transforms some kind of

acceptor machine A into a deterministic FSA $A' = (Q', \Sigma, \delta, Q_I, F)$. It builds the different multiple ESs of A and uses algorithm 8.3 *fsm_ create_ state* for creating a single state in A' for each one, keeping a map ζ_m of multiple ESs to single FSA states. The initial steps of the algorithm perform the following operations:

- initialize A' as the empty FSA,

- if X_I is empty, return A' as is or, otherwise, proceed,

- build V_t as the initial SES X_I and E_t as the corresponding queue for ε-closure computation,

- extend V_t with its ε-closure,

- create the initial state r_t of A',

- map V_t to r_t,

- initialize V_m, the set of every computed SES, as $\{V_t\}$, and

- initialize E_m, the queue of unexplored SES corresponding to V_t.

The rest of the algorithm works in a similar fashion than algorithm 7.3 *fsm-_interlaced_ eclosure*: while there are SESs V_s to explore within E_m, dequeue the next one, compute the reachable SESs V_t from V_s and add them to V_m and, if not already present, enqueue them into E_m as well. Step by step, the loop iteration performs the following operations:

- dequeue the next unexplored V_s,

- retrieve r_s, the state of A' corresponding to V_s,

- call algorithm 8.2 *fsm_ recognize_ every_ symbol* in order to build functions ζ_t and ζ_e, the former mapping symbols $\sigma \in \Sigma$ to target SESs V_t such that $V_t = C_\varepsilon(\Delta(V_s, \sigma))$, and the latter mapping the input symbols to the corresponding queues for ε-closure computation,[5]

- for each input symbol that has been mapped to a non-empty SES V_t,

---

[5]In practice we implement a single map returning both the set and queue instead of having to search inside two separate maps for each object.

- retrieve V_t,
- retrieve its corresponding queue,
- extend V_t with its ε-closure,
- add the resulting V_t to V_m and, if not already present, enqueue V_t into E_m as well, create the corresponding state $r_t \in A'$ and map V_t to r_t,
- otherwise, retrieve the state $r_t \in A'$ corresponding to V_t and,
- finally, add transition (r_s, σ, r_t) to A'.

The algorithm is not applicable for cases in which the number of ESs to explored is infinite. It is applicable for any FSA since FSA ESs are states in Q, which is a finite set. The other cases are discussed in their respective sections.

Determinization of lexical acceptors is slightly more complex: let $t = (q_s, \xi, q_t)$ and $t' = (q_s, \xi', q_t')$ be two transitions of an acceptor A such that $\xi \neq \xi'$, for the case of letter acceptors the realization of t and t' is exclusive (the input symbol is either ξ, ξ' or none of them), while for the case of lexical acceptors this is not necessarily true; for instance, let

$$\xi = \text{<V:1>}, \quad \text{and} \tag{8.17}$$
$$\xi' = \text{<V:p>}, \tag{8.18}$$

any verb in both first person and plural will match both masks. Each set of outgoing transitions from the same state must be replaced by a set of equivalent transitions whose realizations are exclusive. For instance, let the former transitions t and t' be the only ones outgoing from q_s and having the former defined lexical masks ξ and ξ', we first compute their corresponding intersection and differences:

$$\xi \wedge \xi' = \text{<V:1p>} \tag{8.19}$$
$$\xi \wedge \neg\xi' = \text{<V:1s>} \tag{8.20}$$
$$\neg\xi \wedge \xi' = \text{<V:2p:3p>} \tag{8.21}$$

Then, we replace t and t' by the following transitions:

$$(q_s, \text{<V:1p>}, q_t) \tag{8.22}$$
$$(q_s, \text{<V:1p>}, q_t') \tag{8.23}$$
$$(q_s, \text{<V:1s>}, q_t) \tag{8.24}$$
$$(q_s, \text{<V:2p:3p>}, q_t) \tag{8.25}$$

Algorithm 8.1 fsm_determinize(A)   ▷ theorem 8

Input: A, an acceptor having Σ as input alphabet, X as ES domain, X_I as initial SES, X_F as acceptor SES, Δ as consuming transition function on SESs and C_ε as ε-closure function on SES,

Output: $A' = (Q', \Sigma, \delta', Q'_I, F')$, a deterministic FSA equivalent to A

 1: initialize A' as the empty FSA on alphabet Σ
 2: **if** $X_I \neq \emptyset$ **then**
 3:     $V_t \leftarrow X_I$
 4:     $E_t \leftarrow X_I$
 5:     $V_t \leftarrow$ fsm_interlaced_eclosure(V_t, E_t)
 6:     $r_t \leftarrow$ fsm_create_state(true, $V_t \cap X_F \neq \emptyset$)
 7:     $\zeta_m(V_t) \leftarrow r_t$
 8:     $V_m \leftarrow \{V_t\}$
 9:     $E_m \leftarrow \{V_t\}$
10:     **while** ($E_m \neq \emptyset$) **do**
11:         $V_s \leftarrow$ dequeue(E_m)
12:         $r_s \leftarrow \zeta_m(V_s)$
13:         $(\zeta_t, \zeta_e) \leftarrow$ fsm_recognize_every_symbol(V_s)
14:         **for each** $\sigma : \zeta_t(\sigma) \notin \{\bot, \emptyset\}$ **do**
15:             $V_t \leftarrow \zeta_t(\sigma)$
16:             $E_t \leftarrow \zeta_e(\sigma)$
17:             $V_t \leftarrow$ fsm_interlaced_eclosure(V_t, E_t)
18:             **if** add($V_m, V_t,$) **then**
19:                 enqueue(E_m, V_t)
20:                 $r_t \leftarrow$ fsm_create_state(false, $V_t \cap X_F$)
21:                 $\zeta_m(V_t) \leftarrow r_t$
22:             **else**
23:                 $r_t \leftarrow \zeta_m(V_t)$
24:             **end if**
25:             $\delta'(r_s, \sigma) \leftarrow \{r_t\}$
26:         **end for**
27:     **end while**
28: **end if**

Algorithm 8.2 fsm_recognize_every_symbol(V_s)

Input: V_s, a source SES
Output: $\zeta_t : \Sigma \to \mathcal{P}(Q)$, a map of input symbols to target SESs such that
  $\zeta_t(\sigma) = \Delta(V_s, \sigma)$
    ζ_e, a map of input symbols to queues of ESs corresponding to the
  SESs of ζ_t for ε-closure computation
1: $\zeta_t \leftarrow \emptyset$
2: $\zeta_e \leftarrow \emptyset$
3: **for each** $(x_t, \sigma) : x_t \in \Delta(V_s, \sigma)$ **do**
4:     add_enqueue_es($\zeta_t(\sigma), \zeta_e(\sigma), x_t$)
5: **end for**

Algorithm 8.3 fsm_create_state(is_initial, is_final)

Input: is_initial, future value of predicate $r \in Q'_I$
      is_final, future value of predicate $r \in F'$
Output: r, the new FSM state
1: $r \leftarrow$ new_state(Q')
2: add(Q', r)
3: **if** is_initial **then**
4:     add(Q'_I, r)
5: **end if**
6: **if** is_final **then**
7:     add(F', r)
8: **end if**

This procedure requires the lexical mask formalism to be closed under the intersection and the difference, which is not the case of the lexical masks formalism of the Intex and Unitex systems or that of chapter 6: the difference of two lexical masks may result in a subset of tokens that cannot be represented by any lexical mask of those formalisms. Blanc (2006, sec. 2.5, p. 29) gives an alternate and closed definition of lexical masks, along with the algorithms for the computation of the intersection and the difference. As we will explain in the next chapter (sec. 10.7, p. 199), we have chosen a simpler determinization method which simply consists in regarding any transition label as a letter, including ε-moves based on mandatory or forbidden blank ε-predicates, hence the procedure applies to both letter and lexical machines. As drawback, such determinization method may not (and usually will not) result in a "totally" deterministic machine due to non-exclusive lexical masks of transitions outgoing from the same state. However, it must be taken into account that some machines cannot be "totally" determinized but, at least, this "partial" determinization will remove certain kinds of ε-moves which may lead to infinite loops when applying the machines. More information on determinization of lexical FSAs and FSAs with predicates, in general, can be found in Blanc (2006, sec. 2.6, p. 37) and van Noord (2000, sec. 2, p. 5), respectively.

8.6   Minimization

Following van de Snepscheut (1985, sec. 3.1, p. 67), minimizing a FSA A can be achieved by performing the following sequence of operations to A: reverse, determinize, reverse and determinize again. In practice, each state structure stores its outgoing transitions as a map of input symbols to target states. The machine is first to be reversed, replacing the maps of outgoing transitions by maps of incoming transitions and swapping the initial and acceptance sets of states. The following determinize and reverse operations can be condensed into a single algorithm; it suffices to perform the following modifications to algorithm 8.1 *fsm_determinize*:

- swap both arguments of calls to *fsm_create_state*, that is, create initial states of A' as acceptor states and vice-versa, and

- replace $\delta'(r_s, \sigma) \in \{r_t\}$ by $\delta'(r_t, \sigma) \in \{r_s\}$, that is, add the transitions reversed.

Finally the unmodified determinization algorithm is to be applied. Note that just applying twice the determinize-reverse operation may not yield a deterministic FSA: the last determinization must be performed to the non-reversed machine in order to make sure that the resulting machine will have a unique initial state (otherwise, the resulting FSA will have a single acceptance state but, possibly, several initial states). The same minimization algorithm applies for the rest of the machines but using the determinization algorithm proposed for each one; therefore, no further details on minimization will be necessary. We will not go into further details since the main subject of this dissertation is the optimization of the algorithms of application of local grammars rather than their minimization algorithms: we only require to minimize a grammar one time before its application, while the algorithms of application are to be executed once for each sentence. We conclude the chapter with an example of minimization *à la* van de Snepscheut of a lexical FSA (figure 8.2).

Figure 8.2: Minimization *à la* van de Snepscheut of a FSA recognizing SMS command requests, regarding lexical masks and ε-predicates as letters; from top to bottom, (a) original FSA, (b) reversed FSA, (c) reversed-determinized-reversed FSA and (d) reversed-determinized-reversed-determinized FSA.

Chapter 9

Tries

Re*trie*val trees (Fredkin, 1960), or tries, are a special kind of FSAs which have been commonly used for the representation of dictionaries or finite sets of words. This particular application, along with alternative data structures, has been described in chapter 4. In this chapter, we will first give a formal definition of trie and then present a new application of this data structure: the optimization of the algorithms of application of FSMs using string-like data.[1] We have given a brief description of this optimization in Sastre and Forcada (2009, sec. 4.1). Experimental results for each applicable algorithm will be given in chapter 20; these results show speedups up to 30%. Greater speedups might be obtained by using ternary search trees (Bentley and Sedgewick, 1997) instead of tries, though describing and implementing this optimization by means of tries is more straightforward. We leave the adaptation of this material for the case of ternary search trees to a future work.

Definition 138 (Trie). *Given a finite set of strings $S \in \Sigma^*$, we define the trie representing S as the FSA $(Q, \Sigma, \delta, q_\varepsilon, F)$ such that*

- Q *contains a state q_α for each prefix α of each string in S,*

- $\delta(q_\alpha, \sigma) = q_{\alpha\sigma}$ *iff $\alpha\sigma$ is a prefix of some string in S, and*

- $F = \{q_\alpha \in Q : \alpha \in S\}$.

---

[1] By "string-like" we mean any data structure consisting of an empty sequence of elements or a non-empty sequence of elements that can be built by appending elements to the empty sequence.

177

Figure 9.1: Trie representing the set of words {'ago', 'at', 'by', 'for', 'in', 'of', 'off', 'on', 'out', 'to'}.

For the sake of simplicity, we define tries with a single initial state q_ε instead of a set of initial states $Q_I = \{q_\varepsilon\}$, an with a transition function returning single states $q_{\alpha\sigma}$ instead of sets of states $\{q_{\alpha\sigma}\}$.

By construction, a trie accepts only the strings in S, is trimmed (definition 118, p. 144), is deterministic (see definition 92, p. 135) and has no cycles since interpretations share only the subpaths consuming their common prefixes. (see figure **??**).

9.1   Optimizing string processing with tries

Most of the algorithms of application of FSMs that we will present throught this dissertation build SESs whose ESs have one or more string-like components, namely partial outputs and stacks of states. During the initialization stage of these algorithms, a first SES V_0 is built containing the ESs in some set X_I. The string-like components of these ESs are empty sequences in all cases. Then, V_0 is extended with its ε-closure and a loop consuming the input symbols starts: for each input symbol, a new SES V_{i+1} is derived from V_i and extended with its ε-closure. In all cases, let β be a string-like component

of an ES x_t that is to be derived from an ES x_s, β is built from the corresponding string-like component α of x_s by either copying α, copying only some prefix of α or copying a prefix of α and then appending some suffix. Once x_t is built, it is to be added to a SES, which implies to compare β with the corresponding string-like components of the ESs already in the SES in order to avoid for duplicates, as explained in chapter 2. The cost of all these operations is proportional to the length of β.

Since there exists a bijective correspondence between trie states and strings (acceptor state q_α accepts string α and no other string), we can represent string-like components α and β as pointers to states q_α and q_β of a trie, respectively; string copies and comparisons will be then reduced to pointer copies and comparisons, taking a constant time (a single clock cycle in most cases). In case β is not a simple copy of α but some suffix is also to be added to or removed from α, we simply follow the pointer towards q_α and traverse or create the trie path corresponding to that suffix in order to retrieve pointer to q_β, hence saving the cost of either copying the unmodified prefix of α or traversing the trie path corresponding to that prefix. Tries accepting only the empty sequence are initially built for each kind of string-like component, and new paths are added to the tries as new suffixes are to be appended to the already accepted sequences. Each string-like component β is built from a previously built string-like component α as follows:

1. if $\beta = \alpha$ then pointer to q_α is simply copied,

2. if $\beta = \alpha\sigma$ then transition $t = (q_\alpha, \sigma, q_\beta)$ is followed in order to retrieve the pointer to q_β, previously adding t and q_β to the trie if not already present (see algorithm 9.1),

---

Algorithm 9.1 concat_trie_string_and_symbol(q_α, σ)

Input: q_α, the trie state corresponding to string α
$\quad\quad\;\;$ σ, a trie input symbol
Output: $q_{\alpha\sigma}$, the trie state corresponding to string $\alpha\sigma$
 1: $q_{\alpha\sigma} \leftarrow \delta(q_\alpha, \sigma)$
 2: **if** $q_{\alpha\sigma} = \bot$ **then**
 3: $\quad\quad q_{\alpha\sigma} \leftarrow$ fsm_create_state(false, true)
 4: $\quad\quad \delta(q_\alpha, \sigma) \leftarrow q_{\alpha\sigma}$
 5: **end if**

Algorithm 9.2 concat_trie_string_and_string(q_α, β)

Input: q_α, the trie state corresponding to string α
$\quad\quad\quad \beta = \sigma_1 \ldots \sigma_l$, the string to concatenate to q_α
Output: $q_{\alpha\beta}$, the trie state corresponding to string $\alpha\beta$
  1: $q_{\alpha\beta} \leftarrow q_\alpha$
  2: $i \leftarrow 1$
  3: **while** $i \neq l + 1 \wedge (q_\alpha \leftarrow \delta(q_{\alpha\beta}, \sigma_i)) \neq \perp$ **do**
  4: $\quad\quad q_{\alpha\beta} \leftarrow q_\alpha$
  5: $\quad\quad i \leftarrow i + 1$
  6: **end while**
  7: **while** $i \neq l + 1$ **do**
  8: $\quad\quad q_\alpha \leftarrow q_{\alpha\beta}$
  9: $\quad\quad q_{\alpha\beta} \leftarrow$ fsm_create_state(false, true)
10: $\quad\quad \delta(q_\alpha, \sigma_i) \leftarrow q_{\alpha\beta}$
11: $\quad\quad i \leftarrow i + 1$
12: **end while**

---

3. if $\beta = \alpha\sigma_1 \ldots \sigma_l$ then we explore the already present trie path

$$q_\alpha \xrightarrow{\sigma_1} q_{\alpha\sigma_1} \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_j} q_{\alpha\sigma_j}, \tag{9.1}$$

then construct path

$$q_{\alpha\sigma_j} \xrightarrow{\alpha\sigma_{j+1}} q_{\alpha\sigma_{j+1}} \xrightarrow{\alpha\sigma_{j+2}} \ldots \xrightarrow{\alpha\sigma_l} q_{\alpha\sigma_l} \tag{9.2}$$

and finally return pointer to $q_{\alpha\sigma_1 \ldots \sigma_l}$ (see algorithm 9.2), and

4. if $\beta\sigma = \alpha$ then pointer to q_β is retrieved by reversely following transition $(q_\beta, \sigma, q_\alpha)$, operation that can be efficiently performed if the data structure representing q_α stores a pointer to q_β.

Obviously, the first and last cases have a constant time. Appending a symbol σ to a string α represented by a trie state q_α mainly requires a binary search within the map of symbols to target states directly reachable from q_α. As seen in chapter 2, this search has a logarithmic cost w.r.t. the number of outgoing transitions from q_α. Appending a string $\sigma_1 \ldots \sigma_l$ requires to perform j binary searches, where j is the greatest natural number such that the trie contains a path corresponding to $\sigma_1 \ldots \sigma_j$, and then $l - j$ additions of a state and a transition to the trie, each one having a constant time.

For the case of dictionary representation, acceptance flags of trie states are used for distinguishing between complete words and mere prefixes; however, this distinction is not needed for the optimization of string management by means of tries: the set of strings corresponding to the whole consumption of the input will be given by the pointers to trie states within the ESs accepting the whole input. Since we have defined tries as FSAs, the concatenation algorithms always set these flags to true, just to give some value. In practice, we simply do not use any acceptance flags.

9.2 Extracting strings from tries

Depending on the algorithm of application of FSMs, some string-like components of the ESs accepting the input are to be given as result, namely the output sequences generated by algorithms of application of FSMs with letter output (to be seen in chapter 11). Other string-like components are only required for the implementation of derivation mechanisms, namely the stacks of return states constructed by some algorithms of application of FSMs with recursive calls (to be seen in chapters 12–14). These components can be simply thrown away along with their respective tries once the algorithm execution ends. In case the sequences to return are to be represented as arrays instead of pointers to trie nodes, a further processing is necessary in order to generate the corresponding trie strings. Of course, if the strings represented by every state of the trie were to be returned, it would be better to use normal strings since the beginning, but this will not be the case: only those sequences corresponding to the consumption of the whole input are to be returned. Rather than generating the language of the trie, the last computed SES is to be traversed in order to search for the acceptor ESs, and the pointers within these ESs representing the strings to extract are to be followed in order to explore only the relevant trie paths. Summarizing, using tries for the representation of sequences will result in a performance gain as long as the number of sequences to extract is small enough w.r.t. the number of sequences represented by the trie.

By definition, every trie state has a unique incoming transition except for q_ε, which has none. Therefore, the symbols of a string α represented by a pointer to a trie state q_α can be retrieved in reverse order by following the pointer and then by reversely following each incoming transition up to state q_ε (see figure **??**, p. 178). Retrieving α instead of α^R is slightly more com-

plex. Algorithm 9.3 *recursive_retrieve_trie_string* implements a possible
solution based on recursivity. The algorithm takes a state q_α and a counter
of traversed incoming transitions i, having 0 as default value, and returns an
array a containing α and a natural number j equal to $|\alpha|$. Counter i can also
be seen as the index of this recursive call, starting with 0. As long as $q_\sigma \neq q_\varepsilon$,
the algorithm calls itself with the source state of the transition incoming to
q_α and $i + 1$ as counter. During the call in which q_ε is reached, i is equal to
$|\alpha|$. At this point, an array a of length i is initialized and returned, along
with the value of i. The array is then filled with the symbols of α in direct
order, one symbol after each return from a recursive call: let σ be the last
symbol of the string represented by the state q_α during recursive call with
index i, σ is to be assigned to $a[j - i - 1]$, the 'mirror' position of i within a.
Summarizing, the algorithm traverses the path from q_ε up to q_α in reverse
order in order to compute the length of α, then initializes array a and fills it
in direct order by taking its steps back.

Algorithm 9.3 recursive_retrieve_trie_string($q_\alpha, i = 0$)

Input: q_α, the trie state whose string α is to be retrieved
        i, the string length counter having 0 as default value
Output: a, an array of input symbols storing α
        j, the final string length
 1: **if** $\exists (q_\beta, \sigma) : \delta(q_\beta, \sigma) = q_\alpha$ **then**
 2:    $(a, j) \leftarrow$ recursive_retrieve_trie_string($q_\beta, i + 1$)
 3:    $a[j - i - 1] \leftarrow \sigma$ ▷ first buffer index is 0
 4: **else**
 5:    $a \leftarrow$ create_array(i)
 6:    $j \leftarrow i$
 7: **end if**

Finally, a simpler solution could be implemented if α's length could be
retrieved by simply following the pointer to q_α; we extend the data structure
representing each state q_α within the trie with a field storing q_α's *depth*, that
is, the length of the path starting at q_ε and ending at q_α. Since tries have only
pure consuming transitions, q_α's depth is equal to $|\alpha|$. Upon the initialization
of a trie, the depth of its initial state is set to zero. Algorithms 9.1 *concat-
_trie_string_and_symbol* and 9.2 *concat_trie_string_and_string* are mod-
ified so that, each time a new state q_β is created with an incoming transition
$\delta(q_\alpha, \sigma)$, q_β's depth is set to q_α's plus one. Constructing an array a containing

a string α can then be done with a single traversal, as illustrated in algorithm 9.4 *retrieve_trie_string*: array a is first initialized with a length equal to the depth of q_α, and then a loop fills the array from the last position up to the first one while reversely following each incoming transition.

Algorithm 9.4 retrieve_trie_string(q_α)

Input: q_α, the trie state whose string is to be retrieved
Output: a, an array storing α
         j, the final string length
1: $a \leftarrow$ create_array(depth(q_α))
2: **for** $i = $ depth(q_α) $- 1$ to 0 step -1 **do** $\triangleright$ first buffer index is 0
3:     $a[i] \leftarrow \sigma : \delta(q_\beta, \sigma) = q_\alpha$
4:     $q_\alpha \leftarrow q_\beta : \delta(q_\beta, \sigma) = q_\alpha$
5: **end for**

9.3 A not-so-efficient concatenation case

As we will see, some of the algorithms of application of FSMs generating letter sequences require an additional concatenation case: appending a trie string β to another trie string α (e.g.: in figure **??**, p. 178, appending q_{to} to q_{in} in order to obtain q_{into}). In this case, knowing q_β's depth will not avoid the hassle of traversing backwards and then forward the path from q_ε to q_β: this path is to be appended to the path corresponding to α in direct order. Algorithm 9.5 *concat_trie_strings* performs this operation, based on algorithm 9.3 *recursive_retrieve_trie_string*: it recursively calls itself with the states before q_β up to reaching q_ε, then either explores or creates a path analogous to the path from q_ε up to q_β in direct order, starting from q_α; after returning from each recursive call, algorithm 9.1 *concat_trie_string_and-_symbol* is called in order to either reach the next state or to create it along with its incoming transition, if not already present. In practice, performance has dropped for the case of algorithms using this kind of concatenation.

Algorithm 9.5 concat_trie_strings(q_α, q_β)

Input: q_α, the trie state corresponding to a string α
           q_β, the trie state corresponding to a string β
Output: $q_{\alpha\beta}$, the trie state corresponding to string $\alpha\beta$
 1: **if** $\exists(q_\gamma, \sigma) : \delta(q_\gamma, \sigma) = q_\beta$ **then**
 2:      $q_{\alpha\beta} \leftarrow$ concat_trie_strings(q_α, q_γ)
 3:      $q_{\alpha\beta} \leftarrow$ concat_trie_string_and_symbol($q_{\alpha\beta}, \sigma$)
 4: **else** $q_{\alpha\beta} \leftarrow q_\alpha$
 5: **end if**

Chapter 10

Finite-state transducers with blackboard output

FSTBOs are a generalization of finite-state transducers where outputs may be any kind of objects (for instance, strings for the case of letter transducers) and transitions may perform any kind of transformation to a current output (transitions of letter transducers may append a symbol to the current output). FSTBOs can be seen as augmented transition networks (Woods, 1969) without recursive calls where register sets —the blackboards— are not only used in order to implement more complex transition functions but are to be given as output. We give here the general definitions for any kind of output, we present in the next chapter FSTs with string output as a particular case of blackboard output, and finally give in chapters 17, 18 and 19 the general guidelines for the definition of weighted, unification and composite output machines as other particular cases of blackboard output. References to other works are given for each particular case in their respective sections.

Definition 139 (FSTBO). *A FSTBO $(Q, \Sigma, \Gamma, B, B_K, \delta, Q_I, F)$ is a special type of FSM (definition 46, p. 121) whose set of labels Ξ takes its elements from the set of input/output pairs $(\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\mathrm{id}_B\})$, where*

- Σ *is a finite input alphabet,*

- ε *is the empty input symbol,*

- $\Gamma : B \to B$ *is a finite output alphabet of functions γ on finite blackboards $b \in B$,*

- id_B *the identity function on blackboards, and*

- $b_\emptyset$, *is the empty blackboard,*

- $B_K \subseteq (B - \{b_\emptyset\})$ *is a (possibly empty) set of illegal or* killing *black-boards.*

Transitions that produce killing blackboards cannot be traversed: during the execution of an algorithm of application of a FSTBO, explorations of paths reaching such blackboards are killed.

10.1 Transitions

Definition 140 (Consuming transition)**.** *Following definition 51 (p. 123), transitions in $Q \times (\Sigma \times (\Gamma \cup \mathrm{id}_B)) \times Q$, that is, which consume an input symbol, are called* consuming *transitions.*

Definition 141 (Generating transition)**.** *Transitions in $Q \times ((\Sigma \cup \{\varepsilon\}) \times \Gamma) \times Q$, that is, having a non-empty output, are called* generating *transitions.*

Definition 142 (Translating transition)**.** *Transitions in $Q \times (\Sigma \times \Gamma) \times Q$, that is, both consuming and generating, are called* translating *or* substituting *transitions.*

Definition 143 (Deleting transition)**.** *Transitions in $Q \times (\Sigma \times \{\mathrm{id}_B\}) \times Q$, that is, consuming transitions that do not generate, are called* deleting *transitions.*

Definition 144 (ε-transition)**.** *Following definition 53 (p. 124), transitions in $Q \times (\{\varepsilon\} \times (\Gamma \cup \{\mathrm{id}_B\})) \times Q$, that is, which do not consume input but may or may not generate output, are called ε-transitions.*

Definition 145 (Inserting transition)**.** *Transitions in $Q \times (\{\varepsilon\} \times \Gamma) \times Q$, that is, generating ε-transitions, are called* inserting *transitions.*

Definition 146 (ε^2-transition)**.** *Transitions in $Q \times \{(\varepsilon, \mathrm{id}_B)\} \times Q$, that is, non-generating ε-transitions, are called ε^2-transitions.*

10.2 Graphical representation

FSTBO transition labels may include an output as well as an input, as has been seen in definition 139 (p. 185). In the classic representation format, the transition label is formed by an input/output pair of codes separated by a colon (see figure 10.1(b)). The representation of non-generating transitions is not modified, that is, the empty output is represented by the absence of the colon and output code rather than by a colon followed by some code representing an empty output.

Unitex and Intex graphs may associate a single output to each box, representing them as text labels with bold fonts (by default) under the corresponding boxes (see figure 10.1(a)). In case a box contains multiple input labels, the same output label is associated to each input label.

10.3 Sequences of transitions

Definition 147 (Generating path). *A generating path is a path containing at least one generating transition.*

Definition 148 (Generating ε-path). *A generating ε-path is a generating path without consuming transitions.*

Definition 149 (ε^2-path). *An ε^2-path is a path whose transitions are all ε^2-transitions.*

Definition 150 (Generating cycle). *A generating cycle is a closed generating path.*

Definition 151 (Generating ε-cycle). *A generating ε-cycle is a closed generating ε-path.*

Definition 152 (ε^2-cycle). *An ε^2-cycle is a closed ε^2-path.*

10.4 Behaviour

Due to blackboard management and the fact that it might be possible to arrive to a state q through different paths that generate different blackboards, ESs for FSTBOs are composed by a state $q \in Q$ plus the blackboard generated from an initial ES up to the ES.

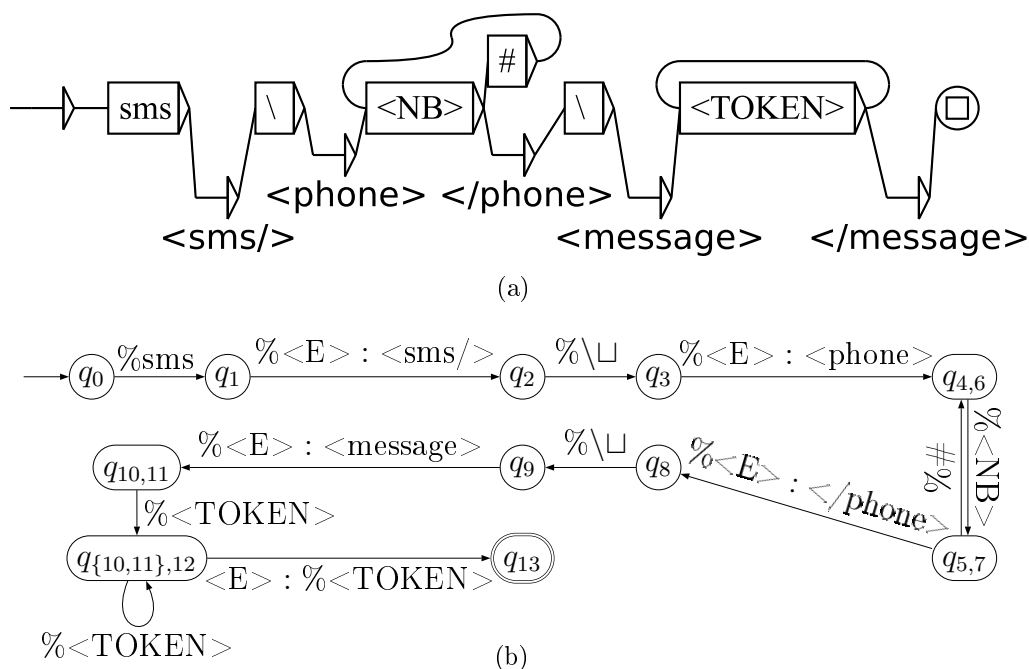Figure 10.1:   (a) Unitex graph marking SMS requests and delimiting their phone and message arguments by inserting XML tags, and (b) equivalent pseudo-minimized lexical FSTSO. These objects are extended versions of the ones in figures 7.1(a) and 7.1(c) (p. 126) Pseudo-minimization is performed as explained in section 8.6 (p. 174) combined with the pseudo-determinization that will be described in section 10.7.

Definition 153 (Execution state)**.** *FSTBO ESs are pairs* $(q, b) \in (Q, B)$.

Definition 154 (Illegal SES)**.** *The illegal SES of a FSTBO* $(Q, \Sigma, \Gamma, B, B_K,$ $\delta, Q_I, F)$ *is* $(Q \times B_K)$, *that is, the set of all ES having a killing blackboard.*

Definition 155 (Δ)**.** *The Δ function for FSTBOs is a simple direct-derivation function on SESs (definition 98, p. 137) with*

- $x_s = (q_s, b_s)$,

- $x_t = (q_t, b_t)$, *and*

- $d = q_t \in \delta(q_s, (\sigma, \gamma)) \wedge b_t = \gamma(b_s) \wedge b_t \notin B_K$.

As we can see, the last condition of predicate d, $b_t \notin B_K$, prevents $\Delta(V, \sigma)$ from returning ESs with killing blackboards.

Definition 156 (D)**.** *The D function for FSTBOs is a simple direct-derivation function on SESs (definition 98, p. 137) with*

- $x_s = (q_s, b_s)$,

- $x_t = (q_t, b_t)$, *and*

- $d = q_t \in \delta(q_s, (\varepsilon, \gamma)) \wedge b_t = \gamma(b_s) \wedge b_t \notin B_K$.

As for $\Delta(V, \sigma)$, the last condition of predicate d prevents $D(V)$ from returning ESs with killing blackboards.

Lemma 10 (Infinite ε-closure)**.** *The ε-closure of a FSTBO SESs V is infinite if there exists an ES x_s within V or ε-reachable from an ES of V such that*

- *there exists an ε-cycle p passing through x_s,*

- *starting from x_s, for every traversal of cycle p output functions always return non-killing blackboards, and*

- *non-identity output functions return a different blackboard at each cycle traversal.*

Proof. Let A be a FSTBO having a generating ε-cycle

$$p = t_0 t_1 \ldots t_{n-1} = (q_0, (\varepsilon, \gamma_0), q_1)(q_1, (\varepsilon, \gamma_1), q_2) \ldots (q_{n-1}, (\varepsilon, \gamma_n), q_n),$$

where $\gamma_j \in (\Gamma \cup \mathrm{id}_B)$ and there is at least one $\gamma_k \in \Gamma$ (non-identity output function) for $j, k = 0 \ldots n - 1$; let $(q_0, b_{0,0}) \in V_0$ a non-illegal ES ($b_{0,0} \notin B_K$) such that

- $(q_j, b_{i,j})$ is a reachable ES through path $p^i t_0 \ldots t_j - 1$, for $i \geq 0$ and $j = 0 \ldots n - 1$, and

- if $\gamma_j \neq \mathrm{id}_B$ then $b_{i,j} = b_{i',j}$ iff $i = i'$, for $i, i' \geq 0$ and $j = 0 \ldots n - 1$, that is, every output function other than the identity produce a new blackboard at each traversal of the cycle.

Following the development of the iterative ε-closure of V_0 (lemma 4, p. 140) adapted for FSTBOs (definition 156 of D function),

$$C_\varepsilon(V_0) = V_n \quad \text{such that} \quad V_{i+1} = V_i \cup \{(q_t, b') : q_t \in \delta(q_s, (\sigma, \gamma)) \wedge$$
$$(q_s, b) \in V_0 \wedge b' = \gamma(b) \wedge b' \notin B_K\}, \quad i = 0 \ldots n, \quad (10.1)$$

where the last condition $b' \notin B_K$ can be omitted since we suppose that only non-killing blackboards are produced,[1] it holds that

$$
\begin{aligned}
(q_0, b_{0,0}) &\in V_0 \\
(q_1, b_{0,1}) &\in V_1 \\
&\vdots \\
(q_{n-1}, b_{0,n-1}) &\in V_{n-1} \\
(q_0, b_{1,0}) &\in V_n \\
(q_1, b_{1,1}) &\in V_{n+1} \\
&\vdots \\
(q_{n-1}, b_{1,n-1}) &\in V_{2n-1} \\
(q_0, b_{2,0}) &\in V_{2n} \\
&\vdots \\
(q_0, b_{i,j}) &\in V_{in+j}
\end{aligned}
$$

Since p is a cycle, when traversing path $p^i t_0 \ldots t_k$ an ES $(q_{k+1}, b_{i,k+1})$ with the same state q_{k+1} is produced for each $i \geq 0$. However, since function γ_k is a non-identity function always returning a different blackboard, every ES $(q_{k+1}, b_{i,k+1})$ is different and therefore the ε-closure is indefinitely incremented with at least one ES per cycle traversal. By definition, if an ES $(q, b_{0,0})$ is ε-reachable from an ES of V_0, then there is a SES V_i that contains the ES. Since the ε-closure of V_0 contains the ε-closure of V_k and the ε-closure of V_k is infinite, so it is the ε-closure of V_0. $\qquad\square$

---

[1]if an ES is reachable, then it is legal and therefore has a non-killing blackboard

Lemma 11 (Finite ε-closure). *Under conditions other than those expressed in the previous lemma, the ε-closure of a FSTBO SES is finite.*

Proof. Let A be a FSTBO and (q_0, b_0) an ES in V_0; let p be any ε-path

$$(q_0, (\varepsilon, \gamma_0), q_1), (q_1(\varepsilon, \gamma_1), q_2) \ldots (q_{n-1}, (\varepsilon, \gamma_{n-1}), q_n), \qquad (10.2)$$

of A, by developing the iterative ε-closure of V_0 we obtain the ESs

$$
\begin{aligned}
(q_0, b_0) &\in V_0 \\
(q_1, \gamma_1(\gamma_0(b_0))) &\in V_1 \\
&\vdots \\
(q_n, \gamma_{n-1}(\ldots(\gamma_1(\gamma_0(b_0)))\ldots)) &\in V_n \\
&\vdots
\end{aligned}
$$

where one of these cases holds:

- After a finite number of transitions have been traversed, a killing blackboard is produced. Since a finite number of ESs have been derived up to this point and no more ESs can be derived once an illegal ES is reached, the ε-closure is finite.

- If $\gamma_j = \mathrm{id}_B$ for $j = 0 \ldots n-1$, then $b_{0,0} = b_{0,1} = \ldots b_{0,n-1} = b_{1,0} = \ldots = b_{i,j}$, that is, every ε-reachable ES from $(q_0, b_{0,0})$ has the same blackboard. Therefore, $C_\varepsilon(V_0)$ is a finite set since it is a subset of $(Q \times \{b_{0,0}\})$, which is finite. In this case the proof is the same than for $FSAs$ (proof of lemma 7, p. 163), which is based in the fact that the result of the ε-closure is a subset Q, which is finite.

- If $\gamma_j \in (\Gamma \cup \{\mathrm{id}_B\})$ but $q_j = q_k$ iff $j = k$, that is, there are no ε-cycles having an ε-reachable ES from q_0, the ε-closure is also finite if the ε-paths are finite, which is true since FSTBOs have a finite number of transitions.

- Finally, if $\gamma_j \in (\Gamma \cup \{\mathrm{id}_B\})$ and $\gamma_j \in \Gamma$ iff $q_j \neq q_k$ for $j \neq k$, that is, there are no generating ε-cycles having an ε-reachable ES from q_0, the ε-closure is also finite since it is the union of the second and third cases, which are also finite: the path can be decomposed into concatenations of ε^2-paths, with cycles or not, and generating ε-paths without cycles, each one adding a finite SES to the ε-closure.

□

Theorem 9. *The ε-closure is always finite for FSTBOs without generating ε-cycles.*

Definition 157 (Initial and acceptance SESs). *Given the sets of initial and acceptance states of a FSTBO, Q_I and F, its initial and acceptance SESs are $(Q_I \times \{b_\emptyset\})$ and $(F \times (B - B_K))$, respectively.*

Definition 158 (Execution machine). *The execution machine of a FSTBO A is defined as for the generic execution machine (definition 105, p. 142) without any other kind of transitions than pure consuming transitions and pure ε-transitions, thus its definition is equal to that of a FSA except for the possibility of having an infinite set of states, transitions and acceptance states.*

Note that the execution machine of a FSTBO A does not require to define transitions with output functions since the resulting output blackboards are coded inside the state labels; for instance, if A contains a transition $(q_s, (\sigma, \gamma), q_t)$ such that q_s is reachable from some initial state by generating blackboard b_s, then $\mathcal{X}(A)$ contains a transition $((q_s, b_s), \sigma, (q_t, \gamma(b_s)))$.

Definition 159 (τ). *We define $\tau(A)$, the language of translations of a FSTBO A, as the set of input/output pairs $(w, b) \in (\Sigma^* \times (B - B_K))$ such that w is recognized and translated into blackboard b by A, that is, an acceptance ES is reached from an initial ES by consuming w and generating blackboard b:*

$$\tau(A) = \{(w, b) : (q_f, b) \in \Delta^*((Q_I \times \{b_\emptyset\}), w) \cap (F \times B)\}. \qquad (10.3)$$

Definition 160 (ω). *We define $\omega(A, w)$, the translations or language of blackboards of a word w for a FSTBO A, as the set of blackboards (SB) $b \in (B - B_K)$ such that (w, b) belongs to the translations of A:*

$$\omega(A, w) = \{b : (w, b) \in \tau(A)\}. \qquad (10.4)$$

Definition 161 (τ_R). *Let x be an ES of a FSM A, we define $\tau_R(x)$, the right translations from x, as*

$$\tau_R(x) = \{(w, b) : x_f \in \Delta^*(\{x\}, w) \cap X_F\}. \qquad (10.5)$$

Definition 162 (ω_R). *Let x be an ES of a FSM A, we define $\omega_R(x, w)$, the right translations of w from x, as*

$$\omega_R(x, w) = \{b : (w, b) \in \tau_R(x)\}. \tag{10.6}$$

Definition 163 (Translator machine). *We say that a machine or an algorithm of application of a machine is a translator iff its purpose is to implement a map $L \to \mathcal{P}(L')$, that is, a map of words of a language L to sets of words of a language L'.*

In a larger sense, we could say that even acceptor machines are also translators: acceptor machines translate input sequences to Booleans (acceptance or rejection). However, generating more complex output than simple Booleans introduces some additional complexities that make worth the distinction; for instance, the possibility of infinite ε-closures and, as we will see in section 10.7, the impossibility of determinizing certain machines.

In definition 93 (p. 135) we introduced the concept of equivalence between machines, and then we formally explained the equivalence between pure acceptor machines (definition 110, p. 143). Once defined what a translator machine is, we can give the last definition of equivalence between machines:

Definition 164 (Equivalent pure translator machines). *We say two pure translator machines A and A' are equivalent iff $\tau(A) = \tau(A')$.*

Other machines than FSAs and FSTBOs that we will present in this dissertation will simply have other kinds of transitions, but will finally be either acceptors defining a language or translators defining a map between two languages.

10.5 Recognized languages

FSTBOs may be designed to express additional restrictions on the input language through the killing-blackboard mechanism in order to go beyond regular languages. Indeed, for every Turing machine there exists an equivalent FSTBO (in the acceptor sense), as for augmented transition networks (Woods, 1969, sec. 1.7.9, p. 39). Following Hopcroft et al. (2000, p. 319), we briefly define Turing machines as follows:

Definition 165 (Turing machine). *A Turing machine is a structure $M = (Q, \Sigma, \Gamma, \delta, q_0, g_\emptyset, F)$ where*

- Q *is a finite set of states,*

- Γ *is a finite tape alphabet,*

- $g_\emptyset \in \Gamma$ *is the blank symbol or default tape symbol,*

- $\Sigma \subseteq \Gamma - \{g_\emptyset\}$ *is a finite input alphabet,*

- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *is a finite transition function, where L and R represent a left or right shift, respectively,*

- $q_0 \in Q$ *is the unique initial state, and*

- $F \subseteq Q$ *is the set of acceptor states.*

ESs are triplets in $(Q, \Gamma^*, \mathbb{N})$: a state, a tape and a head position. The unique ES is $(q_0, \sigma_1 \ldots \sigma_n, 0)$ and the acceptance SES is $(F \times \Gamma^*, \mathbb{N})$.

Informally, a Turing machine is a kind of FSM with a potentially infinite tape and a bidirectional read/write head. Initially, the input is to be contained in a segment of the tape and every other tape cell to contain a special default symbol. Transitions can or cannot be taken depending on the current state as well as the tape symbol at the current head position. Traversing a transition involves to bring the machine to the transition's target state, to overwrite the tape symbol at the current head position and to shift the head position either to the left or to the right. By potentially infinite we mean that the tape head can be shifted in both directions any number of positions, though for a given machine and input only a finite number of shifts must be necessary if we are to apply the machine in practice. Instead of storing an infinite tape, which would be impossible, we initially store a tape having the same length as the input and, each time a symbol is to be read beyond the limits, the tape is first incremented with an extra cell containing the default symbol.

Theorem 10 (FSTBO and Turing machine equivalence)**.** *For every Turing machine there exists a FSTBO recognizing an equivalent language.*

We basically use blackboards to represent the tape and current head position, and output functions to perform the corresponding modifications on the tape and the head position as well as to produce killing blackboards whenever the symbol specified in the transition does not correspond to the

tape symbol at the current head position. We define an initial substructure of the FSTBO for consuming the whole input and loading it into the tape, and another substructure corresponding to the original Turing machine structure where its behaviour is emulated by ε-transitions that operate on the blackboards.

Proof. Let $M = \{Q, \Gamma, g_\emptyset, \Sigma, \delta, q_0, F\}$ be a Turing machine, we build a FSTBO $A = (Q', \Sigma', \Gamma', B, B_K, \delta', Q_I, F')$ as follows:

- $\Sigma' = \Sigma \cup \{\$\}$, the input alphabet of the original Turing machine plus a special symbol marking the end of input (EOI), where every original input $\sigma_1 \sigma_2 \ldots \sigma_{l-1} \sigma_l \in \Sigma^l$ is first to be replaced by $\sigma_l \sigma_{l-1} \ldots \sigma_2 \sigma_1 \$$, that is, the original input in reverse order followed by the EOI mark,

- $Q' = Q \cup \{q_0'\}$, the Turing machine original states plus a FSTBO initial state needed for loading the input into the blackboard's tape,

- $F' = F$,

- $Q_I = \{q_0'\}$

- $B = (\Gamma^* \times \mathbb{N})$, a tape and a head position,

- $B_K = \{\bot\}$, the killing blackboard,

- $\Gamma' = \{\mathrm{id}_B\} \cup \{\gamma_{r,w,s}' : (r, w, s) \in (\Gamma \times \Gamma \times \{L, R\})$, the functions operating on blackboards, where

  - $\gamma_{r,w,s}'(b_s)$ returns $\bot$ if r, the symbol to read, is not equal to the tape symbol at the head position in b_s,

  - otherwise builds b_t, the blackboard to return, by copying b_s, then overwritting b_t's tape symbol at the head position with w and, finally, shifting b_t's head position one cell to the left, if $s = L$, or to the right, if $s = R$,

  - cells containing $g_\emptyset$ are automatically appended to the tape when accessing positions beyond the limits, and

- transitions are defined as follows:

- $q_0' \in \delta'(q_0', (\sigma, \gamma_{b,\sigma,L}'))$, for each $\sigma \in \Sigma$; these transitions load the input in reverse order onto the tape up to the EOI mark, keeping the machine in the initial state q_0',

- $q_0 \in \delta'(q_0, (\$, \gamma_{b,b,R}'))$; this transition detects the EOI mark, positions the head on the last copied symbol and brings the machine to the initial state of the Turing machine to emulate, and

- $q_t \in \delta'(q_s, (\varepsilon, \gamma_{r,w,s}'))$ iff $(q_t, r) \in \delta(q_s, w, s)$; traversing a FSTBO transition is also conditioned by the current tape symbol by the killing-blackboard mechanism.

$\square$

10.6   Translating a string

Based on algorithm 7.5 *fsm_recognize_string* (p. 153) adapted for FSTBOs, algorithm 10.1 *fstbo_translate_string* computes the set of possible translations of a given input string. It uses algorithm 10.2 *fstbo_translate_symbol*, an adaptation of algorithm 7.6 *fsm_recognize_symbol* (p. 154) for FSTBOs, in order to compute the Δ function, and algorithm 10.3 *fstbo_interlaced_eclosure*, an adaptation of algorithm 7.3 *fsm_interlaced_eclosure* (p. 151) for FSTBOs, in order to compute the ε-closure. Finally, algorithm 10.4 *add_enqueue_esbo* is used in the Δ and ε-closure algorithms instead of algorithm 7.4 *add_enqueue_es* (p. 151) in order to add derived ESs with blackboard output;[2] both algorithms perform the same operation but the former checks whether the blackboard is not a killing one before adding the ES. When building the initial SES, the routine *unconditionally_add_enqueue_es* seen in section 7.9 (p. 152) is used instead of an equivalent routine for ESs with blackboard output since, by definition, every initial ES has a non-killing blackboard: $b_\emptyset$.

Apart from the adaptation of the algorithm for FSTBOs and the killing blackboard test, the main difference lies in the post-processing of the last computed SES: rather than looking for the first acceptance ES in order to accept the word, we extract every output associated to any acceptance ES in order to build the set of translations of the input word. Since the SESs V_i cannot contain ESs having a killing blackboard, it is only necessary to check

---

[2]In *add_enqueue_esbo*, 'esbo' stands for ES with blackboard output.

whether their state q is accepting or not. If no acceptance ES is found then an empty set of translations is returned. The domain of application of the translator algorithm must be reduced to FSTBOs not having generating ε-cycles in order to ensure that the algorithm execution will finish. For the case of NLP this is not an issue since generating ε-cycles would lead to infinite sets of interpretations of natural language sentences, which makes no sense.

Algorithm 10.1 fstbo_translate_string$(\sigma_1 \ldots \sigma_l) \triangleright \omega(A, \sigma_1 \ldots \sigma_l)$, def. 160

Input: $\sigma_1 \ldots \sigma_l$, an input string of length l
Output: T, the translations of $\sigma_1 \ldots \sigma_l$
 1: $V \leftarrow \emptyset$
 2: $E \leftarrow \emptyset$
 3: **for each** $q \in Q_I$ **do**
 4:     unconditionally_add_enqueue_es$(V, E, (q, b_\emptyset))$
 5: **end for**
 6: fstbo_interlaced_eclosure(V, E)
 7: $i \leftarrow 0$
 8: **while** $V \neq \emptyset \wedge i < l$ **do**
 9:     $V \leftarrow$ fstbo_recognize_symbol(V, E, σ_{i+1})
10:     $i \leftarrow i + 1$
11:     fstbo_interlaced_eclosure(V, E)
12: **end while**
13: $T \leftarrow \emptyset$
14: **for each** $(q, b) \in V : q \in F$ **do**
15:     add(T, b)
16: **end for**

The algorithm can be further improved by using the trie string management shown in section 9.1 (p. 178) for the representation of string-like structures of output blackboards; FSTSOs are the simplest applicable case since their blackboards are strings (see section 11.5, p. 215).

10.6.1   From breadth-first to depth-first

Algorithm 10.5 *fstbo_depth_first_translate_string* (along with algorithm 10.6 *fstbo_depth_first_translate_suffix*) is another algorithm computing the translations of an input sequence for a given FSTBO but performing a depth-first traversal of the machine instead of a breadth-first one. We simply modify

---

Algorithm 10.2 fstbo_translate_symbol(V, E, σ) $\triangleright \Delta(V, \sigma)$, def. (155)

Input: V, a SES

         E, the empty queue of unexplored ESs

         σ, the input symbol to translate

Output: W, the set of reachable ESs from V by consuming σ

         E after enqueuing the ESs of W

 1: $W \leftarrow \emptyset$
 2: **for each** $(q_s, b_s) \in V$ **do**
 3:     **for each** $(q_t, \gamma) : q_t \in \delta(q_s, (\sigma, \gamma))$ **do**
 4:        add_enqueue_esbo$(W, E, (q_t, \gamma(b_t)))$
 5:     **end for**
 6: **end for**

Algorithm 10.3 fstbo_interlaced_eclosure(V, E) $\triangleright C_\varepsilon(V)$

Input: V, the SES whose ε-closure is to be computed

         E, the queue of unexplored ESs containing every ES in V

Output: V after computing its ε-closure

         E after emptying it

 1: **while** $E \neq \emptyset$ **do**
 2:     $(q_s, b_s) \leftarrow$ dequeue(E)
 3:     **for each** $(q_t, \gamma) : q_t \in \delta(q_s, (\varepsilon, \gamma))$ **do**
 4:        add_enqueue_esbo$(V, E, (q_t, \gamma(b_s)))$
 5:     **end for**
 6: **end while**

Algorithm 10.4 add_enqueue_esbo(V, E, x_t)

Input: V, the SES where the ES is added

         E, the queue of unexplored ESs

         x_t, the ES to add to V

Output: V after adding the ES, if legal

         E after enqueuing the ES, if new and legal

 1: **if** blackboard$(x_t) \notin B_K$ **then**
 2:     **if** add(V, x_t) **then**
 3:        enqueue(E, x_t)
 4:     **end if**
 5: **end if**

---

the generic depth-first recognizer algorithms (section 7.9.1, p. 156) so that they do not stop after reaching the first acceptor ES by consuming the whole input; instead, the algorithm is to continue until reaching all those acceptor ESs and to return a set of translations composed by all of their corresponding blackboards. Algorithms 7.8 *fsm_ depth_ first_ recognize_ string* and 7.9 *fsm_ depth_ first_ recognize_ suffix* are to be modified as follows:

- a set of translations T is to be returned instead of a Boolean value,

- when detecting an acceptor ES, its blackboard is to be added to T instead of returning true,

- calls to algorithm 10.6 *fstbo_ depth_ first_ translate_ suffix* are to be performed without evaluating the returned value and without returning any value, and

- the instruction returning false is to be removed (T is implicitly returned since it is an input/output variable).

Algorithm 10.5 fstbo_depth_first_translate_string$(\sigma_1 \ldots \sigma_l)$ ▷ $\omega(A, \sigma_1 \ldots \sigma_l)$, def. 160

Input: $\sigma_1 \ldots \sigma_l$, an input string of length l
Output: T, the translations of $\sigma_1 \ldots \sigma_l$
 1: **for each** $x \in X_I$ **do**
 2:     fstbo_depth_first_translate_suffix$(\sigma_1 \ldots \sigma_l, 1, x, T)$
 3: **end for**

10.7 Determinization

Deterministic transducers are commonly known as *sequential* transducers; following Mohri (1997, sec 2.1), we define sequential FSTBOs as follows:

Definition 166 (Sequential FSTBO). *Let A be a FSTBO $(Q, \Sigma, \Gamma, B, B_K,$ $\delta, Q_I, F)$, we say A is sequential iff it has deterministic input, that is, let A' be a FSA $(Q, \Sigma, \delta', Q_I, F)$ such that*

$$q_t \in \delta'(q_s, \sigma) \quad \Longleftrightarrow \quad q_t \in \delta(q_s, (\sigma, \gamma)) \quad and \tag{10.7}$$
$$q_t \in \delta'(q_s, \varepsilon) \quad \Longleftrightarrow \quad q_t \in \delta(q_s, (\varepsilon, \gamma)), \tag{10.8}$$

Algorithm 10.6 fstbo_depth_first_translate_suffix$(\sigma_1 \ldots \sigma_l, i, (q_s, b_s), T)$
$\triangleright \omega_R(x_s, \sigma_i \ldots \sigma_l)$, def. 162

Input: $\sigma_1 \ldots \sigma_l$, an input string of length l
        i, the index of the first suffix symbol
        b_s, an ES with q_s as reached state
        T, a set of translations
Output: T after adding the right translations of $\sigma_i \ldots \sigma_l$ from x_s
 1: **if** $i > l \wedge (q_s, b_s) \in X_F$ **then**
 2:    add(T, b_s)
 3: **end if**
 4: **if** $i \leq l$ **then**
 5:    **for each** $x_t \in \Delta(\{(q_s, b_s)\}, \sigma_i)$ **do**
 6:        fstbo_depth_first_translate_suffix$(\sigma_1 \ldots \sigma_l, i + 1, x_t, T)$
 7:    **end for**
 8: **end if**
 9: **for each** $x_t \in D(\{x_s\})$ **do**
10:    fstbo_depth_first_translate_suffix$(\sigma_1 \ldots \sigma_l, i, x_t)$
11: **end for**

A' *is deterministic.*

In general, FSTBOs representing natural language grammars are not determinizable due to their ambiguity.

Corollary 6. *Let A and A' be the machines of the previous definition; then,*

$$\forall w \in \Sigma^*[|\omega(A, w)| \leq 1], \tag{10.9}$$

since A' is deterministic and, hence, it may contain no more than one interpretation of w.

Corollary 7. *Let A be a non-sequential FSTBO, if $|\omega(A, w)| > 1$ for some input sequence w then there exists no sequential FSTBO equivalent to A.*

Note that a FSTBO A may have two different interpretations for the same input sequence w, yet associate a single output to w; for instance, if A maps input sequences to scores by adding some amount of points at each transition, different paths may generate the same score by adding the same points in different order.

Sequential transducers can be generalized by introducing the possibility of generating at most one additional output right after accepting an input sequence, where the output is given by a map of acceptor states to additional outputs (Schützenberger, 1977). Such transducers are called *subsequential*. Choffrut (1977, 1978) characterized the class of transducers performing subsequential transductions, hence being determinizable. Such characterization implicitly defines an algorithm for the construction of an equivalent subsequential transducer. This algorithm has been explicitly given by several authors (Berstel, 1979; Mohri, 1996; Roche and Schabes, 1997). Mohri (1994a) extended the definition of subsequential transducers to p-subsequential transducers, transducers associating up to p additional outputs to each acceptor state, in order to allow for a 'quasi-determinization' of FSTBOs representing a special class of ambiguous languages.

Definition 167 (p-subsequential FSTBO)**.** *A p-subsequential FSTBO is a structure $(Q, \Sigma, \Gamma, B, B_K, \delta, \{q_I\}, F, \rho)$ where $(Q, \Sigma, \Gamma, B, B_K, \delta, \{q_I\}, F)$ is a sequential FSTBO and*

$$\rho : F \to \mathcal{P}(\Gamma) \tag{10.10}$$

is a function mapping acceptor states to sets of up to p additional output functions.

Corollary 8 (Mohri, 1994b, sec. 4)**.** *A p-subsequential FSTBO $(Q, \Sigma, \Gamma, B, B_K, \delta, Q_I, F, \rho)$ can be seen as a FSTBO $(Q \cup \{q_\$\}, \Sigma \cup \{\$\}, \Gamma, B, B_K, \delta', Q_I, \{q_\$\})$ where*

- *input symbol $ explicitly represents the end of input,*

- *$q_\$$ is an additional state and the only acceptor state, and*

- *δ' defines the same transitions than δ plus an additional transition $(q_f, (\$, \gamma), q_\$)$, for each (q_f, γ) such that $q_f \in F$ and $\gamma \in \rho(q_f)$.*

Though not deterministic in the strict sense, p-subsequential transducers (with $p > 1$) can be applied to an input sequence as deterministic machines until reaching an acceptor state: only a single ES must be computed for each input symbol; once the whole input is consumed, if an acceptor ES is reached then the set of translations is built by combining the output of the ES with the outputs mapped to the acceptor state of the ES.

Definition 168 (τ). *We define $\tau(A)$, the language of translations of a p-subsequential FSTBO A, as*

$$\tau(A) = \{(w, b') : (q_f, b) \in \Delta^*((Q_I \times \{b_\emptyset\}), w) \cap (F \times B) \wedge$$
$$b' = \gamma(b) \wedge b' \notin B_K \wedge \gamma \in \rho(q_f)\}. \quad (10.11)$$

Algorithms for the construction of p-subsequential transducers equivalent to string-to-string, string-to-weight and string-to-string-and-weight transducers,[3] have also been given by Mohri (1996, 1997). These algorithms are similar to the common determinization algorithm: they join together transitions sharing the same source state and label, and then join as well the corresponding target states. In order to join transitions consuming the same input but performing different output transformations, the transformations are totally or partially delayed to the subsequent transitions. If not totally delayed, the non-delayed partial transformations must be equal so that the transitions can be joined; for instance, let γ and γ' be two output transformations appending strings $\alpha\beta$ and $\alpha\beta'$, respectively, only the generation of suffixes β and β' is to be delayed. Target states are coupled with the corresponding delayed transformations. When taking these couples as source states for joining their corresponding outgoing transitions, the delayed transformations are added to the transition output labels. Delayed transformations of acceptor states will be the additional transformations to perform once the whole input has been consumed. Summarizing, no output transformation is generated until enough input symbols are observed in order to make sure that the right transformation is performed.

Figure 10.2 illustrates a string-to-string transducer along with its equivalent subsequential transducer.[4] Output labels α simply indicate that string α is to be appended to the current output. Under each state of the sequential transducer, the corresponding set of couples state/delayed output have been included. The initial state r_0 corresponds to couple (q_0, ε), that is, to have reached state q_0 with no delayed output. Transitions $(q_0, (a, b), q_1)$ and $(q_0, (a, a), q_2)$ are joined by delaying the generation of a and b. Reaching state r_1 is equivalent to have reached state q_1 with delayed output b or state q_2 with delayed output a. Transitions $(q_1, (b, b), q_0)$, $(q_2, (a, a), q_2)$ and $(q_2, (a, b), q_0)$

---

[3]Transducers implementing maps of strings to either strings, weights, or both string and weights, respectively, where weights represent scores or probabilities; weighted machines will be the object of chapter 18.

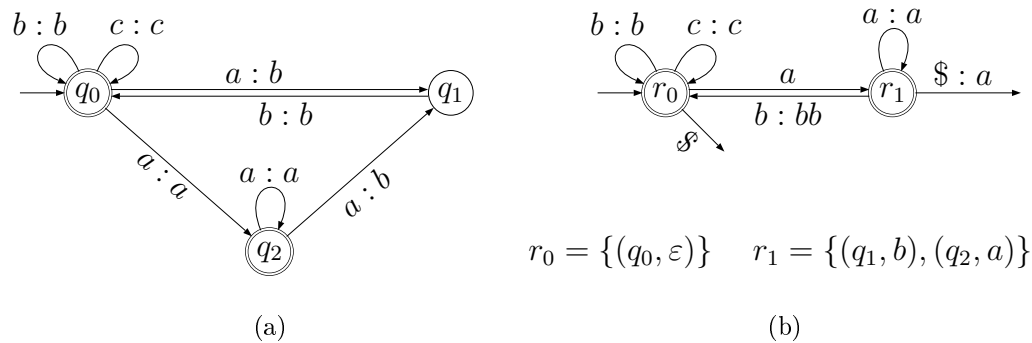[4]Example extracted from (Mohri, 1996).

Figure 10.2: (a) Non-deterministic string-to-string transducer and (b) equivalent subsequential transducer; transitions consuming \$, the end-of-input, represent the additional outputs to generate after accepting the input.

receive these delayed outputs, becoming $(q_1, (b, bb), q_0)$, $(q_2, (a, aa), q_2)$ and $(q_2, (a, ab), q_0)$. Then, the two latter transitions are joined by delaying only the generation of the second symbol, which results in reaching exactly the same set of couples $\{(q_1, b), (q_2, a)\}$. The other transitions have unique inputs, thus their outputs do not need to be delayed and hence reach the set of couples $\{(q_0, \varepsilon)\}$. Acceptor states of the sequential transducer are those having at least one couple (q_f, α) with $q_f \in F$, and the additional outputs to generate are those delayed outputs α: nothing for r_0 and a for r_1.

Figure 10.3 illustrates a string-to-string transducer along with its equivalent ∞-subsequential transducer.[5] Outputs of transitions $(q_0, (a, x), q_0)$ and $(q_0, (a, x), q_0)$ are totally delayed at each step, resulting in different target states r_i with 2^i different delayed outputs.

An alternative to quasi-determinization is *lazy* or on-the-fly determinization (Mohri et al., 2002; Jussila et al., 2005), which consists in determinizing the explored paths of the machine during its application; supposing that a grammar is always applied to the same subset of input sequences, only the corresponding paths will be determinized yet keeping a finite machine since the input sequences are finite. When applying the machine for the first time, the cost of determinizing the corresponding substructures will be added to the cost of applying the machine as if it was deterministic. Successive applications will take advantage of the already determinized substructures, saving the determinization cost. However, the machine may grow in size excessively

---

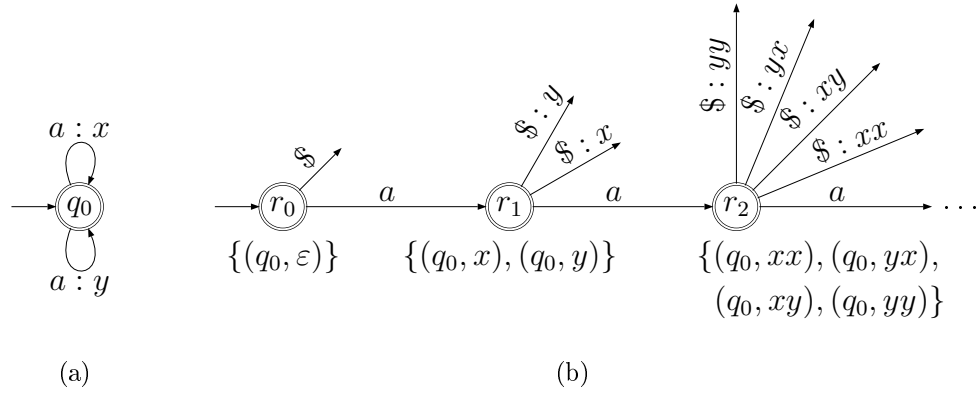[5]Example extracted from (Blanc, 2006, p. 69).

Figure 10.3: (a) Non-deterministic string-to-string transducer and (b) equivalent ∞-subsequential transducer. Transitions consuming \$, the end-of-input, represent the additional outputs to generate after accepting the input.

once applied to a certain amount of input sequences. Lazy determinization is the solution adopted by the Outilex system (Blanc, 2006, sec. 2.8.4, p. 68).

The solution we present here is the one used by the Unitex and Apertium systems (Garrido-Alenda et al., 2002), which consists in determinizing the machine's underlying FSA instead of the machine itself.[6]

Definition 169 (Underlying FSA). *Let* $A = (Q, \Sigma, \Gamma, B, B_K, \delta, Q_I, F)$ *be a FSTBO, we define its underlying FSA as* $(Q, ((\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})) - \{(\varepsilon, \varepsilon)\}, \delta, Q_I, F)$ *with* $(\varepsilon, \varepsilon)$ *as the empty symbol; in other words, FSTBO input/output pairs become FSA input symbols except for* $(\varepsilon, \varepsilon)$ *which becomes the empty symbol.*

This process may only perform a partial determinization of the FSTBO: let $(q_s, (\sigma, \gamma), q_t)$ and $(q_s, (\sigma, \gamma'), q_t)$ be two transitions of a FSTBO, these transitions will not be joined together since (σ, γ) and (σ, γ') will be interpreted as two different input symbols. However, it is sure that, for every FSTBO, this procedure will end up with a finite machine since FSA determinization always ends up with a deterministic FSA. One important advantage of FSA

---

[6]This kind of determinization is performed in Unitex whenever compiling a graph (see Paumier, 2008, sec. 6.2, p. 105), though this is not mentioned in the manual. In Garrido-Alenda et al. (2002), this determinization procedure is mentioned in the context of the interNOSTRUM machine translator; Apertium is another machine translator that has evolved from interNOSTRUM and which has inherited this feature.

determinization is that ε-moves are removed, avoiding the need for ε-closure computation during further FSA applications. Determinizing a FSTBO as its underlying FSA will not remove every ε-move, since generating transitions are treated as consuming transitions, but will at least remove every FSTBO ε^2-transition.

10.8 Minimization

Mohri (1994b) also defined an algorithm for the minimization of transducers by constructing equivalent p-subsequential transducers. As for the case of determinization, the same problem remains: for some transducers, p is infinite. Obviously, if we treat the FSTBO as its underlying FSA then minimization à la van de Snepscheut (section 8.6, p. 174) can be normally performed.

10.9 Blackboard set processing

The set of explored paths during the recognition of an input sequence for a FSTBO without killing blackboards depends uniquely on the input symbols to consume and not on the generated blackboards. By defining killing blackboards, the set of explored paths may be reduced but not extended. However, since ESs contain the blackboard generated up to reaching the FSTBO state q, multiple ESs $x_i = (q, b_i)$ are possible for the same FSTBO state q; moreover, a path p starting at q will allow for multiple execution paths, each one starting at an x_i. Therefore, algorithm 10.1 *fstbo_translate_string* (p. 197) may perform several explorations of p, while it is possible to explore p a single time in order to build the set of blackboards (SBs) it generates. We extend the FSA processing for FSTBO blackboard set processing (BSP) by constructing a function ζ_{B_i} for each FSA SES V_i that maps FSA ESs — which are in fact simple FSA states— to SBs, rather than storing generated blackboards within each ES.

Definition 170 (Z_B). *Given a FSTBO $(Q, \Sigma, \Gamma, B, B_K, \delta, Q_I, F)$, we define Z_B as the set of every partial map ζ_B of FSA ESs in Q to SBs in $\mathcal{P}(B)$.*

Definition 171 (BSP SES). *We define the equivalent BSP SES V_B of a FSTBO SES V as a pair (V', ζ_B) where $V' \subseteq Q$ is a FSA SES —a set of FSA states— and $\zeta_B \in Z_B$ is a function mapping states to SBs such that*

$$V_B = (V', \zeta_B) : V' = \{q : (q, b) \in V\} \wedge \zeta_B(q) = \{b : (q, b) \in V\}, \quad (10.12)$$

which is equivalent to say that

$$V_B = (V', \zeta_B) : \bigcup_{q \in V'} \{q\} \times \zeta_B(q) = V. \qquad (10.13)$$

In BSP, performing a derivation from a state q_s due to a consuming transition $\delta(q_s, (\sigma, \gamma)) \to q_t$, or due to an ε-transition $\delta(q_s, (\varepsilon, \gamma)) \to q_t$, the γ function is to be applied to every blackboard in $\zeta_B(q_s)$.

Definition 172 (γ on SBs). *Given a function γ on blackboards, we extend the definition of γ for SBs as follows:*

$$\gamma : \mathcal{P}(B) \to \mathcal{P}(B)$$

$$\gamma(B_s) = \{b_t : b_s \in B_s \wedge b_t = \gamma(b_s) \wedge b_t \notin B_K\} \qquad (10.14)$$

Note that the application of γ discards killing blackboards; hence γ may return an empty SB, in which case there is no ES to be derived.

Definition 173 (BSP Δ). *We redefine the FSTBO Δ function for BSP SESs as follows:*

$$\Delta : (\mathcal{P}(Q) \times Z_B) \times \Sigma \to (\mathcal{P}(Q) \times Z_B),$$

such that

$$\Delta((V, \zeta_B), \sigma) = (V', \zeta_B') : V' = \{q_t : \zeta_B'(q_t) \neq \emptyset\} \wedge$$
$$\zeta_B'(q_t) = \bigcup_{\gamma : q_t \in \delta(q_s, (\sigma, \gamma)) \wedge q_s \in V} \gamma(\zeta_B(q_s)) \quad (10.15)$$

The computation of Δ traverses every path of length 1 having a state of V_i as source state and consuming σ_{i+1} in order to build $(V_{i+1}, \zeta_{B_{i+1}})$ from (V_i, ζ_{B_i}). However, the computation of the ε-closure traverses every ε-path of any length having any state of V_i as start state, which allows for different derivation paths to share subpaths. These ε-paths can be explored without repeating the traversal of shared subpaths by following a topological sort (definition 81, p. 130) of the corresponding ε-closure-substructure (definition 106, p. 143). However, only acyclic substructures can be topologically sorted (lemma 1, p. 131). Let A be a FSTBO and A' be the FSA equal to A after removing its output alphabet and transition outputs, cycles in the ε-closure-substructures of $\mathcal{X}(A')$ come from cycles in A, which can be of two

forms: generating ε-cycles and non-generating ε-cycles (ε^2-cycles); however, the former must be forbidden in order to avoid infinite ε-closures, and the latter can be removed by determinizing A regarding it as its underlying FSA (definition 169, p. 204). Forbidding generating ε-cycles does not reduce the capability of the formalism for the representation of natural language grammars, since they allow for generating an infinite output from a finite input (e.g. an infinite parse tree for a given sentence), which makes no sense.

Theorem 11 (ε^2-cycle removal). *For every FSTBO with ε^2-cycles there exists an equivalent FSTBO without ε^2-cycles which can be obtained by determinizing the underlying FSA (definition 169, p. 204).*

Theorem 12 (Existence of a topological sort). *Considering lemma 1 (p. 131) and theorem 11, for every FSTBO without generating ε-cycles there exists at least one equivalent FSTBO A such that, given A' the FSA obtained from A after removing its output alphabet and transition outputs, there exists at least one topological sort for every ε-closure-substructure (definition 106, p. 143) of $\mathcal{X}(A')$.*

Recall that the execution machine of a FSA is the FSA itself (definition 135, p. 164); therefore, $\mathcal{X}(A') = A'$ since A' is a FSA.

The definition of D for BSP is almost the same than the previous definition of Δ for BSP; for the case of D, no input symbol is to be consumed, ε-transitions are considered instead of consuming ones, and a BSP SES is derived from a single source state and SB instead of from a BSP SES:

Definition 174 (BSP D). *We redefine the FSTBO D function for BSP as follows:*

$$D : Q \times B \to (\mathcal{P}(Q) \times Z_B)$$

$$D(q_s, B_s) = (V', \zeta'_B) : V' = \{q_t : \zeta'_B(q_t) \neq \emptyset\} \wedge$$
$$\zeta'_B(q_t) = \bigcup_{\gamma : q_t \in \delta(q_s, (\varepsilon, \gamma))} \gamma(B_s), \quad (10.16)$$

that is, $D(q_s, B_s)$ returns a pair (V', ζ'_B) where ζ'_B is a function mapping each ε-reachable state q_t from q_s to the set of blackboards

For the case of BSP, we iteratively compute the ε-closure of a BSP SES (V_0, ζ_{B_0}) by computing at each iteration the ε-reachable states q_t from a

unique source state q_i, and by increasing $\zeta_B(q_t)$ with every blackboard generated by ε-reaching q_t. The state to be taken as q_s for each iteration is given by a topological sort of the corresponding $C_\varepsilon(V_0)$-substructure.

Definition 175 (BSP ε-closure). *Given a BSP SES (V_0, ζ_{B_0}) of a FSTBO A, A' the FSA equal to A after removing its output alphabet and transition outputs, and $x_0, \ldots, x_n$ a topological sort of the $C_\varepsilon(V_0)$-substructure of $\mathcal{X}(A)$, we redefine the ε-closure for BSP as follows:*

$$C_\varepsilon(V_0, \zeta_{B_0}) = (V_n, \zeta_{B_n}) : (V'_{i+1}, \zeta'_{B_{i+1}}) = D(x_i, \zeta_{B_i}(x_i)) \wedge V_{i+1} = V_i \cup V'_{i+1} \wedge$$
$$\forall x_t \in V_{i+1}[\zeta_{B_{i+1}}(x_t) = \zeta_{B_i}(x_t) \cup \zeta'_{B_{i+1}}(x_t)], \quad i = 0 \ldots n-1. \quad (10.17)$$

Note that ESs x_i and x_t of A' are in fact FSA states since A' is a FSA. In the equation,

- V'_{i+1} is the set of ε-reachable FSA states x_t from x_i,

- V_{i+1} accumulates every ε-reached FSA state from V_0 up to iteration $i+1$ (the union of V_0 with $V'_1, V'_2, \ldots, V'_{i+1}$),

- $\zeta_{B_{i+1}}$ maps each state $x_t \in V_{i+1}$ to the SB generated by ε-reaching x_t from any state $x_s \in V_0$, where states x_s have been reached prior to the computation of the ε-closure by generating SB $\zeta_{B_0}(x_s)$, and

- $\zeta'_{B_{i+1}}$ maps the states $x_t \in V'_{i+1})$ to the blackboards that have been generated by directly ε-reaching x_t from x_i after having reached x_i by generating SBs $\zeta_{B_i}(x_i)$.

Theorem 13 (BSP ε-closure equivalence). *Let V be a non-BSP SES of a FSTBO A such that there exists a topological sort of the $C_\varepsilon(V)$-substructure of A, and let V_B be a BSP SES of A, then the equivalence of V and V_B implies the equivalence of $C_\varepsilon(V)$ and $C_\varepsilon(V_B)$.*

Proof. Let V be a non-BSP SES of a FSTBO, $V_B = (V_0, \zeta_{B_0})$ its equivalent BSP SES, (Q', δ') the $C_\varepsilon(V)$-substructure and $q_0, \ldots, q_n$ a topological sort of (Q', δ'). By definition of ε-closure-substructure, V contains at least every state in Q' that is unreachable from any other state in Q' through any path within (Q', δ'), and therefore so it does V_0: if q is such an unreachable state and it does not belong to V, then it cannot be derived during the computation of the ε-closure and therefore cannot belong to Q'. By definition of

topological sort, q_0 is such an unreachable state and therefore belongs to V_0. V_1 contains every state in V_0 plus every ε-reachable state from q_0 by generating at least one non-killing blackboard. If q_1 is one of the unreachable states then it belongs to V_0 and therefore to V_1 as well; otherwise q_1 must be ε-reachable from q_0 since, by definition of topological sort, it cannot be ε-reached from any q_i with $i > 1$. Following the same reasoning for V_i and V_{i+1} with $i = 1 \ldots n-1$, we deduce $q_{i+1} \in V_{i+1}$ and therefore $V_n = Q'$.

Function ζ_{B_0} maps every state in V_0 to a SB so that the equivalence is kept w.r.t. V. Since $q_0 \in V_0$, it holds that $\{q_0\} \times \zeta_{B_0}(q_0) = \{(q_0, b) \in V\}$, that is, ζ_0 is a complete map for q_0. If q_1 is one of the unreachable states, $\zeta_{B_0}(q_1)$ contains every blackboard that can be generated up to reaching q_1, and therefore so it does ζ_{B_1}. Otherwise ζ_{B_0} may or may not be a complete map for q_1, but it is sure that ζ_{B_1} is: by definition of topological sort, every ε-path reaching q_1 from a state of V_0 is completely traversed once every ε-derivation from q_0 is computed, and therefore every generated blackboard for q_1 has been added to $\zeta_{B_1}(q_1)$. Following the same reasoning for V_i and V_{i+1} with $i = 1 \ldots n-1$, we deduce $\zeta_{B_{i+1}}$ is a complete map for q_{i+1} with $i = 1 \ldots n-1$, and therefore (V_n, ζ_{B_n}) is equivalent to $C_\varepsilon(V)$. $\square$

In section 7.8 (p. 148) we gave an efficient definition of ε-closure based on ε-expansions; the main idea consisted in using only the ESs in $E = D(V) - V$ as source ESs in order to try to reach new ESs, since the ESs in V had already been used as source ESs and, hence, no new ESs would be derived from them. For the case of BSP, $D(q_i, \zeta_{B_i})$ returns states and maps that are not already present in (V_i, ζ_{B_i}), hence there is no need for an ε-expansion-based definition.

BSP requires to follow a topological sort of the execution machine substructures involved in the recognition of a string. The topological sort can be computed as these substructures are explored, but it is necessary to know first which substructures of the whole execution machine are going to be explored. Executing the machine in order to find these substructures and then executing it again by means of BSP makes no sense. However, there are cases in which it is sure that the whole execution machine will be explored, for instance when computing the whole language of a trimmed machine (without useless states or transitions; a simple method for the generation of the language of a FSA will be given in the next chapter). One may compute the set of outputs for a given machine and input sequence as another kind of machine recognizing this set, for instance a FSA for the case of FSTBOs or an output FPRTN for the case of RTNBOs (see chapters 15 and 16). In particular, the

former case is of interest since these output FPRTNs can be computed in polynomial time even when representing exponential languages. Once this machine is built, an output enumeration can be efficiently constructed by computing the represented language through BSP.

Chapter 11

Finite-state transducers with string output

We present in this section FSTSOs as a special kind of FSTBO in which blackboards are strings, output functions append a symbol to the output string, and there are no killing blackboards. These FSTSOs correspond to the definition of letter transducer given, for instance, in Roche and Schabes (1997, p. 14). As we have seen in section 10.7 (p. 199), other types of FSTs are possible such as sequential transducers (definition 166, p. 199), subsequential transducers (Schützenberger, 1977) and p-subsequential transducers (definition 167, p. 201), though all of them can be turned into an equivalent letter transducer (corollary 8, p. 201). Additionally, deterministic augmented letter transducers (Garrido-Alenda and Forcada, 2002) are a more general type of letter transducers, due to the included lookahead mechanism for input segmentation. FSTSOs have multiple applications (Mohri, 1997; Karttunen, 2001) such as parsing (Silberztein, 1993), information extraction (Hsu and Chang, 1999; Friburger and Maurel, 2002, 2004), phonology (Kaplan and Kay, 1994; Karttunen, 1993), morphology (Karttunen et al., 1992; Karttunen, 1993), spelling correction (Oflazer, 1996), speech processing (Mohri et al., 1996) and grammatical inference (Oncina et al., 1993; Oncina, 1998). We are mainly interested in parsing and information extraction by using string output for enriching texts with meta-information, for instance by inserting XML (Bray et al., 2008) tags that explicitly represent the syntactic

211

structure of the text sentences or identify the information to be extracted.[1] XML tags can be efficiently treated as output symbols instead of strings by representing them as pointers to the states of a trie, as explained in chapter 9.

Definition 176 (FSTSO). *A FSTSO $(Q, \Sigma, \Gamma, \delta, Q_I, F)$ is a special type of FSMs (definition 46, p. 121) whose set of labels Ξ is the set of input/output pairs $(\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})$, where Σ is a finite input alphabet, Γ a finite output alphabet and ε the empty symbol. FSTSOs can be seen as FSAs augmented with string output or as a particular type of FSTBO where*

- *functions in Γ always perform the concatenation of an output symbol to the current blackboard; for the sake of simplificity, we consider that Γ contains output symbols g rather than functions on blackboards, and output labels g represent the concatenation of a symbol g to the current blackboard,*

- *the identity function on blackboards id_B concatenates the empty symbol ε to the current blackboard; we will therefore use ε instead of id_B in order to explicitly state that a transition does not modify the current output,*

- *$B = \Gamma^*$, that is, blackboards are sequences of zero, one or more output symbols,*

- *$B_K = \emptyset$, that is, there are no killing output strings, and*

- *$b_\emptyset = \varepsilon$, that is, the empty blackboard is the empty string.*

11.1   Transitions

FSTSO transitions are a particular case of FSTBO transitions (section 10.1, p. 186):

- consuming transitions (definition 140, p. 186): $Q \times (\Sigma \times (\Gamma \cup \{\varepsilon\})) \times Q$,

- generating transitions (definition 141, p. 186): $Q \times ((\Sigma \cup \{\varepsilon\}) \times \Gamma) \times Q$,

---

[1]An example of grammar recognizing SMS command requests and delimiting phone number and message to send has been shown in figure 10.1, p. 188

- translating or substituting transitions (definition 142, p. 186): $Q \times (\Sigma \times \Gamma) \times Q$,

- deleting transitions (definition 143, p. 186): $Q \times (\Sigma \times \{\varepsilon\}) \times Q$,

- ε-transitions (definition 144, p. 186): $Q \times (\{\varepsilon\} \times (\Gamma \cup \{\varepsilon\})) \times Q$,

- inserting transitions (definition 145, p. 186): $Q \times (\{\varepsilon\} \times \Gamma) \times Q$ and

- ε^2-transitions (definition 146, p. 186): $Q \times (\{\varepsilon\} \times \{\varepsilon\}) \times Q$.

Substituting, deleting and inserting operations have been commonly used to give a measure of the difference between two strings: the *edit distance*, also called the Levenshtein distance (Levenshtein, 1966); the edit distance between two strings is equal to the minimal number of symbol substitutions, deletions and insertions to be performed in order to transform one string into the other. Edit distance is the basis of approximate string matching. An extensive discussion on this subject can be found in Navarro (2001).

11.2 Sequences of transitions

As for FSTSO transitions, FSTSO paths are a particular case of FSTBO paths. Every definition in section 10.3 (p. 187) can be straightforwardly adapted by replacing FSTBO transitions by their corresponding FSTSO transitions, hence we will not give more details here.

11.3 Behaviour

Definition 177 (Execution state). *FSTSO execution states are pairs $(q, z) \in (Q, \Gamma^*)$.*

Definition 178 (Δ). *The Δ function for FSTSOs is a simple direct-derivation function on SESs (definition 98, p. 137) with*

- $x_s = (q_s, z_s)$,

- $x_t = (q_t, z_t)$, *and*

- $d = q_t \in \delta(q_s, (\sigma, g)) \wedge z_t = z_s g$,

where $g \in \Gamma \cup \{\varepsilon\}$.

Definition 179 (D). *The D function for FSTSOs is itself a simple direct-derivation function on SESs (definition 98, p. 137) with*

- $x_s = (q_s, z_s)$,

- $x_t = (q_t, z_t)$, *and*

- $d = q_t \in \delta(q_s, (\varepsilon, g)) \wedge z_t = z_s g$,

where $g \in \Gamma \cup \{\varepsilon\}$.

Lemma 12 (Finite and infinite ε-closure). *The ε-closure of a FSTSO SESs V is infinite iff there exists an ES (q, z) within V or ε-reachable from an ES of V such that q is traversed by a generating ε-cycle.*

Proof. Since the FSTSO ε-closure function is a particular case of the FSTBO ε-closure function, this proof is a particular case of proofs of lemmas 10 (p. 189) and 11 (p. 191). □

Theorem 14. *The ε-closure is always finite for FSTSOs without generating ε-cycles.*

Recall that generating ε-cycles allow for infinite translations of finite input sequences, which makes no sense for the case of natural language grammars (e.g.: associating an infinite parse tree to a finite sentence).

Definition 180 (Initial and acceptance SESs). *Given the sets of initial and acceptance states of a FSTSO, Q_I and F, its initial and acceptance SESs are $(Q_I \times \{\varepsilon\})$ and $(F \times \Gamma^*)$, respectively.*

Note that, since FSTSOs have no killing blackboards, there is no need to restrict the outputs of acceptor ESs.

Definition 181 (τ). *We define $\tau(A)$, the language of translations of a FSTSO A, as the set of input/output sequence pairs $(w, z) \in \Sigma^* \times \Gamma^*$ such that w is recognized and translated into z by A, that is, the set of input/output sequence pairs such that the whole consumption of w reaches at least one acceptance ES from at least one initial ES through a path that generates z:*

$$\tau(A) = \{(w, z) : (q_f, z) \in \Delta^*((Q_I \times \{\varepsilon\}), w) \cap (F \times \Gamma^*)\}. \tag{11.1}$$

Definition 182 (ω). *We define $\omega(A, w)$, the translations of a word w for a FSTSO A, as the set of output sequences $z \in \Gamma^*$ such that (w, z) belongs to the translations of A:*

$$\omega(A, w) = \{z : (w, z) \in \tau(A)\}, \tag{11.2}$$

with $\tau(A)$ of the previous definition.

11.4   Recognized languages

In section 10.5 (p. 193) we proved that the killing-blackboard mechanism could be used in order to emulate Turing machines. Since FSTSOs define no killing blackboards, the same languages can be recognized by means of FSTSOs than by means of FSAs.

11.5   Translating a string

Algorithms for string translation with FSTSOs, either by means of a breadth-first or a depth-first exploration, can be easily derived from the corresponding FSTBO algorithms (algorithms 10.1 *fstbo_translate_string* and 10.5 *fstbo-_depth_first_translate_string*, pp. 197 and 199) by taking into account the differences between FSTSOs and FSTBOs listed in definition 176 (p. 212). These algorithms can be further improved by using the trie string management shown in section 9.1 (p. 178) since the involved concatenations consist in appending a symbol to a string, that is, one of the cases in which trie string management is faster than normal string concatenation.[2]

11.6   Language generation

The procedure for the generation of the language of a FSA described here is meant to be extended in further chapters for other machines, namely RTNs (chapter 12) and output FPRTNs (chapter 16). Output FPRTNs are a kind of finite state machines that efficiently represent the set of outputs generated by applying a RTNBO (a RTN with blackboard output). The language of

---

[2]Provided that the number of final output strings to generate is small enough w.r.t. the number of partial output strings to compute.

such output FPRTNs is later to be generated in order to extract the effective list of outputs. Moreover, this procedure will be the base for the extraction of the top-ranked output represented by a weighted output FPRTN (chapter 18).

The whole language represented by a FSA can be easily computed by transforming the FSA into a FSTSO as explained in the following theorem.

Theorem 15 (Language generation). *Let $A = (Q, \Sigma, \delta, Q_I, F)$ be a FSA and $A' = (Q', \Sigma', \Gamma, \delta', Q'_I, F')$ a FSTSO such that*

- $Q' = Q$, $Q'_I = Q_I$ *and* $F' = F$,

- $\Sigma' = \emptyset$,

- $\Gamma = \Sigma$, *and*

- $q_t \in \delta'(q_s, (\varepsilon, \sigma))$ *iff* $q_t \in \delta(q_s, \sigma)$,

then it holds that

$$L(A) = \omega_(A', \varepsilon) \tag{11.3}$$

Proof. Let it be the FSA and FSTSO of the previous theorem, and an input sequence $w = \sigma_1 \ldots \sigma_l \in \Sigma^*$. FSA A contains a path of the form

$$p = p_0(q_0, \sigma_1, q_1) p_1(q_1, \sigma_2, q_2) \ldots p_{l-1}(q_{l-1}, \sigma_l, q_l), \tag{11.4}$$

where paths p_i for $i = 0 \ldots l - 1$ are ε-paths or empty paths, iff FSTSO A' contains a path of the form

$$p' = p'_0(q_0, (\varepsilon, \sigma_1), q_1) p_1(q_1, (\varepsilon, \sigma_2), q_2) \ldots p_{l-1}(q_{l-1}, (\varepsilon, \sigma_l), q_l), \tag{11.5}$$

where paths p'_i for $i = 0 \ldots l - 1$ are ε^2-paths or empty paths. Therefore, a path p within A consumes w iff its equivalent path p' within A' translates ε into w. Finally, p is an interpretation within A iff A' is an interpretation within A', and therefore A recognizes w iff A' translates ε into w. □

Following this equivalence, any algorithm computing the translations of an input sequence for a FSTSO can be easily transformed into an algorithm computing the language of a FSA by considering every FSA transition as an ε-transition generating the original input, and computing the translations of ε instead of the translations of a given input sequence. Notice that recognizing the empty string does not require to apply function Δ and only

requires to apply the ε-closure once. Translator algorithms for language generation can be reduced to the computation of the ε-closure of the initial SES plus the extraction of the outputs from the generated acceptance ESs. Algorithm 11.1 *fsa_language* is such a simplified adaptation of the breadth-first algorithm 10.1 *fstbo_translate_string* adapted for string output.

Algorithm 11.1 fsa_language(A)   ▷ $L(A)$, eq. (136)

Input: $A = (Q, \Sigma, \delta, Q_I, F)$, a FSA
Output: L, the language of A
 1: $V \leftarrow \emptyset$
 2: $E \leftarrow \emptyset$
 3: **for each** $q \in Q_I$ **do**
 4:     unconditionally_add_enqueue_es($V, E, (q, \varepsilon)$)
 5: **end for**
 6: **while** $E \neq \emptyset$ **do**
 7:     $(q_s, w) \leftarrow$ dequeue(E)
 8:     **for each** $q_t \in \delta(q_s, \varepsilon)$ **do**
 9:         add_enqueue_es($V, E, (q_t, w)$)
10:     **end for**
11:     **for each** $(q_t, \sigma) : q_t \in \delta(q_s, \sigma)$ **do**
12:         add_enqueue_es($V, E, (q_t, w\sigma)$)
13:     **end for**
14: **end while**
15: $L \leftarrow \emptyset$
16: **for each** $(q, w) \in V : q \in F$ **do**
17:     add(L, w)
18: **end for**

The domain of application of the resulting algorithm is derived from the domain of application of the original algorithm: FSTSOs containing generating ε-cycles involved during the computation of the ε-closure are excluded from the domain since they lead to infinite ε-closures (see lemma 12, p. 214). Note that pruned FSAs leading to such FSTSOs by following the transformation of theorem 15 (p. 216) are in fact FSAs with consuming cycles, that is, FSAs representing infinite languages (see theorem 7, p. 164). As for the original algorithm, this algorithm can also be improved with the trie string management shown in section 9.1 (p. 178). BSP of FSTBOs (section 10.9, p. 205) can also be applied here since the substructure of the machine to

be explored for language generation is known: the whole machine, provided that the machine is trimmed (definition 118, p. 144).

Chapter 12

Recursive transition networks

RTNs (Woods, 1970) are finite-state machines equivalent to pushdown automata (Oettinger, 1961; Schützenberger, 1963; Evey, 1963; but see Hopcroft et al., 2000, chap. 6, p. 219) and CFGs (briefly described in appendix B, page 405). A major advantage of RTNs over CFGs is the ability to merge common parts of many CFG rules; consequently, not only a greater efficiency of representation is achieved but more efficient algorithms of application since separate processing of common parts is also factored out (Woods, 1969, sec. 1.7.3, p. 40). As stated in appendix B, CFGs can be extended with regular expressions in order to also allow for a more compact representation. However, the same advantages and disadvantages of FSAs over regular expressions (chapter 8, p. 161) take place here for RTNs over extended CFGs (ECFGs): it is faster and less cumbersome to manually write simple grammars as ECFGs with a text editor than as RTNs by means of some graphic interface (such as the ones of the Intex, Unitex and Outilex systems), but certain grammars can be more readable when graphically represented as RTNs than when represented as ECFGs with complex regular expressions (see figure 12.1).[1] Indeed, the graphical representation of RTNs used in the Intex, Unitex and Outilex systems (the graphs described throught sections 7.2, 10.2 and 12.2, pp. 124, 187 and 225, respectively) has been optimized in order to give a very intuitive view of natural language grammars.

We present here RTNs as FSAs extended with a subroutine jump mechanism. This mechanism allows for a better structuring of the grammar as well as for reusing grammar fragments: subgrammars or grammar blocks are

---

[1]Example extracted from (Paumier, 2004)

$$S \to E \mid F \qquad\qquad E \to acG \qquad\qquad G \to H \mid SdI$$
$$H \to SdccH \mid f \qquad\qquad I \to ccSdI \mid g \qquad\qquad F \to bdJ$$
$$J \to K \mid ccL \qquad\qquad K \to ccSdK \mid g \qquad\qquad L \to SdccL \mid f$$

(a) CFG

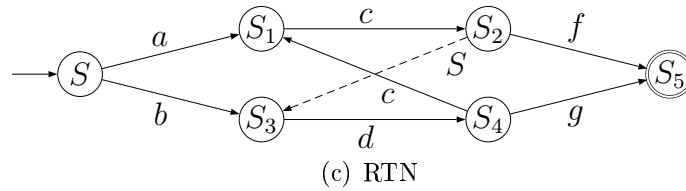$$S \to (ac((Sdcc)^* f + Sd(ccSd)^* g)) + (bd((ccSd)^* g + cc(Sdcc)^* f))$$

(b) ECFG

(c) RTN

Figure 12.1: Equivalent (a) CFG, (b) ECFG and (c) RTN.

defined for local structures, and other subgrammars may be defined from a higher point of view by means of subroutine jumps to lower level subgrammars (see figure 12.3, p. 224, for a simple example of structured RTN). Examples of local structures are Korean time adverbs (Jung, 2005), French location adverbs (Constant, 2003b), French measure expressions (Constant, 2003b) and Greek frozen adverbs (Voyatzi, 2006). As for any piece of software, readability, reusability and well-structuring are crucial for the construction and maintenance of large and complex grammars, such as natural language ones.[2] We give a formal definition similar to the informal one given by Woods (1970) but labelling call transitions with sets of states instead of single states, which facilitates the definition of reverse RTNs. Woods (1969, sec. 3.3, p. 82) gives another definition of RTN which is straightforwardly derived from the definition of CFG, facilitating the redefinition of Earley's (1970) parser for the RTN case.[3] We derive in section 12.10 an alternative definition of the Earley parser based on the FSA-like definition of RTNs and the generic breadth-first algorithm of application of FSMs (sec. 7.9, p. 152).

The application of a RTN may not only result in the acceptance or the rejection of a sentence but also in a description of the sentence structure,

---

[2]Extensive material on concepts and principles of software design can be found in Pressman (2001, chap. 13, p. 335)

[3]The Earley parser was originally conceived for the application of CFGs; see appendix C (p. 411) for a brief description and a discussion on the original Earley parser.

represented as the path or paths that allow to recognize the sentence. In this case, grammar and sentence structures are analogous. However, this prevents from possible restructurings of the grammar that may boost the machine application, such as the weak Greibach normal form for RTNs (Paumier, 2004),[4] since different grammar structures yield different sentence structures in spite of not altering the set of accepted sentences. We have chosen to represent sentence structures as output XML tags bounding the sentence components instead of subgrammar labels (see figure 13.2, p. 259). Hence, it is not required to explicitly define subgrammars nor subgrammar labels (analogous to CFG non-terminals); we call a subgrammar by specifying its set of subinitial states. Let this set be Q_c, each subgrammar is implicitly defined as the machine substructure reachable from Q_c, including Q_c and excluding the substructures of other called subgrammars. We permit subgrammars to share states and transitions in order to support grammar optimizations such as the weak Greibach normal form. However, RTN subgrammars are intended to be disjoint before applying such optimizations for the sake of modularity. Common parts of subgrammars could be avoided by simply replicating such parts, but that would imply a loss of efficiency.

Definition 183 (RTN). *A RTN $(Q, \Sigma, \delta, Q_I, F)$ is a FSA $(Q, \Sigma, \delta, Q_I, F)$ (definition 128, p. 162) extended with a subroutine jump mechanism: its set of transition labels Ξ takes its elements from $(\Sigma \cup \{\varepsilon\}) \cup \mathcal{P}(Q)$, where*

- *labels of the form $\Sigma \cup \{\varepsilon\}$ have the same interpretation as in the case of FSAs, and*

- *labels of the form $\mathcal{P}(Q)$ represent subroutine jumps or calls to state sets (definition 187 in the next section).*

12.1 Transitions

Definition 184 (Consuming transition). *Following definition 51 (p. 123), transitions in $Q \times \Sigma \times Q$, that is, which consume an input symbol, are called consuming transitions.*

---

[4]This weak Greibach normal form is an adaptation of the ECFG Greibach normal form (Albert et al., 1998), which in turn is an extension of the CFG Greibach normal form (Greibach, 1965; Koch and Blum, 1997).

Consuming transitions correspond to terminal symbols within the body of CFG production rules (e.g.: production '$N \rightarrow$ garden' of CFG of figure 12.2 and transition $(q_{N_0}, \%\text{garden}, q_{N_1})$ of figure 12.3).

Definition 185 (Explicit ε-transition). *Following definition 53 (p. 124), transitions in $Q \times \{\varepsilon\} \times Q$, that is, which do not consume input, are called explicit ε-transitions.*

ε-transitions correspond to ε symbols within the body of CFG production rules.

Definition 186 (Implicit ε-transition). *Within the context of RTNs there are two kinds of implicit ε-transitions, that is, transitions that are implicitly defined by the RTN and which do not require to consume input when traversed: push transitions (definition 189) and pop transitions (definition 190).*

Definition 187 (Call transition). *Call transitions are transitions of the form $(q_s, Q_c, q_t) \in Q \times \mathcal{P}(Q) \times Q$ and represent a subroutine jump to a set of states Q_c, that is, the recursive application of the whole RTN taking Q_c as set of initial states before bringing the machine to state q_t. The exact behaviour of call transitions is governed by the RTN implicit ε-transitions.*

Call transitions correspond to non-terminal symbols within the body of CFG production rules (e.g.: non-terminal symbol 'PP' in production '$VP \rightarrow VP\ PP$' of CFG of figure 12.2 and transition $(q_{VP_1}, q_{PP_0}, q_{VP_3})$ of figure 12.3).

Definition 188 (Subinitial set of states). *We say a subset of states Q_c of a machine A is a subinitial set of states (SS) of A iff A contains at least one call to Q_c.*

Subinitial SSs correspond to CFG non-terminals expanding into one or more right-hand sides; every rule left-hand side with the same non-terminal symbol is condensed into a single subinitial SS (e.g.: heads of productions '$VP \rightarrow VP\ PP$' and '$VP \rightarrow V\ NP$' of CFG of figure 12.2 and subinitial SS $\{q_{VP_0}\}$ of figure 12.3).

Definition 189 (Push transition). *Push transitions are implicit ε-transitions which take place each time a state having at least one outgoing call transition is reached: for each call transition (q_s, Q_c, q_r), and for each state $q_c \in Q_c$, the machine implicitly defines a push transition $(q_s, q_r\downarrow, q_c)$ which brings the machine from source state q_s to called state q_c, without input consumption. Additionally, the transition pushes return state q_r onto the stack, action that we represent as $q_r\downarrow$. Push transitions are subroutine jump initializers.*

$$VP \rightarrow VP\ PP \qquad\qquad NP \rightarrow DET\ N \qquad\qquad PP \rightarrow PREP\ NP$$
$$VP \rightarrow V\ NP \qquad\qquad NP \rightarrow NP\ PP$$

$$N \rightarrow \text{monkey} \qquad\qquad V \rightarrow \text{watch} \qquad\qquad PREP \rightarrow \text{in}$$
$$N \rightarrow \text{telescope} \qquad\qquad\qquad\qquad\qquad\qquad PREP \rightarrow \text{with}$$
$$N \rightarrow \text{garden} \qquad\qquad DET \rightarrow \text{the}$$

Figure 12.2: A left-recursive CFG representing a toy grammar which recognizes sentence "Watch the monkey with the telescope in the garden", among others, with non-terminal *VP* as the grammar's start symbol; *VP* stands for verb phrase, *NP* for noun phrase and *PP* for prepositional phrase.

Definition 190 (Pop transition). *Pop transitions are implicit ε-transitions which take place each time an acceptance state $q_f \in F$ is reached during a subroutine jump: for each pair of states $(q_f, q_r) \in F \times Q$, the machine implicitly defines a pop transition $(q_f, q_r\uparrow, q_r)$ which pops state q_r from the stack and brings the machine to state q_r, with q_r as the state at the top of the stack.*

Definition 191 (Realization of call transitions). *A call transition (q_s, Q_c, q_r), or simply a call to Q_c, is realizable iff there exists at least one realizable path p starting with one of the corresponding transitions pushing q_r onto the stack and ending with a transition popping the previously pushed q_r from the same stack position. If p exists then we say call to Q_c is realizable through path p.*

Definition 192 (Call completion). *During the process of application of a machine with calls, we say a call is* uncompleted *or* unresolved *when a path has been executed up to realizing the corresponding push transitions, but not up to realizing any of the corresponding pop transitions; we call the process of realizing a pop transition a* call completion *or* resolution.

Definition 193 (ε-call). *We say a call to a subinitial SS Q_c is an ε-call, a deletable call or an ε-realizable call iff it is realizable through an ε-path (see definition 69, p. 129).*

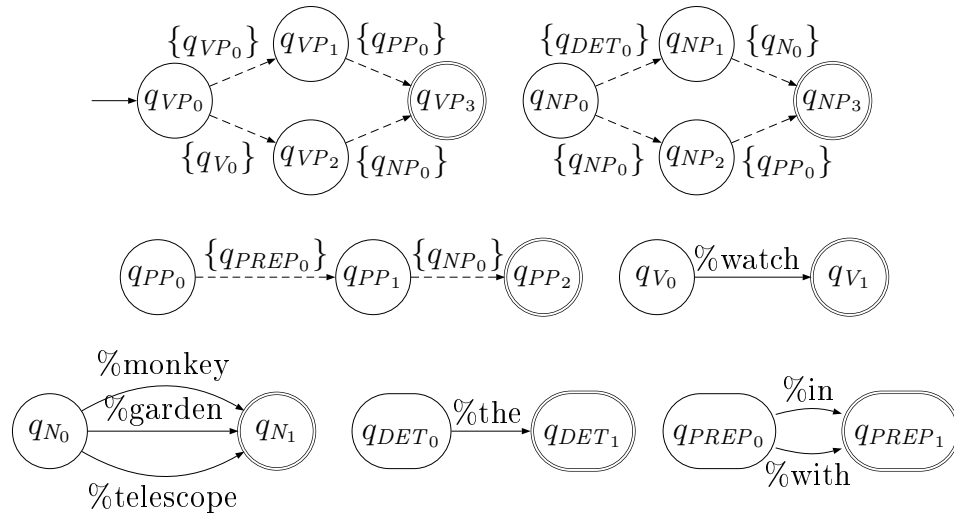These ε-calls correspond to deletable non-terminals within the bodies of CFG production rules.

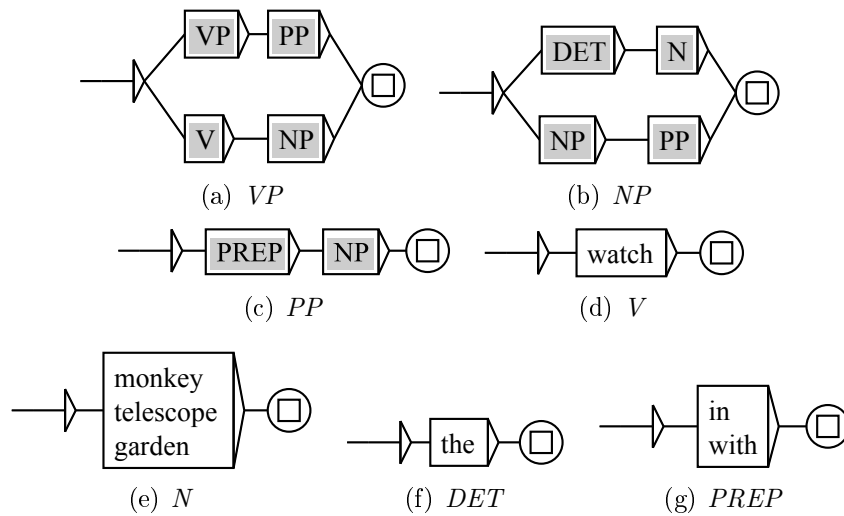**Figure 12.3:** Left-recursive RTN equivalent to CFG of figure 12.2.

(a) *VP*

(b) *NP*

(c) *PP*

(d) *V*

(e) *N*

(f) *DET*

(g) *PREP*

Figure 12.4: Unitex set of graphs equivalent to CFG of figure 12.2.

12.2 Graphical representation

Unitex and Intex graphs represent calls to subgraphs as subgraph identifiers with shaded background (see figure 12.4). We represent call transitions as dashed arrows labeled with the called subinitial SS (see figure 12.3).

There are cases in which it can be useful to explicitly represent implicit transitions (push and pop transitions for the case of RTNs), for instance when graphically representing an execution trace for the visualization of the execution paths generated by an algorithm of application of a machine (e.g.: the execution trace in figure 12.7, p. 238). ESs and transitions between ESs are represented as for states and transitions between states of a machine, though labels of ESs may be structures rather than single elements. We usually mark useless ESs (those who do not derive any acceptance ES) with two crossed lines. Push transitions are represented as dotted arrows, and pop transitions as thick arrows. Both types of transitions are labeled with the return state of the call they implement, that is, the pushed state for the case of push transitions and the popped state for the case of pop transitions. For the latter case, since the popped and target states are the same, the transition label can be omitted. For the case of algorithms that use other more complex subroutine jump mechanisms than the one based on a stack, such as the Earley-like ones (the Earley RTN case will be described in section 12.10), push and pop transitions may be labeled with structures more complex than the return state (e.g.: execution trace of figure 12.7, p. 238).

12.3 Sequences of transitions

Definition 194 (Explicit path). *An explicit path is a path composed by explicit transitions (definition 185, p. 222).*

Note that explicit paths are not necessarily realizable since call transitions may not be realizable.

Definition 195 (ε-path). *Following definition 69 (p. 129), within the context of RTNs, an ε-path is a path that can be traversed without input consumption, that is, whose transitions are either implicit ε-transitions, explicit ε-transitions or ε-call transitions.*

Definition 196 (Explicit ε-path). *An explicit ε-path is both and ε-path and an explicit path.*

Definition 197 (Call cycle). *Let p be a path having q_s as start state, we say p is a call cycle iff its last transition is a push transition having q_s as target state and, during the whole cycle, the pushed state is never popped from that position of the stack.*

Definition 198 (Call ε-cycle). *A call ε-cycle is both an ε-path and a call cycle.*

Definition 199 (Recursive call). *We say a call to a SS Q_c within a RTN A is recursive iff there exists at least one call cycle within A starting at a state of Q_c.*

RTN recursive calls correspond to CFG productions of the form '$A \to \alpha A \beta$'.

Definition 200 (Left-recursive call). *We say a call to a SS Q_c within a RTN A is left-recursive iff there exists at least one call ε-cycle within A starting at a state of Q_c.*

RTN left-recursive calls correspond to CFG productions of the form '$A \to A\alpha$' (e.g.: production '$VP \to VP\ PP$' of CFG of figure 12.2 and call transition $(q_{VP_0}, q_{VP_0}, q_{VP_1})$ of figure 12.3).

Definition 201 (Right-recursive call). *A call to a SS Q_c is right-recursive iff it is realizable through a path $(q_s, q_r\downarrow, q_c)pp'p''$, where p is a call cycle, p' is a path completing call to Q_c and p'' is an ε-path.*

RTN right-recursive calls correspond to CFG productions of the form $A \to \alpha A$.

Definition 202 (Deletable recursion). *We say a path completing a call is a deletable recursion iff it implies the call to be both left- and right-recursive, that is, it is a path having the same form than the one of the previous definition but p is not only a cycle but also an ε-cycle.*

RTN deletable recursions correspond to CFG productions of the form $A \to A$; we call them *deletable* since they do not contribute anything to the grammar description: saying that the structure of A is equal to its own structure does not clarify what A can be made of.

Definition 203 (Recursive machine). *We say a machine is recursive iff it contains at least one recursive call, left-recursive iff it contains at least one left-recursive call, and right-recursive if it contains at least one right-recursive call.*

Definition 204 (Recursion degree). *The recursion degree of a machine with call transitions is equal to the maximum number of useful self-concatenations or consecutive traversals of its call cycles.*

12.4 Substructures

Definition 205 (Submachine). *Let Q_c be a set of initial states of a machine or one of its subinitial SSs, we define its Q_c-submachine as the machine substructure composed by Q_c and every state and transition of every explicit path starting at a state of Q_c.*

A RTN submachine corresponds to a Unitex graph or to a subset of productions of a CFG containing every production having a particular non-terminal as head (e.g.: the set of productions of CFG of figure 12.2 starting with *VP*, graph *VP* of figure 12.4 and $\{q_{VP_0}\}$-submachine of figure 12.3).

Definition 206 (Axiom submachine). *We define the axiom submachine of a machine A as its Q_c-submachine such that Q_c is the set of initial states of A. Let A represent a grammar, the axiom submachine corresponds to the grammar's axiom or start symbol (see definition 288 in appendix B, p. 405).*

12.5 Behaviour

Definition 207 (Execution state). *RTN ESs are pairs $(q, \pi) \in (Q \times Q^*)$ where π is a stack of return states, λ being the empty stack.*

The realization of RTN transitions falls into the FSM general categories of pure consuming transitions and pure ε-transitions (definitions 87 and 88, pp. 133 and 134, respectively) except for push, pop and call transitions.

Definition 208 (Push transition realization). *A push transition $(q_s, q_r\downarrow, q_c)$ is realizable from ES (q_s, π), for any stack π, and its realization brings the machine to ES $(q_t, \pi q_r)$.*

Definition 209 (Pop transition realization). *A pop transition $(q_s, q_r\!\uparrow, q_c)$ is realizable from ES (q_s, π) iff $\pi = \pi' q_r$ and $q_s \in F$, and its realization brings the machine to ES (q_r, π').*

Since the realization of call transitions depends on the realization of push and pop transitions, as well as on the paths transitively connecting push transitions to pop transitions, we rather deduce whether a call transition can or cannot be realized rather than adding a separate definition.

Lemma 13 (Call transition realization). *A call transition (q_s, Q_c, q_r) is realizable from an ES (q_s, π) by bringing the machine to ES (q_r, π), for any stack π, iff there exists at least one explicit path p starting at a state $q_c \in Q_c$ and ending at an acceptor state such that p is composed by*

- *either consuming transitions or explicit ε-transitions, or*

- *either consuming transitions, explicit ε-transitions or call transitions realizable from the ES reached just before each corresponding call,*

the second case requiring for each call transition at least one finite sequence of recursively realizable calls so that the last call of the sequence falls into the first case.

Proof. Let $t = (q_s, Q_c, q_r)$ be a call transition of a RTN A, and

$$p = (q_s, q_r\!\downarrow, q_c) p'(q_s, q_r\!\uparrow, q_r)$$

a path inside A with $q_c \in Q_c$. The push transition is realizable by pushing q_r onto the stack. If p' is composed only by consuming transitions and/or explicit ε-transitions then it is also realizable. The pop transition is realizable by popping the previously pushed state q_r since p' does not modify the stack. Let p' be composed by a unique call transition t' such that t' is realizable through a path

$$p' = (q_s', q_r'\!\downarrow, q_c') p''(q_s', q_r'\!\uparrow, q_r').$$

If p'' is composed only by consuming transitions and explicit ε-transitions, the realization of t' falls into the first case. The realization of t' momentarily modifies the stack by pushing a return state but popping it again, thus the same reasoning than for the first case applies here for the realization of t. If t' is realizable through another call which is completable through another call and so on recursively, t is realizable as long as the last call is

completable through a path only composed by consuming transitions and/or explicit ε-transitions. Since the realization of each call leaves the stack as before the call, a path composed by any sequence of calls is realizable as long as every call is individually realizable, the presence of consuming transitions and explicit ε-transitions in between not modifying this fact since they are always realizable and do not modify the stack. Obviously, for any other cases t is not realizable, either because there is no path reaching an acceptance state which would allow for the realization of the corresponding pop transition or because the path traverses a non-realizable call transition. $\qquad\square$

Definition 210 (Δ). *The Δ function for RTNs is a simple direct-derivation function on SESs (definition 98, p. 137) with*

- $x_s = (q_s, \pi)$,

- $x_t = (q_t, \pi)$, *and*

- $d = q_t \in \delta(q_s, \sigma)$.

Definition 211 (D). *The D function for RTNs is composed by 3 simple direct-derivation functions on SESs (definition 98, p. 137), D_ε with*

- $x_s = (q_s, \pi)$,

- $x_t = (q_t, \pi)$, *and*

- $d = q_t \in \delta(q_s, \varepsilon)$,

D_{push} *with*

- $x_s = (q_s, \pi)$,

- $x_t = (q_c, \pi q_t)$, *and*

- $d = q_t \in \delta(q_s, Q_c) \wedge q_c \in Q_c$,

and D_{pop} with

- $x_s = (q_f, \pi q_r)$,

- $x_t = (q_r, \pi)$, *and*

- $d = q_f \in F$.

Lemma 14 (Infinite ε-closure). *The ε-closure of a RTN SES V is infinite if there exists an ES (q, π) within V or ε-reachable from an ES of V such that q has an outgoing left-recursive call transition.*

Recursive-descent parsers applying left-recursive CFGs may fall into infinite loops (Aho et al., 1986, sec. 2.4, p. 47). Since RTNs and CFGs are equivalent, the same problem arises for the case of the base top-down breadth-first and top-down depth-first acceptors (algorithms 7.5 and 7.8, pp. 153 and 157) when adapted for the application of RTNs.

Proof. The proof is analogous to the one for FSTBOs with generating ε-cycles (proof of lemma 10, p. 189). Left-recursive calls behave as generating ε-cycles: it is possible to traverse infinite times and without consuming input the sequence of states that form the cycle, but for each cycle traversal the stack of return states will be incremented with at least one new return state from the left-recursive call —as happened with the increasing output sequence; hence, each successive traversal of the cycle will generate new ESs with larger stacks.   □

Lemma 15 (Finite ε-closure). *The ε-closure of a RTN SES V is finite iff there is no ES (q, π) within V or ε-reachable from V such that q has an outgoing left-recursive call transition.*

Proof. As we have seen for FSAs, ε-paths having only explicit ε-transitions do not yield infinite SESs since every ES derived through an explicit ε-transition (q_s, ε, q_t) from an ES (q_s, π) is of the form $(q_t, \pi) \in (Q \times \{\pi\})$, which is a finite set. Even if the ε-path is a cycle, during the first path traversal every possible ES will be added to the ε-closure and further traversals will not add new ESs. The completion of a non-left-recursive call through an ε-path does not produce an infinite SES either. Let $(q_0, \pi) \in V_0$ be an ES from where we find a call transition (q_0, Q_c, q_n) that is completed through an ε-path

$$(q_0, q_n\downarrow, q_1), (q_1, \varepsilon, q_2) \ldots (q_{n-2}, \varepsilon, q_{n-1}), (q_{n-1}, q_n\uparrow, q_n), \qquad (12.1)$$

that is, an ε-path whose first transition is a push transition initiating the call $(q_1 \in Q_c)$, the last one is a pop transition that returns from the call, and the middle transitions are explicit ε-transitions; following the iterative ε-closure

of V_0, it holds that

$$
\begin{array}{rcl}
(q_0, \pi) & \in & V_0 \\
(q_1, \pi q_n) & \in & V_1 \\
(q_2, \pi q_n) & \in & V_2 \\
& \vdots & \\
(q_{n-1}, \pi q_n) & \in & V_{n-1} \\
(q_n, \pi) & \in & V_n.
\end{array}
$$

As we can see, every ES produced during the call belongs to the domain $(Q \times \pi q_n)$, which is also finite even if the ε-path that completes the call has ε-cycles. As well, if the ε-path contains a finite succession of calls that are always completed by means of explicit ε-paths, the total number of ESs is also finite since each call produces a finite number of ESs. If any of these calls can also be completed through a non-left-recursive subcall which is completed through an explicit ε-path and/or successive non-left-recursive calls, the number of ESs is finite as well since the number of ESs given by the subcall is finite. Any finite number of subcalls will produce as well a finite number of ESs. Finally, non-left-recursive calls that cannot be completed by means of an ε-path give also a finite number of ESs, since it is a subcase of non-left-recursive calls that can be completed through ε-paths (the ε-closure explores some part of the calls but not up to completing them). □

Theorem 16. *Following lemmas 14 and 15, the ε-closure is always finite for non-left-recursive RTNs.*

Definition 212 (Initial and acceptance SESs)**.** *Given the sets of initial and acceptance states of a RTN, Q_I and F, its initial and acceptance SESs are $(Q_I \times \{\lambda\})$ and $(F \times \{\lambda\})$, respectively, with λ the empty stack.*

Definition 213 (Execution machine)**.** *The execution machine of a RTN A is defined as for the generic execution machine (definition 105, p. 142) without any other kind of transitions than pure consuming transitions and pure ε-transitions, thus its definition is equal to that of a FSA though possibly having an infinite set of states, transitions and acceptance states.*

As for the FSTBO case (definition 158, p. 192), the execution machine of a RTN does not require to define call, push or pop transitions since they are replaced by pure ε-transitions that point to states which already include the resulting stack after pushing or popping the corresponding return state.

Definition 214 (L). *Following definition 107 (p. 143), we define $L(A)$, the language accepted by a RTN A, as*

$$L(A) = \{w \in \Sigma^* : \Delta^*((Q_I \times \{\lambda\}), w) \cap (F \times \{\lambda\}) \neq \emptyset\}. \qquad (12.2)$$

Lemma 16 (Infinite recursion degree). *The recursion degree of a RTN having at least one useful call cycle is infinite.*

Proof. Let A be a RTN containing the structure of figure 12.5 so that

$$p = p_a \ (q_{s_1}, q_{r_1}\downarrow, q_{c_1}) \ p_b \ (q_{s_2}, q_{r_2}\downarrow, q_{c_1}) \ p_c \ (q_{f_3}, q_{r_2}\uparrow, q_{r_2}) \ p_d \ (q_{f_2}, q_{r_1}\uparrow, q_{r_1}) \ p_e$$

is a path within A, path $p_b \ (q_{s_2}, q_{r_2}\downarrow, q_{c_1})$ is a call cycle, and the realization of p produces the sequence of ESs

$$
\begin{array}{ccc}
(q_0, \lambda) & \ldots & (q_{s_1}, \pi_1) \\
(q_{c_1}, \pi_1 q_{r_1}) & \ldots & (q_{s_2}, \pi_1 q_{r_1} \pi_2) \\
(q_{c_1}, \pi_1 q_{r_1} \pi_2 q_{r_2}) & \ldots & (q_{f_3}, \pi_1 q_{r_1} \pi_2 q_{r_2}) \\
(q_{r_2}, \pi_1 q_{r_1} \pi_2) & \ldots & (q_{f_2}, \pi_1 q_{r_1}) \\
(q_{r_1}, \pi_1) & \ldots & (q_{f_1}, \lambda).
\end{array}
$$

Path p is an interpretation within A since $(q_0, \lambda) \in X_I$ and $(q_{f_1}, \lambda) \in X_F$. By suppressing the call cycle we obtain path

$$p_0 = p_a \ (q_{s_1}, q_{r_1}\downarrow, q_{c_1}) \ p_c \ (q_{f_3}, q_{r_1}\uparrow, q_{r_1}) \ p_e,$$

whose realization produces the sequence of ESs

$$
\begin{array}{ccc}
(q_0, \lambda) & \ldots & (q_{s_1}, \pi_1) \\
(q_{c_2}, \pi_1 q_{r_1}) & \ldots & (q_{f_3}, \pi_1 q_{r_1}) \\
(q_{r_1}, \pi_1) & \ldots & (q_{f_1}, \lambda).
\end{array}
$$

As for p, path p_0 is an interpretation within A. By self-concatenating the call cycle $k \geq 1$ times we obtain the infinite family of paths

$$
\begin{aligned}
p_k = p_a \ (q_{s_1}, q_{r_1}\downarrow, q_{c_1}) \ (p_b \ (q_{s_2}, q_{r_2}\downarrow, q_{c_1}))^k \ p_c \ (q_{f_3}, q_{r_2}\uparrow, q_{r_2}) \\
(p_d \ (q_{f_2}, q_{r_2}\uparrow, q_{r_2}))^{k-1} \ p_d \ (q_{f_2}, q_{r_1}\uparrow, q_{r_1}) \ p_e
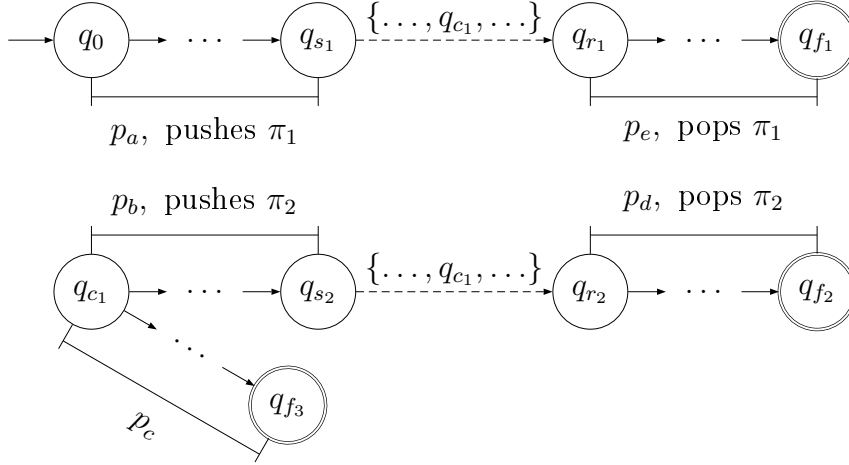\end{aligned}
$$

Figure 12.5: RTN generic structure containing infinite interpretations due to recursive call to $\{\ldots, q_{c_1}, \ldots\}$.

The realization of these paths produce the following family of sequences of ESs...

$$
\begin{aligned}
(q_0, \lambda) &\quad \ldots \quad (q_{s_1}, \pi_1) \\
(q_{c_1}, \pi_1 q_{r_1}) &\quad \ldots \quad (q_{s_2}, \pi_1 q_{r_1} \pi_2) \\
(q_{c_1}, \pi_1 q_{r_1} \pi_2 q_{r_2}) &\quad \ldots \quad (q_{s_2}, \pi_1 q_{r_1} \pi_2 q_{r_2} \pi_2) \\
(q_{c_1}, \pi_1 q_{r_1} \pi_2 q_{r_2} \pi_2 q_{r_2}) &\quad \ldots \quad (q_{s_2}, \pi_1 q_{r_1} \pi_2 q_{r_2} \pi_2 q_{r_2} \pi_2) \\
&\quad \vdots \\
(q_{c_1}, \pi_1 q_{r_1} (\pi_2 q_{r_2})^{k-1}) &\quad \ldots \quad (q_{s_2}, \pi_1 q_{r_1} (\pi_2 q_{r_2})^{k-1} \pi_2) \\
(q_{c_1}, \pi_1 q_{r_1} (\pi_2 q_{r_2})^{k} \pi_2 q_{r_2}) &\quad \ldots \quad (q_{f_3}, \pi_1 q_{r_1} (\pi_2 q_{r_2})^{k} \pi_2 q_{r_2}) \\
(q_{r_2}, \pi_1 q_{r_1} (\pi_2 q_{r_2})^{k} \pi_2) &\quad \ldots \quad (q_{f_2}, \pi_1 q_{r_1} (\pi_2 q_{r_2})^{k}) \\
(q_{r_2}, \pi_1 q_{r_1} (\pi_2 q_{r_2})^{k-1} \pi_2) &\quad \ldots \quad (q_{f_2}, \pi_1 q_{r_1} (\pi_2 q_{r_2})^{k-1}) \\
&\quad \vdots \\
(q_{r_2}, \pi_1 q_{r_1} \pi_2) &\quad \ldots \quad (q_{f_2}, \pi_1 q_{r_1}) \\
(q_{r_1}, \pi_1) &\quad \ldots \quad (q_{f_1}, \lambda),
\end{aligned}
$$

...and therefore paths p_k constitute an infinite family of interpretations within A. $\qquad\square$

Theorem 17 (Possible recursion degrees). *The recursion degree of a RTN is either zero or infinite.*

In chapter 15 we will present other kind of RTNs —FPRTNs— having a not so obvious set of possible recursion degrees: zero, one or infinite. The proof for the case of FPRTNs is an extension of the proof for the case of RTNs.

Theorem 18 (Cardinality of the interpretation set). *Given the previous theorem and the theorems 4 (p. 145) and 6 (p. 164) on the cardinality of the interpretation set for FSMs and for FSAs, the number of interpretations of a RTN is infinite iff it contains at least one useful non-call cycle or its recursion degree is not zero.*

Theorem 19 (Cardinality of the language). *Given theorem 5 (p. 146), since FSAs allow for the realization of any of its transitions, the language of a RTN is infinite iff it contains at least one useful consuming cycle, which in this case can be a call cycle as well.*

12.6   Reverse RTN

Definition 215 (Reverse RTN). *Let A be a RTN $(Q, \Sigma, \delta, Q_I, F)$ with disjoint submachines; we define A^R, the canonical reverse of A, as a RTN $(Q, \Sigma, \delta', Q_I', F')$ such that*

- *Q_I' is the set of acceptance states of A's axiom submachine,*

- *F' is the union of Q_I and every subinitial state of A,*

- *A^R contains a consuming transition or explicit ε-transition t iff A contains transition t^R, and*

- *A^R contains a call transition (q_s, Q_c, q_t) with F_c as the set of acceptance states of its Q_c-submachine iff A contains a call transition (q_t, F_c, q_s) with Q_c as the set of acceptance states of its F_c-submachine.*

Push and pop transitions are implicitly defined by the previous call transitions.

Lemma 17 (Reverse RTN). *Let A be a RTN with disjoint submachines, A^R is a reverse of A.*

Proof. The proof for the case of words which are recognized by means of paths containing FSA transitions, that is, without subroutine jumps, is the same than for FSAs (proof of lemma 9, p. 165). Let $t = (q_s, Q_c, q_t)$ be a call transition within A such that t is realizable through a path

$$(q_s, q_t\downarrow, q_c)p(q_f, q_f\uparrow, q_t)$$

with $q_c \in Q_c$, and p does not contain push, pop or call transitions; RTN A^R contains a call (q_t, F_c, q_s) with F_c equal to the set of acceptance states of the Q_c-submachine of A, and this call is realizable through a path

$$(q_t, q_s\downarrow, q_f)p^R(q_c, q_s\uparrow, q_s)$$

that consumes w^R. No other words are recognized by a call to F_c due to the reversal of other submachine than the Q_c-submachine since submachines are disjoint. Note that given two non-disjoint submachines of A for Q_c and Q_c' with F_c and F_c' as sets of acceptor states, reversed submachines F_c and F_c' of A^R may reach states that are not reachable by the non-reversed submachines of Q_c and Q_c'. If p contains a finite recursion degree n of calls, the same reasoning is to be applied n recursive times. Finally, if A recognizes a word w through a path p starting at a state $q_s \in Q_I$ and ending at a state $q_t \in F$, then A^R recognizes w^R through a path p' starting at $q_t \in Q_I'$ and ending at $q_s \in F'$. Consequently, $L^R(A) = L(A^R)$ is true. $\square$

As stated before, non-disjoint submachines can be made disjoint by replicating their shared substructures, thus any RTN can be reversed as explained above. Anyway, we will not need to reverse any machine with non-disjoint submachines since the grammars we will treat are built as sets of disjoint Unitex's graphs.

12.7 Recognizing a string

The base breadth-first and depth-first acceptor algorithms 7.5 (p. 153) and 7.8 (p. 157) can be adapted for RTNs as explained in section 7.9 (p. 152), but excluding left-recursive RTNs from their domain of application in order to avoid infinite ε-closures. The difference between RTNs and the simplest FSMs, FSAs, is the subroutine jump mechanism, which is implemented by adding a couple of ε-moves (push and pop transitions) that operate on a stack. The main modification to be done to the base breadth-first acceptor

lies in the computation of the ε-closure, which we show in algorithm 12.1 *rtn_interlaced_eclosure*. The adaptation of the depth-first base acceptor can be straightforwardly performed by following the definition of $D(V)$ for RTNs.

Algorithm 12.1 rtn_interlaced_eclosure(V, E) $\triangleright C_\varepsilon(V)$

Input: V, the SES whose ε-closure is to be computed
        E, the queue of unexplored ESs containing every ES in V
Output: V after computing its ε-closure
        E after emptying it
 1: **while** $E \neq \emptyset$ **do**
 2:     $(q_s, \pi) \leftarrow \text{dequeue}(E)$
                                        $\triangleright$ EXPLICIT ε-TRANSITIONS
 3:        **for each** $q_t \in \delta(q_s, \varepsilon)$ **do**
 4:            add_enqueue_es$(V, E, (q_t, \pi))$
 5:        **end for**
                                        $\triangleright$ PUSH-TRANSITIONS
 6:        **for each** $(q_r, Q_c) : q_r \in \delta(q_s, Q_c)$ **do**
 7:            **for each** $q_c \in Q_c$ **do**
 8:                add_enqueue_es$(V, E, (q_c, \pi q_r))$
 9:            **end for**
10:        **end for**
                                        $\triangleright$ POP TRANSITIONS
11:        **if** $\pi = \pi' q_r \wedge q_s \in F$ **then**
12:            add_enqueue_es$(V, E, (q_r, \pi'))$
13:        **end if**
14: **end while**

Figure 12.7 is a graphical representation of the execution trace of the breadth-first acceptor algorithm adapted for RTNs, for RTN of figure 12.6 and input *aabb*. As we can see in the execution trace, the number of concurrent explorations of the RTN doubles each time an *a* is consumed. Even though this number is reduced each time a *b* is consumed, the number of generated ESs increases exponentially w.r.t. the length of input $a^n b^n$.[5] Determinizing the RTN would have avoided this duplication, keeping a linear

---

[5]This is a minimal theoretical case whose purpose is to illustrate the problem of the exponential output generation; an example of exponential output generation for the case of natural language grammars has been given in section 1.5.4, p. 19.
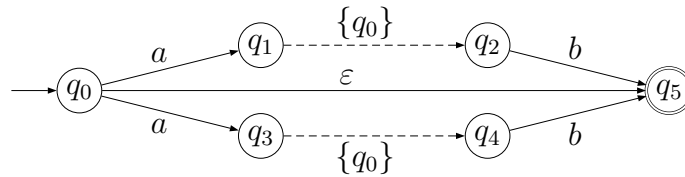
Figure 12.6: Non-deterministic RTN with two calls whose computation can be factored out; solid transitions represent either consuming transitions or explicit ε-transitions, and dashed transitions represent call transitions.

relation between the number of ESs and the input length. However, determinization may be too complex or even impossible, for instance for machines on an alphabet of predicates rather than symbols or machines with output (this has been discussed in sections 8.5 and 10.7, pp. 166 and 199). Assuming that the leftmost transitions of each ES within the trace are the first transitions of each ES, the depth-first acceptor will just generate the leftmost execution path; therefore, its execution cost will be linear for this case. Note that for other cases the algorithm may explore an exponential number of paths consuming some input prefix before finding the first input interpretation, hence its asymptotic cost is yet exponential.

Both acceptor algorithms can be further improved by representing stacks as pointers to the states of a trie of RTN state sequences, as explained in section 9.1 (p. 178).

Left recursion allows for a natural way of modelling many natural language structures (e.g.: see CFG in figure 12.2, p. 223), but the algorithm we have presented here is not able to process left-recursive RTNs. There exist algorithms that transform any left-recursive CFG into an equivalent non-left-recursive CFG; the classic algorithm can be found in Aho et al. (1986, p. 176), and a more efficient algorithm in Moore (2000). Since RTNs and CFGs are equivalent formalisms, left-recursion removal is also possible for RTNs (see figures 12.8 and 12.9). In order to deal with left recursion, a RTN processing system may simply forbid left-recursive RTNs —the solution adopted by the Unitex system— or implement some algorithm for the removal of left recursion. However, we may be interested not only in the recognition of a sentence but in determining the sentence's structure (identifying the sentence's constituents and groupings), which might be coded within the RTN as a precise sequence of call transitions. In this case, if we transform the grammar in order to remove left recursion then we modify the sequence of subroutine calls
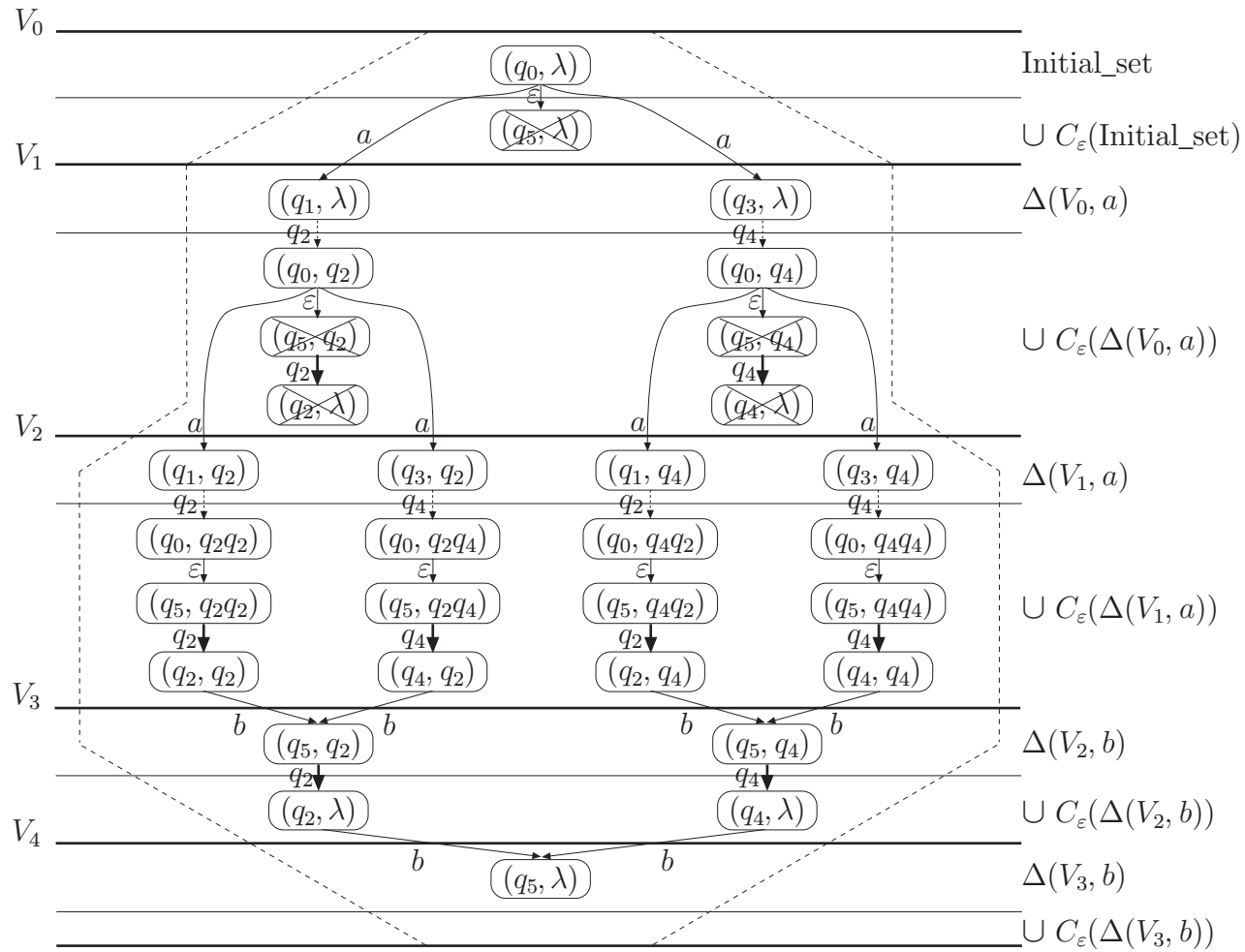
Figure 12.7: Execution trace of the RTN breadth-first acceptor algorithm for the RTN of figure 14.1 and input *aabb*. Solid, dotted and bold trace transitions correspond, respectively, to the exploration of the RTN explicit transitions, push transitions and pop transitions.

recognizing the sentences and therefore the resulting sentence structure. Another possibility is to extend RTNs for output generation —the object of the next chapter— and to code the sentence structures as output tags inserted in the right places, for instance XML tags bounding each sentence constituent. In that case, transforming the RTN structure does not modify the resulting sentence structure as long as both machines are equivalent. This is analogous to the elimination of left recursion from syntax-directed translation schemes described in Aho et al. (1986, chap. 2, p. 25).[6] In section 12.10 we present a more efficient algorithm of application of RTNs which is also able to process left-recursive RTNs, saving the hassle of left-recursion detection and removal.

12.8 Flattening

Flattening is a possible transformation to perform on a RTN in order to accelerate its application. This operation is already implemented in the Unitex system (Paumier, 2008).

Definition 216 (Flattening)**.** *Flattening a RTN A consists in replacing every call transition $t = (q_s, Q_c, q_t)$ in A by an exclusive copy of A's Q_c-submachine, as follows:*

- *remove t,*

- *for each state q of the Q_c-submachine, create a new state r, and make r an acceptor state if so it is q,*

- *for each transition (q'_s, ξ, q'_t) within the Q_c-submachine, add transition (r'_s, ξ, r'_t), with r'_s and r'_t the previously created states corresponding to states q_s and q_t,*

- *for each state $q_c \in Q_c$, add transition (q_s, ε, q_c), and*

- *for each acceptance state q_f of the Q_c-submachine, add transition (q_f, ε, q_t).*

Call transitions of added copies of submachines are to be recursively replaced as previously described.

---

[6]Syntax-directed translation schemes are CFGs extended with some kind of output generation mechanism that recreates the syntactic structure of the sentences they are applied to.

$$\begin{array}{lll}
\text{VP} \to \text{VP PP} & \text{VP} \to \text{V NP VP}' & \text{VP}' \to \text{PP VP}' \\
\text{VP} \to \text{V NP} & \text{VP} \to \text{V NP} & \text{VP}' \to \text{PP} \\
& & \\
\text{NP} \to \text{NP PP} & \text{NP} \to \text{DET N NP}' & \text{VP}' \to \text{PP NP}' \\
\text{NP} \to \text{DET N} & \text{NP} \to \text{DET N} & \text{NP}' \to \text{PP}
\end{array}$$

$$\Longrightarrow$$

Figure 12.8: Left-recursion of CFG of figure 12.2 (p. 223) can be removed by replacing the production rules at the left by those at the right; however, the grammar's structure is modified and new "artificial" symbols VP′ and NP′ are added.

(a) Left-recursive VP

(b) Left-recursive NP

(c) Non-left-recursive VP

(d) Non-left-recursive NP

(e) VP_prime

(f) NP_prime

Figure 12.9: Unitex graphs equivalent to those of figure 12.8: graphs (a) and (b) correspond to the productions at the left, and graphs (c), (d), (e) and (f) correspond to the productions at the right.

As we can see, flattening transforms the RTN into a FSA, thus allowing for a full determinization of the RTN. However, this process is not applicable to RTNs with recursive calls since the replacement of these calls by their corresponding Q_c-submachines would never end. However, the maximum number of recursive realizations of calls for a given set of input sequences is to be finite since input sequences are to be finite. The Unitex system allows for setting up a maximum number of recursive replacements in order to perform, at least, a partial flattening of recursive RTNs; recursive calls beyond this level are not replaced, hence the resulting RTNs accept the same languages. It must be taken into account that flattening a RTN with ambiguous and recursive calls increases its size exponentially w.r.t. the number of recursive replacements to perform. The number of recursive replacements is to be set to the greatest number lower than or equal to the maximum expected number of recursive realizations of calls such that the resulting RTN is small enough to be handled. The MovistarBot grammar is not recursive, hence every call can be removed by flattening it, but the number of states and transitions is increased by a factor of 4.1 and 10.1, respectively.

12.9 Determinization

When applied to RTNs, the generic determinization algorithm seen in section 8.5 (p. 166) not only performs a determinization but also a flattening; therefore, it can only be applied to non-recursive RTNs. However, we are also interested in applying recursive RTNs; as for FSTBOs (section 10.7, p. 199), we pseudo-determinize recursive RTNs by determinizing their underlying FSAs. If the RTN is to be also flattened, better results can be obtained by flattening first the RTN and then pseudo-determinizing it.

Definition 217 (Underlying FSA). *Let $A = (Q, \Sigma, \delta, Q_I, F)$ be a RTN, we define its underlying FSA as $(Q, \Sigma \cup \mathcal{P}(Q), \delta, Q_I, F)$, that is, RTN input symbols and RTN subinitial SSs become FSA input symbols.*

Note that, since call transitions are interpreted as consuming transitions, determinizing a RTN's underlying FSA does not pseudo-determinize the called submachines. Algorithm 8.2 *fsm_recognize_every_symbol* (p. 173) needs to be modified so that when σ is a subinitial SS Q_c, algorithm 8.1 *fsm_determinize* (p. 172) is executed again on RTN A but taking Q_c as initial SS, if this has not already been done. In other words, the different

submachines are to be separately pseudo-determinized, assuming that they are disjoint. Algorithm 8.1 *fsm_ determinize* is to take RTN A as a global argument for each execution, and it is to take two new optional arguments:

- the SS Q_c to be taken as initial SS, with Q_I as default value for the first execution of the algorithm, and

- a global map ζ_I of subinitial SSs $Q_c \in \mathcal{P}(Q)$ to subinitial states $r_c \in Q'$, taking the empty map as default value.

Note that, since the RTN is being treated as its underlying FSA, ESs in the algorithm are simple RTN states, namely: $X_I = Q_I$, $X_F = F$, $x_t \in Q$ and $V_s, V_t \in \mathcal{P}(Q)$.

In algorithm 8.1 *fsm_ determinize*, state $r_t \in Q'$ is created before the `while` loop as the initial state of A'. In the RTN version, this is to be done only for the first algorithm execution. Moreover, map $\zeta_I(Q_c) = r_t$ is to be added for every execution. The presence of this map is to be checked in algorithm 8.2 *fsm_ recognize_ every_ symbol* so that determinization of the Q_c-submachine is not started multiple times. Additionally, the creation of call transitions of machine A' is to be given a special treatment: the original algorithm would create a call transition (r_s, Q_c, r_c), where Q_c is some subinitial SS of A, but transition $(r_s, \zeta_I(Q_c), r_c)$ is to be created instead, where $\zeta_I(Q_c)$ is the subinitial state of A' corresponding to Q_c. At this point, it is sure that $\zeta_I(Q_c)$ is already defined since algorithm 8.2 *fsm_ recognize_ every- _ symbol* has been previously called, and this algorithm triggers the determinization of every Q_c-submachine which has not already been triggered, for every call transition having as source any of the states of A which have been condensed into state r_s of A'. Finally, an infinite loop due to left-recursive calls is not possible since call transitions are treated as consuming transitions: computing the subinitial state r_c corresponding to a subinitial SS Q_c implies to compute the ε-closure of Q_c, which does not traverse call transitions. Furthermore, mapping $\zeta_I(Q_c) = r_c$ is immediately defined afterwards, which prevents from initiating the determinization of the Q_c-submachine more than one time.

12.10 Earley-like processing

Finite-state automata can give a more compact representation of a set of sequences by factoring out common prefixes and suffixes of the accepted

sequences. RTNs can also factor infixes by defining one subautomaton for each repeated infix, and by using transitions calling the set of initial states of the corresponding subautomaton each time the infix is to be recognized. However, it is up to the parsing algorithm to detect that the same set of initial states is being called from multiple points of the grammar for the same input point so that the call is processed only once; for instance, in RTN of figure 12.6 (p. 237) both calls to $\{q_0\}$ could be computed only once (per recursion level). Inspired by Earley's (1970) CFG parser, we show here a modified and more efficient version of the base acceptor algorithm 7.5 for FSMs (p. 153) which is able to process left-recursive RTNs (see figure 12.11) without falling into an infinite loop, and which factors out the computation of infix calls of parallel explorations of the RTN. Our algorithm differs from the Earley-like parser for RTNs given by Woods (1969) in that

- it is based on a FSA-like definition of RTNs rather than on a CFG-like one,

- RTNs with ε-moves are supported, and

- calls are performed towards subinitial SSs instead of single states, which facilitates the definition of the canonical reverse RTN (or the reverse exploration of a RTN)

We have already presented a version of this algorithm for RTNs with string output in Sastre and Forcada (2007, 2009). A brief description of the original Earley parser is given in appendix C (p. 411), and a brief comparative discussion w.r.t. other parsing algorithms has been given in section 1.4.6, p. 16.

We mainly modify the subroutine jump mechanism, which is a part of the ε-closure computation. We replace the use of a stack of return states by a more complex representation of the ESs and a chart storing every computed SESs V during each iteration of the algorithm. When a call transition to a SS Q_c is to be traversed, two kinds of ESs are generated: one *paused* and one or more *active* ESs:

- the paused ES represents a hypothetical return from the call that is not to be resumed until the call is completed, and

- the active ESs initialize the call from each called $q_c \in Q_c$ and the current input, if call to Q_c has not already been initialized at this input point.

Each call is computed only once for each paused ES waiting for its comple-
tion, and each time the call is completed the corresponding paused ESs are
resumed.

Definition 218 (Earley execution state). *ESs for Earley-like RTN processing
are quadruplets in $(Q \times (\mathcal{P}(Q) \cup \{\lambda\}) \times \mathcal{P}(Q) \times \mathbb{N})$, where quadruplets of
the form $(q_s, \{\lambda\}, Q_h, i)$ are called* active *ESs and quadruplets of the form
(q_r, Q_c, Q_h, i) are called* paused *ESs. In the quadruplets,*

- *the first term, q_s or q_r, is the current state of the ES: the source state
  for active ESs and the return-from-call state for paused ESs,*

- *the second term, Q_c or λ, is the called SS Q_c whose completion this
  paused ES is waiting for, or λ if this is an active ES,*

- *Q_h or hypothesis SS is the last called SS whose associated calls will be
  completed once an acceptance state is reached, and*

- *i is the number of consumed input symbols at the moment of initi-
  ating the last call to Q_h, that is, when generating the last active ES
  (q_s, λ, Q_s, i) from where this either active or paused ES is derived.*

ESs of the original Earley parser (the chart items) include a second index
j such that $\sigma_{i+1} \ldots \sigma_j$ is the input interval that has been consumed since the
initialization of the last call up to the ES, for an input sequence $\sigma_1 \ldots \sigma_l$.
Since ESs are grouped into SESs such that V_j contains every generated ES
after consuming j symbols, we retrieve j from the index of V_j rather than
explicitly representing it inside every ES.

Definition 219 (Earley Δ). *The Δ function for RTN Earley-like processing,
the equivalent to Earley's scanner, is a simple direct-derivation function on
SESs (definition 98, p. 137) with*

- *$x_s = (q_s, \lambda, Q_h, i)$,*

- *$x_t = (q_t, \lambda, Q_h, i)$, and*

- *$d = q_t \in \delta(q_s, \sigma)$.*

Notice that the Δ function does not apply to paused ESs: there is nothing
to do with paused ESs until the call they depend on is completed. Input

symbols are only to be consumed from active ESs, and paused ESs are to wait for call completions. Notice as well that neither the hypothesis state nor the input position i are modified: they remain the same until entering into or coming out from a call.

Definition 220 (Earley D). *The $D(V_k)$ function for RTN Earley-like processing is composed by 3 simple direct-derivation functions on SESs (definition 98, p. 137):*

- *the explicit ε-transition processor, $D_\varepsilon(V_k)$ with*

  - $x_s = (q_s, \lambda, Q_h, j)$,
  - $x_t = (q_t, \lambda, Q_h, j)$, *and*
  - $d = q_t \in \delta(q_s, \varepsilon)$,

- *the equivalent to Earley's predictor, $D_{\text{push}}(V_k)$ with*

  - $x_s = (q_s, \lambda, Q_h, j)$,
  - $x_t = (q_c, \lambda, Q_c, k)$ *or* $x_t = (q_r, Q_c, Q_h, j)$, *meaning that both target ESs are derived from x_s if p holds, and*
  - $d = q_r \in \delta(q_s, Q_c) \wedge q_c \in Q_c$, *and*

- *the equivalent to Earley's completer, $D_{\text{pop}}(V_k)$ with*

  - $x_s = (q_f, \lambda, Q_h, j)$,
  - $x_t = (q_r, \lambda, Q'_h, i)$, *and*
  - $d = q_f \in F \wedge (q_r, Q_h, Q'_h, i) \in V_j$,

  where D_{pop} is retroactive, *that is, if during the computation of $D_{\text{push}}(V_k)$ a paused ES (q_r, Q_c, Q_h, j) is added to V_k due to a call to a SS Q_c that has already been ε-completed,[7] its resumed ES (q_r, λ, Q_h, j) is to be retroactively added to V_k as well.*

Retroactive ε-completion is discussed in more detail in the next section.

---

[7] Completions within the same SES V_k are only possible if no input is consumed during the whole call.

Definition 221 (Earley initial and acceptance SESs). *Given the sets of initial and acceptance states of a RTN, Q_I and F, its initial and acceptance SESs for Earley-like processing are $(Q_I \times \{\lambda\} \times \{Q_I\} \times \{0\})$, the ESs starting a call to any initial state before consuming any input symbol, and $(F \times \{\lambda\} \times \{Q_I\} \times \{0\})$, the ESs from where those initial calls would pop, respectively.*

Definition 222 (Earley execution machine). *The Earley execution machine of a RTN is a FPRTN. Its definition and the construction of the execution substructures for a particular input sequence will be described in chapter 15.*

For the Earley case, it does not suffice to replace call, push and pop transitions by pure ε-transitions; consider an ε-path having several consecutive and deletable calls to the same subinitial SS Q_c (see RTN of figure 12.10); during the computation of an ε-closure involving this path, call to Q_c is computed only once and therefore a unique structure resolving this call is built in the execution machine: the execution machine still needs call transitions in order to be able to return to the right state once the call is completed. Moreover, the call may be completed through different paths that do not necessarily consume the same amount of input symbols, resulting in multiple return states corresponding to different input points. FPRTNs perform an additional test in order to forbid pop transitions that bring the machine to return states corresponding to different input points than those of the acceptor states that precede them. More details will be given in chapter 15.

Definition 223 (Earley L). *Following definition 107 (p. 143), we define $L(A)$ —the language of a RTN A— through Earley-like processing as*

$$L(A) = \{w \in \Sigma^* : \Delta^*((Q_I \times \{\lambda\} \times \{Q_I\} \times \{0\}), w) \cap$$
$$(F \times \{\lambda\} \times \{Q_I\} \times \{0\}) \neq \emptyset\}. \quad (12.3)$$

12.11   Earley acceptor algorithm

Algorithm 12.2 *rtn_ earley_ recognize_ string* is a sequence acceptor implementing predicate $w \in L(A)$ through Earley-like processing (definition 223, p. 246). It uses algorithm 12.3 *rtn_ earley_ recognize_ symbol* for computing the Earley-like Δ function (definition 219), the equivalent to Earley's scanner, and algorithm 12.4 *rtn_ earley_ interlaced_ eclosure* for computing

the Earley-like ε-closure (generic FSM ε-closure in definition 100, p. 138, using the Earley-like D function in definition 220, p. 245), which includes both Earley's predictor (push transition processor) and completer (pop transition processor). Moreover, it includes an ε-transition processor for explicit ε-transition support as well as an ε-completer for handling deletable calls; both components are missing in the original Earley parser since:

- it does not support CFGs with either directly or indirectly deletable non-terminals, and

- the empty symbol is used only for the definition of directly deletable non-terminals (e.g.: $A \to \varepsilon$).[8]

Notice that the main difference w.r.t. the FSM acceptor, algorithm 7.5 (p. 153), is the way in which the ε-closure is computed. Finally, *add_ enqueue_ es* and *unconditionally_ add_ enqueue_ es* are the small routines seen in sections 7.8 (p. 148) and 7.9 (p. 152) for conditionally or unconditionally adding an ES to a SES.

Following the predictor-completer mechanism, every call started at a SES V_i to the same SS is computed only once. Without the possibility of ε-completing a call, calls started in V_i are completed during the computation of V_{i+1} or later, and therefore after every paused ES is added to V_i. Each time the call is completed, every paused ES in V_i depending on the call is searched in order to be resumed. If calls can be ε-completed then they can be started and completed during the computation of the same SES; therefore, paused ESs depending on the call might be added to the SES *after* the call is completed, and therefore remain paused. In order to avoid this, ε-completed calls must be marked in order to retroactively resume subsequent paused ESs. Algorithm *rtn_ earley_ interlaced_ eclosure* builds a set T containing the called subinitial SSs Q_c that have been ε-completed during the computation of the ε-closure of the SES V_k.[9] The ε-completer inside the completer adds Q_c to T for each ES (q_s, λ, Q_c, i) that triggers the call completion in V_k with $i = k$: since at V_i we have consumed i symbols and ES (q_s, λ, Q_c, i) indicates that the call started when i symbols where consumed, the call is being completed without input consumption. The ε-completer inside the

---

[8]A non-directly deletable non-terminal B can still be indirectly deletable: the grammar productions may allow for rewriting B as a directly deletable non-terminal, which in turn can be rewritten as the empty symbol.

[9]In practice we add the pointer to the set object representing Q_c.

predictor immediately resumes every paused ES (q_r, Q_c, Q_h, i) added to V_i so that $Q_c \in T$.

A discussion on extending Earley's CFG parser for supporting CFGs with deletable non-terminals can be found in Aycock and Horspool (2002), as well as an example of execution illustrating the problem. In the paper, a list of deletable non-terminals is to be previously built so that calls to such non-terminals are immediately completed. In our case, we have followed a different approach since the algorithm we give here is to be further extended for output generation, which will require an efficient procedure for the computation of such outputs rather than simply completing the deletable calls prematurely. We give in figure 12.10 an equivalent example to that given by Aycock and Horspool (2002) in order to illustrate the problem for the case of RTNs and how we have solved it. An example implying a left-recursive RTN equivalent to the one for CFGs in appendix C (p. 411) is shown in figure 12.11. Functions in the rightmost column of execution traces —except function $deletable(Q_c)$— represent the derivation mechanism that has been followed in order to produce the ES in the same line, where the arguments are the index of the ESs from where this ES has been derived:

- $recognize(i, \sigma)$: derived from ES i by taking a transition consuming input symbol σ,

- $\varepsilon\text{-}transition(i)$: derived from ES i by taking an ε-transition,

- $call(i)$ and $pause(i)$: the active and paused ESs, respectively, derived from ES i by taking a call transition, and

- $resume(i, j)$ and $\varepsilon\text{-}resume(i, j)$: derived by resuming paused ES i due to reaching ES j which triggers the call completion, the former by the completer and the latter by the ε-completer (inside the predictor).

Function $deletable(Q_c)$ accompanies function $resume(i, j)$ and indicates that, upon resuming ES i, the ε-completer (inside the completer) has detected that call to SS Q_h is deletable. Notice that when deriving an ES that is already present in the corresponding SES, there is no line added to the trace and therefore the derivation mechanism for that ES does not appear in the trace; for instance, in figure 12.11 a call is performed from the initial ES in 1 which produces the paused ES in 2 and an active ES that is already present in 1.

A graphical representation of the execution trace of the Earley-like acceptor algorithm for RTN of figure 12.6 and input $aabb$ —the same case seen

---

Algorithm 12.2 rtn_earley_recognize_string($\sigma_1 \ldots \sigma_l$) $\quad \triangleright \sigma_1 \ldots \sigma_l \in L,$
def. (223)

Input: $\sigma_1 \ldots \sigma_l$, an input string of length l
Output: r, a Boolean indicating whether the input string belongs to L
 1: allocate_memory_for_chart(V^{l+1})
 2: $V_0 \leftarrow \emptyset$
 3: $E \leftarrow \emptyset$
 4: **for each** $(q_c \in Q_I)$ **do**
 5:     unconditionally_add_enqueue_es($V_0, (q_c, \lambda, Q_I, 0)$)
 6: **end for**
 7: $V \leftarrow$ rtn_earley_interlaced_eclosure($V^{l+1}, E, 0$)
 8: $k \leftarrow 0$
 9: **while** $V_k \neq \emptyset \wedge k < l$ **do**
10:     $V_{k+1} \leftarrow$ rtn_earley_recognize_symbol(V_k, E, σ_{k+1})
11:     $k \leftarrow k + 1$
12:     rtn_earley_interlaced_eclosure(V^{l+1}, E, k)
13: **end while**
14: $r \leftarrow$ false
15: **for each** $(q_s, \lambda, Q_I, 0) \in V_k$ **do**
16:     $r \leftarrow r \vee q_s \in F$
17: **end for**

Algorithm 12.3 rtn_earley_recognize_symbol(V, E, σ) $\quad \triangleright \Delta(V, \sigma),$
def. (219)

Input: V, a SES
         E, the empty queue of unexplored ESs
         σ, the input symbol to recognize
Output: W, the set of reachable states from V by consuming σ
Output: E after enqueuing the ESs of W
 1: $W \leftarrow \emptyset$
 2: **for each** $(q_s, \lambda, Q_h, i) \in V$ **do**
 3:     **for each** $q_t \in \delta(q_s, \sigma)$ **do**
 4:         add_enqueue_es($W, E, (q_t, \lambda, Q_h, i)$)
 5:     **end for**
 6: **end for**

Algorithm 12.4 rtn_earley_interlaced_eclosure(V^{l+1}, E, k) $\triangleright C_\varepsilon(V_k)$

Input: V^{l+1}, the chart

      E, the queue of unexplored ESs containing every ES in V_k

      k, the index of the SES, V_k, whose ε-closure is to be computed

Output: V^{l+1} after adding to V_k its ε-closure

      E after emptying it

 1: $T \leftarrow \emptyset$

 2: **while** $E \neq \emptyset$ **do**

 3:    $(q_s, \lambda, Q_h, j) \leftarrow \text{dequeue}(E)$

                                       $\triangleright$ EXPLICIT ε-TRANSITIONS

 4:    **for each** $q_t \in \delta(q, \varepsilon)$ **do**

 5:        add_enqueue_es($V_k, E, (q_t, \lambda, Q_h, j)$)

 6:    **end for**

                                             $\triangleright$ PREDICTOR

 7:    **for each** $(q_r, Q_c) : q_r \in \delta(q_s, Q_c)$ **do**

 8:        **if** add($V_k, (q_r, Q_c, Q_h, j)$) **then**

 9:            **if** $Q_c \notin T$ **then**

10:                **for each** $q_c \in Q_c$ **do**

11:                    add_enqueue_es($V_k, E, (q_c, \lambda, Q_c, k)$)

12:                **end for**

                                        $\triangleright \varepsilon$-COMPLETER

13:            **else**

14:                add_enqueue_es($V_k, E, (q_r, \lambda, Q_h, j)$)

15:            **end if**

16:        **end if**

17:    **end for**

                                           $\triangleright$ COMPLETER

18:    **if** $q_s \in F$ **then**

19:        **for each** $(q_r, Q_h, Q'_h, i) \in V_j$ **do**

20:            add_enqueue_es($V_k, E, (q_r, \lambda, Q'_h, i)$)

                                        $\triangleright \varepsilon$-COMPLETER

21:            **if** $i = k$ **then**

22:                add(T, Q_h)

23:            **end if**

24:        **end for**

25:    **end if**

26: **end while**

V_0		
1 :	$(q_0, \lambda, \{q_0\}, 0)$	initial ES
2 :	$(q_1, \{q_5\}, \{q_0\}, 0)$	pause(1)
3 :	$(q_5, \lambda, \{q_5\}, 0)$	call(1)
4 :	$(q_6, \{q_7\}, \{q_5\}, 0)$	pause(3)
5 :	$(q_7, \lambda, \{q_7\}, 0)$	call(3)
6 :	$(q_8, \lambda, \{q_7\}, 0)$	ε-transition(5)
7 :	$(q_6, \lambda, \{q_5\}, 0)$	resume(4, 6); deletable($\{q_7\}$)
8 :	$(q_1, \lambda, \{q_0\}, 0)$	resume(2, 7); deletable($\{q_5\}$)
9 :	$(q_2, \{q_5\}, \{q_0\}, 0)$	pause(8); call already in 3
10 :	$(q_2, \lambda, \{q_0\}, 0)$	ε-resume(7, 9)
11 :	$(q_3, \{q_5\}, \{q_0\}, 0)$	pause(10); call already in 3
12 :	$(q_3, \lambda, \{q_0\}, 0)$	ε-resume(7, 11)
13 :	$(q_4, \{q_5\}, \{q_0\}, 0)$	pause(12); call already in 3
14 :	$(q_4, \lambda, \{q_0\}, 0)$	ε-resume(7, 13)
V_1		
15 :	$(q_6, \lambda, \{q_5\}, 0)$	recognize(3, a)
16 :	$(q_1, \lambda, \{q_0\}, 0)$	resume(2, 15)
17 :	$(q_2, \lambda, \{q_0\}, 0)$	resume(9, 15); resume(20, 27)
18 :	$(q_3, \lambda, \{q_0\}, 0)$	resume(11, 15); resume(22, 27)
19 :	$(q_4, \lambda, \{q_0\}, 0)$	resume(13, 15); resume(23, 27); acceptance ES
20 :	$(q_2, \{q_5\}, \{q_0\}, 0)$	pause(16)
21 :	$(q_5, \lambda, \{q_5\}, 1)$	call(16)
22 :	$(q_3, \{q_5\}, \{q_0\}, 0)$	pause(17); call already in 21
23 :	$(q_4, \{q_5\}, \{q_0\}, 0)$	pause(18); call already in 21
24 :	$(q_6, \{q_7\}, \{q_5\}, 1)$	pause(21)
25 :	$(q_7, \lambda, \{q_7\}, 1)$	call(21)
26 :	$(q_8, \lambda, \{q_7\}, 1)$	ε-transition(25)
27 :	$(q_6, \lambda, \{q_5\}, 1)$	resume(24, 26); deletable($\{q_7\}$)

Figure 12.10: RTN with deletable calls (left) and execution trace of algorithm 12.2 *rtn_ earley_ recognize_ string* for this RTN and input a (right); without the ε-completer, greyed ESs would be missing and the input rejected.

V_0		
$1:$ $(q_0, \lambda, \{q_0\}, 0)$	initial ES
$2:$ $(q_1, \{q_0\}, \{q_0\}, 0)$	pause(1); call already in 1
V_1		
$3:$ $(q_2, \lambda, \{q_0\}, 0)$	recognize$(1, b)$
$4:$ $(q_1, \lambda, \{q_0\}, 0)$	resume$(2, 3)$
V_2		
$5:$ $(q_2, \lambda, \{q_0\}, 0)$	recognize$(4, a)$
$6:$ $(q_1, \lambda, \{q_0\}, 0)$	resume$(2, 5)$
V_3		
$7:$ $(q_2, \lambda, \{q_0\}, 0)$	recognize$(6, a)$
$8:$ $(q_1, \lambda, \{q_0\}, 0)$	resume$(2, 7)$
$\vdots$		
V_l		
$2l+1:$ $(q_2, \lambda, \{q_0\}, 0)$	recognize$(2l, a)$; acceptance ES
$2l+2:$ $(q_1, \lambda, \{q_0\}, 0)$	resume$(2, 2l+1)$

Figure 12.11: Left-recursive RTN recognizing the language ba^n and execution trace of algorithm 12.2 *rtn_ earley-_recognize_ string* for this RTN and input ba^l.

in section 12.7— is shown in figure 12.12. Note that paused ESs are not represented as states of the trace but as labels of the push and pop transitions; their purpose is to annotate the required information upon starting a call for popping from it later. As for the base acceptor algorithm for RTNs, the number of parallel explorations is duplicated each time an a is consumed, but when performing the parallel calls to $\{q_0\}$ the algorithm creates a unique exploration path for the call. Once the call is completed, the two exploration paths are joined after consuming a b. The average number of parallel explorations is kept constant w.r.t. the length of input $a^n b^n$, thus the algorithm has a linear execution time for this RTN instead of exponential as for the base acceptor algorithm.

Note that the exponential explosion is due to multiple nesting levels of subgrammar calls combined with multiple interpretations within the called subgrammars: grammar of figure 12.6 produces 2^n different outputs with n as the number of nesting levels of call to $\{q_0\}$. The amount of such nesting levels that take place when parsing natural language sentences cannot be expected to be very high, since we find difficult to understand sentences involving a high number of ambiguous nesting levels and therefore we usually avoid to formulate such complex sentences. However, since the speedup due to factoring out the computation of subgrammar calls increases exponentially w.r.t. the number of nesting levels, and general natural language grammars involve ambiguous calls to heavy-weighted subgrammars, the performance gain can be expected to be considerable even for low nesting levels.

Since RTNs and CFGs are equivalent and the presented Earley-like algorithm for RTNs is an almost straightforward adaptation of Earley's CFG parser, we can expect the same asymptotic cost than that of the original parser: polynomial (n^3) in the worst case, but linear for many natural language input sentences and grammars. The algorithm cannot be further optimized using the trie string management seen in section 9.1 (p. 178) since it neither generates output nor relies on a stack of return states for recursive call management.

12.12 Earley-like determinization

Even though Earley-like processing avoids falling into infinite loops during the ε-closure computation, it does not prevent infinite loops when it is applied to the generic determinization algorithm (section 8.5, p. 166): Earley-like ESs

Figure 12.12: Execution trace of the RTN Earley-like acceptor —algorithm 12.2— for the RTN of figure 12.6 and input *aabb*. Thick dashed arrows link push transitions with their corresponding pop transitions. Paused ESs decorate push and pop transitions.

belong to sequences of SES $V_0 \ldots V_l$, where V_0 contains the ESs before consuming any input symbol, V_1 after consuming the first input symbol and so on, so even though two ESs belonging to two different SES V_i and V_j might be equal, they are not regarded as the same ES (an Earley-like determinization algorithm would require to extend ESs with another field storing the index of the SES they belong to). If the RTN can recognize input sequences of any length (it contains cycles that can be consecutively realized any number of times for some finite input), the determinization algorithm will try to generate an unbounded number of ESs, increasing the index of the SES they belong to each time an input symbol is consumed. This feature makes breadth-first processing a better choice for RTN determinization. Another possibility is to determinize only the paths consuming the first n input symbols, though we have not studied it. A similar approach called *prefix overlay transducers* (POTs) is presented in Marschner (2007) for RTNs with output. The main idea is that the search space of a RTN representing a natural language grammar gets reduced as sentence words are recognized, since the first words condition the following ones; therefore, if we are to partially determinize a RTN at the expense of increasing its size, we can expect a greater search space reduction by fully determinizing the paths recognizing the first n words than by flattening the first n recursive calls and then determinizing the RTN's underlying FSA.

Chapter 13

Recursive transition networks with blackboard output

We present here RTNBOs as a generalization of output generation with RTNs by combining the definitions and properties of FSTBOs (chapter 10) and RTNs (previous chapter). RTNBOs can be seen as an alternative definition of augmented transition networks (Woods, 1969): both formalisms extend RTNs with registers that store information generated during their application, and both formalisms define extra conditions to the traversal of transitions which depend on the values stored in the registers. RTNs with string output are presented in chapter 14 as a particular case of output generation. RTNs with composite output, weighted RTNs and RTNs with unification processes can be defined as special kinds of RTNBOs. The guidelines for obtaining such definitions will be given in chapters 17, 18 and 19, respectively.

Definition 224 (RTNBO). *A RTNBO $(Q, \Sigma, \Gamma, B, B_K, \delta, Q_I, F)$ is a FSTBO $(Q, \Sigma, \Gamma, B, B_K, \delta, Q_I, F)$ (definition 139, p. 185) extended with a subroutine jump mechanism, as for RTNs (definition 183, p. 221) w.r.t. FSAs (definition 128, p. 162): the set of transition labels Ξ takes its elements from the set $((\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\mathrm{id}_B\})) \cup \mathcal{P}(Q)$, where*

- *labels in $((\Sigma \cup \{\varepsilon\}) \times \Gamma) \cup \mathcal{P}(Q)$ are the same than for the case of FSTBOs and,*

- *labels in $\mathcal{P}(Q)$ represent subroutine jumps or calls to state sets, as for the case of RTNs.*

13.1 Transitions

In the definition of RTNBO, transitions that consume input and/or generate output are not allowed to modify the stack of return states and vice-versa. This way, transition definitions for the case of FSTBOs and RTNs can be reused for the case of RTNBOs. Transitions that either consume input or generate output are inherited from the FSTBO case, which are

- consuming transitions (definition 140, p. 186): $Q \times (\Sigma \times (\Gamma \cup \{\mathrm{id}_B\})) \times Q$,

- generating transitions (definition 141, p. 186): $Q \times ((\Sigma \cup \varepsilon) \times \Gamma) \times Q$,

- translating or substituting transitions (definition 142, p. 186): $Q \times (\Sigma \times \Gamma) \times Q$,

- deleting transitions (definition 143, p. 186): $Q \times (\Sigma \times \{\mathrm{id}_B\}) \times Q$ and

- inserting transitions (definition 145, p. 186): $Q \times (\{\varepsilon\} \times \Gamma) \times Q$.

Transitions that modify the stack are inherited from the RTN case, which are

- call transitions (definition 187, p. 222): $Q \times \mathcal{P}(Q) \times Q$,

- push transitions (definition 189, p. 222): $Q \times Q\downarrow \times Q$,

- pop transitions (definition 190, p. 223): $Q \times Q\uparrow \times Q$ and

- implicit ε-transitions (definition 186, p. 222): push or pop transitions.

Finally, transitions that neither consume input nor generate output nor modify the stack have the same form than FSTBO ε^2-transitions but fall into the category of RTN explicit ε-transitions (definition 185, p. 222):

- explicit ε^2-transitions (definition 146, p. 186): $Q \times (\{\varepsilon\} \times \{\mathrm{id}_B\}) \times Q$.

Definition 225 (ε^2-call). *We say a call to a subinitial SS Q_c is an ε^2-call or ε^2-realizable call iff it is realizable through an ε^2-path (see definition 149, p. 187).*

(a) NP

(b) N

Figure 13.1: Graphs of subfigures 12.4(b) and 12.4(e) (p. 224) after inserting XML output tags for marking the sentence compounds they recognize.

Figure 13.2: RTNBO fragments equivalent to the graphs of figure 13.1.

13.2 Graphical representation

The graphical representation of RTNBOs is a combination of the representations of FSMs (section 7.2, p. 124), FSTBOs (section 10.2, p. 187) and RTNs (section 12.2, p. 225), both for the classical representation as for the Unitex and Intex graphs. Figures 13.1 and 13.2 show some fragments of the graph and RTN shown in section 12.2 (p. 225) but extended with XML output tags in order to mark the recognized sentence compounds.

13.3 Sequences of transitions

Definitions given on the sequences of transitions of FSTBOs (section 10.3, p. 187) and RTNs (section 12.3, p. 225) also apply for the case of RTNBOs.

13.4   Behaviour

The definitions in this section correspond to the application of RTNBOs without Earley processing (as for the non-Earley application of RTNs described in section 12.5, p. 227). The Earley-like application of RTNBOs will be described in section 13.9.

Definition 226 (Execution state). *RTNBO ESs are triplets $(q, b, \pi) \in (Q \times B \times Q^*)$ where b is an output blackboard, $b_\emptyset$ being the empty blackboard, and π is a stack of return states, λ being the empty stack.*

Definition 227 (Illegal SES). *As for FSTBOs (definition 154, p. 189), the illegal SES of a RTNBO $(Q, \Sigma, \Gamma, B, B_K, \delta, Q_I, F)$ is $(Q \times B_K \times Q^*)$, that is, the set of all ES having a killing blackboard.*

Definition 228 (Δ). *The Δ function for RTNBOs is a simple direct-derivation function on SESs (definition 98, p. 137) with*

- $x_s = (q_s, b_s, \pi)$,

- $x_t = (q_t, b_t, \pi)$, *and*

- $d = q_s \in \delta(q_s, (\sigma, \gamma)) \wedge b_t = \gamma(b_s) \wedge b_t \notin B_K$.

The RTNBO Δ function behaves as the FSTBO Δ function (definition 155, p. 189): ESs are extended with a stack which is in fact not modified.

Definition 229 (D). *The D function for RTNBOs is composed by 3 simple direct-derivation functions on SESs (definition 98, p. 137), D_ε with*

- $x_s = (q_s, b_s, \pi)$,

- $x_t = (q_t, b_t, \pi)$, *and*

- $d = q_s \in \delta(q_s, (\varepsilon, \gamma)) \wedge b_t = \gamma(b_s) \wedge b_t \notin B_K$.

D_{push} *with*

- $x_s = (q_s, b, \pi)$,

- $x_t = (q_c, b, \pi q_t)$, *and*

- $d = q_t \in \delta(q_s, Q_c) \wedge q_c \in Q_c$,

and D_{pop} with

- $x_s = (q_f, b, \pi q_r)$,

- $x_t = (q_r, b, \pi)$, *and*

- $d = q_f \in F$.

As can be seen, D_ε is defined as function D for FSTBOs (definition 156, p. 189) but extended with an stack of return states which is left intact, and functions D_{push} and D_{pop} are defined as the ones for RTNs (see definition 211, p. 229) but extended with an output blackboard which is not modified.

Lemma 18 (Infinite ε-closure). *The ε-closure of a RTNBO SES V is infinite if there exists an ES (q, b, π) within V or ε-reachable from V such that q is traversed by a generating ε-cycle holding the conditions expressed in lemma 10 (p. 189) and/or q has an outgoing left-recursive call transition, as for RTNs (lemma 14, p. 230).*

Proof. The proof for the case involving generating ε-cycles can be obtained by extending FSTBOs ESs of proof of lemma 10 (p. 189) with a stack of return states π that does not change. The case involving left-recursive calls can be obtained by extending RTN ESs of proof of lemma 14 (p. 230) with a non-killing output blackboard that does not change. Since both cases lead to infinite ε-closures, the combination of both defines several ε-paths adding an infinite SES to the ε-closure, and therefore leading to infinite ε-closures as well. □

Lemma 19 (Finite ε-closure). *Under conditions other than those expressed in the previous lemma, the ε-closure of a RTNBO SES is finite.*

Proof. This proof is also a mixture of the analogous proofs for FSTBOs (proof of lemma 11, p. 191) and RTNs (proof of lemma 15, p. 230). If we only consider paths that do not modify the stack of return states, by extending the proof for FSTBOs with stacks that do not change we see that every ε-path not containing a generating ε-cycle such as the described in the proof lead to finite ε-closures. If we only consider paths that do modify the stack of return states or do not modify it but do not generate output, by extending the proof for RTNs with non-killing blackboards that do not change we see that every ε-path not containing left-recursive calls lead to finite ε-closures

as well. Finally, a path being composed by the concatenation of a finite sequence of paths of both kinds will lead to the finite union of the finite SES for each individual path, thus also leading to a finite ε-closure. □

Theorem 20. *The ε-closure is finite for non-left-recursive RTNBOs which do not contain generating ε-cycles such as the ones described in lemma 10 (p. 189).*

As already mentioned, such generating ε-cycles do not make sense in natural language grammars since they allow for infinite translations of finite input sequences (e.g.: finite sentences with infinite parse trees). Therefore, forbidding such cycles does not limit the natural language grammars that can be represented but ensures that the execution of the algorithms of application of RTNBOs will end.

Definition 230 (Initial and acceptance SESs). *Given the sets of initial and acceptance states of a RTNBO, Q_I and F, its initial and acceptance SESs are $(Q_I \times \{b_\emptyset\} \times \{\lambda\})$ and $(F \times B \times \{\lambda\})$, respectively.*

Definition 231 (τ). *We define $\tau(A)$, the language of translations of a RTNBO A, as the set of input/output pairs $(w, b) \in (\Sigma^* \times B)$ such that w is recognized and translated into b by A, that is, the set of input/output pairs such that the whole consumption of w reaches at least one acceptance ES from at least one initial ES through a path that generates b:*

$$\tau(A) = \{(w, b) : (q_f, b, \pi) \in \Delta^*((Q_I \times \{b_\emptyset\} \times \{\lambda\}), w) \cap (F \times B \times \{\lambda\})\}. \quad (13.1)$$

Definition 232 (ω). *We define $\omega(A, w)$, the translations of a word w for a RTNBO A, as the set of output sequences $b \in B$ such that (w, b) belongs to the translations of A:*

$$\omega(A, w) = \{b : (w, b) \in \tau(A)\}, \quad (13.2)$$

with $\tau(A)$ of the previous definition.

13.5   Translating a string

Algorithm 13.1 *rtnbo_ translate_ string* performs a breadth-first application of a RTNBO to an input string in order to obtain its set of translations. It is an almost straightforward adaptation of the breadth-first FSTBO translator (algorithm 10.1, p. 197) but with the following differences:

- the initial set of ESs has empty stacks of return states added to each ES,

- the use of the Δ and ε-closure functions adapted for RTNBOs,

- in the last loop, we extract the output blackboards of the ESs of the last V_i that have both an empty stack of return states and an acceptance state.[1]

Algorithm *rtnbo_ translate_ symbol* can be easily deduced from algorithm 10.2 *fstbo_ translate_ symbol* (p. 198) by extending ESs with a stack that does not change, and algorithm *rtnbo_ interlaced_ eclosure* can be easily deduced from algorithm 12.1 *rtn_ interlaced_ eclosure* by extending ESs with an output blackboard that

- might be modified for the case of explicit ε-transitions, as for consuming transitions in algorithm 10.2 *fstbo_ translate_ symbol*, and

- is not modified for the other cases (push and pop transitions).

Algorithm 10.5 (p. 199), the depth-first translator algorithm for FSTBOs, can yet be used for RTNBOs by simply replacing the implementation of the Δ and D functions for the case of RTNBOs.

As for the RTN breadth-first and depth-first acceptor algorithms (section 12.7, p. 235), these algorithms can be further improved by representing the stacks of return states as pointers to the nodes of a trie (see section 9.1, p. 178). As for the case of FSTBOs, blackboard fields containing data sequences may also be represented as pointers to trie nodes.

13.6 Flattening

There is no difference between flattening a RTNBO and flattening a RTN (section 12.8, p. 239) since this process applies only to call transitions, which are defined as for RTNs, and submachines are to be copied as is, that is, without interpreting their content.

---

[1]As for FSTBOs, it is not necessary to check whether the blackboards belong to B_K or not since, by definition, every ES in V_i is legal.

Algorithm 13.1 rtnbo_translate_string($\sigma_1 \ldots \sigma_l$) $\triangleright \, \omega(A, \sigma_1 \ldots \sigma_l)$, eq. (13.2)

Input: $\sigma_1 \ldots \sigma_l$, an input string of length l
Output: T, the translations of $\sigma_1 \ldots \sigma_l$
  1: $V \leftarrow \emptyset$
  2: $E \leftarrow \emptyset$
  3: **for each** $q_c \in Q_I$ **do**
  4:       unconditionally_add_enqueue_es($V, E, (q_c, b_\emptyset, \lambda)$)
  5: **end for**
  6: rtnbo_interlaced_eclosure(V, E)
  7: $i \leftarrow 0$
  8: **while** $E \neq \emptyset \wedge i < l$ **do**
  9:       $V_i \leftarrow$ rtnbo_translate_symbol(V, E, σ_{i+1})
 10:       $i \leftarrow i + 1$
 11:       rtnbo_interlaced_eclosure(V, E)
 12: **end while**
 13: $T \leftarrow \emptyset$
 14: **for each** $(q, b, \lambda) \in V : q \in F$ **do**
 15:       add(T, b)
 16: **end for**

13.7 Determinization

This section is a combination of the determinization sections for FSTBOs (section 10.7, p. 199) and RTNs (section 12.9, p. 241); we are interested in determinizing the RTNBO regarding it as its underlying FSA.

Definition 233 (Underlying FSA)**.** *Let $A = (Q, \Sigma, \Gamma, B, B_K, \delta, Q_I, F)$ be a RTNBO, we define its underlying FSA as $(Q, (((\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})) - \{(\varepsilon, \varepsilon)\}) \cup \mathcal{P}(Q), \delta, Q_I, F)$ with $(\varepsilon, \varepsilon)$ as the empty symbol, that is, RTNBO input/output pairs and RTNBO subinitial SSs become FSA input symbols except for $(\varepsilon, \varepsilon)$, which becomes the empty symbol.*

13.8 Blackboard set processing

We present here blackboard set processing (BSP) of RTNBOs as an extension of the FSTBO case (section 10.9, p. 205). As for FSTBOs, we traverse the RTNBO as a RTN, and we dinamically build a map ζ_B associating each RTN ES with the set of blackboards (SB) that can be generated by reaching the ES from an initial ES through any path. When deriving an ES x_t from an ES x_s we must make sure that $\zeta_B(x_s)$ is completely built so that every blackboard to be generated by this derivation is added to $\zeta_B(x_t)$. ESs derived by consuming i symbols are reached before the ones derived by consuming j symbols, for $0 \leq i < j$, and therefore the Δ function respects this restriction. We only require to pay special attention to the way in which the ε-closure is computed: ESs must be ε-derived by following a topological sort of the ε-closure-substructures of A''. The relation between the cycles in A and the cycles in A'' is not so straightforward as for the FSTBO case due to the presence of a stack inside RTN ESs.

Definition 234 (Z_B)**.** *Given a RTNBO $(Q, \Sigma, \Gamma, B, B_K, \delta, Q_I, F)$, we define Z_B as the set of every partial map ζ_B of RTN ESs $Q \times Q^*$ to SBs in $\mathcal{P}(B)$.*

Definition 235 (BSP SES)**.** *We define the equivalent BSP SES V_B of a RTNBO SES V as a pair (V', ζ_B) where $V' \subseteq Q \times Q^*$ is a RTN SES and $\zeta_B \in Z_B$ is a function mapping states to SBs such that*

$$V_B = (V', \zeta_B) : V' = \{(q, \pi) : (q, b, \pi) \in V\} \wedge \zeta_B((q, \pi)) = \{b : (q, b, \pi) \in V\},$$
$$(13.3)$$

which is equivalent to say that

$$V_B = (V', \zeta_B) : \bigcup_{(q,\pi) \in V'} \{q\} \times \zeta_B((q,\pi)) \times \{\pi\} = V. \qquad (13.4)$$

The definition of γ on SBs does not change w.r.t. the FSTBO case (definition 172, p. 206).

Definition 236 (BSP Δ). *We redefine the RTNBO Δ function for blackboard set processing as follows:*

$$\Delta : (\mathcal{P}(Q \times Q^*) \times Z_B) \times \Sigma \to (\mathcal{P}(Q \times Q^*) \times Z_B),$$

such that

$$\Delta((V, \zeta_B), \sigma) = (V', \zeta_B') : V' = \{x : \zeta_B'(x) \neq \emptyset\} \wedge$$
$$\zeta_B'((q_t, \pi)) = \bigcup_{\gamma : q_t \in \delta(q_s, (\sigma, \gamma)) \wedge (q_s, \pi) \in V} \gamma(\zeta_B((q_s, \pi))), \quad (13.5)$$

As for the FSTBO case, the existence of a topological sort for the computation of the ε-closures depends on the possibility of removing the ε-cycles of the RTNBO. As for FSTBOs, RTNBO ε-cycles with generation are simply forbidden in order to avoid infinite ε-closures. RTNBO ε^2-cycle removal is slightly different from the FSTBO case due to the presence of call transitions.

Theorem 21 (ε^2-cycle removal). *For every non-left-recursive RTNBO with ε^2-cycles not involving deletable calls there exists an equivalent non-left-recursive RTNBO without ε^2-cycles which can be obtained by determinizing the RTNBO regarding it as its underlying FSA.*

Proof. RTNBO ε^2-cycles can be divided into two classes: call ε^2-cycles and non-call ε^2-cycles. Call ε^2-cycles correspond to left-recursive calls, which are forbidden since they lead to infinite ε-closures. Non-call ε^2-cycles can be divided again into two classes: the ones that involve deletable calls and the ones that do not. ε^2-cycles belonging to the former class are forbidden, and the ones belonging to the latter class are the same than the ε^2-cycles found in the FSTBO case, and therefore can be removed by determinizing the RTNBO as its underlying FSA. $\qquad \square$

RTNBO ε^2-cycles with deletable calls are not removed when determinizing its underlying FSA: such ε^2-cycles are reduced to an ε^2-cycle composed only by deletable calls, where the subinitial SS of each call is replaced by a single state that is both subinitial and final. However, such cycles do not contribute anything to the grammar description. We simply forbid their presence in order to support BSP.

Theorem 22 (Existence of a topological sort). *Considering lemma 1 (p. 131) and the previous theorem, for every non-left-recursive RTNBO without ε-cycles involving deletable calls and/or output generation there exists an equivalent RTNBO A such that, given A' the RTN obtained from A after removing its output alphabet and transition outputs, there exists at least one topological sort for every ε-closure-substructure of $\mathcal{X}(A')$.*

As for the FSTBO case (definition 174, p. 207), we define function D for BSP for a single source RTN ES and its associated SB instead of a RTN SES and a mapping of RTN ESs to SBs since we iteratively compute the ε-closure ES by ES, following a topological sort.

Definition 237 (BSP D). *We define function D for RTNBO BSP as follows:*

$$D : ((Q \times Q^*) \times Z_B) \to (\mathcal{P}(Q \times Q^*) \times Z_B),$$

such that

$$
\begin{aligned}
D(x_s, B_s) = (V', \zeta'_B) : V' = V_\varepsilon \cup V_{\text{push}} \cup V_{\text{pop}} \wedge \\
\forall x_t \in V'[\zeta'_B(x_t) = \zeta_{B_\varepsilon}(x_t) \cup \zeta_{B_{\text{push}}}(x_t) \cup \zeta_{B_{\text{pop}}}(x_t)] \wedge \\
(V_\varepsilon, \zeta_{B_\varepsilon}) = D_\varepsilon(x_s, B_s) \wedge (V_{\text{push}}, \zeta_{B_{\text{push}}}) = D_{\text{push}}(x_s, B_s) \wedge \\
(V_{\text{pop}}, \zeta_{B_{\text{pop}}}) = D_{\text{pop}}(x_s, B_s) \quad (13.6)
\end{aligned}
$$

$$
\begin{aligned}
D_\varepsilon((q_s, \pi), B_s) = (V', \zeta'_B) : V' = \{x : \zeta'_B(x) \neq \emptyset\} \wedge \\
\zeta'_B((q_t, \pi)) = \bigcup_{\gamma : q_t \in \delta(q_s, (\varepsilon, \gamma))} \gamma(B_s) \quad (13.7)
\end{aligned}
$$

$$
\begin{aligned}
D_{\text{push}}((q_s, \pi), B_s) = (V', \zeta'_B) : V' = \{(q_c, \pi q_t) : q_t \in \delta(q_s, Q_c)\} \wedge q_c \in Q_c \wedge \\
\forall x \in V'[\zeta'_B(x) = B_s] \quad (13.8)
\end{aligned}
$$

$$
D_{\text{pop}}((q_s, \pi q_r), B_s) = \begin{cases} (\{(q_r, \pi)\}, \zeta'_B) : \zeta'_B(q_r, \pi) = B_s & q_s \in F \\ (\emptyset, \zeta'_B) : \forall x[\zeta'_B(x) = \emptyset] & q_s \notin F \end{cases} \quad (13.9)
$$

Definition 238 (BSP ε-closure). *The definition of BSP ε-closure is the same than for FSTBOs (definition 175, p. 208) but replacing FSTBOs by RTNBOs, FSAs by RTNs, FSA ESs by RTN ESs and the BSP D function for FSTBOs by the one for RTNBOs.*

The proof of equivalence between BSP and non-BSP ε-closures for RTNBO breadth-first processing is analogous to the one for the FSTBO case (proof of theorem 13, p. 208).

13.9   Earley-like processing

RTNBOs perform the recognition of an input sequence as RTNs do, but also compute its associated output blackboards as result of applying to the empty blackboard $b_\emptyset$ the composition of the sequence of output functions found during the traversal of the RTNBO. Earley-like RTN processing (section 12.10, p. 242) factors out the computation of parallel calls to the same state by pausing every calling ES, then starting a new and single processing for the call and finally resuming the paused ESs each time the call is completed. For the case of RTNBOs we factor out as well the computation of the output blackboard of common calls. However, it is necessary to define a blackboard composition operator so that, for each call completion, the blackboards computed during the call can be combined with the blackboards of the paused ESs to be resumed in order to proceed with the exploration of the RTNBO:

Definition 239 (Blackboard composition operator). *Given a RTNBO (Q, Σ, Γ, B, B_K, δ, Q_I, F), its blackboard composition operator $\circ$ is a binary function on blackboards*

$$\circ : B \times B \to B$$

such that given two blackboards

$$b = (\gamma_m \circ \gamma_{m-1} \circ \ldots \circ \gamma_0)(b_\emptyset) \tag{13.10}$$

and

$$b' = (\gamma'_n \circ \gamma'_{n-1} \circ \ldots \circ \gamma'_0)(b_\emptyset) \tag{13.11}$$

it holds that

$$b \circ b' = (\gamma'_n \circ \gamma'_{n-1} \circ \ldots \circ \gamma'_0 \circ \gamma_m \circ \gamma_{m-1} \circ \ldots \circ \gamma_0)(b_\emptyset).^2 \tag{13.12}$$

---

[2]Recall that the notation of function composition reverses the function specification w.r.t. the order in which they are applied, that is, $(\gamma_m \circ \gamma_{m-1} \circ \ldots \circ \gamma_0)(b_\emptyset) = \gamma_m(\gamma_{m-1}(\ldots(\gamma_0(b_\emptyset))\ldots))$

In other words, the blackboard composition operator enables to separately compute partial results corresponding to consecutive segments of the sequence of output functions and then combine those partial results as if the sequence of output functions was applied sequentially to the empty blackboard. The concrete definition of the blackboard composition operator depends on the kind of concrete machine.

Lemma 20 (Associative blackboard composition operator). *Let $\bullet$ be a binary operator on blackboards; if every output function of a RTNBO A is of the form $\gamma_{b_r}(b_l) = b_l \bullet b_r$,[3] and $\bullet$ is associative, then $\bullet$ is the blackboard composition operator of A.*

Proof. Let output functions of a RTNBO B be defined as in the lemma, then it holds that

$$\gamma_{b_r}(b_\emptyset) = b_r \quad \text{and} \tag{13.13}$$
$$\mathrm{id}_B(b_l) = b_l,[4] \tag{13.14}$$

which imply that

$$b_\emptyset \bullet b_r = b_r \quad \text{and} \tag{13.15}$$
$$b_l \bullet b_\emptyset = b_l, \tag{13.16}$$

that is, $b_\emptyset$ is the identity element w.r.t. $\bullet$. Let blackboards b and b' be defined as

$$b = (\gamma_{b_m} \circ \gamma_{b_{m-1}} \circ \ldots \circ \gamma_{b_0})(b_\emptyset) \quad \text{and} \tag{13.17}$$
$$b' = (\gamma_{b'_n} \circ \gamma_{b'_{n-1}} \circ \ldots \circ \gamma_{b'_0})(b_\emptyset), \tag{13.18}$$

then it holds that

$$b = b_\emptyset \bullet b_0 \bullet \ldots \bullet b_{m-1} \bullet b_m = b_0 \bullet \ldots \bullet b_{m-1} \bullet b_m \quad \text{and} \tag{13.19}$$
$$b' = b_\emptyset \bullet b_0 \bullet \ldots \bullet b'_{n-1} \bullet b'_n = b'_0 \bullet \ldots \bullet b'_{n-1} \bullet b'_n. \tag{13.20}$$

If $\bullet$ is associative, then it holds that

$$b \bullet b' = b_\emptyset \bullet b_0 \bullet \ldots \bullet b_{m-1} \bullet b_m \bullet b'_0 \bullet \ldots \bullet b'_{n-1} \bullet b'_n \tag{13.21}$$
$$= (\gamma_{b'_n} \circ \gamma_{b'n-1} \circ \ldots \circ \gamma_{b'_0} \circ \gamma_{b_m} \circ \gamma_{b_{m-1}} \circ \ldots \circ \gamma_{b_0})(b_\emptyset), \tag{13.22}$$

---

[3]Blackboards b_l and b_r stand for left and right operands, respectively.

[4]Recall that $b_\emptyset$ stands for the empty blackboard and that id_B stands for the identity function on blackboards.

and therefore $\bullet$ is the blackboard composition operator of A (definition 239).

$\square$

The previous lemma is applied to the definition of the blackboard composition operator for every particular output case treated in this dissertation, namely weights (the object of chapter 18) and feature structures built by means of unification processes (the object of chapter 19). RTNBOs whose output blackboards do not allow for the definition of a blackboard composition operator can still be applied efficiently if such blackboards are weighted and only the top-ranked blackboard is to be returned. More details will be given in chapter 18.

Definition 240 (Earley execution state). *ESs for Earley-like RTNBO processing are ESs for Earley-like RTN processing (definition 207, p. 227) augmented with a blackboard in B representing the output generated up to the ES, in particular structures in $(Q \times B \times (\mathcal{P}(Q) \cup \{\lambda\}) \times \mathcal{P}(Q) \times \mathbb{N})$.*

Definition 241 (Earley Δ). *The Δ function for RTNBO Earley-like processing, the equivalent to Earley's scanner, is a simple direct-derivation function on SESs (definition 98, p. 137) with*

- $x_s = (q_s, b_s, \lambda, Q_h, i)$,

- $x_t = (q_t, b_t, \lambda, Q_h, i)$, *and*

- $d = q_t \in \delta(q_s, (\sigma, \gamma)) \wedge b_t = \gamma(b_s) \wedge b_t \notin B_K$.

The RTNBO Earley-like Δ function behaves as the RTN Earley-like Δ function (definition 219, p. 244) extended with blackboard processing analogously to the way in which the FSA Δ function (definition 132, p. 163) is extended with blackboard output for FSTBO processing (definition 155, p. 189).

Definition 242 (Earley D). *The $D(V_k)$ function for RTNBO Earley-like processing is composed by 3 simple direct-derivation functions on SESs (definition 98, p. 137):*

- *the explicit ε-transition processor, $D_\varepsilon(V_k)$ with*

  - $x_s = (q_s, b_s, \lambda, Q_h, i)$,
  - $x_t = (q_t, b_t, \lambda, Q_h, i)$, *and*
  - $d = q_t \in \delta(q_s, (\varepsilon, \gamma)) \wedge b_t = \gamma(b_s) \wedge b_t \notin B_K$,

- *the equivalent to Earley's predictor, D_{push} with*

  - *$x_s = (q_s, b_s, \lambda, Q_h, j)$,*
  - *$x_t = (q_c, b_\emptyset, \lambda, Q_c, k)$ or $x_t = (q_r, b_s, Q_c, Q_h, j)$, meaning that both target ESs are derived from x_s if p holds, and*
  - *$d = q_r \in \delta(q_s, Q_c) \wedge q_c \in Q_c$, and*

- *the equivalent to Earley's completer, D_{pop} with*

  - *$x_s = (q_f, b_f, \lambda, Q_h, j)$,*
  - *$x_t = (q_r, b_r, \lambda, Q'_h, i)$, and*
  - *$d = q_f \in F \wedge (q_r, b_s, Q_h, Q'_h, i) \in V_j \wedge b_r = b_s \circ b_f \wedge b_r \notin B_K$,*

  where $\circ$ is the blackboard composition operator and D_{pop} is retroactive, as for RTN Earley-like processing (definition 220, p. 245).

Definition 243 (Earley initial and acceptance SESs). *Given the sets of initial and acceptance states of a RTNBO, Q_I and F, its initial and acceptance SESs for Earley-like processing are $(Q_I \times \{b_\emptyset\} \times \{\lambda\} \times \{Q_I\} \times \{0\})$, the ESs starting a call to any initial state before consuming any input symbol or generating any output, and $(F \times B \times \{\lambda\} \times \{Q_I\} \times \{0\})$, the ESs from where those initial calls would pop, respectively.*

Definition 244 (Earley τ). *We define $\tau(A)$ —the language of translations of a RTNBO A— through Earley-like processing as*

$$\tau(A) = \{(w, b) : (q_f, b, \lambda, Q_I, 0) \in \Delta^*((Q_I \times \{b_\emptyset\} \times \{\lambda\} \times \{Q_I\} \times \{0\}), w) \cap$$
$$(F \times B \times \{\lambda\} \times \{Q_I\} \times \{0\})\}. \quad (13.23)$$

Definition 245 (Earley ω). *We define $\omega(A, w)$ —the translations of a word w for a RTNBO A— through Earley-like processing as*

$$\omega(A, w) = \{b : (w, b) \in \tau(A)\}, \quad (13.24)$$

with $\tau(A)$ of the previous definition.

13.10 Earley translator algorithm

Algorithm 13.2 *rtnbo_ earley_ translate_ string* is a sequence translator implementing the Earley-like version of $\omega(A, w)$ (definition 244, p. 271), which we have obtained by extending the corresponding Earley-like sequence acceptor algorithm for RTNs (algorithm 12.2, p. 249) with blackboard output. As for the acceptor algorithm, it uses algorithm 13.3 *rtnbo_ earley_ translate_ symbol* for computing the Earley-like Δ function (definition 241, p. 270), and algorithm 13.4 *rtnbo_ earley_ interlaced_ eclosure* for computing the Earley-like ε-closure (generic FSM ε-closure in definition 100, p. 138, using the Earley-like D function in definition 242, p. 270). Finally, *add_ enqueue-_ esbo* and *unconditionally_ add_ enqueue_ es* are the small routines seen in sections 10.6 (p. 196) and 7.9 (p. 152) for conditionally or unconditionally adding an ES to a SES.

The differences between the Earley-like acceptor algorithm (algorithm 12.2) and the translator version (algorithm 13.2) are enumerated below:

- ESs are extended with a blackboard element, which is $b_\emptyset$ for the initial ESs, so when arriving to the same state through different paths it is possible to have several ESs due to different blackboards,

- processing a transition with output requires to apply a γ function to the blackboard of the source ES,

- as for FSTBOs, illegal ESs (containing killing blackboards) are rejected,

- instead of a Boolean, the result of the algorithm is the set of blackboards (SB) containing every blackboard of every acceptance ES in the last V,

- given the set of paused ESs W_{push} of a SES V_k having called the same SS Q_c, and the set of ESs W_{pop} from where the call has been popped, the algorithm computes the composition of every pair of blackboards of every pair of ESs in $W_{\text{push}} \times W_{\text{pop}}$, which raises its asymptotic cost from polynomial to exponential, and

- the ε-closure algorithm requires to build the list of not only the ε-completed calls but their corresponding outputs as well, so the composition of blackboards can be performed when retroactively ε-completing a call inside the predictor.

Algorithm 13.2 rtnbo_earley_translate_string($\sigma_1 \ldots \sigma_l$) $\omega(A, \sigma_1 \ldots \sigma_l)$, eq. (13.24)

Input: $\sigma_1 \ldots \sigma_l$, an input string of length l
Output: T, the translations of $\sigma_1 \ldots \sigma_l$
 1: allocate_memory_for_chart(V^{l+1})
 2: $V_0 \leftarrow \emptyset$
 3: $E \leftarrow \emptyset$
 4: **for each** $(q_c \in Q_I)$ **do**
 5:     unconditionally_add_enqueue_es($V_0, E, (q_c, b_\emptyset, \lambda, Q_I, 0)$)
 6: **end for**
 7: $V_0 \leftarrow$ rtnbo_earley_interlaced_eclosure($V^{l+1}, E, 0$)
 8: $k \leftarrow 0$
 9: **while** $V_k \neq \emptyset \wedge k < l$ **do**
10:     $V_{k+1} \leftarrow$ rtnbo_earley_translate_symbol(V_k, E, σ_{k+1})
11:     $k \leftarrow k + 1$
12:     rtnbo_earley_interlaced_eclosure(V^{l+1}, E, k)
13: **end while**
14: $T \leftarrow \emptyset$
15: **for each** $(q_s, b_s, \lambda, Q_I, 0) \in V_k : q_s \in F$ **do**
16:     add(T, b_s)
17: **end for**

Algorithm 13.3 rtnbo_earley_translate_symbol(V, E, σ) $\qquad \triangleright \Delta(V, \sigma)$, def. (241)

Input: V, a SES
            E, the empty queue of unexplored ESs
            σ, the input symbol to translate
Output: W, the set of reachable ESs from V by consuming σ
            E after enqueuing the ESs of W
 1: $W \leftarrow \emptyset$
 2: **for each** $(q_s, b_s, \lambda, Q_h, j) \in V_k$ **do**
 3:     **for each** $(q_t, \gamma) : q_t \in \delta(q_s, (\sigma, \gamma))$ **do**
 4:         add_enqueue_esbo($W, E, (q_t, \gamma(b_s), \lambda, Q_h, j)$)
 5:     **end for**
 6: **end for**

Algorithm 13.4 rtnbo_earley_interlaced_eclosure(V^{l+1}, E, k)   ▷ $C_\varepsilon(V_k)$

Input: V^{l+1}, the chart

  E, the queue of unexplored ESs containing every ES in V_k

  k, the index of the SES whose ε-closure is to be computed

Output: V^{l+1} after adding to V_k its ε-closure

  E after emptying it

1: $T \leftarrow \emptyset$
2: **while** $E \neq \emptyset$ **do**
3:     $(q_s, b_s, \lambda, Q_h, j) \leftarrow$ dequeue(E)

                                  ▷ EXPLICIT ε-TRANSITIONS

4:     **for each** $(q_t, g) : q_t \in \delta(q_s, (\varepsilon, g))$ **do**
5:         add_enqueue_esbo($V_k, E, (q_t, \gamma(b_s), \lambda, Q_h, j)$)
6:     **end for**

                                          ▷ PREDICTOR

7:     **for each** $(q_r, Q_c) : q_r \in \delta(q_s, Q_c)$ **do**
8:         **if** add($V_k, (q_r, b_s, Q_c, Q_h, j)$) **then**
9:             **if** $\nexists b_c : (Q_c, b_c) \in T$ **then**
10:                 **for each** $q_c \in Q_c$ **do**
11:                     add_enqueue_esbo($V_k, E, (q_c, b_\emptyset, \lambda, Q_c, k)$)
12:                 **end for**

                                           ▷ ε-COMPLETER

13:             **else**
14:                 **for each** $b_c : (Q_c, b_c) \in T$ **do**
15:                     add_enqueue_esbo($V_k, E, (q_r, b_s \circ b_c, \lambda, Q_h, j)$)
16:                 **end for**
17:             **end if**
18:         **end if**
19:     **end for**

                                          ▷ COMPLETER

20:     **if** $q_s \in F$ **then**
21:         **for each** $(q_r, b_c, Q_h, Q'_h, i) \in V_j$ **do**
22:             add_enqueue_esbo($V_k, E, (q_r, b_s \circ b_c, \lambda, Q'_h, i)$)

$\triangleright \varepsilon$-COMPLETER

23:	**if** $i = k$ **then**
24:	$\text{add}(T, (Q_h, b_s))$
25:	**end if**
26:	**end for**
27:	**end if**
28:	**end while**

As for the RTN Earley-like case (section 12.11, p. 246), the optimization of sequence representation by means of tries (section 9.1, p. 178) cannot be applied here to the management of stacks of states since the Earley translator algorithm for RTNBOs (algorithm 13.2) does not use such stacks. It might be applicable to the representation of string-like outputs, though for Earley-like processing this modification might decrease efficiency rather than optimizing the algorithm since it falls into the not-so-efficient case described in section 9.3 (p. 183). A more detailed discussion will be given in section 14.7 (p. 289) for the case of RTNSOs.

13.11   Earley-like blackboard set processing

Analogous to breadth-first blackboard set processing (BSP) of RTNBOs (section 13.8, p. 265), we present here Earley-like BSP of RTNBOs as an extension of the FSTBO case (section 10.9, p. 205). The execution machines of the corresponding machines without output for both the FSTBO and RTNBO breadth-first cases (FSAs and RTNs, respectively) are FSAs. However, for the case of Earley-like RTN processing the resulting execution machine is an output FPRTN, a subclass of FPRTNs that we will study in chapter 16.[5] Defining a topological sort for output FPRTN substructures and finding the necessary conditions for its existence is not so straightforward as for FSAs due to the presence of call transitions and the particular way in which they are constructed. We just present here the equations for Earley-like BSP of

---

[5]To be exact, the Earley-like execution machine of a RTN is an "input" FPRTN: a machine built as for output FPRTNs but taking as transition labels the inputs of the RTN instead of the outputs of the original RTNBO. Moreover, output FPRTNs are built for recognizing only the translations generated by the RTNBO for a given input sequence, and the execution machine consumes every input sequence the RTN can consume. More details will be given in chapter 16.

RTNBOs, supposing that there exists such a topological sort for the ε-closure substructures of the corresponding output FPRTNs. An application of these equations will be given in section 16.3 (p. 329) for the generation of the language of an output FPRTN, along with a definition of topological sort for output FPRTN substructures and the necessary conditions for its existence.

Definition 246 (Z_B). *Given a RTNBO $(Q, \Sigma, \Gamma, B, B_K, \delta, Q_I, F)$, we define Z_B as the set of every partial map ζ_B of RTN Earley ESs $(Q \times B \times (\mathcal{P}(Q) \cup \{\lambda\}) \times \mathcal{P}(Q) \times \mathbb{N})$ to SBs in $\mathcal{P}(B)$.*

Definition 247 (Earley blackboard set processing). *We define the equivalent Earley BSP SES V_B of a RTNBO Earley SES V as follows:*

$$V_B = (V', \zeta_B) : V' = \{(q_s, \lambda, Q_h, i) : (q_s, b, \lambda, Q_h, i) \in V\} \cup \quad (13.25)$$
$$\{(q_s, Q_c, Q_h, i) : (q_s, b, Q_c, Q_h, i) \in V\} \wedge \quad (13.26)$$
$$\zeta_B(q_s, \lambda, Q_h, i) = \{b : (q_s, b, \lambda, Q_h, i) \in V\} \wedge \quad (13.27)$$
$$\zeta_B(q_s, Q_c, Q_h, i) = \{b : (q_s, b, Q_c, Q_h, i) \in V\}, \quad (13.28)$$

which is equivalent to say that

$$V_B = (V', \zeta_B) : (\bigcup_{(q_s, \lambda, Q_h, i) \in V'} \{q_s\} \times \zeta_B(q_s, \lambda, Q_h, i) \times \{\lambda\} \times \{Q_h\} \times \{i\}) \cup$$
$$(\bigcup_{(q_s, Q_c, Q_h, i) \in V'} \{q_s\} \times \zeta_B(q_s, Q_c, Q_h, i) \times \{Q_c\} \times \{Q_h\} \times \{i\}) = V. \quad (13.29)$$

The definition of γ on SBs (definition 172, p. 206) does not change.

Definition 248 (Earley BSP Δ). *We redefine the RTNBO Δ function for Earley-like BSP as follows:*

$$\Delta : (\mathcal{P}(Q \times \{\lambda\} \times \mathcal{P}(Q) \times \mathbb{N}) \times Z_B) \times \Sigma \to (\mathcal{P}(Q \times \{\lambda\} \times \mathcal{P}(Q) \times \mathbb{N}) \times Z_B),$$

such that

$$\Delta((V, \zeta_B), \sigma) = (V', \zeta_B') : V' = \{(q_t, \lambda, Q_h, i+1) : \zeta_B'(q_t, \lambda, Q_h, i+1) \neq \emptyset\} \wedge$$
$$\zeta_B'(q_t, \lambda, Q_h, i+1) = \bigcup_{\gamma : q_t \in \delta(q_s, (\sigma, \gamma)) \wedge (q_s, \lambda, Q_h, i) \in V} \gamma(\zeta_B(q_s, \lambda, Q_h, i)) \quad (13.30)$$

Definition 249 (Earley BSP D)**.** *The D function for BSP Earley RTNBO processing is defined as follows:*

$$D : ((Q \times Q^*) \times Z_B) \to (\mathcal{P}(Q \times Q^*) \times Z_B)$$

$$
\begin{aligned}
D((q_s, \lambda, q_h, j), B_s) = (V', \zeta'_B) : V' &= V_\varepsilon \cup V_{\text{push}} \cup V_{\text{pop}} \wedge \\
\zeta'_B(q_t, \pi) &= \zeta_{B_\varepsilon}(q_t, \pi) \cup \zeta_{B_{\text{push}}}(q_t, \pi) \cup \zeta_{B_{\text{pop}}}(q_t, \pi) \wedge \\
(V_\varepsilon, \zeta_{B_\varepsilon}) = D_\varepsilon((q_s, \pi), B_s) \wedge (V_{\text{push}}, \zeta_{B_{\text{push}}}) &= D_{\text{push}}((q_s, \pi), B_s) \wedge \\
(V_{\text{pop}}, \zeta_{B_{\text{pop}}}) &= D_{\text{pop}}((q_s, \pi), B_s) \quad (13.31)
\end{aligned}
$$

$$
\begin{aligned}
D_\varepsilon((q_s, \lambda, q_h, j), B_s) = (V', \zeta'_B) : V' &= \{(q_t, \lambda, q_h, j) : \zeta'_B(q_t, \lambda, q_h, j) \neq \emptyset\} \wedge \\
\zeta'_B(q_t, \lambda, q_h, j) &= \bigcup_{\gamma : q_t \in \delta(q_s, (\varepsilon, \gamma))} \gamma(B_s) \quad (13.32)
\end{aligned}
$$

$$
\begin{aligned}
D_{\text{push}}((q_s, \lambda, q_h, j), B_s) = (V', \zeta'_B) : V' &= \{(q_c, \lambda, q_c, k), (q_t, q_c, q_h, j) : \\
(q_s, \lambda, q_h, j) &\in V_k \wedge q_t \in \delta(q_s, q_c)\} \wedge \\
(\zeta'_B(q_c, \lambda, q_c, k) = \{b_\emptyset\} \wedge \zeta'_B(q_t, q_c, q_h, j) = B_s) &\iff \\
((q_s, \lambda, q_h, j) &\in V_k \wedge q_t \in \delta(q_s, q_c)) \quad (13.33)
\end{aligned}
$$

$$
D_{\text{pop}}((q_s, \lambda, q_h, j), B_s) =
\begin{cases}
(V', \zeta') : V' = \{(q_r, \lambda, q'_h, i) : \\
\quad (q_r, q_h, q'_h, i) \in V_j\} \wedge \zeta'(q_r, \pi) = B_s & q_s \in F \\
(\emptyset, \zeta') : \zeta'(q_s, \lambda, q_h, j) = \emptyset & q_s \notin F
\end{cases}
$$
$$(13.34)$$

Definition 250 (BSP ε-closure)**.** *The definition of BSP ε-closure is the same than for breadth-first BSP of RTNBOs (definition 238, p. 268) but replacing RTN ESs by RTN Earley ESs and the BSP breadth-first D function by the corresponding Earley one.*

The proof of equivalence between Earley-like BSP and non-BSP ε-closures for RTNBO Earley-like processing is analogous to the one for the FSTBO case (proof of theorem 13, p. 208).

Chapter 14

Recursive transition networks with string output

We present here RTNSOs as a special case of RTNBOs in the same way we have presented FSTSOs as a special case of FSTBOs in chapter 11. We have published brief descriptions of RTNSOs —as well as of the breadth-first and Earley-like algorithms of application of RTNSOs we describe here— in Sastre and Forcada (2007, 2009).

Definition 251 (RTNSO). *A RTNSO $(Q, \Sigma, \Gamma, \delta, Q_I, F)$ is a special type of FSM (definition 46, p. 121) with a stack, where the set of labels Ξ of the machine take its elements from $((\Sigma \cup \{\varepsilon\} \times (\Gamma \cup \{\varepsilon\})) \cup \mathcal{P}(Q)$, Σ is an input alphabet, Γ an output alphabet, ε the empty symbol and Q the finite SS of the RTNSO. RTNSOs can be seen as a special case of RTNBOs in the same way FSTSOs are a special case of FSTBOs (see definition 176, p. 212).*

14.1 Transitions

RTNSO transitions are a particular case of RTNBO transitions (section 13.1, p. 258) as FSTSO transitions are a particular case of FSTBO transitions (see section 11.1, p. 212). Possible RTNSO transition types are:

- consuming transitions: $Q \times (\Sigma \times (\Gamma \cup \{\varepsilon\})) \times Q$,

- generating transitions: $Q \times ((\Sigma \cup \{\varepsilon\}) \times \Gamma) \times Q$,

- translating or substituting transitions: $Q \times (\Sigma \times \Gamma) \times Q$,

279

- deleting transitions: $Q \times (\Sigma \times \{\varepsilon\}) \times Q$,

- inserting transitions: $Q \times (\{\varepsilon\} \times \Gamma) \times Q$,

- call transitions: $Q \times \mathcal{P}(Q) \times Q$,

- push transitions: $Q \times Q{\downarrow} \times Q$,[1]

- pop transitions: $Q \times Q{\uparrow} \times Q$,[2]

- implicit ε-transitions: push or pop transitions, and

- explicit ε^2-transitions: $Q \times (\{\varepsilon\} \times \{\varepsilon\}) \times Q$.

14.2   Sequences of transitions

Analogously, RTNSO paths are a particular case of RTNBO paths. Every definition in section 13.3 (p. 259) is inherited by replacing RTNBO transitions by their corresponding RTNSO transitions.

14.3   Behaviour

Definition 252 (Execution state). *RTNSO ESs are triplets $(q, z, \pi) \in (Q \times \Gamma^* \times Q^*)$ where z is a sequence of output symbols, ε being the empty output, and π is a stack of return states, being λ the empty stack.*

Definition 253 (Δ). *The Δ function for RTNBOs is a simple direct-derivation function on SESs (definition 98, p. 137) with*

- $x_s = (q_s, z, \pi)$,

- $x_t = (q_t, zg, \pi)$, *and*

- $d = q_s \in \delta(q_s, (\sigma, g))$,

where $g \in \Gamma \cup \{\varepsilon\}$. The RTNSO Δ function behaves as the FSTSO Δ function but extending its processing with stacks of return states that are in fact left untouched.

---

[1]Recall that $q_c{\downarrow}$ represents to push state q_c onto the stack.
[2]Recall that $q_r{\uparrow}$ represents to pop state q_r out of the stack.

Definition 254 (*D*). *The D function for RTNSOs is composed by 3 simple direct-derivation functions on SESs (definition 98, p. 137), D_ε with*

- $x_s = (q_s, z, \pi)$,

- $x_t = (q_t, zg, \pi)$, *and*

- $d = q_s \in \delta(q_s, (\varepsilon, g))$,

where $g \in \Gamma \cup \{\varepsilon\}$, D_{push} with

- $x_s = (q_s, z, \pi)$,

- $x_t = (q_c, z, \pi q_t)$, *and*

- $d = q_t \in \delta(q_s, Q_c) \wedge q_c \in Q_c$,

and D_{pop} with

- $x_s = (q_f, z, \pi q_r)$,

- $x_t = (q_r, z, \pi)$, *and*

- $d = q_f \in F$.

Lemma 21 (Infinite ε-closure). *The ε-closure of a RTNSO SES V is infinite if there exists an ε-reachable ES (q, z, π) such that q has an outgoing left-recursive call transition and/or is traversed by a generating ε-cycle.*

Proof. Since the RTNSO ε-closure function is a particular case of the RTNBO ε-closure function, this proof is a particular case of proof of lemma 18 (p. 261) for RTNBOs. The ESs derived during the computation of the ε-closure are not explicitly required to be legal since RTNSOs do not define killing blackboards. Additionally, non-identity output functions are neither explicitly required to always generate new blackboards since, for the case of string output, this is always true: $zg \neq z$ iff $g \neq \varepsilon$. □

Lemma 22 (Finite ε-closure). *For conditions other than those expressed in the previous lemma, the ε-closure of a RTNSO SES is finite.*

Proof. Since the RTNSO ε-closure function is a particular case of the RTNBO ε-closure function, this proof is a particular case of proof of lemma 19 (p. 261) for RTNBOs. □

Theorem 23. *The ε-closure is always finite for non-left-recursive RTNSOs without generating ε-cycles.*

Definition 255 (Initial and acceptance SESs). *Given the sets of initial and acceptance states of a RTNSO, Q_I and F, its initial and acceptance SESs are $(Q_I \times \{\varepsilon\} \times \{\lambda\})$ and $(F \times \Gamma^* \times \{\lambda\})$, respectively.*

Definition 256 (τ). *We define $\tau(A)$, the language of translations of a RTNSO A, as the set of input/output pairs $(w, z) \in (\Sigma^* \times \Gamma^*)$ such that w is recognized and translated into z by A, that is, the set of input/output pairs such that the whole consumption of w reaches at least one acceptance ES from at least one initial ES through a path that generates z:*

$$\tau(A) = \{(w, z) : (q_f, z, \pi) \in \Delta^*((Q_I \times \{\varepsilon\} \times \{\lambda\}), w) \cap (F \times \Gamma^* \times \{\lambda\})\}. \quad (14.1)$$

Definition 257 (ω). *We define $\omega(A, w)$, the translations of a word w for a RTNSO A, as the set of output sequences $z \in \Gamma^*$ such that (w, z) belongs to the translations of A:*

$$\omega(A, w) = \{z : (w, z) \in \tau(A)\}, \quad (14.2)$$

with $\tau(A)$ of the previous definition.

14.4   Translating a string

Algorithm 14.1 *rtnso_translate_string* is a specialization of algorithm 13.1 *rtnbo_translate_string* (p. 264) for the computation of the set of string translations for a given RTNSO an input sequence. Algorithms for computing the Δ and ε-closure functions can be easily derived from their RTNBO counterparts (see section 13.5, p. 262). Notice that, since RTNSOs do not define killing strings, routine *add_enqueue_es* (section 7.8, p. 148) can be used instead of routine *add_enqueue_esbo* (section 10.6, p. 196) in order to add derived ESs to the current SES: both routines perform the same operation but the former one does not verify whether the derived ESs contain killing blackboards or not.

   Figure 14.2 is a graphical representation of the execution trace of algorithm 14.1 *rtnso_translate_string*, for RTNSO of figure 14.1 and input *aabb*. This RTNSO cannot be determinized as for the RTN case in section 12.7

Algorithm 14.1 rtnso_translate_string($\sigma_1 \ldots \sigma_l$) $\triangleright \omega(A, \sigma_1 \ldots \sigma_l)$,
eq. (14.2)

Input: $\sigma_1 \ldots \sigma_l$, an input string of length l
Output: T, the translations of $\sigma_1 \ldots \sigma_l$
1: $V \leftarrow \emptyset$
2: $E \leftarrow \emptyset$
3: **for each** $q_c \in Q_I$ **do**
4:     unconditionally_add_enqueue_es($V, E, (q_c, \varepsilon, \{\lambda\})$)
5: **end for**
6: rtnso_interlaced_eclosure(V, E)
7: $i \leftarrow i + 1$
8: **while** $V_i \neq \emptyset \wedge i < l$ **do**
9:     $V \leftarrow$ rtnso_translate_symbol(V, E, σ_{i+1})
10:     $i \leftarrow i + 1$
11:     rtnso_interlaced_eclosure(V, E)
12: **end while**
13: $T \leftarrow \emptyset$
14: **for each** $(q, z, \pi) \in V$ **do**
15:     **if** $q \in F \wedge \pi = \lambda$ **then**
16:         add(T, z)
17:     **end if**
18: **end for**

Figure 14.1: Example of ambiguous RTNSO corresponding to RTN of figure 12.6 (p. 237) extended with string output; labels of the form $x : y$ represent an input/output pair (e.g.: $a : \{$ for transition $(q_0, (a, \{), q_1))$ and dashed transitions represent a call to the state specified by the label (e.g.: transition (q_1, q_0, q_2)). Input a can be interpreted as [or { and b as] or }.

(p. 235) since transitions labeled with the same input symbols define different outputs. As for the RTN case, the number of parallel parses is doubled each time an a is consumed, but is not reduced after consuming each b due to the different outputs of the ESs. The number of generated ESs is also exponential w.r.t. the length of input $a^n b^n$, as for the RTN case.

The algorithm can be further improved with the trie string management seen in section 9.1 (p. 178), which in this case may be applied to both the output strings and the stack of return states.

14.5 Language generation

In section 11.6 (p. 215) we described how to adapt an FSTSO translator algorithm in order to obtain an algorithm for the generation of the language of a FSA. We follow here an analogous procedure for the construction of an algorithm for the generation of the language a RTN, that is, by adapting a RTNSO translator algorithm.

Theorem 24 (Language generation). *Let $A = (Q, \Sigma, \delta, Q_I, F)$ be a RTN and $A' = (Q', \Sigma', \Gamma, \delta', Q'_I, F')$ a RTNSO such that*

- $Q' = Q$, $Q'_I = Q_I$, $F' = F$,

- $\Sigma' = \emptyset$,

- $\Gamma = \Sigma$,

- $q_t \in \delta'(q_s, (\varepsilon, \gamma))$ *iff* $q_t \in \delta(q_s, \sigma)$, *and*

Figure 14.2: Execution trace of the RTNSO breadth-first translator algorithm 14.1 for the ambiguous RTNSO of figure 14.1) and input *aabb*. Solid, dotted and bold trace transitions correspond, respectively, to the exploration of the RTN explicit transitions, push transitions and pop transitions.

- $q_t \in \delta'(q_s, Q_c)$ *iff* $q_t \in \delta(q_s, Q_c)$,

then it holds that

$$L(A) = \omega(A', \varepsilon) \tag{14.3}$$

Proof. The proof is analogous to the one for the FSA/FSTSO case (proof of theorem 15, p. 216). In this case, paths p_i and p'_i for $i = 0 \ldots l - 1$ may also contain push and pop transitions modifying a stack of return states. However, it still holds that p is an interpretation within A iff p' is an interpretation within A'. □

Algorithm 14.2 *rtn_ language* is an adaptation of the breadth-first translator algorithm 14.1 for the computation of the language of a RTN. As for the FSA/FSTSO case, the domain of application is given by the original algorithm, that is, the algorithm cannot compute the language of RTNs containing useful consuming cycles and/or useful left-recursive calls. As for the original algorithm, this algorithm can also be improved with the trie string management shown in section 9.1 (p. 178).

14.6   Earley-like processing

We adapt here the RTNBO Earley-like processing equations (section 13.9, p. 268) for the RTNSO case, that is, replacing blackboards with strings. We mainly remove the killing blackboard mechanism and define the blackboard composition operator as the string concatenation operator.

Definition 258 (String composition operator). *We define the blackboard composition operator (definition 239, p. 268) for the case of RTNSOs as the string concatenation operator since it is a particular case of lemma 20 (p. 269).*

Definition 259 (Earley execution state). *ESs for Earley-like RTNSO processing are ESs for Earley-like RTNBO processing (definition 226, p. 260) where the blackboards are strings in Γ^*, that is, structures in $(Q \times \Gamma^* \times (\mathcal{P}(Q) \cup \{\lambda\}) \times \mathcal{P}(Q) \times \mathbb{N}$.*

Definition 260 (Earley Δ). *The Δ function for RTNSO Earley-like processing, the equivalent to Earley's scanner, is a simple direct-derivation function on SESs (definition 98, p. 137) with*

Algorithm 14.2 rtn_language(A) $\triangleright$ $L(A)$, eq. (12.2)

Input: $A = (Q, \Sigma, \delta, Q_I, F)$, a RTN
Output: L, the language of A
1:   $V \leftarrow \emptyset$
2:   $E \leftarrow \emptyset$
3:   **for each** $q_c \in Q_I$ **do**
4:      unconditionally_add_enqueue_es($V, E, (q_c, \varepsilon, \{\lambda\})$)
5:   **end for**
6:   **while** $E \neq \emptyset$ **do**
7:      $(q_s, w, \pi) \leftarrow$ dequeue(E)
                                 $\triangleright$ EXPLICIT ε-TRANSITIONS
8:      **for each** $q_t \in \delta(q_s, \varepsilon)$ **do**
9:         add_enqueue_es($V, E, (q_t, w, \pi)$)
10:     **end for**
                                   $\triangleright$ CONSUMING TRANSITIONS
11:     **for each** $(q_t, \sigma) : q_t \in \delta(q_s, \sigma)$ **do**
12:       add_enqueue_es($V, E, (q_t, w\sigma, \pi)$)
13:     **end for**
                                   $\triangleright$ PUSH-TRANSITIONS
14:     **for each** $(q_r, Q_c) : q_r \in \delta(q_s, Q_c)$ **do**
15:       **for each** $q_c \in Q_c$ **do**
16:         add_enqueue_es($V, E, (q_c, w, \pi q_r)$)
17:       **end for**
18:     **end for**
                                   $\triangleright$ POP TRANSITIONS
19:     **if** $\pi = \pi' q_r \wedge q_s \in F$ **then**
20:       add_enqueue_es($V, E, (q_r, w, \pi')$)
21:     **end if**
22:   **end while**
23:   $L \leftarrow \emptyset$
24:   **for each** $(q, w, \lambda) \in V : q \in F$ **do**
25:     add(L, w)
26:   **end for**

- $x_s = (q_s, z, \lambda, Q_h, i)$,

- $x_t = (q_t, zg, \lambda, Q_h, i)$, *and*

- $d = q_t \in \delta(q_s, (\sigma, g))$,

where $g \in (\Gamma \cup \{\varepsilon\})$.

Definition 261 (Earley D). *The $D(V_k)$ function for RTNSO Earley-like processing is composed by 3 simple direct-derivation functions on SESs (definition 98, p. 137):*

- *the explicit ε-transition processor, $D_\varepsilon(V_k)$ with*

  - $x_s = (q_s, z, \lambda, Q_h, i)$,
  - $x_t = (q_t, zg, \lambda, Q_h, i)$, *and*
  - $d = q_t \in \delta(q_s, (\varepsilon, g))$,

  where $g \in (\Gamma \cup \{\varepsilon\})$,

- *the equivalent to Earley's predictor, D_{push} with*

  - $x_s = (q_s, z_s, \lambda, Q_h, j)$,
  - $x_t = (q_c, \varepsilon, \lambda, Q_c, k)$ *or* $x_t = (q_r, z_s, Q_c, Q_h, j)$, *meaning that both target ESs are derived from x_s if p holds, and*
  - $d = q_r \in \delta(q_s, Q_c) \wedge q_c \in Q_c$, *and*

- *the equivalent to Earley's completer, D_{pop} with*

  - $x_s = (q_f, z_f, \lambda, Q_h, j)$,
  - $x_t = (q_r, z_s z_f, \lambda, Q'_h, i)$, *and*
  - $d = q_f \in F \wedge (q_r, z_s, Q_h, Q'_h, i) \in V_j$,

  where D_{pop} is retroactive, *as for the RTN case definition 220 (p. 245).*

Definition 262 (Earley initial and acceptance SESs). *Given the sets of initial and acceptance states of a RTNSO, Q_I and F, its initial and acceptance SESs for Earley-like processing are $(Q_I \times \{\varepsilon\} \times \{\lambda\} \times \{Q_I\} \times \{0\})$, the ESs starting a call to any initial state before consuming any input symbol and generating any output, and $(F \times \Gamma^* \times \{\lambda\} \times \{Q_I\} \times \{0\})$, the ESs from where those initial calls would pop, respectively.*

Definition 263 (Earley τ). *Following definition [244](p. [271]), we define $\tau(A)$, the language of translations of a RTNSO A through Earley-like processing, as*

$$\tau(A) = \{(w, z) : (q_f, z, \lambda, Q_I, 0) \in \Delta^*((Q_I \times \{\varepsilon\} \times \{\lambda\} \times \{Q_I\} \times \{0\}), w) \cap$$
$$(F \times \Gamma^* \times \{\lambda\} \times \{Q_I\} \times \{0\}). \quad (14.4)$$

Definition 264 (ω). *We define $\omega(A, w)$, the translations of a word w for a RTNSO A through Earley-like processing, as*

$$\omega(A, w) = \{z : (w, z) \in \tau(A)\}, \quad (14.5)$$

with $\tau(A)$ of the previous definition.

14.7 Earley translator algorithm

Algorithm [14.3](#) *rtnso_ earley_ translate_ string* is a sequence translator implementing the Earley-like $\omega(A, w)$ function (definition [263]), which we have obtained by removing the killing blackboard mechanism of the Earley-like translator for RTNBOs (algorithm [13.2], p. [273]), and by replacing blackboard management by string management. Analogously to the RTNBO algorithm, it uses algorithm [14.4](#) *rtnso_ earley_ translate_ symbol* for computing the Earley-like Δ function (definition [260]) and algorithm [14.5](#) *rtnso-_ earley_ interlaced_ eclosure* for computing the Earley-like ε-closure (generic FSM ε-closure in definition [100], p. [138], using Earley-like D function in definition [261], p. [288]). Finally, the routines *add_ enqueue_ es* and *unconditionally-_ add_ enqueue_ es* seen in sections [7.8](#) (p. [148]) and [7.9](#) (p. [152]), respectively, are used for conditionally or unconditionally adding derived ESs to the current SES without checking for illegal strings since there are none defined. We have already presented the resulting algorithm in Sastre and Forcada (2007, 2009).

Figure [14.3](#) is a graphical representation of the execution trace of the Earley-like translator algorithm adapted for RTNSOs, for RTNSO of figure [14.1](#) and input *aabb*. As for the RTN case (without output generation, section [12.11](#), p. [246]), the number of parallel explorations is duplicated each time the ambiguous symbol a is to be translated but then the common call to SS $\{q_0\}$ is factored out, reducing again the number of parallel explorations to one. However, when completing a call to $\{q_0\}$ the two outputs generated

---

Algorithm 14.3 rtnso_earley_translate_string($\sigma_1 \ldots \sigma_l$) $\triangleright \omega(A, \sigma_1 \ldots \sigma_l)$,
eq. (14.5)

Input: $\sigma_1 \ldots \sigma_l$, an input string of length l
Output: T, the translations of $\sigma_1 \ldots \sigma_l$
 1: allocate_memory_for_chart(V^{l+1})
 2: $V_0 \leftarrow \emptyset$
 3: $E \leftarrow \emptyset$
 4: **for each** $(q_c \in Q_I)$ **do**
 5:     unconditionally_add_enqueue_es($V_0, E, (q_c, \varepsilon, \lambda, Q_I, 0)$)
 6: **end for**
 7: rtnso_earley_interlaced_eclosure($V^{l+1}, E, 0$)
 8: $k \leftarrow 0$
 9: **while** $V_k \neq \emptyset \wedge k < l$ **do**
10:     $V_{k+1} \leftarrow$ rtnso_earley_translate_symbol(V_k, E, σ_{k+1})
11:     $k \leftarrow k + 1$
12:     rtnso_earley_interlaced_eclosure(V^{l+1}, E, k)
13: **end while**
14: $T \leftarrow \emptyset$
15: **for each** $(q_s, z, \lambda, Q_I, 0) \in V_k : q_s \in F$ **do**
16:     add(T, z)
17: **end for**

Algorithm 14.4 rtnso_earley_translate_symbol(V, E, σ) $\triangleright \Delta(V, \sigma)$,
def. (260)

Input: V, a SES
            E, the empty queue of unexplored ESs
            σ, the input symbol to translate
Output: W, the set of reachable ESs from V by consuming σ
              E after enqueuing the ESs of W
 1: $W \leftarrow \emptyset$
 2: **for each** $(q_s, z, \lambda, Q_h, j) \in V$ **do**
 3:     **for each** $(q_t, g) : q_t \in \delta(q_s, (\sigma, g))$ **do**
 4:         add_enqueue_es($W, E, (q_t, zg, \lambda, Q_h, j)$)
 5:     **end for**
 6: **end for**

Algorithm 14.5 rtnso_earley_interlaced_eclosure(V^{l+1}, E, k) $\quad \triangleright C_\varepsilon(V_k)$

Input: V^{l+1}, the chart

$\qquad E$, the queue of unexplored ESs containing every ES in V_k

$\qquad k$, the index of the SES, V_k, whose ε-closure is to be computed

Output: V^{l+1} after adding to V_k its ε-closure

$\qquad E$ after emptying it

1: $T \leftarrow \emptyset$
2: **while** $E \neq \emptyset$ **do**
3: $\quad (q_s, z, \lambda, Q_h, j) \leftarrow \text{dequeue}(E)$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \triangleright$ EXPLICIT ε-TRANSITIONS
4: $\quad$ **for each** $(q_t, g) : q_t \in \delta(q_s, (\varepsilon, g))$ **do**
5: $\qquad \text{add\_enqueue\_es}(V_k, E, (q_t, zg, \lambda, Q_h, j))$
6: $\quad$ **end for**
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \triangleright$ PREDICTOR
7: $\quad$ **for each** $(q_r, Q_c) : q_r \in \delta(q_s, Q_c)$ **do**
8: $\qquad$ **if** $\text{add}(V_k, (q_r, z, Q_c, Q_h, j))$ **then**
9: $\qquad\quad$ **if** $\nexists z' : (Q_c, z') \in T$ **then**
10: $\qquad\qquad$ **for each** $q_c \in Q_c$ **do**
11: $\qquad\qquad\quad \text{add\_enqueue\_es}(V_k, E, (q_c, \varepsilon, \lambda, Q_c, k))$
12: $\qquad\qquad$ **end for**
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \triangleright \varepsilon$-COMPLETER
13: $\qquad\quad$ **else**
14: $\qquad\qquad$ **for each** $z' : (Q_c, z') \in T$ **do**
15: $\qquad\qquad\quad \text{add\_enqueue\_es}(V_k, E, (q_r, zz', \lambda, Q_h, j))$
16: $\qquad\qquad$ **end for**
17: $\qquad\quad$ **end if**
18: $\qquad$ **end if**
19: $\quad$ **end for**
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \triangleright$ COMPLETER
20: $\quad$ **if** $q_s \in F$ **then**
21: $\qquad$ **for each** $(q_r, z', Q_h, Q'_h, i) \in V_j$ **do**
22: $\qquad\quad \text{add\_enqueue\_es}(V_k, E, (q_r, zz', \lambda, Q'_h, i))$

$\triangleright \varepsilon$-COMPLETER

23:             **if** $i = k$ **then**
24:                 $\mathrm{add}(T, (Q_h, z'))$
25:             **end if**
26:         **end for**
27:     **end if**
28: **end while**

during the call are combined with the outputs generated before the call, resulting in an exponential explosion of ESs to compute. The algorithm saves an exponential number of steps w.r.t. the breadth-first RTNSO algorithm in section 14.4 (p. 282) by factoring out push transitions, but still suffers from an exponential explosion of ESs to compute upon popping; since the number of outputs increases exponentially w.r.t. the input length, it is inevitable to perform an exponential number of operations if one is to generate the effective list of translations. In the next chapter we show how to delay as much as possible the exponential explosion by building the translation set as some kind of finite-state machine recognizing the language of translations but factoring out the common output subsequences. Such exponential explosion is possible in natural language grammars, for instance due to unresolved prepositional phrase attachments. An example illustrating such situation has been given in section 1.5.4 (p. 19).

Online applications such as the MovistarBot (section 1.2, p. 6) are not necessarily required to reduce the average cost of analyzing user input sentences but to ensure that each sentence is processed in a short time interval, since users are not willing to wait more than a few seconds for an answer. As well, we must guarantee that the server running the NLP software will not be collapsed due to a particular user sentence having a specially high parsing cost. Since nowadays computers have at least two processing units, another possible solution is to concurrently execute two different parsing algorithms (e.g.: a top-down depth-first parser and an Earley-like parser) and to retrieve the result from the one that finishes first, aborting the execution of the other algorithm. This way we can obtain a "combined" algorithm that both minimizes the average and maximum execution times.

Figure 14.4 is another example of execution of the Earley-like algorithm for RTNSOs equivalent to the example of figure 12.10 (p. 251) for RTNs. The transition consuming a now also generates output symbol A, and the ε-

Figure 14.3: Execution trace of the RTNSO Earley-like translator algorithm 14.3 for the ambiguous RTNSO of figure 14.1 and input *aabb*. Thick dashed arrows link push transitions with their corresponding pop transitions. Paused ESs decorate push and pop transitions.

transition generates now output symbol E. The four possible interpretations of a have a different associated output: $AEEE$, $EAEE$, $EEAE$ and $EEEA$, respectively. As we can see, the trace for this example contains more steps than its RTN equivalent due to the concatenation of different pairs of outputs upon call completions. As for the RTN case, without the ε-completer it would have not been possible to derive the acceptance ESs, and therefore the algorithm would have returned an empty set of translations of a.

Figure 14.5 is the example equivalent to that of figure 12.11 (p. 252) but for a simple left-recursive RTNSO —instead of a simple left-recursive RTN— that translates ba^n into BA^n. In this case, the trace is the same but adding the increasing output to the ESs.

Finally, optimizing the RTNSO Earley-like translator algorithm by means of trie string management (section 9.1, p. 178) is a specific case of the RTNBO Earley-like case (section 13.10, p. 272): output strings may be represented as pointers to the nodes of a trie. As long as no calls are performed, the algorithm behaves as the FSTSO translator algorithm (section 11.5, p. 215): each transition adds at most one symbol to the output string in course, thus it is only required to jump from the corresponding output string node to one of its successors, or just to stay in the same node. When performing a call, new explorations starting with the empty output string are created, and also one symbol is appended at most during the call if no other calls or completions are performed. However, upon completion it is required to append the strings produced during the call to the strings produced before the call. This corresponds to the not-so-efficient case discussed in section 9.3 (p. 183): since string symbols can only be accessed in reverse order, it is necessary to reorder the whole string before adding it. The whole purpose of the optimization consists in transforming vectorial operations into scalar ones, but when concatenating two strings of a trie we must perform two vectorial operations: to reorder the string to append and then to append it. For the cases in which there are a few calls to perform, this problem will have no meaningful impact, but neither will the factoring out of common calls.

14.8   Earley-like language generation

Algorithm 14.6 *rtn_ earley_ language* is an adaptation of the Earley-like translator algorithm 14.3 for the computation of the language of a RTN, as seen with the breadth-first algorithm in section 14.5 (p. 284). Note that since the

algorithm only translates the empty sequence, every input index generated during the algorithm application will be zero. Therefore, we can remove input indexes from ESs and consider every input index equal to zero, that is, a unique SES V_0 is computed during the whole algorithm execution. In the original algorithm, the completer compared the input index of the current ES with the index of the current SES in order to either execute or not the ε-completer, but in this case we can skip this test since every completion will be in fact an ε-completion. As for the original algorithm, this algorithm also applies to left-recursive RTNs, and adding trie string management may or may not accelerate its execution depending on the calls to process.

q0 → {q5} → q1 → {q5} → q2 → {q5} → q3 → {q5} → q4

q5, a : A {q7} → q6

q7, ε : E → q8

	V_0	
$1:$	$(q_0, \varepsilon, \lambda, \{q_0\}, 0)$	initial ES
$2:$	$(q_1, \varepsilon, \{q_5\}, \{q_0\}, 0)$	pause(1)
$3:$	$(q_5, \varepsilon, \lambda, \{q_5\}, 0)$	call(1)
$4:$	$(q_6, \varepsilon, \{q_7\}, \{q_5\}, 0)$	pause(3)
$5:$	$(q_7, \varepsilon, \lambda, \{q_7\}, 0)$	call(3)
$6:$	$(q_8, E, \lambda, \{q_7\}, 0)$	generate$(5, E)$
$7:$	$(q_6, E, \lambda, \{q_5\}, 0)$	resume(4, 6); deletable(q_7, E)
$8:$	$(q_1, E, \lambda, \{q_0\}, 0)$	resume(2, 7); deletable(q_5, E)
$9:$	$(q_2, E, \{q_5\}, \{q_0\}, 0)$	pause(8); call already in 3
$10:$	$(q_2, EE, \lambda, \{q_0\}, 0)$	ε-resume(7, 9)
$11:$	$(q_3, EE, \{q_5\}, \{q_0\}, 0)$	pause(10); call already in 3
$12:$	$(q_3, EEE, \lambda, \{q_0\}, 0)$	ε-resume(7, 11)
$13:$	$(q_4, EEE, \{q_5\}, \{q_0\}, 0)$	pause(12); call already in 3
$14:$	$(q_4, EEEE, \lambda, \{q_0\}, 0)$	ε-resume(7, 13)
	V_1	
$15:$	$(q_6, A, \lambda, \{q_5\}, 0)$	translate$(3, a : A)$
$16:$	$(q_1, A, \lambda, \{q_0\}, 0)$	resume(2, 15)
$17:$	$(q_2, EA, \lambda, \{q_0\}, 0)$	resume(9, 15)
$18:$	$(q_3, EEA, \lambda, \{q_0\}, 0)$	resume(11, 15)
$19:$	$(q_4, EEEA, \lambda, \{q_0\}, 0)$	resume(13, 15); acceptance ES

20 :	$(q_2, A, \{q_5\}, \{q_0\}, 0)$	pause(16)
21 :	$(q_5, \varepsilon, \lambda, \{q_5\}, 1)$	call(16)
22 :	$(q_3, EA, \{q_5\}, \{q_0\}, 0)$	pause(17); call already in 21
23 :	$(q_4, EEA, \{q_5\}, \{q_0\}, 0)$	pause(18); call already in 21
24 :	$(q_6, \varepsilon, \{q_7\}, \{q_5\}, 1)$	pause(21)
25 :	$(q_7, \varepsilon, \lambda, \{q_7\}, 1)$	call(21)
26 :	$(q_8, E, \lambda, \{q_7\}, 1)$	generate(25, E)
27 :	$(q_6, E, \lambda, \{q_5\}, 1)$	resume(24, 26)
28 :	$(q_2, AE, \lambda, \{q_0\}, 0)$	resume(20, 27); deletable($\{q_5\}, E$)
29 :	$(q_3, EAE, \lambda, \{q_0\}, 0)$	resume(22, 27); deletable($\{q_5\}, E$)
30 :	$(q_4, EEAE, \lambda, \{q_0\}, 0)$	resume(23, 27); deletable($\{q_5\}, E$); acceptance ES
31 :	$(q_3, AE, \{q_5\}, \{q_0\}, 0)$	pause(28); call already in (21)
32 :	$(q_3, AEE, \lambda, \{q_0\}, 0)$	ε-resume(27, 31)
33 :	$(q_4, EAE, \{q_5\}, \{q_0\}, 0)$	pause(30); call already in (21)
34 :	$(q_4, EAEE, \lambda, \{q_0\}, 0)$	ε-resume(27, 33); acceptance ES
35 :	$(q_4, AEE, \{q_5\}, \{q_0\}, 0)$	pause(32); call already in (21)
36 :	$(q_4, AEEE, \lambda, \{q_0\}, 0)$	ε-resume(27, 35); acceptance ES

Figure 14.4: RTN with deletable calls of figure 12.10 extended with string output and execution trace of algorithm 12.2 *rtn_ earley_ recognize_ string* for this RTNSO and input a; without the ε-completer, greyed ESs would be missing and the input would be rejected.

V_0		
$1:$	$(q_0, \varepsilon, \lambda, \{q_0\}, 0)$	initial ES
$2:$	$(q_1, \varepsilon, \{q_0\}, \{q_0\}, 0)$	pause(1); call already in 1
V_1		
$3:$	$(q_2, B, \lambda, \{q_0\}, 0)$	translate$(1, b : B)$
$4:$	$(q_1, \varepsilon, \lambda, \{q_0\}, 0)$	resume$(2, 3)$
V_2		
$5:$	$(q_2, BA, \varepsilon, \lambda, \{q_0\}, 0)$	translate$(4, a : A)$
$6:$	$(q_1, BA, \varepsilon, \lambda, \{q_0\}, 0)$	resume$(2, 5)$
V_3		
$7:$	$(q_2, BAA, \lambda, \{q_0\}, 0)$	translate$(6, BAA)$
$8:$	$(q_1, BAA, \lambda, \{q_0\}, 0)$	resume$(2, 7)$
$\vdots$		
V_l		
$2l + 1:$	$(q_2, BA^{l-1}, \lambda, \{q_0\}, 0)$	translate$(2l, a : A)$; acceptance ES
$2l + 2:$	$(q_1, BA^{l-1}, \lambda, \{q_0\}, 0)$	resume$(2, 2l + 1)$

Figure 14.5: Left-recursive RTNSO translating ba^n into BA^n and execution trace of algorithm 14.3 *rtnso_ earley-_ translate_ string* for this RTNSO and input ba^l.

Algorithm 14.6 rtn_earley_language(A)   ▷ $L(A)$, eq. (12.3)

Input: $A = (Q, \Sigma, \delta, Q_I, F)$, a RTN
Output: L, the language of A
1:   $V \leftarrow \emptyset$
2:   $E \leftarrow \emptyset$
3:   **for each** $(q_c \in Q_I)$ **do**
4:      unconditionally_add_enqueue_es($V, E, (q_c, \varepsilon, \lambda, Q_I)$)
5:   **end for**
6:   $T \leftarrow \emptyset$
7:   **while** $E \neq \emptyset$ **do**
8:      $(q_s, w, \lambda, Q_h) \leftarrow \text{dequeue}(E)$
                                 ▷ EXPLICIT ε-TRANSITIONS
9:      **for each** $q_t \in \delta(q_s, \varepsilon)$ **do**
10:        add_enqueue_es($V_k, E, (q_t, w, \lambda, Q_h)$)
11:      **end for**
                                    ▷ CONSUMING TRANSITIONS
12:      **for each** $q_t \in \delta(q_s, \sigma)$ **do**
13:        add_enqueue_es($V_k, E, (q_t, w\sigma, \lambda, Q_h)$)
14:      **end for**
                                        ▷ PREDICTOR
15:      **for each** $(q_r, Q_c) : q_r \in \delta(q_s, Q_c)$ **do**
16:        **if** add($V, (q_r, w, Q_c, Q_h)$) **then**
17:          **if** $\nexists w' : (Q_c, w') \in T$ **then**
18:            **for each** $q_c \in Q_c$ **do**
19:              add_enqueue_es($V, E, (q_c, \varepsilon, \lambda, Q_c)$)
20:            **end for**
                                       ▷ ε-COMPLETER
21:          **else**
22:            **for each** $w' : (Q_c, w') \in T$ **do**
23:              add_enqueue_es($V, E, (q_r, ww', \lambda, Q_h)$)
24:            **end for**
25:          **end if**
26:        **end if**
27:      **end for**

$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ ▷ COMPLETER

28:     **if** $q_s \in F$ **then**

29:         **for each** $(q_r, w', Q_h, Q'_h) \in V$ **do**

30:             add_enqueue_es$(V, E, (q_r, ww', \lambda, Q'_h)$

$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ ▷ ε-COMPLETER

31:             add$(T, (Q_h, w'))$

32:         **end for**

33:     **end if**

34: **end while**

35: $L \leftarrow \emptyset$

36: **for each** $(q_s, w, \lambda, Q_I) \in V : q_s \in F$ **do**

37:     add(L, w)

38: **end for**

Chapter 15

Filtered-popping recursive transition networks

In the previous chapter we have shown the inevitable exponential explosion upon applying a RTNSO generating an exponential number of outputs w.r.t. an input increasing in length, even for an Earley-like algorithm of application. In order to avoid this explosion, we propose to represent the set of outputs as some kind of finite-state machine. This machine should have the same structure than the trace of the Earley-like recognizer, so the algorithm building it would keep the asymptotic cost of the Earley-like recognizer, $O(n^3)$. This machine cannot be a FSA, since it should have a subroutine jump mechanism: the exponential explosion due to the combination of outputs upon call completions must be avoided by representing common output infixes as substructures of the machine that are called rather than explicitly represented multiple times. It neither can be a RTN since, when executing the algorithm, common calls may be completed by consuming input segments of different lengths, but the combination of the blackboards generated before, during and after a call must correspond to the translation of consecutive input segments (see figure 15.1). It is a filtered-popping RTN or FPRTN, a new kind of finite-state machine we have defined as a RTN where states are associated to input indexes, and popping transitions cannot be traversed unless both the source and target states are associated to the same input indexes. We have briefly presented both FPRTNs and the algorithm building them in Sastre (2009).

Once the FPRTN is built, one can compute the effective list of outputs, if necessary, by generating the FPRTN language. Obviously, this operation will

Figure 15.1: At the left, an ambiguous RTNSO, and at the right, a FPRTN recognizing the language of translations of *abc* for this RTNSO. Boxes contain the key of the state they are attached to. FPRTN push and pop transitions are explicitly represented as dotted and thick arrows, respectively. Only pop transitions corresponding to connected input segments are allowed: pop transitions from r_7 to r_5 and from r_9 to r_3 are forbidden since the former skips the translation of c and the latter translates c twice.

still have an exponential cost, as the size of the output set is exponential, but the FPRTN can be pruned first in order to avoid the construction of partial blackboards that uniquely correspond to misinterpretations of the input. We will study in chapter 16 the subclass of FPRTNs built by the algorithm presented here and give two efficient algorithms for generating their languages. Moreover, we will give in chapter 18 an algorithm able to extract only the top-ranked blackboard of a weighted FPRTN, provided that the grammar is a RTNBO where blackboards include weight output.

Definition 265 (FPRTN). *A FPRTN $(Q, K, \Sigma, \delta, \kappa, Q_I, F)$ is a special type of FSMs (definition 46, p. 121) with a stack, a set of keys K and a $\kappa : Q \to K$ function that maps states to keys in K, whose set of labels Ξ take its elements from $(\Sigma \cup \{\varepsilon\} \cup Q)$, where Σ is a finite input alphabet, Q the finite states of the FPRTN and ε the empty symbol. FPRTNs can be seen as an extension of RTNs (definition 183) by associating keys to states and by adding a filter to the pop transitions so that they take place only if the keys of the acceptor and return states match.*

15.1 Transitions

FPRTN transition definitions are the same than the ones for RTNs (section 12.1, p. 221) except for popping transitions, which we redefine below.

Definition 266 (Filtered-pop transition). *Filtered-pop transitions $(q_f, q_r\uparrow, q_r)$ are implicit ε-transitions which take place each time an acceptance state q_f is reached while executing a call having q_r as return state such that $\kappa(q_f) = \kappa(q_r)$, that is, the keys of the acceptance and return states match. When traversing a filtered-pop transition, the state q_r at the top of the stack of return states is popped out and the machine is taken to the popped state q_r without consuming any input symbol.*

15.2 Graphical representation

We represent FPRTNs as RTNs with a box attached to each state, each box containing the key of the corresponding state (see figure 15.1). Filtered-pop transitions are represented as RTN pop transitions.

15.3 Sequences of transitions

FPRTN paths and cycles are defined as for RTNs (section 12.3, p. 225). Recursive calls for the case of RTNs lead to an infinite set of interpretations within the machine, since realizable call cycles can be traversed an infinite number of times. For the case of FPRTNs, we will show in the next section how filtered-pop transitions add additional restrictions upon the number of times call cycles can be realized.

15.4 Behaviour

FPRTNs behave as RTNs except for the pop transitions. Therefore, the only difference is the way in which the D_{pop} function is computed.

Definition 267 (D). *The $D(V)$ function for FPRTNs is defined as for RTNs (definition 211, p. 229) except for the predicate of its D_{pop} component, which is redefined as*

- $d = q_f \in F \wedge \kappa(q_f) = \kappa(q_r)$,

that is, by adding the key-matching restriction.

Lemma 23 (Infinite ε-closure). *The ε-closure of a FPRTN SES V is infinite if there exists an ES $x = (q, \pi)$ within V or ε-reachable from an ES of V such that there exists an ε-realizable call cycle whose start state is q.*

Proof. The proof is the same than for RTNs (proof of lemma 14, p. 230) but taking into account that pop transitions involved in reaching ES (q, π), and in ε-realizing the call cycle, require the keys of their source and target states to be equal. □

Lemma 24 (Finite ε-closure). *Under conditions other than those expressed in the previous lemma, the ε-closure of a FPRTN SES is finite.*

Proof. The proof is the same than for RTNs (proof of lemma 15, p. 230) but taking into account that not only the possible realizable paths under the mentioned conditions yield finite SESs, but may be even shorter than those of the RTN case due to non-realizable pop transitions. □

Corollary 9. *The ε-closure is always finite for non-left-recursive FPRTNs.*

Definition 268 (Initial and acceptance SESs). *The initial and acceptance SESs of a FPRTN are defined as for RTNs (definition 212, p. 231).*

Definition 269 (L). *The language of a FPRTN is defined as for RTNs (definition 214, p. 232), though taking into account that pop transitions are filtered.*

Lemma 25 (Infinite recursion degree). *The recursion degree of a FPRTN having at least one useful call cycle p such that p^2 is also useful is infinite.*

Proof. The proof is similar to the one for RTNs (proof of lemma 16, p. 232); let us suppose a FPRTN such as the RTN of figure 12.5 (p. 233) where each state q_k is associated to a key k_k, that is, $\kappa(q_k) = k_k$. In fact, the relevant keys to this proof are k_{f_3}, k_{r_2}, k_{f_2} and k_{r_1}, the ones of the acceptance and return states (except for q_{f_1}, the "global" acceptance state), since they determine whether the pop transitions are realizable or not. Let p be a path within the FPRTN such as the one defined in the proof for RTNs,

$$p = p_a \ (q_{s_1}, q_{r_1}\downarrow, q_{c_1}) \ p_b \ (q_{s_2}, q_{r_2}\downarrow, q_{c_1}) \ p_c \ (q_{f_3}, q_{r_2}\uparrow, q_{r_2}) \ p_d \ (q_{f_2}, q_{r_1}\uparrow, q_{r_1}) \ p_e,$$

where p_b $(q_{s_2}, q_{r_2}\downarrow, q_{c_1})$ is a call cycle. First of all, if the cycle is useful then it must be realizable as well as the pop transition removing the return state that is pushed onto the stack during the realization of the call cycle. If pop transition $(q_{f_3}, q_{r_2}\uparrow, q_{r_2})$ is realizable then $k_{f_3} = k_{r_2}$. For the cycle to be useful, path p must be an interpretation and therefore realizable. As before, if pop transition $(q_{f_2}, q_{r_1}\uparrow, q_{r_1})$ is realizable then $k_{f_2} = k_{r_1}$. If the cycle can be traversed twice and still be useful, then path

$$p_2 = p_a \ (q_{s_1}, q_{r_1}\downarrow, q_{c_1}) \ (p_b \ (q_{s_2}, q_{r_2}\downarrow, q_{c_1}))^2 \ p_c \ (q_{f_3}, q_{r_2}\uparrow, q_{r_2})$$
$$p_d \ (q_{f_2}, q_{r_2}\uparrow, q_{r_2}) \ p_d \ (q_{f_2}, q_{r_1}\uparrow, q_{r_1}) \ p_e$$

must also be an interpretation. Note that the second traversal of the cycle requires an additional realizable pop transition $(q_{f_2}, q_{r_2}\uparrow, q_{r_2})$ in order to be useful, which either implies $k_{f_2} = k_{r_2}$ or path p_d to be empty: the latter case implies the former one since q_{f_2} would be equal to q_{r_2} and therefore both states would be associated to the same key. Obviously, if $k_{f_2} = k_{r_2}$ then it is not only possible to traverse twice the cycle but to traverse it any number of times, hence allowing for the existence of an infinite number of interpretations. If this last equation does not hold then the cycle can still be useful but not its self-concatenations, that is, the cycle will allow for a recursion degree equal to one, but not zero or infinite. $\square$

Theorem 25 (Possible recursion degrees). *According to the previous proof, the recursion degree of a FPRTN is either zero, one or infinite.*

Theorem 26 (Cardinality of the interpretation set). *Given the previous theorem and the theorems on the cardinality of the interpretation set for FSMs (theorem 4, p. 145) and for FSAs (theorem 6, p. 164), the number of interpretations of a FPRTN is infinite iff it contains at least one useful cycle p holding one of the following conditions:*

- *p is uniquely composed by consuming transitions and/or ε-transitions,*

- *p contains realizable push and pop transitions —keys of source and target states of each pop transition are equal— but the execution of p momentarily modifies the stack, that's it, the stack before and after the execution of p is the same, or*

- *p is a call cycle such that p^2 is also useful.*

Theorem 27 (Cardinality of the language). *Given theorem 5 (p. 146), since FSAs allow for the realization of any of its transitions, the language of a FPRTN is infinite iff it contains at least one useful consuming cycle p holding one of the conditions of the previous theorem.*

15.5   Reverse FPRTN

Definition 270 (RFPRTN). *We define filtered-pushing recursive transition networks or reverse FPRTNs (RFPRTNs) as FPRTNs where the filtering is to be applied to push transitions instead of to popping ones.*

Definition 271 (Reverse FPRTN). *Let A be a FPRTN $(Q, K, \Sigma, \delta, \kappa, Q_I, F)$ with disjoint submachines; we define A^R, the canonical reverse of A, as a RFPRTN $(Q, K, \Sigma, \delta', \kappa, Q'_I, F')$ such that $A' = (Q, \Sigma, \delta, Q_I, F)$ is a RTN and $A'^R = (Q, \Sigma, \delta', Q'_I, F')$ is the canonical reverse of A' (definition 215, p. 234).*

Proof. The proof is the same than for RTNs but with a slight difference in the reversal of call transitions and submachines; let $p = (q_s, q_t\downarrow, q_c)p'(q_f, q_t\uparrow, q_t)$ be a path within A completing a call, the keys of q_f and q_t must be equal so that the pop transition can be taken. If A^R would be a FPRTN instead of a RFPRTN, then $p^R = (q_t, q_s\downarrow, q_f)p'^R(q_c, q_s\uparrow, q_s)$ would be a path within A^R which might not be realizable since the keys of states q_s and q_c are not necessarily equal, and therefore A might recognize a word w such that w^R is not recognized by A^R. Moreover, the opposite could also be true: the keys of q_c and q_s could be different while the keys of q_t and q_f would be equal, thus A^R could recognize a word w^R while A would not recognize word w. Consequently, the matching key restriction must be applied to push transitions instead of pop transitions in order to ensure that A^R recognizes the reverse language of A and not any other language, and therefore the canonical reverse of a FPRTN is a RFPRTN and not another FPRTN. Conversely, the reverse of a RFPRTN is a FPRTN and not another RFPRTN.   □

15.6   Translating a string into a FPRTN

Algorithm 15.1 *rtnbo_translate_string_to_fprtn* is an equivalent version of algorithm 13.2 *rtnbo_earley_translate_string* which returns a FPRTN

rather than the effective list of outputs.[1] This algorithm has been derived
from the RTN Earley-like acceptor algorithm 12.2 (p. 249) by adding the
required instructions for building the resulting FPRTN: a FPRTN state is
created for each active ES generated during the algorithm execution, and a
FPRTN transition is added for each explored RTNBO transition deriving an
active ES from another one, where RTNBO output labels become FPRTN in-
put ones. FPRTN state keys are the input indexes at the moment of creation
of each state. Algorithm 15.2 *fprtn_ create_ state* is used for the creation of
FPRTN states instead of algorithm 8.3 *fsm_ create_ state* (p. 173) in order to
also associate the state to a key. From now on, we will call FPRTNs states
output states or OSs. The algorithm builds a function ζ_s mapping pairs
$(k, (q_s, \lambda, q_h, j))$ to OSs so that it is possible to retrieve the OS corresponding
to any of the previously generated ESs. Notice that active ESs belonging
to different SES may be equal (e.g.: $x_s = x'_s = (q_s, \lambda, Q_h, k)$, with $x_s \in V_j$,
$x'_s \in V_k$ and $0 \leq j < k \leq l$), hence we need to specify here the index of the
SES containing them in order to uniquely identify them;[2] as stated in the
paragraph after definition 218 (p. 244), the index of the SES containing an
ES is also a term of the ES, but we have omitted it in order not to repeat
this index for each ES of a SES (we simply use the SES indexes). Notice
as well that output labels are just copied as input labels of the resulting
FPRTN: blackboard functions are not interpreted here but just annotated
in order to be executed in a further stage of treatment. The same algorithm
is valid for RTNs with string output, weight output, unification processes or
any combination of these output types; different algorithms will be required
for the partial or total generation of the language recognized by the FPRTN,
but not for the construction the FPRTN itself.

The algorithm first allocates memory for storing the parsing chart: a vec-
tor of $l + 1$ SESs. It builds two additional OSs, initial OS r_s and the "global"
acceptance OS r_f, where r_s is associated to index 0 and r_f to index l, the
input length. Then it adds to V_0 the initial SES X_I for RTNs using the rou-
tine *unconditionally_ add_ enqueue_ link_ es_ os* (algorithm 15.3) in order to

---

[1]We only define the FPRTN in algorithm 15.1 *rtnbo_ translate_ string_ to_ fprtn* and
then treat it in the other algorithms as a global variable in order to avoid repetition.

[2]In practice we do not build a map object representing ζ_s but just add an extra field
to active ESs in order to store the pointer to the corresponding OS, so retrieving this OS
does not involve a search inside a map but just to follow the pointer. The only purpose
of this field is to accelerate the retrieval of OSs, so whenever comparing two active ESs of
the same SES for equality this field is not taken into account.

add each ES. This routine extends routine *unconditionally_ add_ enqueue_ es* (algorithm 7.7) by unconditionally creating an OS r_c, as well as the specified ES x_s, and by adding the corresponding map to ζ_s (the link between the ES and the OS).[3] Let R_c be the set of OSs (SOS) corresponding to X_I, the algorithm adds a call to R_c from r_s to r_f. This construction represents a call to the grammar's axiom, which can only be realized by consuming the entire input: since r_f is associated to index l, the input length, pop transitions to r_f from states whose index is less than l —and consequently associated to ESs reached before consuming the whole input— will not be realizable. Afterwards, the same iterative process of the RTN Earley-like algorithm is followed here in order to build the SESs V_0 to V_l, but using algorithm 15.4 *rtnbo_ translate_ symbol_ to_ fprtn* and algorithm 15.6 *rtnbo- _ interlaced_ eclosure_ to_ fprtn* instead of the equivalent RTN ones for the implementation of the Δ and ε-closure functions. Since the number of realizable filtered-pop transitions of the resulting FPRTN is finite (the ones found during the algorithm execution), we explicitly define them so that further FPRTN postprocessing does not require to search for them again. The last loop of the Earley-like RTN acceptor is modified so that for each acceptor ES in the last SES a filtered-pop transition is added towards the "global" acceptor OS, r_f.

Algorithm 15.4 *rtnbo_ translate_ symbol_ to_ fprtn* is an almost straightforward adaptation of algorithm 12.3 *rtn_ earley_ recognize_ symbol* (p. 249) for the construction of a FPRTN. For each active ES $x_s = (q_s, \lambda, Q_h, j)$ in V (which is in fact V_k, the last computed SES), it first retrieves its associated OS r_s. Then, for each consuming transition $(q_s, (\sigma, g), q_t)$ it derives ES (q_t, λ, Q_h, j) and adds it to SES W (which will be V_{k+1}, the next SES) using algorithm 15.5 *add_ enqueue_ link_ es_ os*, the extended version of routine *add_ enqueue_ es* (algorithm 7.4). Besides adding an ES to a SES and enqueuing it for further processing if the ES was not already present in the SES, it also creates its output state r_t with key $k + 1$ and adds the corresponding map to the ζ_s function, or just returns the former created OS if the ES was already present in the SES. Finally, algorithm 15.4 *rtnbo_ translate_ symbol_ to_ fprtn* adds the FPRTN transition (r_s, g, r_t) which represents the possible partial translation of input symbol σ_{k+1} into $g \in \Gamma \cup \{\mathrm{id}_B\}$, since states r_s and r_t are associated to input indexes k and $k + 1$, respectively.

---

[3]As stated before, in practice we only fill the additional field of the active ES with the pointer to the OS we have just created.

Algorithm 15.1 rtnbo_translate_string_to_fprtn($\sigma_1 \ldots \sigma_l$) $\triangleright$
$\omega(A, \sigma_1 \ldots \sigma_l)$

Input: $\sigma_1 \ldots \sigma_l$, an input string of length l
Output: $A' = (Q', \mathbb{N}, \Gamma, \delta', \kappa, Q'_I, F')$, the FPRTN recognizing $\omega(A, \sigma_1 \ldots \sigma_l)$
 1: allocate_memory_for_chart(V^{l+1})
 2: $r_s \leftarrow$ fprtn_create_state(true, false, 0)
 3: $r_f \leftarrow$ fprtn_create_state(false, true, l)
 4: $V_0 \leftarrow \emptyset$
 5: $E \leftarrow \emptyset$
 6: $R_c \leftarrow \emptyset$
 7: **for each** $(q_c \in Q_I)$ **do**
 8:    $r_c \leftarrow$ unconditionally_add_enqueue_link_es_os($V_0, E, 0, (q_c, \lambda, Q_I, 0)$)
 9:    add(R_c, r_c)
10: **end for**
11: $\delta'(r_s, R_c) \leftarrow \{r_f\}$
12: rtnbo_interlaced_eclosure_to_fprtn($V^{l+1}, E, 0$)
13: $k \leftarrow 0$
14: **while** $V_k \neq \emptyset \wedge k < l$ **do**
15:    $V_{k+1} \leftarrow$ rtnbo_translate_symbol_to_fprtn(V_k, E, k, σ_{k+1})
16:    $k \leftarrow k + 1$
17:    rtnbo_interlaced_eclosure_to_fprtn(V^{l+1}, E, k)
18: **end while**
19: **for each** $x_s \in V_k : x_s = (q_s, \lambda, Q_I, 0) \wedge q_s \in F$ **do**
20:    add($\delta'(\zeta_s(k, x_s), r_f\uparrow), r_f$)
21: **end for**

Algorithm 15.2 fprtn_create_state(is_initial, is_final, k)

Input: is_initial, future value of predicate $r \in Q'_I$
       is_final, future value of predicate $r \in F'$
       k, the state key
Output: r, the new FPRTN state
 1: $r \leftarrow$ fsm_create_state(is_initial, is_final)
 2: $\kappa(r) \leftarrow k$

Algorithm 15.3 unconditionally_add_enqueue_link_es_os(V, E, k, x_t)

Input: V, the SES where the ES is added

  E, the queue of unexplored ESs

  k, the current input position

  x_t, the target ES to add to V

Output: V after adding the ES

  E after enqueuing the ES, if new

  r_t, the target ES OS

 1: unconditionally_add_enqueue_es(V, E, x_t))
 2: $r_t \leftarrow$ fprtn_create_state(false, state(x_t) $\in F, k$)
 3: add($\zeta_s(k, x_t), r_t$)

Algorithm 15.4 rtnbo_translate_symbol_to_fprtn(V, E, k, σ) $\triangleright$
$\Delta(V_k, \sigma_{k+1})$

Input: V, a SES

  E, the empty queue of unexplored ESs

  k, the index of V

  σ, the input symbol to translate

Output: W, the set of reachable ESs from V by consuming σ

  E after enqueuing the ESs of W

 1: $W \leftarrow \emptyset$
 2: **for each** $(q_s, \lambda, Q_h, j) \in V$ **do**
 3:     $r_s \leftarrow \zeta_s(k, (q_s, \lambda, Q_h, j))$
 4:     **for each** $(q_t, g) : q_t \in \delta(q_s, (\sigma, g))$ **do**
 5:         $r_t \leftarrow$ add_enqueue_link_es_os($W, E, k + 1, (q_t, \lambda, Q_h, j)$)
 6:         add($\delta'(r_s, g), r_t$)
 7:     **end for**
 8: **end for**

Algorithm 15.5 add_enqueue_link_es_os(V, E, k, x_t)

Input: V, the SES where the ES is added

   E, the queue of unexplored ESs

   k, the current input position

   x_t, the target ES to add to V

Output: V after adding the ES

   E after enqueuing the ES, if new

   r_t, the target ES OS

 1: **if** add(V, x_t) **then**
 2:    enqueue(E, x_t)
 3:    $r_t \leftarrow$ fprtn_create_state(false, state(x_t) $\in F, k$)
 4:    add($\zeta_s(k, x_t), r_t$)
 5: **else**
 6:    $r_t \leftarrow \zeta_s(k, x_t)$
 7: **end if**

Algorithm 15.6 *rtnbo_interlaced_eclosure_to_fprtn* is a slightly more complex adaptation of algorithm 12.4 *rtn_earley_interlaced_eclosure*. The derivation of active ESs due to explicit ε-transitions is analogous to the derivation through consuming transitions; therefore, the extension is almost the same. However, the extension of the predictor, completer and ε-completer involves considering more ESs than the former case, namely

- $x_s = (q_s, \lambda, Q_h, j) \in V_k$, the current active ES whose RTNBO state q_s has been detected to be final and hence triggering the completion of parallel calls to Q_h,

- $x_p = (q_r, Q_h, Q'_h, i) \in V_j$, a paused ES waiting for the completion of call to Q_h,

- $x_r = (q_r, \lambda, Q'_h, i) \in V_k$, the return active ES result of resuming paused ES x_p,

- $x'_s \in V_j$, the active ES from where call to Q_h was performed, resulting in paused ES x_p, and

- X_c, the set of active ESs initiating the exploration of call to Q_h.

Note that reaching ES x_s may resume several paused ESs, each one having an associated return ES and one or more source ESs of the call; therefore, reaching x_s may trigger several call completions. Generating the corresponding call structure inside the FPRTN involves the following OSs:

- $r_s' = \zeta_s(x_s')$, the source OS of the call, associated to ES x_s',

- $R_c = \{r_c : \zeta_s(x_c) = r_c \wedge x_c \in X_c\}$, the called SOS,

- $r_r = \zeta_s(x_r)$, the return OS, and

- $r_s = \zeta_s(x_s)$, the acceptance OS that triggers the call completion, if the filtered-pop transition is to be explicitly defined.

In order to add the FPRTN call transition (r_s', R_c, r_r) in the completer we are first required to retrieve the involved OSs and SOS. OS r_r is created or just retrieved, if the associated return ES already existed, by routine *add_ en- queue_ link_ es_ os* inside the completer. OS r_s' has been previously created by some derivation mechanism of the algorithm, but its ES x_s' is accessed during the prediction of call to Q_h in order to create paused ES x_p and active ES x_c. Inside the predictor, we build two additional maps so that the completer can retrieve these elements latter: ζ_s' mapping x_p to r_s' and ζ_I mapping x_p to R_c.[4] The *ε-completer* inside the completer does not need to be modified: it just marks call to Q_h as deletable (adds Q_h to T) for the current SES V_k. The *ε-completer* inside the predictor will just create or retrieve return OS r_r in order to add the corresponding FPRTN call transition since the other needed elements are already created or retrieved by the predictor. Filtered-pop transitions due to ε-completions are explicitly defined inside the completer; therefore the ε-completer is not required to define them again.

Notice that for all of the derivation mechanisms it might be possible to reach the same RTNBO state through different paths generating different output sequences. For the RTNBO Earley-like translator this implied generating several ESs instead of only one: one for each different output since outputs are a part of the ESs. For the algorithm generating a FPRTN, outputs are represented as FPRTN transitions rather than being stored inside

---

[4]As was done for map ζ_s (footnote 2, p. 307), in practice we do not implement two map objects representing ζ_s' and ζ_I but extend paused ESs with two fields storing the pointers to the corresponding OS and SOS. Those fields are not either taken into account when comparing paused ESs for equality.

Algorithm 15.6 rtnbo_interlaced_eclosure_to_fprtn$(V^{l+1}, E, k) \triangleright C_\varepsilon(V_k)$

Input: V^{l+1}, the chart

        E, the queue of unexplored ESs containing every ES in V_k

        k, the index of the SES V_k

Output: V^{l+1} after adding to V_k its ε-closure

        E after emptying it

1: $T \leftarrow \emptyset$
2: **while** $E \neq \emptyset$ **do**
3:     $(q_s, \lambda, Q_h, j) \leftarrow \text{dequeue}(E)$
4:     $r_s \leftarrow \zeta_s(k, (q_s, \lambda, Q_h, j))$

                                  $\triangleright$ EXPLICIT ε-TRANSITIONS

5:     **for each** $(q_t, g) : q_t \in \delta(q_s, (\varepsilon, g))$ **do**
6:         $r_t \leftarrow \text{add\_enqueue\_link\_es\_os}(V_k, E, k, (q_t, \lambda, Q_h, j))$
7:         $\text{add}(\delta'(r_s, g), r_t)$
8:     **end for**

                                        $\triangleright$ PREDICTOR

9:     **for each** $(q_r, Q_c) : q_r \in \delta(q_s, Q_c)$ **do**
10:      **if** $\text{add}(V_k, (q_r, Q_c, Q_h, j))$ **then**
11:         $\zeta'_s(k, (q_r, Q_c, Q_h, j)) \leftarrow \{r_s\}$
12:         $R_c \leftarrow \zeta_I(k, Q_c)$
13:         **if** $R_c = \perp$ **then**
14:            $R_c \leftarrow \emptyset$
15:            **for each** $q_c \in Q_c$ **do**
16:                $r_c \leftarrow \text{add\_enqueue\_link\_es\_os}(V_k, E, k, (q_c, \lambda, Q_c, k))$
17:                $\text{enqueue}(R_c, r_c)$
18:            **end for**
19:            $\zeta_I(k, Q_c) \leftarrow R_c$

                                      $\triangleright$ ε-COMPLETER

20:         **else if** $\exists r_f : (Q_c, r_f) \in T$ **then**
21:            $r_r \leftarrow \text{add\_enqueue\_link\_es\_os}(V_k, E, k, (q_r, \lambda, Q_h, j))$
22:            $\text{add}(\delta'(r_s, R_c), r_r)$
23:            **for each** $r_f : (Q_c, r_f) \in T$ **do**
24:                $\text{add}(\delta'(r_f, r_r\uparrow), r_r)$
25:            **end for**
26:         **end if**

```
27:            else
28:                 enqueue(ζ'_s(k, (q_r, Q_c, Q_h, j)), r_s)
29:            end if
30:        end for
                                                         ▷ COMPLETER
31:        if q_s ∈ F then
32:            for each (q_r, Q_h, Q'_h, i) ∈ V_j do
33:                 r_r ← add_enqueue_link_es_os(V_k, E, k, (q_r, λ, Q'_h, i))
34:                 R'_s ← ζ'_s(j, (q_r, Q_h, Q'_h, i))
35:                 R_c ← ζ_I(Q_h)
36:                 for each r'_s ∈ R'_s do
37:                     add(δ'(r'_s, R_c), r_r)
38:                 end for
39:                 add(δ'(r_s, r_r↑), r_r)
                                                         ▷ ε-COMPLETER
40:                 if i = k then
41:                     add(T, (Q_h, r_s))
42:                 end if
43:             end for
44:        end if
45: end while
```

the ESs. When deriving an ES by generating an output (or empty output), if the ES was not present it is generated as well as its associated OS, and the corresponding transition with the output label is added, but if the ES was already present its OS is just retrieved and a new alternative transition is added (again, if the transition was not already present).

Figure 15.2 is an example of execution of algorithm 15.1 *rtnbo_ translate-_string_ to_ fprtn* for input *aabb* and the RTNSO of figure 14.1 (p. 284).[5] On the left, we have drawn a copy of the RTNSO and, on the right, we have represented the trace of the RTN Earley-like acceptor with its corresponding output FPRTN. Notice that each line contains a RTN ES along with its associated FPRTN state, and that every transition within the trace has its corresponding FPRTN transition with the same transition label but omitting

---

[5]Blackboards are strings and output labels are just output symbols; as stated before, the algorithm is the same for any kind of output since it does not interpret the outputs but just annotates them.

the input symbol. The key of FPRTN state r_0 is 0, which represents the point just before the first input symbol σ_1. Keys k_s and k_t of two FPRTN states r_s and r_t associated to two ESs x_s and x_t such that x_t is directly reachable from x_s are equal iff the transitions that led to x_t from x_s did not involve input consumption; otherwise k_t is equal to $k_s + 1$. The "global" acceptance state r_1 is associated to key 4 so that any interpretation within the FPRTN necessarily corresponds to the whole consumption of input *aabb*. Note that a single RTNBO call results in several FPRTN calls when the RTNBO call can be completed by consuming input segments of different lengths, since several return ESs belonging to different SESs will be produced; for instance, the first realization of call transition $(q_1, \{q_0\}, q_2)$ produces the FPRTN call transitions $(r_4, \{r_6\}, r_8)$ and $(r_4, \{r_6\}, r_{17})$, which differ only in the target state: OS r_8, whose ES belongs to V_1, and OS r_{17}, whose ES belongs to V_3. The first corresponding filtered-pop transition, $(r_7, r_8\uparrow, r_8)$, is only realizable if the FPRTN call represents a translation of the empty input segment right after consuming the first input symbol $(\kappa(r_6) = 1 \wedge \kappa(r_7) = \kappa(r_8) = 1)$, and the second one, $(r_{13}, r_{14}\uparrow, r_{14})$, if the FPRTN call represents the translation of the second and third input symbols $(\kappa(r_6) = 1 \wedge \kappa(r_{16}) = \kappa(r_{17}) = 3)$. Note also that a RTNBO call transition results in a single FPRTN call transition but several filtered-pop transitions when the call is realizable by reaching multiple acceptor states but always consuming the same amount of input symbols.

Figure 15.3 is an example of execution of algorithm 15.1 *rtnbo_ translate_ string_ to_ fprtn*, equivalent to the example of figure 14.4 (p. 297) for the RTNSO with deletable calls. In this example we can appreciate how the deletable call to SS $\{q_5\}$ is computed only once for each SES, and further calls are processed by the ε-completer by just adding the corresponding call transition. As we can see, deletable calls allow for execution paths traversing the same call successive times inside the same SES (e.g.: call to SOS $\{r_3\}$); even though the same ES is reached several times, no call-cycle is present since each call is completed before starting the next one (the return state is popped out before pushing it again); for instance, the following is an execution path reaching state r_3 four times, starting from state r_0 and ending

q_0

$a : \{$ $a : [$

q_1 q_3

$\{q_0\}$ $\varepsilon : *$ $\{q_0\}$

q_2 q_4

$b :\}$ $b :]$

q_5

$\{6\}\{6\}$

V_0

r_0 [0]

r_1

2   $(q_0, \lambda, \{q_0\}, 0)$ $\{r_2\}$ r_2 [0]   Initial_set

$\varepsilon{:}*$ $*$

3   $(q_5, \lambda, \{q_0\}, 0)$ r_3 [0] $\cup\, C_\varepsilon(\text{Initial\_set})$

V_1 $a{:}\{$ $a{:}[$ $\{$ $[$

4   $(q_1, \lambda, \{q_0\}, 0)$ r_4 [1] $\Delta(V_0, a)$

5   $(q_3, \lambda, \{q_0\}, 0)$ r_{17} r_5 [1]

$(q_2, \{q_0\}, \{q_0\}, 0)$ r_8 r_9 r_{15}

6   6

6   $(q_0, \lambda, \{q_0\}, 1)$ $\{r_6\}\{r_6\}$ r_6 [1] $\{r_6\}\{r_6\}$

$(q_4, \{q_0\}, \{q_0\}, 0)$ $*$ $\cup\, C_\varepsilon(\Delta(V_0, a))$

$\varepsilon{:}*$

7   $(q_5, \lambda, \{q_0\}, 1)$ r_7 [1]

4

8   $(q_2, \lambda, \{q_0\}, 0)$ r_8 [1]

5

V_2 9 $(q_4, \lambda, \{q_0\}, 0)$ r_9 [1]

$a{:}\{$ $a{:}[$ $\{$ $[$

10   $(q_1, \lambda, \{q_0\}, 1)$ r_{10} [2] $\Delta(V_1, a)$

11   $(q_3, \lambda, \{q_0\}, 1)$ r_{11} [2]

r_{14}

Figure 15.2: At the left, a copy of the ambiguous RTNSO of figure 14.1 (p. 284) and, at the right, execution trace of algorithm 15.1 rtnbo_translate_string_to_fprtn for this RTNSO and input *aabb*.

at state r_{10}:

$$(r_0, r_1\downarrow, r_2)$$

$$
\begin{array}{lllll}
(r_2, r_7\downarrow, r_3) & (r_3, r_6\downarrow, r_4) & (r_4, E, r_5) & (r_5, r_6\uparrow, r_6) & (r_6, r_7\uparrow, r_7) \\
(r_7, r_8\downarrow, r_3) & (r_3, r_6\downarrow, r_4) & (r_4, E, r_5) & (r_5, r_6\uparrow, r_6) & (r_6, r_8\uparrow, r_8) \\
(r_8, r_9\downarrow, r_3) & (r_3, r_6\downarrow, r_4) & (r_4, E, r_5) & (r_5, r_6\uparrow, r_6) & (r_6, r_9\uparrow, r_9) \\
(r_9, r_{10}\downarrow, r_3) & (r_3, r_6\downarrow, r_4) & (r_4, E, r_5) & (r_5, r_6\uparrow, r_6) & (r_6, r_{10}\uparrow, r_{10})
\end{array}
$$

Transitions have been aligned so that every push transition to state r_3 is placed in the leftmost column and every filtered-pop transition from call to r_3 is placed in the rightmost column. This path corresponds to the traversal of the RTNSO from state q_0 to state q_4 by consuming no input and generating sequence $EEEE$. This path is not an interpretation of input a: a last transition $(r_{10}, r_1\uparrow, r_1)$ is missing which is not realizable due to the different associated keys to states r_{10} and r_1.

Finally, figure 15.4 is an example of execution of algorithm 15.1 *rtnbo- translate_ string_ to_ fprtn* equivalent to the example of figure 14.5 (p. 298) for the left-recursive RTNSO. Even if the resulting FPRTN contains call cycles, it contains a unique and finite interpretation due to filtered-pop transitions:

$$(r_0, r_1\downarrow, r_2)(r_2, r_{2l}\downarrow, r_2)\ldots(r_2, r_8\downarrow, r_2)(r_2, r_6\downarrow, r_2)(r_2, r_4\downarrow, r_2)$$
$$(r_2, B, r_3)(r_3, r_4\uparrow, r_4)$$
$$(r_4, A, r_5)(r_5, r_4\uparrow, r_6)$$
$$(r_6, A, r_7)(r_7, r_4\uparrow, r_8)$$
$$\vdots$$
$$(r_{2l}, A, r_{2l+1})(r_{2l+1}, r_1\uparrow, r_1)$$

The first line contains the sequence of push transitions initializing the axiom call plus the $l-1$ successive calls to r_2; once the stack is filled with the right sequence of return states, transitions of the following lines consume an input symbol an pop out the next return state.

Chapter 16

Output FPRTNs

We study here the relevant particularities of the FPRTNs constructed by algorithm 15.1 *rtnbo_ translate_ string_ to_ fprtn* (described in section 15.6, p. 306), namely the circumstances that lead to FPRTNs that recognize infinite languages; such circumstances should be avoided in order to ensure finite results. We give a pruning algorithm for such FPRTNs in section 16.1, and two efficient language generator algorithms for FPRTNs representing finite languages in sections 16.2 and 16.3. We have briefly presented both the pruning algorithm and an adapted version of the first language generator algorithm in Sastre et al. (2009), in the context of application of the MovistarBot project. The pruning algorithm removes every useless substructure of the FPRTN (definition 120, p. 145), and consequently saves the cost of computing useless partial blackboards. The first language generator is able to avoid the exponential explosion in cases in which the grammar represents a set of sentences where the number of interpretations of each sentence is limited, even for sentences having an exponential number of local ambiguities (ambiguities that are solved after reading a certain amount of input). The second language generator is intended to be as efficient as possible for the worst case; furthermore, this second algorithm will be the base for the construction of another algorithm that definitively avoids the exponential explosion in cases in which the grammar is a weighted machine and only the top ranked blackboard is needed (to be described in chapter 18). Note that many applications require a single interpretation to be returned in spite of ambiguity, for instance automatic translators and conversational agents such as the MovistarBot.

As stated before, paths within O-FPRTNs correspond to the RTN Earley-

V_0

Initial_set

$\cup\, C_\varepsilon(\text{Initial\_set})$

$\Delta(V_0, a)$

V_1

q_0

$\{q_5\}$

q_1

$\{q_5\}$

q_2

$\{q_5\}$

q_3

$\{q_5\}$

q_4

q_5

$a : A$ $\{q_7\}$

q_6

q_7

$\varepsilon : E$

q_8

2 $(q_0, \lambda, \{q_0\}, 0)$

$(q_1, \{q_5\}, \{q_0\}, 0)$

3 $(q_5, \lambda, \{q_5\}, 0)$

$(q_6, \{q_7\}, \{q_5\}, 0)$

4 $(q_7, \lambda, \{q_7\}, 0)$

5 $(q_8, \lambda, \{q_7\}, 0)$

6 $(q_6, \lambda, \{q_5\}, 0)$

7 $(q_1, \lambda, \{q_0\}, 0)$

$(q_2, \{q_5\}, \{q_0\}, 0)$

8 $(q_2, \lambda, \{q_0\}, 0)$

$(q_3, \{q_5\}, \{q_0\}, 0)$

9 $(q_3, \lambda, \{q_0\}, 0)$

$(q_4, \{q_5\}, \{q_0\}, 0)$

10 $(q_4, \lambda, \{q_0\}, 0)$

11 $(q_6, \lambda, \{q_5\}, 0)$

$\{4\}$

$\{3\}$

$\{3\}$

$\{3\}$

$\{3\}$

$\varepsilon : E$

$a : A$

r_0 $\boxed{0}$

$\{r_2\}$ r_1

r_2 $\boxed{0}$

r_{12}

r_7

$\{r_3\}$ r_3 $\boxed{0}$

r_6

$\{r_4\}$ r_4 $\boxed{0}$

E

r_5 $\boxed{0}$

r_6 r_8 $\boxed{0}$

r_{13}

r_7 $\boxed{0}$

r_9

$\{r_3\}$ r_{14}

r_8 $\boxed{0}$

r_{10}

$\{r_3\}$

r_9 $\boxed{0}$

r_{15}

$\{r_3\}$

A

r_{10} $\boxed{0}$

$\{r_3\}\{r_3\}\{r_3\}$ $\{r_3\}$

r_{11} $\boxed{1}$

Figure 15.3: At the left, a copy of the RTN with deletable calls of figure 12.10(a) (p. 251) but with added output and, at the right, execution trace of algorithm 15.1 rtnbo_translate_string_to_fprtn for this RTNSO and input a; pop transitions with arrow tails are created by the ε-completer.

Figure 15.4: At the left, a copy of the left-recursive RTN of figure 12.11 (p. 252) but with added output and, at the right, execution trace of algorithm 15.1 rtnbo_translate_string_to_fprtn for this RTNSO and input ba^l.

like execution paths of a RTNBO for a given finite input sequence, the RTNBO being applied as a RTN but labeling the transitions of the resulting FPRTN with the RTNBO output labels. If we consider the input segment as a linear FSA (see definition 77, p. 130), the FPRTN is a factored representation of the intersection of the languages represented by the input FSA and the RTNBO, that is, a substructure of the execution machine of the RTNBO conditioned upon the input FSA. Boullier and Sagot (2007) present a set of filters to be applied to a CFG for a given input sequence in order to reduce the search space when applying the CFG, for instance the suppression of any grammar rule consuming an input symbol not present in the input sequence. From that point of view, our algorithm performs some kind of filtering of the grammar represented by the RTNBO so that only the paths consuming some prefix of the input remain. It is not a full filtering since not all these paths may be useful for the recognition of the entire input: we still need to prune the resulting FPRTN in order to remove the useless substructures. We apply the RTNBO rather than individually applying a filter to each transition, though the cost of individually applying one or more of the suggested filters plus the application of our algorithm might be less than the application of the algorithm without the previous filtering of individual transitions. The application of the filters proposed by Boullier and Sagot (2007) is left to a future work.

Definition 272 (Output FPRTN). *We say a FPRTN A is an output FPRTN (O-FPRTN) iff there exists a RTNBO B and an input sequence w such that B is the result of the execution of algorithm 15.1 rtnbo_translate_string-_to_fprtn (p. 309) for B and w, and we call (B, w) a source of A.*

Definition 273 (Canonical source of an output FPRTN). *We say a source (B, w) of an O-FPRTN A is a canonical source iff every path of B is explored and every symbol of w is consumed for the generation of A, independently of whether the language of the resulting FPRTN is empty or not.*

Note that all the examples of execution given here are based on canonical sources in order to keep them small. In practice, only a subset of the RTNBO will be explored and the input sequence will not be necessarily recognized. Recall that consuming every input symbol is not a sufficient condition for the generation of a FPRTN recognizing a non-empty language: at least one acceptance ES is also to be reached.

Lemma 26 (Output FPRTN cycles). *Given a canonical source (B, w) of an O-FPRTN A, a path p in A is a cycle iff there exists a path p' in B holding the following properties:*

- *p' is a subpath of some path p'' realizable from an initial ES by consuming some prefix of w,*

- *p' is an ε-cycle, and*

- *during the traversal of p'', the last called SS and the number of input symbols consumed when starting that call are the same either when reaching the start or the end states of p'.*

Moreover, p is a consuming cycle iff p' is a generating path, and p is useful iff p' is w-useful (definition 119, p. 144).

Proof. Obviously, the first condition must hold since paths are added to A as some prefix of w is consumed by B. Let $w_a w_b w_c = w$, $p_a p_b p_c$ be a path in B such that it is realizable from some RTN Earley-like initial ES x_ε of B by consuming w, where execution path $\mathcal{X}(p_a, x_\varepsilon)$ (definition 90, p. 134) reaches ES x_a by consuming w_a, $\mathcal{X}(p_b, x_a)$ reaches ES x_b by consuming w_b and $\mathcal{X}(p_c, x_b)$ reaches ES x_c by consuming w_c. Proving the remaining conditions consists in proving that they hold iff $\mathcal{X}(p_b, x_b)$ is a cycle, that is, $x_a = x_b$ and both x_a and x_b belong to the same SES. Let $x_a = (q_s, Q_c, Q_h, i)$ and $x_b = (q'_s, Q'_c, Q'_h, j)$. Path p_b is a cycle iff $q_s = q'_s$. $Q_c = Q'_c = \lambda$ since only active ESs correspond to FPRTN states. The third condition holds iff $Q_h = Q'_h$ and $i = j$. Finally, x_a and x_b belong to the same SES iff p_b does not consume input.

The two additional propositions are obvious: since FPRTN input symbols are copies of RTNBO output symbols, p consumes iff p' generates and, by construction, every interpretation of w in B produces an interpretation of some translation of w in A, thus relating the usefulness of p and p'. $\square$

Lemma 27 (Possible recursion degrees). *Given a source (B, w) of an O-FPRTN A, the recursion degree of A is infinite iff B contains a w-useful deletable recursion (definition 202, p. 226); otherwise it is either 0 or 1.*

Proof. The key of this proof is that OS keys represent the number of input symbols consumed during the traversal of the source RTNBO up to the generation of the OSs. For an O-FPRTN A to have an infinite recursion degree,

A must contain a path p such as the one shown in proof of lemma 25 (p. 304),

$$p = p_a \ (q_{s_1}, q_{r_1}\downarrow, q_{c_1}) \ p_b \ (q_{s_2}, q_{r_2}\downarrow, q_{c_1}) \ p_c \ (q_{f_3}, q_{r_2}\uparrow, q_{r_2}) \ p_d \ (q_{f_2}, q_{r_1}\uparrow, q_{r_1}) \ p_e,$$

where $p_b \ (q_{s_2}, q_{r_2}\downarrow, q_{c_1})$ is a call cycle and keys k_{f_2} and k_{r_2} are equal. For (B, w) to be a source of A, B must contain a substructure that is explored by consuming some prefix of w, generating A as is. As stated in the previous proof, a cycle in A implies a corresponding explored ε-cycle in B and, since the cycle $p_b \ (q_{s_2}, q_{r_2}\downarrow, q_{c_1})$ in A is a call cycle, the ε-cycle in B must be a call ε-cycle. Since $k_{f_2} = k_{r_2}$ and A contains a path p_d deriving OS q_{f_2} from OS q_{r_2}, the corresponding path within B cannot either consume any input symbol; therefore, the path in B corresponding to p in A is a deletable recursion. Finally, path p in A must be useful, which —by construction— is only possible if the corresponding path in B is w-useful. $\qquad\square$

Theorem 28 (Cardinality of the interpretation set). *Given theorem 26 (p. 305), the number of interpretations of an O-FPRTN with source (B, w) is infinite iff B contains at least one w-useful ε-cycle p holding at least one of the following conditions:*

- *every call initiated within p is completed as well within p, or*

- *there exists an ε-path p' such that pp' is a deletable recursion.*

Theorem 29 (Cardinality of the language). *The language of an O-FPRTN with source (B, w) is infinite iff B contains at least one w-useful generating ε-cycle holding at least one of the conditions mentioned in the previous theorem.*

We must expect a natural language grammar to associate several interpretations to a natural language sentence since natural languages are ambiguous; however, associating infinite interpretations to a natural language sentence makes no sense. Therefore, forbidding the presence of generating ε-cycles in RTNBOs does not restrict the natural languages that can be represented but ensures that the language represented by the resulting O-FPRTNs will be finite since this restriction and the one of the previous theorem are mutually exclusive.

16.1   Pruning

Let A be an O-FPRTN obtained from a source (B, w); by construction, every state in A is reachable from the initial state of A though it may contain

useless states and transitions. Before generating the language represented by A, we prune it so that we save the cost of processing useless paths, and consequently the generation of useless blackboards. By construction, the last transition of every possible interpretation within A is an explicitly defined filtered-pop transition having the "global" acceptor state r_f as target. If B does not recognize w but is only able to consume some prefix of w, A will have no filtered-pop transitions towards r_f. Since A contains no interpretations, every state and transition is useless and is to be removed. Otherwise, we can reversely traverse every interpretation from state r_f towards the initial state and mark every reached state, so that the remaining unmarked states will be the ones to remove as well as every transition having any of these states as either source or target.

Algorithm 16.1 *output_fprtn_prune* removes every useless state and transition of a given O-FPRTN with explicitly defined filtered-pop transitions, following the procedure described above. In order to optimize the reverse traversal of the O-FPRTN, we store at each state object its set of incoming transitions as well as its set of outgoing transitions. The algorithm first checks for the existence of explicitly defined filtered-pop transitions incoming to state r_f and, if none found, calls procedure *clear* in order to perform an indiscriminate removal of states and transitions;[1] otherwise, it proceeds with a selective pruning. The algorithm builds a function ζ mapping states to Booleans which returns whether a state has already been reversely reached from r_f or not, and initializes it with a false value for every state.[2] The algorithm also keeps a queue E of reversely reached but unexplored states, that is, states whose incoming transitions are still to be reversely traversed. The selective pruning starts by marking r_f as reached and enqueuing it into the queue of states to be explored. Then, for each enqueued state it dequeues the next one and reversely traverses one by one its incoming consuming transitions, ε-transitions, call transitions and explicitly defined filtered-pop transitions. Note that through a pop transition we may reversely reach states of a call up to its subinitial states, but no further. By reversely traversing call transitions as well we skip the whole call traversal and directly reach the state before the call. Note that call transitions are added once it is proved that they are realizable, except for the call transition pointing to r_f; how-

---

[1]Note that, in practice, the memory allocated for the O-FPRTN is to be freed sooner or later, so clearing the O-FPRTN is not a waste of time.

[2]In practice, we just extend FPRTN state objects with a Boolean field so that we do not have to search in a map, assigning to it a false default value.

ever, if this call is not realizable then no filtered-pop transitions will reach r_f and therefore the entire FPRTN will be deleted. For each transition, routine *enqueue_ mark_ unexplored_ os* (algorithm 16.2) is called in order to add the reversely reached state to the queue and mark it as reached, if it was not previously reached: since the FPRTN may contain cycles, the algorithm must perform this check before enqueuing the state in order not to fall into an infinite loop. Finally, a last loop iterates over every FPRTN state and removes the unmarked ones as well as their corresponding transitions. Note that, since we are storing incoming as well as outgoing transitions, for each outgoing transition t stored in a state object q_s there is a corresponding incoming transition t' in some state q_t. In order to accelerate transition removal, we also store in the transition data structures the reference towards the corresponding reverse transition data structure.

16.2 Language generation

There are cases in which we can expect the language of the resulting O-FPRTN to be small; for instance, when building a grammar for a particular domain of application one may try to represent only the interpretation of each ambiguous sentence that a human would assume in that context (e.g.: upon receiving a sentence such as '*envía al móvil* 555-555-555 *hola Paco*', which means 'send to the mobile 555-555-555 hello Paco', the MovistarBot should assume that the user is asking to send the SMS '*hola Paco*' to the mobile phone '555555555' and not the SMS '*al móvil* 555-555-555 *hola Paco*' to an unspecified phone number). Once pruned, the resulting O-FPRTN will contain a small number of paths (the possible interpretations), even for grammars representing an exponential number of local ambiguities (ambiguities that are solved after reading enough input symbols): the pruning operation will efficiently remove the O-FPRTN substructures corresponding to local misinterpretations. An efficient language generator algorithm for such cases (low global ambiguity with high/low local ambiguity) can be obtained by modifying algorithm 14.2 *rtn_ language* (p. 287), the breadth-first language generator for RTNs, as follows:

- ESs contain a blackboard instead of a string,

- explicit ε-transitions and consuming transitions are labeled with an output function on blackboards instead of the empty symbol or an

---

Algorithm 16.1 output_fprtn_prune

Input: $A = (Q', K, \Gamma, \delta', \kappa, Q'_I, F')$, the output FPRTN to prune

        r_f, the "global" acceptor state of A

Output: A after removing every useless state and transition

 1: **if** $\nexists r_s : \delta'(r_s, r_f\uparrow) = r_f$ **then**

 2:    clear(A)

 3: **else**

 4:    **for each** $r \in Q'$ **do**

 5:        $\zeta(r) \leftarrow$ false

 6:    **end for**

 7:    $\zeta(r_f) \leftarrow$ true

 8:    $E \leftarrow \emptyset$

 9:    enqueue(E, r_f)

10:    **while** $E \neq \emptyset$ **do**

11:        $r_t \leftarrow$ dequeue(E)

12:        **for each** $(r_s, \gamma) : r_t \in \delta'(r_s, \gamma)$ **do**

13:            enqueue_mark_unexplored_os(E, ζ, r_s)

14:        **end for**

15:        **for each** $r_s : r_t \in \delta'(r_s, \varepsilon)$ **do**

16:            enqueue_mark_unexplored_os(E, ζ, r_s)

17:        **end for**

18:        **for each** $(r_s, r_c) : r_t \in \delta'(r_s, r_c)$ **do**

19:            enqueue_mark_unexplored_os(E, ζ, r_s)

20:        **end for**

21:        **for each** $r_s : r_t \in \delta'(r_s, r_t\uparrow)$ **do**

22:            enqueue_mark_unexplored_os(E, ζ, r_s)

23:        **end for**

24:    **end while**

25:    **for each** $r \in Q' : \neg\zeta(r)$ **do**

26:        remove_state_and_associated_transitions(A, r)

27:    **end for**

28: **end if**

Algorithm 16.2 enqueue_mark_unexplored_os

Input: E, the queue of OSs to explore

$\zeta : Q' \to \mathbb{B}$, a Boolean function returning whether a state in Q' has precedently been reached or not

r, a recently reached OS

Output: ζ, after setting r as reached, if necessary

E, after enqueuing r, if necessary

1: **if** $\neg\zeta(r)$ **then**
2:     $\zeta(r) \leftarrow$ true
3:     enqueue(r)
4: **end if**

---

input symbol, and their traversal applies the output function to the blackboard of the source ES instead of appending a symbol,

- the key-matching condition is to be added to the treatment of popping transitions, that is, popping from an state q_s with a stack π to a state q_r requires $\kappa(q_s) = \kappa(q_r)$ as well as $\pi = \pi' q_r$ and $q_s \in F$, and

- when computing the set of derived ESs from an ES x, derived ESs are modified copies of x except for the last derived ES, which keeps the original data structure representing x.

Note that, when the language to generate contains a unique element, this algorithm builds a unique ES data structure and simply modifies its blackboard field each time a transition is traversed, instead of building a new ES data structure containing a modified copy of the source blackboard. At a first stage of development of the MovistarBot, this algorithm was used in conjunction with a low global-ambiguity grammar and, upon ambiguous sentences, interpretations were chosen randomly: we selected the first one (whatever it corresponded to) from the list of possible interpretations.

16.3 Language generation through BSP

We apply here the equations for Earley-like BSP of RTNBOs for computing the language represented by an O-FPRTN. First of all, we give a definition of topological sort of the O-FPRTN, then study the necessary conditions for

its existence, and finally give an algorithm computing the language of the O-FPRTN.

Definition 274 (Topological sort of an O-FPRTN). *Assuming that calls within a RTNBO are to be explored only once, such as it is done by algorithm 15.1* rtnbo_translate_string_to_fprtn *(309), we define the topological sort of the resulting O-FPRTN as for FSMs (definition 81, p. 130) but redefining relation R as follows:*

- *explicit ε-transitions, consuming transitions, call transitions and realizable filtered-pop transitions from r_s to r_t imply $r_s \, R \, r_t$, and*

- *$r_a \, R \, r_b \wedge r_b \, R \, r_c$ implies $r_a \, R \, r_c$.*

Let $t = (r_s, R_c, r_t)$ be a call transition within an O-FPRTN that is realizable through a path p starting at a state $r_c \in R_c$ and having a last transition of the form $(r_f, r_t\!\uparrow, r_t)$. If the O-FPRTN is to be traversed by following a topological sort, it is obvious that r_s and r_f are to be explored before r_t. Transitively, states in p before r_f are also to come before r_t. However, it makes no difference whether r_s is explored before states in p or the converse, since the call to R_c is computed as an independent application of the machine but taking R_c as set of initial states. Let $t' = (r'_s, R_c, r'_t)$ be a second call to the same SS R_c; depending on how t and t' are placed w.r.t. each other, we distinguish 3 feasible cases:

- Transitions t and t' are parallel calls (e.g.: call transitions $(r_1 0, \{r_1 2\}, r_1 4)$ and $(r_1 1, \{r_1 2\}, r_1 5)$ in figure 15.2, p. 317); in this case, the following relations would be defined in R:[3]

    - $r_s \, R \, r_t$,
    - $r_c \, R \, r_f \, R \, r_t$,
    - $r'_s \, R \, r'_t$ and
    - $r_c \, R \, r_f \, R \, r'_t$.

- Transitions t and t' are non-alternating sequential calls, that is, t comes before t' but t' does not come before t (e.g.: call transitions $(r_1 2, \{r_1 6\}, r_1 3)$ and $(r_1 3, \{r_1 6\}, r_1 4)$ in figure 15.3, p. 321); the corresponding relations defined in R are:

---

[3]For the sake of simplicity, we have abused here the notation of R as done in inequality expressions such as $3 \le x \le y \le 9$ (instead of $3 \le x \wedge x \le y \wedge y \le 9$).

- $r_s \, R \, r_t \, R \, r'_s \, R \, r'_t$ and
- $r_c \, R \, r_f \, R \, r_t \, R \, r'_t$.

- Finally, transition t starts call to R_c, and the completion of the call involves to traverse transition t' (call to R_c is recursive), but filtered-pop transitions do not allow for repeated completions of t' (e.g.: analogous to call transitions $(r_0, \{r_2\}, r_1)$ and $(r_2, \{r_2\}, r_4)$ in figure 12.11, p. 252, but not necessarily requiring that t' starts at r_2 but some transitions later). In this case, an additional path completing call to R_c without further recursion is needed in order to stop the recursion introduced by transition t' (e.g.: in the figure of the previous example, path $r_2 \xrightarrow{B} r_3 \xrightarrow{r_4\Updownarrow} r_4$, but not necessarily requiring that the path starts at the same state than the path traversing t'). Let this path start at a state r'_c and have a last transition of the form $(r'_f, r'_t\uparrow, r'_t)$, the relations in R are:

  - $r_s \, R \, r_t$,
  - $r_c \, R \, r'_s \, R \, r'_t \, R \, r_f \, R \, r_t$, and
  - $r'_c \, R \, r'_f \, R \, r'_t$.

Transition t would initiate the exploration of call to R_c and, once traversed path from r_c to r'_s, transition t' would not initialize any call but just wait for the exploration of path from r'_c to r'_f.

Note that we have defined the topological sort for entire O-FPRTNs and not only for their ε-closure-substructures. In the previous sections on BSP we focused on topologically sorting the ε-closure-substructures since a topological sort for Δ-substructures was already given by the sequence of SESs the different ESs belonged to: consuming transitions derive target ESs x_t from source ESs x_s with $x_s \in V_i$ and $x_t \in V_{i+1}$. We first explored the Δ-substructure of a SES V_i, then the ε-closure-substructure derived from the Δ-substructure by following a topological sort, then the Δ-substructures of V_{i+1} and so on. Once the O-FPRTN is computed, the key associated to each state gives us the index of the SES the corresponding ES belonged to, but the information about which states where produced by the scanner and which ones where produced by the other algorithm components is lost. Since it is necessary to explore the entire O-FPRTN in order to generate its language (provided that it has been previously pruned, and therefore contains no useless substructures), we explore it by following the previously defined

topological sort for entire O-FPRTNs, which is not necessarily the same than the one given by the sequence of SESs plus the topological sort of the ε-closure-substructures.

Theorem 30 (Existence of a topological sort for output FPRTNs). *Considering lemma 1 (p. 131) and theorems 21 (p. 266) and 28, for every RTNBO not having deletable recursions or ε-cycles involving deletable calls and/or output generation there exists an equivalent RTNBO B such that there exists a topological sort for the O-FPRTN generated with source (B, w), for every input sequence w of B.*

Note that forbidding the presence of deletable recursions in a RTNBO does not restrict the set of grammars that can be represented since such paths do not contribute anything to the grammar description (they are equivalent to CFG rules of the form $A \to A$). The same applies for ε-cycles involving deletable call completions. Finally, ε-cycles with output generation are forbidden since they lead to grammars representing sentences with infinite interpretations, which make no sense.

Algorithm 16.3 *output_fprtn_ bsp_ earley_ language* efficiently computes the language represented by a pruned O-FPRTN, based on the RTN Earley-like language generator algorithm in section 14.8 (p. 294), the RTNBO Earley-like BSP equations in section 13.11, the topological sort for O-FPRTNs and Kahn's (1962) topological sorter (a brief description of Kahn's algorithm can be found in appendix D, page 419). The algorithm computes the translations of the empty word, considering every transition of the O-FPRTN as a transition recognizing the empty symbol and applying the associated γ function to the current output blackboard; therefore the algorithm is reduced to the computation of the ε-closure of a set of initial ESs. Moreover, it performs a blackboard set processing (BSP) of the O-FPRTN, that is, it traverses the entire O-FPRTN by following a topological sort, computing every blackboard that can be generated by reaching each particular state q before computing the blackboards of every reachable state from q. The topological sort is computed during the algorithm application as for Kahn's algorithm. Another algorithm for computing a topological sort is described in Cormen et al. (2001, sec. 22.4). Opposite to Kahn's algorithm, this algorithm does not require to compute first the list of unreachable nodes from any other one so that the exploration of the graph is performed from these nodes by following the own topological sort; however, these nodes are known before computing

the language of the O-FPRTN, the initial state r_I, and our language genera-
tor algorithm requires to follow the topological sort as for Kahn's algorithm
in order to correctly compute the language of blackboards.

Rather than performing again an Earley-like processing of the O-FPRTN,
the algorithm takes advantage of the computations already performed to
build the O-FPRTN: the algorithm takes an additional input parameter ζ_s''
which represents a function mapping each pop transition to the set of source
states of the call transitions it completes.[4] This requires the following mod-
ifications in algorithm 15.6 *rtnbo_ interlaced_ eclosure_ to_fprtn*:

- right before the ε-completer inside the completer, insert instruction
  "$\zeta_s''((r_s, r_r\uparrow, r_r)) \leftarrow \zeta_s''((r_s, r_r\uparrow, r_r)) \cup R_s'$" in order to register the source
  states of normally completed call transitions,

- in the ε-completer of the completer, replace instruction "add(T, Q_h)"
  by "add(T, Q_h, r_s)" in order not only to mark deletable calls but also to
  build the corresponding list of acceptor states triggering ε-completions,

- in the ε-completer of the predictor, replace instruction "**else if** $Q_c \in T$
  then" by "**else if** $\exists r_f : (Q_c, r_f) \in T$ **then**", since elements in T are no
  longer elements Q_c but pairs (Q_c, r_f), and insert inside this "**else if**"
  block a block "**for each** $r_f : (Q_c, r_f) \in T$ **do**" with a unique instruction
  "add$(\zeta_s''((r_f, r_r\uparrow, r_r)), r_s)$" in order to register the source states of ε-
  completed call transitions.

Map ζ_s'' is to be defined as an output parameter of algorithm 15.1 *rtnbo-
translate string_ to_fprtn* and, as the other maps, it is treated as a global
variable and implicitly initialized as an empty map.

The algorithm first creates two maps, ζ_n and ζ_B; the former maps each
state to a counter of unexplored incoming transitions to the state, namely
consuming transitions, explicit ε-transitions, call transitions and pop tran-
sitions but not push transitions. The latter map associates each state to
an initially empty SB. Then it initializes call to r_I by adding the empty
blackboard to the SB of r_I and by enqueuing r_I. States are dequeued and
processed one by one, following a topological sort, until the queue is empty.

---

[4]In practice, we extend pop-transition objects with a field containing a reference to-
wards the corresponding set and, when completing a call during the construction of the
O-FPRTN, the set Q'' is filled with the source states associated to every resumed paused
ES.

At each iteration, r_s represents the dequeued state and B_s its SB. First of all, B_s is incremented with every blackboard composition $b'_s \circ b_f$ such that $(b'_s, b_f) \in \zeta_B(r'_s) \times \zeta_B(r_f)$, for every pair of states (r_f, r'_s) such that r_s is the return state of a call transition having r'_s as source state and r_f as a possible acceptor state completing it. Of course, if r_s is not a return state of any call then no blackboard is added to B_s. Then the outgoing transitions of r_s are explored, namely consuming transitions, explicit ε-transitions, call transitions and pop transitions. Let r_t be the target state of these transitions; in every case the counter $\zeta_n(r_t)$ is decremented, and r_t is enqueued iff the new value of $\zeta_n(r_t)$ is zero since, in that case, every blackboard to be added to $\zeta_B(r_t)$ should already have been added, except for the blackboards due to call completions which are added to r_t right after dequeuing it. Each particular case performs the following additional operations:

- for each consuming transition (r_s, γ, r_t), blackboards $\gamma(B_s)$ are added to $\zeta_B(r_t)$,

- for each explicit ε-transition$(r_s, \mathrm{id}_B, r_t)$, blackboards B_s is added to $\zeta_B(r_t)$,

- for each call transition (r_s, R_c, r_t), call to R_c is initiated if it has not been done yet (SBs of every $r_c \in R_c$ will be empty, thus it suffices to check only the first element) and not every transition incoming to r_t has already been explored:[5] for each state $r_c in R_c$ the empty blackboard is added to $\zeta_B(r_c)$ and r_c is enqueued if its counter is zero,[6] and

- no additional operation is performed for each pop transition $(r_s, r_t\uparrow, r_t)$ since composed blackboards are added right after dequeuing each state.[7]

Once every state in the O-FPRTN has been processed, the SB of the last explored state is returned, which, by construction, corresponds to the "global"

---

[5]Note that having explored every incoming transition to r_t implies that every pop transition completing call to R_c has also been explored and therefore call to R_c has already been initiated.

[6]Note that subinitial states in R_c might be reachable from other subinitial states in R_c, so only the unreachable ones should be enqueued at this moment.

[7]Note that reaching an acceptor state that triggers a call completion does not necessarily ensure that every source state of every other call whose completion it might also trigger has already been visited, hence the algorithm does not compute the composed blackboards for a given state r_s until r_s is dequeued.

acceptor state of the O-FPRTN. Note that, as long as the O-FPRTN is pruned before the application of this algorithm, the "global" acceptor state is the only one from where no other state can be reached. We have followed a non-destructive method for the computation of a topological sort of the O-FPRTN by means of associating counters to each state. Kahn's algorithm removes each traversed edge, and enqueues a node once it has no incoming edges. In case there is nothing to be done with the O-FPRTN once its language is computed, the destructive method would be preferred since, anyway, the memory allocated by the O-FPRTN transitions is to be freed sooner or later (as stated in footnote 1, p. 1).

Algorithm 16.3 output_fprtn_bsp_earley_language(A)

Input: $A = (Q', K, \Gamma, \delta', \kappa, \{r_I\}, F')$, an output FPRTN
       ζ''_s, a function mapping each pop transition to the set of source states
  of the call transitions that it completes
Output: L, the language of A

 1: $E \leftarrow \emptyset$
 2: **for each** $r_t \in Q'$ **do**
 3:      $\zeta_n(r_t) \leftarrow |\{(r_s, \gamma, r_t) : r_t \in \delta'(r_s, \gamma)\}|+$
                     $|\{(r_s, \mathrm{id}_B, r_t) : r_t \in \delta'(r_s, \mathrm{id}_B)\}|+$
                     $|\{(r_s, R_c, r_t) : r_t \in \delta'(r_s, R_c)\}|+$
                     $|\{(r_s, r_t\uparrow, r_t) : r_t \in \delta'(r_s, r_t\uparrow)\}|$
 4:      $\zeta_B(r_t) \leftarrow \emptyset$
 5: **end for**
 6: add($\zeta_B(r_I), b_\emptyset$)
 7: enqueue(E, r_I)
 8: **while** $E \neq \emptyset$ **do**
 9:      $r_s \leftarrow$ dequeue(E)
10:      $B_s \leftarrow \zeta_B(r_s)$
11:      **for each** $r_f : r_s \in \delta'(r_f, r_s\uparrow)$ **do** ▷ BLACKBOARD COMPOSITION
12:          **for each** $r'_s \in \zeta''_s((r_f, r_s\uparrow, r_s))$ **do**
13:              **for each** $(b'_s, b_f) \in \zeta_B(r'_s) \times \zeta_B(r_f)$ **do**
14:                  add($\zeta_B(r_s), b'_s \circ b_f$)
15:              **end for**
16:          **end for**
17:      **end for**
18:      **for each** $(r_t, \gamma) : r_t \in \delta'(r_s, \gamma)$ **do** ▷ CONSUMING TRANSITIONS
19:          add($\zeta_B(r_t), \gamma(B_s)$)
20:          $\zeta_n(r_t) \leftarrow \zeta_n(r_t) - 1$
21:          **if** $\zeta_n(r_t) = 0$ **then**
22:              enqueue(E, r_t)
23:          **end if**
24:      **end for**

25:     **for each** $r_t \in \delta'(r_s, \mathrm{id}_B)$ **do** $\triangleright$ EXPLICIT ε-TRANSITIONS
26:          $\mathrm{add}(\zeta_B(r_t), B_s)$
27:          $\zeta_n(r_t) \leftarrow \zeta_n(r_t) - 1$
28:          **if** $\zeta_n(r_t) = 0$ **then**
29:              $\mathrm{enqueue}(E, r_t)$
30:          **end if**
31:     **end for**
32:     **for each** $(r_t, R_c) : r_t \in \delta'(r_s, R_c)$ **do** $\triangleright$ PUSH TRANSITIONS
33:          $\zeta_n(r_t) \leftarrow \zeta_n(r_t) - 1$
34:          **if** $\zeta_n(r_t) = 0$ **then**
35:              $\mathrm{enqueue}(E, r_t)$
36:          **else if** $\zeta_B(\mathrm{first}(R_c)) = \emptyset$ **then**
37:              **for each** $r_c \in R_c$ **do**
38:                  $\mathrm{add}(\zeta_B(r_c), b_\emptyset)$
39:                  **if** $\zeta_n(r_c) = 0$ **then**
40:                      $\mathrm{enqueue}(E, r_c)$
41:                  **end if**
42:              **end for**
43:          **end if**
44:     **end for**
45:     **for each** $r_t : r_t \in \delta'(r_s, r_t\uparrow)$ **do** $\triangleright$ POP TRANSITIONS
46:          $\zeta_n(r_t) \leftarrow \zeta_n(r_t) - 1$
47:          **if** $\zeta_n(r_t) = 0$ **then**
48:              $\mathrm{enqueue}(E, r_t)$
49:          **end if**
50:     **end for**
51: **end while**
52: $L \leftarrow \zeta_B(r_s)$

Chapter 17

Finite-state machines with composite output

We present here FSMs with composite output (FSMCO) as an extension of FSMs with blackboard output for the generation of multiple outputs, either of the same kind or not. FSMCOs can be seen as machines with multiple output tapes: blackboards are structures having a field for each output tape, and each output tape is itself another kind of blackboard.

Definition 275 (FSMCO). *In general, finite-state machines with composite output(FSMCOs) are a particular case of FSMs with blackboard output with*

- $B = (B_0 \times B_1 \times \ldots \times B_n)$, *that is, blackboards $b \in B$ are composite blackboards $(b_0, b_1, \ldots, b_n) \in (B_0 \times B_1 \times \ldots \times B_n)$,*

- $\Gamma = \Gamma_0 \times \Gamma_1 \times \ldots \times \Gamma_n$, *that is, functions $\gamma \in \Gamma$ are composite functions $\gamma(b) = (\gamma_0(b_0), \gamma_1(b_1), \ldots, \gamma_n(b_n))$ that operate on composite blackboards $b = (b_0, b_1, \ldots, b_n)$,*

- $B_K = \{(b_0, b_1, \ldots, b_n) : b_0 \in B_{K0} \vee b_1 \in B_{K1} \vee \ldots \vee b_n \in B_{Kn}\}$, *that is, composite blackboards $b \in B_K$ are those who have at least one killing term $b_i \in B_{Ki}$, and*

- $b_\emptyset = (b_{\emptyset 0}, b_{\emptyset 1}, \ldots, b_{\emptyset n})$, *that is, the empty composite blackboard is the one whose terms are all empty.*

Chapter 18

Weighted finite-state machines

We present here weighted machines as a special case of blackboard processing, analogously to the way in which we have derived FSMs with string output as a particular case of FSMs with blackboard output, namely FST-SOs (chapter 11) from FSTBOs (chapter 10) and RTNSOs (chapter 14) from RTNBOs (chapter 13): blackboards are weights, we assume that there are no killing weights, and functions on blackboards may increment or decrement such weights. Weights represent the cost, score or probability associated to the realization of a transition. We use here weight output in order to define a criterion to select a preferred interpretation upon ambiguous sentences: the cheapest, top-ranked or most likely one, depending on what the weights represent. In the MovistarBot use case (section 1.2, 6), grammars are RTNs with composite output: scores and XML tags that either identify the requested service or delimit the arguments to be extracted; for instance, the expected XML output for sentence

- 'envía Feliz Navidad al 555',

which means 'send Merry Christmas to the 555', is

- 'envía<sms/> <message>hola</message> al <phone>555</phone>'.

This output is to be coupled with a sequence of scores to be added, such that the resulting overall score is greater than those of other possible XML outputs for other sentence interpretations, such as

- 'envía<sms/> <message>Feliz Navidad al 555</message>'.

XML tags can be treated as string output, though we have implemented a slightly more complex kind of blackboard and treatment on blackboards for the sake of efficiency (to be described in chapter 20). Mainly, we consider in this chapter that there are no killing blackboards whatsoever. The problem of killing blackboards will be discussed in the next chapter.

Weights are defined by following some heuristic, thus the selected interpretation is not guaranteed to be the expected one. However, there are applications which require a single interpretation of each parsed sentence in spite of eventual mistakes, for instance machine translators and conversational agents. Since humans are used to deal with imprecise and/or inexact information, a human interested in the content of a text written in an unknown language will still find useful a partially correct machine translation. Human translators can use machine translators in order to partially automatize their work, having only to correct the output returned by the machine translator instead of typing the whole translation from the scratch. In case a chatterbot does not understand or misunderstands a request, the user may try to express his request in a different manner. Note that this kind of situation also happens between humans, though are usually less frequent than between humans and machines. The conversations held with a chatterbot are usually logged and studied by the chatterbot's administrator in order to improve the conversational rules for covering the possible deficiencies. As well, machine translator developers usually provide free online translation services (e.g.: http://translate.google.com) for gathering user translation requests, which are then studied for improving the translation rules.

Definition 276 (WFSM). *In general, weighted finite-state machines (WFSMs) are a particular case of FSMs with blackboard output so that*

- *given a partially-ordered group $(G, \bullet, \prec)$, for instance $(\mathbb{Z}, +, \leq)$ or $(\mathbb{R}^+, \cdot, \leq)$, functions in Γ always perform the binary operation $\bullet$ on an element of G and the current blackboard, which is another element of G; for the sake of simplicity, we consider that Γ contains elements in G rather than functions on blackboards, and output labels $g \in \Gamma$ represent the operation $b \bullet g$ where b is the current blackboard,*

- *the identity function on blackboards id_B performs operation $\bullet$ with its identity element and the current blackboard, for instance 0 for $(\mathbb{Z}, +, \leq)$ and 1 for $(\mathbb{R}^+, \cdot, \leq)$; we write the corresponding identity element instead*

of function id_B *in order to represent that a transition does not modify the current output,*

- $B = G$,

- $B_K = \emptyset$, *that is, there are no killing blackboards, and*

- $b_\emptyset$ *is the identity element of operator* $\bullet$, *for instance* 0 *for* $(\mathbb{Z}, +, \leq)$ *and* 1 *for* $(\mathbb{R}^+, \cdot, \leq)$.

Definition 277 (Weight of a path). *Given a path or sequence of concatenated transitions* $t_0 t_1 \ldots t_n$ *within a WFSM for an ordered group* $(G, \bullet, \prec)$ *so that* w_i *is the weight of transition* t_i, *the weight of the path is* $w_0 \bullet w_1 \bullet \ldots \bullet w_n$.

In the MovistarBot use case, weights represent scores rather than probabilities. Upon ambiguous sentences, the top-ranked output is to be assumed as the right interpretation. We use partially-ordered group $(\mathbb{Z}, +, \leq)$ in order to avoid floating-point operations.

Probabilistic machines (PFSMs), also called stochastic FSMs (SFSMs), are a special kind of weighted machines, though our definition of weighted machine can easily be adapted for PFSMs as follows, based on the definition of probabilistic automata given in Vidal et al. (2005a, sec. 2.2, p. 1015):

Definition 278 (PFSM). *A probabilistic machine (PFSM) is a weighted machine where*

- $(G, \bullet, \prec)$ *is to be defined as* $([0, 1], \cdot, \leq)$, *that is, weights are probabilities represented by real numbers between* 0 *and* 1,[1]

- Q_I *is to be replaced by a function* $P_I : Q \to \mathbb{R}^+$, *which represents the probability of each state to be an initial state,*

- F *is to be replaced by a function* $P_F : Q \to \mathbb{R}^+$, *which represents the probability of each state to be an acceptor state, and*

- *let* $P : (Q \times \Xi \times Q) \to \mathbb{R}^+$ *be the function returning the probability associated to each transition,* P *is to respect the following constraints*

---

[1]For efficiency, probabilities may also be represented by rational numbers, that is, as the quotient of two integer numbers.

so that the machine represents a probability distribution over the set of interpretations (definition 111, p. 144) it contains:

$$\sum_{q \in Q} P_I(q) \;=\; 1, \quad and \tag{18.1}$$

$$P_F(q_s) + \sum_{\xi \in \Xi, q_t \in Q} P(q_s, \xi, q_t) \;=\; 1, \quad \forall q_s \in Q. \tag{18.2}$$

In case the machine is non-deterministic, the probability of an input sequence is computed as the sum of the probabilities of every interpretation recognizing such sequence. In case the machine generates additional output (apart from probabilities), the probability of a translation (into other outputs than the probabilities) is computed as the sum of the probabilities of every interpretation performing such translation.

A more straightforward definition of probabilistic machine w.r.t. the definition of weighted machine can be given by modifying the previous definition as follows:

- instead of replacing the sets of states Q_I and F by functions P_I and P_F, two additional states q_I and q_F are to be added to Q,

- Q_I is to be defined as $\{q_I\}$,

- F is to be defined as $\{q_F\}$, and

- for each state $q \in Q$ two additional transitions are to be added:

  - a transition from q_I to q consuming no input and generating probability $P_I(q)$, and

  - a transition from q to q_F consuming no input and generating probability $P_F(q)$.

Examples of machines representing probability distributions over a set of sequences are weighted automata (Mohri, 1997), probabilistic suffix trees (Ron et al., 1994), probabilistic finite-state automata (Paz, 1971), stochastic or probabilistic automata (Carrasco and Oncina, 1994), hidden Markov models (Rabiner, 1989) and n-grams (Ney, 1992). In the MovistarBot use case, we have used weighted RTNs rather than probabilistic RTNs. We will

not give here more details on PFSMs, but a complete survey can be found in Vidal et al. (2005a,b).

In section 13.10 (p. 272) we presented an Earley-like algorithm of application of RTNBOs. In order to use this algorithm for the case of weight output, we define the weight composition operator as follows:

Definition 279 (Weight composition operator)**.** *Let A be a weighted RTN having $(G, \bullet, \prec)$ as partially ordered group; considering that operator $\bullet$ is associative (by definition of group), we define the blackboard composition operator (definition 239, p. 268) of A as $\bullet$ since it is a particular case of lemma 20 (p. 269).*

Definition 280 (Top path)**.** *Let p be a path within a WFSM A such that p recognizes a sequence α and has q_s and q_t as start and end states, respectively; we say p is a* top path *of A for (q_s, α, q_t) and ES x_s iff the weight generated by executing p from x_s is greater than or equal to the weight generated by the execution from x_s of any other path p' deriving q_t from q_s. We simply say that*

- *p is a top path of A for (α, q_t) when q_s is any initial state of A and x_s any initial ES,*

- *p is a top path of A for α when p and p' are interpretations of A recognizing α, and*

- *p is a top path of A, in general, when p and p' are interpretations of A recognizing any input.*

18.1 Weight assignment

Given two transitions outgoing from the same state of a FSM such that both transitions are realizable upon the same input and context of execution, one may express the preference of one transition over the other by assigning different weights to each transition. In the MovistarBot use case (section 1.2, p. 6), we have manually built a set of grammars —more or less descriptive— and automatically associated weights to the grammar transitions so that the most descriptive transitions —hence the most restrictive or specific— are preferred over those less descriptive. In section 6.4 (p. 115), we have studied the specificity of the different lexical masks, and proposed a weight to assign

to each transition depending on the lexical mask used as input label. This procedure has allowed our NLP engine to deal with ambiguous sentences, as described in the section.

Other possibility is to use a part-of-speech tagger, either stochastic (Church, 1988) or rule based (Brill, 1992), in order to automatically associate weights to transitions during the application of the machine for a given input. Part-of-speech taggers compute the most likely part-of-speech of the words of a sentence. We may associate higher scores to transitions requiring or, at least, not forbidding the part-of-speech chosen by the tagger.

18.2   Extracting the top blackboard of a weighted-output FPRTN

In section 15.6 we presented an algorithm able to compute the set of translations for a given RTNBO and input as an O-FPRTN in time n^3 —in the worst case— even for RTNBOs generating an exponential number of outputs w.r.t. the input length. In chapter 16 we presented two procedures for the generation of the language of outputs represented by an O-FPRTN. Obviously, generating the language of outputs of an O-FPRTN representing an exponential number of outputs will have an exponential worst-case cost. However, end-user applications such as machine translators and chatterbots (namely the MovistarBot), require only a single output to be returned, let it be the most likely or the top-ranked one. We present here an algorithm that is finally able to generate only the top-ranked output represented by a weighted O-FPRTN (WO-FPRTN) in time n^3. Recall that O-FPRTNs are built from a source (B, w) (definition 272, p. 323), and that O-FPRTN input labels are simple copies of the output labels of their respective source RTN-BOs (section 15.6, p. 306). Therefore, we define WO-FPRTNs as follows:

Definition 281 (Weighted-output FPRTN). *Let (B, w) be the source of an output FPRTN A, we say A is a weighted-output FPRTN iff B is a RTNBO with weight output, either as unique output or as one of the outputs of a composite output machine (definition 275, p. 339). For the sake of generality, we define WO-FPRTNs as both O-FPRTNs and WFSMs with*

- *$(W, \bullet, \prec)$ as partially ordered group, that is, with W as set of weights, $\bullet$ as operator on weights and $\prec$ as weight comparator, and*

- $\Gamma \times W$ *as set of output labels, where Γ is a set of output functions that apply on the blackboard components other than the weight.*

Mainly, the procedure we present here is divided into two stages. The first stage consists in traversing the WO-FPRTN in order to find and annotate the WO-FPRTN top path:

Definition 282 (Top path of a WO-FPRTN). *Top paths of WO-FPRTNs are defined as top paths of WFSMs, though taking into account that weights of WO-FPRTN recognize the outputs instead of generating them: weights are the second term of the pairs that form their input labels.*

The top path is annotated by marking at each state the incoming transition that allowed for reaching the state by generating the maximum weight. At a second stage, the top path is reversely traversed in order to generate the top blackboard:

Definition 283 (Top blackboard of a WO-FPRTN). *Let A be a WO-FPRTN and b a blackboard in $L(A)$, we say b is a top blackboard of A iff for every blackboard b' in $L(A)$ the weight component of b is greater or equal than the weight component of b'.*

Note that this forward-and-backtrack procedure is typically followed by other dynamic programming algorithms (Bellman, 1957), such as Wagner and Fischer's (1974) algorithm for the computation of the edit distance between two strings (Levenshtein, 1966).

Since the top blackboard is to be computed by reversely traversing the top path found, output functions on blackboards cannot be applied as is; instead, their converse functions are to be applied:

Definition 284 (Converse of a function on blackboards). *Let $\gamma_1, \gamma_2 \ldots \gamma_n$ be a sequence of functions on blackboards, the converse of γ_i, $\check{\gamma}_i$, is another function on blackboards such that the following equation is satisfied:*

$$(\check{\gamma}_1 \circ \check{\gamma}_2 \circ \ldots \circ \check{\gamma}_n)(b_\emptyset) = (\gamma_n \circ \ldots \circ \gamma_2 \circ \gamma_1)(b_\emptyset) \qquad (18.3)$$

Definition 285 (Converse of a binary operator). *Let $\bullet$ be a binary operator, we define the converse of $\bullet$, $\check{\bullet}$, as another binary operator such that, for all a, b, c,*

$$a \bullet b = c \iff b \check{\bullet} a = c. \qquad (18.4)$$

Corollary 10 (Double converse). *The converse of the converse of a binary operator is the operator itself, that is, $\breve{\breve{\bullet}} = \bullet$.*

Lemma 28. *Pair $(G, \breve{\bullet})$ is a monoid iff so it is $(G, \bullet)$ and, if so, both monoids share the same identity element.*

Proof. For $(G, \bullet)$ to be a monoid, the following two axioms must be satisfied:

- operator $\bullet$ is associative, and

- $\exists e \in G$ such that e is the identity of $\bullet$.

The first axiom is satisfied iff

$$a \bullet (b \bullet c) = (a \bullet b) \bullet c. \tag{18.5}$$

By definition of $\breve{\bullet}$, the following two equations hold:

$$a \bullet (b \bullet c) = (c \breve{\bullet} b) \breve{\bullet} a \tag{18.6}$$
$$(a \bullet b) \bullet c = c \breve{\bullet} (b \breve{\bullet} a), \tag{18.7}$$

which together with the former equation proof the associative condition of $\breve{\bullet}$:

$$(c \breve{\bullet} b) \breve{\bullet} a = c \breve{\bullet} (b \breve{\bullet} a). \tag{18.8}$$

For e to be the identity element of $\bullet$, the following axioms must be satisfied:

$$a \bullet e = a \tag{18.9}$$
$$e \bullet a = a. \tag{18.10}$$

If so, the following equations hold by definition of $\breve{\bullet}$:

$$e \breve{\bullet} a = a \tag{18.11}$$
$$a \breve{\bullet} e = a, \tag{18.12}$$

and therefore e is also the identity of $\breve{\bullet}$. Since $\breve{\breve{\bullet}} = \bullet$, the same reasoning can be applied to prove that $\bullet$ is a monoid iff so it is $\breve{\bullet}$ and that, if so, both share the same identity element. $\qquad\square$

Lemma 29 (Converse operator on blackboards). *Let $(B, \bullet)$ be a monoid with an identity element $b_\emptyset$; if every output function is of the form $\gamma_{b_r}(b_l) = b_l \bullet b_r$, then $\breve{\gamma}_{b_r}(b_l) = b_l \breve{\bullet} b_r$.*

Proof. Let $\gamma_{b_1}, \gamma_{b_2} \ldots \gamma_{b_n}$ be a sequence of functions on blackboards such that $\gamma_{b_i}(b) = b \bullet b_i$. If $n = 1$ then it holds that

$$
\begin{aligned}
(\gamma_{b_n} \circ \gamma_{b_{n-1}} \circ \ldots \circ \gamma_{b_2} \circ \gamma_{b_1})(b_\emptyset) &= ((\ldots((b_\emptyset \bullet b_1) \bullet b_2) \bullet \ldots \bullet b_{n-1}) \bullet b_n) \\
&= b_\emptyset \bullet b_1 \bullet b_2 \bullet \ldots \bullet b_{n-1} \bullet b_n \\
&= b_1 \bullet b_2 \bullet \ldots \bullet b_{n-1} \bullet b_n \\
&= b_n \,\breve{\bullet}\, b_{n-1} \,\breve{\bullet}\, \ldots \,\breve{\bullet}\, b_2 \,\breve{\bullet}\, b_1 \\
&= b_\emptyset \,\breve{\bullet}\, b_n \,\breve{\bullet}\, b_{n-1} \,\breve{\bullet}\, \ldots \,\breve{\bullet}\, b_2 \,\breve{\bullet}\, b_1 \\
&= ((\ldots((b_\emptyset \,\breve{\bullet}\, b_n) \,\breve{\bullet}\, b_{n-1}) \,\breve{\bullet}\, \ldots \,\breve{\bullet}\, b_2) \,\breve{\bullet}\, b_1) \\
&= (\breve{\gamma}_{b_1} \circ \breve{\gamma}_{b_2} \circ \ldots \circ \breve{\gamma}_{b_{n-1}} \circ \breve{\gamma}_{b_n})(b_\emptyset)
\end{aligned}
$$

$\square$

Note that the associative blackboard composition operator described in lemma 20 (p. 269) forms, indeed, a monoid $(B, \bullet)$ with B as the set of blackboards and $b_\emptyset$ as the identity element. Either for string, score or probability output, such a monoid exists:

- the set of output strings with the string concatenation and the empty string as identity element,

- the set of integer numbers with the addition and 0 as the identity element, and

- the set of real numbers with the multiplication and 1 as the identity element.

The case of feature structure output and unification processes will be discussed in the next chapter.

In case a set of output functions cannot be conversed, the top path can be reversely traversed in order to annotate the corresponding outgoing transitions at each state instead of the incoming ones, then traverse the top path in direct order in order to compute the top blackboard with the original output functions. Another possibility is to apply the reverse of the RTNBO representing the grammar to the reversed sentence; the resulting WO-FPRTN will then represent the reverse translations of the sentence, allowing for using the original output functions during the reverse traversal of the top path instead of their respective converse functions.

We have also studied the possibility of first traversing the whole WO-FPRTN in reverse order, that is, to compute a top path of the canonical reverse of the WO-FPRTN (definition 270, p. 306), then reversely traverse that path in order to compute the top blackboard using the original output functions on blackboards instead of their converses. However, the traversal of the WO-FPRTN must follow a topological sort of the whole machine (definition 274, p. 330); while we can ensure the existence of such topological sort for O-FPRTNs (theorem 30, p. 332), that is not the case of their canonical reverses. A possible example is the canonical reverse of the O-FPRTN of figure 15.3 (p. 321); it suffices to apply the algorithm we describe below to that machine in order to realize of this fact.

18.2.1   The algorithm

Algorithm 18.1 *woutput_fprtn_top_reverse_path* is an almost straightforward adaptation of algorithm 16.3 *output_fprtn_bsp_earley_language* (p. 336) for the efficient computation of a top path of a WO-FPRTN. Instead of computing every possible blackboard that can be generated by reaching each state, it stores only the maximum generated weight up to reaching each state and, each time a new maximum is found for a given state, it stores as well the reverse of the transition that reached that state by generating such a maximum weight. The maximum weight is given by map ζ_w, and the reversed top transition by map ζ_t. The top path can be later traversed in reverse order by following the reversed top transition at each state, starting from the "global" acceptor state up to reaching the initial state.

During the initialization phase, the algorithm sets the counters of incoming transitions for each state r_t, $\zeta_n(r_t)$, as for algorithm 16.3 *output_fprtn-_bsp_earley_language*. However, instead of setting the sets of blackboards (SBs) of each state to an empty set, the algorithm sets the maximum weight of each state to the minimum possible weight ($\zeta_t(r_t) \leftarrow w_{\min}$) so that the first computed weight by reaching r_t is set as the new maximum weight of r_t. The top reversed transition of each state r_t, ($\zeta_t(r_t)$), is assumed to be $\perp$ by default, though an implementation of this algorithm may require to explicitly assign a null value. States having an undefined top reversed transition will be those initiating calls within the FPRTN, that is, those whose top reversed transition is a reversed push transition.[2] The algorithm sets the weight of

---

[2]Recall that while pop transitions are explicitly defined in O-FPRTNs for convenience,

the initial state to w_{id}, the weight identity element, instead of setting the SB of r_I to the empty blackboard. A last initialization instruction enqueues state r_I in order to start the WO-FPRTN exploration as for algorithm 16.3 *output_fprtn_bsp_earley_language*.

In spite of being optimized, algorithm 16.3 *output_fprtn_bsp_earley-_language* cannot avoid an exponential cost due to the computation of an exponential number of blackboards in the blackboard composition block: for a given pop transition $(r_f, r_s\uparrow, r_s)$, it adds to the SB of r_s every blackboard $b_s = b'_s \circ b_f$ such that $(b'_s, b_f) \in B'_s \times B_f$, where B_f is the set of blackboards of r_f and B'_s is the set of every blackboard of every source state of every call transition completed by the pop transition. Instead, algorithm 18.1 *woutput-_fprtn_top_reverse_path* first retrieves $r'_{s_{\max}}$, the state having the maximum weight among all the source states of call transitions completed by the pop transition, then computes only the composition of this maximum weight with the maximum weight of r_f. The treatment for each transition incoming to r_s is the same than for algorithm 16.3 *output_fprtn_bsp_earley_language*, though it computes only the maximum weight and stores it along with the reverse of the corresponding transition whenever a new maximum is found. As for algorithm 16.3 *output_fprtn_bsp_earley_language*, algorithm 18.1 *woutput_fprtn_top_reverse_path* uses counters in order to check whether every incoming transition of each state has already been traversed or not; if the WO-FPRTN is not needed for any other treatment, transitions can be simply removed from the WO-FPRTN instead of keeping a set of counters since the memory allocating the transitions is to be freed sooner or later (as stated in footnote 1, p. 326).

Finally, algorithm 18.3 *woutput_fprtn_top_blackboard* computes a top blackboard of a WO-FPRTN $A = (Q', K, \Gamma \times W, \delta', \kappa, \{r_I\}, F')$. First of all, it calls algorithm 18.1 *woutput_fprtn_top_reverse_path* in order to build ζ_t, the map of states to top-reverse transitions, and to retrieve r_F, the "global" acceptor state of A. Afterwards, it traverses the reverse of the computed top path, from r_F up to r_I, by following at each state the top reverse transition defined by ζ_t. Apart from being reversed, the latter operation is similar to the computation of the language of a RTN (section 14.5, p. 284) by means of a breadth-first traversal of a RTNBO (section 13.5, p. 262), though keeping

---

push transitions are not needed to: the reverse traversal of a push transition will simply consist in bringing the machine to the state at the top of the stack and to pop that state out.

a single top ES (r_t, b_t, π) at each iteration instead of a SES containing every possible ES. The initial top ES is set to $(r_F, b_\emptyset, \lambda)$. Then, the sequence of top ESs that compose the top reverse path is iteratively computed. Let the current top ES be $x_t = (r_t, b_t, \pi)$, the next top ES x_s is computed as follows, depending on the type of the next top reversed transition:

- reverse pop transition: since acceptance states of A^R are those not having a top reverse transition, $x_t = (r_r, b_t, \pi')$ with $\pi = \pi' r_r$ iff $\zeta_t = \bot$,

- reverse filtered-push transition: $x_t = (r_f, b_t, \pi r_s)$ iff $\zeta(r_t) = (r_t, \{r_f\}, r_s)$, knowing that a unique top state r_f is called such that $(r_t, r_f\downarrow, r_s)$ is an allowed top filtered-push transition of A^R since $(r_s, r_f\uparrow, r_t)$ is an allowed top filtered-pop transition of A,

- reverse ε-transition: $x_t = (r_s, b_t, r_t)$ iff $\zeta_t(r_t) = (r_t, (\mathrm{id}_B, w), r_s)$, and

- reverse consuming transition: $x_t = (r_t, \breve{\gamma}(b_t), \pi)$ iff $\zeta_t = (r_t, \gamma, r_s)$, applying $\breve{\gamma}$ to b_t instead of γ so that the resulting blackboard belongs to $L(A)$ instead of $L(A^R)$.

Algorithm 18.1 woutput_fprtn_top_reverse_path(A, ζ_s'')

Input: $A = (Q', K, \Gamma \times W, \delta', \kappa, \{r_I\}, F')$, a weighted-output FPRTN

       ζ_s'', a function mapping each pop transition to the set of source states

  of the call transitions that it completes

Output: ζ_w, a map of states to top weights

         ζ_t, a map of states to top reverse transitions

         r_s, the last visited state = the "global" acceptor state of A

1: $E \leftarrow \emptyset$

2: **for each** $r_t \in Q'$ **do**

3:     $\zeta_n(r_t) \leftarrow |\{(r_s, (\gamma, w), r_t) : r_t \in \delta'(r_s, (\gamma, w))\}| +$

           $|\{(r_s, (\mathrm{id}_B, w_{\mathrm{id}}), r_t) : r_t \in \delta'(r_s, (\mathrm{id}_B, w_{\mathrm{id}}))\}| +$

           $|\{(r_s, R_c, r_t) : r_t \in \delta'(r_s, R_c)\}| +$

           $|\{(r_s, r_t\uparrow, r_t) : r_t \in \delta'(r_s, r_t\uparrow)\}|$

4:     $\zeta_w(r_t) \leftarrow w_{\min}$

5: **end for**

6: $\zeta_w(r_I) \leftarrow w_{\mathrm{init}}$

7: enqueue(E, r_I)

8: **while** $E \neq \emptyset$ **do**

9:     $r_s \leftarrow$ dequeue(E)

10:    $w_s \leftarrow \zeta_w(r_s)$

11:    **for each** $r_f : \delta'(r_f, r_s\uparrow)$ **do**   ▷ BLACKBOARD COMP.

12:       $r'_{s_{\max}} \leftarrow$ first$(\zeta_s''((r_f, r_s\uparrow, r_s)))$

13:       **for each** $r'_s \in \zeta_s''((r_f, r_s\uparrow, r_s)) - \{\mathrm{first}(\zeta_s''((r_f, r_s\uparrow, r_s)))\}$ **do**

14:          **if** $\zeta_w(r'_{s_{\max}}) \prec \zeta_w(r'_s)$ **then**

15:             $r'_{s_{\max}} \leftarrow r'_s$

16:          **end if**

17:       **end for**

18:       $w \leftarrow \zeta_w(r'_{s_{\max}}) \bullet \zeta_w(r_f)$

19:       **if** $w_s \prec w$ **then**

20:          $w_s \leftarrow w$

21:          $\zeta_t \leftarrow (r_t, \{r_f\}, r'_{s_{\max}})$

22:       **end if**

23:    **end for**

24:     **for each** $(r_t, (\gamma, w)) : r_t \in \delta'(r_s, (\gamma, w))$ **do** $\triangleright$ CONSUMING TRANS.
25:         $w_t \leftarrow w_s \bullet w$
26:         **if** $\zeta_w(r_t) \prec w_t$ **then**
27:            $\zeta_w(r_t) \leftarrow w_t$
28:            $\zeta_t(r_t) \leftarrow (r_t, (\gamma, w), r_s)$
29:         **end if**
30:         $\zeta_n(r_t) \leftarrow \zeta_n(r_t) - 1$
31:         **if** $\zeta_n(r_t) = 0$ **then**
32:            enqueue(E, r_t)
33:         **end if**
34:      **end for**
35:      **for each** $r_t \in \delta'(r_s, (\mathrm{id}_B, w_{\mathrm{id}}))$ **do** $\triangleright$ ε-TRANSITIONS
36:         **if** $\zeta_w(r_t) \prec w_s$ **then**
37:            $\zeta_w(r_t) \leftarrow w_s$
38:            $\zeta_t(r_t) \leftarrow (r_t, (\mathrm{id}_B, w_{\mathrm{id}}), r_s)$
39:         **end if**
40:         $\zeta_n(r_t) \leftarrow \zeta_n(r_t) - 1$
41:         **if** $\zeta_n(r_t) = 0$ **then**
42:            enqueue(E, r_t)
43:         **end if**
44:      **end for**
45:      **for each** $(r_t, R_c) : r_t \in \delta'(r_s, R_c)$ **do** $\triangleright$ PUSH TRANSITIONS
46:         $\zeta_n(r_t) \leftarrow \zeta_n(r_t) - 1$
47:         **if** $\zeta_n(r_t) = 0$ **then**
48:            enqueue(E, r_t)
49:         **else if** $\zeta_w(\mathrm{first}(R_c)) = w_{\min}$ **then**
50:            **for each** $r_c \in R_c$ **do**
51:               $\zeta_w(r_c) \leftarrow w_{\mathrm{init}}$
52:               enqueue(E, r_c)
53:            **end for**
54:         **end if**
55:      **end for**

56:    **for each** $r_t : r_t \in \delta'(r_s, r_t\uparrow)$ **do**   ▷ POP TRANSITIONS
57:        $\zeta_n(r_t) \leftarrow \zeta_n(r_t) - 1$
58:        **if** $\zeta_n(r_t) = 0$ **then**
59:            enqueue(E, r_t)
60:        **end if**
61:    **end for**
62: **end while**

Algorithm 18.2 top_state(ζ_w, Q'')

Input: ζ_w, the map of states to top weights
        Q'', a set of states
Output: $r_{\max}$, the state in Q'' mapped to the maximum weight
 1: $r_{\max} \leftarrow$ first(Q'')
 2: **for each** $r \in Q'' - \{$first$(Q'')\}$ **do**
 3:     **if** $\zeta_w(r_{\max}) \prec \zeta_w(r)$ **then**
 4:         $r_{\max} \leftarrow r$
 5:     **end if**
 6: **end for**

Algorithm 18.3 woutput_fprtn_top_blackboard(A, ζ_s'')

Input: $A = (Q', K, \Gamma \times W, \delta', \kappa, \{r_I\}, F')$, a weighted-output FPRTN
        ζ_s'', a function mapping each pop transition to the set of source states
    of the call transitions that it completes
Output: b_t, a top blackboard of A
 1: $(\zeta_t, r_F) \leftarrow$ woutput_fprtn_top_reverse_path(A, ζ_s'', ζ_c)
 2: $(r_t, b_t, \pi) \leftarrow (r_F, b_\emptyset, \lambda)$
 3: **while** $r_t \neq r_I$ **do**
 4:     **if** $\zeta_t(r_t) = \bot$ **then let** $\pi = \pi' r_r$ ▷ REVERSE POP TRANS.
 5:         $(r_t, b_t, \pi) \leftarrow (r_r, b_t, \pi')$
 6:     **else if** $\zeta_t(r_t) = (r_t, \{r_f\}, r_s)$ **then**   ▷ REVERSE PUSH TRANS.
 7:         $(r_t, b_t, \pi) \leftarrow (r_f, b_t, \pi r_s)$
 8:     **else if** $\zeta_t(r_t) = (r_t, (\mathrm{id}_B, w_{\mathrm{id}}), r_s)$ **then**   ▷ REVERSE ε-TRANS.
 9:         $(r_t, b_t, \pi) \leftarrow (r_s, b_t, \pi)$
10:     **else let** $\zeta_t(r_t) = (r_t, (\gamma, w), r_s)$ ▷ REVERSE CONSUMING TRANS.
11:         $(r_t, b_t, \pi) \leftarrow (r_s, \breve{\gamma}(b_t), \pi)$
12:     **end if**
13: **end while**

Chapter 19

Unification finite-state machines

We briefly present here machines comprising unification processes as a special case of blackboard output, analogously to the way in which we have presented weighted machines in the previous chapter. Unification is the only kind of blackboard processing presented in this dissertation that makes use of killing blackboards. Since our definitions and algorithms of application of machines with blackboard output take into account this possibility, adapting the algorithms for the case of unification machines is straightforward except for the last and most efficient algorithm we have presented in this dissertation: algorithm 18.3 *woutput_fprtn_top_blackboard*. We briefly describe unification in section 19.1, unification machines in section 19.2, the advantages of unification in section 19.3, and how to adapt the algorithms of application of machines with blackboard output in order to support unification in section 19.4.

19.1   Overview of unification

Unification allows for a compact representation of long-distance relationships and dependencies, that is, relationships and dependencies between input elements that are separated by an arbitrary amount of input rather than being consecutive; for instance, the number agreement between the subject and the verb of a sentence. Algorithms of application of grammar formalisms comprising unification make use of feature structures in order to store linguistic data as it is observed during the analysis of the sentences. Such feature structures are structures of attribute/value pairs (e.g.: number/singular, func-

tion/subject, etc.), where values can be other feature structures. Additionally, feature structures may comprise values that are shared among different attributes, forming complex structures analogous to directed acyclic graphs (examples can be found in Jurafsky and Martin, 2008, chap. 11, p. 391).

Unification is a monotonic operation: the unification of two feature structures results in another feature structure containing every attribute/value pair of both the feature structures to unify. Whenever unifying two feature structures both containing a given attribute, 3 situations are possible:

- only one of the feature structures defines a value for the attribute (the value in the other feature structure is not set), in which case the resulting feature structure will contain such attribute/value pair without duplicating the attribute,

- both feature structures define the same value for the attribute, in which case the resulting feature will contain the attribute paired with the defined value, or

- the feature structures define different values for the attribute, in which case the feature structures cannot be unified due to an inconsistency.

Whenever inconsistencies appear, killing blackboards are to be generated in order to invalidate the analysis that led to them; for instance, a possible attribute name could be 'number agreement', whose value is to be taken from both the subject and the verb of the sentence to analyse. When either the subject or the verb is read, the feature structure of the current analysis is unified with another one that includes a 'number agreement' attribute taking as value the number of the sentence constituent read. In other words, a set of registers is used in order to remember the number of either the subject or the verb so that it can be compared when reading the other sentence constituent.

More information on feature structures, unification, and how to implement them can be found in Jurafsky and Martin (2008, chap. 11, p. 391).

19.2   Unification machines

We define unification machines as follows:

Definition 286 (UFSM). *In general, unification finite-state machines (UF-SMs) are a particular case of FSMs with blackboard output so that*

- *functions in Γ always perform the unification operation ⊔ of a feature structure with the current blackboard, which is a feature structure as well; for simplicity, we consider that Γ contains feature structures rather than functions on blackboards, and output labels $g \in \Gamma$ represent the operation $b \sqcup g$ where b is the current blackboard,*

- *the identity function on blackboards id_B unifies the empty feature structure, [ ], and the current blackboard,*

- *B is the set of feature structures,*

- *$B_K = \{\bot\}$, where ⊥ represents the inconsistent feature structure, that is, the unification of two incompatible structures of features, and*

- *$b_\emptyset = [\,]$, that is, the empty feature structure.*

As for input labels of lexical FSMs (definition 48, p. 122), feature structures in Γ may rather be expressions —called unification equations— *which describe feature structures whose values may be taken from the properties of the read input (e.g.: a feature structure with a 'number agreement' attribute taking its value from the 'number' property of the last read token).*

An example of unification machines are local grammars extended with feature structures and unification processes (Blanc and Constant, 2005; Blanc, 2006); these machines are equivalent to lexical-functional grammars (Kaplan and Bresnan, 1982): they use RTNs instead of CFGs, which are equivalent grammar formalisms, coupled with feature structures and unification. Such local grammars have been used for parsing complex sentences.

19.3   Advantages of unification

Without unification, finite-state machines such as FSAs and RTNs require a separate machine substructure for each possible valid combination of pairs 'attribute/value'; for instance, assuming that the number and gender of two sentence constituents must agree, and that there are only two possible values for these attributes, four machine substructures are required in order to represent the consistent combinations: both constituents are masculine and singular, both are masculine and plural, both are feminine and singular, or both are feminine and plural. Note that such machine substructures require

also to represent the sentence constituents that may appear between the constituents that must agree. When manually constructing a grammar, this implies to copy by hand such substructures in order to define each possible combination. Moreover, as the number of co-occurrent attributes increases, the number of consistent combinations may increase exponentially. Depending on the grammars to define, unification can avoid an important amount of redundancy while avoiding an exponential growth of the grammar.

Unification can also ease the representation of sentence constituents that may appear in an arbitrary order; for instance, in the MovistarBot use case (section 1.2, p. 6), request sentences are composed by some structure identifying the service requested along with other structures containing the service arguments, where the order in which they appear may not necessarily be fixed: sentences 'envía Feliz Navidad al 555-555-555' (send Merry Christmass to the 555-555-555) and 'envía al 555-555-555 Feliz Navidad' ('send to the 555-555-555 Merry Christmass') are equivalent. Note that, while there are only 2 possibilities with 2 arguments that may permute, the number of combinations increases exponentially w.r.t. the number of freely-permutable arguments.

19.4 Supporting unification

As for weight output (previous chapter), the adaptation of the Earley-like algorithm of application of RTNBOs 13.10 (p. 272) for feature structure output and unification processes requires only to define the feature structure composition operator:

Definition 287 (Feature structure composition operator). *We define the blackboard composition operator (definition 239, p. 268) for the case of unification RTNs as $\sqcup$ since it is a particular case of lemma 20 (p. 269).*

Almost every algorithm of application of machines with blackboard output we have presented in this dissertation takes into account the possibility of generating killing blackboards, hence do not require any further modification in order to support unification. The exception is the algorithm computing the top blackboard of a WO-FPRTN (algorithm 18.3 *woutput_ fprtn_ top_ blackboard*, p. 355). Until now, we have considered the following approaches in order to extend this algorithm with unification processes:

- ensuring that the grammar does not associate the highest score to an inconsistent interpretation for every possible input sentence, that is, ensuring that top-ranked blackboards are not killing blackboards by construction of the grammar, and

- extending the algorithm so that further top-ranked blackboards (the second in the raking, the third, etc.) are efficiently computed in case killing blackboards are encountered.

Note that, in case the grammar defines an exponential number of top-ranked killing blackboards, the algorithm will no longer have a polynomial worst-case cost but an exponential one. A combined possibility would be to ensure that the grammar does not define such an exponential number of top-ranked killing blackboards, that is, to ensure that the non-killing top-ranked black-board is one of the k top-ranked ones for some constant k. A last resource would be to define a procedure for the removal of conflicting unification equations, replacing them by the equivalent sequences of machine substructures for each possible combination. However, the same side-effect than that of RTN flattening (section 12.8, p. 239) can be expected: an exponential growth of the grammar. Due to the complexity of the problem, we leave it open to a future work.

Part III

Results & conclusions

Chapter 20

Experimental results

In this chapter we present the results of the experiments we have conducted in order to empirically compare the performances of the different algorithms we have presented. We first give in section 20.1 an overview of the treatment we have performed, recall the different algorithms and algorithm optimizations we have tested, and describe the implementation details and the actual experiment conditions. Finally, we discuss in section 20.2 the observed results, namely the speedup factors for each algorithm and algorithm optimization relative to the simplest algorithm, the algorithm overheads, and the asymptotic cost of the different algorithms.

20.1   Description

Figures 20.1 and 20.2 (pgs. 373–378) compare the performance of each variant of each algorithm of application of RTNs —with and without output— for two versions of the MovistarBot grammar: in both cases the grammar has been pseudo-determinized (section 13.7, pg. 265) but in the latter case it has first been flattened (section 13.6, pg. 263). Since the grammar contains no recursive calls, the flattened version is not an approximation but a FST equivalent to the original RTN. We have applied the MovistarBot grammar to a test corpus mainly composed by sentences requesting for mobile services. Other sentences have been added in order to control over-recognition (they are to be rejected). The grammar is a RTN with string and weight output (chapters 14 and 18, respectively):

- output string symbols are XML tags which either identify the requested

service or delimit the arguments to extract (see figure 10.1(a), pg. 188),
and

- weights are used for choosing a single interpretation (the one with the
  maximum score) for the case of ambiguous sentences.

Translator algorithms compute maps of XML tags to input segments (the
input position at the moment of generating the corresponding opening and
closing XML tags, starting with 1 as the first input token). Additionally,
each map is associated to an overall weight. The map with the highest
overall weight is to be transformed into a command and then passed to the
MovistarBot. This transformation is trivial and has simply been hard-coded
as a C++ function; for instance, the following set of mappings of XML tags to
left-open input intervals is generated for sentence '*envía hola al 555*', among
others:

$$\text{sms} \quad \rightarrow \quad (1,1] \tag{20.1}$$
$$\text{message} \quad \rightarrow \quad (1,2] \tag{20.2}$$
$$\text{phone} \quad \rightarrow \quad (3,4] \tag{20.3}$$

The presence of the first mapping implies that the user is asking to send an
SMS, and the others define the input segments to be used as message and
phone arguments, respectively. For this map, command `sms 555 hola` is to
be generated. Note that the input interval of the first mapping is empty: only
XML tags corresponding to arguments to be outputted need to be mapped
to a non-empty input interval. XML tags identifying the requested service
require only to be present in the map.

   Translator algorithms compute the set of outputs for each possible inter-
pretation of the input sentence, either as an explicit list of outputs (a list of
maps in this case) or as some kind of machine factoring out common parts:
a filtered-popping recursive transition network (FPRTN, chapter 15) having
pairs XML tag/weight as transition labels. They then translate the top-
ranked output into the corresponding MovistarBot command. Additionally,
FPRTN-based algorithms also prune the generated FPRTN (section 16.1,
pg. 325) before generating either the whole set of outputs or the top-ranked
output only, depending on the algorithm. Acceptor algorithms —algorithms
computing only whether the sentence corresponds to a service request or
not, without generating any translation— ignore grammar's output labels

and compute only whether the sentence is a request for a supported online service or not.

Figure 20.3 illustrates the performance drop of the best performing variant of each algorithm, for input $a^n b^n$ with $n = 0 \ldots 15$ and grammar of figure 14.1 (p. 284). In this case, the grammar has weight and string output; the treatment is similar to the MovistarBot cases though sets of weighted strings are to be generated instead of sets of weighted maps, and the top-ranked string is to be returned as is, that is, without being transformed into some command.

Recall that grammar of figure 14.1 is a minimal theoretical grammar whose purpose is to produce an exponential number of outputs w.r.t. the input length; exponential production happens in natural language grammars due to ambiguity that increases exponentially with additional nesting levels of subgrammar calls. Though such nesting levels in natural language grammars are not usually high, significant speedups can be perceived even for low nesting levels due to the exponential nature of the problem: in spite of the small size of grammar of figure 14.1 (6 states and 7 transitions), non-exponential algorithms already perform better than their exponential counterparts for nesting levels greater than 3; lower nesting levels will be required for general natural language grammars, which can easily reach millions of states and transitions.

20.1.1   Algorithms

In the figures, the following short codes and background colors have been used in order to identify each algorithm:

- **depth-first -o** : depth-first acceptor, section 12.7 (pg. 235)

- **depth-first** : depth-first translator, section 13.5 (pg. 262)

- **breadth-first -o** : breadth-first acceptor, section 12.7 (pg. 235)

- **breadth-first** : breadth-first translator, section 13.5 (pg. 262)

- **earley -o** : Earley acceptor, section 12.11 (pg. 246)

- **earley** : Earley translator, section 13.10 (pg. 272)

- **to-fprtn** : to FPRTN translator (prunes the FPRTN but does not generate its language), section 15.6 (pg. 306)

- **to-fprtn-bfe** : to FPRTN translator and FPRTN breadth-first expansion (as 'to-fprtn' but also generating the language of the FPRTN by means of a breadth-first traversal), sections 15.6 (pg. 306) and 16.2 (pg. 327),

- **to-fprtn-zpps** : to FPRTN & ζ_s'' map translator (as 'to-fprtn' but also building a map ζ_s'' and performing some variable initializations required by algorithm 'to-fprtn-bse'), sections 15.6 (pg. 306) and 16.3 (pg. 329),

- **to-fprtn-bse** : to FPRTN translator and blackboard set expansion (as 'to-fprtn-zpps' but also expanding the FPRTN by means of blackboard set processing instead of a breadth-first traversal), sections 15.6 (pg. 306) and 16.3 (pg. 329),

- **to-fprtn-top** : to FPRTN translator and top-blackboard initialization (as 'to-fprtn-zpps' but performing the initializations required by algorithm 'to-fprtn-tbe' instead of 'to-fprtn-bse'), sections 15.6 (pg. 306) and 16.2 (pg. 327),

- **to-fprtn-tbe** : to FPRTN translator and top-blackboard extractor (as 'to-fprtn-top' but also extracting the top-ranked blackboard by a method similar to blackboard set processing), sections 15.6 (pg. 306) and 16.2 (pg. 327).

Note that algorithms with faded colors do not perform the whole chain of treatment, either because they are simple acceptors or because omit some final stages of treatment. We have included them in order to observe the performance drop due to output generation, and to observe the cost of each separate stage of treatment, namely:

- 'earley -o' = cost of computing the Earley acceptor sets of execution states,

- 'to-fprtn' minus 'earley -o' = cost of adding transitions with output labels to the Earley acceptor execution states in order to build an output FPRTN, plus later prunning the FPRTN,

- 'to-fprtn-zpps' minus 'to-fprtn' = cost of building ζ_s'' map and performing some variable initializations for output generation by means of blackboard set processing,

- 'to-fprtn-top' minus 'to-fprtn' = cost of building ζ_s'' map and performing some variable initializations for the generation of the top-blackboard,

- 'to-fprtn-bse' minus 'to-fprtn-zpps' = cost of generating every output accepted by the FPRTN by means of blackboard set processing and then choose the top-ranked one, and

- 'to-fprtn-tbe' minus 'to-fprtn-top' = cost of generating only the top-ranked output within the FPRTN.

For instance, we can see that the cost of computing map ζ_s'' and performing the subsequent variable initializations is negligible. Additionally, algorithms implementing partial treatments establish a performance limit for algorithms performing additional stages (e.g.: 'to-fprtn' cannot be faster than 'earley -o' since it performs the same treatment stages plus some additional ones). In general, it is no use implementing a FPRTN based algorithm in order to surpass an algorithm X if 'to-fprtn' performs worst than X; the implementation of 'to-fprtn' is to be first improved until obtaining a meaningful performance margin w.r.t. the algorithm to surpass.

20.1.2 Algorithm variants

In the figures, parameters other than '-o' identify "minor" algorithm optimizations (the algorithm variants), namely

- **+t**: optimize sequence management by means of **t**ries (chapter 9); applicable to algorithms whose execution states include a stack of return states (namely 'depth-first', 'breadth-first' and the breadth-first expansion of 'fprtn-bfe') and/or include a sequential partial output (in the case exposed here, outputs are not sequences but descriptions of the mobile service the sentence is asking for),

- **-eXXX**: set/map implementation for the management of sets/maps of **e**xecution states (excluding 'depth-first' since it does not build sets or maps of execution states but single execution states), and

- **-bXXX**: set/map implementation for the management of sets/maps of **b**lackboards (the output structures).

The different set/map implementations are

- **std**: the one provided by GNU's implementation of the C++ Standard Template Library, that is, red-black trees (section 2.5, pg. 62) with Cormen's addition algorithm (section 2.3.6, pg. 50),

- **lrb**: our custom implementation based on double-linked red-black trees (section 2.6, pg. 64) with Knuth's addition algorithm (section 2.3.5, pg. 45), and

- **lrb-3w**: as the previous one but using a 3-way comparator (section 2.3.9, pg. 55).

Fully-colored rows highlight the fastest variant of each algorithm.

20.1.3   Implementation details

Every algorithm has been programmed in C++ (Stroustrup, 2000), using the Standard Template Library (see for instance Josuttis, 1999) and some Boost libraries (http://www.boost.org). We have taken advantage of generic programming in order to reuse the source code of each algorithm for every possible variant —sequence, set and map types have been declared as template types. Apart from factoring out the source code, this ensures that the performance difference between the different variants of the same algorithm is exclusively due to the different implementation of sequences (with or without trie optimization), sets and maps. Input and output types have also been declared as template types so that other kind of grammars can be supported in the future (e.g.: with other character codification schemes such as UTF-8, with other kind of lexical masks, with unification processes, etc.). Every algorithm variant has been compiled into a single executable, weighting 5.9 MB, with version 4.3.2 of GNU's g++ compiler. The codes described in the two previous sections are used as parameters in order to choose the algorithm variant to execute.

20.1.4   Experiment conditions

Each algorithm has been applied to the whole corpus several consecutive times in order to obtain meaningful measures: a minimum amount of seconds is spent per algorithm, the number of consecutive applications being counted. Each measure has been taken several times; graph bars of figures 20.1 and 20.2 represent means and error bars represent the minimum and maximum measures. Errors are less than 1% of the observed measure, hence we can consider negligible the error of the speedup factors we will give (below ± 0.001). In figure 20.3, only thick curves representing the means have been drawn; except for $n = 0$, the regions between the maximum and minimum curves for each algorithm are thinner enough to be covered by the corresponding mean curves. These regions are slightly wider for $n = 0$, and a few times wider for algorithm 'depth-first -o'. Anyway, the purpose of this graphic is to compare the performance drops against an exponential grammar rather than giving absolute measures.

The measures not only include the cost of computing the result, but also the cost of freeing the allocated memory; hence, the overhead added by some optimizations and algorithms due to the use of more complex data structures, such as tries and FPRTNs, is fully taken into account. We have used GNU's `mcheck` library and `mtrace` tool in order to ensure that every single byte of dynamically allocated memory is properly freed.[1]

The tests were run on a Ubuntu platform version 8.10 (Intrepid Ibex), 64 bits. The hardware specifications are:[2]

- CPU: Intel® Core™2 Duo E8500, 3.16 GHz, 6 MB L2 cache, 64 KB L1 cache

- RAM: 8 GBs, DIMM DDR Synchronous 1066 MHz (0.9 ns)

Each test consumed no more than 18 MB of RAM for the case of the MovistarBot grammar and corpus, and less than the RAM size for the exponential case (more than 8 GBs are needed for some algorithms with exponential worst-case costs and $n \geq 20$). The pseudo-determinized version of the MovistarBot grammar has 1359 states and 3141 transitions, and the flattened and pseudo-determinized version has 5504 states and 31702 transitions. The

---

[1]http://www.gnu.org/s/libc/manual/html_node/Allocation-Debugging.html#Allocation-Debugging

[2]As listed by command `lshw`.

corpus contains 168 sentences, with an average of 10.1 tokens per sentence and 4.1 characters per token. Each sentence has an average of 6.9 interpretations for both versions of the MovistarBot grammar.

Parsed sentences per second

#	Parser	opt1	opt2	opt3	Value
1	depth-first	-o	+t		17234 ⋙
2	depth-first	-o			14485 ⋙
3	earley	-o	-elrb		4071 ⋙
4	earley	-o	-elrb-3w		3999
5	earley	-o	-estd		3403
6	to-fprtn-zpps		-elrb		2673
7	to-fprtn-top		-elrb		2667
8	to-fprtn-top		-elrb-3w		2660
9	to-fprtn-zpps		-elrb-3w		2650
10	to-fprtn		-elrb		2641
11	to-fprtn		-elrb-3w		2619
12	to-fprtn-tbe		-elrb	-blrb	2581
13	to-fprtn-tbe		-elrb	-bstd	2550
14	to-fprtn-tbe		-elrb-3w	-bstd	2542
15	to-fprtn-tbe		-elrb-3w	-blrb-3w	2509
16	to-fprtn-tbe		-elrb-3w	-blrb	2508
17	to-fprtn-tbe		-elrb	-blrb-3w	2492
18	breadth-first	-o +t	-elrb		2330
19	breadth-first	-o +t	-elrb-3w		2293
20	to-fprtn-bse		-elrb-3w	-bstd	2122
21	to-fprtn-bse		-elrb	-blrb-3w	2088
22	to-fprtn		-estd		2084
23	to-fprtn-bse		-elrb-3w	-blrb-3w	2083
24	to-fprtn-bse		-elrb-3w	-blrb	2072
25	to-fprtn-bse		-elrb	-blrb	2067
26	to-fprtn-bse		-elrb	-bstd	2064
27	to-fprtn-tbe		-estd	-blrb-3w	2035
28	to-fprtn-top		-estd		2026
29	to-fprtn-zpps		-estd		2025
30	earley		-elrb-3w	-bstd	2002
31	earley		-elrb	-bstd	1996
32	to-fprtn-tbe		-estd	-blrb	1948

Parsed sentences per second

#					value
33	breadth-first	-o	+t	-estd		1901
34	to-fprtn-bfe		+t	-elrb-3w	-bstd	1880
35	to-fprtn-bfe		+t	-elrb	-bstd	1879
36	to-fprtn-tbe			-estd	-bstd	1864
37	earley			-estd	-bstd	1800
38	to-fprtn-bse			-estd	-blrb-3w	1794
39	to-fprtn-bse			-estd	-blrb	1771
40	to-fprtn-bse			-estd	-bstd	1709
41	to-fprtn-bfe		+t	-elrb-3w	-blrb-3w	1676
42	to-fprtn-bfe		+t	-elrb	-blrb-3w	1670
43	to-fprtn-bfe		+t	-elrb	-blrb	1669
44	to-fprtn-bfe		+t	-elrb-3w	-blrb	1667
45	breadth-first	-o		-elrb-3w		1666
46	breadth-first	-o		-elrb		1632
47	to-fprtn-bfe			-elrb-3w	-bstd	1625
48	to-fprtn-bfe		+t	-estd	-bstd	1608
49	to-fprtn-bfe			-elrb	-bstd	1585
50	to-fprtn-bfe			-elrb	-blrb-3w	1467
51	to-fprtn-bfe			-elrb-3w	-blrb-3w	1453
52	earley			-elrb	-blrb	1446
53	to-fprtn-bfe			-estd	-bstd	1446
54	to-fprtn-bfe		+t	-estd	-blrb	1443
55	to-fprtn-bfe			-elrb	-blrb	1442
56	earley			-elrb-3w	-blrb	1442
57	to-fprtn-bfe			-elrb-3w	-blrb	1438
58	earley			-elrb	-blrb-3w	1430
59	to-fprtn-bfe		+t	-estd	-blrb-3w	1424
60	breadth-first	-o		-estd		1423
61	depth-first		+t		-bstd	1408
62	earley			-elrb-3w	-blrb-3w	1408
63	earley			-estd	-blrb-3w	1324
64	earley			-estd	-blrb	1321

Parsed sentences per second

65	to-fprtn-bfe	-estd	-blrb-3w	1283
66	to-fprtn-bfe	-estd	-blrb	1280
67	depth-first		-bstd	1220
68	depth-first	+t	-blrb	1044
69	depth-first	+t	-blrb-3w	1043
70	depth-first		-blrb	919
71	depth-first		-blrb-3w	909
72	breadth-first	+t -elrb	-bstd	835
73	breadth-first	+t -elrb-3w	-bstd	825
74	breadth-first	+t -estd	-bstd	775
75	breadth-first	-elrb-3w	-bstd	656
76	breadth-first	-elrb	-bstd	635
77	breadth-first	+t -elrb-3w	-blrb-3w	604
78	breadth-first	+t -elrb	-blrb-3w	604
79	breadth-first	+t -elrb	-blrb	602
80	breadth-first	+t -elrb-3w	-blrb	598
81	breadth-first	-estd	-bstd	597
82	breadth-first	+t -estd	-blrb-3w	589
83	breadth-first	+t -estd	-blrb	586
84	breadth-first	-elrb-3w	-blrb-3w	522
85	breadth-first	-elrb-3w	-blrb	516
86	breadth-first	-elrb	-blrb-3w	503
87	breadth-first	-elrb	-blrb	494
88	breadth-first	-estd	-blrb-3w	492
89	breadth-first	-estd	-blrb	486

Figure 20.1: Performance comparison of each algorithm variant for the Movis-tarBot corpus and flattened and pseudo-determinized grammar; fully colored rows correspond to the fastest variants, and only intense color rows correspond to algorithms that perform the whole chain of treatment (rows 12–89 except those of algorithm 'breadth-first -o').

Parsed sentences per second

#	Method					Value
1	depth-first	-o	+t			87064 ⋙
2	depth-first	-o				71874 ⋙
3	breadth-first	-o	+t	-elrb		10252 ⋙
4	breadth-first	-o	+t	-elrb-3w		10173 ⋙
5	earley	-o		-elrb		9435
6	earley	-o		-elrb-3w		9347
7	breadth-first	-o	+t	-estd		9294
8	earley	-o		-estd		8500
9	breadth-first	-o		-elrb		7522
10	breadth-first	-o		-elrb-3w		7443
11	breadth-first	-o		-estd		7057
12	to-fprtn			-elrb		6060
13	to-fprtn			-elrb-3w		6013
14	to-fprtn-top			-elrb		5968
15	to-fprtn-zpps			-elrb		5946
16	to-fprtn-top			-elrb-3w		5920
17	to-fprtn-zpps			-elrb-3w		5890
18	to-fprtn-tbe			-elrb	-blrb	5819
19	to-fprtn-tbe			-elrb	-bstd	5801
20	to-fprtn-tbe			-elrb	-blrb-3w	5795
21	to-fprtn-tbe			-elrb-3w	-blrb	5745
22	to-fprtn-tbe			-elrb-3w	-blrb-3w	5743
23	to-fprtn-tbe			-elrb-3w	-bstd	5707
24	to-fprtn-bfe		+t	-elrb	-bstd	5682
25	to-fprtn-bfe		+t	-elrb-3w	-bstd	5673
26	to-fprtn-bfe		+t	-elrb	-blrb	5625
27	to-fprtn-bfe			-elrb	-bstd	5588
28	to-fprtn-bfe		+t	-elrb	-blrb-3w	5582
29	to-fprtn-bfe		+t	-elrb-3w	-blrb	5569
30	to-fprtn-bfe			-elrb-3w	-bstd	5563
31	to-fprtn-bfe		+t	-elrb-3w	-blrb-3w	5535

Parsed sentences per second

32	to-fprtn-bfe		-elrb	-blrb-3w		5518
33	to-fprtn-bfe		-elrb	-blrb		5518
34	to-fprtn		-estd			5508
35	to-fprtn-bfe		-elrb-3w	-blrb		5506
36	to-fprtn-bfe		-elrb-3w	-blrb-3w		5497
37	to-fprtn-top		-estd			5440
38	to-fprtn-zpps		-estd			5423
39	to-fprtn-tbe		-estd	-blrb		5290
40	to-fprtn-tbe		-estd	-bstd		5286
41	to-fprtn-tbe		-estd	-blrb-3w		5249
42	to-fprtn-bfe		-estd	-bstd		5067
43	to-fprtn-bfe	+t	-estd	-bstd		5057
44	to-fprtn-bfe	+t	-estd	-blrb		5050
45	to-fprtn-bfe		-estd	-blrb		5049
46	to-fprtn-bfe		-estd	-blrb-3w		5034
47	to-fprtn-bfe	+t	-estd	-blrb-3w		5030
48	depth-first	+t		-bstd		4660
49	to-fprtn-bse		-elrb	-bstd		4606
50	to-fprtn-bse		-elrb-3w	-bstd		4605
51	to-fprtn-bse		-elrb	-blrb-3w		4441
52	to-fprtn-bse		-elrb	-blrb		4427
53	to-fprtn-bse		-elrb-3w	-blrb-3w		4401
54	to-fprtn-bse		-elrb-3w	-blrb		4400
55	to-fprtn-bse		-estd	-bstd		4269
56	to-fprtn-bse		-estd	-blrb		4131
57	to-fprtn-bse		-estd	-blrb-3w		4117
58	depth-first			-bstd		4006
59	depth-first	+t		-blrb		3397
60	depth-first	+t		-blrb-3w		3395
61	depth-first			-blrb-3w		3054
62	breadth-first	+t	-elrb	-bstd		3047

Parsed sentences per second

63	breadth-first	+t -elrb-3w	-bstd	3044
64	depth-first		-blrb	3031
65	earley	-elrb	-bstd	2875
66	breadth-first	+t -estd	-bstd	2871
67	earley	-elrb-3w	-bstd	2858
68	earley	-estd	-bstd	2765
69	breadth-first	-elrb-3w	-bstd	2516
70	breadth-first	-elrb	-bstd	2513
71	breadth-first	-estd	-bstd	2419
72	breadth-first	+t -elrb	-blrb	2260
73	breadth-first	+t -elrb-3w	-blrb	2255
74	breadth-first	+t -elrb	-blrb-3w	2246
75	breadth-first	+t -elrb-3w	-blrb-3w	2246
76	breadth-first	+t -estd	-blrb-3w	2205
77	earley	-elrb-3w	-blrb	2191
78	earley	-elrb	-blrb	2186
79	earley	-elrb	-blrb-3w	2177
80	breadth-first	+t -estd	-blrb	2173
81	earley	-elrb-3w	-blrb-3w	2148
82	earley	-estd	-blrb-3w	2110
83	earley	-estd	-blrb	2093
84	breadth-first	-elrb-3w	-blrb	1983
85	breadth-first	-elrb	-blrb-3w	1971
86	breadth-first	-elrb	-blrb	1966
87	breadth-first	-elrb-3w	-blrb-3w	1966
88	breadth-first	-estd	-blrb-3w	1928
89	breadth-first	-estd	-blrb	1917

Figure 20.2: Performance comparison of each algorithm variant for the MovistarBot corpus and flattened and pseudo-determinized grammar; fully colored rows correspond to the fastest variants, and only intense color rows correspond to algorithms that perform the whole chain of treatment (rows 18–89 except 34 & 38).

20.2 Interpretation

For the case of the non-flattened grammar, our fastest translator (12 to-fprtn-tbe -elrb -blrb) is 2.12 times faster than the plain depth-first translator (67 depth-first -bstd), the one used by Unitex,[3] while the plain Earley translator (37 earley -estd -bstd), the one used by Outilex,[4] is 1.64 times faster. For the case of the flattened grammar, our fastest translator is 1.45 times faster (position 18) than the plain depth-first translator (position 58), while the plain Earley (position 68) is 1.39 times slower. The speedup factors w.r.t. the plain depth-first algorithm, for each one of the fastest translator variants, are

- non-flattened: 2.12 ('to-fprtn-tbe'), 1.74 ('to-fprtn-bse'), 1.64 ('earley'), 1.54 ('to-fprtn-bfe'), 1.15 ('depth-first') and 0.68 ('breadth-first')

- flattened: 1.45 ('to-fprtn-tbe'), 1.42 ('to-fprtn-bfe'), 1.16 ('depth-first'), 1.15 ('to-fprtn-bse'), 0.76 ('breadth-first') and 0.72 ('earley')

Even for the optimized versions of the Earley and depth-first translators, translator 'to-fprtn-tbe' is the fastest one in both cases, and 'breadth-first' is the worst one.

The performance drop for each stage of treatment of the fastests FPRTN-based translators, for the non-flattened and flattened grammars, and taking the fastest Earley acceptor (3 & 5 'earley -o') as reference, is:

- 35% & 37% for adding output transitions to the Earley trace (10 & 12 'to-fprtn'),

- 35% & 37% (negligible) for additionally building map ζ_s'' and performing the variable initializations required for either extracting the top-ranked blackboard (7 & 14 'to-fprtn-top -elrb') or every blackboard by means of blackboard set processing (6 & 15 'to-fprtn-zpps') and, finally

---

[3]Unitex implements some optimizations to accelereate the evaluation of lexical masks which have not been taken into account here (see Paumier, 2003, Vol. 1, sec. 2.1.2.2, p. 120), though the same optimizations would apply for any algorithm.

[4]Additionally, Outilex's algorithm performs an on-the-fly determinization of the grammar which we have not taken into account (see Blanc, 2006, sec. 2.8.4, pg. 68); this operation accelerates further grammar applications reusing determinized substructures during previous grammar applications at the expense of increasing the grammar size.

Figure 20.3: Performance comparison of the fastest algorithm variants —excluding 'to-fprtn-zpps' and 'to-fprtn-top' since they perform as 'to-fprtn'— for grammar of figure 14.1 and input $a^n b^n$; 'fprtn-tbe' is the only algorithm performing the whole chain of treatment and whose performance does not drop exponentially w.r.t. n, but linearly.

- 37% & 38% for additionally extracting the top-ranked blackboard (12 & 18 'to-fprtn-tbe'), or

- 48% & 51% for additionally extracting every blackboard by means of blackboard set processing (20 & 49 'to-fprtn-bse'), or

- 54% & 40% for additionally extracting every blackboard by means of a breadth-first traversal (34 & 24 'to-fprtn-bfe'),

while the performance drop of the straightforward adaptation of the Earley acceptor for output generation is

- 51% & 70% (30 & 65 'earley').

As we can see, we have managed to extend the original Earley acceptor for the generation of the top-ranked output with a performance drop of 37%–38% (for the MovistarBot grammar) instead of 51%–70%. We can expect a performance drop of up to 48%–51% for an algorithm extracting the m top-ranked blackboards.

The performance drops of the other translator algorithms due to output generation (fastest translators against their respective fastest acceptors) are:

- 79% & 70% for breadth-first traversal (72 & 62 'breadth-first' vs 18 & 3 'breadth-first -o') and

- 92% & 95% for depth-first traversal (61 & 48 'depth-first' vs 1 & 1 'depth-first -o').

As we can see, the breadth-first algorithm is not only less efficient than the FPRTN-based ones, but the performance drop due to output generation is also higher. Obviously, the performance drop of the depth-first translator is the highest since its acceptor-only version stops once the first interpretation is found, instead of searching for every possible interpretation as for the other algorithms.

20.2.1 Overheads

Obviously, more complex algorithms have a greater overhead than simpler ones. Relative overheads between the different algorithms can be observed in figure 20.3 as the different performances for $n = 0$:[5] the algorithms are

---

[5]The lower the performance, the greater the overhead.

just requested to traverse a single ε-transition in order to either accept or to translate the empty string into sequence '*', but more complex algorithms perform additional operations that are amortized for higher values of n. If we compare either translator or acceptor algorithms only, we can see that:

- FPRTN-based algorithms have the greatest overheads, as could be expected due to the construction of an intermediate representation of the set of outputs (the output FPRTN),

- depth-first algorithms have the lowest overheads, as could be expected from the most straightforward algorithms, and

- Earley and breadth-first algorithms have similar intermediate overheads.

Among the FPRTN-based algorithms, 'fprtn-bse', 'fprtn-tbe' and 'fprtn-bfe' have similar overheads; obviously, the difference between these 3 algorithms and algorithm 'to-fprtn' is quite greater since 'to-fprtn' skips the generation of the outputs accepted by the FPRTN. Finally, Earley algorithms have a slightly higher overhead than breadth-first ones due to the use of more complex ESs (5-tuples instead of triplets for the case of the translators, and quadruplets instead of pairs for the case of the acceptors).

20.2.2   Asymptotic costs

The applied grammar for the case of figure 20.3 generates an exponential number of outputs w.r.t. the input length ($|a^n b^n|$). Obviously, every algorithm generating the list of every possible output will have an exponential cost w.r.t. the input length, namely any variants of 'to-fprtn-bse', 'to-fprtn-bfe', 'earley', 'depth-first' and 'breadth-first'. Algorithm 'breadth-first -o' computes acceptance only, yet it generates an exponential number of ESs w.r.t. the input length and, therefore, has an exponential cost. Algorithm 'depth-first -o' generates the same kind of ESs than 'breadth-first -o', but explores only a single potential interpretation at each given moment instead of all of them. Since the first explored path is already found to be an interpretation, it avoids to explore the remaining paths; since the length of this path is linear w.r.t. the input length, the algorithm cost is linear for this case, though for other cases an exponential number of ESs might have to be explored before finding the first interpretation. Algorithm 'earley' computes every ES but following another format which allows for factoring out

common call substructures, which results in a linear number of ESs w.r.t. the input length. Algorithm 'to-fprtn' increments this structure with output transitions, resulting in an output FPRTN accepting every possible output, yet keeping the linear time. Algorithm 'to-fprtn-tbe' traverses the FPRTN in order to extract only the top-ranked output; this operation is also performed in linear time thanks to an adaptation of Kahn's topological sort algorithm for the computation of the top-ranked path within the FPRTN. Blackboard-set expansion is an improvement over the breadth-first traversal of the FPRTN, also based on Kahn's topological sort, obtaining similar results to the Earley translator for complex enough grammars and inputs: for the case of the exponential grammar, n must be equal to or greater than 6, but lower recursion degrees will be enough for grammars having bigger call substructures whose treatment can be factored out; note that the exponential grammar has only 6 states an 7 transitions, while the MovistarBot grammar has 1359 states and 3141 transitions. Indeed, better results are expected for general domain grammars with millions of states and transitions (recall that the MovistarBot grammar mainly represents sentences requesting for some mobile service).

Since 'to-fprtn-tbe' is an adaptation of 'to-fprtn-bse' for extracting only the top-ranked output, we can expect that an efficient implementation of an algorithm generating the m top-ranked blackboards will have a cost between 'to-fprtn-tbe' and 'earley', depending on m. This would allow for a compromise between performance and the maximum number of outputs to generate, for applications taking advantage of multiple outputs upon ambiguity.

20.2.3   Flattening

Flattening the grammar increases performance by a factor between 1.43 and 5.05; Earley algorithms are the less affected (1.43–2.5) and breadth-first and depth-first algorithms are the most affected (3.25–5.05): the efficient treatment of call transitions compensates, to some extent, the lack for this grammar optimization. FPRTN-based algorithms with breadth-first expansion are slightly less affected than the breadth-first translators (3.02–3.94 against 3.65–4.05), since pruning the FPRTN reduces the number of ESs to explore. Finally, algorithms based on FPRTNs and blackboard set processing are slightly more affected than Earley-based ones (2.11–2.84 against 1.43–2.5).

20.2.4   Set and map implementations

The use of double-linked red-black trees instead of the standard red-black trees for the implementation of set and map structures accelerates the iterative traversal of the sets —operation required by all the algorithms but 'depth-first -o'— and allows for selectively removing elements from the structures without having to rebalance the corresponding trees —operation required by the FPRTN-based algorithms only. However, we have used a Knuth-like algorithm (section 2.3.5, pg. 45) instead of a Cormen-like (section 2.3.6, pg. 50) in order to add elements to the trees, the former algorithm requiring additional element comparisons. The Knuth-like algorithm can take advantage of a 3-way comparator (section 2.3.9, pg. 55) in order to reduce the number of comparisons when set elements are sequences: comparing two sequences α and β having a maximum common prefix of length n will require n applications of the 3-way comparator and either n or $2n$ comparisons with a 'less-than' operator (n if $\alpha < \beta$ and $2n$ otherwise, since in the latter case comparison $\beta < \alpha$ will also be performed). In the presented cases, the number of avoided comparisons by using the 3-way comparator does not seem to compensate the additional number of comparisons due to the use of Knuth's algorithm, hence it would be better to simply use Cormen's algorithm, which cannot be improved with a 3-way comparator (we leave this improvement to a future work). For the case of output generation, we have experienced no improvement or even a slow-down when applying either the flattened or the non-flattened version of the MovistarBot grammar (0.7–1.01). For the case of sets of execution states and the non-flattened grammar, we have obtained the same positive results either using or not the 3-way comparator: speedup factors between 1.02 and 1.37, mostly affecting either the FPRTN-based ones not performing a breadth-first exploration of the FPRTN (1.16–1.37) or the non-FPRTN based algorithms not generating output at all (1.15–1.23). For the other algorithms, speedup factors stay between 1.02 and 1.17, the FPRTN-based one being the most affected. For the case of the flattened grammar, speedup factors are quite reduced in every case, staying between 1.02 and 1.12; flattening the grammar results in an important reduction of the impact of this optimization —probably due to a reduction in the size of the computed sets of execution states— though this will not be possible for natural language grammars not focused in a so specific domain such as the sentences requesting for mobile services.

20.2.5 Trie-string optimization

For the MovistarBot use case, outputs are not sequences; hence, the optimization of string management based on tries can only be applied to algorithms using stacks of states, namely the ones performing a breadth-first or depth-first exploration of either the grammar or the generated output FPRTN. The speedup depends on the number of generated stacks and their lengths. Depth-first and breadth-first algorithms generate as many stacks as ESs, and the FPRTN-based algorithm generates at least as much stacks as remaining states within the FPRTN after pruning it, and more if the FPRTN contains shared call substructures. The stack lengths depend on the number of successive unresolved calls. Since the flattened grammar has no calls, the stack lengths are always zero, though it is still more expensive to manage an empty array than a pointer to the root of a trie. Speedup factors for the non-flattened and flattened grammars are 1.16–1.43 and 1.13–1.37 (breadth-first), 1.14–1.19 and 1.11–1.21 (depth-first) and 1.11–1.19 and 1–1.02 (FPRTN-based combined with a breadth-first exploration).

20.2.6 Joint optimizations

Some algorithms can benefit from both the use of more efficient set and map implementations and the trie-based optimization. In those cases, maximum speedups for the flattened and non-flattened versions of the MovistarBot grammar are:

- 'to-fprtn-bse +t -elrb -bstd': 1.3 and 1.12, while '+t' alone yields 1.19 and 1.02 and '-elrb' alone yields 1.17 and 1.12,

- 'breadth-first +t -elrb -bstd': 1.4 and 1.26, while '+t' alone yields 1.31 and 1.21 and '-elrb' alone yields 1.08 and 1.06, and

- 'breadth-first -o +t -elrb': 1.64 and 1.45, while '+t' alone yields 1.43 and 1.36 and '-elrb' alone yields 1.23 and 1.1.

In these cases, the trie-based optimization is more significant than the optimization based on double-linked red-black trees with Knuth's algorithm. We recall that both optimizations can be further improved: the former by using ternary search trees instead of tries and the latter by using Cormen's algorithm instead of Knuth's. Note that modified versions of FPRTN-based algorithms based on blackboard set processing have been considered whole

new algorithms instead of simple optimizations, and that multiple implemen-
tation choices offered by the own programming language have been omitted.
The fact is, the greater the number and complexity of the components con-
stituting an algorithm, the greater the chance to find or to have missed some
applicable optimization. Hence, not only our algorithms based on FPRTNs
and blackboard set processing perform better than the others, but their im-
plementation is more likely to be improved than the one of the other al-
gorithms. It must also be taken into account that further optimizing an
algorithm solving a complex problem, such as natural language parsing, is
analogous to tightening a screw: as we approach the optimal solution —
whatever it is— achieving another quarter of turn requires a considerable
effort, and the screw has already been turned several times in 50 years of
research in natural language parsing.

Chapter 21

Conclusion

This work has focused on the optimization of the algorithms of application of local grammars (Gross, 1997), taking as reference those of the Unitex (Paumier et al., 2009; Paumier, 2008) and Outilex (Blanc and Constant, 2006b,a) systems: a top-down depth-first algorithm (Aho et al., 1986, sec. 4.4, p. 181) and an Earley-like algorithm (Blanc, 2006, sec. 3.5, p. 89), respectively. Local grammars are *recursive transition networks with output* defined on an alphabet of predicates called *lexical masks*. These masks are powerful linguistic operators which ease the construction of natural language grammars: simple expressions can be used in order to represent potentially large sets of words (or *tokens*, chapter 5) complying with a set of constraints on their semantic and/or morphosyntactic properties, which are described in an electronic dictionary (chapter 4).

The adequacy of local grammars for the description of natural language phenomena has already been proved (Roche and Schabes, 1997; Català and Baptista, 2007; Martineau et al., 2007; Laporte et al., 2008b,a). As can be expected from a formalism for the representation of natural language grammars, the application of local grammars requires flexible algorithms, such as those based on top-down (Aho et al., 1986, sec. 4.4, p. 181) and Earley (Earley, 1970) parsers.[1] Other not-so-flexible parsers such as LR (Knuth, 1965), CYK's (Cocke and Schwartz, 1970; Younger, 1967; Kasami, 1965), and Tomita's (Tomita, 1987) are not viable or not so straightforward, either because

- they require the grammars to be deterministic and non-ambiguous,

---

[1] A brief description of the original Earley parser is given in appendix C.

- they require to transform the grammar into some normal form, or

- they require to build a table whose size depends on the size of the input alphabet, which for the case of local grammars can be too large: the potentially infinite set of words and symbols of the language.

21.1   Our contributions

As stated in Boullier (2003), it seems difficult to find a technique that would improve the throughput of context-free parsers due to the huge amount of research that has already been performed by the parsing community. The same applies for the case of recursive transition networks due to their equivalence with context-free grammars. Surpassing both the top-down depth-first and Earley-like algorithms has required to conceive a whole new algorithm and several other optimizations; we have followed an iterative approach, refining the previous solutions until achieving lower execution times.

21.1.1   Formal description of finite-state machines and their algorithms of application

We have started by formally describing the different machines and algorithms of application in order to study their properties, strengths and weaknesses. It must be noted that Unitex's and Outilex's local grammars are not exactly of the same kind: Outilex introduced new kinds of output generation to local grammars, such as weights (Blanc, 2006, sec. 3.3, p. 85) and feature structures built by means of unification processes (Blanc, 2006, sec. 4.3, p. 118). We have built a general mathematical framework for the formal description of any kind of finite-state machine (briefly summarized in section. 1.7.2, p. 25), including machines on an alphabet of lexical masks instead of input symbols (chapter 6); this framework not only copes with the different kind of machines treated by both the Unitex and Outilex systems, but can be used as a base for future extensions. Within this framework, we have first given a general description of finite-state machine (chapter 7), then refined this description for the case of finite-state automata (chapter 8), tries (chapter 9), finite-state transducers (chapters 10 and 11), recursive transition networks with and without output (chapters 12–14) and filtered-popping recursive transition networks (chapters 15 and 16), a new kind of machine; describing first the

simpler automata types is an easier and better-structured approach, allowing for factoring out properties and proofs common to the different machine types:

- finite-state transducers (FSTs) can be obtained by extending finite-state automata (FSAs) with output generation,

- recursive transition networks (RTNs) can be obtained by extending FSAs with a subroutine jump mechanism, which corresponds to the implementation of the evaluation of the equivalent non-terminal symbols of the context-free grammar,[2]

- recursive transition networks with output (RTNOs) can be obtained by combining the two previous extensions, and

- filtered-popping recursive transition networks (FPRTNs) can be obtained by extending RTNs with additional restrictions upon the termination of subroutine jumps.

Based on these formalisms, we have first defined top-down breadth-first algorithms of application; once the equations describing the behaviour of the different machines are given, defining these algorithms is straightforward. Moreover, these algorithms can be easily modified in order to obtain both the top-down depth-first and Earley algorithms, serving as a common base for their formal definition and comparison:

- Top-down depth-first algorithms produce the same steps of execution (or execution states) than the corresponding top-down breadth-first algorithms, but in a different order (following a depth-first exploration of the machine instead of breadth-first). Top-down depth-first algorithms are simpler, requiring to store a single execution state at a time instead of having to manage the sets of every possible execution state for each input prefix, which is more time consuming. In turn, further additions of the same execution state to the sets of execution states are skipped, avoiding having to compute twice every execution that would follow them.

- Earley-like algorithms perform a breadth-first exploration of the grammar, also building sets of execution states, but use a more complex

---

[2]We have given a short description of context-free grammars in appendix B.

representation and management of the execution states. As benefits, they support left-recursive grammars and are able to factor out the computation of subroutine jumps from alternative analyses of the same input prefixes.

As their main weakness, all these algorithms have an exponential worst-case cost, even the Earley-like algorithm in spite of the polynomial worst-case cost (n^3) of the original Earley parser (Sastre and Forcada, 2007, 2009): if the number of outputs to generate increases exponentially w.r.t. the input length, computing the list of outputs for a given input sequence cannot have a worst-case cost below exponential. Such cases occur in natural language grammars: for instance, let the outputs represent tags to be inserted between the sentence constituents in order to build every possible parse tree of the given natural language sentences; the number of trees to generate increases exponentially w.r.t. the number of unresolved prepositional phrase attachments within the sentences (Ratnaparkhi, 1998). As for context-free grammars, the size of the corresponding RTNs does not need to be increased exponentially in order to represent such natural language structures, since their representation is factored out by means of subroutine jumps. The original Earley acceptor keeps an analogous factored representation of the execution state structures (the execution traces), which is lost when extending the execution states with partial outputs: the concurrent analyses that where meant to be joined together during and after the computation of a subroutine jump are now joined during the subroutine jump only, since the combination of the different partial outputs computed before the jump with those computed during the jump result in different execution states after the jump; at least, the internal computation of subroutine jumps is factored out, to some extent.

21.1.2   Trie string management

As a first optimization, we have improved the incremental construction of sequences —namely partial outputs and stacks of return states— by storing them in tries and representing them as pointers to the nodes of the trie corresponding to the end of the sequences (chapter 9); sequence copies and comparisons are then reduced to single operations on pointers, and appending or removing suffixes —the typical involved operations— can be efficiently done on the tries since the pointers give direct access to the end-of-sequence nodes. *Trie string management*, as we have called this optimization, produces

a significant increase in performance for the case of the breadth-first and depth-first parsers, but has a negative impact on the Earley-like algorithms; these algorithms do not use stacks of return states, and factoring out the computation of partial outputs of common subroutine jumps requires an additional operation on the tries which is not so efficient: appending a whole trie branch to another existent trie branch.

21.1.3 A first algorithm of application of local grammars based on filtered-popping recursive transition networks

As a first milestone, we have conceived filtered-popping recursive transition networks (FPRTNs), a new kind of finite-state machine for the compact representation of the sets of outputs (or translations) for a given input sequence and RTNO, and an algorithm computing such FPRTNs in time n^3 in the worst case (Sastre, 2009). FPRTNs are analogous to the shared parse forests (Lang, 1991) that result from the application of a CFG with the original Earley parser, though instead of being custom data structures they are a kind of RTN; therefore, theory and algorithms on graphs, finite-state automata and recursive transition networks can be reused or extended for the case of FPRTNs.

Once a FPRTN is computed, we prune it in order to remove every useless path due to grammar paths accepting some input prefix but not leading to a whole input translation. Due to the ambiguity of the language, this simplifies the FPRTN considerably. The FPRTN can then be graphically represented as a kind of finite-state automaton for its visualization.[3] In order to cope with applications requiring a list of translations rather than a FPRTN, we have conceived a first algorithm for the generation of the language of FPRTNs, based on a breadth-first traversal (section 16.2, p. 327). Obviously, the exponential worst-case cost cannot be avoided if the language of the FPRTN is to be generated, but pruning first the FPRTN avoids the construction of useless partial translations. As for the breadth-first algorithms, we have optimized this algorithm by means of trie string management.

---

[3]Indeed, we have developed a tool for the generation of a Graphviz dot file (Gansner and North, 2000) in order to visualize the resulting FPRTNs, for debugging purposes; more information on Graphviz can be found in http://www.graphviz.org

21.1.4   Implementation and application to the Movistar-Bot project

The given formal descriptions have served as the base for the implementation of the different machine structures and their algorithms of application in an object-oriented programming language: C++ (Stroustrup, 2000), using the Standard Template Library (see, for instance, Josuttis, 1999) and some Boost libraries (http://www.boost.org). We have further adapted these implementations for their exploitation in an industrial natural language application provided by the enterprise Telefónica I+D:[4] the translation of sentences in Spanish requesting for mobile services (e.g.: sending SMSs, downloading games to our mobile phone, subscribing to alert services, etc.) into commands that the MovistarBot —an AIML chatterbot (Wallace, 2004) accessible through Microsoft's Windows Live Messenger— can easily understand (Sastre et al., 2009).[5] As part of the project, we have built the corresponding local grammars for the translation of such request sentences, and built a test corpus in order to verify the grammar coverage and control over-recognition. As benefits, this project has allowed us to test the robustness of our implementations and to compare the different algorithm performances in a final natural language application.

21.1.5   Automatic assignment of weights to grammar transitions

As many other final applications, the MovistarBot takes as input a unique possible translation of each user sentence. Upon ambiguity, it is up to our NLP engine to choose the right interpretation. We have extended the grammar with weight output and implemented a procedure for the automatic assignment of weights to each grammar transition, depending on the specificity of the lexical masks labeling each transition: transitions with more restrictive lexical masks are to be preferred since they result in finer sentence descriptions (section 6.4, p. 115). However, the whole set of translations is still to be computed since an interpretation may start with a fine description but then become coarser than the others. Once every interpretation is com-

---

[4]Telefónica I+D is a research and development enterprise and member of the Telefónica group, leader of the telecommunications market in Spain and Latin America and which also enjoys a significant footprint in Europe.

[5]AIML stands for Artificial Intelligence Mark-up Language.

puted, the *top-ranked* one is chosen. Thanks to this technique, we have been able to properly treat ambiguous sentences requesting for mobile services.

21.1.6 Grammar optimizations

We have measured the performance of the algorithms when applying both a flattened and a non-flattened version of the MovistarBot grammar to the whole corpus of request sentences.[6] Note that, in general, natural language grammars cannot be flattened due to recursivity, though a partial flattening is still possible (recursive calls can be recursively inlined a finite number of times, leaving the call transitions as is within the last inlining). However, this operation increases the grammar size exponentially w.r.t. the amount of call nesting levels to flatten; hence, it may not be viable for large coverage grammars.

Both grammar versions have been pseudo-minimized before their application. We have described this process in section 7.11, p. 159, based on the minimization algorithm given in van de Snepscheut (1985, sec. 3.1, p. 67). This process is mainly based on pseudo-determinization, which in turn consists in determinizing the machine regarding it as a FSA over an input alphabet of RTNO transition labels (input/output pairs and calls). Note that full determinization is not generally possible for the case of machines with output or with recursive calls. We have used both the flattening and pseudo-minimization programs included in Unitex —flattening and then pseudo-minimizing the grammar, or only pseudo-minimizing it— rather than reimplemented the corresponding algorithms. Outilex proposes an *on-the-fly* determinization of the involved grammar substructures during each particular application; however, this will not accelerate the algorithms of application except for repeated applications of the same grammar substructures, since determinization is performed while applying the grammars. We have rather chosen to keep Unitex's approach, which is more general (applicable to any machine), simpler and better-structured (determinization and grammar application are two separated processes).

---

[6]Flattening consists in replacing call transitions (the subroutine jumps) by the whole called machine substructures; this is analogous to function inlining in, for instance, C and C++ programming languages. This procedure has been described in sections 12.8 and 13.6, pp. 239 and 263.

21.1.7   First experimental results

While the Earley-like algorithm had the best performance for the non-flattened grammar, it had the worst one for the flattened grammar.[7] Our FPRTN-based algorithm was the best for the flattened grammar, but came second for the non-flattened grammar. The breadth-first algorithm performed badly in both cases: managing sets of execution states instead of single states, as for the depth-first algorithm, is expensive; the Earley-like algorithm amortizes the added complexity when there are calls whose computations can be factored out and, additionally, the FPRTN-based algorithm also amortizes the added complexity by avoiding the generation of useless partial outputs. However, the added complexity of building the FPRTN and then generating its language is not compensated enough w.r.t. the Earley algorithm, for the non-flattened case.

21.1.8   Double-linked red-black trees with aggressive element removal for efficient set management

GNU's implementation of the Standard Template Library (and many other implementations) use red-black trees for the representation of sets. One of the main drawbacks of this implementation is that, when pruning the FPRTN, the removal of each FPRTN state implies a tree rebalance. We have avoided this rebalancing by using an alternative set representation: double-linked red-black trees (chapter 2); moreover, this structure allows for an *aggressive* element removal: the tree structure is no longer maintained, but only the double links at each tree node. The non-aggressive use of double-linked red-black trees was proposed by Das et al. (2008) for optimizing the iterative traversals of sets. Indeed, the use of double-linked red-black trees has not only accelerated the FPRTN-based algorithm but every algorithm building sets of execution states, that is, every algorithm but depth-first. As result, the speedup factor of the FPRTN-based algorithm w.r.t. the depth first algorithm is even greater, but the Earley-like algorithm has also been improved: the Earley-like algorithm still performs better than the FPRTN-based algorithm for the case of the non-flattened grammar, but not as much as before.

---

[7]See the previous chapter for the exact performance figures.

21.1.9 Blackboard set processing

As a second milestone, we have conceived an efficient method for the generation of the language of a FPRTN we have called *blackboard set processing* (sections 10.9, 13.8 and 16.3, pp. 205, 265 and 329). This method avoids to compute twice the same partial output by following a topological sort of the FPRTN. Additionally, some partial results computed during the construction of the FPRTN can be reused here. The new FPRTN-based algorithm finally performs better than the Earley-like algorithm for the non-flattened grammar, but slightly worse than the depth-first algorithm for the flattened grammar: the additional cost of computing a topological sort of the FPRTN is not sufficiently amortized in this case.

21.1.10 Computing the top-ranked output in time n^3

As a third and final milestone, we have extended blackboard set processing for generating the top-ranked output represented by the FPRTN (section 18.2). The topological sort is used here for generating only the greatest possible weight and for marking the corresponding FPRTN path. This path is then traversed for generating only the top-ranked output. The exponential worst-case cost is finally avoided, reducing it to that of the original Earley parser (n^3). This algorithm is, finally, the best performing for both the flattened and non-flattened versions of the MovistarBot grammar.

21.1.11 Final considerations

It is possible to define more complex algorithms which become aware of situations in which certain computations can be either factored out or avoided. However, noticing such situations does not come without an added cost, which is not amortized if the grammar does not contain structures allowing for them to happen. However, such situations do happen with natural language grammars due to their ambiguity and complexity. For the MovistarBot grammar, which applies to a very restricted domain of the language, our FPRTN-based algorithm already performs better than both the top-down depth-first and the Earley-like algorithms, either for a flattened or a non-flattened version of the grammar. Moreover, this algorithm is the only one having a polynomial worst-case cost instead of exponential. Hence, we expect the performance difference between our algorithm and the others to

be considerably greater for the case of grammars covering a wider spectrum of natural language structures.

21.2   Future work

First of all, we are willing to test our FPRTN-based algorithms with complex and large coverage grammars, such as those that can be semi-automatically built from lexicon-grammar tables for the analysis of simple sentences (Paumier, 2003, sec. 1.3, p. 28). Blanc (2006, sec. 4.5.1, p. 143) follows a similar procedure for the semi-automatic construction of grammars describing complex sentences. However, this procedure will first require extending our algorithm in order to support unification grammars; due to the possibility of generating incompatible feature structures, such extension is not trivial. Apart from unification, other interesting extensions that would allow for an easier and more structured definition of linguistic data are:

- Input sentences represented by means of text automata (acyclic FSAs) instead of sequential inputs (section 5.3.2, p. 101). Apart from lexical ambiguity, text automata represent the possible segmentations of sentences. Up to now, we have coded multi-words and attached words (verbs followed by enclitic pronouns) inside the grammars, hence lexical and syntactic representation layers are not completely separated. This extension has already been done for the Earley translator for RTNs with output by Blanc (2006, sec. 3.5.1, p. 90).

- To modify our algorithms in order to efficiently locate within a text every sequence that is accepted by a RTN with output, and to compute their respective translations,[8] as done by the Unitex (Paumier, 2008, sec. 6.8, p. 137) and Outilex systems (Blanc, 2006, sec. 3.5.1, p. 90). As a workaround, one can define a grammar that first consumes any number of tokens, then calls the grammar representing the sequences to locate, then consumes again any number of tokens.

- To add support for multiwords, for instance as done by Multiflex (Savary, 2009).

---

[8]In French, *application glissante d'une RTN*.

- To add support for attached words (e.g.: enclitic pronouns in Spanish), for instance as done by the Apertium system (see Forcada et al., 2010, chap. 3, p. 17).

Other extensions that would have a considerable impact on the recognition rate of the MovistarBot, and of any chatterbot in general, are:

- Recognizing common abbreviations that are used in conversations based on short text messages (see, for instance, Fairon et al., 2006 and Fairon and Sébastien, 2007).

- Error tolerance, either by using approximate string matching (Levenshtein, 1966; Damerau, 1964; Bentley and Sedgewick, 1997; Baeza-Yates and Navarro, 1998; Mihov and Schulz, 2004) or by relaxing the unification constraints (Fouvry, 2003). In both cases, weights can be used in order to penalize sentence interpretations that assume one or more mistakes.

Finally, possible optimizations that are worth considering for a more efficient application and management of the grammars and the dictionaries are:

- To use alternative data structures to the one proposed by Revuz (1991) that allow for a greater compression rate and, most of all, that allow for modifying dictionary entries directly on the compressed format of the dictionary (Ciura and Deorowicz, 2001; Daciuk et al., 2000, 2005; Carrasco and Forcada, 2002); with Revuz's (1991) approach it is necessary to compress the entire dictionary again in order to take into account any modification, independently of the number of entries that have been changed.

- Use Cormen's/Andersson's addition algorithms (sections 2.3.6 and 2.3.7, pp. 50 and 52) instead of Knuth's (section 2.3.5, p. 45) in order to add elements to set data structures represented by a double-linked red-black tree (section 2.6, p. 64). The former algorithms perform, on the average, a lesser amount of comparisons; hence, they can be expected to be faster.

- To unroll the loops of Cormen's/Andersson's addition algorithm (section 2.3.8, p. 55) and the algorithm for the iterative traversal of sets (section 2.3.4, p. 44) in order to avoid trivial assignments.

- To test efficient structures for the management of sets of elements other than double-linked red-black trees, such as scapegoat trees (Galperin and Rivest, 1993), double-linked B-trees (section 2.7.5, p. 68), treaps (Seidel and Aragon, 1996) and skip lists (Pugh, 1990).

- To implement an efficient set and map library allowing for concurrent accesses (section 2.7.8, p. 70); such library would allow for a trivial extension of the breadth-first, Earley-like and FPRTN-based algorithms for taking advantage of multi-core processors by exploring multiple transitions concurrently. Currently, there exist multiple proposals of parallel versions of well-known parsers such as:

    - LR (Hendrickson, 1995),
    - CYK's (Grishman and Chitrao, 1988; Hill and Wayne, 1991; Janssen et al., 1992),
    - Tomita's (Numazaki and Tanaka, 1990), and
    - Earley's (Janssen et al., 1992; Bruschi and Pighizzini, 1994; Sandstrom, 2004)

- To use ternary search trees (Bentley and Sedgewick, 1997) instead of tries in order to optimize the incremental construction of strings (trie string management, chapter 9). We can expect an increase in performance since ternary search trees require less dynamic memory allocations and deallocations: while data structures representing trie nodes contain a map of letters to other trie nodes,[9] the data structures representing the nodes of a ternary search tree contain a structure having 3 fields.

- To filter the grammar before its application according to the sentence to analyse (Boullier and Sagot, 2007).

- To replace the grammar substructure allowing for the recognition of the n first input symbols by a deterministic transducer (a prefix overlay transducer: Marschner, 2007).

- To transform the grammar into Paumier's (2004) weak Greibach normal form before its pseudo-determinization; handcrafted grammars usually

---

[9]Recall that maps are to be represented by other dynamic structures such as red-black trees.

contain subgrammars whose purpose is to group other subgrammars: they simply offer a choice between multiple subgrammars, forcing the parser to explore multiple calls that may not be able to consume even the next input symbol. The weak Greibach normal form ensures that at least one input symbol is consumed before initiating any call, hence avoiding a completely blind initialization of such sets of subgrammar calls.

- As an alternative to the previous optimization, one can allow to explicitly mark subgrammars that are to be "inlined", as for function inlining in C++. A pre-treatment procedure would replace every call to the marked subgrammars by the own subgrammars. A similar feature is implemented in the Outilex system, although it is not mentioned in the manual (Blanc and Constant, 2006b): individual calls are marked for inlining rather subgrammars.

- To extend the Earley parser with lookahead, as proposed by Leiss (1990), in order to reduce the amount of calls that are explored but do not lead to any interpretation of the input. The weak Greibach normal form may no longer be required once this optimization is implemented.

- To accelerate the Earley parser by means of a guide that "foresees" the grammar rules that will allow for recognizing the whole input sentence (Boullier, 2003). The guide is to have a 100% recall, but not a 100% precision. This guide is to be used in Earley's predictor, avoiding initiating useless subgrammar calls. For instance, the previous optimization can be seen as a kind of guide, though others can be defined: for the case of CFGs, take only into account productions that either rewrite a non-terminal as a sequence of non-terminals or as a sequence of terminals and non-terminals where the non-terminals appear in the remaining input sequence and in the same order.

- To precompute Earley-like ε-closure execution states as LR states (McLean and Horspool, 1996).

- Following (Aycock and Horspool, 2001), to precompute the translations produced during the resolution of deletable calls without consuming

input in order to prematurely complete such calls inside Earley's predictor. Apart from accelerating the computation of the ε-closure, this optimization eliminates the need for an ε-completer.

Part IV

Appendices

Appendix A

Predicate hierarchy and codes

We summarize here the set of lexical masks and predicates —along with their syntax— described in chapter 6.

- Lexical masks
  - token: `[%@]<TOKEN>`
  - Literal masks
    * Literal word mask
      · Case sensitive word masks: `@word`
      · Case insensitive word masks: `%word`
    * Literal symbol masks: `[%@]symbol`
  - Character class masks
    * word: `[%@]<MOT>`
    * digit: `[%@]<NB>`
    * punctuation symbol: `[%@]<PNC>`
    * Case-dependent word masks
      · uppercase: `[%@]<MAJ>`
      · lowercase: `[%@]<MIN>`
      · proper noun: `[%@]<PRE>`
    * Negated character class masks: `[%@]<!...>`
  - Dictionary-based masks
    * known word: `[%@]<DIC>`

* ∗ Constrained dictionary word masks
  * · lemma mask: `[%@]<canonical_form>`
  * · semantic-features mask:
    `[%@]<`$[+-]$`?`$\mathtt{Sem}_1$$[+-]$`...`$[+-]$$\mathtt{Sem}_n$`>`
  * · lemma and semantic-features mask:
    `[%@]<canonical_form.`$[+-]$`?`$\mathtt{Sem}_1$$[+-]$`...`$[+-]$$\mathtt{Sem}_n$`>`
  * · semantic and possible-inflectional-features mask:
    `[%@]<`$[+-]$`?`$\mathtt{Sem}_1$$[+-]$`...`$[+-]$$\mathtt{Sem}_l$`:`$\mathtt{flc}_{11}$`...`$\mathtt{flc}_{1n}$`:...:`
    $\mathtt{flc}_{m1}$`...`$\mathtt{flc}_{mn}$`>`
  * · lemma, semantic and possible-inflectional-features mask:
    `[%@]<canonical_form.`$[+-]$`?`$\mathtt{Sem}_1$$[+-]$`...`$[+-]$$\mathtt{Sem}_l$`:`
    $\mathtt{flc}_{11}$`...`$\mathtt{flc}_{1n}$`:...:`$\mathtt{flc}_{m1}$`...`$\mathtt{flc}_{mn}$`>`
* ∗ Negated dictionary masks: `[%@]<!...>`

- • ε-predicates

  - – blank-insensitive ε-predicate: `[%@]<E>`
  - – Blank-sensitive ε-predicate
    * ∗ mandatory-blank ε-predicate: `[%@]\`$\sqcup$
    * ∗ forbidden-blank ε-predicate: `[%@]#`

Appendix B

Context-free grammars

Context-free grammars (CFGs) are mathematical objects for grammar representation, useful for parsing both formal (Aho et al., 1986; Hopcroft et al., 2000; Brüggemann-Klein and Wood, 2003) and natural languages (Jurafsky and Martin, 2008; Paumier, 2003; Roche, 1999; Roche and Schabes, 1997; Silberztein, 1994). We give here the notation used in appendix C for the description of the Earley parser, which was originally conceived for CFGs. More extensive material on CFGs can be found in Autebert et al. (1997), Hopcroft et al. (2000, chap. 5, p. 169), Sipser (2006, chap. 2, p. 100) and Jurafsky and Martin (2008, chap. 9, p. 319).

Definition 288 (Context-free grammar). *A context-free grammar is a structure $G = (N, T, P, S)$ where*

- *N is a finite alphabet of* non-terminal *symbols,*

- *T is a finite alphabet of* terminal *symbols,*

- *$T \cap N = \emptyset$,*

- *$P : N \to (N \cup T)^*$, with ε as the empty sequence of terminals and/or non-terminals, is a finite* production *application, and*

- *$S \in N$ is the start non-terminal of the grammar, also called the grammar's* axiom *or* start symbol,

Production rules —also called rewrite rules, or simply productions or rules— are expressions of the form $A \to \alpha$; non-terminal A is called the left-side *of*

the rule or rule's head, *and* $\alpha \in (N \cup T)^*$ *is called the right-side of the rule or rule's* body. *Productions describe the possible compositions of non-terminal symbols as a sequence of terminal and/or non-terminal symbols.*

Definition 289 (Production direct derivation). *Consider two sequences* α *and* β *in* $(N \cup T)^*$ *such that* $\alpha = \alpha_1 A \alpha_2$, *with* α_1 *and* α_2 *in* $(N \cup T)^*$ *and* $A \in N$; *we say* β *is* directly derivable *from* α *iff there exists a production of the form* $A \to \gamma$ *and* $\beta = \alpha_1 \gamma \alpha_2$, *and we represent it as* $\alpha \Rightarrow \beta$.

Definition 290 (Production derivation). *Consider two sequences* α *and* β *in* $(N \cup T)^*$; *we say* β *is derivable from* α *iff one of the following conditions holds:*

1. $\alpha = \beta$, *or*

2. $\alpha \Rightarrow \beta$, *or*

3. *there exists a finite sequence of terminals and/or non-terminals* $\beta_1 \ldots \beta_n$ *such that*

   - $\alpha \Rightarrow \beta_1$, *and*
   - $\beta_i \Rightarrow \beta_{i+1}$, *for* $i = 1 \ldots n - 1$, *and*
   - $\beta_n \Rightarrow \beta$.

In order words, we say β *is derivable from* α *iff there exists a possibly empty finite sequence of production applications rewriting* α *as* β, *and we represent it as* $\alpha \stackrel{*}{\Rightarrow} \beta$. *Additionally, we represent as* $\alpha \stackrel{+}{\Rightarrow} \beta$ *the possibility of deriving* β *from* α *by applying one or more rewrite rules, that is, when either the second or the third previous conditions apply but not the first one.*

Definition 291 (Deletable non-terminal). *We say a non-terminal* A *is deletable iff* $A \stackrel{+}{\Rightarrow} \varepsilon$, *that is, either there exists a production of the form* $A \to \varepsilon$ *or a production of the form* $B \Rightarrow \varepsilon$ *with* $A \stackrel{+}{\Rightarrow} B$.

Definition 292 (Language of a CFG). *The language represented by a CFG* G, $L(G)$, *is the set of terminal sequences derivable from the grammar's axiom:*

$$L(G) = \{w \in T^* : S \stackrel{+}{\Rightarrow} w\} \tag{B.1}$$

CFGs allow for structured language definitions. Terminals correspond to the words or symbols of the language and non-terminals to sentence components. Non-terminal definitions are reused in the definition of "higher-level" non-terminals up to the grammar's axiom, which is defined as any complete language sentence. Non-terminals may have several alternative definitions depending on the variability of the sentence component they represent (e.g.: a non-terminal DET representing any determiner would have an alternative definition per determiner). The set of productions below...

$$
\begin{aligned}
DET &\rightarrow \text{the} \\
N &\rightarrow \text{garden} \\
N &\rightarrow \text{house} \\
NP &\rightarrow DET\ N
\end{aligned}
$$

...define determiners as word "the", nouns as either word "garden" or word "house", and noun phrases (NP) as a determiner (DET) followed by a noun (N). If this set of productions would define a complete grammar with (NP) as the grammar's axiom, the possible language sentences would be "the garden" and "the house".

CFGs have a greater generative power than regular expressions. A classic example of language that can be represented with a CFG but not with a regular expression is $a^n b^n$. This language can be represented by means of the following CFG productions:

$$
\begin{aligned}
S &\rightarrow \varepsilon \\
S &\rightarrow aSb
\end{aligned}
$$

A concrete sequence of the language would derived as follows:

$$
S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \ldots a^n Sb^n \ldots a^n b^n. \tag{B.2}
$$

As we can see, CFGs allow for a synchonous generation of the left and right contexts of non-terminals. In other words, CFGs can implement a counter of rewrites on the immediate left context of a non-terminal, which can be consulted during the generation of the immediate right context of the same non-terminal. However, no counters can be implemented with regular expressions: $a^* b^*$ represents any sequence $a^n b^n$ but also any sequence $a^n b^m$ with $n \neq m$. At most, a regular expression of the form

$$
\varepsilon|ab|aabb|aaabbb| \ldots |a^k b^k \tag{B.3}
$$

could represent the language $a^n b^n$ for $n = 0 \dots k$, but not for n beyond k (see pumping lemma for regular expressions in either Hopcroft et al., 2000, sec. 4.1.1, p. 126 or Sipser, 2006, sec. 1.4, p. 77).

There are more powerful formalisms than CFGs, for instance Turing machines. A classic example of language that cannot be represented with a CFG is $a^n b^n c^n$: CFG counters are limited to the immediate left and right contexts of each non-terminal, so it is not possible to access a counter at a greater distance (see pumping lemma for context-free languages in Sipser, 2006, sec. 2.3, p. 123). Turing machines (Turing, 1936) are a formal model of computers as we know them nowadays. These machines are able to access information related to any previously processed input symbol. We briefly describe Turing machines in section 10.5, p. 193; more extensive descriptions can be found in Hopcroft et al. (2000, chap. 8, p. 307) and Sipser (2006, chap. 3, p. 137).

Definition 293 (Extended context-free grammar). *Extended context-free grammars (ECFG) are CFGs where the production bodies may also contain regular expressions.*

ECFGs do not have a greater generative power than CFGs but provide a more compact way of representing a set of productions. For instance, the following CFG

$$
\begin{align}
PREP &\rightarrow \text{in} \tag{B.4}\\
PREP &\rightarrow \text{with} \tag{B.5}\\
PP &\rightarrow PREP\ NP \tag{B.6}\\
PPS &\rightarrow \varepsilon \tag{B.7}\\
PPS &\rightarrow PP\ PPS \tag{B.8}\\
NP &\rightarrow DET\ N\ PPS \tag{B.9}
\end{align}
$$

defines a noun phrase (NP) as a determiner (DET) followed by a noun (N) followed by zero, one or more prepositional phrases (PPS), where a prepositional phrase (PP) is a preposition ($PREP$) followed by a noun phrase. The Kleene star can be used for representing any sequence of prepositional phrases and therefore removing the need for defining non-terminal PPS. As well, the two productions defining prepositions could be joined together with

the disjunction operator. This equivalent ECFG would be

$$PREP \quad \rightarrow \quad \text{in} \mid \text{with} \qquad\qquad (B.10)$$
$$PP \quad \rightarrow \quad PREP\ NP \qquad\qquad (B.11)$$
$$NP \quad \rightarrow \quad DET\ N\ PP^* \qquad\qquad (B.12)$$

Other formalisms equivalent to CFGs are pushdown automata (Oettinger, 1961; Schützenberger, 1963; Evey, 1963),[1] automata with recursive calls (Gallier et al., 2003), syntactic diagrams (see for instance Jensen and Wirth, 1974, chap. 0, p. 3) and RTNs (Woods, 1970).[2]

---

[1] An extensive description of pushdown automata, including the proof of equivalence w.r.t. CFGs, can be found in Hopcroft et al. (2000, chap. 6, p. 219); shorter descriptions can be found in Sipser (2006, sec. 2.2, p. 109) and Autebert et al. (1997, chap. 5, p. 29)

[2] We describe RTNs in chapter 12, p. 219.

Appendix C

Earley's parser

We briefly describe here the Earley parser (Earley, 1970), an efficient algorithm of application of CFGs (see appendix B) without deletable non-terminals for natural language parsing.[1] This description is inspired by the one given in Leiss (1990). A more extensive discussion can be found in Jurafsky and Martin (2008, chap. 10).

Definition 294 (Dotted rule). *Execution states (ESs) are dotted rules $A \to \alpha_1 \bullet \alpha_2, [i, j]$, where α_1 is the prefix of the rule's body that has already been explored, α_2 the unexplored remaining part and $[i, j]$ a closed interval corresponding to the input segment that has been derived from α_1 (see top of fig. ??). We say a dotted rule is* complete *iff the dot is at the end of the rule's body; otherwise we say it is* incomplete. *Given a terminal or non-terminal symbol σ right after the dot of a dotted rule x_s, we say x_s expects symbol σ or σ is the symbol expected by rule x_s.*

Definition 295 (Dotted rule derivation). *Earley's parser is based on three dotted rule derivation mechanisms (see middle of fig. ??):*

- scanner*: from $A \to \alpha_1 \bullet a\alpha_2, [i, j]$ derives $A \to \alpha_1 a \bullet \alpha_2, [i, j+1]$ iff $a = a_{j+1}$, that is, if the expected symbol is terminal a, it scans the input for a and in case of match it shifts the dot right after a and increments the right bound of the input interval one unit,*

- predictor*: from $A \to \alpha_1 \bullet B\alpha_2, [i, j]$ derives $B \to \bullet\beta, [j, j]$ for every rule $B \to \beta$, that is, if the expected symbol is a non-terminal B, it*

---

[1]Though deletable non-terminals are not supported, the idea on how to adapt the algorithm is given in Earley (1970)

expands B by creating the corresponding dotted rules with an empty input interval starting and finishing where the interval of the original dotted rule finishes,

- completer*: from $B \to \beta\bullet, [j,k]$ derives $A \to \alpha_1 B \bullet \alpha_2, [i,k]$ for every dotted rule $A \to \alpha_1 \bullet B\alpha_2, [i,j]$, that is, whenever a dotted rule having a head non-terminal B is complete, reduces B by shifting one position to the right the dot of every dotted rule expecting B and whose input interval finishes where the one of the complete rule starts; the resulting input interval is the concatenation of both intervals.*

Definition 296 (Initial and acceptance SESs). *Given a CFG grammar $G = (N, T, P, S)$, its initial and acceptance Earley SESs are $\{S' \to \bullet S, [0,0]\}$ and $\{S' \to S\bullet, [0,l]\}$, respectively, where S' is a non-terminal not in V_n (see bottom of fig. ??).*

By creating this grammar "super-axiom" symbol, S', Earley's algorithm naturally explores every axiom rule $S \to \alpha$ during the first iteration; once the algorithm execution finishes, successful parses can be identified by tracking back the derivations that have yielded the dotted rule having the super-axiom as head.[2]

Algorithm C.1 *cfg_earley_parser* is the original Earley's parser for CFGs without deletable non-terminals. The algorithm creates a parsing chart or sequence of $l+1$ SESs $V_0 \ldots V_l$ for a given input sentence $a_1 \ldots a_l$ of l words and a CFG grammar G; the grammar is treated as a global variable where terminal symbols are parts-of-speech which are to be compared with the ones of the input words. The algorithm starts by adding to V_0 the super-axiom dotted rule $S' \to S\bullet, [0,0]$ and marking it as unexplored. Then, for each iteration $k = 0 \ldots l$ it explores the dotted rules in V_k, applying for each one the derivation mechanism corresponding to the rule: if the dotted rule expects a terminal symbol, scanner (algorithm C.3 *cfg_earley_scanner*), if it expects a non-terminal, predictor (algorithm C.2 *cfg_earley_predictor*), and if the rule is complete, completer (algorithm C.4 *cfg_earley_completer*). Iterations follow as long as they start with a non-empty V_k, which otherwise would mean that it was not possible to derive terminal a_k from the super-axiom dotted rule, or until every possible derivation has been computed for the whole input.

---

[2]Supposing that the sentence has at least one word and that the whole sentence has been consumed.

Dotted rule:

Scanner:

Predictor:

Completer:

$$a_0 \ldots a_l \in L(G) \iff$$

Figure C.1: Graphical representation of Earley's algorithm; from top to bottom, a dotted rule $A \to \alpha_1 \bullet \alpha_2$ aligned with an input $a_1 \ldots a_l$, the three dotted rule derivation mechanisms and the input string recognition condition.

The predictor derivation mechanism fills V_0 with every dotted rule derived from the super-axiom rule by expanding the expected non-terminals; every dotted rule expecting a terminal is processed by the scanner mechanism and, in case of match with input a_1, the corresponding dotted rules are added to V_1. The completer mechanism cannot be activated during the construction of V_0, since the algorithm does not handle grammars with deletable calls: every completion must involve a complete dotted rule in the current V_k and one or more dotted rules in preceding SESs V_i expecting the head terminal of the complete rule, that is, every complete rule must derive at least one input symbol. Iterations for V_k follow until the computation of V_l or until an empty V_k is derived, which would mean input a_k did not match any expected terminal. V_l will contain every dotted rule derivable from the super-axiom dotted rule and able to derive the whole input sentence. The sentence is recognized if V_l contains the complete super-axiom dotted rule $S' \rightarrow S\bullet, [0, l]$. By tracking back the derivation paths of this dotted rule we can retrieve the derivation trees of every possible interpretation of the sentence.

We end this appendix with a couple of examples of execution: figure **??** illustrates how the Earley parser factors out the exploration of a grammar subtree among two dotted rules, keeping the number of dotted rules per SES constant, and figure **??** illustrates how left-recursive CFGs are handled without falling into an infinite loop.

Algorithm C.1 cfg_earley_parser($\sigma_1 \ldots \sigma_l$)

Input: $\sigma_1 \ldots \sigma_l$, an input string of length l
Output: r, a Boolean indicating whether the input string belongs to L
 1: allocate_memory_for_chart(V^{l+1})
 2: $V_0 \leftarrow \emptyset$
 3: $E \leftarrow \emptyset$
 4: unconditionally_add_enqueue_es($V_0, E, (S' \rightarrow \bullet S, [0,0])$)
 5: $k \leftarrow 0$
 6: $E' \leftarrow \emptyset$
 7: **while** $E \neq \emptyset \wedge k < l$ **do**
 8:     $V_{k+1} \leftarrow \emptyset$
 9:     **repeat**
10:        $x_s \leftarrow$ dequeue(E)
11:        **if** incomplete(x_s) **then**
12:           **if** terminal_symbol_after_dot(x_s) **then**
13:              cfg_earley_scanner(V_k, V_{k+1}, E', x_s)
14:           **else**
15:              cfg_earley_predictor(V_k, E, x_s)
16:           **end if**
17:        **else**
18:           cfg_earley_completer(V^{l+1}, E, x_s)
19:        **end if**
20:     **until** $E = \emptyset$
21:     $k \leftarrow k + 1$
22:     swap(E, E')
23: **end while**

Algorithm C.2 cfg_earley_predictor(V, E, x_s)

Input: V, the current SES or parsing chart element
       E, the current queue of unexplored ESs
       $x_s = (A \rightarrow \alpha_1 \bullet B\alpha_2, [i,j])$, an ESs or dotted rule
Output: V, after expanding non-terminal B
Output: E, after enqueuing the new derived ESs
 1: **for each** $(B \rightarrow \beta) \in G$ **do**
 2:     add_enqueue_es($V, E, (B \rightarrow \bullet\beta, [j,j])$)
 3: **end for**

Algorithm C.3 cfg_earley_scanner(V, W, E', a_{j+1}, x_s)

Input: V, the current SES or parsing chart element
Input: W, the next SES
Input: E', the next queue of unexplored ESs
Input: σ, the input symbol to scan
Input: $x_s = (A \rightarrow \alpha_1 \bullet a\alpha_2, [i, j])$, an ESs or dotted rule
Output: W, after scanning input for terminal σ'
Output: E', after enqueuing the new derived ESs
 1: **if** part $-$ of $-$ speech(a_{j+1}) $= a$ **then**
 2:     add_enqueue_es($W, E', (A \rightarrow \alpha_1 a \bullet \alpha_2, [i, j+1])$)
 3: **end if**

Algorithm C.4 cfg_earley_completer(V^{l+1}, E, x_s)

Input: V^{l+1}, the parsing chart
        E, the current queue of unexplored ESs
        $x_s = (B \rightarrow \beta\bullet), [j, k]$, an ES or dotted rule
Output: V^{l+1}, after expanding non-terminal B
        E, after enqueuing the new derived ESs
 1: **for each** $(A \rightarrow \alpha_1 \bullet B\alpha_2, [i, j]) \in V_j$ **do**
 2:     add_enqueue_es($V_k, E, (A \rightarrow \alpha_1 B \bullet \alpha_2, [i, k])$)
 3: **end for**

V_0			
1	$S' \to \bullet S$	$[0,0]$	initial super-axiom
2	$S \to \bullet aSb$	$[0,0]$	predictor$(1, p_1)$
3	$S \to \bullet aSc$	$[0,0]$	predictor$(1, p_2)$
4	$S \to \bullet x$	$[0,0]$	predictor$(1, p_3)$
V_1			
5	$S \to a \bullet Sb$	$[0,1]$	scanner$(2, a)$
6	$S \to a \bullet Sc$	$[0,1]$	scanner$(3, a)$
7	$S \to \bullet aSb$	$[1,0]$	predictor$(5, p_1)$
8	$S \to \bullet aSc$	$[1,0]$	predictor$(5, p_2)$
9	$S \to \bullet x$	$[1,0]$	predictor$(5, p_3)$
V_2			
10	$S \to a \bullet Sb$	$[1,2]$	scanner$(7, a)$
11	$S \to a \bullet Sc$	$[1,2]$	scanner$(8, a)$
12	$S \to \bullet aSb$	$[2,2]$	predictor$(10, p_1)$
13	$S \to \bullet aSc$	$[2,2]$	predictor$(10, p_2)$
14	$S \to \bullet x$	$[2,2]$	predictor$(10, p_3)$
V_3			
15	$S \to x \bullet$	$[2,3]$	scanner$(14, x)$
16	$S \to aS \bullet b$	$[1,3]$	completer$(15, 11)$
17	$S \to aS \bullet c$	$[1,3]$	completer$(15, 11)$
V_4			
18	$S \to aSb \bullet$	$[1,4]$	scanner$(16, b)$
19	$S \to aS \bullet b$	$[0,4]$	completer$(18, 5)$
20	$S \to aS \bullet c$	$[0,4]$	completer$(18, 6)$
V_5			
21	$S \to aSb \bullet$	$[0,5]$	scanner$(19, b)$
22	$S' \to S \bullet$	$[0,5]$	completer$(21, 1)$

$$
\begin{aligned}
p_1 &: S \to aSb \\
p_2 &: S \to aSc \\
p_3 &: S \to Sx
\end{aligned}
$$

Figure C.2: At the left, a simple left-recursive CFG recognizing the language $a^n x(b|c)^n$ and, at the right, execution trace of Earley's parser for this CFG and input *aaxbb*. Notice that, for this example, a top-down parser would exponentially increase the cardinality of the generated SES whilst Earley's parser manages to keep it constant thanks to the factoring out of the exploration of common grammar subtrees: prediction of symbol S is shared for both productions p_1 and p_2.

V_0			
1	$S' \to \bullet S$	$[0,0]$	initial super-axiom
2	$S \to \bullet Sa$	$[0,0]$	predictor$(1, p_1)$
3	$S \to \bullet b$	$[0,0]$	predictor$(1, p_2)$
V_1			
4	$S \to b\bullet$	$[0,1]$	scanner$(3, b)$
5	$S' \to S\bullet$	$[0,1]$	completer$(4,1)$
6	$S \to S \bullet a$	$[0,1]$	completer$(4,2)$
V_2			
7	$S \to Sa\bullet$	$[0,2]$	scanner$(6, a)$
8	$S' \to S\bullet$	$[0,2]$	completer$(7,1)$
9	$S \to S \bullet a$	$[0,2]$	completer$(7,2)$
V_3			
10	$S \to Sa\bullet$	$[0,3]$	scanner$(9, a)$
11	$S' \to S\bullet$	$[0,3]$	completer$(10,1)$
12	$S \to S \bullet a$	$[0,3]$	completer$(10,2)$
$\vdots$			
V_l			
$3l+1$	$S \to Sa\bullet$	$[0,l]$	scanner$(3l, a)$
$3l+2$	$S' \to S\bullet$	$[0,l]$	completer$(3l+1,1)$
$3l+3$	$S \to S \bullet a$	$[0,l]$	completer$(3l+1,2)$

$$p_1 \; : \; S \; \to \; Sa$$
$$p_2 \; : \; S \; \to \; b$$

Figure C.3: At the left, a simple left-recursive CFG recognizing the language ba^n and, at the right, execution trace of Earley's parser for this CFG and input ba^{l-1}. Notice that, where a top-down parser would enter into an infinite loop due to the left recursion, Earley's parser expands symbol S only once but reduces it several times.

Appendix D

Kahn's topological sorter

We briefly describe here Kahn's algorithm (Kahn, 1962) for topologically sorting a PERT network (PERT, 1958a,b).[1]

Definition 297 (Directed graph). *A directed graph G is an ordered pair (V, A) where V is a set of vertices or nodes and $A \subseteq V \times V$ is a set of ordered pairs of nodes, also called arcs, directed edges or arrows.*

Definition 298 (PERT network). *A PERT network is a directed or undirected graph where nodes represent points in time and edges (n, n') represent tasks that should take place between temporal points n and n', that is, during interval $[n, n']$.*

Considering the directed case, a PERT network is a graphical representation of a set of temporal restrictions over the order in which a set of tasks can be performed; for instance, supposing that graph of figure D.1 represents a PERT network, task of arc $(3, 6)$ is to follow task of arc $(0, 3)$. A topological sort of the network is a total order of the graph nodes respecting the temporal constraints, that is, expressing a possible sequence of execution of every task within the network. Note that such a topological sort is only possible for acyclic graphs.

Algorithm D.1 *kahn_topological_sort* gets as input a directed graph (G, A) and a queue E of graph nodes initially filled with every node having no incoming arcs, and initializes topological sort t as an empty sequence of nodes. Then it dequeues and processes each node in E until there are no more nodes left, traversing the graph by following a possible topological sort.

---

[1]PERT stands for 'program evaluation and review technique'.

Note that any of the nodes initially present in the queue could be the first node of the topological sort. For each dequeued node n, it removes from A every arc of the form (n, m),[2] and enqueues m iff it has no more incoming arcs.[3] If the graph contains no cycles then every arc should have been removed after emptying the queue. If so, the algorithm returns the computed topological sort; otherwise returns $\perp$ in order to indicate that there exists no topological sort for graph G.

Figure D.1 shows an example of directed graph along with the corresponding execution trace of Kahn's algorithm.

Algorithm D.1 kahn_topological_sort(G)

Input: $G = (V, A)$, a directed graph
      E, a queue of nodes initially filled with every node of G without incoming arcs
Output: t, a topological sort of G
 1: $t \leftarrow \varepsilon$
 2: **while** $E \neq \emptyset$ **do**
 3:    $n \leftarrow \text{dequeue}(E)$
 4:    $t \leftarrow tn$
 5:    **for each** $m : (n, m) \in A$ **do**
 6:        $A \leftarrow A - \{(n, m)\}$
 7:        **if** $\nexists n' : (n', m) \in A$ **then**
 8:            $\text{enqueue}(E, m)$
 9:        **end if**
10:    **end for**
11: **end while**
12: **if** $A \neq \emptyset$ **then**
13:    $t \leftarrow \perp$
14: **end if**

---

[2]Tasks starting at n are executed.
[3]Tasks before m are finished, thus tasks starting at m are now available.

n	E	t	action
$\perp$	$0,1,2$	ε	initialize
0	$1,2$	0	$n \leftarrow \mathrm{dequeue}(E)$, $t \leftarrow tn$
0	$1,2$	0	$A \leftarrow A - \{(0,3)\}$
0	$1,2$	0	$A \leftarrow A - \{(0,4)\}$
1	2	$0,1$	$n \leftarrow \mathrm{dequeue}(E)$, $t \leftarrow tn$
1	$2,3$	$0,1$	$A \leftarrow A - \{(1,3)\}$, $\mathrm{enqueue}(E,3)$
2	3	$0,1,2$	$n \leftarrow \mathrm{dequeue}(E)$, $t \leftarrow tn$
2	$3,4$	$0,1,2$	$A \leftarrow A - \{(2,4)\}$, $\mathrm{enqueue}(E,4)$
2	$3,4$	$0,1,2$	$A \leftarrow A - \{(2,5)\}$
3	4	$0,1,2,3$	$n \leftarrow \mathrm{dequeue}(E)$, $t \leftarrow tn$
3	$4,5$	$0,1,2,3$	$A \leftarrow A - \{(3,5)\}$, $\mathrm{enqueue}(E,5)$
3	$4,5$	$0,1,2,3$	$A \leftarrow A - \{(3,6)\}$
3	$4,5,7$	$0,1,2,3$	$A \leftarrow A - \{(3,7)\}$, $\mathrm{enqueue}(E,7)$
4	$5,7$	$0,1,2,3,4$	$n \leftarrow \mathrm{dequeue}(E)$, $t \leftarrow tn$
4	$5,7,6$	$0,1,2,3,4$	$A \leftarrow A - \{(4,6)\}$, $\mathrm{enqueue}(6)$
5	$7,6$	$0,1,2,3,4,5$	$n \leftarrow \mathrm{dequeue}(E)$, $t \leftarrow tn$
7	6	$0,1,2,3,4,5,7$	$n \leftarrow \mathrm{dequeue}(E)$, $t \leftarrow tn$
6	ε	$0,1,2,3,4,5,7,6$	$n \leftarrow \mathrm{dequeue}(E)$, $t \leftarrow tn$

Figure D.1: Acyclic directed graph and execution trace of Kahn's algorithm for this graph.

Bibliography

Abeille, A. (1988). Parsing French with tree adjoining grammar: some linguistic accounts. In *Proceedings of the 12th conference on Computational linguistics*, volume 1 of *COLING '88*, pages 7–12, Stroudsburg, PA, USA. Association for Computational Linguistics.

Abeillé, A. and Blache, P. (2000). Grammaires et analyseurs syntaxiques. In Pierrel, J., editor, *Ingénierie des langues*, pages 51–76. Hermès, Paris.

Adel'son-Vel'skiĭ, G. M. and Landis, E. M. (1962). An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Albahari, J. and Albahari, B. (2010). *C# 4.0: Pocket reference*. O' Reilly, 3rd edition.

Albert, J., Giammarresi, D., and Wood, D. (1998). Extended context-free grammars and normal form algorithms. In Champarnaud, J.-M., Ziadi, D., and Maurel, D., editors, *Workshop on Implementing Automata*, volume 1660 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag.

Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., and Mohri, M. (2007). OpenFst: a general and efficient weighted finite-state transducer library. In Holub, J. and Žďárek, J., editors, *Implementation and Application of Automata*, volume 4783 of *Lecture Notes in Computer Science*, pages 11–23. Springer Berlin / Heidelberg.

Ambati, V. and Kishore, S. (2004). How can academic software research and open source software development help each other? *IEE Seminar Digests*, 2004(908):5–8.

Andersson, A. (1991). A note on searching in a binary search tree. *Software: Practice and Experience*, 21(10):1125–1128.

Andersson, A. (1993). Balanced search trees made simple. In Dehne, F., Sack, J.-R., Santoro, N., and Whitesides, S., editors, *Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, pages 60–71. Springer-Verlag.

Armentano-Oller, C., Corbí-Bellot, A. M., Forcada, M. L., Ginestí-Rosell, M., Montava Belda, M. A., Ortiz-Rojas, S., Pérez-Ortiz, J. A., Ramírez-Sánchez, G., and Sánchez-Martínez, F. (2007). Apertium, una plataforma de código abierto para el desarrollo de sistemas de traducción automática. In Rodríguez Galván, J. R. and Palomo Duarte, M., editors, *Proceedings of the FLOSS International Conference 2007*, pages 5–20. Servicio de Publicaciones de la Universidad de Cadiz.

Autebert, J., Berstel, J., and Boasson, L. (1997). Context-free languages and pushdown automata. In Salomaa, A. and Rozenberg, G., editors, *Handbook of Formal Languages*, volume 1, Word Language Grammar, pages 111–174. Springer-Verlag, Berlin.

Aycock, J. and Horspool, N. (2001). Directly-executable Earley parsing. In Wilhelm, R., editor, *Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 229+. Springer-Verlag.

Aycock, J. and Horspool, N. (2002). Practical Earley parsing. *The Computer Journal*, 45(6):620–630.

Aziz, M. H., Bohez, E. L. J., Parnichkun, M., and Saha, C. (2010). Classification of fuzzy petri nets, and their application. *World Academy of Science, Engineering and Technology*, 72:394–401.

Backus, J. W. (1959). The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *Proceedings of the International Conference on Information Processing*, pages 125–132. UNESCO.

Baeza-Yates, R. A. and Navarro, G. (1998). Fast approximate string matching in a dictionary. In *String Processing and Information Retrieval*, pages 14–22.

Bayer, R. (1972). Symmetric binary B-trees: data structure and maintenance algorithms. *Acta Informatica*, 1:290–306.

Beesley, K. R. and Karttunen, L. (2003). *Finite state morphology.* CSLI Publications, Stanford, California.

Bellman, R. E. (1957). *Dynamic programming.* Princeton University Press, Princeton.

Bender, M. A., Fineman, J. T., Gilbert, S., and Kuszmaul, B. C. (2005). Concurrent cache-oblivious b-trees. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 228–237, New York, NY, USA. ACM.

Bentley, J. and Sedgewick, B. (1997). Fast algorithms for sorting and searching strings. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*.

Bentley, J. L. (1982). *Writing Efficient Programs.* Prentice-Hall, Upper Saddle River, NJ 07458, USA.

Berners-Lee, T., Cailliau, R., Groff, J.-F., and Pollermann, B. (1992). World Wide Web: An information infrastructure for high-energy physics. In *Proceedings of the 2nd Workshop on Artificial Intelligence and Software Engineering for High Energy Physics*, pages 157–164, La Londe, France, January 1992. New Computing Techniques in Physics Research, World Scientific, Singapore. La Londe, France.

Berstel, J. (1979). *Transductions and Context-free Languages.* Teubner Studienbücher, Stuttgart, Germany.

Biswas, D. (2003). Unicode enconding form: A devil in disguise? In *Proceedings of the 23rd Internationalization & Unicode Conference (IUC23)*, Prague, Czech Republic.

Blanc, O. (2006). *Algorithmes d'analyse syntaxique par grammaires lexicalisées: optimisation et traitement de l'ambiguïté.* PhD thesis, Université de Marne-la-Vallée.

Blanc, O. and Constant, M. (2005). Lexicalization of grammars with parametrized graphs. In *Proceedings of the international conference Recent Advances in Natural Language Processing*, pages 117–121, Borovets, Bulgaria.

Blanc, O. and Constant, M. (2006a). Outilex, a linguistic platform for text processing. In *Interactive Presentation Session of Coling-ACL06*, pages 73–76, Morristown, NJ, USA. Association for Computational Linguistics.

Blanc, O. and Constant, M. (2006b). *Outilex platform graphical interface - user guide.*

Blanc, O., Constant, M., and Laporte, E. (2006). Outilex, plate-forme logicielle de traitement de textes écrits. In *Proceedings of TALN'06*, pages 83–92, Leuven, Belgium. UCL Presses universitaires de Louvain.

Blanc, O., Constant, M., and Watrin, P. (2007). A finite-state super-chunker. In Holub, J. and Žďárek, J., editors, *Implementation and Application of Automata*, volume 4783 of *Lecture Notes in Computer Science*, pages 306–308. Springer Berlin / Heidelberg.

Blanco, X. (2000). Les dictionnaires électroniques de l'espagnol (DELASs et DELACs). *Lingvisticæ Investigationes*, 23(2):201–218.

Boons, J.-P., Guillet, A., and Leclère, C. (1976a). *La structure des phrases simples en français : constructions intransitives.* Librairie Droz S.A., Geneve, Switzerland.

Boons, J.-P., Guillet, A., and Leclère, C. (1976b). La structure des phrases simples en français : constructions transitives. Technical Report 6, LADL, Université Paris 7, Paris.

Boons, J.-P., Guillet, A., and Leclère, C. (1992). *La structure des phrases simples en français : constructions transitives locatives.* Librairie Droz S.A., Geneve, Switzerland.

Booth, T. L. (1969). Probabilistic representation of formal languages. In *Proceedings of the 10th Annual Symposium on Switching and Automata Theory (swat 1969)*, pages 74–81, Washington, DC, USA. IEEE Computer Society.

Booth, T. L. and Thompson, R. A. (1973). Applying probability measures to abstract languages. *IEEE Transactions on Computers*, 22(5):442–450.

Boullier, P. (2003). Guided Earley parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT 03)*, pages 43–54, Nancy, France.

Boullier, P. and Sagot, B. (2007). Are very large context-free grammars tractable? In *Proceedings of the 10th International Workshop on Parsing Technologies (IWPT 07)*, Prague, Czech Republic.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). Technical report, World Wide Web Consortium.

Brill, E. (1992). A simple rule-based part of speech tagger. In *HLT '91: Proceedings of the workshop on Speech and Natural Language*, pages 112–116, Morristown, NJ, USA. Association for Computational Linguistics.

Brill, E. (1995). Transformation-based error-driven learning and natural language processing: a case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–565.

Brittain, J. and Darwin, I. F. (2007). *Tomcat: the definitive guide.* O'Reilly, 2nd edition.

Bronson, N. G., Casper, J., Chafi, H., and Olukotun, K. (2010). A practical concurrent binary search tree. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 257–268, New York, NY, USA. ACM.

Bruschi, D. and Pighizzini, G. (1994). A parallel version of Earley's algorithm. Technical Report 114-94, Dipartimento di Scienze dell'Informazione, Univ. of Milan.

Brüggemann-Klein, A. and Wood, D. (2003). On predictive parsing and extended context-free grammars. In Champarnaud, J.-M. and Maurel, D., editors, *Implementation and Application of Automata*, volume 2608 of *Lecture Notes in Computer Science*, pages 277–303. Springer-Verlag.

Butt, M. (2002). Chunk/shallow parsing. Lecture notes.

Carrasco, R. C. and Forcada, M. L. (2002). Incremental construction and maintenance of minimal finite-state automata. *Computational Linguistics*, 28(2):207–216.

Carrasco, R. C. and Oncina, J. (1994). Learning stochastic regular grammars by means of a state merging method. In *ICGI '94: Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, pages 139–152, London, UK. Springer-Verlag.

Català, D. and Baptista, J. (2007). Spanish adverbial frozen expressions. In *Proceedings of the Workshop on A Broader Perspective on Multiword Expressions*, pages 33–40, Prague, Czech Republic. Association for Computational Linguistics.

Choffrut, C. (1977). Une caracterisation des fonctions sequentielles et des fonctions sous-sequentielles en tant que relations rationnelles. *Theoretical Computer Science*, 5(3):325–337.

Choffrut, C. (1978). *Contributions à l'étude de quelques familles remarquables de fonctions rationnelles*. PhD thesis, Université de Paris 7.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124.

Chomsky, N. (1959). On certain formal properties of grammars. *Information and Control*, 2(2):137–167.

Chomsky, N. (1965). *Aspects of the theory of syntax*. MIT Press, Cambridge.

Church, K. W. (1988). A stochastic parts program and noun phrase parser for unrestricted text. In *Proceedings of the second conference on Applied natural language processing*, pages 136–143, Morristown, NJ, USA. Association for Computational Linguistics.

Ciura, M. G. and Deorowicz, S. (2001). How to squeeze a lexicon. *Software—Practice & Experience*, 31(11):1077–1090.

Clavero, M. (2010). Awkward wording. Rephrase: linguistic injustice in ecological journals. *Trends in Ecology & Evolution*, 25(10):552–553. Special Issue: Long-term ecological research.

Cocke, J. and Schwartz, J. T. (1970). Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, New York.

Cohen, D. (1981). On holy wars and a plea for peace. *Computer*, 14:48–54.

Constant, M. (2003a). Converting linguistic systems of relational matrices into finite-state transducers. In *Proceedings of the EACL Workshop on Finite-State Methods in Natural Language Processing*, pages 75–82, Budapest.

Constant, M. (2003b). *Grammaires locales pour l'analyse automatique de textes : Méthodes de construction et outils de gestion*. PhD thesis, Université de Marne la Vallée.

Constant, M. (2009). Microsyntax of Measurement Phrases in French: Construction and Evaluation of a Local Grammar. In *8th International Workshop on Finite-State Methods and Natural Language Processing (FSMNLP'09)*, Pretoria, South Africa.

Constant, M. and Tolone, E. (2008). A generic tool to generate a lexicon for NLP from Lexicon-Grammar tables. In Constant, M., De Gioia, M., Nakamura, T., and Vecchiato, S., editors, *27ème Colloque international sur le Lexique et la Grammaire (LGC'08)*, pages 11–18, L'Aquila, Italy.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to algorithms*. MIT press, Cambridge, Massachusetts, 2nd edition.

Correa, N. (1991). An extension of Earley's algorithm for S-attributed grammars. In *Proceedings of the fifth conference on European chapter of the Association for Computational Linguistics*, EACL '91, pages 299–302, Stroudsburg, PA, USA. Association for Computational Linguistics.

Courtois, B. (1990). Un système de dictionnaires électroniques pour les mots simples du français. *Langue Française*, 87:11–22.

Courtois, B. (2004). Dictionnaires électroniques DELAF anglais et français. In Leclère, C., Laporte, E., Piot, M., and Silberztein, M., editors, *Lexique, Syntaxe et Lexique-Grammaire / Syntax, Lexis & Lexicon-Grammar*, volume 24 of *Lingvisticæ Investigationes Supplementa*, pages 113–123. John Benjamins Publishing Company, Amsterdam, Philadelpia.

Crawford, S. (1983). The origin and development of a concept: the Information Society. *Bulletin of the Medical Library Association*, 71(4):380–385.

Daciuk, J., Maurel, D., and Savary, A. (2005). Dynamic perfect hashing with finite-state automata. In Kłopotek, M. A., Wierzchoń, S., and Trojanowski, K., editors, *Intelligent Information Processing and Web Mining*, volume 31 of *Advances in Soft Computing*, pages 169–178. Springer Berlin / Heidelberg.

Daciuk, J., Mihov, S., Watson, B. W., and Watson, R. E. (2000). Incremental construction of minimal acyclic finite-state automata. *Computational Linguistics*, 26(1):3–16.

Damerau, F. J. (1964). A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176.

Danlos, L. (2005). Automatic recognition of French expletive pronoun occurrences. In *Second International Joint Conference on Natural Language Processing: Companion Volume including Posters/Demos and tutorial abstracts*, pages 73–78, Jeju, Korea.

Das, D., Valluri, M., Wong, M., and Cambly, C. (2008). Speeding up STL set/map usage in C++ applications. In Kounev, S., Gorton, I., and Sachs, K., editors, *Performance Evaluation: Metrics, Models and Benchmarks*, volume 5119 of *Lecture Notes in Computer Science*, pages 314–321. Springer-Verlag.

Davis, M., O'Donovan, W., Fritz, J., and Childress, C. (2000). Linux and open source in the academic enterprise. In *Proceedings of the 28th annual ACM SIGUCCS conference on User services: Building the future*, SIGUCCS '00, pages 65–69, New York, NY, USA. ACM.

Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Devlin, K. (1993). *The joy of sets: Fundamentals of Contemporany set theory*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, 2nd edition.

Dongarra, J. and Hinds, A. R. (1979). Unrolling loops in FORTRAN. *Software: Practice and Experience*, 9(3):219–226.

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Enrique Hamel, R. (2007). The dominance of English in the international scientific periodical literature and the future of language use in science. *AILA Review*, 20(1):53–71.

Evey, R. J. (1963). Application of pushdown-store machines. In *AFIPS '63 (Fall): Proceedings of the November 12-14, 1963, fall joint computer conference*, pages 215–227, New York, NY, USA. ACM.

Fairon, C., Jean, K., and Sébastien, P. (2006). *Le langage SMS*, volume 3.1 of *Cahiers du Cental*. Presses universitaires de Louvain, Louvain-la-Neuve.

Fairon, C. and Paumier, S. (2005). Can we parse without tagging? In Vetulani, Z., editor, *Proceedings of the 2nd Language & Technology Conference (LTC'05)*, pages 473–477, Poznań, Poland. Wydawnictwo Poznańskie Sp. z o.o.

Fairon, C. and Sébastien, P. (2007). De la possibilité de construire un dictionnaire électronique du langage SMS. *Cahiers de Lexicologie*, 90(2):65–72.

Fischer, E. (2000). The evolution of character codes, 1874-1968.

Forcada, M. L. (2006). Open-source machine translation: an opportunity for minor languages. In *Strategies for developing machine translation for minority languages (5th SALTMIL workshop on Minority Languages)*, pages 1–6. organised in conjunction with LREC 2006 (22-28.05.2006).

Forcada, M. L., Bonev, B. I., Ortiz-Rojas, S., Pérez-Ortiz, J. A., Ramírez-Sánchez, G., Sánchez-Martínez, F., Armentano-Oller, C., Montava, M. A.,

and Tyers, F. M. (2010). *Documentation of the open-source shallow-transfer machine translation platform Apertium.* Departament de Llenguatges i Sistemes Informàtics, Universitat d'Alacant.

Forcada, M. L., Tyers, F. M., and Ramírez-Sánchez, G. (2009). The Apertium machine translation platform: five years on. In Pérez-Ortiz, J. A., Sánchez-Martínez, F., and Tyers, F. M., editors, *Proceedings of the First International Workshop on Free/Open-Source Rule-Based Machine Translation*, pages 3–10, Alicante. Departamento de Lenguajes y Sistemas Informáticos, Universidad de Alicante.

Fouvry, F. (2003). Constraint relaxation with weighted feature structures. In *In Proceedings of the 8th International Workshop on Parsing Technologies*, pages 23–25.

Francis, N. W. and Kučera, H. (1982). Frequency analysis of English usage: lexicon and grammar. *Journal of English Linguistics*, 18(1):64–70.

Fredkin, E. (1960). Trie memory. *Communications of the ACM*, 3(9):490–499.

Freksa, C. (1991). Qualitative spatial reasoning. In Mark, D. and Frank, A., editors, *Proceedings of the Conference on Cognitive and Linguistic Aspects of Geographic Space*, pages 361–372. Kluwer Academic Publishers, Dordrecht.

Friburger, N. (2002). *Reconnaissance automatique de noms propres: Application à la classification automatique des textes journalistiques.* PhD thesis, Université de Tours.

Friburger, N. and Maurel, D. (2002). Finite-state transducer cascade to extract proper names in texts. In Watson, B. and Wood, D., editors, *Implementation and Application of Automata*, volume 2494 of *Lecture Notes in Computer Science*, pages 406–410. Springer-Verlag.

Friburger, N. and Maurel, D. (2004). Finite-state transducer cascades to extract named entities in texts. *Theoretical Computer Science*, 313(1):93–104.

Gallier, J. H., Torre, S. L., and Mukhopadhyay, S. (2003). Deterministic finite automata with recursive calls and DPDAs. *Information Processing Letters*, 87(4):187–193.

Galperin, I. and Rivest, R. L. (1993). Scapegoat trees. In *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174.

Gansner, E. R. and North, S. C. (2000). An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233.

Gardent, C., Guillaume, B., Perrier, G., and Falk, I. (2005). Maurice Gross' grammar lexicon and Natural Language Processing. In Vetulani, Z., editor, *Proceedings of the 2nd Language & Technology Conference (LTC'05)*, pages 120–123, Poznań, Poland. Wydawnictwo Poznańskie Sp. z o.o.

Gardent, C., Guillaume, B., Perrier, G., and Falk, I. (2006). Extraction d'information de sous-catégorisation à partir du lexique-grammaire de Maurice Gross. In *Proceedings of TALN'06*, Leuven, Belgium.

Garrido-Alenda, A. and Forcada, M. L. (2002). Comparing nondeterministic and quasideterministic finite-state transducers built from morphological dictionaries. *Procesamiento del lenguaje natural*, 29:73–80.

Garrido-Alenda, A., Forcada, M. L., and Carrasco, R. C. (2002). Incremental construction and maintenance of morphological analysers based on augmented letter transducers. In *Proceedings of TMI 2002 (Theoretical and Methodological Issues in Machine Translation, Keihanna/Kyoto, Japan, March 2002)*, pages 53–62.

Giry-Schneider, J. (1978). *Les nominalisations en français : l'opérateur «faire» dans le lexique*. Librairie Droz S.A., Geneve, Switzerland.

Giry-Schneider, J. (1987). *Les prédicats nominaux en français : les phrases simples à verbe support*. Librairie Droz S.A., Geneve, Switzerland.

Giry-Schneider, J. and Balibar-Mrabti (1993). Classes de noms construits avec avoir. Technical Report 42, LADL, Université Paris 7, Paris.

Gorn, S., Bemer, R. W., and Green, J. (1963). American standard code for information interchange. *Communications of the ACM*, 6(8):422–426.

Graham, S. (2001). From research software to open source. In Wilhelm, R., editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 195–208. Springer Berlin / Heidelberg.

Greibach, S. A. (1965). A new normal-form theorem for context-free phrase structure grammars. *Journal of the ACM*, 12(1):42–52.

Grishman, R. and Chitrao, M. (1988). Evaluation of a parallel chart parser. In *Proceedings of the second conference on Applied natural language processing*, pages 71–76, Morristown, NJ, USA. Association for Computational Linguistics.

Gross, G. (1989). *Les constructions converses du français*. Librairie Droz S.A., Geneve, Switzerland.

Gross, M. (1975). *Méthodes en syntaxe*. Hermann, Paris.

Gross, M. (1982a). Une classification des phrases "figées" du français. *Revue Québécoise de Linguistique*, 11(2):151–185.

Gross, M. (1982b). Une famille d'adverbes figés : les constructions comparatives en comme. *Revue Québécoise de Linguistique*, 13(2):237–269.

Gross, M. (1985). Sur les déterminants dans les expressions figées. *Langages*, 79:89–126.

Gross, M. (1986a). *Grammaire transformationnelle du français, volume 3 : Syntaxe de l'adverbe*. ASSTRIL, Paris.

Gross, M. (1986b). Les nominalisations d'expressions figées. *Langue française*, 69:64–84.

Gross, M. (1988a). Les limites de la phrase figée. *Langages*, 90:7–22.

Gross, M. (1988b). Sur les phrases figées complexes du français. *Langue française*, 77:47–70.

Gross, M. (1993). Les phrases figées en français. *L'information grammaticale*, 59:36–41.

Gross, M. (1996). Lexicon-grammar. In Brown, K. and Miller, J., editors, *Concise Encyclopedia of Syntactic Theories*, pages 224–259. Pergamon Press, Oxford.

Gross, M. (1997). The construction of local grammars. In Roche, E. and Schabes, Y., editors, *Finite State Language Processing*, pages 329–352. MIT Press, Cambridge, MA, USA.

Gross, M. (1999). Lemmatization of compound tenses in English. *Lingvisticæ Investigationes*, 22:71–122.

Gross, M. (2001). Grammaires locales de déterminants nominaux. In Blanco, X., Buvet, P.-A., and Gavriilidou, Z., editors, *Détermination et formalisation*, volume 23 of *Lingvisticæ Investigationes Supplementa*, pages 177–193. John Benjamins Publishing Company, Amsterdam, Philadelpia.

Gross, M. and Senellart, J. (1998). Nouvelles bases statistiques pour les mots du français. In *Actes des 4èmes Journées Internationales d'Analyse Statistique des Données Textuelles (JADT'98)*, pages 335–349. Université de Nice-Sophia-Antipolis.

Guibas, L. J. and Sedgewick, R. (1978). A dichromatic framework for balanced trees. In *SFCS '78: Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Washington, DC, USA. IEEE Computer Society.

Hanke, S. (1999). The performance of concurrent red-black tree algorithms. In Vitter, J. and Zaroliagis, C., editors, *Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*, pages 286–300. Springer-Verlag.

Hansen, J. and Henriksen, A. (2001). The multi map/set of the Copenhagen STL. Technical Report 2001-6, University of Copenhagen.

Harris, Z. S. (1951). *Structural linguistics*. The University of Chicago Press, Chicago & London, and The University of Toronto Press, Toronto 5, Canada.

Harris, Z. S. (1965). Transformational theory. *Language*, 41(3):363–401.

Harris, Z. S. (1968). *Mathematical structures of language*. Wiley, New York, NY, USA.

Harris, Z. S. (1991). *A theory of language and information: a mathematical approach*. Clarendon Press, Oxford.

Hathout, N. and Namer, F. (1998). Automatic construction and validation of French large lexical resources: Reuse of verb theoretical linguistic descriptions. In *Proceedings of the First International Conference on Language Resources and Evaluation*, pages 627–636, Granada, Spain.

Hendrickson, K. J. (1995). A new parallel LR parsing algorithm. In *SAC '95: Proceedings of the 1995 ACM symposium on Applied computing*, pages 277–281, New York, NY, USA. ACM.

Herlihy, M., Lev, Y., Luchangco, V., and Shavit, N. (2006). A provably correct scalable concurrent skip list. In *Proceedings of the OPODIS '06 Conference*.

Hill, J. C. and Wayne, A. (1991). A CYK approach to parsing in parallel: a case study. *SIGCSE Bull.*, 23(1):240–245.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2000). *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2nd edition.

Hsu, C.-N. and Chang, C.-C. (1999). Finite-state transducers for semi-structured text mining. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI) Workshop on Text Mining*, pages 76–82.

Huh, H.-G. (2005). *Délimitation et étiquetage des morphèmes en coréen par ressources linguistiques*. PhD thesis, Université de Marne la Vallée.

Hulden, M. (2009). Foma: a finite-state compiler and library. In *EACL (Demos)*, pages 29–32.

ISO/IEC (1993). Information technology — Coding of moving picture and associated audio for digital storage media at up to about 1.5 mbit/s — Part 3: Audio. Technical Report 11172-3, ISO/IEC, Geneva, Switzerland.

ISO/IEC (1994). Information technology — Digital compression and coding of continuous-tone still images: Requirements and guidelines. Technical Report 10918-1, ISO/IEC, Geneva, Switzerland.

ISO/IEC (1996). Information technology — syntactic metalanguage — extended bnf. Technical Report 14977:1996, ISO/IEC, Geneva, Switzerland.

ISO/IEC (1998). Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1. Technical Report 8859-1, ISO/IEC, Geneva, Switzerland.

ITU (1992). Information technology — Digital compression and coding of continuous-tone still images — requirements and guidelines. Technical Report Recommendation T.81, ITU, Geneva, Switzerland.

Janssen, W., Poel, M., Sikkel, K., and Zwiers, J. (1992). The primordial soup algorithm: a systematic approach to the specification of parallel parsers. In *Proceedings of the 14th conference on Computational linguistics*, pages 373–379, Morristown, NJ, USA. Association for Computational Linguistics.

Jensen, K. and Wirth, N. (1974). *PASCAL user manual and report*. Springer-Verlag New York, Inc., New York, NY, USA. Series Editor-Goos, G. and Series Editor-Hartmanis, J.

Joshi, A. K. (1985). Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions? In Dowty, D. R., Karttunen, L., and Zwicky, A. M., editors, *Natural Language Parsing: Psychological, Computational, and Theoretical Perspectives*, pages 206–250. Cambridge University Press, Cambridge.

Joshi, A. K., Levy, L. S., and Takahashi, M. (1975). Tree adjunct grammars. *Journal of Computer and System Sciences*, 10(1):136–163.

Josuttis, N. M. (1999). *The C++ Standard Library: a tutorial and reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Jung, E.-J. (2005). *Grammaire des adverbes de duree et de date en coréen*. PhD thesis, Université de Marne-la-Vallée.

Jurafsky, D. and Martin, J. H. (2008). *Speech and Language Processing (Prentice Hall Series in Artificial Intelligence)*. Prentice Hall, 2nd edition.

Jussila, T., Heljanko, K., and Niemelä, I. (2005). BMC via on-the-fly determinization. *International Journal on Software Tools for Technology Transfer*, 7(2):89–101.

Kahn, A. B. (1962). Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562.

Kaplan, R. M. and Bresnan, J. (1982). Lexical-Functional Grammar: A formal system for grammatical representation. In Bresnan, J., editor, *The Mental Representation of Grammatical Relations*, pages 173–281. The MIT Press, Cambridge, MA. Reprinted in Mary Dalrymple, Ronald M. Kaplan, John Maxwell, and Annie Zaenen, eds., *Formal Issues in Lexical-Functional Grammar*, 29–130. Stanford: CSLI Publications. 1995.

Kaplan, R. M. and Kay, M. (1994). Regular models of phonological rule systems. *Computational Linguistics*, 20:331–378.

Karttunen, L. (1993). Finite-state constraints. In Goldsmith, J., editor, *The Last Phonological Rule*, pages 173–194. University of Chicago Press.

Karttunen, L. (2001). Applications of finite-state transducers in natural language processing. In Yu, S. and Paun, A., editors, *Implementation and Application of Automata*, volume 2088 of *Lecture Notes in Computer Science*, pages 34–46. Springer-Verlag.

Karttunen, L., Gaál, T., and Kempe, A. (1997). Xerox finite-state tool. Technical report, Xerox Research Centre Europe, Grenoble, Meylan, France.

Karttunen, L., Kaplan, R. M., and Zaenen, A. (1992). Two-level morphology with composition. In *Proceedings of the 14th Conference on Computational Linguistics*, pages 141–148, Morristown, NJ, USA. Association for Computational Linguistics.

Kasami, T. (1965). An efficient recognition and syntax analysis algorithm for context free languages. Scientific Report AF CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Massachusetts.

Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In Shannon, C. and Mccarthy, J., editors, *Automata Studies*, number 34 in Annals of Mathematics Studies, pages 3–41. Princeton University Press, Princeton, N.J.

Klein, D. and Manning, C. D. (2005). Natural language grammar induction with a generative constituent-context model. *Pattern Recognition*, 38(9):1407–1419.

Knuth, D. E. (1965). On the translation of languages from left to right. *Information and Control*, 8(6):607–639.

Knuth, D. E. (1968). Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145.

Knuth, D. E. (1998). *The art of computer programming, volume 3: sorting and searching.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2nd edition.

Koch, R. and Blum, N. (1997). Greibach normal form transformation, revisited. In Reischuk, R. and Morvan, M., editors, *STACS 97*, volume 1200 of *Lecture Notes in Computer Science*, pages 47–54. Springer-Verlag.

Laforest, F. and Badr, Y. (2003). Modèle de couplage de documents structurés et de bases de données. le projet DRUID. *Ingénierie des Systèmes d'Information*, 8(5–6):109–126.

Lang, B. (1991). Towards a uniform formal framework for parsing. In *Current Issues in Parsing Technology*, pages 153–171. Kluwer Academic Publishers.

Laporte, E. (2005). In memoriam Maurice Gross. *Archives of Control Sciences*, 15(3):257–278. Special issue on Human Language Technologies as a challenge for Computer Science and Linguistics. Part I. (2nd Language and Technology Conference).

Laporte, E. (2007). Evaluation of a grammar of French determiners. In *27th Congress of the Brazilian Society of Computation (SBC'07)*, pages 1625–1634. Workshop on Information Technology and Human Language (TIL).

Laporte, E. (2009). Lexicons and grammars for language processing: industrial or handcrafted products? In Rezende, L. M., da Silva, B. C. D., and Barbosa, J. B., editors, *Léxico e gramática: dos sentidos à construção da significação (Lexicon and Grammar: from Meanings to the Construction of Signification)*, Trilhas Lingüísticas, 16. São Paulo: Cultura Acadêmica.

Laporte, E. and Monceaux, A. (2000). Elimination of lexical ambiguities by grammars: the ELAG system. *Lingvisticæ Investigationes*, 22(2):341–367.

Laporte, E., Nakamura, T., and Voyatzi, S. (2008a). A French corpus annotated for multiword expressions with adverbial function. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC '08), Towards a Shared Task for Multiword Expressions (MWE 2008)*, pages 48–51, Marrakech, Morocco.

Laporte, E., Nakamura, T., and Voyatzi, S. (2008b). A French corpus annotated for multiword nouns. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC '08), Towards a Shared Task for Multiword Expressions (MWE 2008)*, pages 27–30, Marrakech, Morocco.

Laporte, E., Ranchhod, E., and Yannacopoulou, A. (2008c). Syntactic variation of support verb constructions. *Lingvisticæ Investigationes*, 31(2):173–185.

Larsen, K. S., Ottmann, T., and Soisalon-Soininen, E. (2001). Relaxed balance for search trees with local rebalancing. *Acta Informatica*, 37(10):743–763.

Leclère, C. (2002). Organization of the lexicon-grammar of French verbs. *Lingvisticæ Investigationes*, 25(1):29–48.

Leclère, C. (2003). The Lexicon-Grammar of French verbs: a syntactic database. In Yuji Kawaguchi, Susumu Zaima, T. T. K. S. M. U., editor, *Proceedings of the First International Conference on Linguistic Informatics*, Linguistic Informatics 1, pages 33–46, Tokyo, Japan.

Lefevre, C. (2009). LHC: the guide. http://cdsweb.cern.ch/record/1165534.

Leiner, B. M., Cerf, V. G., Clark, D. D., Kahn, R. E., Kleinrock, L., Lynch, D. C., Postel, J., Roberts, L. G., and Wolff, S. (2009). A brief history of the Internet. *SIGCOMM Computer Communication Review*, 39(5):22–31.

Leiss, H. (1990). On Kilbury's modification of Earley's algorithm. *ACM Transactions on Programming Languages and Systems*, 12(4):610–640.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707–710.

Liang, S. (1999). *Java Native Interface: Programmer's Guide and Reference.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Lindén, K., Silfverberg, M., and Pirinen, T. (2009). HFST tools for morphology — An efficient open-source package for construction of morphological analyzers. In Mahlow, C. and Piotrowski, M., editors, *State of the Art in Computational Morphology*, volume 41 of *Communications in Computer and Information Science*, pages 28–47. Springer Berlin Heidelberg.

Lombardy, S. and Sakarovitch, J. (2002). V$\overline{\text{AU}}$C$\underline{\text{ANS}}\overline{\text{ON}}$-G, *A package for drawing automata.*

Lynge, S. (2004). Implementing the AVL trees for the CPH STL. Technical Report 2004-1, University of Copenhagen.

Machlup, F. (1962). *The production and distribution of knowledge in the United States.* Princeton University Press, Princeton, N.J.

Marcus, G. F. (2004). Before the word. *Nature*, 431:745.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics - Special Issue on Using Large Corpora: II*, 19(2):313–330.

Marschner, C. (2007). Efficiently matching with local grammars using prefix overlay transducers. In Holub, J. and Žďárek, J., editors, *Implementation and Application of Automata*, volume 4783 of *Lecture Notes in Computer Science*, pages 314–316. Springer-Verlag.

Martineau, C., Tolone, E., and Voyatzi, S. (2007). Les entités nommées: usage et degrés de précision et de désambiguïsation. In *Proceedings of the 26th International Conference on Lexis and Grammar*, pages 105–112, Bonifacio, Corse.

Mason, O. (2004). Automatic processing of local-grammar patterns. In *Proceedings of the Annual Colloquium for the UK Special Interest Group for Computational Linguistics*, pages 166–171, Birmingham, U.K.

McLean, P. and Horspool, R. N. (1996). A faster Earley parser. In Gyimóthy, T., editor, *Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 281–293. Springer.

Mesfar, S. (2007). Named entity recognition for arabic using syntactic grammars. In Kedad, Z., Lammari, N., Métais, E., Meziane, F., and Rezgui, Y., editors, *Natural Language Processing and Information Systems*, volume 4592 of *Lecture Notes in Computer Science*, pages 305–316. Springer Berlin / Heidelberg.

Mihov, S. and Schulz, K. U. (2004). Fast approximate search in large dictionaries. *Computational Linguistics*, 30(4):451–477.

Mohri, M. (1994a). Compact representations by finite-state transducers. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pages 204–209, Morristown, NJ, USA. Association for Computational Linguistics.

Mohri, M. (1994b). Minimization of sequential transducers. In Crochemore, M. and Gusfield, D., editors, *Combinatorial Pattern Matching*, volume 807 of *Lecture Notes in Computer Science*, pages 151–163. Springer Berlin / Heidelberg.

Mohri, M. (1996). On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, 2(1):61–80.

Mohri, M. (1997). Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

Mohri, M., Pereira, F., and Riley, M. (1998). A rational design for a weighted finite-state transducer library. In Wood, D. and Yu, S., editors, *Automata Implementation*, volume 1436 of *Lecture Notes in Computer Science*, pages 144–158. Springer Berlin / Heidelberg.

Mohri, M., Pereira, F., and Riley, M. (2002). Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69–88.

Mohri, M., Pereira, O., and Riley, M. (1996). Weighted automata in text and speech processing. In *In ECAI-96 Workshop*, pages 46–50. John Wiley and Sons.

Molinier, C. and Levrier, F. (2000). *Grammaire des adverbes : description des form en -ment*. Librairie Droz S.A., Geneve, Switzerland.

Mont-Reynaud, B. (1976). Removing trivial assignments from programs. Technical Report STAN-CS-76-544, Computer Science Department, Stanford University, Stanford, California.

Moore, R. C. (2000). Removing left recursion from context-free grammars. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, pages 249–255, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Moseley, C., editor (2010). *Atlas of the World's languages in danger.* UNESCO Publishing, Paris, 3rd edition.

Movva, R. and Lai, W. (1999). MSN Messenger Service 1.0 Protocol.

Nam, J.-S. and Choi, K.-S. (1997). A local grammar-based approach to recognizing of proper names in Korean texts. In *Proceedings of the Fifth Workshop on Very Large Corpora*, pages 273–288. ACL/Tsing-hua University/Hong-Kong University of Science and Technology.

Naur, P., Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A., and Woodger, M. (1960). Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314.

Navarro, G. (2001). A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88.

Navarro, J. R., González, J., Picó, D., Casacuberta, F., de Val, J. M., Fabregat, F., Pla, F., and Tomás, J. (2004). SisHiTra: A hybrid machine translation system fromSpanish to Catalan. In Vicedo, J. L., Martínez-Barco, P., Muñoz, R., and Saiz Noeda, M., editors, *Advances in Natural Language Processing*, volume 3230 of *Lecture Notes in Computer Science*, pages 349–359. Springer-Verlag.

Nenadić, G. (2000). Local grammars and parsing coordination of nouns in serbo-croatian. In Sojka, P., Kopecek, I., and Pala, K., editors, *Text, Speech and Dialogue*, volume 1902 of *Lecture Notes in Computer Science*, pages 143–158. Springer Berlin / Heidelberg.

Newman, H. B., Barczyk, A., and Mughal, A. (2010). Advancement in networks for HEP community. *Journal of Physics: Conference Series*, 219(6):062023.

Ney, H. (1992). Stochastic grammars and pattern recognition. In Laface, P. and Mori, R. D., editors, *Proceedings of the NATO Advanced Study Institute*, pages 313–344. Springer-Verlag.

Numazaki, H. and Tanaka, H. (1990). A new parallel algorithm for generalized LR parsing. In *Proceedings of the 13th conference on Computational linguistics*, pages 305–310, Morristown, NJ, USA. Association for Computational Linguistics.

Oettinger, A. G. (1961). Automatic syntactic analysis and the pushdown store. In Jakobson, R., editor, *Proceedings of Symposia on Applied Mathematics*, volume 12, pages 104–129, Providence, Rhode Island. American Mathematical Society.

Oflazer, K. (1996). Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22:73–89.

Oflazer, K. and Yılmaz, Y. (2004a). Developing finite-state NLP systems with a graphical environment. In Aykanat, C., Dayar, T., and Körpeoglu, I., editors, *Computer and Information Sciences - ISCIS 2004*, volume 3280 of *Lecture Notes in Computer Science*, pages 147–156. Springer Berlin / Heidelberg.

Oflazer, K. and Yılmaz, Y. (2004b). Vi-xfst: a visual regular expression development environment forXerox finite-state tool. In *Proceedings of the 7th Meeting of the ACL Special Interest Groupin Computational Phonology: Current Themes in Computational Phonology and Morphology*, SIGMorPhon '04, pages 86–93, Stroudsburg, PA, USA. Association for Computational Linguistics.

Oncina, J. (1998). The data driven approach applied to the OSTIA algorithm. In Honavar, V. and Slutzki, G., editors, *Grammatical Inference*, volume 1433 of *Lecture Notes in Computer Science*, pages 50–56. Springer Berlin / Heidelberg.

Oncina, J., García, P., and Vidal, E. (1993). Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15:448–458.

Ortiz-Rojas, S., Forcada, M. L., and Sánchez, G. R. (2005). Construcción y minimización eficiente de transductores de letras a partir de diccionarios con paradigmas. *Procesamiento del Lenguaje Natural*, 35:51–57.

Paul, W., Vishkin, U., and Wagener, H. (1983). Parallel dictionaries on 2-3 trees. In Diaz, J., editor, *Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*, pages 597–609. Springer-Verlag.

Paumier, S. (2003). *De la reconnaissance de formes linguistiques à l'analyse syntaxique*. PhD thesis, Université de Marne-la-Vallée.

Paumier, S. (2004). Weak Greibach normal form of recursive transition networks. In *Proceedings of Journées Montoises d'Informatique Théorique*, Liège.

Paumier, S. (2006). *Unitex - Manuel d'utilisation*. First version: 2004.

Paumier, S. (2008). *Unitex 2.1 User Manual*. Université de Marne-la-Vallée.

Paumier, S., Nakamura, T., and Voyatzi, S. (2009). UNITEX, a corpus processing system with multi-lingual linguistic resources. In *eLexicography in the 21st century: new challenges, new applications (eLEX'09)*, pages 173–175.

Paz, A. (1971). *Introduction to probabilistic automata (Computer science and applied mathematics)*. Academic Press, Inc., Orlando, FL, USA.

PERT (1958a). Program evaluation research task, summary report, phase I. Special Projects Office, Bureau of Naval Weapons, Department of the Navy, U. S. Goverment Printing Office, Washington D. C.

PERT (1958b). Program evaluation research task, summary report, phase II. Special Projects Office, Bureau of Naval Weapons, Department of the Navy, U. S. Goverment Printing Office, Washington D. C.

Poibeau, T. (2006). Dealing with metonymic readings of named entities. In *Proceedings of the 28th Annual Conference of the Cognitive Science Society (CogSci 2006)*, Vancouver, Canada.

Pressman, R. S. (2001). *Software Engineering: A Practitioner's Approach.* McGraw-Hill Higher Education, 5th edition.

Pugh, W. (1990). Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, pages 257–286.

Ranchhod, E., Carvalho, P., Mota, C., and Barreiro, A. (2004). Portuguese large-scale language resources for NLP applications. In *Proceedings of LREC 2004*, pages 1755–1758, Lisbon, Portugal.

Ratnaparkhi, A. (1998). Statistical models for unsupervised prepositional phrase attachment. In *ACL-36: Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics*, pages 1079–1085, Morristown, NJ, USA. Association for Computational Linguistics.

Raymond, E. S. (1999). *The cathedral and the bazaar.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition.

Raymond, E. S. (2001). *The cathedral and the bazaar: Musings on Linux and open source by an accidental revolutionary.* O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Revuz, D. (1991). *Dictionnaires et lexiques: méthodes et algorithmes.* PhD thesis, Université Paris 7.

Revuz, D. (1992). Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189.

Roche, E. (1993). Une représentation par automate fini des textes et des propriétés transformationnelles des verbes. *Lingvisticæ Investigationes*, 17(1):189–222.

Roche, E. (1999). *Finite-state transducers: parsing free and frozen sentences*, chapter 11, pages 108–120. Cambridge University Press, New York, NY, USA.

Roche, E. and Schabes, Y. (1997). *Finite-State Language Processing.* MIT Press, Cambridge, MA, USA.

Ron, D., Singer, Y., and Tishby, N. (1994). Learning probabilistic automata with variable memory length. In *In Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory*, pages 35–46, New Brunswick, New Jersey. ACM Press.

Sagot, B. and Tolone, E. (2009). Exploitation des tables du Lexique-Grammaire pour l'analyse syntaxique automatique. *Arena Romanistica, Journal of Romance Studies*, 4:302–312.

Sandstrom, G. (2004). A parallel extension of Earley's parsing algorithm. http://www.cs.earlham.edu/~sandsgr/files/proposal.pdf.

Sarkar, V. (2001). Optimized unrolling of nested loops. *International Journal of Parallel Programming*, 29(5):545–581.

Sastre, J. M. (2006a). Computer tools for the management of lexicon-grammar databases. In *Proceedings of TALN'06*, pages 600–608, Leuven, Belgium.

Sastre, J. M. (2006b). HOOP: a Hyper-Object Oriented Platform for the management of linguistic databases. Presentation in 25th Lexis and Grammar Conference, Palermo, Italy, September 6-9. Abstract available for download at http://www-igm.univ-mlv.fr/~sastre/publications/sastre06b.zip.

Sastre, J. M. (2009). Efficient parsing using filtered-popping recursive transition networks. In Maneth, S., editor, *Implementation and Application of Automata*, volume 5642 of *Lecture Notes in Computer Science*, pages 241–244. Springer-Verlag.

Sastre, J. M. and Forcada, M. L. (2007). Efficient parsing using recursive transition networks with output. In Vetulani, Z., editor, *Proceedings of the 3rd Language & Technology Conference (LTC'07)*, pages 280–284, Poznań, Poland. Wydawnictwo Poznańskie Sp. z o.o.

Sastre, J. M. and Forcada, M. L. (2009). Efficient parsing using recursive transition networks with output. In Vetulani, Z. and Uszkoreit, H., editors, *Human Language Technology. Challenges of the Information Society*,

volume 5603 of *Lecture Notes in Artificial Intelligence*, pages 192–204. Springer-Verlag. Extended version.

Sastre, J. M., Sastre, J., and García, J. (2009). Boosting a chatterbot understanding with a weighted filtered-popping network parser. In Vetulani, Z., editor, *Proceedings of the 4th Language & Technology Conference (LTC'09)*, pages 74–78, Poznań, Poland. Wydawnictwo Poznańskie Sp. z o.o.

Savary, A. (2008). Computational inflection of multi-word units. a contrastive study of lexical approaches. *Linguistic Issues in Language Technology*, 1(2):1—-53.

Savary, A. (2009). Multiflex: A multilingual finite-state tool for multi-word units. In Maneth, S., editor, *Implementation and Application of Automata*, volume 5642 of *Lecture Notes in Computer Science*, pages 241–244. Springer-Verlag.

Scacchi, W. (2010). The future of research in free/open source software development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 315–320, New York, NY, USA. ACM.

Schabes, Y. (1992). Stochastic tree-adjoining grammars. In *Proceedings of the workshop on Speech and Natural Language*, HLT '91, pages 140–145, Stroudsburg, PA, USA. Association for Computational Linguistics.

Schabes, Y. and Joshi, A. K. (1988). An Earley-type parsing algorithm for tree adjoining grammars. In *Proceedings of the 26th annual meeting on Association for Computational Linguistics*, pages 258–269, Morristown, NJ, USA. Association for Computational Linguistics.

Schmid, H. (1994). Probabilistic part-of-speech tagging using decision trees. In *Proceedings of the International Conference on New Methods in Language Processing*, pages 44–49.

Schützenberger, M. P. (1963). On context-free languages and push-down automata. *Information and Control*, 6(3):246–264.

Schützenberger, M. P. (1977). Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, 4(1):47–57.

Seidel, R. and Aragon, C. R. (1996). Randomized search trees. *Algorithmica*, 16:464–497.

Shieber, S. M. (1985). Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *Proceedings of the 23rd annual meeting on Association for Computational Linguistics*, pages 145–152, Morristown, NJ, USA. Association for Computational Linguistics.

Shiers, J. (2007). The Worldwide LHC Computing Grid (worldwide LCG). *Computer Physics Communications*, 177(1–2):219–223.

Silberztein, M. D. (1993). *Dictionnaires électroniques et analyse automatique de textes. Le système INTEX*. Masson, Paris.

Silberztein, M. D. (1994). INTEX: a corpus processing system. In *Proceedings of the 15th conference on Computational linguistics*, pages 579–583, Morristown, NJ, USA. Association for Computational Linguistics.

Silberztein, M. D. (1998). INTEX: An integrated FST toolbox. In Wood, D. and Yu, S., editors, *Automata Implementation*, volume 1436 of *Lecture Notes in Computer Science*, pages 185–197. Springer Berlin / Heidelberg.

Silberztein, M. D. (2003a). Finite-state description of the French determiner system. *Journal of French Language Studies*, 13:221–246.

Silberztein, M. D. (2003b). *NooJ manual*.

Silberztein, M. D. (2004). *INTEX manual*.

Silberztein, M. D. (2005a). NooJ: a linguistic annotation system for corpus processing. In *Proceedings of HLT/EMNLP 2005 Interactive Demonstrations*, pages 10–11, Vancouver, British Columbia, Canada. Association for Computational Linguistics.

Silberztein, M. D. (2005b). NooJ's dictionaries. In Vetulani, Z., editor, *Proceedings of the 2nd Language & Technology Conference (LTC'05)*, pages 291–295, Poznań, Poland. Wydawnictwo Poznańskie Sp. z o.o.

Silberztein, M. D. (2007). An alternative approach to tagging. In Kedad, Z., Lammari, N., Métais, E., Meziane, F., and Rezgui, Y., editors, *Natural Language Processing and Information Systems*, volume 4592 of *Lecture Notes in Computer Science*, pages 1–11. Springer Berlin / Heidelberg.

Sipser, M. (2006). *Introduction to the Theory of Computation.* Thomson Course Technology, Cambridge, 2nd edition.

Sleator, D. D. and Tarjan, R. E. (1985). Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686.

Spuler, D. (1993). The optimal binary search tree for Andersson's search algorithm. *Acta Informatica*, 30(5):405–407.

Spuler, D. A. (1992). The best algorithm for searching a binary search tree. Technical Report 1992-3, Computer Science Dept., James Cook University.

Stolcke, A. (1995). An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21:165–201.

Stroustrup, B. (2000). *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition.

Sun, L., Korhonen, A., and Krymolowski, Y. (2008). Automatic classification of English verbs using rich syntactic features. In *Proceedings of the Third International Joint Conference on Natural Language Processing*, volume 2, pages 769–774, Hyderabad, India.

Swift, J. (1726). *Travels into several remote nations of the world, in four parts. By Lemuel Gulliver, first a surgeon, and then a captain of several ships.* Benjamin Motte, first edition.

The Unicode Consortium (2007). *The Unicode Standard: Version 5.0.0.* Addison-Wesley Developers Press, Boston, MA, USA.

Tomita, M. (1987). An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1-2):31–46.

Traboulsi, H. N. (2009). Arabic named entity extraction: A local grammar-based approach. In *IMCSIT*, pages 139–143.

Triplett, J., McKenney, P. E., and Walpole, J. (2010). Scalable concurrent hash tables via relativistic programming. *ACM SIGOPS Operating Systems Review*, 44(3):102–109.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265.

van de Snepscheut, J. L. A. (1985). *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag. PhD thesis, Eindhoven University of Technology.

van Noord, G. (1997). FSA utilities: A toolbox to manipulate finite-state automata. In Raymond, D., Wood, D., and Yu, S., editors, *Automata Implementation*, volume 1260 of *Lecture Notes in Computer Science*, pages 87–108. Springer Berlin / Heidelberg.

van Noord, G. (2000). Treatment of epsilon moves in subset construction. *Computational Linguistics*, 26(1):61–76.

van Noord, G. and Gerdemann, D. (2001). Finite state transducers with predicates and identities. *Grammars*, 4(3):263–286.

Venkova, T. (2000). A local grammar disambiguator of compound conjunctions as a pre-processor for deep analysers. In Hinrichs, E., Meurers, D., and Wintner, S., editors, *Proceedings of ESSLLI-2000 Workshop on Linguistic Theory and Grammar Implementation*, pages 239–254, Birmingham, U.K.

Vetulani, Z. and Uszkoreit, H., editors (2009). *Human Language Technology. Challenges of the Information Society. Third Language and Technology Conference, LTC 2007, Poznań, Poland, October 5–7, 2007, revised selected papers*, volume 5603 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag.

Vidal, E., Thollard, F., de la Higuera, C., Casacuberta, F., and Carrasco, R. C. (2005a). Probabilistic finite-state machines-part I. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(7):1013–1025.

Vidal, E., Thollard, F., de la Higuera, C., Casacuberta, F., and Carrasco, R. C. (2005b). Probabilistic finite-state machines-part II. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(7):1026–1039.

Vijay-Shanker, K. (1992). Using descriptions of trees in a tree adjoining grammar. *Computational Linguistics*, 18(4):481–517.

Vijay-Shanker, K. and Joshi, A. K. (1988). Feature structures based tree adjoining grammars. In *Proceedings of the 12th conference on Computational linguistics*, volume 2 of *COLING '88*, pages 714–719, Stroudsburg, PA, USA. Association for Computational Linguistics.

von Krogh, G. and Spaeth, S. (2007). The open source software phenomenon: Characteristics that promote research. *The Journal of Strategic Information Systems*, 16(3):236–253.

von Krogh, G. and von Hippel, E. (2006). The promise of research on open source software. *Management Science*, 52:975–983.

Voyatzi, S. (2006). *Description morpho-syntaxique et sémantique des adverbes figés de phrase en vue d'un système d'analyse automatique des textes grecs*. PhD thesis, Université de Marne-la-Vallée.

Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173.

Wallace, R. (2004). The elements of AIML style. ALICE AI Foundation.

Wein, R. (2005). Efficient implementation of red-black trees with split and catenation operation. Technical report, Tel-Aviv University.

Woods, W. A. (1969). Augmented transition networks for natural language analysis. Technical Report CS-1, Harvard Computation Laboratory.

Woods, W. A. (1970). Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606.

Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208.

Zadeh, L. A. (2009). Toward human level machine intelligence-Is it achievable? The need for a paradigm shift. *International Journal of Advanced Intelligence*, 1(1):1–26.

Index