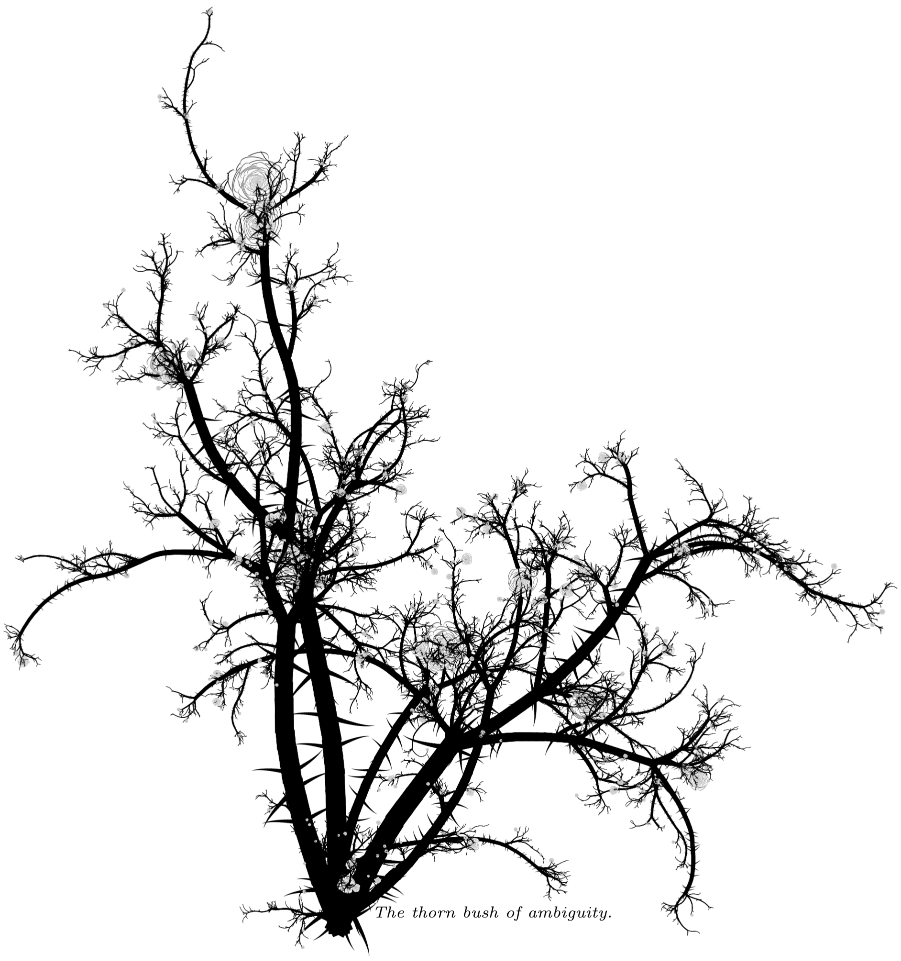
Approximating Context-Free Grammars for Parsing and Verification


The thorn bush of ambiguity.

Sylvain Schmitz

Laboratoire I3S, Université de Nice - Sophia Antipolis & CNRS, France

Ph.D. Thesis

Université de Nice - Sophia Antipolis, École Doctorale « Sciences et Technologies de l'Information et de la Communication »

Informatique	**Subject**
Jacques Farré, Université de Nice - Sophia Antipolis	**Promotor**

Referees

Eberhard Bertsch, Ruhr-Universität Bochum

Pierre Boullier, INRIA Rocquencourt

Defense

September 24th, 2007	**Date**
Sophia Antipolis, France	**Place**

Jury

Pierre Boullier, INRIA Rocquencourt

Jacques Farré, Université de Nice - Sophia Antipolis

José Fortes Gálvez, Universidad de Las Palmas de Gran Canaria

Olivier Lecarme, Université de Nice - Sophia Antipolis

Igor Litovsky, Université de Nice - Sophia Antipolis

Mark-Jan Nederhof, University of St Andrews

Géraud Sénizergues, Université de Bordeaux I

Approximating Context-Free Grammars for Parsing and Verification

Abstract

Programming language developers are blessed with the availability of efficient, powerful tools for parser generation. But even with automated help, the implementation of a parser remains often overly complex.

Although programs should convey an unambiguous meaning, the parsers produced for their syntax, specified by a context-free grammar, are most often nondeterministic, and even ambiguous. Facing the limitations of traditional deterministic parsers, developers have embraced general parsing techniques, capable of exploring every nondeterministic choice, but offering no unambiguity warranty. The real challenge in parser development lies then in the proper identification and treatment of ambiguities—but these issues are undecidable.

The grammar approximation technique discussed in the thesis is applied to nondeterminism and ambiguity issues in two different ways. The first application is the generation of noncanonical parsers, less prone to nondeterminism, mostly thanks to their ability to exploit an unbounded context-free language as right context to guide their decision. Such parsers enforce the unambiguity of the grammar, and furthermore warrant a linear time parsing complexity. The second application is ambiguity detection in itself, with the insurance that a grammar reported as unambiguous is actually so, whatever level of approximation we might choose.

Key Words

Ambiguity, context-free grammar, noncanonical parser, position automaton, grammar engineering, verification of infinite systems

ACM Categories

D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Syntax ; D.3.4 [*Programming Languages*]: Processors—Parsing ; F.1.1 [*Computation by Abstract Devices*]: Models of Computation; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.4.2 [*Mathematical Logic and Formal Languages*]: Grammars and Other Rewriting Systems

Approximation de grammaires algébriques pour l'analyse syntaxique et la vérification

Résumé

La thèse s'intéresse au problème de l'analyse syntaxique pour les langages de programmation. Si ce sujet a déjà été traité à maintes reprises, et bien que des outils performants pour la génération d'analyseurs syntaxiques existent et soient largement employés, l'implémentation de la partie frontale d'un compilateur reste encore extrêmement complexe.

Ainsi, si le texte d'un programme informatique se doit de n'avoir qu'une seule interprétation possible, l'analyse des langages de programmation, fondée sur une grammaire algébrique, est, pour sa part, le plus souvent non déterministe, voire ambiguë. Confrontés aux insuffisances des analyseurs déterministes traditionnels, les développeurs de parties frontales se sont tournés massivement vers des techniques d'analyse générale, à même d'explorer des choix non déterministes, mais aussi au prix de la garantie d'avoir bien traité toutes les ambiguïtés grammaticales. Une difficulté majeure dans l'implémentation d'un compilateur réside alors dans l'identification (non décidable en général) et le traitement de ces ambiguïtés.

Les techniques décrites dans la thèse s'articulent autour d'approximations des grammaires à deux fins. L'une est la génération d'analyseurs syntaxiques non canoniques, qui sont moins sensibles aux difficultés grammaticales, en particulier parce qu'ils peuvent exploiter un langage algébrique non fini en guise de contexte droit pour résoudre un choix non déterministe. Ces analyseurs rétablissent la garantie de non ambiguïté de la grammaire, et en sus assurent un traitement en temps linéaire du texte à analyser. L'autre est la détection d'ambiguïté en tant que telle, avec l'assurance qu'une grammaire acceptée est bien non ambiguë quel que soit le degré d'approximation employé.

Mots clefs

Ambiguïté, grammaire algébrique, analyseur syntaxique non canonique, automate de positions, ingénierie des grammaires, vérification de systèmes infinis

Foreword

This thesis compiles and corrects the results published in several articles (Schmitz, 2006; Fortes Gálvez et al., 2006; Schmitz, 2007a,b) written over the last four years at the Laboratoire d'Informatique Signaux et Systèmes de Sophia Antipolis (I3S). I am indebted for the excellent working environment I have been offered during these years, and further grateful for the advice and encouragements provided by countless colleagues, friends, and family members.

I am indebted to many friends and members of the laboratory, notably to Ana Almeida Matos, Jan Cederquist, Philippe Lahire, Igor Litovsky, Bruno Martin and Sébastien Verel, and to the members of my research team, Carine Fédèle, Michel Gautero, Vincent Granet, Franck Guingne, Olivier Lecarme, and Lionel Nicolas, who have been extremely helpful and supportive beyond any possible repay. I especially thank Igor Litovsky and Olivier Lecarme for accepting to examine my work and attend to my defense. I am immensely thankful to Jacques Farré for his insightful supervision, excellent guidance, and extreme kindness, that greatly exceed everything I could have wished for.

I thank José Fortes Gálvez for his fruitful collaboration, his warm welcome on the occasion of my visit in Las Palmas, and his acceptance to work on my thesis jury. I gratefully acknowledge several insightful email discussions with Dick Grune, who taught me more on the literature on parsing technologies than anyone could wish for, and who first suggested that studying the construction of noncanonical parsers from the NFA to DFA point of view could bring interesting results. I am also thankful to Terence Parr for our discussions on LL(*), to Bas Basten for sharing his work on the comparison of ambiguity checking algorithms, and to Claus Brabrand and Anders Møller for our discussions on ambiguity detection and for granting me access to their tool.

I am obliged to the many anonymous referees of my articles, whose remarks helped considerably improving my work, and I am much obliged to Eberhard Bertsch and Pierre Boullier for their reviews of the present thesis, and the numerous corrections they offered. I am honored by their interest in my work, as well as by the interest shown by the remaining members of my jury, Mark-Jan Nederhof and Géraud Sénizergues.

Finally, I thank all my friends and my family for their support, and my beloved Adeline for bearing with me through these years.

Sophia Antipolis, October 16, 2007

Contents

Introduction 1

"If you want a red rose", said the Tree, "you must build it out of music by moonlight, and stain it with your own heart's-blood. You must sing to me with your breast against a thorn."
Oscar Wilde, *The Nightingale and the Rose*

1.1 Motivation

Grammars are pervasive in software development as syntax specifications for text inputs. From a context-free grammar, software components like parsers, program translators, pretty-printers and so forth can be generated instead of hand-coded, greatly improving reliability and readability. *Grammarware*, as coined by Klint et al. (2005) to comprise grammars and grammar-dependent software, is blessed with a rich body of literature, and by the wide availability of automated tools.

Regardless of their services over the years, classical parser generators in the line of YACC (Johnson, 1975) have seen their supremacy contested by several authors, who pointed out their inadequacy in addressing the modern challenges in grammar engineering (Parr and Quong, 1996; van den Brand et al., 1998; Aycock, 2001; Blasband, 2001). Accordingly, different parsing techniques were investigated, and Tomita's Generalized LR (GLR) algorithm, originally aimed towards natural language processing, found many proponents. With the broad availability of general parser generators, like SDF (Heering et al., 1989), Elkhound (McPeak and Necula, 2004) or GNU Bison (Donnely and Stallman, 2006) to name a few, it might seem that the struggles with the dreaded report

```
grammar.y: conflicts: 223 shift/reduce, 35 reduce/reduce
```

are now over. General parsers simulate the various nondeterministic choices in parallel with good performance, and return all the legitimate parses for the input.

What the above naive account overlooks is that all the legitimate parses according to the grammar might not always be correct in the intended language. With programming languages in particular, a program is expected to have a unique interpretation, and thus a single parse should be returned. Nevertheless, the grammar developed to describe the language is often ambiguous: ambiguous grammars are more concise and readable (Aho et al., 1975). The language definition should then include some *disambiguation* rules to decide which parse to choose (Klint and Visser, 1994). But we cannot always decide where such rules are needed (Cantor, 1962; Chomsky and Schützenberger, 1963; Floyd, 1962a).

The difficulties raised by ambiguities were noted by McPeak and Necula (2004) and they devote a few lines to the problem:

> [There is] the risk that ambiguities would end up being more difficult to debug than conflicts. After all, at least conflicts are decidable. Would the lure of quick prototyping lead us into a thorn bush without a decidable boundary?
>
> Fortunately, ambiguities have not been a problem. By reporting conflicts, Elkhound provides hints as to where ambiguities may lie, which is useful during initial grammar development. But as the grammar matures, we find ambiguities by parsing inputs, and understand them by printing the parse forest (an Elkhound option).

Indeed, many conflicts are caused by ambiguities, and with a careful study of these numerous conflict cases and with extensive testing, the experienced user should be able to determine whether disambiguation is required or not. But as long as there remains a single conflict, there is no formal insurance that all ambiguities were caught. Quoting Dijkstra (1972),

> Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness.

Facing a thorn bush of ambiguity, we advocate wearing gloves. They come in two shapes in this thesis.

1. The first is to generate *noncanonical* parsers, with less conflicts, thus making the knotty process of eliminating conflicts less of a burden. One can more realistically attempt to remove all the conflicts—and thus to obtain the insurance that no ambiguity remains. Noncanonical parsers rely on the remaining text to help choose between parsing

actions; they are able to perform parsing actions in this right context, and pull back in order to use the information gathered there to help with resolving earlier conflicts.

2. The second is to detect ambiguities as such. Since the problem is undecidable, some approximations, in the form of false positives or false negatives, are unavoidable. Our method is conservative in that it cannot return false negatives, and it is therefore safe to employ a grammar it reports as unambiguous.

1.2 Overview

After reviewing some essential background on parsing techniques (Chapter 2) we describe four practical cases occurring in the syntax of Java (Gosling et al., 1996), C++ (ISO, 1998) and Standard ML (Milner et al., 1997) that illustrate the limitations of traditional LALR(1) parsing in Section 3.1. Different approaches to handling these issues are discussed in the remaining of Chapter 3, providing an overview of the "recent" developments in parsing techniques. The three technical chapters of this thesis, which we introduce in greater extent in the following paragraphs, are dedicated to the approximation of grammars (Chapter 4), parser generation (Chapter 5) and ambiguity detection (Chapter 6). We conclude with some final remarks and future work in Chapter 7, and we gather some usual definitions and notational conventions in Appendix A.

Position Automata Both the generation of a noncanonical parser and the detection of ambiguities can be seen as the result of a static analysis on the grammar. Guided by the intuition that most parsing techniques operate a form of left-to-right depth-first walk in the set of all the derivation trees of the grammar, we define the *position graph* of a grammar as the set of all these walks, and a *position automaton* as a quotient of a position graph by an equivalence relation between positions in derivation trees (Chapter 4). We obtain various levels of approximation when we choose refined or coarser equivalences, and thus position automata provide a fairly general approximation framework, in which several classical parser generation techniques can be expressed. In particular, the states of a position automaton generalize the notion of *items* usually employed to denote a position in a grammar.

Besides the generation of noncanonical parsers and the detection of ambiguity, we apply position automata to two small problems:

1. the recognition of derivation trees by means of a finite state automaton (Section 4.3), a problem inspired by a similar issue in streaming XML processing (Segoufin and Vianu, 2002). This issue fits the possibilities

of position automata rather well, and as a result of our initial investigations, we managed to give a characterization, in terms of position equivalences, of which context-free grammars can have their derivation trees recognized by a finite state machine;

2. the generation of canonical bottom-up parsers (Section 5.1). In this domain we simply show how to generate parsers for LR(k) grammars (Knuth, 1965) and strict deterministic grammars (Harrison and Havel, 1973).

Noncanonical Parsing We present two different noncanonical parsing techniques, namely Noncanonical LALR(1) parsing in Section 5.2 and Shift-Resolve parsing in Section 5.3.

Noncanonical LALR(1) With NLALR(1) parsing, we investigate how a noncanonical parser can be obtained from a canonical bottom-up one. The generated parsers generalize the single other practical noncanonical parsing method, namely Noncanonical SLR(1) parsing (Tai, 1979).

Shift-Resolve Parsing The main weakness of most noncanonical parsing techniques is their bound on the length of a reduced lookahead window. With Shift-Resolve parsing, developed jointly with José Fortes Gálvez and Jacques Farré, we exploit a position graph directly, and generate parsers where the lookahead lengths are independent and computed as needed for each parsing action.

Ambiguity Detection In Chapter 6, we present a conservative algorithm for the detection of ambiguities in context-free grammars. The algorithm is designed to work on any position automaton, allowing for various degrees of precision in the ambiguity report, depending on the chosen position equivalence. We compare our procedure formally with other means to detect ambiguities, namely bounded generation of sentences (Gorn, 1963; Cheung and Uzgalis, 1995; Schröer, 2001; Jampana, 2005) and LR-Regular testing (Čulik and Cohen, 1973; Heilbrunner, 1983), and we report on the experimental results we obtained with an implementation of our algorithm as an option in GNU Bison.

General Background 2

The chapter first provides a broad and somewhat historical introduction to the main topics of the thesis, namely context-free grammars and deterministic parsers (Section 2.1). We focus then on the specific needs of parsers for programming languages, and present the classical parsing techniques designed during the *Golden Age* of parsing research (Section 2.2). A reader familiar with these ideas might prefer to fast-forward to the following Chapter 3, where we examine the shortcomings of these techniques, the more recent developments they triggered, and the remaining issues that motivated our research.

2.1 Topics

This first section attempts to provide an intuition of the main topics of the thesis: context-free grammars and deterministic parsers. Though the research we report is mostly motivated by the syntax of programming languages, the notions we use were introduced in the early attempts at understanding the nature of natural languages. We believe these notions to be easier to grasp when presented in their original perspective, and we will illustrate the basic concepts with examples in plain English.

This quick overview is by no means an exhaustive presentation of the subjects in formal languages. The interested reader is directed to the reference works of Harrison (1978), Hopcroft and Ullman (1979) or Sippu and Soisalon-Soininen (1988) for a more thorough treatment, while Appendix A collects the formal definitions of the terms used here.

2.1.1 Formal Syntax

Our view of language is *syntactic*. If we consider the two sequences

The child ate a tomato .
*A tomato the child ate .

we will identify the first as a correct English sentence, and the second, starred sequence as ungrammatical—and probably told by Yoda. The meaning, or *semantics*, of a sentence is irrelevant. For instance, the sentence

A tomato ate the child .

is correct English, though wholly absurd since tomatoes are not carnivorous—unless you consider 1978's cult movie *Attack of the Killer Tomatoes!* to be a realistic account of events.[1] Likewise, the recognition of the word "tomato" as a noun from its sequence of letters "t o m a t o" is a *lexical* issue—which might be problematic in presence of homographs, like "a row" and "to row".

How many syntactically correct English sentences do exist? Intuitively, there is an infinite number of such sentences. A *formal language* (Definition A.3 on page 164) is a set of correct sentences, all of finite length, and all built using the same finite *vocabulary* (Definition A.2 on page 164). Many sentences in this infinite set are related, as for instance

The child ate a red tomato .
The child ate a round red tomato .
The child ate a big round red tomato .
...

This last sentence could grow indefinitely if we added more and more adjectives qualifying our "tomato". What we just observed is a *rule* for rewriting correct sentences into new correct sentences. A *rewriting system* (Definition A.4 on page 164) is a finite set of such rules. A rewriting system can act as a grammar for a language if the exact set of all the correct sentences can be obtained by applying the rewrite rules.

As he investigated which models could account for these rules, Chomsky (1956) observed that sentences were hierarchically structured in *phrases*. For instance, "a tomato" is a noun phrase (*NP*) and, when prefixed with the verb "ate", it forms a verb phrase (*VP*). In turn, together with the noun phrase "The child", they form a complete sentence (*S*). Figure 2.1 on the next page, which exhibits this phrase structure, is also called a *derivation tree*. Instead of working solely on the sentence level, the rules in *phrase structure grammars* (Definition A.5 on page 164) are also able to act on the intermediate phrase structures, also called *nonterminals* by opposition with the *terminals* on the sentence level. We denote the rule allowing the

[1]The fact that in some contexts both the ungrammatical and the absurd sequences could be considered correct and meaningful is a hint at the sheer complexity of natural languages.

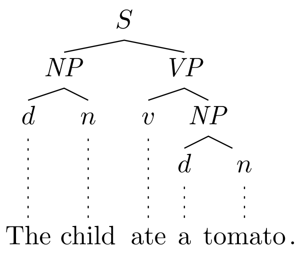


Figure 2.1: Phrase structure of the sentence “The child ate a tomato .”.

---

sentence to be developed into the sequence of a noun phrase and a verb phrase by

$$S \to NP\ VP \tag{2.1}$$

where the *left part* and *right part* of rule (2.1) are S and $NP\ VP$ respectively.

2.1.1.1 Context-Free Grammars

Chomsky (1959) further formalized how the rewriting rules in phrase structure grammars could be restricted, resulting in the famous *Chomsky hierarchy*. One of the possible restrictions stipulates that a rewriting rule should have a single nonterminal on its left part and any sequence of terminal or nonterminal symbols on its right part. Clearly, rule (2.1) complies with this requirement. Grammars where all the rules do comply are called *context-free*, or *type 2* (CFG, Definition A.7 on page 165).

A context-free grammar generating our sentence “The child ate a tomato .” could be the grammar with rules[2]

$$S \to NP\ VP,\ NP \to d\, n,\ VP \to v\, NP. \tag{$\mathcal{G}_1$}$$

In this grammar d, n and v are the terminal symbols standing for determinants, nouns and verbs, S, NP and VP the nonterminal symbols, and among them S is the *axiom*, from which all sentences are derived. Grammar $\mathcal{G}_1$ is rather limited since the only sentences it can generate are always of form $d\,n\,v\,d\,n$. If we wanted to be able to generate the sentence “The child ate a red tomato .”, we would need for instance the grammar with rules

$$S \to NP\ VP,\ NP \to d\, AN,\ AN \to a\, AN \,|\, n,\ VP \to v\, NP, \tag{$\mathcal{G}_2$}$$

where $AN \to a\, AN \,|\, n$ denotes the two rules $AN \to a\, AN$ and $AN \to n$. Note that, with this new grammar, we can further generate the sentences with as

[2]Names in capital letters denote nonterminal symbols. See Section A.1.4 on page 165 for our notations.

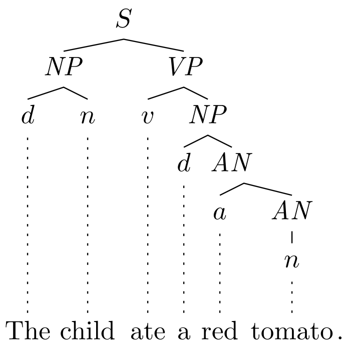


Figure 2.2: Derivation tree for the sentence “The child ate a red tomato .” according to $\mathcal{G}_2$.

many adjectives qualifying our “tomato” as we want by applying the rule $AN \rightarrow a\ AN$ again and again before choosing $AN \rightarrow n$. Figure 2.2 presents a derivation tree using $\mathcal{G}_2$.

Context-free grammars are not so good for the description of natural languages (Shieber, 1985), but they are well studied and employed in many fields. In particular, the syntax of programming languages is almost always described by context-free grammars. This popularity can be traced back to the original formalization of the syntax and semantics of ALGOL 60, when Backus (1959) and Naur (1960) first used a notation to describe its syntax, notation since known as the *Backus-Naur Form* (BNF). Ginsburg and Rice (1962) proved later that context-free grammars and grammars in BNF were equivalent.

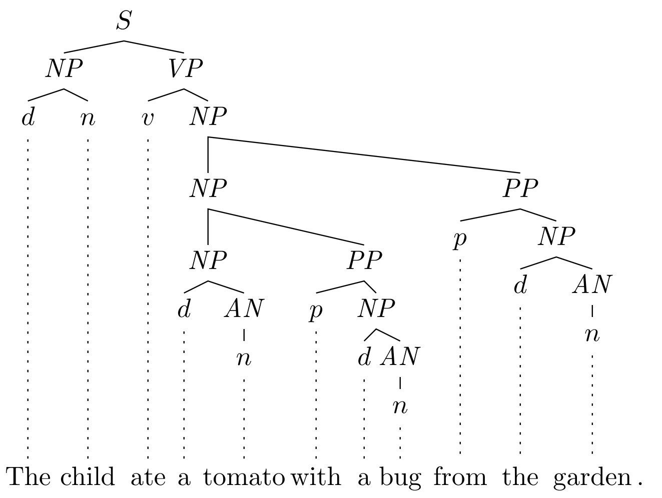
2.1.1.2 Ambiguity

Our grammar $\mathcal{G}_2$ is still rather poor. We might want to qualify our “tomato” further, as in “The child ate a tomato with a bug .”. A grammar allowing this kind of constructions could have the rules

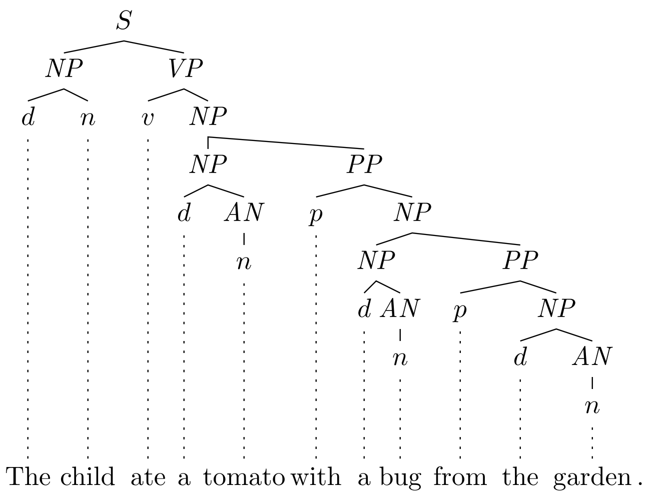
$$S \rightarrow NP\ VP,\ NP \rightarrow d AN \mid NP PP,\ AN \rightarrow a AN \mid n,\ VP \rightarrow v NP,\ PP \rightarrow p NP. \quad (\mathcal{G}_3)$$

Note that the new preposition phrases qualifying a noun phrase can also be added indefinitely using the rule $NP \rightarrow NP\ PP$, and indeed we can imagine the sentence “The child ate a tomato with a bug from the garden .”. But here we just introduced another phenomenon: is the “tomato” “from the garden”, or is the “bug” “from the garden”? We have two different syntactic interpretations for this sentence—there are two different derivation trees, shown in Figures 2.3a and 2.3b.[3] The sentence is *ambiguous*.

[3]The sentence has yet another syntactic interpretation, where “the child” is eating *in company of* “a bug from the garden”. Our grammar $\mathcal{G}_3$ could exhibit this last interpreta-



(a) Derivation tree where the "tomato" comes "from the garden".



(b) Derivation tree where the "bug" comes "from the garden".

Figure 2.3: Derivation trees according to $\mathcal{G}_3$.

Syntactic ambiguity often occurs in natural languages, and its presence is expected. But it can also occur in the syntax of programming languages, which is quite worrying. Testing a context-free grammar for ambiguity is an *undecidable* problem, as proven by Cantor (1962), Chomsky and Schützenberger (1963), and Floyd (1962a). However, approximative answers can be provided to the question. An important part of this thesis is devoted to the detection of syntactic ambiguities in context-free grammars.

tion if we added the rule $VP \rightarrow VP\ PP$.

2.1.2 Parsing

We now look at languages from a different perspective. Instead of developing a grammar describing how the English language is structured, we can set up a device that recognizes correct English sentences and rejects incorrect ones. Just as we could define some rules to generate a correct sentence from another correct one, we can devise rules recognizing correct sentences. A formal language can be described just as well by such a *recognizer* (Definition A.4 on page 164). A *parser* is a recognizer that also outputs the sentence structure it uncovered. Thus the input of a parser for English could be $d\ n\ v\ d\ n$, corresponding to “The child ate a tomato .”, and its output the tree of Figure 2.1 on page 7—in which case we would call it a *parse tree*. A parser has two missions: insuring the sentence correctness, and exhibiting its structure for further analysis.

2.1.2.1 Pushdown Automata

A recognizer for a language somehow works by reversing the generative rules exercised in the grammar. There are thus clear correspondences between each restricted class of phrase structure grammars and their recognizers. Chomsky (1962) proved that the recognizers for context-free languages were *pushdown automata* (PDA, Definition A.10 on page 166).

Indeed, parsers for the languages generated by context-free grammars commonly use a *pushdown stack* holding the roots of some partially computed trees. Two main strategies are possible, namely

bottom-up parsers use the stack to remember which subtrees have already been recognized; they work their way from the terminal tree leaves and try to *reduce* up to the tree root, while

top-down parsers use the stack to remember which subtrees have yet to be recognized; they work their way from the tree root and try to *predict* a correct derivation matching the tree leaves.

Table 2.1 on the next page presents how a bottom-up parser could recognize the sentence “The child ate a tomato .”. This *shift-reduce parser* (Definition A.12 on page 167) has two types of rules, *shift* to push the next input symbol on top of the stack, and *reduce* to replace the topmost symbols of the stack by a nonterminal producing them. The special symbol \$ denotes the bottom of stack and end of sentence boundaries. Each reduction thus connects tree nodes to their direct father in the parse tree.

The shift-reduce parser we just introduced is *nondeterministic*: as long as we have some input symbol left, we can always choose to apply the shift rule, and whenever a complete grammar rule right part is on top of the stack, we can choose to apply a reduction. As explained in Table 2.1, there are times

Table 2.1: Recognition of "The child ate a tomato ." by a shift-reduce parser.

parsing stack	input	rules
$\$$	$d\,n\,v\,d\,n\,\$$	shift
There is no complete grammar rule right part in the stack, thus the shift-reduce parser should shift in order to complete its stack contents:		
$\$\,d$	$n\,v\,d\,n\,\$$	shift
$\$\,d\,n$	$v\,d\,n\,\$$	reduce $NP{\rightarrow}d\,n$
We now have a complete right part for rule $NP{\rightarrow}d\,n$. We have just recognized a noun phrase; we can replace $d\,n$ by NP in the stack. Note however that we could have tried to shift v as well, but then the parse would not have been successful.		
$\$\,NP$	$v\,d\,n\,\$$	shift
$\$\,NP\,v$	$d\,n\,\$$	shift
$\$\,NP\,v\,d$	$n\,\$$	shift
$\$\,NP\,v\,d\,n$	$\$$	reduce $NP{\rightarrow}d\,n$
Once more, we have the constituents of a noun phrase on the top of the stack.		
$\$\,NP\,v\,NP$	$\$$	reduce $VP{\rightarrow}v\,NP$
$\$\,NP\;VP$	$\$$	reduce $S{\rightarrow}NP\;VP$
$\$\,S$	$\$$	
We have successfully recognized the entire sentence according to $\mathcal{G}_1$.		

when the parser has to choose between two different actions. In the case of an ambiguous sentence, two different correct parses of the whole sentence are possible, and therefore the parser is bound to be nondeterministic.

The common description of a nondeterministic machine is that it is advised by some magical oracle telling which way to choose. A more pragmatic view is that an external mechanism allows the machine to test all possibilities, and to reject the choices that end up being wrong after all. Two kinds of external mechanisms exist:

backtrack we choose one possibility and keep working until it fails—in which case we undo the job performed since the last choice and try a different possibility at this point—or definitely succeeds, and

parallelism we keep all the possibilities in parallel and discard the failing ones on the fly.

Instead of resorting to a costly pseudo-nondeterminism mechanism, we can also try to generate a more advanced form of the shift-reduce parser, where the parser would be a bit more demanding about choosing a shift or a reduction. Chapter 5 on page 81 is dedicated to the construction of advanced deterministic parsers.

2.1.2.2 The Determinism Hypothesis

The few examples seen so far might give the impression that natural languages cannot be parsed deterministically, and that human understanding would exert some form of pseudo-nondeterminism. A radically different view was given by Marcus (1980) as he formulated the *determinism hypothesis*. The claim of Marcus is that humans "normally" parse deterministically from left to right, never altering any previous decision—which rules out backtracking—and always making a single choice at each point—which rules out parallelism. Marcus describes in detail a parser for English to sustain his theory, Parsifal.

This theory is not incompatible with the existence of syntactic ambiguity: deterministic parsing is conditioned by the availability of precise lexical and semantic information, so for instance we would know from the context whether "the bug" was really "from the garden". And if that fails, then the sentence would also have to be reparsed by a human. Marcus advances the same argument for unambiguous sentences for which Parsifal is nondeterministic: a human being would also need some kind of backtrack; these are called *garden path* sentences, like for instance

Have the children given a tomato by their parents.

The sentence usually needs a conscious reparsing after the reader led on a garden path "Have the children given a tomato . . . ?" reads "by", and has to reinterpret the whole sentence.

The determinism hypothesis has been influential, more on the general concepts than on the details of Parsifal (see for instance (Pritchett, 1992) for a criticism of the latter). Actual natural language processors try to exploit as much lexical and semantic information as possible in order to decrease the huge numbers of possible parses for ambiguous sentences.

Several elements make the determinism hypothesis interesting to us. First, Parsifal was later shown to behave like a *noncanonical* parser and to accept context-free languages (Nozohoor-Farshi, 1986, 1987; Leermakers, 1992; Bertsch and Nederhof, 2007), which relates it to the parsing methods advocated in this thesis. Second, we are going to shift our interests to programming languages in the next section. In their case, we are going to formulate a variant of the determinism hypothesis: we argue that programming languages should be unambiguous.

2.2 Scope

Following the invention of context-free grammars and their basic properties as summed up in the previous section, an important research effort was directed towards investigating their adequacy as syntax specifications for programming languages. This effort culminated during the seventies with the large adoption of LALR(1) parser generators, and the supremacy of YACC (*Yet Another Compiler Compiler*, Johnson, 1975) in particular.

Accordingly, we present in this section an overview of the major results of this Golden Age of parsing research; the reference books by Grune and Jacobs (2007) on parsing techniques, and Sippu and Soisalon-Soininen (1990) on parsing theory are warmly recommended to any reader interested by an in-depth treatment of the subject. We first emphasize the importance of parsing techniques in the development of programming languages, and the specific requirements that stem from this particular application (Section 2.2.1). We then present the classical deterministic parsing methods (Section 2.2.2), and illustrate how they were extended to handle ambiguous constructs (Section 2.2.3).

2.2.1 Parsers for Programming Languages

In a wide sense, programming languages encompass all the artificial languages developed in order to communicate with a computer: general programming languages, like C (Kernighan and Ritchie, 1988), Java (Gosling et al., 1996) or Standard ML (Milner et al., 1997), but also data description languages, like LaTeX (Lamport, 1994) or XML (Yergeau et al., 2004), or various other domain specific languages, like SQL (ISO, 2003) for databases. The communication in these languages with computers is usually text-based, hence the need for tools that allow to interpret or translate the texts. A component is mandatory in all these tools in order to exhibit the relevant structure from the text before interpretation or translation can occur; this component is commonly referred to as the *front end*.

2.2.1.1 Front Ends

From the definition of parsing given in Section 2.1.2 on page 10, front ends and parsers are clearly intimately related. Front ends are usually decomposed into three different tasks, summed up in Figure 2.4 on the next page (e.g. (Aho and Ullman, 1972)):

lexical analysis (or scanning or tokenizing) recognizes the most basic constituents of the syntax, like keywords, numbers, operators, or comments, just like we needed to recognize that a “tomato” was a noun in

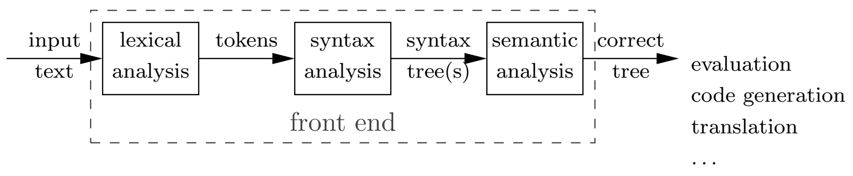


Figure 2.4: The elements of a front end.

English. The result of this task is a sequence of tokens, or sometimes a directed acyclic graph in case of an unresolved lexical ambiguity;

syntax analysis (or parsing) checks the program against the syntax of the language and builds parse trees from the tokens;

semantic analysis (or bookkeeping or symbol table operations) checks that the various identifiers are employed correctly with respect to their names, scopes or types; we could compare this task with checking that plurals are used consistently in an English sentence.

The separation of the front end in three distinct tasks is a result of the impossibility to express some syntactic requirements using context-free grammars only. As noted by Floyd (1962b), the fact that the identifier xxxx—which could be arbitrarily long—in the following ALGOL 60 code snippet has to be declared prior to its use cannot be enforced by a context-free grammar:[4]

begin real xxxx; xxxx := xxxx **end**

The separation is thus mostly motivated by the limitations of context-free grammars in the central parsing step—some performance concerns are also taken into consideration.

Real world programming languages are seldom amenable to a clear separation into the three tasks of the front end. For instance, type and variable names can collide and create ambiguities in C if the lexer does not differentiate them—using the table of symbols—before feeding the parser. Keywords in FORTRAN IV are not reserved and can also be interpreted as identifiers, making the separation impossible. Extensions to context-free grammars spawned from the separation issues: there exist scannerless parsers (Salomon and Cormack, 1989; Visser, 1997), and some table of symbols operations can be integrated in the parsing phase using different variations on context-free grammars, e.g. attribute grammars (Knuth, 1968; Watt, 1980), van Wijngaarden grammars (1975), affix grammars (Koster, 1971),

[4]If identifiers were of bounded length, a context-free grammar enumerating all cases could be written.

facet grammars (Bailes and Chorvat, 1993) or boolean grammars (Okhotin, 2004; Megacz, 2006). Some have dropped context-free grammars completely, in favor of a different, recognitive syntax formalism (Birman and Ullman, 1973; Ford, 2004), for which efficient parsers can be constructed (Ford, 2002; Grimm, 2006).

We consider nonetheless the classical central view of the parser for a context-free grammar, but we will allow some interplay with the other tasks of the front end.

2.2.1.2 Requirements on Parsers

Some specificities of programming languages determine the desirable properties of front ends, and thus imply some properties on the parsers.

Reliability First of all, front ends have to be reliable. They are used in compilers, in processes that will output programs running safety-critical missions. They are blessed with one of the richest bodies of computing literature, resulting in their being in one of the few fields with automated tools generating program source (the front ends themselves) directly from specifications (of the lexer, parser and typing system). Advanced parsers being rather complex programs, the safest practice is to generate them from the grammatical specifications.

Direct generation The current development of programming languages is mostly centered on their semantic aspects. In this view, syntax provides the user interface with the language, and should naturally reflect the semantics. A second tendency is to create many small domain specific languages. These trends require easy to use parsing tools, where some of the burden of the syntax issues is shifted from the developer to the parser generator. In other words, grammatical specifications should not have to be adapted to the parser generator.

Unambiguous Programs (or document descriptions, or databases queries etc.) should have a unique meaning. A lot of software already behaves erratically, it is quite unnecessary to allow a program text to have several interpretations. This is reflected in Figure 2.4 on the facing page by the fact that the front end outputs a single correct tree. Since front ends have to be reliable, we should detect any possibility for the language to be ambiguous. The implication on parsers is simple:

- Either the parser always outputs a single parse tree—the parser itself is unambiguous—, and the overall process is guaranteed to be unambiguous. As mentioned before, an exact ambiguity detection is impossible in general (Cantor, 1962; Chomsky and Schützenberger, 1963; Floyd, 1962a). There is however a simple way to guarantee unambiguity: deterministic parsers are always unambiguous.

- Or, just as some lexical or semantic information was sometimes necessary with Parsifal for the determinism hypothesis to hold, the parser and the two other constituents of the front end might have to communicate in order to gather all the information necessary for the front end to output a single tree. The parser could output several parse trees, provided we have the insurance that subsequent processing will pick at most one correct tree from the lot. This requires an (approximative) ambiguity detection mechanism telling where disambiguation is required. We call such a parser *pseudo deterministic*, by contrast with the pseudo nondeterministic methods mentioned in Section 2.1.2.

Performance Last of all, programs can be huge. The successive inclusions of header files for several libraries in C can result in programs weighting a hundred megabytes after preprocessing. The front end should therefore be reasonably fast, even with today's computing power. Parsers working in time proportional with the size of the input text will be preferred.

In short, ideal parsers for programming languages are,

1. generated directly from a formal grammar,

2. deterministic or pseudo deterministic, and

3. fast, preferably working in linear time.

Parsing is a mature field, and there are numerous techniques meeting more or less these requirements. The confrontation with practical issues (Section 2.2.2) makes however some tradeoffs necessary. Our main purpose in this thesis is to improve parsing methods on this three points basis. Error correction, robustness, modularity, or incremental processing are also highly appreciated in the modern uses of parsers, and are subjects of study by themselves.

2.2.2 Deterministic Parsers

In the classical meaning, deterministic parsers are based on *deterministic pushdown automata* (DPDA, studied by Ginsburg and Greibach, 1966, Definition A.11 on page 167). The literature abounds with techniques trying to construct a deterministic pushdown automaton from a context-free grammar. These parsers are often allowed to have a peek further than the next input symbol, and are thus defined with a parameter k representing the length of their lookahead window. In practice the parameter k is generally set to one.

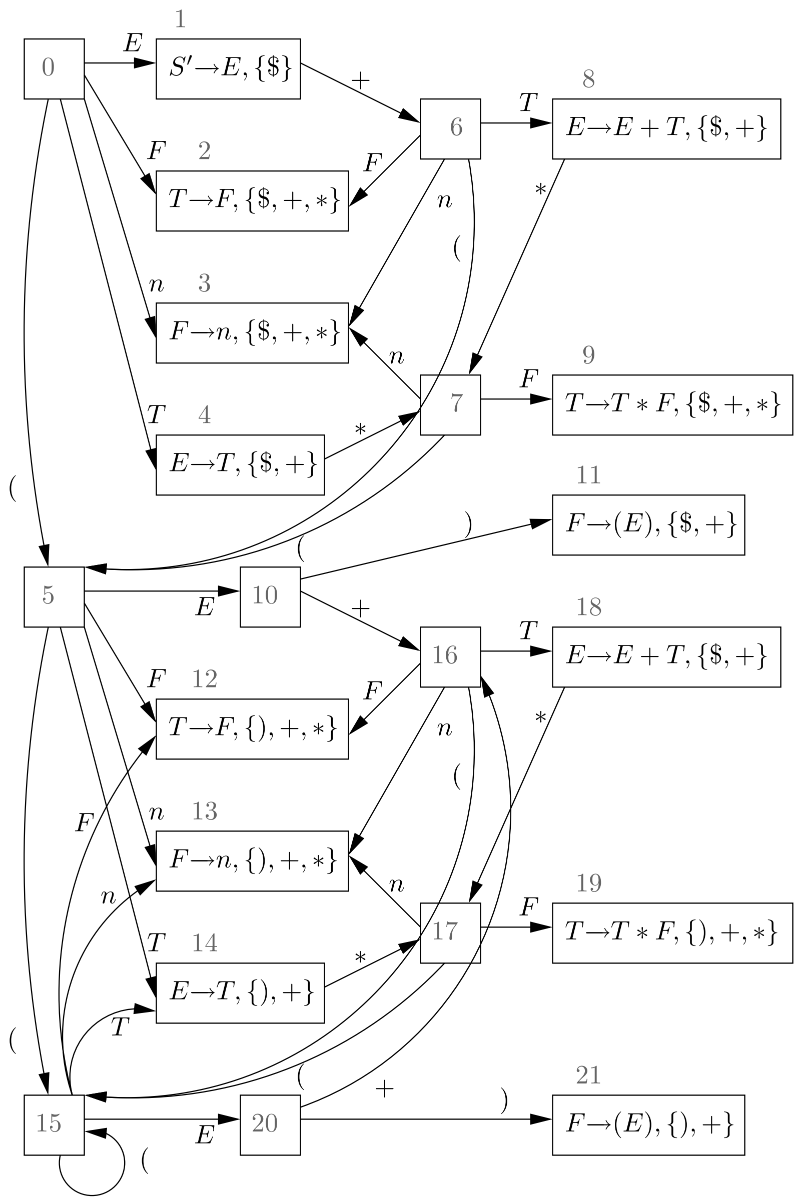

0
1
$S'\rightarrow E, \{\$\}$
2
$T\rightarrow F, \{\$, +, *\}$
3
$F\rightarrow n, \{\$, +, *\}$
4
$E\rightarrow T, \{\$, +\}$
5
6
7
8
$E\rightarrow E+T, \{\$, +\}$
9
$T\rightarrow T*F, \{\$, +, *\}$
10
11
$F\rightarrow (E), \{\$, +\}$
12
$T\rightarrow F, \{), +, *\}$
13
$F\rightarrow n, \{), +, *\}$
14
$E\rightarrow T, \{), +\}$
15
16
17
18
$E\rightarrow E+T, \{\$, +\}$
19
$T\rightarrow T*F, \{), +, *\}$
20
21
$F\rightarrow (E), \{), +\}$

Figure 2.5: LR(1) automaton for grammar $\mathcal{G}_4$.

2.2.2.1 LR(k) Parsers

The most general classical deterministic parsing method is the LR(k) one designed by Knuth (1965, see also Section A.2 on page 168). Its generation algorithm outputs a DPDA-based parser if such a parser exists for the provided grammar. In this sense, the LR(k) construction predates any other deterministic parser construction. We will often consider LR(k) parsers as the best representatives of the classical deterministic parsing techniques.

The key concept behind LR(k) parsing is that the language of all the correct stacks appearing during the operations of a shift-reduce parser is a *regular language*, thus can be processed by the means of a *deterministic finite automaton* (DFA, Definition A.9 on page 166). Depending on the state of the DFA recognizing the stack contents, a decision to shift or to reduce might be taken. A LR(k) parser infers its parsing decisions at each parsing point from the entire left context of the point—through the state of the DFA recognizing the stack contents—, and from the k next symbols of the input. If all this information does not leave any choice open, then the parser is deterministic.

Figure 2.5 on the previous page presents the underlying automaton of the LR(1) parser for the grammar with rules

$$E\to T\,|\,E+T,\ T\to F\,|\,T*F,\ F\to n\,|\,(E). \qquad (\mathcal{G}_4)$$

Grammar $\mathcal{G}_4$ is a grammar for arithmetic expressions. The reduction decisions of the LR(1) parser are displayed in the figure with the correct lookahead symbols for the reduction to occur. For instance $E\to E+T, \{\$, +\}$ in state 8 indicates that a reduction of $E+T$ to E can occur in state 8 if the next input symbol is \$ or +. Shifts are possible whenever a transition labeled by a terminal symbol appears. This LR(1) parser is deterministic; grammar $\mathcal{G}_4$ is LR(1). Table 2.2 presents how we can apply the automaton to parse the sentence "n+n*n".

2.2.2.2 Other Deterministic Parsers

LR(k) parsers tend to be exceedingly large, even by today's standards, and are seldom used in practice.

LL(k) Parsers It is therefore worth mentioning the top-down LL(k) parsing method of Lewis II and Stearns (1968), and its variant the *Strong LL(k)* method of Kurki-Suonio (1969) and Rosenkrantz and Stearns (1970). Their popularity resides mostly in the ease of writing and reading such parsers, which allows them to be programmed by hand. The parsing algorithms are simple enough to be implemented as libraries (Leijen and Meijer, 2001; de Guzman, 2003), and are easier to extend with semantic treatment (Lewis II et al., 1974).

SLR(k) and LALR(k) Parsers The LR(k) parsing method works bottom-up, and we should mention its weaker variants *Simple LR(k)* and *LookAhead LR(k)* proposed by DeRemer (1969, 1971) as alternatives yielding much smaller parsers. Figure 2.6 on page 20 presents the underlying automaton

Table 2.2: Successful parse of “n+n*n” using the LR(1) automaton presented in Figure 2.5 on page 17.

parsing stack	input	rules
⓪	$n + n * n\,\$$	shift

The automaton is in the state 0 corresponding to an empty stack. A shift will bring us to state 3. Notice that each LR(1) state has a unique entering symbol, thus we can stack states instead of input symbols.

parsing stack	input	rules
⓪③	$+ n * n\,\$$	reduce $F{\rightarrow}n$

The reduction is correct since the next input symbol, +, is indicated as a valid lookahead symbol. The reduction pops ③, corresponding to n, from the top of the stack, and pushes ②, corresponding to F.

parsing stack	input	rules
⓪②	$+ n * n\,\$$	reduce $T{\rightarrow}F$
⓪④	$+ n * n\,\$$	reduce $E{\rightarrow}T$

Here we would have the choice between a shift to state 7 and the reduction using $E{\rightarrow}T$; since the next input symbol is +, we know we have to choose the reduction.

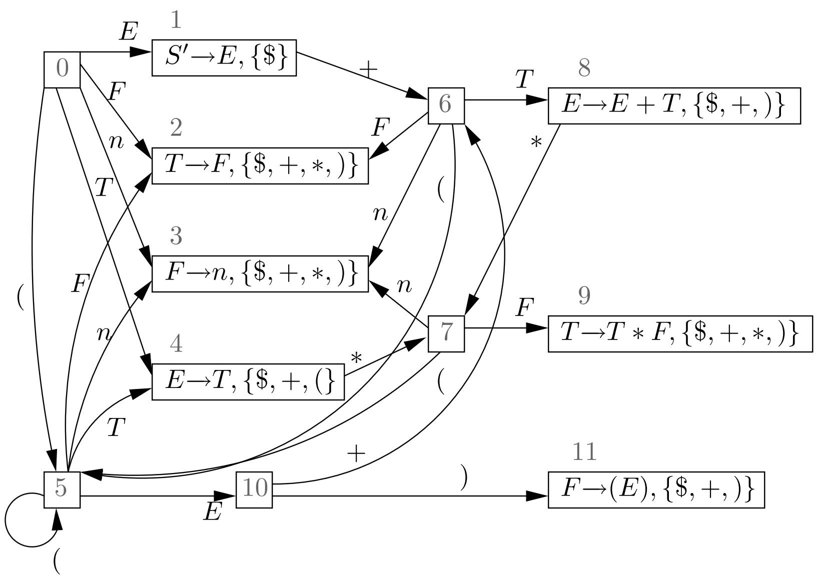
parsing stack	input	rules
⓪①	$+ n * n\,\$$	shift
⓪①⑥	$n * n\,\$$	shift
⓪①⑥③	$* n\,\$$	reduce $F{\rightarrow}n$
⓪①⑥②	$* n\,\$$	reduce $T{\rightarrow}F$
⓪①⑥⑧	$* n\,\$$	shift

This time the choice between shift and reduction is done in favor of a shift since the lookahead symbol does not match the expected symbols for the reduction.

parsing stack	input	rules
⓪①⑥⑧⑦	$n\,\$$	shift
⓪①⑥⑧⑦③	$\$$	reduce $F{\rightarrow}n$
⓪①⑥⑧⑦⑨	$\$$	reduce $T{\rightarrow}T * F$
⓪①⑥⑧	$\$$	reduce $E{\rightarrow}E + T$
⓪①	$\$$	reduce $S'{\rightarrow}E$

for the LALR(1) parser for grammar $\mathcal{G}_4$. The reduction decisions accept a few more lookahead symbols than the LR(1) parser of Figure 2.5 on page 17, and there are only twelve states instead of twenty-two. This LALR(1) parser is also deterministic—a full LR(1) parser is unnecessary for $\mathcal{G}_4$. The parsing steps of sentence “n+n*n” presented in Table 2.2 are exactly the same as the steps using the LALR(1) automaton of Figure 2.6 on the next page since the automata differ only on their treatment of parentheses.

According to a statistical study by Purdom (1974), the size of LALR(1) parsers for practical grammars is linear, while they retain most of the power of full LR(1) parsers. Their popularity is attested by the fame of the LALR(1) parser generator YACC (Johnson, 1975) and of its free counterpart



1
$S' \to E, \{\$\}$
2
$T \to F, \{\$, +, *,)\}$
3
$F \to n, \{\$, +, *,)\}$
4
$E \to T, \{\$, +, (\}$
8
$E \to E + T, \{\$, +,)\}$
9
$T \to T * F, \{\$, +, *,)\}$
11
$F \to (E), \{\$, +,)\}$
0
5
6
7
10

Figure 2.6: LALR(1) automaton for grammar $\mathcal{G}_4$.

GNU Bison (Donnely and Stallman, 2006). Most programming languages are distributed with a tool suite that includes a variant of YACC.

Left Corner, Precedence, Bounded Context, and Others Older bottom-up methods like precedence parsers (Floyd, 1963; Wirth and Weber, 1966; Ichbiah and Morse, 1970; McKeeman et al., 1970) or bounded context parsers (Floyd, 1964; Graham, 1974) are seldom used. Nevertheless, like left corner parsers (Rosenkrantz and Lewis II, 1970; Soisalon-Soininen and Ukkonen, 1979), their compactness makes them sometimes more suitable for pseudo nondeterministic parsing than LR-based methods.

The size of full LR parsers can be contained by merging compatible states (Pager, 1977), or more drastically by having states encode less information and inspecting the stack (Kannapinn, 2001), down to the situation where an arbitrarily deep stack inspection is allowed, in Discriminating Reverse parsing (Fortes Gálvez, 1998).

2.2.2.3 Issues with Deterministic Parsers

Some of the mentioned parsers offer additional fall back mechanisms to deal with nondeterminism, but when working in strict deterministic operation, all process their input in linear time of the input size, and thus respect requirements 2 and 3 on page 16. Quoting Aho and Ullman (1972, Chapter 5),

> We shall have to pay a price for this efficiency, as none of the classes of grammars for which we can construct these efficient parsers generate all the context-free grammars.

Indeed, we will see that requirement 1 on page 16 is not enforced, and proof that the next statement of Aho and Ullman is not quite true

> However, there is strong evidence that the restricted classes of grammars for which we can construct these efficient parsers are adequate to specify all the syntactic features of programming languages that are normally specified by context-free grammars.

2.2.3 Ambiguity in Programming Languages

The very first source of nondeterminism in parsers for programming languages is the presence of ambiguities in the context-free grammar.

As noted by Cantor (1962), an ambiguity issue was already plaguing the ALGOL 60 specification (Naur, 1960), in the form of the now classical *dangling else* ambiguity.

Example 2.1. Figure 2.7a and Figure 2.7b on the following page present two possible parse trees for the same correct ALGOL 60 conditional statement:[5]

if a **then for** i := 1 **step** 1 **until** N **do if** b **then** c:= 0 **else** k := k + 1

Should the "**else** k := k + 1" be part of the "**if** a **then**" (high attachment) or of the "**if** b **then**" (low attachment) conditional statement? □

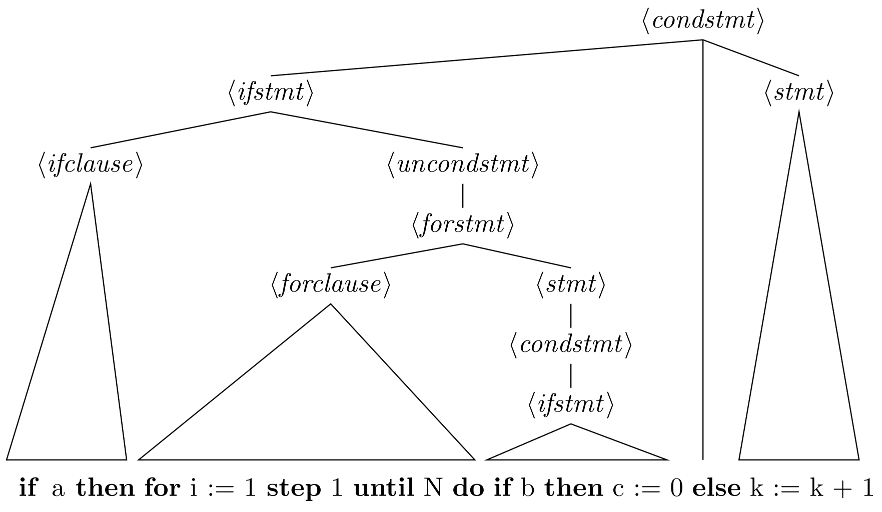
The low attachment solution is clearly the one intended in the specification. The grammar proposed in the specification attempts to enforce it by distinguishing unconditional statements from the others, but fails in the case of the for statements. A solution would be to split the for statement derivations into two cases, with one allowing conditional statements and one not allowing them.

Another solution is to provide some external requirements telling which possibility should be preferred. This *precedence scheme* strategy is quite popular since its proposal by Aho et al. (1975) and Earley (1975) independently, and its implementation in YACC (Johnson, 1975). Ambiguous grammars are smaller, easier to read and to maintain than their disambiguated counterparts for the description of expressions and conditional statements, as we can witness from the comparison between the ambiguous grammar $\mathcal{G}_5$ with rules

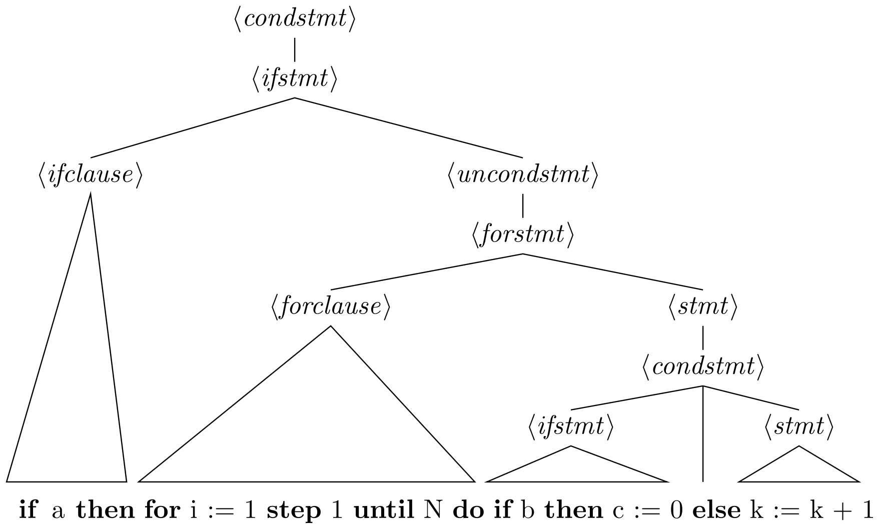
$$E \rightarrow E + E \,|\, E * E \,|\, n \,|\, (E) \qquad (\mathcal{G}_5)$$

and its disambiguated version $\mathcal{G}_4$. The external requirements can specify

[5]Names between angle brackets denote nonterminal symbols.


⟨condstmt⟩
⟨ifstmt⟩
⟨stmt⟩
⟨ifclause⟩
⟨uncondstmt⟩
⟨forstmt⟩
⟨forclause⟩
⟨stmt⟩
⟨condstmt⟩
⟨ifstmt⟩
if a **then** **for** i := 1 **step** 1 **until** N **do** **if** b **then** c := 0 **else** k := k + 1

(a) High attachment.

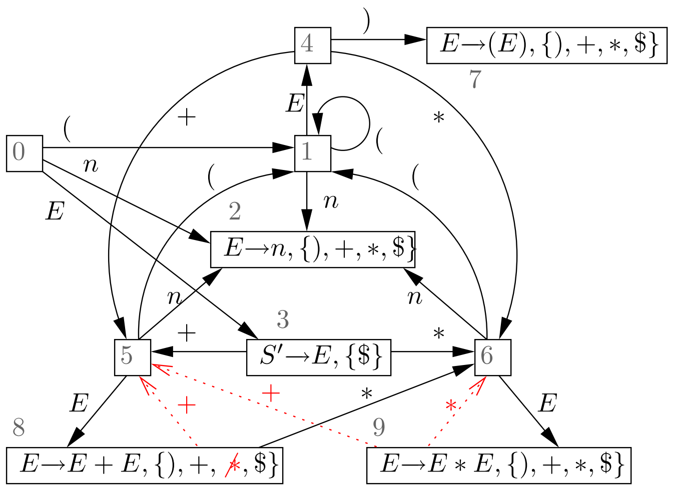

⟨condstmt⟩
⟨ifstmt⟩
⟨ifclause⟩
⟨uncondstmt⟩
⟨forstmt⟩
⟨forclause⟩
⟨stmt⟩
⟨condstmt⟩
⟨ifstmt⟩
⟨stmt⟩
if a **then** **for** i := 1 **step** 1 **until** N **do** **if** b **then** c := 0 **else** k := k + 1

(b) Low attachment.

Figure 2.7: Ambiguity in ALGOL 60.

the associativity and level of precedence of operators, and the preference for a low attachment in conditionals. They are directly enforced in the parsers.

In the case of $\mathcal{G}_5$, we need to specify that multiplication has higher precedence than addition, and that both are left associative in order to have the same parse trees as with $\mathcal{G}_4$. The LALR(1) automaton for $\mathcal{G}_5$ is then smaller, as we can see in Figure 2.8 on the next page, and needs less operations in order to parse the example sentence "n+n*n".



0
1
4
5
6
$E \to (E), \{), +, *, \$\}$
7
2
$E \to n, \{), +, *, \$\}$
3
$S' \to E, \{\$\}$
8
$E \to E + E, \{), +, \cancel{*}, \$\}$
9
$E \to E * E, \{), +, *, \$\}$
(
n
E
+
*
)

Figure 2.8: LALR(1) automaton for $\mathcal{G}_5$. The precedence and associativity requirements rule out the transitions and lookahead symbols in red.

This classical form of ambiguity is well understood; parsers for such grammars are pseudo deterministic and satisfy our three requirements of page 16.[6] Precedence disambiguation has been further studied by Aasa (1995), while a different pseudo deterministic scheme based on the rejection of parse trees is described by Thorup (1994).

The recent popularity of parallel pseudo nondeterministic solutions has accelerated the spread of the general disambiguation filters proposed by Klint and Visser (1994)—at the cost of a violation of requirement 2 on page 16. Van den Brand et al. (2003) present several ambiguous cases motivating the use of filters: typedefs in C (Kernighan and Ritchie, 1988), offside rules in Haskell (Peyton Jones, 2003), and dangling constructions in COBOL (IBM, 1993; Lämmel and Verhoef, 2001). We will discuss these techniques more thoroughly in Section 3.3.2 on page 41.

2.2.3.1 Further Nondeterminism Issues

Ambiguity does not account for all the nondeterminism issues in the syntax of programming languages. The next most common issue is the need for a lookahead of length longer than one symbol. Parr and Quong (1996) describe several instances of the issue. Parser generators like ANTLR (Parr, 2007)

[6]There is a corner case with this disambiguation scheme than can lead to nonterminating parsers (Soisalon-Soininen and Tarhio, 1988). All grammars cannot be handled directly, but the case is quite unlikely to appear in any respectable grammar for a programming language.

or Jikes (Charles, 1991) support user-set lookahead lengths in their SLL(k) and LALR(k) parsers.

Lookahead issues are poorly handled by disambiguation schemes. Parser developers are often forced to modify the grammar. While the algorithmic techniques of Mickunas et al. (1976) could be applied to transform a LR(k) grammar into a LR(1) or another suitable form, the transformation is often considered unpractical, and does not handle unbounded lookahead cases. The issue is thus solved by writing a grammar for a superset of the language, and relying on post processing to reject the unwanted cases.

We will see several examples of syntax issues appearing in programming languages that show the limitations of classical deterministic methods in Section 3.1. Known solutions to these issues imply relaxing one or several of the requirements enunciated on page 16.

Advanced Parsers and their Issues 3

Following the Golden Age of parsing research for programming languages presented in Chapter 2, it has been widely acknowledged amongst computer scientists that the issue was closed. Indeed, developing a small compiler project with YACC is rather easy, and has been taught in many computer science curricula.

Why parsing still gathers some research interest comes from the confrontation with real-life difficulties. As explained by Scott and Johnstone (2006) in their introduction entitled *The Death and Afterlife of Parsing Research*, many programming languages are challenging to parse using the classical deterministic techniques.

We are going to sustain this claim by examining several cases in Section 3.1. The issues presented there are symptomatic of the expressivity gap between what can be specified using a context-free grammar, and what can be actually parsed with a classical deterministic parser, like a YACC-generated LALR(1) parser. Transforming a grammar until we can generate a deterministic parser for it is arduous, and can obfuscate the attached semantics. Moreover, as illustrated by these unbounded lookahead cases, some languages are simply not deterministic. In short, requirement 1 is poorly handled by traditional deterministic parsing methods.

We present then several advanced parsing techniques that give better results in this regard. Two approaches are possible:

- Advanced deterministic parsing methods reduce the width of the expressivity gap, i.e. cases where requirement 1 is violated appear less often, hopefully to the point where the only syntax issues remaining in practice are always caused by ambiguities. The produced parsers are still deterministic, but use more powerful models than DPDAs (Section 3.2).

- Pseudo nondeterministic parsing methods can handle any context-free grammar; therefore they are also called *general* methods. However, they violate requirement 2 (Section 3.3).

3.1 Case Studies

We present four small case studies of difficulties with the syntax of programming languages. All four instances present a LR conflict that requires an unbounded lookahead for its resolution.

3.1.1 Java Modifiers

Programming language specifications usually provide a grammar already acceptable for a LALR(1) parser generator, and not the "natural" grammar they would have used if they had not been constrained. Gosling et al. (1996) interestingly provide both versions of the grammar and present which changes were undertaken in order to obtain a LALR(1) grammar in their Section 19.1.

3.1.1.1 Grammar

Java classes, fields, methods, constructors and interfaces can be declared using modifiers like "**public**" or "**static**". Not all the modifiers can be applied in any declaration. The rules corresponding to the fields and methods declarations in the Java specification are shown in Figure 3.1 on the next page. Among other requirements, the rules explicitly forbid "**abstract**" as a field modifier, while some modifiers are acceptable for fields and methods alike, e.g. "**public**".

3.1.1.2 Input Examples

Example 3.1. Consider now the partial Java input:

```
public class LookaheadIssue {
    public static int length
```

When a deterministic parser sees the terminal string "**public static int** length" in its lookahead window, it cannot tell whether ε should be reduced to a $\langle \textit{FieldModifiers} \rangle$ as in:

```
    public static int length = 1;
```

or to a $\langle \textit{MethodModifiers} \rangle$ as in:

```
    public static int length (String s) {
        return s.length();
    }
```

$$\begin{array}{rcl}
\langle \mathit{FieldDeclaration} \rangle & \rightarrow & \langle \mathit{FieldModifiers} \rangle \; \langle \mathit{Type} \rangle \; \langle \mathit{VariableDeclarators} \rangle \; ; \\
\langle \mathit{FieldModifiers} \rangle & \rightarrow & \langle \mathit{FieldModifiers} \rangle \; \langle \mathit{FieldModifier} \rangle \\
 & | & \varepsilon \\
\langle \mathit{FieldModifier} \rangle & \rightarrow & \mathbf{public} \\
 & | & \mathbf{protected} \\
 & | & \mathbf{private} \\
 & | & \mathbf{final} \\
 & | & \mathbf{static} \\
 & | & \mathbf{transient} \\
 & | & \mathbf{volatile} \\
\langle \mathit{MethodHeader} \rangle & \rightarrow & \langle \mathit{MethodModifiers} \rangle \; \langle \mathit{ResultType} \rangle \; \langle \mathit{MethodDeclarator} \rangle \\
\langle \mathit{MethodModifiers} \rangle & \rightarrow & \langle \mathit{MethodModifiers} \rangle \; \langle \mathit{MethodModifier} \rangle \\
 & | & \varepsilon \\
\langle \mathit{MethodModifier} \rangle & \rightarrow & \mathbf{public} \\
 & | & \mathbf{protected} \\
 & | & \mathbf{private} \\
 & | & \mathbf{static} \\
 & | & \mathbf{abstract} \\
 & | & \mathbf{final} \\
 & | & \mathbf{native} \\
 & | & \mathbf{synchronized} \\
\langle \mathit{ResultType} \rangle & \rightarrow & \langle \mathit{Type} \rangle \\
 & | & \mathbf{void}
\end{array}$$

Figure 3.1: Fields and methods declarations in Java. $\langle \mathit{FieldDeclaration} \rangle$ and $\langle \mathit{MethodHeader} \rangle$ appear in the same context, and both $\langle \mathit{VariableDeclarators} \rangle$ and $\langle \mathit{MethodDeclarator} \rangle$ first derive identifiers.

The same happens for the reductions of "**public**" and "**static**". A similar problem happens after reading "**int**" and seeing the identifier "length" as lookahead symbol, where the parser does not know whether to reduce to $\langle \mathit{Type} \rangle$ or to $\langle \mathit{ResultType} \rangle$. □

This set of grammar productions is not LR(k) for any fixed k since we could repeat the modifiers indefinitely, growing the inconclusive lookahead string "**public static int** length" to an arbitrary length larger than the chosen k length.

3.1.1.3 Resolution

The solution of Gosling et al. to this problem was to combine all kinds of modifiers in a single $\langle \textit{Modifier} \rangle$ nonterminal, and to check whether they are allowed in their context at a later stage of compiler analysis. The $\langle \textit{MethodHeader} \rangle$ production was then split into two rules to distinguish the case of a "**void**" return type from a $\langle \textit{Type} \rangle$ return type.

3.1.2 Standard ML Layered Patterns

Unlike the grammars sometimes provided in other programming language references, the Standard ML grammar defined by Milner et al. (1997, Appendix B) is not put in LALR(1) form. In fact, it clearly values simplicity over ease of implementation, and includes highly ambiguous rules like $\langle \textit{dec} \rangle \rightarrow \langle \textit{dec} \rangle\ \langle \textit{dec} \rangle$.

The definition of Standard ML was carefully reviewed by Kahrs (1993), who presents in his Section 8.4 two issues with the syntax of the language. We consider here the first of these two issues, which arises in the syntax of layered patterns.

3.1.2.1 Grammar

The set of expunged grammar rules illustrating this first issue is quite small:

$$\begin{array}{rcl}
\langle \textit{pat} \rangle & \rightarrow & \langle \textit{atpat} \rangle \\
& | & \langle \textit{pat} \rangle : \langle \textit{ty} \rangle \\
& | & \textit{vid}\ \langle \textit{tyop} \rangle\ \textbf{as}\ \langle \textit{pat} \rangle \\
\langle \textit{atpat} \rangle & \rightarrow & \textit{vid} \\
\langle \textit{tyop} \rangle & \rightarrow & : \langle \textit{ty} \rangle \\
& | & \varepsilon
\end{array}$$

where $\langle \textit{pat} \rangle$ denotes a pattern, $\langle \textit{atpat} \rangle$ an atomic pattern, *vid* a value identifier and $\langle \textit{ty} \rangle$ a type.

3.1.2.2 Input Examples

Example 3.2. We consider now the partial input

val f = **fn** triple : int∗int∗int

A deterministic parser cannot distinguish whether the *vid* "triple" is an atomic pattern $\langle \textit{atpat} \rangle$, with an associated type "int∗int∗int", as in

val f = **fn** triple : int∗int∗int => triple

or is the start of a more complex layered pattern as in

val f = **fn** triple : int∗int∗int **as** (_, _, z) => z + 1 □

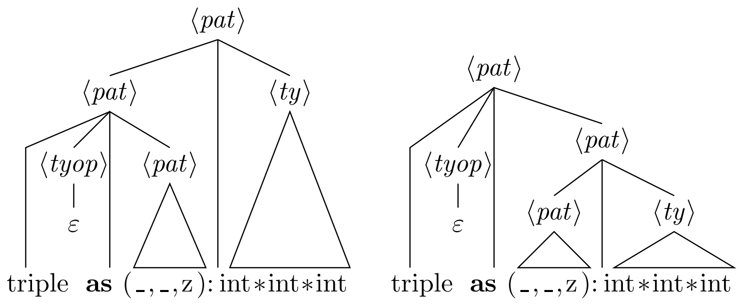

⟨pat⟩
⟨pat⟩
⟨ty⟩
⟨tyop⟩
⟨pat⟩
ε
triple as (_,_,z): int∗int∗int
⟨pat⟩
⟨tyop⟩
⟨pat⟩
ε
⟨pat⟩
⟨ty⟩
triple as (_,_,z): int∗int∗int

Figure 3.2: Ambiguity in the pattern syntax of Standard ML.

Types ⟨*ty*⟩ in Standard ML can be quite involved expressions, of arbitrary length. The inconclusive lookahead string "int∗int∗int" can be longer than the fixed k of a LR(k) parser. Further intricacy derives from the dangling ambiguity of the syntax, where "triple **as** (_,_,z): int∗int∗int" can be understood by attaching the type "int∗int∗int" to the pattern "(_,_,z)" or to the entire layered pattern "triple **as** (_,_,z)", as demonstrated in Figure 3.2.

3.1.2.3 Resolution

The solution often adopted by Standard ML compilers is to replace the rule ⟨*pat*⟩→*vid* ⟨*tyop*⟩ **as** ⟨*pat*⟩ by ⟨*pat*⟩→⟨*pat*⟩ ⟨*tyop*⟩ **as** ⟨*pat*⟩, and exclude after parsing the cases where the pattern is not a variable.

Example 3.3. As noted by Kahrs, the incorrect sentence

val f = **fn** (y) **as** z => z

is allowed by a few implementations because the pattern "(y)" is semantically identical to the value identifier "y". Nonetheless, almost all of the compilers we tested correctly identified the problem and reported an error. □

3.1.3 C++ Qualified Identifiers

Some of the renewed interest in parsing techniques came from the trickiness of parsing C++. First designed as a preprocessor for C, the language evolved into a complex standard (ISO, 1998). Its rather high level of syntactic ambiguity calls for nondeterministic parsing methods, and therefore the published grammar makes no attempt to fit in the LALR(1) class.

3.1.3.1 Grammar

We are interested in one particular issue with the syntax of qualified identifiers, corresponding to the (simplified) grammar rules

$$\begin{array}{rcl}
\langle \mathit{id\_expression} \rangle & \to & \langle \mathit{unqualified\_id} \rangle \\
 & | & \langle \mathit{qualified\_id} \rangle \\
\langle \mathit{unqualified\_id} \rangle & \to & \mathit{identifier} \\
 & | & \langle \mathit{template\_id} \rangle \\
\langle \mathit{qualified\_id} \rangle & \to & \langle \mathit{nested\_name\_specifier} \rangle\ \langle \mathit{unqualified\_id} \rangle \\
\langle \mathit{nested\_name\_specifier} \rangle & \to & \langle \mathit{unqualified\_id} \rangle {::} \langle \mathit{nested\_name\_specifier} \rangle \\
 & | & \langle \mathit{unqualified\_id} \rangle {::} \\
\langle \mathit{template\_id} \rangle & \to & \mathit{identifier}\ {<} \langle \mathit{template\_argument} \rangle {>} \\
\langle \mathit{template\_argument} \rangle & \to & \langle \mathit{id\_expression} \rangle
\end{array}$$

Qualified identifiers ⟨*qualified_id*⟩ are qualified through a sequence of unqualified identifiers ⟨*unqualified_id*⟩ separated by double colons “::”, before the identifier itself. Moreover, each unqualified identifier can be a template identifier ⟨*template_id*⟩, where the argument of the template, between angle brackets “<” and “>”, might be a ⟨*qualified_id*⟩ as well.

3.1.3.2 Input Examples

Example 3.4. A LALR(1) shift/reduce conflict appears with this set of rules. A parser fed with “A::”, and seeing an identifier “B” in its lookahead window, has a nondeterministic choice between

- reducing “A::” to a single ⟨*nested_name_specifier*⟩, in the hope that “B” will be the identifier qualified by “A::”, as in “A::B<C::D>”, and
- shifting, in the hope that “B” will be a specifier of the identifier actually qualified, for instance “E” in “A::B<C::D>::E”.

An informed decision requires an exploration of the specifier starting with “B” in the search of a double colon symbol. The need for an unbounded lookahead occurs if “B” is the start of an arbitrarily long template identifier.

Note that the double colon token might also appear inside a template argument. Considering that the conflict could also arise there, as after reading “A<B::” in “A<B::C<D::E>::F>::G”, we see that it can be arduous to know whether a “::” symbol is significant for the resolution of the conflict or not. □

3.1.3.3 Resolution

We can trivially amend the rules of ⟨*nested_name_specifier*⟩:

$$\begin{array}{rcl}
\langle \mathit{nested\_name\_specifier} \rangle & \to & \langle \mathit{nested\_name\_specifier} \rangle \langle \mathit{unqualified\_id} \rangle {::} \\
 & | & \langle \mathit{unqualified\_id} \rangle {::}
\end{array}$$

$$\begin{array}{rcl}
\langle \mathit{dec} \rangle & \rightarrow & \textbf{fun}\ \langle \mathit{fvalbind} \rangle \\
\langle \mathit{fvalbind} \rangle & \rightarrow & \langle \mathit{sfvalbind} \rangle \\
 & | & \langle \mathit{fvalbind} \rangle\ '|'\ \langle \mathit{sfvalbind} \rangle \\
\langle \mathit{sfvalbind} \rangle & \rightarrow & \mathit{vid}\ \langle \mathit{atpats} \rangle = \langle \mathit{exp} \rangle \\
\langle \mathit{atpats} \rangle & \rightarrow & \langle \mathit{atpat} \rangle \\
 & | & \langle \mathit{atpats} \rangle\ \langle \mathit{atpat} \rangle \\
\langle \mathit{exp} \rangle & \rightarrow & \textbf{case}\ \langle \mathit{exp} \rangle\ \textbf{of}\ \langle \mathit{match} \rangle \\
 & | & \mathit{vid} \\
\langle \mathit{match} \rangle & \rightarrow & \langle \mathit{mrule} \rangle \\
 & | & \langle \mathit{match} \rangle\ '|'\ \langle \mathit{mrule} \rangle \\
\langle \mathit{mrule} \rangle & \rightarrow & \langle \mathit{pat} \rangle \mathrel{=>} \langle \mathit{exp} \rangle \\
\langle \mathit{pat} \rangle & \rightarrow & \mathit{vid}\ \langle \mathit{atpat} \rangle \\
\langle \mathit{atpat} \rangle & \rightarrow & \mathit{vid}
\end{array}$$

Figure 3.3: Standard ML function value binding and case expressions.

Switching to left recursion removes the conflict. This correction was made by the Standards Committee in 2003.[1] Interestingly, the C++ grammar of the Elsa parser (McPeak and Necula, 2004) still employs a right recursion.

3.1.4 Standard ML Case Expressions

A second case where an unbounded lookahead is needed by a Standard ML parser in order to make the correct decision is described by Kahrs. The issue arises with alternatives in function value binding and **case** expressions.

3.1.4.1 Grammar

We consider the relevant rules given in Figure 3.3. The rules describe Standard ML declarations $\langle \mathit{dec} \rangle$ for functions, where each function name *vid* is bound, for a sequence $\langle \mathit{atpats} \rangle$ of atomic patterns, to an expression $\langle \mathit{expr} \rangle$ using the rule $\langle \mathit{sfvalbind} \rangle \rightarrow \mathit{vid}\ \langle \mathit{atpats} \rangle = \langle \mathit{exp} \rangle$. Different function value bindings can be separated by alternation symbols "|". Standard ML **case** expressions associate an expression $\langle \mathit{exp} \rangle$ with a $\langle \mathit{match} \rangle$, which is a sequence of matching rules $\langle \mathit{mrule} \rangle$ of form $\langle \mathit{pat} \rangle \mathrel{=>} \langle \mathit{exp} \rangle$, separated by alternation symbols "|".

[1] See `http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#125`. The correction was not motivated by this conflict but by an ambiguity issue, and the fact that the change eliminates the conflict seems to be a fortunate coincidence.

3.1.4.2 Input Examples

Example 3.5. Using mostly these rules, the filter function of the SML/NJ Library could be written (Lee, 1997) as:

```
datatype 'a option = NONE | SOME of 'a

fun filter pred l =
  let
    fun filterP (x::r, l) =
        case (pred x)
          of SOME y => filterP(r, y::l)
           | NONE => filterP(r, l)
      | filterP ([], l) = rev l
  in
    filterP (l, [])
  end
```

The Standard ML compilers consistently reject this correct input, often pinpointing the error at the equal sign in "| filterP ([], l) = rev l". □

If we implement our set of grammar rules in a traditional LALR(1) parser generator like GNU Bison (Donnely and Stallman, 2006), it reports a single shift/reduce conflict, a nondeterministic choice between two parsing actions:

```
state 20
    6 exp: "case" exp "of" match .
    8 match: match . '|' mrule

    '|'  shift, and go to state 24
    '|'       [reduce using rule 6 (exp)]
```

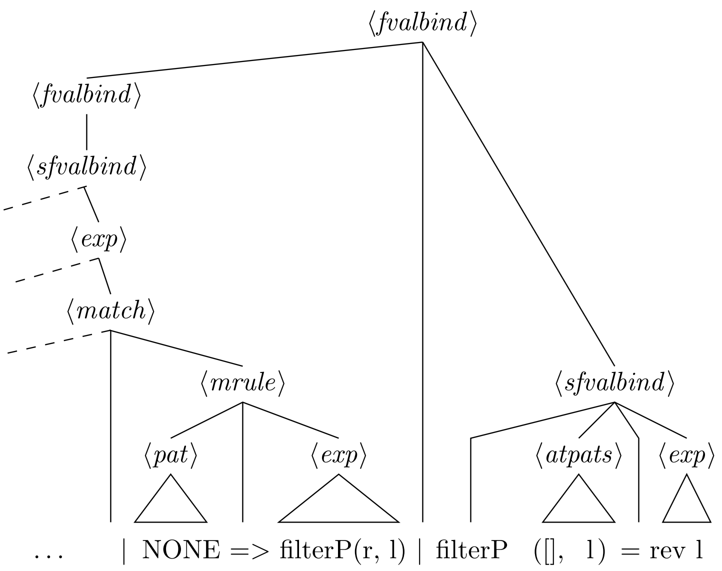
The conflict takes place just before "| filterP ([], l) = rev l" with the program of Example 3.5.

If we choose one of the actions—shift or reduce—over the other, we obtain the parses drawn in Figure 3.4 on the next page. The shift action is chosen by default by Bison, and ends on a parse error when seeing the equal sign where a double arrow was expected, exactly where the Standard ML compilers report an error.

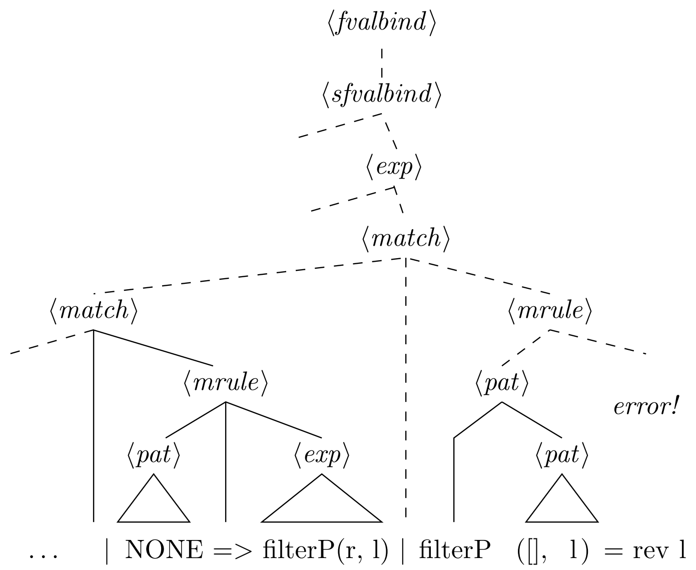
Example 3.6. To make matters worse, we also have to cope with a dangling ambiguity:

case a **of** b => **case** b **of** c => c | d => d

In this expression, should the dangling "d => d" matching rule be attached to "**case** b" or to "**case** a"? The Standard ML definition indicates that the matching rule should be attached to "**case** b". In this case, the shift should be chosen rather than the reduction. □



(a) Correct parse tree when reducing.



(b) Attempted parse when shifting.

Figure 3.4: Partial parse trees corresponding to the two actions in conflict in Example 3.5 on the preceding page.

Our two examples show that we cannot blindly choose one particular action over the other. Nonetheless, this puzzle is not totally inextricable: we could make the correct decision if we had more information at our disposal.

The "=" sign in the lookahead string "| filterP ([], l) = rev l" indicates that the alternative is at the topmost function value binding ⟨*fvalbind*⟩ level, and not at the "**case**" level, or it would be a "=>" sign. But the sign can be arbitrarily far away in the lookahead string: an atomic pattern ⟨*atpat*⟩ can derive a sequence of tokens of unbounded length. The conflict requires an unbounded lookahead.

3.1.4.3 Resolution

As stated earlier, the Standard ML compilers uniformly reject any input that exercises this syntax issue. The issue is one of the few major defects of the Standard ML definition according to a survey by Rossberg (2006):

> There is no comment on how to deal with the most annoying problem in the full grammar, the infinite lookahead required to parse combinations of function clauses and **case** expressions [...] [Parsing] this would either require horrendous grammar transformations, backtracking, or some nasty and expensive lexer hack.

The solution called upon by the maintainers of Standard ML compilers and by Rossberg is to correct the definition, by indicating explicitly that the inputs currently rejected are indeed incorrect.

The previous syntax issues of our case studies were not extremely serious. The solution of Gosling et al. to the Java modifiers issue hindered the reliability of the front end since an *ad hoc* treatment was required. The solution to the first Standard ML issue resulted in some incorrect inputs to be accepted by a few compilers, but with no effect on the semantics. This second Standard ML issue has a dire consequence: some Standard ML programs matching the specification are rejected by the compilers.

3.2 Advanced Deterministic Parsing

In order to handle unbounded lookahead issues, advanced deterministic parsing techniques depart from the DPDA model. We review next three families of advanced deterministic parsing methods.

3.2.1 Predicated Parsers

The use of predicates to guide the parser first stemmed from attribute grammars parsers (e.g. Ganapathi, 1989) and was thus concerned with semantics. *Syntactic predicates* on the other hand were implemented in ANTLR by Parr and Quong (1995). Such predicates are explicitly given by the grammar developer to impersonate an oracle that directs the parser in nondeterministic cases.

For instance, the following ANTLR predicate distinguishes Java method declarations:

```
((MethodModifier)* ResultType MethodDeclarator)?
```

Predicates can match a non regular context-free language, and can be used to solve all our lookahead issues. Once the predicate is matched, normal parsing resumes, potentially reparsing a large part of the input. Syntactic predicates can be highly inefficient at parsing time, potentially yielding exponential processing time, though the bad performance could probably be amended using memoization techniques.

Rather than performance, which is after all our third and least important requirement, our most serious concern is that syntactic predicates are error-prone: the grammar writer manually interferes with the parser. Ideally, the parser generator should infer such predicates directly from the context-free grammar. This is exactly what the regular lookahead and noncanonical constructions attempt.

3.2.2 Regular Lookahead Parsers

The *LR-Regular* parsers defined by Čulik and Cohen (1973) base their decisions on the entire remaining input. Unbounded lookahead strings can be discriminated as being part of distinct regular languages. If a finite set of such disjoint regular lookahead languages—called a *regular partition*—is enough to decide in all cases, then deterministic parsing is possible. LR(k) parsing is then a special case of LRR, where the lookahead strings are partitioned according to their k first symbols.

The original idea of Čulik and Cohen relies on a two scan parsing algorithm. The first scan is done right to left by a DFA and yields a string of symbols identifying the blocks of the partition. The result of this first scan is a sentence of a LR(0) grammar, which can be constructed from the original grammar. The second scan is done left to right by a DPDA for this LR(0) grammar. The overall process runs in linear time. The same generalization can be applied to LL(k) parsing to yield *LL-Regular* parsers (Čulik, 1968; Jarzabek and Krawczyk, 1975; Nijholt, 1976; Poplawski, 1979).

Regular Separability Problem The existence of a regular partition that distinguishes the possible lookahead strings is known as the regular *separability problem*, and is undecidable (Hunt III, 1982). We cannot tell for an arbitrary context-free grammar whether it is LRR (or LLR). The construction of regular lookahead parsers thus relies on heuristics in order to produce a suitable regular partition, and even if one exists, it might fail to find it. Moreover, the two scans parsing algorithm is rather unpractical. The LRR method of Čulik and Cohen is mostly of theoretical interest.

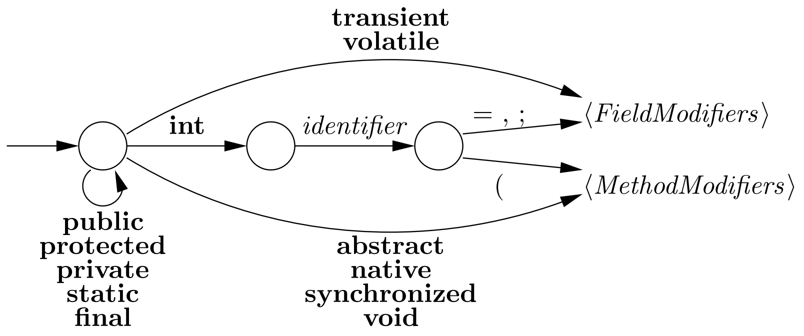


Figure 3.5: Discriminating lookahead DFA for the conflict between $\langle \textit{MethodModifiers} \rangle \rightarrow \varepsilon$ and $\langle \textit{FieldModifiers} \rangle \rightarrow \varepsilon$ in Java.

The conflict with C++ qualified identifiers we studied in Section 3.1.3 on page 29 is an instance of a conflict with non regularly separable lookahead languages. This particular example is therefore out of the grasp of LR-Regular parsing.

Practical Implementations Nonetheless, there are practical implementations of the LRR principle: the SLR(k)-based method of Kron et al. (1974), the XLR method of Baker (1981), the RLR method of Boullier (1984), the method of Seité (1987), the LAR method of Bermudez and Schimpf (1990) and the BCLRR method of Farré and Fortes Gálvez (2001). They all rely on a DFA to explore the remaining input in an attempt to discriminate between conflicting parsing actions. Figure 3.5 presents a DFA solving the unbounded lookahead issue in Java presented earlier in Section 3.1.1 on page 26. Parsing is done in a single sweep, but an unbounded lookahead might be consulted in order to decide of some actions. The lookahead DFA results from a regular approximation of the language expected next by the parser, and its computation is the main difference between the various practical LRR methods. A DFA exploration is also at the heart of the LL(*) algorithm introduced in the upcoming version 3 of ANTLR (Parr, 2007).

Last of all, it is worth noting that a small performance issue can arise, since the DFA-based implementations can be tricked into a quadratic parsing time behavior (see Section 5.3.1 on page 115).

3.2.3 Noncanonical Parsers

Rather than relying on a DFA to explore the remaining input like a LRR parser would do, a noncanonical parser can enter this remaining text and parse it. When confronted with nondeterminism, a bottom-up noncanonical parser is able to suspend a reduction decision, and to start parsing the remaining input. The parser performs some reductions there, and can resume to the conflict point and use nonterminal symbols—resulting from the reduction of a possibly unbounded amount of input—in its lookahead window to solve the initial nondeterministic choice.

Two-Stack Pushdown Automata The noncanonical parsing mechanism is implemented by means of a *two-stack pushdown automaton* (2PDA, introduced by Colmerauer (1970) for noncanonical parsing, and employed in some older parsing methods, e.g. by Griffiths and Petrick (1965)): the first *parsing stack* corresponds to the usual pushdown stack, while the second *input stack* holds the initial input and partially reduced input when the parser resumes to an earlier position. Different noncanonical parser generators build different controls for the 2PDA, just like the different deterministic parser generators do for PDA. The two stacks allow to return to a previously suspended decision point and try to solve the conflict in the light of the newly reduced symbols that just appeared in the lookahead window. Bounding the distance to this suspended decision point guarantees that the parser will run in linear time of the length of the input. Table 3.1 on the following page presents the use of a 2PDA to parse a problematic Java input following the grammar rules of Figure 3.1 on page 27.

History of Noncanonical Parsing The idea of noncanonical parsing can be traced back to an ending remark of Knuth (1965), which gave later birth to LR(k, t) parsing—meaning the parser can suspend up to t successive reduction decisions. Noncanonical parsing methods have been devised as extensions to many bottom-up parsing methods since, starting with the total precedence of Colmerauer (1970) extending precedence parsing, followed by BCP(m, n) of Williams (1975) for bounded context, LR(k, ∞) and *FSPA(k)* (Finite State Parsing Automaton) of Szymanski (1973) for LR(k), and NSLR(1) of Tai (1979) for SLR(1).

Szymanski and Williams (1976) further studied a general framework for noncanonical parsing, with a negative result: there is no algorithmic construction for LR(k, ∞) and FSPA(k) parsers, which denote respectively unconstrained noncanonical LR(k) parsing and noncanonical LR(k) parsing with a finite control. LR(k, ∞) is clearly not computable, whereas FSPA(k) can be compared to full LRR: in noncanonical parsing, we need an accurate regular approximation of the *parsing stack language*, and thus we hit again the wall of the undecidability of the regular separability problem.

Other Uses Noncanonical parsers can be constructed for random sentential forms and not just for canonical right or left sentential forms. They are thus appropriate for many parsing-related problems where more flexibil-

Table 3.1: Recognition of "**private int** n = 0;" by a noncanonical parser with $k = 2$ symbols of lookahead.

parsing stack	input stack	rules
\$	**private int** n=0; \$	shift
We have the choice between the reductions $\langle \mathit{FMods}\rangle \rightarrow \varepsilon$ and $\langle \mathit{MMods}\rangle \rightarrow \varepsilon$; in doubt, we suspend this decision and shift in hope of finding the answer further in the input.		
\$ **private**	**int** n=0; \$	shift
\$ **private int**	n=0; \$	reduce $\langle \mathit{IntegralType}\rangle \rightarrow$**int**
Though we are still in the dark regarding whether we are in a field or in a method declaration, the "**int**" keyword can be reduced in the same way in both cases. The two topmost symbols of the parsing stack are pushed onto the input stack where they form the new 2-lookahead window.		
\$	**private** $\langle \mathit{IntegralType}\rangle$ n=0; \$	shift
However, the 2-lookahead window does not provide enough information to decide anything new, and we have to shift again. Let us fast-forward to the interesting case, where we have just reduced "n = 0" to $\langle \mathit{VDecls}\rangle$. The 2-lookahead window now holds $\langle \mathit{Type}\rangle$, which could appear for fields as well as for methods, followed by $\langle \mathit{VDecls}\rangle$, which is specific of field declarations.		
\$ **private**	$\langle \mathit{Type}\rangle\langle \mathit{VDecls}\rangle$; \$	reduce $\langle \mathit{FMod}\rangle \rightarrow$**private**
\$	$\langle \mathit{FMod}\rangle\langle \mathit{Type}\rangle\langle \mathit{VDecls}\rangle$; \$	reduce $\langle \mathit{FMods}\rangle \rightarrow \varepsilon$
The remaining parsing steps do not exhibit any noncanonical behavior, and the input is successfully parsed.		

ity comes handy: parallel parsing (Fischer, 1975; Schell, 1979), scannerless parsing (Salomon and Cormack, 1989), multi-axiom grammars parsing (Rus and Jones, 1998), robust parsing (Ruckert, 1999), or incremental parsing (Overbey, 2006).

Comparison with LRR Parsing A comparison with practical LRR implementations ends in a draw: while the "lookahead" exploration of noncanonical parsers is more accurate since it recognizes context-free languages instead of regular ones, a noncanonical parser is helpless if the lookahead string cannot be reduced. In other words, noncanonical parsing is more powerful on the language level, but incomparable on the grammar level. Furthermore, the parse example of Table 3.1 illustrates the need for $k = 2$ symbols of reduced lookahead, while in practice noncanonical methods are limited to a single symbol of lookahead. The two exceptions are the Generalized Piecewise LR (GPLR, Schell, 1979) algorithm and the noncanonical extension of Farré and Fortes Gálvez (2004) to DR(k) parsing: both emulate an unbounded reduced lookahead length, but both can exhibit a quadratic parsing time behavior. The ideal deterministic parsing technique remains to be found; meanwhile, general parsing techniques might be the solution.

3.3 General Parsing

Backtracking (Brooker and Morris, 1960; Irons, 1961; Kuno, 1965) or recognition matrix (Cocke and Schwartz, 1970; Younger, 1967; Kasami, 1965) parsers are some of the oldest parsing algorithms. Pseudo nondeterminism was then mandatory in order to palliate the shortcomings of naive parsing techniques, like the shift reduce parser presented in Section 2.1.2. The introduction of practical deterministic parsers then reduced the need for pseudo nondeterminism, while offering much better performance.

Parallel pseudo nondeterminism was later investigated by Earley (1970) and Lang (1974). The performance of their parsers makes them usable in practice, since they work in linear time for deterministic portions of the grammars and cubic time at worst. At first, parallel nondeterministic parsing was not considered for programming languages, but found its niche in natural language processing. Later, the quickly increasing computational power has encouraged their adoption by a growing number of programming languages parser developers.

3.3.1 Families of General Parsers

3.3.1.1 Backtracking Parsers

Both LL and LR algorithms can be easily extended with a backtracking mechanism—in fact, most LL parsing tools implement a backtracking engine (Breuer and Bowen, 1995; Leijen and Meijer, 2001; de Guzman, 2003; Parr, 2007), while there exist quite a few backtracking LR tools (Grosch, 2002; Spencer, 2002; Dodd and Maslov, 2006; Thurston and Cordy, 2006).

Some backtracking engines work on a first-match basis on an *ordered* grammar, and thus return the first possible parse of an ambiguous sentence. The ordering principle for filtering retry points is pushed to the limit with the backtracking technique of Birman and Ullman (1973) and Ford (2002). Ensuring that the ordering corresponds to the desired disambiguated parse can be arduous, while returning all parses is usually deemed too computationally expensive—see however Johnstone and Scott (1998).

3.3.1.2 Chart Parsers

The offspring of the Earley parsing algorithm can be found in the family of *chart* or *dynamic* parsing algorithms (Kay, 1980; Graham et al., 1980; Pereira and Warren, 1983; Sikkel, 1997), preeminent in the field of natural language processing.

The chart contains collections of partial parses—or *items*—that correspond to the input read so far. Table 3.2 on the next page presents the item sets of an Earley parser (see the definitions in Section A.1.6) for the ambiguous sentence "x **as** z: int". The sentence is accepted in two different ways, corresponding to the items $(\langle pat\rangle \rightarrow \langle pat\rangle : \langle ty\rangle \bullet, 0, 5)$ and $(\langle pat\rangle \rightarrow vid\ \langle tyop\rangle\ \textbf{as}\ \langle pat\rangle \bullet, 0, 5)$ that appear in the last set.

There are many implementations of the original Earley algorithm available for programming languages parsing, though they seem less popular than GLR parsers.

3.3.1.3 Generalized LR Parsers

The parallel nondeterministic algorithm of Lang has found its practical realization in the *Generalized LR* parsing algorithm of Tomita (1986). The multiple parallel executions of the parser are merged into a so-called *graph structured stack* (GSS) that allows to contain the running complexity. If GLR parsing was originally aimed towards natural language processing, it has since found an enthusiastic response in the field of programming languages parsing. The low amount of nondeterminism in most programming languages makes them an ideal target for GLR parsing, avoiding the maintenance of too many parallel stacks. An example parse on the Standard ML program of Section 3.1.4 is given in Table 3.3 on page 42.

Various variations of Tomita's algorithm have been defined, correcting some performance or ε rules issues; Scott and Johnstone (2006) present one such variant and a nice survey of the GLR parsing history. Parsing performance reported there, and also by McPeak and Necula (2004) for their Elkhound parser generator, are within reasonable bounds when compared to deterministic parsers. Besides Elkhound, there are many more implementations available, notably in the ASF+SDF framework (Heering et al., 1989) and GNU Bison (Donnely and Stallman, 2006).

The distinction between chart parsers and GLR parsers is a bit artificial; all exert some form of memoization (or *tabulation*) to store intermediate results and obtain polynomial space and time complexity bounds (Nederhof and Satta, 2004). Generalized LR parsers can thus be seen as Earley parsers with a precomputed component. The distinction is further blurred when one considers generalized left corner parsers (Nederhof, 1993; Moore, 2004).

General parsing is rather flexible, and is employed for various parsing problems, including incremental parsing (Wagner and Graham, 1997), lexical ambiguity (Aycock and Horspool, 2001), or scannerless parsing (van den Brand et al., 2002). Nevertheless, as hinted at the beginning of the section, there remains one open issue with general parsing algorithms: they do not shield the parser programmer from ambiguities.

Table 3.2: Earley item sets for the input “x **as** z: int” according to the Standard ML grammar subset described in Section 3.1.2.1 on page 28.

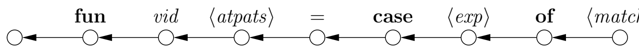
	$\langle \mathit{pat}\rangle \rightarrow \bullet \langle \mathit{atpat}\rangle, 0, 0$
	$\langle \mathit{pat}\rangle \rightarrow \bullet \langle \mathit{pat}\rangle : \langle \mathit{ty}\rangle, 0, 0$
	$\langle \mathit{pat}\rangle \rightarrow \bullet\, \mathit{vid}\ \langle \mathit{tyop}\rangle\ \textbf{as}\ \langle \mathit{pat}\rangle, 0, 0$
	$\langle \mathit{atpat}\rangle \rightarrow \bullet\, \mathit{vid}, 0, 0$
x	
	$\langle \mathit{pat}\rangle \rightarrow \mathit{vid} \bullet \langle \mathit{tyop}\rangle\ \textbf{as}\ \langle \mathit{pat}\rangle, 0, 1$
	$\langle \mathit{atpat}\rangle \rightarrow \mathit{vid} \bullet, 0, 1$
	$\langle \mathit{tyop}\rangle \rightarrow \bullet \langle \mathit{ty}\rangle, 1, 1$
	$\langle \mathit{tyop}\rangle \rightarrow \bullet, 1, 1$
	$\langle \mathit{pat}\rangle \rightarrow \langle \mathit{atpat}\rangle \bullet, 0, 1$
	$\langle \mathit{pat}\rangle \rightarrow \langle \mathit{pat}\rangle \bullet : \langle \mathit{ty}\rangle, 0, 1$
	$\langle \mathit{pat}\rangle \rightarrow \mathit{vid}\ \langle \mathit{tyop}\rangle \bullet \textbf{as}\ \langle \mathit{pat}\rangle, 0, 1$
as	
	$\langle \mathit{pat}\rangle \rightarrow \mathit{vid}\ \langle \mathit{tyop}\rangle\ \textbf{as} \bullet \langle \mathit{pat}\rangle, 0, 2$
	$\langle \mathit{pat}\rangle \rightarrow \bullet \langle \mathit{atpat}\rangle, 2, 2$
	$\langle \mathit{pat}\rangle \rightarrow \bullet \langle \mathit{pat}\rangle : \langle \mathit{ty}\rangle, 2, 2$
	$\langle \mathit{pat}\rangle \rightarrow \bullet\, \mathit{vid}\ \langle \mathit{tyop}\rangle\ \textbf{as}\ \langle \mathit{pat}\rangle, 2, 2$
	$\langle \mathit{atpat}\rangle \rightarrow \bullet\, \mathit{vid}, 2, 2$
z	
	$\langle \mathit{pat}\rangle \rightarrow \mathit{vid} \bullet \langle \mathit{tyop}\rangle\ \textbf{as}\ \langle \mathit{pat}\rangle, 2, 3$
	$\langle \mathit{atpat}\rangle \rightarrow \mathit{vid} \bullet, 2, 3$
	$\langle \mathit{tyop}\rangle \rightarrow \bullet \langle \mathit{ty}\rangle, 3, 3$
	$\langle \mathit{tyop}\rangle \rightarrow \bullet, 3, 3$
	$\langle \mathit{pat}\rangle \rightarrow \langle \mathit{atpat}\rangle \bullet, 2, 3$
	$\langle \mathit{pat}\rangle \rightarrow \langle \mathit{pat}\rangle \bullet : \langle \mathit{ty}\rangle, 2, 3$
	$\langle \mathit{pat}\rangle \rightarrow \mathit{vid}\ \langle \mathit{tyop}\rangle\ \textbf{as}\ \langle \mathit{pat}\rangle \bullet, 0, 3$
	$\langle \mathit{pat}\rangle \rightarrow \bullet \langle \mathit{pat}\rangle \bullet : \langle \mathit{ty}\rangle, 0, 3$
	$\langle \mathit{pat}\rangle \rightarrow \mathit{vid}\ \langle \mathit{tyop}\rangle \bullet \textbf{as}\ \langle \mathit{pat}\rangle, 2, 3$
:	
	$\langle \mathit{pat}\rangle \rightarrow \langle \mathit{pat}\rangle : \bullet \langle \mathit{ty}\rangle, 2, 4$
	$\langle \mathit{pat}\rangle \rightarrow \langle \mathit{pat}\rangle : \bullet \langle \mathit{ty}\rangle, 0, 4$
int	
	$\langle \mathit{pat}\rangle \rightarrow \langle \mathit{pat}\rangle : \langle \mathit{ty}\rangle \bullet, 2, 5$
	$\langle \mathit{pat}\rangle \rightarrow \langle \mathit{pat}\rangle : \langle \mathit{ty}\rangle \bullet, 0, 5$
	$\langle \mathit{pat}\rangle \rightarrow \mathit{vid}\ \langle \mathit{tyop}\rangle\ \textbf{as}\ \langle \mathit{pat}\rangle \bullet, 0, 5$
	$\langle \mathit{pat}\rangle \rightarrow \langle \mathit{pat}\rangle \bullet : \langle \mathit{ty}\rangle, 0, 5$
	$\langle \mathit{pat}\rangle \rightarrow \langle \mathit{pat}\rangle \bullet : \langle \mathit{ty}\rangle, 2, 5$

3.3.2 Parse Forests

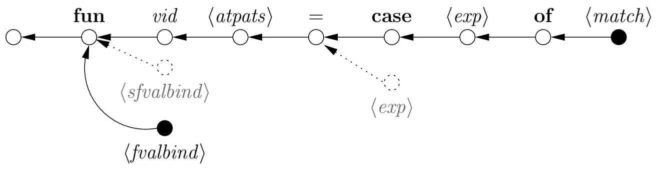
The result of a general parser might be a collection of parse trees—a *parse forest*—where a single parse tree was expected, hampering the reliability of

Table 3.3: Generalized LALR(1) parse of the Standard ML input from Example 3.5 on page 32.

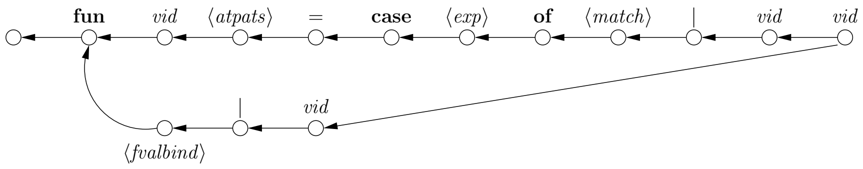
Let us start in the inconsistent state with a shift/reduce conflict: we performed the goto to ⟨*match*⟩, and at this point, we have a single stack.



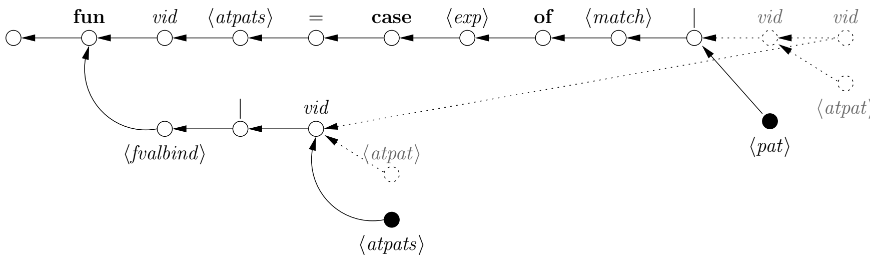
We need now to further perform all the possible reductions: the reduction in conflict creates an new GSS node for the goto on ⟨*exp*⟩. Owing to the possibility of a shift, we need to keep the nodes up to the ⟨*match*⟩ goto. The state in this node demands another reduction using rule ⟨*sfvalbind*⟩→*vid* ⟨*atpats*⟩ = ⟨*exp*⟩, and the previously created node is made inactive, while we create a new node for the goto of ⟨*sfvalbind*⟩. Again, the LALR(1) state in this new node demands yet another reduction, this time to ⟨*fvalbind*⟩, and the node is made inactive, while we add a new node. We have thus two different stack frontiers, filled in black in the following figure.



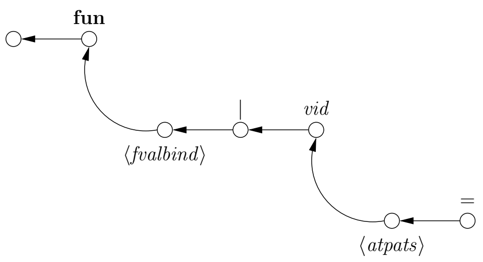
Both these nodes accept to shift the next tokens "|", *vid*, and *vid*. This last shift brings us to a stack with a single frontier node.



Performing all the possible reductions from this node separates again the two cases; in one, we end with a frontier node reached after a goto on ⟨*pat*⟩, whereas in the other, the frontier node is reached after a goto on ⟨*atpats*⟩.



Only the second of these two frontier nodes is compatible with the next shift of the equal sign "=". The other stack is discarded. The parser has now a single stack, as if it had guessed the right parsing action in the conflict state.



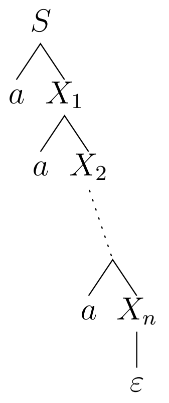


Figure 3.6: The set of parse trees for sentence a^n with respect to $\mathcal{G}_6$.

the computations that follow the parsing phase. With the spread of general parsing tools, the need to handle parse forests arose.

3.3.2.1 Shared Forests

The number of distinct parses for a single sentence can be exponential in the length of the sentence.[2] For instance, the following grammar by Wich (2005) allows exactly 2^n parse trees for a sentence a^n:

$$S \rightarrow aS \,|\, aA \,|\, \varepsilon, \;\; A \rightarrow aS \,|\, aA \,|\, \varepsilon. \qquad (\mathcal{G}_6)$$

Each parse tree for the sentence a^n has the form shown in Figure 3.6, with the $X_1, X_2, \ldots, X_n$ variables independently drawn from the set $\{S, A\}$, yielding 2^n different trees.

Such an explosive upper bound is not unrealistic, at least for natural languages: Moore (2004) reports a startling average number of 7.2×10^{27} different parses for sentences of 5.7 tokens on average, using a large coverage English grammar (Marcus et al., 1993).

Sharing In order to restrain this complexity, we could *share* the nodes of different trees in the parse forest that derive the same string from the same nonterminal: in the case of $\mathcal{G}_6$, we would obtain a single tree as in Figure 3.6, with each X_i variable replaced by two shared nodes for the trees rooted with S and A and deriving a^i. The shared forest now fits in a linear space.

Shared forests have in general a size bounded by $\mathcal{O}(n^{p+1})$ with n the length of the sentence and p the length of the longest rule rightpart in the

[2]In the case of a *cyclic* grammar, with a nonterminal A verifying $A \Rightarrow^+ A$, the number of trees can even be unbounded. The tabulation and shared forest representations have therefore to manage cycles in order to guarantee termination.

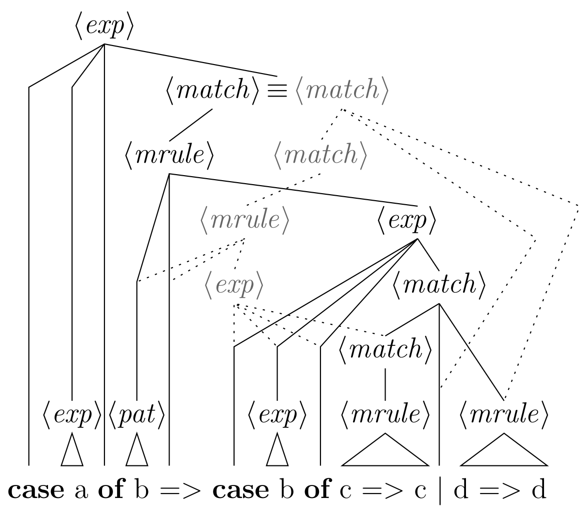


Figure 3.7: The shared parse forest for the input of Example 3.6.

grammar; one could further encode the forest in a binary form in order to obtain a $\mathcal{O}(n^3)$ bound (Billot and Lang, 1989). Figure 3.7 shows a shared forest for the ambiguity of Example 3.6, with a topmost $\langle match \rangle$ node that merges the two alternative interpretations of the input.

Grammar Forest A convenient way to generate a shared forest for a given grammar $\mathcal{G}$ and input string w is to generate the rules of a *grammar forest* $\mathcal{G}_w$ that derives only w, and furthermore whose parse trees are isomorphic to the parse trees of the original grammar $\mathcal{G}$.

Usually, the nonterminal symbols of $\mathcal{G}_w$ are then triples (A, i, j) such that nonterminal A could derive the portion of w between position i and position j. For instance, in the forest grammar of Figure 3.8 on the next page that describes the shared forest of Figure 3.7, the nonterminal $(\langle match \rangle, 3, 15)$ has two different rules corresponding to the shared $\langle match \rangle$ node in Figure 3.7.

Alternatively, the grammar forest can be encoded in binary form as well by adding more nonterminals.

3.3.2.2 Disambiguation Filters

Klint and Visser (1994) developed the general notion of *disambiguation filters* that reject some of the trees of the parse forest, with the hope of ending the selection process with a single tree. Such a mechanism is implemented in one form or in another in many GLR tools, including SDF (van den Brand et al., 2002), Elkhound (McPeak and Necula, 2004), and Bison (Donnely and Stallman, 2006).

$$\begin{aligned}
(\langle exp\rangle, 0, 15) &\rightarrow (\textbf{case}, 0, 1)\ (\langle exp\rangle, 1, 2)\ (\textbf{of}, 2, 3)\ (\langle match\rangle, 3, 15)\\
(\langle exp\rangle, 1, 2) &\Rightarrow^* (\text{a}, 1, 2)\\
(\langle match\rangle, 3, 15) &\rightarrow (\langle mrule\rangle, 3, 15)\\
(\langle match\rangle, 3, 15) &\rightarrow (\langle match\rangle, 3, 11)\ (|, 11, 12)\ (\langle mrule\rangle, 12, 15)\\
(\langle mrule\rangle, 3, 15) &\rightarrow (\langle pat\rangle, 3, 4)\ (\texttt{=>}, 4, 5)\ (\langle exp\rangle, 5, 15)\\
(\langle match\rangle, 3, 11) &\rightarrow (\langle mrule\rangle, 3, 11)\\
(\langle mrule\rangle, 12, 15) &\Rightarrow^* (\text{d}, 12, 13)\ (\texttt{=>}, 13, 14)\ (\text{d}, 14, 15)\\
(\langle pat\rangle, 3, 4) &\Rightarrow^* (\text{b}, 3, 4)\\
(\langle exp\rangle, 5, 15) &\rightarrow (\textbf{case}, 5, 6)\ (\langle exp\rangle, 6, 7)\ (\textbf{of}, 7, 8)\ (\langle match\rangle, 8, 15)\\
(\langle mrule\rangle, 3, 11) &\rightarrow (\langle pat\rangle, 3, 4)\ (\texttt{=>}, 4, 5)\ (\langle exp\rangle, 5, 11)\\
(\langle exp\rangle, 6, 7) &\Rightarrow^* (\text{b}, 6, 7)\\
(\langle match\rangle, 8, 15) &\rightarrow (\langle match\rangle, 8, 11)\ (|, 11, 12)\ (\langle mrule\rangle, 12, 15)\\
(\langle exp\rangle, 5, 11) &\rightarrow (\textbf{case}, 5, 6)\ (\langle exp\rangle, 6, 7)\ (\textbf{of}, 7, 8)\ (\langle match\rangle, 8, 11)\\
(\langle match\rangle, 8, 11) &\rightarrow (\langle mrule\rangle, 8, 11)\\
(\langle mrule\rangle, 8, 11) &\Rightarrow^* (\text{c}, 8, 9)\ (\texttt{=>}, 9, 10)\ (\text{c}, 10, 11)
\end{aligned}$$

Figure 3.8: The grammar forest that describes the shared forest of Figure 3.7 on the facing page.

Unexpected ambiguities are acute with GLR parsers that compute semantic attributes as they reduce partial trees. The GLR implementations of GNU Bison (Donnely and Stallman, 2006) and of Elkhound (McPeak and Necula, 2004) are in this situation. Attribute values are synthesized for each parse tree node, and in a situation like the one depicted in Figure 3.7 on the facing page, the values obtained for the two alternatives of a shared node have to be merged into a single value for the shared node as a whole. The user of these tools should thus provide a *merge* function that returns the value of the shared node from the attributes of its competing alternatives.

Failure to provide a merge function where it is needed forces the parser to choose arbitrarily between the possibilities, which is highly unsafe. Another line of action is to abort parsing with a message exhibiting the ambiguity; this can be set with an option in Elkhound, and it is the behavior of Bison.

Example 3.7. Let us suppose that the user has found out the ambiguity of Example 3.6, and is using a disambiguation filter (in the form of a merge function in Bison or Elkhound) that discards the dotted alternative of Figure 3.7, leaving only the correct parse according to the Standard ML definition. A simple way to achieve this is to check whether we are reducing using rule $\langle match\rangle{\rightarrow}\langle match\rangle'|'\langle mrule\rangle$ or with rule $\langle match\rangle{\rightarrow}\langle mrule\rangle$. Filters of this variety are quite common, and are given a specific `dprec` directive in Bison, also corresponding to the `prefer` and `avoid` filters in SDF2 (van den Brand et al., 2002).

The above solution is unfortunately unable to deal with yet another form of ambiguity with $\langle match \rangle$, namely the ambiguity encountered with the input:

case a **of** b => b | c => **case** c **of** d => d | e => e

Indeed, with this input, the two shared $\langle match \rangle$ nodes are obtained through reductions using the same rule $\langle match\rangle \rightarrow \langle match\rangle '|' \langle mrule\rangle$. Had we trusted our filter to handle all the ambiguities, we would be running our parser under a sword of Damocles. $\square$

This last example shows that a precise knowledge of the ambiguous cases is needed for the development of a reliable GLR parser. While the problem of detecting ambiguities is undecidable, conservative answers could point developers in the right direction.

Grammar Approximations 4

Be it for advanced deterministic parser generation or for ambiguity detection, a static analysis of the context-free grammar at hand is involved. Both issues are rather challenging compared to e.g. the simple problem of identifying useless symbols in the grammar: they require us to *approximate* the grammar into a more convenient, finite state, representation. We describe in this chapter a theoretical framework for grammar approximations, a groundwork for the more practical considerations of the following chapters.

Such approximations should somehow preserve the structure of the original grammar, and thus we start this chapter by considering the derivation trees of context-free grammars (Section 4.1). Then, we define *position graphs* as labeled transition systems (LTS) that describe depth-first traversals inside these trees (Section 4.1.2), and consider their quotients, the *position automata*, for our grammar abstractions (Section 4.2). The investigation of the properties of position automata brings us a bit farther than really needed in the next chapters, but the questions it raises are not wholly abstruse: we describe next a simple application of position automata that employs some of the advanced results of the chapter: the recognition of parse trees by means of finite state automata (Section 4.3). We end the chapter by examining a few related models in Section 4.4.

4.1 Derivation Trees

Chomsky (1956) emphasized from the very beginning that phrase structure grammars describe the structure of the language rather than merely its sentences. The transformations he defined to yield a natural language in its full complexity act on derivation trees of a context-free backbone (Chomsky, 1961). This notion of *tree language* generated by a context-free grammar, i.e. its set of derivation trees, was later studied in its own right by Thatcher (1967), and is still an active research area, especially for XML applications (Neven, 2002; Murata et al., 2005; Schwentick, 2007). Our own interest in the subject stems from the two following facts:

1. the derivation trees are the desired result of a parsing process,

2. a grammar is ambiguous if and only if it derives more than one tree with the same yield.

4.1.1 Bracketed Grammars

Considering the classical sentence "She saw the man with a telescope.", a simple English grammar $\mathcal{G}_7 = \langle N, T, P, S\rangle$[1] that generates this sentence could have the rules in P

$$\begin{aligned} S &\xrightarrow{2} NP\ \ VP\\ NP &\xrightarrow{3} d\ n\\ NP &\xrightarrow{4} pn\\ NP &\xrightarrow{5} NP\ \ PP\\ VP &\xrightarrow{6} v\ NP\\ VP &\xrightarrow{7} VP\ \ PP\\ PP &\xrightarrow{8} pr\ NP, \end{aligned} \tag{$\mathcal{G}_7$}$$

where the nonterminals in N, namely S, NP, VP, and PP, stand respectively for a sentence, a noun phrase, a verb phrase, and a preposition phrase, whereas the terminals in T, namely d, n, v, pn, and pr, denote determinants, nouns, verbs, pronouns, and prepositions. Two interpretations of our sentence are possible: the preposition phrase "with a telescope" can be associated to "saw" or to "the man", and thus there are two derivation trees, shown in Figure 4.1 on the facing page.

The purpose of a parser is to retrieve the explicit structure of this sentence. In the same way, an ambiguity verification algorithm should detect that two different structures seem possible for it.

Tree structures are easier to handle in a flat representation, where the structural information is described by a *bracketing* (Ginsburg and Harrison, 1967): each rule $i = A{\rightarrow}\alpha$ (also denoted $A\xrightarrow{i}\alpha$) of the grammar is surrounded by a pair of opening and closing brackets d_i and r_i.

[1]All our grammars $\mathcal{G}$ are context-free, reduced, and of form $\langle N, T, P, S\rangle$. Our notational conventions are summed up in Section A.1.4 on page 165.

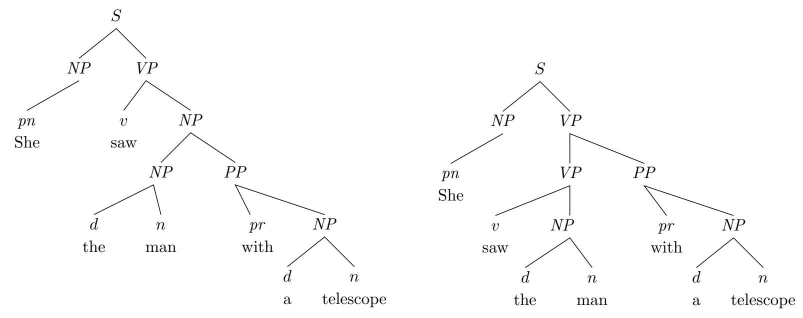


Figure 4.1: Two trees yielding the sentence “She saw the man with a telescope.” with $\mathcal{G}_7$.

Definition 4.1. The *bracketed grammar* of a grammar $\mathcal{G}$ is the context-free grammar $\mathcal{G}_b = \langle N, T_b, P_b, S\rangle$ where $T_b = T \cup T_d \cup T_r$ with $T_d = \{d_i \mid i \in P\}$ and $T_r = \{r_i \mid i \in P\}$, and $P_b = \{A \xrightarrow{i} d_i \alpha r_i \mid A \xrightarrow{i} \alpha \in P\}$. We denote derivations in $\mathcal{G}_b$ by $\Rightarrow_b$.

We define the homomorphism h from V_b^* to V^* by $h(d_i) = \varepsilon$ and $h(r_i) = \varepsilon$ for all i in P, and $h(X) = X$ otherwise, and denote by δ_b (resp. w_b) a string in V_b^* (resp. T_b^*) such that $h(\delta_b) = \delta$ in V^* (resp. $h(w_b) = w$ in T^*).

Using the rule indices as subscripts for the brackets, the two trees of Figure 4.1 are represented by the following two sentences of the augmented bracketed grammar for $\mathcal{G}_7$:[2]

$$d_1\, d_2\, d_4\, pn\, r_4\, d_6\, v\, d_5\, d_3\, d\, n\, r_3\, d_8\, pr\, d_3\, d\, n\, r_3\, r_8\, r_5\, r_6\, r_2\, r_1 \tag{4.1}$$

$$d_1\, d_2\, d_4\, pn\, r_4\, d_7\, d_6\, v\, d_3\, d\, n\, r_3\, r_6\, d_8\, pr\, d_3\, d\, n\, r_3\, r_8\, r_7\, r_2\, r_1. \tag{4.2}$$

The existence of an ambiguity can be verified by checking that the image of these two different sentences by h is the same string $pn\, v\, d\, n\, pr\, d\, n$.

Covers As a pleasant consequence of the unambiguity of bracketed grammars—they are LL(1)—, the set of strings in V_b^* that derives a given string δ_b in V_b^* has a minimal element $\widehat{d_b}$ with respect to the order induced by $\Rightarrow_b^*$. We call this minimal element a *cover* of δ_b, defined by

$$\widehat{\delta_b} = \min_{\Rightarrow_b^*}\{\gamma_b \mid \gamma_b \Rightarrow_b^* \delta_b\}. \tag{4.3}$$

For instance, in $\mathcal{G}_7$, the minimal cover of $d_5\, NP\, d_8\, pr\, d_3\, d\, n\, r_3\, r_8$ (note the unmatched d_5 symbol) is $d_5\, NP\, PP$: $d_5\, NP\, PP \Rightarrow_b^* d_5\, NP\, d_8\, pr\, d_3\, d\, n\, r_3\, r_8$ and there does not exist any string γ in V_b^* such that $\gamma \Rightarrow_b^* d_5\, NP\, PP$ holds.

[2]In the case of bracketed grammars, the *augmented* bracketed grammar $\mathcal{G}'_b$ augments the sets T_d and T_r with the symbols d_1 and r_1 respectively, and augments the set of rules with the rule $S' \rightarrow d_1 S r_1$. Symbols d_1 and r_1 act as beginning and end of sentence markers.

A string δ in V^* might have many different bracketings δ_b, each with its cover; accordingly, we define the set of covers of a string δ as

$$\text{Covers}(\delta) = \{\widehat{\delta_b} \mid h(\delta_b) = \delta\}, \tag{4.4}$$

and denote by $\widehat{\delta}$ an element of Covers(δ). The following proposition motivates the introduction of the previous definitions:

Proposition 4.2. *Let $\mathcal{G} = \langle N, T, P, S\rangle$ be a context-free grammar. A string w in T^* belongs to $\mathcal{L}(\mathcal{G})$ if and only if S belongs to* Covers(w).

Alternate Definitions Our definition of bracketed grammars diverges from the standard one (Ginsburg and Harrison, 1967) on one point: we enforce distinct opening and closing brackets for each rule, whereas a single closing symbol would suffice. As a matter of fact, a single opening symbol and a single closing one would have been enough in order to describe the tree structure, i.e. we could have considered *parenthesis grammars* (McNaughton, 1967) instead. Nonetheless, the idea behind our brackets is that they also represent parsing decisions: the d_i symbols represent *predictions* in top-down or Earley parsing, whereas the r_i ones represent *reductions* in bottom-up parsing and *completions* in Earley parsing. Making these symbols explicit allows us to preserve this behavior throughout our approximations; we will devote more space to this topic in Section 4.2.2.

4.1.2 Position Graphs

Let us consider again the two sentences (4.1) and (4.2) and how we can read them step by step on the trees of Figure 4.1 on the preceding page. This process is akin to a left to right walk in the trees, between *positions* to the immediate left or immediate right of a tree node. For instance, the dot in

$$d_1\, d_2\, d_4\, pn\, r_4\, d_6\, v\, d_5\, d_3\, d\, n\, r_3 \cdot d_8\, pr\, d_3\, d\, n\, r_3\, r_8\, r_5\, r_6\, r_2\, r_1 \tag{4.5}$$

identifies the position between *NP* and *PP* in the middle of the left tree of Figure 4.1 on the previous page.

Positions Although a dotted sentence of $\mathcal{G}_b$ like (4.5) suffices to identify a unique position in the derivation tree for that sentence, it is convenient to know that this position is immediately surrounded by the *NP* and *PP* symbols. We therefore denote by $x_b d_i (\genfrac{}{}{0pt}{}{\alpha}{u_b} \cdot \genfrac{}{}{0pt}{}{\alpha'}{u'_b}) r_i x'_b$ the position identified by $x_b d_i u_b \cdot u'_b r_i x'_b$ such that the derivations

$$S' \Rightarrow_b^* x_b A x'_b \overset{i}{\Rightarrow}_b x_b d_i \alpha\alpha' r_i x'_b,\ \alpha \Rightarrow_b^* u_b \text{ and } \alpha' \Rightarrow_b^* u'_b \tag{4.6}$$

hold in $\mathcal{G}'_b$. The strings α and α' in V^* are covers of u_b and u'_b. Likewise, it is worth noting that $\widehat{x}_b$ is in $d_1(V \cup T_d)^*$ and $\widehat{x}'_b$ in $(V \cup T_r)^* r_1$.

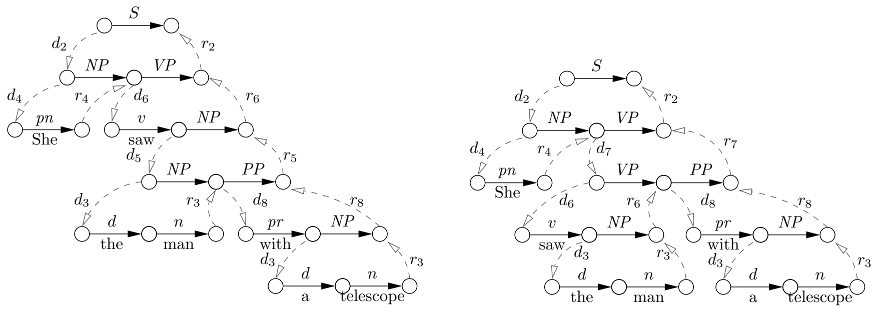


Figure 4.2: The two subgraphs of the position graph of $\mathcal{G}_7$ corresponding to the two trees of Figure 4.1 on page 49.

Using this notation, the position identified by (4.5) is denoted by

$$d_1\,d_2\,d_4\,pn\,r_4\,d_6\,v\,d_5({}_{\overset{NP}{d_3\,d\,n\,r_3}}\bullet{}_{\overset{PP}{d_8\,pr\,d_3\,d\,n\,r_3\,r_8}})r_5\,r_6\,r_2\,r_1. \tag{4.7}$$

The parsing literature classically employs *items* to identify positions in grammars; for instance, $[NP\overset{5}{\rightarrow}NP\bullet PP]$ is the LR(0) item (Knuth, 1965) corresponding to position (4.7). There is a direct connection between these two notions: items are equivalence classes of positions (Section 4.2).

Transitions Transitions from one position to the other can then be performed upon reading the node label, or upon deriving from this node, or upon returning from such a derivation. We have thus three types of transitions: symbol transitions $\overset{X}{\rightarrowtail}$, derivation transitions $\overset{d_i}{\rightarrowtail}$, and reduction transitions $\overset{r_i}{\rightarrowtail}$. The set of all these positions in all parse trees along with the transition relation is a *position graph.* Figure 4.2 presents two portions of the position graph for $\mathcal{G}_7$; the position identified by the dot in (4.5) is now a vertex in the left graph.

For instances of labeled transition relations, we can consider again position (4.7), and see in Figure 4.2 that it is related by $\overset{d_8}{\rightarrowtail}$ to the position

$$d_1\,d_2\,d_4\,pn\,r_4\,d_6\,v\,d_5\,d_3\,d\,n\,r_3\,d_8(\,\bullet{}_{\overset{pr}{pr}\,\overset{NP}{d_3\,d\,n\,r_3}})r_8\,r_5\,r_6\,r_2\,r_1, \tag{4.8}$$

and by $\overset{PP}{\rightarrowtail}$ to the position

$$d_1\,d_2\,d_4\,pn\,r_4\,d_6\,v\,d_5({}_{\overset{NP}{d_3\,d\,n\,r_3}\,\overset{PP}{d_8\,pr\,d_3\,d\,n\,r_3\,r_8}}\bullet)r_5\,r_6\,r_2\,r_1. \tag{4.9}$$

The latter is related in turn to the position

$$d_1\,d_2\,d_4\,pn\,r_4\,d_6({}_{\overset{v}{v}\,\overset{NP}{d_5\,d_3\,d\,n\,r_3\,d_8\,pr\,d_3\,d\,n\,r_3\,r_8\,r_5}}\bullet)r_6\,r_2\,r_1 \tag{4.10}$$

by $\overset{r_5}{\rightarrowtail}$.

Definition 4.3. The *position graph* $\Gamma = \langle \mathcal{N}, \rightarrowtail \rangle$ of a grammar $\mathcal{G}$ associates the set $\mathcal{N}$ of positions with the relation $\rightarrowtail$ labeled by elements of V_b, defined by

$$x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b}\bullet\genfrac{}{}{0pt}{}{X\alpha'}{v_b u'_b})r_i x'_b \overset{X}{\rightarrowtail} x_b d_i(\genfrac{}{}{0pt}{}{\alpha X}{u_b v_b}\bullet\genfrac{}{}{0pt}{}{\alpha'}{u'_b})r_i x'_b \qquad \text{iff } X \in V, X \Rightarrow_b^* v_b,$$

$$x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b}\bullet\genfrac{}{}{0pt}{}{B\alpha'}{v_b u'_b})r_i x'_b \overset{d_j}{\rightarrowtail} x_b d_i u_b d_j(\bullet\genfrac{}{}{0pt}{}{\beta}{v_b})r_j u'_b r_i x'_b \qquad \text{iff } B \xrightarrow{j} \beta \text{ and } \beta \Rightarrow_b^* v_b,$$

$$x_b d_i u_b d_j(\genfrac{}{}{0pt}{}{\beta}{v_b}\bullet)r_j u'_b r_i x'_b \overset{r_j}{\rightarrowtail} x_b d_i(\genfrac{}{}{0pt}{}{\alpha B}{u_b v_b}\bullet\genfrac{}{}{0pt}{}{\alpha'}{u'_b})r_i x'_b$$

$$\text{iff } B \xrightarrow{j} \beta, \alpha \Rightarrow_b^* u_b \text{ and } \alpha' \Rightarrow_b^* u'_b.$$

We label paths in Γ by the sequences of labels on the individual transitions.

Let us denote the two sets of positions at the beginning and end of the sentences by $\mu_s = \{d_1(\bullet\genfrac{}{}{0pt}{}{S}{w_b})r_1 \mid S \Rightarrow_b^* w_b\}$ and $\mu_f = \{d_1(\genfrac{}{}{0pt}{}{S}{w_b}\bullet)r_1 \mid S \Rightarrow_b^* w_b\}$. For each sentence w_b of $\mathcal{G}_b$, a ν_s in μ_s is related to a ν_f in μ_f by $\nu_s \overset{S}{\rightarrowtail} \nu_f$. Our position graph is thus an infinite collection of subgraphs, one for each derivation tree of $\mathcal{G}$, each with its own start and final position, like the two subgraphs shown in Figure 4.2 on the previous page.

Paths in Derivation Trees Positions being defined with respect to a unique derivation tree, paths between positions can be considered within this tree. This gives rise to some simple results on the relations between the various paths between two positions of the position graph and the possible derivations in the original grammar.

Let us consider in the left tree of Figure 4.2 on the preceding page the two positions (4.11) defined as follows

$$d_1\, d_2\, d_4\, pn\, r_4\, d_6\, v\, d_5(\bullet\genfrac{}{}{0pt}{}{NP\ PP}{d_3\, d\, n\, r_3\, d_8\, pr\, d_3\, d\, n\, r_3\, r_8})r_5\, r_6\, r_2\, r_1 \tag{4.11}$$

and (4.9) defined earlier. Lemma 4.4 shows that the existence of the path

$$NP\ d_8\ pr\ NP\ r_8 \tag{4.12}$$

between these positions and of the derivation $NP\ PP \Rightarrow_b^* NP\ d_8 pr\ NP\ r_8$ implies that $NP\ PP$ is also a path between positions (4.11) and (4.9).

Lemma 4.4. *Let ν and ν' be positions in $\mathcal{N}$, and δ_b and γ_b be bracketed strings in V_b^* with $\nu \overset{\delta_b}{\rightarrowtail} \nu'$. If $\gamma_b \Rightarrow_b^* \delta_b$ in $\mathcal{G}_b$, then $\nu \overset{\gamma_b}{\rightarrowtail} \nu'$.*

Proof. Suppose $\gamma_b \Rightarrow_b^n \delta_b$; we proceed by induction on n the number of individual derivation steps. If $n = 0$, then $\delta_b = \gamma_b$ and the property holds. Let now $\gamma_b = \rho_b A \sigma_b \overset{i}{\Rightarrow}_b \rho_b d_i \alpha r_i \sigma_b \Rightarrow_b^{n-1} \delta_b$; using the induction hypothesis, $\nu \overset{\rho_b}{\rightarrowtail} \nu_1 \overset{d_i}{\rightarrowtail} \nu_3 \overset{\alpha}{\rightarrowtail} \nu_4 \overset{r_i}{\rightarrowtail} \nu_2 \overset{\sigma_b}{\rightarrowtail} \nu'$. By Definition 4.3, $\nu_1 \overset{A}{\rightarrowtail} \nu_2$ and thus $\nu \overset{\gamma_b}{\rightarrowtail} \nu'$. □

Conversely, note that the existence of path (4.12) and of the derivation $NP\ d_8pr\ NP\ r_8 \Rightarrow_b^* d_4\ pn\ r_4\ d_8pr\ NP\ r_8$ does not imply the existence of such a path $d_4\ pn\ r_4\ d_8pr\ NP\ r_8$ between positions (4.11) and (4.9), as can be checked on Figure 4.2 on page 51. Neither does the existence of the path $d_3\ d\ n\ r_3\ PP$ between our two positions imply a derivation relation with (4.12). The only conclusive case appears if one considers a *terminal* path between two positions, as demonstrated in the following lemma.

Lemma 4.5. *Let ν and ν' be positions in $\mathcal{N}$, δ_b be a bracketed string in V_b^*, and w_b a terminal bracketed string in T_b^* such that $\nu \xrightarrow{w_b} \nu'$. The path $\nu \xrightarrow{\delta_b} \nu'$ holds in Γ if and only if $\delta_b \Rightarrow_b^* w_b$.*

Proof. Using path $\nu \xrightarrow{w_b} \nu'$ and $\delta_b \Rightarrow_b^* w_b$ with Lemma 4.4, we know that the path $\nu \xrightarrow{\delta_b} \nu'$ exists in Γ. Conversely, if we suppose that $\nu \xrightarrow{\delta_b} \nu'$ holds in Γ, then it is easy to check inductively that $\delta_b \Rightarrow_b^* w_b$. □

4.2 Position Automata

In order to perform static checks on our grammar, we are going to approximate its context-free language by a regular language. This shift brings us to a domain where most problems (equivalence, containment, ambiguity, ...) become decidable.

In relation with the preceding section on position graphs, we want a finite structure instead of our infinite position graph. The approximation is the result of an equivalence relation applied to the positions of the graph, such that the equivalence classes become the states of a position automaton (PA), an operation known as *quotienting*. If the chosen equivalence relation is of finite index, then our position automaton is a *finite state automaton* (FSA, see Definition A.8 on page 166).

4.2.1 Quotients of Position Graphs

Definition 4.6. The (nondeterministic) *position automaton* (PA) $\Gamma/\equiv$ of a grammar $\mathcal{G}$ using the equivalence relation $\equiv$ on $\mathcal{N}$ is a LTS $\langle Q, V_b, R, Q_s, Q_f\rangle$ where

- Q, the *state alphabet*, is the set $\mathcal{N}/\equiv\ = \{[\nu]_\equiv \mid \nu \in \mathcal{N}\}$ of non-empty equivalence classes $[\nu]_\equiv$ over $\mathcal{N}$ modulo the equivalence relation $\equiv$,
- V_b is the *input alphabet*,
- R in $Q\,V_b \times Q$ is the *set of rules*

$$\rightarrowtail/\equiv\ = \{q\chi \vdash q' \mid \exists \nu \in q, \exists \nu' \in q', \nu \xrightarrow{\chi} \nu'\},$$

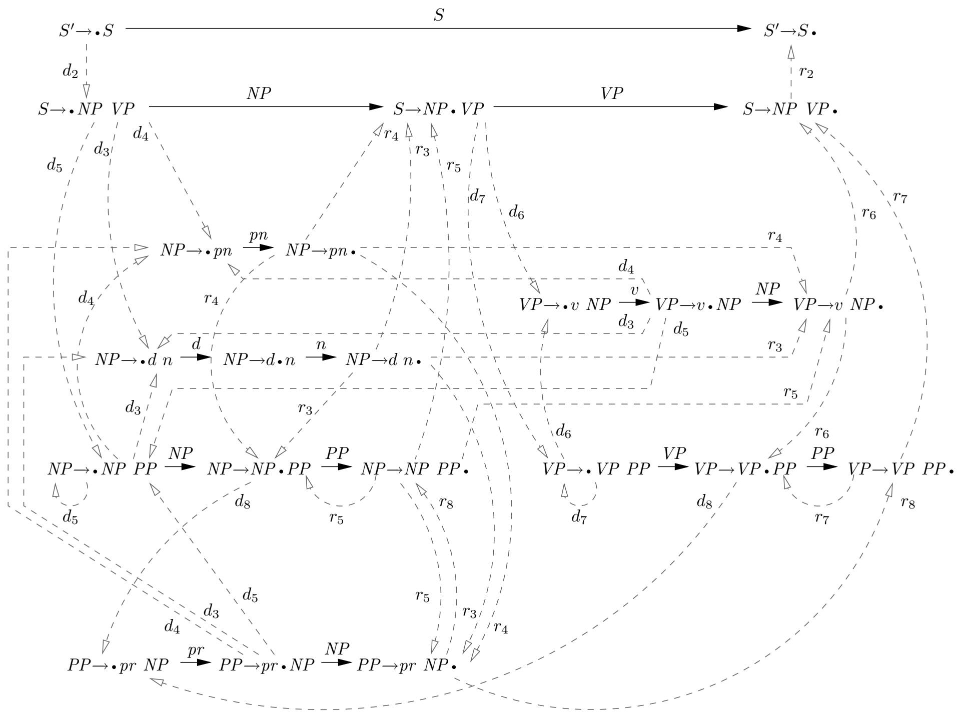


Figure 4.3: The nondeterministic position automaton for $\mathcal{G}_7$ using item_0. Dashed arrows denote d_i and r_i transitions in the automaton, whereas plain arrows denote transitions on symbols in V.

- $Q_s = \mu_s/{\equiv} = \{[\nu_s]_\equiv \mid \nu_s \in \mu_s\}$ is the set of *initial states*, and
- $Q_f = \mu_f/{\equiv} = \{[\nu_f]_\equiv \mid \nu_f \in \mu_f\}$ is the set of *final states.*

4.2.1.1 The item_0 Example

For instance, an equivalence relation that results in a FSA similar to a nondeterministic LR(0) automaton (Hunt III et al., 1974, 1975)—the difference being the presence of the r_i transitions—is item_0 defined by

$$x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b}\bullet\genfrac{}{}{0pt}{}{\alpha'}{u'_b})r_i x'_b \;\; \mathsf{item}_0 \;\; y_b d_j(\genfrac{}{}{0pt}{}{\beta}{v_b}\bullet\genfrac{}{}{0pt}{}{\beta'}{v'_b})r_j y'_b \text{ iff } i = j \text{ and } \alpha' = \beta'. \tag{4.13}$$

The equivalence classes in $\mathcal{N}/\,\mathsf{item}_0$ are the LR(0) items. Figure 4.3 presents the nondeterministic automaton for $\mathcal{G}_7$ resulting from the application of item_0 as equivalence relation. Our position (4.7) is now in the equivalence class represented by the state labeled by $NP{\rightarrow}NP\bullet PP$ in this figure. One can further see that position (4.8) in the left tree of Figure 4.2

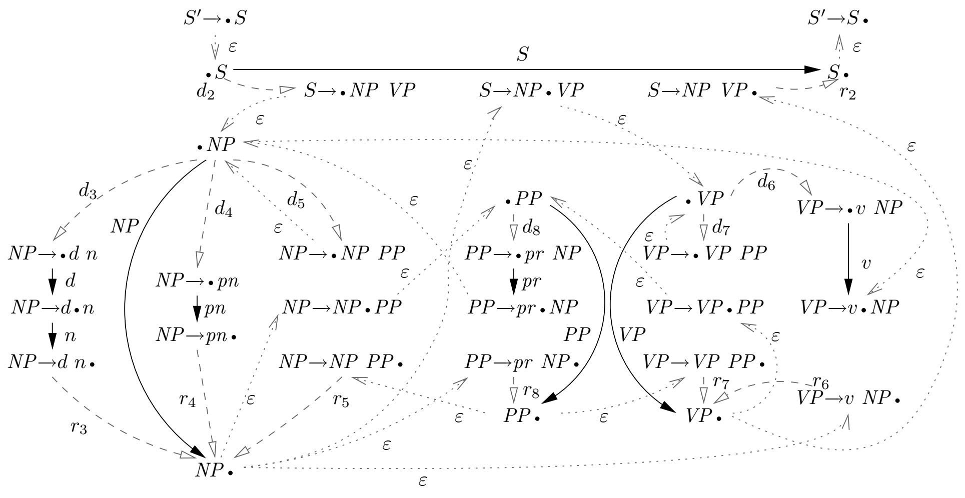


Figure 4.4: An alternate position automaton for $\mathcal{G}_7$ using item_0; dotted arrows denote the additional ε transitions.

and position

$$d_1\, d_2\, d_4\, pn\, r_4\, d_7\, d_6\, v\, d_3\, d\, n\, r_3\, r_6\, d_8(\underset{pr\, d_3\, d\, n\, r_3}{\overset{pr\ NP}{\bullet}})r_8\, r_7\, r_2\, r_1 \tag{4.14}$$

in the right one are equivalent by item_0, and thus both belong to the equivalence class labeled by $PP{\rightarrow}\bullet pr\ NP$ in Figure 4.3 on the facing page.

Example 4.7. The position automaton Γ/item_0 for grammar $\mathcal{G}$ is then constructed with

- state alphabet $Q = \mathcal{N}/\mathsf{item}_0 = \{[A{\rightarrow}\alpha\bullet\alpha'] \mid A{\rightarrow}\alpha\alpha' \in P\}$,
- input alphabet V_b,
- set of rules R

$$\begin{aligned}&\{[A{\rightarrow}\alpha\bullet X\alpha']X \vdash [A{\rightarrow}\alpha X\bullet\alpha'] \mid A{\rightarrow}\alpha X\alpha' \in P\}\\ \cup&\{[A{\rightarrow}\alpha\bullet B\alpha']d_i \vdash [B{\rightarrow}\bullet\beta] \mid i = B{\rightarrow}\beta \in P, A{\rightarrow}\alpha B\alpha' \in P\}\\ \cup&\{[B{\rightarrow}\beta\bullet]r_i \vdash [A{\rightarrow}\alpha B\bullet\alpha'] \mid i = B{\rightarrow}\beta \in P, A{\rightarrow}\alpha B\alpha' \in P\},\end{aligned}$$

- sets of initial and final states $Q_s = \mu_s/\mathsf{item}_0 = \{[S'{\rightarrow}\bullet S]\}$ and $Q_f = \mu_f/\mathsf{item}_0 = \{[S'{\rightarrow}S\bullet]\}$ respectively. □

Size Optimization Due to the wide use of items in parsing constructions, item_0 is of particular interest, and even deserves a glance at a practical optimization: A more compact construction for item_0 position automata adds intermediate states of form $[\bullet A]$ and $[A\bullet]$ to reduce the overall number of transitions. We will further discuss the topic of the size of position automata in Section 4.2.1.3 on page 58. Figure 4.4 shows the resulting position automaton for $\mathcal{G}_7$.

Example 4.8. We construct an alternate position automaton $\Gamma/\mathsf{item}_0{}'$ for grammar $\mathcal{G}$ by redefining

- the state alphabet to $Q = \mathcal{N}/\mathsf{item}_0 \cup \{[\bullet A] \mid A \in N\} \cup \{[A\bullet] \mid A \in N\}$, and
- the set of rules R to

$$\begin{aligned}
&\{[A{\rightarrow}\alpha\bullet X\alpha']X \vdash [A{\rightarrow}\alpha X\bullet\alpha'] \mid A{\rightarrow}\alpha X\alpha' \in P\}\\
\cup&\{[A{\rightarrow}\alpha\bullet B\alpha']\varepsilon \vdash [\bullet B] \mid B \in N, A{\rightarrow}\alpha B\alpha' \in P\}\\
\cup&\{[\bullet B]d_i \vdash [B{\rightarrow}\bullet\beta] \mid i = B{\rightarrow}\beta \in P\}\\
\cup&\{[B{\rightarrow}\beta\bullet]r_i \vdash [B\bullet] \mid i = B{\rightarrow}\beta \in P\}\\
\cup&\{[B\bullet]\varepsilon \vdash [A{\rightarrow}\alpha B\bullet\alpha'] \mid B \in N, A{\rightarrow}\alpha B\alpha' \in P\}.
\end{aligned}$$

The sets V_b, Q_s, and Q_f remain unchanged. □

The number of transitions in this variant construction is now bounded by $\mathcal{O}(|\mathcal{G}|)$. The equivalence of the two constructions is straightforward, and the variant construction is the one typically used for nondeterministic LR automata (Hunt III et al., 1974; Grune and Jacobs, 2007) or LC parsers (Sikkel, 1997).

4.2.1.2 Languages Recognized by Position Automata

Position automata are in general labeled transition systems, but when their number of states is finite—or equivalently when the position equivalence is of finite index—, they end up being finite automata, and thus fulfill our purpose of approximating our context-free grammar by a regular language.

We study here some of the properties of our approximated language, which hold in both the finite and the infinite case. We will further consider the classes of tree languages and nested word languages that position automata define in Section 4.3.1 on page 65.

Regular Superlanguage We denote by $\models$ the relation between configurations of a LTS, such that $qaw \models q'w$ if and only if there exists a rule $qa \vdash q'$ in R.

Lemma 4.9. *Let $\Gamma = \langle\mathcal{N}, \rightarrowtail\rangle$ be a position graph and $\Gamma/{\equiv}$ its quotient by $\equiv$. Let ν, ν' be positions in $\mathcal{N}$ and δ_b a bracketed string in V_b^*. If $\nu \overset{\delta_b}{\rightarrowtail} \nu'$, then $[\nu]_\equiv\delta_b \models^* [\nu']_\equiv$.*

Proof. Straightforward induction on the number of individual steps in $\nu \overset{\delta_b}{\rightarrowtail} \nu'$. □

We then obtain the counterpart of Lemma 4.4 for position automata, with the restriction that the paths under study should be valid, i.e. they should be possible between grammar positions.

Corollary 4.10. *Let $\mathcal{G}$ be a grammar, $\mathcal{G}_b$ its bracketed grammar, and $\Gamma/\!\equiv\, = \langle Q, V_b, R, Q_s, Q_f \rangle$ its position automaton using $\equiv$.*

Let ν and ν' be positions in $\mathcal{N}$, and δ_b and γ_b be bracketed strings in V_b^* with $\nu \overset{\delta_b}{\rightarrowtail} \nu'$. If $\gamma_b \Rightarrow_b^* \delta_b$ in $\mathcal{G}_b$, then $[\nu]_\equiv \gamma_b \models^* [\nu']_\equiv$. □

Let us recall that the language recognized by a LTS $\mathcal{A} = \langle Q, \Sigma, R, Q_s, Q_f \rangle$ is $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \exists q_s \in Q_s, \exists q_f \in Q_f, q_s w \models^* q_f\}$. In the finite case, $\mathcal{L}(\Gamma/\!\equiv)$ is trivially a regular word language. We study how this language can be related to the original grammar.

Theorem 4.11. *Let $\mathcal{G}$ be a grammar, $\mathcal{G}_b$ its bracketed grammar, and $\Gamma/\!\equiv$ its position automaton using $\equiv$. The language of $\mathcal{G}_b$ is included in the terminal language recognized by $\Gamma/\!\equiv$, i.e. $\mathcal{L}(\mathcal{G}_b) \subseteq \mathcal{L}(\Gamma/\!\equiv) \cap T_b^*$.*

Proof. Let w_b be a sentence of $\mathcal{G}_b$. We consider the positions $\nu_s = d_1(\bullet \overset{S}{\underset{w_b}{}})r_1$ and $\nu_f = d_1(\overset{S}{\underset{w_b}{}} \bullet)r_1$ related by $\nu_s \overset{w_b}{\rightarrowtail} \nu_f$. By Lemma 4.9, $[\nu_s]_\equiv w_b \models^* [\nu_f]_\equiv$. By Definition 4.6, $[\nu_s]_\equiv$ and $[\nu_f]_\equiv$ are in Q_s and Q_f respectively, thus w_b is accepted by $\Gamma/\!\equiv$, i.e. w_b is in $\mathcal{L}(\Gamma/\!\equiv) \cap T_b^*$. □

As witnessed when processing the incorrect input

$$d_2\ d_4\ pn\ r_4\ d_6\ v\ d_3\ d\ n\ r_3\ r_6\ d_8\ pr\ d_3\ d\ n\ r_3\ \ r_8\ r_7\ r_2, \tag{4.15}$$

with the position automaton of Figure 4.3 on page 54, there can be strings in $\mathcal{L}(\Gamma/\!\equiv) \cap T_b^*$ that are not in $\mathcal{L}(\mathcal{G}_b)$, i.e. this inclusion is proper in general.

Sentential Forms Let us denote the set of *sentential forms* generated by a (reduced) context-free grammar $\mathcal{G}$ by $\mathcal{SF}(\mathcal{G}) = \{\alpha \in V^* \mid S \Rightarrow^* \alpha\}$. By Corollary 4.10 and Theorem 4.11, the language recognized by a position automaton contains the set of sentential forms generated by the corresponding bracketed grammar.

Corollary 4.12. *Let $\mathcal{G}$ be a context-free grammar and $\Gamma/\!\equiv$ its position automaton using $\equiv$. The set of sentential forms generated by $\mathcal{G}_b$ is included in the language recognized by $\Gamma/\!\equiv$, i.e. $\mathcal{SF}(\mathcal{G}_b) \subseteq \mathcal{L}(\Gamma/\!\equiv)$.* □

4.2.1.3 Lattice of Equivalence Relations

As a pleasant consequence of employing equivalence relations and quotients for position automata, many simple but powerful algebraic results can be exercised to prove some properties of position automata.

For instance, an equivalence relation on $\mathcal{N}$ is a subset of $\mathcal{N} \times \mathcal{N}$, and as such can be compared to other equivalence relations via set inclusion $\subseteq$. The set $\mathrm{Eq}(\mathcal{N})$ of all the equivalence relations on $\mathcal{N}$ is then a *complete lattice* when one employs set inclusion as partial order (Ore, 1942; Grätzer, 1978).

Bounds The largest (and coarsest) element in $\mathrm{Eq}(\mathcal{N})$, denoted by $\top$, always results in a single equivalence class, while the smallest (and finest) equivalence relation, denoted by $\bot$, is the identity on $\mathcal{N}$:

$$x_b d_i ({}^{\alpha}_{u_b} \bullet {}^{\alpha'}_{u'_b}) r_i x'_b \;\top\; y_b d_j ({}^{\beta}_{v_b} \bullet {}^{\beta'}_{v'_b}) r_j y'_b \tag{4.16}$$

$$x_b d_i ({}^{\alpha}_{u_b} \bullet {}^{\alpha'}_{u'_b}) r_i x'_b \;\bot\; y_b d_j ({}^{\beta}_{v_b} \bullet {}^{\beta'}_{v'_b}) r_j y'_b \text{ iff } x_b d_i u_b \bullet u'_b r_i x'_b = y_b d_j v_b \bullet v'_b r_j y'_b. \tag{4.17}$$

Of course, for most grammars—i.e. all the grammars that generate an infinite language—, $\bot$ is not of finite index, and thus cannot be applied in practice.

Lattice Operations The lattice structure provides two operations for combining equivalence relations into new ones. Given a set E of equivalence relations in $\mathrm{Eq}(\mathcal{N})$, the least upper bound or *join* of these relations is the equivalence relation defined by

$$\nu \bigvee E \; \nu' \text{ iff } \exists n \in \mathbb{N}, \exists \equiv_0, \dots, \equiv_n \in E, \exists \nu_0, \dots, \nu_{n+1} \in \mathcal{N}, \\ \nu = \nu_0, \nu' = \nu_{n+1}, \text{ and } \nu_i \equiv_i \nu_{i+1} \text{ for } 0 \le i \le n. \tag{4.18}$$

Conversely, the greatest lower bound or *meet* of E is the equivalence relation

$$\nu \bigwedge E \; \nu' \text{ iff } \forall \equiv_i \in E, \nu \equiv_i \nu'. \tag{4.19}$$

When E has only two elements $\equiv_a$ and $\equiv_b$, we write more simply $\equiv_a \vee \equiv_b$ and $\equiv_a \wedge \equiv_b$ for their join and meet respectively.

This very ability to combine equivalence relations makes our grammatical representation highly generic, and allows for various trade-offs. For instance, finer equivalence relations are obtained when using the meet of two equivalence relations.

Example 4.13. Let us define the lookahead position equivalence look_k for $k \geq 0$ by

$$x_b d_i({}^{\alpha}_{u_b} \bullet {}^{\alpha'}_{u'_b}) r_i x'_b \;\; \mathsf{look}_k \;\; y_b d_j({}^{\beta}_{v_b} \bullet {}^{\beta'}_{v'_b}) r_j y'_b \text{ iff } k : x' = k : y'. \tag{4.20}$$

For $k \geq 0$, we define item_k as a refinement of item_0 by

$$\mathsf{item}_k {=} \mathsf{item}_0 \wedge \mathsf{look}_k \; . \tag{4.21}$$

The look_k equivalence relation emulates a lookahead window of k terminal symbols in the remaining context of the position, and item_k thus corresponds to a LR(k) precision. □

Position Equivalences The significant orderings between equivalence relations are those orderings that remain true for all the position sets $\mathcal{N}$ of all the context-free grammars $\mathcal{G}$. Let us define a *position equivalence* as a function $\equiv$ that associates to a particular position set $\mathcal{N}$ an instance equivalence relation $\overset{\mathcal{N}}{\equiv}$ in Eq($\mathcal{N}$); item_0, $\top$ and $\bot$ are actually position equivalences according to this definition.

We can define a partial order $\subseteq$ on position equivalences by $\equiv_a \subseteq \equiv_b$ if and only if $\overset{\mathcal{N}}{\equiv}_a \subseteq \overset{\mathcal{N}}{\equiv}_b$ for all $\mathcal{N}$, and observe that it also defines a complete lattice by extending Equation 4.18 and Equation 4.19 in a natural way. The context usually prevents mistaking $\overset{\mathcal{N}}{\equiv}$ with $\equiv$, and in these situations, we denote $\overset{\mathcal{N}}{\equiv}$ by $\equiv$.

Position Automata Sizes Let us remark that our lattice of equivalence relations is related to a lattice of nondeterministic position automata sizes: if $\equiv_a \subseteq \equiv_b$, then the equivalence classes of $\equiv_b$ are unions of equivalence classes of $\equiv_a$, and thus $|\mathcal{N}/{\equiv_a}\,| \geq |\mathcal{N}/{\equiv_b}\,|$ holds on the equivalences' index.

If we define the *size* of a FSA $\mathcal{A} = \langle Q, \Sigma, R, Q_s, Q_f \rangle$ by $|\mathcal{A}| = \max(|Q|, |R|)$, then, $\equiv_a \subseteq \equiv_b$ implies $|\Gamma/{\equiv_a}| \geq |\Gamma/{\equiv_b}|$. The function that maps an equivalence relation $\equiv$ of finite index in Eq($\mathcal{N}$) to the size $|\Gamma/{\equiv}|$ is thus *antitone*.

The smallest nondeterministic position automaton is therefore $\Gamma/\top$, of size exactly $|V_b|$, where $|V_b| = |V| + 2|P|$. More generally, the size of a position automaton is bounded by

$$\mathcal{O}(|\mathcal{N}/{\equiv}\,|^2\,|V_b|). \tag{4.22}$$

Considering the case of item_0, the number of equivalence classes is the size of $\mathcal{G}$ $|\mathcal{G}| = \sum_{A \to \alpha \in P} |A\alpha|$, and the bound (4.22) becomes $\mathcal{O}(|\mathcal{G}|^2(|V|{+}2|P|))$. It can be tightened to

$$\mathcal{O}(|\mathcal{G}|\,|P|) \tag{4.23}$$

by observing that a given state can only have a single transition over some symbol in V, and can be further optimized to $\mathcal{O}(|\mathcal{G}|)$ with the alternate construction seen in Section 4.2.1.1 on page 54.

4.2.2 Parsing with Position Automata

A first, simple, application of position automata is to provide a generalization to the classical parsing algorithms. By translating grammar positions $A{\rightarrow}\alpha{\scriptstyle\bullet}\alpha'$ into states of a position automaton, it is straightforward to adapt the rules of a shift-reduce or Earley parser to employ position automata instead.

4.2.2.1 Shift-Reduce Parsing

Context-free languages are characterized by PushDown Automata (PDA), presented in Definition A.10 on page 166 and already introduced in Section 2.1.2.1 on page 10. The classical constructions for pushdown automata from a context-free grammar can be reworded to work from a position automaton. We present here the construction of a shift-reduce pushdown automaton, along the lines of Definition A.12 on page 167.

Definition 4.14. Let $\mathcal{G} =$ be a grammar and $\Gamma/{\equiv} = \langle Q, V_b, R, Q_s, Q_f\rangle$ its position automaton for the equivalence relation $\equiv$.

The *shift-reduce recognizer* for $\Gamma/{\equiv}$ is a pushdown automaton $M(\equiv)$ of form $\langle Q, T, R', Q_s, \{qq' \mid q \in Q_s, q' \in Q_f\}, \$, \|\rangle$ where the set of rules R' is the union of the

***shift* rules** $\{q\|a \vdash qq'\| \mid a \in T, qa \vdash q' \in R\}$,

***reduce* rules** $\{qq_0 \dots q_n\| \vdash qq'\| \mid \exists i = A{\rightarrow}X_1 \dots X_n \in P, qA \vdash q', q_n r_i \vdash q' \in R\}$, and

empty rules $\{q\| \vdash qq'\| \mid \exists i \in P, qd_i \vdash q' \in R\}$.

A shift-reduce parser is obtained as usual from the recognizer by associating an output effect τ' that maps reduce rules to grammar productions and the other rules to the empty string.

The definition of $M(\equiv)$ is a translation of the construction of a *core restricted* shift-reduce parser: at reduction time, the stack contents are not inspected, but we simply pop the length of the rightpart of the rule at hand. Such parsers are known to be correct whenever $\equiv \subseteq \mathsf{item}_0$ (Heilbrunner, 1981).

4.2.2.2 Earley Parsing

Recognition We present here the case of the Earley recognizer, defined in Section A.1.6 on page 167. The Earley items are now triples (q, i, j), $0 \leq i \leq j \leq n$, where q is a state of a position automaton $\Gamma/{\equiv}$. The set of correct Earley items (or chart) $\mathcal{I}_{\equiv}^{\mathcal{G},w}$ for a string $w = a_1 \cdots a_n$ is now the deduction closure of the system comprising the deduction rules (Init'), (Predict'), (Scan'), and (Complete') instead of (Init), (Predict), (Scan), and (Complete) respectively.

$$\frac{}{(q_s, 0, 0)} \left\{ \; q_s \in Q_s \; \right\} \; (q_s, 0, 0) \to \varepsilon \tag{Init'}$$

$$\frac{(q, i, j)}{(q', j, j)} \left\{ \; q d_k \vdash q' \; \right\} \; (q', j, j) \to \varepsilon \tag{Predict'}$$

$$\frac{(q, i, j)}{(q', i, j+1)} \left\{ \begin{array}{l} qb \vdash q' \\ a_{j+1} = b \end{array} \right\} \; (q', i, j+1) \to (q, i, j)\, b \tag{Scan'}$$

$$\frac{\begin{array}{c}(q, i, j) \\ (q', j, k)\end{array}}{(q'', i, k)} \left\{ \begin{array}{l} q' r_l \vdash q'' \\ l = A \to \alpha \\ qA \vdash q'' \end{array} \right\} \; \begin{array}{l} (q'', i, k) \to (q, i, j)\, (A, j, k) \\ (A, j, k) \to (q', j, k) \end{array} \tag{Complete'}$$

Recognition succeeds if an item $(q_f, 0, n)$ with q_f in Q_f belongs to $\mathcal{I}_{\equiv}^{\mathcal{G},w}$.

Parsing The recognizer is turned into a parser by generating the rules of a binary grammar forest $\mathcal{G}_w$ at each deduction step, as indicated on the right end of the deduction rules (Init'), (Predict'), (Scan'), and (Complete'). The correctness of the parser is clear if $\equiv \subseteq \mathsf{item}_0$: replacing the states q, q', etc. by the corresponding items, we see that our deduction rules are a simple translation of the deduction rules given in Section A.1.6.

4.2.3 Approximating Languages

Position automata provide a framework for the static analysis of context-free grammars. Thanks to the position equivalences, approximations of various degrees can be used, making this framework fairly general.

We present here some formal arguments on how general it is: can any language over-approximation be expressed in the framework? Not quite, for we only allow approximations that retain some of the structure of the grammar. But we are going to see two different ways to obtain partial converses for Theorem 4.11 (resp. Corollary 4.12), i.e., given a language L that over-approximates $\mathcal{L}(\mathcal{G}_b)$ (resp. $\mathcal{SF}(\mathcal{G}_b)$), to find an equivalence relation $\equiv$ such that $\mathcal{L}(\Gamma/\equiv) \cap T_b^*$ (resp. $\mathcal{L}(\Gamma/\equiv)$) is "close" to L.

4.2.3.1 Galois Connection

Our first attempt at a converse relies on the lattice structure of the equivalence relations on a given $\mathcal{N}$. We consider for this the function g^* that maps an equivalence relation $\equiv$ in $\mathrm{Eq}(\mathcal{N})$ to the language $\mathcal{L}(\Gamma/{\equiv})$:

$$g^*(\equiv) = \mathcal{L}(\Gamma/{\equiv}). \tag{4.24}$$

It is straightforward to see that $g^*(\top) = V_b^*$, and that $g^*(\bot) = \mathcal{SF}(\mathcal{G}_b)$.

Let us denote the set of the languages over V_b between $\mathcal{SF}(\mathcal{G}_b)$ and V_b^* by $[\mathcal{SF}(\mathcal{G}_b), V_b^*]$. The interval $[\mathcal{SF}(\mathcal{G}_b), V_b^*]$ is partially ordered by language inclusion $\subseteq$, and is also a complete lattice with union and intersection as join and meet operations respectively.

Let us consider a set of equivalence relations E included in $\mathrm{Eq}(\mathcal{N})$; then $g^*(\bigwedge E)$ is the language of $\Gamma/\bigwedge E$. Since the equivalence classes of $\bigwedge E$ are the intersections of the equivalence classes of the individual relations in E, it is rather straightforward that

$$\mathcal{L}(\Gamma/\textstyle\bigwedge E) = \displaystyle\bigcap_{\equiv_i \in E} \mathcal{L}(\Gamma/{\equiv_i}), \tag{4.25}$$

i.e. that g^* preserves meets. Regarding joins, they are not preserved by g^*, and all we can show is that $\bigcup_{\equiv_i \in E} \mathcal{L}(\Gamma/{\equiv_i}) \subseteq \mathcal{L}(\Gamma/\bigvee E)$.

Since $\mathrm{Eq}(\mathcal{N})$ is a complete lattice and g^* preserves meets, then g^* has a unique coadjoint g_* such that for all equivalence relations $\equiv$ in $\mathrm{Eq}(\mathcal{N})$ and for all languages L in $[\mathcal{SF}(\mathcal{G}_b), V_b^*]$, $g_*(L) \subseteq {\equiv}$ if and only if $L \subseteq g^*(\equiv)$. This unique coadjoint is defined by

$$g_*(L) = \bigwedge\{\equiv \mid L \subseteq g^*(\equiv)\}. \tag{4.26}$$

Such a pair of functions was coined a *Galois connection* by Ore (1944),[3] and is usually written

$$[\mathcal{SF}(\mathcal{G}_b), V_b^*] \underset{g_*}{\overset{g^*}{\leftrightarrows}} \mathrm{Eq}(\mathcal{N}). \tag{4.27}$$

As a consequence of this connection, we know that

- both g^* and g_* are monotone: for $\equiv_a$, $\equiv_b$ in $\mathrm{Eq}(\mathcal{N})$, $\equiv_a \subseteq \equiv_b$ implies $\mathcal{L}(\Gamma/{\equiv_a}) \subseteq \mathcal{L}(\Gamma/{\equiv_b})$ and for L_a, L_b in $[\mathcal{SF}(\mathcal{G}_b), V_b^*]$, $L_a \subseteq L_b$ implies $g_*(L_a) \subseteq g_*(L_b)$,
- g_* and g^* are mutual quasi-inverses: $g^* \circ g_* \circ g^* = g^*$ and $g_* \circ g^* \circ g_* = g_*$,
- $g_* \circ g^*$ is decreasing: for $\equiv$ in $\mathrm{Eq}(\mathcal{N})$, $g_*(\mathcal{L}(\Gamma/{\equiv})) \subseteq {\equiv}$, and
- $g^* \circ g_*$ is increasing: for L in $[\mathcal{SF}(\mathcal{G}_b), V_b^*]$,

$$\mathcal{SF}(\mathcal{G}_b) \subseteq L \subseteq \mathcal{L}(\Gamma/g_*(L)). \tag{4.28}$$

[3] Ore (1944) and Birkhoff (1940) originally gave an equivalent, contravariant formulation.

Equation 4.28 is a partial converse to Corollary 4.12. The Galois connection provides a very general means to obtain an equivalence relation from a language that includes $\mathcal{SF}(\mathcal{G}_b)$: no provision was made regarding the regularness, or even the context-freeness, of the languages in $[\mathcal{SF}(\mathcal{G}_b), V_b^*]$. A similar connection holds if we consider the adjoint function g'^* that maps an equivalence relation $\equiv$ in $\mathrm{Eq}(\mathcal{N})$ to the language $\mathcal{L}(\Gamma/\!\equiv) \cap T_b^*$ in $[\mathcal{L}(\mathcal{G}_b), T_b^*]$, thus providing a partial converse to Theorem 4.11 instead.

4.2.3.2 Approximating with Position Automata

The previous attempt at finding a converse for Corollary 4.12 and Theorem 4.11 is arguably too general: it is difficult to tell anything about $g_*(L)$. In particular, it is not clear whether $g_*(L)$ is of finite index if L is a regular language over V_b.

We present here a more pragmatic approach for this last case: if a language L is regular, then it is recognized by a FSA $\mathcal{A}$ such that $L = \mathcal{L}(\mathcal{A})$. We are going to see how to derive an equivalence relation from $\mathcal{A}$. But first, we proceed to issue some conditions on $\mathcal{A}$.

Approximate Automata Let $w = a_1 \cdots a_n$ be a sentence in the language of a FSA $\mathcal{A}$; if $q_0 a_1 \cdots a_n \vDash q_1 a_2 \cdots a_n \vDash \cdots \vDash q_n$, q_0 in Q_s and q_n in Q_f, then $q_0 \cdots q_n$ in Q^* is an *accepting computation* of $\mathcal{A}$ on w. A sentence w is *ambiguous* in $\mathcal{A}$ if it has at least two different accepting computations; otherwise, it is *unambiguous.*

Definition 4.15. A FSA $\mathcal{A} = \langle Q, V_b, R, Q_s, Q_f \rangle$ *approximates* the grammar $\mathcal{G}$ if

(i) $\mathcal{L}(\mathcal{G}_b) \subseteq \mathcal{L}(\mathcal{A}) \cap T^*$,

(ii) each w_b in $\mathcal{L}(\mathcal{G}_b)$ is unambiguous in $\mathcal{A}$,

(iii) $q\delta_b \vDash^* q'$ in $\mathcal{A}$ and $S \Rightarrow_b^* \alpha_b \gamma_b \beta_b \Rightarrow_b^* \alpha_b \delta_b \beta_b$ in $\mathcal{G}_b$ imply that $q\gamma_b \vDash^* q'$ in $\mathcal{A}$, and

(iv) $\mathcal{A}$ is ε-free, i.e. $R \subseteq Q\, V_b \times Q$.

Clearly, any FSA obtained through the quotient construction of Definition 4.6 and unambiguous on $\mathcal{L}(\mathcal{G}_b)$ approximates $\mathcal{G}$ according to Definition 4.15. Given a FSA $\mathcal{A} = \langle Q, V_b, R, Q_s, Q_f \rangle$ that approximates $\mathcal{G}$, we define the relation $\mathsf{pa}_{\mathcal{A}}$ by

$$\begin{aligned} d_1 x_b ({}^{\alpha}_{u_b} \bullet {}^{\alpha'}_{u'_b}) x'_b r_1 \;\mathsf{pa}_{\mathcal{A}}\; d_1 y_b ({}^{\beta}_{v_b} \bullet {}^{\beta'}_{v'_b}) y'_b r_1 \text{ iff } & \exists q_s, q'_s \in Q_s, \exists q_f, q'_f \in Q_f, \exists q \in Q, \\ & q_s x_b u_b u'_b x'_b \vDash^* q u'_b x'_b \vDash^* q_f \\ & \text{and } q'_s y_b v_b v'_b y'_b \vDash^* q v'_b y'_b \vDash^* q'_f. \end{aligned} \tag{4.29}$$

That $\mathsf{pa}_{\mathcal{A}}$ is an equivalence relation can be verified using the conditions *(ii)* and *(iv)* of Definition 4.15 for proving transitivity; the reflexivity and symmetry properties are obvious, and condition *(i)* is needed for $\mathsf{pa}_{\mathcal{A}}$ to be defined on all the positions of the grammar.

Theorem 4.16. *Let $\mathcal{G}$ be a context-free grammar, and $\Gamma/\!\equiv$ a position automaton by quotienting with $\equiv$ that approximates $\mathcal{G}$. The relations $\equiv$ and $\mathsf{pa}_{\Gamma/\equiv}$ are the same.*

Proof. Let $\nu = d_1x_b({}^{\alpha}_{u_b}{\bullet}{}^{\alpha'}_{u'_b})x'_br_1$ and $\nu' = d_1y_b({}^{\beta}_{v_b}{\bullet}{}^{\beta'}_{v'_b})y'_br_1$ be two positions in $\mathcal{N}$. Then, there exist ν_s, ν'_s in μ_s and ν_f, ν'_f in μ_f such that

$$\nu_s \overset{x_bu_b}{\rightarrowtail} \nu \overset{u'_bx'_b}{\rightarrowtail} \nu_f \text{ and } \nu'_s \overset{y_bv_b}{\rightarrowtail} \nu' \overset{v'_by'_b}{\rightarrowtail} \nu'_f.$$

By Lemma 4.9,

$$[\nu_s]_\equiv x_bu_bu'_bx'_b \vDash^* [\nu]_\equiv u'_bx'_b \vDash^* [\nu_f]_\equiv \text{ and } [\nu'_s]_\equiv y_bv_bv'_by'_b \vDash^* [\nu']_\equiv v'_by'_b \vDash^* [\nu'_f]_\equiv.$$

If $\nu \equiv \nu'$, then let $q = [\nu]_\equiv = [\nu']_\equiv$; the two previous paths comply with the definition of $\mathsf{pa}_{\Gamma/\equiv}$, and thus $\nu \;\mathsf{pa}_{\Gamma/\equiv}\; \nu'$.

If $\nu \;\mathsf{pa}_{\Gamma/\equiv}\; \nu'$, then there exist q_s, q'_s in Q_s, q_f, q'_f in Q_f and q in Q that fulfill the conditions of Equation 4.29. If $[\nu_s]_\equiv \neq q_s$ or $[\nu_f]_\equiv \neq q_f$ or $[\nu]_\equiv \neq q$, and in the absence of ε transitions, then there would be two different accepting paths for $x_bu_bu'_bx'_b$, in violation of condition *(ii)*, and similarly for $y_bv_bv'_by'_b$. Thus $q = [\nu]_\equiv = [\nu']_\equiv$, and $\nu \equiv \nu'$. □

Theorem 4.16 does not mean that, given an approximate FSA $\mathcal{A}$ according to Definition 4.15, then the position automaton obtained by quotienting through $\mathsf{pa}_{\mathcal{A}}$ will be the same; $\Gamma/\mathsf{pa}_{\mathcal{A}}$ will rather be an *expunged* version of $\mathcal{A}$ with respect to the grammar.

Partial Converses The non-empty equivalence classes of $\mathsf{pa}_{\mathcal{A}}$ are the states of $\mathcal{A}$ that appear in an accepting computation of a sentence of $\mathcal{G}_b$, and thus the set of states of $\Gamma/\mathsf{pa}_{\mathcal{A}}$ is a subset of the set of states of $\mathcal{A}$. In particular, this inclusion also holds for the sets of initial and final states. Last of all, the same inclusion holds for the terminal transitions.

Proposition 4.17. *Let $\mathcal{G}$ be a context-free grammar and $\mathcal{A}$ a FSA that approximates $\mathcal{G}$.*

(i) If q is a state of $\Gamma/\mathsf{pa}_{\mathcal{A}}$, then it is a state of $\mathcal{A}$.

(ii) If q_s (resp. q_f) is an initial state (resp. final state) of $\Gamma/\mathsf{pa}_{\mathcal{A}}$, then it is an initial state (resp. final state) of $\mathcal{A}$.

(iii) Let q and q' be two states of $\Gamma/\mathsf{pa}_{\mathcal{A}}$ and χ a symbol in T_b. If a rule $q\chi \vdash q'$ exists in $\Gamma/\mathsf{pa}_{\mathcal{A}}$, then it exists in $\mathcal{A}$.

Condition *(iii)* of Definition 4.15 thus insures that all the transitions of $\Gamma/\mathsf{pa}_{\mathcal{A}}$ also exist in $\mathcal{A}$. As a result, Corollary 4.18 provides a step towards a partial converse to Corollary 4.12 for regular approximations of $\mathcal{SF}(\mathcal{G}_b)$.

Corollary 4.18. *Let $\mathcal{G}$ be a context-free grammar and $\mathcal{A}$ a FSA that approximates $\mathcal{G}$. Then $\mathcal{SF}(\mathcal{G}_b) \subseteq \mathcal{L}(\Gamma/\mathsf{pa}_{\mathcal{A}}) \subseteq \mathcal{L}(\mathcal{A})$.* □

A partial converse to Theorem 4.11 is easily obtained when we note that condition *(iii)* of Definition 4.15 is not needed for $\mathsf{pa}_{\mathcal{A}}$ to be an equivalence relation on $\mathcal{N}$. Therefore, pa is actually defined for any regular language that includes $\mathcal{L}(\mathcal{G}_b)$, so that *(i)* holds: we can always construct an ε-free unambiguous FSA $\mathcal{A}$ for it, so that *(iv)* and *(ii)* hold. By Proposition 4.17, we have the following corollary.

Corollary 4.19. *Let $\mathcal{G}$ be a context-free grammar and L a regular language over T_b such that $\mathcal{L}(\mathcal{G}_b) \subseteq L$. There exists an ε-free unambiguous FSA $\mathcal{A}$ with $\mathcal{L}(\mathcal{A}) = L$ such that $\mathcal{L}(\mathcal{G}_b) \subseteq (\mathcal{L}(\Gamma/\mathsf{pa}_{\mathcal{A}}) \cap T_b^*) \subseteq L$.* □

4.3 Recognizing Derivation Trees

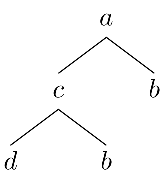
The next two chapters are devoted to two important applications of position automata: parser construction (Chapter 5) and ambiguity detection (Chapter 6), where these subjects will be treated in depth. In this section, we rather demonstrate the usefulness of position automata as regular approximations of context-free grammars, through the study of a simple problem: recognizing well-bracketed sentences of $\mathcal{G}$ by means of a finite automaton.

Although the simple case we consider here is of limited practical interest, it allows to use some of the results on the lattice of equivalence relations (Section 4.2.1.3) and on the characterization of position automata (Section 4.2.3.2) to obtain a complete characterization—in the position automata framework—of which grammars can be recognized by a prefix-correct finite-state machine (Section 4.3.4).

But first we provide some general background on tree languages and overview their relation with position automata.

4.3.1 Tree Languages

Besides regular word languages, finite position automata can define tree languages and nested word languages. In this context, position automata are limited by the absence of any pushdown mechanism, but could still be considered, in particular for validating streaming XML documents (Segoufin



(a) A tree generated by $\mathcal{T}$.

```
<a>
  <c>
    <d/>
    <b/>
  </c>
  <b/>
</a>
```

(b) An XML string generated by $\mathcal{T}$.

Figure 4.5: A tree in the language of $\mathcal{T}$ and its XML yield.

and Vianu, 2002); we will consider a very simple case of this issue in Section 4.3.2.

Regular Tree Languages A ranked[4] *regular tree language* is generated by a normalized *regular tree grammar* $\mathcal{T} = \langle S, N, T, R\rangle$, where each rule in R is of form $A{\rightarrow}a(A_1, \dots, A_n)$, $n \geq 0$ (Comon et al., 2007). Terminals label the nodes of the generated trees, while nonterminals denote node *types*: according to the previous rule, a tree node labeled by a and with children typed by $A_1, \dots, A_n$ in this order can in turn be typed by A. A tree is generated by $\mathcal{T}$ if its root has type S.

Example 4.20. Consider for instance the regular tree grammar with root A and rules

$$\begin{aligned} &A \rightarrow a(AB)\\ &A \rightarrow c(AB)\\ &A \rightarrow d\\ &B \rightarrow b. \end{aligned} \qquad (\mathcal{T})$$

The grammar generates, among others, the tree of Figure 4.5a, which corresponds to the XML string shown in Figure 4.5b. □

We can construct from a regular tree grammar $\mathcal{T} = \langle S, N, T, R\rangle$

1. a context-free grammar $\mathcal{G} = \langle N, T, P, S\rangle$ with the set of productions $P = \{A{\rightarrow}A_1 \dots A_n \mid A{\rightarrow}a(A_1, \dots, A_n) \in R\}$ and
2. a labeling relation l in $P \times T$ defined by $(i, a) \in l$ if and only if a rule $A{\rightarrow}a(A_1, \dots, A_n)$ in R corresponds to the rule $A \xrightarrow{i} A_1 \dots A_n$ in P.

The internal nodes of the derivation trees produced by $\mathcal{G}$, labeled by nonterminals, can be relabeled with l (by considering which production was exercised) in order to obtain the trees in the language of $\mathcal{T}$.

[4]It is straightforward to consider unranked tree languages by simply allowing *extended* context-free grammars, where the rule rightparts are regular expressions over V^*.

In the case of Example 4.20 on the preceding page, we would define the context-free grammar with rules

$$\begin{aligned} A &\xrightarrow{1} AB \\ A &\xrightarrow{2} \varepsilon \\ B &\xrightarrow{3} \varepsilon \end{aligned} \tag{$\mathcal{G}_8$}$$

and the labeling

$$\{(1, a), (1, c), (2, d), (3, b)\}. \tag{4.30}$$

Different classes of tree languages are obtained when one restricts the form of the tree grammar, matching the languages defined by DTDs, XML Schema or Relax NG grammars (Murata et al., 2005).

Position automata correspond to a restricted class of *tree walking automata* (Aho and Ullman, 1971), where the d_i and r_i transitions denote *down* and *up* moves on a symbol $l(i)$, and the remaining terminal transitions denote *right* moves on the given terminal symbol. A single nonterminal transition on A in a position automaton is then mapped to several right moves of the tree walking automaton, one for each symbol $l(i)$ such that A is the leftpart of the rule $i = A{\rightarrow}\alpha$.

The resulting tree walking automaton is restricted since it is not allowed any *left* move, nor to choose which child to visit in a down move—but always selects the leftmost one—, and does not know which child number the current tree node holds. Tree walking automata, with this last restriction (Kamimura and Slutzki, 1981) or without it (Bojańczyk and Colcombet, 2005), do not recognize all the regular tree languages.

Nested Word Languages Nested word languages were introduced by Alur and Madhusudan (2006) as a general model of the *parenthesis languages* (Knuth, 1967) used in software verification and XML. In addition to a word in Σ^*, a *nested word* over Σ contains hierarchical information in the form of a matching relation about which of its letters are properly nested *calls*, *returns*, or *internal* sites. Note that, in a nested word, the symbols at a call site and at its matching return site are not necessarily the same.

The walk performed by a position automaton is identical to the one performed by a *nested word automaton*. Let us define two labeling functions d and r from P to T; for $i = A{\rightarrow}\alpha$, a d_i transition in a position automaton corresponds to a call on symbol $d(i)$, whereas an r_i transition corresponds to a return on $r(i)$. A transition on nonterminal A then corresponds to a transition on one of the possible $d(i)$.

In the taxonomy of Alur (2007), position automata correspond to *flat* nested word automata. As such, they fall short of being (nondeterministic) descriptors for all nested word languages.

4.3.2 Validating XML Streams

4.3.2.1 The XML Case

Segoufin and Vianu (2002) present the problem of validating an XML stream against a DTD under memory constraints. The XML stream is assumed to be the result of another application, for instance of an XSLT transformation, that guarantees its well-formedness, i.e. such that its opening and closing XML tags match.

Nonetheless, well-formedness is not enough to satisfy the constraints of the XML dialect, and the stream should further be validated. Segoufin and Vianu study under which conditions this validation can be performed by a finite-state automaton: they present two necessary conditions for this validation to be possible against a DTD. It is still open whether these conditions are also sufficient (as conjectured by the authors), and which conditions should be enunciated in the more general case of validation against a regular tree language.

4.3.2.2 The Bracketed Case

We consider here a substantially simpler case of prefix-correct validation against a bracketed grammar $\mathcal{G}_b$. Given a well-bracketed string w_b over T_b^*, which could for instance represent a derivation tree obtained as the result of a tree transduction, we want to check whether the sentence it yields belongs to $\mathcal{L}(\mathcal{G}_b)$ or not—we call this *validating* $\mathcal{L}(\mathcal{G}_b)$—by means of a FSA. Furthermore, we want this validation to be prefix-correct, i.e. that during the operation of the FSA, a prefix of a well-bracketed string is processed if and only if it is the prefix of a sentence of $\mathcal{G}_b$. Prefix correctness provides precious information for error detection. It insures that processing ends as soon as the input stops making sense, thus pinpointing a good candidate position for the error. Let us illustrate the issue with a few examples.

Example 4.21. Consider the grammar with rules

$$S\xrightarrow{2}A,\ S\xrightarrow{3}\varepsilon,\ A\xrightarrow{4}B,\ B\xrightarrow{5}A,\ B\xrightarrow{6}\varepsilon, \qquad (\mathcal{G}_9)$$

the regular language $d_2d_4(d_5d_4)^*d_6r_6(r_4r_5)^*r_4r_2 \mid d_3r_3$ contains a well-bracketed string if and only if it belongs to the language of the bracketed grammar of $\mathcal{G}_9$. □

Example 4.22. Consider the grammar with rules

$$S\xrightarrow{2}SS,\ S\xrightarrow{3}\varepsilon, \qquad (\mathcal{G}_{10})$$

the bracketed language of $\mathcal{G}_{10}$ cannot be validated by any regular language. Intuitively, after having recognized the language of S, a regular language

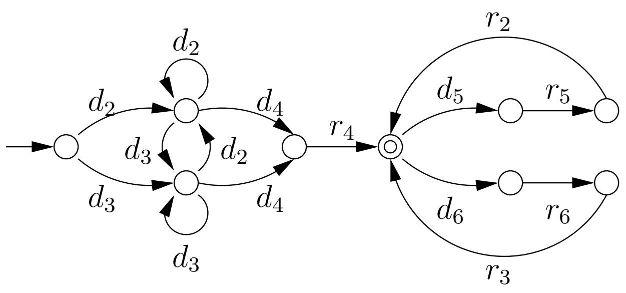


Figure 4.6: A validating FSA for $\mathcal{G}_{11}$.

cannot remember whether it should match the language of a second S, or consider that production 2 is finished. □

Example 4.23. Consider the grammar with rules

$$S\xrightarrow{2}SA,\ S\xrightarrow{3}SB,\ S\xrightarrow{4}\varepsilon,\ A\xrightarrow{5}\varepsilon,\ B\xrightarrow{6}\varepsilon, \qquad (\mathcal{G}_{11})$$

the FSA of Figure 4.6 validates the bracketed language of $\mathcal{G}_{11}$. Nonetheless, this FSA is not prefix-correct: we can match the prefix $d_2d_4r_4d_6r_6$—although it is not the prefix of any bracketed sentence of $\mathcal{G}_{11}$—before seeing r_2 and knowing that the input string is incorrect. Intuitively, no FSA can keep track of whether it should accept d_5 or d_6 after an r_4 symbol at an arbitrary depth. □

Our case of study is simpler than the general validation of an XML stream in several ways:

- since each grammar production i is uniquely identified by the bracketing between d_i and r_i, we are free from the labeling issues of regular tree languages, which might lead to ambiguity in general;
- since we consider context-free grammars and not extended context-free grammars, we are limited to ranked trees: each rule right part is a finite string over V^*;
- last of all, we consider prefix-correct recognition instead of recognition; as demonstrated with $\mathcal{G}_{11}$, this is more restrictive.

4.3.3 A Characterization

We fully characterize which grammars are amenable to prefix-correct validation in terms of equivalence relations in our framework. Our approach relies on a necessary and sufficient condition on equivalence relations that enforces prefix correctness. We were not lucky enough for this condition to define an equivalence relation for all grammars, and we have to accomodate to this limitation. If an equivalence relation on grammar positions satisfies this condition and is of finite index, then it means that the grammar can be validated by means of a prefix-correct finite state automaton, and conversely.

Before we define our condition, we first define a recognitive device for well-bracketed strings: *bracket pushdown automata.*

4.3.3.1 Well-bracketed Strings

We formalize the problem as a recognition problem for a special kind of pushdown automaton (see Definition A.10 on page 166 for the definition of a PDA): a bracket pushdown automaton $\mathcal{B}$ employs its stack solely to remember which opening brackets it should close next. Bracket pushdown automata provide a convenient notation for the processing of well-bracketed strings by labeled transition systems (LTS).

Bracketed Pushdown Automaton We first solve a small technical point: matching opening and closing brackets is not enough to ensure that a bracketed string is well-bracketed. For instance, $d_2\, a\, r_2\, b\, d_3\, c\, r_3$ is well-balanced, but it is not the yield of any derivation tree. The difference is the same as the one between Dyck words and Dyck primes.

In order to handle this technicality, we augment our labeled transition systems with an initial d_1 and a final r_1 transition.

Definition 4.24. Let $\mathcal{A} = \langle Q, V_b, R, Q_s, Q_f\rangle$ be a LTS over the bracketed alphabet V_b. Its *augmented* version $\mathcal{A}' = \langle Q', V_b', R', \{q_s'\}, \{q_f'\}\rangle$ where

- $Q' = Q \cup \{q_s', q_f'\}$ contains two fresh initial and final states q_s' and q_f', and where
- the augmented set of rules R' is the union

$$R \cup \{q_s' d_1 \vdash q_s \mid q_s \in Q_s\} \cup \{q_f r_1 \vdash q_f' \mid q_f \in Q_f\}.$$

Thus, an augmented position automaton $\Gamma/{\equiv'}$ now recognizes a superset of the language of $\mathcal{G}_b'$, the augmented bracketed grammar of $\mathcal{G}$.

Definition 4.25. Let $\mathcal{G}$ be a grammar, $\mathcal{A}$ a LTS over the bracketed alphabet V_b and $\mathcal{A}' = \langle Q', V_b', R', \{q_s'\}, \{q_f'\}\rangle$ its augmented automaton. We define the *bracket pushdown automaton* $\mathcal{B}(\mathcal{A})$ as the PDA $\langle P_b' \cup Q', T_b', R_b, \{q_s'\}, \{q_f'\}, \$, \|\rangle$

where the set of rules R_b is defined as the union

$$\begin{aligned}&\{q\|a \vdash q'\| \mid qa \vdash q' \in R'\}\\ \cup\ &\{q\|d_i \vdash iq'\| \mid qd_i \vdash q' \in R'\}\\ \cup\ &\{iq\|r_i \vdash q'\| \mid qr_i \vdash q' \in R'\}.\end{aligned}$$

Clearly, $\mathcal{B}(\mathcal{A})$ accepts a string w_b in $(T_b')^*$ if and only if w_b is well-bracketed and is in $d_1\ \mathcal{L}(\mathcal{A}) \cap T_b^*\ r_1$. Our general recognition problem can thus be restated as finding $\mathcal{A}$ such that $\mathcal{L}(\mathcal{B}(\mathcal{A})) = \mathcal{L}(\mathcal{G}_b')$. We will consider in particular finite position automata $\Gamma/{\equiv}$ such that $\mathcal{L}(\mathcal{B}(\Gamma/{\equiv})) = \mathcal{L}(\mathcal{G}_b')$. By Theorem 4.11, the inclusion

$$\mathcal{L}(\mathcal{G}_b') \subseteq \mathcal{L}(\mathcal{B}(\Gamma/{\equiv})) \tag{4.31}$$

always holds.

Prefix Correctness An on-line formal machine recognizing a language L is called *prefix correct* if whenever it reaches a configuration by processing some string u, there exists a string v such that uv is in L. In terms of our bracket PDAs, $\mathcal{B}(\mathcal{A})$ is prefix correct if the existence of q in Q' and φ in $P_b'^*$ with

$$\$q_s'\|u_b u_b'\$ \models^* \$\varphi q\|u_b'\$, \tag{4.32}$$

implies the existence of v_b such that $u_b v_b$ is in $\mathcal{L}(\mathcal{G}_b')$.

Since we took the precaution of delimitating our sentence by d_1 and r_1, which are symbols not found in T_b, we know that $\mathcal{L}(\mathcal{B}(\mathcal{A}))$ is prefix-free. Thus the prefix correctness of $\mathcal{B}(\mathcal{A})$ implies its correctness.

Lemma 4.26. *Let $\mathcal{G}$ be a grammar and $\mathcal{A}$ be a LTS over the bracketed alphabet V_b such that $\mathcal{L}(\mathcal{G}_b') \subseteq \mathcal{L}(\mathcal{A}')$. If $\mathcal{B}(\mathcal{A})$ is prefix correct, then $\mathcal{L}(\mathcal{B}(\mathcal{A})) = \mathcal{L}(\mathcal{G}_b')$.*

Proof. Let us first notice that, d_1 and r_1 being symbols not found in T_b, they can only be opening and ending symbols of a sentence of $\mathcal{L}(\mathcal{B}(\mathcal{A}))$. Thus $\mathcal{L}(\mathcal{B}(\mathcal{A}))$ is prefix-free, i.e. none of its sentences can be a proper prefix of another.

Therefore, in an accepting computation

$$\$q_s'\|w_b\$ \models^* \$q_f'\|\$, \tag{4.33}$$

the prefix correctness of $\mathcal{B}(\mathcal{A})$ implies that there exists a bracketed string w_b' in V_b' such that $w_b w_b'$ is in $\mathcal{L}(\mathcal{G}_b')$. Since $\mathcal{L}(\mathcal{G}_b') \subseteq \mathcal{L}(\mathcal{A}')$, and $w_b w_b'$ being well-bracketed (being the yield of a tree of $\mathcal{G}_b'$), $w_b w_b'$ is also in $\mathcal{L}(\mathcal{B}(\mathcal{A}))$. But having both w_b and $w_b w_b'$ in a prefix-free language implies that $w_b' = \varepsilon$. Thus w_b is in $\mathcal{L}(\mathcal{G}_b')$. □

4.3.3.2 The no-mismatch Relation

As seen with the examples of $\mathcal{G}_{10}$ and $\mathcal{G}_{11}$, the main issue for (prefix-correct) validation is to know at all times what language is still acceptable or not. We can afford to allow indifferently two return symbols r_i and r_j since the stack of our bracket pushdown automaton will rule out the wrong one, but otherwise we need to know exactly which symbols are legitimate. The no-mismatch relation thus associates different positions if and only if their expected right language differ first on return symbols, or do not differ at all.

Mismatches Two different symbols X and Y in V_b are in *mismatch* if at least one of them is not a return symbol in T_r:

$$X \text{ mismatch } Y \text{ iff } X \neq Y \text{ and } (X \notin T_r \text{ or } Y \notin T_r). \tag{4.34}$$

We define next the *first mismatch* of two bracketed strings as the pair of suffixes of the two strings starting at the first mismatched symbols. Formally,

$$\begin{aligned}&\text{first-mismatch}(\delta_b, \gamma_b)\\ &= \begin{cases}(X\delta'_b, Y\gamma'_b) & \text{if } \delta_b = X\delta'_b,\ \gamma_b = Y\gamma'_b, \text{ and } X \text{ mismatch } Y,\\ \text{first-mismatch}(\delta'_b, \gamma'_b) & \text{if } \delta_b = X\delta'_b,\ \gamma_b = Y\gamma'_b, \text{ and } X = Y,\\ (\varepsilon, \varepsilon) & \text{otherwise.}\end{cases}\end{aligned} \tag{4.35}$$

Relation We define the no-mismatch relation on $\mathcal{N} \times \mathcal{N}$ by

$$\begin{aligned}&x_b d_i({}^{\alpha}_{u_b} \bullet {}^{\alpha'}_{u'_b}) r_i x'_b \ \text{ no-mismatch } \ y_b d_j({}^{\beta}_{v_b} \bullet {}^{\beta'}_{v'_b}) r_j y'_b\\ &\qquad\qquad \text{iff first-mismatch}(\alpha' r_i \widehat{x'_b}, \beta' r_j \widehat{y'_b}) = (\varepsilon, \varepsilon).\end{aligned} \tag{4.36}$$

In plain English, two positions are related if the covers (see page 49 for the definition of covers) of their remaining input never mismatch. The relation is symmetric and reflexive but not transitive; as a result, we are interested in equivalence relations over $\mathcal{N}$ that verify the condition on the remaining input, i.e. in equivalence relations that are included in no-mismatch.

Let us pause for a moment and consider what such position equivalences entail for our example grammars $\mathcal{G}_9$, $\mathcal{G}_{10}$ and $\mathcal{G}_{11}$, and try to figure out whether the equivalences' index will be finite or not.

For first-mismatch$(\alpha' r_i \widehat{x'_b}, \beta' r_j \widehat{y'_b})$ to be empty, we need in particular $\alpha' = \beta'$. Thus we obtain different equivalence classes for different remaining rule right parts, like $A{\rightarrow}{\bullet}B$ and $B{\rightarrow}{\bullet}A$ in the case of Example 4.21. But there is always a finite number of such cases, since there is a finite number of rules, all with a finite string as right part.

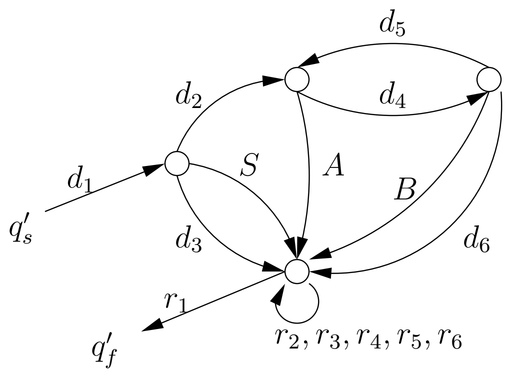


Figure 4.7: The augmented position automaton obtained for $\mathcal{G}_9$ using the no-mismatch equivalence.

In the case where $\alpha' = \beta'$, but $r_i \neq r_j$, the first-mismatch is empty as well. For instance, all the positions corresponding to items $S{\rightarrow}{\bullet}A$ and $B{\rightarrow}{\bullet}A$ end in a single equivalence class. An infinite number of positions that might end in different equivalence classes can thus only be obtained with positions sharing $\alpha' = \beta'$ and $r_i = r_j$. In the case of $\mathcal{G}_9$ in Example 4.21, the remaining languages $\widehat{x}'_b$ and $\widehat{y}'_b$ are in T_r^*, and thus cannot mismatch. Therefore, in this case, no-mismatch is actually transitive, with a finite index. Figure 4.7 shows the augmented position automaton $\Gamma_9/\text{no-mismatch}'$.

Turning to $\mathcal{G}_{10}$ from Example 4.22, a position of form

$$d_1\, d_2^n\, d_3(\,\bullet\,)r_3\, (S\, r_2)^n\, r_1, \tag{4.37}$$

where we show the covers for the remaining context, is not related to any other position of the same form for a different n. Thus no equivalence relation included in no-mismatch can be of finite index.

Last of all, for $\mathcal{G}_{11}$ from Example 4.23, a position of form

$$d_1\, d_2\, d_3^n\, d_4(\,\bullet\,)r_4\, (B\, r_3)^n\, A\, r_2\, r_1 \tag{4.38}$$

for a given value of n is not related to a position of the same form for a different value of n. Thus no equivalence relation included in no-mismatch can be of finite index either.

Lemma 4.27. *Let $\equiv$ be an equivalence relation on $\mathcal{N}$ included in no-mismatch, $i = A{\rightarrow}\alpha$ a rule in P, δ a string in V^*, and q, q' and q'' three states of $\Gamma/{\equiv}$ with $qd_i \vdash q'$.*

(i) If $q'\delta \models^* q''$, then there exists α' in V^* such that $\alpha = \delta\alpha'$.

(ii) If $q'\delta r_i \models^* q''$, then $\alpha = \delta$.

Proof. We first prove *(i)* by induction on the length of δ; obviously, if $\delta = \varepsilon$, then $\alpha = \alpha'$ fits our requirements. For the induction step, we suppose that there exists a state q''' in $\Gamma/{\equiv}$ and a symbol X in V such that

$$q'\delta X \models^* q''X \models q'''. \tag{4.39}$$

By induction hypothesis, $\alpha = \delta\alpha'$, and we need to prove that $X = 1 : \alpha'$. Since $\equiv \subseteq \mathsf{no\text{-}mismatch}$, all the positions in q'' are of form $x_b d_i ({}^{\delta}_{u_b} \bullet {}^{\alpha'}_{u'_b}) r_i x'_b$ for some bracketed strings x_b, u_b, u'_b and x'_b in T_b^*. Thus, for the transition $q''X \vdash q'''$ to be possible, we need $X = 1 : \alpha'$.

We can replace X by r_i in the previous argument to show that $\alpha' = \varepsilon$, and thus that *(ii)* holds. □

Theorem 4.28. *Let* $\equiv$ *be an equivalence relation on* $\mathcal{N}$*. If* $\equiv \subseteq \mathsf{no\text{-}mismatch}$*, then* $\mathcal{B}(\Gamma/{\equiv})$ *is prefix correct.*

Proof. Suppose that $\mathcal{B}(\Gamma/{\equiv})$ is not prefix correct. Thus there exists a rewrite

$$\$q'_s \| u_b X u'_b \$ \models^* \$\varphi' q_2 \| X u'_b \$ \models \$\varphi q_1 \| u'_b \$ \tag{4.40}$$

in $\mathcal{B}(\Gamma/{\equiv})$ for some φ, φ' in $P_b'^*$, and q_1, q_2 in Q', such that there does not exist any bracketed string in $\mathcal{L}(\mathcal{G}'_b)$ with $u_b X$ as a prefix. We assume without loss of generality that there is one such v_b such that $u_b v_b$ is in $\mathcal{L}(\mathcal{G}'_b)$. We can further assume that $v_b \neq \varepsilon$, or the last symbol of u_b would be r_1, but q'_f is the only state of $\Gamma/{\equiv'}$ reachable by r_1 and it has no outgoing transition, thus recognition would have failed on $u_b X$. Let us denote the first symbol of v_b by Y; $v_b = Y v'_b$, and we have the rewrite

$$\$q'_s \| u_b Y v'_b \$ \models^* \$\varphi' p_2 \| Y v'_b \$ \models \$\varphi'' p_1 \| v'_b \$ \tag{4.41}$$

in $\mathcal{B}(\Gamma/{\equiv})$ for some φ'' in $P_b'^*$ and p_1, p_2 in Q'. Indeed, the configurations $\$\varphi' q'_2 \| X u'_b \$$ and $\$\varphi' p_2 \| Y v'_b \$$ are reached by the same string u_b, thus they have the same stack contents φ'.

We know that $X \neq Y$, or $u_b X v'_b$ would be in $\mathcal{L}(\mathcal{G}'_b)$. We further know that at least one of X or Y is not in T_r, or $X = r_i$ and $Y = r_j$ with $i \neq j$, but only one of i or j can be the last symbol of φ', and one of the rewrites (4.40) or (4.41) would be impossible in $\mathcal{B}(\Gamma/{\equiv})$. Therefore,

$$X \mathrel{\mathsf{mismatch}} Y. \tag{4.42}$$

Let d_i be the rightmost symbol of T_d in $\widehat{u_b}$ and let $i = A \to \alpha$. One such symbol d_i exists since d_1 is the first symbol of u_b, but remains unmatched because u_b is not in $\mathcal{L}(\mathcal{G}'_b)$. We note $\widehat{u_b} = \widehat{x_b} d_i \widehat{y_b}$ such that $u_b = x_b d_i y_b$ and

$\widehat{y_b} \in V^*$. We can then detail some steps of rewrites (4.40) and (4.41) by

$$\begin{aligned}\$q'_s\|x_b d_i y_b X u'_b\$ &\models^* \$\psi q_3\|d_i y_b X u'_b\$\\ &\models \$\psi' q_4\|y_b X u'_b\$\\ &\models^* \$\varphi' q_2\|X u'_b\$\\ &\models \$\varphi q_1\|u'_b\$\end{aligned} \tag{4.43}$$

$$\begin{aligned}\$q'_s\|x_b d_i y_b Y v'_b\$ &\models^* \$\psi p_3\|d_i y_b Y v'_b\$\\ &\models \$\psi' p_4\|y_b Y v'_b\$\\ &\models^* \$\varphi' p_2\|Y v'_b\$\\ &\models \$\varphi'' p_1\|v'_b\$\end{aligned} \tag{4.44}$$

where q_3, q_4, p_3, p_4 are states of $\Gamma/{\equiv}$ and ψ and ψ' are sequences of rules in P'^*_b. By Corollary 4.10, $q_4 y_b \models^* q_2$ and $p_4 y_b \models^* p_2$ imply $q_4\widehat{y_b} \models^* q_2$ and $p_4\widehat{y_b} \models^* p_2$. We can thus apply Lemma 4.27 with $\widehat{y_b}X$ and $\widehat{y_b}Y$ in the role of δ or δr_i depending on whether one of X or Y belongs to T_r. In all cases, we have two different, incompatible, expressions for α, and thus a contradiction. □

4.3.4 Optimality

In order to obtain a complete characterization of prefix-correct validation in terms of equivalence relations on grammar positions, we first need to prove the converse of Theorem 4.28.

Lemma 4.29. *Let $\equiv$ be an equivalence relation on $\mathcal{N}$. If $\equiv \not\subseteq$ no-mismatch, then $\mathcal{B}(\Gamma/{\equiv})$ is not prefix correct.*

Proof. Let $\nu = x_b d_i(\begin{smallmatrix}\alpha\\u_b\end{smallmatrix}\bullet\begin{smallmatrix}\alpha'\\u'_b\end{smallmatrix})r_i x'_b$ and $\nu' = y_b d_j(\begin{smallmatrix}\beta\\v_b\end{smallmatrix}\bullet\begin{smallmatrix}\beta'\\v'_b\end{smallmatrix})r_j y'_b$ be two positions related by $\equiv$ but not by no-mismatch: first-mismatch$(\alpha' r_i\widehat{x'_b}, \beta' r_j \widehat{y'_b}) = (X\widehat{z_b}, Y\widehat{z'_b})$ for some symbols X and Y with X mismatch Y and for some bracketed strings $\widehat{z_b}$ and $\widehat{z'_b}$ in $(V\cup T'_r)^*$. Thus $\alpha' r_i\widehat{x'_b} = \widehat{w_b}X\widehat{z_b}$ and $\beta' r_j\widehat{y'_b} = \widehat{w_b}Y\widehat{z'_b}$ for some bracketed string $\widehat{w_b}$ in $(V\cup T_r)^*$.

At least one of the two symbols X or Y is not in T_r; we assume it is X and we define χ as X if X is a terminal in T or d_k if X is a nonterminal in N and k one of its productions. Note that if Y is also a nonterminal in N, it cannot be the leftpart of rule k.

Since $\nu \equiv \nu'$, there exists two states $q = [\nu]_\equiv = [\nu']_\equiv$ and q' in $\Gamma/{\equiv}$ and two sequences of rules φ and φ' in P'^* such that the rewrite

$$\$q'_s\|x_b d_i u_b w_b \chi z_b\$ \models^* \$\varphi\varphi' q\|w_b\chi z_b\$ \models^* \$\varphi q'\|z_b\$ \tag{4.45}$$

holds in $\mathcal{B}(\Gamma/{\equiv})$. Observe that the sequence of rules in φ' matches the symbols of T_r in $\widehat{w_b}$ in reverse order. But then the rewrite

$$\$q'_s\|y_b d_j v_b w_b \chi z_b\$ \models^* \$\varphi''\varphi' q\|w_b\chi z_b\$ \models^* \$\varphi'' q'\|z_b\$ \tag{4.46}$$

is also possible for some sequence of rules φ'' in P'^*, although $y_b d_j v_b w_b \chi$ is not the prefix of any sentence in $\mathcal{L}(\mathcal{G}'_b)$. □

Theorem 4.30. *Let $\mathcal{G}$ be a grammar. There exists a FSA $\mathcal{A}$ over the bracketed alphabet V_b such that $\mathcal{B}(\mathcal{A})$ is prefix correct and $\mathcal{L}(\mathcal{B}(\mathcal{A})) = \mathcal{L}(G'_b)$ if and only if there exists an equivalence relation $\equiv$ of finite index over $\mathcal{N}$ with $\equiv \subseteq \mathsf{no\text{-}mismatch}$.*

Proof. Let $\mathcal{A}$ be a FSA such that $\mathcal{L}(\mathcal{B}(\mathcal{A})) = \mathcal{L}(G'_b)$; then $\mathcal{L}(G_b) \subseteq \mathcal{L}(\mathcal{A})$ or some sentences of $\mathcal{G}_b$ would be missing in $\mathcal{L}(\mathcal{B}(\mathcal{A}))$. Let us consider $\mathsf{pa}_\mathcal{A}$, which is of finite index. By Proposition 4.17, any path in $\Gamma/\mathsf{pa}_\mathcal{A}$ is also possible in $\mathcal{A}$ and thus $\mathcal{B}(\Gamma/\mathsf{pa}_\mathcal{A})$ is prefix correct. By Lemma 4.29, we conclude that $\mathsf{pa}_\mathcal{A} \subseteq \mathsf{no\text{-}mismatch}$.

Conversely, if there exists $\equiv$ of finite index with $\equiv \subseteq \mathsf{no\text{-}mismatch}$, then by Theorem 4.28, $\mathcal{B}(\Gamma/{\equiv})$ is prefix correct. Finally, when we combine Theorem 4.11 with Lemma 4.26, we obtain that $\mathcal{L}(\mathcal{B}(\Gamma/{\equiv})) = \mathcal{L}(G'_b)$. □

Theorem 4.30 illustrates how our framework can be employed to characterize specific properties of formal machines. This general approach could be used to find under which conditions an Earley parser or a shift-reduce parser build from a position automaton (as in Section 4.2.2) would be correct. Furthermore, if we were fortunate enough for these conditions to be met by a *coarsest* position equivalence $\equiv$, one could show the optimality of parsing with $\Gamma/{\equiv}$.

4.4 Related Models

The position automata we introduced in this chapter can be seen as a generalization of many similar representations defined in different contexts. Several of these representations match Γ/item_0 closely, for instance $\vee C$-flow graphs (Gorn, 1963), transition diagrams (Conway, 1963) and networks (Woods, 1970), and item graphs (Farré and Fortes Gálvez, 2001).

We overview here some approaches that allow different levels of precision in the approximation process, as we feel that this is the most important feature of an approximation framework.

4.4.1 Recognition Graphs

Schöbel-Theuer (1994) presents an unifying theory of context-free parsing, based on a *recognition graph* that recognizes the language of the left-sentential forms of the grammar, i.e. of the sentential forms obtained through leftmost derivations from the axiom. A recognition graph is, as one might suspect, a recognitive device for the language of the grammar at hand, and different methods can be employed to generate such graphs that mimic classical parsing techniques like Earley recognition or shift-reduce recognition.

In order to generate finite recognition graphs, Schöbel-Theuer presents a "compression theorem", and indicates the possibility of using different equivalence relations on the graphs' nodes, corresponding to different item models in chart parsing. Schöbel-Theuer further introduces "macros" as precomputed components on recognition graphs, thus allowing him to express nondeterministic LR parsing in his framework. Macro graphs are quotiented recognition graphs as well.

Above all, this work differs with our own on its motivation: recognition graphs are always seen as recognitive devices and not as tools for the static analysis of the grammar. For instance, the various compressions and macro definitions match parsing strategies rather than attempt to produce different levels of approximation. In the same way, the absence of bracketings makes recognition graphs rather unfit for analysis purposes, because too much information can be lost during quotienting.

4.4.2 Parsing Schemata

Sikkel (1997) presents a general model for chart parsing algorithms in the form of deduction systems—the *parsing schemata*—similar to the Earley one defined in Section A.1.6 on page 167. Items in such deduction systems are defined formally as congruence classes of derivation trees that yield a substring of the input to recognize. Parsing schemata can be manipulated directly through refinements, generalizations and filterings, in ways similar to the operations possible on position equivalences.

Parsing schemata are wholly dedicated to the specification of chart parsing algorithms. Instantiated schema items are not sets of positions but sets of partial trees, tied to a subrange of the input string: let $w = a_1 \cdots a_n$ be the input string, an Earley item $(A \rightarrow \alpha \bullet \alpha', k, l)$ is seen as the set of all trees rooted in A, with $a_{k+1} \cdots a_l$ as the yield of the α children, and with α' as the remaining children. This is similar in effect to the equivalence classes defined by the position equivalence $\mathsf{item}_0 \wedge \mathsf{earley}_w$, where earley_w is defined by

$$x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b} \bullet \genfrac{}{}{0pt}{}{\alpha'}{u'_b}) r_i x'_b \ \mathsf{earley}_w \ y_b d_j(\genfrac{}{}{0pt}{}{\beta}{v_b} \bullet \genfrac{}{}{0pt}{}{\beta'}{v'_b}) r_j y'_b \text{ iff } u = v = a_{k+1} \cdots a_l. \tag{4.47}$$

But if the tree sets of parsing schemata are defined in function of the input string, our position equivalences are committed to a left-to-right processing order. The greater flexibility of parsing schemata in this regard allows Sikkel to describe parallel parsing algorithms, where the deduction rules do not enforce any specific processing order.

Nevertheless, this flexibility comes with a price: in order for the parsing schemata to be correct, items should be regular congruence classes for the deduction rules, and the schemata should define all the valid items (completeness) and only valid items (soundness). As explained by Sikkel (1998), soundness can be proven by induction on the deduction rules, but proving completeness is often less simple. With position equivalences however, completeness is usually straightforward as well because the deduction system merely produces a correct bracketing for the sentences it runs through $\Gamma/\equiv$, and $\mathcal{L}(\mathcal{G}_b) \subseteq \mathcal{L}(\Gamma/\equiv)$ is already known.

4.4.3 Regular Approximations

A series of articles (Nederhof, 2000a,b; Mohri and Nederhof, 2001) presents grammatical transformations that result in right-linear grammars that generate a superlanguage of the language generated by the original grammar. The approximation is roughly equivalent to that of item_0, except that it avoids the unnecessary approximation of the language of nonterminals that are not mutually recursive with self-embedding. The approximations can be refined by unfolding some self-embedded nonterminals (Nederhof, 2000b).

Regular approximations, using these grammar transformations or other means, have been used for instance in two-pass parsing algorithms (Čulik and Cohen, 1973; Nederhof, 1998; Boullier, 2003a) or for ambiguity detection (Brabrand et al., 2007).

4.4.4 Abstract Interpretation

Abstract interpretation (Cousot and Cousot, 1977, 2003, 2007) is a unified model for building approximations. The abstract interpretation framework is based on an abstraction function α from a concrete domain to an abstract domain, for instance from the domain of position graphs to the domain of position automata:

$$\alpha_{\equiv}(\Gamma) = \{[\nu]_{\equiv}\chi \vdash [\nu']_{\equiv} \mid \nu \xrightarrow{\chi} \nu' \in \Gamma\}. \tag{4.48}$$

Concrete computations in the concrete domain, for instance $\mathcal{L}(\Gamma)$, can be approached by abstract computations, for instance $\mathcal{L}(\Gamma/\equiv)$, that enforce some soundness properties, for instance $\mathcal{L}(\Gamma) \subseteq \mathcal{L}(\Gamma/\equiv)$. One can attempt to find a best approximation among all the possible sound ones, just like we can try to find a best position equivalence among all the possible sound ones. Our framework for grammar approximations is a specialized form of abstract interpretation.

4.4.5 Systems of Pushdown Automata

Systems of pushdown acceptors were defined by Kuich (1970) as a representation of CFGs amenable to testing for *quasi determinism.* If a system is not immediately quasi deterministic, one of its language and ambiguity preserving transformations can be tested instead. The quasi determinism of a system or of one of its transformations implies the unambiguity of the original grammar.

The approach of Kuich is very close to our own: systems of pushdown acceptors are another interpretation of Γ/item_0, and transformations provide the same flexibility as the choice of an equivalence relation finer than item_0 for us. Kuich presents extensive transformations, including determinization and identification of which nonterminals display self embedding or not.

4.4.6 Item Grammars

Heilbrunner (1981) introduced LR(k) regular *item grammars* in his formalization of LR(k) parsing. Such grammars were also used by Knuth (1965) to compute the LR(k) contexts and in automata form by Hunt III et al. (1974) to test whether a grammar is LR(k) or not. Heilbrunner (1983) presented later item grammars refined for LR-Regular testing. Item grammars correspond to refinements of Γ/item_0.

Parser Generation 5

It had begun with a leaf caught in the wind, and it became a tree; and the tree grew, sending innumerable branches, and thrusting out the most fantastic roots.

J. R. R. Tolkien, *Leaf by Niggle*

As seen with the case studies of Chapter 3, the developments in parsing techniques have not quite answered all the issues that a parser developer has to face. Between the shortcomings of classical LALR(1) parser generators like YACC (Johnson, 1975) and the catch of general parsing (Earley, 1970; Tomita, 1986)—namely the absence of ambiguity detection—, possibilities are pretty scarce. This thesis advocates an almost forgotten way to deal with nondeterminism: the usage of *noncanonical parsers* (already introduced in Section 3.2.3 on page 36). Noncanonical parsers have been thoroughly investigated on a theoretical level by Szymanski and Williams (1976).

Surprisingly, there are very few practical noncanonical parsing methods. Indeed, the only one of clear practical interest is the extension to SLR(1) parsing described by Tai (1979). Considering the three requirements on parsers for programming languages we identified on page 16, noncanonical parsers perform rather well. Two obstacles seem to prevent a wider adoption of noncanonical parsing techniques:

1. there are very few usable noncanonical parser constructions, the Noncanonical SLR(1) one excepted, and the latter is not sufficient even in rather simple cases (see Section 5.2.6.2 on page 112);

2. with a noncanonical parser, the lookahead length issues do not disappear; they are replaced by reduced lookahead length issues, which are just as frustrating (see Section 5.2.6.3 on page 113).

The two parsing methods we describe here target these two issues.

First, we present shortly how to generate classical, canonical, bottom-up parsers from a position automaton (Section 5.1). The flexibility offered by position automata allows for trade-offs between descriptional complexity on the one hand, and robustness against nondeterminism on the other hand.

We consider thereafter how one such canonical parser can be mapped to a noncanonical one in Section 5.2, through the example of Noncanonical LALR(1) parsing. The formulation is general enough to embrace the cases discussed in the first section. This generic NLALR(1) construction improves on NSLR(1) but remains practicable; we detail an efficient construction technique adapted from the LALR(1) construction technique of DeRemer and Pennello (1982) in Section 5.2.3.

At last, we investigate an original parsing technique, shift-resolve parsing, developed jointly with José Fortes Gálvez and Jacques Farré (Section 5.3). Shift-resolve parsing is a novel parsing method with an attractive combination of properties: the produced parsers are deterministic, they can use an unbounded lookahead, and they run in linear time. Their generation is highly generic as it relies on an underlying position automaton.

5.1 Parser Generation from Position Automata

Classical LR parsers are the product of an important observation (Knuth, 1965): the stack language associated with each correct reduction—or *handle*—of the crude, nondeterministic, shift-reduce parser of Definition A.12 on page 167 is a regular language. For each handle, one can construct a finite state automaton that recognizes the compatible stacks. The union of all these automata, the canonical LR automaton, is a handle-finding automaton.

5.1.1 Subset Construction

We present in this section how reduction-finding automata can be generated from position automata. A reduce action of the shift-reduce parser of Section 4.2.2.1 on page 60 is triggered by a reduction transition on some symbol r_i in the position automaton. The generation of a reduction-finding automaton thus corresponds to the determinization of the position automaton, up to r_i transitions.

5.1.1.1 Sets of Reachable States

A deterministic automaton is classically obtained by a *subset construction* (Rabin and Scott, 1959) that associates states of the position automaton that can be reached by reading the same string in the same state set. Let $\mathsf{Reachb}_{\equiv}(\delta_d)$ denote the set of states of a position automaton $\Gamma/{\equiv} = \langle Q, V_b, R, Q_s, Q_f \rangle$ that can be reached from a state in Q_s upon reading a bracketed string δ_d in $(V \cup T_d)^*$:

$$\mathsf{Reachb}_{\equiv}(\delta_d) = \{q \in Q \mid \exists q_s \in Q_s, q_s\delta_d \models^* q\}. \tag{5.1}$$

We extend this function to take a string in V^* as argument by defining

$$\mathsf{Reach}_{\equiv}(\delta) = \bigcup_{\delta_d \in (V \cup T_d)^*, h(\delta_d) = \delta} \mathsf{Reachb}_{\equiv}(\delta_d). \tag{5.2}$$

The reachable states can be computed inductively by

$$\mathsf{Kernel}_{\equiv}(\varepsilon) = Q_s, \tag{5.3}$$

$$\mathsf{Kernel}_{\equiv}(\gamma X) = \{q' \in Q \mid \exists q \in \mathsf{Reach}_{\equiv}(\gamma), qX \vdash q'\}, \tag{5.4}$$

$$\mathsf{Reach}_{\equiv}(\gamma) = \mathsf{Kernel}_{\equiv}(\gamma) \cup \{q' \in Q \mid \exists q \in \mathsf{Reach}_{\equiv}(\gamma), \exists i \in P, qd_i \vdash q'\}. \tag{5.5}$$

Observe that the $\mathsf{Reach}_{\equiv}$ function defines an equivalence relation between strings in V^*: according to this equivalence, two strings sharing the same set of reachable states are related. We overload $\equiv$ to denote this equivalence relation over V^* by

$$\delta \equiv \gamma \text{ iff } \mathsf{Reach}_{\equiv}(\delta) = \mathsf{Reach}_{\equiv}(\gamma). \tag{5.6}$$

Equivalence classes of $\equiv$ over V^* are thus sets of strings that can reach the same states of $\Gamma/{\equiv}$; these equivalence classes in $V^*/{\equiv}$, when non-empty, make the gist of a reduction-finding automaton RFA($\equiv$).

We therefore denote explicitly the states of a reduction-finding automaton as $[\delta]_{\equiv}$, where δ is a string in V^* in the equivalence class $[\delta]_{\equiv}$. Each state of form $[\delta X]_{\equiv}$ has a unique *accessing symbol* X. A pair $([\delta], X)$ in $V^*/{\equiv} \times V$ is a *transition* if and only if $[\delta X]_{\equiv}$ is not the empty equivalence class. We define the *reduction-finding automaton* RFA($\equiv$) as a FSA with set of states $V^*/{\equiv}$, vocabulary V, set of rules $\{[\delta]_{\equiv}X \vdash [\delta X]_{\equiv} \mid [\delta X]_{\equiv} \neq \emptyset\}$, initial states set $\{[\varepsilon]_{\equiv}\}$, and final states set $\{[\delta]_{\equiv} \mid \exists q \in \mathsf{Reach}_{\equiv}(\delta), \exists q' \in Q, \exists i \in P, qr_i \vdash q'\}$.

5.1.1.2 Canonical LR Automata

We show that the $\mathsf{Reach}_{\mathsf{item}_k}$ sets are exactly the Valid_k sets of Definition A.13 on page 168 when one identifies each LR(k) item with an equivalence class of item_k.

Lemma 5.1. *Let $\nu = x_b d_i(\,{}^{\alpha}_{u_b} \bullet {}^{\alpha'}_{u'_b})r_i x'_b$ be a position in $\mathcal{N}$ and γ_d a bracketed string in $(V \cup T_d)^*$. If $q_s\gamma_d \models^* [\nu]_{item_k}$ in $\Gamma/item_k$, then $[A \xrightarrow{i} \alpha \bullet \alpha', k : x']$ is a valid LR(k) item for γ, i.e. $S \underset{\text{rm}}{\Longrightarrow}^* \delta A z \underset{\text{rm}}{\overset{i}{\Longrightarrow}} \delta\alpha\alpha' z = \gamma\alpha' z$ with $k : x' = k : z$ holds in $\mathcal{G}$.*

Proof. We proceed by induction on the length $|\gamma_d|$. If $\gamma_d = \varepsilon$, then $[S \rightarrow \bullet \alpha', \varepsilon]$ is a valid LR(k) item for $\gamma = \varepsilon$. Let us consider for the induction step a position ν' such that $\nu \overset{\chi}{\rightarrowtail} \nu'$ with χ in $V \cup T_d$. Then, $q_s\gamma\chi \models^* [\nu]_{\mathsf{item}_k}\chi \models [\nu']_{\mathsf{item}_k}$, and, by induction hypothesis, $S \underset{\text{rm}}{\Longrightarrow}^* \delta A z \underset{\text{rm}}{\overset{i}{\Longrightarrow}} \delta\alpha\alpha' z = \gamma\alpha' z$ with $k : x' = k : z$ holds in $\mathcal{G}$.

If $\chi = d_j$, then $\alpha' = B\alpha''$ and ν' is of form $x_b d_i u_b d_j(\,\bullet {}^{\beta}_{v_b})r_j u''_b r_i x'_b$ with $u'_b = v_b u''_b$ for some $j = B \rightarrow \beta$ in P. Then, $\gamma\alpha' z = \gamma B\alpha'' z \underset{\text{rm}}{\Longrightarrow}^* \gamma B u'' z \underset{\text{rm}}{\overset{j}{\Longrightarrow}} \gamma\beta u'' z$ holds in $\mathcal{G}$, and $k : u'' x' = k : u'' z$. Thus, for any ν'' in $[\nu']_{\mathsf{item}_k}$, the corresponding item $[B \xrightarrow{j} \bullet \beta, k : u'' x']$ is a valid LR(k) item for γ.

If $\chi = X$ is in V, then $\alpha' = X\alpha''$ and ν' is of form $x_b d_i(\,{}^{\alpha X}_{u_b v_b} \bullet {}^{\alpha''}_{u''_b})r_i x'_b$. Then, $\gamma\alpha' z = \gamma X \alpha'' z$, and, for any ν'' in $[\nu']_{\mathsf{item}_k}$, the corresponding item $[A \xrightarrow{i} \alpha X \bullet \alpha'', k : x']$ is a valid LR(k) item for γX. □

Theorem 5.2. *Let $\nu = x_b d_i(\,{}^{\alpha}_{u_b} \bullet {}^{\alpha'}_{u'_b})r_i x'_b$ be a position in $\mathcal{N}$ and γ a string in V^*. The state $[\nu]_{item_k}$ is in $\mathsf{Reach}_{item_k}(\gamma)$ if and only if $[A \xrightarrow{i} \alpha \bullet \alpha', k : x']$ is in $\mathsf{Valid}_k(\gamma)$.*

Proof. If $S \underset{\text{rm}}{\Longrightarrow}^* \delta A z \underset{\text{rm}}{\overset{i}{\Longrightarrow}} \delta\alpha\alpha' z = \gamma\alpha' z$ with $k : x' = k : z$ holds in $\mathcal{G}$, then a similar derivation holds in $\mathcal{G}_b$. Thus, by Corollary 4.10 on page 57, $q_s\delta_d d_i \alpha \models^* [\nu']_{\mathsf{item}_k}$ with $\nu' = y_b d_i(\,{}^{\alpha}_{v_b} \bullet {}^{\alpha'}_{v'_b})r_i z_b$ with $\delta_d \Rightarrow_b^* y_b$. Then, ν item_k ν' and therefore $[\nu]_{\mathsf{item}_k} = [\nu']_{\mathsf{item}_k}$.

If $[\nu]_{\mathsf{item}_k}$ is in $\mathsf{Reach}_{\mathsf{item}_k}(\gamma)$, then, by Lemma 5.1, $[A \xrightarrow{i} \alpha \bullet \alpha', k : x']$ is a valid LR(k) item for γ. □

Thus, our reduction-finding automaton RFA(item_k) is a handle-finding automaton, and thus it yields a correct parser.

5.1.2 Parsing Machine

Definition 5.3. Let $\mathcal{G}$ be a grammar and $\Gamma/\!\equiv\, = \langle Q, V_b, R, Q_s, Q_f\rangle$ its position automaton for the equivalence $\equiv$. Let $P(\equiv) = (V^*/\!\equiv \cup T \cup \{\$, \|\}, R_p)$ be a rewriting system (see Definition A.4 on page 164) where \$ and $\|$ (the end marker and the stack top, respectively) are not in $V^*/\!\equiv$ nor in T (the set of states and the input alphabet, respectively). A *configuration* of $P(\equiv)$ is a string of the form

$$[\varepsilon]_\equiv[X_1]_\equiv \dots [X_1 \dots X_n]_\equiv \| x\$ \tag{5.7}$$

where $X_1 \dots X_n$ is a string in V^* and x a string in T^*.

We say that $P(\equiv)$ is a *parsing machine* for grammar $\mathcal{G}$ if its initial configuration is $[\varepsilon]_\equiv\|w\$$ with w the input string in T^*, its final configuration is $[\varepsilon]_\equiv[S]_\equiv\|\$$, and if each rewriting rule in R_p is of one of the forms

- *shift* a in state $[\delta]_\equiv$

$$[\delta]_\equiv\|ax \vdash_{\text{shift}} [\delta]_\equiv[\delta a]_\equiv\|x, \tag{5.8}$$

  defined if there are states q and q' of Q with q in $\mathsf{Reach}_\equiv(\delta)$ and $qa \vdash q'$ in R, and if $[\delta a]_\equiv \neq \emptyset$, or

- *reduce* by rule $i = A{\rightarrow}X_1 \dots X_n$ of P in state $[\delta X_1 \dots X_n]_\equiv$

$$[\delta X_1]_\equiv \dots [\delta X_1 \dots X_n]_\equiv\|x \vdash_{A\rightarrow X_1 \dots X_n} [\delta A]_\equiv\|x, \tag{5.9}$$

  defined if there are states q and q' of Q with q in $\mathsf{Reach}_\equiv(\delta X_1 \dots X_n)$ and $qr_i \vdash q'$ in R, and if $[\delta A]_\equiv \neq \emptyset$.

The corresponding parser for the context-free grammar $\mathcal{G}$ is the transducer $\langle P(\equiv), \tau\rangle$ with output alphabet P where $P(\equiv)$ is a parsing machine for $\mathcal{G}$ and τ is defined by $\tau(\vdash_{\text{shift}}) = \varepsilon$ and $\tau(\vdash_{A\rightarrow X_1 \dots X_n}) = A{\rightarrow}X_1 \dots X_n$.

This definition translates the definitions of Section A.2 on page 168 to the realm of position automata. Note that, as a simplification, we did not employ lookaheads in our definition. But they could be taken into account, by defining the language of a FSA state q as $\mathcal{L}(q) = \{w \mid \exists q_f \in Q_f, qw \vDash^* q_f\}$, and by adding the requisite $k : x \in k : h(\mathcal{L}(q'))$ to the conditions of the rules (5.8) and (5.9). We will not prove that the obtained parsing machine with lookaheads are correct, but the results of Section 6.2.3.2 on page 143 seem to indicate that it is the case.

5.1.3 Strict Deterministic Parsers

Geller and Harrison (1977a,b) describe a variant of LR(k) parsing where LR(k) items of form $[A{\rightarrow}{\bullet}\alpha, x]$ with x in $\mathsf{Follow}_k(A)$ can be arbitrarily introduced during the construction of the parsing automaton. Observe that, when we reach a state with $[A{\rightarrow}\alpha{\bullet}, x]$ further in the construction, it cannot result in a reduction unless both the lookahead window and the goto by A are compatible with the current configuration of the parser. Some of the precision of the canonical LR(k) parser, and the correct prefix property, can be relinquished in exchange of a more compact parser. The resulting *characteristic parsers* are correct under the assumption that $\alpha \neq \varepsilon$.

We do not attempt to reproduce this parameterized construction here, but only show how one can construct parsers for strict deterministic grammars.

5.1.3.1 Strict Deterministic Grammars

Geller and Harrison (1977b) describe a simpler class of characteristic parsers build for *strict deterministic grammars* (Harrison and Havel, 1973). Such grammars have a *strict* partition Π on V such that

1. T is an equivalence class of Π, and
2. for any A and B in N and α, β and β' in V^*, if $A \rightarrow \alpha\beta$ and $B \rightarrow \alpha\beta'$ and $A \equiv B \pmod \Pi$, then either
   (a) both β and β' are non null, and $1 : \beta \equiv 1 : \beta' \pmod \Pi$, or
   (b) $\beta = \beta' = \varepsilon$ and $A = B$.

Definition 5.4. [Geller and Harrison (1977b)] Let A be a nonterminal in N. The string α in V^* is a *production prefix* of A if there exists some production $A \rightarrow \alpha\alpha'$ in P for some α' in V^*. The definition is extended to sets of nonterminals $N' \subseteq N$ by saying that α is a production prefix of N' if there exists A in N' such that α is a production prefix of A.

Let $\mathcal{G}$ be a strict deterministic grammar with strict partition Π over V. Then, the item $[A \rightarrow \alpha \bullet \alpha']$ is a *valid strict deterministic item* for $\gamma\alpha$, if there exist a number $n \geq 0$, a decomposition $\gamma = \delta_1 \cdots \delta_n$, and nonterminals $A_0, \ldots, A_n$ such that $A_0 \equiv S \pmod \Pi$, $A_n \equiv A \pmod \Pi$, and

$$\begin{array}{rl}
\delta_1 A_1 & \text{is a production prefix of } [A_0]_\Pi \\
\delta_2 A_2 & \text{is a production prefix of } [A_1]_\Pi \\
& \vdots \\
\delta_n A_n & \text{is a production prefix of } [A_{n-1}]_\Pi \\
\alpha & \text{is a production prefix of } [A_n]_\Pi
\end{array} \tag{5.10}$$

If $n = 0$, this definition reduces to "α is a production prefix of $[A_0]_\Pi$."

5.1.3.2 sd_Π Position Equivalence

As with LR(k) parsing, we define a position equivalence that mirrors the behavior of strict deterministic characteristic parsers. Let sd_Π for Π a strict partition of V be defined by

$$x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b} \bullet \genfrac{}{}{0pt}{}{\alpha'}{u'_b}) r_i x'_b \;\mathsf{sd}_\Pi\; y_b d_j(\genfrac{}{}{0pt}{}{\beta}{v_b} \bullet \genfrac{}{}{0pt}{}{\beta'}{v'_b}) r_j y'_b \text{ iff } \alpha = \beta \text{ and } A \equiv B \pmod \Pi, \tag{5.11}$$

where A and B are the leftparts of $i = A{\rightarrow}\alpha\alpha'$ and $j = B{\rightarrow}\beta\beta'$.

We prove that we construct only valid strict deterministic items with sd_Π in Lemma 5.5. Although the lemma is very similar to Lemma 5.1, a difference is that several items correspond to a single state of Γ/sd_Π.

Lemma 5.5. *Let $\nu = x_b d_i ({}^{\alpha}_{u_b} \bullet {}^{\alpha'}_{u'_b}) r_i x'_b$ be a position in $\mathcal{N}$ and γ_d a bracketed string in $(V \cup T_d)^*$. If $q_s\gamma_d \models^* [\nu]_{\mathsf{sd}_\Pi}$ in Γ/sd_Π, then $[A\overset{i}{\rightarrow}\alpha\bullet\alpha']$ is a valid strict deterministic item for γ.*

Proof. We proceed by induction on the length $|\gamma_d|$. In the base case, $\gamma_d = \varepsilon$, and then $\alpha = u_b = \varepsilon$ and $i = S{\rightarrow}\alpha'$, thus we can choose $n = 0$ in Definition 5.4: ε is a production prefix of $[S]_\Pi$, and $S \equiv S \pmod{\Pi}$ by reflexivity.

For the induction step, we consider a position ν' such that $\nu \overset{\chi}{\rightarrowtail} \nu'$ with χ in $V \cup T_d$. Then, $q_s\gamma\chi \models^* [\nu]_{\mathsf{sd}_\Pi}\chi \models [\nu']_{\mathsf{sd}_\Pi}$, and, by induction hypothesis, there exist $n \geq 0$, a decomposition $\gamma = \delta_1 \cdots \delta_n \alpha$, and nonterminals $A_0,\ldots,A_n$ such that $A_0 \equiv S \pmod{\Pi}$, $A_n \equiv A \pmod{\Pi}$ with $i = A{\rightarrow}\alpha\alpha'$, and (5.10) hold.

If $\chi = d_j$**,** then $\alpha' = B\alpha''$ and ν' is of form $x_b d_i u_b d_j (\bullet {}^{\beta}_{v_b}) r_j u''_b r_i x'_b$ with $u'_b = v_b u''_b$ for some $j = B{\rightarrow}\beta$ in P. Then, let $n' = n + 1$, $A_{n+1} = B$; αB is a production prefix for A, and ε is one for B. Thus, for any ν'' in $[\nu']_{\mathsf{sd}_\Pi}$, the corresponding item $[C{\rightarrow}\bullet\delta]$ is such that $C \equiv B \pmod{\Pi}$, and is thus a valid strict deterministic item for γ.

If $\chi = X$ is in V, then $\alpha' = X\alpha''$ and ν' is of form $x_b d_i ({}^{\alpha X}_{u_b v_b} \bullet {}^{\alpha''}_{u''_b}) r_i x'_b$. Then, αX is also a production prefix for $[A_n]_\Pi$, and, for any ν'' in $[\nu']_{\mathsf{item}_k}$, the corresponding item $[B{\rightarrow}\alpha X\bullet\beta]$ is such that $B \equiv A \pmod{\Pi}$, and is a valid strict deterministic item for γX. □

Lemma 5.6. *If $[A\overset{i}{\rightarrow}\alpha\bullet\alpha']$ is a valid strict deterministic item for a string γ in V^*, then there exist γ_d a bracketed string in $(V \cup T_d)^*$ with $h(\gamma_d) = \gamma$ and ν a position of form $x_b d_i ({}^{\alpha}_{u_b} \bullet {}^{\alpha'}_{u'_b}) r_i x'_b$ in $\mathcal{N}$ such that $q_s\gamma_d \models^* [\nu]_{\mathsf{sd}_\Pi}$ holds in Γ/sd_Π.*

Proof. We sketch the induction on $n+|\gamma|$. If this number is 0, then $\alpha = \varepsilon$ and any item such that $A \equiv S \pmod{\Pi}$ is such that there exists a corresponding position in q_s the only initial state of Γ/sd_Π. For the induction step, we can increase this number either by increasing n or $|\gamma|$. In the first case, it corresponds to following a d_j transition, and in the second to following an X transition in Γ/sd_Π. □

Combining Lemma 5.5 and Lemma 5.6, we see that our construction of a reduction-finding automaton using sd_Π is correct in that it generates a handle-finding automaton if Π is a strict partition (Geller and Harrison, 1977b). A strict deterministic parser can be as much as exponentially smaller than the LR(0) parser for the same grammar.

5.2 Noncanonical LALR(1) Parsers

We present in this section the construction of Noncanonical LALR(1) (thereafter NLALR(1)) parsers from a context-free grammar. LALR parsers were introduced by DeRemer (1969) and Anderson (1972) as practical parsers for deterministic languages. Rather than building an exponential number of LR(k) states, LALR(k) parsers add lookahead sets to the actions of the small LR(0) parser.

LALR(1) Parsing Compared to SLR(1) parsers, LALR(1) machines base their decisions on the current LR(0) state, and thus on an approximation of the input read so far, instead of on a Follow set regardless of the context. They are therefore sensibly less prone to nondeterminism than SLR(1) parsers; we will illustrate this in Section 5.2.6.2 on page 112.

The LALR(1) construction relies heavily on the LR(0) automaton. This automaton provides a nice explanation for LALR lookahead sets: the symbols in the lookahead set for some reduction are the symbols expected next by the LR(0) parser, should it really perform this reduction. A formal definition of LALR(1) parsers can be found in Section A.2.3 on page 169.

Our specific choice of extending LALR(1) parsers is motivated by their wide adoption, their practical relevance, and the existence of efficient and broadly used algorithms for their generation, notably those by Kristensen and Madsen (1981) and DeRemer and Pennello (1982). We express our noncanonical computations in the latter framework and obtain a simple and efficient practical construction.

The additional complexity of generating a NLALR(1) parser instead of a LALR(1) or a NSLR(1) one, as well as the increase of the parser size and the overhead on parsing performances are all quite small. Therefore, the improved parsing power comes at a fairly reasonable price.

5.2.1 Example Parser

Consider for instance grammar $\mathcal{G}_{12}$ with rules

$$S \rightarrow BC \mid AD, A \rightarrow a, B \rightarrow a, C \rightarrow CA \mid A, D \rightarrow aD \mid b, \qquad (\mathcal{G}_{12})$$

generating the language $\mathcal{L}(\mathcal{G}_{12}) = aa^+ \mid aa^*b$.

The state q_1 in the automaton of Figure 5.1 on the facing page is *inadequate*: the parser is unable to decide between reductions $A \rightarrow a$ and $B \rightarrow a$

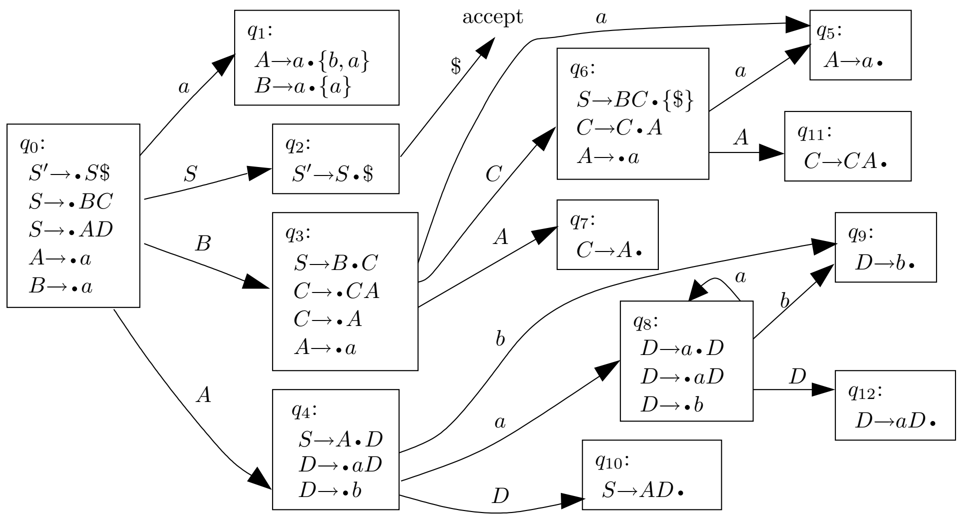

q_0: $S' \to \bullet S\$$ $S \to \bullet BC$ $S \to \bullet AD$ $A \to \bullet a$ $B \to \bullet a$
q_1: $A \to a \bullet \{b, a\}$ $B \to a \bullet \{a\}$
q_2: $S' \to S \bullet \$$
q_3: $S \to B \bullet C$ $C \to \bullet CA$ $C \to \bullet A$ $A \to \bullet a$
q_4: $S \to A \bullet D$ $D \to \bullet aD$ $D \to \bullet b$
q_5: $A \to a \bullet$
q_6: $S \to BC \bullet \{\$\}$ $C \to C \bullet A$ $A \to \bullet a$
q_7: $C \to A \bullet$
q_8: $D \to a \bullet D$ $D \to \bullet aD$ $D \to \bullet b$
q_9: $D \to b \bullet$
q_{10}: $S \to AD \bullet$
q_{11}: $C \to CA \bullet$
q_{12}: $D \to aD \bullet$
accept
a S B A $\$$ C A a b a D a A a b D

Figure 5.1: The LALR(1) automaton for $\mathcal{G}_{12}$.

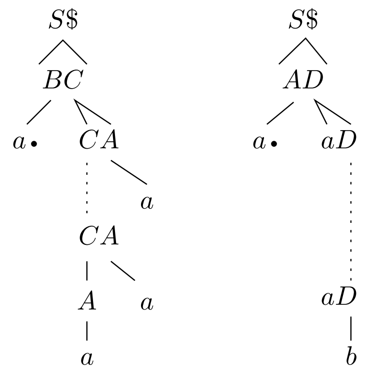


Figure 5.2: The conflict position of state q_1 in the derivation trees of $\mathcal{G}_{12}$.

when the lookahead is a. We see on the derivation trees of Figure 5.2 that, in order to choose between the two reductions, the parser has to know if there is a b at the very end of the input. This need for an unbounded lookahead makes $\mathcal{G}_{12}$ non-LR. A parser using a regular lookahead would solve the conflict by associating the distinct regular lookaheads a^*b and $a^+\$$ with the reductions to A and B respectively.

5.2.1.1 Noncanonical Parsing

At the heart of a noncanonical parser for $\mathcal{G}_{12}$ is the observation that a single lookahead symbol (D or C) is enough: if the parser is able to explore the context on the right of the conflict, and to reduce some other phrases, then, at some point, it will reduce this context to a D (or a C). When coming back to the conflict point, it will see a D (resp. a C) in the lookahead window. The grammar clearly allows an A (resp. a B) in front of an a in state q_1 only if this a was itself produced by a D (resp. a C).

Table 5.1 on the facing page presents a noncanonical parse for a string in $\mathcal{L}(\mathcal{G}_{12})$. A bottom-up parser reverses the derivation steps which lead to the terminal string it parses. The reversal of a derivation $Aaa \Rightarrow aaa$ is the reduction of the *phrase* a in the sentential form aaa to the nonterminal A. For most bottom-up parsers, including LALR ones, these derivations are rightmost, and therefore the reduced phrase is the leftmost one, called the *handle* of the sentential form. Noncanonical parsers allow the reduction of phrases which may not be handles (Aho and Ullman, 1972): in our example, it reverses the derivation $aaA \Rightarrow aaa$.

The noncanonical machine is not very different from the canonical one, except that it uses two stacks. The additional stack, the *input stack*, contains the (possibly reduced) right context, whereas the other stack is the classical *parsing stack*. Reductions push the reduced nonterminal on top of the input stack. There is no goto operation *per se*: the nonterminal on top of the input stack either allows a parsing decision which had been delayed, or is simply shifted.

5.2.1.2 Construction Principles

Let us compute the lookahead set for the reduction $A{\rightarrow}a$ in state q_1. Should the LR(0) parser decide to reduce $A{\rightarrow}a$, it would pop q_1 from the parsing stack (thus be in state q_0), and then push q_4. We read directly on Figure 5.1 on the previous page that three symbols are acceptable in q_4: D, a and b. Similarly, the reduction $B{\rightarrow}a$ in q_1 has $\{C, A, a\}$ for lookahead set, read directly from state q_3.

The intersection of the lookahead sets for the reductions in q_1 is not empty: a appears in both, which means a conflict. Luckily enough, a is not a *totally reduced symbol*: D and C are reduced symbols, read from kernel items in q_4 and q_3. The conflicting lookahead symbol a could be reduced, and later we might see a symbol on which we can make a decision instead. Thus, we shift the lookahead symbol a in order to reduce it and solve the conflict later. All the other symbols in the computed lookaheads allow to make a decision, so we leave them in the lookaheads sets, but we remove a from both sets.

Table 5.1: The parse of the string aaa by the NLALR(1) parser for $\mathcal{G}_{12}$.

parsing stack	input stack	actions
q_0	$aaa\$$	shift
q_0q_1	$aa\$$	shift

The inadequate state q_1 is reached with lookahead a. The decision of reducing to A or B can be restated as the decision of reducing the right context to D or C. In order to perform the latter decision, we shift a and reach the state s_1 of Equation 5.12 where we now expect a^*b and $a^*\$$. We are pretty much in the same situation as before: s_1 is also inadequate. But we know that in front of b or $\$$ a decision can be made:

$q_0q_1s_1$	$a\$$	shift

There is a new conflict between the reduction $A{\rightarrow}a$ and the shift of a to a position $D{\rightarrow}a\bullet D$. We also shift this a. The expected right contexts are still a^*b and $a^*\$$, so the shift brings us again to s_1:

$q_0q_1s_1s_1$	$\$$	reduce using $A{\rightarrow}a$

The decision is made in front of $\$$. We reduce the a represented by s_1 on top of the parsing stack, and push the reduced symbol A *on top* of the input stack:

$q_0q_1s_1$	$A\$$	reduce using $A{\rightarrow}a$

Using this new lookahead, the parser is able to decide another reduction to A:

q_0q_1	$AA\$$	reduce using $B{\rightarrow}a$

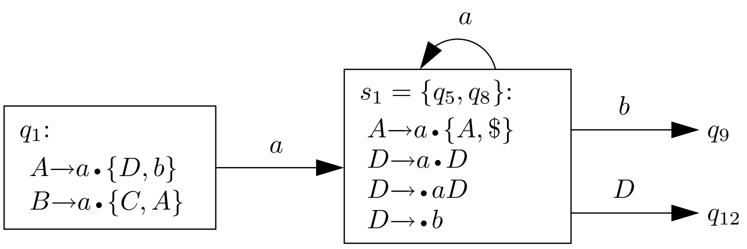
We are now back in state q_1. Clearly, there is no need to wait until we see a completely reduced symbol C in the lookahead window: A is already a symbol specific to the reduction to B:

q_0	$BAA\$$	shift
q_0q_3	$AA\$$	shift
$q_0q_3q_7$	$A\$$	reduce using $C{\rightarrow}A$
q_0q_3	$CA\$$	shift
$q_0q_3q_6$	$A\$$	shift
$q_0q_3q_6q_{11}$	$\$$	reduce using $C{\rightarrow}CA$
q_0q_3	$C\$$	shift
$q_0q_3q_6$	$\$$	reduce using $S{\rightarrow}BC$
q_0	$S\$$	shift, and then accept

Shifting a puts us in the same situation we would have been in if we had followed the transitions on a from both q_3 and q_4, since the noncanonical generation simulates both reductions in q_1. We create a noncanonical transition from q_1 on a to a noncanonical state

$$s_1 = \{q_5, q_8\}, \tag{5.12}$$

that behaves as the union of states q_5 and q_8. State s_1 thus allows a reduction



a
q_1:
$A{\rightarrow}a\,{\bullet}\,\{D,b\}$
$B{\rightarrow}a\,{\bullet}\,\{C,A\}$
a
$s_1 = \{q_5, q_8\}$:
$A{\rightarrow}a\,{\bullet}\,\{A,\$\}$
$D{\rightarrow}a\,{\bullet}\,D$
$D{\rightarrow}\,{\bullet}\,aD$
$D{\rightarrow}\,{\bullet}\,b$
b
q_9
D
q_{12}

Figure 5.3: State q_1 extended for noncanonical parsing.

using $A{\rightarrow}a$ inherited from q_5, and the shifts of a, b and D inherited from q_8. We therefore need to compute the lookaheads for reduction using $A{\rightarrow}a$ in q_5. Using again the LR(0) simulation technique, we see on Figure 5.1 on page 89 that this reduction would lead us to either q_7 or to q_{11}. In both cases, the LR(0) automaton would perform a reduction to C that would lead next to q_6. At this point, the LR(0) automaton expects either the end of file symbol \$, should a reduction to S occur, or an A or an a. The complete lookahead set for the reduction $A{\rightarrow}a$ in q_5 is thus $\{A, a, \$\}$.

The new state s_1 is also inadequate: with an a in the lookahead window, we cannot choose between the shift of a and the reduction $A{\rightarrow}a$. As before, we create a new transition on a from s_1 to a noncanonical state

$$s'_1 = \{q_5, q_8\}. \tag{5.13}$$

State q_5 is the state accessed on a from q_6. State q_8 is the state accessed from q_8 if we simulate a shift of symbol a.

State s'_1 is the same as state s_1, and we merge them. The noncanonical computation is now finished. Figure 5.3 sums up how state q_1 has been transformed and extended. Note that we just use the set $\{q_5, q_8\}$ in a noncanonical LALR(1) automaton; items represented in Figure 5.3 are only there to ease understanding.

5.2.2 Definition of NLALR(1) Parsers

There is a number of differences between the LALR(1) and NLALR(1) definitions. The most visible one is that we accept nonterminals in our lookahead sets. We also want to know which lookahead symbols are totally reduced. Finally, we are adding new states, which are sets of LR(0) states. Therefore, the objects in most of our computations will be LR(0) states.

We denote the LR(0) state reached upon reading γ in the canonical LR(0) handle-finding automaton by $[\gamma]$ instead of $[\gamma]_0$ or $[\gamma]_{\mathsf{item}_0}$. Similarly, we denote the Kernel_0 and Valid_0 sets by Kernel and Valid respectively.

5.2.2.1 Valid Covers

We have recalled in Section A.2 on page 168 that LR(0) states can be viewed as collections of valid prefixes. A similar definition for NLALR(1) states would be nice. However, due to the suspended parsing actions, the language of all prefixes accepted by a noncanonical parser is no longer a regular language. This means the parser only has a regular approximation of the exact parsing stack language. The noncanonical states, being sets of LR(0) states, i.e. sets of equivalence classes on valid prefixes, provide this approximation. We therefore define valid covers as valid prefixes covering the parsing stack language.

Definition 5.7. String γ is a *valid cover* in $\mathcal{G}$ for string δ if and only if γ is a valid prefix and $\gamma \Rightarrow^* \delta$. We write $\hat{\delta}$ to denote some cover of δ and $\text{Cover}(L)$ to denote the set of all valid covers for the set of strings L.

Remember for instance configuration $q_0q_1\|aa\$$ from Table 5.1 on page 91. This configuration leads to pushing state $s_1 = \{q_5, q_8\}$, where both valid prefixes $(B|BC)a$ and Aa^*a of q_5 and q_8 are valid covers for the actual parsing stack prefix aa. Thus in s_1 we cover the parsing stack prefix by $(B \mid BC \mid Aa^*)a$.

This definition would be replaced in the more general case of a reduction-finding automaton RFA($\equiv$) by approximated covers, strings in $V \cup T_d$ that allow to reach the same state in $\Gamma/\equiv$ as the covered string does.

5.2.2.2 Noncanonical Lookaheads

Noncanonical lookaheads are symbols in V'. Adapting the computation of the LALR(1) lookahead sets is simple, but a few points deserve some explanations.

First of all, noncanonical lookahead symbols have to be *non null*, i.e. X is non null if $X \Rightarrow^* ax$. Indeed, null symbols do not provide any additional right context information—worse, they can hide it. If we consider that we always perform a reduction at the earliest parsing stage possible, then they never appear in a lookahead window.

Totally Reduced Lookaheads Totally reduced lookaheads form a subset of the noncanonical lookahead set such that none of its elements can be further reduced. A conflict with a totally reduced symbol as lookahead of a reduction cannot be solved by a noncanonical exploration of the right context, since there is no hope of ever reducing it any further.

We define here totally reduced lookaheads as non null symbols that can follow the right part of the offending rule in a leftmost derivation.

Definition 5.8. The set of *totally reduced lookaheads* for a reduction $A{\rightarrow}\alpha$ in LR(0) state q is defined by

$$\mathrm{RLA}(q, A{\rightarrow}\alpha) = \{X \mid S' \underset{\mathrm{lm}}{\Rightarrow}^* zA\gamma X\omega, \gamma \Rightarrow^* \varepsilon, X \Rightarrow^* ax, \text{ and } q = [\hat{z}\alpha]\}.$$

Derived Lookaheads The derived lookahead symbols are simply defined by extending Equation A.5 to the set of all non null symbols in V.

Definition 5.9. The set of *derived lookaheads* for a reduction $A{\rightarrow}\alpha$ in LR(0) state q is defined by

$$\mathrm{DLA}(q, A{\rightarrow}\alpha) = \{X \mid S' \Rightarrow^* \delta AX\omega, X \Rightarrow^* ax, \text{ and } q = [\hat{\delta}\alpha]\}.$$

We obviously have that

$$\mathrm{LA}(q, A{\rightarrow}\alpha) = \mathrm{DLA}(q, A{\rightarrow}\alpha) \cap T'. \tag{5.14}$$

Conflicting Lookahead Symbols At last, we need to compute which lookahead symbols would make the state inadequate. A noncanonical exploration of the right context is required for these symbols. They appear in the derived lookahead sets of several reductions and/or are transition labels. However, the totally reduced lookaheads of a reduction are not part of this lookahead set, for if they are involved in a conflict, then there is no hope of being able to solve it.

Definition 5.10. The *conflicts lookahead set* for a reduction using $A{\rightarrow}\alpha$ in set s of LR(0) states is defined as

$$\begin{aligned}\mathrm{CLA}(s, A{\rightarrow}\alpha) = \{&X \in \mathrm{DLA}(q, A{\rightarrow}\alpha) \mid q \in s, X \notin \mathrm{RLA}(q, A{\rightarrow}\alpha),\\ &(q, X) \text{ or } (\exists p \in s, \exists B{\rightarrow}\beta \neq A{\rightarrow}\alpha \in P, X \in \mathrm{DLA}(p, B{\rightarrow}\beta))\}.\end{aligned}$$

We then define the *noncanonical lookahead set* for a reduction using $A{\rightarrow}\alpha$ in set s of LR(0) states as

$$\mathrm{NLA}(s, A{\rightarrow}\alpha) = \big(\bigcup_{q\in s} \mathrm{DLA}(q, A{\rightarrow}\alpha)\big) - \mathrm{CLA}(s, A{\rightarrow}\alpha).$$

We illustrate these definitions by computing the lookahead sets for the reduction using $A{\rightarrow}a$ in state $s_1 = \{q_5, q_8\}$ as in Section 5.2.1.2 on page 90: $\mathrm{RLA}(q_5, A{\rightarrow}a) = \{A, \$\}$, $\mathrm{DLA}(q_5, A{\rightarrow}a) = \{A, a, \$\}$, $\mathrm{CLA}(s_1, A{\rightarrow}a) = \{a\}$ and $\mathrm{NLA}(s_1, A{\rightarrow}a) = \{A, \$\}$.

5.2.2.3 Noncanonical States

We saw at the beginning of this section that states in the NLALR(1) automaton were in fact sets of LR(0) states. We denote by $[\![\delta]\!]$ the noncanonical state accessed upon reading string δ in V'^*.

Definition 5.11. The *noncanonical state* $[\![\delta]\!]$ is the set of LR(0) states defined by

$$[\![\varepsilon]\!] = \{[\varepsilon]\} \text{ and}$$
$$[\![\delta X]\!] = \{[\widehat{\gamma A X}] \mid X \in \mathrm{CLA}([\![\delta]\!], A{\rightarrow}\alpha), [\gamma\alpha] \in [\![\delta]\!]\} \ \cup\ \{[\varphi X] \mid [\varphi] \in [\![\delta]\!]\}.$$

The *noncanonical transition* from $[\![\delta]\!]$ to $[\![\delta X]\!]$ on symbol X, denoted by $([\![\delta]\!], X)$, exists if and only if $[\![\delta X]\!] \neq \emptyset$. Reduction $([\![\delta]\!], A{\rightarrow}\alpha)$ exists if and only if there exists a reduction $(q, A{\rightarrow}\alpha)$ and q is in $[\![\delta]\!]$.

Note that these definitions remain valid for plain LALR(1) states since, in absence of a conflict, a noncanonical state is a singleton set containing the corresponding LR(0) state.

A simple induction on the length of δ shows that the LR(0) states considered in the noncanonical state $[\![\delta]\!]$ provide a valid cover for any accessing string of the noncanonical state. It basically means that the actions decided in a given noncanonical state make sense at least for a cover of the real sentential form prefix that is read.

The approximations done when covering the actual sentential form prefix are made on top of the previous approximations: with each new conflict, we need to find a new set of LR(0) states covering the parsing stack contents. This stacking is made obvious in the above definition when we write $\widehat{\gamma A X}$. It means that NLALR(1) parsers are not prefix correct, but prefix cover correct.

Throughout this section, we use the LR(0) automaton to approximate the prefix read so far. We could use more powerful methods—but it would not really be in the spirit of LALR parsing any longer; see Section 5.2.4 on page 103 for alternative methods.

5.2.2.4 NLALR(1) Automata

Here we formalize noncanonical LALR(1) parsing machines. They are a special case of two-stack pushdown automata (2PDA). As explained before, the additional stack serves as an input for the parser, and reductions push the reduced nonterminal on top of this stack. This behavior of reductions excepted, the definition of a NLALR(1) automaton is similar to the LALR(1) one.

Definition 5.12. Let $M = (Q \cup V \cup \{\$, \|\}, R)$ be a rewriting system. A *configuration* of M is a string of the form

$$[\![\varepsilon]\!][\![X_1]\!] \dots [\![X_1 \dots X_n]\!] \| \omega \$$$

where $X_1 \dots X_n$ and ω are strings in V^*. We say that M is a *NLALR(1) automaton* if its initial configuration is $[\![\varepsilon]\!] \| w\$$ with w the input string in

T^*, its final configuration is $[\![\varepsilon]\!][\![S]\!]\|\$$, and if each rewriting rule in R is of the form

- *shift* X in state $[\![\delta]\!]$, defined if there is a transition $([\![\delta]\!], X)$
$$[\![\delta]\!]\|X \vdash_{\text{shift}} [\![\delta]\!][\![\delta X]\!]\|,$$

- or *reduce* by rule $A \to X_1 \dots X_n$ of P in state $[\![\delta X_1 \dots X_n]\!]$ with lookahead X, defined if $A \to X_1 \dots X_n$ is a reduction in $[\![\delta X_1 \dots X_n]\!]$ and lookahead X is in $\text{NLA}([\![\delta X_1 \dots X_n]\!], A \to X_1 \dots X_n)$
$$[\![\delta X_1]\!] \dots [\![\delta X_1 \dots X_n]\!]\|X \vdash_{A \to X_1 \dots X_n} [\![\delta]\!]\|AX.$$

The following rules illustrate Definition 5.12 on the previous page on state s_1 of the NLALR(1) automaton for $\mathcal{G}_{12}$:

$$\begin{array}{lll} s_1\|a & \vdash_{\text{shift}} & s_1 s_1\|, \\ s_1\|b & \vdash_{\text{shift}} & s_1\{q_9\}\|, \\ s_1\|D & \vdash_{\text{shift}} & s_1\{q_{12}\}\|, \\ s_1\|A & \vdash_{A \to a} & \|AA, \text{ and} \\ s_1\|\$ & \vdash_{A \to a} & \|A\$. \end{array} \tag{5.15}$$

According to Definition 5.12, NLALR(1) automata are able to backtrack by a limited amount, corresponding to the length of their window, at reduction time only. We know that noncanonical parsers using a bounded lookahead window operate in linear time (Szymanski and Williams, 1976); the following theorem precisely shows that the total number of rules involved in the parsing of an input string is linear in respect with the number of reductions performed, which itself is linear with the input string length. This theorem uses an output effect τ that outputs the rules used for each reduction performed by M; we then call (M, τ) a NLALR(1) parser.

Theorem 5.13. *Let $\mathcal{G}$ be a grammar and (M, τ) its NLALR(1) parser. If π is a parse of w in M, then the number of parsing steps $|\pi|$ is related to the number $|\tau(\pi)|$ of derivation steps producing w in $\mathcal{G}$ and to the length $|w|$ of w by*
$$|\pi| = 2|\tau(\pi)| + |w|.$$

We start by proving the following lemma inspired by Lemma 5.17 from Sippu and Soisalon-Soininen (1990). The proof of Theorem 5.13 is then immediate when choosing w for α, ε for $X_1 \dots X_n$ and β and S for $Y_1 \dots Y_m$ in Lemma 5.14.

Lemma 5.14. *Let $\mathcal{G}$ be a CFG and (M, τ) its NLALR(1) parser. Further, let $X_1 \dots X_n$, $Y_1 \dots Y_m$, α and β be strings in V^* and π an action string in R^* such that*

$$[\![\varepsilon]\!][\![X_1]\!] \dots [\![X_1 \dots X_n]\!] \| \alpha\$ \ \models_{\pi} [\![\varepsilon]\!][\![Y_1]\!] \dots [\![Y_1 \dots Y_m]\!] \| \beta\$. \tag{5.16}$$

Then,

$$\begin{aligned} &|\pi| = 2|\tau(\pi)| + |\alpha| - |\beta| \textit{ and} \\ &Y_1 \dots Y_m \beta \overset{\tau(\pi)^R}{\Rightarrow} X_1 \dots X_n \alpha \textit{ in } \mathcal{G}. \end{aligned} \tag{5.17}$$

Proof. The proof is by induction on the length of action string π.

If $\pi = \varepsilon$, then Equation 5.17 clearly holds.

Let us prove the induction step. We assume that $\pi = r\pi'$ where r is a single reduce or a single shift action, and where the lemma holds for π'.

If r is a reduce action, then for some number p with $1 \leq p \leq n$ and rule $A \rightarrow X_p \dots X_n$,

$$\begin{aligned} & && [\![\varepsilon]\!][\![X_1]\!] \dots [\![X_1 \dots X_n]\!] \| \alpha\$ \\ &\models_{A \rightarrow X_p \dots X_n} && [\![\varepsilon]\!][\![X_1]\!] \dots [\![X_1 \dots X_{p-1}]\!] \| A\alpha\$ \\ &\models_{\pi'} && [\![\varepsilon]\!][\![Y_1]\!] \dots [\![Y_1 \dots Y_m]\!] \| \beta\$ \text{ in } M. \end{aligned} \tag{5.18}$$

By induction hypothesis,

$$\begin{aligned} &|\pi'| = |\tau(\pi')| + |A\alpha| - |\beta| \text{ and} \\ &Y_1 \dots Y_m \beta \overset{\tau(\pi')^R}{\Rightarrow} X_1 \dots X_{p-1} A\alpha \text{ in } \mathcal{G}. \end{aligned} \tag{5.19}$$

Thus

$$\begin{aligned} &|\pi| = |r| + |\pi'| = |r| + |A| + 2|\tau(\pi')| + |\alpha| - |\beta| \text{ and} \\ &Y_1 \dots Y_m \beta \overset{\tau(\pi')^R}{\Rightarrow} X_1 \dots X_{p-1} A\alpha \overset{A \rightarrow X_p \dots X_n}{\Rightarrow} X_1 \dots X_n \alpha \text{ in } \mathcal{G}, \end{aligned} \tag{5.20}$$

i.e. Equation 5.17 holds.

If r is a shift action, then we rewrite α as $X\omega$ and

$$\begin{aligned} & && [\![\varepsilon]\!][\![X_1]\!] \dots [\![X_1 \dots X_n]\!] \| X\omega\$ \\ &\models_{\text{shift}} && [\![\varepsilon]\!][\![X_1]\!] \dots [\![X_1 \dots X_n]\!][\![X_1 \dots X_n X]\!] \| \omega\$ \\ &\models_{\pi'} && [\![\varepsilon]\!][\![Y_1]\!] \dots [\![Y_1 \dots Y_m]\!] \| \beta\$ \text{ in } M. \end{aligned} \tag{5.21}$$

By induction hypothesis,

$$\begin{aligned} &|\pi'| = |\tau(\pi')| + |\omega| - |\beta| \text{ and} \\ &Y_1 \dots Y_m \beta \overset{\tau(\pi')^R}{\Rightarrow} X_1 \dots X_n X\omega \text{ in } \mathcal{G}. \end{aligned} \tag{5.22}$$

Thus

$$\begin{aligned} &|\pi| = |r| + |\pi'| = 2|\tau(\pi')| + |X| + |\omega| - |\beta| \text{ and} \\ &Y_1 \dots Y_m \beta \overset{\tau(\pi')^R}{\Rightarrow} X_1 \dots X_n X\omega = X_1 \dots X_n \alpha \text{ in } \mathcal{G}, \end{aligned} \tag{5.23}$$

where $|\tau(r)| = 0$ and $|X| + |\omega| = |\alpha|$, thus Equation 5.17 holds. □

Since all the conflict lookahead symbols are removed from the noncanonical lookahead sets NLA, the only possibility for the noncanonical automaton to be nondeterministic would be to have a totally reduced symbol causing a conflict. A context-free grammar $\mathcal{G}$ is NLALR(1) if its NLALR(1) automaton is deterministic, and thus if no totally reduced symbol can cause a conflict.

5.2.3 Efficient Construction

The LALR(1) lookahead sets that are defined in Equation A.5 on page 169 can be efficiently expressed using the following definitions by DeRemer and Pennello (1982), where lookback is a relation between reductions and nonterminal LR(0) transitions, includes and reads are relations between nonterminal LR(0) transitions, and DR—standing for *directly reads*—is a function from nonterminal LR(0) transitions to sets of lookahead symbols.

$$([\delta\alpha], A{\rightarrow}\alpha) \ \mathsf{lookback}\ ([\delta], A), \tag{5.24}$$

$$([\delta\beta], A) \ \mathsf{includes}\ ([\delta], B) \text{ iff } B{\rightarrow}\beta A\gamma \text{ and } \gamma \Rightarrow^* \varepsilon, \tag{5.25}$$

$$([\delta], A) \ \mathsf{reads}\ ([\delta A], C) \text{ iff } ([\delta A], C) \text{ and } C \Rightarrow^* \varepsilon, \tag{5.26}$$

$$\mathrm{DR}([\delta], A) = \{a \mid ([\delta A], a)\}. \tag{5.27}$$

Using the above definitions, we can rewrite Equation A.5 as

$$\mathrm{LA}(q, A{\rightarrow}\alpha) = \bigcup_{(q,A\rightarrow\alpha)\mathsf{lookback}\,\circ\,\mathsf{includes}^*\,\circ\,\mathsf{reads}^*(r,C)} \mathrm{DR}(r, C). \tag{5.28}$$

For instance, considering the reduction using $A{\rightarrow}a$ in q_5 as we did in Section 5.2.1.2 on page 90, since

$$(q_5, A{\rightarrow}a) \ \mathsf{lookback}\ (q_3, A) \ \mathsf{includes}\ (q_3, C) \ \mathsf{includes}\ (q_0, S) \tag{5.29}$$

and since

$$\mathrm{DR}(q_0, S) = \{\$\}, \tag{5.30}$$

we find again that the end of file \$ belongs to the LALR(1) lookahead set for this reduction.

This computation for LALR(1) lookahead sets is highly efficient. It can be entirely performed on the LR(0) automaton, and the union can be interleaved with a fast transitive closure algorithm (Tarjan, 1972) on the includes and reads relations.

5.2.3.1 Computing Lookaheads

Since we have a very efficient and widely adopted computation for the canonical LALR(1) lookahead sets, why not try to use it for the noncanonical ones?

Theorem 5.15.

$$\mathit{RLA}(q, A{\rightarrow}\alpha) = \{X \mid X \Rightarrow^* ax, \psi \Rightarrow^* \varepsilon, [C{\rightarrow}\rho B{\bullet}\psi X\sigma] \in \textsf{\textit{Kernel}}(\delta\rho B) \text{ and } (q, A{\rightarrow}\alpha)\ \textsf{\textit{lookback}} \circ \textsf{\textit{includes}}^*\ ([\delta\rho], B)\}.$$

Theorem 5.15 is an immediate application of Lemma 5.17, and provides a computation for the totally reduced lookaheads. We first quote a technical lemma proven by Sippu and Soisalon-Soininen (1990).

Lemma 5.16. *[Sippu and Soisalon-Soininen (1990)] Let $\mathcal{G} = \langle V, T, P, S\rangle$ be a grammar. Further, let A be a nonterminal, X and Y symbols in V, γ and ψ strings in V^*, y a string in T^*, and π a rule string in P^* such that*

$$A \underset{\text{rm}}{\overset{\pi}{\Longrightarrow}} \gamma X\psi Y y \text{ and } \psi \Rightarrow^* \varepsilon. \tag{5.31}$$

Then there are symbols X' and Y' in V, a rule $r' = B{\rightarrow}\alpha X'\psi' Y'\beta$ in P, and strings γ', α', β' in V^* and y' in T^* such that

$$\begin{aligned} &A \underset{\text{rm}}{\overset{\pi'}{\Longrightarrow}} \gamma' B y' \underset{\text{rm}}{\overset{r'}{\Longrightarrow}} \gamma'\alpha X'\psi' Y'\beta y',\ \gamma'\alpha\alpha' = \gamma,\\ &X' \underset{\text{rm}}{\Longrightarrow^*} \alpha' X,\ \psi' \Rightarrow^* \varepsilon,\ \text{and } Y' \underset{\text{lm}}{\Longrightarrow^*} Y\beta', \end{aligned} \tag{5.32}$$

where $\pi' r'$ is a prefix of π.

Lemma 5.17. *The non null symbol X belongs to $\mathit{RLA}(q, A{\rightarrow}\alpha)$ if and only if there is a rule $C{\rightarrow}\rho B\psi X\sigma$ with $\psi \Rightarrow^* \varepsilon$ and state $[\delta\rho]$ such that*

$$(q, A{\rightarrow}\alpha)\ \textsf{\textit{lookback}} \circ \textsf{\textit{includes}}^*\ ([\delta\rho], B) \text{ and } C{\rightarrow}\rho{\bullet}B\psi X\sigma \in \textsf{\textit{Valid}}(\delta\rho).$$

Proof. This lemma is very similar to Lemma 7.15 from Sippu and Soisalon-Soininen (1990). We first assume that X belongs to $\text{RLA}(q, A{\rightarrow}\alpha)$. Then

$$S' \underset{\text{lm}}{\Longrightarrow^*} zA\gamma X\omega, \gamma \Rightarrow^* \varepsilon, X \Rightarrow^* ax \text{ and } q = [\hat{z}\alpha]; \tag{5.33}$$

we transform this derivation into a rightmost derivation:

$$S' \underset{\text{rm}}{\Longrightarrow^*} \hat{z}A\gamma X y \underset{\text{rm}}{\Longrightarrow^*} \hat{z}Aaxy \text{ where } \omega \Rightarrow^* y. \tag{5.34}$$

Then, we can apply the previous lemma from Sippu and Soisalon-Soininen (1990), hence there exists rule $C{\rightarrow}\rho B\psi X'\sigma$ and strings δ, β, φ and u such that

$$\begin{aligned} &S' \underset{\text{rm}}{\Longrightarrow^*} \delta C u \underset{\text{rm}}{\Longrightarrow} \delta\rho B\psi X'\sigma u, \delta\rho\beta = \hat{z},\\ &B \underset{\text{rm}}{\Longrightarrow^*} \beta A, \psi \Rightarrow^* \varepsilon \text{ and } X' \underset{\text{lm}}{\Longrightarrow^*} X\varphi. \end{aligned} \tag{5.35}$$

Remember that X appeared in the leftmost derivation $S' \underset{\text{lm}}{\Rightarrow}^* zA\gamma X\omega$, where it could not be derived by X' in a leftmost order, thus

$$X' = X \text{ and } \varphi = \varepsilon. \tag{5.36}$$

Thus $C \rightarrow \rho \bullet B\psi X\sigma \in \mathsf{Valid}(\delta\rho)$ and

$$([\hat{z}], A) \text{ includes}^* ([\delta\rho], B), \text{ where} \tag{5.37}$$

$$([\hat{z}\alpha], A \rightarrow \alpha) \text{ lookback } ([\hat{z}], A), \tag{5.38}$$

from which we deduce the result.

Conversely, since $C \rightarrow \rho \bullet B\psi X\sigma \in \mathsf{Valid}(\delta\rho)$,

$$S' \underset{\text{rm}}{\Rightarrow}^* \delta\rho B\psi X\sigma u, \tag{5.39}$$

and since $([\hat{z}\alpha], A \rightarrow \alpha)$ lookback $\circ$ includes* $([\delta\rho], B)$,

$$B \Rightarrow^* \beta A\gamma \text{ with } \gamma \Rightarrow^* \varepsilon. \tag{5.40}$$

Therefore, converting to leftmost derivations,

$$S' \underset{\text{lm}}{\Rightarrow}^* yB\psi X\sigma\omega \text{ where } \delta\rho \Rightarrow^* y \text{ and } \omega \Rightarrow^* u, \tag{5.41}$$

$$\underset{\text{lm}}{\Rightarrow}^* zA\gamma\psi X\sigma\omega \text{ where } \gamma\psi \Rightarrow^* \varepsilon \text{ and } \delta\rho\beta \Rightarrow^* z. \tag{5.42}$$

The non null symbol X complies with the conditions of RLA$([\hat{z}\alpha], A \rightarrow \alpha)$. □

This theorem is consistent with the description of Section 5.2.1.2 on page 90, where we explained that C was a totally reduced lookahead for reduction $B \rightarrow a$ in q_1: indeed, item $S \rightarrow B \bullet C$ is in the kernel of state q_3 accessed by (q_0, B), and $(q_1, B \rightarrow a)$ lookback (q_0, B).

Theorem 5.18. *Let us extend the* directly reads *function of Equation 5.27 to*

$$DR([\delta], A) = \{X \mid ([\delta A], X) \textit{ and } X \Rightarrow^* ax\}; \textit{ then}$$
$$DLA(q, A \rightarrow \alpha) = \bigcup_{(q, A \rightarrow \alpha) \textit{lookback} \circ \textit{includes}^* \circ \textit{reads}^* (r, C)} DR(r, C).$$

Proof. Let the non null symbol $X \Rightarrow^* ax$ be in DLA$(q, A \rightarrow \alpha)$ with $q = [\hat{\delta}\alpha]$. Let us show it is in the directly read set of some related transition. We have

$$S' \Rightarrow^* \delta AX\omega, \tag{5.43}$$

thus

$$S' \underset{\mathrm{rm}}{\Longrightarrow}^* \hat{\delta}AXy \text{ with } \omega \Rightarrow^* y. \tag{5.44}$$

Then, there exist a rule $C \rightarrow \beta BX\sigma$ and strings γ and y' such that

$$S' \underset{\mathrm{rm}}{\Longrightarrow}^* \gamma Cy' \underset{\mathrm{rm}}{\Longrightarrow} \gamma\beta BX\sigma y' \underset{\mathrm{rm}}{\Longrightarrow}^* \gamma\beta BXy \underset{\mathrm{rm}}{\Longrightarrow}^* \gamma\beta Baxy \underset{\mathrm{rm}}{\Longrightarrow}^* \hat{\delta}Aaxy. \tag{5.45}$$

Therefore, $X \in \mathrm{DR}([\gamma\beta], B)$ where $\gamma\beta B$ is a valid cover for $\hat{\delta}A$. By Theorem 5.20, we get the desired result.

Conversely, if X is in $\mathrm{DR}([\gamma], C)$ where γC is a valid cover for $\hat{\delta}A$, then

$$S' \underset{\mathrm{rm}}{\Longrightarrow}^* \gamma CXz \Rightarrow^* \hat{\delta}AXz \Rightarrow^* \delta AXz, \tag{5.46}$$

and thus X is in $\mathrm{DLA}([\hat{\delta}\alpha], A \rightarrow \alpha)$. □

We could have proven this theorem using Lemma 5.17 and the following lemma inspired by Lemma 7.13 of Sippu and Soisalon-Soininen (1990):

Lemma 5.19. *Let*

$$\mathit{DFirst}(\alpha) = \{X \mid \alpha \Rightarrow^* X\omega \Rightarrow^* ax\omega\},$$

then X is in $DR(r, C)$ with $([\gamma], A)$ reads^* (r, C) if and only if $\mathsf{Valid}(\gamma)$ contains an item $B \rightarrow \beta \bullet A\alpha$ and X is in $\mathit{DFirst}(\alpha)$.

The noncanonical lookaheads could then be derived from the totally reduced lookaheads, making the computation of the reads^* relation unnecessary. But we need to compute reads^* in order to get the valid covers for the noncanonical transitions. We might as well use this information for the computation of the noncanonical lookaheads, and avoid the computation of the DFirst sets.

We are still consistent with the description of Section 5.2.1.2 on page 90 since, using this new definition of the DR function, $\mathrm{DR}(q_0, B)$ is $\{a, C, A\}$.

5.2.3.2 Finding the Valid Covers

To find the valid covers that approximate a sentential form prefix using the LR(0) automaton and to find the LALR lookahead sets wind up being very similar operations. As presented in Section 5.2.1.2 on page 90, both can be viewed as simulating the LR(0) parser behavior. The following theorem formalizes this resemblance.

Theorem 5.20. *Let α be a phrase such that $S' \Rightarrow^* \delta AX\omega \Rightarrow \delta\alpha X\omega$, then*[1]

$$\mathit{Cover}([\hat{\delta}]AX) = \{\gamma CX \mid ([\hat{\delta}\alpha], A \rightarrow \alpha)\ \mathsf{lookback} \circ \mathsf{includes}^* \circ \mathsf{reads}^*\ ([\gamma], C)\}.$$

[1]By $[\delta]\alpha$, we denote the set of strings $\{\omega\alpha \mid \omega \in [\delta]\}$.

The proof is immediate using the following lemma.

Lemma 5.21. *Let δA be a valid prefix; then*

$$\mathit{Cover}([\delta]A) = \{\gamma C \mid ([\delta], A)\ \textsf{\textit{includes}}^* \circ \textsf{\textit{reads}}^*\ ([\gamma], C)\}.$$

Proof. Let $([\delta], A)$ $\textsf{includes}^* \circ \textsf{reads}^*$ $([\gamma], C)$, and let us show that γC is a valid cover for δA.

1. if $([\varphi\beta], A)$ $\textsf{includes}$ $([\varphi], B)$ then, since $\varphi B \Rightarrow \varphi\beta A\gamma \Rightarrow \varphi\beta A$, φB is a valid cover for $\varphi\beta A$, and

2. if $([\varphi], B)$ $\textsf{reads}$ $([\varphi B], C)$, then φBC is a valid cover for φB since $\varphi BC \Rightarrow \varphi B$.

Conversely, let δA be a valid prefix and γC a valid cover for δA, and let us show that $([\delta], A)$ $\textsf{includes}^* \circ \textsf{reads}^*$ $([\gamma], C)$. Since δA is a valid prefix and γC is a valid cover for δA,

$$S' \underset{\text{rm}}{\Longrightarrow}^* \gamma Cz \underset{\text{rm}}{\Longrightarrow}^* \delta Az. \tag{5.47}$$

Suppose $C \Rightarrow^* \varepsilon$. Then $\gamma C \underset{\text{rm}}{\Longrightarrow} \gamma \underset{\text{rm}}{\Longrightarrow}^* \delta A$, thus γ ends with a nonterminal and can be rewritten as $\gamma' C'$, and $([\gamma'], C')$ $\textsf{reads}$ $([\gamma], C)$ in this case. This can be iterated over any nullable suffix of γC.

What remains to be proven is that if $S' \underset{\text{rm}}{\Longrightarrow}^* \gamma Cz \underset{\text{rm}}{\Longrightarrow}^* \delta Az$ with $C \not\Rightarrow^* \varepsilon$, then $([\delta], A)$ $\textsf{includes}^*$ $([\gamma], C)$. But in this case $C \underset{\text{rm}}{\Longrightarrow}^* \beta A$ with $\delta = \gamma\beta$, and by Lemma 7.9 of Sippu and Soisalon-Soininen (1990), $([\delta], A)$ $\textsf{includes}^*$ $([\gamma], C)$. □

The valid covers of a reduction can thus be found using relations (5.24), (5.25) and (5.26). This allows us to reuse our lookahead sets computations for the automaton construction itself, as illustrated by the following corollary.

Corollary 5.22. Noncanonical state $[\![\delta]\!]$ *is the set of LR(0) states defined by*

$$\begin{aligned}[\![\varepsilon]\!] &= \{[\varepsilon]\} \textit{ and}\\ [\![\delta X]\!] &= \{[\gamma CX] \mid X \in \mathit{CLA}([\![\delta]\!], A{\rightarrow}\alpha), q \in [\![\delta]\!] \textit{ and}\\ &\qquad (q, A{\rightarrow}\alpha)\ \textsf{\textit{lookback}} \circ \textsf{\textit{includes}}^* \circ \textsf{\textit{reads}}^*\ ([\gamma], C)\}\\ &\quad \cup \{[\varphi X] \mid [\varphi] \in [\![\delta]\!]\}.\end{aligned}$$

5.2.3.3 Practical Construction Steps

We present here a more informal construction, with the main steps leading to the construction of a NLALR(1) parser, given the LR(0) automaton.

1. Associate a noncanonical state $s = \{q\}$ with each LR(0) state q.
2. Iterate while there exists an inadequate[2] state s:
   (a) if it has not been done before, compute the RLA and DLA lookahead sets for the reductions involved in the conflict; save their values for the reduction and LR(0) state involved;
   (b) compute the CLA and NLA lookahead sets for s;
   (c) set the lookaheads to NLA for the reduction actions in s;
   (d)
   - if the NLA lookahead sets leave the state inadequate, meaning there is a conflict on a totally reduced lookahead, then report the conflict, and use a conflict resolution policy or terminate with an error;
   - if CLA is not empty, create transitions on its symbols and create new states if no fusion occurs. New states get new transition and reduction sets computed from the LR(0) states they contain. If these new states result from shift/reduce conflicts, the transitions from s on the conflicting lookahead symbol now lead to the new states.

This process always terminates since there is a bounded number of LR(0) states and thus a bounded number of noncanonical states.

5.2.4 Alternative Definitions

We present here a few variants that also allow the construction of noncanonical LALR-based parsers.

5.2.4.1 Leftmost LALR(1) Alternative

If, instead of deciding a reduction as early as possible, we waited until we saw a totally reduced symbol in the lookahead window, then we would have defined a variant called leftmost LALR(1) parsing by analogy with leftmost SLR(1) parsing (Tai, 1979).

[2]We mean here inadequate in the LR(0) sense, thus no lookaheads need to be computed yet.

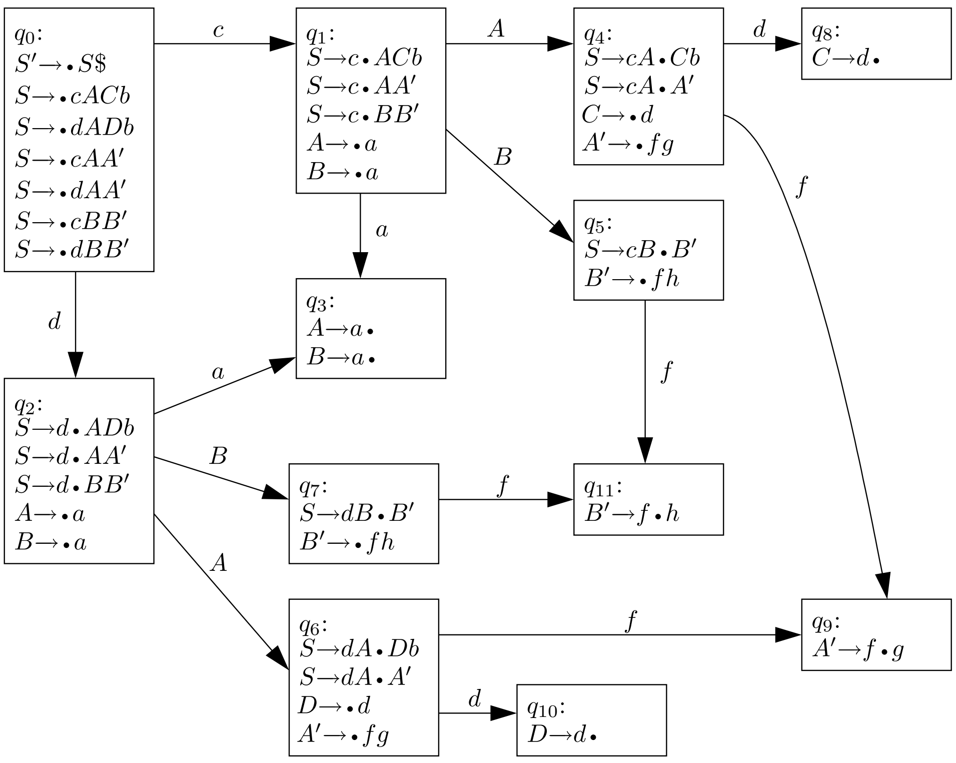

q_0: $S'\to\bullet S\$$ $S\to\bullet cACb$ $S\to\bullet dADb$ $S\to\bullet cAA'$ $S\to\bullet dAA'$ $S\to\bullet cBB'$ $S\to\bullet dBB'$
c
q_1: $S\to c\bullet ACb$ $S\to c\bullet AA'$ $S\to c\bullet BB'$ $A\to\bullet a$ $B\to\bullet a$
A
q_4: $S\to cA\bullet Cb$ $S\to cA\bullet A'$ $C\to\bullet d$ $A'\to\bullet fg$
d
q_8: $C\to d\bullet$
B
q_5: $S\to cB\bullet B'$ $B'\to\bullet fh$
f
a
q_3: $A\to a\bullet$ $B\to a\bullet$
d
a
q_2: $S\to d\bullet ADb$ $S\to d\bullet AA'$ $S\to d\bullet BB'$ $A\to\bullet a$ $B\to\bullet a$
B
q_7: $S\to dB\bullet B'$ $B'\to\bullet fh$
f
f
q_{11}: $B'\to f\bullet h$
A
q_6: $S\to dA\bullet Db$ $S\to dA\bullet A'$ $D\to\bullet d$ $A'\to\bullet fg$
f
q_9: $A'\to f\bullet g$
d
q_{10}: $D\to d\bullet$

Figure 5.4: Partial LR(0) automaton for $\mathcal{G}_{13}$.

In order to generate a Leftmost LALR(1) (LLALR(1)) parser, we have to change the definition with

$$\begin{aligned}\mathrm{CLA}(s, A\to\alpha) = {} & \{X \in \mathrm{DLA}(q, A\to\alpha) \mid q \in s \\ & \text{and } X \notin \mathrm{RLA}(q, A\to\alpha)\} \text{ and}\end{aligned} \tag{5.48}$$

$$\mathrm{NLA}(s, A\to\alpha) = \bigcup_{q\in s} \mathrm{RLA}(q, A\to\alpha). \tag{5.49}$$

LSLR(1) parsers are strictly weaker than NSLR(1) parsers (Tai, 1979). The same is true for LLALR(1) parsers compared to NLALR(1) parsers. The delayed resolution can cause a conflict. Grammar $\mathcal{G}_{13}$ with rules

$$\begin{gathered} S\to cACb \,|\, dADb \,|\, cAA' \,|\, dAA' \,|\, cBB' \,|\, dBB', \\ A\to a,\ B\to a,\ C\to d,\ D\to d,\ A'\to fg,\ B'\to fh \end{gathered} \tag{$\mathcal{G}_{13}$}$$

illustrates this case. Figure 5.4 presents a portion of the LR(0) automaton for $\mathcal{G}_{13}$.

The noncanonical lookaheads for the reductions in state q_3 are

$$\begin{aligned}
\mathrm{RLA}(q_3, A{\rightarrow}a) &= \{C, D, A'\},\\
\mathrm{DLA}(q_3, A{\rightarrow}a) &= \{C, D, d, A', f\},\\
\mathrm{RLA}(q_3, B{\rightarrow}a) &= \{B'\} \text{ and}\\
\mathrm{DLA}(q_3, B{\rightarrow}a) &= \{B', f\}.
\end{aligned}$$

In NLALR(1) parsing, noncanonical state $\{q_3\}$ has a transition on f to noncanonical state $\{q_9, q_{11}\}$, where the conflict is then solved by shifting g or h and reducing to A' or B'.

In LLALR(1) parsing, we compute transitions for all the lookahead symbols that are not completely reduced. With $\mathcal{G}_{13}$, this is the case of d, which appears in $\mathrm{DLA}(q_3, A{\rightarrow}a)$ but not in $\mathrm{RLA}(q_3, A{\rightarrow}a)$. Then, there is a transition on d from $\{q_3\}$ to $\{q_8, q_{10}\}$. This new state has a reduce/reduce conflict where both $\mathrm{RLA}(q_8, C{\rightarrow}d)$ and $\mathrm{RLA}(q_{10}, D{\rightarrow}d)$ contain symbol b. Grammar $\mathcal{G}_{13}$ is not LLALR(1).

5.2.4.2 Item-based Alternative

In the item-based variant, noncanonical states are collections of valid LR(0) items instead of sets of LR(0) states. This allows to merge states with identical Kernel item sets as defined in (A.1–A.2), and to produce more compact automata. There is however an impact on the power of the parsing method. Consider grammar with rules

$$S{\rightarrow}AC\,|\,BDa\,|\,cCa\,|\,cD,\;\; A{\rightarrow}a,\;\; B{\rightarrow}a,\;\; C{\rightarrow}b,\;\; D{\rightarrow}b. \qquad (\mathcal{G}_{14})$$

Figure 5.5 on the following page shows the LALR(1) automaton for $\mathcal{G}_{14}$.

Using the state-based NLALR(1) definition, we just add a transition on b from noncanonical state $\{q_1\}$ to noncanonical state $\{q_9, q_{11}\}$ where the reductions of b to C and D are associated with lookahead symbols \$ and a respectively.

Using the item-based definition, the state reached by a transition on b from state $\{A{\rightarrow}a{\bullet}, B{\rightarrow}a{\bullet}\} = \mathsf{Valid}(a)$ is state $\{C{\rightarrow}b{\bullet}, D{\rightarrow}b{\bullet}\} = \mathsf{Valid}(ab)$ where both symbols a and \$ are possible lookaheads for both reductions using $C{\rightarrow}b$ and $D{\rightarrow}b$. These symbols are totally reduced, and the generated parser for $\mathcal{G}_{14}$ is nondeterministic.

A solution for combining the strength of the state-based approach with the smaller automata of the item-based variant could be to consider the lookahead sets associated with each reduction item. We would then merge states with the same noncanonical LALR(1) item sets instead of the same LR(0) item sets. Since this would be comparable to a state merge once the construction is over, compression techniques for parsing tables seem a better choice.

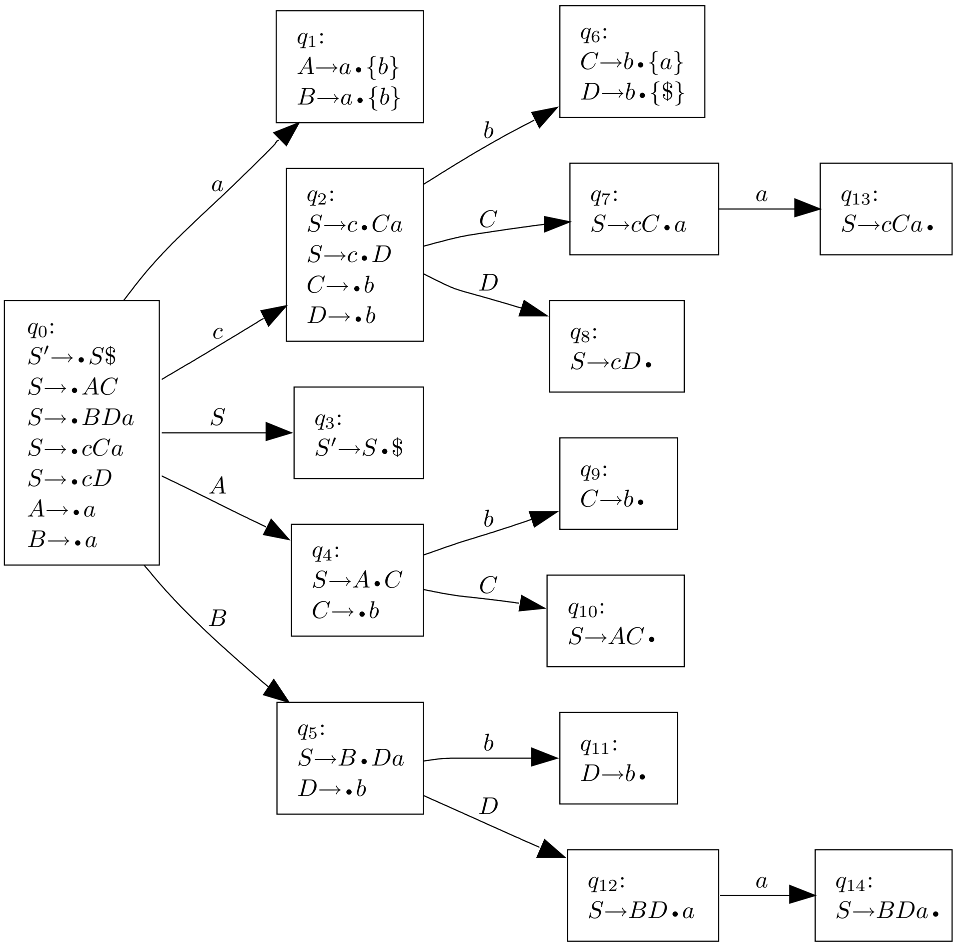

q_1: $A{\rightarrow}a\bullet\{b\}$ $B{\rightarrow}a\bullet\{b\}$
q_6: $C{\rightarrow}b\bullet\{a\}$ $D{\rightarrow}b\bullet\{\$\}$
q_2: $S{\rightarrow}c\bullet Ca$ $S{\rightarrow}c\bullet D$ $C{\rightarrow}\bullet b$ $D{\rightarrow}\bullet b$
q_7: $S{\rightarrow}cC\bullet a$
q_{13}: $S{\rightarrow}cCa\bullet$
q_8: $S{\rightarrow}cD\bullet$
q_0: $S'{\rightarrow}\bullet S\$$ $S{\rightarrow}\bullet AC$ $S{\rightarrow}\bullet BDa$ $S{\rightarrow}\bullet cCa$ $S{\rightarrow}\bullet cD$ $A{\rightarrow}\bullet a$ $B{\rightarrow}\bullet a$
q_3: $S'{\rightarrow}S\bullet\$$
q_9: $C{\rightarrow}b\bullet$
q_4: $S{\rightarrow}A\bullet C$ $C{\rightarrow}\bullet b$
q_{10}: $S{\rightarrow}AC\bullet$
q_5: $S{\rightarrow}B\bullet Da$ $D{\rightarrow}\bullet b$
q_{11}: $D{\rightarrow}b\bullet$
q_{12}: $S{\rightarrow}BD\bullet a$
q_{14}: $S{\rightarrow}BDa\bullet$
a
c
S
A
B
b
C
D
a
b
C
b
D
a

Figure 5.5: LALR(1) automaton for $\mathcal{G}_{14}$.

5.2.4.3 Transition-based Alternative

The two previous modifications we presented were weaker than the regular definition. The transition-based definition is stronger, but the generated noncanonical parsers are not isomorphic to LR(0) parsers on the conflict-free portions of the parser.

In the transition-based alternative, noncanonical states are sets of LR(0) transitions. All computations would then be based on LR(0) transitions instead of LR(0) states. We express this alternative in terms of the relations of Section 5.2.3 on page 98 because they involve LR(0) transitions.

First, we would augment our grammars with rule $S'{\rightarrow}\$S\$$ in order to have an incoming transition to the initial LR(0) state [\$].

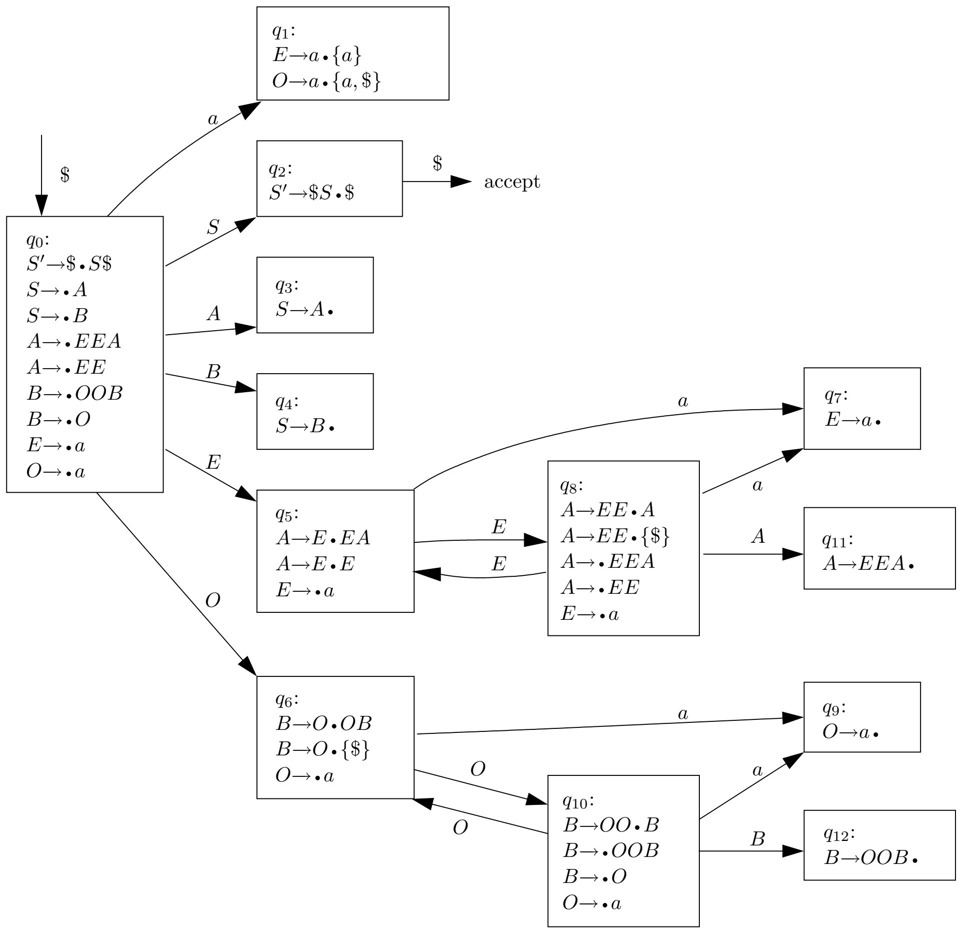

q_1: $E{\to}a{\bullet}\{a\}$ $O{\to}a{\bullet}\{a,\$\}$
q_2: $S'{\to}\$S{\bullet}\$$
accept
q_0: $S'{\to}\${\bullet}S\$$ $S{\to}{\bullet}A$ $S{\to}{\bullet}B$ $A{\to}{\bullet}EEA$ $A{\to}{\bullet}EE$ $B{\to}{\bullet}OOB$ $B{\to}{\bullet}O$ $E{\to}{\bullet}a$ $O{\to}{\bullet}a$
q_3: $S{\to}A{\bullet}$
q_4: $S{\to}B{\bullet}$
q_5: $A{\to}E{\bullet}EA$ $A{\to}E{\bullet}E$ $E{\to}{\bullet}a$
q_6: $B{\to}O{\bullet}OB$ $B{\to}O{\bullet}\{\$\}$ $O{\to}{\bullet}a$
q_7: $E{\to}a{\bullet}$
q_8: $A{\to}EE{\bullet}A$ $A{\to}EE{\bullet}\{\$\}$ $A{\to}{\bullet}EEA$ $A{\to}{\bullet}EE$ $E{\to}{\bullet}a$
q_9: $O{\to}a{\bullet}$
q_{10}: $B{\to}OO{\bullet}B$ $B{\to}{\bullet}OOB$ $B{\to}{\bullet}O$ $O{\to}{\bullet}a$
q_{11}: $A{\to}EEA{\bullet}$
q_{12}: $B{\to}OOB{\bullet}$

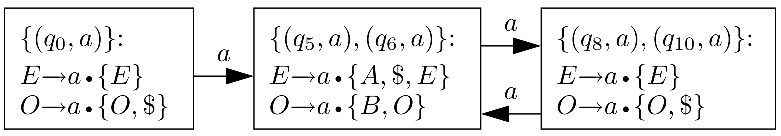
Figure 5.6: The LALR(1) automaton for $\mathcal{G}_{15}$.

Then, we would redefine the lookback relation. For a reduction using $A{\to}\alpha$, instead of looking back from q to any transition $([\delta], A)$ such that there is a path accessing q on $\delta\alpha$, we provide a transition (p, Y) accessing q and want to find transitions $([\delta], A)$ with a path starting in $[\delta]$ and accessing q on $\delta\alpha$ and ending by transition (p, Y).

$$((p, Y), A{\to}\alpha) \text{ lookback } (q, A) \text{ iff } p = [\varphi], q = [\delta] \text{ and } \delta\alpha = \varphi Y, \tag{5.50}$$

where $([\varphi Y], A{\to}\alpha)$ is a reduction using $A{\to}\alpha$ in state $[\varphi Y]$. All noncanonical lookaheads computations would then be modified accordingly.

At last, we would redefine noncanonical states as sets of LR(0) transitions



$\{(q_0,a)\}$:
$E \rightarrow a \bullet \{E\}$
$O \rightarrow a \bullet \{O,\$\}$
a
$\{(q_5,a),(q_6,a)\}$:
$E \rightarrow a \bullet \{A,\$,E\}$
$O \rightarrow a \bullet \{B,O\}$
a
a
$\{(q_8,a),(q_{10},a)\}$:
$E \rightarrow a \bullet \{E\}$
$O \rightarrow a \bullet \{O,\$\}$

Figure 5.7: State q_1 of $\mathcal{G}_{15}$ transformed and extended for noncanonical parsing.

defined by

$$\begin{aligned}
[\![\$]\!] &= \{([\varepsilon],\$)\} \text{ and}\\
[\![\delta X]\!] &= \{([\gamma C],X) \mid X \in \mathrm{CLA}([\![\delta]\!],A{\rightarrow}\alpha),(p,Y)\in[\![\delta]\!] \text{ and}\\
&\quad ((p,Y),A{\rightarrow}\alpha)\ \mathsf{lookback}\circ\mathsf{includes}^*\circ\mathsf{reads}^*\ ([\gamma],C)\}\\
&\quad \cup \{([\varphi Y],X) \mid ([\varphi],Y)\in[\![\delta]\!]\}.
\end{aligned} \tag{5.51}$$

Let us illustrate these definitions on an example. Consider Grammar $\mathcal{G}_{15}$ with rules

$$S{\rightarrow}A\,|\,B,\ A{\rightarrow}EEA\,|\,EE,\ B{\rightarrow}OOB\,|\,O,\ E{\rightarrow}a,\ O{\rightarrow}a; \tag{$\mathcal{G}_{15}$}$$

its LALR(1) automaton is presented in Figure 5.6 on the preceding page. Using the state-based NLALR(1) construction, we would create a transition on symbol a from state $\{q_1\}$ to state $\{q_7,q_9\}$, where totally reduced symbol \$ would appear in the lookahead window of both reductions using $E{\rightarrow}a$ and $O{\rightarrow}a$.

Using the transition-based construction, noncanonical state $\{(q_0,a)\}$ corresponding to the LR(0) state q_1 would have a transition on symbol a to $\{(q_5,a),(q_6,a)\}$ where \$ appears only in the lookahead window of reduction using $E{\rightarrow}a$. Noncanonical state $\{(q_5,a),(q_6,a)\}$ would also have a transition on a to $\{(q_8,a),(q_{10},a)\}$ where \$ appears only in the lookahead window of reduction using $O{\rightarrow}a$. At last, state $\{(q_8,a),(q_{10},a)\}$ would have a transition on a back to $\{(q_5,a),(q_6,a)\}$. Figure 5.7 recapitulates these computations.

But we also notice that LR(0) state q_5 would be split into noncanonical states $[\![\$E]\!] = \{(q_0,E)\}$ and $[\![\$EE(EE)^*E]\!] = \{(q_8,E)\}$. This construction is not really LALR in spirit. In fact, it could be generalized by having n-tuples of LR(0) transitions as noncanonical states, improving the precision of the lookahead computations at the expense of an increased size for the resulting automaton. One could also view these n-tuples of transitions as $(n{+}1)$-tuples of states, in the line of the LR-Regular lookahead computations of Boullier (1984) and Bermudez (1991).

5.2.5 NLALR(1) Grammars and Languages

We compare here the power of NLALR(1) parsing with the power of various other parsing methods.

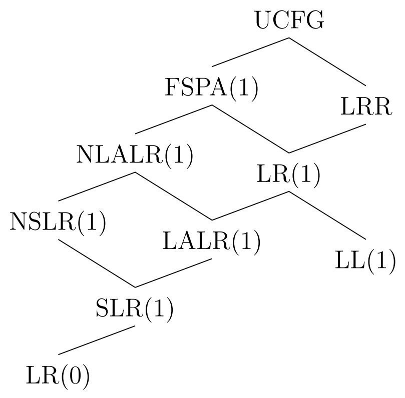


Figure 5.8: Lattice of inclusions between grammar classes.

A *grammar class* is a set of grammars, usually the set for which a deterministic parser can be generated by some construction. We denote by $\mathcal{C}_{\text{NLALR(1)}}$ the class of grammars accepted by a NLALR(1) parser; we compare this class with other grammar classes. The lattice of Figure 5.8 sums up this comparisons.

Unambiguous Grammars We told in Section 5.2.2.4 on page 95 that a grammar was NLALR(1) if and only if no totally reduced symbol could cause a conflict. As will be seen in the next comparison, NLALR(1) grammars are all FSPA(1) grammars, which are all unambiguous. The class of NLALR(1) grammars is thus included in the class of all unambiguous grammars. Considering then the unambiguous and non-NLALR(1) palindrome grammar with rules

$$S \rightarrow aSa \,|\, bSb \,|\, \varepsilon, \qquad (\mathcal{G}_{16})$$

we get the proper inclusion

$$\mathcal{C}_{\text{NLALR(1)}} \subset \mathcal{C}_{\text{UCFG}}. \qquad (5.52)$$

FSPA(1) Grammars Grammars that are parsable using a finite-state parsing automaton and k symbols of lookahead window have been introduced in (Szymanski and Williams, 1976). Informally presented, a FSPA(k) grammar can be parsed noncanonically using a regular approximation of the parsing stack contents and k symbols of lookahead. This parsing method is thus the most general noncanonical parsing method using a bounded lookahead and a finite representation of the parsing stack language. However, the problem of deciding whether a given grammar is FSPA(k), even for a given k, is undecidable in general.

Noncanonical LALR(1) parsing complies with this definition; moreover, grammar $\mathcal{G}_{15}$ is FSPA(1) but not NLALR(1), hence the following proper inclusion for grammars without any null nonterminal

$$\mathcal{C}_{\text{NLALR}(1)} \subset \mathcal{C}_{\text{FSPA}(1)}. \tag{5.53}$$

LALR(1) Grammars Equality 5.14 and Definition 5.11 together imply that any LALR(1) grammar has a NLALR(1) automaton with singleton noncanonical states and empty CLA lookahead sets.

Grammar $\mathcal{G}_{12}$ is an example of a non-LALR(1) and NLALR(1) grammar; thus the proper inclusion

$$\mathcal{C}_{\text{LALR}(1)} \subset \mathcal{C}_{\text{NLALR}(1)}. \tag{5.54}$$

NSLR(1) Grammars Noncanonical SLR(1) parsers (Tai, 1979) are built in a very similar way to NLALR(1) grammars, though they are item-based. Their lookahead computation is however less precise as it uses the noncanonical Follow sets instead of the RLA and DLA lookahead sets. Here are the noncanonical Follow sets, corrected (Salomon and Cormack, 1989) to exclude null symbols:

$$\text{RFollow}(A) = \{X \mid S' \underset{\text{lm}}{\Longrightarrow}^* zA\gamma X\omega, \gamma \Rightarrow^* \varepsilon, X \Rightarrow^* ax\} \tag{5.55}$$

$$\text{DFollow}(A) = \{X \mid S' \Rightarrow^* \delta AX\omega, X \Rightarrow^* ax\}. \tag{5.56}$$

Given a LR(0) state q with a possible reduction using $A{\rightarrow}\alpha$, clearly

$$\text{RLA}(q, A{\rightarrow}\alpha) \subseteq \text{RFollow}(A), \tag{5.57}$$

and thus any NSLR(1) grammar is also NLALR(1). Grammar $\mathcal{G}_{12}$ given on page 88 is an example of a non-NSLR(1) and NLALR(1) grammar; hence the proper inclusion

$$\mathcal{C}_{\text{NSLR}(1)} \subset \mathcal{C}_{\text{NLALR}(1)}. \tag{5.58}$$

LL(1) and LR(1) Grammars On one hand, Grammar $\mathcal{G}_{12}$ was shown to be non-LR(k) and thus non-LL(k), but is NLALR(1). On the other hand, many classical non-LALR grammars are not NLALR(1) either, since these examples rely on a faulty approximation done in the construction of the underlying LR(0) machine. For instance, the grammar with rules

$$\begin{gathered} S{\rightarrow}aA\,|\,bB,\ A{\rightarrow}Cc\,|\,Dd,\ B{\rightarrow}Cd\,|\,Dc, \\ C{\rightarrow}FE,\ D{\rightarrow}FH,\ E{\rightarrow}\varepsilon,\ F{\rightarrow}\varepsilon,\ H{\rightarrow}\varepsilon \end{gathered} \tag{$\mathcal{G}_{17}$}$$

is SLL(1) and thus LL(1) and LR(1) but not LALR(k) for any k nor NLALR(1).

Grammar class $\mathcal{C}_{\text{NLALR}(1)}$ is therefore incomparable with both grammar classes $\mathcal{C}_{\text{LL}(1)}$ and $\mathcal{C}_{\text{LR}(1)}$.

LRR Grammars LR-Regular parsers (defined by Čulik and Cohen (1973), and previously mentioned in Section 3.2.2 on page 35) are able to use a regular cover for an unbounded lookahead exploration. Like with $\mathcal{C}_{\mathrm{FSPA}(k)}$, the membership in the grammar class is an undecidable problem. However, some practical approximations have been developed (Bermudez and Schimpf, 1990).

Grammar $\mathcal{G}_{17}$ being LR(1), it is *a fortiori* LRR. Grammar $\mathcal{G}_{18}$ with rules shamelessly copied from Szymanski and Williams (1976)

$$S\to AC\,|\,BD,\ A\to a,\ B\to a,\ C\to bC\,|\,bD,\ D\to bDc\,|\,bc \qquad (\mathcal{G}_{18})$$

is not LRR but is NLALR(1). Grammar class $\mathcal{C}_{\mathrm{NLALR}(1)}$ is therefore incomparable with grammar class $\mathcal{C}_{\mathrm{LRR}}$.

Deterministic Languages The NLALR(1) language class is the set of all languages for which there exists a NLALR(1) grammar.

All the deterministic languages can be recognized using a deterministic SLR(1) machine; since $\mathcal{C}_{\mathrm{SLR}(1)} \subset \mathcal{C}_{\mathrm{LALR}(1)}$, they can also be parsed by a deterministic NLALR(1) machine. This inclusion is proper: Tai (1979) presents several nondeterministic languages which are NSLR(1) and are thus NLALR(1). Using a transformation in the line of Bermudez and Logothetis (1989), we should be able to prove that any NLALR(1) language is NSLR(1) as well.

LRR Languages Grammar $\mathcal{G}_{19}$ with rules

$$\begin{array}{l} S\to ACD\,|\,A'C',\ A\to aAB\,|\,aB,\ B\to b,\ C\to F,\ D\to dD\,|\,d,\\ A'\to aA'B'\,|\,aB',\ B'\to bB'',\ B''\to b,\ C'\to F,\ F\to cFd\,|\,cd, \end{array} \qquad (\mathcal{G}_{19})$$

is NLALR(1) and even NSLR(1), but generates the non-LR-Regular language (Čulik and Cohen, 1973)

$$\{a^n b^n c^m d^{m+l} \mid n, m, l \geq 1\} \cup \{a^n b^{2n} c^m d^m \mid n, m \geq 1\}. \qquad (\mathcal{L}(\mathcal{G}_{19}))$$

Szymanski and Williams (1976) proved that any LR-Regular language is also a BCP language, and we suspect that the class of BCP languages coincides with the class of NLALR(1) languages.

5.2.6 Practical Considerations

5.2.6.1 Parser Size

Let us dedicate a few words to the size of the generated parsers. Since NLALR(1) states are sets of LR(0) states, we find an exponential function of the size of the LR(0) automaton as an upper bound on the size of

$$\begin{array}{rcl}
\langle \mathit{statement} \rangle & \rightarrow & \langle \mathit{labeled\_statement} \rangle \\
& \mid & \langle \mathit{expression\_statement} \rangle \, ; \\
\langle \mathit{labeled\_statement} \rangle & \rightarrow & \mathit{identifier} : \langle \mathit{statement} \rangle \\
& \mid & \textbf{case} \ \langle \mathit{constant\_expression} \rangle : \langle \mathit{statement} \rangle \\
\langle \mathit{expression\_statement} \rangle & \rightarrow & \langle \mathit{expression} \rangle \\
\langle \mathit{expression} \rangle & \Rightarrow^* & \langle \mathit{primary\_expression} \rangle \\
\langle \mathit{primary\_expression} \rangle & \rightarrow & \mathit{identifier} \\
\langle \mathit{conditional\_expression} \rangle & \rightarrow & \langle \mathit{logical\_or\_expression} \rangle \, ? \, \langle \mathit{expression} \rangle : \\
& & \langle \mathit{conditional\_expression} \rangle
\end{array}$$

Figure 5.9: The syntax of labeled statements in ANSI C.

the NLALR(1) automaton. This bound seems however pretty irrelevant in practice. The NLALR(1) parser generator needs to create a new state for each lookahead causing a conflict, which does not happen so often. All the grammars we studied created transitions to canonical states very quickly afterwards. Experimental results with NSLR(1) parsers show that the increase in size is negligible in practice (Tai, 1979).

Furthermore, noncanonical parsers can be exponentially more succinct than canonical ones for the same grammar. Consider for instance the grammar family with rules

$$S \rightarrow cSc \,|\, dSd \,|\, AA' \,|\, BB',\ A \rightarrow a,\ B \rightarrow a,\ A' \rightarrow c^{i-1}a,\ B' \rightarrow c^{i-1}b \qquad (\mathcal{G}_{20}^{i})$$

inspired by a grammar family designed for similar considerations by Bertsch and Nederhof (2007). A conflict between the reductions by $A \rightarrow a$ and $B \rightarrow a$ is avoided by a canonical LR(k) parser if and only if $k \geq i$. But such a parser contains a different state with an item $[S \rightarrow cSc\bullet, u]$ for each u in $\{c, d\}^k$, whereas the NLALR(1) parsers for $\mathcal{G}_{20}^{i}$ contain a single such state with item $[S \rightarrow cSc\bullet]$.

5.2.6.2 SLR Versus LALR

During our tests on ambiguity detection, we identified a number of cases where a noncanonical SLR lookahead computation would bode worse than a canonical LALR one. A case that recurs in the syntax of ANSI C (Kernighan and Ritchie, 1988, Appendix A.13) and of Java (Gosling et al., 1996, Chapter 19) is a conflict in the syntax of labeled statements.

Labeled C Statements We present the relevant portions of the ANSI C grammar in Figure 5.9. A shift/reduce conflict occurs when the LR(0) parser attempts to recognize a $\langle$*statement*$\rangle$ starting with an *identifier*. With a colon symbol as the lookahead symbol, a SLR(1) parser cannot solve the conflict. Indeed, a $\langle$*primary_expression*$\rangle$ can appear before a colon symbol ":" in only two contexts: in a "**case**" labeled statement, or in a ternary $\langle$*conditional_expression*$\rangle$.

Nevertheless, due to the presence of the "**case**" keyword or of the question mark "?", neither of these contexts can be mistaken with an *identifier* at the beginning of a $\langle$*statement*$\rangle$, and hence a LALR(1) parser knows it can shift safely.

Weakness of NLSR(1) Parsing Observe that the colon symbol is reduced and would thus appear in the RFollow($\langle$*primary_expression*$\rangle$) lookahead set of a NSLR(1) parser. This portion of the C syntax is NLALR(1) but not NSLR(1).

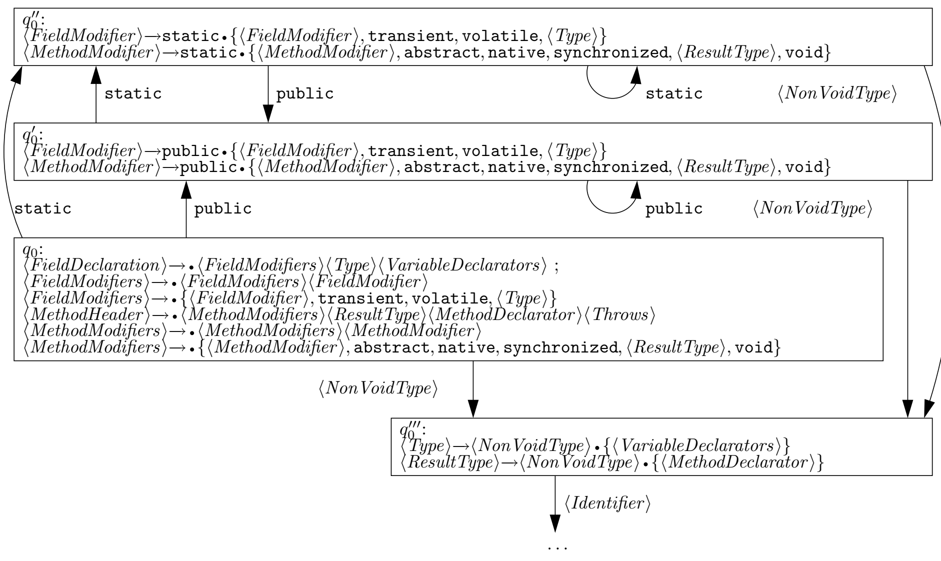
We could devise a transformation of the grammar in the line of Bermudez and Logothetis (1989), where the LR(0) automaton is effectively encoded in the grammar, and the RFollow set would then coincide with the RLA set. This transformation is meant as a simpler way to compute LALR lookahead sets, and not as an explicit grammar rewrite, which would clearly impair the grammar's readability.

5.2.6.3 Need for More Lookahead

Let us consider again the syntax of Java modifiers presented in Section 3.1.1 on page 26. This excerpt of the Java specification grammar is NLALR(2), but can be made NLALR(1) with the simple reassignment of the productions of $\langle$*Type*$\rangle$ to $\langle$*NonVoidType*$\rangle$, the addition of the unit production $\langle$*Type*$\rangle\to\langle$*NonVoidType*$\rangle$, and the change from $\langle$*Type*$\rangle$ to $\langle$*NonVoidType*$\rangle$ in the derivation of $\langle$*ResultType*$\rangle$:

$$\begin{gathered}\langle \mathit{Type}\rangle\to\langle \mathit{NonVoidType}\rangle\\ \langle \mathit{ResultType}\rangle\to\langle \mathit{NonVoidType}\rangle \mid \mathbf{void}\end{gathered}$$

We can generate a NLALR(1) parser for the modified set of grammar productions. Figure 5.10 on the next page presents a small portion of the NLALR(1) automaton. On the partial input "**public static int** length" presented in Section 3.1.1.2 on page 26, it will shift the "**public**" and "**static**" tokens, shift "**int**" and reduce it to a $\langle$*NonVoidType*$\rangle$ and shift it, reduce "length" and parts of what follows to either a $\langle$*VariableDeclarator*$\rangle$ or to a $\langle$*MethodDeclarator*$\rangle$. The parser will come back to this point later, but now it is able to solve all pending conflicts. The nonterminal $\langle$*NonVoidType*$\rangle$ can be reduced to $\langle$*Type*$\rangle$ or $\langle$*ResultType*$\rangle$, and "**static**" followed by **public** to a $\langle$*FieldModifier*$\rangle$ or a $\langle$*MethodModifier*$\rangle$ each. At last, ε and these two nonterminals are reduced to a single $\langle$*FieldModifiers*$\rangle$ or $\langle$*MethodModifiers*$\rangle$.



q_0'':
⟨FieldModifier⟩→static•{⟨FieldModifier⟩, transient, volatile, ⟨Type⟩}
⟨MethodModifier⟩→static•{⟨MethodModifier⟩, abstract, native, synchronized, ⟨ResultType⟩, void}
static
public
static
⟨NonVoidType⟩
q_0':
⟨FieldModifier⟩→public•{⟨FieldModifier⟩, transient, volatile, ⟨Type⟩}
⟨MethodModifier⟩→public•{⟨MethodModifier⟩, abstract, native, synchronized, ⟨ResultType⟩, void}
static
public
public
⟨NonVoidType⟩
q_0:
⟨FieldDeclaration⟩→•⟨FieldModifiers⟩⟨Type⟩⟨VariableDeclarators⟩ ;
⟨FieldModifiers⟩→•⟨FieldModifiers⟩⟨FieldModifier⟩
⟨FieldModifiers⟩→•{⟨FieldModifier⟩, transient, volatile, ⟨Type⟩}
⟨MethodHeader⟩→•⟨MethodModifiers⟩⟨ResultType⟩⟨MethodDeclarator⟩⟨Throws⟩
⟨MethodModifiers⟩→•⟨MethodModifiers⟩⟨MethodModifier⟩
⟨MethodModifiers⟩→•{⟨MethodModifier⟩, abstract, native, synchronized, ⟨ResultType⟩, void}
⟨NonVoidType⟩
q_0''':
⟨Type⟩→⟨NonVoidType⟩•{⟨VariableDeclarators⟩}
⟨ResultType⟩→⟨NonVoidType⟩•{⟨MethodDeclarator⟩}
⟨Identifier⟩
...

Figure 5.10: Partial NLALR(1) automaton for the Java fields and methods declaration syntax of Figure 3.1 on page 27.

This real-life example should demonstrate the suitability of NLALR(1) parsers for practical purposes. We notice in particular that the noncanonical version of the automaton has only six more states, one for each common modifier and one for ⟨*NonVoidType*⟩, than its canonical counterpart for the modified grammar snippet. We would of course have preferred a solution with greater lookahead capabilities, avoiding any modification to the original specification grammar. There is consolation in the fact that the grammar transformation is very simple when compared to the usual transformations required in order to get canonical LALR(1) grammars, but we are going to consider better solutions to this issue in the following section.

5.3 Shift-Resolve Parsers

As described in the previous section, noncanonical parsing allows to circumvent the limitation to bounded lookaheads in bottom-up parsers, but to keep the unambiguity guarantee. However, as we noted in Section 5.2.6.3 on the previous page, the preset bound on the reduced lookahead length—in practice the bound is $k = 1$—hampers the power of the noncanonical methods.

Only two noncanonical parsing methods allowing unbounded right context explorations are known: the Generalized Piecewise LR (GPLR) parsers of Schell (1979) and Overbey (2006), and the Noncanonical Discriminating Reverse (NDR) parsers of Farré and Fortes Gálvez (2004). In both methods, parsing is performed in quadratic time at worst. The alternative to noncanonical parsing is regular lookahead parsing, with a finite state automaton that explores an unbounded right context. Again, we lose the linear parsing time guarantee with practical implementations (Section 5.3.1).

Following the requirements enunciated on page 16, we want to have our cake and eat it too: we want linear time parsing, ambiguity detection, and no user defined bound on the lookahead length. Shift-resolve parsing is a new combination of the regular and noncanonical strategies that achieves all these properties. The originality of shift-resolve parsing lies in the following points.

- We propose a new parsing action, *resolve* (Section 5.3.2 on page 117), which combines the classical reduction with a pushback, i.e. it rewinds the stack down to the point where the reduction should take place. The exact amount of pushback is not fixed, but computed for each reduction as a minimal necessary length.
- By promoting the resolve action as a replacement for the reduce action, our parsers properly integrate noncanonical resolutions in the right context exploration. One could fear that a quadratic time complexity would stem from this combination. We avoid it by ensuring that the pushback lengths remain bounded.
- We present the construction of shift-resolve parsers as the determinization of a position automaton (Section 5.3.3 on page 119). We thus benefit from the flexibility of having a lattice of possible approximations (Section 4.2.1.3 on page 58). Hence, our method is highly generic and allows for tradeoffs between descriptional complexity and classes of accepted grammars.

5.3.1 Time Complexity Matters

We consider the LR(1) grammar with rules

$$\begin{gathered} S\xrightarrow{2}ACa,\ S\xrightarrow{3}BDb,\ A\xrightarrow{4}AD, \\ A\xrightarrow{5}a,\ B\xrightarrow{6}BC,\ B\xrightarrow{7}b,\ C\xrightarrow{8}c,\ D\xrightarrow{9}c, \end{gathered} \qquad (\mathcal{G}_{21})$$

generating the regular language $ac^+a|bc^+b$.

Nondeterminism Grammar $\mathcal{G}_{21}$ can require an unbounded lookahead if we consider LR(0)-based parsing methods, like for instance the LALR(1) automaton shown in Figure 5.11 on the following page. The automaton has a single *inadequate* state q_6 with items $C{\rightarrow}c{\boldsymbol{\cdot}}$ and $D{\rightarrow}c{\boldsymbol{\cdot}}$, reachable after

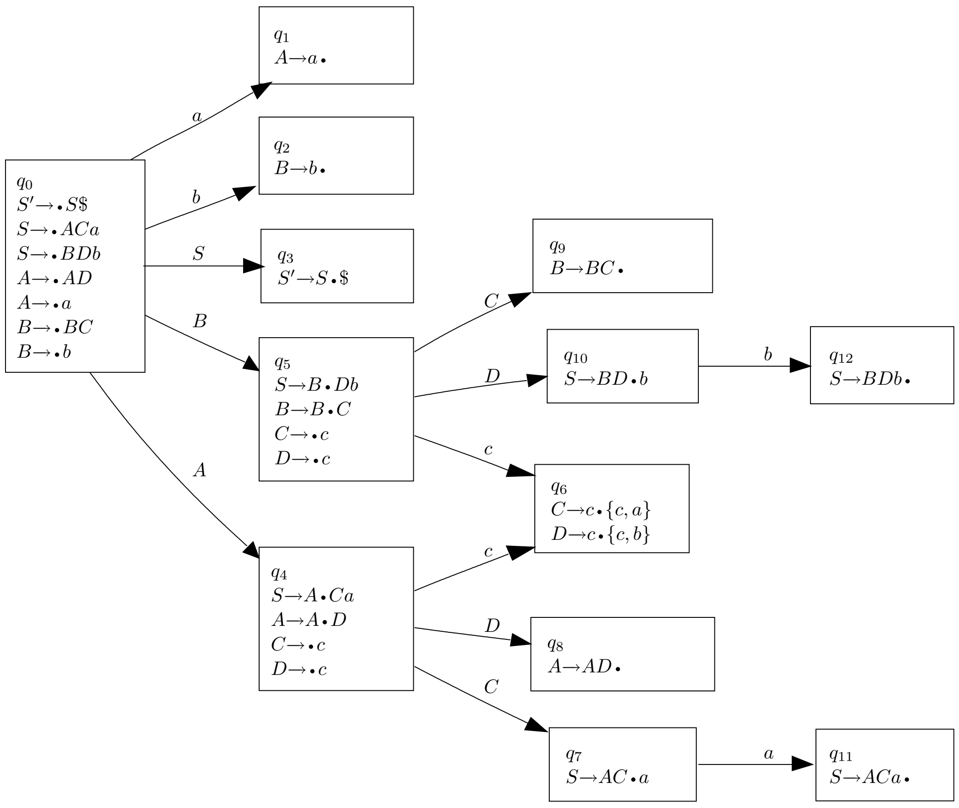


Figure 5.11: The LALR(1) automaton for $\mathcal{G}_{21}$.

reading both prefixes Ac and Bc. After reading Ac, the exact lookahead for the reduction to C is a, while the one for the reduction to D is c^+a. After reading Bc, the lookaheads are c^+b and b respectively. Thus, if we rely on the LR(0) automaton, then we need an unbounded terminal lookahead length in order to choose between the reduction to C or D, when seeing the last input symbol a or b after a sequence c^+.

Noncanonical Parsers Grammar $\mathcal{G}_{21}$ is not LALR(1). If we try to use more advanced parsers, $\mathcal{G}_{21}$ is not NSLR(1) (Tai, 1979) nor NLALR(1)—it is NSLR(2). Having to deal with such conflicts, which could almost be treated, requiring only one additional symbol of lookahead, is a common and frustrating issue. The example of the Java modifiers in Section 5.2.6.3 on page 113 was another instance of this phenomenon.

LR-Regular Parsers An implementation of unbounded regular lookahead explorations based on LR(0) approximations, for instance a R(2)LR(0) machine (Boullier, 1984), can associate the lookaheads $c^+a\,|\,b$ with the reduction to D and $a\,|\,c^+b$ with the reduction to C.

If we feed such a LR-Regular parser with the valid input string acc^na, it will first reduce the initial symbol a to A and shift the first c, thus reaching the inadequate state q_6. At this point, it will need to explore the entire right context c^na until the last a to decide the reduction of this first c to D. After the reduction of AD to A and the shift of the next c symbol, the parser has once more to resolve the same conflict in q_6 with an exploration of $c^{n-1}a$ until it reads the last a. The same phenomenon will happen until exhaustion of all the c symbols in the input. Thus the time complexity of parsing acc^na is quadratic.

More elaborated LR-Regular implementations (Boullier, 1984; Bermudez and Schimpf, 1990; Farré and Fortes Gálvez, 2001) can be defeated by slightly modified versions of $\mathcal{G}_{21}$ and present the same quadratic time complexity. We can consider for instance the grammar family

$$S\to ACc^ia\,|\,BDc^ib,\;\; A\to AD\,|\,a,\;\; B\to BC\,|\,b,\;\; C\to c,\;\; D\to c, \qquad (\mathcal{G}_{21}^i)$$

which is R(h)LR(k) for $h \geq 2$ and all k, but for which the RLR parser runs in quadratic time for $k \leq i$.

The question whether practical LR-Regular parsers, that employ a FSA for lookahead exploration, can work in quadratic time or not was not very clear: Seité (1987) on one hand and Boullier (1984) and Farré and Fortes Gálvez (2001) on the other hand claimed that their methods worked in linear time, but relied on results by Čulik and Cohen (1973) and Heilbrunner (1981) respectively that did not apply to their case. Grammar $\mathcal{G}_{21}$ and its variants settle this issue.

5.3.2 Example Parser

Overview We make with shift-resolve parsing the simplifying choice of always using completely reduced lookahead symbols: symbols as they appear in the grammar rule we are exploring, and cannot be reduced without reducing the entire rule, as with Leftmost LALR(1) parsing (see Section 5.2.4.1 on page 103).

As usual in noncanonical parsing (Aho and Ullman, 1972), a deterministic two-stack model is used to hold the current sentential form. The parsing (or left) stack corresponds to the traditional LR stack, while the input (or right) stack initially contains the input string. Two operations allow to move symbols from the top of one stack to the top of the other: a *shift* of a symbol from the input stack to the parsing stack, and a *pushback* of a bounded number of symbols the other way around. A *reduction* using rule $A\to\alpha$ removes the topmost $|\alpha|$ symbols from the parsing stack and pushes A on top of the input stack.

Table 5.2: Shift-resolve parsing table for $\mathcal{G}_{21}$.

	\$	a	b	c	S	A	B	C	D
q_0		s_4	s_5		s_1	s_2	s_3		
q_1	r_1’0								
q_2				s_8				s_6	s_7
q_3				s_8				s_9	s_{10}
q_4				s_8				r_5’0	r_5’0
q_5				s_8				r_7’0	r_7’0
q_6		s_{11}		s_8					
q_7				s_8				r_4’0	r_4’0
q_8		r_8’0	r_9’0	s_8				s_{12}	s_{13}
q_9				s_8				r_6’0	r_6’0
q_{10}			s_{14}	s_8					
q_{11}	r_2’0								
q_{12}		r_9’1		s_8				r_8’1	r_8’1
q_{13}			r_8’1	s_8				r_9’1	r_9’1
q_{14}	r_3’0								

We compute, for each reduction, the minimal bounded reduced lookahead length needed to discriminate it from other parsing actions. This lookahead exploration is properly integrated in the parser. Once the parser succeeds in telling which action should have been done, we either keep parsing if it was a shift, or need to reduce at an earlier point. The pushback brings the parser back at this point; we call the combination of a pushback and a reduction a *resolution*.

No cost is paid in terms of computational complexity, since shift-resolve parsers are linear in the length of the input text. A simple proof is that the only re-explored symbols are those pushed back. Since pushback lengths are bounded, and since each reduction gives place to a single pushback, the time linearity is clear since the number of reductions is linear with the input length.

Example Table 5.2 contains the parse table for shift-resolve parsing according to $\mathcal{G}_{21}$; the state numbers it indicates are not related to the LR(0) states of Figure 5.11 on page 116, but to the states in the shift-resolve parser construction. The table is quite similar to a LR(1) table, with the additional pushback length information, but describes a parser with much more lookahead information. States are denoted by q_i; shift entries are denoted as s_i where i is the new state of the parser; resolve entries are denoted as r_i’j where i is the number of the rule for the reduction and j

Table 5.3: The parse of the string $acca$ by the shift-resolve parser for $\mathcal{G}_{21}$.

parsing stack	input stack	actions
q_0	$acca\$$	s_4
$q_0 a q_4$	$cca\$$	s_8
$q_0 a q_4 c q_8$	$ca\$$	s_8
$q_0 a q_4 c q_8 c q_8$	$a\$$	r_8’0

We have just reached the first phrase in $acca\$$ that we can resolve with a completely reduced lookahead. This lookahead is a, and indeed it cannot be reduced any further in the rule $S{\rightarrow}ACa$. The lookahead allows the decision of resolving $C{\rightarrow}c$. The newly reduced nonterminal is pushed on the input stack, as usual in noncanonical parsing.

$q_0 a q_4 c q_8$	$Ca\$$	s_{12}
$q_0 a q_4 c q_8 C q_{12}$	$a\$$	r_9’1

We have here a non-null pushback: the resolve action r_9’1, which would have needed an unbounded terminal lookahead, is solved using the stacked C and the lookahead a. The pushback of length 1 emulates a reduced lookahead inspection of length 2.

$q_0 a q_4$	$DCa\$$	r_5’0
q_0	$ADCa\$$	s_2
$q_0 A q_2$	$DCa\$$	s_7
$q_0 A q_2 D q_7$	$Ca\$$	r_4’0
q_0	$AC\$$	s_2
$q_0 A q_2$	$Ca\$$	s_6
$q_0 A q_2 C q_6$	$a\$$	s_{11}
$q_0 A q_2 C q_6 a q_{11}$	$\$$	r_2’0
q_0	$S\$$	s_1
$q_0 S q_1$	$\$$	r_1’0, accept

the pushback length. The reduction according to rule $S'\overset{1}{\rightarrow}S$ indicates that the input is successfully parsed. Table 5.3 details the parsing steps on the valid input $acca$. Symbols are interleaved with states in the parsing stack in order to ease the reading, and are not actually used.

The originality of shift-resolve parsing resides in that Table 5.2 is not the result of a very precise position equivalence; in fact, we used the worst approximation we tolerate, namely item_0. Still, the parsing time is linear and no preset lookahead length was necessary.

5.3.3 Generation

We now describe how to extract a deterministic shift-resolve parser from a position automaton $\Gamma/{\equiv} = \langle Q, V_b, R, Q_s, Q_f \rangle$. Figure 5.12 is not a Rorschach test but the position automaton $\Gamma_{21}/\,\mathsf{item}_0$ for $\mathcal{G}_{21}$ using the equivalence

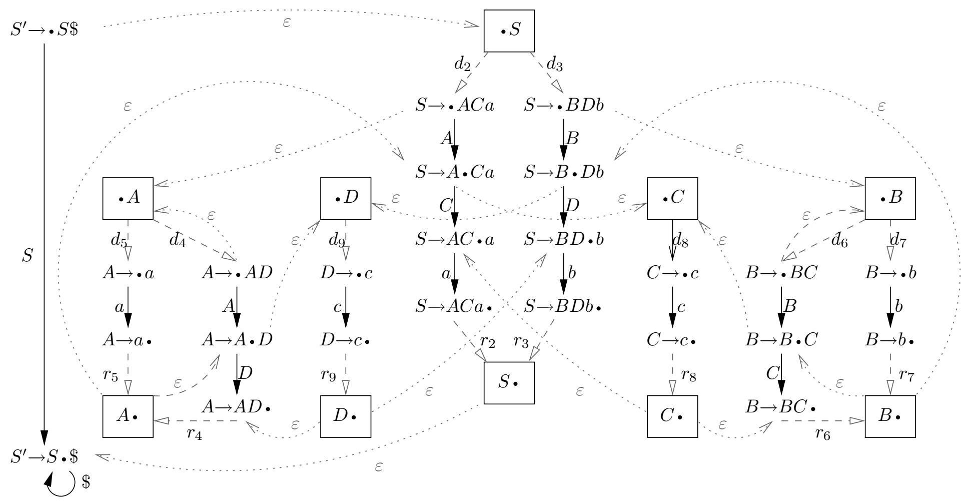


Figure 5.12: Position automaton for $\mathcal{G}_{21}$ using item_0.

relation item_0 between positions, more exactly the result of the variant construction detailed in Example 4.8 on page 56. We augment this position automaton with a cycle $q_f\$ \vdash q_f$ for all q_f in Q_f.

The shift-resolve parser generation algorithm is based on a subset construction that combines the states of the position automaton.

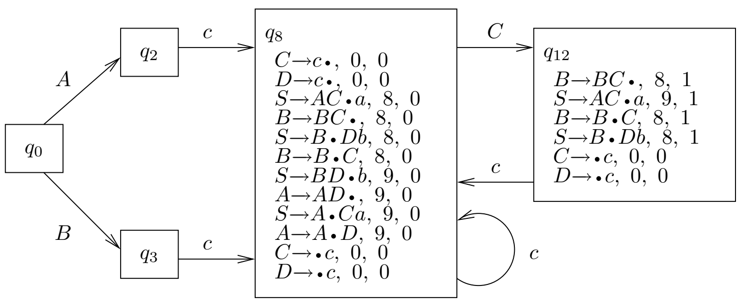
States of the Shift-Resolve Parser The states of the shift-resolve parser are sets of *items* $[p, sr, d]$, where

1. p is an equivalence class on $\mathcal{N}$ using $\equiv$—i.e. a state in $\Gamma/\equiv$—,
2. sr a parsing action—either a production number or 0 to code a shift—, and
3. d is a nonnegative integer to code the distance to the resolution point—a pushback length.

By convention, we assume that d is null whenever sr denotes a shift.

The initial state's item set is computed as $I_{q_0} = \mathsf{close}(\{[p_s, 0, 0] \mid p_s \in Q_s\})$, where the closure $\mathcal{C}$ of an item set I is the minimal set such that

$$\begin{aligned}\mathsf{close}(I) = I&\\ &\cup \{[p', 0, 0] \mid [p, sr, d] \in \mathsf{close}(I), pd_i \vdash p'\}\\ &\cup \{\iota \mid [p, sr, d] \in \mathsf{close}(I), pr_i \vdash p', \neg(\mathsf{null}(i) \text{ and } \mathsf{null}(I)),\\ &\quad ((sr = 0 \text{ and } \iota = [p', i, 0]) \text{ or } (sr \neq 0 \text{ and } \iota = [p', sr, d]))\},\end{aligned} \tag{5.59}$$



q_0
A
B
q_2
q_3
c
c
q_8
$C\!\to\! c\,\bullet$, 0, 0
$D\!\to\! c\,\bullet$, 0, 0
$S\!\to\! AC\,\bullet\, a$, 8, 0
$B\!\to\! BC\,\bullet$, 8, 0
$S\!\to\! B\,\bullet\, Db$, 8, 0
$B\!\to\! B\,\bullet\, C$, 8, 0
$S\!\to\! BD\,\bullet\, b$, 9, 0
$A\!\to\! AD\,\bullet$, 9, 0
$S\!\to\! A\,\bullet\, Ca$, 9, 0
$A\!\to\! A\,\bullet\, D$, 9, 0
$C\!\to\,\bullet\, c$, 0, 0
$D\!\to\,\bullet\, c$, 0, 0
C
c
c
q_{12}
$B\!\to\! BC\,\bullet$, 8, 1
$S\!\to\! AC\,\bullet\, a$, 9, 1
$B\!\to\! B\,\bullet\, C$, 8, 1
$S\!\to\! B\,\bullet\, Db$, 8, 1
$C\!\to\,\bullet\, c$, 0, 0
$D\!\to\,\bullet\, c$, 0, 0

Figure 5.13: Item sets of some states of the shift-resolve parser for Grammar $\mathcal{G}_{21}$.

where, by noting $\mathcal{L}$ the terminal language produced by a sequence of symbols, we discard superfluous ε-reductions with the help of the conditions

$$\begin{aligned} \mathsf{null}(i) \quad &\text{iff} \quad A \overset{i}{\to} \alpha,\ \mathcal{L}(\alpha) = \{\varepsilon\} && (5.60)\\ \mathsf{null}(I) \quad &\text{iff} \quad \forall [[x_b d_i (\genfrac{}{}{0pt}{}{\alpha X}{u_b} \bullet \genfrac{}{}{0pt}{}{\beta}{u'_b}) r_i x'_b]_{\equiv}, sr, d] \in I,\ \mathcal{L}(X) = \{\varepsilon\}. && (5.61)\end{aligned}$$

Transition from state item set I with symbol X in V is defined as follows.

$$\begin{aligned}\Delta(I,X) = \mathsf{close}(\{[p', sr, d'] \mid [p, sr, d] \in I, pX \vdash p',\\ ((sr = 0 \text{ and } d' = 0) \text{ or } (sr \neq 0 \text{ and } d' = d+1))\}). \quad (5.62)\end{aligned}$$

Figure 5.13 presents the details of the item sets computations for states q_8 and q_{12} of the shift-resolve parser presented in Table 5.2. For instance, in q_8, the closure on item $[C\!\to\! c\,\bullet,\ 0,\ 0]$ adds the items $[S\!\to\! AC\,\bullet\, a,\ 8,\ 0]$ and $[B\!\to\! BC\,\bullet,\ 8,\ 0]$ with the rule number of the reduction according to $C\!\to\! c$. The closure on the latter item adds the items $[S\!\to\! B\,\bullet\, Db,\ 8,\ 0]$ and $[B\!\to\! B\,\bullet\, C,\ 8,\ 0]$, and the closure on these two items adds the items $[D\!\to\,\bullet\, c,\ 0,\ 0]$ and $[C\!\to\,\bullet\, c,\ 0,\ 0]$. The transition on C to q_{12} leaves us with a kernel set comprising $[B\!\to\! BC\,\bullet, 8, 1]$ and $[S\!\to\! AC\,\bullet\, a, 9, 1]$ with incremented pushback lengths, on which we apply the same closure operations.

Parser Table Parser table entries, i.e. shifts and resolves, are computed from the item set I_q of each state q as follows.

$$T(q,X) = \begin{cases} \textbf{resolve } r & \text{with pushback } d\\ & \text{if } \forall [p, sr, d] \in I_q \text{ with } pX \vdash p', sr = r,\\ \textbf{accept} & \text{if } \forall [p, sr, d] \in I_q \text{ with } pX \vdash p', sr = 1 \text{ and } d = 0,\\ \textbf{shift} \text{ to } q' & \text{such that } I_{q'} = \Delta(I_q, X)\\ & \text{otherwise}\end{cases} \quad (5.63)$$

An experimental parser generator (with a much finer position equivalence) was implemented by José Fortes Gálvez, and is currently available from the Internet at the address: `http://serdis.dis.ulpgc.es/~ii-pl/ftp/dr`.

5.3.4 Shift-Resolve Grammars

Rejection Condition A grammar is ShRe($\equiv$) if and only if there does not exist two different states with identical item sets except for some pushback length(s).

It follows that the worst-case space complexity of the shift-resolve parser for $\mathcal{G}$ is $O(2^{|\Gamma/\equiv||P|})$. More powerful shift-resolve parsers can be obtained at the price of descriptional complexity if we add to the condition that one such state should be Δ-reachable from the other.

Grammar Classes The classes of ShRe(item_k)—item_k is defined in Equation 4.21 on page 59—grammars are not comparable with the classes of LR(k) grammars. For instance, we can produce a shift-resolve parser for the grammar with rules

$$S{\rightarrow}AC\,|\,BCb,\ A{\rightarrow}d,\ B{\rightarrow}d,\ C{\rightarrow}aCb\,|\,c \qquad (\mathcal{G}_{22})$$

using item_0, but $\mathcal{G}_{22}$ is not LR(k) for any value of k—as a matter of fact, it is not LR-Regular either.

Conversely, for $k > 0$, we can put an unbounded number of null nonterminals between a conflict and its resolution. For instance, the grammar with rules

$$S{\rightarrow}Aa\,|\,Bb,\ A{\rightarrow}cAE\,|\,c,\ B{\rightarrow}cBE\,|\,c,\ E{\rightarrow}\varepsilon \qquad (\mathcal{G}_{23})$$

is LR(1) but not ShRe($\equiv$) for any equivalence $\equiv$: once we reach the a or b symbol allowing to resolve, we would need to pushback an unbounded number of E symbols in order to have the c we intend to reduce on top of the parsing stack.

A simplification we made in the shift-resolve construction makes it possible for a LR(0) grammar not to be ShRe(item_k). This is the case for the grammar with rules

$$S\xrightarrow{2}Sa,\ S\xrightarrow{3}B,\ A\xrightarrow{4}a,\ B\xrightarrow{5}dBA,\ B\xrightarrow{6}b. \qquad (\mathcal{G}_{24})$$

Figure 5.14 shows how the resolution in a shift-resolve state with a single possible reduction (here $B{\rightarrow}b$) can be tricked into a useless exploration of the right context caused by the item_k approximations. The issue could be tackled very simply on the subset construction level, if we tested whether following r_i transitions in the nondeterministic automaton was necessary for a resolution, and if not, filled the entire parser table line with this resolution.

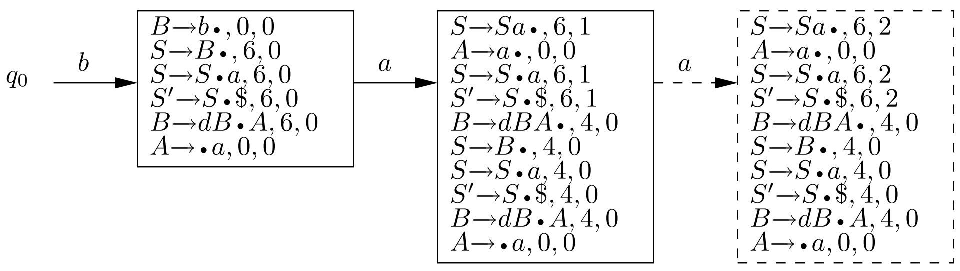


Figure 5.14: Item sets exhibiting the inadequacy of $\mathcal{G}_{24}$ using item_0.

5.3.5 Example: SML Case Expressions

In order to better illustrate and motivate the generation of a shift-resolve parser, we present its working on the conundrum of Standard ML case expressions, already described in Section 3.1.4 on page 31. For this, we consider item_0 approximations, and the construction of item sets starting with a kernel containing the two items in conflict in the LALR case

$$[\langle exp\rangle{\to}\textbf{case}\ \langle exp\rangle\ \textbf{of}\ \langle match\rangle\bullet, 0, 0] \tag{5.64}$$

$$[\langle match\rangle{\to}\langle match\rangle\bullet\text{'|'}\ \langle mrule\rangle, 0, 0]. \tag{5.65}$$

We apply the closure computation of Equation 5.59 to this set of items. The state of (5.64) has an r_5 transition to the state labeled by the dotted rule $\langle sfvalbind\rangle{\to}vid\ \langle atpats\rangle = \langle exp\rangle\bullet$, in the position graph, and thus we add the item

$$[\langle sfvalbind\rangle{\to}vid\ \langle atpats\rangle = \langle exp\rangle\bullet, 5, 0] \tag{5.66}$$

that commands a reduction with empty pushback length. This item adds

$$[\langle fvalbind\rangle{\to}\langle fvalbind\rangle\ \text{'|'}\ \langle sfvalbind\rangle\bullet, 5, 0] \tag{5.67}$$

to the itemset. The new item keeps the same reduction action 5, since it has to be performed *before* the reduction according to $\langle sfvalbind\rangle{\to}vid\ \langle atpats\rangle = \langle exp\rangle$ can take place. In turn, we add the items

$$[\langle fvalbind\rangle{\to}\langle sfvalbind\rangle\bullet, 5, 0] \tag{5.68}$$

$$[\langle fvalbind\rangle{\to}\langle fvalbind\rangle\bullet\text{'|'}\ \langle sfvalbind\rangle, 5, 0] \tag{5.69}$$

to the item set.

This last item (5.69) has a transition on the same symbol “|” in the position automaton as item (5.65), and thus we can apply the Δ transition function of Equation 5.62 to a kernel of an item set containing

$$[\langle \mathit{match} \rangle {\rightarrow} \langle \mathit{match} \rangle \text{ '|'} \bullet \langle \mathit{mrule} \rangle, 0, 0] \tag{5.70}$$

$$[\langle \mathit{fvalbind} \rangle {\rightarrow} \langle \mathit{fvalbind} \rangle \text{ '|'} \bullet \langle \mathit{sfvalbind} \rangle, 5, 1]. \tag{5.71}$$

Note that the pushback length has increased. This new item set contains further

$$[\langle \mathit{mrule} \rangle {\rightarrow} \bullet \langle \mathit{pat} \rangle \texttt{ => } \langle \mathit{exp} \rangle, 0, 0] \tag{5.72}$$

$$[\langle \mathit{pat} \rangle {\rightarrow} \bullet \mathit{vid}\ \langle \mathit{atpat} \rangle, 0, 0] \tag{5.73}$$

and

$$[\langle \mathit{sfvalbind} \rangle {\rightarrow} \bullet \mathit{vid}\ \langle \mathit{atpats} \rangle = \langle \mathit{exp} \rangle, 0, 0] \tag{5.74}$$

reached by d-transitions in the position graph. Their associated actions and pushback lengths are set to zero in order for them to recognize their rule rightparts. Note that, in this item set, we have a single item with $\langle \mathit{sfvalbind} \rangle$ as next expected symbol. According to the parsing table computation of Equation 5.63, there is a **resolve** r_5'1 entry for this item set and symbol. The initial conflict is solved in this case.

Let us return to the previous item set; the closure on item (5.64) produces another item by r_5:

$$[\langle \mathit{mrule} \rangle {\rightarrow} \langle \mathit{pat} \rangle \texttt{ => } \langle \mathit{exp} \rangle \bullet, 5, 0], \tag{5.75}$$

which in turn will bring two more items to the item set:

$$[\langle \mathit{match} \rangle {\rightarrow} \langle \mathit{mrule} \rangle \bullet, 5, 0] \tag{5.76}$$

$$[\langle \mathit{match} \rangle {\rightarrow} \langle \mathit{match} \rangle \bullet \text{'|' } \langle \mathit{mrule} \rangle, 5, 0]. \tag{5.77}$$

This last item (5.77) shares its position with (5.65), but with different actions. Hence, we know that at some point, we are bound to find two item sets differing only on the different pushback lengths. As we already know from Section 3.1.4 on page 31, this portion of the syntax of Standard ML is ambiguous, and bound to be rejected by the shift-resolve construction. Better, we have now a first lead on how to design better ambiguity tests than a standard LALR(1) construction, a subject we will treat in Chapter 6.

Ambiguity Detection 6

One especially desirable feature in a grammar for a programming language is freedom from ambiguity.
Ginsburg and Ullian (1966)

Syntactic ambiguity allows a sentence to have more than one syntactic interpretation. A classical example is the sentence “She saw the man with a telescope.”, where the phrase “with a telescope” can be associated to “saw” or to “the man”. The presence of ambiguities in a context-free grammar can severely hamper the reliability or the performance of the tools built from it. Sensitive fields, where CFGs are used to model the syntax, include for instance language acquisition (Cheung and Uzgalis, 1995), RNA analysis (Reeder et al., 2005; Brabrand et al., 2007), or controlled natural languages (ASD, 2005).

But our main interest in the issue stems from programming languages, as illustrated with the case studies of Section 3.1.2 and Section 3.1.4. With the recent popularity of general parsing methods, ambiguity issues tend to be overlooked. If this might seem acceptable for well established languages, for which the scrutiny of many implementors has pinpointed all ambiguous constructs, there always remains a risk of runtime exceptions if an unexpected ambiguity appears. The avoidance of such problems is clearly a desirable guarantee.

While proven undecidable by Cantor (1962), Chomsky and Schützenberger (1963) and Floyd (1962a), the problem of testing a context-free grammar for ambiguity can still be tackled approximatively. The approximations may result in two types of errors: *false negatives* if some ambiguities are left undetected, or *false positives* if some detected “ambiguities” are not actual ones.

In this chapter, we present an algorithm for the conservative detection of ambiguities, only allowing false positives. Our general approach is that of the verification of an infinite system: we build a finite position automaton (as described in Chapter 4) to approximate the grammar, and check for ambiguities in this abstract structure. Although the ambiguity of a position automaton, when brackets are ignored, is already a conservative test for ambiguities in the original grammar (Section 6.1), our verification improves on this immediate approach by ignoring some spurious paths (Section 6.2). We establish formal comparisons with several ambiguity checking methods, namely

- the bounded-length detection schemes (Gorn, 1963; Cheung and Uzgalis, 1995; Schröer, 2001; Jampana, 2005), which are not conservative tests,
- the LR-Regular condition (Čulik and Cohen, 1973), and
- the horizontal and vertical ambiguity condition (Brabrand et al., 2007).

Such comparisons are possible thanks to the generality of the position automaton model.

Finally, we present succinctly the implementation of the noncanonical unambiguity test in GNU Bison (Donnely and Stallman, 2006),[1] and we comment the experimental results we obtained in Section 6.3.

6.1 Regular Unambiguity

Ambiguity in a CFG is characterized as a property of its derivation trees: if two different derivation trees yield the same sentence, then we are facing an ambiguity.

In general, an ambiguity in a grammar $\mathcal{G}$ is thus the existence of two different sentences w_b and w'_b of $\mathcal{G}_b$ such that $w = w'$. For instance, we can consider again grammar $\mathcal{G}_7$ from Chapter 4 and the ambiguous sentence "She saw the man with a telescope.", obtained by two different bracketed sentences (repeated from page 49)

$$d_1\, d_2\, d_4\, pn\, r_4\, d_6\, v\, d_5\, d_3\, d\, n\, r_3\, d_8\, pr\, d_3\, d\, n\, r_3\, r_8\, r_5\, r_6\, r_2\, r_1 \tag{6.1}$$

$$d_1\, d_2\, d_4\, pn\, r_4\, d_7\, d_6\, v\, d_3\, d\, n\, r_3\, r_6\, d_8\, pr\, d_3\, d\, n\, r_3\, r_8\, r_7\, r_2\, r_1. \tag{6.2}$$

We can design a conservative ambiguity verification if we approximate the language $\mathcal{L}(\mathcal{G}_b)$ with a super language and look for such sentences in the

[1]The modified Bison source is available from the author's web page, at the address `http://www.i3s.unice.fr/~schmitz/`.

super language. By Theorem 4.11 on page 57, if such two sentences exist in $\mathcal{L}(\mathcal{G}_b)$, they also do in $\mathcal{L}(\Gamma/{\equiv}) \cap T_b^*$, and thus we can look for their existence in a finite position automaton $\Gamma/{\equiv}$. We call a CFG with no such pair of sentences *regular ≡-unambiguous*, or RU(≡) for short.

6.1.1 Regular Mutual Accessibility

The basic way to detect ambiguity in a NFA is to consider a graph where couples of states serve as vertices (Even, 1965). For consistency with the treatment of noncanonical unambiguity in the next section, we overview a method to find such sentences in $\Gamma/{\equiv}$ using an accessibility relation between couples of states. This kind of relations were used for LR(k) testing by Sippu and Soisalon-Soininen (1990, Chapter 10).

Definition 6.1. Let $\Gamma/{\equiv} = \langle Q, V_b, R, Q_s, Q_f \rangle$ be a position automaton for a grammar $\mathcal{G}$. The *regular mutual accessibility relation* rma is defined over Q^2 as the union $\mathsf{mad} \cup \mathsf{mar} \cup \mathsf{mat}$, where the primitive accessibility relations are

derivation $\mathsf{mad}{=}\mathsf{madl} \cup \mathsf{madr}$, where (q_1, q_2) madl (q_3, q_2) if and only if there exists i in P such that $q_1 d_i \vdash q_3$, and symmetrically for madr, (q_1, q_2) madr (q_1, q_4) if and only if there exists i in P such that $q_2 d_i \vdash q_4$,

reduction $\mathsf{mar}{=}\mathsf{marl} \cup \mathsf{marr}$, where (q_1, q_2) marl (q_3, q_2) if and only if there exists i in P such that $q_1 r_i \vdash q_3$, and symmetrically for marr, (q_1, q_2) marr (q_1, q_4) if and only if there exists i in P such that $q_2 r_i \vdash q_4$,

terminal mat, defined by (q_1, q_2) mat (q_3, q_4) if and only if there exists a in T, $q_1 a \vdash q_3$ and $q_2 a \vdash q_4$.

Lemma 6.2. *Let q_1, q_2, q_3, q_4 be states in Q, and u_b and v_b strings in T_b^* such that $q_1 u_b \vDash^* q_3$, $q_2 v_b \vDash^* q_4$. The relation (q_1, q_2) rma^* (q_3, q_4) holds if and only if $u = v$.*

Only if part. We proceed by induction on the number of steps n in the relation (q_1, q_2) rma^n (q_3, q_4). In the base case, $q_1 = q_3$ and $q_2 = q_4$, thus $u_b = v_b = \varepsilon$ and $u = v = \varepsilon$ holds.

For the induction step, we suppose that (q_1, q_2) rma^n (q_3, q_4) rma (q_5, q_6) for some states q_5, q_6 in Q and symbols χ, χ' in $T_b \cup \{\varepsilon\}$ such that $q_3 \chi \vdash q_5$ and $q_4 \chi' \vdash q_6$. By induction hypothesis, $u = v$. Depending on the exact relation used in the last step, we have the cases

derivation one of χ, χ' is a symbol in T_d and the other is ε, thus $uh(\chi) = u$ and $vh(\chi') = v$;

reduction one of χ, χ' is a symbol in T_r and the other is ε, thus $uh(\chi) = u$ and $vh(\chi') = v$;

terminal $\chi = \chi' = a$ in T, thus $uh(\chi) = ua$ and $vh(\chi') = va$.

In all the three cases, $uh(\chi) = vh(\chi')$. $\square$

If part. We proceed by induction on the length $|u_b| + |v_b|$. In the base case, $u_b = v_b = \varepsilon$ and thus $q_1 = q_3$ and $q_2 = q_4$. Hence (q_1, q_2) rma^* (q_3, q_4) holds.

For the induction step, there are three atomic ways to increase the length $|u_b|+|v_b|$ while keeping $u = v$: add a derivation symbol d_i to one of u_b, v_b, add a reduction symbol r_i to one of u_b, v_b, or add a terminal symbol a to both u_b and v_b. The three cases are handled by mad, mar and mat respectively. $\square$

The following theorem then holds when taking q_1 and q_2 in Q_s, q_3, q_4 in Q_f, and u_b, v_b in $\mathcal{L}(\mathcal{G}_b)$ in Lemma 6.2.

Theorem 6.3. *Let $\Gamma/{\equiv} = \langle Q, V_b, R, Q_s, Q_f\rangle$ be a position automaton for a grammar $\mathcal{G}$. If $\mathcal{G}$ is ambiguous, then there exist q_s, q'_s in Q_s and q_f, q'_f in Q_f such that (q_s, q'_s) rma^* (q_f, q'_f).*

6.1.2 Practical Concerns

In order to detect ambiguities in $\mathcal{G}$ according to Theorem 6.3, the steps one should follow are then:

1. Construct $\Gamma/{\equiv}$.

2. Compute the image of rma^* $(\{(q_s, q'_s)\})$ for (q_s, q'_s) in Q_s^2; this computation costs at worst $\mathcal{O}(|\Gamma/{\equiv}|^2)$ (Tarjan, 1972).

3. Explore this image and find sequence of rma steps that denote different bracketed strings, and report these sequences.

The issue with the report step of this algorithm is that the rma^* relation presents a lot of redundancies. For instance, the image rma^* (Q_s^2) is symmetric and reflexive, and we could save half of the computations by using this property. In general, we need to identify "equivalent" rma computations.

Right Bracketed Equality Case In practice, many position equivalences include a *right bracketed equality* equivalence $\mathsf{rightb}_=$ defined by

$$x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b} \bullet \genfrac{}{}{0pt}{}{\alpha'}{u'_b}) r_i x'_b \;\mathsf{rightb}_=\; y_b d_j(\genfrac{}{}{0pt}{}{\beta}{v_b} \bullet \genfrac{}{}{0pt}{}{\beta'}{v'_b}) r_j y'_b \text{ iff } u'_b x'_b = v'_b y'_b. \tag{6.3}$$

Let us consider now two different parse trees in $\mathcal{G}$ represented by two different strings w_b and w'_b in $\mathcal{L}(\mathcal{G}_b)$ with $w = w'$. The sentences w_b and w'_b share a longest common suffix v_b, such that $w_b = u_b r_i v_b$ and $w'_b = u'_b r_j v_b$ with $i \neq j$. If $\mathsf{rightb}_= \subseteq {\equiv}$, then Q_f is a singleton set $\{q_f\}$, and there is a single state q in Q such that $q_s u_b r_i v_b \models^* q_i r_i v_b \models q v_b \models^* q_f$ and

$q'_s u'_b r_j v_b \models^* q_j r_j v_b \models q v_b \models^* q_f$ for some q_s, q'_s in Q_s and q_i, q_j in Q. By Lemma 6.2, (q_s, q'_s) rma^* (q_i, q_j) rma^* (q_f, q_f). But we have now some precise information about where the ambiguity was detected. We state this variant of the algorithm formally in the following proposition.

Proposition 6.4. *Let $\Gamma/{\equiv} = \langle Q, V_b, R, Q_s, Q_f \rangle$ be a position automaton for a grammar $\mathcal{G}$ obtained for an equivalence relation $\equiv\, \supseteq \mathsf{rightb}_=$. If $\mathcal{G}$ is ambiguous, then there exist q_s, q'_s in Q_s and q_i, q_j, q in Q such that (q_s, q'_s) rma^* (q_i, q_j), $q_i r_i \vdash q$, $q_j r_j \vdash q$ and $i \neq j$.*

If we are only interested in verifying that a grammar is unambiguous, and not in returning a report on the potential ambiguities, then we can stop constructing the image rma^* (Q_s^2) as soon as we find a pair (q_i, q_j) matching the conditions of Proposition 6.4.

Whether we stop prematurely or not, the overall complexity of the algorithm is $\mathcal{O}(|\Gamma/{\equiv}|^2)$, since all we have to do in the last step of the algorithm is to find a (q_i, q_j) in the image rma^* (Q_s^2).

Merge Functions Using Proposition 6.4, we can report couples (q_i, q_j) instead of the full sequences of rma steps. Furthermore, all the reductions that should use merge actions appear in this set (see Section 3.3.2.2 on page 44 about merge functions).

Indeed, let $w_b = \alpha_0 a_1 \alpha_1 a_2 \cdots \alpha_{n-1} a_n \alpha_n$ and $w'_b = \beta_0 a_1 \beta_1 a_2 \cdots \beta_{n-1} a_n \beta_n$ where the a_k symbols are terminals in T and the α_k and β_k strings are brackets in $(T_d \cup T_r)^*$. A *merge* of productions $i \neq j$ occurs between r_i and r_j at position l, i.e.

$$\alpha_l = \alpha'_l r_i \alpha''_l \text{ and } \beta_l = \beta'_l r_j \beta''_l \tag{6.4}$$

if the matching d_i and d_j symbols also occur at a single position k, i.e.

$$\alpha_k = \alpha'_l d_i \alpha''_k \text{ and } \beta_k = \beta'_k d_j \beta''_k. \tag{6.5}$$

Such a pair of reduction symbols i and j implies the existence of a pair (q_i, q_j) in rma^* (Q_s^2).

Example We consider the syntax of layered Standard ML patterns, described in Section 3.1.2 on page 28. We add a rule $\langle ty \rangle \xrightarrow{8} tyvar$ to the rules of Section 3.1.2.1, and we obtain the position automaton depicted in Figure 6.1 on the next page with the alternate construction of Section 4.2.1.1 on page 54 for item_0. Thanks to this construction, the regular ambiguity detection algorithm runs in time $\mathcal{O}(|\mathcal{G}|^2)$ in the worst case.

The two bracketed sentences representing the trees shown in Figure 6.2 on the next page are

$$d_3\, d_4\, vid\, d_7\, r_7\, \textsf{as}\, d_2\, d_5\, vid\, r_5\, r_2\, r_4 : d_8\, tyvar\, r_8\, r_3$$

$$d_4\, vid\, d_7\, r_7\, \textsf{as}\, d_3\, d_2\, d_5\, vid\, r_5\, r_2 : d_8\, tyvar\, r_8\, r_3\, r_4$$

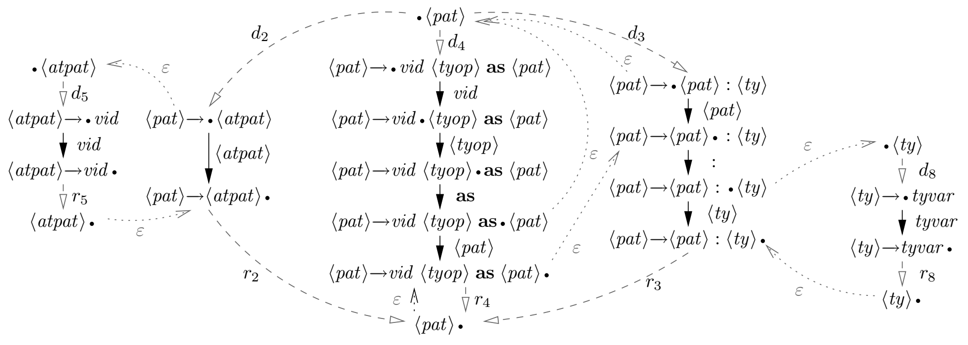


Figure 6.1: A portion of the position automaton for SML layered patterns using item_0.

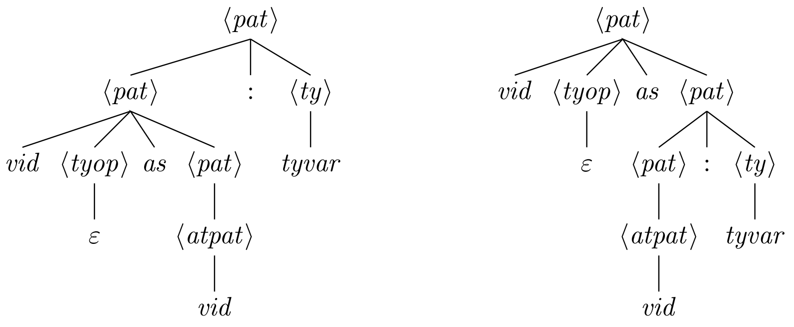


Figure 6.2: An ambiguity in the syntax of layered patterns.

We can follow the paths in Figure 6.1 and see that

$$([\bullet\langle pat\rangle],[\bullet\langle pat\rangle])\ \mathsf{rma}^*\ ([\langle pat\rangle{\rightarrow}\langle pat\rangle\ \langle ty\rangle\bullet],[\langle pat\rangle{\rightarrow}vid\ \langle tyop\rangle\ \textbf{as}\ \langle pat\rangle\bullet]).$$

Since furthermore $[\langle pat\rangle{\rightarrow}\langle pat\rangle\ \langle ty\rangle\bullet]$ and $[\langle pat\rangle{\rightarrow}vid\ \langle tyop\rangle\ \textbf{as}\ \langle pat\rangle\bullet]$ have transitions to $[\langle pat\rangle\bullet]$ on r_3 and r_4 respectively, we can conclude that it is possible to have an ambiguity arising from the use of productions 3 and 4, and we should devise a merge function to handle this case. Figure 6.2 confirms this potential ambiguity to be very real.

6.1.3 Comparisons

Being conservative is not enough for practical uses; after all, a program that always answers that the tested grammar is ambiguous is a conservative test. We compare here our regular unambiguity test with other means to prove the unambiguity of a context-free grammar—and identify issues that we will address in the next section.

6.1.3.1 Horizontal and Vertical Ambiguity

Brabrand et al. (2007) recently proposed an ambiguity detection scheme also based on regular approximations of the grammar language. Its originality lies in the decomposition of the ambiguity problem into two (also undecidable) problems, namely the horizontal and vertical ambiguity problems. The detection method then relies on the fact that a context-free grammar is unambiguous if and only if it is horizontal and vertical unambiguous. The latter tests are performed on a regular approximation of the grammar obtained through the technique of Mohri and Nederhof (2001). We first recall the definitions of horizontal and vertical ambiguity, and then translate them in terms of position automata.

Definition 6.5. [Brabrand et al. (2007)] A context-free grammar is *vertically unambiguous* if and only if, for all A in N with two different productions $A \to \alpha_1$ and $A \to \alpha_2$ in P, $\mathcal{L}(\alpha_1) \cap \mathcal{L}(\alpha_2) = \emptyset$.

It is *horizontally unambiguous* if and only if, for all productions $A \to \alpha$ in P, and for all decompositions $\alpha = \alpha_1\alpha_2$, $\mathcal{L}(\alpha_1) \bowtie \mathcal{L}(\alpha_2) = \emptyset$, where $\bowtie$ is the language *overlap* operator defined by $L_1 \bowtie L_2 = \{xyz \mid x, xy \in L_1, y \in T^+,\ \text{and}\ yz, z \in L_2\}$.

Definition 6.6. The automaton $\Gamma/{\equiv}$ is vertically ambiguous if and only if there exist an A in N with two different productions $i = A \to \alpha_1$ and $j = A \to \alpha_2$, and the bracketed strings x_b, x'_b, u_b, u'_b, w_b, and w'_b in T_b^* with $w = w'$ such that

$$[x_b d_i(\,\bullet\genfrac{}{}{0pt}{}{\alpha_1}{u_b})r_i x'_b]_\equiv w_b \models^* [x_b d_i(\genfrac{}{}{0pt}{}{\alpha_1}{u_b}\bullet\,)r_i x'_b]_\equiv \text{ and}$$

$$[x_b d_j(\,\bullet\genfrac{}{}{0pt}{}{\alpha_2}{u'_b})r_j x'_b]_\equiv w'_b \models^* [x_b d_j(\genfrac{}{}{0pt}{}{\alpha_2}{u'_b}\bullet\,)r_j x'_b]_\equiv.$$

The automaton $\Gamma/{\equiv}$ is horizontally ambiguous if and only if there is a production $i = A \to \alpha$ in P, a decomposition $\alpha = \alpha_1\alpha_2$, and the bracketed strings u_b, u'_b, v_b, v'_b, w_b, w'_b, x_b, x'_b, y_b, y'_b, z_b and z'_b in T_b^* with $v = v'$, $w = w'$, $y = y'$, $|y| \geq 1$ and $v_b y_b w_b \neq v'_b y'_b w'_b$ such that

$$[x_b d_i(\,\bullet\genfrac{}{}{0pt}{}{\alpha_1\alpha_2}{u_b u'_b})r_i x'_b]_\equiv v_b y_b w_b \models^* [x_b d_i(\genfrac{}{}{0pt}{}{\alpha_1}{u_b}\bullet\genfrac{}{}{0pt}{}{\alpha_2}{u'_b})r_i x'_b]_\equiv y_b w_b \models^* [x_b d_i(\genfrac{}{}{0pt}{}{\alpha_1\alpha_2}{u_b u'_b}\bullet\,)r_i x'_b]_\equiv$$

$$[x_b d_i(\,\bullet\genfrac{}{}{0pt}{}{\alpha_1\alpha_2}{z_b z'_b})r_i x'_b]_\equiv v'_b y'_b w'_b \models^* [x_b d_i(\genfrac{}{}{0pt}{}{\alpha_1}{z_b}\bullet\genfrac{}{}{0pt}{}{\alpha_2}{z'_b})r_i x'_b]_\equiv w'_b \models^* [x_b d_i(\genfrac{}{}{0pt}{}{\alpha_1\alpha_2}{z_b z'_b}\bullet\,)r_i x'_b]_\equiv.$$

Theorem 6.7. *Let $\mathcal{G}$ be a context-free grammar and $\Gamma/{\equiv}$ its position automaton. If $\mathcal{G}$ is RU($\equiv$), then $\Gamma/{\equiv}$ is horizontally and vertically unambiguous.*

Proof. If $\Gamma/{\equiv}$ is vertically ambiguous, then $x_b d_i w_b r_i x'_b$ and $x_b d_j w'_b r_j x'_b$ are two different sentences in $\mathcal{L}(\Gamma/{\equiv}) \cap T_b^*$ with $xwx' = xw'x'$, and thus $\mathcal{G}$ is regular $\equiv$-ambiguous. If $\Gamma/{\equiv}$ is horizontally ambiguous, then $x_b d_i v_b y_b w_b r_i x'_b$ and $x_b d_i v'_b y'_b w'_b r_i x'_b$ are two different sentences in $\mathcal{L}(\Gamma/{\equiv}) \cap T_b^*$ with $xvywx' = xv'y'w'x'$, and thus $\mathcal{G}$ is regular $\equiv$-ambiguous. □

Theorem 6.7 shows that the horizontal and vertical ambiguity criteria result in a better conservative ambiguity test than regular $\equiv$-ambiguity, although at a higher price: $\mathcal{O}(|\mathcal{G}|^5)$ in the worst case. Owing to these criteria, the technique of Brabrand et al. accomplishes to show that the palindrome grammar with rules

$$S \rightarrow aSa \,|\, bSb \,|\, a \,|\, b \,|\, \varepsilon \qquad (\mathcal{G}_{25})$$

is unambiguous, which seems impossible with our scheme.

6.1.3.2 LL-Regular Testing

In spite of their popularity as an alternative to the bottom-up parsers of the LR(k) family, top-down parser construction tests (Rosenkrantz and Stearns, 1970; Sippu and Soisalon-Soininen, 1982) would not be very relevant for practical ambiguity detection: the class of LL(k) grammars is arguably not large enough. The exception is the class of LL-Regular grammars (Jarzabek and Krawczyk, 1975; Nijholt, 1976; Poplawski, 1979), defined similarly to LR-Regular grammars (already mentioned in Section 3.2.2 on page 35) by generalizing the LL(k) condition. In particular, the Strong LL-Regular condition (Poplawski, 1979) has given birth to the LL(*) algorithm of the upcoming version 3 of ANTLR and to the ambiguity detection tool shipped with its IDE, ANTLRWorks (Bovet and Parr, 2007).

LL-Regular Condition In this section, we explicit the (non) relation between the LLR condition and the regular ambiguity test. Given a *left congruence* $\cong$ —i.e. if $x \cong y$ then $zx \cong zy$—that defines a finite regular partition Π of T^*—i.e. Π is a finite set $\{\pi_1, \ldots, \pi_n\}$ where each π_i is a regular set of strings in T^*—, a grammar $\mathcal{G}$ is LL(Π) if and only if

$$S \underset{\text{lm}}{\Longrightarrow}^* zA\delta \underset{\text{lm}}{\Longrightarrow} z\alpha\delta \underset{\text{lm}}{\Longrightarrow}^* zx, S \underset{\text{lm}}{\Longrightarrow}^* zA\delta \underset{\text{lm}}{\Longrightarrow} z\beta\delta \underset{\text{lm}}{\Longrightarrow}^* zy \text{ and } x \cong y \pmod{\Pi} \qquad (6.6)$$

imply

$$\alpha = \beta. \qquad (6.7)$$

This definition properly generalizes the LL(k) condition.

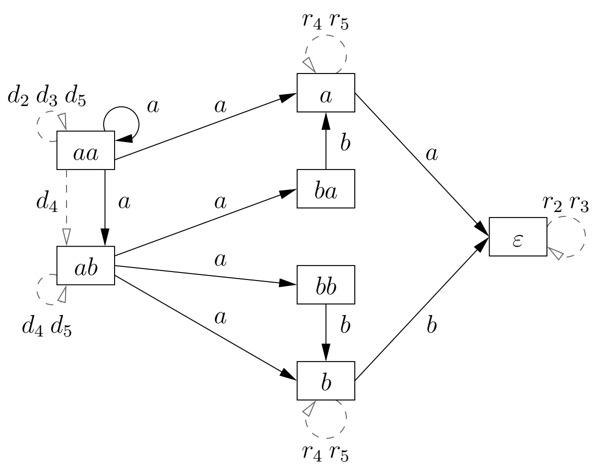


Figure 6.3: The terminal subautomaton of $\Gamma_{27}/\mathsf{right}_2$.

Comparison We define accordingly

$$x_b d_i({}^{\alpha}_{u_b} \bullet {}^{\alpha'}_{u'_b}) r_i x'_b \ \ \mathsf{right}_\Pi \ \ y_b d_j({}^{\beta}_{v_b} \bullet {}^{\beta'}_{v'_b}) r_j y'_b \text{ iff } u'x' \cong v'y' \ (\mathrm{mod}\ \Pi). \tag{6.8}$$

Special cases of right_Π partition T^* by considering two strings x and y to be congruent if and only if $k : x = k : y$, thus defining a right_k position equivalence.

Let us consider the grammar with rules

$$S\xrightarrow{2}BAb,\ A\xrightarrow{3}a,\ A\xrightarrow{4}aa,\ B\xrightarrow{5}bBa,\ B\xrightarrow{6}b. \tag{$\mathcal{G}_{26}$}$$

It is SLL(2) and thus LL(2) but not RU($\mathsf{item}_0 \wedge \mathsf{right}_2$), as witnessed by the two different sentences $d_2\,d_5\,b\,d_6\,b\,r_6\,a\,r_5\,d_3\,a\,r_3\,b\,r_2$ and $d_2\,d_5\,b\,d_6\,b\,r_6\,d_4\,a\,a\,r_4\,b\,r_2$ recognized by $\Gamma_{26}/(\mathsf{item}_0 \wedge \mathsf{right}_2)$. On the other hand, the grammar with rules

$$S\xrightarrow{2}aAb,\ S\xrightarrow{3}aAa,\ A\xrightarrow{4}ab,\ A\xrightarrow{5}a \tag{$\mathcal{G}_{27}$}$$

is not LL(2), but is RU(right_2) (see Figure 6.3).

On Using right_Π The incomparability of the RU(right_Π) grammar class with the LL(Π) one is an issue with the RU criterion rather than with the precision of right_Π. We could design a LL(Π) test on $\Gamma/\mathsf{right}_\Pi$ in the vein of the test described by Heilbrunner (1983).

Lemma 6.8. *Let $\nu = x_b d_i({}^{\alpha}_{u_b} \bullet {}^{\alpha'}_{u'_b}) r_i x'_b$ be a position in $\mathcal{N}$, w_b a bracketed string in T_b^*, and q_f a final state of $\Gamma/\mathsf{right}_\Pi$. If $[\nu]_{\mathsf{right}_\Pi} w_b \vDash^* q_f$ in $\Gamma/\mathsf{right}_\Pi$, then $wr_1 \cong u'x'$ (mod Π).*

Proof. We proceed by induction on n the number of steps in $[\nu]_{\mathsf{right}_\Pi} w_b \models^n q_f$. If $n = 0$, then ν is in μ_f, $w_b = u'_b = \varepsilon$ and $x'_b = r_1$. We consider for the induction step the path $[\nu]_{\mathsf{right}_\Pi} \chi w_b \models q w_b \models^n q_f$, where χ is in T_b. Using the induction hypothesis, any $\nu' = y_b d_j(\genfrac{}{}{0pt}{1}{\beta}{v_b} \bullet \genfrac{}{}{0pt}{1}{\beta'}{v_b}) r_j y' b$ in q is such that $w r_1 \cong v' y' \pmod{\Pi}$.

If $\chi \in T_d$, any ν' in q is such that $\beta = v_b = \varepsilon$ and $v' y' \cong u' x' \pmod{\Pi}$, and the lemma holds trivially by transitivity of $\cong$. If $\chi = a$, then $\nu = x_b d_i(\genfrac{}{}{0pt}{1}{\alpha}{u_b} \bullet \genfrac{}{}{0pt}{1}{a\alpha'}{a u'_b}) r_i x'_b$ and any ν' in q is such that $v' y' \cong u' x' \pmod{\Pi}$. Since Π is a left congruence, $w r_1 \cong u' x' \pmod{\Pi}$ implies that $a w r_1 \cong a u' x' \pmod{\Pi}$ and the lemma holds. If $\chi \in T_r$, then $\nu = x_b d_i(\genfrac{}{}{0pt}{1}{\alpha}{u_b} \bullet\ ) r_i x'_b$ and any ν' in q is such that $v' y' \cong x' \pmod{\Pi}$, and the lemma holds. □

Theorem 6.9. *Let $\mathcal{G}$ be a context-free grammar and $\Gamma/\mathsf{right}_\Pi$ its position automaton using right_Π. If $\Gamma/\mathsf{right}_\Pi$ is vertically ambiguous, then $\mathcal{G}$ is not LL(Π).*

Proof. By Definition 6.6, if $\Gamma/\mathsf{right}_\Pi$ is vertically ambiguous, then there exist an A in N with two different productions $i = A \rightarrow \alpha_1$ and $j = A \rightarrow \alpha_2$, and the bracketed strings x_b, x'_b, u_b, u'_b, w_b, and w'_b in T_b^* with $w = w'$ such that the states $q_1 = [x_b d_i(\ \bullet \genfrac{}{}{0pt}{1}{\alpha_1}{u_b}) r_i x'_b]_{\mathsf{right}_\Pi}$, $q_2 = [x_b d_j(\ \bullet \genfrac{}{}{0pt}{1}{\alpha_2}{u'_b}) r_j x'_b]_{\mathsf{right}_\Pi}$, $q_3 = [x_b d_i(\genfrac{}{}{0pt}{1}{\alpha_1}{u_b} \bullet\ ) r_i x'_b]_{\mathsf{right}_\Pi}$, and $q_4 = [x_b d_j(\genfrac{}{}{0pt}{1}{\alpha_2}{u'_b} \bullet\ ) r_j x'_b]_{\mathsf{right}_\Pi}$ verify

$$q_1 w_b \models^* q_3 \text{ and } q_2 w'_b \models^* q_4. \tag{6.9}$$

By Lemma 4.9 on page 56, there exist q_f and q'_f in Q_f such that $q_3 r_i x'_b / r_1 \models^* q_f$ and $q_4 r_j x'_b / r_1 \models^* q'_f$, and thus

$$q_1 w_b r_i x'_b / r_1 \models^* q_f \text{ and } q_2 w'_b r_j x'_b / r_1 \models^* q'_f. \tag{6.10}$$

By Lemma 6.8, $w(x'/r_1) r_1 \cong u x' \pmod{\Pi}$ and $w'(x'/r_1) r_1 \cong u' x' \pmod{\Pi}$. Since $w = w'$ and by transitivity of $\cong$, the conditions of Equation 6.6 are verified, but $\alpha_1 \neq \alpha_2$, and therefore $\mathcal{G}$ is not LL(Π). □

6.1.3.3 Bounded Length Detection Schemes

Many algorithms specifically designed for ambiguity detection look for ambiguities in all sentences up to some length (Gorn, 1963; Cheung and Uzgalis, 1995; Schröer, 2001; Jampana, 2005). As such, they fail to detect ambiguities beyond that length: they allow false negatives. Nonetheless, these detection schemes can vouch for the ambiguity of any string shorter than the given length; this is valuable in applications where, in practice, the sentences are of a small bounded length. The same guarantee is offered by the equivalence relation prefix_m defined for any fixed length m by[2]

$$x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b}\bullet\genfrac{}{}{0pt}{}{\alpha'}{u'_b})r_i x'_b \;\mathsf{prefix}_m\; y_b d_j(\genfrac{}{}{0pt}{}{\beta}{v_b}\bullet\genfrac{}{}{0pt}{}{\beta'}{v'_b})r_j y'_b \text{ iff } m :_b x_b u_b = m :_b y_b v_b. \tag{6.11}$$

Provided that $\mathcal{G}$ is not left-recursive, Γ/prefix_m is finite.

Lemma 6.10. *Let q_s be a state in Q_s, q a state in Q, and w_b be a string in T_b^* such that $q_s w_b \models^* q$ in Γ/prefix_m. If $|w| \leq m$, then for all ν in q, there exists ν_s in q_s such that $\nu_s \overset{w_b}{\rightarrowtail} \nu$.*

Proof. We proceed by induction on the number n of steps in $q_s w_b \models^n q$. If $n = 0$, then $w_b = \varepsilon$ and the lemma holds.

For the induction step, we consider $q_s w_b \chi \models^{n-1} q\chi \models q'$ with χ in T_b. For all ν' in q', there exists a position ν such that $\nu \overset{\chi}{\rightarrowtail} \nu'$. Since $|h(w_b\chi)| \leq m$, all the positions ν' in q' share the same left context $m :_b w_b\chi = w_b\chi$; thus any position ν such that $\nu \overset{\chi}{\rightarrowtail} \nu'$ has $m :_b w_b = w_b$ for left context, and belongs to q. We only need to invoke the induction hypothesis in order to find an appropriate ν_s in q_s such that $\nu_s \overset{w_b}{\rightarrowtail} \nu \overset{\chi}{\rightarrowtail} \nu'$. □

Lemma 6.10 yields immediately Theorem 6.11, which implies that reported potential ambiguities in sentences of lengths smaller than m are always actual ambiguities.

Theorem 6.11. *Let w_b and w'_b be two bracketed sentences in $\mathcal{L}(\Gamma/\mathsf{prefix}_m) \cap T_b^*$ with $w = w'$ and $|w| \leq m$. Then w_b and w'_b are in $\mathcal{L}(\mathcal{G}_b)$.*

6.1.3.4 LR Grammars

For practical applications, an ambiguity detection algorithm should perform better than the traditional tools in the line of YACC. Indeed, many available grammars for programming languages were coerced into the LALR(1) grammar class (modulo precedence and associativity disambiguation), and an ambiguity detection tool should not report any ambiguity for such grammars.

In this regard, the regular ambiguity test presented above performs unsatisfactorily: when using the item_0 equivalence, it finds some LR(0) grammars ambiguous, like for instance $\mathcal{G}_{28}$ with rules

$$S \xrightarrow{2} aAa,\ S \xrightarrow{3} bAa,\ A \xrightarrow{4} c. \tag{$\mathcal{G}_{28}$}$$

The sentences $d_2 a d_4 c r_4 a r_2$ and $d_2 a d_4 c r_4 a r_3$ are both in $\mathcal{L}(\Gamma_{28}/\mathsf{item}_0) \cap T_b^*$, as witnessed on Figure 6.4 on the next page.

[2] We define the *bracketed prefix* $m :_b x_b$ of a bracketed string x_b as the longest string in $\{y_b \mid x_b = y_b z_b \text{ and } |y| = m\}$ if $|x| > m$ or simply x_b if $|x| \leq m$.

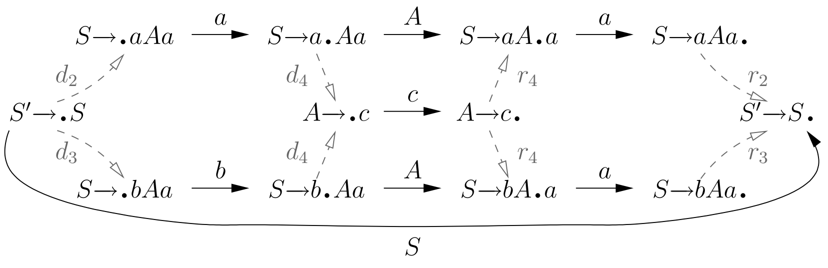


Figure 6.4: The position automaton $\Gamma_{28}/\mathsf{item}_0$ for $\mathcal{G}_{28}$.

6.2 Noncanonical Unambiguity

The LR algorithm (Knuth, 1965) hints at a solution to the weakness of regular unambiguity testing: we could consider nonterminal symbols in our verification and thus avoid spurious paths in the position automaton. A single direct step using a nonterminal symbol represents *exactly* the context-free language derived from it, much more accurately than any regular approximation we could make for this language.

6.2.1 Common Prefixes with Conflicts

Let us consider again the two sentences (6.1) and (6.2), but let us dismiss all the d_i symbols; the two sentences (6.12) and (6.13) we obtain are still different:

$$pn\; r_4\; v\; d\; n\; r_3\; pr\; d\; n\; r_3\; r_8\; r_5\; r_6\; r_2\;\; r_1 \tag{6.12}$$

$$pn\; r_4\; v\; d\; n\; r_3\; r_6\; pr\; d\; n\; r_3\; r_8\; r_7\; r_2\;\; r_1. \tag{6.13}$$

They share a longest common prefix $pn\, r_4\, v\, d\, n\, r_3$ before a *conflict*[3] between pr and r_6.

Observe that the two positions in conflict could be reached more directly in a PA by reading the prefix $NP\; v\; NP$. We obtain the two sentential forms

$$NP\; v\; NP\; pr\; d\; n\; r_3\; r_8\; r_5\; r_6\; r_2\;\; r_1 \tag{6.14}$$

$$NP\; v\; NP\; r_6\; pr\; d\; n\; r_3\; r_8\; r_7\; r_2\;\; r_1. \tag{6.15}$$

We cannot however reduce our two sentences to two identical sentential forms: our common prefix with one conflict $pn\, r_4\, v\, d\, n\, r_3\, r_6$ would reduce to a different prefix $NP\; VP$, and thus we do not reduce the conflicting reduction symbol r_6.

[3] Our notion of conflict coincides with that of LR(0) conflicts when one employs item_0.

The remaining suffixes $pr\ d\ n\ r_3\ r_8\ r_5\ r_6\ r_2\ \ r_1$ and $pr\ d\ n\ r_3\ r_8\ r_7\ r_2\ \ r_1$ share again a longest common prefix $pr\,d\,n\,r_3\,r_8$ before a conflict between r_5 and r_7; the common prefix reduces to PP, and we have the sentential forms

$$NP\ v\ NP\ PP\ r_5\ r_6\ r_2\ \ r_1 \tag{6.16}$$

$$NP\ v\ NP\ r_6\ PP\ r_7\ r_2\ \ r_1. \tag{6.17}$$

Keeping the successive conflicting reduction symbols r_5, r_6 and r_7, we finally reach a common suffix $r_2\ \ r_1$ that cannot be reduced any further, since we need to keep our conflicting reductions. The image of our two different reduced sentential forms (6.16) and (6.17) by h is a common sentential form $NP\ v\ NP\ PP$, which shows the existence of an ambiguity in our grammar.

We conclude from our small example that, in order to give preference to the more accurate direct path over its terminal counterpart, we should only follow the r_i transitions in case of conflicts or in case of a common factor that cannot be reduced due to the earlier conflicts. This general behavior is also the one displayed by noncanonical parsers.

In general, we consider two different sentences w_b and w'_b of $\mathcal{G}_b$ such that $w = w'$. They share a longest common prefix u_b with the d_i symbols ignored such that $w_b = u_b r_i v_{b,1}$ and $w'_b = u_b v'_{b,1}$ with $r_i \neq 1 : v'_{b,1}$. Let us call $u_{b,1} = u_b r_i$ and $u'_{b,1} = u_b$ the shortest common prefixes with one conflict. The remaining portions $v_{b,1}$ and $v'_{b,1}$, if different, also have a pair of shortest common prefixes $u_{b,2}$ and $u'_{b,2}$ with one conflict, so that $u_{b,1}u_{b,2}$ and $u'_{b,1}u'_{b,2}$ are shortest common prefixes with two conflicts.

Using the previous example but keeping the d_i symbols, the pairs of shortest common prefixes with one conflict are successively

$$u_{b,1} = d_1 d_2 d_4\ pn\ r_4 d_6\ v\ d_5 d_3\ d\ n\ r_3 \text{ and } u'_{b,1} = d_1 d_2 d_4\ pn\ r_4 d_7 d_6\ v\ d_3\ d\ n\ r_3 r_6,$$
$$u_{b,2} = d_8\ pr\ d_3\ d\ n\ r_3 r_8 r_5 \text{ and } u'_{b,2} = d_8\ pr\ d_3\ d\ n\ r_3 r_8,$$
$$u_{b,3} = r_6 \text{ and } u'_{b,3} = \varepsilon,$$
$$u_{b,4} = \varepsilon \text{ and } u'_{b,4} = r_7,$$

at which point there only remains a common suffix $v_b = r_2\ r_1$. With explicit d_i symbols, one can verify that the d_1 and d_2 symbols matching the r_1 and r_2 symbols of v_b are not in v_b, and thus that no reduction could occur inside v_b. Our initial sentences (6.1) and (6.2) are decomposed as $u_{b,1}u_{b,2}u_{b,3}u_{b,4}v_b$ and $u'_{b,1}u'_{b,2}u'_{b,3}u'_{b,4}v_b$.

The decomposition is not unique, but that does not hamper the soundness of our algorithm. The following proposition formalizes the decomposition we just performed.

Proposition 6.12. *Let w be a sentence of a context-free grammar $\mathcal{G}$ with two different parse trees represented by strings w_b and w'_b in V_b^*: $w = w'$.*

Then there exists $t \geq 1$ such that $w_b = u_{b,1} \cdots u_{b,t} v_b$ and $w'_b = u'_{b,1} \cdots u'_{b,t} v_b$ where $u_{b,1} \cdots u_{b,t}$ and $u'_{b,1} \cdots u'_{b,t}$ are shortest common prefixes of w_b and w'_b with t conflicts, and v_b is a common suffix of w_b and w'_b. □

6.2.2 Accessibility Relations

As in regular unambiguity testing, we implement the idea of common prefixes with conflicts in the mutual accessibility relations classically used to find common prefixes (Sippu and Soisalon-Soininen, 1990, Chapter 10). Mutual accessibility relations are used to identify couples of states accessible upon reading the same language from a starting couple (q_s, q'_s), which brings the complexity of the test down to a quadratic function in the number of transitions, and avoids the potential exponential blowup of a PA determinization.

We first solve two technical points.

1. The case where reduction transitions should be followed after a conflict is handled by considering pairs over $\mathbb{B} \times Q$ instead of Q: the boolean tells whether we followed a d_i transition for some rule i since the last conflict. In order to improve readability, we write $q\chi \vdash q'$ for q and q' in $\mathbb{B} \times Q$ if their states allow this transition to occur. The predicate $\backslash q$ in $\mathbb{B}$ denotes that we are allowed to ignore a reduction transition. Our starting couple (q_s, q'_s) has its boolean values initially set to true, and is thus in $(\mathsf{true} \times Q_s)^2$.

2. At the heart of the technique is the idea that return transitions should be followed only in case of a "conflict". In order to define what constitutes a conflict between two states (q_1, q_2) with $q_1 r_i \vdash q_3$, we recast the result of Theorem 5.9 by Aho and Ullman (1972) for the LR(0) case into the position automata framework: we translate the EFF_0 computation by a condition on the existence of a path $q_2 z \vDash^+ q_4$ with $z = y\chi$, y in T_d^*, χ in $T \cup T_r$ and $\chi \neq r_i$. We define accordingly the predicate

$$\mathsf{eff}(q, i) = \exists q' \in Q, y \in T_d^*, \chi \in T \cup T_r \text{ s.t. } qy\chi \vDash^+ q' \text{ and } \chi \neq r_i. \tag{6.18}$$

   This definition corresponds to the notion of conflict employed in the previous section, where symbols in T_d were ignored.

Definition 6.13. The primitive mutual accessibility relations over $(\mathbb{B} \times Q)^2$ are

shift mas defined by (q_1, q_2) mas (q_3, q_4) if and only if there exists X in V such that $q_1 X \vdash q_3$ and $q_2 X \vdash q_4$

derivation $\mathsf{mad}=\mathsf{madl} \cup \mathsf{madr}$ where (q_1, q_2) madl (q_3, q_2) if and only if $q_1 d_i \vdash q_3$ or $q_1 \varepsilon \vdash q_3$ and $\searrow q_3$ and symmetrically for madr, (q_1, q_2) madr (q_1, q_4) if and only if $q_2 d_i \vdash q_4$ or $q_2 \varepsilon \vdash q_4$, and $\searrow q_4$,

reduction mar defined by (q_1, q_2) mar (q_3, q_4) if and only if there exists i in P such that $q_1 r_i \vdash q_3$ and $q_2 r_i \vdash q_4$, and furthermore $\neg \searrow q_1$ or $\neg \searrow q_2$, and then $\neg \searrow q_3$ and $\neg \searrow q_4$,

conflict $\mathsf{mac}=\mathsf{macl} \cup \mathsf{macr}$ with (q_1, q_2) macl (q_3, q_2) if and only if there exist i in P, such that $q_1 r_i \vdash q_3$, $\mathsf{eff}(q_2, i)$ and $\neg \searrow q_3$, and symmetrically for macr, (q_1, q_2) macr (q_1, q_4) if and only if there exist i in P such that $q_2 r_i \vdash q_4$, $\mathsf{eff}(q_1, i)$, and $\neg \searrow q_4$.

The global *mutual accessibility relation* ma is defined as $\mathsf{mas} \cup \mathsf{mad} \cup \mathsf{mar} \cup \mathsf{mac}$.

These relations are akin to the item construction of a LR parser: the relation mas corresponds to a shift, the relation mad to an item closure, the relation mar to a goto, and the relation mac to a LR conflict. Note that, instead of a single boolean value, we could have considered vectors of boolean values, one for each possible rule i, or even bounded stacks of such vectors, that would have recorded the last d_i transitions followed in the accessibility relations. Then, the reduction relation mar could have operated much more precisely by checking that the last d_i seen matches the r_i transition under consideration. In practice, spurious ambiguity reports could be avoided when using these vectors. Observe however that, in order to prove the soundness of our algorithm, we only need to consider correct bracketed sentences, and thus upon reading a r_i transition in one such sentence, we know that the last d_j transition we followed was such that $i = j$, and the single boolean value suffices. Finally, let us point out that in our implementation (Section 6.3.2 on page 149) we stop the ma^* computation before mar has a chance to occur, and thus we do not even need this single boolean value.

Let us denote by map the union $\mathsf{mas} \cup \mathsf{mad} \cup \mathsf{mar}$. We explicit the relation between ma and common prefixes with t conflicts in the following lemma.

Lemma 6.14. *Let w_b and w'_b be two different sentences of $\mathcal{G}_b$ with a pair of shortest common prefixes with t conflicts $u_{b,1} \cdots u_{b,t}$ and $u'_{b,1} \cdots u'_{b,t}$. Furthermore, let ν_s and ν'_s be the corresponding starting positions in μ_s, and $u_{b,t} = u_b r_i$ and $u'_{b,t} = u_b$.*

Then, there exist ν_r, ν_t and ν'_t in $\mathcal{N}$ with

(i) $\nu_s \xrightarrow{u_{b,1} \cdots u_{b,t-1} u_b} \nu_r \xrightarrow{r_i} \nu_t$ *and* $\nu'_s \xrightarrow{u'_{b,1} \cdots u'_{b,t-1} u_b} \nu'_t$,

(ii) $([\nu_s]_\equiv, [\nu'_s]_\equiv)\ (\mathsf{mas} \cup \mathsf{mad})^* \circ (\mathsf{mac} \circ \mathsf{map}^*)^{t-1}([\nu_r]_\equiv, [\nu'_t]_\equiv)$, *and*

(iii) $([\nu_r]_\equiv, [\nu'_t]_\equiv)\ \mathsf{mac}\ ([\nu_t]_\equiv, [\nu'_t]_\equiv)$.

Proof. We first note that *(i)* always holds in Γ, and that together with the fact that $u_{b,1} \cdots u_{b,t}$ and $u'_{b,1} \cdots u'_{b,t}$ are longest common prefixes with t conflicts, it implies that *(iii)* holds. Let us then prove *(ii)* by induction on the number of conflicts t.

We can show using a simple induction on the length $|u_b|$ that, for $t = 1$, the common prefix u_b is such that $([\nu_s]_\equiv, [\nu'_s]_\equiv)\ (\mathsf{mas} \cup \mathsf{mad})^*\ ([\nu_r]_\equiv, [\nu'_1]_\equiv)$. If this length is zero, then $([\nu_s]_\equiv, [\nu'_s]_\equiv)\ \mathsf{mad}\ ([\nu_r]_\equiv, [\nu'_1]_\equiv)$ and thus *(ii)* holds. We then consider three atomic ways to increase this length while keeping u_b a common prefix: add an a symbol, a d_i, or an r_i symbol. The first two cases are clearly handled by mas and mad. In the third case, using Lemma 4.4 on page 52, there exist ν_A and ν'_A in $\mathcal{N}$ such that $\nu_s \xrightarrow{v_b} \nu_A \xrightarrow{A} \nu_r$ and $\nu'_s \xrightarrow{v_b} \nu'_A \xrightarrow{A} \nu'_1$. Applying the induction hypothesis, we see that $([\nu_s]_\equiv, [\nu'_s]_\equiv)\ (\mathsf{mas} \cup \mathsf{mad})^*\ ([\nu_A]_\equiv, [\nu'_A]_\equiv)$, and since furthermore $([\nu_A]_\equiv, [\nu'_A]_\equiv)\ \mathsf{mas}\ ([\nu_r]_\equiv, [\nu'_1]_\equiv)$, *(ii)* holds for ν_r and ν'_1.

Let us now prove the induction step for $t > 1$. By induction hypothesis and *(iii)*, $([\nu_s]_\equiv, [\nu'_s]_\equiv)\ (\mathsf{mas} \cup \mathsf{mad})^* \circ (\mathsf{mac} \circ \mathsf{map}^*)^{t-2} \circ \mathsf{mac}\ ([\nu_{t-1}]_\equiv, [\nu'_{t-1}]_\equiv)$, and we only need to prove that $([\nu_{t-1}]_\equiv, [\nu'_{t-1}]_\equiv)\ \mathsf{map}^*\ ([\nu_r]_\equiv, [\nu'_t]_\equiv)$. We proceed again by induction on the length of the common prefix u_b. The initial step for $|u_b| = 0$ is clear, and the induction step where we add an a or a d_i symbol also. The case where we add an r_i symbol triggers the use of mar if at least one of the two states verifies $\neg \searrow q$. Otherwise, we did not follow any r_i transition since the last d_i one, and thus Lemma 4.4 on page 52 applies. In all cases, *(ii)* holds. □

Let us call a grammar $\mathcal{G}$ such that $(q_s, q'_s)\ (\mathsf{mad} \cup \mathsf{mas})^* \circ\ \mathsf{mac} \circ \mathsf{ma}^*\ (q_f, q'_f)$ does not hold in $\Gamma/{\equiv}$ for any q_s and q'_s in Q_s and q_f and q'_f in Q_f *noncanonically* $\equiv$*-unambiguous*, or NU($\equiv$) for short. We just need to combine Lemma 6.14 with Proposition 6.12 in order to prove our main result:

Theorem 6.15. *Let $\mathcal{G}$ be a context-free grammar and $\equiv$ a position equivalence relation. If $\mathcal{G}$ is ambiguous, then $\mathcal{G}$ is not NU($\equiv$).*

Complexity The complexity of our algorithm depends mostly on the equivalence relation we choose to quotient the position graph. Supposing that we choose an equivalence relation $\equiv$ of finite index and of decidable computation of complexity $\mathcal{C}(\Gamma/{\equiv})$, then we need to build the image $\mathsf{ma}^*\ (\{(q_s, q'_s)\})$. This step and the search for a conflict in this image can both be performed in time $\mathcal{O}(|\Gamma/{\equiv}|^2)$. The overall complexity of our algorithm is thus $\mathcal{O}(\mathcal{C}(\Gamma/{\equiv}) + |\Gamma/{\equiv}|^2)$.

The complexity $\mathcal{C}(\Gamma/\mathsf{item}_0)$ of the construction of the position automaton Γ/item_0 is linear with the size of the resulting nondeterministic position

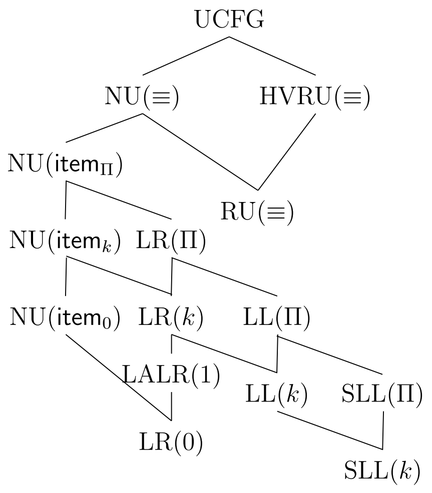


Figure 6.5: Context-free grammar classes inclusions. The classes parameterized by k, Π and $\equiv$ denote the full classes for any fixed k, any finite regular partition Π of T^*, and any position equivalence relation $\equiv$ with finite index respectively.

---

automaton. The overall complexity of our ambiguity detection algorithm when one uses item_0 is therefore $\mathcal{O}(|\mathcal{G}|^2)$.

6.2.3 Comparisons

We compare here our ambiguity detection algorithm with some of the other means to test a context-free grammar for ambiguity we are aware of. We first establish the edge of our algorithm over the regular ambiguity test of Section 6.1. Then, we show that our method performs better than LR-Regular testing when using an appropriate position equivalence.

The lattice of context-free grammar classes inclusions presented in Figure 6.5 sums up the results of our comparisons. Practical results will be discussed in Section 6.3.3.

6.2.3.1 Regular Ambiguity

Theorem 6.17, along with the example of $\mathcal{G}_{28}$, shows a strict improvement of our method over the simple algorithm discussed in Section 6.1.

Lemma 6.16. *Let q_1, q_2, q_3 and q_4 be states in Q such that (q_1, q_2) ma^* (q_3, q_4). Then, there exist u_b and u'_b in T_b^* such that $u = u'$, $q_1 u_b \models^* q_3$ and $q_2 u'_b \models^* q_4$.*

Proof. We proceed by induction on the number of steps n in (q_1, q_2) ma^n (q_3, q_4). If $n = 0$, then $q_1 = q_3$ and $q_2 = q_4$, hence $u_b = u'_b = \varepsilon$ fit our requirements.

Let us prove the induction step. Suppose we have two states q_5 and q_6 such that (q_3, q_4) ma (q_4, q_6), and, using the induction hypothesis, two strings u_b and u'_b in T_b^* such that $u = u'$, $q_1 u_b \vDash^* q_3$ and $q_2 u'_b \vDash^* q_4$. Let us find v_b and v'_b two strings in T_b^* such that $v = v'$, $q_3 v_b \vDash^* q_5$ and $q_4 v'_b \vDash^* q_6$ for each of the primitive mutual accessibility relations. For mas, $v_b = v'_b$ such that $X \Rightarrow^* v_b$ in $\mathcal{G}_b$ fit; for mad, $v_b = v'_b = d_i$ do; for mar, $v_b = v'_b = r_i$ do; at last, for macl, $v_b = r_i$ and $v'_b = \varepsilon$ do and symmetrically for macr. □

Theorem 6.17. *If $\mathcal{G}$ is RU($\equiv$), then it is also NU($\equiv$).*

Proof. Since the relation $(\mathsf{mad} \cup \mathsf{mas})^* \circ \mathsf{mac} \circ \mathsf{ma}^*$ that defines noncanonical $\equiv$-ambiguity is included in ma^*, Lemma 6.16 also applies to it. Therefore, if $\mathcal{G}$ is noncanonically $\equiv$-ambiguous, then there are two strings u_b and u'_b in T_b^* such that $u = u'$ and $q_s u_b \vDash^* q_f$ and $q_s u'_b \vDash^* q'_f$, i.e. u_b and u'_b are in $\mathcal{L}(\Gamma/{\equiv}) \cap T_b^*$. Note that the presence of the first occurrence of mac in the relation implies that the two bracketed strings u_b and u'_b are different, which concludes the proof. □

Horizontal and Vertical Ambiguity Owing to its strong criteria, the technique of Brabrand et al. (noted HVRU($\equiv$) in Figure 6.5 on the preceding page) accomplishes to show that the palindrome grammar $\mathcal{G}_{25}$ is unambiguous, which seems impossible with noncanonical unambiguity. On the other hand, even when they employ *unfolding* techniques to refine their grammar approximations, they are always limited to regular approximations, and fail to see that the LR(0) grammar with rules

$$S{\rightarrow}AA,\ A{\rightarrow}aAa\,|\,b \qquad (\mathcal{G}_{29})$$

is unambiguous. The two techniques are thus incomparable, and might benefit from each other.

Bounded Length Detection Schemes Since noncanonical unambiguity refines regular unambiguity, Theorem 6.11 shows that reported potential ambiguities with Γ/prefix_m in sentences of lengths smaller than m are always actual ambiguities.

Outside of the specific situation of languages that are finite in practice, bounded length detection schemes can be quite costly to use. The performance issue can be witnessed with the two families of grammars $\mathcal{G}_{30}^n$ and $\mathcal{G}_{31}^n$ with rules

$$S{\rightarrow}A\,|\,B_n,\ A{\rightarrow}Aaa\,|\,a,\ B_1{\rightarrow}aa,\ B_2{\rightarrow}B_1B_1,\ \ldots,\ B_n{\rightarrow}B_{n-1}B_{n-1} \qquad (\mathcal{G}_{30}^n)$$

$$S\to A\,|\,B_n a,\ A\to Aaa\,|\,a,\ B_1\to aa,\ B_2\to B_1B_1,\ \ldots,\ B_n\to B_{n-1}B_{n-1},\quad (\mathcal{G}_{31}^n)$$

where $n \geq 1$. In order to detect the ambiguity of $\mathcal{G}_{31}^n$, a bounded length algorithm would have to explore all strings in $\{a\}^*$ up to length $2^n + 1$. Our algorithm correctly finds $\mathcal{G}_{30}^n$ unambiguous and $\mathcal{G}_{31}^n$ ambiguous in time $\mathcal{O}(n^2)$ using item_0.

LL-Regular Testing The class of LL-Regular grammars considered in Section 6.1.3.2 is a proper subset of the class of LR-Regular grammars we consider next (Heilbrunner, 1983).

6.2.3.2 LR(k) and LR-Regular Testing

Conservative algorithms do exist in the programming language parsing community, though they are not primarily meant as ambiguity tests. Nonetheless, a full LALR or LR construction is often used as a practical test for non ambiguity (Reeder et al., 2005). The LR(k) testing algorithms (Knuth, 1965; Hunt III et al., 1974, 1975) are much more efficient in the worst case and provided our initial inspiration.

Our position automaton is a generalization of the item grammar or nondeterministic automaton of these works, and our test looks for ambiguities instead of LR conflicts, resulting in a much more efficient test. Let us consider again $\mathcal{G}_{30}^n$: it requires a LR(2^n) test for proving its unambiguity, but it is simply NU(item_0).

LR-Regular Condition One of the strongest ambiguity tests available is the LR-Regular condition (see Section 3.2.2 on page 35 for a presentation, and the works of Čulik and Cohen (1973) and Heilbrunner (1983)): instead of merely checking the k next symbols of lookahead, a LRR parser considers regular equivalence classes on the entire remaining input to infer its decisions. Given Π a finite regular partition of T^* that defines a *left congruence* $\cong$ for string concatenation (if this is not the case, a refinement of Π which is also a left congruence can always be constructed and used instead), a grammar $\mathcal{G}$ is LR(Π) if and only if

$$S \underset{\text{rm}}{\Longrightarrow}^* \delta Ax \underset{\text{rm}}{\Longrightarrow} \delta\alpha x, S \underset{\text{rm}}{\Longrightarrow}^* \gamma By \underset{\text{rm}}{\Longrightarrow} \gamma\beta y = \delta\alpha z \text{ and } x \cong z \ (\mathrm{mod}\,\Pi) \quad (6.19)$$

implies

$$A\to\alpha = B\to\beta, \delta = \gamma \text{ and } y = z. \quad (6.20)$$

This definition is a proper generalization of the LR(k) condition. Practical implementations (Baker, 1981; Boullier, 1984; Seité, 1987; Bermudez and Schimpf, 1990; Farré and Fortes Gálvez, 2001) of the LRR parsing method actually compute, for each inadequate LR state, a finite state automaton that attempts to discriminate between the x and z regular lookaheads. The final states of this automaton act as the partitions of Π.

$\textsf{\textit{item}}_\Pi$ *equivalence* Our test for ambiguity is strictly stronger than the LR(Π) condition with the equivalence relation $\textsf{item}_\Pi{=}\textsf{item}_0 \cap \textsf{look}_\Pi$, where $\textsf{look}_\Pi$ is defined by

$$x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b}\bullet\genfrac{}{}{0pt}{}{\alpha'}{u'_b})r_i x'_b \;\; \textsf{look}_\Pi \;\; y_b d_j(\genfrac{}{}{0pt}{}{\beta}{v_b}\bullet\genfrac{}{}{0pt}{}{\beta'}{v'_b})r_j y'_b \text{ iff } x' \cong y' \pmod{\Pi}. \tag{6.21}$$

This position equivalence generalizes $\textsf{item}_k$ defined on page 59.

We first state properties of $\Gamma/\textsf{item}_\Pi$ in terms of left and right contexts of its equivalence classes in the following two lemmas.

Lemma 6.18. *Let $\Gamma/\textsf{\textit{item}}_\Pi$ be a position automaton for $\mathcal{G}$ using $\textsf{\textit{item}}_\Pi$, and q_s be the initial state of $\Gamma/\textsf{\textit{item}}_\Pi$, q a state in Q and δ_b a bracketed string in $(V \cup T_d)^*$. If $q_s\delta_b \models^* q$ in $\Gamma/\textsf{\textit{item}}_\Pi$, then there exists a position $\nu = x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b}\bullet\genfrac{}{}{0pt}{}{\alpha'}{u'_b})r_i x'_b$ such that $d_1\delta_b = \hat{x}_b d_i \alpha$.*

Proof. We proceed by induction on the length $|\delta_b|$. For the basis, $\delta_b = \varepsilon$ and thus any ν in q_s is of form $d_1(\bullet\genfrac{}{}{0pt}{}{S}{w_b})r_1$ for some bracketed sentence w_b, and $d_1\delta_b = d_1$ holds.

For the induction step, we consider the path $q_s\delta_b\chi \models^* q'\chi \models q$ in $\Gamma/\textsf{item}_\Pi$ for some χ in $V \cup T_d$. Since we are using $\textsf{item}_\Pi$, we fix β' in V^*, j in P and π in Π such that $q' = \{y_b d_j(\genfrac{}{}{0pt}{}{\beta}{v_b}\bullet\genfrac{}{}{0pt}{}{\beta'}{v'_b})r_j y'_b \mid y' \in \pi\}$. By induction hypothesis, there exists a position ν' in q' such that $d_1\delta_b = \hat{y}_b d_j \beta$. In fact, since we are using $\textsf{item}_\Pi$, there are many positions of this form in q'.

Two possibilities arise, depending on whether χ is in T_d or in V.

If $\chi = d_i$ in T_d for $i = A{\rightarrow}\alpha$, then the positions in q' verify $\beta' = A\beta''$, including the positions that verify the induction hypothesis. There exist $\nu'' = z_b d_j(\genfrac{}{}{0pt}{}{\beta}{w_b}\bullet\genfrac{}{}{0pt}{}{\beta'}{w'_b})r_j z'_b$ in q' and $\nu = x_b d_i(\bullet\genfrac{}{}{0pt}{}{\alpha'}{u'_b})r_i x'_b$ in q such that $\nu'' \overset{\chi}{\rightarrowtail} \nu$ and $x' \cong w'z' \pmod{\Pi}$.

There also exists $\nu' = y_b d_j(\genfrac{}{}{0pt}{}{\beta}{v_b}\bullet\genfrac{}{}{0pt}{}{\beta'}{v'_b})r_j y'_b$ in q' that verifies the induction hypothesis $d_1\delta_b = \hat{y}_b d_j \beta$, and with $v'_b = w'_b$ derived from the same β' as ν'', and with $y' \cong z' \pmod{\Pi}$. Since $\cong$ is a left congruence, $v'y' \cong w'z' \pmod{\Pi}$, and by transitivity $v'y' \cong x' \pmod{\Pi}$. The position ν''' such that $\nu' \overset{\chi}{\rightarrowtail} \nu'''$ verifies $\nu'''\ \textsf{item}_\Pi\ \nu$, and thus also belongs to q. This position ν''' has $y_b d_j v_b$ for left context, with $\widehat{y_b d_j v_b d_i} = \hat{y}_b d_j \beta d_i = d_1 \delta_b d_i = d_1\delta_b\chi$, and therefore ν''' verifies the lemma's condition.

If $\chi = X$ in V, then all the positions ν' in q' verify $\beta X = \alpha$, $\beta' = X\alpha'$ and $i = j$, and there exists a position $\nu = x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b}\bullet\genfrac{}{}{0pt}{}{\alpha'}{u'_b})r_i x'_b$ in q such that $\nu' \overset{\chi}{\rightarrowtail} \nu$ for each one of them, with $x_b = y_b$ and $x'_b = y'_b$, including for the positions ν' that verify the induction hypothesis. In the latter case, $d_1\delta_b\chi = \hat{y}_b d_j \beta\chi = \hat{y}_b d_j \beta X = \hat{y}_b d_j \alpha = \hat{y}_b d_i \alpha = \hat{x}_b d_i \alpha$. □

Lemma 6.19. *Let $\Gamma/\textsf{\textit{item}}_\Pi$ be a position automaton for $\mathcal{G}$ using $\textsf{\textit{item}}_\Pi$, and q_f be the final state of $\Gamma/\textsf{\textit{item}}_\Pi$, $\nu = x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b}\bullet\genfrac{}{}{0pt}{}{\alpha'}{u'_b})r_i x'_b$ a position of $\mathcal{N}$ and*

w_b a bracketed terminal string in T_b^*. If $[\nu]_{\mathsf{item}_\Pi} w_b \models^* q_f$, then there exists z_b in T_b^* such that $\alpha' r_i \hat{x}'_b \Rightarrow_b^* z_b$ and $w \cong z$ (mod Π).

Proof. We proceed by induction on the length $|w_b|$. For the induction basis, we consider $w_b = \varepsilon$, thus ν in μ_f is of form $d_1(\genfrac{}{}{0pt}{}{S}{u_b} \bullet)r_1$, with lookahead r_1 such that $h(r_1) = \varepsilon \cong \varepsilon$ by reflexivity.

For the induction step, we consider a transition $q\chi \vdash q'$ with χ in T_b and q and q' two states of Γ/item_Π. Any position ν in q has a transition on this symbol χ to some position ν' (not necessarily in q'); in order to apply the induction hypothesis, we examine the case where $[\nu']_{\mathsf{item}_\Pi} w_b \models^* q_f$. Then, by induction hypothesis, $\nu' = y_b d_j(\genfrac{}{}{0pt}{}{\beta}{v_b} \bullet \genfrac{}{}{0pt}{}{\beta'}{v'_b})r_j y'_b$ is such that there exists z_b in T_b^*, $\beta' r_j \hat{y}'_b \Rightarrow_b^* z_b$ and $z \cong w$ (mod Π). Three cases arise depending on χ.

If $\chi = a$ in T, then $\nu = y_b d_j(\genfrac{}{}{0pt}{}{\alpha}{u_b} \bullet \genfrac{}{}{0pt}{}{a\beta'}{av'_b})r_j y'_b$ verifies $a\beta' r_j \hat{y}'_b \Rightarrow_b^* a z_b$ and $az \cong aw$ (mod Π) since $\cong$ is a left congruence.

If $\chi = d_j$ in T_d, then $\nu = x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b} \bullet \genfrac{}{}{0pt}{}{B\alpha'}{d_j v'_b r_j u'_b})r_i x'_b$ is such that $y'_b = u'_b r_i x'_b$, thus $B\alpha' r_i \hat{x}'_b \Rightarrow_b d_j \beta' r_j \alpha' r_i \hat{x}'_b = d_j \beta' r_j \hat{y}'_b \Rightarrow_b^* d_j z_b$ with $z \cong w$ (mod Π).

If $\chi = r_i$ in T_r, then $\nu = x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b} \bullet)r_i v'_b r_j y'_b$ is such that $r_i \widehat{v'_b r_j y'_b} = r_i \beta' r_j \hat{y}'_b \Rightarrow_b^* r_i z_b$ with $z \cong w$ (mod Π). □

Theorem 6.20. *If $\mathcal{G}$ is LR(Π), then it is also NU(item_Π).*

Proof. Let us suppose that $\mathcal{G}$ is noncanonically item_Π-ambiguous. We have the relation

$$(q_s, q'_s)\ (\mathsf{mas} \cup \mathsf{mad})^*\ (q_1, q_2)\ \mathsf{mac}\ (q_3, q_2)\ \mathsf{ma}^*\ (q_f, q'_f). \tag{6.22}$$

The first part (q_s, q_s) $(\mathsf{mas} \cup \mathsf{mad})^*$ (q_1, q_2) of Equation 6.22 implies that $q_s \delta_d d_i \alpha \models^* q_1$ and $q_s \gamma_d d_j \beta \models^* q_2$ with $\delta\alpha = \gamma\beta$ for some δ_d and γ_d in $(V \cup T_d)^*$, i and j in P, and α, β in V^* such that $i = A \rightarrow \alpha\alpha'$ and $j = B \rightarrow \beta\beta'$. By Lemma 6.18, there exist $\nu_1 = x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b} \bullet \genfrac{}{}{0pt}{}{\alpha'}{u'_b})r_i x'_b$ in q_1 and $\nu_2 = y_b d_j(\genfrac{}{}{0pt}{}{\beta}{v_b} \bullet \genfrac{}{}{0pt}{}{\beta'}{v'_b})r_j y'_b$ in q_2 such that

$$d_1 \delta_d d_i \alpha = \hat{x}_b d_i \alpha \text{ and } d_1 \gamma_d d_j \beta = \hat{y}_b d_j \beta. \tag{6.23}$$

Furthermore, the relation (q_1, q_2) mac (q_3, q_2) indicates that $q_1 r_i \vdash q_3$ for some i in P: ν_1 is necessarily of form $x_b d_i(\genfrac{}{}{0pt}{}{\alpha}{u_b} \bullet)r_i x'_b$. Among the various possible positions ν_2 in q_2, there is at least one that verifies $v'_b r_j = y''_d \chi y'''_b$ with y''_d in T_d^*, χ in $T \cup T_r$ and y'''_b in T_b^*, such that the predicate (q_1, i) also enforces $\chi \neq r_i$.

Let us now consider the second part (q_1, q_2) mac (q_3, q_2) ma^* (q_f, q'_f) of Equation 6.22. By Lemma 6.16, there exist two bracketed strings w_b and w'_b

with $w = w'$ such that $q_1 w_b \vDash^* q_f$ and $q_2 w'_b \vDash^* q'_f$. By Lemma 6.19, there exists z_b and z'_b in T_b^* such that

$$r_i \hat{x}'_b \Rightarrow_b^* z_b \text{ and } \beta' r_j \hat{y}'_b \Rightarrow_b^* z'_b, \tag{6.24}$$

with $w \cong z \pmod{\Pi}$ and $w' \cong z' \pmod{\Pi}$. By transitivity of the congruence relation,

$$z \cong z' \pmod{\Pi}. \tag{6.25}$$

We combine the equations (6.23) and (6.24) and obtain the derivations in $\mathcal{G}$

$$S \underset{\text{rm}}{\Longrightarrow}^* \delta A z \overset{i}{\underset{\text{rm}}{\Longrightarrow}} \delta\alpha z \text{ and } S \underset{\text{rm}}{\Longrightarrow}^* \gamma B z'' \underset{\text{rm}}{\Longrightarrow} \gamma\beta\beta' z'' \underset{\text{rm}}{\Longrightarrow}^* \delta\alpha z', \tag{6.26}$$

for some z'' in T^*.

We follow now the classical argument of Aho and Ullman (1972, Theorem 5.9) and study the cases where β' is ε, in T^+, or contains a nonterminal as a factor.

If $\beta' = \varepsilon$, then our equations (6.26) and (6.25) fit the requirements of Equation 6.19. Nevertheless, $v' = \varepsilon$ and $\chi = r_j \neq r_i$ implies $i \neq j$, violating the requirements of Equation 6.20.

If $\beta' = v'$ **is in** T^+, then once again we meet the conditions of Equation 6.19. Nevertheless, in this case, $z' = v'z'' \neq z''$, hence violating the requirements of Equation 6.20.

If there is at least one nonterminal C in β', then

$$S' \underset{\text{rm}}{\Longrightarrow}^* \gamma\beta v_1 C v_3 z'' \underset{\text{rm}}{\Longrightarrow}^* \gamma\beta v_1 v_2 v_3 z'' = \delta\alpha v_1 v_2 v_3 z'' = \delta\alpha z'. \tag{6.27}$$

Then, Equation 6.19 holds (with $k = C{\rightarrow}\rho$ instead of $B{\rightarrow}\beta$). Remember that $r_i \neq \chi$, thus either $v_1 v_2 = \rho = \varepsilon$, $\chi = r_k$ but $k \neq i$ and Equation 6.20 cannot hold, or $v_1 v_2 \neq \varepsilon$, but the last condition of Equation 6.20 would require $v_3 z'' = z'$. □

Strictness Let us consider now the grammar with rules

$$S{\rightarrow}AC \,|\, BCb,\ A{\rightarrow}a,\ B{\rightarrow}a,\ C{\rightarrow}cCb \,|\, cb. \tag{$\mathcal{G}_{32}$}$$

Grammar $\mathcal{G}_{32}$ is not LRR: the right contexts $c^n b^n\$$ and $c^n b^{n+1}\$$ of the reductions using $A{\rightarrow}a$ and $B{\rightarrow}a$ cannot be distinguished by regular covering sets. Nevertheless, our test on $\Gamma_{32}/\mathsf{item}_0$ shows that $\mathcal{G}_{32}$ is not ambiguous.

Similarly, the portion of the ISO C++ grammar described in Section 3.1.3 on page 29 is not LR-Regular, but is proven unambiguous by our test with item_0 approximations.

6.3 Practical Results

We implemented a noncanonical ambiguity detection algorithm in C as a new option in GNU Bison that triggers an ambiguity detection computation instead of the parser generation. The output of this verification on the subset of the Standard ML grammar given in Section 3.1.4 on page 31 is:

```
2 potential ambiguities with LR(0) precision detected:
    (match -> mrule . , match -> match . '|' mrule )
    (match -> match . '|' mrule , match -> match '|' mrule . )
```

From this ambiguity report, two things can be noted: that user-friendliness is not a strong point of the tool in its current form, and that the two detected ambiguities correspond to the two ambiguities of Example 3.6 on page 32 and Example 3.7 on page 45. Furthermore, the reported ambiguities do not mention anything visibly related to the difficult conflict of Example 3.5 on page 32.

6.3.1 Example Test

Our ambiguity checking algorithm attempts to find ambiguities as two different parse trees describing the same sentence. Of course, there is in general an infinite number of parse trees with an infinite number of derived sentences, and we use a position automaton to approximate the possible walks through the trees.

We detail here the algorithm on the relevant portion of our grammar, and consider to this end item_0 approximations: our equivalence classes are LR(0) items. A dot in a grammar production $A{\rightarrow}\alpha{\bullet}\beta$ can also be seen as a position in an elementary tree—a tree of height one—with root A and leaves labeled by $\alpha\beta$. When moving from item to item, we are also moving inside all the syntax trees that contain the corresponding elementary trees. The moves from item to item that we describe in the following can be checked on the trees of Figure 3.4 on page 33 and Figure 3.7 on page 44. The resulting ambiguity detection is very similar to the case study of Section 5.3.5 on page 123.

Since we want to find two different trees, we work with pairs of concurrent items, starting from a pair $(S{\rightarrow}{\bullet}\langle dec\rangle\ ,\ \ S{\rightarrow}{\bullet}\langle dec\rangle\ )$ at the beginning of all trees, and ending on a pair $(S{\rightarrow}\langle dec\rangle\ {\bullet},\ \ S{\rightarrow}\langle dec\rangle\ {\bullet})$. Between these, we pair items that could be reached upon reading a common sentence prefix, hence following trees that derive the same sentence.

Example Run Let us start with the couple of items reported as being in conflict by Bison; just like Bison, our algorithm has found out that the two positions might be reached by reading a common prefix from the beginning of the input:

$$(\langle match\rangle{\rightarrow}\langle match\rangle \bullet \text{ '|' } \langle mrule\rangle,\quad \langle exp\rangle{\rightarrow}\textbf{case } \langle exp\rangle \textbf{ of } \langle match\rangle \bullet) \tag{6.28}$$

Unlike Bison, the algorithm attempts to see whether we can keep reading the same sentence until we reach the end of the input. Since we are at the extreme right of the elementary tree for rule $\langle exp\rangle{\rightarrow}\textbf{case } \langle exp\rangle \textbf{ of } \langle match\rangle$, we are also to the immediate right of the nonterminal $\langle exp\rangle$ in some rule right part. Our algorithm explores all the possibilities, thus yielding the three couples:

$$(\langle match\rangle{\rightarrow}\langle match\rangle \bullet \text{ '|' } \langle mrule\rangle,\quad \langle mrule\rangle{\rightarrow}\langle pat\rangle{=}{>}\langle exp\rangle \bullet) \tag{6.29}$$

$$(\langle match\rangle{\rightarrow}\langle match\rangle \bullet \text{ '|' } \langle mrule\rangle,\quad \langle exp\rangle{\rightarrow}\textbf{case } \langle exp\rangle \bullet \textbf{of } \langle match\rangle) \tag{6.30}$$

$$(\langle match\rangle{\rightarrow}\langle match\rangle \bullet \text{ '|' } \langle mrule\rangle,\quad \langle sfvalbind\rangle{\rightarrow}vid\ \langle atpats\rangle = \langle exp\rangle \bullet) \tag{6.31}$$

Applying the same idea to the pair (6.29), we should explore all the items with the dot to the right of $\langle mrule\rangle$.

$$(\langle match\rangle{\rightarrow}\langle match\rangle \bullet \text{ '|' } \langle mrule\rangle,\quad \langle match\rangle{\rightarrow}\langle mrule\rangle \bullet) \tag{6.32}$$

$$(\langle match\rangle{\rightarrow}\langle match\rangle \bullet \text{ '|' } \langle mrule\rangle,\quad \langle match\rangle{\rightarrow}\langle match\rangle \text{ '|' } \langle mrule\rangle \bullet) \tag{6.33}$$

At this point, we find $\langle match\rangle{\rightarrow}\langle match\rangle \bullet \text{ '|' } \langle mrule\rangle$, our competing item, among the items with the dot to the right of $\langle match\rangle$: from our approximations, the strings we can expect to the right of the items in the pairs (6.32) and (6.33) are the same, and we report the pairs as potential ambiguities.

Our ambiguity detection is not over yet: from (6.31), we could reach successively (showing only the relevant possibilities):

$$(\langle match\rangle{\rightarrow}\langle match\rangle \bullet \text{ '|' } \langle mrule\rangle,\quad \langle fvalbind\rangle{\rightarrow}\langle sfvalbind\rangle \bullet) \tag{6.34}$$

$$(\langle match\rangle{\rightarrow}\langle match\rangle \bullet \text{ '|' } \langle mrule\rangle,\quad \langle fvalbind\rangle{\rightarrow}\langle fvalbind\rangle \bullet \text{ '|' } \langle sfvalbind\rangle) \tag{6.35}$$

In this last pair, the dot is to the left of the same symbol, meaning that the following item pair might also be reached by reading the same string from the beginning of the input:

$$(\langle match\rangle{\rightarrow}\langle match\rangle \text{ '|' } \bullet\langle mrule\rangle,\quad \langle fvalbind\rangle{\rightarrow}\langle fvalbind\rangle \text{ '|' } \bullet\langle sfvalbind\rangle) \tag{6.36}$$

The dot being to the left of a nonterminal symbol, it is also at the beginning of all the right parts of the productions of this symbol, yielding successively:

$$(\langle mrule\rangle{\rightarrow}{\scriptstyle\bullet}\langle pat\rangle{=>}\langle exp\rangle,\ \ \langle fvalbind\rangle{\rightarrow}\langle fvalbind\rangle\ \text{'|'}\ {\scriptstyle\bullet}\langle sfvalbind\rangle) \tag{6.37}$$

$$(\langle mrule\rangle{\rightarrow}{\scriptstyle\bullet}\langle pat\rangle{=>}\langle exp\rangle,\ \ \langle sfvalbind\rangle{\rightarrow}{\scriptstyle\bullet}\, vid\ \langle atpats\rangle = \langle exp\rangle) \tag{6.38}$$

$$(\langle pat\rangle{\rightarrow}{\scriptstyle\bullet}\, vid\ \langle atpat\rangle,\ \ \langle sfvalbind\rangle{\rightarrow}{\scriptstyle\bullet}\, vid\ \langle atpats\rangle = \langle exp\rangle) \tag{6.39}$$

$$(\langle pat\rangle{\rightarrow}vid{\scriptstyle\bullet}\langle atpat\rangle,\ \ \langle sfvalbind\rangle{\rightarrow}vid{\scriptstyle\bullet}\langle atpats\rangle = \langle exp\rangle) \tag{6.40}$$

$$(\langle pat\rangle{\rightarrow}vid{\scriptstyle\bullet}\langle atpat\rangle,\ \ \langle atpats\rangle{\rightarrow}{\scriptstyle\bullet}\langle atpat\rangle) \tag{6.41}$$

$$(\langle pat\rangle{\rightarrow}vid\langle atpat\rangle{\scriptstyle\bullet},\ \ \langle atpats\rangle{\rightarrow}\langle atpat\rangle{\scriptstyle\bullet}) \tag{6.42}$$

$$(\langle mrule\rangle{\rightarrow}\langle pat\rangle{\scriptstyle\bullet}{=>}\langle exp\rangle,\ \ \langle atpats\rangle{\rightarrow}\langle atpat\rangle{\scriptstyle\bullet}) \tag{6.43}$$

$$(\langle mrule\rangle{\rightarrow}\langle pat\rangle{\scriptstyle\bullet}{=>}\langle exp\rangle,\ \ \langle sfvalbind\rangle{\rightarrow}vid\ \langle atpats\rangle{\scriptstyle\bullet} = \langle exp\rangle) \tag{6.44}$$

Our exploration stops with this last item pair: its concurrent items expect different terminal symbols, and thus cannot reach the end of the input upon reading the same string. The algorithm has successfully found how to discriminate the two possibilities in conflict in Example 3.5 on page 32.

6.3.2 Implementation

The example run detailed above relates couples of items. This relation is the mutual accessibility relation ma of Definition 6.13 on page 138, defined as the union of several primitive relations:

mas for terminal and nonterminal shifts, holding for instance between pairs (6.35) and (6.36), but also between (6.41) and (6.42),

mad for downwards closures, holding for instance between pairs (6.36) and (6.37),

mac for upwards closures in case of a conflict, i.e. when one of the items in the pair has its dot to the extreme right of the rule right part and the concurrent item is different from it, holding for instance between pairs (6.29) and (6.32). Formally, our notion of a conflict coincides with that of Aho and Ullman (1972, Theorem 5.9).

The algorithm constructs the image of the initial pair $(S'{\rightarrow}{\scriptstyle\bullet}S,\ S'{\rightarrow}{\scriptstyle\bullet}S)$ by the ma^* relation. If at some point we reach a pair holding twice the same item from a pair with different items, we report an ambiguity. Since this occurs as soon as we find a mac relation that reaches the same item twice, the mar relation and the boolean flag of Definition 6.13 are not needed.

The eligible single moves from item to item are in fact the transitions in the position automaton for item_0. The size of the ma relation is bounded by the square of the size of this PA. Let $|\mathcal{G}|$ denote the size of the context-free grammar $\mathcal{G}$, i.e. the sum of the length of all the rules right parts, and $|P|$ denote the number of rules; then, in the LR(0) case, the algorithm time and space complexity is bounded by $\mathcal{O}((|\mathcal{G}|\ |P|)^2)$.

Implementation Details The experimental tool currently implements the algorithm with LR(0), SLR(1), and LR(1) items. Although the space required by LR(1) item pairs is really large, we need this level of precision in order to guarantee an improvement over the LALR(1) construction. Beyond the NU test, the tool also implements a LR and a LRR test using the same item pairing technique as our NU algorithm:

- the LR test simply computes the image of Q_s^2 by $(\mathsf{mas} \cup \mathsf{mad})^*$ and reports a conflict for each couple in this image related to some couple in Q^2 by mac;
- the LRR test computes the image of Q_s^2 by $(\mathsf{mas} \cup \mathsf{mad})^* \circ \mathsf{mac} \circ \mathsf{rma}^*$ and looks for conflicts in this image.

The implementation changes several details:

- We construct a position automaton using the alternative construction of Section 4.2.1.1 on page 54, where states are either the items of form $A \rightarrow \alpha {\bullet} \beta$, or some nonterminal items of form ${\bullet}A$ or $A{\bullet}$. For instance, a nonterminal item would be used when computing the mutual accessibility of (6.29) and before reaching (6.32):
$$(\langle match\rangle \rightarrow \langle match\rangle {\bullet}\ \text{'|'}\ \langle mrule\rangle,\quad \langle mrule\rangle{\bullet}). \tag{6.45}$$
The size of the PA then becomes bounded by $\mathcal{O}(|\mathcal{G}|)$ in the LR(0) and SLR(1) case, and $\mathcal{O}(|\mathcal{G}||T|^2)$—where $|T|$ is the number of terminal symbols—in the LR(1) case, and the complexity of the algorithm is thus bounded by the square of these numbers.
- We consider the associativity and static precedence directives (Aho et al., 1975) of Bison and thus we do not report resolved ambiguities.
- Although we could use a pure notion of a conflict as in Section 6.2.3.2 and rely on the exploration to correctly employ lookaheads, we implemented a more traditional notion of a conflict that integrates lookaheads. The prime motivation for this is that it makes the implementation of static precedence and associativity rules straightforward.
- We order our items pairs to avoid redundancy in reduce/reduce conflicts. As mentioned in Section 6.2.1 for common prefixes with conflicts, our decomposition is not unique. In a reduce/reduce conflict, we can choose to follow one reduction or the other, and we might find a

Table 6.1: Reported ambiguities several toy grammars.

Grammar	actual class	Bison	HVRU	NU(item_0)
$\mathcal{G}_{25}$	non-LRR	6	0	9
$\mathcal{G}_{29}$	LR(0)	0	1	0
$\mathcal{G}_{30}^n$	LR(2^n)	1	0	0
$\mathcal{G}_{31}^n$	ambiguous	1	1	1
$\mathcal{G}_{32}$	non-LRR	1	1	0

point of ambiguity sooner or later depending on this choice. The same issue was met by McPeak and Necula (2004) with Elkhound, where a strict bottom-up order was enforced using an ordering on the nonterminals and the portion of the input string covered by each reduction.

We solve our issue in a similar fashion, the difference being that we do not have a finite input string at our disposal, and thus we adopt a more conservative ordering using the right corner relation. We say that A dominates B, noted $A \searrow B$, if there is a rule $A \rightarrow \alpha B$; our order is then $\searrow^*$. In a reduce/reduce conflict between reductions to A and B, we follow the reduction of A if $A \not\searrow^* B$ or if both $A \searrow^* B$ and $B \searrow^* A$.

6.3.3 Experimental Results

The choice of a conservative ambiguity detection algorithm is currently rather limited. Several parsing techniques define subsets of the unambiguous grammars, and beyond LR(k) parsing, two major parsing strategies exist: LR-Regular parsing (Čulik and Cohen, 1973), which in practice explores a regular cover of the right context of LR conflicts with a finite automaton (Bermudez and Schimpf, 1990), and noncanonical parsing (Szymanski and Williams, 1976), where the exploration is performed by the parser itself. Since we follow the latter principle with our algorithm, we call it a noncanonical unambiguity (NU) test.

A different approach, unrelated to any parsing method, was proposed by Brabrand et al. (2007) with their horizontal and vertical unambiguity check (HVRU). Horizontal ambiguity appears with overlapping concatenated languages, and vertical ambiguity with non-disjoint unions; their method thus follows exactly how the context-free grammar was formed. Their intended application is to test grammars that describe RNA secondary structures (Reeder et al., 2005).

Table 6.2: Reported potential ambiguities in the RNA grammars discussed by Reeder et al. (2005).

Grammar	actual class	Bison	HVRU	NU(item_1)
$\mathcal{G}_1$	ambiguous	30	6	14
$\mathcal{G}_2$	ambiguous	33	7	13
$\mathcal{G}_3$	non-LRR	4	0	2
$\mathcal{G}_4$	SLR(1)	0	0	0
$\mathcal{G}_5$	SLR(1)	0	0	0
$\mathcal{G}_6$	LALR(1)	0	0	0
$\mathcal{G}_7$	non-LRR	5	0	3
$\mathcal{G}_8$	LALR(1)	0	0	0

6.3.3.1 Toy Grammars

The formal comparisons of our algorithm with various other methods given in Section 6.2.3 are sustained by several small grammars. Table 6.1 compiles the results obtained on these grammars. The “Bison” column provides the total number of conflicts (shift/reduce as well as reduce/reduce) reported by Bison, the “HVRU” column the number of potential ambiguities (horizontal or vertical) reported by the HVRU algorithm, and the “NU(item_0)” column the number of potential ambiguities reported by our algorithm with LR(0) items.

For completeness, we also give the results of our tool on the RNA grammars of Reeder et al. (2005) in Table 6.2.

6.3.3.2 Programming Languages Grammars

We ran the LR, LRR and NU tests on seven different ambiguous grammars for programming languages:

Pascal an ISO-7185 Pascal grammar retrieved from the `comp.compilers` FTP at `ftp://ftp.iecc.com/pub/file/`, LALR(1) except for a dangling else ambiguity,

Mini C a simplified C grammar written by Jacques Farré for a compilers course, LALR(1) except for a dangling else ambiguity,

ANSI C (Kernighan and Ritchie, 1988, Appendix A.13), also retrieved from the `comp.compilers` FTP. The grammar is LALR(1), except for a dangling else ambiguity. The **ANSI C’** grammar is the same grammar modified by setting typedef names to be a nonterminal, with a single production $\langle \textit{typedef-name} \rangle \rightarrow \textit{identifier}$. The modification re-

Table 6.3: Number of initial LR(0) conflicting pairs remaining with the LR, LRR and NU tests employing successively LR(0), SLR(1), LALR(1), and LR(1) precision.

Precision	LR(0)			SLR(1)			LALR(1)	LR(1)		
Test	LR	LRR	NU	LR	LRR	NU	LR	LR	LRR	NU
Pascal	119	55	55	5	5	5	1	1	1	1
Mini C	153	11	10	5	5	4	1	1	1	1
ANSI C	261	13	2	13	13	2	1	1	1	1
ANSI C'	265	117	106	22	22	11	9	9	-	-
Standard ML	306	163	158	130	129	124	109	109	107	107
Small Elsa C++	509	285	239	25	22	22	24	24	-	-
Elsa C++	973	560	560	61	58	58	53	-	-	-

flects the fact that GLR parsers should not rely on the *lexer hack* for disambiguation.

Standard ML, extracted from the language definition (Milner et al., 1997, Appendix B). As mentioned in Section 3.1.2 on page 28, this is a highly ambiguous grammar, and no effort whatsoever was made to ease its implementation with a parser generator.

Elsa C++, developed with the Elkhound GLR parser generator (McPeak and Necula, 2004), and a smaller version without class declarations nor function bodies. Although this is a grammar written for a GLR parser generator, it allows deterministic parsing whenever possible in an attempt to improve performance.

In order to provide a better ground for comparisons between LR, LRR and NU testing, we implemented an option that computes the number of initial LR(0) item pairs in conflict—for instance pair (6.28)—that can reach a point of ambiguity—for instance pair (6.32)—through the ma relation. Table 6.3 presents the number of such initial conflicting pairs with our tests when employing LR(0) items, SLR(1) items, and LR(1) items. We completed our implementation by counting conflicting LR(0) item pairs for the LALR(1) conflicts in the parsing tables generated by Bison, which are shown in the LALR(1) column of Table 6.3.

This measure of the initial LR(0) conflicts is far from perfect. In particular, our Standard ML subset has a single LR(0) conflict that mingles an actual ambiguity with a conflict requiring an unbounded lookahead exploration: the measure would thus show no improvement when using our test. The measure is not comparable with the numbers of potential ambiguities reported by NU; for instance, NU(item_1) would report 89 potential ambiguities for Standard ML, and 52 for ANSI C'.

Although we ran our tests on a machine equipped with a 3.2GHz Xeon and 3GiB of physical memory, several tests employing LR(1) items exhausted the memory. The explosive number of LR(1) items is also responsible for a huge slowdown: for the small Elsa grammar, the NU test with SLR(1) items ran in 0.22 seconds, against more than 2 minutes for the corresponding LR(1) test (and managed to return a better conflict report).

6.3.3.3 Micro-Benchmarks

Basten (2007) compared several means to detect ambiguities in context-free grammars, including our own implementation in GNU Bison, the AMBER generative test (Schröer, 2001), and the MSTA LR(k) parser generator (Makarov, 1999).

Initial Experiments Also confronted with the difficulty of measuring ambiguity in a meaningful way, he opted for a micro-benchmark approach, performing the tests on more than 36 small unambiguous grammars from various sources. These grammars being small, and combining the results returned by the tools, he was able to decide they were unambiguous. The conservative accuracy ratios he obtained with our tool, computed as the number of grammars correctly classified as unambiguous, divided by the number of tested grammars, were of 61%, 69%, and 86% in LR(0), SLR(1), and LR(1) mode respectively. This compares rather well to the LR(k) tests, where the ratio drops to 75%, with attempted k values as high as 50. Interestingly, our LRR test with LR(1) precision chokes on the same grammars as the LR(k) tests, and obtains the same 75% ratio. Furthermore, the grammars on which the NU(item_1) test failed were all of the same mold (1-, 2-, and 4-letters palindromes, and the RNA grammars $\mathcal{G}_3$ and $\mathcal{G}_7$ of Reeder et al. (2005)).

The good results Basten obtained with AMBER on a second set of ambiguous grammars emphasizes the interest for a mixed strategy, where the paths to potential ambiguities in ma^* could be employed to try to generate ambiguous sentential forms. The running time of AMBER on a full programming language grammar is currently rather prohibitive; running a generator on the portions of the grammar that might present an ambiguity according to our tool could improve it drastically. The initial experiments run by Basten in this direction are highly encouraging.

A Larger Collection We gathered a few more unambiguous grammars from programming languages constructs in order to improve the representativity of Basten's grammar collection. A first set of seven unambiguous grammars was found in the comp.compilers archive when querying the word "conflict" and after ruling out ambiguous grammars and LL-related conflicts:[4]

90-10-042 an excerpt of the YACC syntax, which has an optional semicolon as end of rule marker that makes it LR(2);

98-05-030 a non LR excerpt of the Tiger syntax;

98-08-215 a LR(2) grammar;

03-02-124 a LR(2) excerpt of the C# grammar;

03-09-027 a LR(2) grammar;

03-09-081 a LR(3) grammar;

05-03-114 a LR(2) grammar (it seems like its author actually meant to write a non LR grammar but mistyped it).

A second set of nine grammars was compiled using grammars from this thesis and from the literature on LR-Regular and noncanonical parsing techniques:

Ada "is" a LR(2) snippet of the Ada syntax (ANSI, 1983), pointed out by Baker (1981) and Boullier (1984);

Ada calls a non LR fragment of the Ada syntax, pointed out by Boullier (1984);

C++ qualified identifiers a non LR-Regular portion of the C++ syntax (ISO, 1998) that we described in Section 3.1.3 on page 29;

Java modifiers a non LR excerpt of the Java syntax that we described in Section 3.1.1 on page 26, and that was detailed by Gosling et al. (1996) in their Sections 19.1.2 and 19.1.3;

Java names a non LR excerpt given in their Section 19.1.1;

Java arrays a LR(2) excerpt given in their Section 19.1.4;

Java casts a LR(2) excerpt given in their Section 19.1.5;

Pascal typed a LR(2) grammar for Pascal variable declarations that enforces type correctness, given by Tai (1979);

Set expressions a non LR grammar that distinguishes between arithmetic and set expressions, given by Čulik and Cohen (1973).

Table 6.4: Number of conflicts obtained with Bison, Brabrand et al.'s tool, and our tool in LRR and NU mode with various precision settings. The LR(k) column indicates instead the smallest value of k such that the grammar is LR(k) or "non LR" if no such value exists. Improvement rates are computed by only considering non-LALR(1) grammars.

Method	LR		HVRU	LRR	NU		
Precision	LALR(1)	LR(k)		LR(1)	LR(0)	SLR(1)	LR(1)
90-10-042	2	LR(2)	0	14	7	7	6
98-05-030	1	non LR	10	26	0	0	0
98-08-215	1	LR(2)	0	0	0	0	0
03-02-124	1	LR(2)	0	0	0	0	0
03-09-027	2	LR(2)	0	0	0	0	0
03-09-081	2	LR(3)	0	0	0	0	0
05-03-114	1	LR(2)	0	0	0	0	0
Ada "is"	1	LR(2)	0	0	0	0	0
Ada calls	1	non LR	0	0	1	0	0
C++ qualified IDs	1	non LRR	5	21	0	0	0
Java modifiers	31	non LR	0	0	3	0	0
Java names	1	non LR	0	0	0	0	0
Java arrays	1	LR(2)	0	0	0	0	0
Java casts	1	LR(2)	0	0	0	0	0
Pascal typed	1	LR(2)	0	0	0	0	0
Set expressions	8	non LR	19	119	2	2	2
Accuracy/improvement	0%	62%	81%	75%	75%	87%	87%
Overall accuracy	50%	69%	69%	75%	65%	75%	87%
Overall improvement	0%	42%	69%	50%	58%	65%	73%

We ran several conservative ambiguity detection tests[5] on Basten's grammar collection and on our small collection. Table 6.4 shows the results of our micro-benchmarks. Our small collection contains only non-LALR(1) grammars, and as such the accuracy of the various tools can also be seen as an improvement ratio over LALR(1). The overall accuracy and improvement scores take into account the complete collection of 53 grammars using both our grammars and Basten's; 26 grammars are not LALR(1) in this full collection.

The ability to freely specify lookahead lengths in a LR(k) parser improves over LALR(1) parsing, but much less than the other methods that take an unbounded lookahead into account. An interesting point is that the

[4]The names `xx-xx-xxx` are the message identifiers on the archive, and are accessible on the web at the adress `http://compilers.iecc.com/comparch/article/xx-xx-xxx`.

[5]The horizontal and vertical ambiguity check of Brabrand et al. was run *as is*, without manual grammar unfolding, which would have improved its accuracy.

results of our tool in LR(1) precision with Brabrand et al.'s horizontal and vertical ambiguity check are not highly correlated, and a simple conjunction of the two tools would obtain an overall 88% improvement rate, or 94% on our small collection only. Let us finally point out that a much larger grammar collection would be needed in order to obtain more trustworthy micro-benchmark results, and that such results might still not be very significant for large, complex grammars, where the precision of a method seems to be much more important than for our small grammars.

6.3.4 Current Limitations

Our implementation is still a prototype. We describe several planned improvements (Section 6.3.4.1 and Section 6.3.4.2), followed by a small account on the difficulty of considering dynamic disambiguation filters and merge functions in the algorithm (Section 6.3.4.3).

6.3.4.1 Ambiguity Report

As mentioned in the beginning of Section 6.3.1 on page 147, the ambiguity report returned by our tool is hard to interpret.

A first solution, already partially supported by Brabrand et al. (2007), is to attempt to generate actually ambiguous inputs that match the detected ambiguities. The ambiguity report would then comprise of two parts, one for proven ambiguities with examples of input, and one for the potential ambiguities. The generation should only follow item pairs from which the potential ambiguities are reachable through ma relations, and stop whenever finding the ambiguity or after having explored a given number of paths.

Displaying the (potentially) ambiguous paths in the grammar in a graphical form is a second possibility. This feature is implemented by ANTLRWorks, the development environment for the upcoming version 3 of ANTLR (Bovet and Parr, 2007).

6.3.4.2 Running Space

The complexity of our algorithm is a square function of the grammar size. If, instead of item pairs, we considered deterministic states of items like LALR(1) does, the worst-case complexity would rise to an exponential function. Our algorithm is thus more robust.

Nonetheless, practical computations seem likely to be faster with LALR(1) item sets: a study of LALR(1) parsers sizes by Purdom (1974) showed that the size of the LALR(1) parser was usually a linear function of the size of the grammar. Therefore, all hope of analyzing large GLR grammars—like the Cobol grammar recovered by Lämmel and Verhoef (2001)—is not lost.

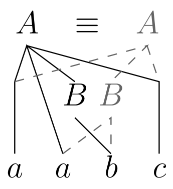


Figure 6.6: The shared parse forest for input *aabc* with grammar $\mathcal{G}_{33}$.

The theory behind noncanonical LALR parsing (Section 5.2 on page 88) translates well into a special case of our algorithm for ambiguity detection, and future versions of the tool should implement it.

6.3.4.3 Dynamic Disambiguation Filters

Our tool does not ignore potential ambiguities when the user has declared a merge function that might solve the issue. The rationale is simple: we do not know whether the merge function will actually solve the ambiguity. Consider for instance the rules

$$A{\rightarrow}aBc\,|\,aaBc,\ B{\rightarrow}ab\,|\,b. \tag{$\mathcal{G}_{33}$}$$

Our tool reports an ambiguity on the item pair $(B{\rightarrow}ab{\bullet},\quad B{\rightarrow}b{\bullet})$, and is quite right: the input *aabc* is ambiguous. As shown in Figure 6.6, adding a merge function on the rules of B would not resolve the ambiguity: the merge function should be written for A.

If we consider arbitrary productions for B, a merge function might be useful only if the languages of the alternatives for B are not disjoint. We could thus improve our tool to detect some useless merge declarations. On the other hand, if the two languages are not equivalent, then there are cases where a merge function is needed on A—or even at a higher level. Ensuring equivalence is difficult, but could be attempted in some decidable cases, namely when we can detect that the languages of the alternatives of B are finite or regular, or using bisimulation equivalence (Caucal, 1990).

Conclusion 7

I know nothing except the fact of my ignorance.
Socrates, by Diogenes Laertius, *Lives of Eminent Philosophers*

The nondeterminism of parsers for programming languages is the prime reason behind the recent momentum of general parsing techniques, where nondeterminism is handled by tabulation methods. Nevertheless, the nondeterminism issue is not eradicated, but appears in subtler ways, with the new possibility of accepting ambiguous grammars.

In this thesis, we have shown two ways to better deal with nondeterminism while keeping the guarantee of unambiguity: the use of noncanonical parsers, through the examples of the Noncanonical LALR(1) and Shift-Resolve constructions, and a conservative ambiguity detection algorithm. In both cases, we relied on a simple approximation model of the paths in the derivation trees of the grammar, allowing a better separation of concerns between the precision of a method, in terms of the objects it manipulates—the states of a position automaton in our framework—, and how it exploits the information carried by these objects.

The two noncanonical parsing techniques that we have studied contribute to the range of practical noncanonical parsing methods, until now mostly reduced to Noncanonical SLR(1) parsing. Our Noncanonical LALR(1) construction is more than just a step from simple lookaheads to contextual ones, as it can be seen as a generic construction of a noncanonical parser from a canonical one. The Shift-Resolve parsing construction further exploits position automata to compute the reduced lookahead lengths it needs for each parsing action, but keeps these lengths finite in order to preserve the linear parsing time complexity. It solves a major drawback of noncanonical parsers, which are either limited to a reduced lookahead window of fixed length, or in contrary not limited at all but with a quadratic parsing time complexity in the worst case.

Our conservative algorithm for ambiguity detection follows some of the principles we designed in noncanonical parser constructions. But we are relieved from the need to actually produce a parser, and thus we do not need to keep track of where conflicts with pending resolutions lie. We exert this freedom to explore in the right context of conflicts as far as needed. Thanks to the generality of the position automata framework, other means for ambiguity detection can be compared formally to our technique, and we prove that it generalizes in particular the LR-Regular tests. The practical experiments conducted so far support the adequacy of the algorithm, and suggest several developments that should be undertaken.

Further Research

We mention here several points related to the work presented in the thesis that seem worthy of further research.

Time Complexity Shift-Resolve parsers enforce a linear time parsing bound by ensuring that pushback lengths remain bounded. Could a linear time bound be enforced for a larger class of grammars by allowing unbounded pushbacks in some cases?

A related question of interest is whether there exists a linear bound on the parsing time for NU($\equiv$) grammars. This holds true for variants of Earley parsing when the grammar is LR(Π) (Leo, 1991), but the proof cannot be translated as such to the noncanonical case.

Ambiguity Detection Many practical improvements to our procedure are mentioned in Section 6.3.4, but we repeat here the two most important ones:

1. switching to a NLALR(1)-like construction in order to avoid the space explosion that was witnessed when using LR(1) items, and

2. generating examples of ambiguous sentences, from which refinements of the position automaton could be constructed on the relevant parts.

Noncanonical Languages Let us call a language $\mathcal{L}(\mathcal{G})$ a *noncanonical language* if there exists an equivalence relation $\equiv$ of finite index on the positions of $\mathcal{G}$ such that $\mathcal{G}$ is NU($\equiv$). Does the class of noncanonical languages properly include the class of, for instance, the Leftmost SLR(1) languages (Tai, 1979)? Or does the grammatical hierarchy collapse in language space?

A point worth investigating is the comparison with Church-Rosser languages (McNaughton et al., 1988), which are characterized by formal machines not so unlike the two stack model we use for noncanonical parsing: shrinking two-pushdown automata (Niemann and Otto, 2005) and restarting automata (Niemann and Otto, 1999). It seems that the class of noncanonical languages would be strictly included in the class of Church-Rosser languages. This would establish that palindrome languages, which are not Church-Rosser (Jurdziński and Loryś, 2007), are indeed beyond the grasp of our ambiguity detection technique.

Regular Approximations The search for good regular approximations of context-free languages is a research domain on its own. The position automaton framework provides a simple unifying theory for the various approximation methods. It is appealing to try to apply it to different formalisms. In this line of work, our initial results on the approximation of XML languages should be completed to embrace the general case of regular tree languages. Another application would be the approximation for formalisms employed in natural language processing, resulting in symbolic *supertaggers* (Boullier, 2003b; Bonfante et al., 2004).

A Formal Definitions

"When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean–neither more nor less."

Lewis Carroll, *Through the Looking-Glass*

A.1 Elements of Formal Language Theory

The reader is as always directed to reference books by Harrison (1978), Hopcroft and Ullman (1979) and Sippu and Soisalon-Soininen (1988) for a more thorough treatment.

A.1.1 Formal Languages

Definition A.1. The *concatenation* of two elements x and y of a set M, denoted by xy, is an associative binary operation. An element ε is the *identity* for this operation if, for all x in V, $\varepsilon x = x\varepsilon = x$. The set M along with the concatenation operation and its identity is a *monoid*. In a monoid M, the n iterated applications of the concatenation operation with the same element x, denoted by x^n, is defined by $x^0 = \varepsilon$ and $x^n = xx^{n-1}$ for $n > 0$.

The concatenation operation can be extended to subsets of M: if A and B are subsets of M, then $AB = \{xy \mid x \in A \text{ and } y \in B\}$. The iterated concatenation is defined accordingly. Then, the closure of a subset A is defined as $A^* = \cup_{n=0}^{\infty} A^n$. If $A^* = M$, then A is a basis of M. The subset A generates M freely if for all x of M there is exactly one sequence $x_1 \cdots x_n$ of elements of A such that $x = x_1 \cdots x_n$; $A^* = M$ is then called a *free monoid*.

Definition A.2. An *alphabet* or *vocabulary* is a finite nonempty set of symbols.

Definition A.3. A *formal language* $\mathcal{L}$ over an alphabet Σ is any subset of the free monoid Σ^*.

A.1.2 Rewriting Systems

Definition A.4. A *rewriting system* is a pair $\langle V, R\rangle$ where

- V is the *alphabet* and
- R a finite set of *rules* in $V^* \times V^*$.

A *generative rewriting system* $\langle V, R, S\rangle$ has a distinguished *starting set* S included in V^*. We denote its rules by $\alpha{\rightarrow}\beta$ for α and β in V^*. The *derivation* relation $\delta\alpha\gamma \overset{i}{\Rightarrow} \delta\beta\gamma$ holds if the rule $i = \alpha{\rightarrow}\beta$ exists in R; we drop the superscript i when irrelevant. We denote the transitive reflexive closure of $\Rightarrow$ by $\overset{\varphi}{\Rightarrow}$ with φ in R^*, or simply by $\Rightarrow^*$ if the sequence of rule applications is irrelevant. The language *generated* by $\langle V, R, S\rangle$ is $\mathcal{L} = \{\beta \mid \exists\alpha \in S, \alpha \Rightarrow^* \beta\}$; more specialized generative rewriting systems will redefine this language.

A *recognitive rewriting system* $\langle V, R, F\rangle$ has a distinguished *final set* F included in V^*. We denote its rules by $\alpha \vdash \beta$ for α and β in V^*. The *rewrite* relation $\delta\alpha\gamma \overset{i}{\models} \delta\beta\gamma$ holds if the rule $i = \alpha \vdash \beta$ exists in R; we drop the superscript i when irrelevant. We denote the transitive reflexive closure of $\models$ by $\overset{\varphi}{\models}$ with φ in R^*, or simply by $\models^*$ if the sequence of rule applications is irrelevant. The language *recognized* by $\langle V, R, F\rangle$ is $\mathcal{L} = \{\alpha \mid \exists\beta \in F, \alpha \models^* \beta\}$; more specialized recognitive rewriting systems will redefine this language.

A *transducer* with output alphabet Σ^* is a pair $\langle M, \tau\rangle$ where M is a recognitive rewriting system $\langle V, R, S\rangle$ and τ a homomorphism from R^* to Σ^*. It produces output $\tau(\varphi)$ for input α if there exists β in F such that $\alpha \overset{\varphi}{\models} \beta$ in M.

A.1.3 Grammars

Definition A.5. A *phrase structure grammar* is a 4-tuple $\langle N, T, P, S\rangle$ where

- N is the *nonterminal alphabet*,
- T the *terminal alphabet*, $T \cap N = \emptyset$,
- $V = N \cup T$ is the *vocabulary*,
- P is a set of *productions* in $V^* \times V^*$, and

- S in N is the *start symbol* or *axiom.*

A phrase structure grammar is a generative rewriting system $\langle V, P, \{S\}\rangle$ and it generates the language $\mathcal{L} = \{w \in T^* \mid S \Rightarrow^* w\}$.

Definition A.6. A *right-linear grammar* is a phrase structure grammar $\mathcal{G} = \langle N, T, P, S\rangle$ where P the set of productions is restricted to $(N \times T^*) \cup (N \times T^* N)$.

A *left-linear grammar* is defined similarly by having P restricted to $(N \times T^*) \cup (N \times NT^*)$.

A *regular grammar* is either a right-linear grammar or a left-linear grammar. A language is *regular language* if it is generated by some regular grammar.

Definition A.7. A *context-free grammar* (CFG) is a phrase structure grammar $\mathcal{G} = \langle N, T, P, S\rangle$ where P the set of productions is restricted to $N \times V^*$. A language is *context-free* if it is generated by some context-free grammar.

A grammar is *reduced* if for all symbols X in V, $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$ for some α and β in V^* and w in T^*. We always consider our grammars to be reduced.

The *augmented grammar* of a context-free grammar $\mathcal{G} = \langle N, T, P, S\rangle$ is the context-free grammar $\mathcal{G}' = \langle N', T', P', S'\rangle$ where

- $N' = N \cup \{S'\}$ with S' a new nonterminal symbol,
- $T' = T \cup \{\$\}$ with $\$$ a new terminal symbol, and
- $P' = P \cup \{S' \rightarrow \$S\$\}$.

A *rightmost derivation* $\delta A x \overset{i}{\underset{\text{rm}}{\Longrightarrow}} \delta \alpha x$ holds if the production $i = A \rightarrow \alpha$ exists in P and x is a terminal string in T^*. Similarly, a *leftmost derivation* $x A \delta \overset{i}{\underset{\text{lm}}{\Longrightarrow}} x \alpha \delta$ holds if the production $i = A \rightarrow \alpha$ exists in P and x is a terminal string in T^*.

A.1.4 Notational Conventions

We present here our notational conventions regarding context-free grammars:

- terminal symbols in T are denoted by small case Latin letters of the beginning of the alphabet a, b, c, d, or by names in small case letters, as in *term*;
- nonterminal symbols in V are denoted by capital Latin letters of the beginning of the alphabet A, B, C, D, or by names in capital letters or between angle brackets, as in *VP* or $\langle$*nonterminal*$\rangle$;

- terminal strings in T^* are denoted by small case Latin letters of the end of the alphabet u, v, w, x, y, z;
- nonterminal strings in V^* are denoted by small case Greek letters of the beginning of the alphabet α, β, γ, δ, and further ρ and σ;
- individual rules are denoted by index letters i, j;
- rule sequences are denoted by φ, ψ, π;
- the length of a string w is denoted by $|w|$;
- the prefix of length k of a string w is denoted by $k : w$, and the suffix by $w : k$; if $|w| < k$, then $k : w = w : k = w$;
- the first set of length k of a sequence α in V^* is $\mathsf{First}_k = \{k : w \mid \alpha \Rightarrow^* w\}$.

A.1.5 Automata

Definition A.8. A *finite state automaton* (FSA) is a 5-tuple $\langle Q, T, R, Q_s, Q_f\rangle$ where

- Q is the (finite) *state set*,
- T is the *input alphabet*,
- R is the (finite) set of *rules* in $QT^* \times Q$,
- $Q_s \subseteq Q$ is the set of *initial states*, and
- $Q_f \subseteq Q$ is the set of *final states*.

In the case where Q and R are not finite, we rather call $\langle Q, T, R, Q_s, Q_f\rangle$ a *labeled transition system* (LTS).

Finite state automata and labeled transition systems are recognitive rewriting systems $\langle Q \cup T, R, Q_f\rangle$ and recognize the language $\mathcal{L} = \{w \in T^* \mid \exists q_s \in Q_s, \exists q_f \in F, q_s w \models^* q_f\}$. In practice, one usually considers rules in $Q(T \cup \{\varepsilon\}) \times Q$.

Definition A.9. A *deterministic finite automaton* (DFA) is a finite-state automaton $\langle Q, T, R, Q_s, Q_f\rangle$ such that $Q_s = \{q_s\}$ is a singleton and R does not contain any two rules $qx \vdash q_1$ and $qy \vdash q_2$ with q_1 different from q_2 and one of x, y a prefix of the other.

Definition A.10. A *pushdown automaton* (PDA) is a 6-tuple $\langle Q, T, R, I, F, \$, \|\rangle$ where

- Q is the set of *stack symbols*,

- T is the *input alphabet*,
- R is the set of *rules* in $(Q^* \cup \$Q^*)\|(T^* \cup T^*\$) \times (Q^* \cup \$Q^*)\|(T^* \cup T^*\$)$,
- I is the set of *initial stack content* in 2^{Q^*},
- F is the set of *final stack contents* in 2^{Q^*},
- $\$$ is the *end marker*, and
- $\|$ is the *stack delimiter*.

A pushdown automaton is a recognitive rewriting system $\langle Q \cup T \cup \{\$, \|\}, R, \$F\|\$\rangle$ and it recognizes the language $\mathcal{L} = \{w \in T^* \mid \exists \gamma_s \in I, \exists \gamma_f \in F, \$\gamma_s\|w\$ \models^* \$\gamma_f\|\$\}$. As with FSAs, the traditional definition employs a finite Q and R sets, but we might want to consider the more general case as well.

Definition A.11. A *deterministic pushdown automaton* (DPDA) is a pushdown automaton $\langle Q, T, R, \{\gamma_s\}, F, \$, \|\rangle$ such that R does not contain any two rules $\alpha\|x \vdash \alpha'\|x'$ and $\beta\|y \vdash \beta'\|y'$ with one of x, y a prefix of the other and one of α, β a suffix of the other.

Definition A.12. The *shift-reduce recognizer* for the context-free grammar $\mathcal{G} = \langle N, T, P, S\rangle$ is a pushdown automaton $\langle V, T, R, \{\varepsilon\}, \{S\}, \$, \|\rangle$ where the set of rules R is the union of the *shift* rules $\{\|a \vdash a\| \mid a \in T\}$ and the *reduce* rules $\{\alpha\| \vdash A\| \mid A \to \alpha \in P\}$.

The *shift-reduce parser* for the context-free grammar $\mathcal{G} = \langle N, T, P, S\rangle$ is the transducer $\langle M, \tau\rangle$ with output alphabet P where M is the shift-reduce recognizer for $\mathcal{G}$ and τ is defined by $\tau(\|a \vdash a\|) = \varepsilon$ and $\tau(\alpha\| \vdash A\|) = A \to \alpha$.

A.1.6 Earley Recognizer

An elegant definition for the Earley recognizer (Earley, 1970) was given among others by Pereira and Warren (1983) and Sikkel (1997) through deduction rules.

An *Earley item* is a triple $(A \to \alpha \bullet \beta, i, j)$, $0 \leq i \leq j \leq n$, meaning that the input $w = a_1 \cdots a_n$ allows a derivation from A to start at a_i, and that $\alpha \Rightarrow^* a_i \cdots a_j$. The deduction steps allow to compute the set (or chart) of all the correct Earley items $\mathcal{I}^{\mathcal{G},w}$ as the deduction closure of the deduction system, using as premises the existence of some other correct Earley items in $\mathcal{I}^{\mathcal{G},w}$, and when some additional side conditions are met.

$$\frac{}{(S \to \bullet \alpha, 0, 0)} \left\{ \; S \to \alpha \in P \right. \tag{Init}$$

$$\frac{(A \to \alpha \bullet B\alpha', i, j)}{(B \to \bullet \beta, j, j)} \left\{ \; B \to \beta \in P \right. \tag{Predict}$$

$$\frac{(A \to \alpha \,\bullet\, b\alpha', i, j)}{(A \to \alpha b \,\bullet\, \alpha', i, j+1)} \left\{ \; a_{j+1} = b \right. \qquad \text{(Scan)}$$

$$\frac{\begin{array}{c}(A \to \alpha \,\bullet\, B\alpha', i, j)\\ (B \to \beta \,\bullet\,, j, k)\end{array}}{(A \to \alpha B \,\bullet\, \alpha', i, k)} \qquad \text{(Complete)}$$

Recognition succeeds if $(S \to \alpha \,\bullet\,, 0, n)$ appears in $\mathcal{I}^{\mathcal{G},w}$ for some rule $S \to \alpha$ of the axiom S.

A.2 LR(k) Parsing

The best resource on the formal aspects of LR(k) parsing is the reference book by Sippu and Soisalon-Soininen (1990).

A.2.1 Valid Items and Prefixes

Definition A.13. A pair $[A \to \alpha \,\bullet\, \beta, y]$ is a *valid LR(k) item* for string γ in V'^* if

$$S' \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \delta A z \underset{\text{rm}}{\Longrightarrow} \delta\alpha\beta z = \gamma\beta z \text{ and } k : z = y.$$

If such a derivation holds in $\mathcal{G}$, then γ in V'^* is a *valid prefix*.

The set of valid items for a given string γ in V'^* is denoted by $\mathsf{Valid}_k(\gamma)$. Two strings δ and γ are equivalent if and only if they have the same valid items.

The valid item sets are obtained through the following computations:

$$\mathsf{Kernel}_k(\varepsilon) = \{[S' \to \,\bullet\, S, \$]\}, \tag{A.1}$$

$$\mathsf{Kernel}_k(\gamma X) = \{[A \to \alpha X \,\bullet\, \beta, y] \mid [A \to \alpha \,\bullet\, X\beta, y] \in \mathsf{Valid}_k(\gamma)\}, \tag{A.2}$$

$$\begin{aligned}\mathsf{Valid}_k(\gamma) &= \mathsf{Kernel}_k(\gamma)\\ &\quad \cup \{[B \to \,\bullet\, \omega, x] \mid [A \to \alpha \,\bullet\, B\beta, y] \in \mathsf{Valid}_k(\gamma) \text{ and } x \in \mathsf{First}_k(\beta y)\}.\end{aligned} \tag{A.3}$$

A.2.2 LR Automata

LR automata are pushdown automata that use equivalence classes on valid prefixes as their stack alphabet Q. We therefore denote explicitly states of a LR parser as $q = [\delta]$, where δ is some valid prefix in q the state reached upon reading this prefix.

Let $M = (Q \cup T \cup \{\$, \|\}, R)$ be a rewriting system where \$ and $\|$ (the end marker and the stack top, respectively) are not in Q nor in T (the set of states and the input alphabet, respectively). A *configuration* of M is a string of the form

$$[\varepsilon][X_1]\dots[X_1\dots X_n]\|x\$ \tag{A.4}$$

where $X_1 \dots X_n$ is a string in V^* and x a string in T^*.

Definition A.14. We say that M is a *LR(k) automaton* for grammar $\mathcal{G}$ if its initial configuration is $[\varepsilon]\|w\$$ with w the input string in T^*, its final configuration is $[\varepsilon][S]\|\$$, and if each rewriting rule in R is of one of the forms

- *shift* a in state $[\delta]$
$$[\delta]\|ax \vdash_{\text{shift}} [\delta][\delta a]\|x,$$
defined if there is an item of form $[A\to\alpha\bullet a\beta, y]$ in $\mathsf{Valid}_k(\delta)$ with $k : ax \in \mathsf{First}_k(a\beta y)$, and if $[\delta a] \neq \emptyset$, or
- *reduce* by rule $A\to X_1 \dots X_n$ of P in state $[\delta X_1 \dots X_n]$
$$[\delta X_1]\dots[\delta X_1\dots X_n]\|x \vdash_{A\to X_1\dots X_n} [\delta A]\|x,$$
defined if $[A\to X_1\dots X_n\bullet, k : x]$ is in $\mathsf{Valid}_k(\delta X_1\dots X_n)$, and if $[\delta A] \neq \emptyset$.

A.2.3 LALR Automata

In case of inadequate states in the LR(0) automaton, LALR(1) lookahead sets are computed for each reduction in conflict in hope of yielding a deterministic LALR(1) automaton. The *LALR(1) lookahead set* of a reduction using $A\to\alpha$ in state q is

$$\text{LA}(q, A\to\alpha) = \{1 : z \mid S' \underset{\text{rm}}{\Longrightarrow}^* \delta A z \text{ and } q = [\delta\alpha]\}. \tag{A.5}$$

The rewriting system M is a *LALR(1) automaton* for grammar $\mathcal{G}$ if it is a LR(0) automaton with another possible form of rules

- *reduce* by rule $A\to X_1 \dots X_n$ of P in state $[\delta X_1 \dots X_n]$ with lookahead a
$$[\delta X_1]\dots[\delta X_1\dots X_n]\|a \vdash_{A\to X_1\dots X_n} [\delta A]\|a, \tag{A.6}$$
defined if $A\to X_1\dots X_n\bullet$ is in $\mathsf{Valid}_0(\delta X_1\dots X_n)$, and lookahead a is in $\text{LA}([\delta X_1\dots X_n], A\to X_1\dots X_n)$.

Synthèse en français

Motivation

Les grammaires sont omniprésentes lors de tout développement logiciel comportant une spécification syntaxique. À partir d'une grammaire algébrique, des composants logiciels comme des analyseurs syntaxiques, des traducteurs de programmes ou encore des enjoliveurs peuvent être générés automatiquement plutôt que programmés manuellement, avec une qualité et une lisibilité bien supérieures. Le domaine de l'ingénierie des grammaires et des logiciels les utilisant, baptisé *grammarware* par Klint et al. (2005), bénéficie à la fois d'une littérature abondante et de nombreux outils pour automatiser les tâches de programmation.

En dépit des nombreux services qu'ils ont pu rendre au cours des années, les générateurs d'analyseurs syntaxiques classiques de la lignée de YACC (Johnson, 1975) ont vu leur suprématie contestée par plusieurs auteurs, qui ont relevé leur inadéquation avec les problèmes actuels de l'ingénierie des grammaires (Parr et Quong, 1996; van den Brand et al., 1998; Aycock, 2001; Blasband, 2001). Par la suite, les praticiens ont pris en considération de nouvelles techniques d'analyse syntaxique, et l'analyse LR Généralisée (GLR) de Tomita, qui visait initialement le traitement des langues naturelles, a trouvé de nombreux adeptes. Grâce à l'utilisation de générateurs d'analyseurs syntaxiques généralisés, comme SDF (Heering et al., 1989), Elkhound (McPeak et Necula, 2004) ou GNU Bison (Donnely et Stallman, 2006), on pourrait penser que les luttes contre les décomptes de conflits tels que

```
grammar.y: conflicts: 223 shift/reduce, 35 reduce/reduce
```

appartiennent maintenant au passé. Les analyseurs syntaxiques généralisés simulent les différents choix non-déterministes en parallèle avec des bonnes performances, et fournissent toutes les analyses valides du texte d'entrée.

L'exposé un peu naïf de la situation donné ci-dessus oublie le fait que toutes les analyses valides selon la grammaire ne le sont pas toujours dans le langage cible. Dans le cas des langages informatiques en particulier, un programme est supposé n'avoir qu'une unique interprétation, et donc une seule analyse syntaxique devrait être obtenue. Néanmoins, la grammaire mise au point pour décrire le langage est souvent ambiguë : en effet, les grammaires ambiguës sont plus concises et plus lisibles (Aho et al., 1975). De ce fait, la définition d'un langage devrait inclure des *règles de désambiguation* pour choisir quelle analyse est correcte (Klint et Visser, 1994). Mais l'on ne peut pas toujours décider quand ces règles sont nécessaires (Cantor, 1962; Chomsky et Schützenberger, 1963; Floyd, 1962a).

McPeak et Necula (2004) ont reconnu les difficultés posées par les ambiguïtés syntaxiques, et ils leur consacrent quelques lignes :

> [Il y a] le risque que les ambiguïtés finissent par être plus difficiles à éliminer que les conflits. Après tout, les conflits sont au moins décidables. Est-ce que l'attrait d'un développement rapide pourrait nous piéger dans des fourrés d'épines sans limites décidables ?
>
> Fort heureusement, les ambiguïtés n'ont pas posé problème. En signalant les conflits rencontrés, Elkhound donne des indices sur les emplacements possibles des ambiguïtés, ce qui est utile pendant le développement initial de la grammaire. Puis, alors que la grammaire mûrit, nous trouvons des ambiguïtés en testant des entrées, et nous les comprenons en imprimant les forêts d'analyse (une option offerte par Elkhound).

Effectivement, beaucoup de conflits sont typiquement causés par des ambiguïtés, et en procédant à un examen attentif de ces nombreux cas de conflit et à des tests intensifs, un utilisateur expérimenté devrait pouvoir déterminer si un règle de désambiguation est nécessaire ou non. Mais aussi longtemps qu'il reste un seul conflit, il n'y a aucune garantie formelle que toutes les ambiguïtés ont bien été traitées. Pour citer Dijkstra (1972),

> Tester des programmes peut se révéler très efficace pour révéler des problèmes, mais est irrémédiablement inadéquat pour démontrer leur absence. Le seul moyen d'améliorer la confiance que l'on peut placer dans un programme est de fournir une preuve convaincante de sa validité.

Dans le cas d'une confrontation avec un fourré d'épines, nous recommandons de porter des gants. Nous les proposons sous deux formes dans cette thèse.

1. La première est la génération d'analyseurs *non canoniques*, moins sujets au non déterminisme, pour lesquels le long processus d'élimination des conflits est moins difficile. On peut ainsi plus raisonnablement espérer éliminer la totalité des conflits, et obtenir ainsi la garantie qu'aucune ambiguïté ne peut subsister. Les analyseurs non canoniques

s'appuient sur la totalité du texte restant à traiter pour décider entre plusieurs actions d'analyse. Ils sont capables d'analyser une partie de ce contexte droit, avant de revenir aux points de conflit et d'utiliser les informations ainsi obtenues pour les résoudre.

2. La seconde est la détection d'ambiguïtés en tant que telle. Ce problème étant indécidable, des approximations, sous la forme de faux positifs ou de faux négatifs, sont inévitables. Notre méthode est prudente dans le sens qu'elle ne peut pas retourner de faux positifs, et il est donc sans danger d'employer une grammaire qu'elle reconnaît comme non ambiguë.

Contenu de la thèse

Après un exposé des bases essentielles de l'analyse syntaxique (Chapitre 2), nous décrivons dans la Section 3.1 quatre cas pratiques issus des syntaxes de Java (Gosling et al., 1996), C++ (ISO, 1998) et Standard ML (Milner et al., 1997) qui illustrent les limites de l'analyse traditionnelle LALR(1). Différentes approches permettant de traiter ces problèmes sont présentées dans le reste du Chapitre 3, fournissant ainsi un aperçu des développements « récents » des méthodes d'analyse syntaxique. Les trois chapitres techniques de cette thèse, que nous présentons de manière plus approfondie par la suite, sont dédiés à l'approximation de grammaires algébriques (Chapitre 4), la génération d'analyseurs syntaxiques (Chapitre 5), et la détection d'ambiguïtés (Chapitre 6). Nous concluons la thèse par quelques remarques et pistes de recherche dans les deux prochaines sections, et nous rappelons les définitions et conventions de notations usuelles dans l'Appendice A.

Automates de positions La génération d'analyseurs syntaxiques non canoniques et la détection d'ambiguïtés peuvent toutes deux être vues comme le résultat d'une analyse statique de la grammaire. Guidés par l'intuition que la plupart des techniques d'analyse syntaxique effectuent à un parcours en profondeur de gauche à droite dans l'ensemble de tous les arbres de dérivation de la grammaire, nous définissons le *graphe de positions* d'une grammaire comme l'ensemble de ces parcours, et un *automate de positions* comme le quotient d'un graphe de positions par une relation d'équivalence entre positions des arbres de dérivation (Chapitre 4). Nous obtenons des niveaux d'approximation arbitraires en choisissant des relations d'équivalence plus fines ou plus grossières : les automates de positions fournissent un cadre général pour approximer des grammaires, cadre dans lequel plusieurs techniques classiques de génération d'analyseurs syntaxiques peuvent être exprimées. En particulier, les états d'un automate de positions généralisent les

items habituellement employés pour indiquer des positions dans une grammaire.
Outre la génération d'analyseurs syntaxiques non canoniques et la détection d'ambiguïtés, nous appliquons les automates de positions à deux problèmes :

1. La reconnaissance d'arbres de dérivations par le biais d'un automate à états finis (Section 4.3), un problème inspiré par le traitement de flux XML (Segoufin et Vianu, 2002). Ce problème est assez révélateur des possibilités des automates de positions, et le résultat de nos recherches initiales est une caractérisation, en termes d'automates de positions, de la famille de grammaires algébriques qui peuvent voir leurs arbres de dérivation reconnus au moyen d'un automate à états finis.
2. La génération d'analyseurs syntaxiques ascendants canoniques (Section 5.1). Dans ce domaine nous montrons simplement comment générer des analyseurs pour les grammaires LR(k) (Knuth, 1965) et strictement déterministes (Harrison et Havel, 1973) à partir d'automates de positions.

Analyse syntaxique non canonique Nous présentons deux méthodes d'analyse non canoniques différentes, à savoir l'analyse LALR(1) non canonique en Section 5.2 et l'analyse par Décalage-Résolution en Section 5.3.

LALR(1) non canonique Avec l'analyse NLALR(1), nous explorons comment obtenir un analyseur non canonique à partir d'un analyseur ascendant canonique. Les analyseurs générés généralisent l'unique autre méthode d'analyse non canonique praticable, à savoir l'analyse SLR(1) non canonique (Tai, 1979).

Décalage-Résolution Le principal défaut des techniques d'analyse non canoniques est la limite qu'elles placent sur la longueur de leur fenêtre d'exploration. Avec l'analyse par Décalage-Résolution, développée en collaboration avec José Fortes Gálvez et Jacques Farré, nous exploitons le graphe de positions directement, et générons des analyseurs où les longueurs des fenêtres d'exploration sont indépendantes et calculées selon les besoins de chaque action d'analyse.

Détection d'ambiguïtés Au long du Chapitre 6, nous présentons un algorithme prudent pour la détection d'ambiguïtés dans les grammaires algébriques. L'algorithme fonctionne sur n'importe quel automate de positions, permettant ainsi des degrés de précision variés lors de la détection en fonction de l'équivalence de positions choisie. Nous comparons formellement notre procédure aux autres moyens de détecter des ambiguïtés, à savoir la génération de phrases de longueur bornée (Gorn, 1963; Cheung et Uzgalis, 1995; Schröer, 2001; Jampana, 2005) et les tests LR-Réguliers

(Čulik et Cohen, 1973; Heilbrunner, 1983), et nous rendons compte des résultats expérimentaux que nous avons obtenus avec notre implémentation de l'algorithme sous la forme d'une option dans GNU Bison.

Conclusion

La présence de choix non déterministes dans les analyseurs syntaxiques pour les langages de programmation est la première raison de l'engouement récent pour les techniques d'analyse généralisée, dans lesquelles le non déterminisme est traité par des méthodes tabulées. Néanmoins, le problème du non déterminisme n'est pas éradiqué, mais réapparaît plus subtilement avec la nouvelle possibilité d'accepter des grammaires ambiguës.

Nous avons montré dans cette thèse deux moyens de mieux traiter le non déterminisme tout en préservant la garantie d'une syntaxe non ambiguë : l'emploi d'analyseurs syntaxiques non canoniques, au travers des constructions des analyseurs LALR(1) non canoniques et à Décalage-Résolution, et un algorithme prudent pour la détection d'ambiguïtés. Dans les deux cas, nous nous sommes appuyés sur un modèle simple d'approximation des chemins dans les arbres de dérivation de la grammaire, permettant une meilleure séparation des préoccupations entre la précision d'une méthode en termes d'objets manipulés (les états d'un automate de positions dans notre cadre théorique), et la manière qu'a la méthode d'exploiter l'information portée par ces objets.

Les deux techniques d'analyse syntaxique non canonique que nous avons étudiées contribuent au choix de méthodes d'analyse non canoniques praticables, choix qui était jusqu'à présent réduit à la seule analyse SLR(1) non canonique. Notre construction LALR(1) non canonique est plus qu'une simple évolution de fenêtres d'analyses simples vers des fenêtres contextuelles, mais peut être vue comme la construction générique d'un analyseur non canonique à partir d'un analyseur canonique. La construction d'analyseurs à Décalage-Résolution exploite de manière approfondie les automates de positions pour calculer les longueurs de fenêtres nécessaires pour chaque action d'analyse, mais garde ces longueurs finies afin de préserver une complexité de traitement en temps linéaire. Cette méthode résout un défaut majeur des analyseurs non canoniques, qui soit sont limités à des fenêtres de symboles réduits d'une longueur fixée à l'avance, soit au contraire ne sont pas limités du tout mais peuvent alors travailler en temps quadratique dans le pire des cas.

Notre algorithme prudent pour la détection d'ambiguïtés applique certains principes mis au point pour l'analyse non canonique. Nous sommes cependant libérés du besoin de construire un analyseur, et de ce fait nous pouvons appliquer une exploration non limitée sans avoir à nous souvenir des emplacements des conflits. Grâce à la flexibilité offerte par les automates de positions, les autres moyens de détecter les ambiguïtés peuvent être comparés formellement à notre technique, et nous avons montré en particulier qu'elle généralise les tests LR-Réguliers. Les expérimentations menées jusqu'ici confirment l'intérêt de l'algorithme, et suggèrent plusieurs améliorations.

Pistes de recherche

Nous mentionnons ici plusieurs points, liés au travail présenté dans la thèse, qui semblent mériter une recherche approfondie.

Complexité en temps Les analyseurs par Décalage-Résolution garantissent un traitement en temps linéaire en assurant que les longueurs de résolution restent bornées. Est-ce qu'on pourrait préserver le temps linéaire pour une plus grande classe de grammaires en acceptant des résolutions de longueurs non bornées dans certains cas ?

Une question assez proche est de déterminer s'il existe une méthode d'analyse syntaxique pour les grammaires NU($\equiv$) qui garantisse un traitement en temps linéaire. C'est le cas pour une analyse basée sur l'analyse d'Earley quand la grammaire est LR(Π) (Leo, 1991), mais la démonstration ne tient pas dans le cas non canonique.

Détection d'ambiguïtés Plusieurs améliorations pratiques de notre procédure sont mentionnées dans la Section 6.3.4, mais nous répétons ici les deux plus importantes :

1. passer à une construction de style NLALR(1) afin d'éviter l'explosion du nombre d'états constatée avec les items LR(1), et
2. générer des exemples de phrases ambiguës, à partir desquelles des raffinements de l'automate de positions pourraient être construits sur les portions concernées par les ambiguïtés.

Langages non canoniques Définissons un langage $\mathcal{L}(\mathcal{G})$ comme *non canonique* s'il existe une relation d'équivalence $\equiv$ d'index fini sur les positions de $\mathcal{G}$ telle que $\mathcal{G}$ soit NU($\equiv$). Est-ce que la classe des langages non canoniques inclut proprement la classe, par exemple, des langages SLR(1) réduits (Tai, 1979) ? Ou bien est-ce que la hiérarchie de classes de grammaires se réduit à une seule classe de langages ?

Un point méritant une investigation est la comparaison avec les langages Church-Rosser (McNaughton et al., 1988), qui sont caractérisés par des machines formelles déterministes assez semblables au modèles à deux piles que nous utilisons pour l'analyse non canonique : les automates mincissants à deux piles (Niemann et Otto, 2005) et les automates à redémarrage (Niemann et Otto, 1999). Il semble que la classe des langages non canoniques soit strictement incluse dans la classe des langages Church-Rosser. Ceci permettrait d'établir que les langages palindromes, qui ne sont pas Church-Rosser (Jurdziński et Loryś, 2007), sont effectivement hors de portée de notre technique de détection d'ambiguïtés.

Approximations rationnelles La recherche de bonnes approximations rationnelles est un domaine qui a sa propre importance. Le cadre théorique des automates de position offre une abstraction simple pour les méthodes de sur-approximation. L'appliquer à d'autres formalismes que les grammaires algébriques est intéressant. Dans ce thème de recherche, nos résultats préliminaires sur l'approximation des langages XML devraient être approfondis pour porter sur le cas général des langages d'arbres rationnels. Une autre application possible serait d'approximer les formalismes employés en traitement automatique des langues naturelles, ce qui permettrait de produire des super étiqueteurs symboliques (Boullier, 2003b; Bonfante et al., 2004).

Bibliography

Aasa, A., 1995. Precedences in specifications and implementations of programming languages. *Theoretical Computer Science*, 142(1):3–26. doi: 10.1016/0304-3975(95)90680-J. Cited on page 23.

AeroSpace and Defence Industries Association of Europe, 2005. *ASD Simplified Technical English.* Specification ASD-STE100. Cited on page 125.

Aho, A.V. and Ullman, J.D., 1971. Translations on a context-free grammar. *Information and Control*, 19(5):439–475. doi: 10.1016/S0019-9958(71)90706-6. Cited on page 67.

Aho, A.V. and Ullman, J.D., 1972. *The Theory of Parsing, Translation, and Compiling. Volume I: Parsing.* Series in Automatic Computation. Prentice Hall. ISBN 0-13-914556-7. Cited on pages 13, 20, 21, 90, 117, 138, 146, 149.

Aho, A.V., Johnson, S.C., and Ullman, J.D., 1975. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(8):441–452. doi: 10.1145/360933.360969. Cited on pages 2, 21, 150, 172.

Alur, R. and Madhusudan, P., 2006. Adding nesting structure to words. In Dang, Z. and Ibarra, O.H., editors, *DLT'06, 10th International Conference on Developments in Language Theory*, volume 4036 of *Lecture Notes in Computer Science*, pages 1–13. Springer. ISBN 3-540-35428-X. doi: 10.1007/11779148_1. Cited on page 67.

Alur, R., 2007. Marrying words and trees. In *PODS'07, 26th ACM Symposium on Principles of Database Systems*, pages 233–242. ACM Press. ISBN 978-1-59593-685-1. doi: 10.1145/1265530.1265564. Cited on page 67.

Anderson, T., 1972. *Syntactic Analysis of LR(k) Languages.* PhD thesis, Department of Computing Science, University of Newcastle upon Tyne. Cited on page 88.

ANSI, 1983. *Reference Manual for the Ada Programming Language ANSI/MIL-STD-1815A-1983*. Springer. URL `http://www.adahome.com/Resources/refs/83.html`. Cited on page 155.

Aycock, J., 2001. Why bison is becoming extinct. *ACM Crossroads*, 7(5): 3–3. doi: 10.1145/969637.969640. Cited on pages 1, 171.

Aycock, J. and Horspool, R.N., 2001. Schrödinger's token. *Software: Practice & Experience*, 31(8):803–814. doi: 10.1002/spe.390. Cited on page 40.

Backus, J.W., 1959. The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM Conference. In *IFIP Congress*, pages 125–131. Cited on page 8.

Bailes, P.A. and Chorvat, T., 1993. Facet grammars: Towards static semantic analysis by context-free parsing. *Computer Languages*, 18(4):251–271. doi: 10.1016/0096-0551(93)90019-W. Cited on page 15.

Baker, T.P., 1981. Extending lookahead for LR parsers. *Journal of Computer and System Sciences*, 22(2):243–259. doi: 10.1016/0022-0000(81)90030-1. Cited on pages 36, 143, 155.

Basten, H.J.S., 2007. Ambiguity detection methods for context-free grammars. Master's thesis, Centrum voor Wiskunde en Informatica, Universiteit van Amsterdam. Cited on pages 154, 156.

Bermudez, M.E. and Logothetis, G., 1989. Simple computation of LALR(1) lookahead sets. *Information Processing Letters*, 31(5):233–238. doi: 10.1016/0020-0190(89)90079-3. Cited on pages 111, 113.

Bermudez, M.E. and Schimpf, K.M., 1990. Practical arbitrary lookahead LR parsing. *Journal of Computer and System Sciences*, 41(2):230–250. doi: 10.1016/0022-0000(90)90037-L. Cited on pages 36, 111, 117, 143, 151.

Bermudez, M.E., 1991. A unifying model for lookahead LR parsing. *Computer Languages*, 16(2):167–178. doi: 10.1016/0096-0551(91)90005-T. Cited on page 108.

Bertsch, E. and Nederhof, M.J., 2007. Some observations on LR-like parsing with delayed reduction. *Information Processing Letters*, 104(6):195–199. doi: 10.1016/j.ipl.2007.07.003. Cited on pages 12, 112.

Billot, S. and Lang, B., 1989. The structure of shared forests in ambiguous parsing. In *ACL'89, 27th Annual Meeting of the Association for Computational Linguistics*, pages 143–151. ACL Press. doi: 10.3115/981623.981641. Cited on page 44.

Birkhoff, G., 1940. *Lattice Theory*, volume 25 of *American Mathematical Society Colloquium Publications*. AMS Press, first edition. Cited on page 62.

Birman, A. and Ullman, J.D., 1973. Parsing algorithms with backtrack. *Information and Control*, 23(1):1–34. doi: 10.1016/S0019-9958(73)90851-6. Cited on pages 15, 39.

Blasband, D., 2001. Parsing in a hostile world. In *WCRE'01, 8th Working Conference on Reverse Engineering*, pages 291–300. IEEE Computer Society. doi: 10.1109/WCRE.2001.957834. Cited on pages 1, 171.

Bojańczyk, M. and Colcombet, T., 2005. Tree-walking automata do not recognize all regular languages. In *STOC'05, thirty-seventh Symposium on Theory of Computing*, pages 234–243. ACM Press. ISBN 1-58113-960-8. doi: 10.1145/1060590.1060626. Cited on page 67.

Bonfante, G., Guillaume, B., and Perrier, G., 2004. Polarization and abstraction of grammatical formalisms as methods for lexical disambiguation. In *COLING'04, 20th International Conference on Computational Linguistics*, pages 303–309. ACL Press. doi: 10.3115/1220355.1220399. Cited on pages 161, 177.

Boullier, P., 1984. *Contribution à la construction automatique d'analyseurs lexicographiques et syntaxiques*. Thèse d'État, Université d'Orléans. Cited on pages 36, 108, 117, 143, 155.

Boullier, P., 2003a. Guided Earley parsing. In *IWPT'03, 8th International Workshop on Parsing Technologies*, pages 43–54. URL `ftp://ftp.inria.fr/INRIA/Projects/Atoll/Pierre.Boullier/earley_final.pdf`. Cited on page 78.

Boullier, P., 2003b. Supertagging: A non-statistical parsing-based approach. In *IWPT'03, 8th International Workshop on Parsing Technologies*, pages 55–65. URL `ftp://ftp.inria.fr/INRIA/Projects/Atoll/Pierre.Boullier/supertaggeur_final.pdf`. Cited on pages 161, 177.

Bovet, J. and Parr, T.J., 2007. ANTLRWorks: An ANTLR grammar development environment. *Software: Practice & Experience*. To appear. Cited on pages 132, 157.

Brabrand, C., Giegerich, R., and Møller, A., 2007. Analyzing ambiguity of context-free grammars. In Holub, J. and Žďárek, J., editors, *CIAA'07, 12th International Conference on Implementation and Application of Automata*, volume 4783 of *Lecture Notes in Computer Science*, pages 214–225. Springer. ISBN 978-3-540-76335-2. doi: 10.1007/978-3-540-76336-9_21. Cited on pages 78, 125, 126, 131, 132, 142, 151, 156, 157.

Breuer, P.T. and Bowen, J.P., 1995. A PREttier Compiler-Compiler: generating higher-order parsers in C. *Software: Practice & Experience*, 25(11): 1263–1297. doi: 10.1002/spe.4380251106. Cited on page 39.

Brooker, R.A. and Morris, D., 1960. An assembly program for a phrase structure language. *The Computer Journal*, 3(3):168–174. doi: 10.1093/comjnl/3.3.168. Cited on page 39.

Cantor, D.G., 1962. On the ambiguity problem of Backus systems. *Journal of the ACM*, 9(4):477–479. doi: 10.1145/321138.321145. Cited on pages 2, 9, 15, 21, 125, 172.

Caucal, D., 1990. Graphes canoniques de graphes algébriques. *RAIRO - Theoretical Informatics and Applications*, 24(4):339–352. URL `http://www.inria.fr/rrrt/rr-0872.html`. Cited on page 158.

Charles, P., 1991. *A Practical method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery.* PhD thesis, New York University. URL `http://jikes.sourceforge.net/documents/thesis.pdf`. Cited on page 24.

Cheung, B.S.N. and Uzgalis, R.C., 1995. Ambiguity in context-free grammars. In *SAC'95*, *Symposium on Applied Computing*, pages 272–276. ACM Press. ISBN 0-89791-658-1. doi: 10.1145/315891.315991. Cited on pages 4, 125, 126, 134, 174.

Chomsky, N., 1956. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124. doi: 10.1109/TIT.1956.1056813. Cited on pages 6, 47.

Chomsky, N., 1959. On certain formal properties of grammars. *Information and Control*, 2(2):137–167. doi: 10.1016/S0019-9958(59)90362-6. Cited on page 7.

Chomsky, N., 1961. On the notion "rule of grammar". In Jakobson, R., editor, *Structure of Language and its Mathematical Aspects*, volume XII of *Proceedings of Symposia in Applied Mathematics*, pages 6–24. AMS. ISBN 0-8218-1312-9. Cited on page 48.

Chomsky, N., 1962. Context-free grammars and pushdown storage. Quarterly Progress Report 65, Research Laboratory of Electronics, M.I.T. Cited on page 10.

Chomsky, N. and Schützenberger, M.P., 1963. The algebraic theory of context-free languages. In Braffort, P. and Hirshberg, D., editors, *Computer Programming and Formal Systems*, Studies in Logic, pages 118–161. North-Holland Publishing. Cited on pages 2, 9, 15, 125, 172.

Cocke, J. and Schwartz, J.T., 1970. *Programming languages and their compilers.* Courant Institute of Mathematical Sciences, New York University. Cited on page 39.

Colmerauer, A., 1970. Total precedence relations. *Journal of the ACM*, 17 (1):14–30. doi: 10.1145/321556.321559. Cited on page 37.

Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M., 2007. *Tree Automata Techniques and Applications.* URL `http://www.grappa.univ-lille3.fr/tata`. Cited on page 66.

Conway, M.E., 1963. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408. doi: 10.1145/366663.366704. Cited on page 76.

Cousot, P. and Cousot, R., 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77, 4th Annual Symposium on Principles of Programming Languages*, pages 238–252. ACM Press. doi: 10.1145/512950.512973. Cited on page 78.

Cousot, P. and Cousot, R., 2003. Parsing as abstract interpretation of grammar semantics. *Theoretical Computer Science*, 290(1):531–544. doi: 10.1016/S0304-3975(02)00034-8. Cited on page 78.

Cousot, P. and Cousot, R., 2007. Grammar analysis and parsing by abstract interpretation. In Reps, T., Sagiv, M., and Bauer, J., editors, *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *Lecture Notes in Computer Science*, pages 178–203. Springer. ISBN 978-3-540-71315-9. doi: 10.1007/978-3-540-71322-7_9. Cited on page 78.

Čulik, K., 1968. Contribution to deterministic top-down analysis of context-free languages. *Kybernetika*, 4(5):422–431. Cited on page 35.

Čulik, K. and Cohen, R., 1973. LR-Regular grammars—an extension of LR(k) grammars. *Journal of Computer and System Sciences*, 7(1):66–96. doi: 10.1016/S0022-0000(73)80050-9. Cited on pages 4, 35, 78, 111, 117, 126, 143, 151, 155, 175.

de Guzman, J., 2003. *Spirit User's Guide.* URL `http://spirit.sourceforge.net/`. Cited on pages 18, 39.

DeRemer, F. and Pennello, T., 1982. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Sys-*

tems, 4(4):615–649. doi: 10.1145/69622.357187. Cited on pages 82, 88, 98.

DeRemer, F.L., 1969. *Practical Translators for LR(k) Languages.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts. URL `http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-065.pdf`. Cited on pages 18, 88.

DeRemer, F.L., 1971. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460. doi: 10.1145/362619.362625. Cited on page 18.

Dijkstra, E.W., 1972. The humble programmer. *Communications of the ACM*, 15(10):859–866. doi: 10.1145/355604.361591. ACM Turing Award Lecture. Cited on pages 2, 172.

Dodd, C. and Maslov, V., 2006. *BTYACC – backtracking YACC.* Siber Systems. URL `http://www.siber.com/btyacc/`. Cited on page 39.

Donnely, C. and Stallman, R., 2006. *Bison version 2.3.* URL `http://www.gnu.org/software/bison/manual/`. Cited on pages 1, 20, 32, 40, 44, 45, 126, 171.

Earley, J., 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102. doi: 10.1145/362007.362035. Cited on pages 39, 40, 81, 167.

Earley, J., 1975. Ambiguity and precedence in syntax description. *Acta Informatica*, 4(2):183–192. doi: 10.1007/BF00288747. Cited on page 21.

Even, S., 1965. On information lossless automata of finite order. *IEEE Transactions on Electronic Computers*, EC-14(4):561–569. doi: 10.1109/PGEC.1965.263996. Cited on page 127.

Farré, J. and Fortes Gálvez, J., 2001. A bounded-connect construction for LR-Regular parsers. In Wilhelm, R., editor, *CC'01, 10th International Conference on Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 244–258. Springer. URL `http://springerlink.com/content/e3e8g77kxevkyjfd`. Cited on pages 36, 76, 117, 143.

Farré, J. and Fortes Gálvez, J., 2004. Bounded-connect noncanonical discriminating-reverse parsers. *Theoretical Computer Science*, 313(1):73–91. doi: 10.1016/j.tcs.2003.10.006. Cited on pages 38, 115.

Fischer, C.N., 1975. *On Parsing Context Free Languages in Parallel Environments.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York. URL `http://historical.ncstrl.org/tr/ps/cornellcs/TR75-237.ps`. Cited on page 38.

Floyd, R.W., 1962a. On ambiguity in phrase structure languages. *Communications of the ACM*, 5(10):526. doi: 10.1145/368959.368993. Cited on pages 2, 9, 15, 125, 172.

Floyd, R.W., 1962b. On the nonexistence of a phrase structure grammar for ALGOL 60. *Communications of the ACM*, 5(9):483–484. doi: 10.1145/368834.368898. Cited on page 14.

Floyd, R.W., 1963. Syntactic analysis and operator precedence. *Journal of the ACM*, 10(3):316–333. doi: 10.1145/321172.321179. Cited on page 20.

Floyd, R.W., 1964. Bounded context syntactic analysis. *Communications of the ACM*, 7(2):62–67. doi: 10.1145/363921.363927. Cited on page 20.

Ford, B., 2002. Packrat parsing: simple, powerful, lazy, linear time. In *ICFP'02, 7th International Conference on Functional Programming*, pages 36–47. ACM Press. ISBN 1-58113-487-8. doi: 10.1145/581478.581483. Cited on pages 15, 39.

Ford, B., 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL'04, 31st Annual Symposium on Principles of Programming Languages*, pages 111–122. ACM Press. ISBN 1-58113-729-X. doi: 10.1145/964001.964011. Cited on page 15.

Fortes Gálvez, J., 1998. *A Discriminating Reverse Approach to LR(k) Parsing*. PhD thesis, Universidad de Las Palmas de Gran Canaria and Université de Nice-Sophia Antipolis. Cited on page 20.

Fortes Gálvez, J., Schmitz, S., and Farré, J., 2006. Shift-resolve parsing: Simple, linear time, unbounded lookahead. In Ibarra, O.H. and Yen, H.C., editors, *CIAA'06, 11th International Conference on Implementation and Application of Automata*, volume 4094 of *Lecture Notes in Computer Science*, pages 253–264. Springer. ISBN 3-540-37213-X. doi: 10.1007/11812128_24. Cited on page ix.

Ganapathi, M., 1989. Semantic predicates in parser generators. *Computer Languages*, 14(1):25–33. doi: 10.1016/0096-0551(89)90028-3. Cited on page 34.

Geller, M.M. and Harrison, M.A., 1977a. Characteristic parsing: A framework for producing compact deterministic parsers, I. *Journal of Computer and System Sciences*, 14(3):265–317. doi: 10.1016/S0022-0000(77)80017-2. Cited on page 85.

Geller, M.M. and Harrison, M.A., 1977b. Characteristic parsing: A framework for producing compact deterministic parsers, II. *Journal of Computer* *and System Sciences*, 14(3):318–343. doi: 10.1016/S0022-0000(77)80018-4. Cited on pages 85, 86, 88.

Ginsburg, S. and Rice, H.G., 1962. Two families of languages related to ALGOL. *Journal of the ACM*, 9(3):350–371. doi: 10.1145/321127.321132. Cited on page 8.

Ginsburg, S. and Greibach, S., 1966. Deterministic context-free languages. *Information and Control*, 9(6):620–648. doi: 10.1016/S0019-9958(66)80019-0. Cited on page 16.

Ginsburg, S. and Ullian, J., 1966. Ambiguity in context free languages. *Journal of the ACM*, 13(1):62–89. doi: 10.1145/321312.321318. Cited on page 125.

Ginsburg, S. and Harrison, M.A., 1967. Bracketed context-free languages. *Journal of Computer and System Sciences*, 1:1–23. doi: 10.1016/S0022-0000(67)80003-5. Cited on pages 48, 50.

Gorn, S., 1963. Detection of generative ambiguities in context-free mechanical languages. *Journal of the ACM*, 10(2):196–208. doi: 10.1145/321160.321168. Cited on pages 4, 76, 126, 134, 174.

Gosling, J., Joy, B., and Steele, G., 1996. *The Java^{TM} Language Specification.* Addison-Wesley, first edition. ISBN 0-201-63451-1. URL `http://java.sun.com/docs/books/jls/`. Cited on pages 3, 13, 26, 28, 34, 112, 155, 173.

Graham, S.L., 1974. On bounded right context languages and grammars. *SIAM Journal on Computing*, 3(3):224–254. doi: 10.1137/0203019. Cited on page 20.

Graham, S.L., Harrison, M., and Ruzzo, W.L., 1980. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462. doi: 10.1145/357103.357112. Cited on page 39.

Grätzer, G., 1978. *General Lattice Theory*, volume 52 of *Lehrbücher und Monographien aus dem Gebiete der exakten Wissenschaften: Mathematische Reihe.* Birkhäuser. ISBN 3-7643-0813-3. Cited on page 58.

Griffiths, T.V. and Petrick, S.R., 1965. On the relative efficiencies of context-free grammar recognizers. *Communications of the ACM*, 8(5):289–300. doi: 10.1145/364914.364943. Cited on page 37.

Grimm, R., 2006. Better extensibility through modular syntax. In *PLDI'06, Conference on Programming Language Design and Implementation*, pages 38–51. ACM Press. ISBN 1-59593-320-4. doi: 10.1145/1133981.1133987. Cited on page 15.

Grosch, J., 2002. Lark – an LALR(2) parser generator with backtracking. Document No. 32, CoCoLab - Datenverarbeitung. URL `ftp://www.cocolab.com/products/cocktail/doc.pdf/lark.pdf`. Cited on page 39.

Grune, D. and Jacobs, C.J.H., 2007. *Parsing Techniques*. Monographs in Computer Science. Springer, second edition. ISBN 0-387-20248-X. Cited on pages 13, 56.

Harrison, M.A. and Havel, I.M., 1973. Strict deterministic grammars. *Journal of Computer and System Sciences*, 7(3):237–277. doi: 10.1016/S0022-0000(73)80008-X. Cited on pages 4, 86, 174.

Harrison, M.A., 1978. *Introduction to Formal Language Theory*. Series in Computer Science. Addison-Wesley. ISBN 0-201-02955-3. Cited on pages 5, 163.

Heering, J., Hendriks, P.R.H., Klint, P., and Rekers, J., 1989. The syntax definition formalism SDF–Reference Manual—. *ACM SIGPLAN Notices*, 24(11):43–75. doi: 10.1145/71605.71607. Cited on pages 1, 40, 171.

Heilbrunner, S., 1981. A parsing automata approach to LR theory. *Theoretical Computer Science*, 15(2):117–157. doi: 10.1016/0304-3975(81)90067-0. Cited on pages 60, 79, 117.

Heilbrunner, S., 1983. Tests for the LR-, LL-, and LC-Regular conditions. *Journal of Computer and System Sciences*, 27(1):1–13. doi: 10.1016/0022-0000(83)90026-0. Cited on pages 4, 79, 133, 143, 175.

Hopcroft, J.E. and Ullman, J.D., 1979. *Introduction to Automata Theory, Languages, and Computation*. Series in Computer Science. Addison-Wesley. ISBN 0-201-02988-X. Cited on pages 5, 163.

Hunt III, H.B., Szymanski, T.G., and Ullman, J.D., 1974. Operations on sparse relations and efficient algorithms for grammar problems. In *15th Annual Symposium on Switching and Automata Theory*, pages 127–132. IEEE Computer Society. Cited on pages 54, 56, 79, 143.

Hunt III, H.B., Szymanski, T.G., and Ullman, J.D., 1975. On the complexity of LR(k) testing. *Communications of the ACM*, 18(12):707–716. doi: 10.1145/361227.361232. Cited on pages 54, 143.

Hunt III, H.B., 1982. On the decidability of grammar problems. *Journal of the ACM*, 29(2):429–447. doi: 10.1145/322307.322317. Cited on page 35.

IBM Corporation, 1993. *VS COBOL II Application Programming Language Reference*, 4 edition. Document number GC26-4047-07. Cited on page 23.

Ichbiah, J.D. and Morse, S.P., 1970. A technique for generating almost optimal Floyd-Evans productions for precedence grammars. *Communications of the ACM*, 13(8):501–508. doi: 10.1145/362705.362712. Cited on page 20.

Irons, E.T., 1961. A syntax directed compiler for ALGOL 60. *Communications of the ACM*, 4(1):51–55. doi: 10.1145/366062.366083. Cited on page 39.

ISO, 1998. *ISO/IEC 14882:1998: Programming Languages — C++*. International Organization for Standardization, Geneva, Switzerland. Cited on pages 3, 29, 155, 173.

ISO, 2003. *ISO/IEC 9075-*:2003: Database Languages — SQL*. International Organization for Standardization, Geneva, Switzerland. Cited on page 13.

Jampana, S., 2005. Exploring the problem of ambiguity in context-free grammars. Master's thesis, Oklahoma State University. URL `http://e-archive.library.okstate.edu/dissertations/AAI1427836/`. Cited on pages 4, 126, 134, 174.

Jarzabek, S. and Krawczyk, T., 1975. LL-Regular grammars. *Information Processing Letters*, 4(2):31–37. doi: 10.1016/0020-0190(75)90009-5. Cited on pages 35, 132.

Johnson, S.C., 1975. YACC — yet another compiler compiler. Computing science technical report 32, AT&T Bell Laboratories, Murray Hill, New Jersey. Cited on pages 1, 13, 19, 21, 81, 171.

Johnstone, A. and Scott, E., 1998. Generalised recursive descent parsing and follow-determinism. In Koskimies, K., editor, *CC'98, 7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 16–30. Springer. ISBN 978-3-540-64304-3. doi: 10.1007/BFb0026420. Cited on page 39.

Jurdziński, T. and Loryś, K., 2007. Lower bound technique for length-reducing automata. *Information and Computation*, 205(9):1387–1412. doi: 10.1016/j.ic.2007.02.003. Cited on pages 161, 177.

Kahrs, S., 1993. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, University of Edinburgh, LFCS. URL `http://www.lfcs.inf.ed.ac.uk/reports/93/ECS-LFCS-93-257/`. Cited on pages 28, 29, 31.

Kamimura, T. and Slutzki, G., 1981. Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, 49(1):10–51. doi: 10.1016/S0019-9958(81)90438-1. Cited on page 67.

Kannapinn, S., 2001. *Reconstructing LR Theory to Eliminate Redundance, with an Application to the Construction of ELR Parsers*. PhD thesis, Technical University of Berlin, Department of Computer Sciences. URL `http://edocs.tu-berlin.de/diss/2001/kannapinn_soenke.htm`. Cited on page 20.

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, Massachussetts. Cited on page 39.

Kay, M., 1980. Algorithm schemata and data structures in syntactic processing. Technical Report CSL-80-12, Xerox Palo Alto Research Center. Cited on page 39.

Kernighan, B.W. and Ritchie, D.M., 1988. *The C Programming Language*. Prentice-Hall. ISBN 0-13-110362-8. Cited on pages 13, 23, 112, 152.

Klint, P. and Visser, E., 1994. Using filters for the disambiguation of context-free grammars. In Pighizzini, G. and San Pietro, P., editors, *ASMICS Workshop on Parsing Theory*, Technical Report 126-1994, pages 89–100. Università di Milano. URL `http://citeseer.ist.psu.edu/klint94using.html`. Cited on pages 2, 23, 44, 172.

Klint, P., Lämmel, R., and Verhoef, C., 2005. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology*, 14(3):331–380. doi: 10.1145/1072997.1073000. Cited on pages 1, 171.

Knuth, D.E., 1965. On the translation of languages from left to right. *Information and Control*, 8(6):607–639. doi: 10.1016/S0019-9958(65)90426-2. Cited on pages 4, 17, 37, 51, 79, 82, 136, 143, 174.

Knuth, D.E., 1967. A characterization of parenthesis languages. *Information and Control*, 11(3):269–289. doi: 10.1016/S0019-9958(67)90564-5. Cited on page 67.

Knuth, D.E., 1968. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145. doi: 10.1007/BF01692511. Cited on page 14.

Koster, C.H.A., 1971. Affix grammars. In Peck, J.E.L., editor, *ALGOL 68 Implementation*, pages 95–109. North-Holland. ISBN 0-7204-2045-8. Cited on page 14.

Kristensen, B.B. and Madsen, O.L., 1981. Methods for computing LALR(k) lookahead. *ACM Transactions on Programming Languages and Systems*, 3(1):60–82. doi: 10.1145/357121.357126. Cited on page 88.

Kron, H.H., Hoffmann, H.J., and Winkler, G., 1974. On a SLR(k)-based parser system which accepts non-LR(k) grammars. In Siefkes, D., editor, *4 Jahrestagung der Gesellschaft für Informatik*, volume 26 of *Lecture Notes in Computer Science*, pages 214–223. Springer. ISBN 978-3-540-07141-9. doi: 10.1007/3-540-07141-5_224. Cited on page 36.

Kuich, W., 1970. Systems of pushdown acceptors and context-free grammars. *Elektronische Informationsverarbeitung und Kybernetik*, 6(2):95–114. Cited on pages 78, 79.

Kuno, S., 1965. The predictive analyzer and a path elimination technique. *Communications of the ACM*, 8(7):453–462. doi: 10.1145/364995.365689. Cited on page 39.

Kurki-Suonio, R., 1969. Notes on top-down languages. *BIT Numerical Mathematics*, 9(3):225–238. doi: 10.1007/BF01946814. Cited on page 18.

Lämmel, R. and Verhoef, C., 2001. Semi-automatic grammar recovery. *Software: Practice & Experience*, 31:1395–1438. doi: 10.1002/spe.423. Cited on pages 23, 157.

Lamport, L., 1994. *LaTeX: A Document Preparation System*. Addison-Wesley, second edition. ISBN 0-201-52983-1. Cited on page 13.

Lang, B., 1974. Deterministic techniques for efficient non-deterministic parsers. In Loeckx, J., editor, *ICALP'74, 2nd International Colloquium on Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer. ISBN 3-540-06841-4. doi: 10.1007/3-540-06841-4_65. Cited on pages 39, 40.

Lee, P., 1997. *Using the SML/NJ System*. Carnegie Mellon University. URL `http://www.cs.cmu.edu/~petel/smlguide/smlnj.htm`. Cited on page 32.

Leermakers, R., 1992. Recursive ascent parsing: from Earley to Marcus. *Theoretical Computer Science*, 104(2):299–312. doi: 10.1016/0304-3975(92)90127-2. Cited on page 12.

Leijen, D. and Meijer, E., 2001. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht. URL `http://www.cs.uu.nl/~daan/parsec.html`. Cited on pages 18, 39.

Leo, J.M.I.M., 1991. A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. *Theoretical Computer Science*, 82(1):165–176. doi: 10.1016/0304-3975(91)90180-A. Cited on pages 160, 176.

Lewis II, P.M. and Stearns, R.E., 1968. Syntax-directed transduction. *Journal of the ACM*, 15(3):465–488. doi: 10.1145/321466.321477. Cited on page 18.

Lewis II, P.M., Rosenkrantz, D.J., and Stearns, R.E., 1974. Attributed translations. *Journal of Computer and System Sciences*, 9(3):279–307. doi: 10.1016/S0022-0000(74)80045-0. Cited on page 18.

Makarov, V., 1999. *MSTA (syntax description translator)*. URL `http://cocom.sourceforge.net/msta.html`. Cited on page 154.

Marcus, M.P., 1980. *A Theory of Syntactic Recognition for Natural Language*. Series in Artificial Intelligence. MIT Press. ISBN 0-262-13149-8. Cited on page 12.

Marcus, M.P., Marcinkiewicz, M.A., and Santorini, B., 1993. Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, 19(2):313–330. URL `http://www.aclweb.org/anthology/J93-2004`. Cited on page 43.

McKeeman, W.M., Horning, J.J., and Wortman, D.B., 1970. *A Compiler Generator*. Series in Automatic Computation. Prenctice-Hall. ISBN 13-155077-2. Cited on page 20.

McNaughton, R., 1967. Parenthesis grammars. *Journal of the ACM*, 14 (3):490–500. doi: 10.1145/321406.321411. Cited on page 50.

McNaughton, R., Narendran, P., and Otto, F., 1988. Church-Rosser Thue systems and formal languages. *Journal of the ACM*, 35(2):324–344. doi: 10.1145/42282.42284. Cited on pages 160, 176.

McPeak, S. and Necula, G.C., 2004. Elkhound: A fast, practical GLR parser generator. In Duesterwald, E., editor, *CC'04, 13th International Conference on Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 73–88. Springer. ISBN 3-540-21297-3. doi: 10.1007/b95956. Cited on pages 1, 2, 31, 40, 44, 45, 151, 153, 171, 172.

Megacz, A., 2006. Scannerless boolean parsing. In Boyland, J. and Sloane, A., editors, *LDTA'06, 6th Workshop on Language Descriptions, Tools and Applications*, volume 164(2) of *Electronic Notes in Theoretical Computer Science*, pages 97–102. Elsevier. doi: 10.1016/j.entcs.2006.10.007. Cited on page 15.

Mickunas, M.D., Lancaster, R.L., and Schneider, V.B., 1976. Transforming LR(k) grammars to LR(1), SLR(1), and (1,1) Bounded Right-Context grammars. *Journal of the ACM*, 23(3):511–533. doi: 10.1145/321958.321972. Cited on page 24.

Milner, R., Tofte, M., Harper, R., and MacQueen, D., 1997. *The definition of Standard ML*. MIT Press, revised edition. ISBN 0-262-63181-4. Cited on pages 3, 13, 28, 153, 173.

Mohri, M. and Nederhof, M.J., 2001. Regular approximations of context-free grammars through transformation. In Junqua, J.C. and van Noord, G., editors, *Robustness in Language and Speech Technology*, volume 17 of *Text, Speech and Language Technology*, chapter 9, pages 153–163. Kluwer Academic Publishers. ISBN 0-7923-6790-1. URL `http://citeseer.ist.psu.edu/mohri00regular.html`. Cited on pages 78, 131.

Moore, R.C., 2004. Improved left-corner chart parsing for large context-free grammars. In *New Developments in Parsing Technology*, pages 185–201. Springer. ISBN 1-4020-2293-X. doi: 10.1007/1-4020-2295-6_9. Cited on pages 40, 43.

Murata, M., Lee, D., Mani, M., and Kawaguchi, K., 2005. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704. doi: 10.1145/1111627.1111631. Cited on pages 48, 67.

Naur, P., editor, 1960. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314. doi: 10.1145/367236.367262. Cited on pages 8, 21.

Nederhof, M.J., 1993. Generalized left-corner parsing. In *EACL'93, 6th conference of the European chapter of the Association for Computational Linguistics*, pages 305–314. ACL Press. ISBN 90-5434-014-2. doi: 10.3115/976744.976780. Cited on page 40.

Nederhof, M.J., 1998. Context-free parsing through regular approximation. In *FSMNLP'98, 2nd International Workshop on Finite State Methods in Natural Language Processing*, pages 13–24. URL `http://www.aclweb.org/anthology/W98/W98-1302.pdf`. Cited on page 78.

Nederhof, M.J., 2000a. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1):17–44. doi: 10.1162/089120100561610. Cited on page 78.

Nederhof, M.J., 2000b. Regular approximation of CFLs: a grammatical view. In Bunt, H. and Nijholt, A., editors, *Advances in Probabilistic and* *other Parsing Technologies*, volume 16 of *Text, Speech and Language Technology*, chapter 12, pages 221–241. Kluwer Academic Publishers. ISBN 0-7923-6616-6. URL `http://odur.let.rug.nl/~markjan/publications/2000d.pdf`. Cited on page 78.

Nederhof, M.J. and Satta, G., 2004. Tabular parsing. In Martín-Vide, C., Mitrana, V., and Păun, G., editors, *Formal Languages and Applications*, volume 148 of *Studies in Fuzziness and Soft Computing*, pages 529–549. Springer. ISBN 3-540-20907-7. arXiv:cs.CL/0404009. Cited on page 40.

Neven, F., 2002. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46. doi: 10.1145/601858.601869. Cited on page 48.

Niemann, G. and Otto, F., 1999. Restarting automata, Church-Rosser languages, and representations of r.e. languages. In *DLT'99*, *4th International Conference on Developments in Language Theory*, pages 103–114. World Scientific. ISBN 981-02-4380-4. Cited on pages 160, 177.

Niemann, G. and Otto, F., 2005. The Church-Rosser languages are the deterministic variants of the growing context-sensitive languages. *Information and Control*, 197(1–2):1–21. doi: 10.1016/j.ic.2004.09.003. Cited on pages 160, 177.

Nijholt, A., 1976. On the parsing of LL-Regular grammars. In Mazurkiewicz, A., editor, *MFCS'76*, *5th International Symposium on Mathematical Foundations of Computer Science*, volume 45 of *Lecture Notes in Computer Science*, pages 446–452. Springer. ISBN 3-540-07854-1. doi: 10.1007/3-540-07854-1_213. Cited on pages 35, 132.

Nozohoor-Farshi, R., 1986. On formalizations of Marcus' parser. In *COLING'86*, *11th International Conference on Computational Linguistics*, pages 533–535. ACL Press. doi: 10.3115/991365.991520. Cited on page 12.

Nozohoor-Farshi, R., 1987. Context-freeness of the language accepted by Marcus' parser. In *ACL'87*, *25th Annual Meeting of the Association for Computational Linguistics*, pages 117–122. ACL Press. doi: 10.3115/981175.981192. Cited on page 12.

Okhotin, A., 2004. A boolean grammar for a simple programming language. Technical Report 2004-478, Queen's University, Kingston, Ontario, Canada. URL `http://www.cs.queensu.ca/TechReports/Reports/2004-478.pdf`. Also in *AFL'05*. Cited on page 15.

Ore, O., 1942. Theory of equivalence relations. *Duke Mathematical Journal*, 9(3):573–627. doi: 10.1215/S0012-7094-42-00942-6. Cited on page 58.

Ore, O., 1944. Galois connexions. *Transactions of the American Mathematical Society*, 55(3):493–513. doi: 10.2307/1990305. Cited on page 62.

Overbey, J., 2006. Practical, incremental, noncanonical parsing: Celentano's method and the Generalized Piecewise LR parsing algorithm. Master's thesis, Department of Computer Science, University of Illinois. URL `http://jeff.over.bz/papers/2006/ms-thesis.pdf`. Cited on pages 38, 114.

Pager, D., 1977. A practical general method for constructing LR(k) parsers. *Acta Informatica*, 7(3):249–268. doi: 10.1007/BF00290336. Cited on page 20.

Parr, T.J. and Quong, R.W., 1995. ANTLR: A predicated-LL(k) parser generator. *Software: Practice & Experience*, 25(7):789–810. doi: 10.1002/spe.4380250705. Cited on page 34.

Parr, T.J. and Quong, R.W., 1996. LL and LR translators need $k > 1$ lookahead. *ACM SIGPLAN Notices*, 31(2):27–34. doi: 10.1145/226060.226066. Cited on pages 1, 23, 171.

Parr, T.J., 2007. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers. ISBN 0-9787392-5-6. Cited on pages 23, 36, 39.

Pereira, F.C.N. and Warren, D.H.D., 1983. Parsing as deduction. In *ACL'83, 21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144. ACL Press. doi: 10.3115/981311.981338. Cited on pages 39, 167.

Peyton Jones, S., editor, 2003. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press. ISBN 0-521-82614-4. URL `http://haskell.org/definition/haskell98-report.pdf`. Also in *Journal of Functional Programming*, 13(1). Cited on page 23.

Poplawski, D.A., 1979. On LL-Regular grammars. *Journal of Computer and System Sciences*, 18(3):218–227. doi: 10.1016/0022-0000(79)90031-X. Cited on pages 35, 132.

Pritchett, B.L., 1992. *Grammatical Competence and Parsing Performance*. The University of Chicago Press. ISBN 0-226-68441-5. Cited on page 12.

Purdom, P., 1974. The size of LALR(1) parsers. *BIT Numerical Mathematics*, 14(3):326–337. doi: 10.1007/BF01933232. Cited on pages 19, 157.

Rabin, M.O. and Scott, D., 1959. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3 (2):114–125. URL `http://www.research.ibm.com/journal/rd/032/ibmrd0302C.pdf`. Cited on page 83.

Reeder, J., Steffen, P., and Giegerich, R., 2005. Effective ambiguity checking in biosequence analysis. *BMC Bioinformatics*, 6:153. doi: 10.1186/1471-2105-6-153. Cited on pages 125, 143, 151, 152, 154.

Rosenkrantz, D.J. and Lewis II, P.M., 1970. Deterministic left corner parsing. In *11th Annual Symposium on Switching and Automata Theory*, pages 139–152. IEEE Computer Society. Cited on page 20.

Rosenkrantz, D.J. and Stearns, R.E., 1970. Properties of deterministic top-down grammars. *Information and Control*, 17(3):226–256. doi: 10.1016/S0019-9958(70)90446-8. Cited on pages 18, 132.

Rossberg, A., 2006. Defects in the revised definition of Standard ML. Technical report, Saarland University, Saarbrücken, Germany. URL `http://ps.uni-sb.de/Papers/paper_info.php?label=sml-defects`. Cited on page 34.

Ruckert, M., 1999. Continuous grammars. In *POPL'99*, *26th Annual Symposium on Principles of Programming Languages*, pages 303–310. ACM Press. ISBN 1-58113-095-3. doi: 10.1145/292540.292568. Cited on page 38.

Rus, T. and Jones, J.S., 1998. PHRASE parsers from multi-axiom grammars. *Theoretical Computer Science*, 199(1–2):199–229. doi: 10.1016/S0304-3975(97)00273-9. Cited on page 38.

Salomon, D.J. and Cormack, G.V., 1989. Scannerless NSLR(1) parsing of programming languages. In *PLDI'89*, *Conference on Programming Language Design and Implementation*, pages 170–178. ACM Press. ISBN 0-89791-306-X. doi: 10.1145/73141.74833. Cited on pages 14, 38, 110.

Schell, R.M., 1979. *Methods for constructing parallel compilers for use in a multiprocessor environment.* PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign. Cited on pages 38, 114.

Schmitz, S., 2006. Noncanonical LALR(1) parsing. In Dang, Z. and Ibarra, O.H., editors, *DLT'06*, *10th International Conference on Developments in Language Theory*, volume 4036 of *Lecture Notes in Computer Science*, pages 95–107. Springer. ISBN 3-540-35428-X. doi: 10.1007/11779148_10. Cited on page ix.

Schmitz, S., 2007a. Conservative ambiguity detection in context-free grammars. In Arge, L., Cachin, C., Jurdziński, T., and Tarlecki, A., editors, *ICALP'07, 34th International Colloquium on Automata, Languages and Programming*, volume 4596 of *Lecture Notes in Computer Science*, pages 692–703. Springer. ISBN 978-3-540-73419-2. doi: 10.1007/978-3-540-73420-8_60. Cited on page ix.

Schmitz, S., 2007b. An experimental ambiguity detection tool. In Sloane, A. and Johnstone, A., editors, *LDTA'07, 7th Workshop on Language Descriptions, Tools and Applications*. URL `http://www.i3s.unice.fr/~mh/RR/2006/RR-06.37-S.SCHMITZ.pdf`. To appear in *Electronic Notes in Theoretical Computer Science*. Cited on page ix.

Schöbel-Theuer, T., 1994. Towards a unifying theory of context-free parsing. In Pighizzini, G. and San Pietro, P., editors, *ASMICS Workshop on Parsing Theory*, Technical Report 126-1994, pages 89–100. Università di Milano. URL `http://citeseer.ist.psu.edu/117476.html`. Cited on pages 76, 77.

Schröer, F.W., 2001. AMBER, an ambiguity checker for context-free grammars. Technical report, compilertools.net. URL `http://accent.compilertools.net/Amber.html`. Cited on pages 4, 126, 134, 154, 174.

Schwentick, T., 2007. Automata for XML—a survey. *Journal of Computer and System Sciences*, 73(3):289–315. doi: 10.1016/j.jcss.2006.10.003. Cited on page 48.

Scott, E. and Johnstone, A., 2006. Right nulled GLR parsers. *ACM Transactions on Programming Languages and Systems*, 28(4):577–618. doi: 10.1145/1146809.1146810. Cited on pages 25, 40.

Segoufin, L. and Vianu, V., 2002. Validating streaming XML documents. In *PODS'02, twenty-first Symposium on Principles of Database Systems*, pages 53–64. ACM Press. ISBN 1-58113-507-6. doi: 10.1145/543613.543622. Cited on pages 3, 65, 68, 174.

Seité, B., 1987. A Yacc extension for LRR grammar parsing. *Theoretical Computer Science*, 52:91–143. doi: 10.1016/0304-3975(87)90082-X. Cited on pages 36, 117, 143.

Shieber, S.M., 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8(3):333–343. doi: 10.1007/BF00630917. Cited on page 8.

Sikkel, K., 1997. *Parsing Schemata - a framework for specification and analysis of parsing algorithms.* Texts in Theoretical Computer Science -

An EATCS Series. Springer. ISBN 3-540-61650-0. Cited on pages 39, 56, 77, 167.

Sikkel, K., 1998. Parsing schemata and correctness of parsing algorithms. *Theoretical Computer Science*, 199(1–2):87–103. doi: 10.1016/S0304-3975(97)00269-7. Cited on page 78.

Sippu, S. and Soisalon-Soininen, E., 1982. On LL(k) parsing. *Information and Control*, 53(3):141–164. doi: 10.1016/S0019-9958(82)91016-6. Cited on page 132.

Sippu, S. and Soisalon-Soininen, E., 1988. *Parsing Theory, Vol. I: Languages and Parsing*, volume 15 of *EATCS Monographs on Theoretical Computer Science*. Springer. ISBN 3-540-13720-3. Cited on pages 5, 163.

Sippu, S. and Soisalon-Soininen, E., 1990. *Parsing Theory, Vol. II: LR(k) and LL(k) Parsing*, volume 20 of *EATCS Monographs on Theoretical Computer Science*. Springer. ISBN 3-540-51732-4. Cited on pages 13, 96, 99, 101, 102, 127, 138, 168.

Soisalon-Soininen, E. and Ukkonen, E., 1979. A method for transforming grammars into LL(k) form. *Acta Informatica*, 12(4):339–369. doi: 10.1007/BF00268320. Cited on page 20.

Soisalon-Soininen, E. and Tarhio, J., 1988. Looping LR parsers. *Information Processing Letters*, 26(5):251–253. doi: 10.1016/0020-0190(88)90149-4. Cited on page 23.

Spencer, M., 2002. *Basil: A Backtracking LR(1) Parser Generator*. URL `http://lazycplusplus.com/basil/`. Cited on page 39.

Szymanski, T.G., 1973. *Generalized Bottom-Up Parsing*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York. URL `http://historical.ncstrl.org/tr/ps/cornellcs/TR73-168.ps`. Cited on page 37.

Szymanski, T.G. and Williams, J.H., 1976. Noncanonical extensions of bottom-up parsing techniques. *SIAM Journal on Computing*, 5(2):231–250. doi: 10.1137/0205019. Cited on pages 37, 81, 96, 109, 111, 151.

Tai, K.C., 1979. Noncanonical SLR(1) grammars. *ACM Transactions on Programming Languages and Systems*, 1(2):295–320. doi: 10.1145/357073.357083. Cited on pages 4, 37, 81, 103, 104, 110, 111, 112, 116, 155, 160, 174, 176.

Tarjan, R.E., 1972. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160. doi: 10.1137/0201010. Cited on pages 98, 128.

Thatcher, J.W., 1967. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 1(4):317–322. doi: 10.1016/S0022-0000(67)80022-9. Cited on page 48.

Thorup, M., 1994. Controlled grammatic ambiguity. *ACM Transactions on Programming Languages and Systems*, 16(3):1024–1050. doi: 10.1145/177492.177759. Cited on page 23.

Thurston, A.D. and Cordy, J.R., 2006. A backtracking LR algorithm for parsing ambiguous context-dependent languages. In *CASCON'06, Centre for Advanced Studies on Collaborative research*. IBM Press. doi: 10.1145/1188966.1188972. Cited on page 39.

Tomita, M., 1986. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers. ISBN 0-89838-202-5. Cited on pages 1, 40, 81, 171.

van den Brand, M.G.J., Sellink, A., and Verhoef, C., 1998. Current parsing techniques in software renovation considered harmful. In *IWPC'98, 6th International Workshop on Program Comprehension*, pages 108–117. IEEE Computer Society. ISBN 0-8186-8560-3. doi: 10.1109/WPC.1998.693325. Cited on pages 1, 171.

van den Brand, M., Scheerder, J., Vinju, J.J., and Visser, E., 2002. Disambiguation filters for scannerless generalized LR parsers. In Horspool, R.N., editor, *CC'02, 11th International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer. ISBN 3-540-43369-4. URL `http://www.springerlink.com/content/03359k0cerupftfh/`. Cited on pages 40, 44, 46.

van den Brand, M., Klusener, S., Moonen, L., and Vinju, J.J., 2003. Generalized parsing and term rewriting: Semantics driven disambiguation. In Bryant, B. and Saraiva, J., editors, *LDTA'03, 3rd Workshop on Language Descriptions, Tools and Applications*, volume 82(3) of *Electronic Notes in Theoretical Computer Science*, pages 575–591. Elsevier. doi: 10.1016/S1571-0661(05)82629-5. Cited on page 23.

van Wijngaarden, A., editor, 1975. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, 5(1–3):1–236. doi: 10.1007/BF00265077. Cited on page 14.

Visser, E., 1997. Scannerless generalized-LR parsing. Technical Report P9707, University of Amsterdam. URL `http://citeseer.ist.psu.edu/visser97scannerles.html`. Cited on page 14.

Wagner, T.A. and Graham, S.L., 1997. Incremental analysis of real programming languages. In *PLDI'97, Conference on Programming Language Design and Implementation*, pages 31–43. ACM Press. ISBN 0-89791-907-6. doi: 10.1145/258915.258920. Cited on page 40.

Watt, D.A., 1980. Rule splitting and attribute-directed parsing. In Jones, N.D., editor, *Workshop on Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 363–392. Springer. ISBN 3-540-10250-7. doi: 10.1007/3-540-10250-7_29. Cited on page 14.

Wich, K., 2005. *Ambiguity Functions of Context-Free Grammars and Languages*. PhD thesis, Institut fur Formale Methoden der Informatik, Universität Stuttgart. URL `ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/DIS-2005-01/DIS-2005-01.pdf`. Cited on page 43.

Williams, J.H., 1975. Bounded context parsable grammars. *Information and Control*, 28(4):314–334. doi: 10.1016/S0019-9958(75)90330-7. Cited on page 37.

Wirth, N. and Weber, H., 1966. EULER: a generalization of ALGOL and its formal definition. *Communications of the ACM*, 9(1):Part I: 13–25, and Part II: 89–99. doi: 10.1145/365153.365162. Cited on page 20.

Woods, W.A., 1970. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606. doi: 10.1145/355598.362773. Cited on page 76.

Yergeau, F., Cowan, J., Bray, T., Paoli, J., Sperberg-McQueen, C.M., and Maler, E., 2004. Extensible markup language (XML) 1.1. Recommendation, W3C. URL `http://www.w3.org/TR/xml11`. Cited on page 13.

Younger, D.H., 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208. doi: 10.1016/S0019-9958(67)80007-X. Cited on page 39.

Index

Cover art generated using the Context-Free Art software and the following (ambiguous) weighted context-free grammar.

```
# $Id: bush.cfdg,v 1.5 2007/06/28 17:24:06 schmitz Exp $
# Thorn bush with roses.
#
# Adapted from demo1.cfdg, rose.cfdg and tangle.cfdg of the
# Context-Free Art software distribution, and as such is a derivative
# work of GPL'ed files.  So you know the drill:
#
# Copyright (C) 2006, 2007 Sylvain Schmitz
#
# This file is free software; you can redistribute it and/or modify it
# under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This file is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
# General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this file; if not, write to the Free Software Foundation,
# Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301  USA

startshape bush
rule bush {
  seed {}
  seed {rotate  20}
  seed {rotate -30}
  fork {}
  seed {size 0.5}
}

rule seed {branch {}}
rule seed {branch {rotate  1}}
rule seed {branch {rotate -1}}
rule seed {branch {rotate  2}}
rule seed {branch {rotate -2}}
rule seed {fork {}}

rule branch {lbranch {flip 90}}
rule branch {lbranch {}}

rule lbranch 3  {wood {} lbranch {y 0.885 rotate 0.1 size 0.99}}
rule lbranch 3  {wood {} lbranch {y 0.885 rotate 0.2 size 0.99}}
rule lbranch 3  {wood {} lbranch {y 0.885 rotate 4   size 0.99}}
rule lbranch 1.2  {wood {}   fork {}}
rule lbranch 0.02 {rose {size 6 b 1}}
rule wood     {SQUARE {rotate 1} SQUARE {rotate -1} SQUARE {}}
rule wood 0.5 {
  SQUARE {rotate  1}
  SQUARE {rotate -1}
  SQUARE {}
  thorns { size 0.3 }
  thorns { flip 90 size 0.3 }
}

rule fork {branch {} branch {size 0.5 rotate  40}}
rule fork {branch {} branch {size 0.5 rotate -40}}
rule fork {branch {size 0.5 rotate -20} branch {}}
rule fork {branch {size 0.5 rotate -50} branch {size 0.5 rotate 20}}
rule fork {
  branch {size 0.7 y 0.1 rotate  30}
  branch {size 0.7 y 0.1 rotate -30}
}

rule thorns     {thorn {} thorn {rotate 70 size 0.8}}
rule thorns 0.8 {
  thorn {size 0.7}
  thorn {rotate 120}
  thorn {rotate 190 size 0.9}
}

rule thorn {lthorn {}}
rule thorn {lthorn {flip 90}}

rule lthorn {SQUARE {} lthorn {y 0.9 size 0.9 rotate 1}}

rule rose {flower {}}
rule rose {flower {rotate  10}}
rule rose {flower {rotate -10}}

rule flower {
  petal {}
  petal {r  90}
  petal {r 180}
  petal {r 270}
  flower {r 14 size 0.9 b 1}
}

rule petal {petal1 {r  15 x 0.9 s 0.9 1.8}}
rule petal {petal1 {r -15 x 0.9 s 0.9 1.8}}

rule petal1 {
  CIRCLE {}
  CIRCLE {s 0.975 x -0.025 b -0.8}
  CIRCLE {s 0.95  x -0.05  b  1}
  CIRCLE {s 0.925 x -0.075 b  0.01|}
}
```