A Discriminating Reverse Approach to LR(k) Parsing

Fortes Gálvez, José

November 1998

Preface

> *(...) then there is a finite state automaton that can, by reading the grammar symbols on the stack from top to bottom, determine what handle, if any, is on top of the stack.*
>
> Erratum in [ASU86, page 221], but correct!

Original shift-reduce parsers used the notion of precedence, which in general terms is based on the idea that the combination of several top of stack symbols plus some lookahead is used to decide whether to shift or to reduce, and with which rule to reduce. The main weakness of this approach is that the source of information used in the decisions is *bounded* to some small window around the top of the stack, and then it is frequent for programming language grammars the existence of some cases where a wider context is needed to uniquely determine the parsing action. Although limited, these methods were very popular for their efficiency until the advent of LR methods, which greatly surpassed them in parsing power, because they use (almost) all the stack plus lookahead necessary information to take a parsing decision. Since then, precedence and related methods have been almost forgotten, and most research efforts have concentrated onto reducing the storage needs of LR parsers while preserving their parsing power, without much success, since classic full LR(1) methods have the disadvantage of producing unacceptable large automata in relation to grammar size, specially when building parsers for programming language grammars. Consequently, the currently recognized method of choice continues to be LALR(1), where a practical compromise is found by reducing automaton size at the cost of some parsing power.

In this thesis we further investigate this subject, presenting an original approach to shift-reduce parsing through a combination of both above approaches. Ours tries to preserve as much as possible the efficiency of resources used to take a parsing decision, while obtaining the maximum parsing power —full LR(k)— which is possible when all the useful stack plus lookahead information is available. For that purpose, an automaton is built to *begin reading the stack from its top* to decide, in combination with the lookahead, which parsing action to perform next. Each automaton state has the information of what parsing actions are compatible with the portion of the stack read thus far. As soon as the next stack symbol or the lookahead is enough to decide the parsing action to perform, no further states are built, since the parsing action will be unique from this point on. Therefore, the discriminating reverse parser does not need to recognize the whole stack, but only its action discriminating part, which leads to a much smaller automaton, since classical full LR methods build a full-stack recognizing automaton such that the relevant parsing information is condensed on the states, which are used for the parsing decisions in combination with the current lookahead.

The new method is here formally proved to be correct for the general full LR(k), $k \geq 0$, case. It is also shown to be linear with input length, be it correct or not, and to be extensible to accept ambiguous grammars where the designer can define a choice of disambiguation rules.

An experimental parser generator was built by the author in order to test our method. Practical results obtained by us and independently are in accordance with the predicted parsing power and small automata sizes. For large LALR(1) programming language grammars, the number of states of the discriminating reverse parsers have been found to be around 40% of those corresponding to `yacc`-generated parsers. Non-LALR(1) LR(1) grammars, although indeed usually more complicated, pose no specially relevant problem to the discriminating reverse LR(1) parsers.

In theory, parsing efficiency compares unfavorably with classic parsing methods because the latter do not search in the stack. However, the reverse automaton deeply searches in the stack —relatively short for programming language grammars— only in the worst theoretical cases. Even for those worst cases, that should not happen frequently since, in the case of a hard left context configuration, the resulting nonterminal will normally be different for the alternative actions, which usually is enough for not to search subsequently so deeply. Experimental results for practical grammars show that the average stack depth of exploration per parsing action is quite stable around about 2 stack symbols.

Errors are always detected, obviously, although the method has not in principle the correct-prefix property, as it is the rule for precedence methods. Nevertheless, error-free inputs are indeed always well parsed, and reductions are always correct. It is felt that well-known techniques of error recovery for precedence parsers might be adapted to the discriminating reverse approach.

Document contents

Chapter 1 introduces in the first place definitions and terminology to be used throughout the text. Chapter 2 reviews relevant practical existing parsing techniques, and how they are used in available parser generation tools. Chapter 3 serves as motivation for our method and to introduce some useful notions. Successive chapters progressively develop the technique. Chapter 4 shows the case for LR(0) parsing power, and introduces much of the concepts that are developed in Chapter 5, which is devoted to develop in detail the most important practical case to obtain LR(1) parsing power, with the presentation of numerous examples. Amongst the several possible alternatives, the most promising one, where lookahead use is deferred until it is sufficient to decide the parsing action, is adopted and fully developed. In Chapter 6, performance results from study of cases and from experimental results is shown and evaluated. Chapter 7 shows then the general LR(k) case, which is formally proved, including its linear time complexity on input length. The exposition ends with several possibilities of extensions and variants briefly presented in Chapter 8, and with the general conclusions in Chapter 9, where further lines of research around the thesis subject are mentioned. The document includes three Appendixes containing some automata listings, a brief user manual for the experimental tool developed by the author, and experimental results independently obtained.

Some chapters are organized presenting the concepts in a relatively informal manner first, with details and examples, and leaving for the last sections the formal treatment with algorithms and proofs.

Contents

Chapter 1

Definitions

Let us present in the first place the basic definitions, as well as some terminology and conventions, that will be used throughout this thesis.

Let $\Sigma = \{Z_1, \ldots, Z_r\}$ be a finite set of r different *symbols*. A *sequence* is an ordered concatenation of symbols in Σ. Two sequences $\alpha = X_1 \cdots X_l$ and $\beta = Y_1 \cdots Y_m$ can be concatenated producing the new sequence

$$\alpha\beta = X_1 \cdots X_l Y_1 \cdots Y_m.$$

Only finite sequences will be considered. The number of symbols in a sequence $\alpha = X_1 \cdots X_l$ will be represented as $|\alpha| = l$. If X is a symbol, then X^k (for k nonnegative integer) is the sequence of k X's, i.e., $|X^k| = k$. The *empty* or *null* sequence will represented as ε, i.e., $|\varepsilon| = 0$.

A *language* is a (possibly infinite) set of sequences, called *sentences*. The k-th *power*, $k \geq 0$, of Σ is a language defined as follows:

$$\Sigma^k = \{X_1 \cdots X_k \mid X_1, \ldots, X_k \in \Sigma\},$$

i.e., all the possible sequences of length k on Σ. The sets of all (nonempty and possibly empty, respectively) sentences of any length on Σ are defined as follows:

$$\Sigma^+ = \bigcup_{k>0} \Sigma^k,$$

and

$$\Sigma^* = \Sigma^0 \cup \Sigma^+.$$

A *dotted* sequence is a symbol sequence with a *dot* to indicate a position in the sequence. It can be written $\alpha \bullet \beta$ or $X_1 \cdots X_l \bullet X_{l+1} \cdots X_m$, with $0 \leq l \leq m$, i.e., one or both sub-sequences may be empty.

Definition 1.1 *A* context-free grammar $\mathcal{G}$ *is a 4-tuple* (V_T, V_N, P, S)*, where*

- V_T *is a finite set of* terminal *symbols,*

- V_N, $V_N \cap V_T = \emptyset$, *is a finite set of* nonterminal *symbols, (*$V = V_T \cup V_N$ *is called the* vocabulary*),*

- P, $P \subset V_N \times V^*$, *is a finite set of* productions *(also called* rules*), which are normally written as* $A \rightarrow \alpha$ *(*A *and* α *will be called the rule* leftpart *and* rightpart*, respectively), and*

- $S \in V_N$ *is the* start *symbol.*

□

Throughout the text the usual convention for notation of symbols and sequences will be used, as follows:

- Symbols in V_T will be represented with the first Latin letters in lower case, i.e., a, b, c, $\ldots$,

- Symbols in V_N will be represented with the first Latin letters in upper case, i.e., A, B, C, $\ldots$,

- Symbols in V will be represented with the last Latin letters in upper case, i.e., Z, Y, X, $\ldots$,

- Sequences in $V_T{}^*$ will be represented with the last Latin letters in lower case, i.e., z, y, x, $\ldots$, and, finally,

- Sequences in V^* will be represented with Greek letters in lower case, i.e., α, β, γ, $\ldots$

Definition 1.2 *Given a grammar* $\mathcal{G}$*, a* single derivation *is represented as* $\beta \Rightarrow \gamma$*, such that* $\beta = \beta_1 A \beta_2$*,* $A \rightarrow \alpha \in P$*, and* $\gamma = \beta_1 \alpha \beta_2$*; then it is said that* α *is* reduced *to* A*. A derivation is called* rightmost *iff* $\beta_2 \in V_T{}^*$*, and can be written as* $\beta \underset{\mathrm{rm}}{\Rightarrow} \gamma$*, or* leftmost *iff* $\beta_1 \in V_T{}^*$*, and can be written as* $\beta \underset{\mathrm{lm}}{\Rightarrow} \gamma$*.*

A derivation $\beta \Rightarrow^* \delta$ *belongs to the reflexive, transitive closure of single derivations and, if all the single derivations involved are rightmost, it can be written as* $\beta \underset{\mathrm{rm}}{\Rightarrow}{}^* \delta$*. A sequence* α *is said to be* nullable *if* $\mathcal{G}$ *permits* $\alpha \underset{\mathrm{rm}}{\Rightarrow}{}^+ \varepsilon$*, then a rule* $A \rightarrow \alpha$ *would also be said nullable. A nonterminal is said to be* null *if it only produces the empty sequence.* □

A *reduction* is a derivation considered the other way. We shall usually consider single reductions, as a consequent, we shall usually employ the general term reduction to refer to them when there is no ambiguity. A terminal sequence is a *phrase* if it is reduced to one nonterminal.

Definition 1.3 *A grammar* $\mathcal{G}$ *is well-constructed (usually called* reduced*) iff for every* $A \in V_N$*,* $S \Rightarrow^* \alpha A \beta$ *and* $A \Rightarrow^+ x \in V_T{}^*$*.* □

Definition 1.4 *The* language $\mathcal{L}$ *generated by a grammar* $\mathcal{G}$ *is defined as*

$$\mathcal{L}(\mathcal{G}) = \{x | S \Rightarrow^* x \in V_T{}^*\}.$$

Those φ *such that* $S \Rightarrow^* \varphi$ *are called* sentential forms *and, if* $S \underset{\text{rm}}{\Rightarrow}^* \varphi$, rightmost *sentential forms. In* $\zeta\beta z$, β *is called a* handle *iff* $S \underset{\text{rm}}{\Rightarrow}^* \zeta A z \underset{\text{rm}}{\Rightarrow} \zeta\beta z$. □

Definition 1.5 *A grammar* $\mathcal{G}$ *is said to be* ambiguous *if there is some* $x \in \mathcal{L}$ *with at least two different rightmost derivations.* □

If $\mathcal{G}$ is unambiguous, then each rightmost sentential form has a unique handle.

As it is customary, to avoid the difficulties arising for parsing decisions on the extremes of forms, and thus to accept precisely the desired family of grammars, a grammar augmentation scheme is used. The idea is to impose some criteria to the actual (augmented) grammar to be used for parser generation and parsing, as not having the start symbol S in any rightpart or that $S \Rightarrow^+ S$ is impossible. In order to achieve this goal, we have found appropriate and general the following scheme.

Definition 1.6 *Given a grammar* $\mathcal{G} = (V_T, V_N, P, S)$, *the grammar* $\mathcal{G}' = (V_T{}', V_N{}', P', S')$ *is defined the augmented grammar for* $\mathcal{G}$, *such that*

$$\begin{aligned} V_T{}' &= \{\vdash, \dashv\} \cup V_T \\ V_N{}' &= \{S'\} \cup V_N \\ P' &= \{S' \rightarrow \vdash S \dashv\} \cup P \end{aligned}$$

where neither S', $\vdash$, *nor* $\dashv$ *are in* $V_N \cup V_T$. □

Some authors, e.g., [AU73a, WG84], use the augmentation scheme $S' \rightarrow S$ and others, e.g., [GJ90], use $S' \rightarrow S \dashv$. The above definition is in fact the augmentation scheme that is normally used in precedence methods, e.g., [AU73a]. It is symmetrical, which is interesting in general, but in particular to us, for we are to compare two approaches that work in different directions. It includes into the grammar the information necessary in practice of whether the beginning/end of the form has been reached. The elimination of $\vdash$ would force us to introduce a minor inelegancy in the algorithm at some bottom-of-stack decisions. And with respect to $\dashv$, its removal would in principle exclude some grammars (e.g. grammar of Figure 4.2 in page 42) of being considered in the family of LR(0) grammars, as is most commonly the case [GJ90, WG84], although some reputed authors [AU73a] do not share this opinion.

Another possibility is to put $\dashv^k$ $(k > 1)$ to the right of the sentence. Thus there will be always k symbols of lookahead, even when reading the last symbols of the input sequence to parse. Normally, this augmentation scheme should only have interest for practical reasons when implementing a parser using up to k lookahead symbols, as in parsing LR(k) grammars, but it will have no theoretical influence, as it is evident that just one $\dashv$ suffices to signal that the end of the input sequence has already been reached.

When not indicated otherwise in the context, the term *grammar* will be used to refer to a reduced context-free grammar. Grammars will normally be considered in their augmented form.

Convention 1.7 *Each rule in P' is assigned a different number i, $|P'| \geq i \geq 1$. Accordingly, it can be written $A \xrightarrow{i} \alpha$ or $A \overset{i}{\Longrightarrow} \alpha$. In particular, $S' \xrightarrow{1} \vdash S \dashv$.* □

Definition 1.8 *Prefix of length at most k of a sequence $x = X_1 \cdots X_m$:*

$$k : x = X_1 \cdots X_{\min\{k,m\}}$$

A prefix *(or* head*) is a prefix of some length.* □

Definition 1.9 *Suffix of length at most k of a sequence $x = X_m \cdots X_1$:*

$$x : k = X_{\min\{k,m\}} \cdots X_1$$

A suffix *(or* tail*) is a suffix of some length.* □

Definition 1.10

$$\mathrm{First}_k(\alpha) = \{k : x \mid \alpha \Rightarrow^* x\}.$$

$$\mathrm{Follow}_k(A) = \{k : x \dashv^k \mid S' \Rightarrow^* \alpha A x\}.$$

□

Chapter 2

Revision of Practical Parsers

Before presenting our method, this chapter will be devoted to review other parsing methods. Because of its great interest from the theoretical and practical viewpoints, parsing has attracted the attention of many theoretical computer science researchers as well as many computing practitioners. They have produce a large amount of experience, and have developed a large variety of parsing methods. It will be indeed out of the reach of this chapter to try to survey all these works. Instead, we shall concentrate on practical parsers for programming language processors, that is, especially those which are powerful in terms of grammars which can be accepted, efficient in terms of time and space, or which are actually used for parser generation within compiler construction tools.

2.1 Practical parsers for compiler construction

When planning the construction of a parser by hand or the construction of an automatic parser generator for compiler construction, the designer must consider several aspects in his choice of a parsing method.

A first obvious aspect is parsing power. The wider the spectrum of grammars that the parser is capable of analyze, the easier it is for the user to accommodate its language design to the grammar formalism. Nevertheless, during this process of grammar design, transformation, and adaptation, it is usually essential for the user to understand the parsing mechanism involved. In this sense, clarity and simplicity of the parsing method is important for a better and wider tool use.

A second aspect is efficiency, both at generation and parsing time, and from the space and time viewpoints. Of course, these aspects are relative to the computing power and resources of the target machine.

And a final aspect, but no less important, is easiness of integration with the other modules of the compiler construction tool, as lexical analysis, error processing and semantic processing.

To automate error processing is a difficult task, since in general it is dependent on the language design and on its usage. Moreover, error recovery techniques are usually dependent on the parsing method, thus the existence of well-founded and tested techniques already developed is another important aspect to consider.

Semantic processing usually consists of semantic actions directly written by the compiler designer in some programming language, or by computations automatically derived from the attribute grammar formalism. This is an important aspect for the parsing method, with relation to the ability to accept the insertion of semantic actions within the rules and to allow the required order of attribute computation.

In our survey of parsing methods we shall concentrate in those which can be considered more relevant from the above points of view. We shall also briefly mention the most relevant parser generators within compiler construction tools that use them.

2.1.1 Survey of practical methods

Almost all parser generators use the Backus-Naur Form (BNF) notation (or a slight variant of it) to specify context-free grammars (CFG). Some generators, particularly amongst those using top-down parsing, use Extended BNF (EBNF) notation, or a variant, by allowing regular expressions in the rightparts of productions. But this usually complicates parsing and do not necessarily simplify other aspects, as the semantic specification.

A context-free language (CFL) is a language for which a CFG exists. Although parsing algorithms for the general class of CFG have been found [Sak62, Gre64, Ung68, Döm68, Ear70, Lan74, Tom86], in general terms, they are considered not sufficiently efficient for practical usage, specially for programming language grammars.

A more practical, although more reduced, class of languages is that of deterministic context-free languages, that is, those languages for which a deterministic pushdown automaton exists. Many families of grammars can be used for the family of deterministic languages [AU73a]. Amongst all the shift-reduce grammars classes with bounded lookahead, LR is the largest class including all the others. In particular LR(1) includes all the other grammars when using a one-symbol lookahead, because in LR(1) all the necessary information in the stack plus lookahead is used for the parsing decision, while other methods use a limited part of this information. Furthermore, every LR(k) can be converted to an LR(1) grammar [MLS76], although the general transformation is usually impractical[1]. We should also mention the class of right parsable grammars, that is, the class of grammars for which there exists a deterministic pushdown automaton, which is a

[1]The transformation into bounded context and precedence grammars is studied in [Flo64, MLS76, Wyr86].

proper superclass of LR grammars.

But, as Aho & Ullman put it [AU73a, page 271], LR grammars are those which can be parsed *naturally* bottom-up and deterministically, i.e., the parse tree is built bottom-up as the input stream is read from left to right, while otherwise the method must ("*unnaturally*") resort to store a certain quantity (maybe large or even unbounded) of information about the input before being able to decide and produce the corresponding derivations.

It is also a well-known fact that the family of LR(k) grammars is larger than the LL(k) grammars, i.e., those which can be parsed top-down deterministically in a natural way. LL(1) in particular is a popular class of grammars, for which oneself can rather easily produce by hand a "recursive descent" parser. Unfortunately, this class of grammars is rather reduced. Nonetheless, some consider it enough and even adequate for a human to "parse", —and in fact practical top-down compiler generation tools do exist— and for having confidence about what is going on during parsing in practice when constructing by hand a compiler, in comparison with the rather obscure functioning of table-driven parsers. On the other hand, we can mention many cases where the possibility of using a rather large class of grammars is quite convenient. It is clear that, from the point of view of the grammar designer, it is rather cumbersome to struggle with the syntax difficulties of his parser generator, whereas he would prefer to concentrate on the language he is designing. Moreover, when adding attributes or actions to the grammar to construct a real compiler, it is important to have the flexibility that a rather large class of grammars offers.

LR, and LR(1) in particular, has thus received great attention from the theoretical and practical points of view. Much effort has been put to develop efficient LR(1) parsers. Unfortunately, the canonical LR method normally produces a huge number of automaton states. Thus a compromise between parsing power and parser size was finally found with the class of LALR(1) grammars, a rather difficultly characterizable class of grammars: those with an LR(0) automaton where the lookahead information derived from the corresponding LR(1) states is enough. Clearly, although quite powerful, LALR(1) is a proper subclass of LR(1), since as it is evident from the construction algorithm from an LR(1) parser by merging item sets with equal LR(0) section, part of the stack information that may be needed is lost in the process to reduce the LR(1) automaton size. It is not so widely known that LALR(1) is not a superclass of LL(1)[2] (they are incomparable), although every LL(1) grammar without nonterminals producing only the empty string is also LALR(1) [Cha87, Bea82]. Note also that every ε-free LL(1) is LR(0), thus also LALR(1). On the other hand, for $k = 2$ and greater, there are ε-free LL(k) grammars which are not LALR(k) [SSS90],

[2] The opposite was (erroneously) stated in [AJ74] and [Hor76]. Counterexamples are given in [Hei77] and [HIS78].

even though they all are LR(k), evidently.

Nonetheless, other methods with LR(1) power have been tried, quite complicated in general, notably Pager's [Pag77c], and also Seité's [Sei87]. The more recent Tomita's method [Tom86] uses a somehow limited-power automaton of the LR family to drive an essentially nondeterministic parser —thus resulting in increased parsing power— which is implemented in the form of a dynamic graph representing a number of currently-valid parallel stacks where their common sections are joined.

In following sections, we shall study in more detail these practical parsing methods, with special emphasis on LR, the most powerful class amongst them.

2.2 LR parsing essentials

LR theory is today well established, and it will be impossible for us to even mention all the relevant works done in this area. The reader is referred to reputed works in the form of books or articles devoted to any aspect of this subject, of which the bibliography at the end of this document may serve as a selection of the most relevant works. Here we shall content ourselves with merely reviewing the main ideas and to make some considerations that may be of interest in relation to the work presented in this document.

2.2.1 Definition of the problem

Bottom-up parsers build the parsing tree, through a sequence of single reductions, from the end nodes of the sentence towards the start symbol at the tree root. *Shift-reduce* parsing is a practical form of bottom-up in which, during a left-to-right scan of the input symbol sequence, the parser makes reductions as soon as the corresponding rightparts, or handles, have been explored.

When a sentence is shift-reduce parsed, a *stack* contains a prefix of the current rightmost sentential form. Terminal symbols are *shifted*, i.e., pushed, onto the stack until the stack contains the handle at its top. While the stack contains a handle, it is reduced and substituted by its corresponding production leftpart. Thus, the main question in designing a shift-reduce parser is how to decide in every case the next parsing action, i.e., whether to shift or to reduce, and, in the latter case, which rule to apply.

Proposition 2.1 *Given an unambiguous reduced grammar $\mathcal{G}$, if*

$$S' \underset{\text{rm}}{\Rightarrow}^* \zeta A z \underset{\text{rm}}{\Rightarrow} \zeta \alpha z = \tau y z,$$

the correct shift-reduce parsing action for current stack contents τ is to reduce using $A \rightarrow \alpha$ if $y = \varepsilon$, or to shift otherwise

Proof. If $y = \varepsilon$ then stack τ ends with handle α, which has to be reduced to A according to the derivation. Otherwise, since the handle is not yet on top of the stack, the correct action is to shift. □

In LR(k) grammars, as soon as the handle is read in a left-to-right scan of the sentence, its appropriate reduction can always be uniquely determined from the head of the sentential form —corresponding to the head of the sentence— already read and at most k more *lookahead* terminal symbols to the right of the handle. If there is no reduction to perform, the appropriate action is to shift the next terminal symbol into the stack. Obviously, for the $k = 0$ case, i.e., LR(0) grammars, the stack contents always suffices, without having to look ahead to take a parsing decision.

Accordingly, the name "LR(k)" is intended to indicate that, during a *L*eft-to-right scan, it can always be decided (from what has been already been read, plus k symbols of look-ahead) the correct sequence of reductions for the *R*ightmost derivation of the sentence.

Definition 2.2 *A grammar $\mathcal{G}$ is said to be LR(k), for some $k \geq 0$, iff*

$$\left.\begin{array}{l} S' \underset{\text{rm}}{\Rightarrow}^* \zeta A z \underset{\text{rm}}{\Rightarrow} \zeta \alpha z, \\ S' \underset{\text{rm}}{\Rightarrow}^* \zeta' A' z' \underset{\text{rm}}{\Rightarrow} \zeta \alpha y, \ \textit{and} \\ k : z = k : y \end{array}\right\} \ \textit{implies}\ \zeta' A' z' = \zeta A y.$$

□

In other words, if during parsing the stack contents is $\zeta\alpha$ and current (compatible) lookahead is w ($= k : z = k : y$ above), where α is a compatible handle, then it is unique, as well as the nonterminal A to which is has to be reduced. Otherwise, the only possibility is to shift. Obviously, if current lookahead is incompatible with the stack contents, an error can be detected.

LR(k) grammars are unambiguous, since the handle is always unique for every sentential form (i.e., when $z = y$ in the above definition).

A grammar is *LR* if it is LR(k) for some $k \geq 0$. Thus LR is the largest class of grammars whose sentences can be shift-reduce parsed using bounded lookahead.

For a better comprehension on how a stack recognizing automaton (be it direct or reverse) can be built from the grammar rules, Figure 2.1 will help to visualize the internal structure of the typical parse tree in the middle of shift-reduce parsing process of a sentence from a unambiguous grammar. It is quite easy to see that, during the rightmost derivation of a sentence z, a structure as shown is produced. A typical sequence for the first part of derivation $S \underset{\text{rm}}{\Rightarrow}^* \varphi w \underset{\text{rm}}{\Rightarrow}^* z$ is as follows: first $S \underset{\text{rm}}{\Rightarrow} \alpha_1 A_1 \beta_1$, then $\beta_1 \underset{\text{rm}}{\Rightarrow}^* x_1$, then $A_1 \underset{\text{rm}}{\Rightarrow} \alpha_2 A_2 \beta_2$, and so on, until $A_{m-1} \underset{\text{rm}}{\Rightarrow} \alpha_m A_m \beta_m$, then $\beta_m \underset{\text{rm}}{\Rightarrow}^* x_m$, then $A_m \underset{\text{rm}}{\Rightarrow} \alpha\beta$, and finally $\beta \underset{\text{rm}}{\Rightarrow}^* x$. As in Figure 3.3, let φ be the stack contents,

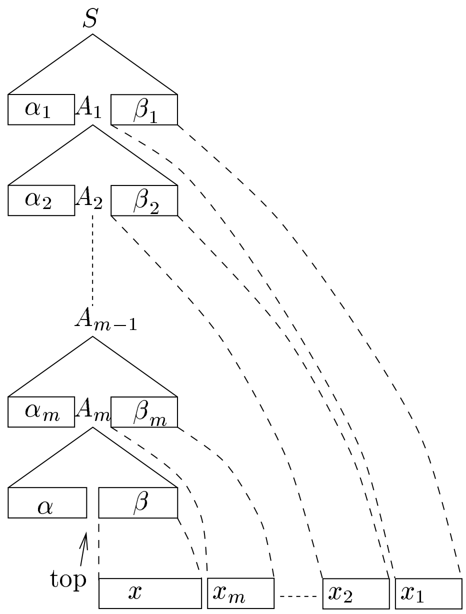


Figure 2.1: Internal structure of a typical bottom-up parsing tree.

$\varphi = \alpha_1 \cdots \alpha_m \alpha$ —i.e., the stack includes until this α in A_m's rightpart— and let $w = x x_m \cdots x_1$. In shift-reduce parsing, either the stack has reached the right end of a rightpart, and then reduces it, or else the next input symbol is shifted on top of the stack. In the latter case, $\beta \underset{\text{rm}}{\Rightarrow}^* x \neq \varepsilon$, and in the former case, $\beta = \varepsilon = x$ in the Figure. In the particular case when an empty rightpart is to be reduced we may think that handle ε already "is" in the stack top (in the Figure $\alpha\beta = \varepsilon = x$), and the top of the stack only goes until the right of α_m (actually to the last nonempty α_i); then ε is reduced to A_m, which is pushed onto the top of the stack.

2.2.2 Canonical LR

As we have seen, in shift-reduce parsing, the bottom-up parser uses some contextual information from current stack-plus-lookahead configuration to decide whether it has to shift or to reduce (and with what rule). Else the detection of a syntactic error is signalled in order to trigger the corresponding recovery action. In (full, direct) LR parsing, we want to use all the available information in current stack-plus-lookahead configuration to take such a parsing decision. In other words, if, according to the grammar under consideration, the parsing action is uniquely determined by the current configuration, then the action shall be identified by the parser. This implies that the grammar should be such that all the language sentences sharing any stack-plus-lookahead configuration at some moment during parsing have to share the same action. In order to use all the available information, the idea of D. Knuth was to build a bottom-to-top stack recognizer to carry such information in its states. We shall present here his method, although in manner slightly different form his paper [Knu65], where he introduces and characterizes LR(k) grammars and languages, shows how to build an LR finite-state automaton and how the LR parser operates as a pushdown automaton.

As we have seen, a legal parsing stack is composed of a sequence of heads of rightparts. Thus, a stack recognizer can be built by precisely considering how the current point of exploration in the stack corresponds to a point of exploration within the grammar rightparts. To indicate such a point, we shall use a *rightpart dot* as in $[A \rightarrow \alpha \bullet \beta]$ (where $A \rightarrow \alpha\beta$ is a grammar rule). Such dotted rules are conventionally called, following Knuth, *items*. In general, to an exploration point within a legal stack will correspond a nonempty set of items.

If at the end of the stack we have (amongst others) an item of the form $[A \rightarrow \alpha \bullet]$, it is clear that a legal possibility —according to the stack just read— is to reduce at this moment using rule $A \rightarrow \alpha$. If we want to use the lookahead information, items must predict which are the possible lookahead sequences compatible with this reduction at this point. Thus, for the general case of LR(k) parsing, $k \geq 0$, items will have the form $[A \rightarrow \alpha \bullet \beta, w]$, where $|w| \leq k$ (if we consider the input end followed by k end markers, i.e., $\dashv^k$, then $|w| = k$)[3]. Accordingly, at the stack top, an item such as $[A \rightarrow \alpha \bullet \beta, w]$ will indicate,

- when $\beta = \varepsilon$, reducing with rule $A \rightarrow \alpha$ only if current lookahead is w, and
- when $\beta = a\beta_0$, to shift next input symbol only if current lookahead is in $\mathrm{First}_k(\beta w)$[4].

In order to build such a recognizer, we may build a (nondeterministic) finite-state automaton (FA), such that each state has an associated item. Since the stack is recognized from the bottom, the automaton initial state has the associated item $[S' \rightarrow \vdash \bullet S \dashv]$. It remains indeed the question of how to build the transitions. Evidently, an item of the form $[A \rightarrow \alpha \bullet X\beta, w]$ will have an *X-transition* to item $[A \rightarrow \alpha X \bullet \beta, w]$. But if $X = B \in V_N$, we have to consider also the case where the stack contains a prefix of some μ, such that $B \underset{\mathrm{rm}}{\Rightarrow}^+ \mu$. For this purpose, we have to consider all the rightparts of B, and obviously

[3]Sometimes, it is practical to consider such items in groups, if they differ on the lookahead only, and thus an item group would have the form $[A \rightarrow \alpha \bullet \beta, \{w_1, \ldots, w_m\}]$.

[4]For LR(1) parsers, this condition reduces to the existence of an item of the form $[A \rightarrow \alpha \bullet a\beta_0, b]$, such that current lookahead is a. Moreover, this simpler condition can also be applied (at the cost of some delay in error detection) to LR(k) parsers, $k > 1$, if shifting is considered only after lookahead sequences leading to reductions have been checked.

to put the dot at the left end of the rightparts. We must also predict the legal lookahead sequences to be found *after* having read that rightpart, that is, for all rightparts of the form $B \rightarrow \gamma$, we must build items of the form $[B \rightarrow \cdot \gamma, x]$, for all possible $x \in \mathrm{First}_k(\beta w)$. Of course, the transitions from state of item $[A \rightarrow \alpha \cdot B\beta, w]$ to all these $[B \rightarrow \cdot \gamma, x]$ takes place without moving our current stack exploration point, hence they are ε-transitions.

Nondeterministic automata, as the one resulting from the above construction, are usually considered impractical for programming language parsing. Moreover, it is not evident from them what combinations of items are possible in practice, and thus whether there exists the possibility to arrive to the stack end with some combination of items containing a *conflict*, i.e., different parsing actions indicated by different items for the same lookahead. Evidently, such a conflict indicates that the grammar is not LR(k), and that, either more right-hand context is needed to single out the correct parsing action for the sentence under analysis, or (in case of ambiguous grammars) some sentences may actually have more than one possible rightmost parse, and thus different sequences of parsing actions.

In order to avoid such inconveniences, the classical transformation [HU79] of this (nondeterministic) FA to a deterministic finite-state automaton (DFA) can be applied. In practice, this DFA, which is called the Knuth *canonical* LR(k) automaton, is directly built. Its closure C_k, initial state q_0, and transition function d are shown, for the sake of reference, as follows:

$$C_k(q) = q \cup \{[D \rightarrow \cdot \varphi,\ k : yw] \mid [A \rightarrow \alpha \cdot \beta,\ w] \in q \text{ and } \beta \underset{\mathrm{rm}}{\Rightarrow}^* Dy \underset{\mathrm{rm}}{\Rightarrow} \varphi y\}$$

$$q_0 = C_k(\{[S' \rightarrow \vdash \cdot S \dashv,\ \varepsilon]\})$$

$$d(q, X) = C_k(\{[A \rightarrow \alpha X \cdot \beta, w] \mid [A \rightarrow \alpha \cdot X\beta, w] \in q\})$$

We can distinguish two subsets for state items: the *kernel* items, i.e., those in the above constructions for d and q_0 before computing C_k, which are characterized by not having the dot at the left end of the rightpart; and the non-kernel items, which are those produced by C_k (which correspond in the FA to all items accessed from ε-transitions). Since the latter can be obtained from the former, they are not usually stored in order to save space and time.

During parsing, the stack only progressively changes its topmost symbols, be it by pushing a shifted input symbol, or by reducing, and thus substituting several (or none) topmost symbols constituting the handle for the corresponding reducing-to nonterminal. Evidently, the rest of the stack remains unchanged, and thus it is a useless work to re-read it again for each parsing action. Accordingly, in practice, instead of storing the symbols themselves in the stack, they are replaced by the states to which they have led. Thus, after a reduction, a number of states equal to the length of the reducing handle is deleted from the stack top, and then we retrieve on the top the state from which to take a transition on the corresponding reducing-to nonterminal. In this way, LR parsing time is linear with the input length.

Parsing tables have two parts: the *action* part for shifts and reductions, and the *go-to* part for transitions to be taken on leftpart nonterminals immediately after a reduction has taken place. LR parsing tables in general are easily produced from the item-set contents of each state. Shifts and go-to's for items with the dot to the left of a terminal or nonterminal, respectively, and reductions for items whose dot is at the right end of the rightpart.

A relevant property of these parsers is that they always have a valid stack legally deriving the part of the input sequence read thus far, which in theory includes the lookahead. Accordingly, as early as the stack-plus-lookahead becomes illegal, an error might be detected. This is often called the "*immediate error detection*" property, what we consider is somehow misleading. Consider for instance a single error, such as a missing token at some point in the input. It is possible that after its actual position the input may still be a valid prefix of some sentence, and thus the parser will continue parsing, evidently up to a point where the input is no longer a valid prefix. Whether this second point has much relation with the actual error point depends on the grammar. Nonetheless, as we shall see, this early detection mechanism is usually loosened for practical reasons and, in general, for a syntactically incorrect input, the parser detects an error when it is unable to continue having a valid stack, preserving what is called the *correct-prefix* property. Note finally that the need of preserving a valid sequence of states in the stack represents a certain level of inflexibility for the possible error recovery mechanisms, in comparison with non-correct-prefix methods.

Lower bounds for the size of LR(k) parsers, $k \geq 0$, is shown in [Ukk83] to be exponential with grammar size. LR(k) parsers, for $k > 1$, lead to a combinatorial explosion of states and are generally considered impractical for programming language analysis. Even the canonical LR(1) automaton has been traditionally considered too large, and even more costly to build. But LR(0) is too limited for practical purposes. Thus research work has been mainly centered into trying to overcome the size problem of LR(1), which can be considered its most important disadvantage. Most efforts have gone into merging states, either during construction, after full construction, or once the parsing table has been built. In the first two cases, most techniques merge states by unioning the item sets from two or more states to form a new state. Fortunately, this parser frequently permits this kind of mergers without loosing its LR(1) parsing power, nor the correct-prefix property. In practice, however, a more drastic approach is taken, and mergers are done even at the cost of some parsing power, leading to LALR(1) parsers. P. Purdom argues in [Pur74], from experimental size analysis on LALR(1) parsers, that in practice grammars for programming languages have LR parsers that grow only linearly. In the following sections we shall review some of the main efforts done in these both main directions in order to reduce parser size.

2.2.3 Size compression

The problem of reducing the size of LR parsers has been the object of intense research from many points of view. We shall merely present in this and subsequent sections a small summary of the most relevant practical results. In this section we consider in particular those optimizations using a (mostly) standard method of automaton construction, and at the same time preserving the original parsing power.

One of the approaches that has received considerable research attention is the (virtual) "elimination" of *unit* reductions, i.e., reductions using rules with a single-symbol rightpart. Usually, these kind of rules are not semantically significant. They are very frequent in programming language grammars: in expression syntax when implementing operator precedence in the form of rule hierarchy, and when grouping several independent language structures under a more general concept. Essentially, these techniques consist in avoiding to perform a *chain* of such reductions, and effectively substituting the corresponding sequence of parser steps by a single step with the same final result. In general terms, it is very difficult to eliminate all unit reductions and preserve at the same time the correct-prefix property, and some compromise is settled. These techniques have the additional advantage of speeding up the parser, and in fact it is precisely this aspect which is sometimes seeked to optimize —then, some approaches may produce even larger tables (a straightforward one is grammar rewriting to eliminate unit rules). The reader is referred for the details of the different techniques to [SSS90, Cha87, AU73a, AEH73, AU73b, Pag74, Dem75, Pag77a, KSS79, SS80, Tok81, Sch84, Hei85].

In a different approach, J. Witaszek [Wit88] proposes to reduce the LR automaton size by postponing error checking. Since finding the optimum automaton transformation is very costly, a heuristic is proposed there.

LR parsing tables for practical programming language grammars are quite sparse in general. This single fact allows to apply standard sparse matrix compression techniques. However, LR tables possess special properties that made them prone to special forms of compression [AU73a, WG84, Cha87, SSS90]. Usually, several action-part rows are identical and can be merged, if we consider compression of the action and go-to parts separately, as it is usual. Some table entries are never consulted —they are called *inessential* or *don't care* entries. This is the case for all empty entries in the go-to part. This allows to merge similar go-to columns when they differ only in those entries where the other column has empty entries. In the action part, most of the entries are in effect error entries, i.e., they might be consulted, and an error might be detected in that moment. But some of the action-part empty entries may actually be found to be inessential entries [AJU72, AU73a, AU73b, SS82, SSS90]. This permits additional action rows to be merged, when they differ only in places where the other row has an inessential entry. An optimization of this idea is to put in a separate *failure* matrix [Jol74, WG84] the information of whether an entry is an error, a true entry, or an inessential entry; this table filters access during parsing to the original table. In this way, all the original table entries may be considered inessential, which opens further possibilities of row merger (optimal merger requires incompatibility-graph analysis). In [Pag70], all empty entries are considered inessential, but the resulting parser is non-correct-prefix, thus symbols are also saved in the stack in order to perform handle verifications.

An interesting property of LR parsers is that reductions which are correct at a given state for some lookahead, can be also safely done for other (illegal) lookaheads [DeR69, AU77], without affecting to the parser's validity and correct-prefix property[5].

Actually, some states can only perform reductions with the same rule, in principle for legal lookaheads only; they are called *LR(0) reduce states*. Considering the above property, we may consider the possibility of always reducing in these single-reduce states, for whatever the lookahead. Then, these states can be removed from the table if we add a new kind of action, *shift-reduce* r, which replaces transitions (shifts and go-to's) in states originally having transitions to these reduce states, and which obviously represents the combined action of transition and r-reduction [Cha87, Gro84].

Some possibilities of size optimization depend on the actual data structures used to implement the parsing tables [Cha87, AU73a, AHS86]. A simple solution, at the cost of parsing speed, is to replace action rows by search lists where empty entries are not considered. These lists have a last *default-action* entry, which typically indicate an error or a default reduction (using again the above property) to replace the most frequent reduction in the row. Numbers identifying states can be in general renumbered to avoid indirection when accessing their corresponding search lists. In the go-to part, search lists replace columns, and the default action is the most frequent action in the column. Other techniques try to avoid the cost of list searching with different approaches [AHS86, KM89].

When implementing parsing tables in general, a concern is speed. An immediate solution, usually at the cost of an increase in size, is to directly produce code instead of tables [AJ74, AU77]. In this line, it has been proposed a dual to recursive descent named *recursive ascent* [Rob88, KA88], as a form of implementing LR parsers, where recursive procedures mimic the evolution of the LR parsing stack. Although there is a potential speed-up, the resulting automata have to be mechanically generated and the resulting code is far from intuitive, in comparison with recursive descent.

Fortunately, most compression techniques for full LR parsers are equally

[5]But they affect to the immediate error detection property, what may be important depending on the error recovery technique. An approach to overcome this is buffering reductions [FL88], which are validated when a shift is made.

applicable to other parsers in the LR family as well. However, the above techniques do not solve the complexity problem of the full-LR construction process itself.

Another possibility is to simplify grammars and thus reduce parser size by accepting ambiguous grammars [AJU75, AU77, ASU86, KNPS88]. An implementation of this idea is found in the parser generator yacc (see next section for more details).

In other approaches to the size problem of LR parsers, smaller full LR parsers are directly built using some special mechanism, or non-full LR parsers are used, and then full LR parsing power is obtained by additional mechanisms. Some of the most relevant techniques following these approaches will be reviewed in Section 2.3.

2.2.4 LALR(1)

One of the first approaches to the size problem of full LR(1) parsers was SLR(1) [DeR69, DeR71]. This method uses the LR(0) automaton, and for items of the form $[A \to \alpha \bullet]$ decides a reduction when current lookahead is in $\mathrm{Follow}(A)$. In spite of the fact that $\mathrm{Follow}(A)$ does not take into consideration current-stack context information, SLR(1) parsers are surprisingly powerful. Unfortunately, practical programming language grammars usually produce a number of conflicts in SLR(1) parsers, and thus these are generally considered impractical for them.

LALR(1) [DeR69] is also based on the LR(0) automaton, and appeared then as a effort to relate reduction lookahead to the stack context to improve parsing power further from SLR(1). In fact, LR(1) automaton state item sets are equal to the LR(0) states', if we ignore the lookahead in LR(1) items, i.e., if we only consider their *core* or LR(0) parts. Moreover, the stack language that both accept is indeed the same. Thus, to each LR(0) state corresponds in general several LR(1) states which obviously carry significant stack context information. Accordingly, the idea of LALR(1) is to exploit this information, by means of merging these LR(1) states sharing the same core, which leads to a new automaton with LR(1) items "within" LR(0) states. Of course, in the process some contextual information gets lost in general. But, if as a consequence of such merging process no conflict is introduced, then we get a working parser. Actually, these conflicts can only be of the type reduce-reduce, since shift-reduce conflicts would also be already present in the original LR(1) parser, and thus the grammar were not LR(1), anyway. Since they are not very frequent, LALR(1) parsers have been finally found a reasonable compromise between complexity and parsing power. But when these conflicts appear, they are difficult to trace, and it is unfortunate that full LR(1) parsers usage is not extended, for they would allow what we feel is the most powerful, and at the same time most natural, parsing mechanism for programming languages.

The described method of construction of LALR(1), i.e., to completely build the full LR(1) parser, and then to merge states with the same core, is normally discarded because of the state explosion of full LR(1). The possibility of state merging "on the fly", i.e., during the process constructing states, forces to re-build portions of the parser or to propagate lookahead symbols from the merged states [AEH73]. Efficient algorithms for the construction of LALR(1) parsers have been developed, as those shown in [AU77, KM81, DP82, PCC85, Ive87, BL89]. Most are based on initially building the LR(0) states, and to reconstruct from it the LALR(1) automaton by computing the item lookahead sets. Since their invention, LALR(1) parser generation became quite popular, in part probably because of the wide availability of practical implementations, beginning with yacc, which served of inspiration to many other LALR(1) parser generators. We shall survey here some of the most relevant ones.

YACC

The parser generator yacc [Joh75, LMB92] was originally developed by S.C. Johnson in the AT&T Bell Laboratories (USA) as a utility of the operating system Unix, in conjunction with the compatible lexical analyzer generator lex.

The algorithm [AU77] used in yacc initially builds the LR(0) states and items, which are then augmented with the LALR(1) lookahead symbols. This is done by means of a process of *generation*, from items of the form $[A{\rightarrow}\alpha{\bullet}B\beta]$, of lookahead symbols in $\mathrm{First}(\beta)-\{\varepsilon\}$ into items resulting from closure; and a subsequent process of iterative propagation of lookahead symbols through items where $\beta{\Rightarrow}^*\varepsilon$ into items resulting from closure and transition.

A user-controlled disambiguation mechanism is incorporated to accept ambiguous —hence non-LALR(1)— grammars. However, when using this mechanism, parser correctness becomes in part under the user's responsibility. The user probably needs to know whether the conflict is due to an ambiguity, and not to a non-LALR(1)-ness. For instance, an LR(1) grammar is equally treated as an ambiguous one, and this is not evident to the user, since the tool cannot help on this.

In order to indicate the conflict-resolution choices, every terminal symbol or *token* may have been specified some level of precedence and an associativity (left, right or none). By default, in case of reduce-reduce conflict, the parser reduces the rule specified higher in the text. Every rule can also have a precedence and an associativity, which can be explicitly specified or else is taken from the rightmost token in the rule with a precedence assigned. Shift-reduce conflicts are resolved according to the relative precedences of the terminal to shift and the rule to reduce and, if they are equal, then associativity is used, and when this mechanism cannot be applied, by default a shift is generated.

The generated parser is table-driven (with some compression) in C. Semantic actions in the form of C statements can be inserted at the end or in any other point in a rightpart. In the latter case, since semantic actions are only invoked at the moment of reductions, the system inserts in the rightpart, at the point of the semantic action, a new nonterminal producing only the null sequence, whose reduction triggers the semantic action. Such a transformation may render the grammar unparsable. The system allows a straightforward form of synthesized attributes to be used with C code.

Error recovery must be handled in its most part by the user, who may insert at convenient places within rightparts a special `error` token and the corresponding C code. When a syntactic error is found, some stack is discarded until finding a place where the `error` token can be shifted. Then *resynchronization* takes place according to symbols following the token in the rightpart.

Yacc is probably the most widely known compiler construction tool. Many other implementations for different systems and languages have been made. We may cite some notable free implementations as GNU bison (which includes some practical enhancements), BSD unix byacc, and versions for languages other than C, as bison++ for C++, ayacc for Ada, or mlyacc for Standard ML, as well as commercial implementations as MK yacc or Yacc++ (which accepts EBNF rightparts and uses C++). Yacc has also been modified to do backtrack [Mer93], and other parser generators are also based on it, at least at the user interface level, as it is shown below.

Rie

Rie [SIN95] is a free compiler generator developed by a group leadered by M. Sassa (Tokyo Institute of Technology, Japan). It has been written as an extension of bison to accept ECLR-attributed grammars [SIN87], a variant of LR-attributed grammars where optimization of attribute evaluation is achieved by introducing a kind of equivalence relation amongst inherited attributes. As in LR-attributed grammars [JM80], attributes can be evaluated in a single pass during LR parsing, without making a parse tree, at the moment of LR state transitions or reductions.

Eli

The Eli system [GHL+92, KPJ98] is a free compiler construction tool developed and maintained by groups leadered by W. Waite (University of Colorado at Boulder, USA), U. Kastens (University of Paderborn, Germany), and A. Sloane (Macquarie University, Sydney, Australia). It presents the originality that its overall functioning is controlled by an expert system, called Odin [CO90]. Other principal components of the system are the lexical analyzer GLA [Heu86], the LALR(1) parser generator PGS [GK88, KM89] and the attribute evaluator GAG or LIGA [KHZ82]. GAG produces an internal grammar specification with semantic actions attached to productions, which is input to the parser generator PGS, whose produced parsers build the parse tree and then attribute computation takes place.

In order to accept non-LR(1) grammars, a grammar and tree rewriting algorithm called tris has been developed and integrated into the Eli system. The implementation described in [HHM92] builds a canonical LR(1) automaton, and tries to transform the grammar into an LR(1) one within a fixed number of steps, each time processing a conflict. If the resulting grammar is not LALR(1) (the input needed by the PGS parser), then the algorithm gives up. If the transformation succeeds, at parsing time the grammar transformation is unfolded in order to make the process transparent to the user. According to the cited paper, the implementation is considered inefficient for large grammars, and the intent of building a new implementation directly based on LALR(1) is announced.

Cocktail

Cocktail is a tool set developed mainly by J. Grosch at the GMD National Research Center for Information Technology at Karlsruhe, Germany. It is originally freely available, and there exists a commercial version.

The freely available version main modules are: the scanner generator rex, the parser generators lalr and ell[6], the syntax tree generator ast, the attribute-evaluator generator ag, and the syntax tree transformer puma. Produced code is C or Modula-2.

The parser generator lalr [Gro90] directly implements LALR(1) parsing tables in the form of target programming language code, obtaining reported speeds about two or three times faster than (table-driven) yacc, with an increase of about 25 to 40 percent in size. The user interface is very similar to yacc. However, lalr also allows extended BNF, includes automatic error reporting, recovery and repair, and in the case of conflicts a derivation tree is displayed instead of states and items.

The commercial version contains the parser generator Lark instead of lalr. According to the available description, Lark generates table-driven parsers for LALR(1) and LR(1) grammars, semantic predicates can control parsing by conditions, and backtracking parsing with unlimited lookahead is supported. Produced code is augmented with C++, and in part Ada or Eiffel.

[6] A LL(1) parser generator [Gro89].

2.3 Space-efficient LR parsers

We shall describe now several particularly interesting methods for producing space-efficient LR parsers.

2.3.1 Pager's methods

D. Pager (University of Hawaii, USA) published in the 70's some papers on developing practical methods to construct full direct LR(k) parsers as small and efficiently as possible. His main result was reproduced in the 80's by S. Heilbrunner (University of the Federal Armed Forces Munich, Germany) within the general theoretical framework of item grammars and parsing automata. In this context, in the 90's W.-J. Park (Korea Advanced Institute of Science and Technology, South Korea) et al. tackle again the problem of building a minimal LR(k) parser.

We have already cited [Pag70]. One of Pager's main contributions was [Pag77b], where he developed a "progressive" method whereby he constructed in the first instance an LR(0) automaton, then the LALR(1) automaton, and finally, through state splitting in case of conflict, obtained if necessary a full LR(1) automaton. For that purpose, he makes use of the notion of "lane" of items, which connects items related by closure and transition, and which permits to follow the propagation of item lookahead symbols. Unfortunately, the algorithms he describes are of considerable complexity [GJ90], and to our knowledge, they have not been implemented in practical parser generators.

Pager also developed an alternative method [Pag77c], with a different approach, which is considered his main contribution, and thus it is the method that we are going to describe here.

Compatibility criteria for state merger

The main idea is to start building a full LR(1) parser, and to merge "on the fly" newly created states with pre-existing states in order to avoid the recognized unacceptable computational complexity of directly building a canonical LR(1) parser and then to check for state compatibility in order to perform state mergers.

Pager considers state items (which he calls *configurations*), grouped in sets with the same LR(0) part or *core*. These groups are called *config-groups*. Following Pager's notation, config-group numbered r for state S is noted $CG_{r,S}$, and is represented by their common core plus the union set of their lookahead symbols, U_r. Two compatibility criteria are introduced: *weak* and *strong* compatibility, which are applied on states with the same core. The former criterion implies the latter on LR(1) grammars, and both can be applied on kernel (*nucleus*) items only. Merger of course means that a new state S'' replaces two previous states S and S', containing the same core and the union of their lookahead sets, i.e., $U''_r = U_r \cup U'_r$.

In the weak compatibility criterion, two states S and S' are compatible if, for all different i, j for their config-groups, at least one of the following is true:

(a) $U_i \cap U'_j = \emptyset$ and $U'_i \cap U_j = \emptyset$, or

(b) $U_i \cap U_j \neq \emptyset$, or

(c) $U'_i \cap U'_j \neq \emptyset$.

The rationale for this criterion is that if the "underlying" LR(1) parser has no conflict, then it is impossible to introduce a conflict by merging two weak compatible states. First of all, note that merger cannot introduce a shift-reduce conflict, for that would imply this same conflict in some of the states to merge. Thus, only reduce-reduce conflicts can be introduced. Conditions (b) and (c) —which are in fact the same condition, applied to both states— guarantee that there cannot be a conflict for the new config-group pair $CG_{i,S''}$-$CG_{j,S''}$, since in some pre-merger state there is no conflict even though they share some lookahead. If none of both conditions hold, then there is in principle the possibility that a merger-introduced common lookahead for pair $CG_{i,S''}$-$CG_{j,S''}$ would produce a conflict in this or some successor state. This eventuality is ruled out by condition (a).

A drawback of weak compatibility is that for some grammars it may produce automata which are larger than those produced by genuine SLR(1) or LALR(1) parser generators, or his own above described [Pag77b]. In order to avoid this, the strong criterion is suggested.

For the strong compatibility criterion, Pager considers the rule right-part sections to the right of the config-group (i.e., item) dots, which he calls *scanned strings*, in order to define the right-hand context languages (sets of *strong rightmost descendants*) which can be accepted by the LR(1) automaton —because they are produced by scanned strings— leading to a reduction on precisely the corresponding config-group lookahead sets. Two scanned strings *share a descendant* if they have some common strong right-most descendants whose last production is different.

Accordingly, in the strong compatibility criterion both weak compatibility conditions (b) and (c) are substituted for the following condition:

(b') The scanned strings for $CG_{i,S}$ and $CG_{j,S}$ do not share a descendant.

If (b') does not hold, then there will be a conflict in some state "originating" from config-groups i and j for their shared lookahead. Accordingly, Pager shows that this is the widest possible criterium to be employed with his algorithm[7]. Section 5.6.2 shows from page 131 the LR(1) non-LALR(1)

[7] This may seem to imply that one would always get a minimal-state automaton, as it is (erroneously) stated in [Hei81] for a proposed alternative, nondeterministic algorithm. See the ongoing discussion and [PLC93] to see that this is not the case.

grammar (and the corresponding Pager's and discriminating-reverse parsing tables) that is used in his paper to illustrate weak compatibility.

In his original paper [Pag77c], Pager proposes a recursive procedure in order to test condition (b'), although in [Hei81] it is proposed to compute such a relation as the closure of a specific relation μ on pairs of LR(0) items.

One problem after merging states is that in principle successor states are no longer valid, thus leading to the possibility of iteratively re-building portions (which may be large) of the automaton, incurring on inefficiency, which is what it was intended to avoid in the first place. In order to avoid this, Pager devises a *context propagation procedure*[8]. In it, lanes are used to compute how new merged lookahead symbols will propagate, and to build virtual successor states, whose compatibility with the original successor guarantees that the successor can be updated with the above computed propagated lookahead symbols. Unfortunately, such a compatibility is not guaranteed[9], and in theory it may happen that a true construction of a new successor is otherwise necessary. (Pager mentions that he never found the need for such constructions in practical tests.)

In his algorithm [Pag77c], Pager merges current state with the *first* already constructed state which is found to be compatible (except during context propagation from a merged states, what may lead to further mergers). Although not commented in the paper, the resulting automaton —and, what is more important, its size—, for both the weak and strong compatibility criteria, might depend on the order in which candidate states are checked for compatibility. As we have seen, both criteria share condition (a). If state S is incompatible with state T, and S is merged with another state S' compatible with T, the new resulting state S'' may "inherit" from S incompatibility with T. See, in effect, that if (a) does not hold, then there is some pair of config-groups from S and T sharing some lookahead. Since a merger simply adds new lookahead symbols to S, the corresponding config-group of S'' will still share the same lookahead sets with that of T. In order to illustrate this, consider states A, B, C, and D, such that they share the same core, but pairs (A, D), (B, C), and (C, D) are incompatible, while the remaining pairs —i.e., (A, B), (A, C), and (B, D)— are compatible[10]. It is thus possible to merge AC and then BD —and their corresponding successors, of course. But if AB is merged in the first place, then it is impossible to proceed with more mergers, because AB inherits the incompatibilities with

[8] Which, contrary to what is stated in the paper, apparently does not work for weak compatibility [Rei98].

[9] The actual problem with an X-successor is that it may already have suffered a merger which may have rendered itself weak-incompatible with an X-successor of the new merged state. In fact, Heilbrunner [Hei81] does not actually build transitions until all the final automaton states have been built.

[10] This may happen, for instance, if the corresponding lookahead sets for the corresponding couples of config-groups are: $(\{a\}, \{b\})$, $(\{c\}, \{d\})$, $(\{a\}, \{c\})$, and $(\{b\}, \{a\})$, for A, B, C, and D, respectively.

C and D, and these are themselves incompatible. Thus, in this second case more states are obtained, which typically provokes a multiplication of part of their successors. This example can be enlarged for more states giving poorer results. Unfortunately, it seems quite difficult to avoid that pseudo-random suboptimality and to retain the idea of merging on the fly. Even if we have the whole canonical LR(1) machine already constructed, in the general case it seems in principle of intractable computational complexity to find the best mergers in order to produce an optimal result. In effect, this problem has been studied in [PLC93] for the general LR(k) case, where a criterion is applied *after* completely building the an LR(k) automaton, but to find an efficient algorithm for even a suboptimal solution is left for future research.

Pager claims in his paper that the generalization for the general LR(k) case is straightforward. But, as N.P. Chapman shows [Cha87], for $k > 1$ it is necessary to consider shift-reduce conflicts also. Chapman cites [Sch] for that generalization, which shows that it is no longer sufficient to use kernel items only.

Pager also claims in the paper that, according to his practical results, the size of his parsers and the number of configurations evaluated are of the order of magnitude of the corresponding ones for LR(0) parsers.

LR

The previously described parser was implemented by C. Wetherell and A. Shannon at the Lawrence Livermore Laboratory (California, USA), in a parser generator called LR [WS81], using the weak compatibility criterion [FL88]. It was written in ANSI standard FORTRAN 66 for portability, what conditioned the implementations and limited the character set of its BNF specifications. No error recovery is provided. Its authors report that in practice the number of states is about twice the number of productions in the grammar, and that generation time is at least linear with grammar size. No table compression is performed. LR originated from a precedence-based generator which was later modified by DeRemer based on SLR(k) [DeR71]. It has been used for commercial developments and for the generation of Ada and PL/I parsers.

It may surprise that neither weak nor strong compatibility have been used (more) in parser generators, in spite of its theoretical advantages. We speculate that it might be due to the fact that weak compatibility automata do not guarantee size results obtained with LALR(1) generators, and that it is felt that logical and computational complexities associated with strong compatibility do not compensate for the gain obtained in parsing power. In the same vein, S. Sippu and E. Soisalon-Soininen, in their monograph on LR(k) and LL(k) parsing [SSS90], cite Pager's work and the formalization by S. Heilbrunner [Hei81] only as a last bibliographic note, even though considerable space is devoted to the construction, implementation and optimization of practical LR(1) (full or not) parsers.

2.3.2 Seité's method

We cite here specially the work of B. Seité [Sei87, Sei86] because, within the proposal of a method to produce small and powerful parsers, some kinds of top-to-bottom automata are considered, amongst other notions. Actually, Seité's purpose is to build parsers for (a subset of) LRR grammars[11] [ČC73], which can be defined as follows.

Definition 2.3 *A grammar $\mathcal{G}$ is said to be LRR if there is a regular partition δ of ${V_T}^*$ such that*

$$\left.\begin{array}{l} S' \underset{\mathrm{rm}}{\Rightarrow}^* \zeta A z \underset{\mathrm{rm}}{\Rightarrow} \zeta\alpha z, \\ S' \underset{\mathrm{rm}}{\Rightarrow}^* \zeta' A' z' \underset{\mathrm{rm}}{\Rightarrow} \zeta\alpha y, \ \textit{and} \\ z \equiv y \pmod{\delta} \end{array}\right\} \ \textit{implies}\ \zeta' A' z' = \zeta A y.$$

□

In the paper, it is considered that to be limited to the grammar class accepted by yacc is arduous, results often in an unintelligible semantic representation, is dangerous as one may obtain the grammar of a super-language, and is reserved for specialists. However, it is recognized that building LR(k) parsers for $k \geq 1$ is in principle impractical, and that merging compatible states and the splitting of conflicting states are two important contributions by Pager, as we have seen. Moreover, other methods to recognize subsets of LRR grammars, as XLR [Bak81], RLR [Bou84], and LAR [BS86, BS90], which also build an LR(0) automaton and use regular covers for the right-hand context, fail to accept simple LR(1) or LR(2) grammars, which is attributed to ignoring information on the stack. In order to overcome this drawback, it is proposed to build, for each conflicting LR(0) state, a specific automaton to explore the corresponding right-hand context, such that, when it cannot resolve, one or more case-specific automata that begin reading some stack prefix from its bottom take over to try to resolve. The resulting parsing power is, by construction, at least LALR(1), with acceptance of some LR(k), $k \geq 1$, grammars, as well as some non-LR(k) grammars within the LRR family. The method can be considered also as a supplement to tools like yacc.

The construction of an LR0(X) automaton is presented. This automaton is the partial reverse of the LR(0) automaton, accepting the reverse stack (suffix) language with the symbol X at the top, and permitting to determine which item sets exist for every LR(0) states with incoming X-transitions. In the construction of this automaton, all the LR(0) items with X to the

[11] The set of LRR languages strictly includes the set of deterministic languages and is strictly included in the set of unambiguous languages.

left of the dot are considered in the initial state, and each one is given an index number. The construction proceeds until the bottom of the stack is reached, or alternatively, stops when in a state the items and the indexes are *equidistributed*, i.e., all the items have the same set of indexes. A slight variant of this automaton, named Aut0pref (for "prefix") will be used to help in the LRR parsing technique. It is also shown how to build a reverse LR(0) recognizer for a given LR(0) item.

Some results are also presented on LR(k) parsing, such as its characterization by using the extension of the above idea to LR$k(X)$, and by showing how to compute the k-lookaheads of LR(k) states on the LR(0) automaton through the use of products of matrices on the LR(0) items in order to solve LR(0) conflicts. The item k-lookaheads are progressively computed as the construction advances, which, in this case, may terminate when in a state the collection is equidistributed and the lookaheads are *saturated*. Finally, the construction of the Aut$k$pref automaton is shown and an example is given of its use to analyze conflicts by building a top-to-bottom automaton starting from the items in conflict.

2.3.3 Tomita's method

We briefly describe here, for the sake of completeness, the efficient and practical parsing method [Tom86, Tom87] developed by M. Tomita (Carnegie-Mellon University, USA), also known as *generalized LR*, GLR, to be used with ambiguous grammars, with application to natural language parsing.

We may view Tomita's algorithm mainly as a form of controlling the combinatorial explosion of the inherent nondeterminism of parsing sentences from ambiguous grammars [Lan74], with the help of a parser of the (direct) LR family, e.g., SLR(1). Evidently, if the grammar is ambiguous, or simply non-SLR(1), the parsing table contains conflicting (i.e., nondeterministic) entries. If we nonetheless use this table, in principle we should multiply our stacks each time the parser consults a nondeterministic table entry. Tomita devises a practical, near linear method for sharing common stack prefixes, and collapsing stack tops into a single stack top when two or more stacks arrive to a same state. The result is called a "graph-structured-stack". The method works in a breadth-first manner, such that parsing progresses simultaneously on the input for the different tops, pruning the lines that cannot shift, and replicating the stacks tops for all the possible reductions before shifting. In the end, the different parses of the sentence are produced. It is intended to work interactively, and compares very favorably with the generalized parser by Earley [Ear70] on the typical context of natural language sentences (which are very small for programming language standards).

Unfortunately, this approach seems of limited applicability to the field of programming language processing, where in principle ambiguous grammars are unacceptable. However, it is a small, efficient and general method that could be used also for unambiguous grammars: although the method cannot beforehand tell the user whether the grammar is ambiguous —which is well-known to be undecidable [AU73a]—, some powerful test could be applied to verify whether it belongs to some large family of unambiguous grammars, as full LR, for instance [Knu65, SSS90]; but then we could only work safely within this class of grammars. Even so, several temporarily different parallel parses during an important part of the analysis should normally be unacceptable from the point of view of semantic processing.

2.4 Other parsers

We shall briefly survey in this section other practical classes of parsers: LL, LC and the family of bounded methods. As we know, full LR grammars include all of the grammars acceptable by these parsers.

2.4.1 LL parsers

Top-down parsers build the parsing tree, by a sequence of single derivations, from the start symbol at the tree root towards the end nodes of the sentence. We shall consider here the practical approach, dual of shift-reduce parsing, in which the parser performs the derivations while scanning the input symbol sequence from left to right —i.e., the parser performs a *leftmost* derivation. In order to do this, it maintains a stack of predicted terminals and nonterminals, in reverse order. Topmost terminals are verified to coincide with the next input symbol, and then popped off the stack. Topmost nonterminals are substituted, according to the grammar, by one of their rightparts (in reverse order). How to decide, from the context information available, which rightpart to apply is the main question in designing these parsers. The most powerful of them with bounded lookahead is LL(k) [LS68], where all the information from the input read thus far is used. It can be defined as follows (we consider extended by analogy our definitions for rightmost derivations to leftmost ones).

Definition 2.4 *A grammar $\mathcal{G}$ is said to be LL(k), for some $k \geq 0$, iff*

$$\left.\begin{array}{l} S' \underset{\rm lm}{\Rightarrow}^* zA\zeta \underset{\rm lm}{\Rightarrow} z\alpha\zeta \Rightarrow^* zy, \\ S' \underset{\rm lm}{\Rightarrow}^* zA\zeta \underset{\rm lm}{\Rightarrow} z\alpha'\zeta \Rightarrow^* zx, \ \textit{and} \\ k : y = k : x \end{array}\right\} \ \textit{implies}\ \alpha = \alpha'.$$

□

In the above definition, $A\zeta$ is our reverse stack of predicted symbols, containing A on its top. As we see, for any stack configuration, the input read so far, z, plus k lookahead symbols, always suffices to decide which is the rightpart to use for A. That is, the above definition is equivalent to

$$\text{First}_k(\alpha\zeta) \cap \text{First}_k(\alpha'\zeta) = \emptyset,$$

for all ζ such that $S' \underset{\text{lm}}{\Rightarrow}^* zA\zeta$, and all pairs α, α' of A rightparts. The reader is referred to [SSS90] for the theory on LL(k) parsing. An important result is that LL(k) grammars are properly included in LR(k). This is easy to see intuitively, since in general LL parsers have to decide before reading the phrase of some nonterminal, while LR parsers may delay this decision until the whole phrase has been read, and thus have at their disposal possibly essential context information that is out of the reach of the LL parser.

There also exist the duals of SLR(k) and LALR(k), SLL(k) (called strong-LL) [RS70] and LALL(k) [SSS90]. For instance, in SLL(k), the current context ζ is not considered, that is, we put a strong(er) condition[12]:

$$\text{First}_k(\alpha\text{Follow}_k(A)) \cap \text{First}_k(\alpha'\text{Follow}_k(A)) = \emptyset.$$

Typically, $k = 1$ in practice. SLL(1) is implemented in tabular form or, more frequently, in the form of a recursive descent parser, where each procedure corresponds to a nonterminal, resulting in intuitive and simple parsers [Woo70, DM81, SSS90, Wir96, Ter97]. The classes of SLL(1) and LL(1) grammars coincide; however, LL(1) parsers have the immediate error detection property, whereas SLL(1) ones only have the correct-prefix property [GJ90].

Coco/R

Coco/R [Mös91] is a free tool by H. Mössenböck (Johannes Kepler University of Linz, Austria). It evolved from two predecessors: the scanner generator Alex and the parser generator Coco [RM89]. Its design puts the emphasis on simplicity and efficiency over parsing power. Accordingly, the parser generator produces recursive descent code from an EBNF description including the lexical description and offers help to integrate attributes into semantic actions. The user may specify *synchronization points* to seek in case of syntactic error in order to continue parsing, and *weak symbols*, which are known to be often mistyped or missing, and thus the parser accepts to continue from its successors. Coco/R uses Oberon, and there is another version for C. A similar tool for Modula-2 named Coco-2 also emerged [DP90].

PCCTS

In his PhD thesis [Par93], T.J. Parr reproduces the following result by B.M. Brosgol [Bro74]: if different null nonterminals are inserted at the left ends

[12] Note that we are extending our First_k operator to sets of sequences: $\text{First}_k(\cup\alpha_i) = \cup\text{First}_k(\alpha_i)$.

of all rightparts, the resulting grammar is LR(k) if, and only if, the original grammar is LL(k). In other words, if we allow to arbitrarily insert null nonterminals within rightparts in order to trigger semantic actions, we can warranty that no LR(k) conflict is introduced only if the original grammar is LL(k)[13]. Reasoning in this way, LR(k) grammars could not be rewritten in general as LR($k-1$), hence LL(k) would properly include LALR(k) and LR($k-1$). From these considerations, and from practical examples, Parr concludes [Par93, PQ96] that practical parsers need $k > 1$ lookahead.

In his thesis, a method is proposed to overcome the exponential combination of k-lookahead sequences by using adaptable lengths, testing one terminal at a time by using specific DFAs, lists of lookahead sequences, and caching, resulting in a heterogeneous method of near linear complexity. He develops his method for SLL(k) grammars, which are considered the method of choice in practice, because of their simplicity of construction and use, and shows that it is equally applicable to the general class of LL(k) and LR(k) grammars and their usual subsets.

ANTLR is the parser used in the Purdue Compiler-Construction Tool Set (PCCTS) [PDC92, Par96]. It accepts EBNF grammars, and its parsing power has evolved since the first versions. That described in [PQ95] uses *pred-LL(k)* grammars. Its parsing method is basically the one described above for LL(k) grammars, augmented with syntactic and semantic *predicates*, which the user may specify in case of conflict to indicate which rightpart to prefer. A syntactic predicate is a regular expression including nonterminals which has to be matched by an unbounded exploration of the rest of the input in order to select its associated rightpart; in fact, this is a form of selective backtracking. Semantic predicates take the form of a boolean function.

ANTLR uses C++. As it is usual in top-down parsers, the user can insert its semantic code within the rightpart symbols, and the system assists in making L-attributed translations [LRS74].

PCCTS if free software. Other tools within PCCTS are DLG, a lexical analyzer generator, and SORCERER, a simple tree parser and rewrite generator.

A variant of PCCTS is JavaCC, a free product of Sun Microsystems, developed by S. Sankar (parser generator) and others, for use with Java, and which has the particularity that grammar rule notation can be similar to Java code. A re-implementation of PCCTS, also named ANTLR, for use with Java, is also at the time of this writing under development by Parr.

It is clear that LL parsers need much more right-hand context than LR parsers if we need a powerful parser. The solution given with these tools may be almost automatic, by extending at the user request the value of k

[13]Evidently, this is the worst possible case in theory. In our opinion this is too pessimistic in practice, and this single result should not bring us to give up using more powerful methods as LR.

(which is quite limited as solution if we compare with LR), or almost manual in the form of predicates, and the users has no help from the system as to whether his solution is correct[14]. In our opinion, parser generators should help more the user by automatic means, and warranty as much as possible that the parser will always work well.

2.4.2 LC parsers

In a Left Corner (LC) parser [RL70, Dem77], production rightparts are considered in two sections, a prefix and a suffix. The former is called the "left corner". In LC parsing, the prefix is parsed bottom-up, while the suffix is parsed top-down. Thus LC parsers lie somehow between LL and LR parsers, which in fact may be considered extreme cases of LC. In general, when a lookahead of length k is used, we write LC(k), as usual.

An interesting aspect of LC parsers is in connection with the points within production rightparts at which semantic actions can be specified. These points are called *free positions* [PB80]. They mark the possible ends of left corners which can be used for LC parsing. Moreover, as we know, in LL(k) grammars, all positions within rightparts are free. This has suggested the possibility of transformation of LC(k) grammars, or a larger class called predictive LR(k), PLR(k), into LL(k) grammars [SSU79].

LC parsers are often criticized for having the tendency of combining the problems of both top-down and bottom-up parsing [GJ90]. Here we review a recent work in which the opposite opinion is indeed supported. We would also like to cite the works of E. Ukkonen on weak-PLR(k) grammars [Ukk79] and of K. Inoue on Least LC(k) grammars [IF83], which also approach parsing LR(k) grammars from the viewpoint of left corner parsing.

Recursive ascent-descent

N. Horspool (University of Victoria, Canada) extends the notion of generalized left corner parsing, GLC [Dem77], which uses SLR(1) as the bottom-up parsing method, to LAXLC(1) and XLC(1), which instead use LALR(1) and LR(1), respectively. In [Hor93] the algorithm to compute the possible *recognition points* (i.e., free positions) for the rightparts is shown.

Normally, the leftmost recognition point is used for parsing. The method actually produces the usual (LA)LR tables if the right ends of rightparts (which, as it is known, are always free positions) is always used. During construction of the LAXLC(1) parser, if it happens that the grammar is not LALR(1), then it is suggested to test whether the grammar is LL(1). If it is, then all the positions are known to be free!

[14] As a JavaCC tutorial (`http://www.suntest.com/JavaCC/DOC/lookahead.html`) says: *"it assumes you know what you are doing."*

The theoretical advantage of the method is that it can parse LALR(1) and LR(1) grammars with the use of smaller tables. Although the paper considers recursive ascent parsers [Rob88, KA88] as impractical, it is proposed there to implement LAXLC(1) and XLC(1) in the form of recursive (ascent and descent) procedures to be automatically generated, and where the programmer can freely insert its semantic code. In our opinion, the author disregards the inconveniences of this approach when successive grammar transformations take place during the development process.

2.4.3 Bounded methods

Under the general name of bounded methods, we identify the family of shift-reduce methods which only use a bounded context around the stack top in order to decide each parsing action [AU73a]. They are amongst the earliest parsing techniques. We may cite the intuitive and simple technique of *operator precedence* [Flo63], where *operators* —i.e. terminal symbols— guide parsing through their binary precedence relations. In the *simple precedence* [WW66, Pai64] and *weak precedence* methods [IM70, ADU72], binary precedence relations are extended to nonterminals also, and permit to detect whether the stack top may contain a handle. Usually, these methods impose additional strong conditions on the grammar rightparts: they cannot be empty[15] and have to be *uniquely invertible* (i.e., all rightparts have to be different). These (1,1) *Wirth-Weber* precedence relations were also *extended* to (m, n) precedence, where relations apply to two sequences, one for the m topmost stack symbols and the other for an n-symbol lookahead. Another extension involved two steps, one to detect the presence of a handle, and a second one to identify the rule to reduce, leading to *mixed-strategy precedence* (MSP) methods [MHW70, ADU72]. Other solutions are the "bounded-context" methods [Flo64, Iro64, Gra64, Loe70, Gra74, AU73a], as (m, n)-*bounded context*, BC(m, n), where the rule to reduce is included in the relation itself between m topmost symbols and n lookahead symbols, and the more general (m, n)-*bounded-right-context*, (m, n)-BRC, where a context of m symbols to the left of the handle is considered.

In spite of their limitations, these methods were reasonably powerful and efficient, and were very popular. In fact, (1,1)-BRC, uniquely invertible (2,1)-precedence and simple MSP generate the class of deterministic CFLs [AU73a]. The precedence tables can be linearized to optimize space, and the so-called *Floyd-Evans production language* [Flo61, Eva64] was used in practice to specify the parsing algorithms by means of a series of pattern tests and transformation actions. However, "unbounded" parsers are clearly superior, since (almost) all the stack information is at their disposal. Consequently, all these "bounded" shift-reduce parsers were superseded when

[15] *Epsilon* precedence methods were developed (much) later [MW90, Mil91].

a reasonably-sized LALR(1) parser was devised and an efficient algorithm to build it was found. However, LALR(1) does not even include BC(2,1) grammars, for instance, in spite of its very small bound[16].

Cigale

Cigale [FL92] is a front-end generation toolkit developed by a group leadered by O. Lecarme (University of Nice-Sophia Antipolis, France), a system that originated in the "système de Montréal" [LB74] by the same author. Its design approach is to offer an efficient and practical compiler building and assembly workshop to build nice implementations. The system comes with pre-built procedures for almost all aspects of front-ends for usual constructs of imperative languages (the most used ones in the software industry), which the programmer may take as such or adapt to his needs.

The components of Cigale are the scanner generator Lexigen, the parser generator Syntgen, the system assembler Compgen, and the systematic test-case generator Proggen.

Syntgen has originally used Floyd-Evans productions to implement a variant of MSP using weak precedence relations, extended with bounded left-hand contexts to determine the rule to reduce[17]. There is a set of possible wider-context exploration actions that the user can specify to solve conflicts in a semi-automatic manner. Semantic actions can be inserted in the form of Pascal code, and help is provided to perform L-attributed translations.

The system incorporates automatic error recovery [Mau86] inspired in the method of Graham and Rhodes [GR75], by estimating the point of error, condensing left-hand and right-hand contexts, and applying a least-cost reparation.

Cigale provides tools to generate on-the-fly the stack-based EM intermediate code of the Amsterdam Compiler Kit (ACK) [TSKS83], whose tools are used for back-end generation: peephole and global optimizer, back-end generator, target optimizer, assembler, and link editor.

[16] This is the case for the example grammar of Figure 3.1 in page 34.

[17] It was precisely the intent to automatically extend this rule-to-reduce decision process to its maximum possibilities which gave place to the development of a reverse automaton. Eventually [FG91], the reverse automaton was found to be powerful enough to incorporate the whole decision process, i.e., to trivially include also the decision of whether to shift [FG92b]. Thus, in some manner, it incorporated the information of precedence relations when this information is useful for decision, and then precedence relations actually became unnecessary.

Chapter 3

Introduction to the Approach

In this chapter we present first examples of non-LALR(1) grammars, which are nevertheless LR(1), in order to show that there exist some practical language constructs which need full LR(1) parsing. In the next sections we shall briefly show how we intend to approach the problem of full LR parsing by means of a automaton that scans a stack suffix from its top, and, finally, we shall define some concepts that will be useful for the development of our method in subsequent chapters.

3.1 Difficulties with LALR(1) parsing

Let us see examples of non-LALR(1) grammars in order to motivate the research of practical full LR parsers. In Chapter 5, after presenting the construction of the discriminating reverse (full) LR(1) parser, these examples will be developed, and some others will be also shown.

A small example of an LR(1), non-LALR(1) grammar is shown in Figure 3.1. When given to yacc, the following result is produced:

```
5: reduce/reduce conflict (red'ns 6 and 7 ) on $end
state 5
        B :  a_A
        A :  a_    (6)
        A : _    (7)

        b  reduce 7
        a  shift 10
        .  reduce 6

        A  goto 9
```

Here yacc cannot distinguish the derivations

$$\mathtt{S}\Rightarrow\mathtt{A}\Rightarrow\mathtt{a}$$

```
S : A | B 'b' | 'b' B | 'b' A 'c' ;
B : 'a' A ;
A : 'a' | ;
```

Figure 3.1: $\mathcal{G}_{sba}$: small LR(1), non-LALR(1) grammar.

```
prog : 'with' declist 'eval' explist ;
declist : decl ';' declist | decl ;
explist : exp ';' explist | exp ;
decl : var ':' type | type '=' type ;
exp : var '=' exp | var '+' var | var
    | type '(' exp ')' | '(' exp ')' ;
var : 'id' ;
type : 'id' ;
```

Figure 3.2: $\mathcal{G}_{pdt}$: LR(1), non-LALR(1) grammar with conflict on `id`.

and

$$\mathtt{S}\Rightarrow\mathtt{bB}\Rightarrow\mathtt{baA}\Rightarrow\mathtt{ba},$$

and in the former case, after shifting `a` into the stack, instead of reducing `a` to `A`, a default reduction is performed, while this reduction is only legal when the lookahead is `b`. Note that the problem cannot be solved by changing the default reduction, and that the above conflict is independent of whether `c` is equal to `b` or not.

A more practical example of LR(1), non-LALR(1) grammar in the area of programming languages is shown in Figure 3.2. When given to `yacc` the following result is produced:

```
7: reduce/reduce conflict (red'ns 13 and 14 ) on =
state 7
        var :  id_    (13)
        type :  id_    (14)

        (  reduce 14
        .  reduce 13
```

That is, the LALR(1) automaton cannot decide in state 7 in front of `=` as lookahead symbol but, since the grammar is LR(1), it should be possible to deduce from the stack contents what is the correct reduction to perform. If all the legal possibilities according to the grammar are considered, it can be seen that `yacc` cannot distinguish in that state between those stack-plus-lookahead configurations containing in the last position one of the following sequences:

```
    ... ( id _ =
 ... eval id _ =
    ... = id _ =
... exp ; id _ =
```

which always indicate that the unique possible action, according to the grammar, is `reduce 13`, from these other possibilities:

```
  ... with id _ =
... decl ; id _ =
```

which signal to `reduce 14`. Nonetheless, it is evident that a simple analysis of the topmost stack contents, up to 3 symbols from the top in the worst case (when there is a `';'` as the second-to-top symbol), would permit to take a decision.

In the discriminating reverse approach, the stack will be reversely explored from its top in order to discriminate amongst all the possible stack-plus-lookahead configurations. It is evident that, being able to use all the stack-plus-lookahead information, the discriminating reverse method can be applied to any LR(1) grammar, be it LALR(1) or not.

The (conflictive) LALR(1) automaton for the above grammar has 34 states, and the canonical LR(1) automaton has 49 states, while our reverse automaton only needs 15 states to discriminate amongst all the possible parsing actions. See from page 119 in Section 5.6.2 for more details about parsing this example grammar, where more examples of LR(1) non-LALR(1) grammars are shown.

As we have seen, LL(1) parsers use all the input already read, up to *before* a phrase, plus 1 symbol lookahead, to take a decision, actually a prediction on the rule producing that phrase, while LR(1) parsers additionally use the phrase itself, plus 1 symbol lookahead. Thus, their parsing power is clearly superior, and in fact LR(1) grammars properly include all LL(1) grammars. Nevertheless, there are some LL(1) grammars[1] which are not LALR(1). We can understand that this problem arises from the fact that LALR(1) might *loose* left-hand context information because of LR(1) state merging, information that may be available to the LL(1) parser. Fortunately, this cannot happen if the grammar does not contain any nonterminal exclusively producing the empty string. Although these kind of rules are rarely used in practice, they may be sometimes of interest. For instance, yacc itself internally augments the grammars with this kind of rules in order to trigger semantic actions in the middle of rightparts.

Direct —but also reverse, as we shall see— LR(1) parsing tables and item sets of states have the property that they are essentially correct, in the sense that, if there is an item in a state or a transition in a table,

[1] Section 5.6.3 shows an example of such grammars.

this implies that this is a true, legal possibility from the grammar at that state. Unfortunately, LALR(1) parsers do not have this property, that is, the generator user cannot be sure, when analyzing a table or item set —e.g. because of a syntax conflict—, that all the possibilities shown in the state are actually possible for every path coming from the initial state. Obviously, this can only be a source of confusion, even if the user is aware of this circumstance.

We can conclude that LR(1) is powerful, simple to understand, and natural, while LALR(1), although somehow less powerful in practice, is not easy to understand when a problem comes from a state merger while the grammar is still LR(1). Moreover, since the generation algorithm does not initially produce a true LR(1) parser, but an LR(0) one, there can be no indication on the origin of the conflict. This is usually misleading for the LALR(1) generator user —of course, SLR(1) has the same problem, aggravated. Thus, there are some LR(1) grammar cases missing for LALR(1) grammars which should, and can, be naturally expected to be parsable, and which are relevant to the programming language grammar designer.

In this thesis we develop a *discriminating reverse* approach to parse LR(k) grammars. We will give special attention to the practical class of full LR(1) grammars —whether or not they are LALR(1) will be of not much significance. Grammars like the one shown above will be of course accepted and the parsing tables generated. In this new approach, before each parsing action, the topmost part of the stack is scanned from its top until a point where, considering the lookahead, a parsing decision can be taken. User-controlled ambiguity-resolution can be also incorporated onto the method, as it will be also shown.

3.2 The discriminating reverse approach

Let us consider the general problem of shift-reduce parsing for a sentence generated by a context-free grammar. We can informally define this problem as the problem of deciding from the current rightmost sentential form, the next parsing action to perform, i.e., whether to shift or to reduce, and in this latter case which rule to apply for the reduction.

It is generally considered that in practice backtracking and/or unbounded lookahead are too costly to implement. So we shall restrict ourselves to precisely no-backtracking, finite lookahead parsing, and thus we are confronted with the problem of deciding the next parsing action from the information contained in the parsing stack plus k lookahead terminals. As wee have seen, the class of grammars that precisely allows to always take a parsing decision on this information are called LR(k) grammars.

We shall take here an original approach for parsing the general class of LR(k) grammars, which is inspired in both the initial precedence and

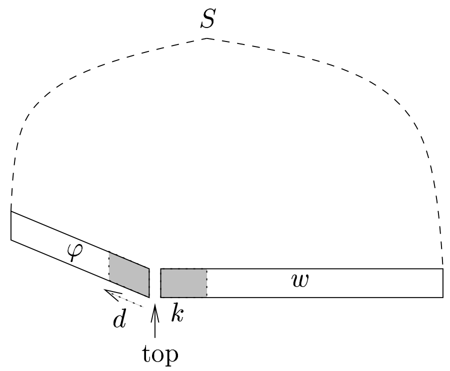


Figure 3.3: Left-hand and right-hand contexts for bottom-up parsing decision.

bounded-context approaches, and in the item set technique of conventional LR parsers. First of all, it is quite natural to initially try to take a decision from the symbols next to the current point of exploration, as precedence and bounded-context approaches do. After all, it is precisely in this area were the parsing action will take effect. But the inconvenience is that, sometimes, it is necessary to have more information, and this eventually makes a big difference in parsing power. So the idea is to look for this information, but *only if necessary.* On the other hand, the very idea of full (direct) LR(k) parsing is to build a full recognizer for the parsing stack, in order to condensate in its topmost state all the relevant stack information for subsequent parsing decisions. The inconvenient is that, for full LR(k) parsing, $k > 0$, the number of cases to consider, that is, the number of different automaton states, is considered too big for practical programming language grammars. Since in our approach only a minimal partial —quite marginal, as experience with precedence parsers shows— stack explorer is to be built, it is clear that an important improvement on size reduction is to be expected for the same parsing power.

Figure 3.3 illustrates the differences amongst these approaches. As we have seen, the essential problem in shift-reduce parsing is what procedure to use to determine the next parsing action to perform, that is to say, whether the parser has to shift, to reduce using a certain grammar rule or to signal a syntactic error. In general, there will be a sentential form consisting of φ, that is, the current stack contents, and w, the rest of the input to be read. Precedence and bounded-context methods use a bounded number of symbols around the top of the stack to determine the parsing action:

$$Action(\varphi : d, k : w) \quad \text{with fixed } d.$$

and thus have a limited parsing power. Classical, direct LR(k) methods employ a deterministic finite-state automaton recognizing the full stack language, and determine the next parsing action from the current automaton state and the lookahead:

$$Action(state(\varphi), k : w).$$

The relevant parsing information is coded into the state to allow this decision, but this normally provokes a huge number of states.

In our method, we extract the relevant information in another way: for each parsing action, a stack scan is started from the top of the stack, which will end as early as the stack suffix read so far plus the current lookahead determine a unique parsing action, according to the grammar:

$$Action(\varphi : d, k : w) \text{ for minimal } d = f(\varphi, k : w).$$

Since the exploration is minimal for discrimination, the automaton is usually far from recognizing the whole language of legal reverse stacks. Hence the resulting automaton should normally have a relatively small number of states, in comparison with a direct full LR automaton. Moreover, its full LR parsing power can be easily predicted, since according to the LR property, the stack plus lookahead contents will always allow to take a decision.

But how can such a stack explorer be built? In the method presented here, a reverse –that is, from the parsing stack top towards its bottom– finite state automaton is built. The purpose of this automaton is to discriminate, amongst the different possibilities of parsing action, which are compatible with the portion of stack read up to the current state, and to stop when the number of possibilities reduces to one.

But is it possible to build such a reverse automaton? Let us consider for the sake of simplicity the case $k = 0$. It is a well-known fact, and immediately clear from the conventional LR(0) construction, that the language of legal parsing stack configurations is a regular language $\mathcal{L}$, since it is accepted by the (direct, bottom-to-top) automaton. Furthermore, if the grammar under consideration is LR(0), then each parsing action i, $i = 0..n$, has an associated regular sublanguage $\mathcal{L}_i$, that is the language of stack contents which ends up in a state whose action indicated by the automaton is precisely i. Obviously, every $\mathcal{L}_i$ is a subset of $\mathcal{L}$. But the key point for us is that all pairs $\mathcal{L}_i$, $\mathcal{L}_j$, $i \neq j$, are disjoint, for otherwise there would exist a legal stack ending in a state with at least two different actions, contrary to the fact that the grammar is LR(0). In other words, these $\mathcal{L}_i$ form a partition of $\mathcal{L}$, such that each $\mathcal{L}_i$ is an equivalence class with respect to the parsing action corresponding to i. Thus, the problem of deciding the next parsing action can be restated in terms of deciding to which $\mathcal{L}_i$ the current stack belongs. It is evident that the conventional LR(0) automaton does precisely this, by completely reading (having read, really) the stack, and telling the action to perform in its topmost state[2] . But it is clear that there is no need to read

[2]Since our grammars are context-free, the context information which can be useful for

the whole stack if exploration is started from its top, as far as the action to perform is identified.

It might still remain the general question of whether a regular language can be reversely recognized by a finite automaton. We can constructively answer yes, for it is possible to build a reverse automaton from the direct one by exchanging the initial and final properties of its states.

3.3 Formal preliminaries

The following statements are applicable to our *discriminating reverse* parsers, which use a deterministic finite-state automaton to partially explore the stack from its top, and then decide a parsing action (or signal a syntactic error), possibly using some finite lookahead.

Definition 3.1 *A* stack form *is a pair* (α, z)*, also written* $\alpha{\mid}z$*. In such a stack form,* α *represents the current stack contents, and* z *the remaining input to be read. The current stack contents before the action,* α*, can be also written in dotted form, that is, as* $\eta{\bullet}\sigma{\mid}z$*,* $\alpha = \eta\sigma$*, to indicate that the discriminating reverse automaton has explored the stack suffix* σ *to decide the action. Symbols in* σ *are said to be* visited *during that exploration.* □

During shift-reduce parsing of a sentence, stack form $\alpha{\mid}z$ implies that the current rightmost sentential form is αz.

Definition 3.2 *Given an discriminating reverse automaton* $\mathcal{A}$ *for a reduced grammar* $\mathcal{G}$*, a* parser action *is a pair of stack forms* $(\alpha'{\mid}z', \alpha{\mid}z)$*, also written as* $\alpha'{\mid}z' \overset{i}{=\!\!\dashv} \alpha{\mid}z$*, representing the reduce or shift action* i *performed on stack form* $\alpha{\mid}z$ *producing* $\alpha'{\mid}z'$*. If the action is to reduce to some nonterminal* A*, it is said to be* generated *in that action.* □

Please note that our notation intentionally represents from right to left the sequence of transformations that the parser performs on the stack, in order to write the forms in the same order as it is customary for derivations. According to the above definition, when parsing a sentence, $\alpha'{\mid}z' =\!\!\dashv \alpha{\mid}z$ implies $\alpha' z' \underset{\mathrm{rm}}{\Rightarrow} \alpha z$ when $\alpha' z' \neq \alpha z$.

Definition 3.3 *For a given grammar* $\mathcal{G}$*, stack form* $\tau{\mid}yz$ *and shift-reduce parsing action* i *are said to be* compatible *iff there is some rightmost derivation*

$$S' \underset{\mathrm{rm}}{\Rightarrow}^* \zeta A z \underset{\mathrm{rm}}{\Rightarrow} \zeta\alpha z = \tau y z$$

decision tends to be local, thus topmost stack symbols should typically be more informational than deeper stack symbols. In fact, an imaginary discriminating *direct* parser (with or without the correct prefix property) would actually need to reach the stack top to be able to decide, since any valid stack is preceded during parsing by all its prefixes, each one requiring a (usually different) decision.

such that i is to reduce τ's suffix α to A for $y = \varepsilon$ or to shift otherwise. $\square$

Of course, for unambiguous grammars, there is only one compatible action for each legal stack form.

The following (usual) convention is assumed.

Convention 3.4 *The discriminating reverse parser stops when it finds an error or when it accepts the input by reducing to S'.* $\square$

Chapter 4

Discriminating Reverse LR(0)

In order to introduce the reader to the ideas of the discriminating reverse approach, it may be useful to start by considering the simpler case of parsing LR(0) grammars. In the next chapter we shall concentrate on the particularly important and practical case of LR(1) grammars, and finally generalize the method from a mainly theoretical viewpoint for the general case of LR(k) grammars.

In this chapter we first study, with the help of a simple expression language as a running example, how to build a *reverse* —i.e., from top to bottom of stack— recognizing automaton for the language of stack contents such that the next parsing action is a given one. Next, we shall see how to build an all-action automaton, such that in each state it has the set of parsing actions which are compatible with the stack portion already read from its top. This automaton allows to know the parsing action to actually perform as soon as the next-state set of actions contains only a single action. Of course, these single-action states do not need actually to be built, but only the useful discriminating part (and maybe a handle-verification extension).

Finally, the actual algorithms for automaton construction and its use for parsing are given, and the method is formally proved for this LR(0) case.

4.1 Single-action reverse stack recognizer

Let us first consider the problem of building a reverse recognizer for a given action i, that is to say, a deterministic finite-state automaton that reads the current parsing stack (which is the head of a rightmost sentential form during bottom-up parsing of a sentence) from the current top of stack until its bottom, and accepts precisely the language of legal stack contents such that the next parsing action is i. According to Convention 1.7 we shall code with $i = 1..n$ the action of reducing with grammar rule numbered i; and we shall use $i = 0$ for coding the shift action.

Let us consider first this latter case, and let $A \xrightarrow{i} \alpha$, with $\alpha = X_1..X_m$.

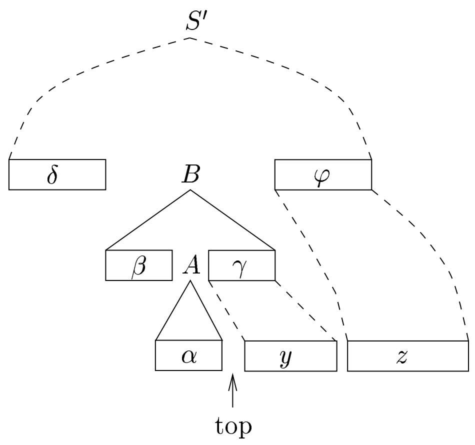


Figure 4.1: Typical parsing tree for $A \rightarrow \alpha$ reduce action.

$$\begin{array}{ll}
1: & S' \rightarrow \vdash S \dashv \\
2: & S \rightarrow S - T \\
3: & S \rightarrow T \\
4: & T \rightarrow (S) \\
5: & T \rightarrow a
\end{array}$$

Figure 4.2: $\mathcal{G}_{exp}$: LR(0) grammar for simple expression language.

It is evident that for the next parsing action to be i, the handle α must be on top of the stack, and thus the automaton has to initially accept the symbols in α, in reverse, i.e., from X_m to X_1. Then, to continue exploration past the left end of this handle, the automaton has to consider all the possibilities of legal stack symbols to the left of A, i.e., the reducing rule leftpart, and thus all the places of A in the grammar rules' rightparts have to be considered, such as $B \rightarrow \beta A \gamma$; see Figure 4.1 for a typical syntax tree configuration where current stack is $\delta\beta\alpha$ and the next parsing action is to reduce using rule $A \rightarrow \alpha$. Now automaton construction can continue for the symbols of β, from right to left. The process should evidently be repeated when the left ends of the several possible β's are reached.

In order to make more explicit to the reader the problem we are considering, let us try first to build "by hand" a recognizer for a simple case. Let us use a simple expression language as a running example. Thus, consider the augmented grammar of Figure 4.2.

If we want to reversely recognize the language of legal stacks when the action to perform is for instance to reduce according to rule $3: \ S \rightarrow T$, it is evident that the first symbol to accept is T, for it constitutes the handle that must be present on top of the stack in order to perform the reduction.

Next we must consider what can legally appear to the left of the S to

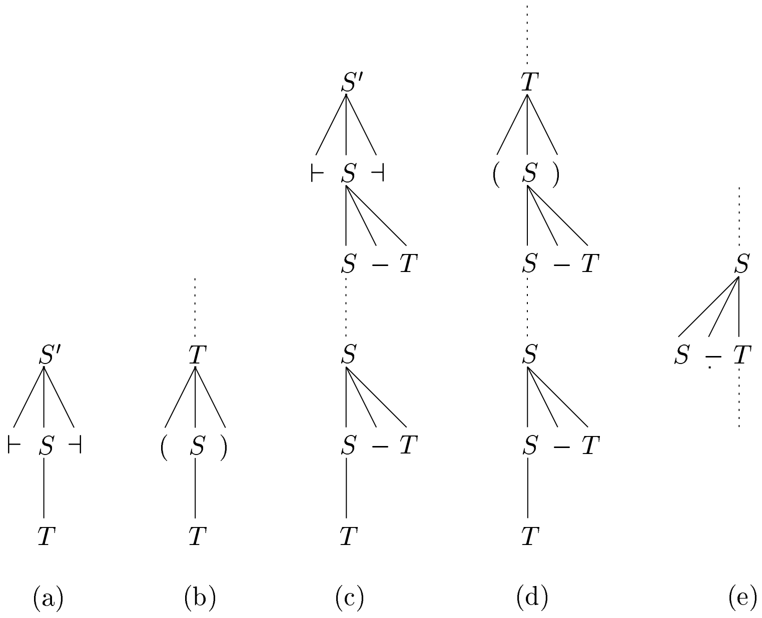


Figure 4.3: Tree portions for handle T.

which the handle T is to be reduced; see Figure 4.3. Accordingly, we have to consider several possibilities depending on the rule containing the reducing-to S: $\vdash$ for rule 1 and "(" for rule 4, as is shown in trees (a) and (b), and rule 2. Rule 2 is a special case, for no symbol is to the left of S in the rightpart. Nonetheless we should in general consider the rule leftpart symbol to identify some possible stack symbols to its left. In this case this symbol is again S, which we are just considering, and thus the next lower stack symbol will be, as we have already seen, $\vdash$ or "(", as is shown in trees (c) and (d). For subsequent left symbols we can see that tree portions (b) and (d) of Figure 4.3 are recursive with all tree portions of the same figure, and thus $\vdash$ or a number of more "(" can appear.

Finally we have to consider the case of T in rightpart of rule 2, as is shown in tree portion (e). It is possible at this point to read "$-$", S, and then, since the leftpart is S, we are in the same situation as in all cases after having reduced T to S, i.e., tree portions (a) to (d). In summary, we can build the recognizing automaton of Figure 4.4.

4.1.1 Items

The above example suggests to approach the automatic construction of this kind of recognizer by using an item notation of the general form $[A \rightarrow \beta \bullet \gamma]$, which resembles to the notation for the conventional LR parser construction, but which in our case has important differences in meaning: the dot indicates the current point of *stack* exploration, and it moves from right to left.

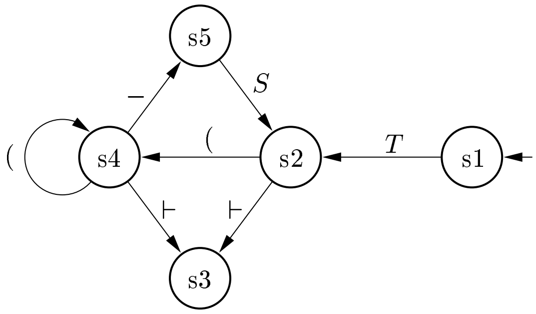


Figure 4.4: Reverse recognizing automaton for action-3 stack language.

In our example grammar, for the above example parsing action "reduce using rule 3", in the initial state we would just have the item

State 1
$[S{\rightarrow}T\,\bullet]$

where the dot indicates that, effectively, since we are going to begin exploration from the top of the stack, the dot is placed to the right of the handle T and since it is going to move towards the left, i.e., towards the stack bottom, in the next state we shall (initially) have

State 2
$[S{\rightarrow}\bullet T]$

As it was previously said, in general, for a reducing action, the first states correspond to the exploration of the symbols present in the rightpart to reduce, from right to left, until the exploration dot reaches the left end of the rightpart, as in state 2 above.

The item approach also clearly suggests to use an item-set technique for state construction, whereby each automaton state is associated to a set of items, and two states are equivalent when their corresponding item sets are equivalent. In our reverse scan, a closure operation $\mathcal{C}$ is thus to be considered, which incorporates new items to the set when the dot is at the *left* end of a rule rightpart, as it is the case of state 2 above. Thus, in our case, since item $[S{\rightarrow}\bullet T]$ has the dot at the left end of the rule rightpart, we have to look for appearances of the rule leftpart symbol S within grammar rules, that is, rules $S'{\rightarrow}\vdash S\dashv$, $S{\rightarrow}S-T$ and $S{\rightarrow}(S)$, with the dot to the left of the S in order to continue with the stack symbols which can legally appear to the left of the reducing-to S. Here we also find recursivity in the case of item $[S{\rightarrow}\bullet S-T]$, which induces to look again for grammar rules with S in their rightpart, but this logically adds no new item. All of this evidently corresponds with our previous study of cases. So, the complete set of items

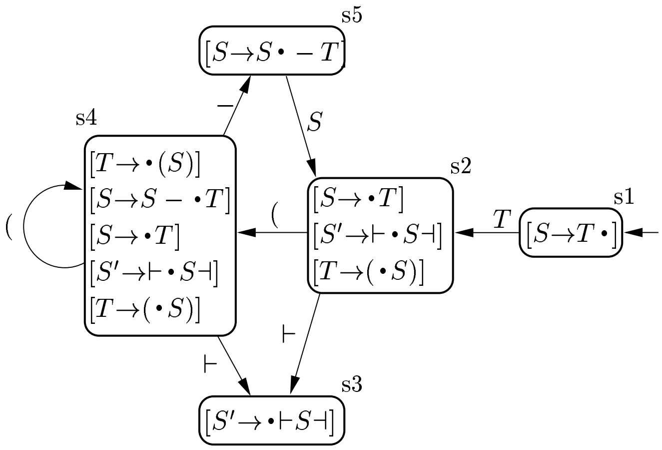


Figure 4.5: Action-3 stack recognizing automaton including item sets.

for state 2 is the following

$$
\begin{array}{l}
\text{State 2} \\
\quad [S \rightarrow \bullet T] \\
\quad [S \rightarrow \bullet S - T] \\
\quad [S' \rightarrow \vdash \bullet S \dashv] \\
\quad [T \rightarrow (\bullet S)]
\end{array}
$$

Note that item $[S \rightarrow \bullet S - T]$ is actually *equivalent* to item $[S \rightarrow \bullet T]$, for the useful part of the item lies to the left of the dot, and in the leftpart nonterminal, but in this case both are equal for both items. Accordingly, for our purposes, in general items $[A \rightarrow \alpha \bullet \beta]$ and $[A \rightarrow \alpha \bullet \gamma]$ are completely equivalent, independently of β and γ. Accordingly, we could have alternatively chosen to write $[A \rightarrow \alpha \bullet]$, $[A \rightarrow \alpha]$, or simply $[A,\ \alpha]$, notations which possibly reflect more formally item meaning. On the other hand, the notation used here seems to us more conventional and intelligible.

Thus, considering item equivalence, in our example we shall write:

$$
\begin{array}{l}
\text{State 2} \\
\quad [S \rightarrow \bullet T] \\
\quad [S' \rightarrow \vdash \bullet S \dashv] \\
\quad [T \rightarrow (\bullet S)]
\end{array}
$$

Item set construction may be continued in this way, producing new item sets corresponding to new states, until finally producing the automaton of Figure 4.4, which is shown again in Figure 4.5, including the resulting item sets of each state.

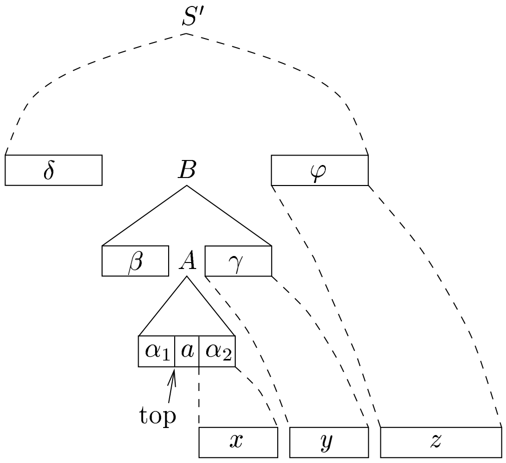


Figure 4.6: Typical parsing tree for shift action.

Shift action

See Figure 4.6 for a typical tree configuration where the next parsing action is to shift, that is, terminal symbol a in the figure will be read and put on top of the stack. Note that if α_2 is a (possibly empty) sequence of terminals, then $\alpha_1 a \alpha_2$ will be the next handle to reduce; otherwise the next handle (which may be ε, of course) to reduce will be found somewhere in x. In all cases, one or more symbols are to be shifted until having the handle on top of the stack.

Let us now consider the construction of a reverse recognizing automaton for all the legal stack configurations where the next shift-reduce parsing action is to shift. Note that in this case, there is no need to make a distinction amongst the different terminal symbols: the next input symbol is just read and its value put onto the stack top. The implication for the initial state of our building procedure is simple: we have to consider all possible cases of shifting a terminal, that is, all the cases of rules of the form $A{\rightarrow}\alpha_1 a \alpha_2$ (see again Figure 4.6) where there is some terminal a which might be the next input terminal, which implies to consider all terminals in every rule of the grammar. For our example grammar, this means that for the initial state for the shift action, the following items must be initially considered:

State 1

$[S'{\rightarrow}\bullet{\vdash}S{\dashv}]$
$[S'{\rightarrow}{\vdash}S\bullet{\dashv}]$
$[S{\rightarrow}S\bullet - T]$
$[T{\rightarrow}\bullet(S)]$
$[T{\rightarrow}(S\bullet)]$
$[T{\rightarrow}\bullet a]$

Note that items $[T\to\bullet(S)]$ and $[T\to\bullet a]$ are actually equivalent, for they have the same leftpart and the same (null, in this case) rightpart section to the left of the dot. Thus, one of them may be removed, let it be $[T\to\bullet a]$. Moreover, since the dot is at the left end of the rightpart, a closure computation is needed in order to be able to construct the next states after transition on the current (top of) stack symbol. Thus, the following set of items is obtained for the initial state:

State 1

$[S'\to\bullet\vdash S\dashv]$
$[S'\to\vdash S\bullet\dashv]$
$[S\to S\bullet-T]$
$[T\to\bullet(S)]$
$[T\to(S\bullet)]$
$[S\to S-\bullet T]$
$[S\to\bullet T]$
$[T\to(\bullet S)]$
$[S'\to\vdash\bullet S\dashv]$

Other states are constructed in the same way as we have already seen for the case of reducing action. For instance, after transition on S, we get:

State 2

$[S'\to\vdash\bullet S\dashv]$
$[T\to(\bullet S)]$
$[S\to\bullet S-T]$

As above, item set construction may be continued in this way, producing new item sets corresponding to new states, until finally producing the automaton of Figure 4.7.

4.1.2 Closure

Let us summarize the above ideas of state construction. The closure of an item set q can be recursively computed as the smallest set satisfying

$$\mathcal{C}(q) = q \cup \{[B\to\beta\bullet C\gamma] \mid [C\to\bullet\varphi] \in \mathcal{C}(q) \text{ and } B\to\beta C\gamma \in P'\}$$

Figure 4.8 illustrates this closure computation when the stack suffix already scanned is σ, and some left end of rightpart α may have been reached.

Thus, we can use the closure operator to compute item sets of new states after a lower stack symbol transition as follows:

$$q' = \delta(q, X) = \mathcal{C}(\{[B\to\beta\bullet X\gamma] \mid [B\to\beta X\bullet\gamma] \in q\})$$

Figure 4.9 illustrates this dot displacement when scanning next stack symbol X when the stack suffix already scanned is σ. If in the figure $\alpha=\varepsilon$, then a closure computation is needed.

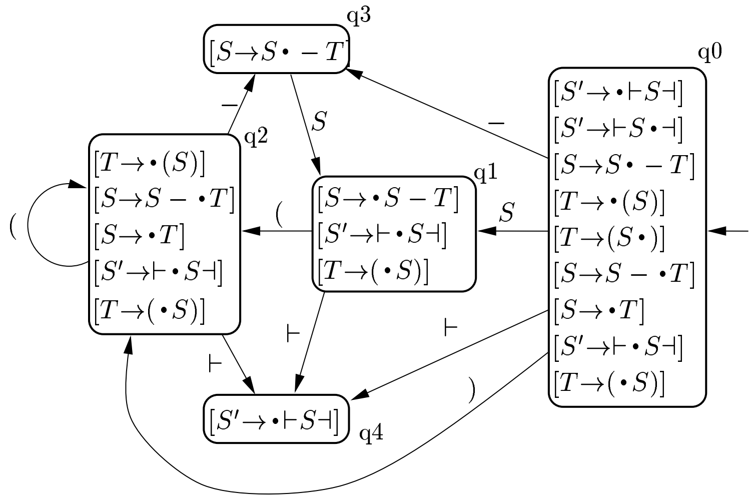


Figure 4.7: Action-0 (i.e., shift) stack recognizing automaton including item sets.

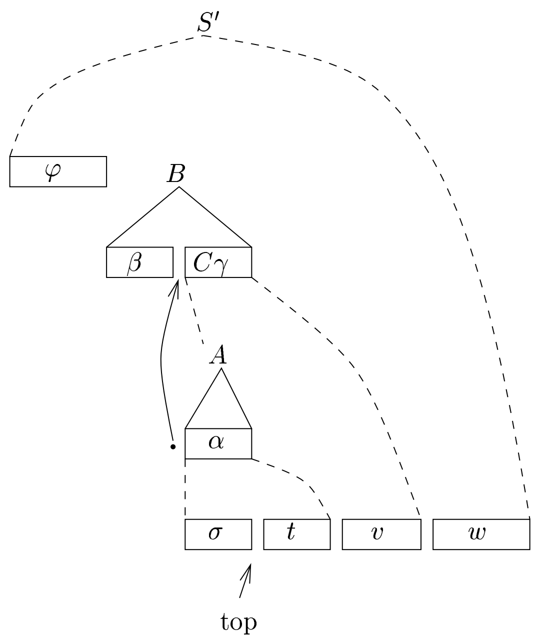


Figure 4.8: Typical tree for closure after scanning stack suffix σ.

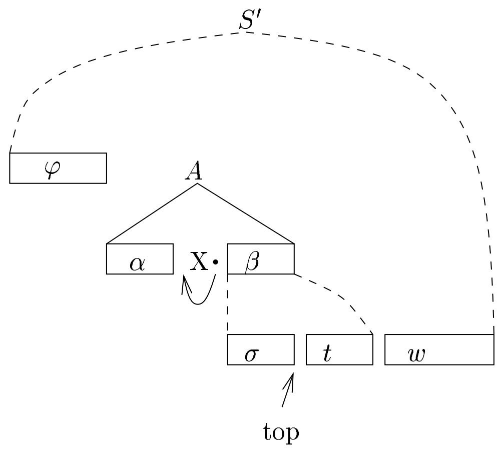


Figure 4.9: Typical tree for transition on next stack symbol X.

Finally, note that, after closure computation, items of the form $[A \to \bullet \alpha]$ are not useful anymore[1].

4.1.3 Initial states

The reverse stack recognizing automaton for reducing $A \to \alpha$ is simply built from initial state q_0 as follows

$$q_0 = \mathcal{C}(\{[A \to \alpha \bullet]\})$$

Note that this definition includes the case $\alpha = \varepsilon$, thanks to the use of the closure operation.

Now let us consider the case of building a stack recognizer for the shift action. We must indeed consider every possible case of shift, what in our item notation means, for the initial state, to consider every possible position of the dot to the left of a terminal symbol, and thus initial state q_0 is defined as

$$q_0 = \mathcal{C}(\{[A \to \alpha \bullet a\beta] \mid A \to \alpha a \beta \in P'\})$$

4.2 Full stack reverse recognizer with action discrimination

Now let us return to our initial general problem of deciding, through a stack scan from its top, the next parsing action to perform. As we have already

[1] This is only valid for the LR(0) case.

seen, we are able to build a recognizing automaton $\mathcal{A}_i$, for each of the stack languages associated to each of the possible parsing actions coded as $i = 0..n$.

It is evident that a possibility is to run all these automata $\mathcal{A}_i$ in parallel from the current stack top, and to stop when only one automaton A_j still accepts the suffix read thus far, which would evidently signal to perform action j. This will in effect be a nondeterministic automaton accepting the union of the several single-action stack languages.

But the same result can be obtained by combining all of the above single-action automata into a *single* deterministic automaton carrying all the cases in parallel, that is, accepting the full language of legal stack contents for all the parsing actions, but at the same time carrying in the states the information of what actions are still compatible with the suffix already read leading to the state, for if this information were lost during its construction, it would be useless for our discriminating purpose. In other words, states should not be considered equivalent when the language to be accepted from that point on is the same but the sets of compatible actions are different.

In order to keep track of the set of compatible actions corresponding to each state of this full stack recognizer, our reverse items will be augmented with their associated action, and thus take now the general form $[i,\ A{\to}\alpha{\bullet}\beta]$. It is easy to see that this new combined automaton can be directly built as follows:

$$\mathcal{C}(q) = q \cup \{[i,\ B{\to}\beta{\bullet}C\gamma] \mid [i,\ C{\to}{\bullet}\varphi] \in \mathcal{C}(q) \text{ and } B{\to}\beta C\gamma \in P'\}$$

$$q' = \delta(q, X) = \mathcal{C}(\{[i,\ B{\to}\beta{\bullet}X\gamma] \mid [i,\ B{\to}\beta X{\bullet}\gamma] \in q\})$$

$$q_0 = \mathcal{C}(\{[0,\ A{\to}\alpha{\bullet}a\beta] \mid A{\to}\alpha a\beta\} \ \cup\ \{[i,\ A{\to}\alpha{\bullet}] \mid A\xrightarrow{i}\alpha\})$$

The all-action initial state q_0 contains all the single-action initial states previously seen, with indication of their corresponding action in the item augmented form. Closure and transition functions are almost identical to the single-action case, but now must preserve this action information for new items generated by both functions.

It should be evident that this automaton, after having reversely read some stack suffix σ, is in a state whose item set is precisely the union of the item sets of the above single-action automata after reading the same σ, and thus, the set of compatible actions of a state is just the union of the actions in its items, that is to say, which parsing actions are still compatible with the language of stack suffixes leading to the state. It should be also clear that, for our discriminating purpose, it is not necessary to build states with unique actions, that is, when all the items correspond to the same action. Only states with more than one compatible action will actually constitute our discriminating automaton, as it will be shown in the next section.

Note that every possible position of the dot in all rules is considered for our initial state, because

- for reducing actions the dot is considered to the right of all rightparts, and (because of the closure) to the left of every nonterminal N such that $N \Rightarrow^{+} \varepsilon$, in all rightparts containing it, and

- for the shift action the dot will be considered to the left of every terminal symbol in all rightparts, and (because of the closure) also to the left of every nonterminal N such that $N \Rightarrow^{+} x_N$, for some $x_N \neq \varepsilon$, in all rightparts containing it

(remember that in a reduced grammar every nonterminal $N \Rightarrow^{+} x$). Note that normally not every position in every rule is actually needed, because of item equivalence.

Finally, let us consider the case where some string in an action-related sublanguage is a suffix of another string in another action sublanguage, that is, $\sigma \in \mathcal{L}_i$ and $\rho\sigma \in \mathcal{L}_j$, with $\rho \neq \varepsilon$. In this case, if current stack is σ, both automata $\mathcal{A}_i$ and $\mathcal{A}_j$ are still compatible at the end of σ. It is to overcome this minor technical inconvenience that a bottom-of-stack marker $\vdash$ has been introduced as a new terminal symbol in our grammar augmentation scheme, as described in Convention 1.6. Since every string now begins by $\vdash$ and this symbol cannot appear in the middle of any string, the above suffix problem disappears in practice.

Figure 4.10 shows the full stack recognizing deterministic automaton for grammar $\mathcal{G}_{exp}$, built according to the above definition. The full listing of states including item sets, actions and transitions is shown in Appendix A.1 (page 219). The sets within the states represent the sets of compatible actions associated to the states, which are derived as explained from the item sets of the states. States shown with dark background constitute the interesting, discriminating part of the automaton, which is the only part that will be ultimately needed in practice. Note that it is a very small part of the whole automaton, as it is typically the case. For the sake of comparison, Figure 4.11 shows a minimal reverse recognizing automaton built for the same reverse stack language, but without taking in consideration the above sets of compatible actions, i.e., states have been considered equivalent independently of these sets, and thus this automaton cannot discriminate actions, but just recognize whether the stack is legal or not. Finally, Figure 4.12 shows the *direct* (bottom-to-top of stack) full stack recognizer automaton, i.e., the canonical LR(0) automaton, built by using the classical approach of construction from the same grammar $\mathcal{G}_{exp}$.

4.3 The discriminating reverse automaton

As shown in the preceding section, in order to determine the next parsing action through a partial scan of the current parsing stack from its top, we only need to begin constructing the full recognizing automaton with action

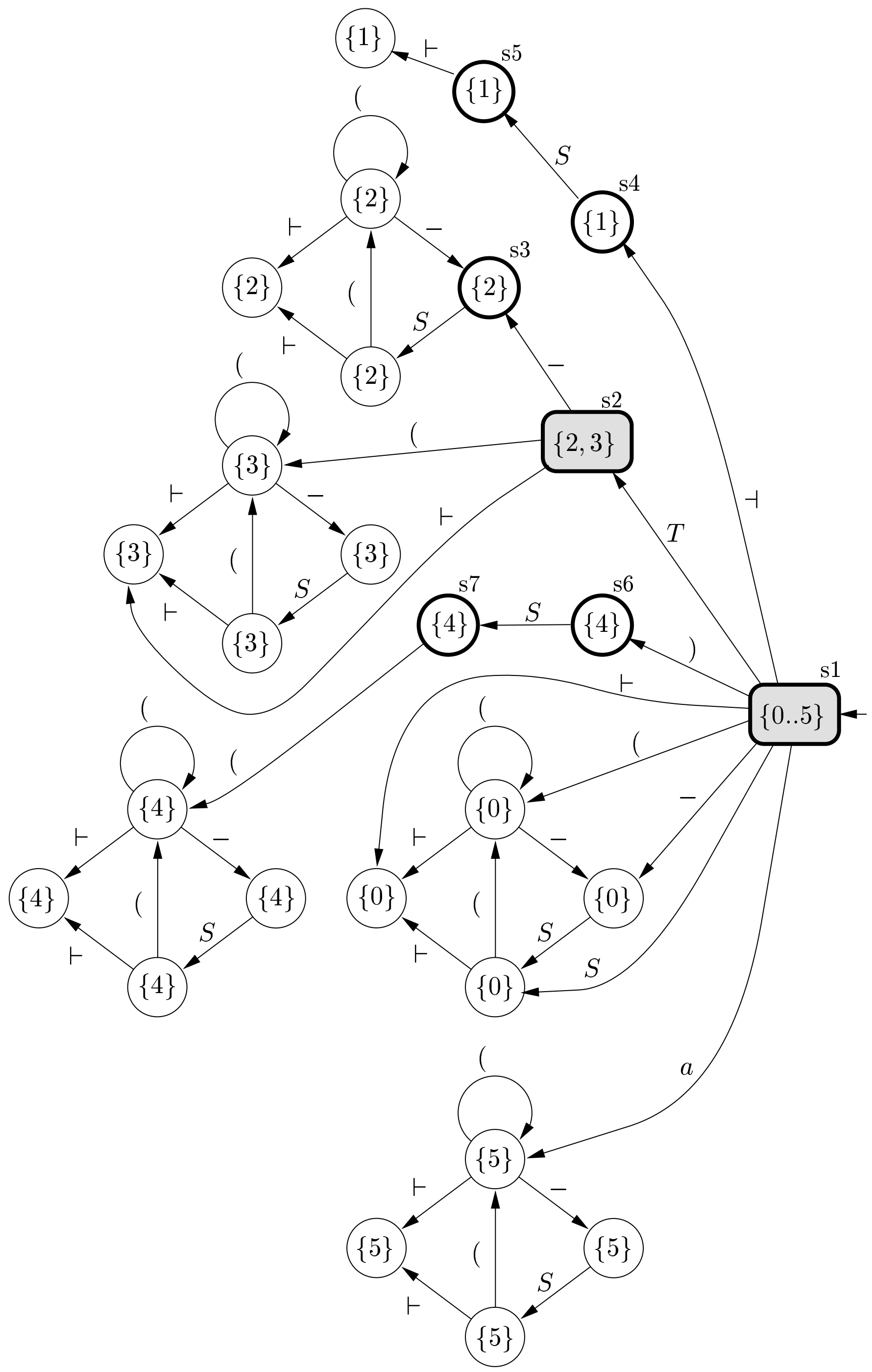


Figure 4.10: Reverse recognizing automaton for all-action stack language of grammar $\mathcal{G}_{exp}$.

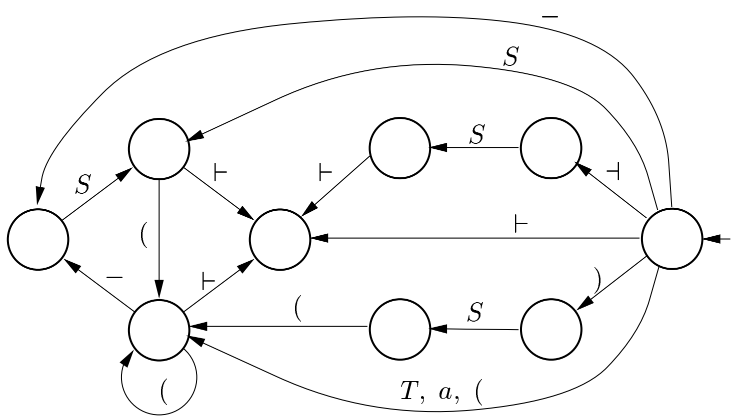


Figure 4.11: Minimal reverse recognizing automaton for stack language of grammar $\mathcal{G}_{exp}$.

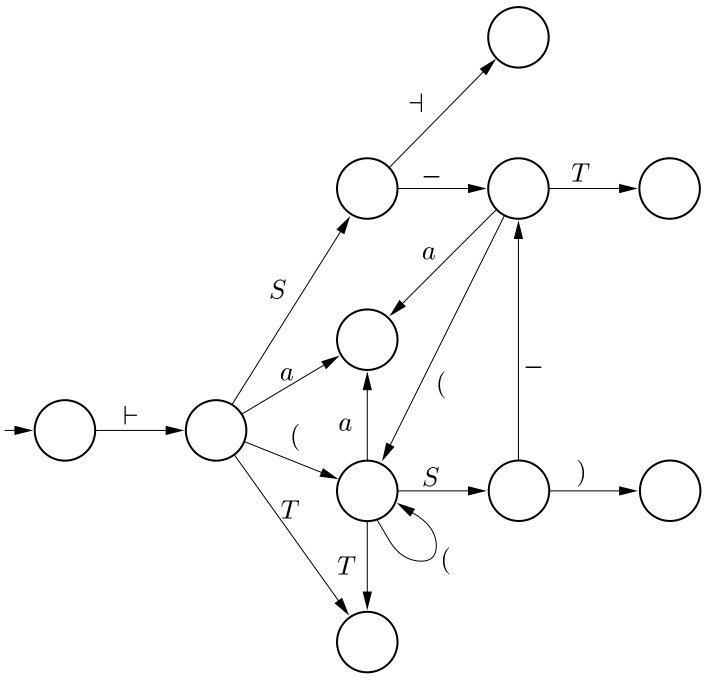


Figure 4.12: Canonical direct LR(0) automaton for grammar $\mathcal{G}_{exp}$.

	$\vdash$	$\dashv$	S	$-$	T	(	)	a
s1	0	1	0	0	s2	0	4	5
s2	3			2		3		

Figure 4.13: Discriminating reverse LR(0) parsing table.

discrimination, and at the same time inhibit the construction of states whose number of associated actions is already reduced to one. The actual construction algorithm will be shown in Section 4.4.1. For our grammar example, as we have seen, the discriminating reverse automaton just consists of the 2 states with dark background shown in Figure 4.10, and it is also represented in the form of a parsing table in Figure 4.13. The parsing table reflects in its transition and action entries the corresponding transitions shown in the automaton. On state s1, with stack symbol T, the table indicates a transition to state s2. In this particular case the rest of entries are actions, actually coding shift (0) or the rule to reduce (1..5) —reducing rule 1 signals a successful end of parsing. For instance, on state s2, if the next stack symbol is the bottom of stack marker $\vdash$, the action entry is 3, in accordance with the transition in the automaton of Figure 4.10 leading to a state whose only compatible action is 3.

Note that, since the automaton-derived parsing table is used to partially scan the stack, terminal and nonterminal symbols of the grammar are not treated differently, for they are just symbols of the stack language. Thus in the structure of the table they are treated indifferently. In following chapters, for discriminating reverse LR(k) parsers with $k > 0$, a two-part table will be shown to conveniently use the k-lookahead in the parsing action decision process.

Item sets corresponding to automaton states of Figures 4.13 and 4.10 are shown in Figure 4.14. Remember that an item like $[0, S \to S - \bullet T]$ indicates that, according to the grammar, in the current state, "$-$" is compatible with action 0, i.e., to shift. Then it is easy to verify that according to the set of items in state 1, every symbol —with the exception of T— to the left of the dot is compatible with only one action, thus allowing to already decide that action in case it is the current top-of-stack symbol. Symbol T in state 1 has two compatible actions, 2 and 3, and thus further stack exploration is needed. After moving the dot to the left past T in items $[2, S \to S - T \bullet]$ and $[3, S \to T \bullet]$ and computing the closure, state 2 item set is obtained, which already allows to decide on the next stack symbol, i.e., to reduce with rule 3 for $\vdash$ or "(", or with rule 2 for "$-$". Actually, this means that grammar $\mathcal{G}_{exp}$ is BC$(2, 0)$, for in the worst case only 2 topmost stack symbols suffice to decide the next parsing action.

It should be noted that in several cases it is possible to determine that the only possible action is a reduction like action 2 in state 2, but the presence of the whole handle in top of the stack has not actually been verified. Obviously,

State 1 $\{0, 1, 2, 3, 4, 5\}$
$[1, S' \to \vdash S \dashv \bullet]$
$[2, S \to S - T \bullet]$
$[3, S \to T \bullet]$
$[4, T \to (S) \bullet]$
$[5, T \to a \bullet]$
$[0, S' \to \vdash S \bullet \dashv]$
$[0, S' \to \bullet \vdash S \dashv]$
$[0, S \to S \bullet - T]$
$[0, T \to (S \bullet)]$
$[0, T \to \bullet (S)]$
$[0, S \to S - \bullet T]$
$[0, S \to \bullet T]$
$[0, S' \to \vdash \bullet S \dashv]$
$[0, T \to (\bullet S)]$

State 2 $\{2, 3\}$
$[2, S \to S - \bullet T]$
$[3, S \to \bullet T]$
$[3, S' \to \vdash \bullet S \dashv]$
$[3, T \to (\bullet S)]$

Figure 4.14: Item sets of discriminating portion of reverse automaton.

	$\vdash$	$\dashv$	S	$-$	T	(	)	a
s1	0	s4	0	0	s2	0	s6	5
s2	3			s3		3		
s3			2					
s4			s5					
s5	1							
s6			s7					
s7						4		

Figure 4.15: Discriminating reverse LR(0) parsing table including full right-part verification.

this verification must be done in order to perform legal reductions, as is typical in precedence methods. Possibly the most practical approach to this is to build a straightforward verifier for these cases. Anyway, the method can be naturally extended to continue exploration until rightparts to reduce are completely scanned, as it will be shown later. Figure 4.15 shows this extended parsing table, where its lower part is only intended for rightpart completion, and whose states correspond to those drawn using bold lines in Figure 4.10.

4.3.1 Parsing

Using the generated tables for actual parsing is straightforward. The parsing table is used to partially explore the stack from its top by following state transitions until an action-code entry is found upon next lower stack symbol; then the corresponding parsing action is actually performed, and the process begins again from the stack top and initial state to decide the next parsing action. The actual parsing algorithm is shown in Section 4.4.1 (page 60).

While it might seem that this stack scanning process introduces an important overhead, this is not the case. Let us see for instance the parsing sequence of sentence "$a - (a - a)$", as is shown in Figure 4.16, where successive parsing stack contents are shown with their corresponding parsing table entries, while the former are partially scanned from right to left according to table 4.15. Entries between brackets are not part of the discriminating process, for they belong to the rightpart verification (lower) part of the parsing table. For instance, in Figure 4.16 when stack contents is "$\vdash S - (S - T$", in state 1 T leads to state 2, where "$-$" already discriminates as the only possible action to reduce with rule 2: $S \rightarrow S - T$; see table of Figure 4.13. In the extended table of Figure 4.15, state 2 leads with "$-$" to state 3 to verify that effectively S is present to complete the handle, and thus the action on S is to reduce with rule 2.

4.3.2 Syntactic error detection

In general, since the parser is not intended to check whether the whole stack is syntactically correct, the stack is not necessarily a viable prefix, and thus syntactic errors might be detected after the first possible point of detection in a left-to-right scan of the input text, that is, it can partially continue parsing even in the presence of an illegal stack. From the point of view of error recovery, this is sometimes [FL88, Ric85] considered as an advantage, because of the flexibility of recovery that other methods, in particular standard LR in general, do not allow in principle.

Since our automaton performs somehow like precedence methods, the well-known error recovery procedures of condensation and repair for these

						$\vdash$	$a-(a-a)\dashv$
						0	
					$\vdash$	a	$-(a-a)\dashv$
						5	
					$\vdash$	T	$-(a-a)\dashv$
					3	s2	
					$\vdash$	S	$-(a-a)\dashv$
						0	
				$\vdash$	S	$-$	$(a-a)\dashv$
						0	
			$\vdash$	S	$-$	(	$a-a)\dashv$
						0	
		$\vdash$	S	$-$	(	a	$-a)\dashv$
						5	
		$\vdash$	S	$-$	(	T	$-a)\dashv$
					3	s2	
		$\vdash$	S	$-$	(	S	$-a)\dashv$
						0	
	$\vdash$	S	$-$	(	S	$-$	$a)\dashv$
						0	
$\vdash$	S	$-$	(	S	$-$	a	$)\dashv$
						5	
$\vdash$	S	$-$	(	S	$-$	T	$)\dashv$
				[2]	s3	s2	
		$\vdash$	S	$-$	(	S	$)\dashv$
						0	
	$\vdash$	S	$-$	(	S	)	$\dashv$
				[4	s7]	s6	
			$\vdash$	S	$-$	T	$\dashv$
				[2]	s3	s2	
					$\vdash$	S	$\dashv$
						0	
				$\vdash$	S	$\dashv$	
				[1	s5]	s4	
						S'	

Figure 4.16: Example of discriminating reverse parsing of "$a-(a-a)$".

$$\begin{array}{ll}
1: & S' \rightarrow \vdash S \dashv \\
2: & S \rightarrow RT \\
3: & R \rightarrow S- \\
4: & R \rightarrow \varepsilon \\
5: & T \rightarrow (S) \\
6: & T \rightarrow a
\end{array}$$

Figure 4.17: $\mathcal{G}_{eps}$: LR(0) grammar with empty rule for same language of $\mathcal{G}_{exp}$.

	$\vdash$	$\dashv$	S	R	T	$-$	(	)	a
s1	4	1	0	0	2	3	4	5	6

Figure 4.18: Discriminating reverse LR(0) parsing table for epsilon grammar $\mathcal{G}_{eps}$.

methods [GR75] should be adaptable to ours, which can produce a condensation around the "point in error" before actually signalling a syntactic error.

4.3.3 Grammars with empty rules

Since inspiration for this technique comes in part from the family of precedence methods, which typically do not accept empty rules in grammars, the reader might tend to think that this is also the case here. Actually, these kind of rules do not present any particular difficulty for the discriminating reverse method. See Figure 4.17 for an example grammar for the same expression language as that of grammar $\mathcal{G}_{exp}$.

The corresponding parsing table is shown in Figure 4.18, and the item set of the single discriminating state for this grammar is shown in Figure 4.19. Item $[4, R \rightarrow \bullet]$ produces, after closure, items $[4, S' \rightarrow \vdash \bullet S \dashv]$ and $[4, T \rightarrow (\bullet S)]$, which allow to decide to perform reduction $4 : \; R \rightarrow \varepsilon$ when the stack top is $\vdash$ or "(".

4.3.4 Non-LR(0) grammar detection

Consider for instance the ambiguous grammar $\mathcal{G}_{amb}$ of Figure 4.20 for the same simple expression language as before.

A portion of the corresponding full stack reverse recognizing automaton is shown in Figure 4.21, where we can appreciate that actions shift and reduce-2 cannot in general be discriminated, even having reached the stack bottom at state 17. Fortunately, normally we should not need to continue automaton construction until a bottom-of-stack state in order to detect an action conflict; instead, this example shows that the following item set allows

State 1 $\{0, 1, 2, 3, 4, 5, 6\}$

$[1, S' \to \vdash S \dashv \bullet]$
$[2, S \to RT \bullet]$
$[3, R \to S - \bullet]$
$[4, R \to \bullet]$
$[5, T \to (S) \bullet]$
$[6, T \to a \bullet]$
$[0, S' \to \vdash S \bullet \dashv]$
$[0, S' \to \bullet \vdash S \dashv]$
$[0, R \to S \bullet -]$
$[0, T \to (S \bullet)]$
$[0, T \to \bullet (S)]$
$[4, S \to \bullet RT]$
$[0, S \to R \bullet T]$
$[4, S' \to \vdash \bullet S \dashv]$
$[4, R \to \bullet S -]$
$[4, T \to (\bullet S)]$

Figure 4.19: Item sets for discriminating reverse automaton of epsilon grammar.

1:	$S' \to \vdash S \dashv$
2:	$S \to S - S$
3:	$S \to (S)$
4:	$S \to a$

Figure 4.20: $\mathcal{G}_{amb}$: ambiguous grammar for same language as $\mathcal{G}_{exp}$.

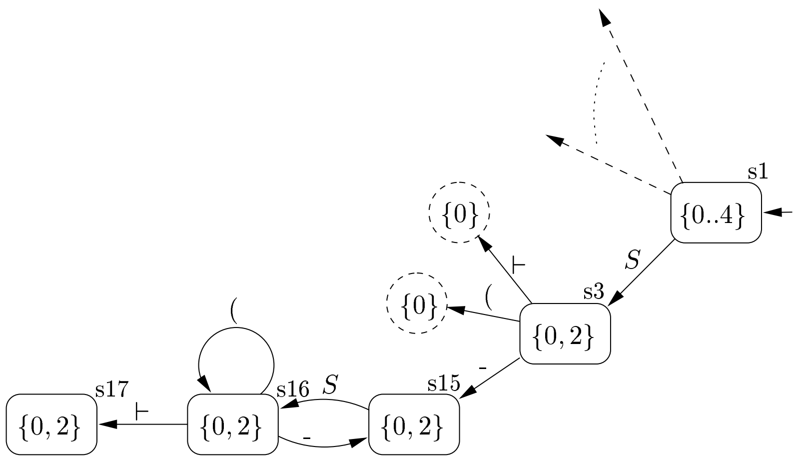


Figure 4.21: Portion of recognizing automaton for ambiguous grammar $\mathcal{G}_{amb}$.

to detect that the grammar is non-LR(0).

$$
\begin{array}{l}
\text{State 3 } \{0,2\} \\
\quad [2, S \to S - \bullet S] \\
\quad [0, S' \to \vdash \bullet S \dashv] \\
\quad [0, S \to \bullet S - S] \\
\quad [0, S \to (\bullet S)] \\
\quad [0, S \to S - \bullet S]
\end{array}
$$

We shall use the term *left-hand context language* to refer to the language of stack prefixes that can be legally found to the left of an item or symbol. In state 3 above, items $[2, S \to S - \bullet S]$ and $[0, S \to S - \bullet S]$, which correspond to different actions, have exactly the same left-hand context language, i.e., both items indicate to read "$-$" then S in the stack, and then the left-hand context language of S. Thus the actual impossibility of discrimination can be already predicted by checking state 3 after its construction, and thus construction of the following conflicting states (like states 15 to 17 in the example) can be avoided.

The method also allows to introduce user-defined conflict resolution for ambiguous grammars, in analogy as how it is done in other parser generators like `yacc`. This will be shown as an extension to our method in Section 8.2. The use of ambiguous grammars is not encouraged in principle, but it may make some specific language structures easier and clearer to write.

4.4 Algorithms

The algorithms developed in this chapter are formally presented in this section, and briefly commented.

4.4.1 Generation algorithm

The generation algorithm for LR(0) grammars is shown in figure 4.22.

Function closure iteratively adds new items to the set when some item has its dot at the left end of the rule rightpart.

Initial state is computed according to the definition. At each state, non-LR(0)-ness of the grammar is checked, and then the sets of actions associated to each next lower stack symbol are computed, and, if unique, used to discriminate the parsing action. The corresponding items are then removed, and the remaining items are used to compute transitions to other states after dot displacement to the left of the transition symbol and closure computation of the new state item sets.

Note that the algorithm is presented with emphasis on clarity rather than efficiency.

```
algorithm Generator of discriminating-reverse LR(0) parsing table
input Reduced grammar 𝒢
output Parsing table ActTrans(state, symbol),
        or Error if the grammar is not LR(0)
function Closure(K);
    I := K
    repeat
        for [i, A→•α] ∈ I do for B→βAγ ∈ P′ do
            add [i, B→β•Aγ] to I
    until no new addition to I
    return I
end Closure;
begin
    Aut:=empty
    K₀ :={[i, A→α•] | A —i→ α ∈ P′} ∪
          {[0, A→α•aβ] | A→αaβ ∈ P′}
    I₀ :=Closure(K₀); add I₀ to Aut
    for I ∈ Aut do
        % non-LR(0) check
        if [i, A→α•β], [j, A→α•γ] ∈ I, i ≠ j then Error
        % decisions upon next stack symbol
        J := I
        for X ∈ V′ do St(X) := {i | [i, A→αX•β] ∈ J}
        for X | St(X) = {i} do
            if i = 0 then ActTrans(I, X):=shift
            else ActTrans (I, X):=reduce(i)
            J := J − {[i , A→αX•β] ∈ J}
        endfor
        % state transitions upon next stack symbol
        for X | [i, A→αX•β] ∈ J do
            K :=Closure({[j, B→γ•Xδ] | [j, B→γX•δ] ∈ J})
            add K to Aut; ActTrans(I, X) := goto(K)
        endfor
    endfor
end
```

Figure 4.22: Generation algorithm of discriminating reverse LR(0) parser.

algorithm Discriminating-reverse LR(0) parser
input Parsing table *ActTrans(state, symbol)*
    and input sequence
output Bottom-up parse of the input sequence,
    or Error if syntactically incorrect
begin
  Stack:=empty; push($\vdash$); read(a)
  repeat
    $q := q_0$; $p := top$
    repeat
      at:=$ActTrans$(q, $Stack$(p))
      case at **of**
        goto(q'): $q := q'$; $p := p - 1$
        shift: push(a); read(a)
        reduce($A \xrightarrow{i} \alpha$): **if** $i = 1$ **then** at:=accept
          else pop(α); push(A)
        nil: Error
      endcase
    until $at \neq$ goto
  until at=accept
end

Figure 4.23: Algorithm of discriminating reverse LR(0) parser.

4.4.2 Parsing algorithm

The parsing algorithm is shown in figure 4.23. It uses a parsing table previously produced according to the above generation algorithm. State item sets themselves are useless at parsing time, and thus states are differently referred to.

In the algorithm, the parsing stack is initialized with a bottom-of-stack marker at the beginning of parsing. Before each parsing action, stack exploration begins from its top at initial state. In each state, the next stack symbol is used for a transition or a parsing action decision. This exploration will eventually finish with deciding a shift, reduce or accept action, or else a syntactic error is found —if, according to the grammar, the scanned stack suffix is illegal. Note that the presence of the rightpart to reduce must be verified before reduction, if the parsing table does not incorporate this section. If it not present, this verification evidently leads to the corresponding error action.

4.5 Formal proofs

In this section, the validity of the discriminating reverse method for LR(0) grammars is formally proven in it essential aspects. First, a formal definition of the construction of single-action reverse automata is given. Then the validity of this construction is proven by means of a characterization theorem of state items. Note that in this chapter a single-automata approach is used in the proofs. In the LR(k) case an all-action combined automata will be used instead.

Definition 4.1 *An LR(0)* actionless *reverse item has the form* $B{\rightarrow}\gamma{\bullet}\varphi$*, where* $B{\rightarrow}\gamma\varphi \in P'$*.* $\square$

Definition 4.2 *An LR(0) (discriminating or action) reverse item has the form* $[i, B{\rightarrow}\gamma{\bullet}\varphi]$*, where* $|P'| \geq i \geq 0$ *and* $B{\rightarrow}\gamma\varphi \in P'$*.* $\square$

In the following formal exposition about the LR(0) case, actionless items will be used for single-action reverse automata. In the more informal previous presentation, both kind of items are used.

Definition 4.3 *Given a grammar* $\mathcal{G}$*, the closure of a set* q *of LR(0) reverse items is defined by:*

$$\mathcal{C}_0(q) = q \cup \{[B{\rightarrow}\gamma{\bullet}\varphi] \mid [A{\rightarrow}{\bullet}\beta] \in q,\ \varphi \underset{\text{rm}}{\Rightarrow}^{*} Av,\ \text{and } B{\rightarrow}\gamma\varphi \in P'\}.$$

$\square$

Note that the above definition is equivalent to the recursive definition in Section 4.1.2.

Definition 4.4 *Given a grammar* $\mathcal{G}$*, the automaton* $\mathcal{A}_0(i)$ *is constructed as follows:*

1. *Initial state:*

$$q_0 = \begin{cases} i = 0: & \mathcal{C}_0(\{A{\rightarrow}\alpha{\bullet}a\beta \mid A{\rightarrow}\alpha a\beta \in P'\}) \\ |P'| \geq i \geq 1: & \mathcal{C}_0(\{A{\rightarrow}\alpha{\bullet} \mid A \xrightarrow{i} \alpha \in P'\}) \end{cases}$$

2. *Transition function:*

$$\delta(q, \sigma) = \begin{cases} \sigma = \varepsilon: & q \\ \sigma = X: & \mathcal{C}_0(\{A{\rightarrow}\alpha{\bullet}X\beta \mid A{\rightarrow}\alpha X{\bullet}\beta \in q\}) \\ \sigma = X\sigma_1: & \delta(\delta(q, \sigma_1), X) \end{cases}$$

$\square$

The following theorem characterizes LR(0) reverse items in terms of their corresponding derivation subtrees, and is most naturally extended to the LR(k) case, although it seems to be possible to put it more simply as follows:

$B{\rightarrow}\beta{\bullet}\rho \in \delta(q_0, \sigma)$ iff $B \xrightarrow{j} \beta\rho$ and $\rho = \sigma y$ or $B{\rightarrow}\beta{\bullet}\rho$ and $\rho \underset{\text{rm}}{\Rightarrow}^{*} \varphi_1 A w \underset{\text{rm}}{\overset{j}{\Longrightarrow}} \sigma y w$, where $y \neq \varepsilon$ when $i = 0$, and $y = \varepsilon$ and $j = i$ when $A \xrightarrow{i} \alpha$.

Its extension to the LR(k) case seems also simple. Another way to put it is the following, although its extension to the LR(k) case seems to be too much complicated.

$$B \to \beta \bullet \rho \in \delta(q_0, \sigma) \text{ iff}$$
$$B \xrightarrow{h} \beta\rho \text{ and } \begin{cases} i = 0: & \rho = \sigma a y_1 \text{ or } \rho \underset{\text{rm}}{\Rightarrow}^* \varphi_1 A w \underset{\text{rm}}{\Rightarrow} \sigma a y_1 w \\ i > 0: & \rho = \sigma \text{ and } h = i \text{ or } \rho \underset{\text{rm}}{\Rightarrow}^* \varphi_1 A w \overset{i}{\underset{\text{rm}}{\Longrightarrow}} \sigma w \end{cases}$$

To interpret the following theorem it is useful to see that, since $\mathcal{G}$ is reduced, $S \underset{\text{rm}}{\Rightarrow}^* \mu B z$, and there are two identical derivations producing the same $\mu\beta\sigma y w z = \mu\varphi\alpha w z$. If the grammar is unambiguous, this is the current rightmost sentential form, where α is the current handle, the current stack contents at a certain moment is $\mu\beta\sigma$ according to the second derivation, and σ the current stack suffix already read by the automaton, with the current stack reverse exploration point shown by the first derivation to be located between β and ρ.

Theorem 4.5 *Given a reduced grammar $\mathcal{G}$ and an automaton $\mathcal{A}_0(i)$ constructed according to Definition 4.4,*

$$B \to \beta \bullet \rho \in \delta(q_0, \sigma) \textit{ iff } \begin{cases} B \xrightarrow{h} \beta\rho, \ \rho \overset{\pi_2}{\underset{\text{rm}}{\Longrightarrow}}{}^* \sigma y w, \\ B \overset{\pi_1}{\underset{\text{rm}}{\Longrightarrow}}{}^* \varphi A w \overset{j}{\underset{\text{rm}}{\Longrightarrow}} \varphi\alpha w, \ \textit{and } \pi_1 j = h\pi_2 \end{cases}$$

where $y \neq \varepsilon$ when $i = 0$, and $y = \varepsilon$ when $A \xrightarrow{i} \alpha$.

Proof. By Convention 1.7, $j = i$ when $A \xrightarrow{i} \alpha$.

"If" part: By induction on $|\sigma|$:

- Induction base with $|\sigma| = 0$. By definition, $\delta(q_0, \sigma) = q_0$. There are two cases:
  - Case $i = 0$. Since $y \neq \varepsilon$, $\rho \underset{\text{rm}}{\Rightarrow}^* y w \neq \varepsilon$, and then $\rho \neq \varepsilon$. There are two cases:
    * Case $\rho = a\rho_1$. Then $B \to \beta \bullet a\rho_1 \in q_0$ by definition.
    * Case $\rho = C\rho_1$. Then $|\pi_2| > 0$, and since $\pi_1 j = h\pi_2$, $\pi_2 = \pi j$ such that

      $$\rho = C\rho_1 \overset{\pi}{\underset{\text{rm}}{\Longrightarrow}}{}^* y_1 A w \overset{j}{\underset{\text{rm}}{\Longrightarrow}} y_1 y_2 w = y w,$$

      where $y_2 = \alpha$. There are two cases:
      · Case $y_1 = \varepsilon$. Then $y_2 = y \neq \varepsilon$, and then $A \to \bullet y_2 \in q_0$ by definition, and since $\rho \overset{\pi}{\underset{\text{rm}}{\Longrightarrow}}{}^* A w$, the theorem holds by definition of $\mathcal{C}_0$.

· Case $y_1 \neq \varepsilon$. Then $|\pi| > 0$. Let j' be the rule in π producing the $1 : y_1 = a$, that is, $\pi = \pi_a j' \pi_b$ such that

$$\rho = C\rho_1 \overset{\pi_a}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \varphi_2 C' w_2 \overset{j'}{\underset{\mathrm{rm}}{\Longrightarrow}} \varphi_2 \varphi_1 a \alpha_1 w_2 \text{ and } \alpha_1 \overset{\pi_b}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* y_1' A w_1$$

where $w_1 w_2 = w$ and $a y_1' = y_1$, what implies $\varphi_2 \varphi_1 = \varepsilon$, and thus $\rho \underset{\mathrm{rm}}{\Rightarrow}{}^* C' w_2$. Since $C' \to \bullet a \alpha_1 \in q_0$ by definition, the theorem holds by definition of $\mathcal{C}_0$.

– Case $A \overset{i}{\longrightarrow} \alpha$. Since $\sigma y = \varepsilon$, $\rho \overset{\pi_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* w$. There are two cases:

∗ Case $|\pi_2| = 0$. Since $\pi_1 j = h$, $j = h$ and $|\pi_1| = 0$, and then $B = \varphi A w$, what implies $\varphi = w = \varepsilon$ and $B = A$, and since $\rho = w = \varepsilon$, $\beta = \alpha$. And the theorem holds because $A \to \alpha \bullet \in q_0$ by definition.

∗ Case $|\pi_2| > 0$. Since $\pi_1 j = h \pi_2$, $\pi_2 = \pi j$ such that

$$\rho \overset{\pi}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* w_1 A w_2 \overset{j}{\underset{\mathrm{rm}}{\Longrightarrow}} w_1 \alpha w_2 = w.$$

But $h\pi = \pi_1$, and thus $B \overset{\pi_1}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \beta w_1 A w_2 = \varphi A w$, and then $w_2 = w$, but since $w_1 \alpha w_2 = w$, $w_1 = \alpha = \varepsilon$. And by definition, $A \to \bullet \in q_0$, and since $\rho \underset{\mathrm{rm}}{\Rightarrow}{}^* A w_2$, the theorem holds by definition of $\mathcal{C}_0$.

- Induction step with the "if" part assumed to hold for $|\sigma| = p$. The reasoning is the same for both cases $i = 0$ ($y \neq \varepsilon$) and $A \overset{i}{\longrightarrow} \alpha$ ($y = \varepsilon$ and $j = i$). Let $|\sigma'| = p + 1$, i.e., $\sigma' = X\sigma$, such that

$$\begin{cases} B' \overset{h'}{\longrightarrow} \beta' \rho',\ \rho' \overset{\pi_2'}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \sigma' y w', \\ B' \overset{\pi_1'}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \varphi' A w' \overset{j}{\underset{\mathrm{rm}}{\Longrightarrow}} \varphi' \alpha w', \text{ and } \pi_1' j = h' \pi_2' \end{cases}$$

Let $q = \delta(q_0, \sigma)$ and $q' = \delta(q_0, \sigma') = \delta(q, X)$ by definition. Since $\rho' \overset{\pi_2'}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* X \sigma y w'$, there are the following two cases:

– There is some ρ such that $\rho' = X\rho$ and $\rho \overset{\pi_2'}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \sigma y w'$. Thus $B' \to \beta' X \rho$, and choosing $\beta = \beta' X$, and since the "if" part is assumed to hold for $|\sigma| = p$, $B' \to \beta' X \bullet \rho \in q$, hence $B' \to \beta' \bullet X \rho \in q'$ by definition of δ.

– There is no ρ such that $\rho' = X\rho$ and $\rho \overset{\pi_2'}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \sigma y w'$. Then $|\pi_2'| > 0$, and thus $\pi_2' = \pi' j$ such that

$$\rho' \overset{\pi'}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \varphi_1 A w' \overset{j}{\underset{\mathrm{rm}}{\Longrightarrow}} \varphi_1 \alpha w' = X \sigma y w'.$$

There are two cases:

- Case $\varphi_1 = \varepsilon$. Then $\alpha = X\sigma y$. Choosing $|\pi_2| = |\pi_1| = 0$, $h = j$, $\beta = X$, and $\rho = \sigma y$, and since the "if" part is assumed to hold for $|\sigma| = p$, $A \rightarrow X \bullet \rho \in q$, and then $A \rightarrow \bullet X\rho \in q'$ by definition of δ, and since $\rho' \underset{\text{rm}}{\Rightarrow}^* Aw'$, the theorem holds by definition of $\mathcal{C}_0$.
- Case $\varphi_1 \neq \varepsilon$. Then $|\pi'| > 0$. Let h be the rule in π' producing the $1 : \varphi_1 = X$, that is, $\pi' = \pi_a h \pi$ such that

$$\rho' \overset{\pi_a}{\underset{\text{rm}}{\Longrightarrow}}{}^* \varphi_2 B z_1 \overset{h}{\underset{\text{rm}}{\Longrightarrow}}{}^* \varphi_2\varphi_3 X\rho z_1 \text{ and } \rho \overset{\pi}{\underset{\text{rm}}{\Longrightarrow}}{}^* \varphi_4 Aw,$$

where $X\varphi_4 = \varphi_1$, what implies $\varphi_2\varphi_3 = \varepsilon$ and $\varphi_4\alpha = \sigma y$ because $\varphi_1\alpha = X\sigma y$. Choosing $\pi_2 = \pi j$ and $\pi_1 = h\pi$,

$$\begin{cases} B \overset{h}{\longrightarrow} X\rho,\ \rho \overset{\pi_2}{\underset{\text{rm}}{\Longrightarrow}}{}^* \sigma y w, \\ B \overset{\pi_1}{\underset{\text{rm}}{\Longrightarrow}}{}^* X\varphi_4 Aw \overset{j}{\underset{\text{rm}}{\Longrightarrow}} X\varphi_4\alpha w, \text{ and } \pi_1 j = h\pi_2, \end{cases}$$

and choosing $\beta = X$ and $\varphi = X\varphi_4$, and since the "if" part is assumed to hold for $|\sigma| = p$, $B \rightarrow X \bullet \rho \in q$, and then $B \rightarrow \bullet X\rho \in q'$ by definition of δ, and since $\rho' \underset{\text{rm}}{\Rightarrow}^* Bz_1$, the theorem holds by definition of $\mathcal{C}_0$.

"Only if" part: By induction on $|\sigma|$:

- Induction base with $|\sigma| = 0$. By definition, $\delta(q_0, \sigma) = q_0$. There are two cases:
  - Case $i = 0$. From the definitions of q_0 and $\mathcal{C}_0$, $B \overset{h}{\longrightarrow} \beta\rho$, with the following two cases:
    - Case $\rho = a\rho_1$. Since $\mathcal{G}$ is reduced, $\rho \overset{\pi_2}{\underset{\text{rm}}{\Longrightarrow}}{}^* at$. There are two cases:
      - Case $|\pi_2| = 0$. The theorem holds choosing $\pi_1 j = h$ and $y = at$.
      - Case $|\pi_2| > 0$. Then $\pi_2 = \pi j$ such that

$$\rho = a\rho_1 \overset{\pi}{\underset{\text{rm}}{\Longrightarrow}}{}^* ay_1 Aw \overset{j}{\underset{\text{rm}}{\Longrightarrow}} ay_1y_2w = at,$$

and the theorem holds choosing $\pi_1 = h\pi$, $y = ay_1y_2$, $\varphi = \beta a y_1$, and $\alpha = y_2$.

    - Case $\rho \overset{\pi}{\underset{\text{rm}}{\Longrightarrow}}{}^* Cw_2$ and $C \overset{j'}{\longrightarrow} a\alpha_1$. Since $\mathcal{G}$ is reduced, $\alpha_1 \overset{\pi_2'}{\underset{\text{rm}}{\Longrightarrow}}{}^* t$. There are two cases:

· Case $|\pi_2'| = 0$. Choosing $\pi_2 = \pi j'$, $\pi_1 = h\pi$, and $j = j'$:

$$\begin{cases} \rho \overset{\pi_2}{\underset{\text{rm}}{\Longrightarrow}}{}^* atw_2, \\ B \overset{\pi_1}{\underset{\text{rm}}{\Longrightarrow}}{}^* \beta C w_2 \overset{j}{\underset{\text{rm}}{\Longrightarrow}} \beta a t w_2, \text{ and } \pi_1 j = h\pi_2, \end{cases}$$

and the theorem holds choosing $y = at$, $w = w_2$, $\varphi = \beta$, $A = C$, and $\alpha = at$.

· Case $|\pi_2'| > 0$. Then $\pi_2' = \pi_1' j$ such that

$$\alpha_1 \overset{\pi_1'}{\underset{\text{rm}}{\Longrightarrow}}{}^* y_1 A w_1 \overset{j}{\underset{\text{rm}}{\Longrightarrow}} y_1 y_2 w_1 = t.$$

Choosing $\pi_2 = \pi j' \pi_1' j$, $\pi_1 = h\pi j' \pi_1'$,

$$\begin{cases} \rho \overset{\pi_2}{\underset{\text{rm}}{\Longrightarrow}}{}^* a y_1 y_2 w_1 w_2, \\ B \overset{\pi_1}{\underset{\text{rm}}{\Longrightarrow}}{}^* \beta a y_1 A w_1 w_2 \overset{j}{\underset{\text{rm}}{\Longrightarrow}} \beta a y_1 y_2 w_1 w_2, \text{ and } \pi_1 j = h\pi_2. \end{cases}$$

And the theorem holds choosing $y = ay_1y_2$, $w = w_1w_2$, $\varphi = \beta a y_1$, and $\alpha = y_2$.

– Case $A \overset{i}{\longrightarrow} \alpha$. From the definition of q_0 and $\mathcal{C}_0$, $B \overset{h}{\longrightarrow} \beta\rho$, with the following two cases:

  * Case $h = i$ and $\rho = \varepsilon$. The theorem holds choosing $|\pi_2| = |\pi_1| = 0$.
  * Case $\rho \overset{\pi}{\underset{\text{rm}}{\Longrightarrow}}{}^* Aw$ and $A \overset{i}{\longrightarrow} \varepsilon$. The theorem holds choosing $\pi_2 = \pi i$, $\pi_1 = h\pi$, and $\varphi = \beta$.

- Induction step with the "only if" part assumed to hold for $|\sigma| = p$. The reasoning is the same for both cases $i = 0$ ($y \neq \varepsilon$) and $A \overset{i}{\longrightarrow} \alpha$ ($y = \varepsilon$ and $j = i$). Let $|\sigma'| = p + 1$, i.e., $\sigma' = X\sigma$, such that $B' \rightarrow \beta' \bullet \rho' \in \delta(q_0, \sigma')$. Let $q = \delta(q_0, \sigma)$ and $q' = \delta(q_0, \sigma') = \delta(q, X)$ by definition. From the definitions of δ and $\mathcal{C}_0$, $B' \overset{h'}{\longrightarrow} \beta' \rho'$, with the following two cases:

  – Case $\rho' = X\rho$ and $B' \rightarrow \beta' X \bullet \rho \in q$. Choosing $\beta = \beta' X$ and since the "only if" part is assumed to hold for $|\sigma| = p$,

$$\begin{cases} B' \overset{h'}{\longrightarrow} \beta' X \rho, \ \rho \overset{\pi_2}{\underset{\text{rm}}{\Longrightarrow}}{}^* \sigma y w, \\ B' \overset{\pi_1}{\underset{\text{rm}}{\Longrightarrow}}{}^* \varphi A w \overset{j}{\underset{\text{rm}}{\Longrightarrow}} \varphi \alpha w, \text{ and } \pi_1 j = h' \pi_2, \end{cases}$$

  and thus $\rho' = X\rho \overset{\pi_2}{\underset{\text{rm}}{\Longrightarrow}}{}^* X \sigma y w$.

– Case $\rho' \overset{\pi}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* Bz_1$ and $B \to X \bullet \rho \in q$. Choosing $\beta = X$, and since the "only if" part is assumed to hold for $|\sigma| = p$,

$$\begin{cases} B \overset{h}{\longrightarrow} X\rho, \ \rho \overset{\pi_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \sigma y w, \\ B \overset{\pi_1}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \varphi A w \overset{j}{\underset{\mathrm{rm}}{\Longrightarrow}} \varphi \alpha w, \ \text{and } \pi_1 j = h\pi_2, \end{cases}$$

and then

$$\begin{cases} \rho' \overset{\pi}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* Bz_1 \overset{h}{\underset{\mathrm{rm}}{\Longrightarrow}} X\rho z_1 \overset{\pi_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* X\sigma y w z_1, \\ B' \overset{h'}{\underset{\mathrm{rm}}{\Longrightarrow}} \beta' \rho' \overset{\pi}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \beta' B z_1 \overset{\pi_1}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \beta' \varphi A w z_1. \end{cases}$$

Choosing $\varphi' = \beta'\varphi$, $w' = wz_1$, $\pi_2' = \pi h \pi_2$, and $\pi_1' = h'\pi\pi_1$, then $\pi_1' j = h'\pi\pi_1 j = h'\pi h \pi_2 = h'\pi_2'$.

□

The following definition establishes the set of items which are valid after having read a stack suffix σ. In the next following lemma it will be shown that his definition is correct, according to the above theorem.

Definition 4.6 *The set of* valid *items after having scanned a stack suffix σ is defined as follows:*

$$\mathrm{Valid}(\sigma) = \{[B \to \beta \bullet \rho] \mid [B \to \beta \bullet \rho] \in \delta(q_0, \sigma)\}.$$

□

By the above theorem and the following lemma, every automaton $\mathcal{A}_0(i)$ recognizes precisely the language of stack suffixes corresponding to action i. Actually, this result might be proved directly in a simpler form, by characterizing just σ:

$$\delta(q_0, \sigma) \neq \emptyset \text{ iff } S' \underset{\mathrm{rm}}{\Rightarrow}{}^* \gamma A u \overset{j}{\underset{\mathrm{rm}}{\Longrightarrow}} \eta \sigma y u$$

where $y \neq \varepsilon$ when $i = 0$, and $y = \varepsilon$ and $j = i$ when $A \overset{i}{\longrightarrow} \alpha$.

This connects with Proposition 2.1.

Lemma 4.7 *Given a reduced grammar $\mathcal{G}$,*

$$S' \underset{\mathrm{rm}}{\Rightarrow}{}^* \gamma A u \overset{j}{\underset{\mathrm{rm}}{\Longrightarrow}} \eta \sigma y u \ \textit{iff} \ \begin{cases} B \overset{h}{\longrightarrow} \beta\rho, \ \rho \overset{\pi_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \sigma y w, \\ B \overset{\pi_1}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \varphi A w, \ \textit{and } \pi_1 j = h\pi_2. \end{cases}$$

Proof. **"If" part:** Since $\mathcal{G}$ is reduced and $\pi_1 j = h\pi_2$,

$$S' \underset{\mathrm{rm}}{\Rightarrow}{}^* \mu B z \overset{\pi_1}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \mu\varphi A w z \overset{j}{\underset{\mathrm{rm}}{\Longrightarrow}} \mu\beta\sigma y w z.$$

And the lemma holds choosing $\gamma = \mu\varphi$, $u = wz$ and $\eta = \mu\beta$.

"Only if" part: Let $S' \overset{\pi_a}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \gamma A u \overset{j}{\underset{\mathrm{rm}}{\Longrightarrow}} \gamma\alpha u = \eta\sigma y u$. There are two cases:

- Case $\sigma y = \varepsilon$. The lemma holds choosing $|\pi_1| = |\pi_2| = 0$.
- Case $\sigma y \neq \varepsilon$. Let h be the rule in $\pi_a j$ producing the first symbol of σy, that is, let $\sigma y = X\sigma_1$ and $\pi_a j = \pi_b h \pi_2$ such that

$$S' \overset{\pi_b}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \mu B z \overset{h}{\underset{\mathrm{rm}}{\Longrightarrow}} \mu\beta X \rho_1 z \text{ and } \rho_1 z \overset{\pi_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \sigma_1 u.$$

  It is chosen $\rho = X\rho_1$. There are two cases:

  - Case $|\pi_2| = 0$. Then $\pi_a j = \pi_b h$, and thus $u = z$ and $\rho_1 = \sigma_1$.
  - Case $|\pi_2| > 0$. Then $\pi_2 = \pi j$ such that $\rho_1 \overset{\pi}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \varphi_1 A w \overset{j}{\underset{\mathrm{rm}}{\Longrightarrow}} \varphi_1 \alpha w$, with $X\varphi_1\alpha = \sigma y$, that is, $\rho \overset{\pi_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \sigma y w$, and choosing $\pi_1 = h\pi$ and $\varphi = \beta X \varphi_1$, the lemma holds.

□

Definition 4.8 *Left-hand context language associated to an LR(0) item.*

$$\mathcal{L}([B \rightarrow \beta \bullet \rho]) = \{\mu\beta \mid S' \underset{\mathrm{rm}}{\Rightarrow}{}^* \mu B z\}$$

□

Theorem 4.9 *Left-hand context language associated to an LR(0) item.*

$$\mathcal{L}([B \rightarrow \beta \bullet \rho]) = \{\vdash \mu_0 \mid \delta(\{[B \rightarrow \beta \bullet \rho]\}, \vdash \mu_0) \neq \emptyset\}$$

Proof. □

Definition 4.10 *For a given grammar $\mathcal{G}$, stack suffix σ, is said to be* compatible *with action i iff there is some stack form $\eta\sigma | x$ compatible with i. A stack suffix with at least one compatible action is said to be* valid. □

For LR(0) grammars, valid full-stack configurations always have only one compatible action.

Definition 4.11 *A stack suffix σ is called* discriminant *iff it has only one compatible action. That compatible action is said to be* discriminated *by σ.* □

Note that for sentences, all stack suffix configurations are necessarily compatible with some action.

Using the above definitions, discriminating reverse LR(0) parsing is characterized by the following theorem.

Theorem 4.12 *The discriminating reverse LR(0) automaton for an LR(0) grammar $\mathcal{G}$ performs a parsing action $\eta' \mathbin{|} z' \overset{i}{\Longrightarrow\!\!|} \eta \bullet \sigma \mathbin{|} z$ if, and only if, σ is the shortest valid suffix of $\eta\sigma$ such that σ discriminates action i. Otherwise, an error is signalled after exploring the shortest σ which is incompatible with any action.*

Proof. □

Next lemma shows that in case of a sequence of reductions, the topmost "working" suffix of the stack always remains correct.

Lemma 4.13 *Let a discriminating reverse LR(0) automaton $\mathcal{A}$ for an LR(0) grammar, $\alpha_2 A \mathbin{|} x \Longrightarrow\!\!|^{+} \alpha_1 \mathbin{|} x$, and let σ_2 be the longest suffix of α_2 consisting only of visited or generated stack nodes during that sequence of actions $\Longrightarrow\!\!|^{+}$. Then $S' \underset{\mathrm{rm}}{\Rightarrow}^{*} \eta\sigma_2 A z$.*

Proof. By induction on the number p of parsing steps:

- Induction base with $p = 1$. Let $\alpha_2 A \mathbin{|} x \Longrightarrow\!\!| \eta_1 \bullet \sigma_1 \mathbin{|} x$. By Theorem 4.5,

$$S' \underset{\mathrm{rm}}{\Rightarrow}^{*} \mu\varphi A z \underset{\mathrm{rm}}{\Rightarrow} \mu\varphi\gamma z = \mu\beta\sigma_1 z.$$

  In case $|\sigma_1| > |\gamma|$, $\varphi = \beta\sigma_2$, otherwise $\sigma_2 = \varepsilon$.

- Induction step with some $p > 1$. Thus we suppose $\alpha_2' A' \mathbin{|} x \overset{i}{\Longrightarrow\!\!|} \eta_1' \bullet \sigma_1' \mathbin{|} x$, $A' \overset{i}{\longrightarrow} \gamma'$, $\eta_1'\sigma_1' = \eta_2\sigma_2 A$, $\eta_2\sigma_2 A \mathbin{|} x \Longrightarrow\!\!|^{p} \alpha_1 \mathbin{|} x$, and $S' \underset{\mathrm{rm}}{\Rightarrow}^{*} \eta\sigma_2 A z$. If $|\gamma'| < |\sigma_2 A| > |\sigma_1'|$, then $\eta\sigma_2' A' z \overset{i}{\underset{\mathrm{rm}}{\Longrightarrow}} \eta\sigma_2'\gamma' z = \eta\sigma_2 A z$, which is, by Theorem 4.12, the only legal possibility after exploring σ_1', and thus the lemma holds. Otherwise, as in case $p = 1$, the lemma holds.

□

As a corollary of the above lemma, a correct parsing stack may only become incorrect as a consequence of a shift action, but never from a reduce action. Moreover, since a sequence of reductions is always correct, it is impossible in an LR(0) grammar to produce an unbounded sequence of reductions from empty. As a consequence, the automaton always stop in finite time in front of a finite input sequence. Furthermore, it will be shown for the general LR(k) case that its parsing time complexity is linear with the length of the input, for sentences as well as for syntactically incorrect input sequences. It will be also shown a possible extension to verify the compatibility of the incoming terminal in case of shift action, in order to obtain for the parsing stack the correct prefix property even in case of erroneous inputs, and thus achieving an early error detection.

Chapter 5

Discriminating Reverse LR(1)

Let us consider now the case of LR(1) grammars. This is a particularly important case in practice and thus we shall give it particular attention. Although in principle a relatively simple extension to the LR(0) case can be initially tried, i.e., to build an automaton that begins exploration from the lookahead terminal instead of the topmost symbol, this solution was rejected[1] because a multiplication of states is provoked in the automaton with many similar sub-automata for the different lookahead symbols. Moreover, this early separation of cases is normally quite useless for our purpose of discrimination. Ideally, and in the same original spirit of parsimony of the method, the lookahead terminal should be used *only if it may suffice* to take a decision. In this way, the reverse LR(1) would continue to resemble very much to the LR(0) automaton. In particular it would continue to be quite small, or even smaller in the sense that, if the lookahead terminal is used, the corresponding LR(0) automaton might be pruned at that point. Thus, we shall develop again a *stack suffix* exploration automaton which will use the next stack symbol or the current lookahead terminal to take a decision. For that purpose, the lookahead information will be incorporated to our items, which for the single-action automata will take initially the form $[A{\rightarrow}\alpha\bullet\beta,\ a]$, although some precision will be needed as it will be shown.

5.1 Single-action single-lookahead recognizer

In order to illustrate the details of the LR(1) construction, we shall begin with a simple case. Let us try to build a reverse recognizing automaton for all legal stacks for a given action when the lookahead is a given one. For instance, let us consider again grammar $\mathcal{G}_{exp}$ of Figure 4.2, and let us try to build again a recognizing automaton for stacks where the next parsing action is to reduce using rule 3 in the presence of ")" as the current lookahead

[1]As suggested by O. Lecarme during discussions with the author.

symbol. Let us tentatively use our new item form. Thus we would write:

State 1
$[S \to T \bullet,)]$

Since the dot is not at the beginning of a rule, a closure computation would add nothing. Now let us see the next state, after transition on T. In principle, we have:

State 2
$[S \to \bullet T,)]$

Now a closure is needed to continue with our automaton construction. But note that we cannot add rule $S' \to \vdash S \dashv$ nor rule $S \to S - T$, for T is reduced to S in the presence of ")". Thus only rule $S \to (T)$ can be used in the closure, and thus we finally get

State 2
$[S \to \bullet T,)]$
$[T \to (\bullet S),)]$

Now, after transition on "(", we get initially:

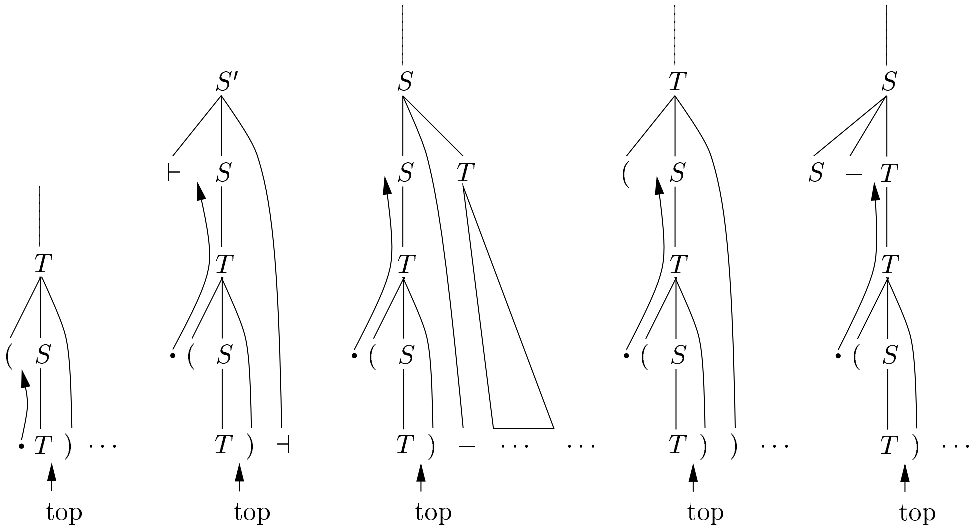
State 3
$[T \to \bullet (S),)]$

Whose closure has to produce indeed $[S \to \bullet T,)]$ and $[S \to S - \bullet T,)]$. But note that the former item is the same as the one we had initially in state 2. But now the situation is different, for the ")" was already considered when in state 2 only item $[T \to (\bullet S),)]$ was included, but not $[S' \to \vdash \bullet S \dashv,)]$ nor $[S \to \bullet S - T,)]$. But now, in state 3, these items are possible for legal stack continuations, and should be included in the closure. Figure 5.1 illustrates both situations when computing the closure, for state 2 and the four cases of state 3. Note that the change consists in that in state 2 the lookahead ")" was not *covered* by the item rightpart T, while in state 3 it is already covered by the item rightpart or, more precisely, by the rightpart portion to the right of the dot; see for instance case (3b). Thus our state 3 would look like

State 3
$[T \to \bullet (S),)]$
$[S \to S - \bullet T,)]$
$[S \to \bullet T,)]$
$[S' \to \vdash \bullet S \dashv,)]$
$[T \to (\bullet S),)]$

5.1.1 The lookahead dot

Now, it is clear that these two cases, lookahead covered and not covered, must be clearly differentiated in the items, in order to be able to construct



(2) (3a) (3b) (3c) (3d)

Figure 5.1: Tree portions for handle T and lookahead ")".

its item set correctly and automatically. For that purpose, in our notation we shall use a *lookahead dot*, such that the noncovered case of state 2 will be represented[2] as $[S \to \bullet T, \bullet)]$, and the covered case of state 3 as $[S \to \bullet T,)\bullet]$.

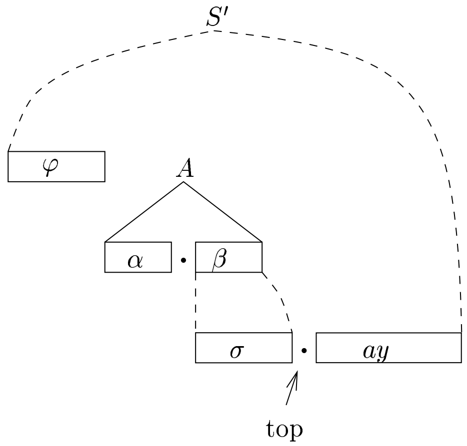
Figure 5.2 illustrates the meaning of the lookahead dot for stack contents $\varphi\alpha\sigma$ after having explored stack suffix σ. You can think of the lookahead dot as placed to the right of the terminal subphrase t still to read produced by β of A's rightpart (i.e., $\beta \underset{\mathrm{rm}}{\Rightarrow}{}^* \sigma t$), such that if $t \neq \varepsilon$ then the item has the form $[A \to \alpha \bullet \beta, a\bullet]$ (i.e., $a = 1 : t$, $t = as$ in lower tree in the figure), and otherwise when $t = \varepsilon$ the item has the form $[A \to \alpha \bullet \beta, \bullet a]$ (i.e., $a = 1 : z$, $z = ay$ in the upper tree in the figure).

Thus, in general, when the dot reaches the left end of the rightpart, as in $[A \to \bullet \alpha, \bullet a]$, rules of the form $B \to \beta A \gamma$ must again be considered for the closure computation. But, as it has just been shown, possibly not every such rule might be compatible with the presence of the item lookahead. In particular, to be compatible, a has to be in $\mathrm{First}(\gamma)$, or in $\mathrm{Follow}(B)$ if $\mathrm{First}(\gamma) \ni \varepsilon$, or both. But even at this point the distinction is important, for if it is precisely the case that $a \in \mathrm{First}(\beta)$, then when the dot reaches the left end of β, *every* rule of the form $C \to \varphi B \mu$ will be compatible, independently of a. The point is whether or not the item lookahead can be *covered* by a phrase of the portion of the item rightpart to the right of the dot.

This is illustrated in Figure 5.3, which shows how the lookahead dot

[2] The original notation, used in some early papers, indicated the noncovered case with the presence of an apostrophe in the item lookahead, as in $[S \to \bullet T,)']$, but the current one is felt to be more intuitive, and is extensible to the LR(k) case.

$[A \to \alpha \bullet \beta, \bullet a]$



$[A \to \alpha \bullet \beta, a \bullet]$

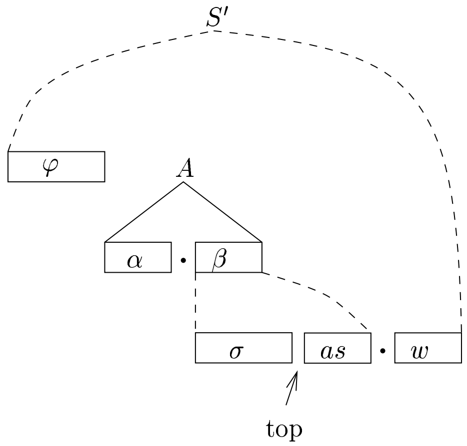


Figure 5.2: Lookahead dot in LR(1) reverse item: noncovered and covered cases.

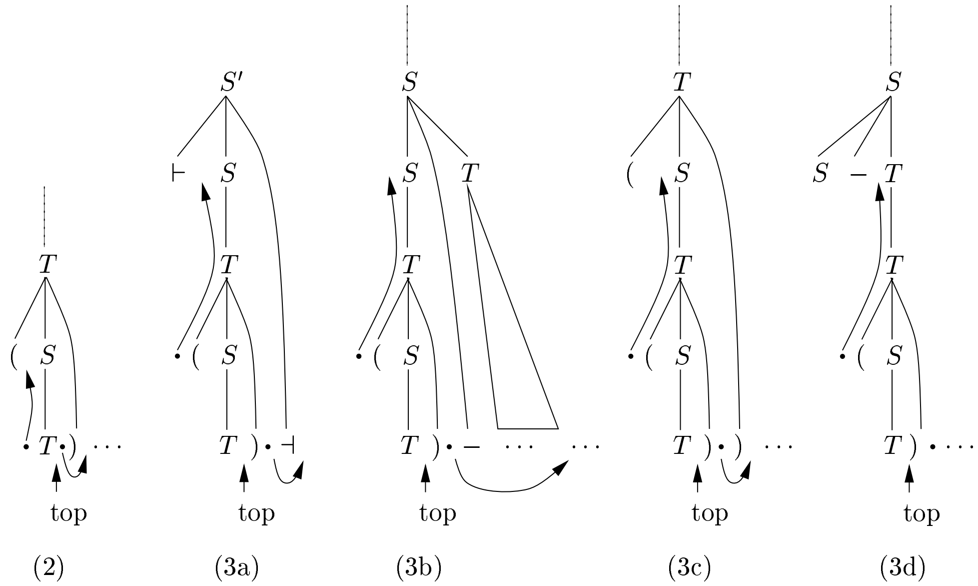


Figure 5.3: Tree portions for handle T and lookahead ")" showing lookahead dot displacement, if any.

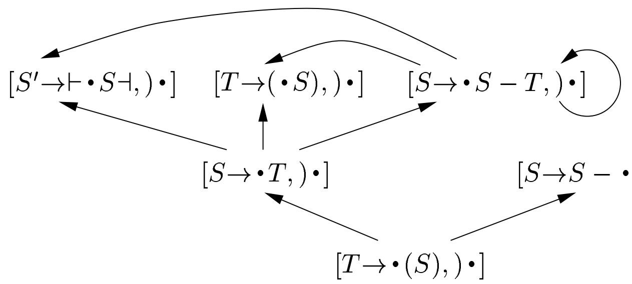


Figure 5.4: Dependency graph from closure of LR(1) reverse items for state 3.

moves during the closure following the corresponding movement of the right-part dot, in the portion trees considered above for automaton states 2 and 3. Diagram of Figure 5.4 shows more schematically the dependence of LR(1) reverse items for state 3 during that closure computation. Note that, as in the LR(0) case, items $[S\to\bullet T,)\bullet]$ and $[S\to\bullet S-T,)\bullet]$ are equivalent, and thus only one of them will be actually used —the former, in the text— in LR(1)-item set of state 3.

The general case for lookahead dot displacement during closure computation is illustrated in Figure 5.5. The lower tree shows that when the lookahead is covered, as in $[A\to\bullet\alpha, a\bullet]$, then it continues to be covered in subsequent items, as in $[D\to\mu\bullet\tau, a\bullet]$. Otherwise, if it is not covered, as in$[A\to\bullet\alpha, \bullet a]$, then it might become covered in a subsequent item of the form $[D\to\mu\bullet\tau, a\bullet]$ (see middle tree of figure) if $\tau \underset{\mathrm{rm}}{\Rightarrow}^{*} Aas$ for some s. Also, as

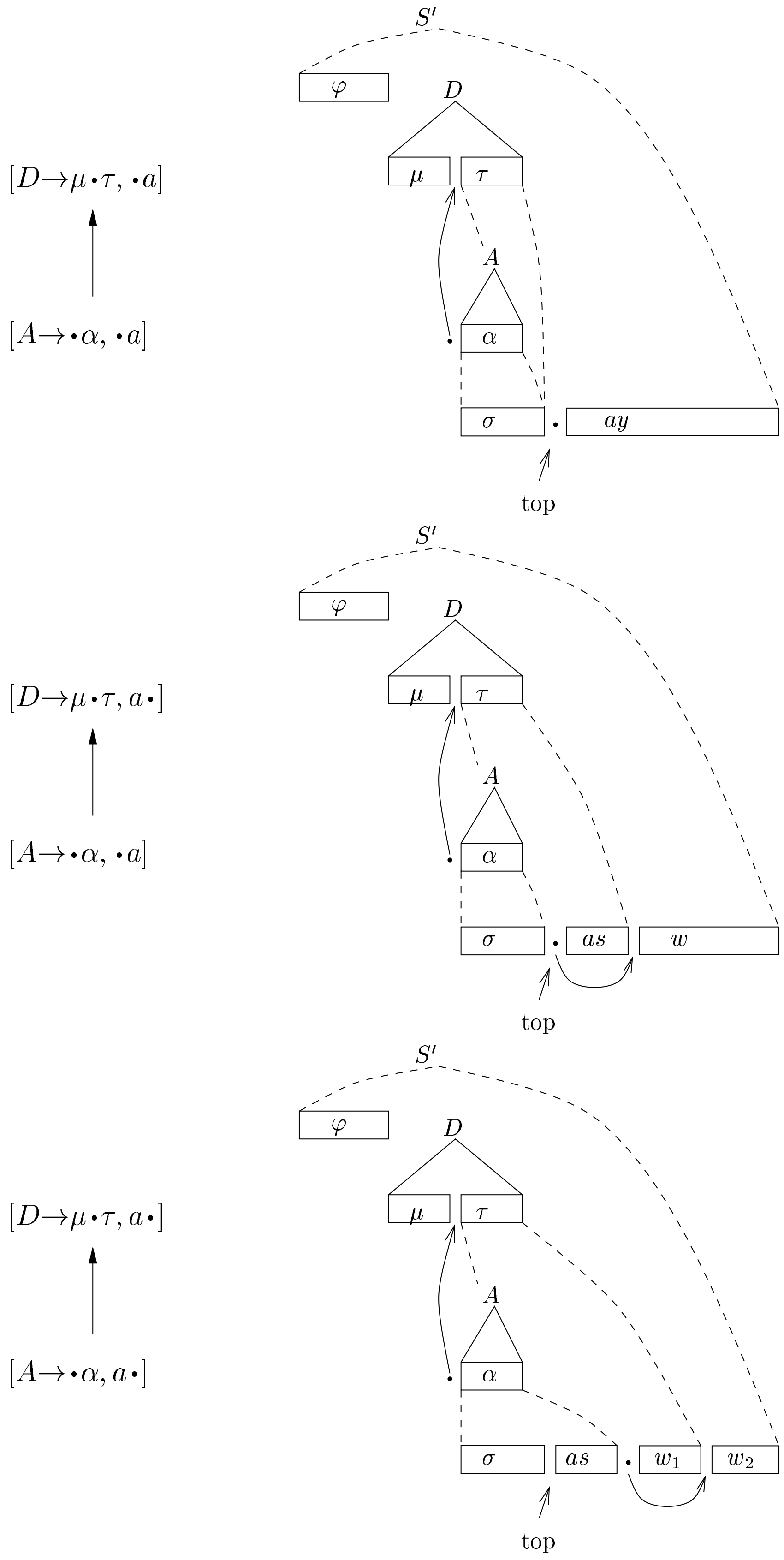


Figure 5.5: Rightpart and lookahead dot displacement during LR(1) reverse item closure.

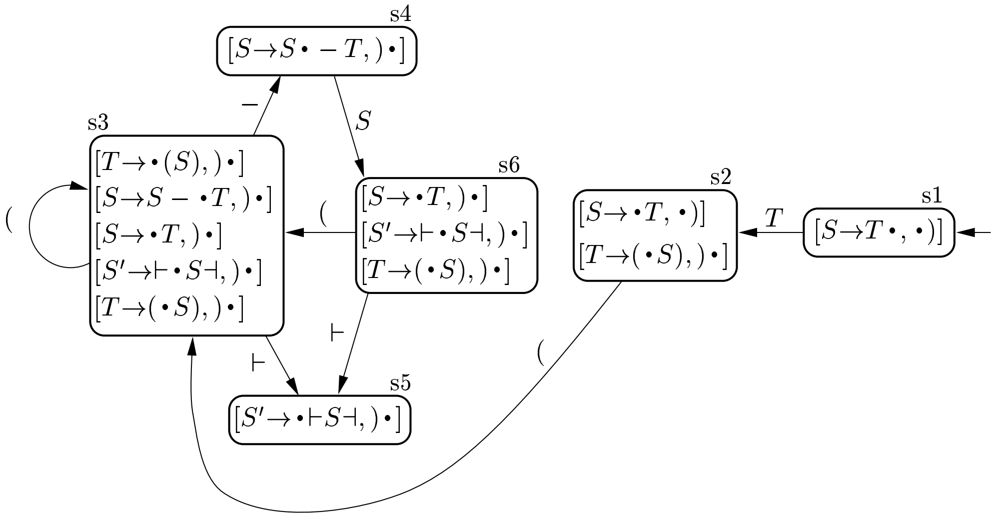


Figure 5.6: Stack recognizing automaton including item sets for action 3 and lookahead ")".

the upper tree of the figure shows, the lookahead might continue noncovered in subsequent items, as in $[D{\rightarrow}\mu\bullet\tau,\bullet a]$, if $\tau \underset{\text{rm}}{\Rightarrow}^* A$. Note that this includes the simple case $\tau = A$ or a chain derivations as $\tau = C_1 \Rightarrow \cdots C_n \Rightarrow A$, or still $\tau \underset{\text{rm}}{\Rightarrow}^* A\eta$, $\eta \underset{\text{rm}}{\Rightarrow}^+ \varepsilon$. Even this latter case of empty derivations immediately to the right of A is irrelevant for the closure under study, for in shift-reduce parsing, α will be first reduced to A before all these empty reductions take place.

Note finally that as consequence of a stack symbol transition, the lookahead dot is not affected before closure. Refer again to Figure 5.2, and consider the case where $\alpha = \alpha_0 X$. There will be a transition on stack symbol X, and thus the rightpart dot will move one symbol to the left, but, as it is evident from the figure, the lookahead dot has to remain in the same position in both the uncovered and covered cases. Concretely, if the item has the form $[A{\rightarrow}\alpha_0 X\bullet\beta,\bullet a]$, it becomes $[A{\rightarrow}\alpha_0\bullet X\beta,\bullet a]$ in the new state after the transition on X, and if it has the form $[A{\rightarrow}\alpha_0 X\bullet\beta, a\bullet]$ it becomes $[A{\rightarrow}\alpha_0\bullet X\beta, a\bullet]$.

Figure 5.6 shows the resulting recognizing automaton including items in this notation. It may be compared with automaton of Figure 4.5, which recognizes the stack for all cases of lookahead for action 3.

5.2 Single-action recognizer

Let us consider again the problem of building a stack recognizing automaton for a reducing action of rule $A \xrightarrow{i} \alpha$. Again, the handle must be on top of stack. But what about the lookahead symbol in the items? Obviously, we must consider every possible legal lookahead, which is precisely the follow set of A, that is, for our initial state we must initially consider all the items of the form $[A \rightarrow \alpha \bullet, \bullet a]$ for every a in Follow(A).

It might seem that nothing has been gained. The point here is now we are in conditions to *distinguish* the different cases for the different lookahead values.

Thus, in our example grammar, to build, using LR(1) reverse items, the stack recognizer for action-3, i.e., to reduce using rule $S \rightarrow T$, the initial state must be the closure of items having as lookahead the symbls in Follow(T) = $\{\dashv, -,)\}$, that is:

$$\begin{array}{l} \text{State 1} \\ \quad [S \rightarrow T \bullet, \bullet \dashv] \\ \quad [S \rightarrow T \bullet, \bullet -] \\ \quad [S \rightarrow T \bullet, \bullet)] \end{array}$$

Figure 5.7 shows the corresponding stack recognizing automaton, including item sets. It may be compared with the previous particular case of lookahead ")" of Figure 5.6, and again with the stack recognizing automaton for the same action using LR(0) reverse items of Figure 4.5. Note that, in this latter case, automata are different, although accepting the same stack language. The reason is indeed that in the LR(1) case the automaton distinguishes the set of lookahead values that are compatible with the state, which will be important in the discriminating automaton. Note that states 2 and 7 of the LR(1) automaton are merged into LR(0) automaton's state 2. If we compare the sets of items for state 2 and 7 of the former automaton, we see that the difference lies in the lookahead symbols, which, although different, provoke no difference to the languages of stack prefixes that can be read from each state, because for those different items the lookahead dot lies to the right, and thus the particular value of the lookahead in those cases does not affect the language, as we have seen. This can be a criterium for state merger, although it might be interesting only in some cases. As we are goint to see, there will be normally few possibilities of merger in the discriminating automaton, and in those cases possibly other simpler merger techniques could be applied directly to the table rather than to the automaton itself.

In essence, the difference lies in the fact that state 2, which is reached after reading stack top T, contains item

$$[S' \rightarrow \vdash \bullet S \dashv, \dashv \bullet],$$

which indicates that lookahead $\dashv$ is compatible with T at that point, but state 7, which is reached after reading one or several "("'s, contains in its place item

$$[S' \rightarrow \vdash \bullet S \dashv,) \bullet],$$

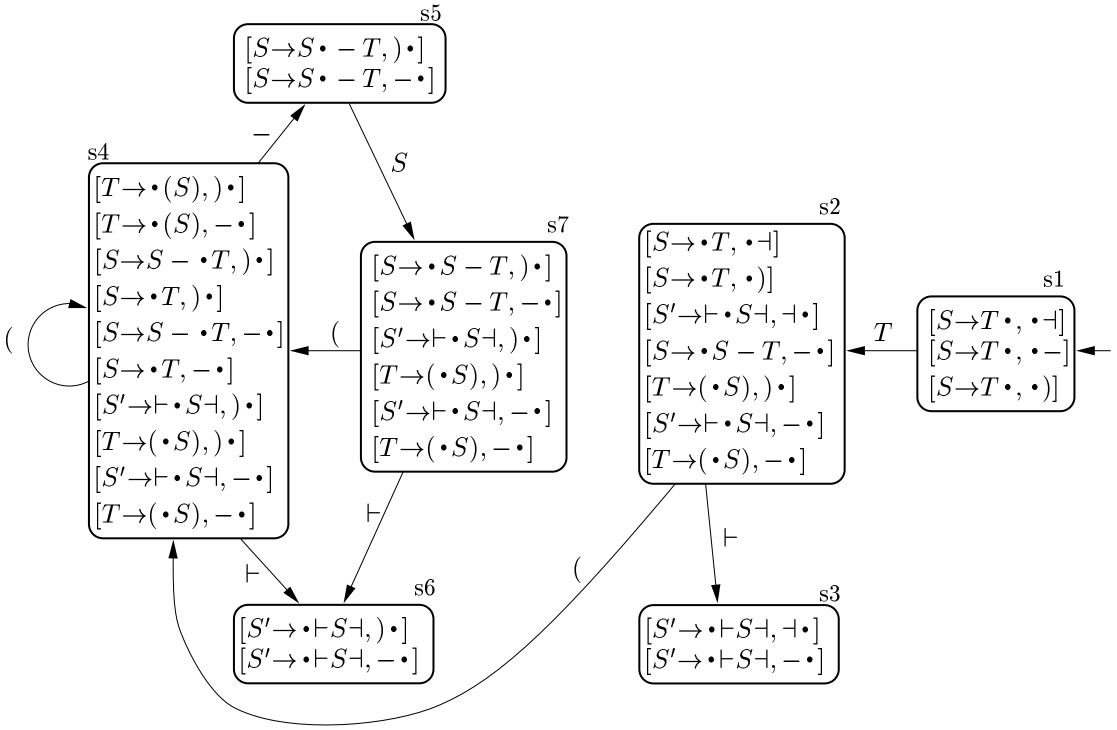


Figure 5.7: Stack recognizing automaton including item sets for action 3.

which indicates that if the bottom of stack is reached ($\vdash$ as next stack symbol), this is only compatible with lookahead ")". If state 2 and 7 were merged, this information, which may be critical for a parsing decision in a non-LR(0) grammar, would be lost.

5.2.1 Equivalence and redundancy of items

In the above examples, the notion of item equivalence has been implicitly used in item set construction. Actually, there is no major difference with the LR(0) case. Two items are equivalent when they are identical except in their respective rightpart section to the right of the dot. In the LR(1) case, this includes that the lookahead (for the reason given above about its importance for discrimination) must be the same, including the position of the lookahead dot, for, as it has been shown; this affects the rules that can be considered during closure computation, and thus the language of stack prefixes that are compatible with that item. In summary, two items are equivalent if they have the form

$$[A \rightarrow \alpha \bullet \beta, \bullet a] \text{ and } [A \rightarrow \alpha \bullet \beta', \bullet a],$$

or the form

$$[A \to \alpha \bullet \beta, a \bullet] \text{ and } [A \to \alpha \bullet \beta', a \bullet].$$

Recall that the only difference between $[A \to \alpha \bullet \beta, \bullet a]$ and $[A \to \alpha \bullet \beta', a \bullet]$ is that in the former case, only rules of the form $B \to \beta A \gamma$ such that a is in First(γ)Follow(B) can be used in the closure, but in the latter case all rules of the form $B \to \beta A \gamma$ can be used. If we compare the languages of stack prefixes that are compatible for each kind of item, the language for $[A \to \alpha \bullet \beta, \bullet a]$ is always included in the language for $[A \to \alpha \bullet \beta', a \bullet]$, which leads us to conclude that, if both are present in the same item set, the former is redundant, for it will produce items (in this and subsequent state item sets) for which there will be always the corresponding item produced by the former. Thus, in summary, we can say that items of the form $[A \to \alpha \bullet \beta, \bullet a]$ are always *redundant* with respect to items of the form $[A \to \alpha \bullet \beta', a \bullet]$, and thus the redundant item might be removed from the set in the presence of the second. This is, for instance, what happened to item $[S \to \bullet T, \bullet -]$ in state 2 of Figure 5.7 after transition on T from item $[S \to T \bullet, \bullet -]$ in state 1. Since the closure produced item $[S \to \bullet S - T, - \bullet]$, $[S \to \bullet T, \bullet -]$ became redundant, and was deleted from the set[3].

5.2.2 Shift action recognizer

Finally, from what has been shown, it is clear how to build a recognizer for shift action. For initial state, as in the LR(0) case, there is an item for each terminal in a rightpart. Now, it is clear that the item lookahead is precisely that terminal, and that the lookahead dot must always be to the right of it, because it is obviously covered by the rule rightpart. Thus, on the following set

$$\begin{array}{l}
[S' \to \bullet \vdash S \dashv, \vdash \bullet] \\
[S' \to \vdash S \bullet \dashv, \dashv \bullet] \\
[S \to S \bullet - T, - \bullet] \\
[T \to \bullet (S), (\bullet] \\
[T \to (S \bullet),) \bullet] \\
[T \to \bullet a, a \bullet]
\end{array}$$

[3]Actually, in this example both items have the rightpart dot at its left end, and thus after closure they become almost useless.

the corresponding closure is computed producing

$$
\begin{array}{l}
\text{State 1} \\
\quad [S' \rightarrow \bullet \vdash S \dashv, \vdash \bullet] \\
\quad [S' \rightarrow \vdash S \bullet \dashv, \dashv \bullet] \\
\quad [S \rightarrow S \bullet - T, - \bullet] \\
\quad [T \rightarrow \bullet (S), (\bullet] \\
\quad [T \rightarrow (S \bullet),) \bullet] \\
\quad [T \rightarrow \bullet a, a \bullet] \\
\quad [S \rightarrow S - \bullet T, (\bullet] \\
\quad [S \rightarrow \bullet T, (\bullet] \\
\quad [S \rightarrow S - \bullet T, a \bullet] \\
\quad [S \rightarrow \bullet T, a \bullet] \\
\quad [S' \rightarrow \vdash \bullet S \dashv, (\bullet] \\
\quad [T \rightarrow (\bullet S), (\bullet] \\
\quad [S' \rightarrow \vdash \bullet S \dashv, a \bullet] \\
\quad [T \rightarrow (\bullet S), a \bullet]
\end{array}
$$

from which the rest of the automaton is produced as before.

To summarize, we formally define below the construction procedure of the reverse stack recognizing automaton for a given action i, $i \geq 0$.

5.2.3 Closure

The closure of a set q of items is defined as follows.

$$
\begin{array}{rcl}
\mathcal{C}(q) = q & \cup & \{[B \rightarrow \beta \bullet C\gamma,\ \bullet a] \mid [C \rightarrow \bullet \varphi,\ \bullet a] \in \mathcal{C}(q) \text{ and } \varepsilon \in \text{First}(\gamma) \text{ and} \\
& & a \in \text{Follow}(B) \text{ and } B \rightarrow \beta C \gamma\} \\
& \cup & \{[B \rightarrow \beta \bullet C\gamma,\ a \bullet] \mid ([C \rightarrow \bullet \varphi,\ a \bullet] \in \mathcal{C}(q) \text{ or} \\
& & [C \rightarrow \bullet \varphi,\ \bullet a] \in \mathcal{C}(q) \text{ and } a \in \text{First}(\gamma)) \text{ and } B \rightarrow \beta C \gamma\}
\end{array}
$$

From a set q of items, a transition on next lower stack symbol X to another item set q' is defined as follows.

$$q' = \delta(q, X) = \mathcal{C}(\{[B \rightarrow \beta \bullet X\gamma,\ a] \mid [B \rightarrow \beta X \bullet \gamma,\ a] \in q\})$$

For the lookahead part of items, we use the convention that if the dot is not represented, it can be in any position. Of course, we consider that item $[B \rightarrow \beta \bullet X\gamma,\ a]$ has the lookahead dot in the same position as the corresponding item $[B \rightarrow \beta X \bullet \gamma,\ a]$.

5.2.4 Initial states

The reverse stack recognizing automaton for reducing $A \rightarrow \alpha$ is built from initial state q_0 as follows

$$q_0 = \mathcal{C}(\{[A \rightarrow \alpha \bullet,\ \bullet a] \mid A \rightarrow \alpha \in P' \text{ and } a \in \text{Follow}(A)\})$$

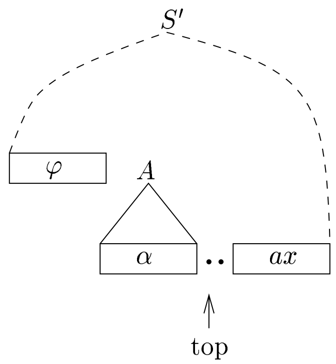


Figure 5.8: Typical parsing tree for reduce action.

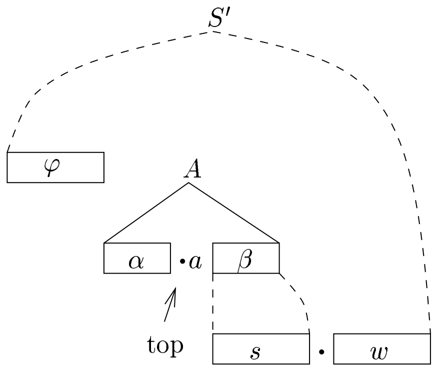


Figure 5.9: Typical parsing tree for shift action.

With respect to the recognizing automaton for the shift action, as in the LR(0) case, we have to consider all the possibilities, which simply means for the LR(1) case to include the lookahead in the items, as follows

$$q_0 = \mathcal{C}(\{[A \rightarrow \alpha \bullet a\beta,\ a \bullet] \mid A \rightarrow \alpha a \beta \in P'\})$$

Of course, the lookahead is the terminal to shift, and the lookahead dot is always put to the right of it, for it is evidently covered by a phrase derived from the rightpart section to the right of the rightpart dot.

Figures 5.8 and 5.9 show the positions of both the rightpart and lookahead dots for initial state items according to the structure of a typical parsing tree, for the reduce and shift actions, respectively.

5.3 Full stack reverse LR(1) recognizer including item action

Now it is easy to apply again the idea for building a single stack recognizing automaton that allows to decide the next parsing action to perform, this time using the new LR(1) reverse items, which means that in each state lookahead compatibility is considered, and thus, even if the stack can be the same for two different actions —i.e., the grammar is not LR(0)—, the lookahead present in the items would permit to decide if the grammar is LR(1).

As in the previous LR(0) case, in order to discriminate amongst all the possible parsing actions, they must be inserted in the items when building this combined automaton. Thus items take the general form $[i,\ A{\to}\alpha{\bullet}\beta,\ {\bullet}a]$ or $[i,\ A{\to}\alpha{\bullet}\beta,\ a{\bullet}]$. The construction of the automaton is presented as follows for the sake of reference, although it should be quite evident to the reader, from what has already been shown.

$$\begin{aligned}\mathcal{C}(q) = q \quad & \cup \quad \{[i,\ B{\to}\beta{\bullet}C\gamma,\ {\bullet}a] \mid [i,\ C{\to}{\bullet}\varphi,\ {\bullet}a] \in \mathcal{C}(q) \text{ and } \varepsilon \in \mathrm{First}(\gamma) \text{ and} \\ & \qquad a \in \mathrm{Follow}(B) \text{ and } B{\to}\beta C\gamma\} \\ & \cup \quad \{[i,\ B{\to}\beta{\bullet}C\gamma,\ a{\bullet}] \mid ([i,\ C{\to}{\bullet}\varphi,\ a{\bullet}] \in \mathcal{C}(q) \text{ or} \\ & \qquad [i,\ C{\to}{\bullet}\varphi,\ {\bullet}a] \in \mathcal{C}(q) \text{ and } a \in \mathrm{First}(\gamma)) \text{ and } B{\to}\beta C\gamma\}\end{aligned}$$

$$q' = \delta(q, X) = \mathcal{C}(\{[i,\ B{\to}\beta{\bullet}X\gamma] \mid [i,\ B{\to}\beta X{\bullet}\gamma] \in q\})$$

$$\begin{aligned}q_0 = \quad & \mathcal{C}(\{[i,\ A{\to}\alpha{\bullet},\ {\bullet}a] \mid A\xrightarrow{i}\alpha \text{ and } a \in \mathrm{Follow}(A)\} \\ & \cup \{[0,\ A{\to}\alpha{\bullet}a\beta,\ a{\bullet}] \mid A{\to}\alpha a\beta\})\end{aligned}$$

As usual, $i = 0$ codes a shift action, and $i > 0$ indicates to reduce using rule $i : A{\to}\alpha$.

The table of Figure 5.10 shows the resulting automaton table (with 68 entries) for grammar $\mathcal{G}_{exp}$, with the set of compatible actions shown besides each state (two states have been considered different when their corresponding item sets are different). The reader might want to compare this table with the direct LR(1) and LALR(1) tables, which are shown in Figures 5.11 and 5.12, with 41 and 28 entries, respectively. Fortunately, the discriminating reverse LR(1) parser will only need 2 states (from 45 states of Figure 5.10) and 10 entries in all, as we shall see.

However, note before that the full LR(1) reverse recognizing automaton can determine, as in the LR(0) case, the parsing action to perform when there remains in a state a unique action after reading some stack suffix —which is signalled by the items all having the same action. Since $\mathcal{G}_{exp}$ is LR(0) this works well, but in general, for a non-LR(0) LR(1) grammar, there are certainly some cases such that two or more actions share some stack configuration, and thus cannot be distinguished except by looking at their corresponding lookahead symbols, which must be necessarily disjoint

		$\vdash$	$\dashv$	S	$-$	T	(	)	a
$\{0,1,2,3,4,5\}$	s1	s8	s2	s6	s7	s3	s9	s4	s5
$\{1\}$	s2			s44					
$\{2,3\}$	s3	s33			s32		s34		
$\{4\}$	s4			s23					
$\{5\}$	s5	s17			s16		s18		
$\{0\}$	s6	s11					s12		
$\{0\}$	s7			s10					
$\{0\}$	s8								
$\{0\}$	s9	s8			s7		s9		
$\{0\}$	s10	s8					s9		
$\{0\}$	s11								
$\{0\}$	s12	s14			s13		s12		
$\{0\}$	s13			s15					
$\{0\}$	s14								
$\{0\}$	s15	s14					s12		
$\{5\}$	s16			s22					
$\{5\}$	s17								
$\{5\}$	s18	s20			s19		s18		
$\{5\}$	s19			s21					
$\{5\}$	s20								
$\{5\}$	s21	s20					s18		
$\{5\}$	s22	s17					s18		
$\{4\}$	s23						s24		
$\{4\}$	s24	s26			s25		s27		
$\{4\}$	s25			s31					
$\{4\}$	s26								
$\{4\}$	s27	s29			s28		s27		
$\{4\}$	s28			s30					
$\{4\}$	s29								
$\{4\}$	s30	s29					s27		
$\{4\}$	s31	s26					s27		
$\{2\}$	s32			s38					
$\{3\}$	s33								
$\{3\}$	s34	s36			s35		s34		
$\{3\}$	s35			s37					
$\{3\}$	s36								
$\{3\}$	s37	s36					s34		
$\{2\}$	s38	s39					s40		
$\{2\}$	s39								
$\{2\}$	s40	s42			s41		s40		
$\{2\}$	s41			s43					
$\{2\}$	s42								
$\{2\}$	s43	s42					s40		
$\{1\}$	s44	s45							
$\{1\}$	s45								

Figure 5.10: Full stack LR(1) reverse recognizer (considering actions in reverse items) for grammar $\mathcal{G}_{exp}$.

	$\dashv$	$-$	(	)	a	S	T
s1			s4		s5	2	3
s2	s6	s7					
s3	r3	r3					
s4			s10		s11	8	9
s5	r5	r5					
s6	r1						
s7			s4		s5		12
s8		s14		s13			
s9		r3		r3			
s10			s10		s11	15	9
s11		r5		r5			
s12	r2	r2					
s13	r4	r4					
s14			s10		s11		16
s15		s14		s17			
s16		r2		r2			
s17		r4		r4			

Figure 5.11: Direct LR(1) parsing table for grammar $\mathcal{G}_{exp}$

	$\dashv$	$-$	(	)	a	S	T
s1			s4		s5	2	3
s2	s6	s7					
s3	r3	r3		r3			
s4			s4		s5	8	3
s5	r5	r5		r5			
s6	r1						
s7			s4		s5		9
s8		s7		s10			
s9	r2	r2		r2			
s10	r4	r4		r4			

Figure 5.12: Direct LALR(1) parsing table for grammar $\mathcal{G}_{exp}$

$$\begin{array}{ll} 1: & S' \to \vdash E \dashv \\ 2: & E \to E + T \\ 3: & E \to T \\ 4: & T \to T * F \\ 5: & T \to F \\ 6: & F \to (E) \\ 7: & F \to a \end{array}$$

Figure 5.13: $\mathcal{G}_{etf}$: LR(1) non-LR(0) grammar for expression language.

because of the LR(1) property. It is evident that the automaton can, after having read the whole stack plus the bottom-of-stack marker $\vdash$, decide what is the correct action from the (action, lookahead) pairs in its state items. But, although LR(1) discrimination is obtained, to have to read the whole stack in this cases is a comparatively much more costly solution in terms of automaton size and exploration time. Of course, this decision can usually be taken much earlier in the stack suffix scan, as it is revealed by simple observation of action sets associated to each state in Figure 5.10. The next section is devoted to such an analysis of the information on each state in order to decide as early as possible.

5.4 The discriminating reverse parser

For our analysis we shall consider the non-LR(0)[4] LR(1) grammar of Figure 5.13. The item set for state 1 of Figure 5.14 has been built according to the procedure of the previous section for the initial state. From the point of view of action decision, each item will be viewed as a triple

$$(\text{action, stack symbol, lookahead}),$$

e.g., an initial state item such as $[0, E \to E + \bullet T, a \bullet]$ can be viewed as the triple $(0, +, a)$. This means that, in initial state, the shift action is compatible with stack symbol $+$ and lookahead a, both present at the same time. From these triples we shall calculate, for each

$$(\text{stack symbol, lookahead})$$

pair, its corresponding set of compatible actions.

The three tables shown in Figure 5.15 contain these sets of compatible actions, which are built as follows, considering all the triples from each state of Figure 5.14. For instance, let us consider the table entry for state 1 corresponding to stack symbol T and lookahead $\dashv$: it contains the set of all actions i that correspond to triples of the form $(i, T, \dashv)$, that is, to items

[4] See its corresponding discriminating reverse LR(0) generation listing in Appendix A.2.

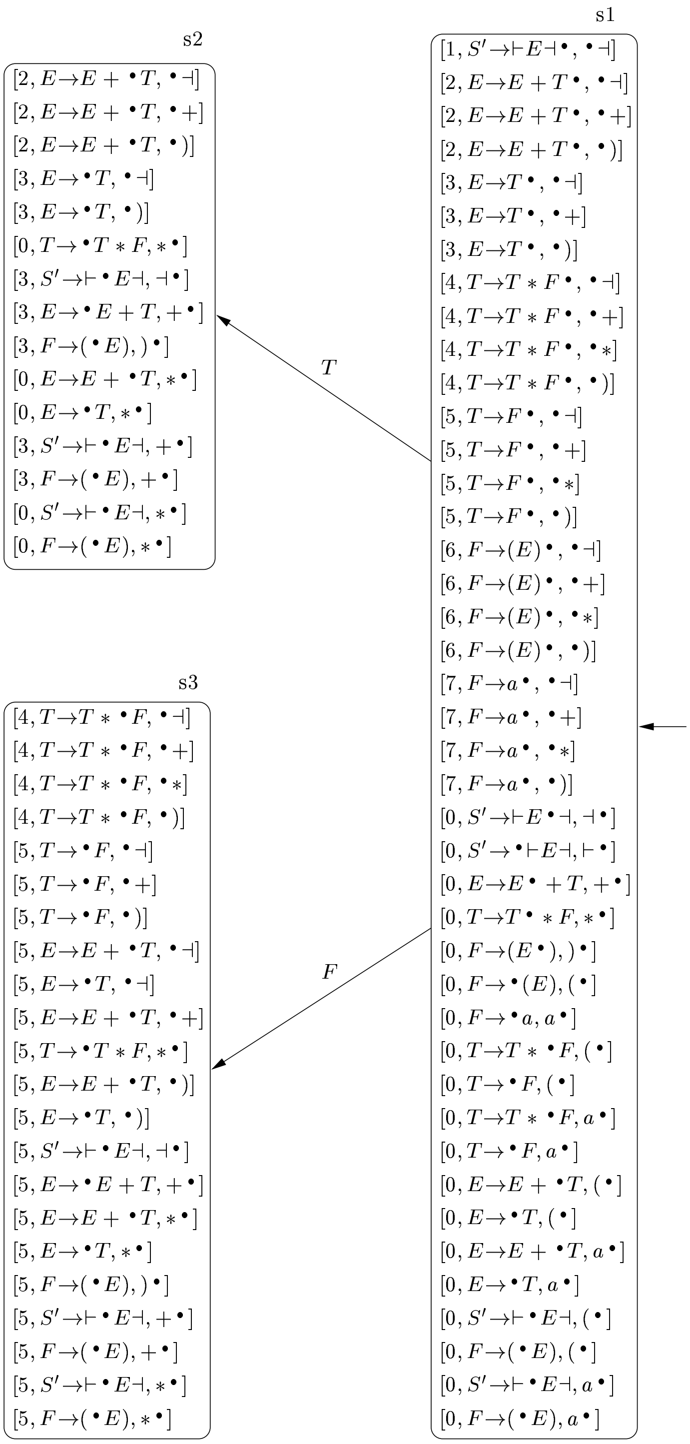


Figure 5.14: Discriminating reverse LR(1) automaton for grammar $\mathcal{G}_{etf}$.

s1	$\dashv$	$+$	$*$	(	)	a
$\vdash$	$\emptyset$	$\emptyset$	$\emptyset$	$\{0\}$	$\emptyset$	$\{0\}$
$\dashv$	$\{1\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$+$	$\emptyset$	$\emptyset$	$\emptyset$	$\{0\}$	$\emptyset$	$\{0\}$
$*$	$\emptyset$	$\emptyset$	$\emptyset$	$\{0\}$	$\emptyset$	$\{0\}$
(	$\emptyset$	$\emptyset$	$\emptyset$	$\{0\}$	$\emptyset$	$\{0\}$
)	$\{6\}$	$\{6\}$	$\{6\}$	$\emptyset$	$\{6\}$	$\emptyset$
a	$\{7\}$	$\{7\}$	$\{7\}$	$\emptyset$	$\{7\}$	$\emptyset$
E	$\{0\}$	$\{0\}$	$\emptyset$	$\emptyset$	$\{0\}$	$\emptyset$
T	$\{2,3\}$	$\{2,3\}$	$\{0\}$	$\emptyset$	$\{2,3\}$	$\emptyset$
F	$\{4,5\}$	$\{4,5\}$	$\{4,5\}$	$\emptyset$	$\{4,5\}$	$\emptyset$

s2	$\dashv$	$+$	$*$	)
$\vdash$	$\{3\}$	$\{3\}$	$\{0\}$	$\emptyset$
$+$	$\{2\}$	$\{2\}$	$\{0\}$	$\{2\}$
(	$\emptyset$	$\{3\}$	$\{0\}$	$\{3\}$

s3	$\dashv$	$+$	$*$	)
$\vdash$	$\{5\}$	$\{5\}$	$\{5\}$	$\emptyset$
$+$	$\{5\}$	$\{5\}$	$\{5\}$	$\{5\}$
$*$	$\{4\}$	$\{4\}$	$\{4\}$	$\{4\}$
(	$\emptyset$	$\{5\}$	$\{5\}$	$\{5\}$

Figure 5.15: Tables of actions which are compatible with lookahead and next stack symbol corresponding to states of Figure 5.14.

of the form $[i, A \rightarrow \alpha T \bullet \beta, \dashv]$ —note that the position of the lookahead dot is irrelevant, for it is only needed for the closure computation, and does not affect the fact that $\dashv$ is the item lookahead—, which are in our case $[2, E \rightarrow E + T \bullet, \bullet\dashv]$ and $[3, E \rightarrow T \bullet, \bullet\dashv]$. That is, we have the triples $(2, T, \dashv)$ and $(3, T, \dashv)$, and thus the table entry for $(T, \dashv)$ contains $\{2, 3\}$. Although in this example there is only one item for each triple (i, X, b), in general there may be several items of the same form $[i, A \rightarrow \alpha_j T \bullet \beta_j, \dashv]$ with different values of α_j. Empty sets evidently correspond to entries for which there is no triple of the corresponding form from the state.

There are still some items in state 1 which cannot correspond to any triple, since their rightpart dot is at the left end. In all but one case they are irrelevant, since precisely the corresponding triple information lies in the items that they produce during closure computation. For instance, the closure for item $[0, F \rightarrow \bullet a, a \bullet]$ generated the following items (which are included in Figure 5.14):

$$
\begin{array}{l}
[0, T \to T * \bullet F, a \bullet] \\
[0, T \to \bullet F, a \bullet] \\
[0, E \to E + \bullet T, a \bullet] \\
[0, E \to \bullet T, a \bullet] \\
[0, S' \to \vdash \bullet E \dashv, a \bullet] \\
[0, F \to (\bullet E), a \bullet]
\end{array}
$$

From them, triples $(0, *, a)$, $(0, +, a)$, $(0, \vdash, a)$, and $(0, (, a)$ are obtained.

The exception to the rule are bottom-of-stack items of the form

$$[S' \to \bullet \vdash S \dashv, b],$$

since their closure produces no new item, yet there is valuable context information in their lookahead b. A solution for this would be to use an augmentation scheme like $S' \to \vdash \vdash S \dashv$. This will turn out to be unnecessary, because the above item form can appear in only two circumstances:

- As a shift in initial state, as is our case in state 1, in the form

  $$[S' \to \bullet \vdash S \dashv, \vdash \bullet].$$

  This simply means that at the beginning of parsing (and only then) the beginning-of-file marker $\vdash$ has to be shifted to the currently empty stack. Since the parsing algorithm actually initializes the stack to contain $\vdash$, this shift action becomes useless. Thus we shall ignore this action in particular.

- At a bottom-of-stack state in the form $[S' \to \bullet \vdash S \dashv, c \bullet]$ for some c. For the ongoing discussion, we may ignore this exception, which will be trivially solved in the solution to the decision problem described in next section.

From these tables of compatible actions we can conclude that if some set is empty, this would mean that, in the current state, this combination is impossible; see for instance in state 1 the set for pair $(+, +)$, whose combination is clearly illegal. But if the set contains a single action, then this would mean that this single action is the only legal possibility, according to the grammar, for that particular (stack symbol, lookahead) pair. For example, for pair $(a, +)$, the only possibility is to reduce using rule $7 : F \to a$. Now, it is clear that many decisions can be taken on such pairs in each state. The number of states which are visited to take a decision would be reduced in comparison with the solution given below. Nevertheless, this approach would possibly imply to build a table for each state, which is considered unnecessarily costly.

However, a closer analysis permits to realize that, actually, many decisions can be taken on just one symbol, instead of a pair. For instance, lookahead "(" or a by themselves allow to decide to shift, without even looking at the stack symbol. But what happens if the current stack symbol is illegal for the current lookahead symbol? This simply means that there is a syntactic error, but that, at this point, passes unnoticed. This is considered a natural compromise in our approach, since the stack is not going to be completely verified anyway. The only real danger would be to decide and then make an illegal reduction, but this cannot happen because, as in the LR(0) case, reductions are always verified if necessary. In the same vein, stack symbols permit to decide by themselves, without having to use the lookahead, as is actually the case in initial state for all stack symbols with the exception of T and F.

5.4.1 The deferred lookahead solution

There are several possibilities to solve this decision problem. We have chosen one of them to investigate and implement which is straightforward and seems the most natural allowing a minimal stack suffix exploration. Other possibilities are mentioned in Section 8.3. Our decision mechanism works, in each state, as follows:

1. The current lookahead is used to try to take a decision.
2. If step 1 fails, then the next stack symbol is used to take either a decision, a transition to a new state, or to signal a syntactic error.

That is, if it is possible, the decision is taken on the lookahead only. If it is not, and knowing that, the value of the stack symbol may allow to decide next parsing action, or has a value which is illegal in this context, giving place to the detection of a syntactic error, or else it does not permit to decide yet and deeper exploration is needed.

As a result, during the decision process for each action, the parser explores some stack suffix from the stack top, using in fact a standard finite-state automaton which follows a sequence of transitions on successive stack symbols, until a decision is taken at some state. The decision may be finally taken on the current lookahead value or on the value of the next stack symbol.

Let us see now in detail how to build our parsing table, according to this decision mechanism, with the help of the tables of compatible actions presented in the previous section.

First, the set of compatible actions is computed for each terminal symbol as lookahead. This is equivalent to compute the union of all sets for each column, as in the table for state 1 in Figure 5.15 in page 88. If such a set of lookahead compatible actions contains a single element for a given lookahead value, this means that this action can be decided if during parsing the current lookahead has that value. Note that this does not allow in all cases to detect an error on the lookahead, because if the current lookahead value cannot decide but is legal, the presence of a legal stack symbol is possible in general. For instance, $*$ does not allow to decide in the referred table, but there are several stack symbols that can legally appear in the presence of that lookahead. In our example for state 1, this step produces the decision to shift if lookahead is "(" or a.

Then, the set of compatible actions is computed for each grammar symbol appearing as next lower stack symbol. Note that if this second step is reached at parsing time, this means that the lookahead *has not decided* in the previous step, i.e., we know that all those cases can be excluded from our set of possible cases, and thus the columns for those lookahead values that permit to decide in the first step need not be considered at this point. In our example, these are columns for "(" and a. Actually, as a consequence, the sets are considered empty after excluding those columns. Moreover, in our example, if the next stack symbol is one of $\vdash$, $+$, $*$ or "(", it implies a syntactic error, for at this point it is sure that the current lookahead is not "(" nor a, and thus we are necessarily in a case of null entry in that table. Accordingly, the compatible action sets for each row are computed, and if the action is unique for some stack symbol, this action can be decided if during parsing the next stack symbol has that value. Note that, as a consequence of excluding those columns, in some cases, the number of resulting compatible actions for a row may become reduced to a single action, while before there were several different actions, all of them but one removed with the columns.

In our example, stack symbol $\dashv$ decides action 1 (i.e., to accept the input and end parsing), and symbols ")", a, and E decide, respectively, actions 6, 7, and 0. Finally, symbols T and F provoke a state transition, since it has not been possible to decide the action on them in this state. Note that a case like $(T, *)$ permits to decide a shift in state 1, but this would have implied to perform the test on a number of pairs of symbols. Nevertheless, in these cases, after state transition, as on T in our example, in the next state, the lookahead symbol, as $*$ in the example, has only one possible compatible action, which is thus immediately decided. This is what happens in state 2, as we shall see below.

In summary, the operations performed from the stack-symbol$\times$lookahead tables of compatible actions are as follows:

1. Build lookahead decisions for lookahead columns with a single action. Remove these columns.

2. Build stack-symbol decisions for stack-symbol rows with a single action, and build transitions for the remaining nonempty rows.

From the point of view of the set of items, the above operations have an immediate effect, if we view them as triples (action, stack symbol, lookahead). Since they represent the different possible cases that are compatible with the current state of exploration, removal of a column means removal from the set of those items containing the lookahead of that column. Then, removal of a row after decision generation means removal of those items whose symbol to the left of the dot corresponds to that row. Thus, the remaining items correspond to those cases still undecided. This remaining set is shown as follows.

$$
\begin{array}{l}
[2, E \to E + T \bullet, \bullet \dashv] \\
[2, E \to E + T \bullet, \bullet +] \\
[2, E \to E + T \bullet, \bullet)] \\
[3, E \to T \bullet, \bullet \dashv] \\
[3, E \to T \bullet, \bullet +] \\
[3, E \to T \bullet, \bullet)] \\
[4, T \to T * F \bullet, \bullet \dashv] \\
[4, T \to T * F \bullet, \bullet +] \\
[4, T \to T * F \bullet, \bullet *] \\
[4, T \to T * F \bullet, \bullet)] \\
[5, T \to F \bullet, \bullet \dashv] \\
[5, T \to F \bullet, \bullet +] \\
[5, T \to F \bullet, \bullet *] \\
[5, T \to F \bullet, \bullet)] \\
[0, T \to T \bullet * F, * \bullet]
\end{array}
$$

This set actually consists of all items for actions 2, 3, 4, and 5, plus item $[0, T \to T \bullet * F, * \bullet]$. It is from this remaining set that the transitions on T and F have been computed, producing, after closure, the sets of items for states 2 and 3 shown in Figure 5.14 of page 87, whose corresponding action set tables are shown in Figure 5.15 of page 88. Items for actions 4 and 5 go to state 3 and the rest of items go to state 2. The same construction as described for state 1 is applied to states 2 and 3. Note that in state 2, the fact that the lookahead is used first permits to decide next on the stack symbol, which would not have been possible in the reverse order. Since no transition to new states is produced in the corresponding table rows, no more states are needed for this grammar, and thus automaton construction ends.

The resulting discriminating reverse LR(1) table is shown in Figure 5.16, containing 16 table entries. The table is divided in two sections. The left-hand section corresponds to the lookahead decisions. At each state, it is entered with the current lookahead symbol. If no action entry is found, then the right-hand section is entered with the next stack symbol. There, an action can be decided or a transition to a new state. If a null entry is found in this right-hand section, then a syntactic error is detected.

The parsing table shown in the Figure corresponds precisely to the construction process just described. For instance, in state 2, shift actions — coded as 0— correspond to lookahead values "(" and a, in accordance with columns of 0's for "(" and a in Figure 5.15.

For the sake of comparison, both direct LR(1) and LALR(1) parsing tables for that same grammar $\mathcal{G}_{etf}$ are presented in Figures 5.17 and 5.18,

	$\dashv$	$+$	$*$	(	)	a	$\vdash$	$\dashv$	E	$+$	T	$*$	F	(	)	a
s1				0		0		1	0		s2		s3		6	7
s2			0				3			2				3		
s3							5			5		4		5		

Figure 5.16: Discriminating reverse LR(1) parsing table for grammar $\mathcal{G}_{etf}$.

	$\dashv$	$+$	$*$	(	)	a	E	T	F
s1				s5		s6	2	3	4
s2	s7	s8							
s3	r3	r3	s9						
s4	r5	r5	r5						
s5				s13		s14	10	11	12
s6	r7	r7	r7						
s7	r1								
s8				s5		s6		15	4
s9				s5		s6			16
s10		s18			s17				
s11		r3	s19		r3				
s12		r5	r5		r5				
s13				s13		s14	20	11	12
s14		r7	r7		r7				
s15	r2	r2	s9						
s16	r4	r4	r4						
s17	r6	r6	r6						
s18				s13		s14		21	12
s19				s13		s14			22
s20		s18			s23				
s21		r2	s19		r2				
s22		r4	r4		r4				
s23		r6	r6		r6				

Figure 5.17: Direct LR(1) parsing table for grammar $\mathcal{G}_{etf}$

with 72 and 46 table entries, respectively.

Returning to the special case of bottom-of-stack items of the form

$$[S' \to \bullet \vdash S \dashv, c \bullet],$$

note that since symbols S' and $\vdash$ are different from other symbols in the grammar, in such a bottom-of-stack state, only items of the form $[S' \to \bullet \vdash S \dashv, c_j \bullet]$ can appear for a $\vdash$ transition, and then items can only vary on the different c_j. Thus discrimination can only take place on the lookahead symbol. According to the above considerations, the generation algorithm (shown below) for the discriminating reverse parser discriminates first on the lookahead, and then on the stack. If it could not discriminate, then the grammar is not LR(1), which is detected before the state item set is analyzed to generate its corresponding row. In fact, the generation algorithm shown

	$\dashv$	$+$	$*$	(	)	a	E	T	F
s1				s5		s6	2	3	4
s2	s7	s8							
s3	r3	r3	s9		r3				
s4	r5	r5	r5		r5				
s5				s5		s6	10	3	4
s6	r7	r7	r7		r7				
s7	r1								
s8				s5		s6		11	4
s9				s5		s6			12
s10		s8			s13				
s11	r2	r2	s9		r2				
s12	r4	r4	r4		r4				
s13	r6	r6	r6		r6				

Figure 5.18: Direct LALR(1) parsing table for grammar $\mathcal{G}_{etf}$

below will use a two-step approach in the construction of the actions for each state, which is equivalent to the process shown here, but which works also for this particular case.

The above procedure is not optimal in general in terms of space for the selected order of tests. Refer again to state 1 of Figure 5.15 of page 88, where the use of the lookahead can be avoided at all, and all decisions could have been made on the stack symbol. Although in this case this is a worst solution in terms of space, in other cases this may produce some gain in the number of entries, but note that it would be at the cost of some parsing time, since the decision would normally be delayed after lookahead check. See Section 8.3 for other variants.

We can conclude that, when exploring with this mechanism of deferred lookahead, in a state, after having read a stack portion, there are three cases for the current lookahead: it is legal and discriminatory (then a decision is taken on it at the state); it is legal but not discriminatory (stack search continues, but the stack suffix plus lookahead is still compatible with items); or it is illegal, i.e., not compatible with the stack tail already read; but then an error has not already been found, although it could have been, for instance, in an exploration beginning from the lookahead. In this latter case, the items are only compatible with the stack suffix, not including the lookahead.

Note that the fact of using the lookahead can produce a smaller automaton for an LR(0) grammar than the discriminating reverse LR(0) automaton, if it causes the elimination of subsequent states present in the LR(0) automaton.

The table of Figure 5.19 shows the corresponding item sets of the discriminating portion of the LR(1) reverse automaton for grammar $\mathcal{G}_{exp}$, whose tables of compatible action sets are shown in Figure 5.20, which leads after the same procedure as described above to the parsing table, containing 10 entries, shown in Figure 5.21. The interested reader might want to compare them with the corresponding LR(0) case, which was shown in Figures 4.14 and 4.13 (see page 54). Of course, table size compares very favorably with the corresponding tables for full LR(1) reverse recognizer of Figure 5.10 (see page 84), and direct LR(1) (Figure 5.11) and LALR(1) (Figure 5.12).

5.4.2 Parsing

Let us first see the parsing example shown in Figure 5.22 of page 98 for sentence

$$\vdash a + (a + a) * a \dashv$$

from grammar $\mathcal{G}_{etf}$ of Figure 5.13 of page 86, whose discriminating reverse LR(1) parsing table is shown in Figure 5.16 of page 93.

According to the parsing algorithm shown below, to decide each action, the automaton is restarted from its initial state in order to scan the stack from its top until a decision is found, with state transitions performed on successive stack symbols. As the figure shows, there are not actually many transitions, for decisions are taken usually quite rapidly. In each state, first the current lookahead (i.e., the leftmost symbol of the rest of input shown in the rightpart of the figure) is tried with the left-hand section of the state row in the parsing table. If no action is found there, then the current stack symbol is used for the left-hand section of the state row. The corresponding action is shown below it in the figure. Actions marked with an asterisk, as the topmost action 0^*, indicate that it is taken on the lookahead, rather than on the stack symbol. For instance, the second action, reduce using rule 7 ($F \rightarrow a$), is taken on topmost a, since lookahead $+$ offers no decision in state 1. The next action, reduce using rule 5 ($T \rightarrow F$), is taken after transition on F in state 1 to state 3, where lookahead $+$ is again useless, and then the decision is finally taken on the the current stack symbol $\vdash$. In some reductions, like rule 2 (see next to the middle of the figure) $E \rightarrow E + T$, some symbol(s) need to be verified, since reducing actions can sometimes be taken before the left end of the handle. These verification steps are represented in the figure with []'s. At the end of this parsing process, since the example input is a correct sentence, the automaton takes decision 1, and successfully ends parsing.

Note that exploration depth does not depend most often (never in this grammar) on the depth of the stack. Moreover, the depth of exploration is usually very small for most grammars and sentences. In the example, the parser has performed 10 shifts and 14 reductions. The average length of reductions is 1.57 symbols, while the average length of rightparts is 2. The average stack exploration depth per shift action is 0.5 symbols, i.e., less than 1, since much of these (shift) decisions are just taken on the lookahead. Finally, the average depth of stack exploration per reduce action is 1.64

State 1 $\{0, 1, 2, 3, 4, 5\}$

$[1, S' \to \vdash S \dashv \bullet, \bullet \dashv]$
$[2, S \to S - T \bullet, \bullet \dashv]$
$[2, S \to S - T \bullet, \bullet -]$
$[2, S \to S - T \bullet, \bullet)]$
$[3, S \to T \bullet, \bullet \dashv]$
$[3, S \to T \bullet, \bullet -]$
$[3, S \to T \bullet, \bullet)]$
$[4, T \to (S) \bullet, \bullet \dashv]$
$[4, T \to (S) \bullet, \bullet -]$
$[4, T \to (S) \bullet, \bullet)]$
$[5, T \to a \bullet, \bullet \dashv]$
$[5, T \to a \bullet, \bullet -]$
$[5, T \to a \bullet, \bullet)]$
$[0, S' \to \vdash S \bullet \dashv, \dashv \bullet]$
$[0, S' \to \bullet \vdash S \dashv, \vdash \bullet]$
$[0, S \to S \bullet - T, - \bullet]$
$[0, T \to (S \bullet),) \bullet]$
$[0, T \to \bullet (S), (\bullet]$
$[0, T \to \bullet a, a \bullet]$
$[0, S \to S - \bullet T, (\bullet]$
$[0, S \to \bullet T, (\bullet]$
$[0, S \to S - \bullet T, a \bullet]$
$[0, S \to \bullet T, a \bullet]$
$[0, S' \to \vdash \bullet S \dashv, (\bullet]$
$[0, T \to (\bullet S), (\bullet]$
$[0, S' \to \vdash \bullet S \dashv, a \bullet]$
$[0, T \to (\bullet S), a \bullet]$

State 2 $\{2, 3\}$

$[2, S \to S - \bullet T, \bullet \dashv]$
$[2, S \to S - \bullet T, \bullet -]$
$[2, S \to S - \bullet T, \bullet)]$
$[3, S \to \bullet T, \bullet \dashv]$
$[3, S \to \bullet T, \bullet)]$
$[3, S' \to \vdash \bullet S \dashv, \dashv \bullet]$
$[3, S \to \bullet S - T, - \bullet]$
$[3, T \to (\bullet S),) \bullet]$
$[3, S' \to \vdash \bullet S \dashv, - \bullet]$
$[3, T \to (\bullet S), - \bullet]$

Figure 5.19: Item sets of discriminating portion of LR(1) reverse automaton.

s1	$\dashv$	$-$	(	)	a
$\vdash$	$\emptyset$	$\emptyset$	$\{0\}$	$\emptyset$	$\{0\}$
$\dashv$	$\{1\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$-$	$\emptyset$	$\emptyset$	$\{0\}$	$\emptyset$	$\{0\}$
(	$\emptyset$	$\emptyset$	$\{0\}$	$\emptyset$	$\{0\}$
)	$\{4\}$	$\{4\}$	$\emptyset$	$\{4\}$	$\emptyset$
a	$\{5\}$	$\{5\}$	$\emptyset$	$\{5\}$	$\emptyset$
S	$\{0\}$	$\{0\}$	$\emptyset$	$\{0\}$	$\emptyset$
T	$\{2,3\}$	$\{2,3\}$	$\emptyset$	$\{2,3\}$	$\emptyset$

s2	$\dashv$	$-$	)
$\vdash$	$\{3\}$	$\{3\}$	$\emptyset$
$-$	$\{2\}$	$\{2\}$	$\{2\}$
(	$\emptyset$	$\{3\}$	$\{3\}$

Figure 5.20: Tables for states of Figure 5.19 of actions which are compatible with lookahead and next stack symbol.

	$\dashv$	$-$	(	)	a	$\vdash$	$\dashv$	S	$-$	T	(	)	a
s1			0		0		1	0		s2		4	5
s2						3			2		3		

Figure 5.21: Discriminating reverse LR(1) parsing table for grammar $\mathcal{G}_{exp}$.

symbols. In all, the average number of state transitions per action is 0.42 states. Considering also the overhead for partial verification of rightparts before reduction, the average depth of stack exploration per reduce action is now 2.00 symbols, with a total average depth of stack exploration per action of 1.38 symbols. This small exploration cost is not particular for this grammar, rather the contrary: these values are most often quite similar to the figures shown here, as it will be shown.

5.4.3 Syntactic error detection

Let us see now the behavior of the discriminating reverse LR(1) parser in the presence of syntactic errors in the input. Let us consider, for instance, the case where the input writer has forgotten the left parenthesis in the preceding sentence. The corresponding sequence of parsing steps is shown in Figure 5.23. Parsing proceeds reducing $E \underset{\text{rm}}{\Rightarrow}^{+} a + a + a$, and the right parenthesis is shifted on top of the stack, which in initial state calls for reducing with rule $6 : F \rightarrow (E)$. At this point the rightpart verification detects the presence of an error. In this example it is easy to tell from the verification that the error is that a left parenthesis is missing. Note that the actual point of error is of course impossible to locate. Actually, this point is surpassed and the error detected several symbols to the right. Even a correct prefix parser would behave somehow like here, because that first section of the input is

						$\vdash$	$a+(a+a)*a\dashv$
						0^*	
					$\vdash$	a	$+(a+a)*a\dashv$
						7	
					$\vdash$	F	$+(a+a)*a\dashv$
					5	s3	
					$\vdash$	T	$+(a+a)*a\dashv$
					3	s2	
					$\vdash$	E	$+(a+a)*a\dashv$
						0	
				$\vdash$	E	$+$	$(a+a)*a\dashv$
						0^*	
			$\vdash$	E	$+$	$($	$a+a)*a\dashv$
						0^*	
		$\vdash$	E	$+$	$($	a	$+a)*a\dashv$
						7	
		$\vdash$	E	$+$	$($	F	$+a)*a\dashv$
					5	s3	
		$\vdash$	E	$+$	$($	T	$+a)*a\dashv$
					3	s2	
		$\vdash$	E	$+$	$($	E	$+a)*a\dashv$
						0	
	$\vdash$	E	$+$	$($	E	$+$	$a)*a\dashv$
						0^*	
$\vdash$	E	$+$	$($	E	$+$	a	$)*a\dashv$
						7	
$\vdash$	E	$+$	$($	E	$+$	F	$)*a\dashv$
					5	s3	
$\vdash$	E	$+$	$($	E	$+$	T	$)*a\dashv$
				[]	2	s2	
		$\vdash$	E	$+$	$($	E	$)*a\dashv$
						0	
	$\vdash$	E	$+$	$($	E	$)$	$*a\dashv$
				[]	[]	6	
			$\vdash$	E	$+$	F	$*a\dashv$
					5	s3	
			$\vdash$	E	$+$	T	$*a\dashv$
					0^*	s2	
		$\vdash$	E	$+$	T	$*$	$a\dashv$
						0^*	
	$\vdash$	E	$+$	T	$*$	a	$\dashv$
						7	
	$\vdash$	E	$+$	T	$*$	F	$\dashv$
				[]	4	s3	
			$\vdash$	E	$+$	T	$\dashv$
				[]	2	s2	
					$\vdash$	E	$\dashv$
						0	
				$\vdash$	E	$\dashv$	
				[]	[]	1	
						S'	

Figure 5.22: Discriminating reverse LR(1) parsing of sentence "$a+(a+a)*a$" for grammar $\mathcal{G}_{etf}$.

			$\vdash$	$a+a+a)*a\dashv$
			0^*	
		$\vdash$	a	$+a+a)*a\dashv$
			7	
		$\vdash$	F	$+a+a)*a\dashv$
		5	s3	
		$\vdash$	T	$+a+a)*a\dashv$
		3	s2	
		$\vdash$	E	$+a+a)*a\dashv$
			0	
	$\vdash$	E	$+$	$a+a)*a\dashv$
			0^*	
$\vdash$	E	$+$	a	$+a)*a\dashv$
			7	
$\vdash$	E	$+$	F	$+a)*a\dashv$
		5	s3	
$\vdash$	E	$+$	T	$+a)*a\dashv$
		2	s2	
		$\vdash$	E	$+a)*a\dashv$
			0	
	$\vdash$	E	$+$	$a)*a\dashv$
			0^*	
$\vdash$	E	$+$	a	$)*a\dashv$
			7	
$\vdash$	E	$+$	F	$)*a\dashv$
		5	s3	
$\vdash$	E	$+$	T	$)*a\dashv$
		2	s2	
		$\vdash$	E	$)*a\dashv$
			0	
	$\vdash$	E	$)$	$*a\dashv$
	[error]	[]	6	

Figure 5.23: Discriminating reverse LR(1) parsing of erroneous input (forgotten left parenthesis) for grammar $\mathcal{G}_{etf}$.

a valid head of a sentence. Of course, error detection can also take place during the discriminatory part of the exploration, when consulting the right section of the parsing table with next stack symbol.

In case of error, the discriminating reverse parser may perform more shifts and/or condensations than a correct prefix one. To show this, let us consider the erroneous input

$$\vdash a(a+a)*a\dashv,$$

whose parsing sequence is shown in Figure 5.24. After a shift as the second action in initial state on lookahead "(", a condensation $F \underset{\mathrm{rm}}{\Rightarrow}^{+} (a+a)$ takes place, producing at the end the stack $\vdash aF$. Then, the error is detected when entering the parsing table with stack symbol a in state 3 row, since the two-symbol topmost stack sequence aF is an illegal combination.

					$\vdash$	$a(a+a)*a\dashv$
					0^*	
				$\vdash$	a	$(a+a)*a\dashv$
					0^*	
			$\vdash$	a	$($	$a+a)*a\dashv$
					0^*	
		$\vdash$	a	$($	a	$+a)*a\dashv$
					7	
		$\vdash$	a	$($	F	$+a)*a\dashv$
				5	s3	
		$\vdash$	a	$($	T	$+a)*a\dashv$
				3	s2	
		$\vdash$	a	$($	E	$+a)*a\dashv$
					0	
	$\vdash$	a	$($	E	$+$	$a)*a\dashv$
					0^*	
$\vdash$	a	$($	E	$+$	a	$)*a\dashv$
					7	
$\vdash$	a	$($	E	$+$	F	$)*a\dashv$
				5	s3	
$\vdash$	a	$($	E	$+$	T	$)*a\dashv$
			[]	2	s2	
		$\vdash$	a	$($	E	$)*a\dashv$
					0	
	$\vdash$	a	$($	E	$)$	$*a\dashv$
			[]	[]	6	
			$\vdash$	a	F	$*a\dashv$
				error	s3	

Figure 5.24: Discriminating reverse LR(1) parsing of erroneous sentence provoking condensation, for grammar $\mathcal{G}_{etf}$.

$$
\begin{array}{rl}
1: & S' \rightarrow \vdash I \dashv \\
2: & I \rightarrow iEtFeI \\
3: & I \rightarrow iEtI \\
4: & I \rightarrow E \\
5: & F \rightarrow iEtFeF \\
6: & F \rightarrow E \\
7: & F \rightarrow \varepsilon \\
8: & E \rightarrow TR \\
9: & R \rightarrow R + T \\
10: & R \rightarrow \varepsilon \\
11: & T \rightarrow (E) \\
12: & T \rightarrow a
\end{array}
$$

Figure 5.25: $\mathcal{G}_{ift}$: LR(1), non-LR(0) grammar for if-then-else with expressions.

5.4.4 Grammars with empty rules

The grammar shown in Figure 5.25 serves as example of a non-LR(0) LR(1) grammar with empty rules, for a simple language of conditionals including additive expressions (i.e., i, t, and e, stand for "if", "then", and "else", respectively). Here the dangling else problem has been "syntactically solved" as shown.

The corresponding discriminating reverse LR(1) parsing table is shown in Figure 5.26, containing 33 entries. As in the LR(0) case, empty rules present no special problem. After discarding the beginning of a I or a E (i.e., lookahead values i, "(", and a, in initial state), the only possibility with t or e as topmost stack symbol is to reduce rule 7: $F \rightarrow \varepsilon$ in initial state, as the following subset of items shows:

$$
\begin{array}{l}
[7, F \rightarrow \bullet, \bullet e] \\
[7, I \rightarrow iEt \bullet FeI, e \bullet] \\
[7, F \rightarrow iEt \bullet FeF, e \bullet] \\
[7, F \rightarrow iEtFe \bullet F, \bullet e]
\end{array}
$$

Otherwise, if the topmost stack symbol is T, then (and only then) only reductions 9: $R \rightarrow R+T$ and 10: $R \rightarrow \varepsilon$ are possible, the latter obviously only when the next stack symbol is $+$.

The reader can verify all of this at parsing time in Figure 5.27, which shows the sequence of parsing steps for sentence $\vdash ia+atiatea\dashv$, with 11 shifts and 15 reductions of an average length of 1.60 symbols (while the grammar has an average length of rightparts of 2.45 symbols). The average depth of stack exploration per shift action is 0.64 symbols, and the average depth of stack exploration per reduce action is 1.40 symbols. In all, the average

	$\dashv$	i	t	e	$+$	(	)	a	$\vdash$	$\dashv$	I	i	E	t	F	e	T	R	$+$	(	)	a
s1		0				0		0		1	s2		s3	7	s4	7	s6	s5			11	12
s2									0					3		2						
s3	4		0	6			0															
s4														0		5						
s5	8		8	8	0		8															
s6									10			10		10		10			9	10		

Figure 5.26: Discriminating reverse LR(1) parsing table for grammar $\mathcal{G}_{ift}$.

number of state transitions per action is 0.58 states. Considering also the overhead for partial verification of rightparts before reduction, the average depth of stack exploration per reduce action is now 2.07 symbols, with a total average depth of stack exploration for action of 1.47 symbols. As the reader can verify, these figures are very similar to our previous example for grammar $\mathcal{G}_{etf}$.

5.4.5 Erroneous inputs and empty reductions

As we have seen, in the deferred lookahead solution for parsing action discrimination, only in some cases the action is decided on current lookahead. This implies its compatibility with the explored stack suffix, according to the grammar. But when the parsing action is decided on next stack symbol, the parser will already have verified only that current lookahead *is not* amongst a certain set of values, i.e., the lookahead values in the lookahead section of the states in the path from initial state to the decision state.

For incorrect inputs, at some moment during parsing, the stack plus lookahead configuration will become illegal. This has to lead, obviously, to the detection of a syntactic error at this moment or later. This may happen immediately, but the parser may also decide to perform an action if the current stack (suffix) is legal, when the current "illegal" lookahead value is not in the verified set. All the decided actions are indeed syntactically correct in the context of the explored stack suffix, and since the presence on top of the stack of the handle to reduce is always verified, the reductions are also locally correct, and can only build a locally correct subtree from an LR(1) grammar. As far as the parser consumes input, this can only lead to the detection of a syntactic error at some time, since otherwise a correct and complete tree would have been built, what is impossible from an incorrect input. The only remaining case is when no input is consumed, i.e., the parser might be induced by an erroneous input to perform and endless sequence of reductions from ε. Fortunately, this can be easily avoided, as we shall see in this section.

To illustrate with a simple example how this problem may appear, we shall use grammar $\mathcal{G}_{ier}$ (see Figure 5.28), whose discriminating reverse LR(1) table is shown in Figure 5.29. This grammar has the particularity that, for some sentences leading to AB as topmost stack symbols, reduction $14 : C \rightarrow \varepsilon$

										$\vdash$	$ia+atiatea\dashv$
										0^*	
									$\vdash$	i	$a+atiatea\dashv$
										0^*	
								$\vdash$	i	a	$+atiatea\dashv$
										12	
								$\vdash$	i	T	$+atiatea\dashv$
									10	s6	
							$\vdash$	i	T	R	$+atiatea\dashv$
									0^*	s5	
						$\vdash$	i	T	R	$+$	$atiatea\dashv$
										0^*	
					$\vdash$	i	T	R	$+$	a	$tiatea\dashv$
										12	
					$\vdash$	i	T	R	$+$	T	$tiatea\dashv$
								[]	9	s6	
							$\vdash$	i	T	R	$tiatea\dashv$
									$[]8^*$	s5	
								$\vdash$	i	E	$tiatea\dashv$
									0^*	s3	
							$\vdash$	i	E	t	$iatea\dashv$
										0^*	
						$\vdash$	i	E	t	i	$atea\dashv$
										0^*	
					$\vdash$	i	E	t	i	a	$tea\dashv$
										12	
					$\vdash$	i	E	t	i	T	$tea\dashv$
									10	s6	
				$\vdash$	i	E	t	i	T	R	$tea\dashv$
									$[]8^*$	s5	
					$\vdash$	i	E	t	i	E	$tea\dashv$
									0^*	s3	
				$\vdash$	i	E	t	i	E	t	$ea\dashv$
										7	
			$\vdash$	i	E	t	i	E	t	F	$ea\dashv$
									0	s4	
		$\vdash$	i	E	t	i	E	t	F	e	$a\dashv$
										0^*	
	$\vdash$	i	E	t	i	E	t	F	e	a	$\dashv$
										12	
	$\vdash$	i	E	t	i	E	t	F	e	T	$\dashv$
									10	s6	
$\vdash$	i	E	t	i	E	t	F	e	T	R	$\dashv$
									$[]8^*$	s5	
	$\vdash$	i	E	t	i	E	t	F	e	E	$\dashv$
									4^*	s3	
	$\vdash$	i	E	t	i	E	t	F	e	I	$\dashv$
					[]	[]	[]	[]	2	s2	
						$\vdash$	i	E	t	I	$\dashv$
							[]	[]	3	s2	
									$\vdash$	I	$\dashv$
									0	s2	
								$\vdash$	I	$\dashv$	
								[]	[]	1	

Figure 5.27: Discriminating reverse LR(1) parsing of sentence "$ia+atiatea$" for grammar $\mathcal{G}_{ift}$.

$$
\begin{array}{rl}
1: & S' \to \vdash S \dashv \\
2: & S \to ABd \\
3: & S \to aBc \\
4: & S \to cABCc \\
5: & S \to bCa \\
6: & S \to aBCAa \\
7: & S \to cAb \\
8: & S \to bCABb \\
9: & A \to e \\
10: & A \to \varepsilon \\
11: & B \to f \\
12: & B \to \varepsilon \\
13: & C \to g \\
14: & C \to \varepsilon
\end{array}
$$

Figure 5.28: $\mathcal{G}_{ier}$: LR(1) grammar for unbounded sequence of empty reductions.

can be decided on that A. According to the parsing table, the parser takes this decision when current lookahead does not belong to set $\{d, a, b, e, g\}$ —although, actually, this can only happen in a sentence for c as current lookahead—, since otherwise, the decision would have been taken on the current lookahead in state 5. In a similar way, for inputs leading to BC on top of the stack with lookahead not in set $\{c, b, e, f\}$ —in a sentence, only a can appear as lookahead— reduction $10 : A \to \varepsilon$ is decided on that B in state 6. Finally, for inputs leading to CA on top of the stack with lookahead not in set $\{d, a, c, f, g\}$ —in a sentence, only b can appear on lookahead— reduction $12 : B \to \varepsilon$ is decided on that C in state 7. Evidently, the above three reductions cannot happen in sequence for a sentence, since they can only happen with different lookahead values[5]. But it may happen for a non-sentence if the lookahead is not in the union of the above three sets, that is to say, if lookahead is $\dashv$. Thus, if it is possible for a non-sentence to lead to a stack containing AB, BC, or CA, as its two topmost symbols with $\dashv$ as lookahead, then an unbounded sequence of reductions might be triggered. This is what happens in our example when the parser receives the erroneous input $\vdash ef \dashv$, as Figure 5.30 shows. A sequence of locally correct parsing steps corresponding to derivation $AB \underset{\mathrm{rm}}{\Rightarrow}^{+} ef$ is performed. But, from this point on, a cyclic sequence of empty reductions follows, repeatedly generating the sequence ABC.

Even though this can only happen for a very specific kind of grammars and for very specific erroneous inputs, the possibility of the above behavior is normally unacceptable. In the general case, this kind of situations can

[5]Otherwise the grammar would be ambiguous, thus non-LR(1).

	$\dashv$	d	a	c	b	e	f	g	$\vdash$	$\dashv$	S	A	B	d	a	c	C	b	e	f	g
s1									s8	1	0	s7	s5	2	s3	s2	s6	s 4	9	11	13
s2				10	10	0	10	10					3				4				
s3			12	12		12	0	12				6					5				
s4			14		14	14	14	0				7	8								
s5		0	14		0	14		0				14			0						
s6				0	10	0	10						10					0			
s7		12	0	12			0	12								0	12				
s8		10	0	0	0	0	10														

Figure 5.29: Discriminating reverse LR(1) parsing table for grammar $\mathcal{G}_{ier}$, without verifications for loops of reductions from empty.

					$\vdash$	$ef\dashv$
				0^*	s8	
				$\vdash$	e	$f\dashv$
					9	
				$\vdash$	A	$f\dashv$
				0^*	$s7$	
			$\vdash$	A	f	$\dashv$
					11	
			$\vdash$	A	B	$\dashv$
				14	$s5$	
		$\vdash$	A	B	C	$\dashv$
				10	$s6$	
	$\vdash$	A	B	C	A	$\dashv$
				12	$s7$	
$\vdash$	A	B	C	A	B	$\dashv$
				14	$s5$	

Figure 5.30: Discriminating reverse LR(1) parsing without lookahead verification of erroneous sentence provoking unbounded sequence of empty reductions, for grammar $\mathcal{G}_{ier}$.

occur because some reductions, within a sequence of reductions from empty, are performed without verifying compatibility with current lookahead. Otherwise, i.e., if current lookahead were verified, all those nullable reductions, if performed, would be always correct in that suffix plus lookahead context, but their sequence would never be unbounded, since this is impossible for an LR(1) grammar.

Accordingly, in the discriminating reverse parser, lookahead will be verified (at least) in those cases, and thus sequences of reductions from the empty string will be necessarily bounded, whatever the erroneous input and stack configuration of the parser. In order to achieve that desired parser behavior, some simple solutions are possible. All of them can be implemented at parser generation time, with slight to null effect on parsing efficiency, depending on the grammar. Here three possibilities are given:

- Never reduce on a stack symbol. That is, inhibit reductions from being decided on the stack symbol, such that if it were normally possible to take a decision already, instead of its decision a transition would be performed on a new state where it would be taken only on the lookahead part of the table, for the compatible lookahead values only, obviously, according to the resulting state item set. This solution, although very simple, is considered too brute force, unnecessarily increasing the number of states.

- Detect the possibility of this kind of loops from a directed graph. In our example, detect that "state" $[AB]$ (corresponding to stack configuration with AB on top) may lead with lookahead in $\{\dashv, e, f\}$ to "state" $[BC]$. Note that these configurations are finite for an LR(1) grammar. Now, detect the possibility of loops in the graph. And for these loops transform one reduction on stack symbol in the parsing table into a transition to new state deciding on lookahead. This solution, with minimal impact on table size and parsing efficiency, is also considered extreme, unnecessarily complicated, and somehow computationally costly at generation time. Thus, an intermediate solution of compromise has been chosen:

- Never decide a nullable reduction, i.e., of the form $A\rightarrow\nu$, with $\nu\Rightarrow^*\varepsilon$, on a nullable stack symbol B, i.e., such that $B\Rightarrow^+\varepsilon$. That is, always verify lookahead compatibility, and thus local correction of the reduction, in any such rule which might take part in such a hypothetical loop of reductions from empty. Note that in order to enter such a loop, an unbounded sequence of nonterminals must be produced in the stack, and these reductions can only be of that form. If these are in these cases always verified, then any such incipient loop is immediately cut through the usual syntactic error detection mechanism. Clearly, this solution is easy to implement at low cost and, since these cases should be quite rare in practice, this would normally lead to none or very small impact on the number of parser states and thus on parsing efficiency. Refer to Section 7.5 for a general correctness proof for this solution.

It will be shown that discriminating reverse parsing has a linear time complexity for parsing. The above class of solutions does not affect this property, since the number of reductions from empty is bounded, and the number of parsing decisions is also linear with the input length, correct or not.

To illustrate how our solution can be implemented, see for instance item set for state 5 corresponding to the above parsing table.

State 5

$$
\begin{array}{l}
[0, S \rightarrow A \bullet Bd, d \bullet] \\
[0, S \rightarrow a \bullet Bc, c \bullet] \\
[0, S \rightarrow bCA \bullet Bb, b \bullet] \\
[14, S \rightarrow a \bullet BCAa, a \bullet] \\
[14, S \rightarrow cA \bullet BCc, c \bullet] \\
[14, S \rightarrow a \bullet BCAa, e \bullet] \\
[0, S \rightarrow cA \bullet BCc, g \bullet] \\
[0, S \rightarrow a \bullet BCAa, g \bullet]
\end{array}
$$

After generating the corresponding table entries for the lookahead section of the parsing table, the remaining items are used to decide on the stack symbol section.

$$
\begin{array}{l}
[0, S \rightarrow a \bullet Bc, c \bullet] \\
[14, S \rightarrow cA \bullet BCc, c \bullet]
\end{array}
$$

Obviously, a decides 0, and A decides 14, as Figure 5.29 shows. In our solution, the latter decision would be inhibited, since 14 corresponds to a nullable reduction, and A itself is nullable. Thus, instead of the decision, a transition is entered in the table on stack symbol A to a new state containing only

State 11

$$[14, S \rightarrow c \bullet ABCc, c \bullet]$$

where, obviously, the normal table generation procedure will generate decision 14 on *lookahead* c. Thus, the parser will take this decision only when current lookahead is compatible. The resulting corrected parsing table is shown in Figure 5.31, which has three new states for verification: states 9 to 11. Note that, obviously, their stack symbol section has no decision, and thus the parser, following its normal algorithm, will signal the error if no decision is taken when consulting the lookahead section, thus, cutting from its very inception the possibility of an endless loop of reductions from empty. Figure 5.32 shows that now, for erroneous entry $\vdash ef \dashv$, the parser detects the syntactic error with no cycle of nullable reductions.

	$\dashv$	d	a	c	b	e	f	g	$\vdash$	$\dashv$	S	A	B	d	a	c	C	b	e	f	g
s1									s8	1	0	s7	s5	2	s3	s2	s6	s4	9	11	13
s2				10	10	0	10	10					3				4				
s3			12	12		12	0	12				6					5				
s4			14		14	14	14	0				7	8								
s5		0	14		0	14		0				s11			0						
s6				0	10	0	10						s10					0			
s7		12	0	12			0	12								0	s9				
s8		10	0	0	0	0	10														
s9					12																
s10			10																		
s11				14																	

Figure 5.31: Discriminating reverse LR(1) parsing table without possibility of unbounded sequence of empty reductions, for grammar $\mathcal{G}_{ier}$.

		$\vdash$	$ef\dashv$
	0^*	s8	
	$\vdash$	e	$f\dashv$
		9	
	$\vdash$	A	$f\dashv$
	0^*	s7	
$\vdash$	A	f	$\dashv$
		11	
$\vdash$	A	B	$\dashv$
error	s11	s5	

Figure 5.32: Discriminating reverse LR(1) parsing of erroneous sentence, with bounded sequence of nullable reductions, for grammar $\mathcal{G}_{ier}$.

Note that for our example grammar, just one of theses verification states would actually suffice to cut, with a bounded length, any such incipient loop of reductions, and thus to detect the syntactic error. But this would require to apply a solution with deeper analysis at generation time, as described above.

5.4.6 Non-LR(1) grammar detection

If a grammar is not LR(1), then there exists some legal input for which it is impossible for the discriminating reverse LR(1) parser to decide some action, even after having reversely scanned the whole stack. At that point, there would be at least two conflicting legal actions for the same lookahead value. Thus, in the corresponding *"bottom-of-stack"* state, there would be two items of the form

$$[i, S'{\to}\vdash{\bullet}S\dashv, a{\bullet}], [j, S'{\to}\vdash{\bullet}S\dashv, a{\bullet}], \text{ with } i \neq j.$$

But, as we saw in Chapter 4 for the LR(0) case, we can actually detect this condition earlier during parser construction, and thus to avoid constructing a possibly large and unnecessary section of the automaton until such states are built. The idea is to detect, using only the information within the state items themselves, whether two parsing actions share some of these stacks.

Left-hand context languages

We define the left-hand context language $\mathcal{L}$ of an item as the language of all possible stack prefix continuations that can be found when exploring the stack from the current point of exploration, down to the stack bottom, supposing that the current input is a sentence compatible with the item. More precisely, considering item equivalence (i.e., the rightpart section to the right of the dot being irrelevant), correctness of our item set construction procedure indeed implies that

$$\mathcal{L}([i,\ B{\to}\beta{\bullet}\rho,\ X{\bullet}Y]) = \{\zeta\beta \mid S' \underset{\mathrm{rm}}{\Rightarrow}^* \zeta Bz \text{ and } 1: X{\bullet}z = X{\bullet}Y\},$$

with $XY = a$. That is, there are two cases: either $X = \varepsilon$ and $Y = a$ (thus $x{\bullet}Y = {\bullet}a$) or $X = a$ and $Y = \varepsilon$ (thus $X{\bullet}Y = a{\bullet}$).

According to this definition, and always considering item equivalence,

$$\mathcal{L}([i,\ B{\to}\beta{\bullet}\rho,\ a{\bullet}]) \begin{cases} = \mathcal{L}([j,\ B{\to}\beta{\bullet}\mu,\ a{\bullet}]) \\ \supseteq \mathcal{L}([j,\ B{\to}\beta{\bullet}\mu,\ {\bullet}a]) \end{cases}$$

The former simply reflects item equivalence. While the latter shows that left-hand context languages for items with "$a{\bullet}$" as lookahead include those with "${\bullet}a$" as lookahead. Accordingly, since $\mathcal{L}$ is never empty for a state item,

$$\mathcal{L}([i,\ B{\to}\beta{\bullet}\rho,\ a{\bullet}]) \cap \mathcal{L}([j,\ B{\to}\beta{\bullet}\mu,\ {\bullet}a]) = \mathcal{L}([j,\ B{\to}\beta{\bullet}\mu,\ {\bullet}a]) \neq \emptyset.$$

A test for items in conflict

According to the above definitions, we can immediately see that if the left-hand context languages of two items for a different action of a same state have a nonempty intersection, and their lookahead symbol is the same, then we can predict that, for those sentences giving place to the stack configuration corresponding to the intersection, it will be impossible to decide the action from the information in the stack plus lookahead. That in turn implies that the grammar is not LR(1)[6]. Thus, we define two such items to be *in conflict* when

$$\mathcal{L}([i,\ A{\rightarrow}\alpha{\bullet}\mu,\ X{\bullet}Y]) \cap \mathcal{L}([j,\ B{\rightarrow}\beta{\bullet}\rho,\ X{\bullet}Y]) \neq \emptyset,\ i \neq j.$$

Moreover, it is cleat that if the grammar is not LR(1), at initial state there will be necessarily at least two items in conflict, for otherwise it would be always possible to discriminate next parsing action. But although left-hand context languages are regular, we consider it unnecessarily costly to compute them in order to always detect this condition at the initial state. Thus we propose the compromise solution of testing for conflicts directly from the item information, i.e., in our generation algorithm we shall test for the following condition: the existence in a same state item set of two items of the form:

$$[i,\ B{\rightarrow}\beta{\bullet}\rho,\ X{\bullet}Y],\ [j,\ B{\rightarrow}\beta{\bullet}\mu,\ X{\bullet}Y],\ \text{with } i \neq j.$$

It is evident that this will happen at least at a bottom-of-stack state, as we have seen above. In fact, it usually happens at initial state or quite early during automaton construction. Moreover, in order to promote a prompt non-LR(1)-ness detection, in our implementation we use a breadth-first state construction[7].

In Section 8.2, the above test will be use for ambiguous grammars to detect such types of items in conflict in order to apply default or designer-supplied disambiguation rules.

An example non-LR(1) grammar

Let us consider for instance the non-LR(1) LR(2) grammar of Figure 5.33. Its discriminating reverse LR(1) initial state contains the item set shown in Figure 5.34. As we said, often the non-LR(1) condition is found to already hold in initial state for some pair(s) of items. It is not the case for this grammar, even though a stronger test should detect that left-hand context

[6] As we have seen above, every non-LR(1) grammar has this property at least at a bottom-of-stack state.

[7] There is a small disadvantage: non-consecutive ordering of state numbers, as it can be appreciated listings in Appendix A.

$$
\begin{array}{ll}
1: & S' \to \vdash S \dashv \\
2: & S \to Ac \\
3: & S \to aA \\
4: & A \to aB \\
5: & A \to abc \\
6: & B \to b
\end{array}
$$

Figure 5.33: $\mathcal{G}_{lrt}$: non-LR(1) LR(2) grammar.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{State 1 } \{0,1,2,3,4,5,6\}} \\
 & [1, S' \to \vdash S \dashv \bullet, \bullet \dashv] \\
 & [2, S \to Ac \bullet, \bullet \dashv] \\
 & [3, S \to aA \bullet, \bullet \dashv] \\
 & [4, A \to aB \bullet, \bullet \dashv] \\
 & [4, A \to aB \bullet, \bullet c] \\
 & [5, A \to abc \bullet, \bullet \dashv] \\
 & [5, A \to abc \bullet, \bullet c] \\
 & [6, B \to b \bullet, \bullet \dashv] \\
(*) & [6, B \to b \bullet, \bullet c] \\
 & [0, S' \to \vdash S \bullet \dashv, \dashv \bullet] \\
 & [0, S' \to \bullet \vdash S \dashv, \vdash \bullet] \\
 & [0, S \to A \bullet c, c \bullet] \\
 & [0, S \to \bullet aA, a \bullet] \\
 & [0, A \to \bullet aB, a \bullet] \\
(*) & [0, A \to ab \bullet c, c \bullet] \\
 & [0, A \to a \bullet bc, b \bullet] \\
 & [0, B \to \bullet b, b \bullet] \\
 & [0, S' \to \vdash \bullet S \dashv, a \bullet] \\
 & [0, S \to a \bullet A, a \bullet]
\end{array}
$$

Figure 5.34: Discriminating reverse LR(1) automaton initial-state item set for grammar $\mathcal{G}_{lrt}$.

languages for items marked with "($*$)" actually intersect. As a consequence, the generator must then build, after transition on b,

$$\begin{array}{ll} \multicolumn{2}{l}{\text{State } 4\ \{0,6\}} \\ & [6, B\to\bullet b,\ \bullet\dashv] \\ & [6, B\to\bullet b,\ \bullet c] \\ * & [0, A\to a\bullet bc, c\bullet] \\ & [6, A\to a\bullet B,\ \bullet\dashv] \\ * & [6, A\to a\bullet B,\ \bullet c] \end{array}$$

Now the test succeeds to detect non-LR(1)-ness from the pair of items marked. Note that the rightpart sections to the left of the dot are equal, but those to the right of the dot are different. And that the lookahead dots are in different positions, that is, although the left-hand context language for the upper conflicting item is $\{\vdash a, \vdash aa\}$, the lower conflicting item's left-hand context language, $\{\vdash a\}$, is contained in it. Thus, although for sentence $\vdash aabc\dashv$ it would suffice to look deeper in the stack to be able to decide action 0, our test implies some intersection of the left-hand context languages, what implies the existence of sentences —in our case, $\vdash abc\dashv$ and $\vdash abcc\dashv$ — for which an LR(1) parser cannot decide in some stack plus lookahead configuration — $\vdash ab\,|\,c$, in our case. But we can see that the grammar is LR(2), since for this non-LR(1) configuration it would suffice to look at the second lookahead symbol to decide.

For ambiguous grammars —thus, non-LR(1)—, the problem is the same from the point of view of the discriminating reverse LR(1) parser. There are some sentences for which at some point there is some configuration with more that one legal possibility. In this case it is due to the possibility of building different syntax trees, thus different derivations. Accordingly, the functioning of the test is the same as we have just seen. Section 8.2.4 shows an ambiguous grammar whose conflicting items are detected in order to apply disambiguation rules.

5.5 Algorithms

This section describes the generation and parser algorithms developed in this chapter.

5.5.1 Parsing table generation

In the generation algorithm shown in Figure 5.35, items represented without the lookahead or action dot mean both cases are considered—e.g. "$\bullet b$" and "$b\bullet$"—, since the distinction is sometimes irrelevant. Remember also the convention by which "item" $[j,\ B\to\gamma\bullet X\delta,\ b]$ has the lookahead dot in the same position as the corresponding "item" $[j,\ B\to\gamma X\bullet\delta,\ b]$.

```
algorithm Generator of discriminating-reverse LR(1) parsing table
input Reduced grammar 𝒢
output Parsing table ActTrans(state,kind_ of_ symbol,symbol),
        or Error if the grammar is not LR(1)
function Closure(J)
    I := J
    repeat
        for [i, A→•α, •a] ∈ I do for B→βAγ ∈ P′ do
            if a ∈ First(γ) then add [i, B→β•Aγ, a•] to I
            else if ε ∈ First(γ) and a ∈ Follow(B) then add [i, B→β•Aγ, •a] to I
        for [i, A→•α, a•] ∈ I do for B→βAγ ∈ P′ do add [i, B→β•Aγ, a•] to I
    until no new additions to I
    return I
end Closure
begin
    N_n:={A | A⇒⁺ε}; N_r:={i | A –i→ α and α⇒*ε}
    Aut:=empty
    K_0 :={[i, A→α•, •a] | A –i→ α ∈ P′ and a ∈ Follow(A)} ∪
          {[0, A→α•aβ, a•] | A→αaβ ∈ P′}
    I_0 :=Closure(K_0); add I_0 to Aut
    for I ∈ Aut do
        % non-LR(1) check
        if [i, A→α•β, a], [j, A→α•γ, a] ∈ I,  i ≠ j then Error
        % decisions upon current lookahead
        for a ∈ V_T′ do La(a) := {i | [i, A→α•β, a] ∈ I}
        J := I
        for a | La(a) = {i} do
            if i = 0 then ActTrans (I,lookahead,a):=shift
            else ActTrans(I,lookahead,a):=reduce(i)
            J := J − {[i, A→α•β, a] ∈ J}
        % decisions upon next stack symbol
        for X ∈ V′ do St(X) := {i | [i, A→αX•β, a] ∈ J}
        for X | St(X) = {i} and  not (X ∈ N_n and i ∈ N_r) do
            if i = 0 then ActTrans(I,stack,X):=shift
            else ActTrans (I,stack,X):=reduce(i)
            J := J − {[i, A→αX•β, a] ∈ J}
        % state transitions upon next stack symbol
        for X | [i, A→αX•β, a] ∈ J do
            K :=Closure({[j, B→γ•Xδ, b] | [j, B→γX•δ, b] ∈ J})
            add K to Aut; ActTrans(I,stack,X) := goto(K)
end
```

Figure 5.35: Generation algorithm of discriminating reverse LR(1) parser.

Note that, although the closure operation as previously defined allows both items $[A \rightarrow \alpha \bullet \beta, \bullet a]$ and $[A \rightarrow \alpha \bullet \beta, a \bullet]$ in the same state set, the former item is redundant for automaton construction when the latter item is also present, since, according to Lemma 7.11 the latter item includes all the possible continuations (i.e., legal remaining stack contents) of the former item. In the closure function the redundant item is implicitly eliminated during the first **for** loop.

The algorithm reflects the method discussed in preceding sections. In each state, first the lookahead symbol is used to try to decide the parsing action if all the situations with a given lookahead symbol signal the same action. For the remaining cases in which the lookahead fails, the next possible stack symbols are used to discriminate, if possible, the parsing action (i.e., if all the remaining situations with a given next stack symbol indicate the same action), with the exception that nullable reductions are inhibited on nullable stack symbols (N_r and N_n sets, respectively). Then, transitions to other states are computed on the remaining values for next stack symbol.

Finally, note that the algorithm is presented with emphasis on clarity rather than efficiency.

5.5.2 Parsing algorithm

The parsing algorithm shown in Figure 5.36 uses a previously generated action and transition table. State item sets themselves are useless at parsing time, and thus states are differently referred to. Note that a part of the handle[8] to reduce should be verified if it has not been completely explored.

The parsing stack is initialized with a bottom-of-stack marker at the beginning of parsing. Before each parsing action, stack exploration begins from the top and initial state. In each state, checking the lookahead is made first, else the next stack symbol is used for a parsing action or transition. This exploration will eventually finish with deciding a shift, reduce or accept action, or else a syntactic error is found.

In general, since the parser is not intended to check whether the whole stack is syntactically correct, the stack is not necessarily a viable prefix.

5.6 Parsers for some grammar classes

Next sections show the use of the preceding algorithm of parser generation to produce the discriminating reverse LR(1) parsing tables for several interesing

[8]The discriminating reverse automaton can only shift on lookahead in initial state, for each lookahead must be shifted at some time, thus a reduction cannot be the only possibility for a lookahead symbol and thus a reduction cannot be decided only on it. Thus at least one (handle) symbol has been read at that point. This is quite a lot: practical grammars have an average rightpart length of 2–3 symbols only, and the average length of handles is usually smaller than 2.

```
algorithm Discriminating-reverse LR(1) parser;
input Parsing table ActTrans(state,kind_of_symbol,symbol)
        and input sequence
output Bottom-up parse of the input sequence,
         or Error if syntactically incorrect
begin
    Stack:=empty; push(⊢); read(a)
    repeat
        q := q0; p := top
        repeat
            at:=ActTrans(q,lookahead,a)
            if at=nil then at:=ActTrans(q,stack,Stack(p))
            case at of
                goto(q'): q := q'; p := p − 1
                shift: push(a); read(a)
                reduce(A -i-> α): if i = 1 then at:=accept
                    elsif not ontop(α) then Error
                    else pop(α); push(A)
                nil: Error
        until at ≠ goto
    until at=accept
end
```

Figure 5.36: Algorithm of discriminating reverse LR(1) parser.

classes of grammars. The parsing tables for a non-SLR(1) grammar and several non-LALR(1) grammars are shown, which are LR(1) or even LL(1).

5.6.1 LALR(1) non-SLR(1) grammars

The grammars shown up to this point in this chapter are all SLR(1) and LALR(1). Remember that LALR(1), SLR(1) and LR(0) share the same automaton (i.e., the same states and transitions on terminal and nonterminal symbols), and that LALR(1) parsing tables entries are in theory a subset of SLR(1) entries. However, for the preceding grammars, their respective parsing LALR(1) and SLR(1) parsing tables are equal for each grammar, containing exactly the same entries[9]. See, for instance, the direct SLR(1) and LALR(1) parsing table for grammar $\mathcal{G}_{ift}$ shown in Figure 5.37, containing 90 entries. This may be surprising. Note, however, that LALR(1) actually improves only in some cases the context information of the parser in order to compute the lookahead sets used for action decision. The LALR(1) generator has to do this *without* changing the LR(0) automaton nor, obviously, its number of states. In some cases, this is simply too restrictive, since evidently the LR(0) parser generator does not use any lookahead information to build

[9]Of course, this is not the case for many grammars. For instance, if in grammar $\mathcal{G}_{ift}$, the rule $F \rightarrow E$ is replaced by $F \rightarrow TR$, then the SLR(1) and LALR(1) parsing tables have 93 and 98 entries, respectively.

	$\dashv$	i	t	e	$+$	(	)	a	I	E	F	T	R
s1		s3				s6		s7	2	4		5	
s2	s8												
s3						s6		s7		9		5	
s4	r4												
s5	r10		r10	r10	r10		r10						10
s6						s6		s7		11		5	
s7	r12		r12	r12	r12		r12						
s8	r1												
s9			s12										
s10	r8		r8	r8	s13		r8						
s11							s14						
s12		s17		r7		s6		s7	16	18	15	5	
s13						s6		s7				19	
s14	r11		r11	r11	r11		r11						
s15				s20									
s16	r3												
s17						s6		s7		21		5	
s18	r4			r6									
s19	r9		r9	r9	r9		r9						
s20		s3				s6		s7	22	4		5	
s21			s23										
s22	r2												
s23		s17		r7		s6		s7	16	18	24	5	
s24				s25									
s25		s17		r7		s6		s7	22	18	26	5	
s26				r5									

Figure 5.37: Direct SLR(1) and LALR(1) parsing table for grammar $\mathcal{G}_{ift}$

its states. In this sense, it could be said that LALR(1) is somehow in the middle between LR(0) and LR(1).

In many compiler texts, example grammars to introduce the LALR(1) technique are also SLR(1) grammars, and even share the same tables. A grammar from [TS85] is shown in Figure 5.38. Its SLR(1) parsing table is shown in Figure 5.39, containing a conflict in state 4 between reducing rules 3 and 6. The corresponding LALR(1) table is identical, except in this conflicting entry, where the LALR(1) table contains the correct reduction 3. The conflict clearly originated in that, although $\dashv$ is in Follow(T), state 4 is only reached from initial state through a transition on f, and thus, if the lookahead symbol is $\dashv$ (i.e., the sentence is $\vdash f \dashv$), the only possibility is to

$$\begin{array}{ll}
1: & S' \rightarrow \vdash S \dashv \\
2: & S \rightarrow E = E \\
3: & S \rightarrow f \\
4: & E \rightarrow T \\
5: & E \rightarrow E + T \\
6: & T \rightarrow f \\
7: & T \rightarrow T * f
\end{array}$$

Figure 5.38: $\mathcal{G}_{tas}$: LALR(1) non-SLR(1) grammar.

	$\dashv$	$=$	f	$+$	$*$	S	E	T
s1			s4			2	3	5
s2	s6							
s3		s7		s8				
s4	{r3,r6}	r6		r6	r6			
s5	r4	r4		r4	s9			
s6	r1							
s7			s11				10	5
s8			s11					12
s9			s13					
s10	r2			s8				
s11	r6	r6		r6	r6			
s12	r5	r5		r5	s9			
s13	r7	r7		r7	r7			

Figure 5.39: Direct SLR(1) parsing table for grammar $\mathcal{G}_{tas}$ with conflict in state 4.

	$\dashv$	$=$	f	$+$	$*$	$\vdash$	$\dashv$	S	E	$=$	f	T	$+$	$*$
s1			0				1	0	s2		s3	s4		
s2	2	0		0										
s3						s5				6			6	7
s4					0	4				4			5	
s5	3	6		6	6									

Figure 5.40: Discriminating reverse LR(1) parsing table for grammar $\mathcal{G}_{tas}$.

reduce it to S, and not to T.

The corresponding discriminating reverse LR(1) parsing table is shown in Figure 5.40. Here, the problem is solved in state 5 after two transitions: on top-of-stack (and handle) f and the bottom-of-stack marker $\vdash$. This is needed because state 3, reached from initial state after transition on f, contains the following item set

State 3

$[3, S \to \bullet f, \bullet \dashv]$
$[6, T \to \bullet f, \bullet \dashv]$
$[6, T \to \bullet f, \bullet =]$
$[6, T \to \bullet f, \bullet +]$
$[7, T \to T * \bullet f, \bullet \dashv]$
$[7, T \to T * \bullet f, \bullet =]$
$[7, T \to T * \bullet f, \bullet +]$
$[7, T \to T * \bullet f, \bullet *]$
$[3, S' \to \vdash \bullet S \dashv, \dashv \bullet]$
$[6, E \to \bullet T, \bullet \dashv]$
$[6, E \to E + \bullet T, \bullet \dashv]$
$[6, E \to \bullet T, \bullet =]$
$[6, E \to E + \bullet T, \bullet =]$
$[6, E \to E + \bullet T, \bullet +]$
$[6, T \to \bullet T * f, * \bullet]$
$[6, S \to E = \bullet E, \bullet \dashv]$
$[6, S \to \bullet E = E, = \bullet]$
$[6, E \to \bullet E + T, + \bullet]$
$[6, E \to \bullet T, * \bullet]$
$[6, E \to E + \bullet T, * \bullet]$
$[6, S' \to \vdash \bullet S \dashv, = \bullet]$
$[6, S \to \bullet E = E, + \bullet]$
$[6, S \to E = \bullet E, + \bullet]$
$[6, S \to \bullet E = E, * \bullet]$
$[6, S \to E = \bullet E, * \bullet]$
$[6, S' \to \vdash \bullet S \dashv, + \bullet]$
$[6, S' \to \vdash \bullet S \dashv, * \bullet]$

From this, a transition to state 5 on $\vdash$ is needed, containing the following items:

State 5

$[3, S' \to \bullet \vdash S \dashv, \dashv \bullet]$
$[6, S' \to \bullet \vdash S \dashv, = \bullet]$
$[6, S' \to \bullet \vdash S \dashv, + \bullet]$
$[6, S' \to \bullet \vdash S \dashv, * \bullet]$

In this case, it is clear that the lookahead should finally decide. This state actually serves as an example of bottom-of-stack state, where, as said above, only items of the form $[i, S' \to \bullet \vdash S \dashv, e]$ can appear. The generator simply discriminates on the lookahead, because at this point, since the grammar is LR(1), it must allow to discriminate amongst all the legal possibilities. This is the case in state 5, where lookahead $\dashv$ calls for reduction 3 —the decision that LALR(1) was unable to build—, while lookahead values $=$, $+$, and $*$, call for reduction 6.

1: $S' \rightarrow \vdash prog \dashv$
2: $prog \rightarrow \textbf{with}\ declist\ \textbf{eval}\ explist$
3: $declist \rightarrow decl; declist$
4: $declist \rightarrow decl$
5: $explist \rightarrow exp; explist$
6: $explist \rightarrow exp$
7: $decl \rightarrow var : type$
8: $decl \rightarrow type = type$
9: $exp \rightarrow var = exp$
10: $exp \rightarrow var + var$
11: $exp \rightarrow var$
12: $exp \rightarrow (exp)$
13: $exp \rightarrow type(exp)$
14: $var \rightarrow id$
15: $type \rightarrow id$

Figure 5.41: $\mathcal{G}_{pdt}$: LR(1) non-LR(0) programming-like grammar.

5.6.2 Non-LALR(1) LR(1) grammars

In this section several examples of LR(1) grammars which are not LALR(1) are presented. They include the examples of Chapter 3, the example from Pager's paper on merger compatibility [Pag77c] reviewed in Section 2.3.1, and the standard example of non-LALR(1) LR(1) grammar. We shall present their corresponding parsing tables and briefly analize the origin of the non-LALR(1)-ness.

A small programming-like grammar

Figure 5.41 shows again grammar $\mathcal{G}_{pdt}$ of Figure 3.2 (page 34). Its corresponding yacc-generated automaton[10] is shown in Figure 5.42. The output from yacc shows a conflict in state 7. Thus it is not a LALR(1) grammar although it is LR(1), and thus a valid discriminating reverse LR(1) automaton can be built, which is shown in Figure 5.43, and whose corresponding discriminating reverse LR(1) parsing table is shown in Figure 5.44, containing 68 entries. It can be compared with the direct LR(1) parsing table shown in Figure 5.45, containing 119 entries.

From the direct LR(1) items, we can see that the LALR(1) conflict appears in the LALR(1) state resulting from the merger of the following three

[10] Note that the automaton shown has been generated by yacc, which usually compresses the the table. The actual "LALR(1)" table has 34 states and contains 94 entries.

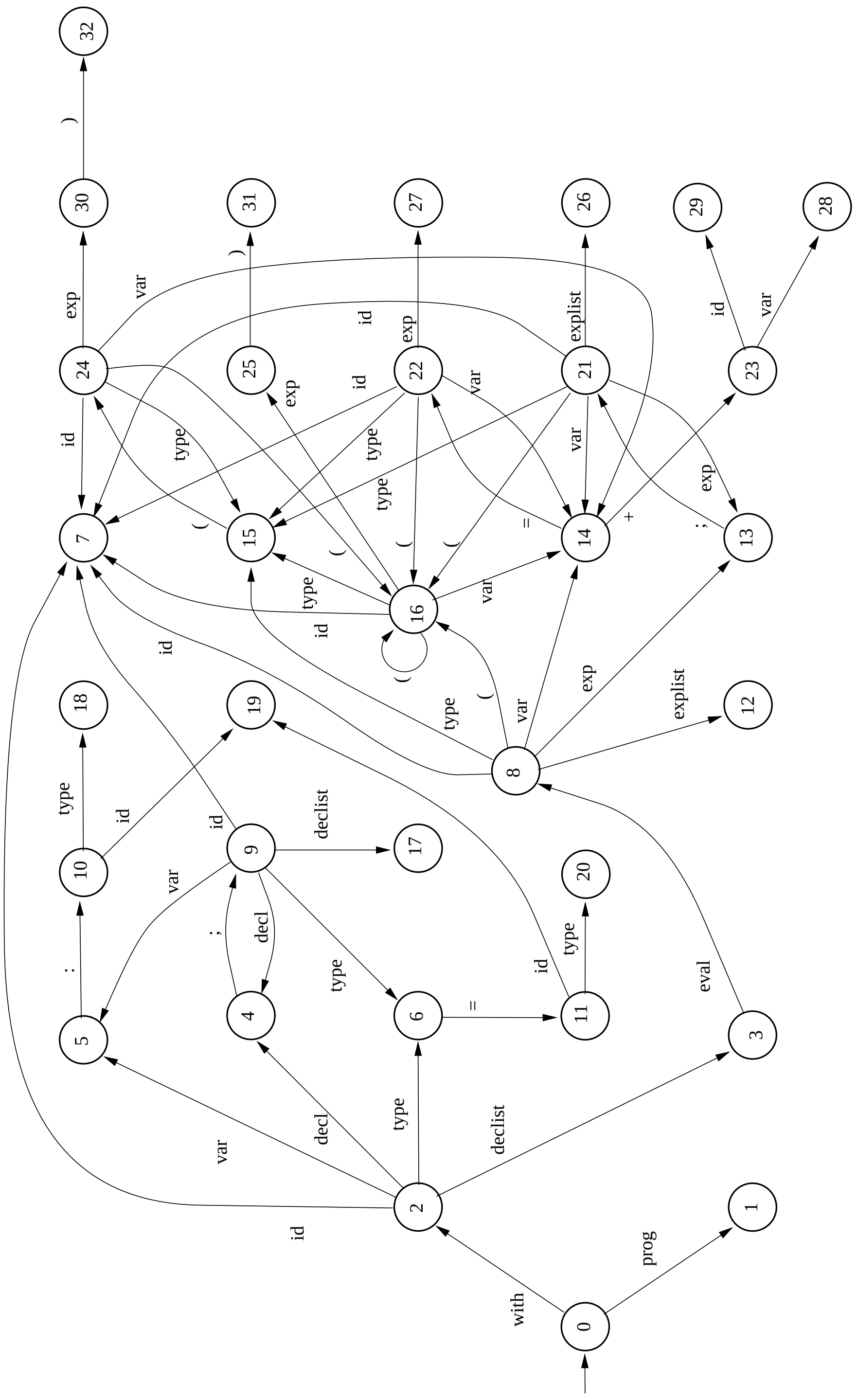


Figure 5.42: Yacc-generated automaton for LR(1) non-LALR(1) grammar with conflict in state 7.

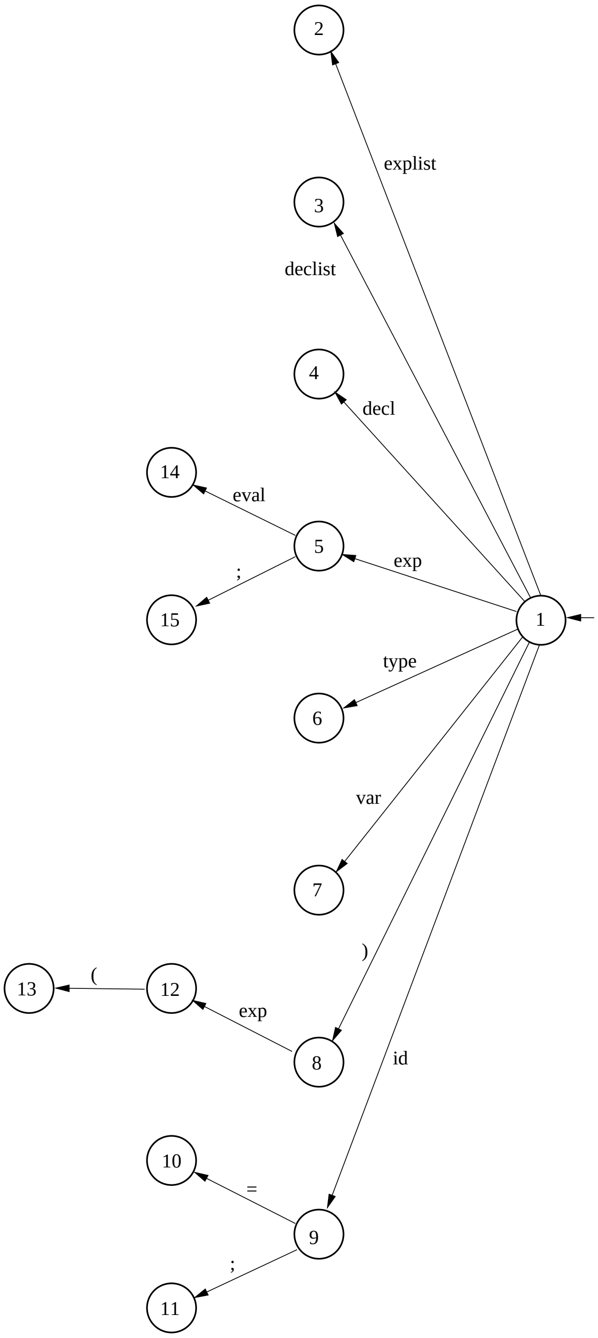


Figure 5.43: Discriminating reverse LR(1) automaton for non-LALR(1) grammar.

	$\dashv$	*with*	*eval*	;	:	=	+	(	)	*id*
s1		0								0
s2										
s3										
s4			4	0						
s5										
s6						0		0		
s7					0	0	0			
s8										
s9	14		15		14		14	15	14	
s10						14				
s11				14						
s12										
s13										
s14	6			0						
s15	6			0						

	$\vdash$	$\dashv$	*prog*	*with*	*declist*	*eval*	*explist*	*decl*	;	*exp*	*var*	:	*type*	=	+	(	)	*id*
s1		1	0		s3	0	s2	s4	0	s5	s7		s6	0		0	s8	s9
s2						2			5									
s3				0					3									
s4																		
s5						s14			s15					9		0		
s6												7		8				
s7						11			11					11	10	11		
s8										s12								
s9				15		14			s11			15		s10	14	14		
s10											14		15					
s11								15		14								
s12																s13		
s13						12			12				13	12		12		
s14																		
s15																		

Figure 5.44: Discriminating reverse LR(1) parsing table for grammar $\mathcal{G}_{pdt}$.

	⊣	*with*	*eval*	;	:	=	+	(	)	*id*	*prog*	*declist*	*explist*	*decl*	*exp*	*var*	*type*
s1		s3									2						
s2	s4																
s3										s9		5		6		7	8
s4	r1																
s5			s10														
s6			r4	s11													
s7					s12												
s8						s13											
s9					r14	r15											
s10								s17		s19			14		15	16	18
s11										s9		20		6		7	8
s12										s22							21
s13										s22							23
s14	r2																
s15	r6			s24													
s16	r11			r11		s25	s26										
s17								s29		s31					27	28	30
s18								s32									
s19	r14			r14		r14	r14	r15									
s20			r3														
s21			r7	r7													
s22			r15	r15													
s23			r8	r8													
s24								s17		s19			33		15	16	18
s25								s17		s19					34	16	18
s26										s36						35	
s27									s37								
s28						s38	s39		r11								
s29								s29		s31					40	28	30
s30								s41									
s31						r14	r14	r15	r14								
s32								s29		s31					42	28	30
s33	r5																
s34	r9			r9													
s35	r10			r10													
s36	r14			r14													
s37	r12			r12													
s38								s29		s31					43	28	30
s39										s45						44	
s40									s46								
s41								s29		s31					47	28	30
s42									s48								
s43									r9								
s44									r10								
s45									r14								
s46									r12								
s47									s49								
s48	r13			r13													
s49									r13								

Figure 5.45: Direct LR(1) parsing table for grammar $\mathcal{G}_{pdt}$.

LR(1) states:

State 9
$[var \to id\bullet, :]$
$[type \to id\bullet, =]$

State 19
$[var \to id\bullet, =]$
$[var \to id\bullet, +]$
$[var \to id\bullet, ;]$
$[type \to id\bullet, (]$
$[var \to id\bullet, \dashv]$

State 31
$[var \to id\bullet, =]$
$[var \to id\bullet, +]$
$[var \to id\bullet,)]$
$[type \to id\bullet, (]$

In the state resulting from the merger, items $[type \to id\bullet, =]$ and $[var \to id\bullet, =]$ call for two different, conflicting reductions on the same lookahead =.

The origin of the problem is, of course, that some relevant information from the left-hand context is lost during direct LR(1) state merger. According to the grammar,

$$\ldots id = \ldots$$

can appear in two possible contexts: a type declaration —which is preceded by the context keyword **with** introducing a declaration section— or an expression assignment —then preceded by the keyword **eval**. But this context information is lost during LALR(1) construction, although for the reverse stack exploration, as the reverse automaton and table of Figures 5.43 and 5.44 show, the current context is rapidly detected following the initial state transition on *id* to state 9. Actually, the exploration does not need to reach a context keyword like **with** or **eval**, but it suffices to find some clue about the current context. For instance, state 9 contains, amongst others, the following two items with lookahead symbol =:

$$[14, exp \to (\bullet exp), = \bullet]$$
$$[14, exp \to type(\bullet exp), = \bullet]$$

which permit to see at this point that "(" is only legal within the expression section, and thus *id* has to reduce to *var* instead of *type*. Nevertheless, neither stack symbol = nor ";" permit to decide in state 9. After transition on =, state 10 shows the following set of items:

State 10
$$[15, decl{\rightarrow}type\bullet = type, \bullet;]$$
$$[14, exp{\rightarrow}var\bullet = exp, \bullet;]$$
$$[14, exp{\rightarrow}var\bullet = exp, = \bullet]$$

Thus nonterminal *var* permits to decide that the current context is the expression section, and thus *id* has to reduce to *var* (rule 14), while nonterminal *decl* permits to decide that the context is the declaration section, and thus *id* has to reduce to *type* (rule 15). This is a typical situation where the different context contain different nonterminals which permit to rapidly determine the current context, and thus the right parsing action. A similar situation appears in state 11, which contains the following set of items:

State 11
$$[14, explist{\rightarrow}exp\bullet; explist, ; \bullet]$$
$$[14, explist{\rightarrow}exp\bullet; explist, = \bullet]$$
$$[15, declist{\rightarrow}decl\bullet; declist, = \bullet]$$

In this case, nonterminals *exp* and *decl* evidently signal the respective contexts for expressions and declarations.

In fact, right recursion is used in grammar $\mathcal{G}_{pdt}$ for *declist* and *explist*. Nevertheless, changing to left recursion would not change the LR(1), non-LALR(1) character of the grammar. There would also be no impact on the work for the reverse discrimination, although this work might be less in some cases if the stack is shorter, as is normally the case when using left recursion. Again, changing the grammar to more complex expressions would not solve the problem of context detection just discussed.

From a practical point of view, not only LR(1) grammar are more powerful and thus a full LR(1) parser offers more flexibility to the designer to write his grammar according to what the application demands, but at the same time, the kind of error messages he receives in these cases from his LALR(1) parser generation tool are more confusing. It is not easily understandable that an important syntactical context information as **with** or **eval** in our example, has been neglected or "forgotten" by his parser generator because of some (usually virtual) merger of some unseen LR(1) states and items. What is more, he cannot distinguish non-LALR(1)-ness from ambiguities. On the other hand, of course, these non-LALR(1)-ness problems do not exist for LR(1) grammars, but when the grammar is not LR(1) —maybe ambiguous or LR(k), $k > 1$—, the problem is much clearer and simpler to state and understand: since at this point (current point of exploration plus lookahead) there are several possibilities which are all legal, it is impossible to decide.

$$
\begin{array}{ll}
1: & S' \rightarrow \vdash S \dashv \\
2: & S \rightarrow aAa \\
3: & S \rightarrow bAb \\
4: & S \rightarrow aBb \\
5: & S \rightarrow bBa \\
6: & A \rightarrow c \\
7: & B \rightarrow c
\end{array}
$$

Figure 5.46: $\mathcal{G}_{nla}$: a usual example of LR(1) non-LALR(1) grammar.

	$\dashv$	a	b	c	S	A	B
s1		s3	s4		2		
s2	s5						
s3				s8		6	7
s4				s8		9	10
s5	r1						
s6		s11					
s7			s12				
s8		{r6,r7}	{r7,r6}				
s9			s13				
s10		s14					
s11	r2						
s12	r4						
s13	r3						
s14	r5						

Figure 5.47: Direct LALR(1) parsing table for grammar $\mathcal{G}_{nla}$ with two conflicts in state 8.

Moreover, for a human it will be equally impossible to decide, unless he reads ahead for a while and comes back, which is feeled to be unnatural. On the other hand, it is quite natural for a human to "remember" (from what has been read above the current point) what is the current context, and thus to interpret correctly the text. In this sense, we may say that LR(1) is much more natural (in some sense the most powerful and natural, at the same time) than LALR(1).

The usual example grammar

Grammar $\mathcal{G}_{nla}$ shown in Figure 5.46 is the usual example of LR(1) non-LALR(1) grammar. Figure 5.47 shows the corresponding direct LALR(1) parsing table with two conflicts for rules 6 and 7 at state 8. The corresponding discriminating reverse and direct LR(1) parsing tables for this grammar are shown in Figures 5.48 and 5.49, respectively.

Evidently, the problem with this grammar is that both left-hand and right-hand context are necessary in order to know the right reduction for c.

	$\dashv$	a	b	c	$\vdash$	$\dashv$	S	a	A	b	B	c
s1				0	0	1	0	s2	0	s3	0	s4
s2									2		5	
s3									3		4	
s4								s5		s6		
s5		6	7									
s6		7	6									

Figure 5.48: Discriminating reverse LR(1) parsing table for $\mathcal{G}_{nla}$.

	$\dashv$	a	b	c	S	A	B
s1		s3	s4		2		
s2	s5						
s3				s8		6	7
s4				s11		9	10
s5	r1						
s6		s12					
s7			s13				
s8		r6	r7				
s9			s14				
s10		s15					
s11		r7	r6				
s12	r2						
s13	r4						
s14	r3						
s15	r5						

Figure 5.49: Direct LR(1) parsing table for $\mathcal{G}_{nla}$.

			$\vdash$	$acb\dashv$
			0	
		$\vdash$	a	$cb\dashv$
			0^*	
	$\vdash$	a	c	$b\dashv$
	7^*	s5	s4	
	$\vdash$	a	B	$b\dashv$
			0	
$\vdash$	a	B	b	$\dashv$
	[]	4	s3	
		$\vdash$	S	$\dashv$
			0	
	$\vdash$	S	$\dashv$	
	[]	[]	1	

Figure 5.50: Discriminating reverse LR(1) parsing of sentence $\vdash acb\dashv$ for grammar $\mathcal{G}_{nla}$.

In the discriminating reverse LR(1) automaton, after transition on c from the intial state, state 4 is reached, containing the following item set (the whole list of item sets is shown in Appendix A).

State 4

$$\begin{array}{l} [6, A\to\bullet c, \bullet a] \\ [6, A\to\bullet c, \bullet b] \\ [7, B\to\bullet c, \bullet a] \\ [7, B\to\bullet c, \bullet b] \\ [6, S\to a\bullet Aa, a\bullet] \\ [6, S\to b\bullet Ab, b\bullet] \\ [7, S\to b\bullet Ba, a\bullet] \\ [7, S\to a\bullet Bb, b\bullet] \end{array}$$

Here we have four triples: $(6, a, a)$, $(6, b, b)$, $(7, b, a)$, and $(7, a, b)$, which of course would permit to decide in this state. Nonetheless, this decision is taken in two steps, according to the generation and parsing algorithms shown before. In particular, after transition on a, state 5 is reached, containing the following items:

State 5

$$\begin{array}{l} [6, S\to\bullet aAa, a\bullet] \\ [7, S\to\bullet aBb, b\bullet] \\ [6, S'\to\vdash\bullet S\dashv, a\bullet] \\ [7, S'\to\vdash\bullet S\dashv, b\bullet] \end{array}$$

Here, clearly lookahead a calls for reduction 6, while b calls for reduction 7. Something analogous occurs in state 7.

The reader might want to see Figure 5.50, where the parsing process for sentence $\vdash acb\dashv$ is shown.

$$
\begin{array}{ll}
1: & S' \rightarrow \vdash S \dashv \\
2: & S \rightarrow A \\
3: & S \rightarrow Bb \\
4: & S \rightarrow bB \\
5: & S \rightarrow bAc \\
6: & B \rightarrow aA \\
7: & A \rightarrow a \\
8: & A \rightarrow \varepsilon
\end{array}
$$

Figure 5.51: $\mathcal{G}_{sba}$: LR(1), non-LALR(1) grammar.

	$\dashv$	b	c	a	$\vdash$	$\dashv$	S	A	B	b	c	a
s1				0	s6	1	0	s2	s4	s3	5	s5
s2		6	0		2							6
s3	3		8									
s4	4	0										
s5			7		s7					8		7
s6	8	0										
s7	7	8										

Figure 5.52: Discriminating reverse LR(1) parsing table for $\mathcal{G}_{sba}$.

A variant

Figure 5.51 shows again grammar $\mathcal{G}_{sba}$ of Figure 3.1 (page 34), which is a variant of the grammar we have just seen. This time, the LALR(1) conflict involves an empty rule[11] —numbered 8— and a nonempty rule —numbered 7. The corresponding discriminating reverse and direct LR(1) parsing tables for this grammar are shown in Figures 5.52 and 5.53, respectively. Figure 5.54 shows the corresponding direct LALR(1) parsing table, showing the conflict in state 6.

A more complicated grammar

Figure 5.55 shows a more complicated non-LALR(1) LR(1) grammar [WG84]. The corresponding discriminating reverse and direct LR(1) parsing tables for this grammar are shown in Figures 5.56 and 5.57, respectively. Figure 5.58 shows the corresponding direct LALR(1) parsing table showing the conflict in state 6, which results from the merger of direct LR(1) states 6, 11, 16, and 24, while the discriminating reverse LR(1) parser quickly solves the decision problem after transition on top-of-stack (and handle) b to state 3, where the current lookahead or else the second stack symbol suffices to decide, as its table of compatible actions of Figure 5.59 clearly shows.

[11] Actually, both rules may be empty in an LALR conflict. Moreover, in general any number of rules may be involved in an LALR conflict.

	$\dashv$	b	c	a	S	A	B
s1	r8	s5		s6	2	3	4
s2	s7						
s3	r2						
s4		s8					
s5			r8	s11		10	9
s6	r7	r8		s13		12	
s7	r1						
s8	r3						
s9	r4						
s10			s14				
s11	r8		r7	s16		15	
s12		r6					
s13		r7					
s14	r5						
s15	r6						
s16	r7						

Figure 5.53: Direct LR(1) parsing table for $\mathcal{G}_{sba}$.

	$\dashv$	b	c	a	S	A	B
s1	r8	s5		s6	2	3	4
s2	s7						
s3	r2						
s4		s8					
s5			r8	s6		10	9
s6	{r7,r8}	r8	r7	s12		11	
s7	r1						
s8	r3						
s9	r4						
s10			s13				
s11	r6	r6					
s12	r7	r7					
s13	r5						

Figure 5.54: Direct LALR(1) parsing table for grammar $\mathcal{G}_{sba}$ with conflict in state 6.

$$\begin{array}{rl}
1: & S' \rightarrow \vdash A \dashv \\
2: & A \rightarrow aBcB \\
3: & A \rightarrow B \\
4: & A \rightarrow D \\
5: & B \rightarrow b \\
6: & B \rightarrow Ff \\
7: & D \rightarrow dE \\
8: & E \rightarrow FcA \\
9: & E \rightarrow FcE \\
10: & F \rightarrow b
\end{array}$$

Figure 5.55: $\mathcal{G}_{wag}$: LR(1), non-LALR(1) grammar.

	$\dashv$	a	c	b	f	d	$\vdash$	$\dashv$	A	a	B	c	D	b	F	f	d	E
s1		0		0		0		1	s5		s2		4	s3	0	6		s4
s2			0				3					s6						
s3	5				10					5		10					10	
s4												9					7	
s5							0					8						
s6											2				3			

Figure 5.56: Discriminating reverse LR(1) parsing table for $\mathcal{G}_{wag}$.

Pager's algorithm example grammar

In Section 2.3.1 we reviewed the method by Pager for merging "on the fly" LR(1) states according to a weak or strong compatibility criterion. Figure 5.60 shows the non-LALR(1) LR(1) grammar that Pager proposes in his paper [Pag77c] to illustrate his algorithm. Figure 5.61 shows the corresponding LALR(1) parser, which shows a conflict in state 17. The corresponding canonical LR(1) parser is shown in Figure 5.62, with 126 entries.

In effect, Pager's algorithm with compatibility criteria produces a conflict-free LR(1) parser with slightly more states than the LALR(1) conflicting table, as it is usually the case for non-LALR(1) grammars, and even for some LALR(1) grammars. In this case both the weak and strong compatibility tables are equal. In fact, the number of states for LALR(1), which is the same as the LR(0) automaton, is an absolute minimum for direct LR(1) automata resulting from compatibility criteria for states with same core, as Pager's.

Figure 5.63 shows Pager's parsing table, with 67 entries. The corresponding discriminating reverse LR(1) parsing table, with 30 entries, is shown in Figure 5.64.

	$\dashv$	a	c	b	f	d	A	B	D	F	E
s1		s3		s6		s8	2	4	5	7	
s2	s9										
s3				s11				10		12	
s4	r3										
s5	r4										
s6	r5				r10						
s7					s13						
s8				s16						15	14
s9	r1										
s10			s17								
s11			r5		r10						
s12					s18						
s13	r6										
s14	r7										
s15			s19								
s16			r10								
s17				s6				20		7	
s18			r6								
s19		s3		s24		s8	21	4	5	23	22
s20	r2										
s21	r8										
s22	r9										
s23			s19		s13						
s24	r5		r10		r10						

Figure 5.57: Direct LR(1) parsing table for $\mathcal{G}_{wag}$.

	$\dashv$	a	c	b	f	d	A	B	D	F	E
s1		s3		s6		s8	2	4	5	7	
s2	s9										
s3				s6				10		7	
s4	r3										
s5	r4										
s6	r5		{r5,r10}		r10						
s7					s11						
s8				s14						13	12
s9	r1										
s10			s15								
s11	r6		r6								
s12	r7										
s13			s16								
s14			r10								
s15				s6				17		7	
s16		s3		s6		s8	18	4	5	20	19
s17	r2										
s18	r8										
s19	r9										
s20			s16		s11						

Figure 5.58: Direct LALR(1) parsing table for grammar $\mathcal{G}_{wag}$ with conflict in state 6.

s3	$\dashv$	c	f
$\vdash$	$\{5\}$	$\emptyset$	$\{10\}$
a	$\emptyset$	$\{5\}$	$\{10\}$
c	$\{5\}$	$\{10\}$	$\{10\}$
d	$\emptyset$	$\{10\}$	$\emptyset$

Figure 5.59: Table of compatible actions for state 3 of Figure 5.56.

$$
\begin{array}{rl}
1: & S' \to \vdash X \dashv \\
2: & X \to aYd \\
3: & X \to aZc \\
4: & X \to aT \\
5: & X \to bYe \\
6: & X \to bZd \\
7: & X \to bT \\
8: & Y \to tW \\
9: & Y \to uX \\
10: & Z \to tu \\
11: & T \to uXa \\
12: & W \to uV \\
13: & V \to \varepsilon
\end{array}
$$

Figure 5.60: $\mathcal{G}_{pag}$: non-LALR(1) LR(1) grammar by Pager.

	$\dashv$	a	d	c	b	e	t	u	X	Y	Z	T	W	V
s1		s3			s4				2					
s2	s5													
s3							s9	s10		6	7	8		
s4							s9	s10		11	12	13		
s5	r1													
s6			s14											
s7				s15										
s8	r4	r4	r4			r4								
s9								s17					16	
s10		s3			s4				18					
s11						s19								
s12			s20											
s13	r7	r7	r7			r7								
s14	r2	r2	r2			r2								
s15	r3	r3	r3			r3								
s16			r8			r8								
s17			{r10,r13}	r10		r13								21
s18		s22	r9			r9								
s19	r5	r5	r5			r5								
s20	r6	r6	r6			r6								
s21			r12			r12								
s22	r11	r11	r11			r11								

Figure 5.61: LALR(1) parsing table for LR(1) grammar $\mathcal{G}_{pag}$, with conflict in state 17.

5.6.3 Non-LALR(1) LL(1) grammars

As we know, LR(1) grammars properly contain all LL(1) grammars, although this is not exactly the case for LALR(1) grammars. Figure 5.65 shows an example[12] of such LL(1) grammars which are not LALR(1). The reader can easily verify that, since $\text{First}(C) = \{\varepsilon, f\}$, $\text{First}(D) = \{\varepsilon, g\}$, and $\text{Follow}(F) = \text{Follow}(G) = \{a, b\}$ the LL(1) parser can always decide the rightparts to predict for A or B, and then for F or G.

The corresponding direct LALR(1) parsing table is shown in Figure 5.66, presenting two conflicts in state 9 for empty reductions to F and G in front of lookahead symbols a and b. This state "results" from merging two full LR(1) states containing item sets

$$
\begin{array}{l}
[C \to E \bullet F, a] \\
[D \to E \bullet G, b] \\
[F \to \bullet, a] \\
[F \to \bullet f, a] \\
[G \to \bullet, b] \\
[G \to \bullet g, b]
\end{array}
$$

[12] Inspired from non-LALR(k) SLL(1) grammar of exercise 8.51 in [SSS90].

	$\dashv$	a	d	c	b	e	t	u	X	Y	Z	T	W	V
s1		s3			s4				2					
s2	s5													
s3							s9	s10		6	7	8		
s4							s14	s15		11	12	13		
s5	r1													
s6			s16											
s7				s17										
s8	r4													
s9								s19					18	
s10		s21			s22				20					
s11						s23								
s12			s24											
s13	r7													
s14								s26					25	
s15		s28			s29				27					
s16	r2													
s17	r3													
s18			r8											
s19			r13	r10										30
s20		s31	r9											
s21							s9	s35		32	33	34		
s22							s14	s39		36	37	38		
s23	r5													
s24	r6													
s25						r8								
s26			r10			r13								40
s27		s31				r9								
s28							s9	s44		41	42	43		
s29							s14	s48		45	46	47		
s30			r12											
s31	r11													
s32			s49											
s33				s50										
s34		r4	r4											
s35		s21			s22				51					
s36						s52								
s37			s53											
s38		r7	r7											
s39		s28			s29				54					
s40						r12								
s41			s55											
s42				s56										
s43		r4				r4								
s44		s21			s22				57					
s45						s58								
s46			s59											
s47		r7				r7								
s48		s28			s29				60					
s49		r2	r2											
s50		r3	r3											
s51		s61	r9											
s52		r5	r5											
s53		r6	r6											
s54		s61				r9								
s55		r2				r2								
s56		r3				r3								
s57		s62	r9											
s58		r5				r5								
s59		r6				r6								
s60		s62				r9								
s61		r11	r11											
s62		r11				r11								

Figure 5.62: Canonical LR(1) parsing table for non-LALR(1) grammar $\mathcal{G}_{pag}$.

	$\dashv$	a	d	c	b	e	t	u	X	Y	Z	T	W	V
s1		s3			s4				2					
s2	s5													
s3							s9	s10		6	7	8		
s4							s14	s10		11	12	13		
s5	r1													
s6			s15											
s7				s16										
s8	r4	r4	r4			r4								
s9								s18					17	
s10		s3			s4				19					
s11						s20								
s12			s21											
s13	r7	r7	r7			r7								
s14								s22					17	
s15	r2	r2	r2			r2								
s16	r3	r3	r3			r3								
s17			r8			r8								
s18			r13	r10										23
s19		s24	r9			r9								
s20	r5	r5	r5			r5								
s21	r6	r6	r6			r6								
s22			r10			r13								23
s23			r12			r12								
s24	r11	r11	r11			r11								

Figure 5.63: Pager's weak compatibility LR(1) parsing table for non-LALR(1) grammar $\mathcal{G}_{pag}$.

	$\dashv$	a	d	c	b	e	t	u	$\vdash$	$\dashv$	X	a	Y	d	Z	c	T	b	e	t	W	u	V
s1					0		0	0	0	1	s4	11	0	s2	0	3	s3		5		8	s5	12
s2													2		6								
s3												4						7					
s4	0	0	9			9																	
s5		0		10		13														s6			
s6												13						10					

Figure 5.64: Discriminating reverse LR(1) parsing table for non-LALR(1) grammar $\mathcal{G}_{pag}$.

$$
\begin{array}{rl}
1: & S' \to \vdash S \dashv \\
2: & S \to aA \\
3: & S \to bB \\
4: & A \to Ca \\
5: & A \to Db \\
6: & B \to Da \\
7: & B \to Cb \\
8: & C \to EF \\
9: & D \to EG \\
10: & E \to \varepsilon \\
11: & F \to \varepsilon \\
12: & F \to f \\
13: & G \to \varepsilon \\
14: & G \to g
\end{array}
$$

Figure 5.65: $\mathcal{G}_{llr}$: non-LALR(1) LL(1) grammar.

	$\dashv$	a	b	f	g	S	A	B	C	D	E	F	G
s1		s3	s4			2							
s2	s5												
s3		r10	r10	r10	r10		6		7	8	9		
s4		r10	r10	r10	r10			10	12	11	9		
s5	r1												
s6	r2												
s7		s13											
s8			s14										
s9		{r11,r13}	{r13,r11}	s17	s18							15	16
s10	r3												
s11		s19											
s12			s20										
s13	r4												
s14	r5												
s15		r8	r8										
s16		r9	r9										
s17		r12	r12										
s18		r14	r14										
s19	r6												
s20	r7												

Figure 5.66: Direct LALR(1) parsing table for LL(1) grammar $\mathcal{G}_{llr}$ with conflicts in state 9.

	$\dashv$	a	b	f	g	$\vdash$	$\dashv$	S	a	A	b	B	C	D	E	F	G	f	g
s1						0	1	0	s2	2	s3	3	0	0	s6	s4	s5	12	14
s2		10	10	10	10								4	6					
s3		10	10	10	10								7	5					
s4		8	8																
s5		9	9																
s6				0	0				s7		s8								
s7		11	13																
s8		13	11																

Figure 5.67: Discriminating reverse LR(1) parsing table for $\mathcal{G}_{llr}$.

and

$$\begin{array}{l}
[D{\rightarrow}E{\bullet}G, a]\\
[C{\rightarrow}E{\bullet}F, b]\\
[G{\rightarrow}{\bullet}, a]\\
[G{\rightarrow}{\bullet}g, a]\\
[F{\rightarrow}{\bullet}, b]\\
[F{\rightarrow}{\bullet}f, b]
\end{array}$$

into the referred LALR(1) state 9. The discriminating reverse LR(1) parser, whose table is shown in Figure 5.67, easily solves this decision problem in the same way of preceding examples, in states 7 and 8, which are reached after transition from initial state on stack symbols E and then a or b.

Again, the problem for the LALR(1) parser is that some essential information from the input string —i.e., whehter there is an a or a b as left-hand context— is lost due to state merger, although this information is directly available to the LL(1) parser for deciding between the two rightparts of S. Note that in order for the grammar to be both LL(1) and non-LALR(1), it is critical the presence in rules for C and D of nonterminal E deriving only ε.

5.7 Formal statements

For the sake of completeness and reference, the most important definitions and properties regarding the LR(1) discriminating reverse automaton are given in this section. Theorems are just enunciated. They are proved for the general LR(k) case in Section 7.5. Alternatively, some conventions and definitions given here are also applicable to the general case.

Definition 5.1

$$\mathrm{First}(\alpha) = \{1 : x \mid \alpha{\Rightarrow^*}x\}.$$

$$\mathrm{Follow}(A) = \{1 : x{\dashv} \mid S'{\Rightarrow^*}\alpha A x\}.$$

□

Definition 5.2 *A grammar $\mathcal{G}$ is said to be LR(1), iff*

$$\left.\begin{array}{l} S' \underset{\mathrm{rm}}{\Rightarrow}^* \zeta A z \underset{\mathrm{rm}}{\Rightarrow} \zeta \alpha z, \\ S' \underset{\mathrm{rm}}{\Rightarrow}^* \zeta' A' z' \underset{\mathrm{rm}}{\Rightarrow} \zeta \alpha y, \ \textit{and} \\ 1 : z = 1 : y \end{array}\right\} \textit{implies } \zeta' A' z' = \zeta A y.$$

□

Definition 5.3 *An LR(1) reverse item has one of the following two forms:*

$$\begin{array}{l} [i, B \to \gamma \bullet \varphi, \bullet a], \ \textit{or} \\ [i, B \to \gamma \bullet \varphi, a \bullet], \end{array}$$

where $|P'| \geq i \geq 0$, and $B \to \gamma\varphi \in P'$.

Alternatively, it can be equally defined as having the form $[i, B \to \gamma \bullet \varphi, x \bullet y]$, with $|xy| = 1$. □

The following is an extension of the prefix operator which results in convenient notation for reverse automata.

Definition 5.4 *Given a* dotted *sequence $x \bullet y$,*

$$1 : x \bullet y = x' \bullet y' \ \textit{iff}\ x' = 1 : x \ \textit{and}\ x'y' = 1 : xy.$$

□

Definition 5.5 *Given a grammar $\mathcal{G}$, the closure of a set q of LR(1) reverse items is defined by:*

$$\begin{array}{rl} \mathcal{C}(q) = q \ \cup & \{[i,\ B \to \beta \bullet \gamma,\ xy \bullet z] \mid [i,\ A \to \bullet \alpha,\ x \bullet yz] \in q,\ \gamma \underset{\mathrm{rm}}{\Rightarrow}^* Av, \\ & B \to \beta\gamma \in P',\ S' \underset{\mathrm{rm}}{\Rightarrow}^* \varphi B w,\ \textit{and}\ xy \bullet z = 1 : xv \bullet w\dashv\}. \end{array}$$

□

Definition 5.6 *Given a grammar $\mathcal{G}$, the automaton $\mathcal{A}$ is constructed as follows:*

1. *Initial state:*

$$\begin{array}{rcl} q_0 & = & \mathcal{C}(\{[0,\ A \to \alpha \bullet a\beta,\ a] \mid A \to \alpha a \beta \in P'\} \\ & & \cup \{[i,\ A \to \alpha \bullet,\ 1 : \bullet w\dashv] \mid A \xrightarrow{i} \alpha \in P' \ \textit{and}\ S' \underset{\mathrm{rm}}{\Rightarrow}^* \varphi A w\}) \end{array}$$

2. *Transition function:*

$$\delta(q, \sigma) = \left\{ \begin{array}{ll} \sigma = \varepsilon : & q \\ \sigma = X : & \mathcal{C}(\{[i,\ A \to \alpha \bullet X\beta,\ x \bullet y] \mid \\ & \qquad [i,\ A \to \alpha X \bullet \beta,\ x \bullet y] \in q\}) \\ \sigma = X\sigma_1 : & \delta(\delta(q, \sigma_1), X) \end{array} \right.$$

□

Theorem 5.7 *Given a grammar $\mathcal{G}$ and an automaton $\mathcal{A}$ constructed according to Definition 5.6,*

$$[i,\ B{\rightarrow}\beta{\bullet}\rho,\ v{\bullet}x] \in \delta(q_0,\sigma)\ \textit{iff}\ \begin{cases} B\xrightarrow{h}\beta\rho,\ \rho\underset{\mathrm{rm}}{\overset{\pi_2}{\Longrightarrow}}{}^{*}\sigma yw, \\ B\underset{\mathrm{rm}}{\overset{\pi_1}{\Longrightarrow}}{}^{*}\varphi Aw,\ \pi_1 j = h\pi_2, \\ S'\underset{\mathrm{rm}}{\Rightarrow}{}^{*}\mu Bz,\ \textit{and}\ v{\bullet}x = 1 : yw{\bullet}z\dashv \end{cases}$$

when $i = 0$ and $y \neq \varepsilon$ or $i = j$ and $y = \varepsilon$.

Proof. □

Definition 5.8 *For a given grammar $\mathcal{G}$, stack suffix plus lookahead $\sigma | a$, is said to be* compatible *with action i iff there is some stack form $\eta\sigma | ax$ compatible with i. A stack suffix plus lookahead with at least one compatible action is said to be* valid. □

For LR(1) grammars, valid full-stack plus lookahead configurations always have only one compatible action.

Definition 5.9 *A stack suffix plus lookahead $\sigma | a$ is called* discriminant *iff*

1. *it has only one compatible action, or else,*
2. *for all but one of the actions compatible with σ, for each compatible $\sigma | a'$, there is some suffix σ_0, $\sigma = X\sigma_1\sigma_0$, such that $\sigma_0 | a'$ has a unique compatible action and $a' \neq a$.*

That compatible action is said to be discriminated *by $\sigma | a$.* □

Note that, according to the definition, $\sigma | a$ does not need to be compatible with any action, when condition 1 does not apply. According to the following proposition, this is the case only for erroneous inputs, since for sentences, all stack suffix plus lookahead configurations are necessarily compatible with some action.

Proposition 5.10 *If stack suffix plus lookahead $\sigma | a$ discriminates, and at the same time is compatible with, action i, then i is the only compatible action.*

Proof. □

Using the above definition, discriminating reverse parsing is characterized by the following theorem.

Theorem 5.11 *The discriminating reverse LR(1) automaton for an LR(1) grammar $\mathcal{G}$ performs a parsing action $\eta' \,|\, z' \overset{i}{=\!\!=\!\!\!|}\, \eta \bullet \sigma \,|\, z$ if, and only if, σ is the shortest valid suffix of $\eta\sigma$ such that $\sigma \,|\, 1 : z$ discriminates action i. Otherwise, an error is signalled after exploring the shortest σ such that condition 2 in Definition 5.9 holds for all compatible actions, if any, with no exception.*

Proof. $\square$

Theorem 5.12 *The discriminating reverse LR(1) automaton for an LR(1) reduced grammar $\mathcal{G}$ parses any input x in time $O(|x|)$.*

Proof. $\square$

Corollary 5.13 *The discriminating reverse LR(k) automaton does not loop forever on any finite input.*

Proof. $\square$

Theorem 5.14 *The discriminating reverse LR(1) automaton for an LR(1) reduced grammar $\mathcal{G}$ parses any input x in time $O(|x|)$, including rightpart verification.*

Proof. $\square$

Chapter 6

Performance

In this chapter, the practical validity of the method will be analyzed from several points of view. We shall first study different cases of grammars that may lead the reverse parser to do some specially deep explorations, and verify in all cases the linearity[1] of the method. We shall also briefly analyze the behavior in case of erroneous input. Next, the implementation of an experimental tool by the author is described, and the test results obtained are shown in detail and evaluated. The main conclusion of this chapter is that small parser size and exploration time are verified in practice. Discriminating reverse full LR tables are considerably smaller than LALR(1) tables. Not only the method is linear, but in practice the overhead due to stack exploration is negligible, with an average depth per action of 2 symbols.

6.1 Parsing efficiency

Parsing efficiency is one of the major concerns for the discriminating reverse approach, since the idea of exploring the stack-plus-lookahead to take a parsing decision is intrinsic to the method. In contrast, in the classical approach no exploration is performed (just looking at the topmost stack symbol or code plus the current lookahead terminal) or a small stack exploration can be performed, whose scope is bounded from the outset.

As it has been shown in previous sections, in practical cases this concern is not really relevant. Nonetheless, it may be important to know from a theoretical point of view what are the possible weaknesses of the method regarding stack exploration cost. Experimental tests always show a constant —and very small— stack depth exploration per parsing action, independently of the length of the parsed sentences. This strongly suggests that the method is linear in exploration cost with the length of the sentence[2]. In

[1]Linearity is formally proved in Chapter 7 for the general LR(k) case

[2]As in fact it is, as we shall formally prove for the general case in Chapter 7.

$$\begin{array}{ll} 1: & S' \rightarrow \vdash S \dashv \\ 2: & S \rightarrow \varepsilon \\ 3: & S \rightarrow Saaaaa \end{array}$$

Figure 6.1: $\mathcal{G}_{aaa}$: grammar with rightpart-dependent average depth of exploration.

this section some performance results are shown from examples for extreme cases.

The number of steps n that the discriminating automaton has to perform to recognize a whole sentence can be represented in terms of the number of stack exploration steps, i.e., the sum of depths of exploration for each action:

$$n = \sum_{i=1}^{s+r} d_i,$$

where s and r are the number of shift and reduce actions, respectively. Obviously, s is equal to the length l of the sentence, and it is a well-known fact [AU73a] that $r = O(l)$, thus $s + r = O(l)$.

Note that if the discriminating automaton has no loop, i.e., it is just like a decision tree, then the depth of stack exploration is bounded to some fixed value $D = \max(d_i)$, and thus

$$n \leq \sum_{i=1}^{s+r} D = D(s+r) = DO(l) = O(l),$$

that is,

$$n = \sum_{i=1}^{s+r} d_i = O(l).$$

Note also that this bound D is grammar-dependent. See, for instance, the grammar of Figure 6.1, whose discriminating reverse parsing table is shown in Figure 6.2. For a stack like

$$S\ a^m,\ 0 \leq m \leq 5,$$

the parser has to explore "counting" a's until finding the S to decide whether to shift or to reduce. Thus, $D = 6$ in this case, and in general $D = A + 1$, where A is the number of a's in rule 3, which is obviously constant for each grammar of this kind. Furthermore, in the average, the number of exploration steps per action is, quite approximately, $(A+1)/2$, which implies that no general bound exists for the average depth of exploration per action —although for "normal", practical grammars, this average depth is usually approximately 2.

	$\vdash$	$\dashv$	a	$\vdash$	$\dashv$	S	a
s1	0			3	1	0	s2
s2		2				0	s3
s3						0	s4
s4						0	s5
s5						0	2

Figure 6.2: Discriminating reverse LR(1) table for grammar $\mathcal{G}_{aaa}$.

$$
\begin{array}{ll}
1: & S' \rightarrow \vdash S \dashv \\
2: & S \rightarrow aA \\
3: & S \rightarrow bB \\
4: & A \rightarrow cA \\
5: & A \rightarrow d \\
6: & B \rightarrow cB \\
7: & B \rightarrow d
\end{array}
$$

Figure 6.3: $\mathcal{G}_{oue}$: grammar for unbounded exploration.

Let us first recognize that for some decisions, the discriminating reverse parser may have to explore an unbounded stack depth. For this to happen, it suffices with the existence of a loop in the discriminating automaton, i.e., the discriminating part includes one or more loops of the recognizing automaton. Consider the grammar of Figure 6.3. For stack contents

$$\vdash ac^m d,$$

the parser enters the loop in state 4, to reach the bottom of the stack to determine whether there is an a or a b in order to reduce d to A or B, respectively. The depth of exploration is unbounded, it depends only of the number of c's, which can be arbitrary large. But note that this loop is entered only once in any given sentence, and thus the whole sum of stack symbols explored for the parsing of the whole sentence is still linear with the length of the sentence. That is, all decisions are of bounded depth D, with the exception of the j-th decision, where $d_j = O(l)$. Thus, as before,

$$n = d_j + \sum_{i \neq j}^{s+r} d_i = O(l) + O(l) = O(l).$$

Now consider grammar of Figure 6.5, whose discriminating reverse parsing table is shown in Figure 6.6. If stack contents is

$$a4\ a^m,$$

in the presence of the end-of-file marker $\dashv$, the parser reduces $6: \; B0 \rightarrow \varepsilon$, and then $9: \; B1 \rightarrow B0$, both in state 1, resulting in stack contents

$$a4\ a^m\ B1.$$

	$\vdash$	$\dashv$	a	b	c	d	$\vdash$	$\dashv$	S	a	A	b	B	c	d
s1	0		0	0	0	0		1	0		s2		s3		s4
s2										2				4	
s3												3		6	
s4										5		7		s4	

Figure 6.4: Discriminating reverse LR(1) table for grammar $\mathcal{G}_{oue}$.

Then the loop at state 6 is entered to reach $a4$ in the bottom of the stack to decide reduction $12: B2{\to}B1$, then loop at state 7 for reduction $15: B3{\to}B2$, and finally loop at state 8 for reduction $18: B4{\to}B3$. From now on, $B4$, and subsequent $A4's$ are sufficient to determine the reductions, and thus no more complete, unbounded stack explorations are necessary. Thus, 3 decisions have exploration cost $O(l)$, and the rest are bounded to depth $D = 2$. Note that if there had been $a3$ (or $a2$) instead of $a4$ in the bottom of the stack, only 2 (or 1) unbounded stack explorations would have been needed, that is, their number is bounded to 3. Actually, that number is in general bounded to some grammar-dependent constant G —note that to increase this number by one would imply to increase the number of grammar rules. So, for each grammar of this type,

$$n = \sum_{k=1}^{G} d_{j_k} + \sum_{i \neq j_1,\ldots,j_G}^{s+r} d_i = GO(l) + O(l) = O(l).$$

Finally, consider the grammar of Figure 6.7, whose discriminating reverse LR(1) parsing table is shown in Figure 6.8. This grammar provokes an unbounded number of unbounded-depth stack scans. Consider, for instance, the following shift-reduce sequence:

$$\begin{array}{rl}
S' \mid & =\!\!\dashv^{+} \\
\vdash xcAb\cdots bc\underline{Ab\cdots ba} \mid \dashv & =\!\!\dashv^{+} \\
\vdash xc\underline{Ab\cdots bcb\cdots ba} \mid b\cdots ba\dashv & =\!\!\dashv^{+} \\
\vdash xcb\cdots bc\underline{Ab\cdots ba} \mid b\cdots bcb\cdots bab\cdots ba\dashv & =\!\!\dashv^{+} \\
\vdash \underline{xcb\cdots bcb\cdots ba} \mid b\cdots bab\cdots bcb\cdots bab\cdots ba\dashv &
\end{array}$$

There is no bound to the number of a's in a sentence. Each a will be at the top of the stack once and, in each case $f = 1, \ldots, F$, an unbounded stack suffix φ_f (as the above underlined stack suffixes) scan is performed, by looping in states 4 and 5 until finding some x, y, A, or B. But note that these suffixes do not overlap —there is no common stack node that is visited more than once in those unbounded stack scans— and thus $\sum_{f=1}^{F} d_{j_f} = O(l)$, and then the total sum of exploration depths is linear:

$$n = \sum_{f=1}^{F} d_{j_f} + \sum_{i \neq j_1,\ldots,j_F}^{s+r} d_i = O(l) + O(l) = O(l).$$

$$\begin{array}{rl}
1: & S'\to\vdash S\dashv \\
2: & S\to a1A1 \\
3: & S\to a2A2 \\
4: & S\to a3A3 \\
5: & S\to a4A4 \\
6: & B0\to\varepsilon \\
7: & A1\to aA1 \\
8: & A1\to B1 \\
9: & B1\to B0 \\
10: & A2\to aA2 \\
11: & A2\to B2 \\
12: & B2\to B1 \\
13: & A3\to aA3 \\
14: & A3\to B3 \\
15: & B3\to B2 \\
16: & A4\to aA4 \\
17: & A4\to B4 \\
18: & B4\to B3
\end{array}$$

Figure 6.5: $\mathcal{G}_{sue}$: grammar for unbounded explorations, up to 3 per sentence.

6.2 Error detection

These considerations bring us to analyze how the method works in the presence of syntactically erroneous inputs. Of course, since reductions are always verified, it is impossible for the parser to produce a legal parse for an erroneous input, and thus it will eventually signal an error. As in precedence and bounded-context parsers of which this one might be seen as a powerful extension, decisions are usually taken from a small suffix of the parsing stack, plus the lookahead. Thus, it is possible for the parser to perform shift and parsing actions past the state where valid-prefix parsers, as conventional LR for instance, would signal an error. We consider that, in principle, this is not a defect in itself, which can only be perceived as such in relation to the desired error recovery mechanism. Actually, the ability to continue with parsing even in the presence of errors is a flexibility that conventional LR parsers for instance, do not have. Moreover, the well-known and reputed error recovery technique of Graham and Rhodes for weak precedence parsers should be easily adaptable to our parser, given the similarities between the two methods; in particular, reducing past the "point of error" can be interpreted as an "automatic" forward move. In our group, R. O. Cayot is currently working on this subject in his doctoral research.

	$\vdash$	$\dashv$	$a1$	$a2$	$a3$	$a4$	a	$\vdash$	$\dashv$	S	$a1$	$A1$	$a2$	$A2$	$a3$	$A3$	$a4$	$A4$	$B0$	a	$B1$	$B2$	$B3$	$B4$
s1	0		0	0	0	0	0		1	0	6	s2	6	s3	6	s4	6	s5	9	6	s6	s7	s8	17
s2											2									7				
s3													3							10				
s4															4					13				
s5																	5			16				
s6											8		12		12		12			s6				
s7													11		15		15			s7				
s8															14		18			s8				

Figure 6.6: Discriminating reverse LR(1) table for grammar $\mathcal{G}_{sue}$.

1: $S' \rightarrow \vdash S \dashv$
2: $S \rightarrow xA$
3: $S \rightarrow yB$
4: $A \rightarrow a$
5: $A \rightarrow bA$
6: $A \rightarrow cAA$
7: $B \rightarrow a$
8: $B \rightarrow bB$
9: $B \rightarrow cBB$

Figure 6.7: $\mathcal{G}_{uue}$: grammar for unbounded number of unbounded explorations.

	$\vdash$	$\dashv$	x	y	a	b	c	$\vdash$	$\dashv$	S	x	A	y	B	a	b	c
s1	0		0	0					1	0	0	s2	0	s3	s4	0	0
s2											2	6				5	0
s3													3	9		8	0
s4											4	4	7	7		s4	s5
s5											4	4	7	7		s5	s5

Figure 6.8: Discriminating reverse LR(1) table for grammar $\mathcal{G}_{uue}$.

6.3 Implementation

In order to practically test the validity of the stated theoretical predictions for the method, an experimental system has been written by the author. The system allows to generate parsing tables from grammars in BNF format according to the construction algorithm [FG91] of the discriminating reverse LR(1) automaton. In order to allow comparisons with classical LR(1) automata, the capability of generating (non-compressed) parsing tables by the classical algorithm [AU73a, SSS90] has been included in the system. Of course, table-driven parsers for both kinds of automata have also been included, incorporating the computation of several statistical measures. Finally, an original sentence generator also included allows to use easily any new grammar for parsing sentences of any desired length; this is a convenient way to test the method parsing efficiency for different cases.

Clearly, comparison with the classical LR(1) parsing method was a must. But would-be available LR(1) systems use some compression technique. This led the author to write its own classical LR(1) generator in order to make comparisons on the same conditions: parsing power, compression level, implementation language, etc. Thus no attempt has been made to apply known compression methods for classical LR(1) tables [SSS90] or compressing modified versions, e.g., [Pag77c, Pag77b], of the basic generation algorithm. In the former case, table size, parsing efficiency, generation time, etc. would not have been comparable if compression were not also applied to the reverse discriminating method. In both cases, comparative results for the variants can be inferred from the results given here. Anyway, reverse discriminating tables are clearly compressible, although no standard procedure has been yet implemented.

On the practical side, comparison with yacc was also mandatory, since it is the most widely used parser generator, although it is not LR(1) nor LALR(1) [DeR69]. Since yacc accepts some ambiguity in grammars, comparisons are difficult, but at the same time most available grammars are written for yacc. The solution adopted has been to modify yacc grammars as little as possible to eliminate ambiguity and render them LALR(1) for still being acceptable by both systems. This normally involves a slight increase in grammar size. It must also be noted that this comparison is not balanced, since the parsing powers are different, LR(1) being more complex and general than LALR(1). Again, adapting the reverse discriminating method to accept ambiguous grammar ought to allow comparisons to be made on the same conditions. Finally, yacc also compresses the tables it generates, and thus the above remarks for compressed classical LR parsers is also applicable, and since it is also highly optimized, no time comparison is significant with a non-optimized experimental system. Our generator was also extended to produce LALR(1) parsing tables (without compression) and other variants of LR parsers.

In what follows, a more detailed description of the implementation itself is presented. The reader willing to know only about the results and their analysis may skip the rest of this section.

All the possibilities (parser generation and parsing for both the reverse and direct methods, and sentence generation)[3] have been included into a single program with several functioning modes, for ease of use, transport and code sharing and development. The sharing and reusing of code for both the direct and the reverse LR(1) parser generators is specially remarkable, thanks to the similarity of situations, automata, and closure and transition functions.

The program has been written in ISO 7185 Standard Pascal (about 6,000 lines of code), to be portable and compatible with the Cigale compiler generator system [FL92], although its internal structure is modular and easily rewritable to more modern modular languages like Modula-2, and at the same time practically uses only top-level procedures and functions, to be easily rewritable into C. Actually, it has been directly compiled and run on DEC's MIPS machines running Ultrix and DEC's Pascal compiler, on Sun SPARC machines running Solaris and the GNU Pascal compiler, and on PCs running MS-DOS and the Turbo Pascal compiler.

[3] See Appendix B for a brief user manual describing its operation and options.

6.3.1 The parsing table generators

The system parsing table generation part can produce both direct (classical) and reverse discriminating LR(1) parsing tables, from an input file containing the grammar in BNF format. Two files are generated, one containing vocabulary information, and the other containing the parsing table itself, both in internal formats. In the reverse case, full rightpart checking generation can be disabled. Some listings can be output, including the grammar, vocabulary, *First* and *Follow* sets, and automaton states including situations and transition/action tables. Finally, some statistics about the grammar at hand and the generated automata can be obtained, showing the automaton size in terms of number of states, and number of table entries (i.e., the sum of the transitions and action entries for all states). In the case of non-LR(1) grammars, the conflicting states are signalled.

Internally, most auxiliary code is shared for both kinds of automata: user interface, symbol and table management, and grammar processing and checking. Also, a great part of code for situations, states, automata and parsing tables has been shared or reused. Only the kernels of the methods—the closure and the transition/action functions—diverge, with the discriminating reverse method being significantly more complex.

6.3.2 The table-driven parsers

The parser part uses the vocabulary and parsing table files previously generated by one of the two parsing methods, and parses an input sentence text file. Errors are detected and signalled, although no recovery mechanism has been yet implemented. Stack evolution can be output if desired. For the discriminating reverse method, statistical results include the number of shift and reduce actions, the average length of reductions, the maximum and average stack length, the average depth of stack exploration per shift and per reduce actions, and the average number of state transitions per action.

Internally, no space optimization has been attempted on tables, in order to be neutral for parsing time comparisons. All auxiliary code is shared, but the parsing loops for both methods are separated.

6.3.3 The sentence generator

A sentence generator has been included in the system. Nonetheless, its use is not mandatory, since a sentence file can be written using the terminal symbols themselves as they appear in the BNF grammar, or a lexical analyzer for the grammar at hand can be built with the usual techniques or tools.

There are several benefits of using an automatic sentence generator. The user is freed from the burden of writing several sentences for a new grammar, for a slightly adapted grammar, or for a grammar for which it is difficult or impossible to obtain a significant set of sentence samples. If the desired sentence length is extremely big, its generation by hand is absolutely impractical. If the experimenter itself writes the sentences, they might be unintendedly biased in favor of the experimenter's method, or might systematically avoid some constructs that a sentence generator with random capabilities would not. Extreme cases of very involved "pathological" grammars can be easily tested. Finally, not having an automatic sentence generator discourages the application of tests with medium to big sentences, or with new grammars.

Curiously, sentence generators have received little attention in the literature [com91, Mau92]. (Bad) sentence generators can be trivially written, but good sentence generators are difficult, since it is not clear how to generate human-like sentences, i.e., sentences with the form and statistical properties of typical human-written sentences for the (programming) language at hand.

The purely random approach to sentence generation (i.e., equal probability for the nonterminal's different rightparts) showed extremely poor for programming language grammars. Actually, randomly generated sentences use to be just a very small construct with a few huge expressions occupying almost all the sentence.

In essence, the implemented method for sentence generation tries to generate a phrase of a given length by subdividing the task amongst the rightpart nonterminals. Nonterminals are assigned different heuristic generative capabilities related mainly with their level within the grammar hierarchy. Rightpart probabilities and length share are assigned according to these capabilities and the desired length. Generation is performed in a wide-first fashion. All the steps are relatively randomized to avoid systematic bias. The method gives specially good results for grammars with a strong relation between semantics and syntax, and with natural and clear hierarchy of constructs.

The current implementation allows the user to ask for a given or random length of the sentence, to disable randomization with the possibility of giving a seed number, to choose purely random generation, and to ask a number of sentences to be generated. Sentential forms during derivation can be displayed, allowing to check the correct use of the different possibilities of the grammar. See Appendix B for further details.

6.4 Experimental results

Since the main objective was to test the method efficiency for practical use, very artificial grammars, with only theoretical interest to test worst cases, have received only marginal attention, because they typically have no impact on general practice. Accordingly, the presentation of results will be based on three typical case grammars:

A Small expression grammar, very frequently found as part of programming language grammars.

B Medium size grammar, representing application interface grammars or teaching or experimental programming language grammars.

C Large grammar, representing normal programming language grammars.

According to the objective, the tests[4] have been applied on the generation and parsing phases for the above three grammars. Results are compared, specially for size and time, with the classical LR(1) method and, when applicable, with `yacc`. The same comparisons are made for pathological worst-case grammars.

Finally, every attempt has been made to experimentally contrast the predicted LR(1) parsing power, with verification for all cases of the capability of generation for LR(1) grammars, and detection of conflicting state(s) for non-LR(1) grammars.

The following table shows some useful statistics about the three test grammars.

grammar	R	l	t	T	N	$\|G\|$
A	6	2.00	0.83	5	3	12
B	93	2.24	1.12	41	38	208
C	234	2.50	1.24	82	68	585

The meaning of the columns is as follows:

R: number of rules.

l: average length or rightparts.

t: average number of terminal symbols per rightpart.

T: number of terminal symbols in the vocabulary.

N: number of nonterminal symbols in the vocabulary.

$|G|$: grammar size defined[5] as $|G| = R(1 + l)$.

Most of the data shown allows to compute some measure of the grammar size. It is remarkable that practical BNF grammars are found to typically have a small average length of rightparts. The average number of terminal and nonterminal symbols per rightpart allows to make a rough prediction about the proportion in the number of shift and reduce actions during parsing, which is interesting for the discriminating reverse automaton.

[4] These tests were performed on a DECstation 5000/133 running Ultrix 4.2A, and using the DEC Pascal for RISC 1.2 compiler with the optimizing option.

[5] As it is usual [SSS90].

The following table shows the results from running, for the three case grammars, the parser generator for the direct and reverse LR(1) parsers, as well as the results from running `yacc`.

	reverse			direct			yacc	
grammar	S	E	u_g	S	E	u_g	S	E
A	8(3)	23(15)	0.02s	22	71	0.04s	12	26(45)
B	118(72)	781(770)	8.30s	773	6,874	4m40s	173	520(1,324)
C	270(182)	3,145(3,346)	4m57s	1,000+	15,000+	2h+	439	3,155(9,184)

The meaning of the columns is as follows:

S: number of states in the corresponding automaton. Figures between parenthesis for the reverse automaton correspond to the construction without rightpart checking.

E: number of parsing table entries, i.e., the number of entries for state transitions plus the number of entries for parsing actions. Figures between parenthesis correspond, for the reverse automaton, to the construction without rightpart checking, and, in the case of `yacc`, to the corresponding number of LALR(1) table entries without compression (generated by our tool).

u_g: "user" time[6] for program execution, from the `time` Unix command. According to the experimental, non-tuned character of both LR(1) implementations, absolute values are not significant but only their relative values. For the same reason, `yacc` results are not shown, since its values are not comparable.

It is important to note that

The `yacc` number of states and entries shown are as given by its `-v` option. The number of states S is the "states" reported by `yacc`, which is the same as produced by our tool for LALR(1), i.e., `yacc` performed no compression in the number of states. The figures for parsing table entries E have been computed by adding the "shift entries" and "goto entries" reported by `yacc`, and including the "accept" and "reduce" actions in the state list. "Error" entries are not included. Thus, there is a level of compression, since the "entries saved by goto default" are not included into E, and the implicit important compression of the "reduce" entries. It is important to consider this when comparing with other methods, whose tables were not compressed, as explained above.

Note also that direct LR(1) generation results for grammar `C` are not precise since the program aborted because of excessive memory needs.

[6]Times smaller than 1 minute are averaged from 5 runs, each run performing a loop of 5 times the whole program work.

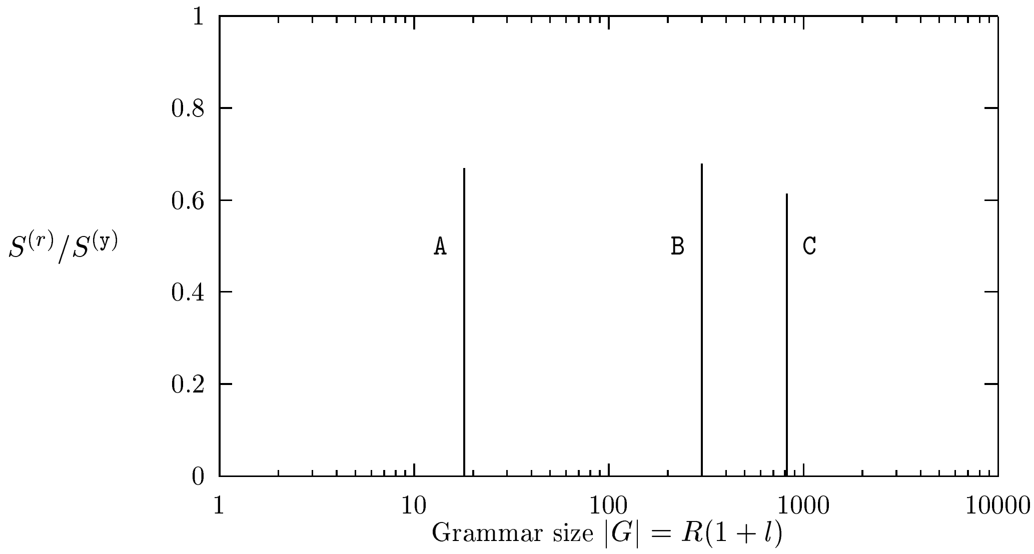


Figure 6.9: Reverse-to-yacc state ratio.

Figure 6.9 shows the ratio of the sizes, in terms of numbers of states, of the discriminating reverse automaton and yacc for the three example grammars. As we can see, the reverse automaton is about 1/3 smaller. Note that in the table only the full rightpart verification version of our automaton is considered. However, the corresponding non-verification figures are significantly smaller, as we have seen in the previous table.

Finally, the table of Figure 6.10 shows the results from running the parsers using the generated parsing tables with the automatically generated sentences as input, for different sizes ranging from small to large, to establish the tendency in the growth rate (if any) of the shown parameters. Special care has been taken in obtaining significant measures for the reverse method. Obviously, since the parsing actions that all the used parsers perform are precisely the same, the corresponding results (A_s, A_r, l_r, $\hat{s}$, and s, in the table) are exactly the same. In conclusion, only the parsing times can differ, and the parsing time for yacc is not shown for the same reason as above. Note also that parsing results in a same row are obtained for the same single sentence, with the exception of time results.

The meanings of the columns in the table of Figure 6.10 are as follows:

A_s: number of shift actions, i.e., the length of the sentence to parse.

A_r: number of reduce actions.

l_r: average length of reductions. Compare with l, the average length of rightparts.

grammar	A_s	A_r	l_r	$\hat{s}$	s	reverse					direct	
						σ_s	σ_r	τ	u_p	u_{p_0}	u_p	u_{p_0}
`A`	10	17	1.59	9	5.32	0.45	2.18	0.93	0.01s	0.11ms	0.02s	0.08ms
	100	136	1.74	9	5.50	0.50	2.10	0.85	0.02s	0.90ms	0.03s	0.67ms
	1,000	1,398	1.72	26	10.01	0.50	2.14	0.87	0.16s	9.05ms	0.15s	6.36ms
	10,000	13,565	1.74	41	14.40	0.50	2.16	0.88	1.58s	89.0ms	1.41s	62.1ms
`B`	10	9	2.00	8	5.89	1.10	2.56	1.26	0.08s	0.08ms	0.54s	0.06ms
	100	121	1.82	19	11.15	1.00	2.44	1.04	0.11s	1.31ms	0.59s	1.03ms
	1,000	1,049	1.95	24	12.01	1.08	2.57	1.08	0.44s	12.6ms	0.88s	11.3ms
	10,000	11,657	1.86	128	47.96	1.05	2.49	1.00	3.57s	0.11s	3.90s	86.9ms
`C`	10	22	1.41	5	3.56	1.60	2.00	1.06	0.19s	0.18ms	n/a	n/a
	100	387	1.26	15	8.82	1.09	1.96	0.90	0.28s	2.44ms	n/a	n/a
	1,000	4,184	1.24	46	19.47	1.21	1.95	0.90	1.18s	28.0ms	n/a	n/a
	10,000	40,761	1.25	97	41.22	1.21	1.95	0.90	9.75s	0.26s	n/a	n/a

Figure 6.10: Experimental parsing results (see explanation of columns in the text).

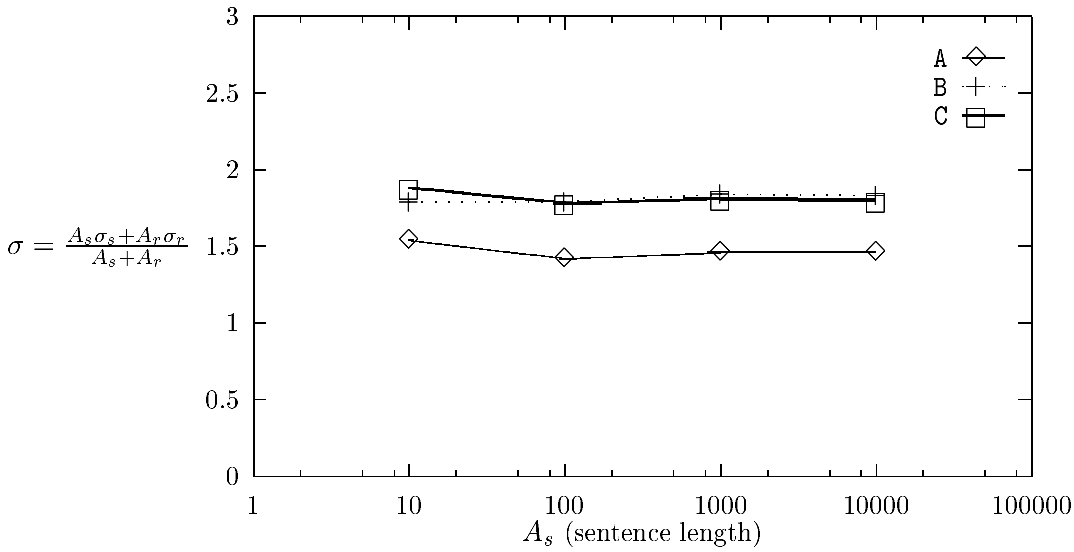


Figure 6.11: Average depth of stack exploration per parsing action.

$\hat{s}$: maximum stack length during parsing.

s: average stack length.

σ_s: average depth of stack exploration per shift action. Its value can be less than 1, since some actions can be decided at the initial state, without looking at the stack.

σ_r: average depth of stack exploration per reduce action (full rightpart exploration is included). Compare with l_r.

τ: average number of state transitions per action.

u_p: "user" time (in seconds) for parsing program execution, averaged from the `time` command. The same considerations as for u_g apply.

u_{p0}: "user" time (in milliseconds) spent strictly for parsing, averaged[7] from the Pascal `clock` function.

Figure 6.11 shows the average depth of stack exploration per parsing action, σ, for automatically generated sentences of different lengths —it can

[7] Separately obtained as the average from 5 different sentences of the same length, each sentence used for 5 runs. Within the program, a strict parsing time is obtained by repeatedly parsing the whole sentence a number of times enough to consume at least 1 second, from which the lexical analyzer time (averaged by using the same repetition procedure) is then subtracted.

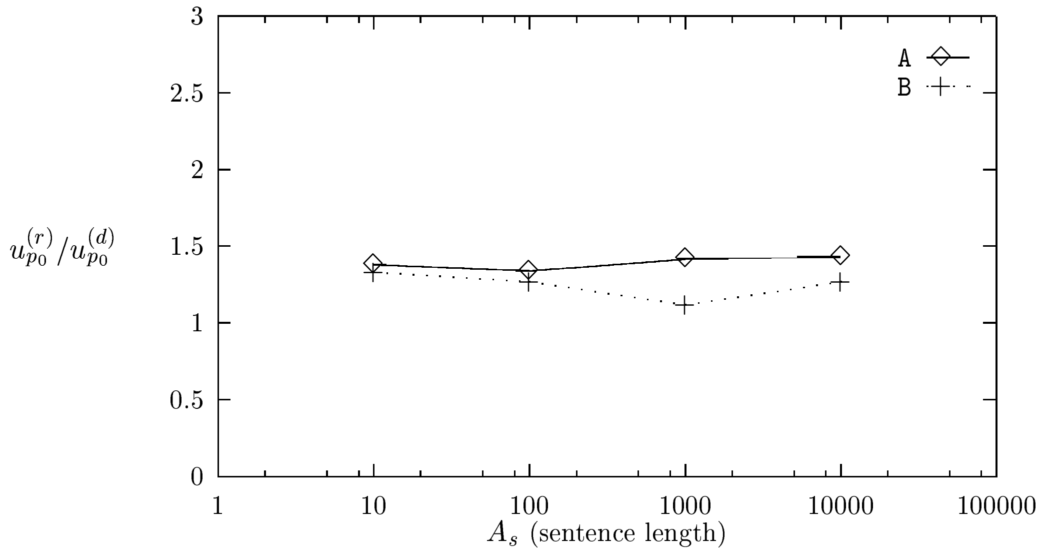


Figure 6.12: Reverse-to-direct parsing time ratio.

be computed as shown in the Figure. As we can see, this average depth is very stable between 1.5 and 2 stack-top symbols.

Finally, Figure 6.12 shows the relative speeds of our reverse and direct LR(1) parsers for different lengths of sentences. Again, it is very stable around 1.3.

The three test grammars are obviously LALR(1) in order to be acceptable by the three kinds of parser used: direct and reverse LR(1), and yacc. Strictly LR(1) grammars, i.e., non-LALR(1) unambiguous grammars, are difficult to find, given the long tradition of yacc use. Small grammars of this kind have been designed to test the method, as well as minor variations of yacc grammars. In all the cases the resulting automata have shown practically the same differences as for the LALR(1) grammars.

Pathological LR(1) grammars have been tested as well, only for theoretical interest, since they are never found in practice. Some were found in the literature as designed to be difficult for classical LR(1), provoking an explosion in the number of states. Normally, the reverse automaton performance were not damaged, in contrast with the classical automaton. On the other hand, a few involved pathological grammars with very long rightparts, have been written, provoking an important increase in stack exploration overhead. This phenomenon is not considered relevant, since, as we have seen, no method is free from pathological worst-case grammars with only theoretical interest. We consider the results in the usual range of grammars and programs, much more relevant for assessing the interest of the method.

The above results on relative automata sizes have been independently confirmed[8] by J. Farré (University of Nice-Sophia Antipolis), which has written a performant implementation of the generator in C++. His tests also show that the average number of state transitions per action is around 1, what is coincident with our result on the average depth of stack exploration.

6.4.1 Evaluation of results

Let us analyze the above results, and try to explain them and derive some conclusions about the usefulness of the new method.

Results from parsing table generation show an enormous improvement in size in comparison with the classical method, both in the number of states and in the number of table entries. The method is even significantly better than yacc for LALR(1) grammars, in spite of the difference in parsing power between the two methods. The main reason to explain the size improvement lies in the discriminating character of the new method, which allows to build the automaton as small as possible, instead of building, as in classical LR(1) or yacc, a full stack-recognizing automaton. Also, the fact that exploration begins where the relevant information for parsing is more significant, and that the lookahead check is deferred until useful, are additional reasons to the size improvement. Memory is not so important nowadays, although it is clear that the reason for not using LR(1) instead of LALR(1) in yacc is precisely the high demand of memory and, consequently, unacceptable generation time. Anyway, a small table can fit in cache memory, what may be an important improvement in parsing performance. Moreover, a small table might be more practical to implement directly into programming code in order to speed up parsing without much impact on memory requirements; which always is a limited resource, specially with applications demanding more and more memory.

The times taken to generate the parsing tables are naturally better, given the big difference in size of the tables to generate, in spite of the fact that the reverse discriminating algorithm is more elaborated. Generation time depends obviously on how well tuned is the construction procedure, the internal tables organization, etc. In a production implementation the effect of table size might be lessened, which would favor classical methods time for generation. Anyway, it may be expected that the overhead for bigger tables will not compensate for the more complex algorithm in practice, and thus the reverse method would still be better. In particular, yacc lookahead generation algorithms, or direct LR compression algorithms in general, are complex enough to increase the final generation time.

Parsing time takes a small part of the processing time of a compiler. Compare for instance, in the table of parsing results, the time used strictly for

[8] See Appendix C.

parsing u_{p_0} with the time of the program u_p which includes lexical analysis, parsing table loading, etc. Anyway, production compilers have to optimize all parts, and then every improvement in time is important. Our comparison for parsing is centered about how important the stack exploration overhead is for the reverse automaton.

A first-sight critique to the method is that it needs to fully explore the rightparts to reduce. However, in practice grammars have a small average length of rightparts l, and thus the impact is small, as the average length of reductions l_r in the test results show. Another critique is that full stack exploration for worst cases is too bad. Experience shows that stacks are very small in comparison with the lengths of sentences, and, which is more important, full exploration of deep stack is practically never performed, except for grammars written ad-hoc. Further analysis of test results shows that all these concerns are of very little practical importance. All the measures of stack exploration (σ_s, σ_r, and τ) are virtually independent of the sentence or stack lengths. It is very remarkable that stack exploration per shift action σ_s is very small, and that the average depth of exploration when reducing σ_r has a very small overhead in comparison with the average length of reductions l_r. Finally, the most important result is that, in practice, the average number of state transitions per action τ is very small, which is directly comparable with direct parsing, with exactly 1 transition per action. And thus, we can conclude that the exploration overhead for the discriminating reverse automaton is negligible in practice.

The registered parsing times reflect the small difference between the reverse and direct approaches, in spite that the reverse method performs a double check (lookahead, then next stack symbol) in each (non-final) state. Results for direct methods are easier to understand, given the simple mechanism to decide the parsing action, although in practice their tables are compressed, which introduces an important overhead, comparable or even worse than stack exploration if the reverse tables are not also compressed. Strictly parsing time results (u_{p_0}) show only a 25% increase in time with respect to the direct method for grammar `B`. This is even smaller than what could be expected from stack exploration measures. A possible explanation for this unexpected effect may be that the reverse parsing table is most frequently used at the initial state and its neighbor states, and thus locality of access improves running times in a machine with cache memory, in comparison with a larger, direct table which is much more randomly accessed. Note finally that in the experimental results loading a bigger parsing table was much more time-consuming than the strict parsing, which explains that the whole parsing time u_p is sometimes significantly better for the reverse method.

So good results for parsing were not expected. Now, they can be analyzed in the light of the other aspects of the method. Actually, as precedence parsers designers foresaw, most parsing actions can be decided with very few symbols around the top of the stack. Our results confirm that old intuition. The automaton structure reflects this fact, since shifts are discriminated after very few transitions (some even at the initial state), and the portion of states not in the reverse rightpart recognizer is only about one third. At the same time, the overhead in the number of states that complete exploration of rightparts when their reduction has been detected in advance is only about one third. In conclusion, the automaton automatically adapts itself to the intricacies of the grammar, which produces in practice very good results in size and in average exploration overhead.

Chapter 7

Discriminating Reverse LR(k)

In this chapter, discriminating reverse parsing is extended for the general case of LR(k) grammars, $k \geq 0$. Reverse items are extended to a general form of reverse LR(k) items where a general form of dotted lookahead is used. As in the particular LR(0) and LR(1) cases already studied, a discriminating reverse automaton is constructed with states associated to sets of reverse LR(k) items.

At the end of the chapter, correctness and linearity proofs are given for the general discriminating reverse LR(k) parser.

7.1 Dotted k-lookahead

A dotted string of at most k symbols is simply a string consisting of k or less symbols with a dot inserted somewhere in the string, i.e., it may be between two substrings, or just at the beginning or ending of the sequence. For instance, $\bullet abc$, $ab\bullet cde$, and $abcd\bullet$ are dotted strings of at most 5 symbols.

For our LR(k) reverse items we shall use dotted k-lookaheads, i.e., lookaheads consisting of dotted strings with at most k symbols.

To obtain our dotted k-lookaheads, a head operation is usually applied on another dotted string, which is usually longer than k, but which may also be equal or smaller than k. Figure 7.1 shows the result of the head operator on a dotted string $x\bullet y$, i.e., $k : x\bullet y = x'\bullet y'$. As the figure shows, $k : x\bullet y$, for $|xy| \geq k$, consists of, either the first k symbols of xy with the dot in the same place if $k \geq |x|$, or else, of the first k symbols of x followed by a dot. Otherwise, if $k > |xy|$ —not shown in the figure— then $k : x\bullet y = x\bullet y$. In conclusion, it works as the usual head operator for strings, without counting the dot as a symbol, which remains in the same position, with the only exception of $k : x\bullet y = x'\bullet$, $x' = k : x$, when $k \leq |x|$.

For instance, $3 : ab\bullet = ab\bullet$, $3 : abcd\bullet ef = abc\bullet$, $3 : abc\bullet de = abc\bullet$, and $3 : ab\bullet cd = ab\bullet c$.

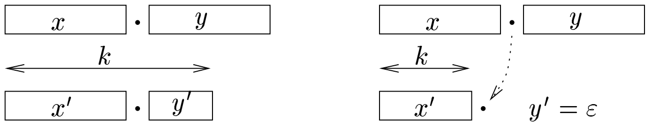


Figure 7.1: Head $x'\!\bullet\! y'$ of length k of a dotted sequence $x\!\bullet\! y$.

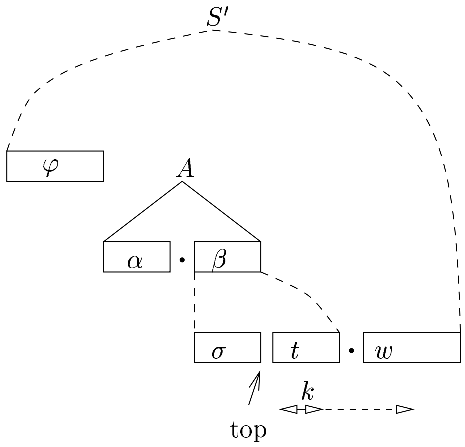


Figure 7.2: Lookahead dot in LR(k) reverse item.

Our general discriminating reverse LR(k) items will simply have the form

$$[i,\ A{\rightarrow}\alpha\!\bullet\!\beta,\ u\!\bullet\! v].$$

The only difference with previous forms lies in the lookahead, which is now a dotted string of (at most) k symbols, i.e., $|uv| \leq k$. The meaning of such items is informally represented in Figure 7.2, where σ represents the stack suffix that has already been explored from the indicated stack top, and thus the item rightpart dot indicates (of course, amongst other possibilities indicated by other items in the current state) the current exploration point within the rightpart of rule $A{\rightarrow}\alpha\beta$. As the figure shows, $\beta \underset{\mathrm{rm}}{\Rightarrow}^* \sigma t$, and then the item lookahead part is $k : t\!\bullet\! w$. Depending on the values of k and on the position of the lookahead dot in $t\!\bullet\! w$ in the figure, $k : t\!\bullet\! w$ may be $t'\bullet$ for a prefix t' of length k of t, $t\!\bullet\! w'$ for a prefix w' of w if $|tw'| = k$, or precisely $t\!\bullet\! w$ if $k \geq |tw|$. In summary the lookahead dot separates the lookahead substring u covered by the item rightpart β to the right of the dot from the rest of the lookahead still not covered.

If we consider that the sentence is followed by k end-of-file markers $\dashv$, then our item lookahead parts will always have $|uv| = k$. This may simplify programming when implementing the method, so we considered this for the algorithm. However, it is unnecessary in theory, because our augmentation scheme $S'{\rightarrow}{\vdash} S{\dashv}$ is enough to have the end of file information, and adding k more such markers adds no more information.

The reader might want to verify that this really is a generalization of the LR(1) reverse item, whose meaning was represented in Figure 5.2 (page 74). If we consider $k = 1$ for lookahead $u \bullet v$, then $|uv| = 1$, and thus one of u or v is empty, and the other is some terminal a, thus either the item lookahead part is $\bullet a$ or $a \bullet$. In the former case $t = \varepsilon$ in Figure 7.2, which corresponds to upper tree of Figure 5.2, and in the latter case $t = as$, which corresponds to lower tree of that same figure.

7.2 Closure and initial state

Once we understand how the reverse item is generalized for the LR(k) case, it is relatively straightforward to generalize the closure operation, and to calculate the initial state for the general discriminating reverse LR(k) automaton.

Figure 7.3 schematically represents a tree during the closure operation. Again, the closure produces new items only when the rightpart dot has reached the left end of the item rule rightpart, in order to compute new items to continue building the automaton by means of decisions or transitions on next stack symbols. This corresponds in the figure to item

$$[i,\ A \to \bullet \alpha,\ u \bullet v],$$

such that $u \bullet v = k : t \bullet w_1 w_2$. Now we can see the need of this form of k-lookahead, since when computing the closure for such an item, only those rules of the form $D \to \mu\tau$ such that the derivation $\tau \underset{\text{rm}}{\Rightarrow}^{+} A w_1$ fits the item lookahead, i.e., such that, either v is a prefix of w_1, or $v = w_1 v'$ and $S' \underset{\text{rm}}{\Rightarrow}^{*} \varphi D w_2$ and v' is a prefix of w_2. In summary, only those rules producing that string v to the right of the lookahead dot can be used in the closure —of course, if $v = \varepsilon$, i.e., u is a prefix of t, then all the rules such that $\tau \underset{\text{rm}}{\Rightarrow}^{+} A w_1$, for whatever w_1, have to be considered. The use of other productions which do not hold to the above condition would evidently mean to consider illegal tree and stack continuations, and thus we would be producing a reverse automaton inconsistent with the grammar. Moreover, in all the above cases, the k-lookahead of the item produced by the closure has to hold to our above item definition, according to the movement of the lookahead dot in the figure tree, that is, the new item will have the form

$$[i,\ D \to \mu \bullet \tau,\ u' \bullet v'],$$

where $u' \bullet v' = k : t w_1 \bullet w_2$, according on the above values of w_1 and w_2.

Consequently, the closure of a set q of LR(k) reverse items can be defined as follows.

$$\begin{aligned}\mathcal{C}_k(q) = q \quad \cup \quad \{&[i,\ D \to \mu \bullet \tau,\ uy \bullet v'] \mid [i,\ A \to \bullet \alpha,\ u \bullet y v'] \in q,\ \tau \underset{\text{rm}}{\Rightarrow}^{*} A w_1,\\ &D \to \mu\tau \in P',\ S' \underset{\text{rm}}{\Rightarrow}^{*} \varphi D w_2,\ \text{and } uy \bullet v' = k : u w_1 \bullet w_2\}.\end{aligned}$$

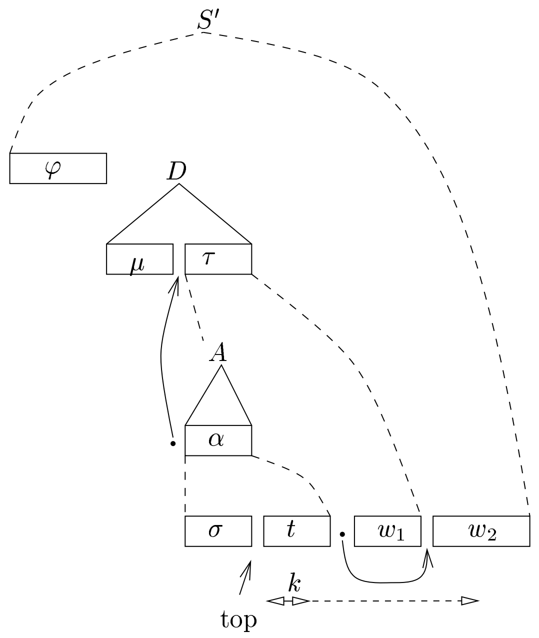


Figure 7.3: Rightpart and lookahead dots in LR(k) reverse item closure.

The reader might want to verify that this k-closure $\mathcal{C}_k$ is a generalization of the reverse LR(1) closure $\mathcal{C}$. Compare Figure 5.5 (page 76) with the above figure. If we consider $k = 1$ for the above items, we can see that

- The upper tree of Figure 5.5 corresponds to $t = w_1 = \varepsilon$, $w_2 = ay$, $u = u' = \varepsilon$, and $v = v' = a$.
- The middle tree of Figure 5.5 corresponds to $t = \varepsilon$, $w_1 = as$, $w_2 = w$, $u = \varepsilon$, $v = a = u'$, and $v' = \varepsilon$.
- The lower tree of Figure 5.5 corresponds to $t = as$, $u = a = u'$, and $v = \varepsilon = v'$.

For a reduce action i, $i > 0$, we simply have to consider for the item lookahead all the possible strings of length k (or less) that can legally follow the reducing nonterminal. According to our item definition, the lookahead dot must be to the left of the lookahead string, for it is obviously non-covered by the (here empty) rightpart section to the right of the rightpart dot. See Figure 7.4 for action corresponding to reducing $A \xrightarrow{i} \alpha$. It is evidently a generalization of the LR(1) case when $k = 1$ and $w = ax$.

Now consider the computation of initial state items for the shift action $i = 0$. Again, we have to consider, before closure, all the possible configurations where the current top of the stack is just before a terminal, i.e., all the rule rightparts of the grammar, and for the rightpart dot all the possible positions

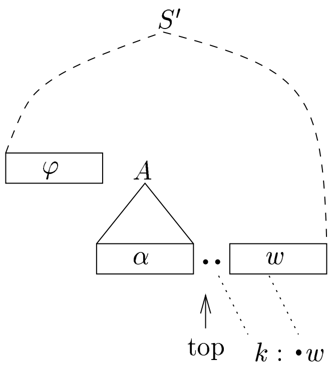


Figure 7.4: Typical parsing tree for reduce action.

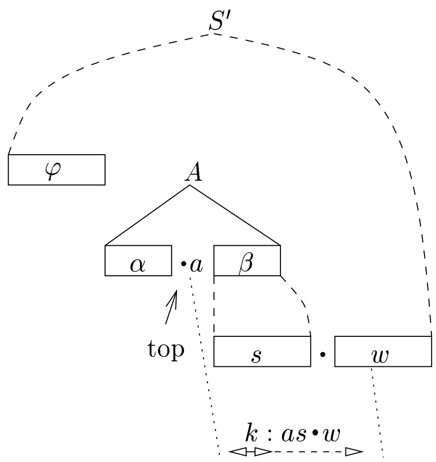


Figure 7.5: Typical parsing tree for shift action.

before a rightpart terminal symbol. For a generic rule $A{\rightarrow}\alpha a\beta$, initial state item $[0,\ A{\rightarrow}\alpha{\bullet}a\beta,\ u{\bullet}v]$ is built. In order to compute its lookahead $u{\bullet}v$, according to the item definition, u is the lookahead substring covered by $a\beta$ (see Figure 7.5), that is, we have to consider all the cases for u and v such that $u{\bullet}v = k : as{\bullet}w$ for $\beta \underset{\text{rm}}{\Rightarrow}^{*} s$ and $S' \underset{\text{rm}}{\Rightarrow}^{*} \varphi A w$.

As we see, this is a little more complicated generalization of the items for initial state in comparison with reverse LR(1), since here we have to consider longer lookahead sequences consisting of the terminal to shift a and all possible substrings beyond it. Another difference is that the lookahead dot may not be at the right end, because part of it might still be non-covered, but of course at least the terminal a to shift is covered. If we make $k = 1$ in $k : as{\bullet}w$ we get the only case $a{\bullet}$ again, as we had seen for the LR(1) case.

In summary, initial state for the reverse discriminating LR(k) automaton

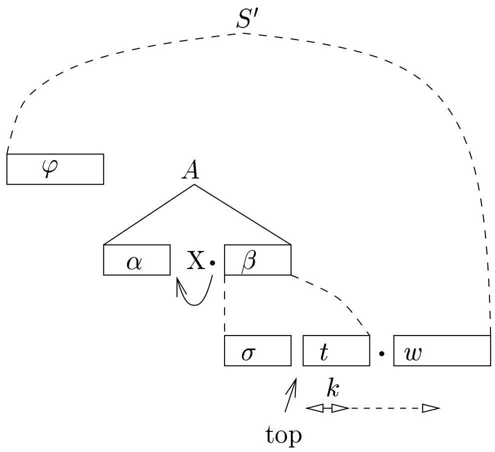


Figure 7.6: Transition on stack symbol X for LR(k) reverse items.

is built as follows.

$$\begin{aligned} q_0 = \ & \mathcal{C}_k(\{[i,\ A{\rightarrow}\alpha\bullet,\ k : \bullet w] \mid A\xrightarrow{i}\alpha \in P' \text{ and } S' \underset{\text{rm}}{\Rightarrow}^* \varphi A w\} \\ & \cup \{[0,\ A{\rightarrow}\alpha\bullet a\beta,\ k : as\bullet w] \mid A{\rightarrow}\alpha a\beta \in P' \text{ and } \beta \underset{\text{rm}}{\Rightarrow}^* s \text{ and } S' \underset{\text{rm}}{\Rightarrow}^* \varphi A w\}) \end{aligned}$$

Finally, a state item set resulting from a transition on stack symbol X is computed as follows by the closure of the items with the rightpart dot moved to the left of X.

$$\delta(q, X) = \mathcal{C}_k(\{[i,\ A{\rightarrow}\alpha\bullet X\beta,\ u\bullet v] \mid [i,\ A{\rightarrow}\alpha X\bullet\beta,\ u\bullet v] \in q\})$$

See Figure 7.6, where $k : t\bullet w = u\bullet v$. As the figure clearly shows, before closure, the position of the lookahead dot is not affected by a transition, as in the LR(1) case.

7.3 An example for $k = 2$

Figure 7.7 shows the discriminating reverse LR(2) automaton with items sets for states, and their corresponding parsing actions and transitions, for grammar $\mathcal{G}_{lrt}$ shown in Figure 5.33 (page 111). This grammar is LR(2) but not LR(1). It was in fact used in Section 5.4.6 to illustrate the detection of non-LR(1) grammars.

7.4 Algorithms

With the procedures given in the above sections, a reverse recognizing automaton can be built which carries in each state items the information of which actions are compatible with the stack suffix read thus far. Moreover, State 1 $\{0,1,2,3,4,5,6\}$

$[1, S' \to \vdash S \dashv \bullet,\ \bullet \dashv\dashv]$	`LA(`$\vdash a$`) = Shift`
$[2, S \to Ac\bullet,\ \bullet \dashv\dashv]$	`LA(`aa`) = Shift`
$[3, S \to aA\bullet,\ \bullet \dashv\dashv]$	`LA(`ab`) = Shift`
$[4, A \to aB\bullet,\ \bullet c\dashv]$	`LA(`bc`) = Shift`
$[4, A \to aB\bullet,\ \bullet \dashv\dashv]$	`LA(`cc`) = Shift`
$[5, A \to abc\bullet,\ \bullet c\dashv]$	`LA(`$b\dashv$`) = Shift`
$[5, A \to abc\bullet,\ \bullet \dashv\dashv]$	`ST(`$\dashv$`) = Reduce 1`
$[6, B \to b\bullet,\ \bullet c\dashv]$	`ST(`B`) = Reduce 4`
$[6, B \to b\bullet,\ \bullet \dashv\dashv]$	`ST(`S`) = Shift`
$[0, S' \to \bullet\vdash S\dashv, \vdash a\bullet]$	`ST(`c`) = Go To 2`
$[0, S' \to \vdash S\bullet\dashv, \dashv\bullet\dashv]$	`ST(`A`) = Go To 3`
$[0, S \to A\bullet c, c\bullet\dashv]$	`ST(`b`) = Go To 4`
$[0, S \to \bullet aA, aa\bullet]$	
$[0, A \to \bullet aB, ab\bullet]$	
$[0, A \to a\bullet bc, bc\bullet]$	
$[0, A \to ab\bullet c, c\bullet c]$	
$[0, A \to ab\bullet c, c\bullet\dashv]$	
$[0, B \to \bullet b, b\bullet c]$	
$[0, B \to \bullet b, b\bullet\dashv]$	
$[0, S' \to \vdash\bullet S\dashv, aa\bullet]$	
$[0, S \to \bullet Ac, ab\bullet]$	
$[0, S \to a\bullet A, ab\bullet]$	
$[0, S \to \bullet Ac, bc\bullet]$	
$[0, S \to a\bullet A, bc\bullet]$	
$[0, A \to a\bullet B, b\bullet\dashv]$	
$[0, S' \to \vdash\bullet S\dashv, ab\bullet]$	
$[0, S' \to \vdash\bullet S\dashv, bc\bullet]$	

State 2 $\{2,5\}$

$[2, S \to A\bullet c,\ \bullet\dashv\dashv]$	`LA(`$c\dashv$`) = Reduce 5`
$[5, A \to ab\bullet c,\ \bullet c\dashv]$	`ST(`A`) = Reduce 2`
$[5, A \to ab\bullet c,\ \bullet\dashv\dashv]$	`ST(`b`) = Reduce 5`

State 3 $\{0,3\}$

$[3, S \to a\bullet A,\ \bullet\dashv\dashv]$	`LA(`$\dashv\dashv$`) = Reduce 3`
$[0, S \to \bullet Ac, c\bullet\dashv]$	`LA(`$c\dashv$`) = Shift`
$[0, S' \to \vdash\bullet S\dashv, c\dashv\bullet]$	

State 4 $\{0,6\}$

$[6, B \to \bullet b,\ \bullet c\dashv]$	`LA(`$\dashv\dashv$`) = Reduce 6`
$[6, B \to \bullet b,\ \bullet\dashv\dashv]$	`ST(`a`) = Go To 5`
$[0, A \to a\bullet bc, c\bullet\dashv]$	
$[6, A \to a\bullet B,\ \bullet c\dashv]$	
$[6, A \to a\bullet B,\ \bullet\dashv\dashv]$	

State 5 $\{0,6\}$

$[0, A \to \bullet abc, c\bullet\dashv]$	`ST(`a`) = Shift`
$[6, A \to \bullet aB,\ \bullet c\dashv]$	`ST(`$\vdash$`) = Reduce 6`
$[0, S \to a\bullet A, c\bullet\dashv]$	
$[6, S \to \bullet Ac, c\bullet\dashv]$	
$[6, S' \to \vdash\bullet S\dashv, c\dashv\bullet]$	

Figure 7.7: Discriminating reverse LR(2) automaton for grammar $\mathcal{G}_{lrt}$, including list of parsing actions and transitions.

in each state, the item lookahead strings uv can be used —without considering the lookahead dot, evidently— to check the current k-lookahead string in order to detect whether the current stack suffix plus lookahead combination is compatible with only one single action.

Again, as in the particular LR(0) and LR(1) cases, if we remove an item from the set, we are restricting the language of remaining stack prefixes that we are to reverse accept to those cases corresponding to the remaining items.

Thus, in the algorithm shown in Figure 7.8, we take a two-step approach completely analogous to the particular cases seen in preceding sections, such that in each state, first the lookahead is used to try to decide next parsing action, and if this fails, next stack symbol is used to decide, perform a transition with the remaining items, or signal an error if at this point it is illegal. Again, each time that a decision is taken on current lookahead o next stack symbol, all the corresponding items with that lookahead o stack symbol, respectively, are removed from the current item set under consideration.

Non-LR(k) grammars are detected with a test analogous to the LR(1) case.

Note that First_k and Follow_k were defined in page 4. One of the differences of this case relies precisely in that, instead of using a single symbol lookahead, as in the LR(1) case, here lookahead sequences of length k are used. This leads to the problem of the exponential combination of such values. In the direct LR(k) case this leads to an huge explosion in the number of states, that we feel should be much smaller for the discriminating reverse LR(k) case, since most of the branches are rapidly pruned. Anyway, the fact of using lookahead sequences of length k rapidly increases the table size in terms of number of columns. This reason alone makes that to apply directly this construction in practice would be usually too much costly, unless some optimization is put in practice.

Again, nullable reductions are inhibited on nullable stack symbols. As it was indicated, this simple mechanism can be replaced for a finer analysis of the possibility of loop of reductions from empty for non-sentences, in order to minimize its impact on automaton size.

7.5 Formal proofs

Definition 7.1 *An LR(k) reverse item has the form* $[i, B \rightarrow \gamma \bullet \varphi, x \bullet y]$*, where* $|P'| \geq i \geq 0$*,* $B \rightarrow \gamma\varphi \in P'$*, and* $|xy| \leq k$*.* $\square$

Lemma 7.2 *If* $v = k : x$*, then* $k : vy = k : xy$*.*

Proof. There are two cases:

- Case $|x| \leq k$. Then $v = x$.

```
discriminating-reverse LR(k) parsing table
    input Reduced grammar G
    output Parsing table ActTrans(state,kind,seq_or_symb),
           or Error if the grammar is not LR(k)
    function Closure(J)
        I := J
        repeat
            for [i, A→•α, x•y] ∈ I do for B→βAγ ∈ P' do
                m := 0
                loop
                    l := |y| − m; v := l : y; w := y : m
                    if v ∈ First_l(γ) and w ∈ Follow_m(B) then exitloop
                    m := m + 1
                    if m > |y| then exitloop
                endloop
                if m ≤ |y| then add [i, B→β•Aγ, xv•w] to I
        until no new additions to I
        return I
    end Closure
    begin
        N_n:={A | A⇒⁺ε}; N_r:={i | A—i→α and α⇒*ε}
        Aut:=empty
        K_0 :={[i, A→α•, •y] | A—i→α ∈ P' and y ∈ Follow_k(A)} ∪
             {[0, A→α•aβ, av•w] | A→αaβ ∈ P', v ∈ First_l(β), w ∈ Follow_m(A) and k = 1 + l + m}
        I_0 :=Closure(K_0); add I_0 to Aut
        for I ∈ Aut do
            % non-LR(k) check
            if [i, A→α•β, x•y], [j, A→α•γ, x'•y'] ∈ I,  i ≠ j and xy = x'y' then Error
            % decisions upon current lookahead
            for z ∈ V_T'^k do La(z) := {i | [i, A→α•β, x•y] ∈ I and z = xy}
            J := I
            for x | La(x) = {i} do
                if i = 0 then ActTrans (I,lookahead,x):=shift
                else ActTrans(I,lookahead,x):=reduce(i)
                J := J − {[i, A→α•β, x] ∈ J}
            % decisions upon next stack symbol
            for X ∈ V' do St(X) := {i | [i, A→αX•β, x•y] ∈ J}
            for X | St(X) = {i} and  not (X ∈ N_n and i ∈ N_r) do
                if i = 0 then ActTrans(I,stack,X):=shift
                else ActTrans (I,stack,X):=reduce(i)
                J := J − {[i, A→αX•β, x•y] ∈ J}
            % state transitions upon next stack symbol
            for X | [i, A→αX•β, x•]y ∈ J do
                K :=Closure({[j, B→γ•Xδ, v•w] | [j, B→γX•δ, v•w] ∈ J})
                add K to Aut; ActTrans(I,stack,X) := goto(K)
    end
```

Figure 7.8: Generation algorithm of discriminating reverse LR(k) parser.

```
algorithm Discriminating-reverse LR(k) parser;
input Parsing table ActTrans(state,kind,seq_ or_ symb)
        and input sequence
output Bottom-up parse of the input sequence,
        or Error if syntactically incorrect
function Next
    read(a)
    x := xa : k
end Next
begin
    Stack:=empty; push(⊢)
    for j := 1 to k do Next
    repeat
        q := q0; p := top
        repeat
            at:=ActTrans(q,lookahead,x)
            if at=nil then at:=ActTrans(q,stack,Stack(p))
            case at of
                goto(q'): q := q'; p := p − 1
                shift: push(1 : x); Next
                reduce(A -i-> α): if i = 1 then at:=accept
                    else pop(α); push(A)
                nil: Error
        until at ≠ goto
    until at=accept
end
```

Figure 7.9: Algorithm of discriminating reverse LR(k) parser.

- Case $|x| > k$. Then $x = x_1 x_2$ with $|x_1| = k$, and then $v = x_1$. And the lemma holds because $k : vy = k : x_1 y = x_1$ and $k : xy = k : x_1 x_2 y = x_1$.

□

The following is an extension of the prefix operator which results in convenient notation for reverse automata.

Definition 7.3 *Prefix of length at most k of a* dotted *sequence:*

$$k : x \bullet y = x' \bullet y' \textit{ iff } x' = k : x \textit{ and } x'y' = k : xy.$$

□

Lemma 7.4 *If $v = k : x$, then $k : vy \bullet z = k : xy \bullet z$.*

Proof. By Lemma 7.2, $v = k : x$ implies $k : vy = k : xy$ and $k : vyz = k : xyz$. □

Definition 7.5 *Given a grammar $\mathcal{G}$, the closure of a set q of LR(k) reverse items is defined by:*

$$\begin{aligned}\mathcal{C}_k(q) = q \quad \cup \quad \{[i,\ B{\to}\beta\bullet\gamma,\ xy\bullet z] \mid [i,\ A{\to}\bullet\alpha,\ x\bullet yz] \in q,\ \gamma \underset{\text{rm}}{\Rightarrow}{}^* Av,\\ B{\to}\beta\gamma \in P',\ S' \underset{\text{rm}}{\Rightarrow}{}^* \varphi Bw,\ \textit{and } xy\bullet z = k : xv\bullet w\}.\end{aligned}$$

□

Alternatively, it can be put $xy \bullet z = k : xv \bullet w'$, with $w' \in \mathrm{Follow}_k(B)$.

Actually, in general, it is not necessary to use rightmost derivations in $S' \underset{\text{rm}}{\Rightarrow}{}^* \varphi Bw$, but that makes proofs simpler.

Note that the phrase of the currently explored rightpart (according to the item under consideration) "cuts" the lookahead string of length k in two pieces: before the dot is the suffix of the phrase, and after the dot the rest of the lookahead string not yet included in the phrase.

Lemma 7.6 *If $k : u \bullet vw = x \bullet t$, then $k : uv \bullet w = xy \bullet z$ with $yz = t$.*

Proof. Let $yz = t$. Then $k : uvw = xt = xyz$. There are the following two cases:

- Case $k \le |uv|$. Then $xt = k : uvw = k : uv$. The lemma holds choosing $y = t$.

- Case $k > |uv|$. Then $x = k : u = u$ and then $k : uv = uv = xv$. And the lemma holds choosing $y = v$.

□

Here a different approach is used by defining a single, all-action stack (suffix) recognizer $\mathcal{A}_k$.

Definition 7.7 *Given a grammar $\mathcal{G}$, the automaton $\mathcal{A}_k$ is constructed as follows:*

1. *Initial state:*

$$\begin{aligned} q_0 \;=\; & \mathcal{C}_k(\{[0,\ A{\to}\alpha{\bullet}a\beta,\ k:av{\bullet}w] \mid A{\to}\alpha a\beta \in P',\ \beta \underset{\text{rm}}{\Rightarrow}^* v,\ \textit{and} \\ & S' \underset{\text{rm}}{\Rightarrow}^* \varphi A w\} \cup \\ & \{[i,\ A{\to}\alpha{\bullet},\ k:{\bullet}w] \mid A \xrightarrow{i} \alpha \in P' \ \textit{and}\ S' \underset{\text{rm}}{\Rightarrow}^* \varphi A w\}) \end{aligned}$$

2. *Transition function:*

$$\delta(q,\sigma) = \begin{cases} \sigma = \varepsilon : & q \\ \sigma = X : & \mathcal{C}_k(\{[i,\ A{\to}\alpha{\bullet}X\beta,\ x{\bullet}y] \mid \\ & \qquad [i,\ A{\to}\alpha X{\bullet}\beta,\ x{\bullet}y] \in q\}) \\ \sigma = X\sigma_1 : & \delta(\delta(q,\sigma_1),X) \end{cases}$$

□

The following theorem generalizes Theorem 4.5 for any k, although obviously the language of stack suffixes are always the same: for $k' > k$, the automaton typically has just more states, because a longer lookahead string is used, what will be interesting for discriminating purposes.

Note again how B's phrase "cuts" the lookahead string into already covered (v) and not yet covered (x) parts.

The single, all-action automaton allows to take a parsing decision in a state when for a given lookahead vx the action i is unique in the state items, and thus it can be used for the same purposes as the pruned automaton from the construction algorithm.

Theorem 7.8 *Given a grammar $\mathcal{G}$ and an automaton $\mathcal{A}_k$ constructed according to Definition 7.7,*

$$[i,\ B{\to}\beta{\bullet}\rho,\ v{\bullet}x] \in \delta(q_0,\sigma) \ \textit{iff} \ \begin{cases} B \xrightarrow{h} \beta\rho,\ \rho \overset{\pi_2}{\underset{\text{rm}}{\Longrightarrow}}{}^* \sigma y w, \\ B \overset{\pi_1}{\underset{\text{rm}}{\Longrightarrow}}{}^* \varphi A w,\ \pi_1 j = h\pi_2, \\ S' \underset{\text{rm}}{\Rightarrow}^* \mu B z,\ \textit{and}\ v{\bullet}x = k:yw{\bullet}z \end{cases}$$

when $i = 0$ and $y \neq \varepsilon$ or $i = j$ and $y = \varepsilon$.

Proof. **"If" part:** By induction on $|\sigma|$:

- Induction base with $|\sigma| = 0$. By definition, $\delta(q_0, \sigma) = q_0$. There are two cases:

  - Case $y \neq \varepsilon$. Since $\rho \underset{\text{rm}}{\Rightarrow}^* yw \neq \varepsilon$, $\rho \neq \varepsilon$. There are two cases:

    * Case $\rho = a\rho_1$. Since $y \neq \varepsilon$ and $\rho \underset{\text{rm}}{\Rightarrow}^* yw$, $y = ay_1$ and $\rho_1 \underset{\text{rm}}{\Rightarrow}^* y_1 w$. Then $[0,\ B \rightarrow \beta \bullet a\rho_1,\ k : ay_1 w \bullet z] \in q_0$ by definition.
    * Case $\rho = C\rho_1$. Then $|\pi_2| > 0$, and since $\pi_1 j = h\pi_2$, $\pi_2 = \pi j$ such that

    $$\rho = C\rho_1 \overset{\pi}{\underset{\text{rm}}{\Longrightarrow}}^* y_1 A w \overset{j}{\underset{\text{rm}}{\Longrightarrow}} y_1 y_2 w = yw.$$

    There are two cases:

      · Case $y_1 = \varepsilon$. Then $y_2 = y \neq \varepsilon$, and since $S' \underset{\text{rm}}{\Rightarrow}^* \mu\beta A w z$, $[0,\ A \rightarrow \bullet y_2,\ k : y_2 \bullet wz] \in q_0$ by definition, and since $\rho \overset{\pi}{\underset{\text{rm}}{\Longrightarrow}}^* Aw$, the theorem holds by definition of $\mathcal{C}_k$ and Lemma 7.6.

      · Case $y_1 \neq \varepsilon$. Then $|\pi| > 0$. Let j' be the rule in π producing the $1 : y_1 = a$, that is, $\pi = \pi_a j' \pi_b$ such that

      $$\rho = C\rho_1 \overset{\pi_a}{\underset{\text{rm}}{\Longrightarrow}}^* \varphi_2 C' w_2 \overset{j'}{\underset{\text{rm}}{\Longrightarrow}} \varphi_2 \varphi_1 a \alpha_1 w_2 \text{ and } \alpha_1 \overset{\pi_b}{\underset{\text{rm}}{\Longrightarrow}}^* y_1' A w_1$$

      where $w_1 w_2 = w$ and $ay_1' = y_1$, what implies $\varphi_2\varphi_1 = \varepsilon$, and thus $\rho \underset{\text{rm}}{\Rightarrow}^* C' w_2$. Then, since $S' \underset{\text{rm}}{\Rightarrow}^* \mu\beta C' w_2 z$ and $\alpha_1 \underset{\text{rm}}{\Rightarrow}^* y_1' y_2 w_1$, $[0,\ C' \rightarrow \bullet a\alpha_1,\ k : ay_1' y_2 w_1 \bullet w_2 z] \in q_0$ by definition, and the theorem holds by definition of $\mathcal{C}_k$ and Lemma 7.6.

  - Case $y = \varepsilon$. Since $\sigma y = \varepsilon$, $\rho \overset{\pi_2}{\underset{\text{rm}}{\Longrightarrow}}^* w$. There are two cases:

    * Case $|\pi_2| = 0$. Since $\pi_1 j = h$, $j = h$ and $|\pi_1| = 0$, and then $B = \varphi A w$, what implies $w = \varepsilon = \rho$. But $[h,\ B \rightarrow \beta \bullet,\ k : \bullet z] \in q_0$ by definition.
    * Case $|\pi_2| > 0$. Since $\pi_1 j = h\pi_2$, $\pi_2 = \pi j$ such that

    $$\rho \overset{\pi}{\underset{\text{rm}}{\Longrightarrow}}^* w_1 A w_2 \overset{j}{\underset{\text{rm}}{\Longrightarrow}} w_1 \alpha w_2 = w.$$

    But $h\pi = \pi_1$, and thus $B \overset{\pi_1}{\underset{\text{rm}}{\Longrightarrow}}^* \beta w_1 A w_2 = \varphi A w$, and then $w_2 = w$, but since $w_1 \alpha w_2 = w$, $w_1 = \alpha = \varepsilon$. And since $S' \underset{\text{rm}}{\Rightarrow}^* \mu\beta A w_2 z$, $[j,\ A \rightarrow \bullet,\ k : \bullet w_2 z] \in q_0$ by definition, and since $\rho \underset{\text{rm}}{\Rightarrow}^* A w_2$, the theorem holds by definition of $\mathcal{C}_k$ and Lemma 7.6.

- Induction step with the "if" part assumed to hold for $|\sigma| = p$. The reasoning is the same for both cases $y \neq \varepsilon$ ($i = 0$) and $y = \varepsilon$ ($i = j$). Let $|\sigma'| = p + 1$, i.e., $\sigma' = X\sigma$, such that

$$\begin{cases} B' \xrightarrow{h'} \beta'\rho',\ \rho' \overset{\pi'_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^{*} \sigma' y w', \\ B' \overset{\pi'_1}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^{*} \varphi' A w',\ \pi'_1 j = h'\pi'_2, \\ S' \underset{\mathrm{rm}}{\Rightarrow}{}^{*} \mu' B' z',\ \text{and } v' \bullet x' = k : yw' \bullet z'. \end{cases}$$

Let $q = \delta(q_0, \sigma)$ and $q' = \delta(q_0, \sigma') = \delta(q, X)$ by definition. Since $\rho' \overset{\pi'_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^{*} X\sigma y w'$, there are the following two cases:

  - There is some ρ such that $\rho' = X\rho$ and $\rho \overset{\pi'_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^{*} \sigma y w'$. Thus $B' \rightarrow \beta' X \rho$, and choosing $\beta = \beta' X$, and since the "if" part is assumed to hold for $|\sigma| = p$, $[i,\ B' \rightarrow \beta' X \bullet \rho,\ k : yw' \bullet z'] \in q$, hence $[i,\ B' \rightarrow \beta' \bullet X\rho,\ k : yw' \bullet z'] \in q'$ by definition of δ.
  - There is no ρ such that $\rho' = X\rho$ and $\rho \overset{\pi'_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^{*} \sigma y w'$. Then $|\pi'_2| > 0$, and thus $\pi'_2 = \pi' j$ such that

$$\rho' \overset{\pi'}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^{*} \varphi_1 A w' \overset{j}{\underset{\mathrm{rm}}{\Longrightarrow}} \varphi_1 \alpha w' = X\sigma y w'.$$

    There are two cases:

    * Case $\varphi_1 = \varepsilon$. Then $\alpha = X\sigma y$. Choosing $|\pi_2| = |\pi_1| = 0$, $h = j$, $\beta = X$, and $\rho = \sigma y$, and since $S' \underset{\mathrm{rm}}{\Rightarrow}{}^{*} \mu' \beta' A w' z'$ and the "if" part is assumed to hold for $|\sigma| = p$, $[i,\ A \rightarrow X \bullet \rho,\ k : y \bullet w' z'] \in q$, and then $[i,\ A \rightarrow \bullet X\rho,\ k : y \bullet w' z'] \in q'$ by definition of δ, and since $\rho' \underset{\mathrm{rm}}{\Rightarrow}{}^{*} A w'$, the theorem holds by definition of $\mathcal{C}_k$ and Lemma 7.6.
    * Case $\varphi_1 \neq \varepsilon$. Then $|\pi'| > 0$. Let h be the rule in π' producing the $1 : \varphi_1 = X$, that is, $\pi' = \pi_a h \pi$ such that

$$\rho' \overset{\pi_a}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^{*} \varphi_2 B z_1 \overset{h}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^{*} \varphi_2 \varphi_3 X \rho z_1 \text{ and } \rho \overset{\pi}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^{*} \varphi_4 A w,$$

      where $wz_1 = w'$ and $X\varphi_4 = \varphi_1$, what implies $\varphi_2\varphi_3 = \varepsilon$ and $\varphi_4 \alpha = \sigma y$ because $\varphi_1 \alpha = X\sigma y$. Choosing $\pi_2 = \pi j$ and $\pi_1 = h\pi$,

$$\begin{cases} B \xrightarrow{h} X\rho,\ \rho \overset{\pi_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^{*} \sigma y w, \\ B \overset{\pi_1}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^{*} X \varphi_4 A w,\ \pi_1 j = h\pi_2, \text{ and} \\ S' \underset{\mathrm{rm}}{\Rightarrow}{}^{*} \mu' \beta' B z_1 z'. \end{cases}$$

Choosing $\beta = X$, $\varphi = X\varphi_4$, $\mu = \mu'\beta'$, and $z = z_1 z'$, and since the "if" part is assumed to hold for $|\sigma| = p$, $[i,\ B\to X\bullet\rho,\ k : yw\bullet z_1 z'] \in q$, and then $[i,\ B\to\bullet X\rho,\ k : yw\bullet z_1 z'] \in q'$ by definition of δ, and since $\rho' \underset{\text{rm}}{\Rightarrow}{}^{*} Bz_1$, the theorem holds by definition of $\mathcal{C}_k$ and Lemma 7.6.

"Only if" part: By induction on $|\sigma|$:

- Induction base with $|\sigma| = 0$. By definition, $\delta(q_0, \sigma) = q_0$. There are two cases:
  - Case $i = 0$. From the definitions of q_0 and $\mathcal{C}_k$, $S' \underset{\text{rm}}{\Rightarrow}{}^{*} \mu Bz$ and $B \xrightarrow{h} \beta\rho$, with the following two cases:
    - Case $\rho = a\rho_1$, $\rho_1 \overset{\pi_2}{\underset{\text{rm}}{\Longrightarrow}}{}^{*} t$, and $v\bullet x = k : at\bullet z$. There are two cases:
      - Case $|\pi_2| = 0$. The theorem holds choosing $\pi_1 j = h$ and $y = at$.
      - Case $|\pi_2| > 0$. Then $\pi_2 = \pi j$ such that

$$\rho = a\rho_1 \overset{\pi}{\underset{\text{rm}}{\Longrightarrow}}{}^{*} ay_1 Aw \overset{j}{\underset{\text{rm}}{\Longrightarrow}} ay_1 y_2 w = at,$$

        and the theorem holds choosing $\pi_1 = h\pi$, $y = ay_1 y_2$, and $\varphi = \beta a y_1$.
    - Case $\rho \overset{\pi}{\underset{\text{rm}}{\Longrightarrow}}{}^{*} Cw_2$, $C \xrightarrow{j'} a\alpha_1$, $\alpha_1 \overset{\pi_2'}{\underset{\text{rm}}{\Longrightarrow}}{}^{*} t$, and $v\bullet x = k : atw_2\bullet z$. There are two cases:
      - Case $|\pi_2'| = 0$. Choosing $\pi_2 = \pi j'$ and $\pi_1 = h\pi$:

$$\begin{cases} \rho \overset{\pi_2}{\underset{\text{rm}}{\Longrightarrow}}{}^{*} atw_2, \\ B \overset{\pi_1}{\underset{\text{rm}}{\Longrightarrow}}{}^{*} \beta C w_2, \text{ and } \pi_1 j' = h\pi_2, \end{cases}$$

        and the theorem holds choosing $y = at$, $w = w_2$, $\varphi = \beta$, and $A = C$.
      - Case $|\pi_2'| > 0$. Then $\pi_2' = \pi_1' j$ such that

$$\alpha_1 \overset{\pi_1'}{\underset{\text{rm}}{\Longrightarrow}}{}^{*} y_1 A w_1 \overset{j}{\underset{\text{rm}}{\Longrightarrow}} y_1 y_2 w_1 = t.$$

        Choosing $\pi_2 = \pi j' \pi_1' j$, $\pi_1 = h\pi j' \pi_1'$,

$$\begin{cases} \rho \overset{\pi_2}{\underset{\text{rm}}{\Longrightarrow}}{}^{*} ay_1 y_2 w_1 w_2, \\ B \overset{\pi_1}{\underset{\text{rm}}{\Longrightarrow}}{}^{*} \beta a y_1 A w_1 w_2, \text{ and } \pi_1 j = h\pi_2. \end{cases}$$

        Choosing $y = ay_1 y_2$, $w = w_1 w_2$, and $\varphi = \beta a y_1$, then $atw_2 = ay_1 y_2 w_1 w_2 = yw$.

- Case $A \xrightarrow{i} \alpha$. From the definition of q_0 and $\mathcal{C}_k$, $S' \underset{\mathrm{rm}}{\Rightarrow}^* \mu B z$ and $B \xrightarrow{h} \beta\rho$, with the following two cases:
  - Case $h = i$, $\rho = \varepsilon$, and $v \bullet x = k : \bullet z$. The theorem holds choosing $|\pi_2| = |\pi_1| = 0$.
  - Case $\rho \overset{\pi}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* A w$, $\alpha = \varepsilon$, and $v \bullet x = k : w \bullet z$. The theorem holds choosing $\pi_2 = \pi i$, $\pi_1 = h\pi$, and $\varphi = \beta$.

- Induction step with the "only if" part assumed to hold for $|\sigma| = p$. The reasoning is the same for both cases $i = 0$ ($y \neq \varepsilon$) and $A \xrightarrow{i} \alpha$ ($y = \varepsilon$ and $j = i$ by Convention 1.7). Let $|\sigma'| = p+1$, i.e., $\sigma' = X\sigma$, such that $[i,\ B' \rightarrow \beta' \bullet \rho',\ v' \bullet x'] \in \delta(q_0, \sigma')$. Let $q = \delta(q_0, \sigma)$ and $q' = \delta(q_0, \sigma') = \delta(q, X)$ by definition. From the definitions of δ and $\mathcal{C}_k$, $S' \underset{\mathrm{rm}}{\Rightarrow}^* \mu' B' z'$ and $B' \xrightarrow{h'} \beta'\rho'$, with the following two cases:

  - Case $\rho' = X\rho$ and $[i,\ B' \rightarrow \beta' X \bullet \rho,\ v' \bullet x'] \in q$. Choosing $\beta = \beta' X$ and since the "only if" part is assumed to hold for $|\sigma| = p$,

$$\begin{cases} B' \xrightarrow{h'} \beta' X \rho,\ \rho \overset{\pi_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \sigma y w, \\ B' \overset{\pi_1}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \varphi A w,\ \pi_1 j = h' \pi_2, \text{ and} \\ v' \bullet x' = k : yw \bullet z', \end{cases}$$

  and thus $\rho' = X\rho \overset{\pi_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* X\sigma y w$.

  - Case $\rho' \overset{\pi}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* B z_1$, $[i,\ B \rightarrow X \bullet \rho,\ v \bullet x] \in q$, and $v' \bullet x' = k : v z_1 \bullet z'$. Choosing $\beta = X$, and since the "only if" part is assumed to hold for $|\sigma| = p$,

$$\begin{cases} B \xrightarrow{h} X\rho,\ \rho \overset{\pi_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \sigma y w, \\ B \overset{\pi_1}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \varphi A w,\ \pi_1 j = h \pi_2, \text{ and} \\ v \bullet x = k : yw \bullet z. \end{cases}$$

  And then

$$\begin{cases} \rho' \overset{\pi}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* B z_1 \overset{h}{\underset{\mathrm{rm}}{\Longrightarrow}} X \rho z_1 \overset{\pi_2}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* X \sigma y w z_1, \\ B' \overset{h'}{\underset{\mathrm{rm}}{\Longrightarrow}} \beta' \rho' \overset{\pi}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \beta' B z_1 \overset{\pi_1}{\underset{\mathrm{rm}}{\Longrightarrow}}{}^* \beta' \varphi A w z_1, \end{cases}$$

  and $v = k : yw$ by definition, what by Lemma 7.4 implies $v' \bullet x' = k : y w z_1 \bullet z'$. Choosing $w' = w z_1$, $\varphi' = \beta' \varphi$, $\pi'_2 = \pi h \pi_2$, and $\pi'_1 = h' \pi \pi_1$, then $\pi'_1 j = h' \pi \pi_1 j = h' \pi h \pi_2 = h' \pi'_2$.

□

The same considerations for the LR(0) case apply here, too. In particular, it can be proved directly the LR(k) value of the method: calling

$$\mathrm{LA}_i(q) = \{vx \mid [i,\ B \to \beta \bullet \rho,\ v \bullet x] \in q\},$$
$$t \in \mathrm{LA}_i(\delta(q_0, \sigma)) \text{ iff } S' \underset{\mathrm{rm}}{\Rightarrow}^* \gamma A u \overset{j}{\underset{\mathrm{rm}}{\Longrightarrow}} \eta\sigma y u \text{ and } t = k : yu,$$

when $i = 0$ and $y \neq \varepsilon$ or $i = j$ and $y = \varepsilon$.

Definition 7.9 *Left-hand context language associated to an LR(k) item.*

$$\mathcal{L}([i,\ B \to \beta \bullet \rho,\ v \bullet x]) = \{\mu\beta \mid S' \underset{\mathrm{rm}}{\Rightarrow}^* \mu B z \textit{ and } k : v \bullet z = v \bullet x\}$$

□

Lemma 7.10 *Left-hand context language associated to an LR(k) item.*

$$\mathcal{L}([i,\ B \to \beta \bullet \rho,\ v \bullet x]) = \{\vdash \mu_0 \mid \delta(\{[i,\ B \to \beta \bullet \rho,\ v \bullet x]\},\ \vdash \mu_0) \neq \emptyset\}$$

Proof. □

Lemma 7.11

$$\mathcal{L}([i,\ B \to \beta \bullet \rho,\ v \bullet x]) \begin{cases} = \mathcal{L}([j,\ B \to \beta \bullet \mu,\ u \bullet x]) \\ \supseteq \mathcal{L}([j,\ B \to \beta \bullet \mu,\ u \bullet wx]) \end{cases}$$

Proof. □

Thus, $\mathcal{L}$ does not depend upon i, ρ, nor v. In this sense, an LR(k) item could also be written as $[B, \beta, x]$, although i and v are still necessary for discriminating purposes.

Definition 7.12 *For a given grammar $\mathcal{G}$, stack suffix (plus lookahead) $\sigma \mid w$, $|w| \geq 0$, is said to be* compatible *with action i iff there is some stack form $\eta\sigma \mid wx$ compatible*[1] *with i. A stack suffix (plus lookahead) with at least one compatible action is said to be* valid. □

For LR(k) grammars, valid full-stack plus k-lookahead configurations always have only one compatible action. The following definition of discriminant stack suffix plus lookahead allows the characterization of (deferred-lookahead) discriminating reverse parsing.

Definition 7.13 *A stack suffix plus lookahead $\sigma \mid w$ is called* deferred-lookahead discriminant *(or simply* discriminant*) iff*

1. *it has only one compatible action, or else,*

[1] See Definition 3.3 in page 39.

2. *for all but one of the actions compatible with σ, for each compatible $\sigma | w'$, there is some suffix σ_0, $\sigma = X\sigma_1\sigma_0$, such that $\sigma_0 | w'$ has a unique compatible action and $w' \neq w$.*

That compatible action is said to be discriminated *by $\sigma | w$.* □

Note that, according to the definition, $\sigma | w$ does not need to be compatible with any action, when condition 1 does not apply. According to the following proposition, this is the case only for erroneous inputs, since for sentences, all stack suffix plus lookahead configurations are necessarily compatible with some action.

Proposition 7.14 *If stack suffix plus lookahead $\sigma | w$ discriminates, and at the same time is compatible with, action i, then i is the only compatible action.*

Proof. If condition 2 in Definition 7.13 does apply, it is not possible for another different action i' to be compatible, since according to the referred definition, all the w' are different from w. In other words, in these circumstances, condition 2 implies condition 1. □

Using the above definition, discriminating reverse parsing is characterized by the following theorem, showing that the discriminating reverse method is optimal in terms of depth of stack search.

Theorem 7.15 *The discriminating reverse LR(k) automaton for an LR(k) grammar $\mathcal{G}$ performs parsing action $\eta' | z' \stackrel{i}{\Longrightarrow}\!\!\dashv \eta \bullet \sigma | z$ if, and only if, σ is the shortest valid suffix of $\eta\sigma$ such that $\sigma | k : z$ discriminates action i. Otherwise, an error is signalled after exploring the shortest*[2] *σ such that condition 2 in Definition 7.13 holds for all compatible actions, if any, with no exception.*

Proof. **"If" part:** Let us first suppose that such σ does exist for current configuration $\eta\sigma | z$. If condition 1 of Definition 7.13 applies then, after exploring σ, there is only one possible action for lookahead $k : z$, and then it is decided by the automaton without further stack exploration, else if condition 2 applies, for all those σ_0, at the corresponding states there are decisions on such w' and then these items are removed, and thus do not reach the stack-symbol decision step of current state, where the decision is taken on the stack symbol as there is only one remaining action i. Otherwise, if there

[2]The theorem does not consider the additional transition to perform in case of nullable reduction in order to verify lookahead compatibility, as shown in Section 5.4.5. Note that the fact of including such transitions does not increase (except by one more symbol, when a syntactic error is detected) the depth of the explored stack suffix, since the new "verification" state only has decision entries on lookahead, but none on subsequent stack symbol.

is no such σ, condition 1 does not apply, and then there is no state visited deciding on the lookahead, and since condition 2 holds for all the compatible actions, there is at some state no remaining item to build a transition to a new state, and then an error is signalled at that point by the automaton.

"Only if" part: The automaton takes a decision on the lookahead when it is the only legal possibility at this state, and thus it is taken at the shortest valid σ such that condition 1 applies. If the decision is taken on the stack symbol, it means that it is the only legal remaining possibility, because, either at this or previous state some other possibilities are discarded after testing that the current value of the lookahead is not valid for them, even though it would have been otherwise discriminant, i.e., such prefixes σ_0. Finally, an error is detected if there is no transition on the lookahead, that is, there is no legal possibility, now that all the other possibilities have been discarded on different lookahead values, different from current $k:z$. $\square$

Of course, the set of actions compatible with $\rho|w$ is included in the set of actions compatible with simply ρ. Consider a nonempty set of compatible actions for some $\rho|w$ during stack suffix exploration. It is possible that for some next stack symbol X, the set of actions compatible with $X\rho|w$ reduces to empty, while the set for $X\rho$ remains nonempty. In this case, since lookahead use is deferred, the error might not be detected at this point, and actually some parsing action might be performed if at some point in the exploration the set of compatible actions for the stack suffix alone reduces to a single action.

The following lemma shows that reductions performed by the discriminating reverse automaton are locally correct.

Lemma 7.16 $\alpha_2|x_2 \Dashv^+ \alpha_1|x_1$ *implies* $\alpha_2 x_2 \underset{\mathrm{rm}}{\Rightarrow}^* \alpha_1 x_1$.

Proof. It is immediate for a single shift $\beta a|y \Dashv \beta|ay$, and for a single reduction $\beta A|y \Dashv \beta\gamma|y$. It is thus correct for a sequence of shifts and reductions. $\square$

As Lemma 4.13 shows, in the LR(0) case, a correct parsing stack can become incorrect as a consequence of a shift action —never from a reduce action—, when the shifted terminal is incompatible with the rest of the stack. But for LR(k), $k > 0$, the deferred use of lookahead without lookahead verification may involve an unnoticed incompatibility of correct stack and current lookahead, which may lead to an incorrect stack as a consequence of a reduce action, as well as from a shift action. Consider for instance two correct stack-plus-lookahead combinations: $\eta\sigma\gamma|aw$ and $\eta'\sigma\gamma|a'w'$, where $\eta \neq \eta'$ and $a \neq a'$. If the actual combination during parsing is, for instance, $\eta'\sigma\gamma|aw$, and in both legal combinations the correct action is to shift after exploring stack suffix $\sigma\gamma$, the stack becomes illegal: $\eta'\sigma\gamma a$. Another case is that the correct action in both cases is to reduce, and thus, while, for instance, both $\eta\sigma A$ and $\eta'\sigma A'$ are correct, if the actual combination during parsing is $\eta'\sigma\gamma \mid aw$, the stack becomes illegal if only $\sigma\gamma$ is explored: $\eta'\sigma A$.

We are going to prove that the discriminating reverse LR(k) parser does not loop forever on any input. Here the deferred lookahead construction described in the text is considered, such that the k-lookahead is used only if it may suffice to decide, as well as in cases of nullable reductions decidable on stack nullable nonterminal[3]. In these cases, that nonterminal is visited to take a transition to a new state where the k-lookahead is used to decide, and thus its compatibility is verified.

Lemma 7.17 *If $\mathcal{G}'$ is LR(k), then there is a bound to all $|\nu|$, such that $S' \underset{\mathrm{rm}}{\Rightarrow}^* \mu\nu z$ and $\nu \Rightarrow^* \varepsilon$.*

Proof. If there is not such a bound, sufficiently long ν implies $A \underset{\mathrm{rm}}{\Rightarrow}^* \nu_0 A z_0$ and $\nu_0 \Rightarrow^* \varepsilon$, with $z_0 \neq \varepsilon$, since otherwise $\mathcal{G}'$ would be ambiguous. But then $A \underset{\mathrm{rm}}{\Rightarrow}^* \nu_0^m A z_0^m$, and thus

$$S' \underset{\mathrm{rm}}{\Rightarrow}^* \mu_1 A z_1 \underset{\mathrm{rm}}{\Rightarrow}^* \mu_1 \nu_0^m A z_0^m z_1 \underset{\mathrm{rm}}{\Rightarrow} \mu_1 \nu_0^m y z_0^m z_1 \underset{\mathrm{rm}}{\Rightarrow}^* \mu_1 y z_0^m z_1,$$

and thus at $\mu_1 \mid y z_0^m z_1$, for sufficiently large m, it is impossible for an LR(k) parser to tell how many ν_0 produce in the stack. $\square$

Next lemma generalizes Lemma 4.13 when k-lookahead compatibility is verified, as is the case of the solution to avoid loops of reductions from empty, showing that in that case the topmost "working" suffix of the stack, plus the k-lookahead, always remains correct. The difference between this general k-lookahead case, and the 0-lookahead case, is that, in our deferred lookahead parser, when the decision is taken on next stack symbol, it is locally correct if we *suppose* that the k-lookahead is locally compatible, otherwise, for an incompatible k-lookahead, in general we cannot guarantee that the decided action is the only possible one for any compatible lookahead.

Lemma 7.18 *Let a discriminating reverse LR(k) automaton $\mathcal{A}_k$ for an LR(k) grammar, $\alpha_2 \mid x \mathrel{=\!\!\!\dashv}^+ \alpha_1 \mid x$, and let $\sigma_2 A$ be the longest suffix of α_2 consisting only of visited or generated stack nodes during the above sequence of actions, such that k-lookahead compatibility is verified for all such reductions. Then $S' \underset{\mathrm{rm}}{\Rightarrow}^* \zeta\sigma_2 A z$ and $k : x = k : z$.*

Proof. By induction on the number p of parsing steps:

- Induction base with $p = 1$. Let $\alpha_2 \mid x \mathrel{=\!\!\!\dashv} \eta_1 \cdot \sigma_1 \mid x$. By Theorem 7.8,

$$S' \underset{\mathrm{rm}}{\Rightarrow}^* \varphi A z \underset{\mathrm{rm}}{\Rightarrow} \varphi\gamma z = \zeta\sigma_1 z$$

[3]See Definition 1.2 in page 2. Note that it is not necessary that they actually produce the empty string according to the reductions made by the parser.

and $k : x = k : z$, since lookahead is verified. In case $|\sigma_1| > |\gamma|$, $\varphi = \zeta\sigma_2$, otherwise $\sigma_2 = \varepsilon$.

- Induction step with some $p > 1$. Thus we suppose

$$\alpha'_2 A' \,\mathbf{|}\, x \overset{i}{=\!\!\dashv} \eta'_1 \bullet \sigma'_1 \,\mathbf{|}\, x,$$

$$A' \xrightarrow{i} \gamma',$$

$$\eta'_1 \sigma'_1 = \eta_2 \sigma_2 A,$$

$$\eta_2 \sigma_2 A \,\mathbf{|}\, x =\!\!\dashv^{p} \alpha_1 \,\mathbf{|}\, x,$$

$$S' \underset{\mathrm{rm}}{\Rightarrow}^{*} \zeta \sigma_2 A z, \text{ and}$$

$$k : x = k : z.$$

If $|\gamma'| < |\sigma_2 A| > |\sigma'_1|$, then $\zeta\sigma'_2 A' z \underset{\mathrm{rm}}{\overset{i}{\Longrightarrow}} \zeta\sigma'_2\gamma' z = \zeta\sigma_2 A z$, which is, by Theorems 7.15 and 7.8, the only legal possibility after exploring σ'_1 and verifying compatibility of $k : x$, and thus the lemma holds. Otherwise, as in case $p = 1$, the lemma holds.

□

Lemma 7.19 *Let a discriminating reverse LR(k) parser for an LR(k) grammar, such that lookahead compatibility is always verified before deciding a nullable reduction after exploring stack suffix $C\sigma$ for nullable C. Then there is a bound to the sequence π of reductions performed by that parser from empty: $\eta\xi \,\mathbf{|}\, x \overset{\pi}{=\!\!\dashv}{}^{+} \eta \,\mathbf{|}\, x$ such that $\xi \underset{\mathrm{rm}}{\overset{\pi}{\Longrightarrow}}{}^{+} \varepsilon$.*

Proof. Let us suppose the lemma to be false. The length of such ξ cannot be bounded, because an unbounded number of reductions from ε would permit to build an unbounded tree, what is impossible for unambiguous grammars, such as LR(k). But if their length is unbounded, after a bounded number of such reductions, all the decisions are taken on nullable nonterminals, since Lemma 7.17 shows that the automaton cannot have loops on paths consisting only on nullable nonterminals. Since k-lookahead compatibility is performed in these cases, from that point on, these sequences of reductions must be correct, according to Lemma 7.18, and thus bounded. □

Theorem 7.20 *Given an LR(k) grammar and its corresponding discriminating reverse LR(k) automaton $\mathcal{A}$, for any finite input x, the parser stops in finite time, either after a successful parsing if x is a sentence, or by signalling a syntactic error otherwise.*

Proof. If x is a sentence from an LR(k) grammar, the number of parsing steps is finite, and the parsing stack is always finite, thus for each parsing step, only a finite number of steps during stack suffix exploration can be performed.

If input x is not a sentence, but is finite, then only a finite number of shifts is possible, as well as a finite number of single reductions to nonterminals A_i producing some input subsequence w_i, i.e., $A_i \underset{\mathrm{rm}}{\Rightarrow}^{+} w_i$. Again, for each of these actions, the number of exploration steps is finite, if the stack contains only these nonterminals and terminals, and thus is finite.

The only remaining case are some sequences of nonterminals resulting from reductions from empty, whose number is bounded, according to Lemma 7.19, and thus the length of such nonterminal sequences. Since such sequences are produced between terminals and/or nonempty-producing nonterminals, whose number is finite as it has been shown above, the total number of stack configurations is finite, as well as their length. Thus, the total number of possible explorations steps is finite. □

7.5.1 Parsing time complexity

In this section, linear time behavior of the discriminating parsing algorithm is formally proved. It is interesting indeed to know time efficiency not only in the presence of correct sentences but also on any possible erroneous input. Remember that in the latter case the stack may contain a sequence that is not a prefix of rightmost sentential form, although every reduction is locally correct, as we have seen.

Definition 7.21 *Given an LR(k) discriminating reverse automaton $\mathcal{A}$ for a reduced grammar $\mathcal{G}$, and a terminal input sequence x_0, a* stack node *is a pair (X, p) —also written $X^{(p)}$— such that*

$$\alpha_p X \,|\, x_p \Longrightarrow\!\!\!\!\dashv^{(p)} \cdots \Longrightarrow\!\!\!\!\dashv^{(1)} \vdash |\, x_0.$$

That is, X is put on top of the stack at the p-th parsing action. □

It should be clear from the definition that a stack node, defined as above, uniquely identifies any node in the stack for any input x, and in particular any node in the syntax tree of x when it is a sentence. It should also be evident that every stack form has a corresponding stack node "form", where each symbol is replaced by its corresponding stack node. Thus, instead of saying that a stack symbol is visited during a reverse stack suffix scan, it is more precise to say that a stack node is visited. Finally, the definition clearly shows that there are as many stack (syntax tree) nodes as parsing actions.

Theorem 7.22 *Given an unambiguous reduced grammar $\mathcal{G}'$, for any derivation $S' \Rightarrow^{(r)} \alpha_r \Rightarrow^{(r-1)} \cdots \Rightarrow^{(1)} \alpha_1 = z$, $r = O(|z|)$.*

Proof. Corollary 4.26 in [SSS90] implies that for any sentence of any grammar there is a derivation whose time complexity is linear with the length of the sentence and thus that there is some syntax tree whose number of nonterminal nodes —each of them corresponding to a derivation step— is linear with the length of the grammar, but for unambiguous grammars this tree is unique. □

The above result is also a corollary of theorem 12.2.1 in [Har78], where derivation linearity is proved for grammars where $A \Rightarrow^+ A$ is impossible, thus including all unambiguous grammars. Another related result is time linearity of direct LR(k) parsing; see theorem 5.13 in [AU73a].

Lemma 7.23 *The number n of stack nodes for a sentence z from an LR(k) reduced grammar $\mathcal{G}$ is $O(|z|)$.*

Proof. From Definition 7.21 and Theorem 7.22, $n = |z| + r = |z| + O(|z|) = O(|z|)$. □

Lemma 7.24 *The number n of stack nodes for input sequence x in a discriminating reverse LR(k) parse is $O(|x|)$.*

Proof. Lemma 7.23 proofs the case when x is a sentence. Otherwise the automaton may left symbols in the stack, whose corresponding stack nodes are:

- $O(|x|)$ for terminals, obviously.
- $O(|x|)$ for trees from nonterminals A_i, $i = 1, \ldots, m$, producing terminal phrases $x_i \neq \varepsilon$, because in fact these are trees for grammar $\mathcal{G}_i = (V_N, V_T, P, A_i)$, which are unambiguous because $\mathcal{G}$ is, and thus $n_i = O(|x_i|)$ from Theorem 7.22, what implies that there are some constants L, M such that if $|x_i| \leq L$ then $n_i < M$ and otherwise $n_i < M|x_i|$, and thus $n = \sum n_i < M \sum |x_i|$. Thus $n = O(|x|)$.
- $O(|x|)$ for trees from nonterminals A_j, $j = 1, \ldots, m'$, producing ε, since, according to Lemma 7.19, there is a bound to their number of reductions —each one corresponding to a stack node— which might be performed between each pair of symbols from the two preceding cases, whose number is $O(|x|)$.

□

Lemma 7.25 *If the number of times that any stack node $X^{(p)}$ of an LR(k) reduced grammar $\mathcal{G}$ is visited during a stack scan is bounded to some value*

K, then the sum of the number of visits in all successive stack scans is linear with the length of the input sequence.

Proof. If there is such a bound K, then for an input sequence x the total number of visits during all successive scans is $v \leq Kn$, where n is the number of stack nodes for x. And by Lemma 7.24 $v \leq KO(|x|)$ and thus $v = O(|x|)$. □

The following two theorems assume that no state is constructed where the stack suffix σ explored is longer than necessary, i.e., that as early as the suffix plus the k-lookahead allows to discriminate, the automaton ends exploration. This is a strong condition, allowing the following two forms of construction:

- The lookahead is used only in a special initial state to separate the different sub-cases. In each other state all items share the same lookahead sequence.
- The "standard" construction, where in each state the lookahead is always checked first to decide if possible, and if this fails then next stack symbol is used to decide, perform a transition to next state, or signal a syntactic error. In the intermediate item set of each state, for each lookahead w there are at least two different actions compatible with it.

And thus linearity is not proved for less restricted constructions, such as the following:

- Lookahead use is delayed until it is sure that it will discriminate in a legal sentence and thus end exploration. In the initial item set of each state with some decision or transition on next stack symbol, there is some lookahead w such that there are at least two different actions compatible with it.
- A reverse quasi-LR(0) automaton is produced and the lookahead used only at bottom-of-stack if necessary. In the item set of a state, each lookahead w might have only a single compatible action.

Nor it is proved for finer discriminating automata, such as the one described in Section 8.3.6.

Lemma 7.26 *For a given LR(k) grammar $\mathcal{G}$ there is a bound to the number of times that a stack node is visited by the discriminating reverse LR(k) automaton to decide all the parsing actions for a sentence.*

Proof. For the sake of contradiction, suppose that for any given integer constant K there is some sentence z with some stack forms containing a stack node $X^{(p)}$ that is visited a number v of times, $v > K$. Thus, for sufficiently large K, for some sentence z, a state q of the discriminating reverse LR(k) automaton is repeatedly visited with the same w as lookahead, then having a transition on X from which the same $X^{(p)}$ is visited, and eventually deciding the same action i. That q has an associated set of different parsing actions compatible with the portion of the stack which has already been scanned, such that this stack suffix plus the lookahead has not yet discriminated, provoking the visit of the next lower stack node $X^{(p)}$, and thus there is at least another action i' which is compatible with the same stack plus k-lookahead. This implies that for all such cases there exists some stack prefix η' which, if scanned from q would led to decide i', from state q. This should be clear by considering that the discriminating reverse automaton can be extended from q by inhibiting deciding i' on lookahead w, and then completely scanning the stack in those cases. All of the above implies that for some sentence z,

$$S' \underset{\mathrm{rm}}{\Rightarrow}^* \eta\sigma_f w y_f \underset{\mathrm{rm}}{\Rightarrow}^* \eta\sigma_g w y_g \underset{\mathrm{rm}}{\Rightarrow}^* z,$$

where σ_f and σ_g are two stack suffixes leading from initial state to such a state q. Note that the above includes the case $\sigma_f w y_f = \sigma_g w y_g$, what implies $i = 0$. Furthermore, there must also exist two rightmost sentential forms,

$$\eta'\sigma_f w y'_f \text{ and } \eta'\sigma_g w y'_g,$$

for action i', which are precisely amongst the other possibilities that have led the automaton to read $X^{(p)}$ from q. But since $\sigma_f \underset{\mathrm{rm}}{\Rightarrow}^* \sigma_g t$, such that $t w y_f = w y_g$, then there is some sentence z' such that

$$S' \underset{\mathrm{rm}}{\Rightarrow}^* \eta'\sigma_f w y'_f \underset{\mathrm{rm}}{\Rightarrow}^* \eta'\sigma_g t w y'_f \underset{\mathrm{rm}}{\Rightarrow}^* z'.$$

But since $k : t w y'_f = w$, it is not possible to tell which of the two possible different actions i and i' is the correct one from stack-plus-k-lookahead contents $\eta'\sigma_g w$, in contradiction with the assumption that the grammar is LR(k). $\square$

Theorem 7.27 *The discriminating reverse LR(k) automaton for an LR(k) reduced grammar $\mathcal{G}$ parses any sentence z in time $O(|z|)$.*

Proof. Immediate from lemmas 7.26 and 7.25. $\square$

Lemma 7.28 *Given a non-valid stack suffix, it cannot become valid as a consequence of a discriminating reverse parsing action.*

Proof. There are two cases:

- Shift action. Non-valid stack suffix σ becomes σa. Since σa is valid, there is some derivation

$$S' \underset{\mathrm{rm}}{\Rightarrow}^* \gamma A u \overset{j}{\underset{\mathrm{rm}}{\Longrightarrow}} \eta\sigma a y u.$$

Let $y' = ay$. Then by Definition 7.12 stack suffix σ is valid, in contradiction with the assumption that it is non-valid.

- Reduce action. Non-valid stack suffix $\sigma\alpha$ becomes σA by reducing using rule $A \xrightarrow{j} \alpha$. Since σA is valid, there is some derivation

$$S' \underset{\mathrm{rm}}{\Rightarrow}^* \eta\sigma A y u \underset{\mathrm{rm}}{\overset{j}{\Longrightarrow}} \eta\sigma\alpha y u.$$

Let $\gamma' = \eta\sigma$, $u' = yu$, $\sigma' = \sigma\alpha$, and $y' = \varepsilon$. Then by Definition 7.12 stack suffix $\sigma\alpha$ is valid, in contradiction with the assumption that it is non-valid.

□

Theorem 7.29 *The discriminating reverse LR(k) automaton for an LR(k) reduced grammar $\mathcal{G}$ parses any input x in time $O(|x|)$.*

Proof. Suppose the statement to be false. Then there is some LR(k) reduced grammar $\mathcal{G}$ whose discriminating reverse LR(k) automaton cannot parse some inputs x in time $O(|x|)$. But by Lemma 7.25 this implies that there is no bound to the number of times that a stack node $X^{(p)}$ is visited by the discriminating reverse automaton considering all the parsing actions, and thus for some input x, a state q of the discriminating reverse LR(k) automaton visiting $X^{(p)}$ is repeatedly visited with the same w as lookahead, and thus deciding the same action i. Let the following be such sequence of parsing actions

$$\cdots \Longrightarrow\!\!\!\mid^+ \eta X^{(p)}\sigma_F \,|\, wy_F \Longrightarrow\!\!\!\mid^+ \cdots \Longrightarrow\!\!\!\mid^+ \eta X^{(p)}\sigma_1 \,|\, wy_1 \Longrightarrow\!\!\!\mid^+ \vdash |\, x,$$

where (possibly illegal) stack forms $\eta X^{(p)}\sigma_f \,|\, wy_f$, $f = 1, \ldots, F$, bring the automaton to such visits, —a syntactic error has not been detected yet. This implies that all $X^{(p)}\sigma_f$ must be valid. By applying iteratively Lemma 7.28, all the intermediate stack suffixes $X^{(p)}\sigma$ are valid. Moreover, according to Lemma 7.16, all the derivations are rightmost, i.e., $\sigma_F \underset{\mathrm{rm}}{\overset{\pi}{\Longrightarrow}}^* \sigma_1 t$.

In q there must be items of the form $[i, A \rightarrow \beta X \bullet \rho, w']$, for some k-lookahead w', —which is by construction compatible with all σ_f—, such that $X^{(p)}$ is visited for transition or decision. Consider $\eta X^{(p)}\sigma_F \,|\, wy_F$. There are two cases:

- w is not compatible with such σ_F (although, because of deferred use of lookahead, the syntactic error is not yet detected at this point). But then $|t| < |w|$ in $\sigma_F \underset{\mathrm{rm}}{\overset{\pi}{\Longrightarrow}}^* \sigma_1 t$, for otherwise w would be compatible at q —i.e., w would be in q items. Since there is no bound to F and the number of shifts is only $|t|$, there is some sub-derivation π', $\pi = \pi_2\pi'\pi_1$, such that $\sigma_{F_2} \underset{\mathrm{rm}}{\overset{\pi'}{\Longrightarrow}}^* \sigma_{F_1}$ ($F_2 \leq F$ and $F_1 \geq 1$), with no bound to

$F' = F_2 - F_1 + 1$. Since in items of state q there is some lookahead w' compatible with σ_{F_2}, it is also compatible with all $\sigma_{F_2}, \ldots, \sigma_{F_1}$.

- w is compatible with such σ_F. In this case, let $w' = w$ and $\pi' = \pi$ ($F' = F_2 = F$ and $F_1 = 1$).

In both cases, there must be some legal stack prefix η' and rest of input y'_{F_2} such that $S' \underset{\text{rm}}{\Rightarrow}^* \eta' X^{(p')} \sigma_{F_2} w' y'_{F_2}$. From this rightmost sentential form, the following derivation can be obtained by applying $\sigma_{F_2} \overset{\pi'}{\underset{\text{rm}}{\Longrightarrow}}^* \sigma_{F_1} t'$:

$$S' \underset{\text{rm}}{\Rightarrow}^* \eta' X^{(p')} \sigma_{F_2} w' y'_{F_2} \overset{\pi'}{\underset{\text{rm}}{\Longrightarrow}}^* \eta' X^{(p')} \sigma_{F_1} w' y'_{F_1} \underset{\text{rm}}{\Rightarrow}^* z.$$

Since w' is in such items in q, $X^{(p')}$ is visited for each such stack suffix $\sigma_{f'}$, $f' = F_1, \ldots, F_2$. But then there is no bound to the number F' of times that $X^{(p')}$ is visited, what is in contradiction with Lemma 7.26. □

Theorem 7.30 *The discriminating reverse LR(k) automaton for an LR(k) reduced grammar $\mathcal{G}$ parses any input x in time $O(|x|)$, including rightpart and nullable reduction verifications.*

Proof. From the fact that there is a maximum length of $\mathcal{G}$'s rightparts, and by Lemma 7.24 and Theorem 7.29. And from the fact that on nullable reduction decisions on nullable nonterminals an additional stack symbol is never visited, except when the added lookahead verification fails, in order to immediately detect the syntactic error. □

Chapter 8

Extensions

In this chapter we shall briefly describe several possibilities of extensions and variants to the method. They are presented for the practical LR(1) case, although they are readily generalized to the general LR(k) case.

In the first place, we shall present the possibility of integration, within the automaton itself, of the verification of rests of handles when the decision can be taken before they have been completely explored. In a second section we shall present a computationally cheap test to detect when two reverse items are in conflict, in order to apply user-specified conflict-resolution rule, e.g., *à la* yacc. Finally, we shall present several ideas to improve in different directions the decision mechanism at each state, ranging from the simple removal of some redundant entries to the possibility of obtaining (often appreciated) immediate error detection.

8.1 Full handle verification

Frequently, our reverse automaton can discriminate a reduction without having completely explored the handle to be reduced. In other words, there may exist legal suffix-plus-lookahead combinations which are only compatible with some reduction $A \to \alpha$, but where the suffixes are themselves proper suffixes of α. Of course, we must be sure of the presence on top of the stack of the whole rightpart to reduce, since otherwise the parser might perform an illegal reduction.

8.1.1 Action dot in items

There is the trivial, and probably quite practical, solution of verifying the rest of the rightpart to reduce through an auxiliary mechanism. This can be very easily implemented. Nonetheless, we also propose an alternative straightforward solution, which is quite natural within the framework of the method, and which, depending on the particular way of implementing the parsing tables, might prove itself useful. This solution is to continue automaton construction, even when the decision can be taken from the suffix read so far, until the corresponding handle is completely read.

We shall show only how to apply this idea to the LR(1) case. It is immediate to extend it to any LR(k), $k \geq 0$.

In our solution items will carry the information of whether the handle to reduce has been already completely explored. For that purpose we extend our item notation with yet another dot, this time in the item action part, as, for instance, in

$$[i\bullet,\ A \rightarrow \alpha \bullet \beta,\ \bullet a].$$

In order to understand our choice for the notation, remember that the closure operation takes place only for items whose rightpart dot has reached the left end of its rightpart. Thus the closure itself serves to detect that the corresponding rightpart has been completely explored. Thus, in our convention, in items produced by the closure computation, this *action dot* will be always placed to the left of the action number i to indicate that the rightpart corresponding to rule i has already been completely explored by the automaton, i.e., the rightpart dot has already surpassed the left end of i's rightpart.

According to this, for initial state items corresponding to a reduce action, before closure they are written with the dot to the right[1] of the action number, as in

$$[i\bullet,\ A \rightarrow \alpha\beta\bullet,\ \bullet a].$$

While the rightpart dot is being moved to the left by the only reason of an automaton state transition, the action dot remains in the same position, until, as we have seen, the closure produces items with the dot to the left.

Thus, it is clear that the generator can tell from the items whether the handle corresponding to a reduce action has already been covered or not, in order to inhibit the generation of the corresponding reducing action until it has been completely covered, and thus always completely exploring the handles before reduction, even if it is not necessary for discrimination. More precisely, a non-covered rightpart of a reducing item implies that

- deciding this reduction on the lookahead is inhibited, and that
- deciding the reduction on the next stack symbol is inhibited, unless it is precisely the leftmost symbol in the rightpart.

In order to use an homogeneous notation, shift action numbers ($i = 0$) also carry the action dot, although for us it is useless, in principle. Since the closure computation always puts it to the left, it was decided that in the

[1] In our original notation, instead of an action dot, an apostrophe was used on the action number. Its presence was equivalent to our action dot on the right, while its absence was equivalent to the action dot on the left.

initial state items for shift actions it was also put to the left of the action number. From that point on, it remains there.

Note that some combinations are not possible in our notation. In particular $[i\bullet,\ A\rightarrow\alpha\bullet\beta, a\bullet]$ is not possible.

To summarize, we formally define below the construction procedure of the reverse stack recognizing automaton using these items.

8.1.2 Closure and initial state

The closure of a set q of items is defined as follows.

$$\begin{aligned}\mathcal{C}(q) = q \quad & \cup \quad \{[\bullet i,\ B\rightarrow\beta\bullet C\gamma,\ \bullet a] \mid [i,\ C\rightarrow\bullet\varphi,\ \bullet a] \in \mathcal{C}(q) \text{ and } \varepsilon \in \mathrm{First}(\gamma) \text{ and} \\ & \qquad a \in \mathrm{Follow}(B) \text{ and } B\rightarrow\beta C\gamma\} \\ & \cup \quad \{[\bullet i,\ B\rightarrow\beta\bullet C\gamma,\ a\bullet] \mid ([\bullet i,\ C\rightarrow\bullet\varphi,\ a\bullet] \in \mathcal{C}(q) \text{ or} \\ & \qquad [C\rightarrow\bullet\varphi,\ \bullet a] \in \mathcal{C}(q) \text{ and } a \in \mathrm{First}(\gamma)) \text{ and } B\rightarrow\beta C\gamma\}\end{aligned}$$

From a set q of items, a transition on the next lower stack symbol X to another item set q' is defined as follows.

$$q' = \delta(q, X) = \mathcal{C}(\{[i,\ B\rightarrow\beta\bullet X\gamma,\ a] \mid [i,\ B\rightarrow\beta X\bullet\gamma,\ a] \in q\})$$

For the action and lookahead parts of items, we use the convention that if one or both dots are not represented, it or both can be in any position. Of course, we consider that item $[i,\ B\rightarrow\beta\bullet X\gamma,\ a]$ has the action and lookahead dots in the same position as the corresponding item $[i,\ B\rightarrow\beta X\bullet\gamma,\ a]$.

A full stack recognizing automaton can be built from the following initial state q_0.

$$\begin{aligned}q_0 = \quad & \mathcal{C}(\{[i\bullet,\ A\rightarrow\alpha\bullet,\ \bullet a] \mid A\xrightarrow{i}\alpha \text{ and } a \in \mathrm{Follow}(A)\} \\ & \cup \{[\bullet 0,\ A\rightarrow\alpha\bullet a\beta,\ a\bullet] \mid A\rightarrow\alpha a\beta\})\end{aligned}$$

Again, for the discriminating reverse LR(1) automaton with full rightpart verification, many states and items will be pruned during the construction process. The difference with the construction of Chapter 5 is that some reduction decisions are inhibited, and thus some items are not eliminated until rightpart verification by the automaton itself.

8.1.3 Reduction inhibition on lookahead or stack symbol

Let us see how to implement reduction inhibition in our two-step decision procedure for each state. First, lookahead is used for decision, as usual, but now reductions take place only if it is verified that the rightpart to reduce has been fully explored. Consider reduce-action items which are not the result of a closure computation, i.e., items with the action dot to the right. In these items, the rightpart coincides with the handle to reduce. Hence, this handle has not been fully explored by the automaton if the rightpart dot is still in the

$$\begin{array}{ll}
1: & S' \rightarrow \vdash S \dashv \\
2: & S \rightarrow Aa \\
3: & S \rightarrow Bb \\
4: & S \rightarrow dcCb \\
5: & A \rightarrow edc \\
6: & B \rightarrow fc \\
7: & C \rightarrow \varepsilon
\end{array}$$

Figure 8.1: $\mathcal{G}_{ldi}$: grammar with overhead for full rightpart verification.

middle of the rightpart, that is, if it is not at its left end. In this case, reduce actions cannot be decided on the lookahead. Accordingly, we calculate the set of actions which are inhibited from decision on the lookahead as follows:

$$R_L := \{i \mid [i\bullet, A \rightarrow \alpha \bullet \beta, \bullet a] \in q \text{ and } |\alpha| > 0\}$$

Consequently, all these items with action i pass on to the second step for stack symbol decision.

Reduction decision on the stack symbol is inhibited in a very similar way, although, now, since we are going to use precisely the next stack symbol, this may be the last symbol in the handle to reduce. So, if $|\alpha| = 1$ above we may already decide on the next stack symbol while still verifying the handle on top of the stack. Thus, we only inhibit reductions on the stack symbol if $|\alpha| > 1$, as follows:

$$R_S := \{i \mid [i\bullet, A \rightarrow \alpha \bullet \beta, \bullet a] \in q \text{ and } |\alpha| > 1\}$$

Examples

Consider for instance, grammar $\mathcal{G}_{ldi}$. Its corresponding item sets are shown in Figure 8.2, which were built according to the above item notation and procedure. For each state, the corresponding set of compatible actions is shown between brackets. Obviously, those states with a single action are states which did not exist in the automaton without part verification.

Both parsing tables, with and without rightpart verification are shown in Figures 8.3 and 8.4, respectively. The latter is almost an extension of the former, where, instead of deciding, if necessary, successive transitions are taken on stack symbols until the corresponding handle is fully verified. See, for instance, states 7 and 8 to complete rightpart for rule 4.

Nevertheless, the incorporation of full rightpart verification into the reverse parser occasionally introduces a minor overhead other decisions. Consider for instance sentence $\vdash dcb\dashv$. When the stack contains $\vdash dc$, and current lookahead is b, in the parser without rightpart verification, there is a transition on c from initial state to state 3, where, on stack symbol d, the decision to reduce using rule 7: $C \rightarrow \varepsilon$ is taken. In the parsing table with rightpart

State 1 $\{0, 1, 2, 3, 4, 5, 6, 7\}$

$[1\bullet, S'\to\vdash S\dashv\bullet, \bullet\dashv]$
$[2\bullet, S\to Aa\bullet, \bullet\dashv]$
$[3\bullet, S\to Bb\bullet, \bullet\dashv]$
$[4\bullet, S\to dcCb\bullet, \bullet\dashv]$
$[5\bullet, A\to edc\bullet, \bullet a]$
$[6\bullet, B\to fc\bullet, \bullet b]$
$[7\bullet, C\to\bullet, \bullet b]$
$[\bullet 0, S'\to\vdash S\bullet\dashv, \dashv\bullet]$
$[\bullet 0, S'\to\bullet\vdash S\dashv, \vdash\bullet]$
$[\bullet 0, S\to A\bullet a, a\bullet]$
$[\bullet 0, S\to B\bullet b, b\bullet]$
$[\bullet 0, S\to dcC\bullet b, b\bullet]$
$[\bullet 0, S\to d\bullet cCb, c\bullet]$
$[\bullet 0, S\to\bullet dcCb, d\bullet]$
$[\bullet 0, A\to ed\bullet c, c\bullet]$
$[\bullet 0, A\to e\bullet dc, d\bullet]$
$[\bullet 0, A\to\bullet edc, e\bullet]$
$[\bullet 0, B\to f\bullet c, c\bullet]$
$[\bullet 0, B\to\bullet fc, f\bullet]$
$[\bullet 7, S\to dc\bullet Cb, b\bullet]$
$[\bullet 0, S'\to\vdash\bullet S\dashv, d\bullet]$
$[\bullet 0, S\to\bullet Aa, e\bullet]$
$[\bullet 0, S\to\bullet Bb, f\bullet]$
$[\bullet 0, S'\to\vdash\bullet S\dashv, e\bullet]$
$[\bullet 0, S'\to\vdash\bullet S\dashv, f\bullet]$

State 5 $\{5, 6, 7\}$

$[5\bullet, A\to ed\bullet c, \bullet a]$
$[6\bullet, B\to f\bullet c, \bullet b]$
$[\bullet 7, S\to d\bullet cCb, b\bullet]$

State 6 $\{5, 7\}$

$[5\bullet, A\to e\bullet dc, \bullet a]$
$[\bullet 7, S\to\bullet dcCb, b\bullet]$
$[\bullet 7, S'\to\vdash\bullet S\dashv, b\bullet]$

State 4 $\{3, 4\}$

$[3\bullet, S\to B\bullet b, \bullet\dashv]$
$[4\bullet, S\to dcC\bullet b, \bullet\dashv]$

State 7 $\{4\}$

$[4\bullet, S\to dc\bullet Cb, \bullet\dashv]$

State 8 $\{4\}$

$[4\bullet, S\to d\bullet cCb, \bullet\dashv]$

State 3 $\{2\}$

$[2\bullet, S\to A\bullet a, \bullet\dashv]$

State 2 $\{1\}$

$[1\bullet, S'\to\vdash S\bullet\dashv, \bullet\dashv]$

State 9 $\{1\}$

$[1\bullet, S'\to\vdash\bullet S\dashv, \bullet\dashv]$

Figure 8.2: Item sets of discriminating reverse LR(1) automaton with full rightpart verification for grammar $\mathcal{G}_{ldi}$.

	$\dashv$	a	b	d	c	e	f	$\vdash$	$\dashv$	S	A	a	B	b	d	c	C	e	f
s1				0	0	0	0		s2	0	0	s3	0	s4		s5	0		
s2										s9									
s3											2								
s4													3				s7		
s5															s6				6
s6			7															5	
s7																s8			
s8															4				
s9								1											

Figure 8.3: Discriminating reverse LR(1) parsing table including rightpart verification for grammar $\mathcal{G}_{ldi}$.

	$\dashv$	a	b	d	c	e	f	$\vdash$	$\dashv$	S	A	a	B	b	d	c	C	e	f
s1				0	0	0	0		1	0	0	2	0	s2		s3	0		
s2													3				4		
s3		5													7				6

Figure 8.4: Discriminating reverse LR(1) parsing table without rightpart verification for grammar $\mathcal{G}_{ldi}$.

verification, however, there are two transitions on c, then d, and then reduction 7 is decided on lookahead b. We can trace down the origin of this effect if we look at the table of compatible actions for state 3 of the non-verifying automaton:

	a	b
d	$\{5\}$	$\{7\}$
f		$\{6\}$

It is clear that first a decision to reduce 5 is taken on lookahead a, and then both 6 and 7 can be decided on the next stack symbol. Actually, the table of compatibles actions (for state 5) for the verifying automaton is the same, but there reduction 5 cannot be decided, since the last symbol of rule 5's rightpart, e, would remain without verification. As a side effect, items for 5 and 7 must remain, and thus reduction 7 cannot be decided now. Fortunately, this problem is always solved in the next state, since then the lookahead suffices to decide, as in our example after transition on d to state 7.

The reader might be interested in comparing the discriminating reverse LR(1) parsing table with full rightpart verification for grammar $\mathcal{G}_{etf}$, which is shown in Figure 8.5, with the corresponding non-verifying parsing table shown in Figure 5.16 (page 93). The list of the corresponding item sets for the verifying automaton is shown in Appendix A.4.

	$\dashv$	$+$	$*$	$($	$)$	a	$\vdash$	$\dashv$	E	$+$	T	$*$	F	$($	$)$	a
s1				0		0		s2	0		s3		s4		s5	7
s2									s9							
s3			0				3			s8				3		
s4							5			5		s7		5		
s5									s6							
s6														6		
s7											4					
s8									2							
s9							1									

Figure 8.5: Discriminating reverse LR(1) with full rightpart verification for grammar $\mathcal{G}_{etf}$.

8.1.4 Algorithms

Figure 8.6 shows the generation algorithm of a discriminating reverse LR(1) parser including full rightpart verification, according to the above explanations. Note that R_L and R_S sets are used for inhibiting reductions on lookahead and stack symbol, respectively.

The parsing algorithm is shown in Figure 8.7. It changes very little from the non-verifying automaton. Now, since rightpart verification at the moment of reduction is incorporated into the parsing table, it is not necessary to include the explicit verification instructions.

8.2 Conflict resolution

An interesting, practical extension for the method would be to be able to accept ambiguous grammars as other parser generators do, for example yacc. This is a slippery issue, however, and the grammar designer must be very cautious, and should know very well what are the consequences of introducing ambiguities, because it will finally rely on him whether the produced parser will function properly or not, since in the case of ambiguous grammars, it is the designer who, directly or indirectly, indicates which option should the parser take in case of ambiguity. We do not pretend to investigate here the subject by itself, nor to speculate about the convenience of using one or another convention for disambiguating grammars. We just present how the method can incorporate a disambiguation mechanism, separating this from the decision of *which* of the ambiguous options should be preferred.

An additional note of warning is given, since LR methods in general, as ours, are not able to discriminate between conflicts originated in non-LR-ess or proper ambiguity of the grammar. If it were the former case, the parser implementor would be surprised to be warned "syntactic errors" from correct sentences, because the conflict-resolution mechanism might choose the wrong parsing action for a conflict originated in grammar non-LR-ness.

algorithm Generator of discriminating-reverse LR(1) parsing table
with full rightpart verification
input Reduced grammar $\mathcal{G}$
output Parsing table *ActTrans(state,kind_ of_ symbol,symbol)*,
or Error if the grammar is not LR(1)
function Closure(J)
  $I := J$
  repeat
    for $[i, A \rightarrow \bullet\alpha, \bullet a] \in I$ **do for** $B \rightarrow \beta A \gamma \in P'$ **do**
      if $a \in \text{First}(\gamma)$ **then** add $[\bullet i, B \rightarrow \beta \bullet A\gamma, a\bullet]$ to I
      else if $\varepsilon \in \text{First}(\gamma)$ and $a \in \text{Follow}(B)$ **then** add $[\bullet i, B \rightarrow \beta \bullet A\gamma, \bullet a]$ to I
    for $[i, A \rightarrow \bullet\alpha, a\bullet] \in I$ **do for** $B \rightarrow \beta A \gamma \in P'$ **do** add $[\bullet i, B \rightarrow \beta \bullet A\gamma, a\bullet]$ to I
  until no new additions to I
  return I
end Closure
begin
  $N_n := \{A \mid A \Rightarrow^+ \varepsilon\}$; $N_r := \{i \mid A \xrightarrow{i} \alpha$ and $\alpha \Rightarrow^* \varepsilon\}$
  Aut:=empty
  $K_0 := \{[i\bullet, A \rightarrow \alpha\bullet, \bullet a] \mid A \xrightarrow{i} \alpha \in P'$ and $a \in \text{Follow}(A)\} \cup$
    $\{[\bullet 0, A \rightarrow \alpha \bullet a\beta, a\bullet] \mid A \rightarrow \alpha a \beta \in P'\}$
  I_0 :=Closure(K_0); add I_0 to Aut
  for $I \in Aut$ **do**
    % non-LR(1) check
    if $[i, A \rightarrow \alpha \bullet \beta, a], [j, A \rightarrow \alpha \bullet \gamma, a] \in I,\ i \neq j$ **then** Error
    % decisions upon current lookahead
    for $a \in {V_T}'$ **do** $La(a) := \{i \mid [i, A \rightarrow \alpha \bullet \beta, a] \in I\}$
    $R_L := \{i \mid [i\bullet, A \rightarrow \alpha \bullet \beta, \bullet a] \in I$ and $|\alpha| > 0\}$
    $J := I$
    for $a \mid La(a) = \{i\}$ and $i \notin R_L$ **do**
      if $i = 0$ **then** $ActTrans\,(I$,lookahead,$a)$:=shift
      else $ActTrans(I$,lookahead,$a)$:=reduce(i)
      $J := J - \{[i, A \rightarrow \alpha \bullet \beta, a] \in J\}$
    % decisions upon next stack symbol
    for $X \in V'$ **do** $St(X) := \{i \mid [i, A \rightarrow \alpha X \bullet \beta, a] \in J\}$
    $R_S := \{i \mid [i\bullet, A \rightarrow \alpha \bullet \beta, \bullet a] \in J$ and $|\alpha| > 1\}$
    for $X \mid St(X) = \{i\}$ and $i \notin R_S$ and not ($X \in N_n$ and $i \in N_r$) **do**
      if $i = 0$ **then** $ActTrans(I$,stack,$X)$:=shift
      else $ActTrans\,(I$,stack,$X)$:=reduce(i)
      $J := J - \{[i, A \rightarrow \alpha X \bullet \beta, a] \in J\}$
    % state transitions upon next stack symbol
    for $X \mid [i, A \rightarrow \alpha X \bullet \beta, a] \in J$ **do**
      K :=Closure($\{[j, B \rightarrow \gamma \bullet X\delta, b] \mid [j, B \rightarrow \gamma X \bullet \delta, b] \in J\}$)
      add K to Aut; $ActTrans(I$,stack,$X)$:= goto(K)
end

Figure 8.6: Generation algorithm of discriminating reverse LR(1) parser including full rightpart verification.

```
algorithm Discriminating-reverse LR(1) parser;
input Parsing table ActTrans(state,kind_of_symbol,symbol)
      and input sequence
output Bottom-up parse of the input sequence,
       or Error if syntactically incorrect
begin
   Stack:=empty; push(⊢); read(a)
   repeat
      q := q0; p := top
      repeat
         at:=ActTrans(q,lookahead,a)
         if at=nil then at:=ActTrans(q,stack,Stack(p))
         case at of
            goto(q'): q := q'; p := p − 1
            shift: push(a); read(a)
            reduce(A -i-> α): if i = 1 then at:=accept
               else pop(α); push(A)
            nil: Error
      until at ≠ goto
   until at=accept
end
```

Figure 8.7: Algorithm of discriminating reverse LR(1) parser for parsing table including full rightpart verification.

Even worst, both kind of origin can be present at the same time.

Nonetheless, we must say that ambiguities can be found of practical interest in very particular cases found in programming languages, as in the well-known "dangling else" problem, which otherwise is somehow cumbersome to represent in an LR grammar, or the simplicity and efficiency of parsing of syntactically ambiguous expressions. Thus we shall content with proposing a method that works in general, and well for these cases in particular, instead of trying to propose a complicated, completely optimized mechanism.

As we have seen in a previous chapter, a non-LR(1) grammar —ambiguous or not— means, from the point of view of our parser, that for some sentences, there are at some point two different actions for the same stack-plus-lookahead configuration, since there are more than a single possible parse. This means that even after reading the whole stack, the decision cannot be taken from the information in the item themselves. We would like to promptly detect whether two items have the same continuation, however. In this way, instead of uselessly pursuing automaton construction, an external disambiguation mechanism would be called for.

8.2.1 Resolvable conflicts

Recall the definition of left-hand context language (page 109) for an item as the set of all possible continuations that can be found when exploring the legal stack from the current point of exploration, until the stack bottom.

The information we can ultimately use to discriminate between two items with different actions are their respective pairs (left-hand context language, lookahead). If the lookaheads are different, the two actions can be discriminated, at worst after having read the whole stack. Let us now suppose that the lookaheads are the same. Then it is clear that for LR(1) grammars, the left-hand context languages of two items with different actions are always disjoint, since otherwise there would be some stack for which it would be impossible to decide. According to this, in page 110 we defined that two items are in conflict when their left-hand context languages are not disjoint. In general, we cannot resolve this kind of conflicts, since in principle there can be cases (i.e., the non-common continuations) where the action is unique.

Let us first consider the case where the two left-hand context languages are equal for the items in conflict. Then it is impossible to discriminate between them, and so it would be useless to continue building the automaton for these two items, and thus it is the moment to apply conflict resolution by applying some decision mechanism, be it explicitly given by the designer in its grammar specification, or by applying some default convention implemented in the grammar generator. Is it the only possible case where resolution can be applied? Note that if the left-hand context language of the item with preferred action includes the other, then it is also possible to decide in favor of the former, because in the non-conflicting cases this will be also the parsing action to perform. Thus, we shall say that in both cases the two items are in *resolvable conflict*.

8.2.2 Non-preferred item elimination

Although the discriminating reverse automaton as defined for LR(1) grammars is pruned as early as the action to parse is uniquely determined, in the case of non-LR(1) grammars there are cases where the action cannot be decided, and the algorithm given in Section 5.5.1 produces the full top-to-bottom recognizer for a subset of the whole stack language. In general, bottom-of-stack states, which are reached after reading the $\vdash$ marker, contain only items of the form:

$$[i,\ S' \to \bullet \vdash S \dashv,\ a \bullet].$$

In a non-LR(1) grammar, an item like this can be in conflict with another item with the form

$$[j,\ S' \to \bullet \vdash S \dashv,\ a \bullet],\ i \neq j.$$

At this point it is clear that if the designer has specified that action i has precedence over action j, the conflict can be resolved. Conflict resolution can be easily integrated in the generation algorithm just by removing these non-preferred items from the state item set before the phase of computing its actions and transitions.

But, as it has been shown, a property of the discriminating reverse method is the possibility of strongly pruning the automaton as early as the action can be determined. Strictly speaking, for ambiguous grammars, in case of resolvable conflict we need not continue automaton construction, and the non-preferred items can simply be removed. In this way, we avoid to build the recognizing automaton for the stack sublanguage beginning from the state containing the resolvable conflict. Accordingly, when two items are *detected* to be in resolvable conflict, the non-preferred one is eliminated from the item set, and the construction procedure continues in exactly the same way. It may surprise that we can just eliminate an item from the set. Note that an item set is precisely a coded form of all the possibilities still in consideration, and thus it is perfectly valid to eliminate an item when one of them is no longer valid for some reason[2].

The practical problem with completely putting in practice this idea is that it would need a possibly costly computation of the relations between the left-hand context languages of items in conflict. Thus, a computationally cheap test is presented in the following section, which works quite well for the usual cases found in practice. Note that, although in some cases two items in resolvable conflict might passed undetected, this only means that the automaton is not pruned, but its construction is continued for that case, in the worst case until bottom-of-stack states, where, as it has been shown, conflicts can be always resolved if the designer provides the corresponding precedence relation. Note that in case of conflict at parsing time, the bottom of stack would be reached in the worst case, and then the parsing action is decided.

8.2.3 A test to detect resolvable conflicts

Two items with different action numbers i and j are detected in resolvable conflict when they have the form:

$$[i,\ A{\to}\alpha{\bullet}\beta,\ {\bullet}a]$$

$$[j,\ A{\to}\alpha{\bullet}\gamma,\ {\bullet}a]$$

or

$$[i,\ A{\to}\alpha{\bullet}\beta,\ a{\bullet}]$$

$$[j,\ A{\to}\alpha{\bullet}\gamma,\ a{\bullet}]$$

[2]The idea of removing an item from a conflicting pair was suggested to the author by J. Farré.

$$
\begin{array}{ll}
1: & S' \rightarrow \vdash S \dashv \\
2: & S \rightarrow \textbf{if } E \textbf{ then } S \textbf{ else } S \\
3: & S \rightarrow \textbf{if } E \textbf{ then } S \\
4: & S \rightarrow E \\
5: & E \rightarrow E + E \\
6: & E \rightarrow E * E \\
7: & E \rightarrow (E) \\
8: & E \rightarrow a
\end{array}
$$

Figure 8.8: $\mathcal{G}_{cie}$: ambiguous grammar for conditionals containing expressions.

Also, if the action i is to be preferred according to the external disambiguating procedure, then items

$$[i,\ A \rightarrow \alpha \bullet \beta,\ a \bullet]$$

$$[j,\ A \rightarrow \alpha \bullet \gamma,\ \bullet a]$$

will also be found in resolvable conflict, because in every continuation of the j item, the i item will be preferred, since its continuations cover those of the j item.

Finally, for case

$$[i,\ A \rightarrow \alpha \bullet \beta,\ \bullet a]$$

$$[j,\ A \rightarrow \alpha \bullet \gamma,\ a \bullet]$$

with preferred i, we verify whether

$$[i,\ A \rightarrow \bullet \delta,\ a \bullet] \in \mathcal{C}([i,\ A \rightarrow \bullet \alpha\beta,\ \bullet a])$$

since then it is clear that, after having read α the corresponding j item will be in resolvable conflict.

8.2.4 An example of ambiguous grammar for if-then-else with expression

To illustrate the workings of this technique, let us use grammar $\mathcal{G}_{cie}$ shown in Figure 8.8, containing two typical cases of ambiguity, one for the well-known dangling else problem of **if-then-else** constructs, and the other for a compact, ambiguous syntax for arithmetic expressions. The former case of ambiguity is solved by associating the **else** to the inner **if**, and the latter is solved by giving preference to the multiplication over addition, and by associating a succession of additions (or multiplications) to the left as it is usual. The grammar designer will have to specify these preferences in some manner, depending on the parser tool conventions. Thus, we shall suppose that these preferences are known to the parser generator. Let us then see how they are implemented in order to select the corresponding items and actions.

The whole listing of state items is shown in the Appendix. From it, we can see that in the initial state the generator finds the following two items to be in conflict:

$$[3, S \to \textbf{if}\ E\ \textbf{then}\ S\,\bullet, \bullet\textbf{else}]$$
$$[0, S \to \textbf{if}\ E\ \textbf{then}\ S\bullet\textbf{else}\ S, \textbf{else}\bullet]$$

Here the preferred action is to shift (action 0) in order to associate the **else** to the inner **if**, instead of reducing (action 3). Thus, this is a case of resolvable conflict, since the left-hand context language of the preferred action covers the left-hand context language of the non preferred action. Accordingly, the first item is removed from item set of initial state.

From the initial state, a transition on top of stack symbol E is generated to state 3, where the following pairs of items in conflict are successively found by the generator:

$$[5, E \to E + \bullet E, \bullet +]$$
$$[0, E \to E + \bullet E, + \bullet]$$

$$[5, E \to E + \bullet E, \bullet *]$$
$$[0, E \to E + \bullet E, * \bullet]$$

$$[6, E \to E * \bullet E, \bullet +]$$
$$[0, E \to E * \bullet E, + \bullet]$$

$$[6, E \to E * \bullet E, \bullet *]$$
$$[0, E \to E * \bullet E, * \bullet]$$

In the first case, to reduce with rule 5: $E \to E + E$ is preferred in order to associate additions to the left, instead of a shift (action 0). The situation is now the opposite to the case of **else**. Here it is the preferred-action left-hand context language which is included in the non-preferred-action left-hand context language. But closure $\mathcal{C}(\{[5, E \to \bullet E + E, \bullet +]\})$ results in the following set of items:

$$[5, E \to \bullet E + E, + \bullet]$$
$$[5, S \to \textbf{if}\bullet E\ \textbf{then}\ S\ \textbf{else}\ S, + \bullet]$$
$$[5, S \to \bullet E, + \bullet]$$
$$[5, E \to E + \bullet E, + \bullet]$$
$$[5, E \to E * \bullet E, + \bullet]$$
$$[5, E \to (\bullet E), + \bullet]$$
$$[5, S' \to \vdash \bullet S \dashv, + \bullet]$$
$$[5, S \to \textbf{if}\ E\ \textbf{then}\bullet S\ \textbf{else}\ S, + \bullet]$$
$$[5, S \to \textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\bullet S, + \bullet]$$

	$\dashv$	*if*	*then*	*else*	$+$	$*$	(	)	a	$\vdash$	$\dashv$	S	*if*	E	*then*	*else*	$+$	$*$	(	)	a
s1		0					0		0		1	s2		s3						7	8
s2										0					s8	2					
s3										s5			0		s7	s6	s4	6	0		
s4	5		5	5	5	0		5													
s5	4				0	0															
s6	4			4	0	0															
s7	4			4	0	0															
s8	3			0																	

Figure 8.9: Discriminating reverse LR(1) parsing table including conflict resolution for grammar $\mathcal{G}_{cie}$.

We see that, effectively, $[5, E \to \bullet E + E, + \bullet]$ is in the set, and thus both left-hand context languages for $[5, E \to E + \bullet E, \bullet +]$ and $[0, E \to E + \bullet E, + \bullet]$ are equal, hence we conclude that we are in resolvable conflict. Accordingly, the generator deletes the former item from the set of state 3.

The second conflict in state 3 is again between rules 5 and 0. This time the shift is preferred in front of lookahead $*$ in order to give preference to the multiplication over the addition. Since the preferred-item language covers the non preferred one, we are again in resolvable conflict.

The two remaining conflicts of state 3 are solved by giving preference to reducing using rule 6: $E \to E * E$, because of preference over the addition, or because left associativity, respectively. In both cases the preferred-item left-hand context language is included in the non-preferred-item left-hand context language, but the closure computation indicates that actually both languages are equal, and thus both conflicts are resolvable, thus resulting in deletion of the shift items.

No more conflicts are found during automaton construction. The resulting discriminating reverse LR(1) parsing table is shown in Figure 8.9.

8.2.5 Generation algorithm

Figure 8.10 shows the corresponding algorithm for parser generation including conflict resolution. Here, instead of announcing an error and giving up when a non-LR(1)-ness is detected because of a conflict, the generator tries before to solve each pair of items in conflict by means of the method shown in the previous sections. It makes use of a boolean function $\mathit{Prec}(i, j)$ which tells whether item with action i has precedence over item with action j, according to preferences, normally specified by the grammar designer. Note that in case of a conflict which needs a closure, but which does not pass this test, then an error might be announced in case we are not interested in continuing automaton construction for these items in conflict until their successor items are in resolvable conflict. Also, function *Prec* might announce an error if it has not been specified a preference for the current conflict under process.

The parsing algorithm itself does not change from the normal case, since conflicts are resolved in the generation phase and thus their resolution is already incorpored into the parsing table.

8.3 Variations on the decision mechanism

We may consider several variants for the deferred-lookahead decision scheme at each state presented in Section 5.4.1. The one shown there seems to be a reasonable compromise, since it is straightforward and rather easy to understand and implement, results are quite good in terms of table size, and stack exploration is minimal. Nevertheless, the resulting decision table for each state may be redundant in a few cases. In this section we shall present first how to avoid easily this redundancy, while preserving the order of tests (first lookahead, then stack symbol). Then we shall show the possibility of optimizing the decision process at each state to reduce the size in number of entries and states, at the cost of parsing time, preserving the order of test, and if no order is predetermined for the different tests of possible values of lookahead and stack symbols.

In more general terms, the approach presented in Chapter 5 is only one amongst many possibilities of discriminating-reverse parsing, each one representing a different compromise between space and time efficiency. We mentioned there the initial possibility of beginning exploration from the lookahead, that we discarded in principle, but that J. Farré has found that may also be practical if the resulting sub-automata are merged when possible. This has the advantage that the lookahead is explored only once, and that the parser is conceptually simpler, more robust, and preserves minimal exploration depth. The reverse is also possible, to use the lookahead at the end of the exploration, not necessarily at the stack bottom, but for instance when we find that the stack is useless (because the left-hand context languages coincide) and then the lookahead is needed. A better approach is also shown in this section, by using the lookahead as a last step in the exploration when we know that it will suffice to take the decision. This variant has the obvious potential advantage that the lookahead is tested only once, if any, during each exploration, but this is not always minimal.

The section finishes with two further variants, which may reveal of practical interest. In the first one we shall explore the possibility of deciding, not to which rule to reduce, but to which nonterminal to reduce, what reduces the number of cases, and thus should reduce automaton size. On the other hand, the second variant enlarges the number of actions involved, by allowing to verify the compatibility of the incoming lookahead, and then producing an immediate error detection (and thus correct-prefix, obviously) parser.

```
algorithm Generator (with conflict resolution) of discriminating-reverse LR(1) parsing table
input Reduced grammar 𝒢 and precedence table Prec(action,action)
output Parsing table ActTrans(state,kind_of_symbol,symbol),
        or Error if unespecified precedence for non-LR(1) conflict
function Closure(J)
    I := J
    repeat
        for [i, A→•α, •a] ∈ I do for B→βAγ ∈ P' do
            if a ∈ First(γ) then add [i, B→β•Aγ, a•] to I
            else if ε ∈ First(γ) and a ∈ Follow(B) then add [i, B→β•Aγ, •a] to I
        for [i, A→•α, a•] ∈ I do for B→βAγ ∈ P' do add [i, B→β•Aγ, a•] to I
    until no new additions to I
    return I
end Closure
begin
    N_n:={A | A⇒⁺ε}; N_r:={i | A -i-> α and α⇒*ε}
    Aut:=empty
    K_0 :={[i, A→α•, •a] | A -i-> α ∈ P' and a ∈ Follow(A)} ∪
          {[0, A→α•aβ, a•] | A→αaβ ∈ P'}
    I_0 :=Closure(K_0); add I_0 to Aut
    for I ∈ Aut do
        % non-LR(1) conflicts
        J := I
        for [j, A→α•γ, a•] ∈ J do
            for [i, A→α•β, a•] ∈ J | i ≠ j do
                if Prec(i, j) then J := J − {[j, A→α•φ, a]}
            for [i, A→α•β, •a] ∈ J | i ≠ j and [i, A→•δ, a•] ∈ Closure([i, A→•αβ, •a]) do
                if Prec(i, j) then J := J − {[j, A→α•φ, a]}% optionally else Error
        for [j, A→α•β, •a] ∈ J do for [i, A→α•β, a] ∈ J | i ≠ j do
            if Prec(i, j) then J := J − {[j, A→α•φ, •a]}
        % decisions upon current lookahead
        for a ∈ V_T' do La(a) := {i | [i, A→α•β, a] ∈ J}
        for a | La(a) = {i} do
            if i = 0 then ActTrans (I,lookahead,a):=shift
            else ActTrans(I,lookahead,a):=reduce(i)
            J := J − {[i, A→α•β, a] ∈ J}
        % decisions upon next stack symbol
        for X ∈ V' do St(X) := {i | [i, A→αX•β, a] ∈ J}
        for X | St(X) = {i} and  not (X ∈ N_n and i ∈ N_r) do
            if i = 0 then ActTrans(I,stack,X):=shift
            else ActTrans (I,stack,X):=reduce(i)
            J := J − {[i, A→αX•β, a] ∈ J}
        % state transitions upon next stack symbol
        for X | [i, A→αX•β, a] ∈ J do
            K :=Closure({[j, B→γ•Xδ, b] | [j, B→γX•δ, b] ∈ J})
            add K to Aut; ActTrans(I,stack,X) := goto(K)
end
```

Figure 8.10: Generation algorithm of discriminating reverse LR(1) parser with conflict resolution.

```
1: S' -> ^ S $
2: S -> G A b
3: S -> G B c
4: S -> G B e
5: S -> H C a
6: S -> H C b
7: S -> H C c
8: S -> H D d
9: G -> g
10: H -> h
11: A -> f
12: B -> f
13: C -> f
14: D -> f
```

Figure 8.11: $\mathcal{G}_{dec}$: example grammar for optimal decision order.

8.3.1 Redundant entry elimination

Consider the grammar of Figure 8.11. The discriminating reverse automaton builds the states and corresponding item list shown in Figure 8.12. The simple two-step decision construction algorithm produces in state 4 the list of actions shown, first by testing the lookahead, and if the current lookahead does not belong to set $\{\mathtt{e}, \mathtt{a}, \mathtt{d}\}$, then the next lower stack symbol is tested to take the decision of reducing using rule 13, or to take a transition to state 5, which is also shown.

Note that in Figure 8.12 action `LA(a) = Reduce 13` is not actually needed, i.e., this entry can be simply removed, since that action corresponds to item

```
13 : S -> H . C a , a .
```

and thus the same action would be performed at the stack test `ST(H) = Reduce 13`. That is, in some cases, table size can be improved even preserving the order of tests in two steps, first lookahead, then stack symbol. Although the improvement seems to be quite marginal in general, it might be interesting to consider this kind of optimization if efficiently implemented.

In summary, the construction process from a state table stack-symbol×lookahead containing the compatible actions would be as follows:

1. Mark lookahead colums with a single action.
2. (a) Compute the union of the action-sets for each row without considering columns marked in step 1.
   (b) Build stack-symbol decisions for rows with single-action unions, and transitions for rows with several-action unions.

```
State 4  [ 11 12 13 14 ]
 11 : A -> . f , . b
 12 : B -> . f , . c
 12 : B -> . f , . e
 13 : C -> . f , . b
 13 : C -> . f , . c
 13 : C -> . f , . a
 14 : D -> . f , . d
 11 : S -> G . A b , b .
 12 : S -> G . B c , c .
 12 : S -> G . B e , e .
 13 : S -> H . C b , b .
 13 : S -> H . C c , c .
 13 : S -> H . C a , a .
 14 : S -> H . D d , d .

LA(e) = Reduce 12
LA(a) = Reduce 13
LA(d) = Reduce 14
ST(H) = Reduce 13
ST(G) = Go To 5


State 5  [ 11 12 ]
 11 : S -> . G A b , b .
 12 : S -> . G B c , c .
 11 : S' -> ^ . S $ , b .
 12 : S' -> ^ . S $ , c .

LA(b) = Reduce 11
LA(c) = Reduce 12
```

Figure 8.12: Discriminating reverse states for grammar $\mathcal{G}_{dec}$with corresponding actions produced by the simple two-step decision algorithm.

(c) Remove the action-set entries included in its corresponding row union.

3. Build lookahead decisions.

Note that the automaton will behave slightly differently in case of error. It might be possible in principle that if the stack symbol were not "`H`" because of an error, then in the first case reduction 13 would be performed in front of "`a`" as lookahead symbol, while in the second case an error would be detected.

The above consideration induces to think that it is possibly more solid to decide on the stack symbol, if possible, and to use the lookahead for decision only if necessary to decide in each state. In effect, we should be in general more confident in basing a decision on a stack symbol, since it has much more possibilities of being correct, for it has appeared after decisions taken in a wider context, than to use a single lookahead terminal. The following section studies this possibility.

8.3.2 Preference for stack-symbol decisions

In this variant the discriminating reverse parser will use the lookahead to take a decision (if this is possible) only in those cases that the next stack symbol does not suffice to decide in the current state. The order of consultations remains the same, first the lookahead section for a decision, and if this fails, then the stack-symbol section, for a decision, a transition or an error detection.

Hence, in this variant one more consultation (corresponding to the stack symbol) to the parsing table may be performed, even though in the standard construction the decision would have been taken in the first consultation (on the lookahead section).

The automaton states and items do not change, and the parser continues to be linear, since at most one more test is done for a linear number of decisions.

A (suboptimal) construction of the parsing table which reflects this idea is as follows:

1. Mark stack-symbol rows with a single action.

2. Build lookahead decisions for columns with a single action and some nonempty entry in some row not marked in step 1. Remove these columns.

3. On the remaining table, build stack symbol decisions for rows with a single action, and transitions for rows with several actions.

Note that we may have redundancy as before. Refer to Figure 8.13, which reflects the different compatible actions for state 4 of Figure 8.12. And there

	a	b	c	d	e
G		{11}	{12}		{12}
H	{13}	{13}	{13}	{14}	

Figure 8.13: Action set table for state 4 of Figure 8.12.

may be further possibilities for optimization. If our table has, for instance, a single row with the same configuration as the lower row of Figure 8.13, it is evident that we could first decide on lookahead `d` and then on stack symbol `H`.

With this idea, at the cost of an additional test in some cases, we may significatively reduce the overall number of entries, as in typical cases when some decision can be taken on several lookahead values, but can also be decided on a single stack-symbol value, thus reducing from several to one the number of entries.

Please note again that decisions are the same, and the only difference lies on whether the lookahead or the next stack symbol is used to take them. As we said before, it seems more robust to use the stack, so this may be a criterium for using this variant.

8.3.3 Test order optimization

If any order is allowed in the generated list of tests to be performed by the parser at each state, the number of tests can be further optimized in some cases. In the same case of Figure 8.12 the following optimal order can be found for actions of state 4:

```
LA(d) = Reduce 14
ST(H) = Reduce 13
LA(b) = Reduce 11
ST(G) = Reduce 12
```

This can be easily seen in Figure 8.13. According to that table, if the first decision is `LA(d) = Reduce 14` and if this test fails because the current lookahead symbol is not a “`d`”, then it is possible to decide that, if the stack symbol is a “`H`” then the only possible action is to reduce using rule 13, for rule 14 is now discarded. In the same vein, if the test on stack “`H`” fails, then reductions 11 and 12 are now opened to decision. Note that these final tests can be made all of them on the current lookahead, but that it would be optimal to first test lookahead “`b`”, and then to use *stack* symbol “`G`”.

This optimization not only reduces, although still probably marginally in general, the number of actions, but also can diminish in some cases the number of states, as it is the case here. Nevertheless, it is probably too costly in computational terms to be used as the decision generation algorithm of choice in general.

The decision order optimization has also consequences on the possibilities of subsequent global parsing table optimization, for the order of test can be critical. It also influences the implementation (and parser efficiency), which in cases like this is necessarily a list of tests, while in the two-step cases an indexed access to table is possible.

Note also that, as usual, behavior in front of syntactic error can vary in some cases depending of them decision method and table implementation.

Finally note that any configuration of state action table can happen. Actually, it is straightforward to build a grammar producing a state with the desired configuration, as was actually made for grammar $\mathcal{G}_{dec}$.

8.3.4 Lookahead postponement

Let us study here the possibility of reducing the number of table consultations if we completely defer the use of the lookahead until it clear that, if tested, will produce a decision (or detect a syntactic error). That is, the parser will only use the stack, much like a reverse LR(0) parser, but will also use the lookahead as a final step. In this way, we may expect to reduce the number of parsing steps to something more than half the "standard" method shown in Chapter 5.

For that purpose, a construction process from a state table stack-symbol×lookahead containing the compatible actions would be as follows:

1. Build stack-symbol decisions for rows with a single action. Remove these rows.

2. Mark lookahead columns with a single action.

3. Build stack-symbol transitions for stack-symbol rows containing some nonempty entry in some column not marked in step 2. Remove these rows.

4. Build lookahead decisions on the remaining table.

Once the parsing table has been built according to this procedure, for each action decision the discriminating reverse parser will only use the stack-symbol section of the table to perform transitions or to take a decision (which of course would end exploration). Only if an empty entry in the stack-symbol section is entered at some state, then the lookahead section for this same state is entered, which results in a decision or the detection of a syntactic error. Since for each state the parser only consults one section of the table (except in a last state, where both sections may be consulted), there is a potential gain in speed in relation to our standard construction, where the situation is the opposite (both sections are consulted, except in a last state where the lookahead section, or both, may be consulted).

However, there is a special effect to note with this mechanism. In step 3 above, items whose action could be decided on the lookahead, are not actually decided in this state when there is one or more different actions in the same stack-symbol row which cannot be decided on the lookahead (nor on the stack-symbol, of course). We may see a simple example of this effect with action tables of Figure 5.15 in page 88. In state 1, effectively, we can decide on the stack symbol for most rows without the need to consult the lookahead. Then transitions to the same states 2 and 3 are calculated, whose tables are identical to those shown in the referred figure. In state 3 everything goes well and the stack symbol always decides. But in state 2, step 1 above cannot decide, step 2 marks the * column, but step 3 needs to make transitions to three new states, carrying in these transitions all the items with lookahead * —because the decision to make each such transition is taken without looking at the lookahead, and, consequently, all the state items must be considered in the transitions. Fortunately, those new states may decide on the stack, or certainly on the lookahead. But if in our original action table for state 2 we had for instance $\{2,3\}$ instead of $\{2\}$ somewhere, the shift could not be decided until these other actions are decided, which may happen some states later. As we see, in these cases, depth of exploration is not minimal, and the automaton may be enlarged. However, this approach could also permit avoiding to construct some states that our original procedure could not, when to decide first on the stack permits to decide then on the lookahead, but not the converse .

This effect, and the fact that exploration lengths are typically very small for practical grammars, as we have seen, should make us relativize the expectation in speed gain of this variant. Moreover, since our exploration may be enlarged in some cases, it is unclear whether the linearity with input length is preserved.

8.3.5 Nonterminal decision

A possible variant of the method in the context of full rightpart verification is to decide to which nonterminal the reduction must be done instead of deciding which rule must be applied for the reduction. It is evident that, since the possible options to decide amongst are fewer, the discriminating automaton should naturally be smaller in general, although in principle it does seem that there would be in normal practice a big improvement in size for this reason, since it would only affect to the decisions amongst rules with the same leftpart. Of course, in order to implement this variant of the method, the full rightpart verification should finally tell which of the rightparts is to be actually reduced. Finally, there is a curious inconvenience with this variant, for if the decision is taken *after* having read the whole rightpart during the reverse exploration, it seems absurd to have to decide which rightpart. Depending on the way in which the full rightpart verification is implemented, this inconvenience can be overcome. For instance, if the verification phase is not done within the automaton itself, but is separately implemented, normally there should be a pointer from the discriminating automaton decision state to the point in this second verification phase where the exploration of the rest of the rightpart is to begin, which in the end signals the rule to reduce. So, every discriminating automaton decision state would signal where to continue exploration, or the action itself.

In conclusion, the flexibility of the method allows be to adapted to the implementation. Depending of how it is done, it might be practical and more efficient to discriminate amongst nonterminals or rules to reduce.

8.3.6 Immediate error detection

Depending on the error recovery technique used, the correct-prefix property or the immediate error detection may be important. As shown in precedent chapters, our method does not guarantee a correct stack for syntactically erroneous sentences. Here we propose the possibility of developing a variant for building discriminating reverse LR parsers with the immediate error detection property.

To facilitate the exposition, we shall consider the LR(1) case, which has the most practical interest in principle, but it can be readily generalized to the general LR(k) case.

First, let us suppose that we have a legal stack-plus-lookahead. As far as the discriminating reverse parser performs reductions only, the resulting stack-plus-lookahead continues to be legal, since the correct parsing action is always performed, because it has found that action to be the only possible legal one in the current stack-plus-lookahead configuration, once it has explored some stack suffix (and maybe used the lookahead). Now, let us consider that the parser decides to perform a shift. Again, it is the (only) correct action, and thus the stack is still legal, but the new lookahead may —due to a syntactic error— render illegal the new stack-plus-lookahead configuration. However, if we verify in some way its compatibility, then we can guarantee that it is still legal after shifting also. Hence, in these conditions, the parser will always have a legal stack-plus-lookahead, and for erroneous inputs will have the immediate error detection property.

In order to integrate this lookahead compatibility verification in our method, we propose to use in our construction, instead of our single shift action, a set of shift-verify(b) actions. When the parser finds one of such actions, it first shifts the current lookahead, and then verifies that the new incoming lookahead symbol is b. Now, instead of discriminating amongst the different reduce(r) and a single shift action, the parser has to discriminate amongst the different reduce(r) and shift-verify(b) actions. For this purpose, we use LR(2) reverse items[3] such that the last terminal in the lookahead is indeed the new lookahead b to verify. For the rest, our discriminating reverse LR(1) automaton is built much like usually, although we must specifically consider discrimination amongst the different shift actions. A straightforward (suboptimal) solution is the following. When in a same state all the shift-action items having a as the first terminal in the lookahead are of the form

$$[0\text{-}b_i, A \rightarrow \alpha \bullet \beta_i, ab_i \bullet],$$

with different b_i, then (and only then) all of these b_i are equally acceptable, because the left-hand context languages for all these items are the same, as we know. Accordingly, the parser has to shift and then check whether the new lookahead is in $\{b_i\}$, detecting an error otherwise. This may be simply incorporated into the parsing table as a new verification row which is consulted once the new lookahead has been read.

The need to start parsing with a legal stack-plus-lookahead is easily solved with an initially empty parsing stack, and by considering that the first input symbol is the begin-of-file marker $\vdash$.

Since with this variant our parser must discriminate further than a simple decision to shift, up to a point in the stack where a set of lookaheads are equally acceptable, deeper explorations will be needed in some cases. It is unclear the impact of this additional work on parsing efficiency, specially whether parsing time linearity on the input length is preserved. Although it is reasonable to expect significantly larger parsing tables with this variant, the fact of having a legal stack-plus-lookahead opens new possibilities for compactation. For instance, only our new verification rows contain error entries, and all the original error entries become in fact inessential, since they are never consulted.

[3] LR($k+1$) reverse items, for the general LR(k) case.

Chapter 9

Conclusions

The *left-to-right rightmost* or LR technique, invented by Donald Knuth when I had the same age as my son has now, appeared very powerful from both the theoretical and practical points of view. Unfortunately, the produced automata, as Knuth anticipated, are too large for grammars describing practical programming languages. Since then, research has concentrated mainly on developing efficient techniques to cope with the size problem. We have analyzed some of the most significant approaches, which, however, have not been widely considered in practice. Instead, it is a somehow deceiving compromise, LALR(1), which has been widely accepted, at the cost of reducing the parsing power to something less than LR, resulting in acceptable automata sizes. LALR(1) is used in tools as widely used as `yacc` and all its descendants, and served as well as inspiration to other parser generators.

We have shown, however, that `yacc` and similar tools have several practical drawbacks. There are programming constructs, typically constructs with similar syntaxes, the interpretation of which depends on the current context, which cannot be accepted unless the parser is full LR(1). Even some recursive-descent grammars are not acceptable by LALR(1). The introduction of ambiguity for some constructs can be also seen as a way to reduce size, but in this case the parser generator does not provides any help to the user, thus it is error prone and cannot be considered a satisfactory solution.

Other parsing techniques are more limited, for example the top-down parsers or the bounded-context bottom-up parsers. The latter, which are amongst the earliest parsing techniques, were based on the evidence that the most important source of parsing information is around the top of the stack. It is precisely this evidence, in combination with the notion of item-set state stolen from Knuth, which eventually gave place to our discriminating reverse approach.

In this thesis, we have shown that, as the mistaken sentence in the first printing of the "dragon book" correctly says, it is possible to build a reverse automaton which, reading the stack from its top, can decide which is the next shift-reduce action to perform. We have shown how to build different kinds of these automata in order to recognize the languages for some actions, and how to combine them in order to recognize the whole stack while carrying the compatible-action information. We have finally shown how, by pruning these automata when the compatible actions reduce to one, we only need to explore a minimal stack suffix in order to decide the next parsing action, resulting in the same power as direct LR(0), LR(1), and the general case LR(k). These automata have been proved to be correct.

We have shown the algorithms used to build these automata (and the corresponding parsing tables), and to detect during construction whether the grammar belongs to the corresponding class. We have put special attention on the practical case of LR(1), by developing a deferred-lookahead approach to the decision mechanism in each state. We have also shown how to cut, with negligible impact on table size, the possibility of empty-reduction loops for some grammars in the presence of particular syntactically erroneous inputs.

We have developed an experimental tool to implement our method for the reverse and direct LR(1) parsers and subsets, with the possibility of automatic generation of nontrivial random sentences, which has been very useful during the preparation of this thesis, in the production and evaluation of parsers, and has also proven to be useful for teaching purposes.

We have studied the efficiency of our method, through a study of cases and through practical experimentation, with the help of our implementation. Our practical results have also been independently verified for large practical programming language grammars, through a different, efficient implementation by J. Farré. We conclude that for practical grammars the size of our discriminating reverse LR(1) parsers are usually between half and one third the size of those generated by the less powerful LALR(1)-based tools. We also conclude that the theoretical disadvantage of stack suffix exploration is negligible in practice, since the average length of exploration is very stable for practical grammars, around 2 topmost stack symbols.

We have shown the correctness of our method for the general LR(k) case, and formally proved that the total sum of exploration steps during parsing is linear with the length of the input to parse, be it correct or erroneous. Contrarily to direct LR(k) parsers, which present a combinatorial explosion of states as k increases, our LR(k) parsers do not grow, or grow only slightly, if the grammar is not LR(k). This mainly theoretical result may be of practical interest for some programming constructs, as the recent tendency in the construction of generation tools demonstrates.

We have also developed several extensions to our method. In the first place, we have shown the algorithm and implemented a variant to integrate, through inhibition of the corresponding reducing actions, the verification of remaining prefixes of handles when the parser can decide to reduce before they have been completely scanned. We have also shown the algorithms and implemented the possibility of accepting ambiguous grammars with user-defined disambiguation rules, through the introduction of the notion of resolvable conflict to remove the items for the non-preferred action. Finally, we have shown several possibilities of optimization of the order in the decision mechanism, and further variants to decide to which nonterminal to reduce, and to discriminate specific shift actions which also predict the acceptable values of the incoming new lookahead, resulting in immediate error detection parsers.

9.1 Further research

We may mention further interesting areas of research around our parsing method. This is our "to-do list":

- As with every other parsing method, in order to incorporate ours into a practical parser generation tool, an adapted error-processing method is absolutely needed. Work is in progress in this area by R.-O. Cayot (University of Nice-Sophia Antipolis). For the same reason, it is interesting to study the facilities offered by our method to incorporate user-defined semantic actions and the automatic evaluation of attributes.

- Another interesting aspect is to develop efficient techniques for the implementation of the generation and parsing phases, and for the optimization of parsing tables. J. Farré. has worked also in this area. Other interesting possibilities to explore are the elimination of unit reductions, and the efficient implementation of k-lookaheads for LR(k) parsing.

- It would also be interesting to do more precise comparisons of results with other efficient LR parsers as Pager's or Seité's, although as we have seen, these cannot be better than LALR(1), while our parser improves over it. Some work has already been done in this line under our supervision by A. Reina.

- To implement some variants of the method which may improve speed, and to obtain experimental results to evaluate them, is also under way under our supervision.

- It would be of interest to formally prove upper bounds for the size of our parsers, and also to prove whether some variants preserve linearity with input length.

- To explore another approach to discrimination, such as the possibility of controlling the automaton growth when beginning exploration from the lookahead, or of enlarging the parsing power beyond LR(k) using our discriminating reverse parser as the basic mechanism, is being pursued by J. Farré (University of Nice-Sophia Antipolis).

- And, finally, to explore the feasibility of using our parser for extended BNF, or as the basic technique with Tomita's method, or for incremental generation and incremental parsing.

Thus it seems that this novel idea of discrimination has very rich and various consequences, and can be the opening of a large research area. It is encouraging to learn that the domain of syntax analysis is still far from being fully explored, contrarily to the common opinion of the last ten or twenty years.

Appendix A

Listings for Generated Automata

This Appendix shows several listings produced by our implementation for several of the automata studied in the main document. The implementation is describen in Section 6.3, and a small manual is shown in the Appendix B.

A.1 Full reverse recognizing automaton for grammar $\mathcal{G}_{exp}$

This listing produced by the implementation for grammar $\mathcal{G}_{exp}$ (see Figure 4.2 in page 42) shows the item sets and the actions and transitions for all the states. It corresponds to the full reverse recognizing automaton shown in Figure 4.10. Note that some state numbers are shown in that figure corresponding to the automaton generated in non-recognizing mode, and thus do not correspond to the state numbering shown here.

```
LR(1) discriminating reverse automaton experimental tool (v7.11.0 12-dec-1996)

Invoked:  /usr/users/fortes/aut/aut gr -gf exp.gra -g8 -g9 -dsaig

Input grammar statistics:
 Number of rules = 4
 Average length of rightparts = 2.00
 Average number of terminal symbols per rightpart = 1.00
 Vocabulary:
  Number of terminal symbols = 4
  Number of nonterminal symbols = 2

1: S' -> ^ S $
2: S -> S - T
3: S -> T
4: T -> ( S )
5: T -> a

Discriminating reverse LR(0) automaton
```

```
State 1  [ 0 1 2 3 4 5 ]
 1 : S' -> ^ S $ .
 2 : S -> S - T .
 3 : S -> T .
 4 : T -> ( S ) .
 5 : T -> a .
 0 : S' -> ^ S . $
 0 : S' -> . ^ S $
 0 : S -> S . - T
 0 : T -> ( S . )
 0 : T -> . ( S )
 0 : S -> S - . T
 0 : S -> . T
 0 : S' -> ^ . S $
 0 : T -> ( . S )

ST($) = Go To 2
ST(T) = Go To 3
ST()) = Go To 4
ST(a) = Go To 5
ST(S) = Go To 6
ST(-) = Go To 7
ST(^) = Go To 8
ST(() = Go To 9

State 9  [ 0 ]
 0 : T -> . ( S )
 0 : S -> S - . T
 0 : S -> . T
 0 : S' -> ^ . S $
 0 : T -> ( . S )

ST(-) = Go To 7
ST(^) = Go To 8
ST(() = Go To 9

State 8  [ 0 ]
 0 : S' -> . ^ S $


State 7  [ 0 ]
 0 : S -> S . - T

ST(S) = Go To 6

State 6  [ 0 ]
 0 : S' -> ^ . S $
 0 : S -> . S - T
 0 : T -> ( . S )

ST(^) = Go To 8
ST(() = Go To 9

State 5  [ 5 ]
 5 : T -> . a
 5 : S -> S - . T
 5 : S -> . T
 5 : S' -> ^ . S $
 5 : T -> ( . S )

ST(-) = Go To 10
ST(^) = Go To 11
```

```
ST(() = Go To 5

State 11  [ 5 ]
 5 : S' -> . ^ S $


State 10  [ 5 ]
 5 : S -> S . - T

ST(S) = Go To 12

State 12  [ 5 ]
 5 : S -> . S - T
 5 : S' -> ^ . S $
 5 : T -> ( . S )

ST(^) = Go To 11
ST(() = Go To 5

State 4  [ 4 ]
 4 : T -> ( S . )

ST(S) = Go To 13

State 13  [ 4 ]
 4 : T -> ( . S )

ST(() = Go To 14

State 14  [ 4 ]
 4 : T -> . ( S )
 4 : S -> S - . T
 4 : S -> . T
 4 : S' -> ^ . S $
 4 : T -> ( . S )

ST(-) = Go To 15
ST(^) = Go To 16
ST(() = Go To 14

State 16  [ 4 ]
 4 : S' -> . ^ S $


State 15  [ 4 ]
 4 : S -> S . - T

ST(S) = Go To 17

State 17  [ 4 ]
 4 : S -> . S - T
 4 : S' -> ^ . S $
 4 : T -> ( . S )

ST(^) = Go To 16
ST(() = Go To 14

State 3  [ 2 3 ]
 2 : S -> S - . T
 3 : S -> . T
 3 : S' -> ^ . S $
 3 : T -> ( . S )
```

```
ST(-) = Go To 18
ST(^) = Go To 19
ST(() = Go To 20

State 20  [ 3 ]
 3 : T -> . ( S )
 3 : S -> S - . T
 3 : S -> . T
 3 : S' -> ^ . S $
 3 : T -> ( . S )

ST(-) = Go To 21
ST(^) = Go To 19
ST(() = Go To 20

State 21  [ 3 ]
 3 : S -> S . - T

ST(S) = Go To 22

State 22  [ 3 ]
 3 : S -> . S - T
 3 : S' -> ^ . S $
 3 : T -> ( . S )

ST(^) = Go To 19
ST(() = Go To 20

State 19  [ 3 ]
 3 : S' -> . ^ S $


State 18  [ 2 ]
 2 : S -> S . - T

ST(S) = Go To 23

State 23  [ 2 ]
 2 : S -> . S - T
 2 : S' -> ^ . S $
 2 : T -> ( . S )

ST(^) = Go To 24
ST(() = Go To 25

State 25  [ 2 ]
 2 : T -> . ( S )
 2 : S -> S - . T
 2 : S -> . T
 2 : S' -> ^ . S $
 2 : T -> ( . S )

ST(-) = Go To 18
ST(^) = Go To 24
ST(() = Go To 25

State 24  [ 2 ]
 2 : S' -> . ^ S $


State 2  [ 1 ]
```

```
 1 : S' -> ^ S . $

ST(S) = Go To 26

State 26  [ 1 ]
 1 : S' -> ^ . S $

ST(^) = Go To 27

State 27  [ 1 ]
 1 : S' -> . ^ S $


End of generation.

Automaton statistics:
 Number of states = 27
 Number of transitions & actions = 45
```

A.2 Discriminating reverse LR(0) generation for non-LR(0) grammar $\mathcal{G}_{etf}$

This listing corresponds to the discriminating reverse LR(0) automaton for LR(1), non-LR(0) grammar $\mathcal{G}_{etf}$ shown in Figure 5.13 (page 86).

```
LR(1) discriminating reverse automaton experimental tool (v9.3.5 21-jun-1998)

Invoked:  ./aut gr0 -gf etf.gra -dai +f

Discriminating reverse LR(0) automaton

State 1  [ 0 1 2 3 4 5 6 7 ]
 1 : S' -> ^ E $ .
 2 : E -> E + T .
 3 : E -> T .
 4 : T -> T * F .
 5 : T -> F .
 6 : F -> ( E ) .
 7 : F -> a .
 0 : S' -> ^ E . $
 0 : S' -> . ^ E $
 0 : E -> E . + T
 0 : T -> T . * F
 0 : F -> ( E . )
 0 : F -> . ( E )
 0 : T -> T * . F
 0 : T -> . F
 0 : E -> E + . T
 0 : E -> . T
 0 : S' -> ^ . E $
 0 : F -> ( . E )

ST($) = Reduce 1
ST()) = Reduce 6
ST(a) = Reduce 7
ST(E) = Shift
ST(*) = Shift
ST(+) = Shift
```

```
ST(^) = Shift
ST(() = Shift
ST(T) = Go To 2
ST(F) = Go To 3

State 3  [ 4 5 ]
 4 : T -> T * . F
 5 : T -> . F
 5 : E -> E + . T
 5 : E -> . T
 5 : S' -> ^ . E $
 5 : F -> ( . E )

ST(*) = Reduce 4
ST(+) = Reduce 5
ST(^) = Reduce 5
ST(() = Reduce 5

State 2  [ 0 2 3 ]
 2 : E -> E + . T
 3 : E -> . T
 0 : T -> . T * F
 3 : S' -> ^ . E $
 3 : F -> ( . E )
 0 : E -> E + . T
 0 : E -> . T
 0 : S' -> ^ . E $
 0 : F -> ( . E )

CONFLICT(S) IN STATE!

End of generation.

Non-LR(0) Grammar!
Parser not generated!
```

A.3 Discriminating reverse LR(1) automaton for grammar $\mathcal{G}_{nla}$

This listing corresponds to the discriminating reverse LR(1) automaton for grammar $\mathcal{G}_{nla}$ shown in Figure 5.46 (page 126).

```
LR(1) discriminating reverse automaton experimental tool (v9.3.5 21-jun-1998)

Invoked:  ./aut gr -gf nla.gra -dais +f

Input grammar statistics:
 Number of rules = 6
 Average length of rightparts = 2.33
 Average number of terminal symbols per rightpart = 1.67
 Vocabulary:
  Number of terminal symbols = 3
  Number of nonterminal symbols = 3

Discriminating reverse LR(1) automaton

State 1  [ 0 1 2 3 4 5 6 7 ]
 1 : S' -> ^ S $ . , . $
 2 : S -> a A a . , . $
```

```
 3 : S -> b A b . , . $
 4 : S -> a B b . , . $
 5 : S -> b B a . , . $
 6 : A -> c . , . a
 6 : A -> c . , . b
 7 : B -> c . , . a
 7 : B -> c . , . b
 0 : S' -> ^ S . $ , $ .
 0 : S' -> . ^ S $ , ^ .
 0 : S -> a A . a , a .
 0 : S -> . a A a , a .
 0 : S -> b A . b , b .
 0 : S -> . b A b , b .
 0 : S -> a B . b , b .
 0 : S -> b B . a , a .
 0 : A -> . c , c .
 0 : B -> . c , c .
 0 : S' -> ^ . S $ , a .
 0 : S' -> ^ . S $ , b .
 0 : S -> a . A a , c .
 0 : S -> b . A b , c .

LA(c) = Shift
ST($) = Reduce 1
ST(S) = Shift
ST(A) = Shift
ST(B) = Shift
ST(^) = Shift
ST(a) = Go To 2
ST(b) = Go To 3
ST(c) = Go To 4

State 4  [ 6 7 ]
 6 : A -> . c , . a
 6 : A -> . c , . b
 7 : B -> . c , . a
 7 : B -> . c , . b
 6 : S -> a . A a , a .
 6 : S -> b . A b , b .
 7 : S -> b . B a , a .
 7 : S -> a . B b , b .

ST(a) = Go To 5
ST(b) = Go To 6

State 6  [ 6 7 ]
 6 : S -> . b A b , b .
 7 : S -> . b B a , a .
 6 : S' -> ^ . S $ , b .
 7 : S' -> ^ . S $ , a .

LA(b) = Reduce 6
LA(a) = Reduce 7

State 5  [ 6 7 ]
 6 : S -> . a A a , a .
 7 : S -> . a B b , b .
 6 : S' -> ^ . S $ , a .
 7 : S' -> ^ . S $ , b .

LA(a) = Reduce 6
LA(b) = Reduce 7
```

```
State 3  [ 3 4 ]
 3 : S -> b A . b , . $
 4 : S -> a B . b , . $

ST(A) = Reduce 3
ST(B) = Reduce 4

State 2  [ 2 5 ]
 2 : S -> a A . a , . $
 5 : S -> b B . a , . $

ST(A) = Reduce 2
ST(B) = Reduce 5

End of generation.

Automaton statistics:
 Number of states = 6
 Number of transitions & actions = 19
```

A.4 Discriminating reverse LR(1) automaton including full rightpart verification for grammar $\mathcal{G}_{etf}$

This listing corresponds to the discriminating reverse LR(1) automaton with full rightpart verification for grammar $\mathcal{G}_{etf}$ shown in Figure 5.13 (page 86).

```
LR(1) discriminating reverse automaton experimental tool (v9.3.5 21-jun-1998)

Invoked:  ./aut gr -gf etf.gra -dai

Discriminating reverse LR(1) automaton

State 1  [ 0 1 2 3 4 5 6 7 ]
 1 . : S' -> ^ E $ . , . $
 2 . : E -> E + T . , . $
 2 . : E -> E + T . , . +
 2 . : E -> E + T . , . )
 3 . : E -> T . , . $
 3 . : E -> T . , . +
 3 . : E -> T . , . )
 4 . : T -> T * F . , . $
 4 . : T -> T * F . , . +
 4 . : T -> T * F . , . *
 4 . : T -> T * F . , . )
 5 . : T -> F . , . $
 5 . : T -> F . , . +
 5 . : T -> F . , . *
 5 . : T -> F . , . )
 6 . : F -> ( E ) . , . $
 6 . : F -> ( E ) . , . +
 6 . : F -> ( E ) . , . *
 6 . : F -> ( E ) . , . )
 7 . : F -> a . , . $
 7 . : F -> a . , . +
 7 . : F -> a . , . *
```

```
 7 . : F -> a . , . )
 . 0 : S' -> ^ E . $ , $ .
 . 0 : S' -> . ^ E $ , ^ .
 . 0 : E -> E . + T , + .
 . 0 : T -> T . * F , * .
 . 0 : F -> ( E . ) , ) .
 . 0 : F -> . ( E ) , ( .
 . 0 : F -> . a , a .
 . 0 : T -> T * . F , ( .
 . 0 : T -> . F , ( .
 . 0 : T -> T * . F , a .
 . 0 : T -> . F , a .
 . 0 : E -> E + . T , ( .
 . 0 : E -> . T , ( .
 . 0 : E -> E + . T , a .
 . 0 : E -> . T , a .
 . 0 : S' -> ^ . E $ , ( .
 . 0 : F -> ( . E ) , ( .
 . 0 : S' -> ^ . E $ , a .
 . 0 : F -> ( . E ) , a .

LA(() = Shift
LA(a) = Shift
ST(a) = Reduce 7
ST(E) = Shift
ST($) = Go To 2
ST(T) = Go To 3
ST(F) = Go To 4
ST()) = Go To 5

State 5  [ 6 ]
 6 . : F -> ( E . ) , . $
 6 . : F -> ( E . ) , . +
 6 . : F -> ( E . ) , . *
 6 . : F -> ( E . ) , . )

ST(E) = Go To 6

State 6  [ 6 ]
 6 . : F -> ( . E ) , . $
 6 . : F -> ( . E ) , . +
 6 . : F -> ( . E ) , . *
 6 . : F -> ( . E ) , . )

ST(() = Reduce 6

State 4  [ 4 5 ]
 4 . : T -> T * . F , . $
 4 . : T -> T * . F , . +
 4 . : T -> T * . F , . *
 4 . : T -> T * . F , . )
 5 . : T -> . F , . $
 5 . : T -> . F , . +
 5 . : T -> . F , . *
 5 . : T -> . F , . )
 . 5 : E -> E + . T , . $
 . 5 : E -> . T , . $
 . 5 : E -> E + . T , . +
 . 5 : T -> . T * F , * .
 . 5 : E -> E + . T , . )
 . 5 : E -> . T , . )
 . 5 : S' -> ^ . E $ , $ .
```

```
 . 5 : E -> . E + T , + .
 . 5 : E -> E + . T , * .
 . 5 : E -> . T , * .
 . 5 : F -> ( . E ) , ) .
 . 5 : S' -> ^ . E $ , + .
 . 5 : F -> ( . E ) , + .
 . 5 : S' -> ^ . E $ , * .
 . 5 : F -> ( . E ) , * .

ST(+) = Reduce 5
ST(^) = Reduce 5
ST(() = Reduce 5
ST(*) = Go To 7

State 7  [ 4 ]
 4 . : T -> T . * F , . $
 4 . : T -> T . * F , . +
 4 . : T -> T . * F , . *
 4 . : T -> T . * F , . )

ST(T) = Reduce 4

State 3  [ 0 2 3 ]
 2 . : E -> E + . T , . $
 2 . : E -> E + . T , . +
 2 . : E -> E + . T , . )
 3 . : E -> . T , . $
 3 . : E -> . T , . +
 3 . : E -> . T , . )
 . 0 : T -> . T * F , * .
 . 3 : S' -> ^ . E $ , $ .
 . 3 : E -> . E + T , + .
 . 3 : F -> ( . E ) , ) .
 . 0 : E -> E + . T , * .
 . 0 : E -> . T , * .
 . 3 : S' -> ^ . E $ , + .
 . 3 : F -> ( . E ) , + .
 . 0 : S' -> ^ . E $ , * .
 . 0 : F -> ( . E ) , * .

LA(*) = Shift
ST(^) = Reduce 3
ST(() = Reduce 3
ST(+) = Go To 8

State 8  [ 2 ]
 2 . : E -> E . + T , . $
 2 . : E -> E . + T , . +
 2 . : E -> E . + T , . )

ST(E) = Reduce 2

State 2  [ 1 ]
 1 . : S' -> ^ E . $ , . $

ST(E) = Go To 9

State 9  [ 1 ]
 1 . : S' -> ^ . E $ , . $

ST(^) = Reduce 1
```

```
End of generation.
```

A.5 Discriminating reverse LR(1) automaton with conflict resolution for grammar $\mathcal{G}_{cie}$

This listing corresponds to the discriminating reverse LR(1) automaton with conflict resolution for grammar $\mathcal{G}_{cie}$ shown in Figure 8.8 (page 202).

```
LR(1) discriminating reverse automaton experimental tool (v9.3.6 24-jun-1998)

Invoked:  ./aut gr -gf elsexp.gra -dai -g7 -c +f

Discriminating reverse LR(1) automaton

 3 : S -> if E then S . , . else
 0 : S -> if E then S . else S , else .
Preferred action = 0

State 1  [ 0 1 2 3 4 5 6 7 8 ]
 1 : S' -> ^ S $ . , . $
 2 : S -> if E then S else S . , . $
 2 : S -> if E then S else S . , . else
 3 : S -> if E then S . , . $
 3 : S -> if E then S . , . else
 4 : S -> E . , . $
 4 : S -> E . , . else
 5 : E -> E + E . , . $
 5 : E -> E + E . , . then
 5 : E -> E + E . , . else
 5 : E -> E + E . , . +
 5 : E -> E + E . , . *
 5 : E -> E + E . , . )
 6 : E -> E * E . , . $
 6 : E -> E * E . , . then
 6 : E -> E * E . , . else
 6 : E -> E * E . , . +
 6 : E -> E * E . , . *
 6 : E -> E * E . , . )
 7 : E -> ( E ) . , . $
 7 : E -> ( E ) . , . then
 7 : E -> ( E ) . , . else
 7 : E -> ( E ) . , . +
 7 : E -> ( E ) . , . *
 7 : E -> ( E ) . , . )
 8 : E -> a . , . $
 8 : E -> a . , . then
 8 : E -> a . , . else
 8 : E -> a . , . +
 8 : E -> a . , . *
 8 : E -> a . , . )
 0 : S' -> ^ S . $ , $ .
 0 : S' -> . ^ S $ , ^ .
 0 : S -> if E then S . else S , else .
 0 : S -> if E . then S else S , then .
 0 : S -> . if E then S else S , if .
 0 : E -> E . + E , + .
 0 : E -> E . * E , * .
 0 : E -> ( E . ) , ) .
```

```
 0 : E -> . ( E ) , ( .
 0 : E -> . a , a .
 0 : S' -> ^ . S $ , if .
 0 : S -> if E then . S else S , if .
 0 : S -> if E then S else . S , if .
 0 : S -> if . E then S else S , ( .
 0 : S -> . E , ( .
 0 : E -> E + . E , ( .
 0 : E -> E * . E , ( .
 0 : E -> ( . E ) , ( .
 0 : S -> if . E then S else S , a .
 0 : S -> . E , a .
 0 : E -> E + . E , a .
 0 : E -> E * . E , a .
 0 : E -> ( . E ) , a .
 0 : S' -> ^ . S $ , ( .
 0 : S -> if E then . S else S , ( .
 0 : S -> if E then S else . S , ( .
 0 : S' -> ^ . S $ , a .
 0 : S -> if E then . S else S , a .
 0 : S -> if E then S else . S , a .

LA(if) = Shift
LA(() = Shift
LA(a) = Shift
ST($) = Reduce 1
ST()) = Reduce 7
ST(a) = Reduce 8
ST(S) = Go To 2
ST(E) = Go To 3
CONFLICT(S) IN STATE!

 5 : E -> E + . E , . +
 0 : E -> E + . E , + .
Preferred action = 5

 5 : E -> E + . E , . *
 0 : E -> E + . E , * .
Preferred action = 0

 6 : E -> E * . E , . +
 0 : E -> E * . E , + .
Preferred action = 6

 6 : E -> E * . E , . *
 0 : E -> E * . E , * .
Preferred action = 6

State 3  [ 0 4 5 6 ]
 4 : S -> . E , . $
 4 : S -> . E , . else
 5 : E -> E + . E , . $
 5 : E -> E + . E , . then
 5 : E -> E + . E , . else
 5 : E -> E + . E , . +
 5 : E -> E + . E , . *
 5 : E -> E + . E , . )
 6 : E -> E * . E , . $
 6 : E -> E * . E , . then
 6 : E -> E * . E , . else
 6 : E -> E * . E , . +
 6 : E -> E * . E , . *
```

```
 6 : E -> E * . E , . )
 0 : S -> if . E then S else S , then .
 0 : E -> . E + E , + .
 0 : E -> . E * E , * .
 0 : E -> ( . E ) , ) .
 4 : S' -> ^ . S $ , $ .
 4 : S -> if E then S else . S , . $
 4 : S -> if E then . S , . $
 4 : S -> if E then . S else S , else .
 4 : S -> if E then S else . S , . else
 0 : S -> if . E then S else S , + .
 0 : S -> . E , + .
 0 : E -> E + . E , + .
 0 : E -> E * . E , + .
 0 : E -> ( . E ) , + .
 0 : S -> if . E then S else S , * .
 0 : S -> . E , * .
 0 : E -> E + . E , * .
 0 : E -> E * . E , * .
 0 : E -> ( . E ) , * .
 0 : S' -> ^ . S $ , + .
 0 : S -> if E then . S else S , + .
 0 : S -> if E then S else . S , + .
 0 : S' -> ^ . S $ , * .
 0 : S -> if E then . S else S , * .
 0 : S -> if E then S else . S , * .

ST(*) = Reduce 6
ST(if) = Shift
ST(() = Shift
ST(+) = Go To 4
ST(^) = Go To 5
ST(else) = Go To 6
ST(then) = Go To 7
CONFLICT(S) IN STATE!

State 7  [ 0 4 ]
 4 : S -> if E . then S , . $
 4 : S -> if E . then S else S , else .
 0 : S -> if E . then S else S , + .
 0 : S -> if E . then S else S , * .

LA($) = Reduce 4
LA(else) = Reduce 4
LA(+) = Shift
LA(*) = Shift

State 6  [ 0 4 ]
 4 : S -> if E then S . else S , . $
 4 : S -> if E then S . else S , . else
 0 : S -> if E then S . else S , + .
 0 : S -> if E then S . else S , * .

LA($) = Reduce 4
LA(else) = Reduce 4
LA(+) = Shift
LA(*) = Shift

State 5  [ 0 4 ]
 4 : S' -> . ^ S $ , $ .
 0 : S' -> . ^ S $ , + .
 0 : S' -> . ^ S $ , * .
```

```
LA($) = Reduce 4
LA(+) = Shift
LA(*) = Shift

State 4  [ 0 5 ]
 5 : E -> E . + E , . $
 5 : E -> E . + E , . then
 5 : E -> E . + E , . else
 5 : E -> E . + E , . +
 5 : E -> E . + E , . )
 0 : E -> E . + E , * .

LA($) = Reduce 5
LA(then) = Reduce 5
LA(else) = Reduce 5
LA(+) = Reduce 5
LA()) = Reduce 5
LA(*) = Shift

State 2  [ 0 2 3 ]
 2 : S -> if E then S else . S , . $
 2 : S -> if E then S else . S , . else
 3 : S -> if E then . S , . $
 0 : S' -> ^ . S $ , $ .
 0 : S -> if E then . S else S , else .

ST(else) = Reduce 2
ST(^) = Shift
ST(then) = Go To 8

State 8  [ 0 3 ]
 3 : S -> if E . then S , . $
 0 : S -> if E . then S else S , else .

LA($) = Reduce 3
LA(else) = Shift

End of generation.

Non-LR(1) Grammar!
Conflict(s) resolved according to preferences.
```

Appendix B

Experimental Tool User Manual

In order to make practical tests and comparisons with other LR methods, an experimental tool has been implemented, which generates parsing tables for discriminating-reverse LR(1), and some funny variants as LR(0) or full-stack reverse recognizers. The program also includes generation of direct canonical LR(1), LR(0), SLR(1), and LALR(1) (uncompressed-)table-driven parsers, and even an automatic sentence generator, which can be used to obtain easily nontrivial random sentences for test purposes.

Efficiency was not amongst the objectives for these implementations, i.e., this is not a production-quality tool, since the main interest is to have a code that could be easily modified and clear to implement directly the algorithms and minimize the possibility of programming errors. It has been found convenient also for didactic purposes.

This Appendix presents the user manual for the latest version of the tool. First, the general layout is presented, showing how the different modules interact, the file organization, the grammar and sentence formats, and the command interface for the different modules. Finally, an example session is reproduced.

B.1 Using the tool

The tool is invoked at the command line prompt with the program name `aut`, followed by a functioning mode, asking for one of the following things:

`aut gr` K To *g*enerate the discriminating-*r*everse parsing tables

`aut gd` K To *g*enerate the usual *d*irect parsing tables

K represents the parser lookahead length, which may be specified as `0` or `1` (default).

`aut s` LENGTH To generate one or more random sentences from the grammar. If LENGTH is specified, sentence(s) with the nearest possible length are generated; otherwise random length sentences are produced.

`aut pr` To *p*arse an input sequence with the discriminating-*r*everse parsing tables

`aut pd` To *p*arse an input sequence with the *d*irect parsing tables

First, one of the parser generators is invoked. Then, the sentence generator can optionally be invoked to produce a random sentence, or an input text file can be manually constructed by using any text editor. Finally, the corresponding parser is invoked with the input file.

B.1.1 Files used by the tool

The tool uses the following four files —the default names in the current directory are shown, although at invocation time other (path) name(s) can be given:

`g.aut` ASCII text file containing the specification grammar in Backus-Naur Form (BNF). This is the input to the generators (see the section below for the format of this file).

`v.aut, p.aut` Vocabulary and parsing-table file, respectively, both in internal format. They are produced by the parser generators and read by the parsers.

`s.aut` ASCII text file containing a sequence to be parsed. It can be manually written or automatically generated by `aut s` (see below for the format of this file).

The use of default names allows to comfortably work without caring about the internal files of the tool. That is, the user can write with an editor the `g.aut` text file containing the grammar —two programs, `yacc2aut` and `aut2yacc`, are available to convert `yacc` grammars to the tool format and vice versa— and successively invoke the different modes, without any new command or edition, to generate parsers or sentences and verify parses.

As previously stated, the user can specify, when appropriate, the precise file (path) names, with the following command line options:

`-gf` FILEPATHNAME *G*rammar *f*ile.

`-vf` FILEPATHNAME *V*ocabulaty *f*ile.

`-pf` FILEPATHNAME *P*arsing-table *f*ile.

`-sf` FILEPATHNAME Input *f*ile to the parser containing a *s*equence of terminals symbols.

The grammar file

The grammar file is the input to the parser or sentence generators. It specifies the structure of legal sentences of the language. The file format is an ASCII text file containing the grammar in BNF notation. More specifically, a grammar is a sequence of rules, where each rule has the form:

```
nonterminal = rightpart { | rightpart } .
```

That is, a nonterminal name, an equal sign, and one or more rightparts (sequences of symbols) separated by vertical bars, all ended with a dot. One rightpart may be empty. Terminal symbols should appear between quotes (`"`) containing the strings as they will literally appear in sentences, formed by any sequence of characters (quotes are represented in the sequence by a pair of quotes). Nonterminal symbols begin with any alphabetical upper or lower case characters or underscore (`_`), and may also contain numerical characters. The start symbol is the first to appear in the file.

Grammars should be well-formed (usually called "reduced" in the literature), otherwise an error is signalled when they are processed. The following tests are made:

- No rightpart can be repeated for the same nonterminal.
- Every terminal symbol should be generated from the start symbol.
- Every nonterminal symbol should derive sequences consisting only of terminals or the empty sequence.

Note also that recursive rules as "`A = A`" are useless and make the grammar ambiguous.

The input files to parse

After the parser has been successfully generated, we might want to test it with an input file. The format of this file to parse is just an ASCII text file containing the terminal symbols exactly as they appear in the grammar (without quotes), separated by spaces or newlines. This is precisely the format produced by the automatic sentence generator.

This simple scheme allows an easy construction of sentences, with no other purpose than to test the parsers. The parser is intended to be integrated within the Cigale system, then standard lexical analyzer features will be available.

B.1.2 Other command line options

This section describes other command line options. Some of them are significant only in some functioning mode, some in several modes. Although this should be evident from the explanations, you can look at the beginning of the example session below to see which case is appropriate.

`-NUMBER` Specify the number of sentences to display (or redirect to a file with `>`). Only the last one is saved to the sentence file. By default NUMBER is 1.

`-c` In case of reverse *c*onflicts for non-LR(K) grammars, interactively ask which action is preferred. Note that conflicting situations are detected in pairs, and so the user is asked.

`+c` By default the generator stops state construction from situations in conflict. With this option the curious user makes the generator to *c*ontinue building states beyond conflicts until bottom-of-stack states.

`+e` A technique using a heuristic estimate of typical l*e*ngths of nonterminal phrases is used by default to generate sentences, with the intention to produce human-like sentences. With this option, that computation is disabled, and sentences are generated purely at random, but then results are typically very poor in comparison with the default.

`+f` Disable *f*ull rightparts computation, that is, do not consider in reverse automata whether or not the lookahead part is covered by the portion of the rightpart to the left of the marker. Without special options activated, this option asks the generator to produce only the discriminating section of the automaton, not the additional states that would be used to verify the presence of the handle on top of the stack.

`-g` DEBUGNUMBER Activate the debug flag DEBUGNUMBER. For developing purposes only.

`-i` With this option the reverse parser is generated from an *i*nitial state interactively specified by the user. State situations need to be specified according to internal codes, which can be displayed using option `-dgv` (see below).

`+l` Produce a (direct) LALR(1) parser instead of an full LR(1) one, by merging states with the same core (non-lookahead) part. Apart from this, no further compression is made, contrary to what is usual in other LALR(1) parser generators, e.g., `yacc`.

`-l` When producing a direct LR(1) parser for a non-LALR(1) grammar, signal states that after merger with previous LR(1) states would lead to LALR(1) conflicts.

`+n` SEED By default, every time the sentence generator is invoked, a different random sentence is produced. Sometimes, repeatability is interesting, and then this option can be used to always produce the same random sentence(s). A SEED value can be optionally specified to obtain more (repeatable) random sentences. The default value of SEED is 0.

`-r a` A full reverse recognizer for the whole stack is produced, which can be compared with the usual discriminating reverse automaton. Either LR(0) or LR(1) algorithm can be used. If suboption "`a`" is not specified, situation actions are not considered during automaton construction, which yields a smaller full-stack recognizer, although useless for action discrimination.

`+s` Produce a (direct) SLR(1) parser instead of an full LR(1) one, by augmenting items within the LR(0) automaton states with lookaheads in the Follow sets. The resulting states correspond to LALR(1) ones, but sometimes the SLR(1) parsing tables have more entries.

`+t` CHAR Produce a file named `paut.tex` containing the parsing table in text form, which can be displayed, browsed with an editor, or printed. If CHAR is specified (usually a dot), then it is used instead of a blank space for empty table entries. See also option `-x`.

`+v` The program produces by default some friendly output. This can be avoided with this option, which may be of interest when the tool is executed from scripts.

`-x` To help in the preparation of documents on this subject, with this option the generator produces displayed data in LaTeX notation, as well as a file named `paut.tex` containing the parsing table in suitable form to be included in a LaTeX document.

Display options

There are several display options that the user can ask for in the command line to show some results or listings. All of them begin with `-d` and then a list of letters telling the display options wanted. As with other options, display options are significant only in some modes. See the example session below for their meaning.

B.2 A sample session

In this section, an example session is shown to give an idea of the dynamic of working with the tool, and about the kind of results that can be obtained.

When the program is run without arguments, the following help is displayed —also, when an error is encountered in the command line, the user is asked whether he wants the help to be displayed.

```
$ aut

LR(1) discriminating reverse automaton experimental tool (v10.0.3 9-10-1998)

Usage:

  aut g r|d[#] {-|+c -d{ailbfgsv} +f -g[#] -i +|-l -r[a] +s -t. +v -x file}
  aut s[#] {-# -d{bdfgvem} +e -g[#] +n[#] +v file}
  aut p r|d {-d{psu} -g[#] +v file}

 Functioning modes:
  g      parser Generation (reads gf; writes vf, pf)
  s      random Sentence generation (reads gf; writes sf), with random or
         given (#) length (quite approximately)
  p      Parse (reads vf, pf, sf) with mostly manual error recovery

 Automaton classes:
  r      Reverse discriminating
  d      Direct (classical)
  #      lookahead length: 0 or 1 (default)

 Files:
  -gf path  BNF Grammar File (default g.aut) with rules like:
               proc_list = "proc" heading ";" proc_list | .
  -sf path  Sentence text File (default s.aut) to parse; spaces separate
            terminals.
  -pf path  Parsing table File (default p.aut) in internal format.
  -vf path  Vocabulary File (default v.aut) in internal format.

 Options (some do not produce a usable parser):
  -#     Number of sentences
  -c     resolve (reverse) Conflicts (by asking)
  +c     Continue (reverse) state construction beyond conflicts
  -d     Display options:
    a     Automaton states; suboptions:
     i     state situations (also called Items)
     l     List actions for (stack symbol, lookahead) pairs
    b      intput BNF grammar
    d      sentential forms during Derivation
    f      First and follow sets
    g      internal Grammar
    p      (action counts and) stack windows during Parsing
    s      Statistics
    u      strict parsing User time
    v      Vocabulary; suboptions:
     e      estimated typical lEnghts of phrases
     m      Minimum and maximum lenghts of phrases
  +e     disable use of typical lEngths of phrases
  +f     disable Full rightparts computation in situations
  -g[#]  debuGging [option; default 0]
  -i     ask (reverse) Initial state
  +l     build (direct) LALR(1) automaton
  -l     signal (direct) LALR(1) conflicts in LR(1) automaton
  +n[#]  seed [Number; default 0]: disable randomize
```

```
  -r[a]  full (reverse) Recognizer [consider Actions]
  +s     build (direct) SLR(1) automaton
  -t.    produce parsing table in plain Text using '.' for empty entries
  +v     disable Verbose output
  -x     produce output and parsing table in LaTeX notation

Author address:  <Fortes Gálvez, José> jfortes@dis.ulpgc.es.
```

Before using the tool, let us have a look at the grammar with which we are going to work:

```
$ cat g.aut
E = E "+" T | T .
T = F R .
R = | "*" F R .
F = "(" E ")" | "a" .
```

Let us generate a random sentence (corresponding to seed number 28) of 15 symbols on default file `s.aut`, and display its derivation steps. You might not get the same sentence as below, but yours should be the same each time you run your program with the same command line below.

```
Invoked:  aut s15 +n28 -dd

<E>:15

<E>:11 + <T>:3

<E>:7 + <T>:3 + <F>:2 <R>:1

<T>:7 + <F>:2 <R>:3 + a

<F>:4 <R>:3 + a * <F>:2 <R>:1 + a

( <E>:3 ) * <F>:2 <R>:1 + a * a + a

( <E>:2 + <T>:2 ) * a + a * a + a

( <T>:2 + <F>:1 <R>:1 ) * a + a * a + a

( <F>:2 <R>:2 + a ) * a + a * a + a

( a * <F>:0 <R>:0 + a ) * a + a * a + a

( a * a + a ) * a + a * a + a

( a * a + a ) * a + a * a + a
```

Let us produce a discriminating-reverse parser for the grammar above, displaying the grammar itself (notice the rule numbers assigned), the automaton transition tables (`LA` and `ST` stand for lookahead and stack symbol, respectively) and associated sets of situations, and some statistics (about the grammar and the generated parser). Some output has been deleted to save space.

```
Invoked:  aut gr -dgais

Input grammar statistics:
 Number of rules = 7
 Average length of rightparts = 1.86
 Average number of terminal symbols per rightpart = 0.71
 Vocabulary:
  Number of terminal symbols = 5
  Number of nonterminal symbols = 4

1: S' -> ^ E $
2: E -> E + T
3: E -> T
4: T -> F R
5: R ->
6: R -> * F R
7: F -> ( E )
8: F -> a

Discriminating reverse LR(1) automaton

State 1  [ 0 1 2 3 4 5 6 7 8 ]
 1 . : S' -> ^ E $ . , . $
 2 . : E -> E + T . , . $
 2 . : E -> E + T . , . +
 2 . : E -> E + T . , . )
 3 . : E -> T . , . $
 3 . : E -> T . , . +
 3 . : E -> T . , . )
 4 . : T -> F R . , . $
 4 . : T -> F R . , . +
 4 . : T -> F R . , . )
 5 . : R -> . , . $
 5 . : R -> . , . +
 5 . : R -> . , . )
 6 . : R -> * F R . , . $
 6 . : R -> * F R . , . +
 6 . : R -> * F R . , . )
 7 . : F -> ( E ) . , . $
 7 . : F -> ( E ) . , . +
 7 . : F -> ( E ) . , . *
 7 . : F -> ( E ) . , . )
 8 . : F -> a . , . $
 8 . : F -> a . , . +
 8 . : F -> a . , . *
 8 . : F -> a . , . )
 . 0 : S' -> ^ E . $ , $ .
 . 0 : S' -> . ^ E $ , ^ .
 . 0 : E -> E . + T , + .
 . 0 : R -> . * F R , * .
```

```
 . 0 : F -> ( E . ) , ) .
 . 0 : F -> . ( E ) , ( .
 . 0 : F -> . a , a .
 . 5 : T -> F . R , . $
 . 5 : R -> * F . R , . $
 . 5 : T -> F . R , . +
 . 5 : R -> * F . R , . +
 . 5 : T -> F . R , . )
 . 5 : R -> * F . R , . )
 . 0 : T -> F . R , * .
 . 0 : R -> * F . R , * .
 . 0 : T -> . F R , ( .
 . 0 : R -> * . F R , ( .
 . 0 : T -> . F R , a .
 . 0 : R -> * . F R , a .
 . 0 : E -> E + . T , ( .
 . 0 : E -> . T , ( .
 . 0 : E -> E + . T , a .
 . 0 : E -> . T , a .
 . 0 : S' -> ^ . E $ , ( .
 . 0 : F -> ( . E ) , ( .
 . 0 : S' -> ^ . E $ , a .
 . 0 : F -> ( . E ) , a .

LA(() = Shift
LA(a) = Shift
ST(a) = Reduce 8
ST(E) = Shift
ST($) = Go To 2
ST(T) = Go To 3
ST(R) = Go To 4
ST()) = Go To 5
ST(F) = Go To 6

        .
        .
        .

State 2  [ 1 ]
 1 . : S' -> ^ E . $ , . $

ST(E) = Go To 10

State 10  [ 1 ]
 1 . : S' -> ^ . E $ , . $

ST(^) = Reduce 1

End of generation.

Automaton statistics:
 Number of states = 10
 Number of transitions & actions = 26
```

Now, let us try the parser with the previously generated sentence and display the parsing stack just before each reduction. The pairs of numbers at the left of each line count the number of shifts and reduces. The terminals between parenthesis before each nonterminal correspond to the first terminal of its corresponding phrase in the sentence. Some parsing statistics are also displayed.

```
Invoked:  aut pr -dps
2,0: ^ ( a          * a + a ) * a + a *
4,1: ^ ( (a)F * a          + a ) * a + a * a +
4,2: ^ ( (a)F * (a)F          + a ) * a + a * a +
4,3: ^ ( (a)F * (a)F (+)R          + a ) * a + a * a +
4,4: ^ ( (a)F (*)R          + a ) * a + a * a +
4,5: ^ ( (a)T          + a ) * a + a * a +
6,6: ^ ( (a)E + a          ) * a + a * a + a $
6,7: ^ ( (a)E + (a)F          ) * a + a * a + a $
6,8: ^ ( (a)E + (a)F ())R          ) * a + a * a + a $
6,9: ^ ( (a)E + (a)T          ) * a + a * a + a $
7,10: ^ ( (a)E )          * a + a * a + a $ $
9,11: ^ (()F * a          + a * a + a $ $ $ $
9,12: ^ (()F * (a)F          + a * a + a $ $ $ $
9,13: ^ (()F * (a)F (+)R          + a * a + a $ $ $ $
9,14: ^ (()F (*)R          + a * a + a $ $ $ $
9,15: ^ (()T          + a * a + a $ $ $ $
11,16: ^ (()E + a          * a + a $ $ $ $ $ $
13,17: ^ (()E + (a)F * a          + a $ $ $ $ $ $ $ $
13,18: ^ (()E + (a)F * (a)F          + a $ $ $ $ $ $ $ $
13,19: ^ (()E + (a)F * (a)F (+)R          + a $ $ $ $ $ $ $ $
13,20: ^ (()E + (a)F (*)R          + a $ $ $ $ $ $ $ $
13,21: ^ (()E + (a)T          + a $ $ $ $ $ $ $ $
15,22: ^ (()E + a          $ $ $ $ $ $ $ $ $ $
15,23: ^ (()E + (a)F          $ $ $ $ $ $ $ $ $ $
15,24: ^ (()E + (a)F ($)R          $ $ $ $ $ $ $ $ $ $
15,25: ^ (()E + (a)T          $ $ $ $ $ $ $ $ $ $
16,26: ^ (()E $          $ $ $ $ $ $ $ $ $ $

Successful end of parsing.

Statistics:
 Number of shift actions = 16
 Number of reduce actions = 26
 Average length of reductions = 1.54
 Maximum stack length = 7
 Average stack length = 3.95
 Average depth of stack exploration per shift action = 0.50
 Average depth of stack exploration per reduce action = 2.00
 Average number of state transitions per action = 0.86
```

Let us now compare the above results with a direct canonical LR(1) parser. First, let us generate the parser, displaying only the automaton tables (`NS` stands for next symbol, that is, current lookahead) and some statistics (the output has been partially deleted).

```
Invoked:  aut gd -das

Input grammar statistics:
 Number of rules = 7
 Average length of rightparts = 1.86
 Average number of terminal symbols per rightpart = 0.71
 Vocabulary:
  Number of terminal symbols = 5
  Number of nonterminal symbols = 4

Direct (classical) LR(1) automaton

State 1

NS(E) = Go To 2
NS(T) = Go To 3
NS(F) = Go To 4
NS(() = Shift 5
NS(a) = Shift 6

State 2

NS($) = Shift 7
NS(+) = Shift 8

        .
        .
        .

State 26

NS(*) = Reduce 7
NS()) = Reduce 7
NS(+) = Reduce 7

State 27

NS()) = Reduce 6
NS(+) = Reduce 6

End of generation.

Automaton statistics:
 Number of states = 27
 Number of transitions & actions = 80
```

Finally, let us try the generated direct parser with the same sentence. In this case, before each parsing stack state its corresponding symbol is shown. As before, the stack is only displayed before each reduction.

```
Invoked:  aut pd -dp
(:5 a:15           * a + a ) * a + a *
(:5 F:13 *:21 a:15           + a ) * a + a * a +
```

```
(:5 F:13 *:21 F:25          + a ) * a + a * a +
(:5 F:13 *:21 F:25 R:27          + a ) * a + a * a +
(:5 F:13 R:20          + a ) * a + a * a +
(:5 T:12          + a ) * a + a * a +
(:5 E:11 +:19 a:15          ) * a + a * a + a $
(:5 E:11 +:19 F:13          ) * a + a * a + a $
(:5 E:11 +:19 F:13 R:20          ) * a + a * a + a $
(:5 E:11 +:19 T:24          ) * a + a * a + a $
(:5 E:11 ):18          * a + a * a + a $ $
F:4 *:10 a:6          + a * a + a $ $ $ $
F:4 *:10 F:17          + a * a + a $ $ $ $
F:4 *:10 F:17 R:23          + a * a + a $ $ $ $
F:4 R:9          + a * a + a $ $ $ $
T:3          + a * a + a $ $ $ $
E:2 +:8 a:6          * a + a $ $ $ $ $ $
E:2 +:8 F:4 *:10 a:6          + a $ $ $ $ $ $ $ $
E:2 +:8 F:4 *:10 F:17          + a $ $ $ $ $ $ $ $
E:2 +:8 F:4 *:10 F:17 R:23          + a $ $ $ $ $ $ $ $
E:2 +:8 F:4 R:9          + a $ $ $ $ $ $ $ $
E:2 +:8 T:16          + a $ $ $ $ $ $ $ $
E:2 +:8 a:6          $ $ $ $ $ $ $ $ $ $
E:2 +:8 F:4          $ $ $ $ $ $ $ $ $ $
E:2 +:8 F:4 R:9          $ $ $ $ $ $ $ $ $ $
E:2 +:8 T:16          $ $ $ $ $ $ $ $ $ $
E:2 $:7          $ $ $ $ $ $ $ $ $ $

Successful end of parsing.
```

Appendix C

Other experimental results

The table below shows the results from experimental tests on several practical programming language grammars, obtained, and kindly provided, by J. Farré (University of Nice-Sophia Antipolis, France) using his efficient implementation in C++ of the generation algorithm for the discriminating reverse LR(1) parser.

	Term	Non Term	Regles	YACC		BISON		DRLR		DRLR-FRP		NDRLR	
				etats	temps	etats	temps	etats	temps	etats	temps	etats	temps
ADA	95	238	459	858	3.0	860	1.0	266	3.08	382	3.38	700	3.2
C	84	64	214	367	4.50	369	2.72	168	2.67	236	2.97	479	
C bis	88	97	309	515	7.23	517	3.47	218	3.88	295	4.97	610	
C++	103	160	608	/	/	1160	14.85	919	17.73	1061	18.37	1640	
ConfETF	9	1	7	14	0.08	16	0.25	5	0.17	9	0.22	7	
CONFLIT	6	3	5	9	0.03	11	0.21	2	0.03	7	0.03	5	
DSL	139	152	368	699	5.52	701	3.03	358	4.30	588	5.22	481	
ETF	7	3	9	18	0.08	14	0.23	4	0.03	10	0.05	11	
GRAM	18	17	41	56	0.18	58	0.33	20	0.10	27	0.12	46	
GROS	161	189	473	800	14.53	802	5.70	401	7.98	536	8.78	473	
IMRA-C	104	124	303	485	9.27	487	3.52	208	4.78	282	5.08	660	
LOO	32	25	67	124	0.38	126	0.57	70	0.23	94	0.35	75	
MINI	46	42	103	190	0.68	192	0.80	102	0.60	132	0.70	152	
M2	73	134	245	386	2.15	388	1.38	196	1.42	284	1.62	371	
PASCAL	62	79	178	321	0.9	323	1.03	159	1.08	239	1.22	346	0.8
REACT-C	61	87	171	291	2.08	293	1.37	88	1.50	130	1.65	235	
VDM	165	183	440	853	41.78	855	9.52	311	8.93	474	9.72	724	

Bibliography

[ADG82] M. Ancona, G. Dodero, and V. Gianuzzi. Building collections of LR(k) items with partial expansion of lookahead strings. *ACM SIGPLAN Notices*, 17(5):24–28, May 1982.

[ADGM91] M. Ancona, G. Dodero, V. Gianuzzi, and M. Morgavi. Efficient construction of LR(k) states and tables. *ACM Transactions on Programming Languages and Systems*, 13(1):150–178, January 1991.

[ADU72] A. V. Aho, P. J. Denning, and J. D. Ullman. Weak and mixed strategy precedence parsing. *Journal of the ACM*, 19(2):225–243, April 1972.

[AEH73] T. Anderson, J. Eve, and J. J. Horning. Efficient LR(1) parsers. *Acta Informatica*, 2:12–39, 1973.

[AFG92] Massimo Ancona, Claudia Fassino, and Vittoria Gianuzzi. Optimization of LR(k) "Reduced parsers". *Information Processing Letters*, 41:13–20, 1992.

[AG81] M. Ancona and V. Gianuzzi. A new method for implementing LR(k) tables. *Information Processing Letters*, 13(4/5):171–176, 1981.

[AHS86] A. M. M. Al-Hussainin and R. T. Stone. Yet another storage technique for LR parsing tables. *Software—Practice and Experience*, 16(4):389–401, 1986.

[AJ74] A. V. Aho and S. C. Johnson. LR parsing. *ACM Computing Surveys*, 6:99–124, 1974.

[AJU72] A. V. Aho, S. C. Johnson, and J. D. Ullman. Optimization of LR(k) parsers. *Journal of Computer and System Sciences*, 6:573–602, 1972.

[AJU75] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(8):441–452, 1975.

[App97] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques.* Cambridge University Press, 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[AU73a] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling.* Prentice-Hall, 1972–1973.

[AU73b] Alfred V. Aho and Jeffrey D. Ullman. A technique for speeding up LR(k) parsers. *SIAM Journal on Computing*, 2(2):106–127, June 1973.

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[Bac76] R. C. Backhouse. An alternative approach to the improvement of LR(k) parsers. *Acta Informatica*, 6(3):277–296, 1976.

[Bac79] Roland C. Backhouse. *Syntax of Programming Languages.* Prentice-Hall, 1979.

[Bak81] Theodore P. Baker. Extending look-ahead for LR parsers. *Journal of Computer and System Sciences*, 22(2):243–259, 1981.

[BBG+60] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur (Ed.), A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960.

[BBGC86] William A. Barrett, Rodney M. Bates, David A. Gustafson, and John D. Couch. *Compiler Construction: Theory and Practice.* Science Research Associates, Chicago, 1986.

[BC79] William A. Barrett and John D. Couch. *Compiler Construction. Theory and Practice.* Science Research Associates, 1979.

[BE76] F. L. Bauer and J. Eickel. *Compiler Construction – an Advanced Course.* Springer Verlag, Berlin, second edition, 1976.

[Bea82] J. C. Beatty. On the relationship between LL(1) and LR(1) grammars. *Journal of the ACM*, 29:1007–1022, 1982.

[BL89] Manuel E. Bermudez and George Logothetis. Simple computation of LALR(1) look-ahead sets. *Information Processing Letters*, 31(5):233–238, 1989.

[Bou84] Pierre Boullier. *Contribution à la construction automatique d'analyseurs lexicographiques et syntaxiques.* PhD thesis, Université d'Orléans, France, 1984. In French.

[Bro74] B. M. Brosgol. Deterministic translation grammars. Technical Report TR-3-74, Center for Research in Computer Technology, Harvard University, 1974.

[BS86] Manuel E. Bermudez and Karl M. Schimpf. A practical arbitrary look-ahead LR parsing technique. *ACM SIGPLAN Notices*, 21(7):136–144, July 1986.

[BS90] Manuel E. Bermudez and Karl M. Schimpf. Practical arbitrary lookahead LR parsing. *Journal of Computer and System Sciences*, 41:230–250, 1990.

[ČC73] Karel Čulik II and Rina Cohen. LR-regular grammars — an extension of LR(k) grammars. *Journal of Computer and System Sciences*, 7:66–96, 1973.

[Cel81] Augusto Celentano. An LR parsing technique for extended context-free grammars. *Computer Languages (Elmsford, N.Y.)*, 6(2):95–107, 1981.

[CH87] Jacques Cohen and Timothy J. Hickey. Parsing and compiling using Prolog. *ACM Transactions on Programming Languages and Systems*, 9(2):125–164, April 1987.

[Cha87] Nigel P. Chapman. *LR Parsing. Theory and Practice*. Cambridge University Press, 1987.

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2:136–167, 1959.

[CO90] G. M. Clemm and L. J. Osterweil. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12:1–25, January 1990.

[Col70] Alain Colmerauer. Total precedence relations. *Journal of the ACM*, 17(1):14–30, January 1970.

[com91] Usenet `comp.compilers` newsgroup postings on sentence generation, May 1991.

[Cor89] Gordon V. Cormack. An LR substring parser for noncorrecting syntax error recovery. *ACM SIGPLAN Notices*, 24(7):161–169, July 1989.

[CRGM78] S. Crespi-Reghizzi, G. Guida, and D. Mandrioli. Noncounting context-free languages. *Journal of the ACM*, 25(4):571–580, October 1978.

[DDQ78] Peter J. Denning, Jack B. Dennis, and Joseph E. Qualitz. *Machines, Languages, and Computation*. Prentice-Hall, 1978.

[Dem75] A. J. Demers. Elimination of single productions and merging nonterminal symbols of LR(1) grammars. *Computer Languages*, 1:105–119, 1975.

[Dem77] Alan J. Demers. Generalized left corner parsing. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 170–182, 1977.

[DeR69] Franklin L. DeRemer. *Practical Translators for LR(k) Languages*. PhD thesis, Dep. Electrical Engineering, M.I.T., Cambridge, Massachusetts, 1969.

[DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, July 1971.

[Döm68] Bálint Dömölki. A universal compiler system based on production rules. *BIT*, 8(4):262–275, October 1968.

[DM81] A. J. T. Davie and R. Morrison. *Recursive Descent Compiling*. Ellis Horwood, 1981.

[DMM88] Pierpaolo Degano, Stefano Mannucci, and Bruno Mojana. Efficient incremental LR parsing for syntax-directed editors. *ACM Transactions on Programming Languages and Systems*, 10(3):345–373, July 1988.

[DP82] Frank DeRemer and Thomas J. Pennello. Efficient computation of LALR(1) lookahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1982.

[DP90] H. Dobler and K. Pirklbauer. Coco-2, a new compiler compiler. *ACM SIGPLAN Notices*, 25(5):82–90, May 1990.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[Eva64] A. Evans, Jr. An ALGOL 60 compiler. *Annual Review in Automatic Programming*, 4:87–124, 1964.

[FG91] José Fortes Gálvez. A discriminating reverse automaton for LR(1) parsing. Research Report 91-23, Laboratoire I3S, Bât. 4, 250 Avenue Albert Einstein, F-06560 Valbonne, France, October 1991.

[FG92a] José Fortes Gálvez. Generating LR(1) parsers of small size. In Uwe Kastens and Peter Pfahler, editors, *Compiler Construction. 4th International Conference, CC'92*, Lecture Notes in Computer Science, vol. 641, pages 16–29. Springer-Verlag, 1992.

[FG92b] José Fortes Gálvez. Hacia la generación de reconocedores LR(1) de reducidas dimensiones. In *PANEL'92, XVIII Conferencia Latinoamericana de Informática*, pages 493–500, Universidad de Las Palmas de Gran Canaria, Spain, 1992. In Spanish.

[FG94a] José Fortes Gálvez. Experimental results on discriminating-reverse LR(1) parsing. In Peter Fritzson, editor, *Proceedings of the Poster Session of CC'94 - International Conference on Compiler Construction*, pages 71–80. Department of Computer and Information Science, Linköping University, March 1994. Research report LiTH-IDA-R-94-11.

[FG94b] José Fortes Gálvez. A note on a proposed LALR parser for extended context-free grammars. *Information Processing Letters*, 50:303–305, 1994.

[FG96] José Fortes Gálvez. A practical small LR parser with action decision through minimal stack suffix scanning. In Jüergen Dassow, editor, *Developments in Language Theory II*. World Scientific, Singapore, 1996. Proceedings of Conference, Magdeburg, Germany 17–21 July 1995.

[Fin85] Gerard D. Finn. Extended use of null productions in LR(1) parser applications. *Communications of the ACM*, 28(9):961–972, September 1985.

[FL88] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting a Compiler*. Benjamin/Cummings, 1988.

[FL92] Carine Fédèle and Olivier Lecarme. Towards a toolkit for building language implementations. *Software—Practice and Experience*, 22(11):911–936, 1992.

[Flo61] R. W. Floyd. A descriptive language for symbol manipulation. *Journal of the ACM*, 8(4):579–584, 1961.

[Flo63] R. W. Floyd. Syntactic analysis and operator precedence. *Journal of the ACM*, 10(3):316–333, 1963.

[Flo64] R. W. Floyd. Bounded context syntactic analysis. *Communications of the ACM*, 7(2):62–67, 1964.

[GH73] James N. Gray and Michael A. Harrison. Canonical precedence schemes. *Journal of the ACM*, 20(2):214–234, April 1973.

[GH77] M. M. Geller and M. A. Harrison. Characteristic parsing: a framework for producing compact deterministic parsers. *Journal of Computer and System Sciences*, 14(3):265–317, June 1977.

[GHJ79] Susan L. Graham, Charles B. Haley, and William N. Joy. Practical LR error recovery. *ACM SIGPLAN Notices*, 14(8):168–175, August 1979.

[GHL+92] Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, 1992.

[GJ88] Dick Grune and Ceriel J. H. Jacobs. A programmer-friendly LL(1) parser generator. *Software—Practice and Experience*, 18(1):29–38, January 1988.

[GJ90] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide.* Ellis Horwood, 1990.

[GK88] J. Grosch and E. Klein. User manual for the PGS-system. Technical report, Gesellschaft für Mathematik und Datenverarbeitung, Research Institute at the University of Karlsruhe, Germany, 1988.

[GM79] Carlo Ghezzi and Dino Mandrioli. Incremental parsing. *ACM Transactions on Programming Languages and Systems*, 1(1):58–70, July 1979.

[GR75] Susan L. Graham and Steven P. Rhodes. Practical syntactic error recovery. *Communications of the ACM*, 18(11):639–650, 1975.

[Gra64] R. M. Graham. Bounded context translation. In *Proc. AFIPS Spring Joint Computer Conference*, pages 17–29. Spartan, New York, 1964.

[Gra74] Susan L. Graham. On bounded right context languages and grammars. *SIAM Journal on Computing*, 3(3):224–254, September 1974.

[Gre64] Sheila A. Greibach. Formal parsing systems. *Communications of the ACM*, 7(8):499–504, August 1964.

[Gri76] David Gries. Error recovery and correction. In F. L. Bauer and J. Eickel, editors, *Compiler Construction – an Advanced Course*, pages 627–638. Springer Verlag, Berlin, second edition, 1976.

[Gro84] K. Groening. Combined actions to reduce LR parser tables (experimental results). *ACM SIGPLAN Notices*, 19(3):42–45, 1984.

[Gro89] J. Grosch. Generators for high-speed front-ends. In D. Hammer, editor, *Compiler Compilers and High-Speed Compilation*, Lecture Notes in Computer Science, vol. 371, pages 81–92. Springer-Verlag, 1989.

[Gro90] J. Grosch. Lalr—a generator for efficient parsers. *Software—Practice and Experience*, 20(11):1115–1135, November 1990.

[Har78] Michael A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, 1978.

[Hei77] S. Heilbrunner. Using item grammars to prove LR(1) theorems. Bericht 7701, Fachbereich Informatik, Hochschüle der Bundeswehr, Munich, 1977.

[Hei79] Stephan Heilbrunner. On the definition of ELR(k) and ELL(k) grammars. *Acta Informatica*, 11:169–176, 1979.

[Hei81] Stephan Heilbrunner. A parsing automata approach to LR theroy. *Theoretical Computer Science*, 15:117–157, 1981.

[Hei85] S. Heilbrunner. Truly prefix-correct chain-free LR(1) parsers. *Acta Informatica*, 22(5):499–536, 1985.

[Heu86] V. P. Heuring. The automatic generation of fast lexical analysers. *Software—Practice and Experience*, 16:801–808, 1986.

[HHM92] A. Gayler Harford, Vincent P. Heuring, and Michael G. Main. A new parsing method for non-LR(1) grammars. *Software—Practice and Experience*, 22(5):419–437, May 1992.

[HIS78] H. B. Hunt III and T. G Szymanski. Corrigendum to 'Lower bounds and reductions between grammar problems'. *Journal of the ACM*, 25:687–688, 1978.

[HKR89] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *ACM SIGPLAN Notices*, 24(7):179–191, July 1989.

[Hor76] J. J. Horning. LR grammars and analysers. In F. L. Bauer and J. Eickel, editors, *Compiler Construction – an Advanced Course*, pages 85–108. Springer Verlag, Berlin, second edition, 1976.

[Hor93] R. Nigel Horspool. Recursive ascent-descent parsing. *Computer Languages*, 18(1):1–15, 1993.

[HRS84] K. Hammond and V. J. Rayward-Smith. A survey on syntactic error recovery and repair. *Computer Languages*, 9(1):51–68, 1984.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[IF83] Kenzo Inoue and Fukumi Fujiwara. On LLC(k) parsing method of LR(k) grammars. *J. Inform. Process.*, 6(4):206–217, 1983.

[IM70] J. D. Ichbiah and S. P. Morse. A technique for generating almost optimal Floyd-Evans productions for precedence grammars. *Communications of the ACM*, 13(8):501–508, 1970.

[Iro64] E. T. Irons. Structural connections in formal languages. *Communications of the ACM*, 7(2):62–67, 1964.

[Ive87] Fred Ives. Response to remarks on recent algorithms for LALR lookahead sets. *ACM SIGPLAN Notices*, 22(8):99–104, 1987.

[JK75] S. Jarzabek and T. Krawczyk. LL-regular grammars. *Information Processing Letters*, 4(2):31–37, 1975.

[JM80] N. D. Jones and M. Madsen. Attribute-influenced LR parsing. Lecture Notes in Computer Science, vol. 94, pages 393–407. Springer, 1980.

[Joh75] Stephen C. Johnson. Yacc—yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, July 1975.

[Jol74] Marc L. Joliat. Practical minimization of LR(k) parser tables. In J. L. Rosenfeld, editor, *Information Processing 74*, pages 376–380. North Holland, 1974.

[KA88] F. E. J. Kruseman Aretz. On a recursive ascent parser. *Information Processing Letters*, 29(4):201–206, November 1988.

[KHZ82] U. Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*. Springer Verlag, 1982.

[KM81] Bent Bruun Kristensen and Ole Lehrmann Madsen. Methods for computing LALR(k) lookahead. *ACM Transactions on Programming Languages and Systems*, 3(1):60–82, January 1981.

[KM89] E. Klein and M. Martin. The parser generating system PGS. *Software—Practice and Experience*, 19(11):1015–1028, 1989.

[KNPS88] K. Koskimies, O. Nurmi, J. Paakki, and S. Sippu. The design of a language processor generator. *Software—Practice and Experience*, 18:107–135, February 1988.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[Knu68] D. E. Knuth. Semantics of context-free languages. *Math. Syst. Th.*, 2(2):127–145, 1968. Correction ibid. Vol. 5, No. 1, pp. 95–95, 1971.

[KPJ98] Uwe Kastens, Peter Pfahler, and Matthias Jung. The Eli system. In Kai Koskimies, editor, *Compiler Construction, 7th International Conference, CC'98*, Lecture Notes in Computer Science, vol. 1383, pages 294–297. Springer, 1998.

[KSS79] Kai Koskimies and Eljas Soisalon-Soininen. On a method for optimizing LR parsers. *Intern. J. Computer Math.*, A7:287–295, 1979.

[LaL77] Wilf R. LaLonde. Regular right part grammars and their parsers. *Communications of the ACM*, 20(10):731–741, October 1977.

[Lan74] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Automata, Languages and Programming*, Lecture Notes in Computer Science, vol. 14, pages 255–269. Springer-Verlag, Berlin, 1974.

[LB74] O. Lecarme and G. V. Bochmann. A (truly) usable and portable compiler writing system. In J. L. Rosenfeld, editor, *Information Processing '74*, pages 218–221. North Holland, 1974.

[LC94] Myung-Joon Lee and Kwang-Moo Choe. Boundedly LR(k)-conflictable grammars. *Acta Informatica*, 31:261–283, 1994.

[LdVHH78] J. Lewi, K. de Vlaminck, J. Huens, and M. Huybrechts. The ELL(1) parser generator and the error-recovery mechanism. *Acta Informatica*, 10:209–228, 1978.

[LK97] Gyung-Ok Lee and Do-Hyung Kim. Characterization of extended LR(k) grammars. *Information Processing Letters*, 64:75–82, 1997.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly and Associates, 2nd edition, 1992.

[Loe70] Jacques Loeckx. An algorithm for the construction of bounded-context parsers. *Communications of the ACM*, 13(5):297–307, May 1970.

[LRS74] P. M. Lewis, D. J. Rosenkratz, and D. J. Stearns. Attributed translations. *Journal of Computer and System Sciences*, 9:279–307, 1974.

[LRS76] P. M. Lewis, D. J. Rosenkratz, and D. J. Stearns. *Compiler Design Theory*. Addison-Wesley, 1976.

[LS68] P. M. Lewis II and R. E. Stearns. Syntax-directed transduction. *Journal of the ACM*, 15(3):465–488, 1968.

[Mau86] Carole Maury. *Traitement automatique des erreurs syntaxiques dans un système d'écriture de compilateurs*. PhD thesis, Université de Nice, France, 1986. In French.

[Mau92] Peter M. Maurer. The design and implementation of a grammar-based data generator. *Software—Practice and Experience*, 23(3):223–244, 1992.

[Mer93] Gary H. Merrill. Parsing non-LR(k) grammars with yacc. *Software—Practice and Experience*, 23(8):829–850, August 1993.

[MF88] Jon Mauney and Charles N. Fischer. Determining the extent of lookahead in syntactic error repair. *ACM Transactions on Programming Languages and Systems*, 10(3):456–469, July 1988.

[MHW70] W. M. McKeeman, J. J. Horning, and D. B. Wortman. *A Compiler Generator*. Prentice-Hall, 1970.

[Mic78] M. Dennis Mickunas. Automatic error recovery for LR parsers. *Communications of the ACM*, 21(6):459–465, June 1978.

[Mil91] Masoud T. Milani. On the descriptive power of simple precedence grammars. *Intern. J. Computer Math.*, 39:29–49, 1991.

[MLS76] M. D. Mickunas, R. L. Lancaster, and V. B. Schneider. Transforming LR(k) grammars to LR(1), SLR(1) and $(1,1)$ bounded right-context grammars. *Journal of the ACM*, 23(3):511–533, July 1976.

[Mol80] K. R. Moll. Left context precedence grammars. *Acta Informatica*, 14:317–335, 1980.

[Mös91] Hanspeter Mössenböck. Coco/R - A generator for production quality compilers. In Dieter Hammer, editor, *Compiler Compilers, Third International Workshop on Compiler Construction, CC'90*, Lecture Notes in Computer Science, vol. 477. Springer, 1991.

[MW90] Masoud T. Milani and David A. Workman. Epsilon weak precedence grammars and languages. *Informatique théorique et Applications/Theoretical Informatics and Applications*, 24(3):241–266, 1990.

[Nij88] Anton Nijholt. *Computers and Languages: Theory and Practice*. North-Holland, 1988.

[NS86] Ikuo Nakata and Masataka Sassa. Generation of efficient LALR parsers for regular right part grammars. *Acta Informatica*, 23:149–162, 1986.

[Pag70] David Pager. A solution to an open problem by Knuth. *Information and Control*, 17:462–473, 1970.

[Pag74] David Pager. On eliminating unit productions from LR(k) parsers. In J. Loeckx, editor, *Automata, Languages and Programming*, Lecture Notes in Compter Science, vol. 14, pages 242–254. Springer-Verlag, Berlin, 1974.

[Pag77a] David Pager. Eliminating unit productions from LR parsers. *Acta Informatica*, 9:31–59, 1977.

[Pag77b] David Pager. The lane-tracing algorithm for constructing LR(k) parsers and ways of enhancing its efficiency. *Information Sciences*, 12:19–42, 1977.

[Pag77c] David Pager. A practical general method for constructing LR(k) parsers. *Acta Informatica*, 7:249–268, 1977.

[Pai64] C. Pair. Trees, pushdown stores and compilation. *RFTI—Chiffres*, 7(3):199–216, 1964.

[Par93] Terence John Parr. *Obtaining Practical Variants for LL(k) and LR(k) for $k > 1$ by splitting the atomic k-tuple.* PhD thesis, Purdue University, West Lafayette, Indiana, 1993.

[Par96] Terence John Parr. *Language Translation Using PCCTS & C++: A Reference Guide.* Automata Publishing, San Jose, CA, 1996.

[PB80] P. Purdom and C. A. Brown. Semantic routines and LR(k) parsers. *Acta Informatica*, 14:299–315, 1980.

[PB81] Paul W. Purdom and Cynthia A. Brown. Parsing extended LR(k) grammars. *Acta Informatica*, 15:115–127, 1981.

[PCC85] Joseph C. H. Park, K. M. Choe, and C. H. Chang. A new analysis of LALR formalisms. *ACM Transactions on Programming Languages and Systems*, 7(1):159–175, January 1985.

[PD78] Thomas J. Pennello and Frank DeRemer. A forward move algorithm for LR error recovery. In *Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 241–254, January 1978.

[PDC92] Terence J. Parr, Henry G. Dietz, and Will E. Cohen. PCCTS 1.00: The Purdue compiler construction tool set. *ACM SIGPLAN Notices*, 27(2):88–165, 1992.

[Pen86] Thomas J. Pennello. Very fast LR parsing. *ACM SIGPLAN Notices*, 21(7):145–151, July 1986.

[Pez89] Stylianos Pezaris. Shift-reduce conflicts in LR parsers. *ACM SIGPLAN Notices*, 24(11):94–95, November 1989.

[PLC93] Woo-Jun Park, Myung-Joon Lee, and Kwang-Moo Choe. On the reduction of LR(k) parsers. *Information Processing Letters*, 47:245–251, 1993.

[Pop79] D. A. Poplawski. On LL-regular grammars. *Journal of Computer and System Sciences*, 18:218–227, 1979.

[PQ95] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software—Practice and Experience*, 25(7):789–810, 1995.

[PQ96] T. J. Parr and R. W. Quong. LL and LR translators need $k > 1$ lookahead. *ACM SIGPLAN Notices*, 31(2):27–34, 1996.

[Pur74] Paul Purdom. The size of LALR(1) parsers. *BIT*, 14:326–337, 1974.

[RD78] G. David Ripley and Frederick C. Druseikis. A statistical analysis of syntax errors. *Computer Languages*, 3:227–240, 1978.

[Rei98] Alejandro Reina, September 1998. Personnal communication.

[Ric85] Helmut Richter. Noncorrecting syntax error recovery. *ACM Transactions on Programming Languages and Systems*, 7(3):478–489, July 1985.

[Rip79] G. David Ripley. Pascal syntax errors data base. Technical report, RCA Laboratories, Princeton, NJ 08540, USA, 1979.

[RL70] D. J. Rosenkrantz and P. M. Lewis II. Deterministic left-corner parsing. In *IEEE Conference Record 11th Annual Symposium on Switching and Automata Theory*, pages 139–152, 1970.

[RM89] P. Rechenberg and H. Mössenböck. *A Compiler Generator for Microcomputers*. Prentice-Hall, 1989.

[Rob88] George H. Roberts. Recursive ascent: an LR analog to recursive descent. *ACM SIGPLAN Notices*, 23(8):23–29, August 1988.

[Rör85] Johannes Rörich. Methods for the automatic construction of error correcting parsers. *Acta Informatica*, 13(2):115–139, February 1985.

[RS70] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top-down grammars. *Information and Control*, 17:226–256, 1970.

[Sak62] Itiroo Sakai. Syntax in universal translation. In *Proceedings 1961 International Conference on Machine Translation of Languages and Applied Language Analysis*, pages 593–608. Her Majesty's Stationary Office, London, 1962.

[SC93] Heung-Chul Shin and Kwang-Moo Choe. An improved LALR(k) parser generation for regular right part grammars. *Information Processing Letters*, 47:123–129, 1993.

[Sch] L. Schmitz. *Theorie Kettenfreier LR-Zerteiler*. PhD thesis, Fachbereich Informatik, Hochschule der Bundeswehr, Munich.

[Sch84] Lothar Schmitz. On the correct elimination of chain productions from LR parsers. *Intern. J. Computer Math.*, 15(2):99–116, 1984.

[Sei86] B. Seité. *Une extension de Yacc, pour analyse des grammaires LRR, qui autorise un examen non borné du contexte droit et de la pile*. PhD thesis, University of Paris VII, April 1986.

[Sei87] B. Seité. A Yacc extension for LRR grammar parsing. *Theoretical Computer Science*, 52:91–143, 1987.

[Shy77] R. K. Shyamasundar. Precedence parsing using Dömölki's algorithm. *Intern. J. Computer Math.*, A6:105–114, 1977.

[Sik97] Klaas Sikkel. *Parsing Schemata: A Framework for Specification and Analysis of Parsing Algorithms*. Springer, 1997.

[SIN87] M. Sassa, H. Ishizuka, and I. Nakata. ECLR-attributed grammars: a practical class of LR-attributed grammars. *Information Processing Letters*, 24:31–41, 1987.

[SIN95] M. Sassa, H. Ishizuka, and I. Nakata. Rie, a compiler generator based on a one-pass-type attribute grammar. *Software—Practice and Experience*, 25(3):229–250, March 1995.

[SMS73] Shoji Sekimoto, Kuniaki Mukai, and Masaru Sudo. A method of minimizing LR(k) parsers. *Systems, Computers and Control*, 4(5):73–80, 1973.

[Spe81] David Spector. Full LR(1) parser generation. *ACM SIGPLAN Notices*, 16(8):58–66, 1981.

[Spe88] David Spector. Efficient full LR(1) parser generation. *ACM SIGPLAN Notices*, 23(12):143–150, 1988.

[SS80] Eljas Soisalon-Soininen. On the space-optimizing effect of eliminating single productions from LR parsers. *Acta Informatica*, 14:157–174, 1980.

[SS82] Eljas Soisalon-Soininen. Inessential error entries and their use in LR parser optimization. *ACM Transactions on Programming Languages and Systems*, 4(2):179–195, April 1982.

[SSS90] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory.* Springer-Verlag, 1988–1990.

[SSU79] Eljas Soisalon-Soininen and Esko Ukkonen. A method for transforming grammars into LL(k) form. *Acta Informatica*, 12:339–369, 1979.

[SW76] Thomas G. Szymanski and John H. Williams. Non-canonical extensions of bottom-up parsing techniques. *SIAM Journal on Computing*, 5(2):231–250, June 1976.

[Ter97] P. D. Terry. *Compilers and Compiler Generators: An Introduction with C++.* International Thomson Computer Press, 1997.

[Tok81] Takehiro Tokuda. Eliminating unit reductions from LR(k) parsers using minimum contexts. *Acta Informatica*, 15:447–470, 1981.

[Tom86] Masaru Tomita. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, Boston, 1986.

[Tom87] Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1–2):31–46, January-June 1987.

[TS85] Jean-Paul Tremblay and P. G. Sorenson. *The Theory and Practice of Compiler Writing.* McGraw-Hill, 1985.

[TSKS83] A. S. Tanenbaum, H. Van Staveren, E. G. Keizer, and J. V. Stevenson. A practical tool kit for making portable compilers. *Communications of the ACM*, 26(9):654–660, 1983.

[TY79] Robert Endre Tarjan and Chi-Chih Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November 1979.

[Ukk79] Esko Ukkonen. A modification of the LR(k) method for constructing compact bottom-up parsers. In Hermann A. Maurer, editor, *Automata, Languages and Programming*, Lecture Notes in Computer Science, vol. 71, pages 646–658. Springer-Verlag, 1979.

[Ukk83] Esko Ukkonen. Lower bounds on the size of deterministic parsers. *Journal of Computer and System Sciences*, 26:152–170, 1983.

[Ung68] S. H. Unger. A global parser for context-free phrase structure grammars. *Communications of the ACM*, 11(4):240–247, April 1968.

[Wet77] Charles Wetherell. Why automatic error correctors fail. *Computer Languages (Elmsford, N.Y.)*, 2:179–186, 1977.

[WG84] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer Verlag, 1984.

[Wir96] Niklaus Wirth. *Compiler Construction*. Addison-Wesley, 1996.

[Wit88] Jacek Witaszek. A practical method for finding the optimum postponement transformation for LR(k) parsers. *Information Processing Letters*, 27(2):63–67, February 1988.

[WM95] Reinhald Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

[Woo70] D. Wood. The theory of left factored languages, Part II. *Computer J.*, 13(1):55–62, 1970.

[Wor81] David A. Workman. SR(s, k) parsers: A class of shift-reduce bounded-context parsers. *Journal of Computer and System Sciences*, 22(1):178–197, 1981.

[WS81] Charles Wetherell and A. Shannon. LR—automatic parser generator and LR(1) parser. *IEEE Transactions on Software Engineering*, SE-7(3):274–278, May 1981.

[WW66] Niklaus Wirth and Helmut Weber. EULER: A generalization of ALGOL and its formal definition. Parts I and II. *Communications of the ACM*, 9(1):13–23 and 89–99, 1966.

[Wyr83] Piotr Wyrostek. On the 'correct prefix property' in precedence parsers. *Information Processing Letters*, 17(3):161–165, October 1983.

[Wyr86] Piotr Wyrostek. Precedence technique is not worse than SLR(1). *Acta Informatica*, 23:361–392, 1986.

[You67] D. H. Younger. Recognition of context-free languages in time n^3. *Information and Control*, 10(2):189–208, February 1967.

Abstract

Since the invention by Donald Knuth of LR(k) parsing — the most powerful practical method that permits to parse the input text as it is being read—, much research effort has concentrated into developing efficient techniques to cope with the size problem when $k \geq 1$, which have not been found however sufficiently satisfactory in practice, where a simpler but less powerful LALR(1) method is firmly established.
This thesis claims that this situation is not entirely satisfactory for programming language parser generation, and proposes a new, discriminating-reverse approach to full LR(k) parsing, by building a finite-state automaton to explore a minimal stack suffix from its top to discriminate each bottom-up parsing action, whose correctness and parsing time linearity are here formally proved.
The method has been implemented and evaluated for practical grammars, resulting in parsers with full LR(1) power considerably smaller than parsers produced by LALR(1)-based generators, as `yacc`, and with a negligible stack exploration overhead.
This has been possible because classical direct parsers build a recognizer for the full bottom-to-top stack-plus-lookahead language, while the discriminating reverse parser typically uses a few topmost stack symbols plus the lookahead, where most parsing information is concentrated, as it was evident to the early designers of precedence parsers.

Keywords: parsing, LR, discriminating reverse, formal languages, compilers