Programming Languages

T.A. Standish Editor

Deterministic Parsing of Ambiguous Grammars

A.V. Aho and S.C. Johnson
Bell Laboratories
and
J.D. Ullman
Princeton University

Methods of describing the syntax of programming languages in ways that are more flexible and natural than conventional BNF descriptions are considered. These methods involve the use of ambiguous context-free grammars together with rules to resolve syntactic ambiguities. It is shown how efficient LR and LL parsers can be constructed directly from certain classes of these specifications.

Key Words and Phrases: programming language specification, parser generation, translator writing systems, syntax analysis, LR parsing, LL parsing, ambiguous grammars

CR Categories: 4.12, 4.22, 5.23

A preliminary version of this paper was presented at the ACM Symposium on the Principles of Programming Languages, October 1973.

The work of J. D. Ullman was partially supported by NSF Grants GJ474 and GJ35570. Authors' addresses: A. V. Aho and S. C. Johnson, Bell Laboratories 2C522, 600 Mountain Avenue, Murray Hill, NJ 07974; J. D. Ullman, Department of Electrical Engineering, Princeton University, Princeton, NJ 08540.

1. Introduction

Using a context-free grammar to help define the syntax of a programming language offers a number of advantages. First, a grammar provides a clear and precise description of those strings that make up a programming language. Second, a grammar is a particularly useful tool for a language designer who wishes to produce a language whose syntax is clean and free of special-case anomalies. Finally, if the grammar is carefully designed, it is possible to construct a parser for the language automatically from the grammar. In fact, several compiler-compilers have been built which incorporate such a facility [1, 7, 11, 25, 26].

Considerable effort has gone into developing efficient parsing methods such as the LL, LR, and precedence based techniques (see [2]). Much of this work has focused on mechanical ways of producing deterministic (no backtracking) parsers for certain classes of context-free grammars. Once one has a grammar in the required class an efficient parser can be constructed automatically.

Unfortunately, the most natural grammar describing a language is frequently not in the class of grammars for which we were able to construct efficient parsers mechanically. With many automatic translator writing systems, the grammar must be rewritten by hand to make it fall into the class of grammars acceptable to the translator writing system. While the necessary manipulations are not hard to do after some practice, they often expand the size of the grammar and introduce seemingly extraneous symbols.

In this paper we suggest using an ambiguous context-free grammar together with disambiguating rules as a method of specifying the syntax of a programming language. We feel that a number of common syntactic constructs in programming languages can be specified more naturally and succinctly by this technique than by using an equivalent unambiguous grammar. The disambiguating rules are based on familiar concepts from operator precedence [13] and recursive descent [4] parsing.

The definitions of LR and LL grammars exclude all ambiguous grammars. However, we shall show how the LR and LL parser construction methods can be easily extended to construct parsers for some useful classes of ambiguous grammars. The resulting parsers are often smaller and more efficient than the corresponding parsers constructed from unambiguous grammars.

In the last part of this paper, we define a generalization of the class of LL languages. This class includes the familiar **if-then, if-then-else** construct not included in the traditional LL class.

2. A Method for Specifying Languages

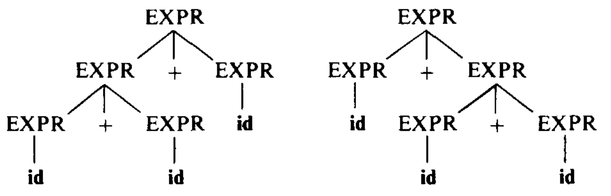
We shall discuss our ideas in an informal manner, assuming the reader is familiar with the rudiments of context-free grammars. (See [2] or [16] for basic definitions.) The term "grammar" by itself hereafter means context-free grammar with no useless symbols.

We recall a grammar is *unambiguous* if and only if no string in the language generated by the grammar has more than one parse tree (or equivalently has more than one rightmost derivation, or equivalently more than one leftmost derivation). Otherwise, the grammar is said to be *ambiguous.*

Example 1. Consider the following grammar G_1 for simple arithmetic expressions involving the operators + and $*$.

$$\begin{aligned} G_1\colon\ \mathrm{EXPR} &\rightarrow \mathrm{EXPR} + \mathrm{EXPR}\\ &\mid \mathrm{EXPR} * \mathrm{EXPR}\\ &\mid (\mathrm{EXPR})\\ &\mid \mathbf{id} \end{aligned}$$

G_1 is ambiguous. For example, the string **id** + **id** + **id** has the following two parse trees:



□

The usual approach to this problem calls for rewriting the grammar to eliminate such ambiguities. We propose instead to select the proper tree by defining functions that accept or reject parse trees of the grammar, so each string has at most one parse tree acceptable to all these functions. We shall call each such function a *disambiguating rule.*

In this paper we shall specify disambiguating rules rather informally. For example, for G_1 we can specify a disambiguating rule that + and $*$ are to be left associative and that $*$ is to have higher precedence than +. Thus this disambiguating rule accepts the first parse tree and rejects the second parse tree shown above.

Defining a language in terms of an ambiguous grammar plus a set of disambiguating rules offers certain advantages. It is possible to find a grammar plus a disambiguating rule that will define a language that is not context-free [15]. However, this is not our intent here. We shall use ambiguous grammars and disambiguating rules to specify traditional programming language constructs more economically and more clearly than would be possible with an equivalent unambiguous grammar.

Example 2. The traditional specification of the simple arithmetic expressions of Example 1 (assuming the disambiguating rule mentioned above) might use the following unambiguous grammar:

$$\begin{aligned} G_2\colon\ \mathrm{EXPR} &\rightarrow \mathrm{EXPR} + \mathrm{TERM}\\ &\mid \mathrm{TERM}\\ \mathrm{TERM} &\rightarrow \mathrm{TERM} * \mathrm{FACTOR}\\ &\mid \mathrm{FACTOR}\\ \mathrm{FACTOR} &\rightarrow (\mathrm{EXPR})\\ &\mid \mathbf{id} \end{aligned}$$

□

This grammar has two more nonterminals, TERM and FACTOR, and two new productions, EXPR → TERM and TERM → FACTOR. Productions of this form with a single nonterminal on the right side are called *single* productions. The introduction of the nonterminals TERM and FACTOR ensures that + and $*$ are each left associative; the single productions do nothing but enforce the operator precedence. These changes make G_2 larger than G_1, and an LR parser directly constructed from G_2 would spend a significant fraction of its time dealing with these single productions. An LR parser constructed directly from G_1 using the techniques of Section 5 would avoid the unnecessary steps.

These remarks assume more significance when we are dealing with programming languages having operators on as many as 12 or more different precedence levels.

Example 3. Suppose we have expressions generated by the following grammar:

$$\begin{aligned} G_3\colon\ \mathrm{EXPR} &\rightarrow \mathrm{EXPR} < \mathrm{EXPR}\\ &\mid \mathrm{EXPR} + \mathrm{EXPR}\\ &\mid \mathrm{EXPR} - \mathrm{EXPR}\\ &\mid \mathrm{EXPR} * \mathrm{EXPR}\\ &\mid \mathrm{EXPR}\ /\ \mathrm{EXPR}\\ &\mid \mathrm{EXPR} \uparrow \mathrm{EXPR}\\ &\mid (\mathrm{EXPR})\\ &\mid \mathbf{id} \end{aligned}$$

Disambiguating rules:	$<$	nonassociating
	$+ \quad -$	left associative
	$* \quad /$	left associative
	$\uparrow$	right associative

Following the grammar are disambiguating rules consisting of a listing of the operators in the grammar; the operators in each line are of higher precedence than all preceding operators. In addition, associated with each operator or set of operators is the associativity of the operator.

For example, the operator $<$ is indicated to be nonassociating, meaning that an expression of the form EXPR $<$ EXPR $<$ EXPR is invalid. The operator $\uparrow$ (exponentiation) is right associative, meaning EXPR $\uparrow$ EXPR $\uparrow$ EXPR is to be interpreted as (EXPR $\uparrow$ (EXPR $\uparrow$ EXPR))

The following unambiguous grammar generates the same language:

$$\begin{aligned} G_4\colon\ E_1 &\rightarrow E_2 < E_2\\ &\mid E_2\\ E_2 &\rightarrow E_2 + E_3\\ &\mid E_2 - E_3\\ &\mid E_3\\ E_3 &\rightarrow E_3 * E_4\\ &\mid E_3\ /\ E_4\\ &\mid E_4\\ E_4 &\rightarrow E_5 \uparrow E_4\\ &\mid E_5\\ E_5 &\rightarrow (E_1)\\ &\mid \mathbf{id} \end{aligned}$$

G_4 has five nonterminals and four single productions $E_i \rightarrow E_{i+1}$, $1 \leq i \leq 4$. □

In most programming languages, expressions are the basic building blocks of statements and programs, so a parser for an unambiguous grammar such as G_4 would spend much of its time dealing with single productions. We could speed-up such a parser by modifying it to avoid applying single productions that have no associated semantic actions [3, 5, 27]. (Joliat [17] observed a 47 percent improvement in his parser for XPL by such an optimization.) Using the techniques of Section 5 we can construct optimized parsers that eliminate most of the reductions by single productions directly from the ambiguous grammars and disambiguating rules, eliminating the optimization step.

Our next example shows how the famous dangling-**else** construct can be handled by an ambiguous grammar and a disambiguating rule suggested by recursive descent parsing.

Example 4. Grammar G_5 generates **if-then, if-then-else** statements. b stands for a Boolean expression and **others** for any other statement.

$$
\begin{aligned}
G_5\colon\ S &\rightarrow \textbf{if } b \textbf{ then } S\\
&\mid \textbf{if } b \textbf{ then } S \textbf{ else } S\\
&\mid \textbf{others}
\end{aligned}
$$

Disambiguating rule: Associate each **else** with the closest unmatched **then**.

This disambiguating rule is the one normally used in languages with this syntactic construct. G_6 is an equivalent unambiguous grammar that incorporates this disambiguating rule into the productions themselves.

$$
\begin{aligned}
G_6\colon\ S &\rightarrow A\\
&\mid B\\
A &\rightarrow \textbf{if } b \textbf{ then } S\\
&\mid \textbf{if } b \textbf{ then } B \textbf{ else } A\\
B &\rightarrow \textbf{if } b \textbf{ then } B \textbf{ else } B\\
&\mid \textbf{others}
\end{aligned}
$$

□

One other application of the use of ambiguous grammars is in recognizing special cases for code optimization. For example, in addition to the productions for the general case we might also include productions for special cases of expressions and statements for which we can construct highly efficient object code. Our disambiguating rule would be to use these special productions wherever possible. The next example shows an instance of this technique in a rather interesting context.

Example 5. In developing a typesetting language for mathematics B. W. Kernighan and L. L. Cherry [19] wished to have superscripts align above subscripts in expressions such as x_i^2. They used productions of the form

$$
\begin{aligned}
G_7\colon\ \text{EXPR} &\rightarrow \text{EXPR } \textbf{sub} \text{ EXPR}\\
&\mid \text{EXPR } \textbf{sup} \text{ EXPR}\\
&\mid \text{EXPR } \textbf{sub} \text{ EXPR } \textbf{sup} \text{ EXPR}
\end{aligned}
$$

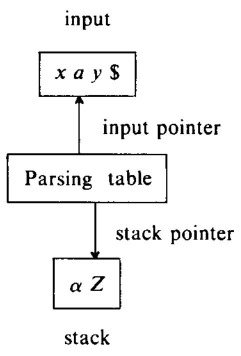
to achieve this effect along with the disambiguating rule that the last production be used wherever possible. Thus the expression of the form x **sub** i **sup** 2 can be interpreted as (1) ((x **sub** i) **sup** 2), (2) (x **sub** (i **sup** 2)), or (3) (x **sub** i **sup** 2). In their system the expression x **sub** i **sup** 2 would be translated into the output $x_i{}^2$ under the first interpretation, the output x_{i^2} under the second interpretation, and the output x_i^2 under the third interpretation. The disambiguating rule causes the third output to be produced (other disambiguating rules resolve the relative precedences and associativities of the operators **sub** and **sup**). The compiler for the typesetting language was produced by YACC, a compiler-compiler written by S. C. Johnson employing the ambiguous LR techniques discussed in this paper. □

3. Simple LR(1) Parsers

We shall discuss LR(1) parsing first since it is one of the most widely applicable deterministic parsing techniques known. In this Section we present the general form of an LR parser. In Section 4 we show how valid LR parsers can be constructed for a certain class of grammars called the simple LR(1) grammars, and in Section 5 we show how an LR parser can be constructed for certain ambiguous grammars using disambiguating rules to resolve parsing action conflicts.

An LR parser can be pictured as having the form shown in Figure 1.

Fig. 1. An LR parser.



The input to an LR parser consists of the string to be parsed followed by a right endmarker \$. The input pointer indicates the current input symbol, initially the leftmost. The parser also has a parsing table and a stack; we draw the top of the stack at the right. Moreover, at every point in time if the input (xay in Figure 1) is in the language, then αZ, the sequence of grammar symbols on the stack, followed by ay, the portion of the input not yet used, will be called the *right sentential form represented by the parser*. In Figure 1, αZay is the right sentential form represented by the parser.

However, in addition to putting grammar symbols on the stack an LR parser places special symbols called *states* on the stack. In fact, at all times the contents of the stack is a sequence of the form $s_0X_1s_1X_2s_2...X_ms_m$ where each s_i is a state and each X_i a grammar symbol. The next move of the parser is dictated by s_m, the state on top of the stack, and a, the current input symbol. This move is found in the parsing table, which consists of two parts, an *action table* and a *goto table*. The type of move is determined by the entry for s_m and a in the action table. The move can be one of four types:

1. *Shift s*. The current input symbol a is shifted onto the stack and the state numbered s is then placed on top of the stack. This move has the effect of advancing the input pointer one symbol to the right.
2. *Reduce by production $A \rightarrow \beta$*. Suppose the right side β is of length $r \geq 0$. Then $2r$ symbols are popped off the stack. That is, $X_{m-r+1}s_{m-r+1} \ldots X_m s_m$ is removed from the stack, leaving $s_0 X_1 s_1 \ldots X_{m-r} s_{m-r}$. The nonterminal A is placed on top of the stack. The goto table for s_{m-r} is then consulted and the state in the entry for s_{m-r} and A is placed on top of the stack. In a properly constructed LR parser, when a reduction by the production $A \rightarrow \beta$ is called for, $X_{m-r+1} \ldots X_m$, the string of grammar symbols removed from the stack, will match β, the right side of the production.
3. *Accept*. The parser halts and announces successful completion of parsing.
4. *Error*. The input string is not in the language being parsed; the parser transfers to an error recovery routine.

In a shift move the right sentential form represented by the parser does not change. In a reduce move it does. Suppose

$$S \overset{*}{\Rightarrow}_{\text{rm}} \alpha A a y \Rightarrow_{\text{rm}} \alpha \beta a y \overset{*}{\Rightarrow}_{\text{rm}} x a y$$

is a rightmost derivation in the grammar at hand. An LR parser would represent the right sentential form $\alpha\beta ay$ with $\alpha\beta$ on the stack and ay as the unexpended input. After the reduction by the production $A \rightarrow \beta$, the parser would represent the right sentential form αAay with αA on the stack.

Initially an LR parser has the input string to be parsed on the input tape with the input pointer at the leftmost symbol on the input. The stack contains only the initial state s_0. The parser then makes a sequence of moves until either an accept action or error action is called for. The sequence of reductions made by the parser up to an accept action is the *parse produced by* the parser for the input string.

Example 6. Consider the following expression grammar G_8, which is G_2 with certain grammar symbols abbreviated, and productions numbered as shown:

G_8: (1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow a$

A parsing action table for G_8 is shown in Table I. As before, \$ represents the right endmarker. In this table, "—" stands for "error," "sh i" stands for "shift i," and "red i" stands for "reduce by production numbered (i)."

Table II is the goto table. The blank entries in the goto table are never referenced. An LR parser using these action and goto tables would parse the string

Table I

	Action table					
State	+	*	(	)	a	\$
0	—	—	sh 4	—	sh 5	—
1	sh 6	—	—	—	—	accept
2	red 2	sh 7	—	red 2	—	red 2
3	red 4	red 4	—	red 4	—	red 4
4	—	—	sh 4	—	sh 5	—
5	red 6	red 6	—	red 6	—	red 6
6	—	—	sh 4	—	sh 5	—
7	—	—	sh 4	—	sh 5	—
8	sh 6	—	—	sh 11	—	—
9	red 1	sh 7	—	red 1	—	red 1
10	red 3	red 3	—	red 3	—	red 3
11	red 5	red 5	—	red 5	—	red 5

Table II

	Goto table		
State	E	T	F
0	1	2	3
1			
2			
3			
4	8	2	3
5			
6		9	3
7			10
8			
9			
10			
11			

$a+a*a$ in the manner shown in Figure 2. Notice that the parse produced by this parser for the input string $a+a*a$ is the reverse of the sequence of productions used in a rightmost derivation of $a+a*a$:

$$E \Rightarrow_{\text{rm}} E+T \Rightarrow_{\text{rm}} E+T*F \Rightarrow_{\text{rm}} E+T*a \Rightarrow_{\text{rm}} E+F*a \Rightarrow_{\text{rm}} E+a*a \Rightarrow_{\text{rm}} T+a*a \Rightarrow_{\text{rm}} F+a*a \Rightarrow_{\text{rm}} a+a*a$$

□

The reader is referred to [1] or [2] for a more thorough discussion of LR parsing.

4. Construction of LR Parsers for Simple LR(1) Grammars

We shall call the sequence of productions used in the reverse of a rightmost derivation of a string w a *right parse* for w. We say an LR parser is *valid* for a grammar if for each string w the parser produces a right parse for w if w is in the language generated by the grammar, and an error indication if w is not in the language generated by the grammar. In this section we shall sketch a technique that will produce valid LR parsers for a class of grammars called the simple LR(1) grammars [7].

Let G be a grammar for which we wish to construct an LR parser. If X is a nonterminal or terminal in a grammar, we define FOLLOW(X) to be the set of termi-

Fig. 2.

Stack	Unexpended input	Right sentential form represented by the parser
0	$a+a*a\$$	$a+a*a$
0a5	$+a*a\$$	$a+a*a$
0F3	$+a*a\$$	F$+a*a$
0T2	$+a*a\$$	T$+a*a$
0E1	$+a*a\$$	E$+a*a$
0E1+6	$a*a\$$	E$+a*a$
0E1+6a5	$*a\$$	E$+a*a$
0E1+6F3	$*a\$$	E+F$*a$
0E1+6T9	$*a\$$	E+T$*a$
0E1+6T9*7	$a\$$	E+T$*a$
0E1+6T9*7a5	$	E+T$*a$
0E1+6T9*7F10	$	E+T*F
0E1+6T9	$	E+T
0E1	$	E

nals that can appear immediately to the right of X in a sentential form. If X can be the rightmost symbol in a sentential form, then we shall include \$ (the right endmarker) in FOLLOW(X).

To construct an LR parser for G, we first augment G with a new starting production of the form $S' \to S$ where S' is a new start symbol and S is the start symbol of G.

To determine the states of the parser we construct the *collection of sets of items* for G. An *item* is a production of G with a dot somewhere in the right side. The dot indicates how much of the right side will have been recognized when the parser is in the state associated with that item.

We define a function CLOSURE on sets of items. If I is a set of items, CLOSURE(I) is the smallest set of items that contains I and has the property:

If $A \to \alpha.B\gamma$ is in CLOSURE(I) and $B \to \beta$ is a production of G, then $B \to .\beta$ is in CLOSURE(I).

An easy way to compute CLOSURE(I) is to begin with I and continue adding new items of the form $B \to .\beta$ to I for each production $B \to \beta$ in G and item $A \to \alpha.B\beta$ currently in I. We continue adding items to I until no more new items can be added. The resulting set is CLOSURE(I).

We need to introduce one other function, the GOTO function on sets of items. If I is a set of items and X a grammar symbol, then GOTO(I,X) $= J$ where J is the closure of the set of items

$$\{A \to \alpha X.\beta \mid A \to \alpha.X\beta \text{ is in } I\}.$$

Computing GOTO(I,X) is straightforward. We take each item in I with an X immediately to the right of the dot. We shift the dot past the X and add the resulting item to J. We then compute the closure of J.

To construct C, the collection of sets of items for the given grammar G, we begin with

$$C = \{I_0 = \text{CLOSURE}(\{S' \to .S\})\}.$$

Then for each set of items I in C and for each grammar symbol X, we compute GOTO(I,X) and add the resulting set of items to C if it is not already there. When no further sets of items can be added by this process, C is the collection of sets of items for the given grammar.

Example 7. The collection of sets of items for G_8 is given below:

I_0: $E' \to .E$
$\quad E \to .E+T$
$\quad E \to .T$
$\quad T \to .T*F$
$\quad T \to .F$
$\quad F \to .(E)$
$\quad F \to .a$

I_1: $E' \to E.$
$\quad E \to E.+T$

I_2: $E \to T.$
$\quad T \to T.*F$

I_3: $T \to F.$

I_4: $F \to (.E)$
$\quad E \to .E+T$
$\quad E \to .T$
$\quad T \to .T*F$
$\quad T \to .F$
$\quad F \to .(E)$
$\quad F \to .a$

I_5: $F \to a.$

I_6: $E \to E+.T$
$\quad T \to .T*F$
$\quad T \to .F$
$\quad F \to .(E)$
$\quad F \to .a$

I_7: $T \to T*.F$
$\quad F \to .(E)$
$\quad F \to .a$

I_8: $F \to (E.)$
$\quad E \to E.+T$

I_9: $E \to E+T.$
$\quad T \to T.*F$

I_{10}: $T \to T*F.$

I_{11}: $F \to (E).$

□

Given C, the collection of sets of items for the given grammar G, we can construct the action and goto tables of an LR parser for G as follows. For each set of items in C, there will be a state. Let state i correspond to the set of items I_i. The parsing action table entries for state i are determined as follows:

1. If I_i contains an item of the form $A \to \alpha.a\beta$ where a is a terminal, then the action of state i on input a is "shift j," where j is the state associated with the set of items GOTO(I_i,a).
2. If I_i contains the item $A \to \alpha.$ and a is in FOLLOW(A), then the action of state i on input a is "reduce by production $A \to \alpha$."
3. If I_i contains the item $S' \to S.$, then the action of state i on the right endmarker \$ is "accept."
4. Otherwise, the action of state s on input a is "error."

The goto table entry for state i on nonterminal A is simply the state associated with the set of items GOTO(I_i,A).

If at most one move is defined for each entry of the parsing action table, then the given grammar G is said to be *simple* LR(1). It can be shown that the LR parser so constructed is a valid parser for G. Moreover, the parser will announce error on input a after having scanned x, provided there is no string in $L(G)$ of the form xaz, for any z; that is, an error is announced as soon as possible.

Example 8. In Example 6 we gave the action table and goto table that would be computed from the collection of sets of items for G_8 given in Example 7. □

5. Resolving Parsing Action Conflicts

Let us now examine what happens when we try to build an LR parser for an ambiguous grammar. First consider the following grammar (which is G_1 with certain grammar symbols renamed):

G_9: (1) $E \rightarrow E + E$
(2) $E \rightarrow E * E$
(3) $E \rightarrow (E)$
(4) $E \rightarrow a$

The collection of sets of items for G_9 is:

I_0: $E' \rightarrow .E$
$E \rightarrow .E{+}E$
$E \rightarrow E{*}E$
$E \rightarrow .(E)$
$E \rightarrow .a$

I_1: $E' \rightarrow E.$
$E \rightarrow E.{+}E$
$E \rightarrow E.{*}E$

I_2: $E \rightarrow (.E)$
$E \rightarrow .E{+}E$
$E \rightarrow .E{*}E$
$E \rightarrow .(E)$
$E \rightarrow .a$

I_3: $E \rightarrow a.$

I_4: $E \rightarrow E{+}.E$
$E \rightarrow .E{+}E$
$E \rightarrow .E{*}E$
$E \rightarrow .(E)$
$E \rightarrow .a$

I_5: $E \rightarrow E{*}.E$
$E \rightarrow .E{+}E$
$E \rightarrow .E{*}E$
$E \rightarrow .(E)$
$E \rightarrow .a$

I_6: $E \rightarrow (E.)$
$E \rightarrow E.{+}E$
$E \rightarrow E.{*}E$

I_7: $E \rightarrow E{+}E.$
$E \rightarrow E.{+}E$
$E \rightarrow E.{*}E$

I_8: $E \rightarrow E{*}E.$
$E \rightarrow E.{+}E$
$E \rightarrow E.{*}E$

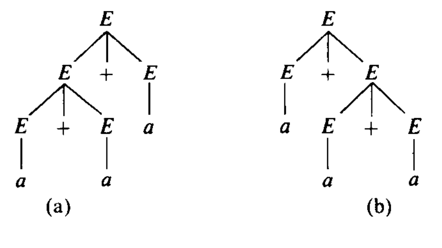
I_9: $E \rightarrow (E).$

As before, we can try to construct the parsing action and goto tables from these sets of items. However, sets of items I_7 and I_8 give rise to entries in the action table with more than one action. These multiply defined entries are represented by 1?, 2?, 3?, and 4? in Table III. Each of the multiply defined entries in this action table happens to represent a shift-reduce conflict:

1?: conflict between "shift 4" and "reduce 1"
2?: conflict between "shift 5" and "reduce 1"
3?: conflict between "shift 4" and "reduce 2"
4?: conflict between "shift 5" and "reduce 2"

In effect, the grammar G_9 will permit many different parses for certain input strings. The multiply defined entries in the action table represent choices which, when uniquely resolved, will result in the selection of one of the possible parses for a given input string.

As an example of these choices, we may observe that the input string $a{+}a$ will eventually cause the parser to place state 7 on top of the stack. If we replace the entry 1? in the action table by "reduce 1," then the input string $a{+}a{+}a$ will be parsed as (a) below.



That is, + would be treated as being left associative. On the other hand, if we replace 1? by "shift 4," then the same string would be parsed as (b) above. Here, + is treated as being right associative.

We could also replace 1? by an error entry. Doing so would make + be a nonassociating operator like .LT. in Fortran. The input string $a{+}a{+}a$ would become illegal.

Table III

	Action table						Goto table
State	+	*	(	)	a	$	E
0	—	—	sh 2	—	sh 3	—	1
1	sh 4	sh 5	—	—	—	accept	
2	—	—	sh 2	—	sh 3	—	6
3	red 4	red 4	—	red 4	—	red 4	
4	—	—	sh 2	—	sh 3	—	7
5	—	—	sh 2	—	sh 3	—	8
6	sh 4	sh 5	—	sh 9	—	—	
7	1?	2?	—	red 1	—	red 1	
8	3?	4?	—	red 2	—	red 2	
9	red 3	red 3	—	red 3	—	red 3	

Table IV

1?	2?	3?	4?	Disambiguating Rule
red 1	sh 5	red 2	red 2	The usual interpretation; + and * are left associative and * has higher precedence
red 1	red 1	sh 4	sh 5	+ left associative, * right associative, + has higher precedence.
red 1	red 1	red 2	red 2	+ and * evaluated left to right
sh 4	sh 5	sh 4	sh 5	+ and * evaluated right to left (as in APL)
—	—	—	—	precedence must be fully specified by parentheses.

Table V

	Action table				Goto table
State	a	b	c	$	S
0	sh 2	—	sh 3	—	1
1	—	—	—	accept	
2	sh 2	—	sh 3	—	4
3	—	red 3	—	red 3	
4	—	?	—	red 2	
5	sh 2	—	sh 3	—	6
6	—	red 1	—	red 1	

Thus the question of how we should resolve the parsing action conflicts present in 1? is directly connected with the question of the desired associativity of the + operator.

Now examine the entry 2?. Similar reasoning shows that, if we wish to associate $a{+}a{*}a$ as $a{+}(a{*}a)$ (i.e. * has higher precedence), then 2? should be "shift 5"; if we wish $a{+}a{*}a$ to be treated as $(a{+}a){*}a$, then 2? should be "reduce 1." Again we also have the option of forbidding this input string by replacing 2? by "error."

In the same fashion, we may observe that the input string $a{*}a$ will eventually cause the parser to place state 8 on top of the stack. The entry 4? dictates the associativity of the * operator, and the entry 3? reflects the relative precedence of + following *.

Fig. 3

Stack	Unexpended input
0	$a+a*a\$$
0a3	$+a*a\$$
0E1	$+a*a\$$
0E1+4	$a*a\$$
0E1+4a3	$*a\$$
0E1+4E7	$*a\$$
0E1+4E7*5	$a\$$
0E1+4E7*5a3	\$
0E1+4E7*5E8	\$
0E1+4E7	\$
0E1	\$

Table IV gives several different resolutions of the ambiguous entries with a brief description of the disambiguating rule used. Thus for a grammar like G_9 we can give a clear meaning to the resolutions of the multiply defined entries in the action table.

Suppose we resolve the entries according to the top row of Table IV. A parse of the input string $a+a*a$ goes as shown in Figure 3.

There is a close correspondence between this parser and the one given for G_8 in Example 6. G_8 uses the first four productions to enforce the left associativity of the $+$ and $*$ operators. G_8 also uses two nonterminals E and T to make $*$ of higher precedence than $+$. However, introducing these nonterminals requires the use of the productions

$E \to T \qquad T \to F$

to allow a factor to be trivially reduced to a term, and a term to be trivially reduced to an expression. These single productions serve no other useful purpose. In Example 6, the LR parser for G_8 made 13 moves parsing $a+a*a$. The parser above makes only ten moves.

When we are parsing we can skip the reductions by these single productions provided we take into account the associativities and precedence levels of the operators. In [3, 5, 27] an algorithmic technique for eliminating reductions by single productions of this nature was presented. If we apply this technique to the simple LR(1) parser for G_8 we obtain the parser for G_9 given above. Thus by using the ambiguous grammar G_9 and introducing precedence information to specify the associativity and relative precedence of $+$ and $*$ we can obtain the reduced parser directly.

This direct procedure has several advantages. First, we have a grammar for our language which is smaller and easier to read. Second, a more efficient parser with fewer states can be directly obtained from the ambiguous grammar and the precedence information than from the equivalent unambiguous grammar. Moreover, it can be shown that this more efficient parser is equivalent in a useful sense to the simple LR parser constructed from the unambiguous grammar [3].

Our analysis of G_9 can be extended to the following class of grammars. Consider a sequence $*_1, *_2, \ldots, *_n$ of n left associative operators and assume $*_i$ has lower precedence than $*_{i+1}$. We can write the ambiguous grammar:

$E \to E *_1 E$
$E \to E *_2 E$
$\ldots$
$E \to E *_n E$
$E \to (E)$
$E \to a$

When we construct the parser for this grammar, we find n^2 ambiguous entries in the action table. Each is of the form: "In state s on input $*_i$, we may either shift or reduce by $E \to E *_j E$." To reflect the given associativities and precedence levels of the operators, we may use the simple resolution rule: "If $i > j$, shift; otherwise, reduce." It can be shown that when the ambiguities are resolved in this way, the resulting parser analyzes expressions giving the proper associativities and precedences to $*_1, \ldots, *_n$. The resolution rule can be easily extended to deal with right associative and nonassociating operators, as well as several operators at each precedence level as indicated in Example 3.

Notice that in a scheme of this nature the precedence information is consulted only where the grammar is ambiguous. Thus, there is no need to specify awkward precedences for terminals such as reserved words or parentheses, as one would have to do in a purely operator precedence scheme.

An analogous technique can be applied to the **if-then-else** grammar G_5.

Example 9. Consider the following version of G_5:

G_{10}: (1) $S \to aSbS$
(2) $S \to aS$
(3) $S \to c$

The collection of sets of items for G_{10} is:

I_0: $S' \to .S$
$S \to .aSbS$
$S \to .aS$
$S \to .c$

I_1: $S' \to S.$

I_2: $S \to a.SbS$
$S \to a.S$
$S \to .aSbS$
$S \to .aS$
$S \to .c$

I_3: $S \to c.$

I_4: $S \to aS.bS$
$S \to aS.$

I_5: $S \to aSb.S$
$S \to .aSbS$
$S \to .aS$
$S \to .c$

I_6: $S \to aSbS.$

The resulting action and goto table are shown in Table V. Since FOLLOW(S) = $\{b, \$\}$, the entry for state 4 on input b is a shift-reduce conflict between "shift 5" and "reduce by $S \to aS$". If we choose to shift on input b, then we shall always associate this b with the last available a. This would correspond to associating an **else** with the most recent unmatched **then.** □

Let us review the main ideas of this section. An ambiguous grammar always gives rise to an LR parser with multiply defined entries in its action table. If we make each multiply defined entry single-valued by choosing only one of the possible actions or error in accordance with some disambiguating rule, we obtain an LR parser that will parse a subset of the language defined by the original grammar; often this is a more economical parser.

Some of the ideas of using operator precedence to resolve ambiguities were developed independently by [9]. The work of [10] should also be mentioned. There, given any operator precedence grammar, one obtains a bottom-up parser that works in an LR(1) style on a skeletal grammar (see [2]). Unfortunately, the method does not guarantee that the language parsed is the same as that defined by the original grammar. It does, however, produce the parser of Example 6 for G_8, so in at least some cases the language is not changed. In [6] conditions are characterized under which the resulting LR parsers for ambiguous grammars recognize the same language as that generated by the grammar.

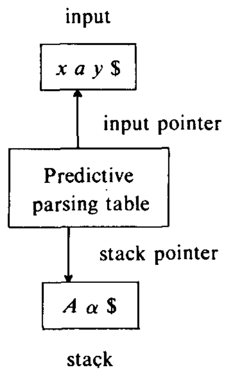
6. LL(1) Grammars and Parsers

We have seen how LR parsers can be made simpler by using ambiguous grammars. However, we could not increase the class of languages that were parsable by deterministic bottom up means for the simple reason that the LR languages are coextensive with the class of languages accepted by deterministic pushdown automata [18], and the latter can simulate an LR parser. There is another, smaller class of grammars, called LL, which has been considered theoretically and used in at least one compiler [21]. In [22] these grammars are considered to possess certain virtues in connection with syntax directed translations definable on them.

A disadvantage of LL grammars is that the languages they accept form a proper subset of the LR languages; there is at least one important construct, the dangling-**else**, which is not representable by an LL grammar. It is therefore significant that by going to ambiguous grammars, we can define a class which properly includes the LL grammars and is capable of handling the dangling-**else** and other constructs. In so doing, we retain the capability possessed by formal grammar methods of being able to prove that our parser accepts the language we say it does, and to obtain this proof mechanically.

An LL(1) grammar is parsed by the device shown in Figure 4, called a *predictive parser*. (We shall omit a discussion of the full class of LL(k) grammars, which are treated similarly [23, 28].) The predictive parser is really a one state pushdown automaton that may scan an input symbol without consuming it [12]. The notion is also equivalent to the DeBakker-Scott schemes [8, 24]. As with the LR parser, the input contains the string to be parsed, followed by the right endmarker. The parser also has a predictive parsing table and a stack; the stack initially contains only the start symbol. In the LL parser, we draw the top of the stack at the left. The special symbol \$ marks the end of the input and of the stack. At every point in time, if the input (xay in Figure 4) is in the language, then x, the portion of the input over which the input pointer has moved, followed by $A\alpha$, the stack contents, will be called *the left sentential form represented by the parser*. In Figure 4 $xA\alpha$ is the left sentential form represented by the parser. Note that x is a string of terminals, A is the leftmost nonterminal, and α is a stringof terminals and nonterminals.

Fig. 4. Predictive parser.



The moves of the parser are of three types:

1. If the current input symbol matches the top stack symbol, pop the stack and move the input pointer one symbol to the right. This move is made only with a terminal on top of the stack and does not change the left sentential form represented by the parser.
2. If there is a nonterminal, say A, on top of the stack, and a is the current input symbol, consult a *parsing table* to determine a particular production $A \rightarrow \beta$ associated with A and a. Replace A on top of the stack by the string β. The input pointer is not advanced. This move causes the parser to represent the next left sentential form.
3. If the current input symbol is \$ and the stack contains only \$, then the parser halts and accepts the original input.

Example 10. Consider the grammar

$$\begin{aligned} S &\rightarrow aA \\ &\mid bB \\ A &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

with the start symbol S. The appropriate parsing table is:

	a	b	\$
S	$S \rightarrow aA$	$S \rightarrow bB$	—
A	$A \rightarrow a$	—	—
B	—	$B \rightarrow b$	—

With input aa the parser would make the following moves:

Unexpended Input	Stack	Left sentential form represented by the parser
$aa\$$	$S\$$	S
$aa\$$	$aA\$$	aA
$a\$$	$A\$$	aA
$a\$$	$a\$$	aa
\$	\$	aa

□

Definition. A grammar is LL(1) if whenever $A \rightarrow \alpha$ and $A \rightarrow \beta$ are distinct productions, the following two conditions hold: (1) There is no terminal a such that both α and β derive strings beginning with a. (2) At most one of α and β is or derives ϵ, the empty string. If, say, $\alpha \Rightarrow^* \epsilon$, then there is no terminal a in FOLLOW(A) such that β derives a string beginning with a.

For an LL(1) grammar, we can construct a predictive parser easily. The entry for (A,a) in the parsing table is the unique production $A \rightarrow \gamma$, if it exists, such that $\gamma \Rightarrow^* a\delta$ for some δ or $\gamma \Rightarrow^* \epsilon$ and a is in FOLLOW(A). If no such γ exists (and there cannot be more than one by the LL(1) definition), then the entry for (A,a) is "error."

Example 11. The (S,a) entry in Example 10 is $S \rightarrow aA$. We observe $aA \Rightarrow^* aA$ (in zero steps), while bB, the other right side for A, does not derive any string beginning with a. □

The LL(1) grammars are a natural class of unambiguous grammars for which nonbacktracking recursive descent parsers can be fabricated, and they have been studied by [18, 23]. However, there are certain ambiguous grammars for which predictive parsing algorithms (nonbacktracking recursive descent parsers) exist and which, like all ambiguous grammars, are not LL(1). We now consider one in detail.

We may rewrite grammar G_5 by "left-factoring" [29, 30] it in the following manner:

G_{11}: $S \rightarrow$ **if** b **then** S ELSECLAUSE
  | **others**
ELSECLAUSE $\rightarrow$ **else** S
  | ϵ

This grammar is ambiguous. For notational simplicity, consider the abbreviated grammar:

G_{12}: $S \rightarrow aSE$
  | c
$E \rightarrow bS$
  | ϵ

In G_{12}, a represents **if** b **then**, b represents **else**, and c represents **others**. The sentence $aacbc$ has the two parse trees shown in Figure 5.

As we have seen, the usual solution to this ambiguity is to specify a disambiguating rule that says an **else** is to be associated with the closest unmatched **then**. In terms of G_{12}, b's are associated with previous a's as in Figure 5(b). A skilled person writing a recursive descent parser for G_{12} would have no trouble implementing this rule. The recursive descent parser would work as follows:

1. Given that we need to find an S on the input, search for aS.
2. If (1) succeeds, check to see whether the next input symbol is b. If so, succeed if and only if the b is followed by a string derivable from S. If b is not the next input symbol, however, succeed without looking further.

The above strategy not only works, it never causes backtracking. Thus it should not be surprising that we can design a predictive parser for G_{12} even though that grammar is ambiguous and hence not LL. Indeed, the language generated by G_{12} is not LL.

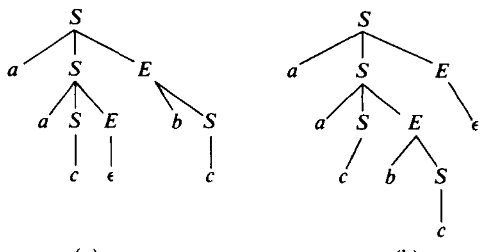
The predictive parser for G_{12} is:

	a	b	c	$
S	$S \rightarrow aSE$	—	$S \rightarrow c$	—
E	—	$E \rightarrow bS$	—	$E \rightarrow \epsilon$

For example, the predictive parser would process $aacbc$ as shown in Figure 6.

Again, the crucial question is how the compiler designer is to know that his attempt to resolve the dangling else or any other ambiguity "works," in the sense that he is not changing the language recognized by the compiler when he causes his parser to throw away certain legal parses. In general, one cannot decide such matters, but we shall see that G_{12} is in a class where the usual resolution can be shown not to change the language defined by the grammar.

Fig. 5. Two parse trees.



(a) (b)

Fig. 6.

Unexpended input	Stack
aacbc$	S$
aacbc$	*a*SE$
acbc$	SE$
acbc$	*a*SEE$
cbc$	SEE$
cbc$	*c*EE$
bc$	EE$
bc$	*b*SE$
c$	SE$
c$	*c*E$
$	E$
$	$

7. An Extension to the LL Class of Grammars

We shall now consider two classes of grammars with predictive parsers. The classes each include the LL(1) grammars and also some ambiguous ones. For these grammars, we can show that the predictive parser we construct accepts the language generated by the grammar. The LL(k) grammars for $k > 1$ can be extended in the same way, and the details should be obvious to the reader who is acquainted with LL(k) grammars.

Suppose we are in a situation where we have derived the left sentential form $wA\gamma$ and there are two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ such that $\alpha\gamma$ and $\beta\gamma$ each derive strings beginning with the terminal a. Then clearly we do not have an LL(1) grammar. However, it is conceivable that in every situation where we are faced with a choice of expanding A by α or β one choice will always enable us to continue the parse to completion if the other choice would.

Definition. We say a grammar G is in *Class 1* if the following conditions hold:
(1) G is not left recursive.[1]
(2) Let $wA\gamma$ be a left sentential form. Suppose $A \rightarrow \alpha$ and $A \rightarrow \beta$ are two productions in G such that a string beginning with the terminal a can be derived from both $\alpha\gamma$ and $\beta\gamma$. Then either $\beta\gamma \Rightarrow^* ax$ implies $\alpha\gamma \Rightarrow^* ax$ for all terminal strings x or $\alpha\gamma \Rightarrow^* ax$ implies $\beta\gamma \Rightarrow^* ax$ for all terminal strings x.

Condition (2) incorporates the disambiguating rule. Informally, (2) says that a predictive parser may always expand A by the more inclusive production on lookahead a. Condition (1) prevents the parser from getting into a loop. Clearly, every LL(1) grammar is in Class 1, since condition (2) is trivially satisfied and no LL(1) grammar is left recursive. However, certain ambiguous grammars such as G_{12} are also in Class 1, as we shall see.

We should observe that it is not immediately apparent if we can effectively determine whether a grammar is in Class 1, and in fact, membership in Class 1 is undecidable, as we shall show. We shall therefore progress to a smaller class of grammars which is effective but contains at least G_{12} in addition to the LL(1) grammars.

Definition. Given a grammar and a nonterminal A, define $F(A)$ to be the set of symbols X (nonterminal or terminal) such that there is some left sentential form $wA\gamma X\delta$ and $\gamma \Rightarrow^* \epsilon$. (The case where γ is itself the empty string is not ruled out.)

Definition. We say a grammar is in *Class 2* if whenever $A \rightarrow \alpha$ and $A \rightarrow \beta$ are distinct productions the following two conditions hold:
(1) There is no terminal a such that both α and β derive a string beginning with a.
(2) At most one of α and β is or derives ϵ. Suppose $\beta \Rightarrow^* \epsilon$ and $\alpha \Rightarrow^* \alpha x$ for some terminal a such that a is in FOLLOW(A). Then for all X in $F(A)$ such that $X \neq A$, X does not derive a string beginning with a. (This rules out the possibility that $X = a$.)

We can easily show that (1) and (2) imply that there is no left recursion. It will also be shown that condition (2) of the Class 1 definition is satisfied. In particular, when condition (2) of the Class 2 definition prevails and we are presented with a choice of expanding A by α or β with a as the next lookahead symbol, we always choose α.

It is easy to compute $F(A)$ for each nonterminal A and thus decide whether a grammar is in Class 2.

Example 12. Recall the grammar G_{12} whose productions are:

$$\begin{aligned} S &\rightarrow a\,S\,E \\ &\mid\ c \\ E &\rightarrow b\,S \\ &\mid\ \epsilon \end{aligned}$$

We shall show that G_{12} is in Class 2, and hence in Class 1. The productions for S satisfy the LL(1) definition, so we certainly get no contradiction of the Class 2 condition. For the E productions, we observe that $E \rightarrow bS$ derives a string beginning with b, and b is also in FOLLOW(E). We must therefore check that E is the only symbol in $F(E)$ that derives strings beginning with b.

We note that E is in $F(E)$, since $S \Rightarrow^*_{lm} aaSEE$, for example. However, there are no other members of $F(E)$, so condition (2) of the Class 2 definition is satisfied. Thus, on lookahead b, we may always expand E to bS rather than ϵ. The predictive parser generated is the one given in Section 6. □

Example 13. Let us consider a block structured language in which an **end** will terminate all open blocks. The following grammar abstracts this situation.

$$\begin{aligned} G_{13}:\quad S &\rightarrow \textbf{begin}\ \text{LIST}\ \textbf{end} \\ &\mid\ \textbf{begin}\ \text{LIST} \\ &\mid\ \textbf{others} \\ \text{LIST} &\rightarrow S;\ \text{LIST} \\ &\mid\ S \end{aligned}$$

Here **others** stands for other types of statements. The left factored form of G_{13} with productions numbered is:

G_{14}: (1) $S \rightarrow$ **begin** LIST END
(2) $S \rightarrow$ **others**
(3) END $\rightarrow$ **end**
(4) END $\rightarrow \epsilon$
(5) LIST $\rightarrow S$ REM
(6) REM $\rightarrow$; LIST
(7) REM $\rightarrow \epsilon$

G_{14} is not LL(1). For example, END $\rightarrow$ **end** obviously derives a string beginning with **end**. But **end** is in FOLLOW(END), since $S \Rightarrow^*$ **begin begin** LIST END **end**, for example. This observation, together with the fact that END $\rightarrow \epsilon$ is a production, violates the LL(1) definition.

However, G_{14} is in Class 2. It is easy to check that no violation of condition (1) occurs. To check condition (2) we find F(END) = F(REM) = {END, REM}. We must check, because of the pair of productions END $\rightarrow$ **end** | ϵ, that REM does not derive a string beginning with **end**. Also, because of REM $\rightarrow$; ⟨LIST⟩ | ϵ, we must check that END does not derive a string beginning with ; . Clearly, neither of these derivations exists so G_{14} is in Class 2.

[1] A grammar is *left recursive* if it has a nonterminal A such that A$\Rightarrow^*$ Aα for some α in a derivation of one or more steps.

The predictive parser for G_{14} is:

	begin	end	others	;	\$
S	1	—	2	—	—
END	—	3	—	4	4
LIST	5	—	5	—	—
REM	—	7	—	6	7

Here the numbers refer to the productions in G_{14}. □

We would now like to show that every Class 2 grammar is in Class 1. The absence of left recursion is easy to demonstrate, and we omit this part of the proof. We must show for Class 2 grammars that whenever $A \rightarrow \alpha$ and $A \rightarrow \beta$ are two productions such that a is in FOLLOW(A), $\beta \Rightarrow^* \epsilon$, α derives strings beginning with a, and

$$S \overset{*}{\Rightarrow}_{\text{lm}} wA\gamma \overset{*}{\Rightarrow}_{\text{lm}} w\beta\gamma \overset{*}{\Rightarrow}_{\text{lm}} wax,$$

then $\alpha\gamma \Rightarrow^* ax$. Since the grammar is in Class 2, we may assume that neither β, nor any other right side for A, except α, derives a string beginning with a. Also no symbol but A in $F(A)$ derives strings beginning with a.

Then the derivation $\beta\gamma \overset{*}{\Rightarrow}_{\text{lm}} ax$ can be written

$$\beta\gamma \overset{*}{\Rightarrow}_{\text{lm}} \gamma \overset{*}{\Rightarrow}_{\text{lm}} X\gamma' \overset{*}{\Rightarrow}_{\text{lm}} ay\gamma' \overset{*}{\Rightarrow}_{\text{lm}} ax$$

That is, $\gamma = \gamma''X\gamma'$, where $\gamma'' \Rightarrow^* \epsilon$ and $X \Rightarrow^* ay$ for some y. Then X is in $F(A)$, so X is A. Since X derives a string beginning with a, the first step in the derivation $X \overset{*}{\Rightarrow}_{\text{lm}} ay$ must be $X \Rightarrow \alpha$. Thus there exists another leftmost derivation $\alpha\gamma \overset{*}{\Rightarrow}_{\text{lm}} ay\gamma$. Since $\gamma = \gamma''X\gamma'$, $\gamma'' \Rightarrow^* \epsilon$, $X \Rightarrow \beta \Rightarrow^* \epsilon$, and γ' derives the string z such that $yz = x$, we have $\gamma \Rightarrow^* z$. Thus $w\alpha\gamma \overset{*}{\Rightarrow}_{\text{lm}} wax$, as was to be proved. We now have the following theorem:

THEOREM 1. *Every Class 2 grammar is in Class 1.*

8. Properties of Class 1 and 2 Grammars

Let us first observe that every Class 1 grammar can be put into a modified Greibach normal form where each production is of the form $A \rightarrow a\alpha$ or $A \rightarrow \epsilon$ and no two productions are of the form $A \rightarrow a\alpha_1 \mid a\alpha_2$. The production $A \rightarrow \epsilon$ will be used by the predictive parser on lookahead a only when there is no production of the form $A \rightarrow a\alpha$.

To put a grammar in this form, we observe that for each nonterminal A on top of the stack the predictive parser on lookahead a will either halt or cause A to be replaced, in one or more steps, by some string of the form $a\alpha$, or by the empty string. To construct the new grammar, for each such situation we create the productions $A \rightarrow a\alpha$ or $A \rightarrow \epsilon$, respectively. The new grammar clearly generates the same language as the old and is in the desired form. Furthermore, the new grammar is still in Class 1.

Example 14. Let us consider G_{14} from Example 13. The only production whose right side is not empty and does not begin with a terminal is LIST → S REM. With LIST on top of the stack and **begin** as the next input symbol, LIST is replaced in two steps by **begin** LIST END REM according to the parser of Example 13. With **others** on top of the stack, LIST is instead replaced by **others** REM. Other inputs cause the parser to halt. We thus give our new grammar the productions

LIST → **begin** LIST END REM
 | **others** REM

for the nonterminal LIST and the productions of G_{14} for nonterminals other than LIST.

We shall thus state without proof the following theorem.

THEOREM 2. *Every Class 1 (Class 2) grammar has an equivalent Class 1 (Class 2) grammar in which every production is either of the form $A \rightarrow a\alpha$ or $A \rightarrow \epsilon$. Moreover, the predictive parser for the grammar does not on lookahead a expand A by $A \rightarrow \epsilon$ unless there is no production of the form $A \rightarrow a\alpha$.*

Using Theorem 2, we can show that certain rather primitive languages do not have a Class 1 grammar.

THEOREM 3. *The language $L = \{a^nb^n \mid n \geq 1\} \cup \{a^nb^{n} \mid n \geq 1\}$ has no Class 1 grammar.*

PROOF. The proof parallels the one given in [28] to show that L is not LL(1) and can thus be omitted. □

Finally, we show that there is no algorithm to tell whether a given grammar is in Class 1.

THEOREM 4. *It is undecidable whether a grammar is in Class 1.*

PROOF. Assume that we have an algorithm to determine whether a grammar is in Class 1. Using this algorithm we shall construct an algorithm to decide for Class 1 grammars G_1 and G_2 whether $L(G_1)$ is included in $L(G_2)$.

First, one may test (see e.g. [2]) for arbitrary CFG's G_1 and G_2, whether

a. $L(G_1)$ contains ϵ, but $L(G_2)$ does not.
b. $L(G_2)$ is empty, but $L(G_1)$ is not.
c. For some terminal a, G_1 generates a string beginning with a, but G_2 generates no such string.

We may begin our test for inclusion with tests for the above three conditions. If any of them hold, then $L(G_1)$ is not included in $L(G_2)$ and our test ends.

Now assume that none of (a), (b), or (c) hold. Let S_1 and S_2 be the start symbols of G_1 and G_2, and assume that G_1 and G_2 have no nonterminals in common. Let S and S' be new nonterminals and # be a new symbol.

Construct a new grammar G with the productions of G_1 and G_2 plus $S \rightarrow S_1 \mid S_2S'$ and $S' \rightarrow \# \mid \epsilon$. Now if $L(G_1)$ is included in $L(G_2)$, then the choice of $S \rightarrow S_2S'$ over $S \rightarrow S_1$ satisfies condition (2) of the definition of Class 1 grammar when the nonterminal is S and the terminal is arbitrary. Also, no other violation of the Class 1 definition can occur for G, since G_1 and G_2 are in Class 1 by hypothesis. However, suppose $L(G_1)$ is not included in $L(G_2)$. Then since conditions (a)–(c) do not hold, we may find aw in $L(G_1)$ but not in $L(G_2)$ and also find ax in $L(G_2)$ for some x. Then consideration of aw tells us that $S \rightarrow S_2S'$ does not take priority over $S \rightarrow S_1$ on lookahead a, and consideration of $ax\,\#$ tells us that neither does $S \rightarrow S_1$ take priority over $S \rightarrow S_2S'$. This situation violates the Class 1 definition. Thus G is in Class 1 if and only if $L(G_1)$ is included in $L(G_2)$.

On the hypothesis that we may test whether G is in Class 1, we would have an algorithm to test set inclusion for Class 1 grammars. However, [14] has shown that inclusion is undecidable even for the simple grammars of [20], a proper subset of the LL grammars and hence of the Class 1 grammars. □

As mentioned, a test for membership in Class 2 is easy.

9. Summary

We have seen that by using ambiguous context-free grammars with some rather simple disambiguating rules we can obtain language definitions that are shorter and easier to comprehend than equivalent definitions in terms of an unambiguous grammars. Moreover, if the disambiguating rules are carefully chosen, we can often resolve the parsing action conflicts in a straightforward way and obtain a parser equivalent to that constructed from an equivalent unambiguous grammar.

A number of open questions remain. In this paper we have used rather informal disambiguating rules that reflect current practices in programming language design. A more general but still easy to use formalism for specifying disambiguating rules would be the next step. We would also be interested in knowing to what extent ambiguous grammars and disambiguating rules should be used to obtain the clearest description of a language.

Received November 1973; revised February 1975

References

1. Aho, A. V. and Johnson, S. C. LR Parsing. *Computing Surveys* 6, 2 (June 1974), 99–124.
2. Aho, A. V. and Ullman, J. D. *The Theory of Parsing, Translation and Compiling. Vol.* 1: *Parsing*. Prentice-Hall, Englewood Cliffs, N. J., 1972. *Vol.* 2: *Compiling*, 1973.
3. Aho, A. V. and Ullman, J. D. A technique for speeding up LR (k) parsers. *SIAM J. Computing* 2, 2 (June 1972), 106–127.
4. Conway, M. E. Design of a separable transition-diagram compiler. *Comm. ACM* 6, 7 (July 1963), 396–408.
5. Demers, A. J. Elimination of single productions and merging nonterminal symbols of LR (1) grammars. *J. Computer Languages* 1, 2 (April 1975).
6. Demers, A. J. Skeletal LR parsing. IEEE Conf. Record of 15th Annual Symposium on Switching and Automata Theory, 1974, pp. 185–198.
7. DeRemer, F. L. Simple LR(k) grammars. *Comm. ACM* 14, 7 (July 1971), 453–460.
8. de Bakker, J. W. and Scott, D. A theory of programs. Unpublished.
9. Earley, J. Ambiguity and precedence in syntax description. Tech Rep. 13, Dep. Computer Science, U. of California, Berkeley, 1973.
10. El Djabri, N. Extending the LR parsing technique to some non-LR grammars. TR 121, Computer Science Lab., Dep. Electr. Eng., Princeton U., Princeton, N.J., 1973.
11. Feldman, J. and Gries, D. Translator writing systems. *Comm. ACM 11*, 2 (Feb. 1968), 77–113.
12. Fischer, M. J. Some properties of precedence languages. Proc. ACM Symposium on Theory of Computing, May 1969, pp. 181–190.
13. Floyd, R. W. Syntactic analysis and operator precedence. *J. ACM 10*, 3 (March 1963), 316–333.
14. Friedman, E. P. The inclusion problem for simple machines. Proc. Eighth Annual Princeton Conference on Information Sciences and Systems, 1974, pp. 173–177.
15. Ginsburg, S. and Spanier, E. H. Control sets on grammars. *Mathematical Systems Theory 2*, 2 (1968), 159–178.
16. Hopcroft, J. E. and Ullman, J. D. *Formal Languages and their Relation to Automata*. Addison-Wesley, Reading, Mass., 1969.
17. Joliat, M. L. Practical minimization of LR(k) parser tables. Proc. IFIP Congress 1974, pp. 376–380.
18. Knuth, D. E. Top-down syntax analysis. *Acta Informatica 1*, 2 (1971), 79–110.
19. Kernighan, B. W. and Cherry, L. L. A system for typesetting mathematics. *Comm. ACM 18*, 3 (Mar 1975), 151–156.
20. Korenjak, A. J. and Hopcroft, J. E. Simple deterministic languages. IEEE Conf. Record of 7th Annual Symposium on Switching and Automata Theory, 1966, pp. 36–46.
21. Lewis, P. M. and Rosenkrantz, D. J. An Algol compiler designed using automata theory. Proc. Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, N.Y., 1971, pp. 75–88.
22. Lewis, P. M., Rosenkrantz, D. J., and Stearns, R. E. Attributed translations. *J. Computer and System Sciences 9*, 3 (Dec. 1974), 279–307.
23. Lewis, P. M. and Stearns, R. E. Syntax directed transduction. *J. ACM 15*, 3 (March 1968), 464–438.
24. Manna, Z., Ness, S., and Vuillemin, J. Inductive methods for proving properties of programs. Proc. ACM Conf. on Proving Assertions About Programs, 1972, pp. 27–50.
25. McKeeman, W. M., Horning, J. J., and Wortman, D. B. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, N.J., 1970.
26. Mickunas, M. D. and Schneider, V. B. A parser generating system for constructing compressed compilers. *Comm. ACM 16*, 11 (Nov. 1973), 667–675.
27. Pager, D. On eliminating unit productions from LR(k) parsers. TR, Information Sciences Program, U. of Hawaii, Honolulu, Hawaii, 1974.
28. Rosenkrantz, D. J. and Stearns, R. E. Properties of deterministic top-down grammars. *Inf. Control 14*, 5 (1969), 226–256.
29. Stearns, R. E. Deterministic top-down parsing. Proc. Fifth Annual Princeton Conf. on Information Sciences and Systems, 1971, pp. 182–188.
30. Wood, D. The theory of left factored languages. *Computer J. 12*, 4 (1969), 349–356 and *13*, 1 (1970), 55–62.