LR Parsing

A. V. AHO and S. C. JOHNSON

Bell Laboratories, Murray Hill, New Jersey 07974

The LR syntax analysis method is a useful and versatile technique for parsing deterministic context-free languages in compiling applications. This paper provides an informal exposition of LR parsing techniques emphasizing the mechanical generation of efficient LR parsers for context-free grammars. Particular attention is given to extending the parser generation techniques to apply to ambiguous grammars.

Keywords and phrases: grammars, parsers, compilers, ambiguous grammars, context-free languages, LR grammars.

CR categories. 4 12, 5 23

1. INTRODUCTION

A complete specification of a programming language must perform at least two functions. First, it must specify the *syntax* of the language; that is, which strings of symbols are to be deemed well-formed programs. Second, it must specify the *semantics* of the language; that is, what meaning or intent should be attributed to each syntactically correct program.

A compiler for a programming language must verify that its input obeys the syntactic conventions of the language specification. It must also translate its input into an object language program in a manner that is consistent with the semantic specification of the language. In addition, if the input contains syntactic errors, the compiler should announce their presence and try to pinpoint their location. To help perform these functions every compiler has a device within it called a *parser*.

A context-free grammar can be used to help specify the syntax of a programming language. In addition, if the grammar is designed carefully, much of the semantics of the language can be related to the rules of the grammar.

There are many different types of parsers for context-free grammars. In this paper we shall restrict ourselves to a class of parsers known as LR parsers. These parsers are efficient and well suited for use in compilers for programming languages. Perhaps more important is the fact that we can automatically generate LR parsers for a large and useful class of context-free grammars. The purpose of this article is to show how LR parsers can be generated from certain context-free grammars, even some ambiguous ones. An important feature of the parser generation algorithm is the automatic detection of ambiguities and difficult-to-parse constructs in the language specification.

We begin this paper by showing how a context-free grammar defines a language. We then discuss LR parsing and outline the parser generation algorithm. We conclude by showing how the performance of LR parsers can be improved by a few simple transformations, and how error recovery and "semantic actions" can be incorporated into the LR parsing framework.

For the purposes of this paper, a *sentence* is a string of *terminal symbols*. Sentences are written surrounded by a pair of single quotes. For example, '*a*', '*ab*', and ',' are sentences. The empty sentence is written ''. Two sentences written contiguously are to be concatenated, thus '*a*' '*b*' is synonymous with

CONTENTS

'*ab*'. In this paper the term *language* merely means a set of sentences.

2. GRAMMARS

A grammar is used to define a language and to impose a structure on each sentence in the language. We shall be exclusively concerned with *context-free grammars*, sometimes called BNF (for Backus-Naur form) specifications.

In a context-free grammar, we specify two disjoint sets of symbols to help define a language. One is a set of *nonterminal symbols*. We shall represent a nonterminal symbol by a string of one or more capital roman letters. For example, LIST represents a nonterminal as does the letter A. In the grammar, one nonterminal is distinguished as a *start* (or *sentence*) symbol.

The second set of symbols used in a context-free grammar is the set of *terminal symbols*. The sentences of the language generated by a grammar will contain only terminal symbols. We shall refer to a terminal or nonterminal symbol as a *grammar symbol*.

A context-free grammar itself consists of a finite set of rules called *productions*. A production has the form

$$\text{left-side} \rightarrow \text{right-side},$$

where left-side is a single nonterminal symbol (sometimes called a syntactic category) and right-side is a string of zero or more grammar symbols. The arrow is simply a special symbol that separates the left and right sides. For example,

$$\text{LIST} \rightarrow \text{LIST ',' ELEMENT}$$

is a production in which LIST and ELEMENT are nonterminal symbols, and the quoted comma represents a terminal symbol.

A grammar is a rewriting system. If $\alpha A \gamma$ is a string of grammar symbols and $A \rightarrow \beta$ is a production, then we write

$$\alpha A \gamma \Rightarrow \alpha \beta \gamma$$

and say that $\alpha A \gamma$ *directly derives* $\alpha \beta \gamma$. A sequence of strings

$$\alpha_0, \alpha_1, \cdots, \alpha_n$$

such that $\alpha_{i-1} \Rightarrow \alpha_i$ for $1 \leqslant i \leqslant n$ is said to be a *derivation* of α_n from α_0. We sometimes also say α_n is *derivable* from α_0.

The start symbol of a grammar is called a *sentential form*. A string derivable from the start symbol is also a *sentential form* of the grammar. A sentential form containing only terminal symbols is said to be a *sentence* generated by the grammar. The *language generated by a grammar* G, often denoted by $L(G)$, is the set of sentences generated by G.

Example 2.1: The following grammar, hereafter called G_1, has LIST as its start symbol:

LIST → LIST ',' ELEMENT
LIST → ELEMENT
ELEMENT → 'a'
ELEMENT → 'b'

The sequence:

LIST ⇒ LIST ',' ELEMENT
⇒ LIST ',a'
⇒ LIST ',' ELEMENT ',a'
⇒ LIST ',b,a'
⇒ ELEMENT ',b,a'
⇒ 'a,b,a'

is a derivation of the sentence 'a,b,a'. $L(G_1)$ consists of nonempty strings of a's and b's, separated by commas.

Note that in the derivation in Example 2.1, the rightmost nonterminal in each sentential form is rewritten to obtain the following sentential form. Such a derivation is said to be a *rightmost derivation* and each sentential form in such a derivation is called a *right sentential form*. For example,

LIST ',b,a'

is a right sentential form of G_1.

If $\alpha A w$ is a right sentential form in which w is a string of terminal symbols, and $\alpha A w \Rightarrow \alpha\beta w$, then β is said to be a *handle* of $\alpha\beta w$ * For example, 'b' is the handle of the right sentential form

LIST ',b,a'

in Example 2.1.

* Some authors use a more restricting definition of handle

A prefix of $\alpha\beta$ in the right sentential form $\alpha\beta w$ is said to be a *viable prefix* of the grammar. For example,

LIST ','

is a viable prefix of G_1, since it is a prefix of the right sentential form,

LIST ',' ELEMENT

(Both α and w are null here.)

Restating this definition, a viable prefix of a grammar is any prefix of a right sentential form that does not extend past the right end of a handle in that right sentential form. Thus we know that there is always some string of grammar symbols that can be appended to the end of a viable prefix to obtain a right sentential form. Viable prefixes are important in the construction of compilers with good error-detecting capabilities, as long as the portion of the input we have seen can be derived from a viable prefix, we can be sure that there are no errors that can be detected having scanned only that part of the input.

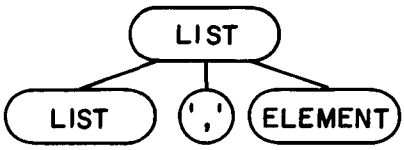
3. DERIVATION TREES

Frequently, our interest in a grammar is not only in the language it generates, but also in the structure it imposes on the sentences of the language. This is the case because grammatical analysis is closely connected with other processes, such as compilation and translation, and the translations or actions of the other processes are frequently defined in terms of the productions of the grammar. With this in mind, we turn our attention to the representation of a derivation by its *derivation tree*.

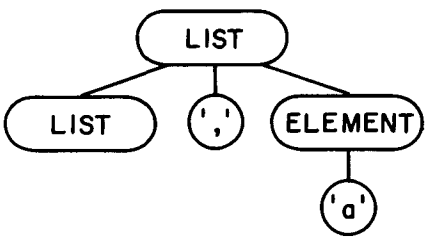
For each derivation in a grammar we can construct a corresponding derivation tree. Let us consider the derivation in Example 2.1. To model the first step of the derivation, in which LIST is rewritten as

LIST ',' ELEMENT

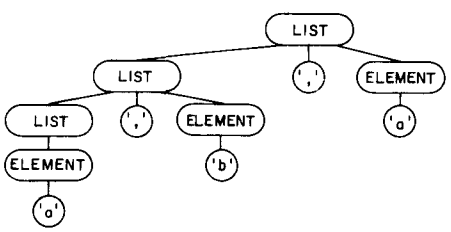
using production 1, we first create a root labeled by the start symbol LIST, and then create three direct descendants of the root, labeled LIST, ',', and ELEMENT:



(We follow historical usage and draw our "root" node at the top.) In the second step of the derivation, ELEMENT is rewritten as 'a'. To model this step, we create a direct descendant labeled 'a' for the node labeled ELEMENT:



Continuing in this fashion, we obtain the following tree:



Note that if a node of the derivation tree is labeled with a nonterminal symbol A and its direct descendants are labeled $X_1, X_2, \cdots, X_n$, then the production.

$$A \rightarrow X_1 X_2 \cdots X_n$$

must be in the grammar.

If $a_1, a_2, \cdots, a_m$ are the labels of all the leaves of a derivation tree, in the natural left-to-right order, then the string

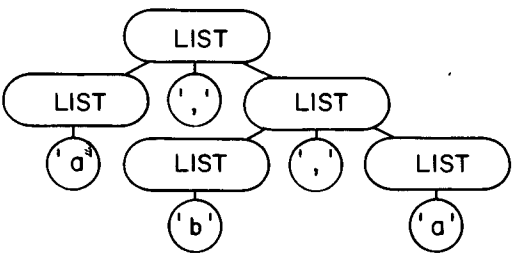
$$a_1 a_2 \cdots a_m$$

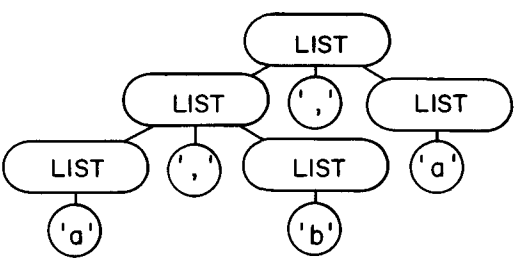
is called the *frontier* of the tree. For example, 'a,b,a' is the frontier of the previous tree. Clearly, for every sentence in a language there is at least one derivation tree with that sentence as its frontier. A grammar that admits two or more distinct derivation trees with the same frontier is said to be *ambiguous*.

Example 3.1: The grammar G_2 with productions

LIST → LIST ',' LIST
LIST → 'a'
LIST → 'b'

is ambiguous because the following two derivation trees have the same frontier.





In certain situations ambiguous grammars can be used to represent programming languages more economically than equivalent unambiguous grammars. However, if an ambiguous grammar is used, then some other rules should be specified along with the grammar to determine which of several derivation trees is to be associated with a given input. We shall have more to say about ambiguous grammars in Section 7.

4. PARSERS

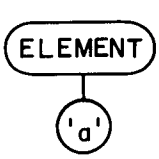
We can consider a parser for a grammar to be a device which, when presented with an input string, attempts to construct a deriva-

tion tree whose frontier matches the input. If the parser can construct such a derivation tree, then it will have verified that the input string is a sentence of the language generated by the grammar. If the input is syntactically incorrect, then the tree construction process will not succeed and the positions at which the process falters can be used to indicate possible error locations.

A parser can operate in many different ways. In this paper we shall restrict ourselves to parsers that examine the input string from left to right, one symbol at a time. These parsers will attempt to construct the derivation tree "bottom-up"; i.e., from the leaves to the root. For historical reasons, these parsers are called *LR parsers*. The "L" stands for "left-to-right scan of the input", the "R" stands for "rightmost derivation." We shall see that an LR parser operates by reconstructing the reverse of a rightmost derivation for the input. In this section we shall describe in an informal way how a certain class of LR parsers, called LR(1) parsers, operate.

An LR parser deals with a sequence of partially built trees during its tree construction process. We shall loosely call this sequence of trees a *forest*. In our framework the forest is built from left to right as the input is read. At a particular stage in the construction process, we have read a certain amount of the input, and we have a partially constructed derivation tree. For example, suppose that we are parsing the input string *'a,b'* according to the grammar G_1. After reading the first *'a'* we construct the tree:

Then we construct:



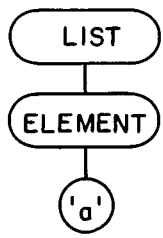
using the production,

ELEMENT → *'a'*

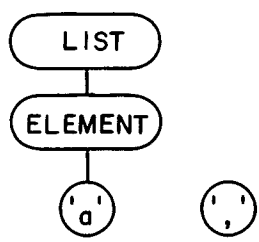
To reflect this parsing action, we say that *'a'* is *reduced to* ELEMENT. Next we use the production

LIST → ELEMENT

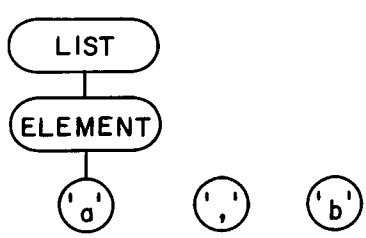
to obtain the tree:



Here, ELEMENT is reduced to LIST. We then read the next input symbol ',', and add it to the forest as a one node tree.



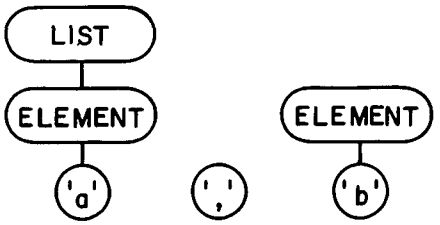
We now have two trees. These trees will eventually become sub-trees in the final derivation tree. We then read the next input symbol *'b'* and create a single node tree for it as well.



Using the production,

ELEMENT → *'b'*

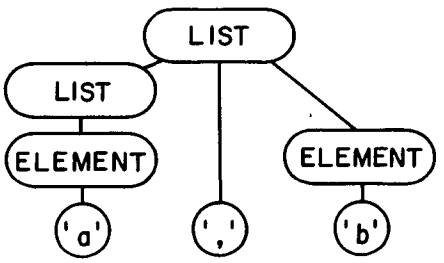
we reduce 'b' to ELEMENT to obtain:



Finally, using the production

LIST → LIST ',' ELEMENT

we combine these three trees into the final tree:



At this point the parser detects that we have read all of the input and announces that the parsing is complete. The rightmost derivation of 'a,b' in G_1 is

$$\begin{aligned} \text{LIST} &\Rightarrow \text{LIST ',' ELEMENT} \\ &\Rightarrow \text{LIST ',}b\text{'} \\ &\Rightarrow \text{ELEMENT ',}b\text{'} \\ &\Rightarrow \text{'}a,b\text{'} \end{aligned}$$

In parsing 'a,b' in the above manner, all we have done is reconstruct this rightmost derivation in reverse. The sequence of productions encountered in going through a rightmost derivation in reverse is called a *right parse*.

There are four types of parsing actions that an LR parser can make; *shift*, *reduce*, *accept* (announce completion of parsing), or announce *error*.

In a shift action, the next input symbol is removed from the input. A new node labeled by this symbol is added to the forest at the right as a new tree by itself.

In a reduce action, a production, such as

$$A \rightarrow X_1 X_2 \cdots X_n$$

is specified; each X_i represents a terminal or nonterminal symbol. A reduction by this production causes the following operations:

(1) A new node labeled A is created.
(2) The rightmost n roots in the forest (which will have already been labeled $X_1, X_2, \cdots, X_n$) are made direct descendants of the new node, which then becomes the rightmost root of the forest.

If the reduction is by a production of the form

$$A \rightarrow ''$$

(i.e., where the right side is the empty string), then the parser merely creates a root labeled A with no descendants.

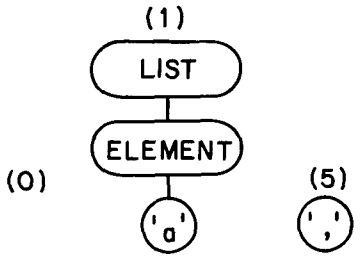
A parser operates by repeatedly making parsing actions until either an accept or error action occurs.

The reader should verify that the following sequence of parsing actions builds the parse tree for 'a,b' in G_1:

(1) Shift 'a'
(2) Reduce by: ELEMENT → 'a'
(3) Reduce by: LIST → ELEMENT
(4) Shift ','
(5) Shift 'b'
(6) Reduce by. ELEMENT → 'b'
(7) Reduce by: LIST → LIST ',' ELEMENT
(8) Accept

We now consider the question of how an LR parser decides what parsing actions to make. Clearly a parsing action can depend on what actions have already been made and on what the next input symbols are. An LR parser that looks at only the next input symbol to decide which parsing action to make is called an LR(1) parser. If it looks at the next k input symbols, $k \geq 0$, it is called an LR(k) parser. To help to make its parsing decisions, an LR parser attaches to the root of each tree in the forest a number called a *state*. The number on the root of the rightmost tree is called the *current state*. In addition, there is an *initial state* to the left of the forest, which helps determine the very first

parsing action. We shall write the states in parentheses above the associated roots. For example,



represents a forest with states. State 5 is the current state, and state 0 is the initial state. The current state and the next input symbol determine the parsing action of an LR(1) parser.

The following table shows the states of an LR(1) parser for G_1, and the associated parsing actions. In this table there is a column labeled '$' with special significance. The '$' stands for the *right endmarker*, which is assumed to be appended to the end of all input strings. Another way of looking at this is to think of '$' as representing the condition where we have read and shifted all of the "real" characters in the input string.

		Next Input Symbol			
		'a'	*'b'*	*','*	*'$'*
Current State	0	shift	shift	error	error
	1	error	error	shift	accept
	2	error	error	Red. 2	Red 2
	3	error	error	Red. 3	Red. 3
	4	error	error	Red. 4	Red. 4
	5	shift	shift	error	error
	6	error	error	Red. 1	Red 1

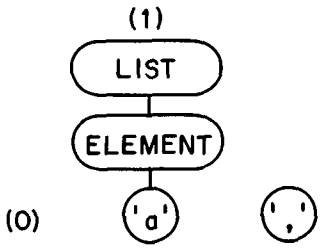
FIG. 1. Parsing Action Table for G_1

The reduce actions are represented as "Red. n" in the above table; the integer n refers to the productions as follows:

(1) LIST → LIST ',' ELEMENT
(2) LIST → ELEMENT
(3) ELEMENT → *'a'*
(4) ELEMENT → *'b'*

We shall refer to the entry for row s and column c as **pa**(s,c). After making either a shift move or a reduce move, the parser must determine what state to attach to the root of the tree that has just been added to the forest. In a shift move, this state is determined by the current state and the input symbol that was just shifted into the forest.

For example, if we have just shifted ',' into the forest



then state 1 and ',' determine the state to be attached to the new rightmost root ','.

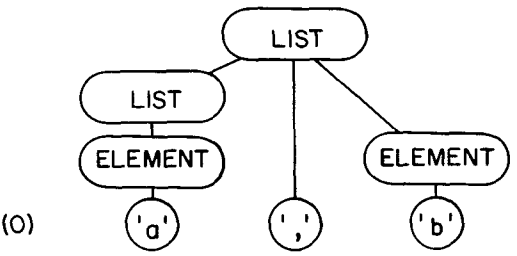
In a reduce move, suppose we reduce by production

$$A \rightarrow X_1 X_2 \cdots X_n$$

When we make nodes $X_1, \cdots, X_n$ direct descendants of the root A, we remove the states that were attached to $X_1, \cdots, X_n$. The state that is to be attached to node A is determined by the state that is now the rightmost state in the forest, and the nonterminal A. For example, if we have just reduced by the production

LIST → LIST ',' ELEMENT

and created the forest



then state 0 and the nonterminal LIST determine the state to be attached to the root LIST. Note that the states previously attached to the direct descendants of the new

root have disappeared, and play no role in the calculation of the new state.

The following table determines these new states for G_1. For reasons that will become apparent later, we shall call this table the *goto table* for G_1.

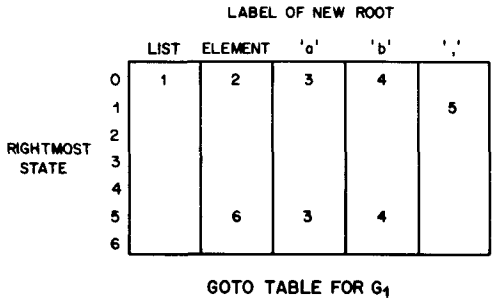


FIG 2. Goto Table for G_1

We shall refer to the entry in the row for state s and column c as **goto**(s, c). It turns out that the entries in the goto table which are blank will never be consulted [Aho and Ullman (1972b)].

An LR parser for a grammar is completely specified when we have given the parsing action table and the goto table. We can picture an LR(1) parser as shown in Fig. 3.

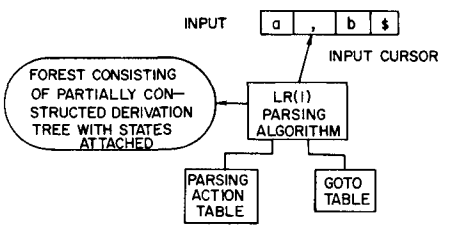


FIG. 3. Pictorial Representation of an LR(1) Parser

The LR(1) parsing algorithm can be summarized as follows:

Initialize: Place the initial state into an otherwise empty forest; the initial state is the current state at the beginning of the parse.

Parsing Action: Examine the parsing action table, and determine the entry corresponding to the current state and the current input symbol. On the basis of this entry (*Shift*, *Reduce*, *Error*, or *Accept*) do one of the following four actions:

Shift: Add a new node, labeled with the current input symbol, to the forest. Associate the state

goto(current state, input)

to this node and make this state the new current state. Advance the input cursor to read the next character. Repeat the step labeled *Parsing Action*.

Reduce: If the indicated production is

$$A \rightarrow X_1 X_2 \cdots X_n$$

add a new node labeled A to the forest, and make the rightmost n roots, $n \geqslant 0$, direct descendants of this new node. Remove the states associated with these roots. If s is the state which is now rightmost in the forest (on the root immediately to the left of the new node), then associate the state

goto(s,A)

with the new node. Make this state the new current state. (Notice that the input character is not changed.) Repeat the step labeled *Parsing Action*.

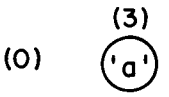
Accept: Halt. A complete derivation tree has been constructed.

Error: Some error has occurred in the input string. Announce error, and then try to resume parsing by recovering from the error. (This topic is discussed in Section 9.)

To see how an LR parser works, let us again parse the input string 'a,b' using the parsing action function **pa** (Figure 1) and the **goto** function (Figure 2).

Initialization: We place state 0 into the forest; 0 becomes the current state.

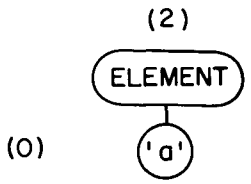
Parsing Action 1: **pa**(0, 'a') = shift. We create a new root labeled 'a' and attach state 3 to it (because **goto**(0, 'a') = 3). We have:



Parsing Action 2: **pa**(3, ',') = reduce 3. We reduce by production (3)

ELEMENT → *'a'*

We examine the state immediately to the left; this is state 0. Since **goto**(0, ELEMENT) = 2, we label the new root with 2. We now have:



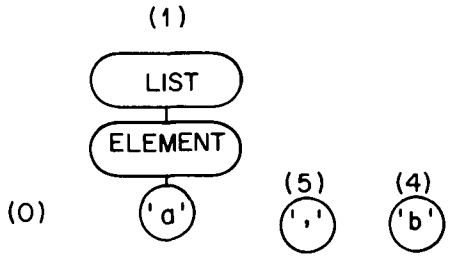
Parsing Action 3: **pa**(2, ',') = reduce 2. We reduce by production (2)

LIST → ELEMENT

goto(0, LIST) = 1, so the new state is 1.

Parsing Action 4: **pa**(1, ',') = shift. We shift and attach state 5.

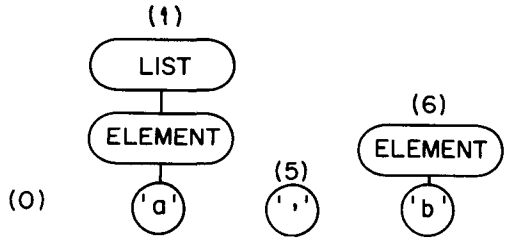
Parsing Action 5: **pa**(5, *'b'*) = shift. We shift and attach state 4. We now have



Parsing Action 6: **pa**(4, '$') = reduce 4. We reduce by production (4)

ELEMENT → *'b'*

goto(5, ELEMENT) = 6, so the new state is 6. We now have



Parsing Action 7: **pa**(6, '$') = reduce 1. We reduce by production (1)

LIST → LIST ',' ELEMENT

The state to the left of the newly created tree is state 0, so the new state is **goto**(0, LIST) = 1.

Parsing Action 8: **pa**(1, '$') = accept. We halt and terminate the parse.

The reader is urged to follow this procedure with another string, such as *'a,b,a'* to verify his understanding of this process. It is also suggested that he try a string which is not in $L(G_1)$, such as *'a,ba'* or *'a,,b'*, to see how the error detection mechanism works. Note that the grammar symbols on the roots of the forest, concatenated from left to right, always form a viable prefix.

Properly constructed LR(1) parsers can parse a large class of useful languages called the *deterministic context-free languages*. These parsers have a number of notable properties:

(1) They report error as soon as possible (scanning the input from left to right).
(2) They parse a string in a time which is proportional to the length of the string.
(3) They require no rescanning of previously scanned input (backtracking).
(4) The parsers can be generated mechanically for a wide class of grammars, including all grammars which can be parsed by recursive descent with no backtracking [Knuth (1971)] and those grammars parsable by operator precedence techniques [Floyd (1963)].

The reader may have noticed that the states can be stored on a pushdown stack, since only the rightmost state is ever used at any stage in the parsing process. In a shift move, we stack the new state. In a reduce move, we replace a string of states on top of the stack by the new state.

For example, in parsing the input string *'a,b'* the stack would appear as follows at each of the actions referred to above. (The top of the stack is on the right.)

Action	*Stack*	*Input*
Initial	0	*'a,b$'*
1	0 3	*',b$'*
2	0 2	*',b$'*
3	0 1	*',b$'*
4	0 1 5	*'b$'*

Action	Stack	Input
5	0 1 5 4	'$'
6	0 1 5 6	'$'
7	0 1	'$'
8	0 1	'$'

Thus, the parser control is independent of the trees, and depends only on a stack of states. In practice, we may not need to construct the derivation tree explicitly, if the translation being performed is sufficiently simple. For example, in Section 10, we mention a class of useful translations that can be performed by an LR parser without requiring the forest to be maintained.

If we wish to build the derivation tree, we can easily do so by stacking, along with each state, the root of the tree associated with that state.

5. REPRESENTING THE PARSING ACTION AND GOTO TABLES

Storing the full action and goto tables straightforwardly as matrices is extremely wasteful of space for large parsers. For example, the goto table is typically nearly all blank. In this section we discuss some simple ways of compacting these tables which lead to substantial savings of space; in effect, we are merely representing a sparse matrix more compactly, using a particular encoding.

Let us begin with the shift actions. If x is a terminal symbol and s is a state, the parsing action on x in state s is shift if and only if **goto**(s, x) is nonblank. We will encode the goto into the shift action, using the notation

shift 17

as a shorthand for "shift and attach state 17 to the new node." By encoding the gotos on terminal symbols as part of the action table, we need only consider the gotos on nonterminal symbols. We will encode them by columns; i.e., by nonterminal symbol name. If, on a nonterminal symbol A, there are nonblank entries in the goto table corresponding to states $s_1, s_2, \cdots, s_n$, and we have

$$s_i' = \textbf{goto}(s_i, A), \quad \text{for} \quad i = 1, \cdots, n$$

then we shall encode the column for A in a pseudo-programming language:

A: **if** (state = s_1) **goto** = s_1'
⋮
if (state = s_n) **goto** = s_n'

The goto table of G_1 would be represented in this format as:

LIST: **if** (state = 0) **goto** = 1
ELEMENT: **if** (state = 0) **goto** = 2
if (state = 5) **goto** = 6

It turns out that [Aho and Ullman (1972b)] whenever we do a goto on A, the state will always be one of $s_1, \cdots, s_n$, even if the input string is in error. Thus, one of these branches will always be taken. We shall return to this point later in this section.

We shall encode parsing actions in the same spirit, but by rows of the table. The parsing actions for a state s will also be represented by a sequence of pseudo-programming language statements. If the input symbols $a_1, \cdots, a_n$ have the associated actions $\text{action}_1, \cdots, \text{action}_n$, then we will write:

s: **if** (input = a_1) **action**$_1$
⋮
if (input = a_n) **action**$_n$

As we mentioned earlier, we shall attach **goto**(s,a_i) onto the action if action_i is shift. Similarly, if we have a reduction by the production $A \rightarrow \alpha$, we will usually write

reduce by $A \rightarrow \alpha$

as the action.

For example, the parsing actions for state 1 in the parser for G_1 are represented by:

1: **if** (input = 'a') **error**
if (input = 'b') **error**
if (input = ',') **shift** 5
if (input = '$') **accept**

At first glance this is no saving over the table, since the parsing action table is usually nearly full. We may make a large saving, however, by introducing the notion of a *default* action in the statements. A default action is simply a parsing action which is done irrespective of the input character; there may be at most one of these in each state, and it will be written last. Thus, in state 1 we have two error actions, a shift

action, and an accept action, we shall make the error action the default. We will write:

```
1: if (input = ',') shift 5
   if (input = $ ) accept
   error
```

There is an additional saving which is possible. Suppose a state has both error and reduce entries. Then we may replace all error entries in that state by one of the reduce entries. The resulting parser may make a sequence of reductions where the original parser announced error but the new parser will announce error before shifting the next input symbol. Thus both parsers announce error at the same position in the input, but the new parser may take slightly longer before doing so.

There is a benefit to be had from this modification; the new parsing action table will require less space than the original. For example, state 2 of the parsing action table for G_1 would originally be represented by:

```
2: if (input = 'a') error
   if (input = 'b') error
   if (input = ',') reduce 2
   if (input = '$') reduce 2
```

Applying this transformation, state 2 would be simply represented as:

```
2: reduce 2
```

Thus in a state with reduce actions, we will always have the shift and accept actions precede the reduce actions. One of the reduce actions will become a default action, and we will ignore the error entries. In a state without reduce actions, the default action will be error. We shall discuss other means of cutting down on the size of a parser in Section 8.

6. CONSTRUCTION OF A PARSER FROM A GRAMMAR

How do we construct the parsing action and goto tables of an LR(1) parser for a given grammar? In this section we outline a method that works for a large class of grammars called the lookahead LR(1) (LALR(1)) grammars.

The behavior of an LR parser, as described in the last section, is dictated by the current state. This state reflects the progress of the parse, i.e., it summarizes information about the input string read to this point so that parsing decisions can be made.

Another way to view a state is to consider the state as a representative of an equivalence class of viable prefixes. At every stage of the parsing process, the string formed by concatenating the grammar symbols on the roots of the existing subtrees must be a viable prefix; the current state is the representative of the class containing that viable prefix.

6.1 Sets of Items

In the same way that we needed to discuss partially built trees when talking about parsing, we will need to talk about "partially recognized productions" when we talk about building parsers. We introduce the notion of *item** to deal with this concept. An item is simply a production with a dot (.) placed somewhere in the right-hand side (possibly at either end). For example,

```
[LIST → LIST . ',' ELEMENT]
[ELEMENT → . 'a']
```

are both items of G_1.

We enclose items in square brackets to distinguish them more clearly from productions.

Intuitively, a set of items can be used to represent a stage in the parsing process; for example, the item

$$[A \rightarrow \alpha \,.\, \beta]$$

indicates that an input string derivable from α has just been seen, and, if we next see an input string derivable from β, we may be able to reduce by the production $A \rightarrow \alpha\beta$.

Suppose the portion of the input that we have seen to this point has been reduced to the viable prefix $\gamma\alpha$. Then the item $[A \rightarrow \alpha \,.\, \beta]$ is said to be *valid for* $\gamma\alpha$ if γA is also a viable prefix. In general, more than one item is valid for a given viable prefix; the set of all items which are valid at a particular

* Some authors have used the term "configuration" for item.

stage of the parse corresponds to the current state of the parser.

As an example, let us examine the viable prefix

LIST ','

in G_1. The item

[LIST → LIST ',' . ELEMENT]

is valid for this prefix, since, setting γ to the empty string and α to LIST ',' in the definition above, we see that γ LIST (which is just LIST) is a viable prefix. In other words, when this item is valid, we have seen a portion of the input that can be reduced to the viable prefix, and we expect to see next a portion of the input that can be reduced to ELEMENT.

The item

[LIST → . ELEMENT]

is not valid for LIST ',' however, since setting γ to LIST ',' and α to the empty string we obtain

LIST ',' LIST

which is not a viable prefix.

The reader can (and should) verify that the state corresponding to the viable prefix LIST ',' is associated with the set of items:

[LIST → LIST ',' . ELEMENT]
[ELEMENT → . 'a']
[ELEMENT → . 'b']

If γ is a viable prefix, we shall use $V(\gamma)$ to denote *the set of items that are valid for* γ. If γ is not a viable prefix, $V(\gamma)$ will be empty. We shall associate a state of the parser with each set of valid items and construct the entries in the parsing action for that state from the set of items. There is a finite number of productions, thus only a finite number of items, and thus a finite number of possible states associated with every grammar G.

6.2 Constructing the Collection of Accessible Sets of Items

We shall now describe a constructive procedure for generating all of the states and, at the same time, generating the parsing action and goto table. As a running example, we shall construct parsing action and goto tables for G_1.

First, we augment the grammar with the production

ACCEPT → LIST

where in general LIST would be the start symbol of the grammar (here G_1). A reduction by this production corresponds to the accept action by the parser.

Next we construct $I_0 = V(\text{''})$, the set of items valid for the viable prefix consisting of the empty string. By definition, for G_1 this set must contain the item

[ACCEPT → . LIST]

The dot in front of the nonterminal LIST means that, at this point, we can expect to find as the remaining input any sentence derivable from LIST. Thus, I_0 must also contain the two items

[LIST → . LIST ',' ELEMENT]
[LIST → . ELEMENT]

obtained from the two productions for the nonterminal LIST. The second of the items has a dot in front of the nonterminal ELEMENT, so we should also add to the initial state the items

[ELEMENT → . 'a']
[ELEMENT → . 'b']

corresponding to the two productions for element. These five items constitute I_0. We shall associate state 0 with I_0.

Now suppose that we have computed $V(\gamma)$, the set of items which are valid for some viable prefix γ. Let X be a terminal or nonterminal symbol. We compute $V(\gamma X)$ from $V(\gamma)$ as follows:

(1) For each item of the form $[A \rightarrow \alpha \,.\, X\beta]$ in $V(\gamma)$, we add to $V(\gamma X)$ the item $[A \rightarrow \alpha X \,.\, \beta]$.

(2) We compute the *closure* of the set of items in $V(\gamma X)$; that is, for each item of the form $[B \rightarrow \alpha \,.\, C\beta]$ in $V(\gamma X)$, where C is a nonterminal symbol, we add to $V(\gamma X)$ the items

$$\begin{array}{l}[C \rightarrow \,.\, \alpha_1] \\ [C \rightarrow \,.\, \alpha_2] \\ \quad\vdots \\ [C \rightarrow \,.\, \alpha_n]\end{array}$$

where $C \rightarrow \alpha_1, \cdots, C \rightarrow \alpha_n$ are all the productions in G with C on the left side. If one of these items is already in $V(\gamma X)$ we do not duplicate this item. We continue to apply this process until no new items can be added to $V(\gamma X)$.

It can be shown that steps (1) and (2) compute exactly the items that are valid for γX [Aho and Ullman (1972a)].

For example, let us compute $I_1 = V(\text{LIST})$, the set of items that are valid for the viable prefix LIST. We apply the above construction with $\gamma = ''$ and $X = \text{LIST}$, and use the five items in I_0.

In step (1) of the above construction, we add the items

[ACCEPT → LIST .]

[LIST → LIST . ',' ELEMENT]

to I_1. Since no item in I_1 has a nonterminal symbol immediately to the right of the dot, the closure operation adds no new items to I_1. The reader should verify that these two items are the only items valid for the viable prefix. We shall associate state 1 with I_1.

Notice that the above construction is completely independent of γ; it needs only the items in $V(\gamma)$, and X. For every set of items I and every grammar symbol X the above construction builds a new set of items which we shall call GOTO(I, X); this is essentially the same goto function encountered in the last two sections. Thus, in our example, we have computed

$$\text{GOTO}(I_0, \text{LIST}) = I_1$$

We can extend this GOTO function to strings of grammar symbols as follows:

$$\text{GOTO}(I, '') = I$$

$$\text{GOTO}(I, \gamma X) = \text{GOTO}(\text{GOTO}(I, \gamma), X)$$

where γ is a string of grammar symbols and X is a nonterminal or terminal symbol. If $I = V(\alpha)$, then $I = \text{GOTO}(I_0, \alpha)$. Thus $\text{GOTO}(I_0, \alpha) \neq \phi$ if and only if α is a viable prefix, where $I_0 = V('')$.

The sets of items which can be obtained from I_0 by GOTO's are called the *accessible sets of items*. We build up the set of accessible sets of items by computing GOTO(I, X), for all accessible sets of items I and grammar symbols X, whenever the GOTO construction comes up with a new nonempty set of items, this set of items is added to the set of accessible sets of items and the process continues. Since the number of sets of items is finite, the process eventually terminates.

The order in which the sets of items are computed does not matter, nor does the name given to each set of items. We will name the sets of items $I_0, I_1, I_2, \cdots$ in the order in which we create them. We shall then associate state i with I_i.

Let us return to G_1. We have computed I_0, which contained the items

[ACCEPT → . LIST]
[LIST → . LIST ',' ELEMENT]
[LIST → . ELEMENT]
[ELEMENT → . 'a']
[ELEMENT → . 'b']

We now wish to compute GOTO(I_0, X) for all grammar symbols X. We have already computed

$$\text{GOTO}(I_0, \text{LIST}) = I_1$$

To determine GOTO(I_0, ELEMENT), we look for all items in I_0 with a dot immediately before ELEMENT. We then take these items and move the dot to the right of ELEMENT. We obtain the single item

[LIST → ELEMENT .]

The closure operation yields no new items since this item has no nonterminal to the right of the dot. We call the set with this item I_2. Continuing in this fashion we find that:

GOTO(I_0, 'a') contains only
[ELEMENT → 'a' .]
GOTO(I_0, 'b') contains only
[ELEMENT → 'b' .]

and GOTO(I_0, ',') and GOTO(I_0, '$') are empty. Let us call the two nonempty sets I_3 and I_4. We have now computed all sets of items that are directly accessible from I_0.

We now compute all sets of items that are accessible from the sets of items just computed. We continue computing accessible sets of items until no more new sets of items

are found. The following table shows the collection of accessible sets of items for G_1:

I_0: [ACCEPT → . LIST]
[LIST → . LIST ',' ELEMENT]
[LIST → . ELEMENT]
[ELEMENT → . 'a']
[ELEMENT → . 'b']

I_1: [ACCEPT → LIST .]
[LIST → LIST . ',' ELEMENT]

I_2: [LIST → ELEMENT .]

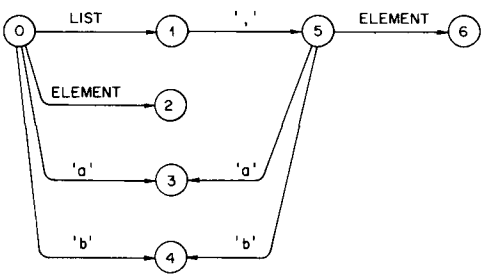
I_3: [ELEMENT → 'a' .]

I_4: [ELEMENT → 'b' .]

I_5: [LIST → LIST ',' . ELEMENT]
[ELEMENT → . 'a']
[ELEMENT → . 'b']

I_6: [LIST → LIST ',' ELEMENT .]

The GOTO function on this collection can be portrayed as a directed graph in which the nodes are labeled by the sets of items and the edges by grammar symbols, as follows:



Here, we used i in place of I_i.
For example, we observe

GOTO(0, ") = 0
GOTO(0, LIST ',') = 5
GOTO(0, LIST ',' ELEMENT) = 6

Observe that there is a path from vertex 0 to a given node if and only if that path spells out a viable prefix. Thus, GOTO(0, 'ab') is empty, since 'ab' is not a viable prefix.

6.3 Constructing the Parsing Action and Goto Tables from the Collection of Sets of Items

The parsing action table is constructed from the collection of accessible sets of items. From the items in each set of items I_s we generate parsing actions. An item of the form

$$[A \rightarrow \alpha \ . \ 'a' \ \beta]$$

in I_s generates the parsing action

if (input = 'a') **shift** t

where GOTO(I_s, 'a') = I_t.

An item with the dot at the right end of the production is called a *completed item.* A completed item $[A \rightarrow \alpha \ .]$ indicates that we may reduce by production $A \rightarrow \alpha$. However, with an LR(1) parser we must determine on what input symbols this reduction is possible. If 'a_1', 'a_2', $\cdots$, 'a_n' are these symbols and 'a_1', 'a_2', $\cdots$, 'a_n' are not associated with shift or accept actions, then we would generate the sequence of parsing actions:

if(input = 'a_1') **reduce by:** $A \rightarrow \alpha$
if(input = 'a_2') **reduce by:** $A \rightarrow \alpha$
$\vdots$
if(input = 'a_n') **reduce by:** $A \rightarrow \alpha$

As we mentioned in the last section, if the set of items contains only one completed item, we can replace this sequence of parsing actions by the default reduce action

reduce by: $A \rightarrow \alpha$

This parsing action is placed after all shift and accept actions generated by this set of items.

If a set of items contains more than one completed item, then we must generate conditional reduce actions for all completed items except one. In a while we shall explain how to compute the set of input symbols on which a given reduction is permissible.

If a completed item is of the form

$$[\text{ACCEPT} \rightarrow S \ .]$$

then we generate the accept action

if(input = '$') **accept**

where '$' is the right endmarker for the input string.

Finally, if a set of items generates no reduce action, we generate the default error statement. This statement is placed after all shift and accept actions generated from the set of items.

Returning to our example for G_1, from I_0 we would generate the parsing actions:

if(input = '*a*') **shift** 3
if(input = '*b*') **shift** 4
error

Notice that these are exactly the same parsing actions as those for state 0 in the parser of Section 4. Similarly, I_3 generates the action

reduce by: ELEMENT → '*a*'

The goto table is used to compute the new state after a reduction. For example, when the reduction in state 3 is performed we always have state 0 to the left of '*a*'. The new state is determined by simply noting that

$$\text{GOTO}(I_0, \text{ELEMENT}) = I_2$$

This gives rise to the code

if(state = 0) **goto** = 2

for ELEMENT in the goto table.

In general, if nonterminal A has precisely the following GOTO's in the GOTO graph:

$$\begin{aligned} \text{GOTO}(I_{s_1}, A) &= I_{t_1} \\ \text{GOTO}(I_{s_2}, A) &= I_{t_2} \\ &\vdots \\ \text{GOTO}(I_{s_n}, A) &= I_{t_n} \end{aligned}$$

then we would generate the following representation for column A of the goto table:

$$\begin{aligned} A\text{: } &\textbf{if}(\text{state} = s_1)\ \textbf{goto} = t_1 \\ &\textbf{if}(\text{state} = s_2)\ \textbf{goto} = t_2 \\ &\qquad\vdots \\ &\textbf{if}(\text{state} = s_n)\ \textbf{goto} = t_n \end{aligned}$$

Thus, the goto table is simply a representation of the GOTO function of the last section, applied to the nonterminal symbols.

We must now determine the input symbols on which each reduction is applicable. This will enable us to detect ambiguities and difficult-to-parse constructs in the grammar, and to decide between reductions if more than one is possible in a given state. In general, this is a complex task; the most general solution of this problem was given by [Knuth (1965)], but his algorithm suffers from large time and memory requirements. Several simplifications have been proposed, notably by [DeRemer (1969 and 1971)], which lack the full generality of Knuth's technique, but can construct practical parsers in reasonable time for a large class of languages. We shall describe an algorithm that is a simplification of Knuth's algorithm which resolves all conflicts that can be resolved when the parser has the states as given above.

6.4 Computing Lookahead Sets

Suppose $[A \rightarrow \alpha \,.\, \beta]$ is an item that is valid for some viable prefix $\gamma\alpha$. We say that input symbol '*a*' is *applicable* for $[A \rightarrow \alpha \,.\, \beta]$ if, for some string of terminals '*w*', both $\gamma\alpha\beta$'*aw*' and γA'*aw*' are right sentential forms. The right endmarker '$' is applicable for $[A \rightarrow \alpha \,.\, \beta]$ if both $\gamma\alpha\beta$ and γA are right sentential forms.

This definition has a simple intuitive explanation when we consider completed items. Suppose input symbol '*a*' is applicable for completed item $[A \rightarrow \alpha \,.]$. If an LR(1) parser makes the reduction specified by this item on the applicable input symbol '*a*', then the parser will be able to make at least one more shift move without encountering an error.

The set of symbols that are applicable for each item will be called the *lookahead set* for that item. From now on we shall include the lookahead set as part of an item. The production with the dot somewhere in the right side will be called the *core* of the item. For example,

([ELEMENT → '*a*' .], {',', '$'})

is an item of G_1 with core

[ELEMENT → '*a*' .]

and lookahead set {',', '$'}.

We shall now describe an algorithm that will compute the sets of valid items for a grammar where the items include their

lookahead sets. Recall that in the last section items in a set of items arose in two ways: by goto calculations, and then by the closure operation. The first type of calculation is very simple; if we have an item of the form

$$([A \rightarrow \alpha \cdot X\beta], L)$$

where X is a grammar symbol and L is a lookahead set, then when we perform the goto operation on X on this item, we obtain the item

$$([A \rightarrow \alpha X \cdot \beta], L)$$

(i.e., the lookahead set is unchanged).

It is somewhat harder to compute the lookahead sets in the closure operation. Suppose there is an item of the form

$$([A \rightarrow \alpha \cdot B\beta], L)$$

in a set of items, where B is a nonterminal symbol. We must add items of the form

$$([B \rightarrow \cdot \delta], L')$$

where $B \rightarrow \delta$ is some production in the grammar. The new lookahead set L' will contain all terminal symbols which are the first symbol of some sentence derivable from any string of the form β 'a', where 'a' is a symbol in L.

If, in the course of carrying out this construction, a set of items is seen to contain items with the same core; e.g.,

$$([A \rightarrow \alpha \cdot \beta], L_1)$$

and

$$([A \rightarrow \alpha \cdot \beta], L_2)$$

then these items are merged to create a single item; e.g., $([A \rightarrow \alpha \cdot \beta], L_1 \cup L_2)$.

We shall now describe the algorithm for constructing the collection of sets of items in more detail by constructing the valid sets of items for grammar G_1. Initially, we construct I_0 by starting with the single item

([ACCEPT → . LIST], {'$'})

We then compute the closure of this set of items. The two productions for LIST give rise to the two items

([LIST → . LIST ',' ELEMENT], {'$'})

and ([LIST → . ELEMENT], {'$'})

The first of these two items gives rise, through the closure operation, to two additional items

([LIST → . LIST ',' ELEMENT], {','})

and ([LIST → . ELEMENT], {','})

since the first terminal symbol of any string derivable from

',' ELEMENT '$'

is always ','. Since all items with the same core are merged into a single item with the same core and the union of the lookahead sets, we currently have the following items in I_0:

([ACCEPT → . LIST], {'$'})
([LIST → . LIST ',' ELEMENT], {',', '$'})
([LIST → . ELEMENT], {',', '$'})

The first two of these items no longer give rise to any new items when the closure operation is applied. The third item gives rise to the two new items:

([ELEMENT → . 'a'], {',', '$'})
([ELEMENT → . 'b'], {',', '$'})

and these five items make up I_0.

We shall now compute

$$I_2 = \text{GOTO}(I_0, \text{'a'}).$$

First we add the item

([ELEMENT → 'a' .], {',', '$'})

to I_2, since 'a' appears to the right of the dot of one item in I_0. The closure operation adds no new items to I_2.

I_2 contains a completed item. The lookahead set {',', '$'} tells us on which input symbols the reduction is applicable.

The reader should verify that the complete collection of sets of items for G_1 is:

I_0: [ACCEPT → . LIST], {'$'}
[LIST → . LIST ',' ELEMENT], {',', '$'}
[LIST → ELEMENT], {',', '$'}
[ELEMENT → . 'a'], {',', '$'}
[ELEMENT → 'b'], {',', '$'}

I_1: [ACCEPT → LIST .], {'$'}
[LIST → LIST . ',' ELEMENT], {',', '$'}

I_2: [LIST → ELEMENT .], {',', '$'}

I_3: [ELEMENT → 'a' .], {',', '$'}

I_4: [ELEMENT → 'b' .], {',', '$'}

I_5: [LIST → LIST ',' . ELEMENT], {',', '$'}
[ELEMENT → 'a'], {',', '$'}
[ELEMENT → . 'b'], {',', '$'}

I_6: [LIST → LIST ',' ELEMENT .], {',', '$'}

Although the situation does not occur here, if we generate a set of items I_t such that I_t has the same set of cores as some other set of items I_s already generated, but $I_s \neq I_t$, then we combine I_s and I_t into a new set of items I by merging the lookahead sets of items with the same cores. We must then compute GOTO(I, X) for all grammar symbols X.

The lookahead sets on the completed items give the terminal symbols for which the reductions should be performed. There is a possibility that there are ambiguities in the grammar, or the grammar is too complex to allow a parser to be constructed by this technique; this causes conflicts to be discovered in the actions of the parser. For example, suppose there is a set of items I_s in which 'a' gives rise to the parsing action shift because GOTO(I_s, 'a') exists. Suppose also that there is a completed item

$$([A \rightarrow \alpha \,.], L)$$

in I_s, and that the terminal symbol 'a' is in the lookahead set L. Then we have no way of knowing which action is correct in state s when we see an 'a'; we may shift 'a', or we may reduce by $A \rightarrow \alpha$. Our only recourse is to report a *shift-reduce* conflict.

In the same way, if there are two reductions possible in a state because two completed items contain the same terminal symbol in their lookahead sets, then we cannot tell which reduction we should do; we must report a *reduce-reduce* conflict.

Instead of reporting a conflict we may attempt to proceed by carrying out all conflicting parsing actions, either by parallel simulation [Earley (1970)] or by backtracking [Pager (1972b)].

A set of items is *consistent* or *adequate* if it does not generate any shift-reduce or reduce-reduce conflicts. A collection of sets of items is *valid* if all its sets of items are consistent; our collection of sets of items for G_1 is valid.

We summarize the parsing action and goto table construction process:

(1) Given a grammar G, augment the grammar with a new initial production

$$\text{ACCEPT} \rightarrow S$$

where S is the start symbol of G.

(2) Let I be the set with the one item

$$([\text{ACCEPT} \rightarrow . \, S], \{\text{'\$'}\})$$

Let I_0 be the closure of I.

(3) Let C, the current collection of accessible sets of items, initially contain only I_0.

(4) For each I in C, and for each grammar symbol X, compute $I' = \text{GOTO}(I,X)$. Three cases can occur:
   a. $I' = I''$ for some I'' already in C. In this case, do nothing.
   b. If the set of cores of I' is distinct from the set of cores of a set of items already in C, then add I' to C.
   c. If the set of cores of I' is the same as the set of cores of some I'' already in A but $I' \neq I''$, then let I''' be the set of items

$$([A \rightarrow \alpha \,.\, \beta], L_1 \cup L_2)$$

such that

$([A \rightarrow \alpha \,.\, \beta], L_1)$ is in I' and
$([A \rightarrow \alpha \,.\, \beta], L_2)$ is in I''.

Replace I'' by I''' in C.

(5) Repeat step 4 until no new sets of items can be added to C. C is called the *LALR(1) collection* of sets of items for G.

(6) From C try to construct the parsing action and goto tables as in Section 6.3.

If this technique succeeds in producing a collection of sets of items for a given grammar in which all sets of items are consistent, then that grammar is said to be an *LALR(1) grammar*. LALR(1) grammars include many important classes of grammars, including the LL(1) grammars [Lewis and Stearns (1968)], the simple mixed strategy precedence grammars [McKeeman, Horning, and Wortman (1970)], and those parsable by operator precedence techniques. Techniques

for proving these inclusions can be found in [Aho and Ullman (1972a and 1973a)].

Step (4) can be rather time-consuming to implement. A simpler, but less general, approach would be to proceed as follows. Let FOLLOW(A) be the set of terminal symbols that can follow nonterminal symbol A in a sentential form. If A can be the rightmost symbol of a sentential form, then '$' is included in FOLLOW($A$). We can compute the sets of items without lookaheads as in Section 6.2. Then in each completed item $[A \rightarrow \alpha\ .]$ we can approximate the lookahead set L for this item by FOLLOW(A) (In general, L is a subset of FOLLOW(A).) The resulting collection of sets of items is called the *SLR(1) collection*. If all sets of items in the SLR(1) collection are consistent, then the grammar is said to be *simple* LR(1) [DeRemer (1971)]. Although not every LALR(1) grammar is simple LR(1), every language generated by an LALR(1) grammar is also generated by a simple LR(1) grammar ([Aho and Ullman (1973a)] contains more details).

7. PARSING AMBIGUOUS GRAMMARS

It is undesirable to have undetected ambiguities in the definition of a programming language. However, an ambiguous grammar can often be used to specify certain language constructs more easily than an equivalent unambiguous grammar. We shall also see that we can construct more efficient parsers directly from certain ambiguous grammars than from equivalent unambiguous grammars.

If we attempt to construct a parser for an ambiguous grammar, the LALR(1) parser construction technique will generate at least one inconsistent set of items. Thus, the parser generation technique can be used to determine that a grammar is unambiguous. That is to say, if no inconsistent sets of items are generated, the grammar is guaranteed to be unambiguous. However, if an inconsistent set of items is produced, then all we can conclude is that the grammar is not LALR(1). The grammar may or may not be ambiguous. (There is no general algorithm to determine if a context-free grammar is ambiguous (see, for example [Aho and Ullman (1972a)]).

Inconsistent sets of items are useful in pinpointing difficult-to-parse or ambiguous constructions in a given grammar. For example, a production of the form

$$A \rightarrow AA$$

in any grammar will make that grammar ambiguous and cause a parsing action conflict to arise from sets of items containing the items with the cores

$$[A \rightarrow A\ A\ .]$$
$$[A \rightarrow A\ .\ A]$$

Constructions which are sufficiently complex to require more than one symbol of lookahead also result in parsing action conflicts. For example, the grammar

$$S \rightarrow A\ \text{`}a\text{'}$$
$$A \rightarrow \text{`}a\text{'} \mid \text{''}$$

is an LALR(2) but not LALR(1) grammar.

Experience with an LALR(1) parser generator called YACC at Bell Laboratories has shown that a few iterations with the parser generator are usually sufficient to resolve the conflicts in an LALR(1) collection of sets of items for a reasonable programming language.

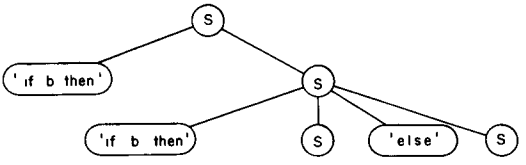
Example 7.1: Consider the following productions for "if-then" and "if-then-else" statements:

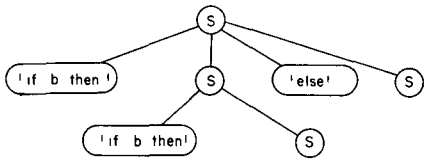
$$S \rightarrow \text{`if } b \text{ then'}\ S$$
$$S \rightarrow \text{`if } b \text{ then'}\ S\ \text{`else'}\ S$$

If these two productions appear in a grammar, then that grammar will be ambiguous; the string

$$\text{`if } b \text{ then if } b \text{ then'}\ S\ \text{`else'}\ S$$

can be parsed in two ways as shown:





In most programming languages, the first phrasing is preferred. That is, each new 'else' is to be associated with the closest "unelsed" 'then'.

A grammar using these ambiguous productions to specify if-then-else statements will be smaller and, we feel, easier to comprehend than an equivalent unambiguous grammar. In addition if a grammar has only ambiguities of this type, then we can construct a valid LALR(1) parser for the grammar merely by resolving each shift-reduce conflict in favor of shift [Aho, Johnson, and Ullman (1973)].

Example 7.2: Consider the ambiguous grammar*

$$S \rightarrow \text{'if } b \text{ then' } S$$
$$S \rightarrow \text{'if } b \text{ then' } S \text{ 'else' } S$$
$$S \rightarrow \text{'}a\text{'}$$

in which each else is to be associated with the last unelsed 'then'. The LALR(1) collection of sets of items for this grammar is as follows:

I_0: [ACCEPT → · S], {'$'}
[S → . 'if b then' S], {'$'}
[S → . 'if b then' S 'else' S], {'$'}
[S → 'a'], {'$'}

I_1 [ACCEPT → S .], {'$'}

I_2: [S → 'if b then' . S], {'else', '$'}
[S → 'if b then' S 'else' S], {'else', '$'}
[S → 'if b then' S], {'else', '$'}
[S → . 'if b then' S 'else' S], {'else', '$'}
[S → . 'a'], {'else', '$'}

I_3: [S → 'a'], {'else', '$'}

I_4: [S → 'if b then' S .], {'else', '$'}
[S → 'if b then' S . 'else' S], {'else', '$'}

I_5: [S → 'if b then' S 'else' S], {'else', '$'}
[S → . 'if b then' S], {'else', '$'}
[S → . 'if b then' S 'else' S], {'else', '$'}
[S → . 'a'], {'else', '$'}

I_6: [S → 'if b then' S 'else' S .], {'else', '$'}

I_4 contains a shift-reduce conflict. On the input 'else', I_4 says that either a shift move to I_5 is permissible, or a reduction by production

$$S \rightarrow \text{'if } b \text{ then' } S$$

is possible. If we choose to shift, we shall associate the incoming 'else' with the last unelsed 'then'. This is evident because the item with the core

$$[S \rightarrow \text{'if } b \text{ then' } S \text{ . 'else' } S]$$

in I_4 gives rise to the shift action.

The complete parsing action table, with the conflict resolved, and the goto table constructed from this collection of sets of items are shown below:

Parsing Action Table

0: **if**(input = 'if b then') **shift** 2
if(input = 'a') **shift** 3
error
1: **if**(input = $) **accept**
error
2: **if**(input = 'if b then') **shift** 2
if(input = 'a') **shift** 3
error
3: **reduce by:** $S \rightarrow$ 'a'
4: **if**(input = 'else') **shift** 5
reduce by: $S \rightarrow$ 'if b then' S
5: **if**(input = 'if b then') **shift** 2
if(input = 'a') **shift** 3
error
6: **reduce by:** $S \rightarrow$ 'if b then' S 'else' S

Goto Table

S: **if**(state = 0) **goto** = 1
if(state = 2) **goto** = 4
goto = 6

Given an ambiguous grammar, with the appropriate rules for resolving the ambiguities we can often directly produce a smaller parser from the ambiguous grammar than from the equivalent unambiguous grammar.

* The following grammar is an equivalent unambiguous grammar:

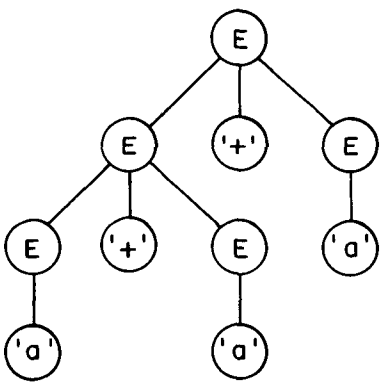
$$S \rightarrow \text{'if } b \text{ then' } S$$
$$S \rightarrow \text{'if } b \text{ then' } S_1 \text{ 'else' } S$$
$$S \rightarrow \text{'}a\text{'}$$
$$S_1 \rightarrow \text{'if } b \text{ then' } S_1 \text{ 'else' } S_1$$
$$S_1 \rightarrow \text{'}a\text{'}$$

However, some of the "optimizations" discussed in the next section will make the parser for the unambiguous grammar as small as that for the ambiguous grammar.

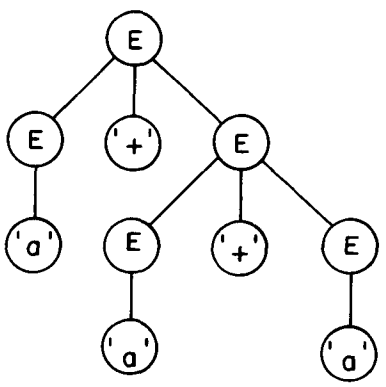
Example 7.3: Consider the following grammar G_3 for arithmetic expressions:

$$E \rightarrow E\ \text{`+'}\ E$$
$$E \rightarrow E\ \text{`*'}\ E$$
$$E \rightarrow \text{`('}E\text{`)'}$$
$$E \rightarrow \text{`}a\text{'}$$

where 'a' stands for any identifier. Assuming that + and * are both left associative and * has higher precedence than +, there are two things wrong with this grammar. First, it is ambiguous in that the operands of the binary operators '+' and '*' can be associated in any arbitrary way. For example, '$a + a + a$' can be parsed as



or as



The first parsing gives the usual left-to-right associativity, the second a right-to-left associativity.

If we rewrote the grammar as G_4:

$$E \rightarrow E\ \text{`+'}\ T$$
$$E \rightarrow E\ \text{`*'}\ T$$
$$E \rightarrow T$$
$$T \rightarrow \text{`('}E\text{`)'}$$
$$T \rightarrow \text{`}a\text{'}$$

then we would have eliminated this ambiguity by imposing the normal left-to-right associativity for + and *. However, this new grammar has still one more defect; + and * have the same precedence, so that an expression of the form '$a+a*a$' would be evaluated as $(a+a)*a$. To eliminate this, we must further rewrite the grammar as G_5:

$$E \rightarrow E\ \text{`+'}\ T$$
$$E \rightarrow T$$
$$T \rightarrow T\ \text{`*'}\ F$$
$$T \rightarrow F$$
$$F - \text{`('}E\text{`)'}$$
$$F \rightarrow \text{`}a\text{'}$$

We can now construct a parser for G_5 quite easily, and find that we have 12 states; if we count the number of parsing actions in the parser (i.e., the sum of the number of shift and reduce actions in all states together with the goto actions) we see that the parser for G_5 has 35 actions.

In contrast, the parser for G_3 has only 10 states, and 29 actions. A considerable part of the saving comes from the elimination of the nonterminals T and F from G_5, as well as the elimination of the productions $E \rightarrow T$ and $T \rightarrow F$.

Let us discuss the resolution of parsing action conflicts in G_3 in somewhat more detail. There are two sets of items in the LALR(1) collection of sets of items for G_3 which generate conflicts in their parsing actions:

$$[E \rightarrow E\ .\ \text{`+'}\ E],\ \{\text{`+'}, \text{`*'}, \text{`)'}, \text{`\$'}\}$$
$$[E \rightarrow E\ .\ \text{`*'}\ E],\ \{\text{`+'}, \text{`*'}, \text{`)'}, \text{`\$'}\}$$
$$[E \rightarrow E\ \text{`+'}\ E\ .],\ \{\text{`+'}, \text{`*'}, \text{`)'}, \text{`\$'}\}$$

and

$$[E \rightarrow E\ .\ \text{`+'}\ E],\ \{\text{`+'}, \text{`*'}, \text{`)'}, \text{`\$'}\}$$
$$[E \rightarrow E\ .\ \text{`*'}\ E],\ \{\text{`+'}, \text{`*'}, \text{`)'}, \text{`\$'}\}$$
$$[E \rightarrow E\ \text{`*'}\ E\ .],\ \{\text{`+'}, \text{`*'}, \text{`)'}, \text{`\$'}\}$$

In both sets of items, shift-reduce conflicts arise on the two terminal symbols '+' and '*'. For example, in the first set of items on an input of '+' we may generate either a reduce action or a shift action. Since we wish + to be left associative, we wish to reduce on this input; a shift would have the effect of delaying the reduction until more of the string had been read, and would imply right associativity. On the input symbol '*', however, if we did the reduction we would end up parsing the string '$a+a*a$' as $(a+a)*a$; that is, we would not give * higher precedence than +. Thus, it is correct to shift on this input. Using similar reasoning, we see that it is always correct to generate a reduce action from the second set of items; on the input symbol '*' this is a result of the left associativity of *, while on the input symbol '+' this reflects the precedence relation between + and *.

We conclude this section with an example of how this reasoning can be applied to our grammar G_1. We noted earlier that the grammar G_2:

LIST → LIST ',' LIST
LIST → 'a'
LIST → 'b'

is ambiguous, but this ambiguity should no longer be of concern. Assuming that the language designer wants to treat ',' as a left associative operator, then we can produce a parser which is smaller and faster than the parser for G_1 produced in the last section. The smaller parser looks like:

Parsing Action Table

0: **if**(input = 'a') **shift** 2
   if(input = 'b') **shift** 3
   error
1: **if**(input = '$') **accept**
   if(input = ',') **shift** 4
   error
2: **reduce by:** LIST → 'a'
3: **reduce by:** LIST → 'b'
4: **if**(input = 'a') **shift** 2
   if(input = 'b') **shift** 3
   error
5: **reduce by:** LIST → LIST ',' LIST

Goto Table

LIST: **if**(state = 0) **goto** = 1
      goto = 5

Notice that we have only 14 parsing actions in this parser, compared to the 16 which we had in the earlier parser for G_1. In addition, the derivation trees produced by this parser are smaller since the nodes corresponding to the nonterminal symbol ELEMENT are no longer there. This in turn means that the parser makes fewer actions when parsing a given input string. Parsing of ambiguous grammars is discussed by [Aho, Johnson, and Ullman (1973)] in more detail.

8. OPTIMIZATION OF LR PARSERS

There are a number of ways of reducing the size and increasing the speed of an LR(1) parser without affecting its good error-detecting capability. In this section we shall list a few of many transformations that can be applied to the parsing action and goto tables of an LR(1) parser to reduce their size. The transformations we list are some simple ones that we have found to be effective in practice. Many other transformations are possible and a number of these can be found in the references at the end of this section.

8.1 Merging Identical States

The simplest and most obvious "optimization" is to merge states with common parsing actions. For example, the parsing action table for G_1 given in Section 5 contains identical actions in states 0 and 5. Thus, it is natural to represent this in the parser as:

0: 5: **if**(input = 'a') **shift** 3
      if(input = 'b') **shift** 4
      error

Clearly the behavior of the LR(1) parser using this new parsing action table is the same as that of the LR(1) parser using the old table.

8.2 Subsuming States

A slight generalization of the transformation in Section 8.1 is to eliminate a state whose parsing actions are a suffix of the actions of another state. We then label the beginning of the suffix by the eliminated state. For example, if we have:

```
n: if(input = 'x') shift p
   if(input = 'y') shift q
   error
```

and

```
m: if(input = 'y') shift q
   error
```

then we may eliminate state m by adding the label into the middle of state n:

```
n: if(input = 'x') shift p
m: if(input = 'y') shift q
   error
```

Permuting the order of these statements can increase the applicability of this optimization. (See Ichbiah and Morse (1970) for suggestions on the implementation of this optimization.)

8.3 Elimination of Reductions by Single Productions

A single production is one of the form $A \rightarrow X$, where A is a nonterminal and X is a grammar symbol. If this production is not of any importance in the translation, then we say that the single production is *semantically insignificant.* A common situation in which single productions arise occurs when a grammar is used to describe the precedence levels and associativities of operators (see grammar G_5 of Example 7.3). We can always cause an LR parser to avoid making these reductions; by doing so we make the LR parser faster, and reduce the number of states. (With some grammars, the size of the "optimized" form of the parsing action table may be greater than the unoptimized one.)

We shall give an example in terms of G_1 which contains the single production

$$\text{LIST} \rightarrow \text{ELEMENT}$$

We shall eliminate reductions by this production from the parser for G_1 found in Section 5. The only state which calls for a reduction by this production is state 2. Moreover, the only way in which we can get to state 2 is by the goto action

ELEMENT: **if**(state = 0) **goto** = 2

After the parser does the reduction in state 2, it immediately refers to the goto action

LIST: **goto** = 1

at which time the current state becomes 1. Thus, the rightmost tree is only labeled with state 2 for a short period of time; state 2 represents only a step on the way to state 1. We may eliminate this reduction by the single production by changing the goto action under ELEMENT to:

ELEMENT: **if**(state = 0) **goto** = 1

so that we bypass state 2 and go directly to state 1. We now find that state 2 can never be reached by any parsing action, so it can be eliminated. Moreover, it turns out here (and frequently in practice as well) that the goto actions for LIST and ELEMENT become *compatible* at this point; that is, the actions do not differ on the same state. It is always possible to merge compatible goto actions for nonterminals; the resulting parser has one less state, and one less goto action.

Example 8.1: The following is a representation of the parsing action and goto tables for an LR(1) parser for G_1. It results from the parsing action and goto tables in Section 5 by applying state merger (Section 8.1), and eliminating the reduction by the single production.

Parsing Action Table

```
0. 5. if (input = 'a') shift 3
      if (input = 'b') shift 4
      error
1:    if (input = ',') shift 5
      if (input = $) accept
      error
3·    reduce by: ELEMENT → 'a'
4     reduce by: ELEMENT → 'b'
6·    reduce by: LIST → LIST ',' ELEMENT
```

Goto Table

```
LIST: ELEMENT: if(state = 0) goto = 1
                      goto = 6
```

These tables are identical with those for the ambiguous version of G_1, after the equal states have been identified. These tables differ only in that the nonterminal symbols LIST and ELEMENT have been explicitly merged in the ambiguous grammar, while the distinction is still nominally made in the tables above.

In the general case, there may be a number of states which call for reductions by the same single production, and there may be other parsing actions in the states which call for these reductions. It is not always possible, in general, to perform these modifications without increasing the number of states; conditions which must be satisfied in order to profitably carry out this process are given in [Aho and Ullman (1973b)]. It is enough for our purposes to notice that if a reduction by a single production $A \rightarrow X$ is to be eliminated, and if this reduction is generated by exactly one set of items containing the item with the core

$$[A \rightarrow X \ .]$$

then this single production can be eliminated. It turns out that the single productions which arise in the representation of operator precedence or associativity can always be eliminated; the result is typically the same as if an ambiguous grammar were written, and the conflicts resolved as discussed in Section 6. However, the ambiguous grammar generates the reduced parser immediately, without needing this optimizing algorithm [Aho, Johnson, and Ullman (1973)].

Other approaches to optimization of LR parsers are discussed by [Aho and Ullman (1972b)], [Anderson (1972)], [Jolliat (1973)], and [Pager (1970)]. [Anderson, Eve, and Horning (1973)], [Demers (1973)], and [Pager (1974)] also discuss the elimination of reductions by single productions.

9. ERROR RECOVERY

A properly designed LR parser will announce that an error has occurred as soon as there is no way to make a valid continuation to the input already scanned. Unfortunately, it is not always easy to decide what the parser should do when an error is detected; in general, this depends on the environment in which the parser is operating. Any scheme for error recovery must be carefully interfaced with the lexical analysis and code generation phases of compilation, since these operations typically have "side effects" which must be undone before the error can be considered corrected. In addition, a compiler should recover gracefully from each error encountered so that subsequent errors can also be detected.

LR parsers can accommodate a wide variety of error recovery stratagems. In place of each error entry in each state, we may insert an error correction routine which is prepared to take some extraordinary actions to correct the error. The description of the state as given by the set of items frequently provides enough context information to allow for the construction of sophisticated error recovery routines.

We shall illustrate one simple method by which error recovery can be introduced into the parsing process. This method is only one of many possible techniques. We introduce error recovery productions of the form

$$A \rightarrow \textbf{error}$$

into the grammar for certain selected nonterminals. Here, **error** is a special terminal symbol. These error recovery productions will introduce items with cores of the form

$$[A \rightarrow \ . \ \textbf{error}]$$

into certain states, as well as introducing new states of the form

$$[A \rightarrow \textbf{error} \ .]$$

When the LR parser encounters an error, it can announce error and replace the current input symbol by the special terminal symbol **error.** The parser can then discard trees from the parse forest, one at a time from right-to-left, until the current state (the state on the rightmost tree in the parse forest) has a parsing action shift on the input **error**. The parser has now reached a state with at least one item of the form

$$[A \rightarrow \ . \ \textbf{error}]$$

The parser can then perform the shift

action and reduce by one of the error recovery productions

$$A \rightarrow \textbf{error}$$

(If more than one error recovery production is present, a choice would have to be specified.) On reducing, the parser can perform a hand-tailored action associated with this error situation. One such action could be to skip forward on the input until an input symbol 'a' was found such that 'a' can legitimately occur either as the last symbol of a string generated by A or as the first symbol of a string that can follow A.

Certain automatic error recovery actions are also possible. For example, the error recovery productions can be mechanically generated for any specified set of nonterminals. Parsing and error recovery can proceed as above, except that on reducing by an error recovery production, the parser can automatically discard input symbols until it finds an input symbol, say 'a', on which it can make a legitimate parsing action, at which time normal parsing resumes. This would correspond to assuming that an error was encountered while the parser was looking for a phrase that could be reduced to nonterminal A. The parser would then assume that by skipping forward on the input to the symbol 'a' it would have found an instance of nonterminal A.

Certain error recovery schemes can produce an avalanche of error messages. To avoid a succession of error messages stemming from an inappropriate recovery, a parser might suppress the announcement of subsequent errors until a certain number of successful shift actions have occurred.

We feel that, at present, there is no efficient general "solution" to the error recovery problem in compiling. We see faults with any uniform approach, including the one above. Moreover, the success of any given approach can vary considerably from application to application. We feel that if a language is cleanly designed and well human-engineered, automatic error recovery will be easier as well.

Particular methods of error recovery during parsing are discussed by [Aho and Peterson (1972)], [Graham and Rhodes (1973)], [James (1972)], [Leinius (1970)], [McGruther (1972)], [Peterson (1972)], and [Wirth (1968)].

10. OUTPUT

In compiling, we are not interested in parsing but rather in producing a translation for the source program. LR parsing is eminently suitable for producing bottom-up translations.

Any translation which can be expressed as the concatenation of outputs which are associated with each production can be readily produced by an LR parser, without having to construct the forest representing the derivation tree. For example, we can specify a translation of arithmetic expressions from infix notation to postfix Polish notation in this way. To implement this class of translations, when we reduce, we perform an output action associated with that production. For example, to produce postfix Polish from G_1, we can use the following translation scheme:

	Production	*Translation*
(1)	$E \rightarrow E$ '+' E	'+'
(2)	$E \rightarrow E$ '*' E	'*'
(3)	$E \rightarrow$ '('E')'	
(4)	$E \rightarrow$ 'a'	'a'

Here, as in Section 7, we assume that + and * are left associative, and that * has higher precedence than +. The translation element is the output string to be emitted when the associated reduction is done. Thus, if the input string

$$'a + a * (a + a)'$$

is parsed, the output will be

$$'aaaa + * +'$$

These parsers can also produce three address code or the parse tree as output with the same ease. However, more complex translations may require more elaborate intermediate storage. Mechanisms for implementing these translations are discussed in [Aho and Ullman (1973a)] and in [Lewis, Rosenkrantz, and Stearns (1973)]. It is our current belief that, if a complicated translation is called for, the best way of imple-

menting it is by constructing a tree. Optimizing transformations can then massage this tree before final code generation takes place. This scheme is simple and has low overhead when the input is in error.

11. CONCLUDING REMARKS

LR parsers belong to the class of shift-reduce parsing algorithms [Aho, Denning, and Ullman (1972)]. These are parsers that operate by scanning their input from left-to-right, shifting input symbols onto a pushdown stack until the handle of the current right sentential form is on top of the stack; the handle is then reduced. This process is continued either until all of the input has been scanned and the stack contains only the start symbol, or until an error has been encountered.

During the 1960s a number of shift-reduce parsing algorithms were found for various subclasses of the context-free grammars. The operator precedence grammars [Floyd (1963]), the simple precedence grammars [Wirth and Weber (1966)], the simple mixed strategy precedence grammars [McKeeman, Horning, and Wortman (1970)], and the uniquely invertible weak precedence grammars [Ichbiah and Morse (1970)] are some of these subclasses. The definitions of these classes of grammars and the associated parsing algorithms are discussed in detail in [Aho and Ullman (1972a)].

In 1965 Knuth defined a class of grammars which he called the LR(k) grammars. These are the context-free grammars that one can naturally parse bottom-up using a deterministic pushdown automaton with k-symbol lookahead to determine shift-reduce parsing actions. This class of grammars includes all of the other shift-reduce parsable grammars and admits of a parsing procedure that appears to be at least as efficient as the shift-reduce parsing algorithms given for these other classes of grammars. [Lalonde, Lee, and Horning (1971)] and [Anderson, Eve, and Horning (1973)] provide some empirical comparisons between LR and precedence parsing that support this conclusion.

In his paper Knuth outlined a method for constructing an LR parser for an LR grammar. However this algorithm results in parsers that are too large for practical use. A few years later [Korenjak (1969)] and particularly [DeRemer (1969 and 1971)] succeeded in substantially modifying Knuth's original parser constructing procedure to produce parsers of practical size. Substantial progress has been made since in improving the size and performance of LR parsers.

The general theory of LR(k) grammars and languages is developed in [Aho and Ullman (1972a and 1973a)]. Proofs of the correctness and efficacy of many of the constructions in this paper can be found there.

Perhaps the biggest advantage of LR parsing is that small, fast parsers can be mechanically generated for a large class of context-free grammars, that includes all other classes of grammars for which nonbacktracking parsing algorithms can be mechanically generated. In addition, LR parsers are capable of detecting syntax errors at the earliest opportunity in a left-to-right scan of an input string, a property not enjoyed by many other parsing algorithms.

Just as we can parse by constructing a derivation tree for an input string bottom-up (from the leaves to the root) we can also parse top-down by constructing the derivation tree from the root to the leaves. A proper subclass of the LR grammars can be parsed deterministically top-down. These are the class of LL grammars, first studied by [Lewis and Stearns (1968)]. LL parsers are also efficient and have good error-detecting capabilities. In addition, an LL parser requires less initial optimization to be of practical size. However, the most serious disadvantage of LL techniques is that LL grammars tend to be unnatural and awkward to construct. Moreover, there are LR languages which do not possess any LL grammar.

These considerations, together with practical experience with an automatic parser generating system based on the principles expounded in this paper, lead us to believe that LR parsing is an important, practical tool for compiler design.

REFERENCES

AHO, A. V., DENNING, P. J., AND ULLMAN, J. D. "Weak and mixed strategy precedence parsing." *J. ACM* **19**, 2 (1972), 225–243.

AHO, A. V., JOHNSON, S. C., AND ULLMAN, J. D. "Deterministic parsing of ambiguous grammars." *Conference Record of ACM Symposium on Principles of Programming Languages* (Oct. 1973), 1–21.

AHO, A. V., AND PETERSON, T. G. "A minimum distance error-correcting parser for context-free languages." *SIAM J. Computing* **1**, 4 (1972) 305–312.

AHO, A. V., AND ULLMAN, J. D. *The Theory of Parsing, Translation and Compiling*, Vol. 1, *Parsing*. Prentice-Hall, Englewood Cliffs, N.J., 1972a.

AHO, A. V., AND ULLMAN, J. D. "Optimization of LR(k) parsers." *J. Computer and System Sciences* **6**, 6 (1972b), 573–602.

AHO, A. V., AND ULLMAN, J. D. *The Theory of Parsing, Translation, and Compiling*, Vol. 2, *Compiling*. Prentice-Hall, Englewood Cliffs, N.J., 1973a.

AHO, A. V., AND ULLMAN, J. D. "A technique for speeding up LR(k) parsers." *SIAM J. Computing* **2**, 2 (1973b), 106–127.

ANDERSON, T. *Syntactic analysis of LR*(k) *languages*. PhD Thesis, Univ. Newcastle-upon-Tyne, Northumberland, England (1972).

ANDERSON, T., EVE, J., AND HORNING, J. J. "Efficient LR(1) parsers." *Acta Informatica* **2** (1973), 12–39.

DEMERS, A. "Elimination of single productions and merging nonterminal symbols of LR(1) grammars." Technical Report **TR-127**, Computer Science Laboratory, Dept. of Electrical Engineering, Princeton Univ., Princeton, N.J., July 1973.

DEREMER, F. L. "Practical translators for LR(k) languages." Project MAC Report MAC TR-65, MIT, Cambridge, Mass., 1969.

DEREMER, F. L. "Simple LR(k) grammars." *Comm. ACM* **14**, 7 (1971), 453–460.

EARLEY, J. "An efficient context-free parsing algorithm." *Comm. ACM* **13**, 2 (1970), 94–102.

FELDMAN, J. A., AND GRIES, D. "Translator writing systems." *Comm. ACM* **11**, 2 (1968), 77–113.

FLOYD, R. W. "Syntactic analysis and operator precedence." *J. ACM* **10**, 3 (1963), 316–333.

GRAHAM, S. L., AND RHODES, S. P. "Practical syntactic error recovery in compilers." *Conference Record of ACM Symposium on Principles of Programming Languages* (Oct. 1973), 52–58.

GRIES, D. *Compiler Construction for Digital Computers*. Wiley, New York, 1971.

ICHBIAH, J. D., AND MORSE, S. P. "A technique for generating almost optimal Floyd-Evans productions for precedence grammars." *Comm. ACM* **13**, 8 (1970), 501–508.

JAMES, L. R. "A syntax directed error recovery method." Technical Report **CSRG-13**, Computer Systems Research Group, Univ. Toronto, Toronto, Canada, 1972.

JOLLIAT, M. L. "On the reduced matrix representation of LR(k) parser tables." PhD. Thesis, Univ. Toronto, Toronto, Canada (1973).

KNUTH, D. E. "On the translation of languages from left to right." *Information and Control* **8**, 6 (1965), 607–639.

KNUTH, D. E. "Top down syntax analysis." *Acta Informatica* **1**, 2 (1971), 97–110.

KORENJAK, A. J. "A practical method of constructing LR(k) processors." *Comm. ACM* **12**, 11 (1969), 613–623.

LALONDE, W. R., LEE, E. S., AND HORNING, J. J. "An LALR(k) parser generator." *Proc. IFIP Congress 71 TA-3*, North-Holland Publishing Co., Amsterdam, the Netherlands (1971), pp. 153–157.

LEINIUS, P. "Error detection and recovery for syntax directed compiler systems." PhD Thesis, Univ. Wisconsin, Madison, Wisc. (1970).

LEWIS, P. M., ROSENKRANTZ, D. J., AND STEARNS, R. E. "Attributed translations." *Proc. Fifth Annual ACM Symposium on Theory of Computing* (1973), 160–171.

LEWIS, P. M., AND STEARNS, R. E. "Syntax directed transduction." *J. ACM* **15**, 3 (1968), 464–488.

MCGRUTHER, T. "An approach to automating syntax error detection, recovery, and correction for LR(k) grammars." Master's Thesis, Naval Postgraduate School, Monterey, Calif., 1972.

MCKEEMAN, W. M., HORNING, J. J., AND WORTMAN, D. B. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, N.J., 1970.

PAGER, D. "A solution to an open problem by Knuth." *Information and Control* **17** (1970), 462–473.

PAGER, D. "On the incremental approach to left-to-right parsing." Technical Report **PE 238**, Information Sciences Program, Univ. Hawaii, Honolulu, Hawaii, 1972a.

PAGER, D. "A fast left-to-right parser for context-free grammars." Technical Report **PE 240**, Information Sciences Program, Univ. Hawaii, Honolulu, Hawaii, 1972b.

PAGER, D. "On eliminating unit productions from LR(k) parsers." Technical Report, Information Sciences Program, Univ. Hawaii, Honolulu, Hawaii, 1974.

PETERSON, T. G. "Syntax error detection, correction and recovery in parsers." PhD Thesis, Stevens Institute of Technology, Hoboken, N.J., 1972.

WIRTH, N. "PL360—a programming language for the 360 computers." *J. ACM* **15**, 1 (1968), 37–74.

WIRTH, N., AND WEBER, H. "EULER—a generalization of ALGOL and its formal definition." *Comm. ACM* **9**, 1 (1966), 13–23, and **9**, 2 (1966), 89–99.