INTRODUCTION

There are certain aspects of language theory that have had, or can have, significant impact on the design and implementation of compilers. These areas are, principally, the subjects of context free grammars and syntax directed translations. It is perhaps to be expected that the deep and interesting theorems of language theory do not usually find application. Rather, it is the definitions of formal constructs and their elementary properties that find use.

In this paper, we will discuss some of the ways in which the language theorist can help the compiler designer. First, we will discuss classes of context free grammars that have fast parsing algorithms and discuss briefly what these algorithms are. Then we shall discuss schemes for specifying the translations which must be performed by the compiler.

CONTEXT FREE GRAMMARS

Let us proceed to the principal formalism of use to compiler writers, the context free grammar.

Definition: A context free grammar (CFG) is a four-tuple \((N, \Sigma, P, S)\), where \(N\) and \(\Sigma\) are finite, disjoint sets of nonterminals and terminals, respectively; \(S\) in \(N\), is the start symbol, and \(P\) is a finite list of productions of the form \(A \rightarrow \alpha\), where \(A\) is in \(N\) and \(\alpha\) in \((N \cup \Sigma)^*\).

Example: A good example grammar, which will be subsequently referred to as \(G_0\), is \((\{E, T, F\}, \{a, (, ), +, *\}, P, E)\), where \(P\) consists of the following productions.

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

\(G_0\) defines arithmetic expressions over operators + and *, with no structured variables, and with \(a\) representing any identifier.

The way in which a CFG serves to define strings of terminal symbols is as follows.

Definition: Let \(G = (N, \Sigma, P, S)\) be a CFG. We define the relation \(\Rightarrow\) on \((N \cup \Sigma)^*\), by: if \(\alpha\) and \(\beta\) are any strings in \((N \cup \Sigma)^*\), and \(A \rightarrow \gamma\) is a production, then \(\alpha A \beta \Rightarrow \alpha \gamma \beta\).

If \(\alpha\) is in \(\Sigma^*\), then \(\alpha A \beta \Rightarrow \alpha \gamma \beta\) and if \(\beta\) is in \(\Sigma^*\), then \(\alpha A \beta \Rightarrow \alpha \gamma \beta\); \(lm\) and \(rm\) stand for leftmost and rightmost, respectively. The transitive closure of any relation \(R\), denoted \(R^*\), is defined by:

1. if \(aRb\), then \(aR^+b\);
2. if \(aR^+b\), and \(bR^+c\), then \(aR^+c\);
3. \(aR^+b\) only if it so follows from (1) and (2).

The reflexive-transitive closure of relation \(R\), denoted \(R^*\), is defined by \(a \Rightarrow^* b\) if and only if \(a = b\) or \(a \Rightarrow^* b\). Thus, we can use \(\Rightarrow^*\), for example, to express the notion that \(\alpha\) can become \(\beta\) by some (possibly null) sequence of replacements of left sides of productions by their right sides.

We will, whenever no ambiguity results, drop the subscript \(G\) from the nine relations \(\Rightarrow\), \(\Rightarrow^*\), \(\Rightarrow_\text{lm}\), \(\Rightarrow_\text{rm}\), and their transitive, and reflexive-transitive closures.

We say \(L(G)\), the language defined by \(G\), is \(\{w \mid \epsilon \Rightarrow^* w \in \Sigma^* \land S \Rightarrow^* w\}\). \(L(G)\) is said to be a context-free language (CFL).

Convention: Unless we state otherwise, the following convention regarding symbols of a context free grammar holds.

1. \(A, B, C, \ldots\) are nonterminals.
2. \(\ldots, X, Y, Z\) are either terminals or nonterminals.
3. \(a, b, c, \ldots\) are terminals.
4. \(u, \ldots, z\) are terminal strings.
(5) \( \alpha, \beta, \gamma, \ldots \) are strings of terminals and/or nonterminals.

(6) \( S \) is the start symbol; except in \( G_n \), where \( E \) is the start symbol.

(7) \( \epsilon \) denotes the empty string.

We may, using this convention, specify a CFG merely by listing its productions. Moreover, we use the shorthand \( A \rightarrow \alpha_1 \mid \cdots \mid \alpha_n \) to stand for the productions \( A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots, A \rightarrow \alpha_n \).

**Definition:** Let \( G = (N, \Sigma, P, S) \) be a CFG. If \( \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n \), then we say there is a derivation of \( \alpha_n \) from \( \alpha_1 \). If \( \alpha_i \Rightarrow \alpha_{i+1} \), \( 1 \leq i < n \), we call this a leftmost derivation, and if \( \alpha_i \Rightarrow \alpha_{i+1} \), \( 1 \leq i < n \), it is a rightmost derivation. Most often, we are interested in the case where \( \alpha_1 = S \). If \( S \Rightarrow a \), then \( a \) is a sentential form. If \( S \Rightarrow a \), it is a left sentential form, and if \( S \Rightarrow a \), it is a right sentential form.

**Example:** Let us consider \( G_0 \), and the string \( a^*a + a \) in \( L(G_0) \). It has the following derivations, among others.

1. \( E \rightarrow E + T \Rightarrow T + T \Rightarrow T + F \Rightarrow T + \epsilon + T + F \Rightarrow T + F + a \Rightarrow F + a + a + a \)
2. \( E \rightarrow E + T \Rightarrow T + T \Rightarrow T + F \rightarrow T + F + T \Rightarrow T + a \rightarrow T + a + T \Rightarrow T + a + T \rightarrow T + a \rightarrow T + a + a \)
3. \( E \rightarrow E + T \rightarrow E + F \rightarrow E + a \rightarrow T + a \rightarrow T + a \rightarrow T + a + a \rightarrow T + a + a + a \)

Derivation (1) is neither rightmost nor leftmost. (2) is leftmost and (3) is rightmost. \( F + a + F \) is a sentential form of \( G_0 \), but happens not to be a left or right sentential form. \( a^*a + a \) is both a left and right sentential form, and is in \( L(G_0) \).

Given a derivation of \( w \) from \( S \) we can construct a parse tree for \( w \) as follows. Let \( S = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n = w \).

1. Begin with a single node labeled \( S \). This node is a trivial tree, and is said to be the tree for \( \alpha_1 \). In general the tree for \( \alpha_i \) will have a leaf corresponding to each symbol of \( \alpha_i \). Initially, the lone symbol \( S \) of \( \alpha_1 \) corresponds to the lone node.
2. Suppose we have constructed the tree for \( \alpha_i \), \( i < n \). Let \( \alpha_{i+1} \) be constructed by replacing some instance of \( A \) in \( \alpha_i \) by \( \beta \). To this instance of \( A \) there corresponds a leaf. Make descendants of that leaf for each symbol of \( \beta \), ordered from the left. If \( \beta \) is the empty string, then we create a single descendant, labeled \( \epsilon \).

The construction of a tree defined above is called top down. We can also construct the same tree bottom up, as follows.

1. Begin with an isolated node corresponding to each symbol of \( \alpha_n \). As this algorithm proceeds, we will have a collection of trees corresponding to each step of the derivation. At the end, there will be one tree for the first sentential form \( \alpha_1 \) (i.e., \( S \)).
2. Suppose we have a collection of trees for \( \alpha_i \), \( 1 \leq i \leq n \), where to each symbol of \( \alpha_i \) there corresponds a root of one of the trees. If \( \alpha_i \) is constructed from \( \alpha_{i-1} \) by replacing \( A \) by \( \beta \), create a new node, labeled \( A \), whose direct descendants are the roots of the trees for the symbols of \( \beta \). The order of symbols in \( \beta \) reflects the order of the descendants. If, however, \( \beta = \epsilon \), create one descendant for the node labeled \( A \), and label the descendant \( \epsilon \).

**Example:** The unique parse tree for \( a^*a + a \) in \( G_0 \) is shown below. Note that the leaves of the tree read \( a^*a + a \), from the left.

\[
\begin{align*}
E & \rightarrow E + T \\
& \rightarrow T + T \\
& \rightarrow T + F \\
& \rightarrow F + a \\
& \rightarrow a + a \\
& \rightarrow a + a + a
\end{align*}
\]

It is easy to show that the top down and bottom up methods of tree construction yield the same tree when applied to one derivation.

An important property of a CFG is that it be unambiguous, that is, no word in its language has more than one parse tree. Because of the way syntax directed translation algorithms work on parse trees, an ambiguous programming language is very likely to provide surprising machine code when (at least some of) its programs are compiled. It is easy to show that the con-
dition of unambiguity is tantamount to saying that each word in the language has a unique leftmost derivation and a unique rightmost derivation.

EARLEY'S ALGORITHM

There is one outstanding method of recognizing the words of an arbitrary CFL and building parse trees, although many others have been proposed. This is the method of Earley. Descriptions of other general methods can be found in References 2 and 3. Earley's method works as follows.

1. Let $G = (N, S, P, S)$ and let $a_1, \ldots, a_n$ be the word we wish to recognize or parse. We construct lists $I_0, I_1, \ldots, I_n$, of items; an item is of the form $[A \rightarrow \alpha \cdot \beta, i]$, where $A \rightarrow \alpha \beta$ is a production and $i$ an integer.

2. Construct $I_0$ as follows.
   (i) Add $[S \rightarrow a_0]$ to $I_0$, for each $S \rightarrow a$ in $P$.
   (ii) If $[A \rightarrow \beta a, 0]$ is on $I_0$, add $[B \rightarrow \beta, 0]$ to $I_0$, if it is not already there, for each $B \rightarrow \beta$ in $P$.

3. Suppose we have constructed $I_{i-1}$. Construct $I_i$ as follows.
   (i) For all $[A \rightarrow \alpha \cdot a_0 \beta, j]$ on $I_{i-1}$, add $[A \rightarrow a_i \cdot \beta, j]$ to $I_i$. Recall that $a_i$ is the $i$th input position.
   (ii) If $[A \rightarrow \alpha, j]$ on $I_i$, add $[B \rightarrow \alpha \cdot \gamma, k]$ to $I_i$, if $[B \rightarrow \beta A \gamma, k]$ is on $I_j$.
   (iii) If $[A \rightarrow \alpha \cdot \beta a_0, j]$ is on $I_i$, add $[B \rightarrow \gamma, i]$ to $I_i$ for all $B \rightarrow \gamma$ in $P$.

We can show that item $[A \rightarrow \alpha \cdot \beta, j]$ is placed on $I_i$ if and only if there is a leftmost derivation

$$S \Rightarrow a_1 \ldots a_{i-1} A \gamma \Rightarrow a_1 \ldots a_{i-1} \gamma.$$ 

As a special case of the above, $a_1 \ldots a_n$ is in $L(G)$ if and only if $[S \rightarrow \alpha, 0]$ is on $I_n$ for some $\alpha$. Thus, it is easy to see how Earley's algorithm can be used to recognize the words of a language. Moreover, once the lists have been constructed, it is possible to build the parse tree of the word. We shall not explore this further here, but the intuitive idea behind the tree construction algorithm is to let the item $[S \rightarrow \alpha, 0]$ on $I_n$ represent the root, then look for those complete (not at the right end) items which lead to its being placed on $I_n$. Make these items correspond to the descendants of the root, and proceed top down, finding "causes" for each item corresponding to a node.

Example: The lists for $G_0$ with input $a \ast a$ are shown below.

<table>
<thead>
<tr>
<th>$I_0$</th>
<th>$I_1$</th>
<th>$I_2$</th>
<th>$I_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E + T, 0$</td>
<td>$F \rightarrow a_0, 0$</td>
<td>$T \rightarrow T \cdot F, 0$</td>
<td>$F \rightarrow a_0, 2$</td>
</tr>
<tr>
<td>$E \rightarrow T, 0$</td>
<td>$T \rightarrow F, 0$</td>
<td>$F \rightarrow (E), 2$</td>
<td>$T \rightarrow T \cdot F, 0$</td>
</tr>
<tr>
<td>$T \rightarrow T \cdot F, 0$</td>
<td>$E \rightarrow T, 0$</td>
<td>$F \rightarrow a_2, 2$</td>
<td>$E \rightarrow T, 0$</td>
</tr>
<tr>
<td>$T \rightarrow F, 0$</td>
<td>$T \rightarrow T \cdot F, 0$</td>
<td>$F \rightarrow a_2, 0$</td>
<td></td>
</tr>
</tbody>
</table>

List $I_3$ is constructed as follows. $[F \rightarrow a_2, 2]$ is added by rule (3i), because $[F \rightarrow a, 2]$ is on $I_2$, $[T \rightarrow T \cdot F, 0]$ is added to $I_3$ by rule (3ii), because $[T \rightarrow T \cdot F, 0]$ is on $I_2$ and $[F \rightarrow a_2, 2]$ is on $I_3$. $[E \rightarrow T, 0]$ and $[T \rightarrow T \cdot F, 0]$ are added because $[E \rightarrow T, 0]$ and $[T \rightarrow T \cdot F, 0]$ are on $I_2$ and $[T \rightarrow T \cdot F, 0]$ is on $I_3$. Since $[E \rightarrow T, 0]$ is on $I_3$, $a_2 \alpha a$ is in $L(G_3)$.

Earley's algorithm can be implemented in $O(n^2)$ steps of a random access computer if the underlying grammar is unambiguous, and in $O(n^2)$ steps for an arbitrary grammar. Moreover, on many grammars of practical interest, Earley's algorithm operates in $O(n)$ steps. It is the fastest known general parsing algorithm.

LL GRAMMARS

We will now begin the study of several subclasses of the context free grammars. None of the three classes we consider is capable of generating all the context free languages. However, confronted with a context free language that is associated with a real programming language, it is highly likely that grammars in the classes to be discussed do exist.

Our first subclass of grammars, called $LL(k)$, for left to right scan, producing a leftmost derivation, with $k$ symbol lookahead, was first examined in Reference 4. The $LL(1)$ case was developed independently in Reference 5. Development of the theory can be found in References 6 and 7. The general idea can be summarized as follows.

Let $G = (N, S, P, S)$ be a CFG, and let $w$ be in $\Sigma^*$. We may attempt to find a leftmost derivation for $w$ by starting with $S$ and trying to proceed to successive left sentential forms. Suppose we have obtained

$$S = a_1 \Rightarrow a_2 \Rightarrow \ldots \Rightarrow a_n,$$

where $a_1 = xA\beta$, and $x$ is a prefix of $w$. If there are several productions with $A$ on the left, we must select the proper one. It is desirable that we be able to do so using only information which we have accumulated so far during the parse (which we represent by saying that we "know what $\beta$ is") and the $k$ symbols of $w$ be-
yond prefix $x$, for some small $k$. If we can always make this decision correctly, we say $G$ is $LL(k)$.

If a grammar is $LL(k)$, we can parse it deterministically in a simple fashion. One pushdown list, which holds the portion of a left sentential form to the right of the prefix of terminals ($A\beta$ in the case of $a_i$ above) is used. (There is also some other information on the list, but we shall not discuss this here. A discussion can be found in in Reference 3.) The input $w$ is scanned left to right, and if $a_i$ has been reached, the input pointer will have scanned over prefix $x$, and is ready to read the first $k$ symbols of $w$. The arrangement is shown below.

![Diagram of pushdown list]

If production $A \rightarrow \gamma$ is chosen to expand $A$, the pushdown list is made to hold $\gamma\beta$. Then, any terminals on top of the pushdown list are compared with the input immediately to the right of the input pointer, and the next left sentential form will be properly represented. The process of expanding the topmost nonterminal on the pushdown list then repeats. Note that while expanding nonterminals, we could construct a parse tree top down if we wished.

It should be noted that the pushdown list may carry information other than grammar symbols, in order to summarize at the top of the list, the important information about what is in the list ($\beta$ in our example). However, this information is not essential, if the grammar is modified properly. We now give the formal definition of an $LL(k)$ grammar.

**Definition:** If $a$ is a string, let $|a|$ denote the length of $a$. Let $a:k$ denote the first $k$ symbols of $a$, or all of $a$ if $|a| < k$. Let $G = (N, \Sigma, P, S)$ be a CFG. We say $G$ is $LL(k)$ if whenever we have the following two derivations

1. $S \Rightarrow^{*} zAa \Rightarrow^{*} z\beta \Rightarrow^{*} xyz$
2. $S \Rightarrow^{*} zAa \Rightarrow^{*} z\gamma a \Rightarrow^{*} zz$

and $y:k = z:k$. Then we may conclude that $\beta = \gamma$.

Stated less formally, suppose we are parsing $w$, and have so far reached left sentential form $xAa$. We have, as indicated in the informal parsing procedure which introduced the section, not scanned $w$ more than $k$ symbols beyond $x$. Thus, either $xy$ or $xz$ could be $w$, as far as we know. Suppose $A \rightarrow \beta$ and $A \rightarrow \gamma$ are two productions. Then the $LL(k)$ definition assures us that independent of $w$, but depending on $x$, $a$ and the first symbols of $w$ beyond $x$ (which are $y:k$, or equivalently, $z:k$), we may uniquely select the proper production with which to replace $A$.

**Example:** Consider the grammar

$$S \rightarrow aBB \mid b$$
$$B \rightarrow bSS \mid a$$

This grammar is $LL(1)$. For suppose we have two derivations (1) and (2), as in the $LL(k)$ definition. If $y:1 = z:1 = a$, and $A$ is $S$, then clearly $aBB$ is both $\beta$ and $\gamma$. If $A$ is $B$, the $\beta = \gamma = a$. If $y:1 = z:1 = b$, then $\beta = \gamma = b$ if $A$ is $S$, and $\beta = \gamma = bSS$ if $A$ is $B$.

**LR GRAMMARS**

Just as the $LL(k)$ grammars are a natural class of grammars for which parse trees can be built deterministically, top down, via leftmost derivations, there is a natural class of grammars, called $LR(k)$, for left to right scan, producing rightmost derivations with $k$ symbol lookahead, for which parse trees can be constructed deterministically bottom up via rightmost derivations. This class of grammars was defined in Reference 8, and the theory, including optimization of LR parsers has been discussed in References 9-13.

**Definition:** Let

$$S \Rightarrow^{*} aAx \Rightarrow a\beta x.$$

Then $\beta$, in the position shown, is said to be a **handle** of right sentential form $a\beta x$. The handle of a right sentential form need not be uniquely defined, but will be, if the grammar is unambiguous.

Given grammar $G = (N, \Sigma, P, S)$ and $w$ in $\Sigma^*$, we could attempt to find a rightmost derivation of $w$, starting with $w$ and working backwards toward $S$. Suppose we have found

$$S \Rightarrow^{*} a_1 a_2 \Rightarrow^{*} a_3 a_4 \Rightarrow^{*} \ldots \Rightarrow a_m = w.$$  

In order to find the right sentential form previous to $a_i$, we must find its handle and replace the handle by the left side of the production used to create the handle. If we can do so, we can recognize and parse using a pushdown list as shown.
Suppose \( \alpha_i \) is \( \beta \alpha \). Then \( \beta \alpha \) will appear on the pushdown list, with \( \alpha \) at the top. (As with LL grammars, there will be some extra information on the list helping us to make parsing decisions. See References 3 and 8.) \( x \) will be a suffix of the input, unscanned to this point. (In the case \( i=m \), i.e., \( \alpha_i = w \), the pushdown list would be empty, with the input pointer at the left end.) The handle of \( \beta \alpha \) cannot appear wholly within \( \beta \), by the definition of a rightmost derivation. Thus, either it is a suffix of \( \beta \alpha \), or its right end is somewhere within \( x \).

Using the extra information on the pushdown list and the \( \alpha \) symbols to the right of the input pointer, the LR\( (k) \) parser concludes either

1. that the handle is not yet on the pushdown list, and an input symbol must be shifted from the input to the pushdown list (i.e., the input pointer moves right, and the symbol which it leaves is placed on top of the pushdown list, or
2. that the handle is now on top of the pushdown list. In this case, the extra information tells us what production was used at the step which created \( \alpha_i \), and we can reduce \( \alpha_i \) to \( \alpha_{i-1} \) by replacing the handle by the appropriate nonterminal.

Whether (1) or (2) applies, the LR\( (k) \) parser repeats the decision whether to shift or reduce. Since we cannot shift indefinitely, we must eventually reduce. If the grammar is LR\( (k) \), the correct decision will be made at each step, and we will trace out a rightmost derivation in reverse order. Note that we can build the parse tree bottom up as we do the reductions. We will now give the formal LR\( (k) \) definition.

**Definition:** Let \( G = (N, \Sigma, P, S) \) be a CFG, and let \( G' = (\mathcal{N} \cup \{S\}, \Sigma, P', S') \) be the augmented grammar for \( G \), defined as follows.

1. \( S' \) is a new nonterminal.
2. \( P' \) is \( P \) preceded by a production numbered 0, namely \( S' \rightarrow S \).

The augmented grammar allows us to treat "reduction" by production \( S' \rightarrow S \) as the successful completion of the algorithm.

Suppose we have two derivations in \( G' \):

1. \( S'_m = \alpha \alpha x \rightarrow \alpha \beta \alpha \)
2. \( S'_m = \gamma \beta y \rightarrow \gamma \delta \gamma \)

and we can write \( \gamma \delta \gamma \) as \( \alpha \beta \gamma ' \), where \( \gamma ': k = x:k \). If we can always conclude that \( \gamma \gamma = \alpha \gamma ' \) (i.e., \( y = y \), \( A = B \) and \( \gamma = \alpha \)), then we say \( G \) is LR\( (k) \).

Stated informally, suppose we are constructing a rightmost derivation of \( w \), in reverse, in such a way that we never observe symbols of \( w \) that are more than \( k \) symbols beyond the handle. Suppose that we have done some reductions, and reduced some prefix of \( w \) to \( \alpha \beta \gamma \). We still do not know what the tail end of \( w \) is; it could be \( x \) or \( y ' \), among other things. However, we do know that the next \( k \) symbols of our current right sentential form are \( x:k \) (equivalently, \( y:k \)). Then if \( G \) is LR\( (k) \), we may determine that \( \beta \) is the handle and must be reduced to \( A \). We can make this determination independent of whether the input actually ends with \( x \) or \( y ' \).

**Example:** Consider the grammar

\[
S \rightarrow aS | a
\]

The grammar is not LR\( (0) \). For in the augmented grammar, we have derivations

\[
S_m = S \rightarrow a
\]

\[S_m = aS \rightarrow aa\]

We may compare these derivations with those of the LR\( (k) \) definition, letting \( \alpha = e \), \( A = B = S \), \( \beta = \gamma = \delta = a \), \( x = y = e \) and \( \gamma ' = a \). Then \( x:0 = y':0 = e \), but \( \gamma \gamma ' \), which is \( a \gamma ' \), is not equal to \( a \gamma ' \), which is \( S \).

However, this grammar is LR\( (1) \). The only possible right sentential forms \( a \beta \gamma \) are:

1. \( \alpha = x = e \), \( \beta = S \), in which the last step replaced \( S' \) by \( S \),
2. \( \alpha = a_i ' \) for some \( i \geq 0 \), \( x = e \), and \( \beta = aS \). (The last step replaced \( S \) by \( aS \),) or
3. \( \alpha = a_i ' \), \( \beta = a \) and \( x = e \), where \( S \) was replaced by \( a \) at the last step.

The above three possibilities hold also if we refer to \( \gamma \), \( \delta \) and \( y ' \) instead of \( \alpha \), \( \beta \) and \( x \). If we have a violation...
of the LR(1) condition, since \( x:1 = e \), we must have \( \gamma_0 = a \beta \) and \( y':1 = e \), but \( \gamma_0 B \neq a A y' \). If \( y':1 = e \), then \( y' = e \). Moreover, as we argued in (i)-(iii) above, we must have \( y = e \) in the right sentential form \( \gamma y \). Thus, \( \gamma_0 = a \beta \).

If \( a \beta = S \), then \( \gamma_0 = S \), and \( y = e \) and \( \delta = S \) follows. Then \( B = A = S' \), and we have \( \gamma_0 B \gamma = a A y' \), which we assumed not to be the case.

If \( \alpha = a' \) and \( \beta = aS \), it similarly follows that \( \gamma = a' \) and \( \delta = aS; A = B = S \). Again \( \gamma_0 B \gamma = a A y' \).

Finally, if \( \alpha = a' \) and \( \beta = a \), we may conclude that \( \gamma = a' \) and \( \delta = a \). Again, \( A = B = S \), and \( \gamma_0 B \gamma = a A y' \).

We conclude that the grammar is LR(1).

### PRECEDENCE GRAMMARS

Many practical classes of grammars admit of shift-reduce type parsing algorithms, as outlined in the previous section, but without the need for extra information on the pushdown list. We shall here discuss several classes that go under the heading of precedence grammars. The initial idea of operator precedence parsing is from Reference 14, while the most common current notion of a precedence grammar first appeared in Reference 15. The theory of precedence parsing has been developed in References 16 through 21. We now give the basic precedence definitions.

**Definition:** Let \( G = (N, \Sigma, P, S) \) be a CFG with no production \( A \rightarrow \epsilon \), no derivation \( A \Rightarrow A \), and no symbols not appearing in the derivation of some word in \( L(G) \). We assume \( \$ \) is not in \( N \cup \Sigma \). We define three precedence relations, \(<\cdot \), \(=\) and \(\cdot>\) on \( N \cup \Sigma \) as follows.

1. \( X \preceq Y \) if and only if there is a production of the form \( A \rightarrow a XY \beta \).
2. \( X <\cdot Y \) if and only if there is a production \( A \rightarrow a XB \beta \) and \( B \Rightarrow Y \gamma \).
3. \( X \cdot> Y \) if and only if \( Y \) is a terminal, there is a production \( A \rightarrow a B Z \beta, B \Rightarrow \gamma X \) and \( Z \Rightarrow \gamma \beta \).

We also have \( S\preceq X \) if \( S \Rightarrow X \alpha \) and \( X \cdot> \$ \) if \( S \Rightarrow \beta \).

A grammar is uniquely invertible if it has no two productions of the form \( A \rightarrow \alpha \beta \) and \( B \rightarrow \alpha \).

We say \( G \) is a precedence grammar if it satisfies the constraints mentioned above (no production \( A \rightarrow \epsilon \), etc.), and the relations \(<\cdot \), \(=\) and \(>\cdot \) are disjoint from one another.

If, in addition, \( G \) is uniquely invertible, we say \( G \) is a simple precedence grammar.

**Example:** Let us consider the grammar with productions \( S \rightarrow AA, A \rightarrow aA | \$ \). It is not a precedence grammar, because we have \( A \cdot> a \) and \( A <\cdot a \). That is, since \( AA \) is a right side, \( A \stackrel{*}{\Rightarrow} aA \) and \( A \stackrel{*}{\Rightarrow} aA \), we have \( A \cdot> a \).

Since \( AA \) is a right side and \( A \stackrel{*}{\Rightarrow} aA \) again, we conclude \( A <\cdot a \). However, the modified grammar

\[
S \rightarrow BA \\
B \rightarrow A \\
A \rightarrow aA | \$ 
\]

is a simple precedence grammar. Its precedence relations are shown below.

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>B</th>
<th>A</th>
<th>a</th>
<th>b</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td><img src="image1" alt="Symbol" /></td>
<td><img src="image2" alt="Symbol" /></td>
<td><img src="image3" alt="Symbol" /></td>
<td><img src="image4" alt="Symbol" /></td>
<td><img src="image5" alt="Symbol" /></td>
<td><img src="image6" alt="Symbol" /></td>
</tr>
<tr>
<td>A</td>
<td><img src="image7" alt="Symbol" /></td>
<td><img src="image8" alt="Symbol" /></td>
<td><img src="image9" alt="Symbol" /></td>
<td><img src="image10" alt="Symbol" /></td>
<td><img src="image11" alt="Symbol" /></td>
<td><img src="image12" alt="Symbol" /></td>
</tr>
<tr>
<td>a</td>
<td><img src="image13" alt="Symbol" /></td>
<td><img src="image14" alt="Symbol" /></td>
<td><img src="image15" alt="Symbol" /></td>
<td><img src="image16" alt="Symbol" /></td>
<td><img src="image17" alt="Symbol" /></td>
<td><img src="image18" alt="Symbol" /></td>
</tr>
<tr>
<td>b</td>
<td><img src="image19" alt="Symbol" /></td>
<td><img src="image20" alt="Symbol" /></td>
<td><img src="image21" alt="Symbol" /></td>
<td><img src="image22" alt="Symbol" /></td>
<td><img src="image23" alt="Symbol" /></td>
<td><img src="image24" alt="Symbol" /></td>
</tr>
<tr>
<td>$</td>
<td><img src="image25" alt="Symbol" /></td>
<td><img src="image26" alt="Symbol" /></td>
<td><img src="image27" alt="Symbol" /></td>
<td><img src="image28" alt="Symbol" /></td>
<td><img src="image29" alt="Symbol" /></td>
<td><img src="image30" alt="Symbol" /></td>
</tr>
</tbody>
</table>

We can parse according to a simple precedence grammar as follows. An input, scanned left to right by a pointer is used, as is a pushdown list. Right sentential forms are represented as in the description of the LR(k) parser. Initially, only $ appears on the pushdown list, and the leftmost symbol of the input is being scanned. The input has $ appended at the right. At each step of the algorithm, the following is done. Let \( X_1 \ldots X_m \) appear on the pushdown list (top right) and \( a_1 \ldots a_r \) be the remaining input.

1. If \( X_m <\cdot a_1 \) or \( X_m \equiv a_1 \), \( a_1 \) is shifted onto the pushdown list, and the process repeats.
2. If \( X_m \cdot> a_1 \), find \( k \) such that \( X_{k-1} <\cdot X_k \equiv X_k+1 \equiv \ldots \equiv X_m \). Let \( A \rightarrow X_k \ldots X_m \) be a production. \( (A \) must be unique if \( G \) is a simple precedence grammar.) Then replace \( X_k \ldots X_m \) by \( A \) on top of the pushdown list.
3. If neither (1) nor (2) apply, accept the input if \( X_1 \ldots X_m = $S \) and \( a_1 \ldots a_r = $ \). Declare an error otherwise.

We conclude that the grammar is LR(1).

---

From the collection of the Computer History Museum (www.computerhistory.org)
SYNTAX DIRECTED TRANSLATIONS

We now turn to formalisms for specifying the code generation phase of a compiler. The general strategy is to work from a parse tree, building some “translations” at each node. One translation at the root of the tree is designated the output of the translation system. The most elementary system, called a (formal) syntax directed translation scheme was first expounded in Reference 22 and formalized in Reference 4. Some theoretical developments and generalizations appear in References 22 through 30.

Definition: A (formal) syntax directed translation scheme (SDTS) is a CFG \( G = (N, \Sigma, P, S) \), together with a translation element for each production. The translation element associated with production \( A \to \alpha \) is a string \( \beta \) in \( (NU\Delta)^* \), where \( \Delta \) is an alphabet of output symbols, disjoint from \( N \). \( \beta \) must be such that there is a one to one mapping from its nonterminals to the nonterminals of \( \alpha \), where each nonterminal is mapped to an identical symbol. If the mapping preserves the (left to right) order of appearance of the nonterminals in \( \alpha \) and \( \beta \), then the SDTS is said to be simple.

We construct a translation from a parse tree in the CFG as follows. An order for the interior nodes is chosen so that each node follows all its descendants in the order. Each node is considered in its turn. If a particular node \( n \) has label \( A \) and its direct descendants have labels \( X_1, \ldots, X_n \), from the left, then

\[ A \to X_1 \ldots X_n \]

is a production. Let \( \beta = Y_1 \ldots Y_m \) be the translation element for that production. The translation at \( n \) is formed from \( Y_1 \ldots Y_m \) by substituting for each \( Y_i \) in \( N \), the translation at the \( j \)th direct descendant of \( n \) if \( Y_i \) is associated with \( X_j \).

The translation defined by the SDTS is the set of pairs \((w, x)\) such that \( w \) is the yield of some parse tree \( T \) and the translation defined at the root of \( T \) is \( x \).

Example: We can build a translation on \( G_0 \) to translate infix arithmetic expressions to postfix (operator follows both operands) expressions.

<table>
<thead>
<tr>
<th>Production</th>
<th>Translation Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \to E + T )</td>
<td>( ET+ )</td>
</tr>
<tr>
<td>( E \to T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( T \to T * F )</td>
<td>( TF* )</td>
</tr>
<tr>
<td>( T \to F )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F \to (E) )</td>
<td>( E )</td>
</tr>
<tr>
<td>( F \to a )</td>
<td>( a )</td>
</tr>
</tbody>
</table>

Since no production of \( G_0 \) has more than one occurrence of the same nonterminal on the right side of any production, the mapping between nonterminals of the translation elements and their productions is obvious. Note that the SDTS is simple.

The string \((a+a)*a\) has the parse tree shown below. Nodes have been numbered for reference. The following order of the interior nodes is acceptable. \( n_0, n_1, n_6, n_9, n_5, n_3, n_4, n_11, n_7, n_2, n_1 \).

The translation at \( n_0 \) is just \( a \), since production \( F \to a \) was applied there, and the associated translation element is \( a \). To compute the translation at \( n_1 \), we substitute the translation at \( n_6 \) for \( F \) in the translation element associated with \( T \to F \). Thus, the translation at \( n_1 \) is \( a \), as is the translation at \( n_4, n_9 \), and \( n_2 \). The translation at \( n_3 \) is computed by substituting the translation at \( n_4 \) for \( E \) and that at \( n_6 \) for \( T \) in the translation element \( ET+ \). Thus, the translation at \( n_3 \) is \( aa+ \).

Proceeding similarly, the translation at \( n_4 \) is \( aa+a* \).

REFERENCES

1 J EARLEY
An efficient context free parsing algorithm
CACM 13 pp 94-102 February 1970
2 T E CHEATHAM
The theory and construction of compilers
3 A V AHO J D ULLMAN
The theory of parsing, translation and compiling
4 P M LEWIS R E STEARNS
Syntax-directed transduction
JACM 15 pp 464-488 July 1968
5 D E KNUTH
Top down syntax analysis
Inl Summer School on Computer Programming
Copenhagen Denmark 1967
6 D J ROSENKRANTZ R E STEARNS
Properties of deterministic top-down grammars
Inf and Control 17 pp 226-256 1970
7 D WOOD
The theory of factored languages
8 D E KNUTH
On the translation of languages from left to right
Inf Control 8 pp 607-639 October 1965
9 A J KORENJAK
A practical method for constructing LR(k) processors
CACM 12 pp 613-623 1969
10 F L De REMER
Practical translators for LR(k) languages
11 F L De REMER
Simple LR(k) grammars
CACM 14 pp 453-459 July 1971
12 A V AHO J D ULLMAN
The care and feeding of LR(K) grammars
Proc 3rd Annual ACM Symposium on Theory of Computing pp 159-170 1971
13 A V AHO J D ULLMAN
A technique for speeding up LR(k) parsers
Unpublished memorandum Bell Telephone Laboratories
Murray Hill NJ 1972
14 R W FLOYD
Syntactic analysis and operator precedence
JACM 10 pp 318-333 July 1963
15 N WIRTH H WEBER
EULER—A generalization of ALGOL, and its formal definition, Part I
CACM 9 pp 13-25 January 1966
16 W M McKEEMAN
An approach to computer language design
Tech Report CS48 Computer Science Dept Stanford University Stanford California
17 M J FISCHER
Some properties of precedence languages
Proc ACM Symp on Theory of Computing pp 181-190 May 1969
18 S L GRAHAM
Extended precedence languages, bounded right context languages and deterministic languages
Proc IEEE 11th Annual Symp on Switching and Automata Theory pp 175-180 October 1970
19 J N GRAY
Precedence parsers for programming languages
PhD Thesis Dept of Computer Science University of California Berkeley California September 1969
20 A V AHO P J DENNING J D ULLMAN
Weak and mixed strategy precedence parsing
JACM to appear April 1972
21 A COLMERAUER
Total precedence relations
JACM 17 pp 14-30 January 1970
22 E T IRONS
A syntax directed compiler for ALGOL 60
CACM 4 pp 51-55 January 1961
23 A V AHO J D ULLMAN
Properties of syntax directed translations
J Computer and System Sciences 3 pp 319-334 August 1969
24 A V AHO J D ULLMAN
Syntax directed translations and the pushdown assembler
J Computer and System Sciences 3 pp 37-56 February 1969
25 L PETRONE
Syntax directed mappings of context free languages
Conference Record of 9th Annual Symposium on Switching and Automata Theory pp 160-175 October 1968
26 A V AHO J D ULLMAN
Translations on a context-free grammar
27 W C ROUNDS
Mappings and grammars on trees
28 J W THATCHER
There's a lot more to finite automata theory than you would have thought
29 D E KNUTH
Semantics of context free languages
Math Systems Theory 2 pp 127-146 June 1968
30 T WILCOX
Generating machine code for high level programming languages
Cornell University Dept of Computer Science TR-71-103 September 1971