EFFICIENT RECURSIVE PARSING

Christoph M. Hoffmann
Computer Science Department
Purdue University
West Lafayette, Indiana 47907

CSD-TR 215

December 1976
Abstract

Algorithms are developed which construct from a given LL(1) grammar a recursive descent parser with as much recursion resolved by iteration as is possible without introducing auxiliary memory. Unlike other proposed methods in the literature designed to arrive at parsers of this kind, the algorithms do not require extensions of the notational formalism nor alter the grammar in any way.

The algorithms constructing the parsers operate in \(O(k \cdot s)\) steps, where \(s\) is the size of the grammar, i.e. the sum of the lengths of all productions, and \(k\) is a grammar-dependent constant. A speedup of the algorithm is possible which improves the bound to \(O(s)\) for all LL(1) grammars, and constructs smaller parsers with some auxiliary memory in form of parameters to some of the routines.

Keywords and phrases: compilers, top down parsers, recursive descent, complexity.
1. Introduction

Among the various proposed parsing techniques, the method of recursive descent has been particularly attractive from a practical point of view. A parser using this method is, in our experience, very readable and hence readily modified to allow flexible extensions handling semantic processing [12], as well as giving good error diagnosis and recover gracefully [6, 14]. Because of its advantages, the method has found acceptance both commercially, e.g. [3], as well as academically, [13], and serves as a basis for research into compiler compilers [10, 11].

A recursive descent parser is usually obtained from an LL(1) grammar $G^+$ by a simple transcription of the productions of $G$, using the LL(1) lookahead sets. Parsers obtained in this way are frequently unnecessarily recursive, as noted in [11], because of the technique of 'left factoring' (see e.g. [16]) applied to make $G$ LL(1). This inherent problem has been attacked in the past by hand modifications of the LL(1) grammar [9, 11], thereby running risk of having to backtrack, or by ad-hoc modifications of the parser itself.

We present algorithms which construct recursive descent parsers directly from an LL(1) grammar avoiding recursion wherever this can be done without introducing auxiliary variables. The resulting parsers are as good as the ones obtained by hand using the methods of [9, 11], without any need to modify $G$. Assuming a machine environment in which recursion is not unduely expensive, the resulting parsers are competitive in speed with table driven techniques such as LL(1) parsers.

The constructor algorithms are shown to operate in $O(k \cdot a)$ steps,

$\uparrow$ Note, however, the results of [1] which show that recursive descent parsing is more powerful than LL(1) parsing.
where $s$ is the size of the grammar (i.e. the sum of the lengths of all productions), and $k$ is a grammar dependent constant. Ways of reducing $k$ to 1 by a modification of the algorithms are indicated. As derived in [8], the lookahead sets for checking that $G$ is LL(1) can be constructed within the same bound, hence the construction of our parsers is faster than, for example, the SLR(1) algorithms of [5].

2. Preliminaries

We use the standard notation for grammars as, for example, in [2]. A context free grammar $G$ is a quadruple $(N, \Sigma, P, S)$, where $N$ is the set of nonterminal symbols, $\Sigma$ the set of terminal symbols, $P$ the set of productions, where each production is an element of $N^* (N \cup \Sigma)^*$ and written $A \rightarrow \alpha$. $S$ is a distinguished element of $N$, the start symbol.

Symbols in $N$ are denoted by $A, B, C, \ldots$ symbols in $\Sigma$ by $a, b, c, \ldots$, strings in $(N \cup \Sigma)^*$ by $\alpha, \beta, \gamma, \ldots$, and strings in $\Sigma^*$ by $u, w, x, \ldots$. The empty string is denoted by $\epsilon$. The size of $G$ is the sum of the lengths of all productions of $G$.

The relation $\Rightarrow$ ("leftmost derives") is defined on $(N \cup \Sigma)^* \times (N \cup \Sigma)^*$ as follows: $wA\beta \Rightarrow w\alpha\beta$ if $A \rightarrow \alpha$ is a production in $P$ and $w$ is in $\Sigma^*$. The relations $\Rightarrow^*$ and $\Rightarrow^*$ are the transitive and the reflexive transitive closure of $\Rightarrow$, respectively.
A string \( w \) in \( \Sigma^* \) is in \( L_A(G) \), the language derived by \( A \) in \( G \), iff \( A \Rightarrow^* w \), and the language derived by \( G \) is the language derived by \( S \) in \( G \) and denoted \( L(G) \).

Given a context-free grammar \( G = (N, \Sigma, P, S) \), we define for each production \( A \rightarrow \alpha \) in \( P \) a lookahead set \( H(A \rightarrow \alpha) \) as follows:

\[
H(A \rightarrow \alpha) = \{ u \text{ in } \Sigma^* \mid S \Rightarrow^* uA \gamma ; A \beta \Rightarrow^* \alpha \Rightarrow^* ax; \}
\]

\( u, ax \text{ in } \Sigma^* \} \).

**Definition** A context-free grammar \( G \) is LL(1) if, for every \( A \in N, A \rightarrow \alpha \text{ and } A \rightarrow \beta \in P \), \( H(A \rightarrow \alpha) \cap H(A \rightarrow \beta) = \emptyset \).

The definition is equivalent to other definitions in the literature, e.g. \([2, 17]\).

Given an LL(1) grammar \( G \), there is a recursive descent parser for \( L(G) \) obtained, loosely speaking, as follows: For every non-terminal \( A \) in \( N \) create a recursive procedure \( A \). The body of \( A \) contains, for every production \( A \rightarrow \alpha \) in \( P \), a statement of the form

\[
\text{if next input symbol is in } H(A \rightarrow \alpha) \text{ then begin} \langle \alpha \rangle; \text{ goto exit end;}
\]

where \( \langle \alpha \rangle \) is a series of statements calling procedures for non-terminals occurring in \( \alpha \) and scanning the input for terminals in \( \alpha \). An example may clarify.

**Example** Let \( N = \{ S, E, E_1, T, T_1, F \} \),
\[
\Sigma = \{ +, *, (, ), i, $ \},
\]
and let \( G_1 \) be the grammar \( (N, \Sigma, P, S) \), where \( P \) consists of the productions
\[
S \rightarrow E \, \$ \\
E \rightarrow T \, E_1 \\
E_1 \rightarrow \epsilon \\
T \rightarrow F \, T_1 \\
F \rightarrow I \\
E_1 \rightarrow + T \, E_1 \\
T_1 \rightarrow \ast F \, T_1 \\
F \rightarrow ( \, E \, )
\]

\( G_1 \) is LL(1). The recursive descent parser for \( L(G_1) \) obtained as described above is as follows.

```plaintext
procedure S; begin
    $; if nextch = '{' then scan else error; end;
procedure E; begin
    T; E_1; end;
procedure E_1; begin
    if nextch in { '+', '-' } then begin
        scan; T; E_1; goto exit;
    end;
    if nextch in { '*', '/' } then goto exit;
    error;
    exit: end E_1;
    ...
...
procedure E; begin
    if nextch in { '+' } then begin
        scan; goto exit; end;
    if nextch in { '*' } then begin
        scan; E_1;
        if nextch = ')\}' then scan else error;
        goto exit; end;
    error;
    exit: end E;
```
Note that the parser parses the string $i + i + i$ in $L(G)$ by a recursion in $E_1$. Evidently, an iteration is sufficient, and is an example of a situation in which recursion should be removed. It can also be seen, that the procedures $E_1$ and $T_1$ should be eliminated eventually.

**Theorem** The parser obtained as described parses $L(G)$ deterministically if $G$ is LL(1).

The theorem follows quite easily from the properties of LL(1) grammars.

3. Constructor Algorithms

In this section we develop the constructor algorithms for fast recursive descent parsing and establish their correctness. A relation $\mathcal{E}$ defined on $N \times N$ is used to determine which nonterminals $A$ permit iteration in the parse of $L_A(G)$ without auxiliary memory. Note that $\mathcal{E}$, as defined below, is analogous to $\mathcal{E}$ used in $L_{ll}$ to compute the lookahead sets for LL(1) grammars.

**Definition** The relation $\mathcal{E}$ is defined for pairs of nonterminals by $A \mathcal{E} B$ iff there is a production $A \rightarrow \alpha B$ in $G$ ($\alpha$ possibly empty).

Let $\mathcal{E}^+$ and $\mathcal{E}^*$ denote the transitive and the reflexive transitive closure of $\mathcal{E}$, respectively.

**Definition** The nonterminal $A$ in $N$ is quasi-regular, if $A \mathcal{E}^* A$. 
Given a nonterminal A, define a directed graph D(A) as follows: Let R(A) be the subset of all those nonterminals B for which A \notin^* B and B \notin^* A. Then the node set of D(A) is R(A) \cup \{X\}, where X is a symbol not in N \cup \Sigma. Furthermore, for every B in R(A), if B \rightarrow_\alpha C is a production of B and C is in R(A), then there is a directed edge in D(A) from B to C labelled \alpha. For all other productions B \Rightarrow_\beta of B, there is a directed edge from B to X in D(A) labelled \beta.

Example The graph D(E1) of E1 in G is

Definition The graph D(A) of A in N is the deriving graph of A.

Intuitively, the deriving graph of A describes the body of the procedure A which parses L_A(G). Note that D(A) is always connected.

Lemma 3.1 D(A) is acyclic if and only if A is not quasi-regular.

Proof If A is quasi-regular, then A \notin^+ A, hence D(A) must contain a cycle. The rest is immediate.

In order to demonstrate the connection between traversals of deriving graphs and derivation trees in G, we decompose derivation trees into 'cuts'. Given a derivation tree T of w in L(G) with an interior node A, an A-cut of T is a tree defined as follows:

1. The node A is the root of the A-cut, and all immediate descendants of A in T are immediate descendants of the root of the A-cut.
(2) If $B$ is a node in the $A$-cut $\mathcal{A}$ and is the rightmost descendant of a node $C$ in $\mathcal{A}$, then the immediate descendants of the corresponding node $B$ in $\mathcal{T}$ are descendants of $B$ in $\mathcal{A}$ provided that $B \in A$.

**Definition** An $A$-cut $\mathcal{A}$ of a derivation tree $\mathcal{T}$ is maximal if, whenever the root $A$ of $\mathcal{A}$ corresponds to a node $A$ in $\mathcal{T}$ which is the rightmost descendant of another node $B$ in $\mathcal{T}$, then $A$ is not in $R(B)$.

**Example** Consider the derivation tree $\mathcal{T}$ of $i+i+iS$ in $G_i$:

\[
\begin{array}{c}
S \\
\vdots \\
E \\
\vdots \\
T \uparrow \\
F \\
\vdots \\
i \\
\vdots \\
T_1 \\
\vdots \\
E \\
\vdots \\
i \\
\vdots \\
T_1 \\
\vdots \\
i \\
\vdots \\
E
\end{array}
\]

Then $\mathcal{T}_1$ is an $E_1$-cut of $\mathcal{T}$ but not maximal, and $\mathcal{T}_2$ is another $E_1$-cut of $\mathcal{T}$ and is maximal:

\[
\begin{array}{c}
E_1 \\
\vdots \\
T \\
\vdots \\
E_1 \\
\vdots \\
i \\
\vdots \\
T_1 \\
\vdots \\
i \\
\vdots \\
E
\end{array} \quad \begin{array}{c}
E_1 \\
\vdots \\
T \\
\vdots \\
E_1 \\
\vdots \\
i \\
\vdots \\
T_1 \\
\vdots \\
i \\
\vdots \\
E
\end{array}
\]

The following lemma should be obvious.

**Lemma 3.2** Every derivation tree $\mathcal{T}$ in $G$ has a unique decomposition into maximal cuts.

**Proof** By induction on the structure of $\mathcal{T}$. 
A simple traversal of the deriving graph \( D(A) \) is a string in 
\( [N \cup E]^* \{X\} \) defining a path from \( A \) to \( X \) in \( D(A) \), that is, a string of alternating node and edge labels spelling out a path in \( D(A) \), with possible repetition, leading from \( A \) to \( X \). A simple traversal \( t \) describes a tree as follows:

1. The tree \( G_0 \) consisting of the root node \( A \) alone is described by the prefix \( A \) of \( t \).

2. If the tree \( G_{i+1} \) is described by the prefix \( t_iB \) of \( t \), and \( t_{i+1} = t_iBC \) is a prefix of \( t \) where the edge label \( \alpha = B_1B_2\ldots B_k \), then the tree \( G_{i+1} \) described by \( t_{i+1} \) is obtained from \( G_i \) by making \( B_1, B_2, \ldots, B_k, C \) the immediate descendants of the rightmost occurrence of \( B \) in \( G_i \) (as leaf) provided \( C \) is not \( X \); otherwise, only \( B \), \( B_1, \ldots, B_k \) descend from \( B \).

**Lemma 3.3** For any simple traversal \( t \) of \( D(A) \) there exists a derivation tree \( T \) of some sentence \( w \) in \( L(G) \) and an \( A \)-cut \( \mathcal{A} \) of \( T \) such that \( t \) describes \( \mathcal{A} \). Conversely, given an \( A \)-cut \( \mathcal{A} \) in a derivation tree \( T \), there is a simple traversal of \( D(A) \) describing the \( A \)-cut.

**Proof** Let \( \mathcal{A}' \) be the tree described by \( t \). Note, that by construction of \( D(A) \), \( \mathcal{A}' \) can be embedded into a derivation tree \( T \). If \( \mathcal{A}' \) is not an \( A \)-cut of \( T \), then this must be because the rightmost leaf of \( \mathcal{A}' \) has descendants in the \( A \)-cut. But the last edge of the traversal must lead to \( X \) in \( D(A) \), hence the rightmost leaf of
is either a terminal or a nonterminal B such that not B $\not\in^* A$

Hence $A'$ is an A-cut.

The converse part is shown by induction on the size of the A-cut and is straight-forward.

Thus, there is a close connection between parts of derivation trees and traversals of deriving graphs. By a suitable notion of substitution we will be able to relate any derivation tree to certain traversals. Let $t_A = t_1^A t_2^A$ be a simple traversal of $D(A)$ and assume that $\alpha$ is an edge label occurring in $D(A)$ where $\alpha = \alpha_1^B \alpha_2^B$.

Let $t_B$ be a simple traversal of $D(B)$. A substitution of $t_B$ into $t_A$ is the string $t_1^A \alpha_1^B t_B \alpha_2^B t_2^A$ and is a traversal of $D(A)$. Note, that $t_B$ is substituted for only one occurrence of B as part of an edge label in $t_A$. A substitution corresponds, in an obvious manner, to attaching the tree $A'_{B}$ described by $t_B$ with its root to the leaf B in the tree described by $t_A$ in the corresponding position. Substitution is extended, permitting the substitution of traversals into traversals. Formalizing these notions is routine and is left to the reader.

If a traversal $\bar{t}$ does not contain edge labels containing non-terminals, then $\bar{t}$ is a complete traversal. No substitution is possible into complete traversals. Complete traversals correspond to trees the leaves of which are all labelled by terminals.
Theorem 3.4 Assume that $L_A(G)$ is not empty. Then the following is true:

1. Given a complete traversal $t$ of $D(A)$, there exists a derivation tree $T$ of $w$ in $L_A(G)$ such that $w$ is obtained from $t$ by deleting all node labels and brackets.

2. Given a derivation tree $T$ of $w$ in $L_A(G)$, there is a complete traversal $t$ of $D(A)$ such that the string obtained from $t$ by deleting all node labels and brackets is $w$.

Proof (1): Given a complete traversal $t$ of $D(A)$, consider the tree $T$ it describes. It is easily verified that $T$ is a derivation tree for $L_A(G)$. Since $t$ is complete, all occurring edge labels are terminal strings, hence $w$ can be obtained from $t$ as described.

Proof (2): This is proved by induction on the structure of $T$ in its decomposition into maximal cuts (Lemma 3.2).

Basis $T$ is an $A$-cut. By Lemma 3.3 there is a simple traversal $t$ of $D(A)$ describing $T$, and, since $T$ is a derivation tree, $t$ is complete. The rest follows easily.

Induction Step Assume $T$ is a derivation tree of $w$ in $L_A(G)$. Decompose $T$ into the maximal $A$-cut $A$ with its root corresponding to the root of $T$, and the trees $T_{B_1}$, $T_{B_2}$, ..., $T_{B_k}$ attached to $A$ at the nonterminal leaves $B_1$, ..., $B_k$. By Lemma 3.3, the simple traversal $t_A$ describes $A$. Also, each tree $T_{B_i}$ is a derivation tree for some string $u_i$ in $L_{B_i}(G)$ (which therefore cannot be empty), hence, by induction hypothesis, there are complete traversals $t_{B_i}$ of $D(B_i)$ satisfying the theorem.
The string obtained from $t_A$ by deleting all node labels and brackets is the frontier of $t_A$ and is $x_1^B x_2^B \ldots x_k^B x_{k+1}^B$ in $\Sigma^*$. Hence $w = x_1 u_1 \ldots u_k x_{k+1}$. Substituting $t_{B_1}$ for $B_1$ in $t_A$ we obtain the desired complete traversal of $D(A)$ satisfying the theorem.

The graphs $D(A)$ are converted into parsing routines as follows: Corresponding to $D(A)$ is a recursive procedure $A$ which predicts a simple traversal of $D(A)$ edge by edge. Associated with each edge label $B_1^B \ldots B_k^B$, is a sequence of computations $\chi_1$ where $\chi_1$ is a call of procedure $B_i$ if $B_i$ is in $N$, otherwise $\chi_1$ checks if the next input symbol is $B_i$ if $B_i$ is in $\Sigma$.

Assuming that the grammar is LL(1), the simple traversal can be predicted deterministically from a one symbol lookahead [16]. We sketch the algorithm generating procedure $A$ below. In it, code $\alpha$ refers to the code generated for the edge label $\alpha$. Note that if the first symbol of $\alpha$ is terminal, the code corresponding to it can be generated to merely advance the input pointer since the presence of the nonterminal in the input has already been checked by interrogating the lookahead set $H(B \rightarrow \ldots)$. If the parse fails at any point, the routine $\text{error}$ is called which is a standard procedure supplied which issues an appropriate message.
Algorithm 1 (Construct Procedure A)

Input: LL(1) grammar $G$, nonterminal $A$.
Output: Procedure A parsing $L_A(G)$ using iteration where possible.

1. [Initialization]
   Initialize list $\mathcal{L}$ to contain $A$ unmarked.
   Emit: "procedure A; begin"

2. [Check if all done]
   If all elements of $\mathcal{L}$ are marked, goto Step 6; otherwise, take the next unmarked $B$ in $\mathcal{L}$ and mark it, emit: "LB:"

3. [Process B]
   For every production $B \rightarrow \beta$ in $G$ perform Step 5. Thereafter, goto Step 5.

4. [Process $B \rightarrow \beta$]
   4.1 Emit: "if nextch in $H(B \rightarrow \beta)$ then begin".
   4.2 If $\beta = \alpha C$, where $C$ is in $N$ and $C \not\rightarrow A$, then code $\alpha$,
      emit: "goto LC;"
      add $C$ to $\mathcal{L}$ unmarked unless $C$ is already in $\mathcal{L}$;
      otherwise,
      code $\beta$.
   4.3 Emit: "end else".

5. [End of processing B]
   Emit: "error; goto exit;". Goto Step 2.

6. [Procedure A coded]
   Emit: "exit: end A;".
   Stop.
The driving algorithm calls Algorithm 1 to construct procedures, beginning with procedure $S$, for only those nonterminals $A$ for which there is a call on $A$. In addition, some simple optimizations are performed to make the code neater, e.g. the lookahead set is not checked for single production nonterminals, a procedure body is substituted as open subroutine if only one call to it is made in the parser, etc. The resulting code parsing $L(G_1)$ is given in the example below.

Example The parser generated from grammar $G_1$ is as follows:

```plaintext
procedure $S$; begin
  $E$; if nextch = 's' then scan else error;
end $S$;

procedure $E$; begin
  $T$;
  $LEI$: if nextch in {'+', '-'}, '(' then $T$; goto $LEI$
  else if nextch in {*, '='} then $T$
  else error;
exit: end $E$;

procedure $T$; begin
  $F$;
  $LT1$: if nextch in {'+', '*'}, '(' then $F$; goto $LT1$
  else if nextch in {*, '='} then $F$
  else error;
exit: end $F$;
```
procedure \( F \); begin
  if nextch in \{ 'i' \} then scan
  else if nextch in \{ '(', ']' \} then begin
    scan; \( F \);
    if nextch = ')' then scan else error
  end
  else error;
end \( F \);

**Theorem 3.4** When calling \( G \), the parser generated by the algorithms described above parses \( L(G) \) deterministically provided that \( G \) is LL(1).

**Proof** Theorem 3.4 in conjunction with the results of [16].

4. Complexity of the Algorithms

The results of [7, 8] have established that the lookahead sets \( H(A \rightarrow \alpha) \) may be constructed for an LL(1) grammar \( G \) in \( O(p \cdot s) \) steps, where \( p \) is the number of terminals and \( s \) is the size of \( G \). Ultimately these results depend on the sparseness of relations on \( N \times N \) which are similar to the relation \( \tilde{Q} \) used in this paper. This is usually the case for grammars. Also in [8], there is a discussion of how to lower the bound to \( O(s) \) in certain machine environments.
On closer inspection of \( \mathcal{B} \) it can be seen that \( \mathcal{C}^* \) may be computed within the same bound, because it is a subset of \( \mathcal{C} \) used in \( \mathcal{B} \), and therefore at least as sparse. Hence we can assume that both the look-

ahead sets and the relation \( \mathcal{C}^* \) can be computed within this bound. Once they are available, the parser may be constructed within the bounds de-

rived below.

**Proposition 4.1** Given nonterminal \( A \), the sets \( \mathcal{H}(B \rightarrow \alpha) \) for all \( B \) in \( \mathcal{R}(A) \), and the relation \( \mathcal{C}^* \), Algorithm 1 constructs procedure \( A \) in \( O(s^*) \) steps, where \( s_A \) is the sum of the lengths of all productions of the nonterminals in \( \mathcal{R}(A) \).

**Proof** Evident.

Let \( |\mathcal{R}(A)| \) denote the number of elements in \( \mathcal{R}(A) \). We define an equivalence relation \( \mathcal{S} \) on \( N \times N \) by \( A \mathcal{S} B \iff A \mathcal{C}^* B \) and \( B \mathcal{C}^* A \). Note that the \( \mathcal{R}(A) \) are the equivalence classes of \( \mathcal{S} \).

**Theorem 4.2** Given the LL(1) grammar \( G \), the lookahead sets \( \mathcal{H}(A \rightarrow \alpha) \) for \( G \) and the relation \( \mathcal{C}^* \), the parser for \( G \) can be constructed as de-

scribed above in \( O(k \cdot s) \) steps, where \( s \) is the size of \( G \), and \( k \) is the cardinality of the largest equivalence class of \( \mathcal{S} \), i.e.

\[
    k = \max \{|\mathcal{R}(A)|\}.
\]

**Proof** Let \( \mathcal{R}(A) = \{A, A_1, \ldots, A_{m-1}\} \). The procedures \( A, A_1, \ldots, A_{m-1} \) can then be constructed in \( O(m \cdot s^*) \) steps, which may be estimated by \( O(k \cdot s_A) \) steps because of \( k \)'s definition.

Let \( B_1, \ldots, B_r \) be a set of representatives of the equivalence classes of \( \mathcal{S} \), i.e.

\[
\begin{align*}
    &\mathcal{R}(B_i) \cap \mathcal{R}(B_j) = \emptyset \quad \text{if } i \neq j, \ i,j \in \{1, \ldots, r\} \\
    &\bigcup_{j \in \{1, \ldots, r\}} \mathcal{R}(B_j) = N
\end{align*}
\]
Then, clearly, \( s_B + s_B + \ldots + s_B = s \), where \( s \) is the size of \( G \).

Hence all procedures \( A \) for every \( A \) in \( N \) can be constructed in \( O(k \cdot s) \) steps.

The constant \( k \) could be fairly large for certain grammars, as seen by considering the grammars \( G_n, n > 1 \), which are as follows.

The nonterminals of \( G_n \) are \( \{S, A_1, \ldots, A_n\} \), the terminals are \( \{a_1, \ldots, a_n, b_1, \ldots, b_n, \$\} \) and the productions are

\[
\begin{align*}
S & \rightarrow A_1 S \\
A_i & \rightarrow a_i A_{i+1} a_i A_{i+1} | b_i \quad i < n \\
A_n & \rightarrow a_n A_1 a_n A_1 | b_n
\end{align*}
\]

Clearly, \( |R(A_j)| = n-1 \).

This points out a potential weakness in our algorithms. In case that the sets \( R(A) \) are large, the code generated becomes bulky. Recall that \( A \) is in \( R(B) \) iff \( B \) is in \( R(A) \). Consequently, the procedures for \( A \) and \( B \) can be made identical except for different entry points. If the set \( R(A) \) contains \( m \) elements, there will be \( m \) such procedures. Clearly a simple scheme can be devised which replaces the \( m \) procedures associated with the nonterminals in \( R(A) \) by a single one. This new procedure, called with a parameter indicating the nonterminal to be expanded, makes an initial transfer to the appropriate label, and is otherwise exactly like procedure \( A \). Thus, the size of the parser stays comparable to that of conventional recursive descent parsers. A careful implementation of this idea would result in a bound \( O(s) \) for the construction. It should be noted, however, that for the grammars of most programming languages the sets \( R(A) \) contain usually not more than one or two elements, which means that for such grammars the algorithms are quite acceptable.
References

1. Aho, A.V., S.C. Johnson and J.D. Ullman
   Deterministic Parsing of Ambiguous Grammars
   CACM 18,8 (Aug. 75) 441-452

2. Aho, A.V., and J.D. Ullman
   The Theory of Parsing, Translation and Compiling, Vol. I
   Prentice Hall, 1973

   Burroughs Corporation, Detroit, 1968

4. Conway, M.E.
   Design of a Separable Transition - Diagram Compiler
   CACM 6,7 (Jul. 63) 396-408

5. DeRemer, F.L.
   Simple LR(k) Grammars
   CACM 14,7 (Jul. 71) 453-460

6. Gries, D.
   Compiler Construction for Digital Computers
   Wiley, 1971

7. Hunt, H.B., T.G. Szymanski and J.D. Ullman
   Operations on Sparse Relations and Efficient Algorithms for Grammar Problems,
   Proc. 15th Symp. on Switching and Aut. Thy, (Oct 74) 127-132

8. Johnson, D.B., and R. Sethi
   A Characterization of LL(1) Grammars
   to appear in BIT
   Top Down Syntactical Analysis
   Acta Informatica 1,2 (1971) 79-110

10. Koster, C.H.A.
    Affix Grammars
    in ALGOL 68 Implementation, Peck, ed., North Holland 1971

11. Koster, C.H.A.
    Using the CDL Compiler Compiler

12. Lewis, P.M.
    Attributed Translation
    J. of Comp. and Sys. Sci. 9,3 (Dec. 74) 279-307

13. Lewis, P.M., and J.D. Rosenkrantz
    An ALGOL Compiler designed using Automata Theory
    Proc. Symp. on Comp. and Automata, Polytechnic Instit. of Brooklyn, New York 1971, p. 75-88

14. Lewis, P.M., J.D. Rosenkrantz and R.E. Stearns
    Compiler Design Theory
    Addison-Wesley, 1976

15. Lewis, P.M. and R.E. Stearns
    Syntax Directed Transduction
    JACM 15,3 (Mar. 68) 438-464

16. Stearns, R.E.
    Deterministic Top Down Parsing

17. Rosenkrantz, D.J., and R.E. Stearns
    Properties of Deterministic Top-Down Grammars
    Inf. and Control 14,5 (1969) 226-256