1-1-1981

An implementation of Graham-Harrison-Ruzzo's LR-type parsing algorithm for context-free languages.

Jusuf Liauw Ibrahim

AN IMPLEMENTATION OF GRAHAM-HARRISON-RUZZO'S LR-TYPE PARSING ALGORITHM FOR CONTEXT-FREE LANGUAGES

by

Jusuf Liauw Ibrahim

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computing Science

Department of Mathematics

Division of Computing and Information Science

Lehigh University

1981

This thesis is accepted and approved in partial fulfilment of the requirements for the degree of Master of Science.

December 11/81
Date

Professor in Charge

Head of the Division

Acknowledgment

The author wishes to thank Professor Samuel L. Gulden for his helpful suggestions and thoughts in the preparation of this paper.

TABLE OF CONTENTS

1. ABSTRACT

AN IMPLEMENTATION OF GRAHAM-HARRISON-RUZZO'S LR-TYPE PARSING ALGORITHM FOR CONTEXT-FREE LANGUAGES

by Jusuf Liauw Ibrahim

An LR-type parsing algorithm by Graham, Harrison, and Ruzzo is implemented in this paper. It works on any cycle-free context-free language, thus requiring no other initial transformation of the grammar. Some background on the theory of context-free grammars is given, and a detailed analysis of the algorithm is also shown. Logically, it consists of two parts, namely, the parsing table generator, and the driver routine (or simply the parser). The parsing table is an upper-triangular 0-origin (n+1)x(n+1) matrix. Its entries are sets of dotted rules. These rules provide the basic information for the parser as it generates the sequence of rightmost parse of an input string with respect to the given grammar.

2. INTRODUCTION

A grammar forms the underlying method in deriving sets of strings of a language. In this paper we shall deal with a particular class of grammars known as context-free grammars (CFG), and an implementation of an LR-type parsing algorithm to recognize the language produced.

An LR parser scans the input string from Left-to-Right and constructs a Rightmost derivation in reverse (hence, the name LR). The LR parsing algorithm used in this paper is due to Graham, Harrison, and Ruzzo [4]. The implementation was done in a way that works on any context-free grammar containing no cyclic productions or useless symbols. (A method is available in [4] which can be used to eliminate useless symbols in a CFG.)

Logically, the parser consists of two parts, a driver routine and a parsing table. The driver routine (or simply the parser) is a recursive procedure which provides a means to "re-construct" the given input string by producing a rightmost parse from the parsing table. If it fails, an error message is announced.

The parsing table, or recognition matrix, is a 0-origin (n+1)x(n+1) upper-triangular matrix, where n is the length of the input string. The entries in the matrix are "dotted rules"

as defined in the next section.

3. BACKGROUND

In this section we present the notation and terminology used in this paper.

A context-free grammar (CFG) is defined to be a 4-tuple $G = (V, T, P, S)$, where:

V is a finite non-empty set called the total vocabulary;

$T \subseteq V$ is a finite nonempty set called the terminals;

$S \in V - T = N$ is called the start symbol, and N the set of nonterminals;

P is a finite set of rules (productions) of the form $A \longrightarrow \alpha$ where A is in N, and α in V^*.

We assume the productions are numbered $1,2,\ldots,p$ in some order.

Referring to a CFG $G = (V, T, P, S)$, the capital letters near the beginning of the alphabet denote nonterminals in N; single lower-case letters a,b,c,..., operator symbols such as +,-,etc., punctuation symbols such as parentheses, brackets, etc., and the digits $0,1,\ldots,9$ denote terminals in T. Capital symbols near the end of the alphabet such as X, Y, Z, represent grammar symbols, that is, either nonterminals or terminals. Small letters near the end of the alphabet, such as u,v,...,z,

represent strings of terminals. Lower-case Greek letters α, β, γ, for example, represent strings of grammar symbols. We use <> for the empty string.

The symbol $\Longrightarrow$ means "derives in one step," thus if $\alpha \Longrightarrow \beta$, then $\alpha = \alpha_1 A \alpha_3$, $\beta = \alpha_1 \alpha_2 \alpha_3$, and $A \longrightarrow \alpha_2$ is a rule in P. If α_3 is in T*, then the replacement is said to be rightmost, and we write $\alpha_1 A \alpha_3 \underset{r}{\Longrightarrow} \alpha_1 \alpha_2 \alpha_3$. Often we wish to say "derives in zero or more steps." For this purpose we use the symbol $\overset{*}{\Longrightarrow}$. It is a relation that denotes the transitive and reflexive closure of the relation $\Longrightarrow$. (e.g. $A \Longrightarrow \alpha \Longrightarrow \ldots \Longrightarrow \delta$ implies $A \overset{*}{\Longrightarrow} \delta$.)

A sentential form is any string in V* derivable from S. A sentence is any terminal sentential form. The language L(G) generated by a context-free grammar G is the set of sentences, i.e. $L(G) = \{w \text{ in } T^* \text{ such that } S \overset{*}{\Longrightarrow} w\}$. It is well known that for every sentence in L(G), there exists a rightmost derivation $S \overset{*}{\Longrightarrow} w$. If for each $A \longrightarrow \alpha$ in P, there exists a derivation $S \overset{*}{\underset{r}{\Longrightarrow}} \beta A v \underset{r}{\Longrightarrow} \beta \alpha v \overset{*}{\Longrightarrow} w$, w in T*, then G contains no useless productions. There are well known methods for detecting and removing useless productions [4,5]. If v is a string consisting of terminals only, then α is said to be a handle of $\beta \alpha v$, and thus $\beta A v$ and $\beta \alpha v$ are right sentential forms of G. In this paper we shall require that G be cycle-free, that is, for each A

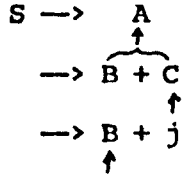
in N, $A \overset{+}{\Longrightarrow} A$ is impossible. A nonterminal A is said to be nullable if $A \overset{*}{\Longrightarrow} \langle\rangle$. Since we shall deal with rightmost derivations only, the subscript r is understood and thus dropped for convenience.

The LR parser is a bottom-up parsing algorithm, that is, given a string of terminals w in L(G), it outputs the sequence of productions in P used to construct a rightmost derivation in reverse. Or equivalently, it can be viewed as attempting to construct a parse tree for w by going from the leaves ("bottom") backwards ("up") to reach the root S, and in the process of so doing it "prunes off" the handle of each right sentential form encountered until the root S is left. The "handle pruning" process produces, in reverse, the derivation obtained by replacing the rightmost nonterminal at each step. The parser uses entries in the parsing table as a "road map" to determine to which "state of parse" it belongs after pruning off a handle.

Example 1. Consider the grammar G1

S —> A
A —> B + C
B —> i
C —> j

If w = i + j then the parse for w is as follows,



—> i + j

The up-arrows indicate the replacement trace (in reverse) when viewed in bottom-up fashion. We now define the concept of "dotted rules." [4]

Definition 1: Let G = (V, T, P, S) be a context-free grammar and let . be a symbol not in V. If A —> $\alpha\beta$ is in P, then A —> $\alpha.\beta$ is a dotted rule of G.

The symbol . is used to separate α from β, where α and β are in V^*. As the input is being read from left-to-right, the α part indicates how much of the production has been seen at a given point in the parsing process, while nothing is yet known about β. For example, A —> .XYZ would indicate that a string derivable from XYZ is expected on the input. A —> X.YZ indicates that a string derivable from X has just been seen on the input, and coming up next, a string derivable from YZ is expected. If A —> XYZ., then a handle has just been found. If A —> <> is in P, we then write the dotted rule A —> . and concatenate . with <>.

The following defines the operations which will be used in the algorithm to construct the parsing table. [4]

Definition 2: Let G = (V, T, P, S) be a context-free grammar. Let Q be a set of dotted rules, and let R $\subseteq$ V.

Define:

$$Q \times R = \{A \longrightarrow \alpha B \beta . \gamma \mid A \longrightarrow \alpha . B \beta \gamma \text{ is in } Q, \; \beta \stackrel{*}{\Longrightarrow} \langle\rangle, \text{ and } B \text{ is in } R\};$$

$$Q * R = \{A \longrightarrow \alpha B \beta . \gamma \mid A \longrightarrow \alpha . B \beta \gamma \text{ is in } Q, \; \beta \stackrel{*}{\Longrightarrow} \langle\rangle, \; B \stackrel{*}{\Longrightarrow} C \text{ for some } C \text{ in } R\}.$$

The x-product will form the appropriate dotted rule as each input character is read and matched against the righthand-side of some production. The *-product is a method of precomputing chain derivations. These products can be extended to the case where both arguments are sets of dotted rules.

Definition 3: Let $G = (V, T, P, S)$ be a context-free grammar and let Q, R be sets of dotted rules. Define:

$$Q \times R = \{A \longrightarrow \alpha B \beta . \gamma \mid A \longrightarrow \alpha . B \beta \gamma \text{ is in } Q, \; \beta \stackrel{*}{\Longrightarrow} \langle\rangle, \text{ and } B \longrightarrow \eta . \text{ is in } R\};$$

$$Q * R = \{A \longrightarrow \alpha B \beta . \gamma \mid A \longrightarrow \alpha . B \beta \gamma \text{ is in } Q, \; \beta \stackrel{*}{\Longrightarrow} \langle\rangle, \; B \stackrel{*}{\Longrightarrow} C \text{ for some } C \text{ in } N \text{ and } C \longrightarrow \eta . \text{ is in } R\}.$$

Note that in both definitions above, $Q \times R \subseteq Q * R$. The algorithm also uses a predictor to guess which dotted rules may be needed next. The predictor depends only on the grammar and

can be precomputed for each A in N or dotted rule.

Definition 4: Let G = (V, T, P, S) be a context-free grammar and let R $\subseteq$ V. Define:

$$\text{predict}(R) = \{C \longrightarrow \gamma.\delta \mid C \longrightarrow \gamma\delta \text{ is in } P,\ \gamma \stackrel{*}{\Longrightarrow} \langle\rangle,\ B \stackrel{*}{\Longrightarrow} C\eta,\ B \text{ is in } R, \text{ and } \eta \text{ is in } V^*\}.$$

If R is a set of dotted rules, then

$$\text{predict}(R) = \text{predict}(\{B \mid A \longrightarrow \alpha.B\beta \text{ is in } R\}).$$

4. ANALYSIS OF THE ALGORITHMS

In this section we will analyze in detail two algorithms: one that generates the parsing table and the other that outputs the rightmost parse sequence. Formal proofs for both algorithms can be found in [4].

Algorithm 1: Let $G = (V, T, P, S)$ be any context-free grammar. Let $w = a_1 \ldots a_n$, where $n \geq 0$ and a_k is in T for each k, $1 \leq k \leq n$, be the string to be recognized. Form an $(n+1)x(n+1)$ matrix $T = (t_{i,j})$ as follows:

```
begin
   t_{0,0} := PREDICT({S});

   for j := 1 to n do
     begin
       comment build col. j, given cols. 0,...,j-1;
scanner:
       for 0 ≤ i ≤ j-1 do

       t_{i,j} := t_{i,j-1} X {a_j}

completer:
       for k := j-1 downto 0 do
         begin

           t_{k,j} := t_{k,j} ∪ t_{k,k} * t_{k,j};

           for i := k-1 downto 0 do

             t_{i,j} := t_{i,j} ∪ t_{i,k} X t_{k,j}
         end;

predictor: t_{j,j} := PREDICT( ∪ {t_{i,j} | 0 ≤ i ≤ j-1})
     end
end.
```

Since the relation $\overset{*}{\Longrightarrow}$ is the transitive-reflexive closure of the relation $\Longrightarrow$, it is natural to use a computation technique that yields as well as preserves this relation. We introduce the computation of the transitive closure of a matrix. As a result, the recognition problem in general reduces to operations on matrices.

Let T be the $(n+1)x(n+1)$ matrix whose entries are sets of dotted rules, and n the length of the input string. We define the transitive closure on T by computing the x- and *-products as specified in Algorithm 1.

The order of computation is as follows. Suppose columns $0,\ldots,j-1$ have been built. To construct column j, the scanner places new dotted rules into the off-diagonal part of the column. Next the completer works its way up the column from $t_{j-1,j}$ to $t_{0,j}$. For $0 \leq k < j$, the entry $t_{k,j}$ is computed by adding the * product of the diagonal entry of the row to the original contents of $t_{k,j}$. (See Figure 1.)

Then $t_{k,j}$ is cross multiplied with $t_{i,k}$ and the result added to $t_{i,j}$ for all rows above $t_{k,j}$. (See Figure 2.)

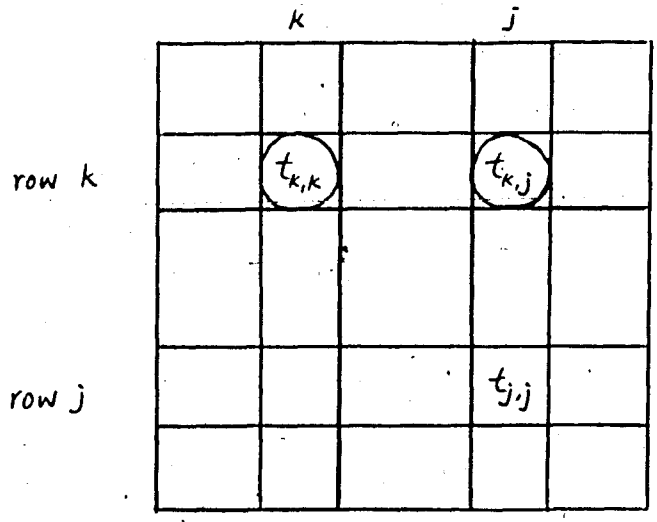


Figure 1.

The computation is repeated for each next entry $t_{k-1,j}$ until it reaches and fills in $t_{0,j}$. Finally, the predictor fills in $t_{j,j}$.

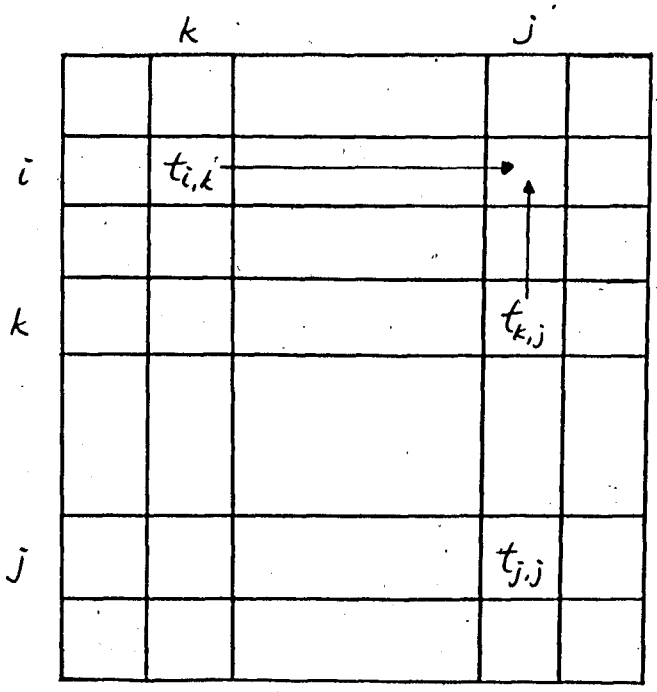


Figure 2.

The implementation program for this algorithm was written in the language Pascal. There are three important global variables used, namely: prod_tab - a two dimensional array used to store the productions 1,2,...,p of the grammar; str_tab - a two dimensional array used to store the input strings; and matrix - a linear array representation of the (n+1)x(n+1) parsing table.

The matrix is a record type of

```
record
  pds: an array of record
                 prodno, dotpos: integer ;
                        end;
  total: integer;
    (* total of dotted rules in that
       matrix element *)
end
```

Internally, each dotted rule is represented by two integers: a production number (prodno) and position of the dot with respect to the leftmost symbol in the production (dotpos). The relation between an entry $t_{i,j}$ in the parsing table and an element in matrix[k] is given by the formula

$$k = (i \times n) - (i(i-1) \text{ div } 2) + j$$

where n is the length of the input string. We now execute the algorithm on a grammar given in the next example.

Example 2. Let G2 be the grammar

S —> AB
A —> <>
B —> CDC
C —> <>
D —> a

and let w = a.

The procedure PREDICT([S]) forms the starting matrix by filling the diagonal entry $t_{0,0}$ with

[S —> .AB, S —> A.B, A —> ., B —> .CDC, B —> C.DC,
C —> ., D —> .a].

These dotted rules form the basis for which the scanner and completer look for handles. Intuitively, a dotted rule such as B —> C.DC indicates the possibility that a string derivable from D will appear next on the input. Since C —> <> is in P, only the empty string <> is derivable from C. The predictor indicates this by placing the dot prior to D - a non-nullable nonterminal. The rest of the algorithm will, in a general sense, build the parsing table by taking the "transitive closure" of the starting matrix.

The outer for loop is executed once, for j = 1. The scanner "reads" the current input character, namely a, in w. Then using x-product of Definition 2, it determines from the previous column entry ($t_{0,0}$) whether or not a is expected now. If it is, there must exist a dotted rule in $t_{0,0}$ of the type $A \rightarrow \alpha.a\beta\gamma$, where γ is in V*, $\alpha \stackrel{*}{\Longrightarrow} <>$ and $\beta \stackrel{*}{\Longrightarrow} <>$. The scanner then forms $A \rightarrow \alpha a\beta.\gamma$ and places it in $t_{0,1}$. We know that D —> .a is in $t_{0,0}$, hence D —> a. is put into $t_{0,1}$. Notice that D —> a. implies a handle has just been found.

The following is Procedure SCANNER:

```
Procedure SCANNER (j,n,strno: integer);
  var
    i,k,p,q: integer;

  procedure fill_mtx;
    (* insert new dotted rule into t[i,j] *)

  begin (* scanner *)
    for i := 0 to j-1 do
      begin
        q := calc_pos(i,j,n);
        p := calc_pos(i,j-1,n);

        with matrix[p] do
          for k := 1 to total do with pds[k] do
            if prod_tab[prodno,dotpos+1] =
                 str_tab[strno,j]
            then fill_mtx(prodno,dotpos);
      end
  end (* scanner *);
```

Next the completer executes the inner for loop once for k = 0. Its purpose is to record all of those productions in P whose right sides contain a string or substring $\alpha B\beta$ which in zero or more steps derive the handle found earlier by the scanner. (We know a handle always derives a string of terminals.) Overall, this action corresponds to recording the replacement trace (in reverse) as shown previously in Example 1.

Using *-product of Definition 3, the completer searches for such strings or substrings $\alpha\beta\beta$ from the diagonal entry $t_{0,0}$ and yields

$$[S \longrightarrow AB.,\quad B \longrightarrow CD.C,\quad B \longrightarrow CDC.].$$

This new set of dotted rules is then added to $t_{0,1}$. Since

-product uses dotted rules from the diagonal entries (formed by the predictor), we know that for each $A \longrightarrow \alpha.B\beta\gamma$ in $t_{k,k}$, $\alpha \stackrel{*}{\Longrightarrow} \langle\rangle$. If $\beta \stackrel{*}{\Longrightarrow} \langle\rangle$, $B \stackrel{*}{\Longrightarrow} C$, $C \longrightarrow \eta .$ is in $t_{k,j}$ (formed by the scanner), and we know that $\eta \stackrel{*}{\Longrightarrow} y$, y in T, then $A \longrightarrow \alpha B\beta.\gamma$ is in $t_{k,k} * t_{k,j}$ because $\alpha B\beta \stackrel{*}{\Longrightarrow} B\beta \stackrel{*}{\Longrightarrow} C \stackrel{*}{\Longrightarrow} \eta \stackrel{*}{\Longrightarrow} y$.

The for loop containing the x-product is not executed. However, for $j \geq 2$, it extends the search up the column where *-product started. If $A \longrightarrow \alpha.B\beta\gamma$ is in $t_{i,k}$, $\alpha \stackrel{*}{\Longrightarrow} x$, x in T*, $\beta \stackrel{*}{\Longrightarrow} \langle\rangle$, $B \longrightarrow \eta .$ is in $t_{k,j}$ (formed by the scanner), and $\eta \stackrel{*}{\Longrightarrow} y$, y in T*, then $A \longrightarrow \alpha B\beta.\gamma$ is in $t_{i,k} \times t_{k,j}$ because $\alpha B\beta \stackrel{*}{\Longrightarrow} xB\beta \Longrightarrow x\eta\beta \stackrel{*}{\Longrightarrow} x\eta \stackrel{*}{\Longrightarrow} xy$.

The Procedure COMPLETER contains three major sub-procedures.

```
Procedure COMPLETER (j,n: integer);
  var
    i,k: integer;

  procedure fixdot;
    (* moves the dot to a new position *)

  procedure chain_deriv(t,B,pr,dot,yy: integer);
    var
      s, C: integer;
    begin
      fix_dot(B,pr,dot,yy);
      t := t + 1;   search(B,t);   s := 2;
      C := prod_tab[t,s];
      if chr(C) in nt_set then
        begin
          if B = C then
            begin
              if prod_tab[t,3] = endmarker
```

```
            then error(6,0)    (* a cycle *)
            else chain_deriv(t,C,pr,dot,yy);
          end
        else if chr(C) in nullables then

          begin
            skip_nullables(t,s);
            if prod_tab[t,s] <> endmarker then
               chain_deriv(0,prod_tab[t,s],pr,dot,yy);
          end
        else chain_deriv(0,C,pr,dot,yy);
      end
  end (* chain_deriv *);

procedure fill_matx;
  (* inserts new dotted rules into t[i,j] *)

procedure get_B;
  (* gets the nonterminal B which chain_derives C
     in *-product of Definition 3 *)
```

```
procedure star_product(k,j,n: integer);
  var
    p,q: integer;    str: boolean;
  begin
    (* perform t[k,j] := t[k,j] U t[k,k] * t[k,j] *)
    p := calc_pos(k,j,n);
    q := calc_pos(k,k,n);
    str := true;  (* a signal to chain derive B *)
    get_B(p,q,p,str);
  end (* star_product *);

procedure cross_product(k,j,n: integer);
  var
    i,r,s,t: integer;    str: boolean;
  begin
    (* perform t[i,j] := t[i,j] U t[i,k] X t[k,j] *)
    for i := k-1 downto 0 do
      begin
        r := calc_pos(i,j,n);
        s := calc_pos(i,j,n);
        t := calc_pos(k,j,n);
        init_tpds;   str := false;
        get_B(s,t,r,str);
      end
  end (* cross product *);

begin (* completer *)
  for k := j-1 downto 0 do
    begin
      init_tpds;
      star_product(k,j,n);
      cross_product(k,j,n);
    end
end (* completer *);
```

Finally the predictor forms a set of dotted rules which will be used by the completer as a basis for its computation of the next column, if any. By taking the union of all column j entries above $t_{j,j}$, the predictor can "see" how much recognition has been performed so far with respect to the input string w. The substring to the left of the dot of each dotted rule in

the union indicates this. A nonterminal to the right of the dot indicates the opposite. Hence, the predictor records the corresponding production in P and thereby provides information for the completer to take action later. In $t_{0,1}$ we see only the nonterminal C is left. Since C —> <> is in P, the predictor need only place C —> . in $t_{1,1}$ and thus completes the construction of the parsing table. The final table for w = a is shown in Figure 3.

$S \longrightarrow \cdot AB$ $A \longrightarrow \cdot$ $S \longrightarrow A \cdot B$ $B \longrightarrow \cdot CDC$ $C \longrightarrow \cdot$ $B \longrightarrow C \cdot DC$ $D \longrightarrow \cdot a$	$D \longrightarrow a \cdot$ $S \longrightarrow AB \cdot$ $B \longrightarrow CD \cdot C$ $D \longrightarrow CDC \cdot$
	$C \longrightarrow \cdot$

Figure 3.

The Procedure PREDICTOR consists of two subprocedures: form_union and pred_union.

```
Procedure PREDICTOR(j,n: integer);
  var
    ttotal: integer;

  procedure form_union;
    (* forms union of {t[i,j] | 0 ≤ i ≤ j-1} *)

  procedure pred_union;
    var
      B,p,s: integer;

      procedure fill_JJ;
        (* inserts new dotted rule in t[j,j] *)

      procedure tjain_deriv;
        (* performs chain derivation of nonterm. B
           in Definition 4 *)

      begin (* pred_union *)
        p := calc_pos(j,j,n);
        for s := 1 to ttotal do with tpds[s] do
          begin
            B := prod_tab[tprodno,tdotpos+1];
            tjain_deriv(B);
          end
      end (* pred union *);

  begin (* predictor *)
    init_tpds;
    ttotal := 0;
    form_union;
    pred_union;
  end (* predictor *);
```

The main program of the procedure BUILD_MATRIX looks like

```
begin
  init_matrix;
  predict_S;
  for j := 1 to n do
    begin
      scanner(j,n,strno);
      completer(j,n);
      predictor(j,n);
    end
 end (* build_matrix *);
```

Example 3. Consider the grammar G3

```
S  —>  E
E  —>  E + T
E  —>  T
T  —>  T * F
T  —>  F
F  —>  (E)
F  —>  a
```

and let w = a * a. Figure 4 gives the parsing table.

Notice that the top row of the table presents the history of the parse as indicated by the underlined dotted rules. Each column of that row reflects the instantaneous "state of the parse" with respect to an appropriate input character. Hence, the parser we construct must be able to enter successively each of those "states of the parse" to produce the rightmost parse sequence.

We now give the parser algorithm.

Algorithm 2: Let $G = (V, T, P, S)$ be a cycle-free, context-free grammar with the productions $\pi_1, \pi_2, \ldots, \pi_p$. Let $w = a_1 a_2 \ldots a_n$, $n > 0$, be a string, where, for $1 \leq i \leq n$, a_i is in T; and let $T = (t_{i,j})$ be the parsing table for w constructed by Algorithm 1.

Define the recursive procedure PARSE(i, j, π_m) which generates a rightmost parse for $A \Longrightarrow \alpha \overset{*}{\Longrightarrow} a_{i+1} \ldots a_j$, where $\pi_m = A \longrightarrow \alpha$, as follows:

```
Procedure PARSE (i, j, π_m = A --> A_1 A_2 ... A_p );
  begin
    output(m);   j_0 := j;
    for d := p_m  downto 1 do
      if A_d is in T
      then j_0 := j_0 - 1
      else
        if k is the greatest integer  i < k < j_0
            such that for some     in V*,

            A_d --> α·  is in t_{k,j}      and

            A --> A_1 A_2 ... A_{d-1} · A_d ... A_p

                        is in t_{i,k}
        then  begin
                PARSE (k, j_0, A_d --> α );
                j_0 := k;
              end;
  end;
```

Output a right parse for w as follows:

```
main program:    if for some α in V*,
                    S --> α·  is in t_{0,n}
                 then PARSE (0, n, S ⟶ α )
                 else output("error").
```

The entry $t_{0,n}$ of the parsing table is the last one filled

in by the scanner and completer. Since the parsing technique we used in constructing the table is bottom-up, there must exist in $t_{0,n}$ a dotted rule $S \longrightarrow \alpha\cdot$ for some α in V^*. Otherwise, we know the input string contains error(s). Traversing "top-down" with respect to the parse tree, the procedure PARSE starts from the production $S \longrightarrow \alpha$. It examines each A in α from right-to-left, where A is in V. If A is a nonterminal then a replacement for it is found by matching entries from the column k,j

$$A_d \longrightarrow \alpha\cdot \text{ in } t_{k,j}$$

and the row i,k

$$A \longrightarrow A_1 A_2 \cdots A_{d-1} \cdot A_d \cdots A_{p_m} \text{ in } t_{i,k}.$$

$A_d \longrightarrow \alpha$ implies there may exist an A in α such that A is a nonterminal. Hence, a recursive call to PARSE is made to find the replacement for such A. This process is repeated until, for some d, we have $A_d \longrightarrow$ a where a is a terminal. When this occurs, the procedure PARSE does not make a call and it terminates.

In Figure 4 we see that $S \longrightarrow E.$ is in $t_{0,3}$, therefore PARSE (0, 3, S —> E) is invoked, yielding (1, PARSE(0, 3, E —> T)). PARSE(0, 3, E —> T) generates (3, PARSE(0, 3, T —> T * F)). Then PARSE(0, 3, T —> T * F) generates (4, PARSE(2, 3, F —> a), PARSE(0, 1, T —> F)). PARSE(2, 3, F —> a) yields 7, and PARSE(0, 1, T —> F) generates (5,

PARSE(0, 1, F —> a)) which yields (5, 7). Thus the output of the algorithm is (1, 3, 4, 7, 5, 7).

The implementation of the parser is as follows:

```
Procedure PARSING (n,strno: integer);
  var
    pno: integer;

  function alpha;
    (* a boolean check for the case of singular
       nonterminal as in Example 4 *)

  function alpfa;
    (* a boolean check for S —> α  *)
```

$S \to \cdot E$ $E \to \cdot E + T$ $E \to \cdot T$ $\underline{T \to \cdot T * F}$ $T \to \cdot F$ $F \to \cdot (E)$ $F \to \cdot a$	$F \to a \cdot$ $E \to E \cdot + T$ $E \to T \cdot$ $\underline{T \to T \cdot * F}$ $T \to F \cdot$ $S \to E \cdot$	$\underline{T \to T * \cdot F}$	$\underline{T \to T * F \cdot}$ $S \to E \cdot$ $E \to E \cdot + T$ $E \to T \cdot$ $T \to T \cdot * F$
		$F \to \cdot (E)$ $F \to \cdot a$	$F \to a \cdot$

Figure 4.

```
function alphatwo;
  (* a boolean check for

       A   —> α    in  t      and
        d                k,j

       A   —>  A  A   ... A    . A  ... A  in t    *)
                1  2       d-1    d      p     i,k
                                          m

Procedure PARSE (i,j,pno: integer);
  var
    pno2,jz,pm,el,k: integer;
  begin
    write(pno:3);  jz := j;
    pm := prod_tab[pno,0];
    for el := pm downto 2 do
      if chr(prod_tab[pno,el]) in t_set
      then jz := jz - 1
      else
        begin
          pno2 := pno;
          if alphatwo(i,jz,el,k,pno2)
          then begin
                 PARSE (k,jz,pno2);
                 jz := k;
               end
        end
  end (* parse *)

begin (* parsing *)
  if alpha(pno) or alpfa(pno)
  then parse(0,n,pno)
  else error(5,strno)
end (* parsing *);
```

The main program for the algorithm is:

```
begin (* main *)
  enter_input;
  print_t_tab;    (* print terminals *)
  print_nt_tab;   (* print nonterms. *)
  for j := 1 to stringnum do
    begin
      build_matrix(str_tab[j,0],j);
      parsing(str_tab[j,0],j);
99:  end;
end (* main *).
```

Let us now examine the behavior of the parser when given an ambiguous grammar.

Example 4: Let G4 be the ambiguous grammar given by

E —> E * E

E —> E + E

E —> a

and let w = a + a * a. Figure 5 gives the parsing table.

A context-free grammar G is said to be ambiguous if for w in L(G), there exists more than one rightmost generations from S. The string w in Example 4 has two rightmost generations from S, or equivalently, it has two "parse trees," namely

E —> E + E

—> E + E * E

—> E + E * a

—> E + a * a

—> a + a * a

yielding the parse sequence (2, 1, 3, 3, 3), and

E —> E * E

—> E * a

MICROBEX CORRECTION GUIDE (M-6)

CORRECTION

The preceding document has been re-photographed to assure legibility and its image appears immediately hereafter.

RP 8868

REMINGTON RAND
OFFICE SYSTEMS DIVISION

The main program for the algorithm is:

```
begin (* main *)
  enter_input;
  print_t_tab;    (* print terminals *)
  print_nt_tab;   (* print nonterms. *)
  for j := 1 to stringnum do
    begin
      build_matrix(str_tab[j,0],j);
      parsing(str_tab[j,0],j);
99:  end;
end (* main *).
```

Let us now examine the behavior of the parser when given an ambiguous grammar.

Example 4: Let G4 be the ambiguous grammar given by

E —> E * E

E —> E + E

E —> a

and let w = a + a * a. Figure 5 gives the parsing table.

A context-free grammar G is said to be ambiguous if for w in L(G), there exists more than one rightmost generations from S. The string w in Example 4 has two rightmost generations from S, or equivalently, it has two "parse trees," namely

E —> E + E

—> E + E * E

—> E + E * a

—> E + a * a

—> a + a * a

yielding the parse sequence (2, 1, 3, 3, 3), and

E —> E * E

—> E * a

—> E + E * a

—> E + a * a

—> a + a * a

yielding the parse sequence (1, 3, 2, 3, 3).

The algorithms we have introduced will always generate the first parse sequence (2, 1, 3, 3, 3) for w. We can find the reasons for this behavior by examining entries in columns 2 and 3 of the parsing table in Figure 5.

In table entry $t_{0,1}$ we see there are two choices presented. The substring E.*E tells us that possibly the next input character is a *, while E.+E guesses a + may appear. However, the scanner in Algorithm 1 was designed in such a way that it does not read beyond the current input character, which in our present case is a +. Therefore, it formed the substring E+.E for the dotted rule in $t_{0,2}$, indicating that a + has just been read, and thus determined once and for all the course of the parse for w. This behavior typically characterizes that of an LR(0) type parser.

MICROMEX CORRECTION SHEET (M-8)

CORRECTION

The preceding document has been re-photographed to assure legibility and its image appears immediately hereafter.

REMINGTON RAND
OFFICE SYSTEMS DIVISION

$\longrightarrow$ E + E * a

$\longrightarrow$ E + a * a

$\longrightarrow$ a + a * a

yielding the parse sequence (1, 3, 2, 3, 3).

The algorithms we have introduced will always generate the first parse sequence (2, 1, 3, 3, 3) for w. We can find the reasons for this behavior by examining entries in columns 2 and 3 of the parsing table in Figure 5.

In table entry $t_{0,1}$ we see there are two choices presented. The substring E.*E tells us that possibly the next input character is a *, while E.+E guesses a + may appear. However, the scanner in Algorithm 1 was designed in such a way that it does not read beyond the current input character, which in our present case is a +. Therefore, it formed the substring E+.E for the dotted rule in $t_{0,2}$, indicating that a + has just been read, and thus determined once and for all the course of the parse for w. This behavior typically characterizes that of an LR(0) type parser.

$E \to \cdot E * E$ $E \to \cdot E + E$ $E \to \cdot a$	$E \to E \cdot * E$ $E \to E \cdot + E$ $E \to a \cdot$	$E \to E + \cdot E$	$E \to E + E \cdot$ $E \to E \cdot * E$ $E \to E \cdot + E$	$E \to E * \cdot E$	$E \to E + E \cdot$ $E \to E \cdot * E$ $E \to E \cdot + E$ $E \to E * E \cdot$
		$E \to \cdot E * E$ $E \to \cdot E + E$ $E \to \cdot a$	$E \to E \cdot * E$ $E \to E \cdot + E$ $E \to a \cdot$	$E \to E * \cdot E$	$E \to E * E \cdot$ $E \to E \cdot * E$ $E \to E \cdot + E$
				$E \to \cdot E * E$ $E \to \cdot E + E$ $E \to \cdot a$	$E \to E \cdot * E$ $E \to E \cdot + E$ $E \to a \cdot$

Figure 5.

Depending on the nature of the language intended to be generated by G, this kind of behavior may prove disadvantageous. One way

to avoid this problem is to rewrite the grammar so that it is unambiguous for all input strings; or, introduce functions which will accept or reject parse trees of the grammar, so each input string has at most one parse tree acceptable to all these functions. Each of the functions is called a disambiguating rule.

For G4 we can specify a disambiguating rule that + and * are to be left associative and that * is to have higher precedence than +. Thus this disambiguating rule accepts the first parse sequence (2, 1, 3, 3, 3) and rejects the second one. More detailed discussion on this subject can be found in [2].

Since E is the only nonterminal used in G4, any one of the three productions may become the "starting production" to be used by the parser. E —> E + E is the most obvious choice to replace the initial substring a + a, and E —> E + E . is in $t_{0,5}$, hence, PARSE(0, 5, E —> E + E) is invoked, yielding (2, PARSE(2, 5, E —> E * E), PARSE(0, 1, E —> a)). Then PARSE(2, 5, E —> E * E) generates (1, PARSE(4, 5, E —> a), PARSE(2, 3, E —> a)), which yields (1, 3, 3). PARSE(0, 1, E —> a) yields 3. Thus the output of the algorithm is (2, 1, 3, 3, 3).

Consistency in the grammar specification can be maintained by introducing another production S —> E for the "starting production." The resulting grammar G5 then looks like

S —> E
E —> E * E
E —> E + E
E —> a.

G5 is said to be the augmented grammar of G4. The purpose of this addition is to indicate to the parsing table generator when it should stop parsing. This occurs when the completer forms the dotted rule S —> E. in $t_{0,3}$. The resulting parse sequence for w is (1, 2, 4, 3, 4, 4).

5. CONCLUSIONS

The recognition algorithm we have analyzed is quite suitable for practical use. It works on any cycle-free context-free language, so no other initial transformation of the grammar is needed. One can write a context-free grammar and have the algorithm generate a parsing table for it within reasonable time and space requirements. (See [4] for detailed discussion on general time and space bounds for context-free language recognition.) In many ways the structure of this algorithm provides more ease and directness for its implementation than the generally available LR(0) parsing algorithm. However, its usefulness in compiler construction is curtailed by the non-linear time bound and large constants required.

The complete computer program developed to implement this algorithm was written in the language Pasal. It is filed and accessible at the office of the Division of Computing and Information Science, Lehigh University, Bethlehem, Pa.

6. LIST OF REFERENCES

[1] Aho, A.V., and Ullman, J.D., Principles of Compiler Design. Addison-Wesley, Reading, Mass., 1978.

[2] Aho, A.V., Johnson, S.C., and Ullman, J.D., "Deterministic parsing of ambiguous grammars," Comm. ACM 18:8, pp. 441-452.

[3] DeRemer, F.L., "Simple LR(k) grammars," Comm. ACM 14:7, pp. 453-460.

[4] Harrison, M.A., Introduction to Formal Language Theory. Addison-Wesley, Reading, Mass., 1978.

[5] Hopcroft, J.E., and Ullman, J.D., Formal Languages and Their Relation to Automata. Addison-Wesley, Reading, Mass., 1969.

[6] Knuth, D.E., "On translation of languages from left to right," Information and Control, 8:6, pp. 607-639.

[7] Korenjak, A.J., "A practical method for constructing LR(k) processors," Comm. ACM 12:11, pp. 613-623.

7. Vita

The author was born to Mr. and Mrs. Philip K.L. Liauw on August 5, 1955 in Jakarta, Indonesia. He earned his B.A. degree in Mathematics from Houghton College (Houghton, N.Y.) in 1977. In the Fall 1978, he began graduate study in Computing Science at Lehigh University. After more than a year of absence he resumed his study in Fall 1980. He was a teaching assistant in Mathematics for the Department of Mathematics of Lehigh University.