1-1-1984

Implementation of Deremer's SLR(1) parser.

Lih-Ling Tzuu

IMPLEMENTATION OF DEREMER'S SLR(1) PARSER

by

LIH-LING TZUU

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

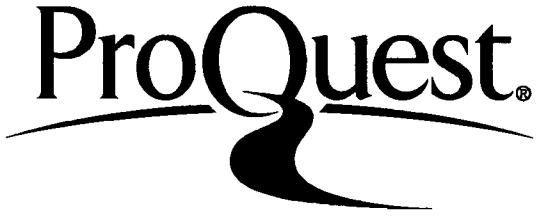
Master of Science

in

Computing Science

Lehigh University

1984



Certificate of Approval

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

May 11, 1984
(date)

Professor in Charge

Head of Division

Table of Contents

ABSTRACT

IMPLEMENTATION OF DEREMER'S SLR(1) PARSER

by Lih-ling Tzuu

A class of context-free grammars, called the "simple LR(K)" or "slr(K)" grammar, defined by Franklin L. DeRemer is implemented in this paper. It works on any cycle-free SLR(K) grammar, thus requiring no other initial transformation of the grammar. Some background on the theory of context-free grammar is given and a detail analysis of the SLR(K) parsing method is also shown. Logically, it consists of three parts, namely, the LR(0) sets of items, the set of parse tables and the driver routine (or simply the parser). The item sets are constructed utilizing a doubly linked-list structure. For the purpose of direct access to next state, the linked list is arranged into a semi-network linkage. The set of parse tables derived from sets of items is presented as an upper triangular (states) x (grammar symbols) matrix. Each table is a pair of functions <f,g>, where f is the action part and g is the goto part. The set of tables provides the information for the parser as it generates the rightmost parse of an input string with respect to the given grammar.

1. INTRODUCTION

A parser is a logic machine that recognizes the language generated from a grammar. In this paper, we shall implement a parser generator for a particular class of context-free grammars known as Simple LR(K) grammars [SLR(K)].

SLR(K) parsing algorithm is an LR parsing which scans the input string from left-to-right and constructs a rightmost derivation in reverse. These parser construction techniques are practical both in speed of parser construction and in the size and speed of parsers produced (hence, the name SLR).

Logically, the parser consists of two parts, a driver routine and a set of parse tables derived from LR(0) sets of items. The set of items are carried by a double-linked list. Its entries contain informations of each "item" relative to the "mark" and the "goto" state.

The set of parse tables consists of tables each containing a pair <f,g> of functions. "f" provides the parser with the "action" to be taken and "g" the state to "goto".

The implementation is done in PASCAL and examples are given for the algorithms analyzed. An extension of this parser-construction technique to cover LR(K) grammars is also mentioned.

2. TERMINOLOGY

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_n, V_t, S, P) where

V_n: a finite non-empty set of symbols called nonterminals;

V_t: a finite non-empty set of symbols distinct from those in V_n called terminals;

S: a distinquished member of V_n called the starting symbol;

P: a finite set of pairs called productions.

Each production is written A --> ω and has a left part A in V_n, and a right part ω in V^*, where $V = V_n \cup V_t$. V^* denotes the set of all strings composed of symbols in V.

Referring to a CFG=(V_n, V_t, P, S), the capital letters near the beginning of the alphabet denote nonterminals in V_n; single lowercase letters a, b, c, ..., operator symbols such as +, -, etc., punctuation symbols such as parentheses, brackets, etc., and the digits 0, 1, ..., 9, denote terminals in V_t. Capital symbols near the end of the alphabet such as X, Y, Z, represent grammar symbols

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, ρ, for example, represent strings of grammar symbols. The empty string is shown as λ.

The symbol ==> means "derives in one step", thus if α ==> β, then $\alpha=\alpha_1 A \alpha_3$, $\beta=\alpha_1 \alpha_2 \alpha_3$, and A --> α_2 is a production. If α_3 is in V_t^*, then the replacement is said to be rightmost, and $\alpha_1 A \alpha_3$ $\Longrightarrow_{rm}$ $\alpha_1 \alpha_2 \alpha_3$ represents this action. We use symbol $\Longrightarrow^*$ for "derives in zero or more steps". It is a relation that denotes the transitive and reflexive closure of the relation ==>. For example, A ==> α ==> ... ==> δ implies A $\Longrightarrow^*$ δ.

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 in V_t^* such that S $\Longrightarrow^*$ w}. It is well known that for every sentence in L(G), there exists a rightmost derivation S $\Longrightarrow^*$ w. If for each A --> α in p, there exists a derivation S $\Longrightarrow^*_{rm}$ βAv $\Longrightarrow_{rm}$ $\beta\alpha$v $\Longrightarrow^*$ w, w in V_t^*, then G has no useless productions. Well-known methods exist for detecting and removing useless productions [5]. If v is a string consisting of terminals only, then α is said to be a handle of $\beta\alpha$v, and thus βAv and $\beta\alpha$v are right sentential forms of G.

In this paper, we assume that G has no useless productions. And we require that G be cycle-free, that is, for each A in V_n, A $==>^+$ A, where $==>^+$ denotes "derives in one or more steps", does not exist. Since we shall deal with rightmost derivations only, the subscript rm is dropped for convenience. Now let us define FIRST, FOLLOW and EFF functions.

If α is any string of grammar symbols, let $FIRST_k(\alpha)$ be the set of terminals that begin strings derived from α. If $\alpha ==>^* \lambda$, then λ is also in $FIRST_k(\alpha)$.

For any nonterminal A, $FOLLOW_k(A)$ is the set of terminals that can appears immediately to the right of A in some sentential form, that is, $S ==>^* \alpha Aa\beta$ for some α and β. If A can be the rightmost symbol in some sentential form, then we add λ to FOLLOW(A).

We define the λ-free first function, $EFF_k(\alpha)$, as follows:

1. If α does not begin with a nonterminal, $EFF_k(\alpha)=FIRST_k(\alpha)$.
2. If α begins with a nonterminal, then then $EFF_k(\alpha)=\{w|$ there is a derivation $\alpha ==>^* \beta ==>wx$, where $\beta=Awx$ for any nonterminal A$\}$, and $w=FIRST_k(wx)$.

Thus, $EFF_k(\alpha)$ captures all members of $FIRST_k(\alpha)$ whose derivation does not involve replacing a leading nonterminal by λ.

3. THE GENERAL NOTION OF LR PARSING

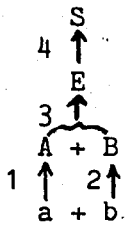
The LR parser is a bottom-up parsing algorithm proceeding in a shift-reduce fashion employing three pushdown lists -- (1) stack of states of parse (2) replacement stack (3) input stack. Shift-reduce parsing consists of shifting input symbols onto the replacement stack until a handle appears on top of the stack. The handle is then reduced, that is, it is replaced by the nonterminal symbol on the left-hand side of the production applied to this action. If no errors occur, this process is repeated until all of the input string is scanned and only the starting symbol S appears on the stack. This bottom-up parsing, equivalently, can be viewed as attempting to construct a parse tree by going from the leaves(bottom) backwards to reach the root(top) S. During the process, the handle of each right sentential form is "pruned off" until the root S is left. Thus given a terminal string w in L(G), the parser outputs the sequence of productions in P used to construct a rightmost derivation in reverse.

Example 1.

Let G(1) be defined by the productions

(1) S --> E
(2) E --> A+B
(3) A --> a
(4) B --> b

Let w=a+b, then the parse for w is as follows:



The up-arrows indicate the replacement trace when viewed in bottom-up fashion. The number besides each arrow indicates the processing sequence.

A set of LR(k) tables forms the basis of the LR(k) parsing algorithm. Each table consists of a pair of functions <f,g> representing each "state of parse". The state of parse on top of the stack associated with the symbol on top of the input stack or replacement stack dictates the behavior of the parsing. The first function f, the parsing action function, given a look-ahead string, tells us what parsing action to take. The action may be to (1) shift the next input symbol onto the replacement, (2) reduce the top symbol(s) of the replacement stack according to a named production, (3) announce completion of the parsing or (4) declare that a syntactic error has been found in the input. The second function g, the goto function, is invoked after each shift action and each reduce action. Given a symbol of the grammar, the goto function returns either the name of another table (state) or an error notation.

4. LR(0) ITEM SETS CONSTRUCTION

The basis for the SLR(k) parser is an LR(0) deterministic finite state machine (DFSM). This machine recognizes viable prefixes of the grammar, that is, prefixes of the right-sentential forms that do not contain any symbols to the right of the handle. The viable prefix therefore serves as a kind of conceptual link between derivations and the LR parsing automaton.

In the construction of this DFSM, each machine state is associated with a set of items, where an item is a production carrying a position marker ".". For example, items generated by production A --> $\alpha\beta$ are

A --> .$\alpha\beta$
A --> α.β
A --> $\alpha\beta$.

The production A --> λ generates only one item, A --> .; intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. For example, A --> .$\alpha\beta$ indicates that we are expecting to see a string derivable from $\alpha\beta$ next on the input. A --> α.β indicates that we have just seen on the input a string derivable from α and that we next expect to see a string derivable from β.

Items are grouped into sets, which give rise to the "states" of

an LR parser.

We now define an augmented grammar and three construction rules for an LR(0) item sets collection -- the start operation, the closure and the goto operation.

If G is a grammar with start symbol S, then G', the augmented grammar for G, is G with a new start symbol S' and S' --> S. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input.

On the implementation of augmented grammar, S is reserved for the new starting symbol for convenience.

The three rules are as follows:

The Start Operation. Let S be the start symbol of the grammar, and S --> E is the start production, then item S --> .E is associated with the start state.

The Closure Operation. If B --> α.Aβ is an item in some state P, then every item of the form A --> .γ must be included in state P. Here A must be a nonterminal symbol, and this rule must be repeated until no more new items can be added to the state.

The Goto Operation. Let X be a terminal or nonterminal symbol

in an item A --> α.Xβ associated with some state P. Then A --> αX.β is associated with a state Q , (possibly the same as P) and a transition

P to Q on symbol X

exists.

For now, let us summarize the construction of the state item sets as implied by the three operations just given.

1. Give the start state a number, and use the start operation to put one item into it. Then use the Closure operation repeatedly if necessary, to get more items into this state. In getting the closure of a state, we look for a nonterminal symbol X that follows the mark ".", and then add items of the form X --> .ω to the state, where X --> ω is a production.

2. We now have one state, consisting of a set of items. Any other state will be constructed by the same process, so we consider it a general state.

3. Use the goto operation to start one or more new states, based on the present state. The idea is to look for items of the form A --> α.Xβ, i.e., items in which some symbol X follows the mark, then build a new state from the item A --> αX.β, i.e., the mark "moved past" the symbol X. This new state incidentally must also contain all the other items formed from items in the old state in which this symbol X follows the mark. For example, if the old state contains the two items

E --> E+T.
E --> .T

then the new state must contain (at least) the items

E --> E+T.

E --> T.

Let the old state be P and the new state be Q, we have the transition P to Q on X.

4. Complete the new state started in step 3 by applying the closure operation repeatedly.

5. Repeat steps 3 and 4 until no more new states are obtained.

Example 2.

Given a grammar G_0 as follows:

1. S --> E
2. E --> E+T
3. E --> T
4. T --> T*F
5. T --> F
6. F --> (E)
7. F --> a

The complete item sets are shown in Fig .1. Fig .2 shows the DFSM for grammar G_0.

```
P                                   (X,Q)
I0:   S --> .E                      (E,1)
      E --> .E+T                    (E,1)
      E --> .T                      (T,2)
      T --> .T*F                    (T,2)
      T --> .F                      (F,3)
      F --> .(E)                    ((,4)
      F --> .a                      (a,5)

I1:   S --> E.                      completed
      E --> E.+T                    (+,6)

I2:   E --> T.                      completed
      T --> T.*F                    (*,7)

I3:   T --> F.                      completed

I4:   F --> (.E)                    (E,8)
      E --> .E+T                    (E,8)
      E --> .T                      (T,2)
      T --> .T*F                    (T,2)
      T --> .F                      (F,3)
      F --> .(E)                    ((,4)
      F --> .a                      (a,5)

I5:   F --> a.                      completed

I6:   E --> E+.T                    (T,9)
      T --> .T*F                    (T,9)
      T --> .F                      (F,3)
      F --> .(E)                    ((,4)
      F --> .a                      (a,5)

I7:   T --> T*.F                    (F,10)
      F --> .(E)                    ((,4)
      F --> .a                      (a,5)

I8:   E --> (E.)                    (),11)
      E --> E.+T                    (+,6)

I9:   E --> E+T.                    completed
      T --> T.*F                    (*,7)

I10:  T --> T*F.                    completed

I11:  F --> (E).                    completed
```

Fig.1. Sets of Items for Grammar G_0

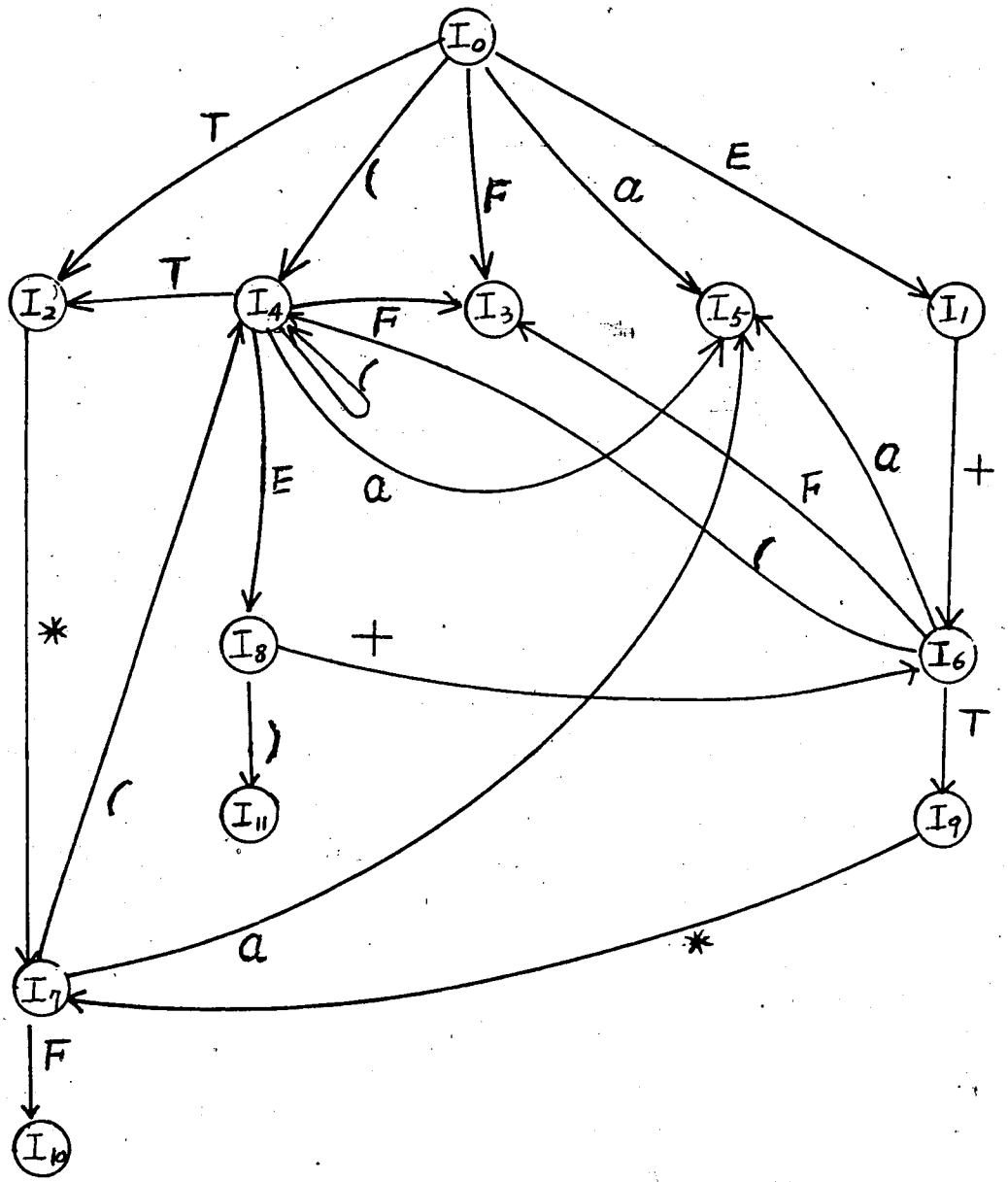


Fig. 2 Deterministic finite state machine (DFSM) for G_o

Example 3.

Consider grammar G_2 with productions as follows:

1. S --> E
2. E --> Aa
3. E --> dAb
4. E --> cb
5. E --> dca
6. A --> c

The sets of items generated from G_2 is shown in Fig.3.

P		(X,Q)
I0:	S --> .E	(E,1)
	E --> .Aa	(A,2)
	A --> .c	(c,3)
	E --> .dAb	(d,4)
	E --> .cb	(c,3)
	E --> .dca	(d,4)
I1:	S --> E.	completed
I2:	E --> A.a	(a,5)
I3:	A --> c.	completed
	E --> c.b	(b,6)
I4:	E --> d.Ab	(A,7)
	A --> .c	(c,3)
	E --> d.ca	(c,8)
I5:	E --> Aa.	completed
I6:	E --> cb.	completed
I7:	E --> dA.b	(b,9)
I8:	E --> dc.a	(a,10)
I9:	E --> dAb.	completed
I10:	E --> dca.	completed

Fig.3. Sets of Items for G_2

For grammar G_0 (Fig.1), states I_1, I_2 and I_9 are called inadequate or inconsistent states, and so does state I_3 in Fig.3 for grammar G_2. In general, an inadequate state is any state containing both a completed item (of the form A --> ω.) and any other incomplete item. An inadequate state represents a conflict in a parsing decision. If the LR states contain no conflict, the grammar is said to be LR(0).

We have to make sure that the parser system defined above establish that the machine constructed recognizes exactly the class of viable prefixes of right-most sentential forms. Following definitions are necessary for this proof.

An item A --> $\beta.\gamma$ is said to be valid for some viable prefix $\alpha\beta$ if and only if some right-most derivation

$$S \Longrightarrow^{*} \alpha A \delta \Longrightarrow \alpha\beta\gamma\delta$$

exists.

A string ω is said to be associated with or valid for some state P in the machine if and only if the machine falls into state P upon scanning ω.

An item is said to be valid for some state P in the machine if and only if it is valid for some viable prefix ω associated with state P.

Now we may state the construction correctness of the machine.

Lemma.

Every state P contains all and only the items valid for P.

Main Theorem.

The machine M (constructed as above) recognizes exactly the class of viable prefixes of grammar G.

Formal proofs for the above lemma and theorem can be found in [3].

Example 4.

Let us consider grammar G_0 (Example 2) again, whose sets of items and GOTO operations are exhibited in Figs.1 and 2. The string E+T* is a viable prefix of G_0. The machine of Fig.2 will be in state I_7 after having read E+T*. State I_7 contains the items

```
T --> T*.F
F --> .(E)
F --> .a
```

which are precisely the items valid for E+T*. To see this, consider the following three rightmost derivations

```
(1) S ==> E
      ==> E+T
      ==> E+T*F
(2) S ==> E
      ==> E+T
```

```
          ==> E+T*F
          ==> E+T*(E)

   (3) S  ==> E
          ==> E+T
          ==> E+T*F
          ==> E+T*a
```

The first derivation shows the validity of T --> T*.F, the second the validity of F --> .(E), and the third the validity of F --> .a for the vaible prefix E+T*. There are no other valid items for E+T*.

In summary, the sets of items become the states of a deterministic finite machine M that recognizes the viable prefixes of the grammar. The GOTO operation becomes the state transition function of M.

5. IMPLEMENTATION OF LR(0) ITEM SETS CONSTRUCTION

We use PASCAL programming language to implement item sets construction according to the system defined in section 4. The structure applied here is a semi-network, double- linked list. Using this structure, a large set of items can be constructed without wasting spaces and the "inext" pointer provides a direct acess to the GOTO state . The structure is shown as follows:

```
Type
     ptr = ^network;
     network = record
                  num: integer;
                  mkr: integer;
                  inext, pnext: ptr;
               end;

{num: item set number (state) for "inext" ptr and a
     single item (production) number for "pnext"}
{mkr: position of the mark "."}
{inext: points to the next item set (state)}
{pnext: points to next single item (production)}
```

Note that in section 4 we have defined the inadequate state that causes a conflict. In the implementation, each pointer structure ("^network" in Fig.3) carrying the item sets number (state) is usually filled with a dummy zero in the "mkr" field since the position of mark is only considered in a single item within a set. Whenever a conflict exists, however, that dummy zero is changed into "-1" in order to indicate that the state is an inadequate state.

Codes for CLOSURE operation, GOTO operation and MAIN procedure for item sets construction are as follows:

```
procedure getclosure(nextsym:char; pnum: integer;
                     cureni: ptr; gen:boolean);

{nextsym: grammar symbol following the mark}
{pnum: production number}
{cureni: current item set under processing}
{gen: denote whether more items need to be generated,
      i.e., getting the CLOSURE, or not}

begin {getclosure}
   while pnum <= plast do  {plast: last production no.}
      begin
         if gram[pnum,0] = nextsym
         then
            begin
               addtoset(pnum, 1, cureni, gen);
               if (gram[pnum, 1] <> nextsym) and
                  (nonterm(gram[pnum, 1])
               then
                  getclosure(gram[pnum,1],2,cureni,gen);
            end;
         pnum := pnum + 1;
      end;
   end{getclosure};

{gram: two-dimentional array of characters carrying
       the grammar.  It serves as a dictionary for

{addtoset: procedure functions adding an unduplicated
           item to the set (state)}


procedure get_inext(var adp, cureni: ptr; gen: boolean)
{This procedure functions GOTO operation.  It finds or
 creates the GOTO state for item "adp"}

var
```

```
   item0: ptr {first item set};
   fond: boolean;


begin{get_inext}
   if adp^.mkr > plen[adp^.num] {length of production}
   then
      adp^.inext := nil
   else
      if not gotosame(cureni^.pnext, adp)
      then
         if not cyclic(mainlok, adp)
         then
            if gen
            then
              search first state to get GOTO state
              for items generated
             else
                goto_newstate(lastloc, adp, inum);
end{get_inext};

{gotosame: the GOTO state is the same with a previous
           item in the set}
{cyclic: the core item of one of previous state is
         different with item denoted by "adp" only
         by "mark is one position behind the mark of
         adp"}

{goto_newstate: creates a new state for "adp" to goto}



procedure itemsets(var mainlok:ptr; var inum: integer);
{Main procedure for item sets construction}

var
   iloc, lastloc, ilok, plok: ptr;

begin{itemsets}
   inum := 0; {first state}
   init_ilok(mainlok, inum);
   iloc := mainlok;
   lastloc := mainlok;
   init_item; {get first item set}
   while iloc <> nil do
      begin
         get_next_item_set;
```

```
            iloc := iloc^.inext;
        end;
    end{build_itemset};
```

6. COSTRUCTION OF AN SLR(K) SET OF PARSING TABLES

Recall that an inadequate state is any state containing both completed item and any other item. If the inadequate state in an LR(0) item sets can be solved by determining the lookaheads, i.e., by computing the FOLLOW sets, for the nonterminal left member of the production in the inadequate state, then we have an SLR(K), where K is the number of lookahead(s), resolution. In this section, we shall concentrate on K = 1.

Definition

Let $G=(V_n, V_t, P, S)$ be a CFG (not necessarily LR(0)). Let S_0 be the LR(0) item sets for G. Let I be any set of items in S_0. Suppose that whenever $A \rightarrow \alpha.\beta$ and $B \rightarrow \gamma.\delta$ are two distinct items in I, one of the following conditions is satisfied:

1. Neither β and δ are λ.
2. $\beta = \lambda$, $\delta \neq \lambda$, and $FOLLOW_k(\beta) \cap EFF_k(\beta FOLLOW_k(A)) = \phi$.
3. $\beta \neq \lambda$, $\delta = \lambda$, and $FOLLOW_k(A) \cap EFF_k(\delta FOLLOW_k(\beta)) = \phi$.
4. $\beta=\delta=\lambda$ and $FOLLOW_k(A) \cap FOLLOW_k(\beta) = \phi$.

Then G is said to be a simple LR(K) grammar [SLR(K)].

Example 5.

Grammar G_0 (example 2) is not SLR(0) because, for example, I_1 contains two items S --> .E and E --> E.+ T and

$$FOLLOW_0(S) = \{\lambda\} = EFF_0[+T\ FOLLOW_0(E)]$$

However, G_0 is SLR(1). To check the SLR(1) condition, it suffies to consider sets of items which

1. Have at least two items, and
2. Have an item with the mark "." at the right-hand end.

Thus, we need to concern ourselves only with inadequate states I_1, I_2, and I_7. For I_1, we observe that $FOLLOW_1(S) = \lambda$ and $EFF_1[+T\ FOLLOW_1(E)] = \{+\}$. Since $\{\lambda\} \cap \{+\} = \phi$, I_1 satisfies condition (3) of the SLR(1) definition. I_2 and I_9 satisfy condition (3) similarly, and so we conclude that G_0 is SLR(1).

Algorithm

Construction of a set of LR(K) tables for an SLR(K) grammar.

Input. An SLR(K) grammar $G = (v_n, V_t, P, S)$ and S_0, the sets of LR(0) items for G.

Output. (τ, T_i), a set of LR(K) tables for G, which we shall call the SLR(K) set of tables for G.

Method. Let I_i be a set of LR(0) items in S_0. The LR(K) table T_i associated with I_i is the pair <f,g> constructed as follows:

1. For all ψ in V_t^{*k},

   a. $f(\psi)$ = shift if $A \rightarrow \alpha.\beta$ is in I_i, $\beta = \lambda$, and ψ is in the set of $EFF_k(\beta\ FOLLOW_k(A))$.

   b. $f(\psi)$ = reduce j if $A \rightarrow \alpha.$ is in I_i, $A \rightarrow \alpha$ is production j in P, and ψ is in $FOLLOW_k(A)$.

c. $f(\lambda)$ = accept if $S' \rightarrow S.$ is in I_i.

d. $f(\psi)$ = error otherwise.

2. For all X in V, g(X) is the table constructed from $GOTO(I_i,X)$.

T_0, the initial table, is the one associated with the set of items containing $S' \rightarrow .S$.

Example 5.

Let us construct the SLR(1) set of tables from the sets of items of Fig.1. We use the name T_i for the table constructed from I_i. We shall consider the construction of T_2 only.

I_2: E --> T.
T --> T.* F

Let T_2 = <f,g>. Since FOLLOW(E) is {+,), λ}, we have f(+) = f[)] = $f(\lambda)$ = reduce 2. Since EFF(*F FOLLOW(T)) = {*}, f(*) = shift. For the other lookaheads, we have f(a) = f[(] = error. For T_2 the only symbol X for which g(X) is defined is X = *. By inspection of Fig.1, it is easy to see that $g(*) = T_7$. The entire set of tables is given in Fig.3.

Example 6.

We attempt to construct an SLR(1) set of tables from the sets of items of Fig.3 in this example. However, when we proceed to the

inadequate state I_3

I_3: A --> c.
E --> c.b

We have FOLLOW(A) $\cap$ EFF(b FOLLOW(E))={a, b}. Following the algorithm to produce a unique parsing action is impossible on the lookaheads a and b. It is therefore not an SLR(1) grammar.

	action						goto							
	a	+	*	(	)		E	T	F	a	+	*	(	)
T0	S	X	X	S	X	X	1	2	3	5	X	X	4	X
T1	X	S	X	X	X	A	X	X	X	X	6	X	X	X
T3	X	3	S	X	3	3	X	X	X	X	X	7	X	X
T3	X	5	5	X	5	5	X	X	X	X	X	X	X	X
T4	S	X	X	S	X	X	X	X	X	X	X	X	X	X
T5	X	7	7	X	7	7	8	2	3	5	X	X	4	X
T6	S	X	X	S	X	X	X	9	3	5	X	X	4	X
T7	S	X	X	S	X	X	X	X	10	5	X	X	4	X
T8	X	S	X	X	S	X	X	X	X	X	6	X	X	11
T9	X	2	S	X	2	2	X	X	X	X	X	7	X	X
T10	X	4	4	X	4	4	X	X	X	X	X	X	X	X
T11	X	6	6	X	6	6	X	X	X	X	X	x	x	x

i in action table: production i;
i in goto table: T_i;
X: error;

Fig.3. SLR(1) parse tables for G_0

Implementation in PASCAL is given as follows:

```
procedure parsetables(var mainlok: ptr; var go_to,
                      action: table; var err: boolean);
{mainlok: carrying entire sets of items}
{action and go_to represent pair of tables <f,g>}
{err: detect syntatic error during processing}

var
   sets, prt:ptr;
   fx: symlit {FOLLOW(X) sets};
   count, i: integer;

begin{parsetables}
   err := false;
   init_table(go_to, action);
   sets := mainlok;
   while sets <> nil do
      begin
         prt := sets^.pnext;
         while (prt <> nil) and (not err) do
            begin
               with prt^ do
                  if inext <> nil
                  {incompleted item, GOTO
                   transition demand}
                  then
                     begin
                        go_to[sets^.num, gram[num,mkr]]
                        := inext^. num;
                        eff(prt, fx, count);
                        if count > 0 then
                          for i := 1 to count do
                            action[sets^.num, fx[i]
                               := shift;
                     end
                  else
                     if sets^.mkr = conflict
                     {inadequate state}
                     then
                        begin
                           count := 0;
                           followx(gram[num,0],fx,count);
                           slr1(go_to, action, err, sets,
                                prt, fx, count);
                        end
                    else {item has been completed}
```

```
                        begin
                           count := 0;
                          followx(gram[num,0],fx,count);
                           for i := 1 to count do
                             action[sets^.num, fx[i]]
                                := num;
                        end;
                prt := prt^.pnext;
                {process next item  in the set}
             end;
          sets := sets^.inext;
          {process next item set (state)}
       end;
end{parsetable};
```

{procedure slr1 checks if the grammar is slr(1) grammar whenever a conflict exists. If the conflict can be solved by slr(1) method, then fill <f,g> with proper entries; otherwise, an error message is given.}

So far we have shown the implementation for k = 1. As to k > 1, the parsing algorithm would look k symbols ahead, rather than just one, whenever necessary to make a parsing decision.

The techniques are described as follows and applied only for inadequate states with over-lapping simple 1-look-ahead sets associated with complete and incomplete items.

Let A --> ω be a complete item and A --> α.Xβ an incomplete item of a state that one-lookahead is inadequate. The lookahead set for A --> ω is $FOLLOW_k(A)$. For A --> α.Xβ, if X is in V_n, the set is {X}, and otherwise is {Xδ V_t^* | the "on X goto" transition is to a

state Q and δ is in a simple (k-1)-lookahead set associated with some complete or incomplete items from Q}.

Some detailed description can be found in [4]. The implementation is left to interested readers.

7. SLR(K) PARSER

In this section, we shall build an SLR(1) parser, i.e., the driver routine. This parser is implemented using entries in the set of parsing tables constructed in section 7.

```
procedure parser(go_to, action: table; inp: symlist;
                 inplen: integer);
{inp: input stack}
{inplen: length of input string}

type
   statestack = array [0..tablelen] of integer;
   repstack = packed array [0..100] of char;

var
   stat: statstack;
   {stack contains states of parse}
   rep: repstack;
   {replacement stack}
   spt, rpt, ipt: integer;
   {stack pointers for stat, rep and inp respectively}
   nextstate, pnum: integer;
   {pnum: production number}
   err: boolean;
   {error detecting}

begin{parser}
   err := false;
   start; {initiate the parser}
   while (rep[rpt] <> startsym) and (not err) do
      if not endofinp(inplen)  {end of input}
      then
         if action[stat[spt], inp[ipt]] = shift
         then
            begin
               shiftinp; {shift top of inp to rep stack}
               pushstat(go_to[stat[spt], rep[rpt])
               {push the go_to state to stat stack}
            end
         else
            if reduce(action[stat[spt], inp[ipt])
            then
               begin
```

```
                apply(action[stat[spt], inp[ipt])
                if rep[rpt] <> startsym then
                pushstat(go_to[stat[spt], rep[rpt]])
            end
        else
            begin
                error_message;
                err := true;
            end;
end{parser};
```

Example 7.

Given a string ω = a+(a*a) associated with grammar G_0 (Example 2), using the set of tables generated in Fig. 3 to run the parser constructed as above. Trace of the parsing is shown in Fig.4.

states	apply	replace	input	operation
0		λ	a+(a*a)	
0		a	+(a*a)	shift
0 5		a	+(a*a)	goto
0	7 F-->a	F	+(a*a)	reduce
0 3		F	+(a*a)	goto
0	5 T-->F	T	+(a*a)	reduce
0 2		T	+(a*a)	goto
0	3 E-->T	E	+(a*a)	reduce
0 1		E	+(a*a)	goto
0 1		E+	(a*a)	shift
0 1 6		E+	(a*a)	goto
0 1 6		E+(	a*a)	shift
0 1 6 4		E+(	a*a)	goto
0 1 6 4		E+(a	*a)	shift
0 1 6 4 5		E+(a	*a)	goto
0 1 6 4	7 F-->a	E+(F	*a)	reduce
0 1 6 4 3		E+(F	*a)	goto
0 1 6 4	5 T-->F	E+(T	*a)	reduce
0 1 6 4 2		E+(T	*a)	goto
0 1 6 4 2		E+(T*	a)	shift
0 1 6 4 2 7		E+(T*	a)	goto
0 1 6 4 2 7		E+(T*a	)	shift
0 1 6 4 2 7 5		E+(T*a	)	goto
0 1 6 4 2 7	7 F-->a	E+(T*F	)	reduce
0 1 6 4 2 7 10		E+(T*F	)	goto
0 1 6 4	4 T-->T*F	E+(T	)	reduce
0 1 6 4 2		E+(T	)	goto
0 1 6 4	3 E-->T	E+(E	)	reduce
0 1 6 4 8		E+(E	)	goto
0 1 6 4 8		E+(E)	λ	shift
0 1 6 4 8 11		E+(E)	λ	goto
0 1 6	6 F-->(E)	E+F	λ	reduce
0 1 6 3		E+F	λ	goto
0 1 6	5 T-->F	E+T	λ	reduce
0 1 6 9		E+T	λ	goto
0	2 E-->E+T	E	λ	reduce
0 1		E	λ	goto
0	1 S-->E	S	λ	reduce
		accepted		

Fig.4. Trace on parsing w=a+(a*a)

8. EXTENSION TO NON-SLR GRAMMARS

In example 6, we have shown that the FOLLOW sets are not sufficient to resolve parsing action conflicts resulting from inconsistent sets of LR(0) items (for grammar G_2). What should we do with non-SLR grammars? There are several techniques that can be considered before abandoning the LR(0) approach to parser design.

One approach would be to try to use local context to resolve ambiguities. It is called LALR(K) method. Note that LALR grammars include all SLR grammars, but not all LR grammars are LALR grammars. Therefore if this approach is unsuccessful, we might attempt to split one set of items into several. In each of the pieces the local context might result in unique parsing decisions.

Since a thorough discussion is beyond the scope of this paper. We shall end up here. Detail examples and the grammar splitting algorithm can be found in [2].

9. CONCLUSION

The parser-constructing techniques we have implemented is superior to other LR parsing methods both in speed and size of the resulting parser. It is sufficiently powerful to be useful for practical grammars and is the easiest to implement. It works on any SLR(K) grammar. However this method is not powerful enough to solve the inadequate states in every LR grammar. Other algorithms extended from this one would be able to give resolution, but the implementations are more elaborate.

The complete computer program developed to implement this algorithm is filed with Proffessor Samuel L. Gulden of the Division of Computer Science, Lehigh University, Bethlehem, PA.

10. VITA

The auther was born to Mr. and Mrs. Yeong-Fu Tzuu on July 16, 1955 in Taipei, Taiwan, R. O. C. She earned her B. A. in Foreign Literature and Languages from National Chung Hsing University (Taichung, Taiwan) in June 1979. In the fall 1980, she began graduate study in Computing Science at Lehigh University.

BIBILIOGRAPHY

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

2. Aho, A.V., and Ullman, J. D., The Theory of Parsing, Translation, and Compiling, Volume II: Compiling, Addison-Wesley, Reading, Mass., 1978.

3. Barrett, W. A., and Couch, J. D., Compiler Construction Theory and Practice, Science Research Association, Inc., 1979.

4. DeRemer, F. L., "Simple LR(K) Grammars," Comm. ACM 14:7, PP. 453-460.

5. Harrison, M.A., Introduction to Formal Language Theory, Addison-wesley, Reading, Mass., 1978.

6. Ibrahim, L. J., "An Implementation of Graham-Harrison-Ruzzo's LR-Type Parsing Algorithm for Context-Free Languages," Lehigh University, Bethlehem, 1981.