Lehigh University
Lehigh Preserve

Theses and Dissertations

1-1-1983

An implementation of a parsing algorithm for LALR grammars.

Ali Mousa Jaber

Follow this and additional works at: http://preserve.lehigh.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
Jaber, Ali Mousa, "An implementation of a parsing algorithm for LALR grammars." (1983). *Theses and Dissertations.* Paper 2337.

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

AN IMPLEMENTATION OF A PARSING ALGORITHM

FOR LALR GRAMMARS

by

ALI MOUSA JABER

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computing Science

Lehigh University

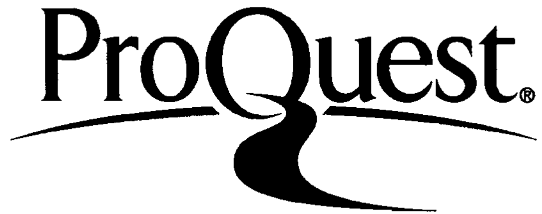
1983

ProQuest Number: EP76613

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.



ProQuest EP76613

Published by ProQuest LLC (2015). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Certificate of Approval

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

May 2, 1983
(date)

Professor in Charge

Chairman of Department

Acknowledgment

The author wishes to thank Professor Samuel L. Gulden for his helpful suggestions and thoughts in preperation of this thesis.

Table of Contents

1. ABSTRACT 1

2. INTRODUCTION 2

3. BACKGROUND 5

3.1 Basic Definitions and Notation 5
- 3.1.1 Alphabets and Strings 5
- 3.1.2 Terminals and Nonterminals 5
- 3.1.3 Production Rules and Grammars 6
- 3.1.4 Languages 7

3.2 LR(k) Grammars 7
- 3.2.1 FIRST set 7
- 3.2.2 Eugmented Grammars 9
- 3.2.3 Definition of LR(k) Grammar 9

4. ANALYSIS OF THE PARSING ALGORITHM 11

4.1 LR(0) Item 11
4.2 LR(1) Item 12
4.3 Sets of LR(1) Items 12
- 4.3.1 CLOUSURE Function 12
- 4.3.2 GOTO Function 13

4.4 Sets of LALR(1) Items 14
4.5 LALR(1) Parsing Table 15
4.6 The Parsing Process 17

5. IMPLEMENTATION OF THE PARSING ALGORITHM 19

5.1 Input and FIRST set 19
- 5.1.1 Procedure GETPROD 20
- 5.1.2 Procedure FIRSTPROD 21
- 5.1.3 Procedure FIND_LASET 23

5.2 Sets of Items 24
- 5.2.1 procedure CLOSURE 25
- 5.2.2 Procedure BUILD_NEWSTATE 27

5.3 Parsing Table 29
5.4 Parsing Process 31

6. CONCLUSIONS 34

7. LIST OF REFERENCES 35

8. VITA 36

1. ABSTRACT

AN IMPLEMENTATION OF A PARSING ALGORITHM FOR LALR GRAMMARS

by Ali M. Jaber

An LALR parsing algorithm presented by Alfred V. Aho, and Jeffrey D. Ullman is presented in this paper. This class of languages is of great importance because the parsing tables obtained are considerably smaller than the LR tables, yet most common syntactic constructs of programming languages can be expressed conveniently by LALR grammar. An introduction about the LR parsing technique and a background on the theory of parsing are given. An analysis of the algorithm is shown.

Logically, the algorithm consists of two parts, the parsing table, and the driver routine. The parsing table is created dynamically using pointers. To implement the concept of the parsing algorithm, a program written in PASCAL is discussed.

2. INTRODUCTION

A grammar forms the underlying method in deriving sets of strings of a formal language. In this paper we shall deal with a special type of grammar known as an LALR grammar which is a subset of a particular class of grammars known as context free grammars (CFG), and we will present an LALR parsing algorithm.

LALR (lookahead_LR) grammars are a subset of LR grammars. An LR parser scans the input from left to right and constructs a rightmost derivation in reverse (hence, the name LR). The LALR parsing algorithm used in this paper is due to Aho, and Ullman.

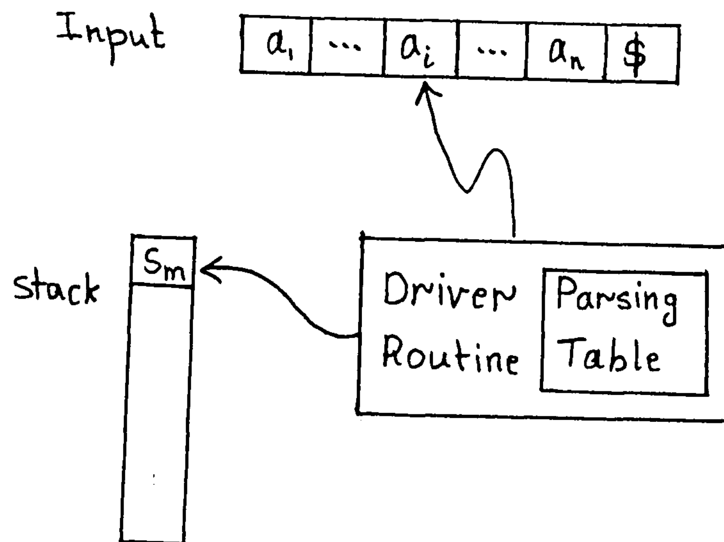


Fig. 2.1 LR parser

Logically, an LR parser consists of two parts, a

driver routine and a parsing table. The driver routine is the same for all LR parsers; only the parsing table changes from one parser to another. The parser has an input, a stack, and a parsing table as shown in Fig. 2.1. The input is read from left to right, one symbol at a time. The stack contains a string of the form $s_0 X_1 s_1 \ldots X_m s_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a symbol called a state. The parsing table consists of two parts, a parsing action function ACTION and a goto function GOTO.

The program driving the LR parser behaves as follows. It determines s_m, the state currently on top of the stack, and a_i, the current input symbol. It then consults ACTION[s_m,a_i], the parsing action table entry for state s_m and input a_i. The entry ACTION[s_m,a_i] can have one of four values:

1. shift s
2. reduce $A \rightarrow \beta$
3. accept
4. error

The function GOTO takes a state and a grammar symbol as argument and produces a state. It is essentially the transition table of a deterministic finite automaton whose input symbols are the terminals and nonterminals of

the grammar.

LR parsers can be constructed to recognize virtually all programming language constructs for which context free grammars can be written.

3. BACKGROUND

3.1 Basic Definitions and Notation

3.1.1 Alphabets and Strings

An alphabet is any set of symbols. An alphabet need not be finite or even countable, but for all practical applications our alphabets will be finite. The symbol Σ is used to designate an alphabet.

A string is a sequence of elements drawn from an alphabet. For example, 01011 is a string over the binary alphabet {0,1}. The empty string is denoted by λ.

DEFINITION We formally define strings over an alphabet Σ in the following manner:

1. λ is a string over Σ.
2. If x is a string over Σ and a is in Σ, then xa is a string over Σ.
3. y is a string over Σ if and only if it is being so follows from (1) and (2).

3.1.2 Terminals and Nonterminals

A terminal is a member of Σ. In order to define structural rules for a grammar we introduce a finite set of objects called nonterminals. The set of nonterminals is disjoint from Σ, and is usually denoted by N.

3.1.3 Production Rules and Grammars

A production rule, or production for short, has the general form X--->Y where X and Y are strings in the terminal and nonterminal sets of a given grammar. The Y may be an empty string, but X cannot be. Productions are used to generate strings in the language.

A grammar is a 4_tuple : G = (N, $\sum$, P, S), where :

N is a finite nonempty set of nonterminals ;

$\sum$ is a finite set of alphabet ;

P is a finite set of productions of the form X--->Y, Where X and Y are in $(N \cup \sum)^*$, and X contains at least one element in N ;

S is called the start symbol ;

We represent a derivation step by the symbol ===> ,a sequence of one or more derivation steps by $\overset{+}{===>}$,and a sequence of zero or more derivation steps by $\overset{*}{===>}$.

DEFINITION A context free grammar (CGF) is denoted by G=(N, $\sum$, P, S), where N, $\sum$,P , and S as defined ab above ; each production is of the form A--->W, where A is a member of of N and W is a string of symbols from $(N \cup \sum)^*$.

An example of a context free grammar is given below :

N={E,T,F}, $\sum$ ={+,*,(,),a}, S=E, and P= the set :

1. E ---> E+T

2. E ---> T

3. T ---> T*F

4. T ---> F

5. F ---> (E)

6. F ---> a

3.1.4 Languages

DEFINITION Let $G=(N, \Sigma, P, S)$ be a grammar. The language generated by G, written L(G), is the set :

$$L(G)=\{w \in \Sigma^* / S \overset{*}{\Longrightarrow} w\}$$

Example : let $G=(N, \Sigma, P, S)$ where :
$N=\{A,C\}$, $\Sigma=\{c,d\}$, S=A, and P=the set:
A ---> CC
C ---> cC \ d

Then, $L(G) = c^*dc^*d$.

3.2 LR(k) Grammars

Before we define the LR grammar, we need to define a useful function $FIRST_k(W)$.

3.2.1 FIRST set

The domain of $FIRST_k$ is some string w in $(N \cup \Sigma)^*$. The function is defined as follows:

$$FIRST_k(W)=\{x \,/ W \overset{*}{==>} xy \;, x,y \in \Sigma^*\}$$

$$|x| = k \text{ if } y \neq \lambda$$
$$|x| \leq k \text{ if } y = \lambda$$

Where the derivations are left_most. That is, $FIRST_k(W)$ for some string W is the set of all leading terminal strings of length k or less in the strings derivable from W.

To compute the FIRST set we can use the following rules:

1. $FIRST_k(aW)=a\,FIRST_{k-1}(W)$ for any string W $, a \in \Sigma$.
2. $FIRST_k(\lambda)= \{\lambda\}$, λ being the empty string.
3. $FIRST_k(XY)=FIRST_k(FIRST_k(X)FIRST(Y))$ $=FIRST_k(X\;\; FIRST_k(Y))=FIRST_k(FIRST_k(X)\;\; Y)$ $X,Y \in \;\; \{N \cup \Sigma\}^*$.
4. Given a production A ---> W in G , $FIRST_k(A)$ contains $FIRST_k(W)$, $A \in N$. $W \in \{N \cup \Sigma\}^*$.

Consider the example in section 3.1.3 , then

$FIRST_1(E) = \{(,a\}$
$FIRST_1(T) = \{(,a\}$
$FIRST_1(F) = \{(,a\}$

For simplicity, we will use FIRST(W) for $FIRST_1(W)$.

3.2.2 Eugmented Grammars

DEFINITION Let $G=(N,\Sigma,P,S)$ be a CFG. We define the augmented grammar derived from G as $G'=(N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$.

The augmented grammar G' is merely G with a new starting production S'--->S, where S' is a new start symbol, not in N. We assume S'--->S is the zeroth production in G' and that the other productions of G are numbered 1,2,3,...,p. We add the starting production so that when a reduction using the zeroth production is called for, we can interpret this "reduction" as a signal to accept.

3.2.3 Definition of LR(k) Grammar

DEFINITION Let $G=(N,\Sigma,P,S)$ be a CFG and let $G'=(N',\Sigma,P',S')$ be its augmented grammar. We say that G is LR(k), $k \geq 0$, if the three conditions

1. $S' \overset{*}{\underset{G'\ rm}{\Longrightarrow}} \alpha AW \underset{G'\ rm}{\Longrightarrow} \alpha \beta W$
2. $S' \overset{*}{\underset{G'\ rm}{\Longrightarrow}} \gamma BX \Longrightarrow \alpha \beta Y$, and
3. $FIRST_k(W) = FIRST_k(Y)$

imply that $\alpha AY = \gamma BX$. (That is, $\alpha = \gamma$, A=B, and X=Y). A grammar is LR if it is LR(k) for some k.

Intuitively this definition says that if $\alpha \beta W$ and

$\alpha\beta Y$ are right sentential forms of the augmented grammar with $FIRST_k(W) = FIRST_k(Y)$ and if $A \rightarrow \beta$ is the last production used to derive $\alpha\beta W$ in a rightmost derivation, then $A \rightarrow \beta$ must also be used to reduce $\alpha\beta Y$ to αAY in a right parse. Since A can derive β independently of W, the LR(k) condition says that there is sufficient information in $FIRST_k(W)$ to determine that $\alpha\beta$ was derived from αA.

4. ANALYSIS OF THE PARSING ALGORITHM

4.1 LR(0) Item

DEFINITION An LR(0) item of a grammar G is a production of G with a dot at some position of the right side of the production. For example, production A--->XYZ generates the four items :

A ---> .XYZ
A ---> X.YZ
A ---> XY.Z
A ---> XYZ.

The production A--->λ (λ empty string) generates only one item A--->λ. .

Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. For example, the first item above would indicate that we are expecting to see a string derivable from XYZ next on the input. The second item would indicate that we have just seen on the input a string derivable from X and we next expect to see a string derivable from YZ.

4.2 LR(1) Item

DEFINITION An LR(1) item of a grammar G is an LR(0) item with a lookahead symbol. The general form of an item is $[A \rightarrow \alpha . \beta, a]$, where $A \rightarrow \alpha\beta$ is a production and a is a terminal or the right end marker \$. $A \rightarrow \alpha . \beta$ is called the core of the LR(1) item.

The lookahead has no effect in an item of the form $[A \rightarrow \alpha . \beta, a]$, where β is not λ, but an item of the form $[A \rightarrow \alpha ., a]$ calls for reduction by $A \rightarrow \alpha$ only if the next input symbol is a .

4.3 Sets of LR(1) Items

Let G' be the augmented grammar for G, to construct the sets of LR(1) items, we need to define two functions, CLOUSURE and GOTO.

4.3.1 CLOUSURE Function

If I is a set of items for a grammar G, then the set of items CLOUSURE(I) is constructed from I by the rules:

1. Every item in I is in CLOUSURE(I).
2. If $[A \rightarrow \alpha . B\beta, a]$ is in CLOUSURE(I) and $B \rightarrow \gamma$ is a production, then add the item $[B \rightarrow .\gamma, b]$ where b in FIRST(βa), if it is not already there.

Intuitively, $[A \rightarrow \alpha . B\beta, a]$ in CLOUSURE(I) indicates that, at some point in the parsing process, we next

expect to see a string derivable from Bβ as input. If B--->γ is a production, we would also expect to see a string derivable from γ at this point. It is for this reason we also include [B--->.γ ,b] in CLOUSURE(I). The procedure CLOUSURE is shown in Fig. 4.1 below:

```
procedure COUSURE(I);

 BEGIN
  REPEAT
   FOR each item [A--->α.Bβ,a] in I, each production
      B--->γ ,and each terminal b in FIRST( βa) such
      that [B--->.γ ,b] is not in I
   DO  add [B--->.γ ,b] to I ;
  UNTIL no more items can be added to I ;
  RETURN I ;
 END;
```

Fig. 4.1

4.3.2 GOTO Function

The function GOTO(I,X) where I is a set of items and X is a grammar symbol is defined to be the clousure of the set of all items [A--->αX.β ,a] such that [A--->α.Xβ,a] is in I. The procedure GOTO is shown in Fig. 4.2 below:

```
procedure GOTO(I,X);

 BEGIN
  let J be the set of items [A--->αX.β ,a],
  such that [A--->α.Xβ,a] is in I ;
  RETURN CLOUSURE(J) ;
 END ;
```

Fig. 4.2

To construct the sets of items, we begin by computing the clousure of {[S'--->.S,$]}. We match the item [S'--->.S,$] with the item [A---> α.Bβ,a] in the procedure clousure. That is, A=S', $\alpha=\lambda$, B=S, $\beta=\lambda$, and a=$. CLOUSURE tells us to add [B--->.γ,b] for each production B--->γ and terminal b in FIRST(βa). The main procedure for constructing the sets of items is shown in Fig. 4.3 below:

```
procedure MAIN ;

 BEGIN
  C := {CLOUSURE({S'--->.S,$})} ;
  REPEAT
  FOR each set of items I in C and each grammar
      symbol X such that GOTO(I,X) is not empty
      and not already in C
  DO add GOTO(I,X) to C ;
  UNTIL no more sets of items can be added to C ;
 END ;
```

Fig. 4.3

4.4 Sets of LALR(1) Items

We are now prepared to give our LALR(1) sets of items construction algorithm. The general idea is to construct the sets of LR(1) items and merge the sets having the same core.

ALGORITHM 4.1
----

input : An augmented grammar G' for the grammar G.

output: The sets of LALR(1) items.

method:

1. Construct $C=\{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items.

2. For each core present among the sets of LR(1) items, find all sets having that core, and replace these sets by their union.

3. The GOTO is constructed as follows: If J is the union of one or more sets of LR(1) items, i.e., $J=I_1 \cup I_2 \cup \ldots \cup I_m$, then the cores of $GOTO(I_1,X)$, $GOTO(I_2,X)$, ..., $GOTO(I_k,X)$ are the same, since $I_1, I_2, \ldots, I_k$ all have the same core. Let K be the union of all sets of items having the same core as $GOTO(I_1,X)$, then GOTO(J,X)=K.

4.5 LALR(1) Parsing Table

We now give the rules whereby the LALR(1) parsing action and goto functions are constructed from the sets of LALR(1) items.

ALGORITHM 4.2

Construction of a canonical LALR(1) parsing table :

input : an augmented grammar G' for the grammar G .

output: If possible, the LALR(1) parsing table, i.e., the action function ACTION and goto function GOTO .

method:

1. construct $C=\{I_0, I_1, \ldots, I_n\}$, the collection of sets of LALR(1) items for G.

2. State i of the parser is constructed from I_i, the parsing actions for state i are determined as follows:

   - If $[A \text{---}> \alpha.a\beta, b]$ is in I_i and $GOTO(I_i, a)=I_j$, then set ACTION[i,a] to "shift j."

   - If $[A \text{---}> \alpha., a]$ is in I_i, then set ACTION[i,a] to "reduce $A \text{---}> \alpha$."

   - If $[S' \text{---}> S., \$]$ is in I_i, then set ACTION[i,$] to "accept."

   If a conflict results from the above rules, the grammar is said not to be LALR(1), and the algorithm is said to fail.

3. The goto transitions for state i are determined as follows: If $GOTO(I_i, A)=I_j$, then GOTO[i,A]=j.

4. All entries not defined by rules (2) through(3) are made "error."

5. The initial state of the parser is the one constructed from the set containing item $[S' \text{---}> .S, \$]$.

The table formed by Algorithm 4.7 is called the LALR(1) parsing table. If there is no parsing action conflict, then the given grammar G is called an LALR(1) grammar.

4.6 The Parsing Process

As mentioned earlier, the parser has an input, a stack, and a parsing table. The parsing table consists of two parts, a parsing action function ACTION and a goto function GOTO.

A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input:

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m , a_i a_{i+1} \ldots a_n \$)$$

The next move of the parser is determined by reading a_i, the current input symbol, and s_m, the state on top of the stack, and by consulting the parsing action table entry ACTION[s_m, a_i]. The configuration resulting after each of the four types of moves are as follows:

1. If ACTION[s_m, a_i]=shift s, the parser executes a shift move, entering the configuration

   $$(s_0 X_1 S_1 X_2 S_2 \ldots X_m s_m a_i s , a_{i+1} \ldots a_n \$)$$

   Here the parser has shifted the current input symbol a_i and the next state s=GOTO[s_m, a_i] onto the stack; a_{i+1} becomes the new current input symbol.

2. If $ACTION[s_m, a_i]$=reduce $A \rightarrow \beta$, then the parser executes a reduce move, entering the configuration

$$(s_0 X_1 S_1 X_2 s_2 \ldots X_{m-r} s_{m-r} A\ s, a_i a_{i+1} \ldots a_n \$)$$

Where $s=GOTO[s_{m-r}, A]$ and r is the length of β. Here the parser first popped 2r symbols off the stack, exposing state s_{m-r}. The parser then pushes both A, the left side of the production, and s, the entry for $ACTION[s_{m-r}, A]$, onto the stack. The current symbol is not changed in a reduce move.

3. If $ACTION[s_m, a_i]$=accept, parsing is completed.

4. If $ACTION[s_m, a_i]$=error, the parser has discovered an error and calls the error procedure.

The parsing is very simple. Initially, the LALR parser is in the configuration $(s_0, a_1 a_2 \ldots a_n \$)$ where s_0 is a designated initial state and $a_1 a_2 \ldots a_n$ is the string to be parsed. The parser executes moves until an accept or error action is encountered.

5. IMPLEMENTATION OF THE PARSING ALGORITHM

The implementation of the algorithm is written in PASCAL. Throughout the discussion of the program, we shall use the following LALR(1) grammar G as an example:

1. S ---> CC
2. C ---> cC
3. C ---> d

The program is divided into four parts, input and FIRST set, sets of items, parsing table, and the parsing process. We will discuss these parts in this chapter.

5.1 Input and FIRST set

The data structure employed in this part is as follows:

```
TSET = SET OF TERMINALS;
NSET = SET OF NONTERMINALS;
RIGHTSTRING = ARRAY [1.. RIGHTPART] OF
GRRANGE;
PRODUCTION = RECORD
                LEFT: NONTERMINALS;
                RIGHT: RIGHTSTRING;
                LN: INTEGER;
              END;
GRAMMAR = RECORD
             AR: ARRAY [0.. NOPROD] OF
             PRODUCTION;
             LENGTH: INTEGER;
           END;
FIRSTSETS = ARRAY [NONTERMINALS] OF
TSET;
```

There are three important procedures in this section.

5.1.1 Procedure GETPROD

Procedure GETPROD is to read the productions from the input file, it will designate the first symbol as the start symbol S, and it will form the augment grammar for the grammar read by adding a new production S'--->S, i.e., production number 0. The new grammar G' is:

1. S' ---> S
2. S ---> CC
3. C ---> cC
4. C ---> d

The nonterminals will be those symbols formed on the left side of the productions and are represented by the set NST. LAMBDASET set holds the nonterminals that produce the empty string λ. Finally, the productions are stored in an array represented by the variable GR from type GRAMMAR. Procedure GETPROD is as follows:

```
PROCEDURE GETPROD;

 VAR
  N: GRRANGE;
  I, J: INTEGER;
  CH: CHAR;

 BEGIN
  I := 0;
  READ(INPUT, CH);
  CHAR_TO_NUM(CH, N);
  GR. AR[I]. LEFT := 70;
  GR. AR[I]. RIGHT[1] := N;
  GR. AR[I]. LN := 1;
```

```
WHILE NOT EOF(INPUT) DO
 BEGIN
  READ(INPUT, CH);
  I := I + 1;
  J := 0;
  GR. AR[I]. LEFT := N;
  NST := NST + [N];
  WHILE NOT EOLN(INPUT) DO
   BEGIN
    READ(INPUT, CH);
    CHAR_TO_NUM(CH, N);
    J := J + 1;
    GR. AR[I]. RIGHT[J] := N;
   END;
  GR. AR[I]. LN := J;
  READLN(INPUT);
  READ(INPUT, CH);
  IF NOT EOF(INPUT)  THEN
   CHAR_TO_NUM(CH, N);
  IF GR. AR[I]. RIGHT[1] = 0  THEN
   LAMBDASET := LAMBDASET + [GR. AR[I
   ]. LEFT];
 END;
GR. LENGTH := I;
END {GETPROD};
```

5.1.2 Procedure FIRSTPROD

To find FIRST(A), where A is a nonterminal, we need procedure FIRSTPROD. This procedure uses the properties of the FIRST function to find FIRST(A). If the first symbol of the production's right side is a terminal, then this symbol is in FIRST(A). Otherwise, if it is a nonterminal and it is not in LAMBDASET then FIRST(A) includes FIRST(this symbol), else procedure FIRSTPROD is called again recurrsively with the same production but ignoring the first symbol in the right side.

Example:

Let P= A--->E+T , E in LAMBDASET, then
afer calling FIRSTPROD(P) , FIRSTPROD
is called again recurrsively with a new
parameter P'= A--->+T .

To find the FIRST set for all tne nonterminals, we repeat procedure FIRSTPROD(A) until no more terminals are added to the FIRST set. This is accomplished through procedure GETFIRST. Procedures FIRSTPROD and GETFIRST are as follows:

```
 PROCEDURE FIRSTPROD(PROD: PRODUCTION);

  VAR
   TEMPPROD: PRODUCTION;
   LP: NONTERMINALS;
   I: 1.. RIGHTPART;

  BEGIN
   LP := PROD. LEFT;
   IF PROD. RIGHT[1] <= TULIMIT
   THEN
    FR[LP] := FR[LP] + [PROD. RIGHT[1]]
   ELSE
    BEGIN
     IF PROD. RIGHT[1] IN LAMBDASET
     THEN
      BEGIN
       IF PROD. LN = 1
       THEN
        FR[LP] := FR[LP] + FR[PROD.
        RIGHT[1]]
       ELSE
        BEGIN
         FR[LP] := FR[LP] + FR[PROD.
         RIGHT[1]] - [0];
         TEMPPROD. LEFT := PROD. LEFT;
         FOR I := 1 TO (PROD. LN - 1) DO
          TEMPPROD. RIGHT[I] := PROD.
          RIGHT[I + 1];
         TEMPPROD. LN := (PROD. LN - 1);
         FIRSTPROD(TEMPPROD);
```

```
        END;
      END
     ELSE
      FR[LP] := FR[LP] + FR[PROD. RIGHT[
      1]];
    END;
  END {FIRSTPROD};

 PROCEDURE GETFIRST;

  VAR
   TEMP: FIRSTSETS;
   OK: BOOLEAN;
   I, J: 1.. NOPROD;
   NI: NONTERMINALS;

  BEGIN
   J := GR. LENGTH;
   REPEAT
    OK := TRUE;
    TEMP := FR;
    FOR I := 1 TO J DO
     FIRSTPROD(GR. AR[I]);
    NI := NLLIMIT;
    FOR NI := NLLIMIT TO NULIMIT DO
     IF TEMP[NI] <> FP[NI]  THEN
      OK := FALSE;
   UNTIL OK;
  END {GETFIRST};
```

5.1.3 Procedure FIND_LASET

In building sets of LALR(1) items, if [A---> α.Bβ, a] in I ,then for each production B---> γ , and terminal b in FIRST(βa), we add [B--->.γ ,b] to I. Procedure FIND_LASET finds the lookahead set for the LR(0) item B--->.γ . Procedure FIND_LASET is as follows:

```
 PROCEDURE FIND_LASET(S: RIGHTSTRING;
 LAS1: TSET; LIN: INTEGER; VAR LAS2:
 TSET);

  VAR
```

```
    TEMPS: RIGHTSTRING;
    I: 1.. RIGHTPART;
    M: INTEGER;

   BEGIN
    IF S[1] <= TULIMIT
    THEN
     IF S[1] = 0
     THEN
      LAS2 := LAS2 + LAS1
     ELSE
      LAS2 := LAS2 + [S[1]]
    ELSE
     BEGIN
      IF S[1] IN LAMBDASET
      THEN
       IF LIN = 1
       THEN
        LAS2 := LAS2 + (FR[S[1]] - [0]) +
        LAS1
       ELSE
        BEGIN
         LAS2 := LAS2 + FR[S[1]] - [0];
         M := LIN - 1;
         FOR I := 1 TO M DO
          TEMPS[I] := S[I + 1];
         FIND_LASET(TEMPS, LAS1, M, LAS2)
        ;
        END
      ELSE
       LAS2 := LAS2 + FR[S[1]];
     END;
   END (FIND_LASET);
```

5.2 Sets of Items

The data structure emloved in this section is as follows:

```
  ITEM = RECORD
           PRNO: 0.. NOPROD;
           DOT: 0.. RIGHTPART;
           LASET: TSET;
           GOO: 0.. NOSTATES;
         END;
  STATE = RECORD
```

```
            ARY: ARRAY [1.. MAXITEMS] OF
            ITEM;
            STATELEN: INTEGER;
          END;
  ITEMSETS = RECORD
               ARR: ARRAY [0.. NOSTATES]
               OF STATE;
               LEN: INTEGER;
             END;
```

There are two important procedures in this section:

5.2.1 procedure CLOSURE

In chapter 4, an algorithm for constructing procedure CLOUSURE(I) was given . Procedure Closure which we are dscussing in this section is not exactly the same. Given the item [A---> α.B β , a] in I , then procedure CLOSURE only adds the item [B--->.γ ,b] to I if it is not already there.

For example: In the grammar G
given the item [C--->c.C,$], then
procedure CLOSURE adds the items
  [C--->.cC,$]
  [C--->.d ,$]
to state I.

Procedure CLOSURE is as follows:

```
  PROCEDURE CLOSURE(TEM: ITEM; VAR STT:
  STATE);

   VAR
    H, I, J, K, L, M: 0.. MAXITEMS;
    LL: INTEGER;
    NN: NONTERMINALS;
    SS: RIGHTSTRING;
    NOTTHERE: BOOLEAN;
```

```
BEGIN
 J := TEM. DOT + 1;
 K := TEM. PRNO;
 IF GR. AR[K]. LN < (J + 1)
 THEN
  BEGIN
   SS[1] := 0;
   LL := 1;
  END
 ELSE
  BEGIN
   FOR H := (J + 1) TO GR. AR[K]. LN
   DO
    BEGIN
     M := H - J;
     SS[M] := GR. AR[K]. RIGHT[H];
    END;
   LL := (GR. AR[K]. LN - J);
  END;
 NN := GR. AR[K]. RIGHT[J];
 FOR I := 1 TO GR. LENGTH DO
  IF GR. AR[I]. LEFT = NN
  THEN
   BEGIN
    NOTTHERE := TRUE;
    FOR L := 1 TO ITEMCOUNT DO
     IF (STT. ARY[L]. PRNO = I) AND (
     (STT. ARY[L]. DOT = 0) OR (GR.
     AR[I]. RIGHT[1] = 0))
     THEN
      BEGIN
       FIND_LASET(SS, TEM. LASET, LL,
       STT. ARY[L]. LASET);
       NOTTHERE := FALSE;
      END;
    IF NOTTHERE
    THEN
     BEGIN
      ITEMCOUNT := ITEMCOUNT + 1;
      WITH STT. ARY[ITEMCOUNT] DO
       BEGIN
        PRNO := I;
        IF (GR. AR[I]. RIGHT[1] = 0)
        AND (GR. AR[I]. LN = 1)
        THEN
         DOT := 1
        ELSE
         DOT := 0;
```

```
            FIND_LASET(SS, TEM. LASET, LL
            , LASET);
          END;
        END;
      END;
    END {CLOSURE};
```

5.2.2 Procedure BUILD_NEWSTATE

The algorithm for building the sets of LALR(1) items states to build the sets of LR(1) items, then the states with the same core are merged to form a new state. The resulting sets of items are called the sets of LALR(1) items. The number of LALR(1) sets of items(states) is the same as the number of LR(0) sets of items and generally it is much smaller than the number of sets of LR(1) items.

For example, for the following LALR(1) grammar

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

The number of sets of LR(1) items is 22, but that for LALR(1) is 12. For a language like ALGOL, the LR table would have several thousand states compared to several hundred in the case of an LALR language.

This shows how important the LALR parsers are and how space consuming it is to construct LR parsers. If we

build the sets of LR items and then merge the states with identical cores, this will consume a lot of space. To avoid this, during the process of building a new state, if already there is a state with identical cores, the new state is merged immediately. By this means we save a lot of space in the process of building the LALR(1) sets of items. All this is done in procedure BUILD_NEWSTATE.

Example:

For the grammar G ,the sets of LALR(1)
items are listed below:

```
0  : S'--->.S    ,{$}       1
     S--->.CC    ,{$}       2
     C--->.cC    ,{c,d}     3
     C--->.d     ,{c,d}     4

1  : S'--->S.    ,{$}

2  : S--->C.C    ,{$}       5
     C--->.cC    ,{$}       3
     C--->.d     ,{$}       4

3  : C--->c.C    ,{c,d,$}   6
     C--->.cC    ,{c,d,$}   3
     C--->.d     ,{c,d,$}   4

4  : C--->d.     ,{c,d,$}

5  : S--->CC.    ,{$}

6  : C--->cC.    ,{c,d,$}
```

5.3 Parsing Table

The data structure employed in this section ia as follows:

```
ACTION =
 (S, R, A, G);
ENTRYPTR = ^. PTR3;
PTR3 = RECORD
         ACT: ACTION;
         N: INTEGER;
       END;
SYMPTR = ^. PTR2;
PTR2 = RECORD
         SYM: GRRANGE;
         NEXT: SYMPTR;
         ENTRY: ENTRYPTR;
       END;
STATEPTR = ^. PTR1;
PTR1 = RECORD
         ST: INTEGER;
         NXT: STATEPTR;
         FIRST: SYMPTR;
       END;
```

The direct way to construct the parsing table is to represent it by a matrix (two dimensional array). By this method it is faster to access the entries in the table, but it is space consuming because most of the table entries are error entries. The other way is to build the table dynamically using pointers as indicated above in the data structure. Here a lot of space is saved, but a longer access time is obtained. Fig. 5.1 shows the LALR(1) parsing table for the grammar G:

	Action			Goto	
State	c	d	$	S	C
0	s3	s4		1	2
1			accept		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

Fig. 5.1 Parsing table

During the process of building the table , the function SEARCH tells if there s a conflict in a certain state of in of the table, which means that the given grammar is not LALR(1). The function SEARCH is as follows:

```
FUNCTION SEARCH(R: GRRANGE): BOOLEAN;

 VAR
  PT: SYMPTR;
  YES: BOOLEAN;

 BEGIN
  YES := FALSE;
  PT := SYMBEG;
  WHILE PT^.. NEXT <> SYMEND DO
   BEGIN
    PT := PT^. NEXT;
    IF PT^.. SYM = R  THEN
     BEGIN
      YES := TRUE;
```

```
        FLAG := PT;
      END;
    END;
   SEARCH := YES;
  END {SEARCH};
```

5.4 Parsing Process

The data structure employed in this section is as follows:

```
TYPE
 STACKTYPE = ARRAY [1.. 50] OF INTEGER
 ;
 INPUTSTRING = RECORD
                 STRING: ARRAY [1.. 20
                 ] OF CHAR;
                 LENGTH: INTEGER;
               END;
```

In the program, the parsing process is done interactively. The user feeds the computer terminal with a string to be parsed and waits for the parsing result. The main procedure in this section is procedure PARSING. since the entry ACTION[s,a] can have one of four values:

1. shift
2. reduce
3. accept
4. error

We can summerize procedure PARSING by the following pseudo pascal code:

```
BEGIN
 READ input string;
 consult table;
 WHILE action <> accept DO
  BEGIN
   IF action=shift THEN
    call SHIFT
   ELSE
    call REDUCE;
   consult table;
  END;
 print an accepting massage;
END;
```

If an error is encounterd, the program stops and prints an error message. Procedure SHIFT shifts the current input symbol on top of the stack and pushes the new state on top of the stack. Procedure SHIFT is as follows:

```
procedure SHIFT;

 BEGIN
  stackptr:=stackptr+1;
  stack[stackptr]:= current symbol;
  stackptr:=stackptr+1;
  stack[stackptr]:= new state;
 END;
```

If the action is to reduce A--->β, then procedure REDUCE first pops 2r symbols off the stack (r is the length of β), then pushes both A, the left side of the production, and s, the new state onto the stack. Procedure REDUCE can be represented as follows:

```
procedure REDUCE;

 BEGIN
  IF β<>λ (empty string) THEN
```

```
    stackptr:=stackptr-2*len of +1
   ELSE
    stackptr:=stackptr+1;
   stack[stackptr]:=A;
   stackptr:=stackptr+1;
   stack[stackptr]:= new state;
  END;
```

Fig. 5.2 shows the moves of the LALR(1) parser for the grammar G on the input string ccdd:

```
     stack        input
     =====        =====
(1)  0            ccdd$  shift
(2)  0c3           cdd$  shift
(3)  0c3c3          dd$  shift
(4)  0c3c3d4         d$  reduce
(5)  0c3c3C6         d$  reduce
(6)  0c3C6           d$  reduce
(7)  0C2             d$  shift
(8)  0C2d4            $  reduce
(9)  0C2C5            $  reduce
(10) 0S1              $  accept .
```

Fig. 5.2

6. CONCLUSIONS

The parsing algorithm which was analyzed is effective and simple in constructing LALR parsers for this class of context free grammars. In many ways the structure of this algorithm provides more ease and directness in its implementation compared to other LALR parsing algorithm. It is apparent from the algorithm that if a grammar fails to generate the parsing table then it is not LALR(1).

Generally, LR parsers can be constructed to recognize virtually all programming language constructs for which context free grammars can be written.

The complete computer program developed to implement the parsing algorithm was written in the language PASCAL . It is filed with professor Samuel L. Gulden at the division of Computing and Information Science, Lehigh University, Bethlehem, Pennsylvania.

7. LIST OF REFERENCES

1. Aho,A.V ,and Ullman,J.D. , Principles of Compiler Design . Addison_ wesley, 1978.

2. Aho,A.V ,and Ullman,J.D. , The Theory of Parsing, Translation, and Compiling , V1 . Bell Telephone Laboratories,incorporated, and J.D. Ullman , 1972.

3. Harrison,M.A. , Introduction to Formal Language Theory. addison_wesley, 1978.

4. Hopcroft,J.E. , and Ullman ,J.D , Introduction to Automata Theory, Languages, and Computation. Addison_wesley, 1979.

5. Barret,W.A. ,and Couch,J.D. , Compiler Construction : Theory and Practice , Science Research Associates, 1979.

8. VITA

The auther was born to Mr. and Mrs. Mousa Ali Jaber on October 03, 1955 in Jerusalem, Jordan. He earned his B. Sc. in Mathematics from University of Jordan (Amman, Jordan) in June 1977. In the fall 1979, he began graduate study in Mathematics at Lehigh University and earned his MS. degree in Mathematics in June 1981. In the Fall 1981, he began graduate study in Computing Science at Lehigh University . He was a teaching assistant in Mathematics for the Department of Mathematics at Lehigh University.