J. of al-anbar university for pure science : Vol.2 : No.1 : 2008

ISSN: **1991-8941**

DESIGN OF NEW MODEL FOR SLR-PARSER

Essam T. Yassen
College of Computer ,University of Anbar.

ABSTRACT: In last decades the applications of the computerized system were widely used in various environments, such real time systems, monitoring system and other. These applications need live answer from the programmable system. The compiler phases represent the heart of any programming language, therefore if we enhance the compilers; we make the execution more efficient.
In this paper we present new model for SLR-Parser, which is the main stage of the compiler phases, because it responsible for the grammatical checking of the program statements and it needs more time than other stages. The new model appears faster than original parse. Also it is less complexity than original parser. Therefore, it is more efficient to use.

Keywords: Design ,Model , SLR-Parser

Introduction

A compiler is a language translator that takes as input a program written in high level language and produces an equivalent program in a low level language.

For example, a compiler may translate a C++ program into an executable program running on MIPS processor.In the process of translation,a compiler goes through several phases:

- **Lexical Analysis (also called Scanning)**
- **Syntax Analysis (also called Parsing)**
- **Semantic Analysis (also called Type Checking)**
- **Intermediate Code Generation**
- **Code Optimization**
- **Code Generation**

The diagram shows the major components of a typical compiler :The job of lexical analyzer is to is to read the source program on character at time and collect these characters to produce as output a stream of Tokens(Tokens are used to represent low-level program units, such as "Keywords", Identifiers", and may other language symbols).The job of the syntax analyzer is to take a stream of tokens produce by the lexical analyzer and build a Parse Tree(or syntax tree)and determines if sentences in a program are constructed properly according to the rules of the source language. The semantic analyzer's job is to attach some meaning to the structure produced by the parser. After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program, this intermediate representation should have two important properties: it should be easy to produce, and easy to translate into target program. the code optimization phase attempt to improve the intermediate code, so that faster running machine code will result. The final phase of compiler is the code generator,at this point the optimized intermediate form of source program is usually translated to

either assembly language or machine language[1],[2]e.

Syntax Analysis (Parsing)

Syntactical analysis is the process of combining the tokens into well formed expressions, statements, and programs and it checks the syntax error, so it recognizes the legal programs syntactically and it rejects illegal ones[3],[4].

Each language has special rules for structure of program-called grammar or syntax. usually, the grammatical phrases of the source program are represented by a parse tree such as the one shown below[5].[6]:

There are two general categories of parsers[1],[5],[6]:

Top down parsers:

A top down parser, such as LL(1) parsing,move from the goal symbol to astring of terminal symbols.in the terminology of trees,this is moving from the root of the tree to a set of the leaves in

the syntax tree for a program.in using full backup we are willing to attempt to create a syntax tree by following branches until the correct set of terminals is reached.in the worst possible case,that of trying to parse a string which is not in the language,all possible combinations are attempted before the failure to parse is recognized.the nature of top down parsin technique is characterized by:

Recursive-Descent parsing:

The general form of top down parsing called recursive-descent that may involve backtracking,that is,making repeated scans of the input.

Example: consider the grammar

$$S \longrightarrow cAd$$
$$A \longrightarrow ab \mid a$$

if the input string is w=cad then:

LL(1) parser:

In this class of top down parser we can parsed by simply looking at the next symbol in the unparsed input string in order to decide which production is to be applied. this method is deterministic in the sense that no backup is required. therefore, to implement this method we must eliminating left recursion and left factoring from grammar.Example: consider the grammar

$$E \longrightarrow E+T \mid T$$
$$T \longrightarrow T*F \mid F$$
$$F \longrightarrow (E) \mid id$$

Nonterminals	**Input symbol**					
	id	**+**	*****	**(**	**)**	**$**
E	TE'			TE'		
E'		+TE'			ϵ	ϵ
T	FT'			FT'		
T'		ϵ	*FT'		ϵ	ϵ
F	id			(E)		

if the input string is w=id+id*id then:

1. **eliminate left recursion and left factoring**
2. **compute First & Follow**
3. **construct Predictive table,as follow:**
4. **implement LL(1) program:**

stack	Input	output
$E	id+id*id$	E $\longrightarrow$
$E'T	id+id*id$	TE'
$E'T'F	id+id*id$	T $\longrightarrow$
$E'T'id	id+id*id$	FT'
$E'T'	+id*id$	F $\longrightarrow$ id
$E'	+id*id$	
$E'T+	+id*id$	T' $\longrightarrow$ ϵ
$E'T	id*id$	E' $\longrightarrow$
$E'T'F	id*id$	+TE'
$E'T'id	id*id$	
$E'T'	*id$	T $\longrightarrow$
$E'T'F*	*id$	FT'
$E'T'F	id$	F $\longrightarrow$
$E'T'id	id$	id
$E'T'	$	
$E'	$	T' $\longrightarrow$
		*FT'
$	$	
		F $\longrightarrow$
		id
		T' $\longrightarrow$ ϵ
		E' $\longrightarrow$ ϵ
		Accept

Bottom Up parsers

Bottom-up parsing simply proceeds in the reverse order of top down parsing, it starts with the symbols of the input sentence and tries to find production right-hand-sides(substrings of a sentential form) that it can replace with a nonterminal. It proceeds until it reduces to the goal, the start symbol.the bottom up parsin technique include[7]:

Handle Pruning

In this technique keep removing handles, replacing them with corresponding left-hand-sides of production, until we reach S,for example: consider the grammar

E→E+E | E*E | (E) | id

Shift-reduce Parsing

In this section ,we introduce a general style of bottom-up syntax analysis,shift-reduce parsing.this technique attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root.there are actually four possible actions a shift-reduce parser can make[1],[6],[8] :

1. **Shift input symbols from buffer to stack until a handle is formed.**
2. **Reduce handle by replacing gramming symbols at top of stack by l.h.s. of production.**
3. **Accept on successful completion of parse.**
4. **Fail on syntax error.**

<u>Shift-Reduce Parsing Example :</u>

E→E+E | E*E | (E) | id

Stack	Input Buffer	Action
$	a+b*c$	Shift
$a	+b*c$	Reduce: E→id
$E	+b*c$	Shift
$E+	b*c$	Shift
$E+b	*c$	Reduce: E→id
$E+E	*c$	Shift(*)
$E+E*	c$	Shift
$E+E*c	$	Reduce: E→id
$E+E*E	$	Reduce: E→E*E
$E+E	$	Reduce: E→E+E
$E	$	Accept

Conflicts during shift-reduce parsing:

There are contex free grammars for which shift-reduce parsing cannot be used. Ambiguous grammars lead to parsing conflicts.Can fix by rewriting grammar or by making appropriate choice of action during parsing. There are two type of conflicts[9]:

1. **Shift/Reduce conflicts: should we shift or reduce? (See previous example (*))**
2. **Reduce/Reduce conflicts: which production should we reduce with?**

   <u>Example:</u>

   stmt → id(param)
   param → id
   expr → id(expr) | id

Stack	Input Buffer	Action
$...id(id	,id)...$	Reduce by ??

Right-sentential form	Handle	Reducing production
a+b*c	a	E→id
E+b*c	b	E→id
E+E*c	c	E→id
E+E*E	E*E	E→E*E
E+E	E+E	E→E+E
E		

Should we reduce to param or to expr ?

LR Parsing

LR parsers are most general non-backtracking shift-reduce parsers known.

• L stands for "Left-to-right scan of input."

• R stands for "Rightmost derivation (in reverse)."

LR parsing is attractive for a variety of reasons :

1. **LR parsers can be constructed to recognize virtually all programming language constructs for which contex-free grammars can be written.**
2. **the LR parsing method is the most general nonbacktracking shift-reduce parsing method known,yet it can be implemented as efficiently as other shift-reduce methods.**
3. **the class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.**
4. **an LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.**

The schematic form of an LR parser is shown in following figure .it consists of an input,an output,a stack,a driver program,and a parsing table that has two parts (*action* and *goto*)[1],[6],[8].

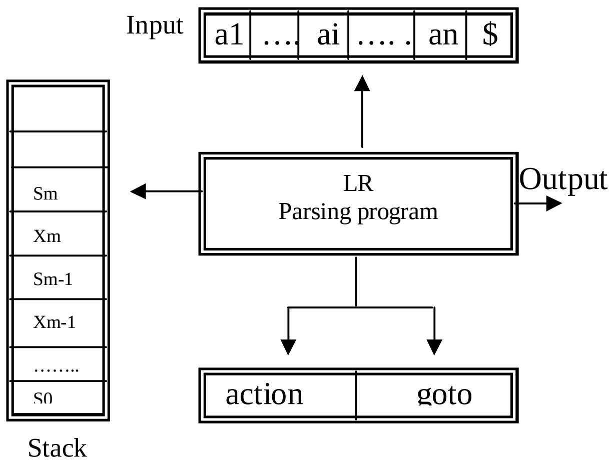


Fig-5- Model of an LR parser

There are three techniques for LR parser depending on the construct of LR parsing table for a grammar :

1. **simple LR parser (SLR for short)**
2. **canonical LR parser**
3. **lookahead LR parser (lalr for short)**

The LR program is the same for all LR parses;only the parsing table changes from one parse to another.The elements in the transition table are labeled with four kinds of actions:

1	**sn**	**Shift into state n**
2	**gn**	**Goto state n**
3	**rk**	**Reduce by rule k**
4	**a**	**Accept**
5		**Error (denoted by a blank entry in the table)**

By using a deterministic finite automaton (DFA), the LR parser know when to shift and when to reduce?the DFA is not applied to the input but to the stack.The edges of the DFA are labeled by the symbols(terminals and nonterminals)that can appear on the stack[1],[5].

The LR parsing algorithm :

The program driving the LR parser behaves as follows.It determines Sm,the state currently on top of the stack,and ai,the current input symbol.It then consults

action[Sm,ai],the transition table action entry , if action is [5]:

- **Shift(n): Advance input one token;push n on stack.**
- **Reduce(k): Pop stack as many times as the number of symbols on the right-hand side of rule k;let X be the left-hand side symbol of rule k;in the state now on top of stack,look up X to get "goto n" ;Push n on top of stack.**
- **Accept: Stop parsing , report success.**
- **Error:Stop parsing, report failure.**

<u>Example</u> : consider grammar

$$\begin{array}{l} \mathbf{E} \longrightarrow \mathbf{T+E} \\ \mathbf{E} \longrightarrow \mathbf{T} \\ \mathbf{T} \longrightarrow \mathbf{x} \end{array}$$

Initially,it will have an empty stack,and the input will be a complete S-sentence followed by $;that is the right-hand side of the S' rule will be on the input.we indicate this as S' $\longrightarrow$. S$ where the dot indicates the current position of the parser. So:

$$\begin{array}{l} \mathbf{0\ S'} \longrightarrow \mathbf{E\$} \\ \mathbf{1\ E} \longrightarrow \mathbf{T+E} \\ \mathbf{2\ E} \longrightarrow \mathbf{T} \\ \mathbf{3\ T} \longrightarrow \mathbf{x} \end{array}$$

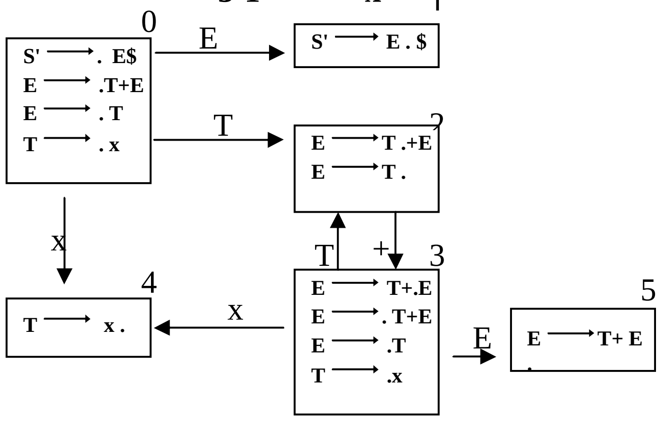


Fig-6- Deterministic finite automaton

Stack	Input	Action
0	x+x $	shift
0S4	+x$	Reduce by **T** $\longrightarrow$ **x**
0T2	+x$	shift
0T2S3	x$	Shift
0T2S3S4	$	Reduce by **T** $\longrightarrow$ **x**
0T2S3T2	$	Reduce by **E** $\longrightarrow$ **T**
0T2S3E5	$	Reduce by **E** $\longrightarrow$ **T+E**
0E1	$	Accept

Fig-8- Parse of input x+x

The New Model for SLR-Parser

The SLR-parser extremely tedious to build by hand, so needs a generator and several specific steps. The following block diagram presents the main stages of the new model for SLR-Parser, which used to parse particular input and give the result depending on the Context Free Grammar build our system that is used for implementing the new model of the SLR-parser:

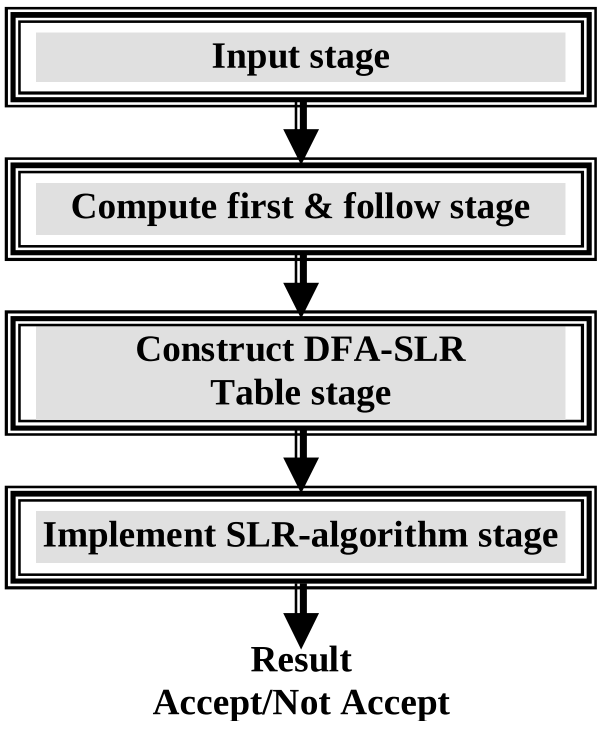

Input stage
Compute first & follow stage
Construct DFA-SLR Table stage
Implement SLR-algorithm stage
Result
Accept/Not Accept

Fig-9-Block diagram of our system

state	x	+	$	E	T
0	S4			g1	g2
1			Accept		
2		S3	r2		
3	S4			g5	g2
4		r3	r3		
5			r1		

Input stage:

In this state the grammar has been reading and the symbols of grammar (terminals and nonterminals) could be specified and give the symbols numerical representation. the advantage of using this method for representation:

1. **Perfect use for storage area, because the numerical representation take fixed size.**
2. **It is more efficient for the parser to work with integer values representing the symbols rather than the actual symbol name(string).**
3. **The numerical representation facilitate to built the SLR-table.**

In this state ,each production of grammar must be on one straight line. Finally,the productions has been numbered.

For example, consider the grammar

$$
\begin{array}{lll}
E \longrightarrow & E+T \mid T & \\
T \longrightarrow & T*F \mid F & \Longrightarrow \\
F \longrightarrow & (\,E\,) \mid id &
\end{array}
\qquad
\begin{array}{l}
1\ E \longrightarrow E+T \\
2\ E \longrightarrow T \\
3\ T \longrightarrow T*F \\
4\ T \longrightarrow F \\
5\ F \longrightarrow (\,E\,) \\
6\ F \longrightarrow id
\end{array}
$$

The numerical representation is:

Symbol name	+	*	(	)	id	E	T	F
Numeric representation	1	2	3	4	5	6	7	8

Compute first & follow stage:

Through this state First & Follow could be detected for each nonteminal.The First should be specified firstly according to the following steps[10]:

1. **If x is terminal, then FIRST(x) is {x}.**
2. **If x $\longrightarrow$ ε is a production ,then add ε to FIRST(x).**
3. **If x is nonterminal and x$\longrightarrow$y1y2 … yk is a production, then place *a* in FIRST(x) if for some *i* ,*a* is in FIRST(yi),and ε is in all of FIRST(y1)… FIRST(yi-1).**

Secondly, the Follow specified according to the following steps:

1. **Place$ in FOLLOW(S),where S is the start symbol.**
2. **If there are a production A $\longrightarrow$ αBß,then everything in FIRST(ß)except for ε is placed in FOLLOW(B).**
3. **If there are a production A$\longrightarrow$ αB ,or a production A$\longrightarrow$ αBß where FIRST(ß) contains ε,then everything in FOLLOW(A) is in FOLLOW(B).**

Construct DFA-SLR table stage

In traditional SLR-parser model this stage implemented in two separate stages, the first one is Construct DFA stage, and the second is Construct SLR-table stage .as follows:

- **Construct DFA stage:**

By using a deterministic finite automaton (DFA)the SLR-parser know when to *shift* and when to *reduce.* the edges of DFA are labeled by symbols of grammar(terminals & nonterminals).In this state, where the input begins with S'(root),that means that it begins with any possible right-hand side of an S-production we indicate that by

$$
\boxed{\begin{array}{l}
S' \longrightarrow .S\$ \\
S \longrightarrow . \\
. \\
.
\end{array}}
$$

Call this state1 or state0,a productions combined with the dot(.) that indicates a position of parser Firstly, for each production in state1 we exam the symbol that occur after dot, there are three cases:

1. **If the symbol is null (the dot has been occurred in the end of right side of production),then there are no new state .**
2. **If the symbol is "$" sign, then there are no new state.**
3. **If the symbol is a terminal or nonterminal, then there are new state, this state start with current production after the dot has been proceeded one step forward. If the symbol has been occurred after the dot(in new position)is nonterminal such as A, then we add all possible right hand side of A to a new state, and so on.**

You must know that any new state must built firstly in a buffer, and we compare it with a previous states in DFA,if there are no similarity situation then the new state is added to DFA and give it a new number equal to number of states in DFA plus one. Finally, we repeat these steps on all new states until the DFA completed.

Algorithm: Construct DFA

Input : BNF grammar

Output: A Deterministic Finite Automaton (DFA)

```
1. S=Initial state          // start with production
                               S' ——→ S$ followed
                               all possible right-
                               hand side of S
2. N=1                      // no. of states in DFA
3. for each production of S
      3.1 a= symbol after dot in
            current production A ——→ ß.a
      3.2 if a is terminal or
                 nonterminal
         3.2.1 Create buffer state start
               with current production
               after proceed dot
               one step A ——→ ßa.
         3.2.2 Append all possible
               right-hand side of
               symbol after dot(in
               new position) into
               buffer state
         3.2.3 Compare buffer state
               with all previous states in
               DFA
         3.2.4 If there are no equal
               case then
            3.2.4.1. Save the number of
                     new state(N+1)and
                     the edge(a)
            3.2.4.2 Append buffer state into
                    DFA
            3.2.4.3 Increment no. of state (N) by
                    one
4.repeate step 3 for all new states
```

- **Construct SLR-table stage:**

The SLR-table is a data structure consist of many rows equal to the number of the states in DFA,also many columns equal to the number of grammar symbols plus "$" sign .As know ,data structure presents fast in information treatment and information retrieve .In this stage SLR-table is constructed .this table had seen as two subtables:

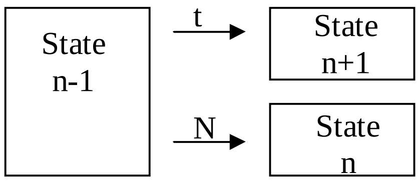
1. **The Action table: consist of many rows equal to number of states in DFA,and many columns equal to number of terminals plus "$" sign(the end of input).**
2. **The Goto table: consist of many rows equal to number of states in DFA,and many columns equal to number of nonterminals.**

the elements(entries) in the SLR-table are labeled with four kinds of actions:

• S_n	**shift into state n**
• g_n	**goto state n**
• r_k	**reduce by production k**

- **a** **accept**
- **error (denoted by blank entry in the table)**

for the construction of this table and the contribution the actions on the tables cells must pass to each state in DFA individually :



- **Shift action & Goto action could be specified according to the edge which has been moved from the current state(n) to the new state.**
  If the edge was terminal symbol (t) then

  Cell[n-1,t]= Sn

  If the edge was nonterminal symbol (N) then

  Cell[n-1,N]= gn
- **If there are production in current state has the form $A \rightarrow ß.$ (the dot in the end of right hand side, ß is any string),then the action is reduce**

  Cell[n-1,f]=rk {f in Follow(A), k is the no. of production}
- **If there are production in current state has the form $A \rightarrow ß.\$$ (the dot occurred before \$ sign, ß is any string),then the action is accept**

  Cell[n-1,\$]=a
- **Finally, any empty cell in row n-1 means error action.**

Repeat the above steps for each state in DFA.

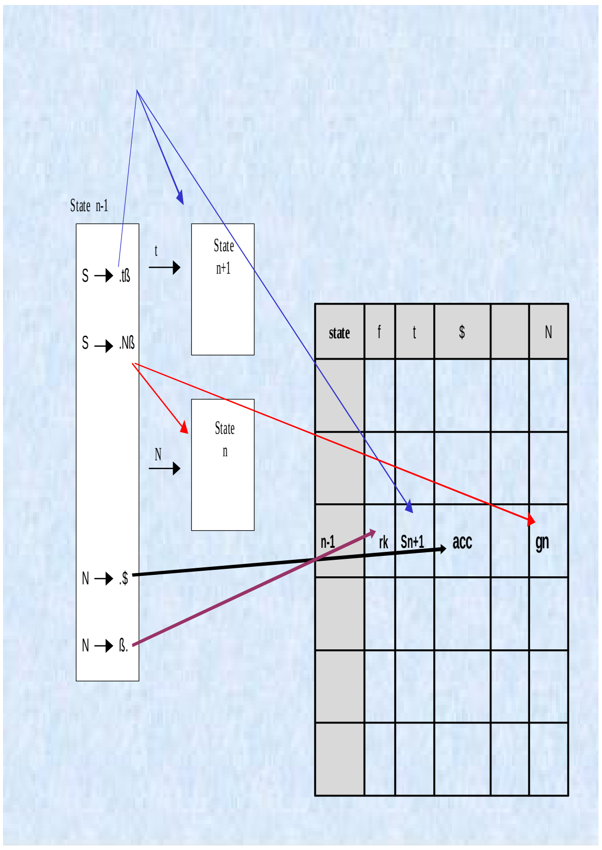
Algorithm: Construct SLR-table

Input : A Deterministic Finite Automaton (DFA)

Output: SLR-table

```
1.For each state in DFA (S)
    1.2 For all production of s
      1.2.1 a= symbol after dot in current production A → ß.a
            // check the class of a
      1.2.2 if a is terminal then
              action of cell[s,a]= shift s'
      1.2.3 if a is nonterminal then
              action of cell[s,a]= goto s'
      1.2.4 if a is ϵ
        1.2.4.1 for all f in follow(A) do
          1.2.4.1.1 action[i,f]=reduce k
                    // k=no. of production
      1.2.5 if a is $ then
              action[i,$]=accept
      1.2.6. else error // denoted by blank
```

through implementing and studying these two stages, it was notice that, we can implement these two stages in one stage (in the same time) instead of two separate stages .that means, while we exam the productions in the current state to specify a new state in DFA,we will specified(immediately) the SLR-table entries(actions).for example, if there is a production $A \rightarrow$ B. ß in current state (n),then we can specify the new state start with the production $A \rightarrow$ Bß. And in the same time we fill sn'(if ß is terminal) or gn'(if ß is nonterminal)in cell [n, ß] of SLR-table. the benefit this work is to decrease the parsing time and consequently to compile time. and the other advantage of this work there is no need to save the edges(output) of current state and its new states because the construction of DFA and SLR-table occurred in the same time.



The suggestion algorithm of DFA and SLR-table is obtained in the following:

Algorithm: Construct DFA-SLR Table

Input : BNF grammar

Output: A Deterministic Finite Automaton (DFA)and SLR-table

```
1. S=Initial state      // start with
      production $S' \longrightarrow . S\$$
      followed      all possible
      right-hand side of S
2. N=1        // no. of states in DFA
3. for each production  of S
   3.1 a= symbol after dot in
         current production $A \longrightarrow ß.a$
   3.2 if a is terminal then
      3.2.1 Create buffer state start
            with current production
            after  proceed  dot one
            step $A \longrightarrow ß a.$
      3.2.2 Append all possible right-
            hand side of symbol after
            dot(in new position) into
            buffer state
      3.2.3 Compare buffer state
            with all previous states in
            DFA
      3.2.4 If there are no equal case
            then
         3.2.4.1 Append buffer state into
                 DFA
         3.2.4.2 Increment no. of state (N) by
                 one
      3.2.5 action of cell[s,a]= shift s'
   3.3 if a is nonterminal then
      3.3.1 Create buffer state start with
            current production after
            proceed   dot  one step
            $A \longrightarrow ß a.$
      3.3.2 Append all possible right-hand
            side of symbol after dot(in new
            position) into buffer state
      3.3.3 Compare buffer state with all
            previous states in DFA
      3.3.4 If there are no equal case then
         3.3.4.1 Append buffer state into DFA
         3.3.4.2 Increment no. of state (N) by
                 one
      3.3.5 action of cell[s,a]= goto s'
   3.4 if a is $\epsilon$
      3.4.1 for all f in follow(A) do
      3.4.2 action[i,f]=reduce k     // k=no.
            of production
   3.5 if a is $ then action[i,$]=accept
   3.6 else error // denoted by blank
4.repeate step 3 for all new states
```

• The Comparison Between New Algorithm and Origin Algorithms Using Computation Complexity:

The computational complexity measure used to measure the complexity of any algorithm and help our to chose the best algorithm for the same problem. In this section we measure the complexity of new algorithm compare to the origin algorithms as shown in [13].

- **The complexity of construct DFA algorithm: this algorithm takes computational complexity as follows:**

T(construct DFA)=n * T(step3)
T(step3)=m* (T(step 3.1)+T(step 3.2))
T(step 3.1)=O(1)
T(step3.2)=O(1)+ T(step3.2.1)+ T(step3.2.2)+ T(step3.2.3) +T(step3.2.4)
T(step3.2.1)=O(1)
T(step3.2.2)=O(m-1)
T(step3.2.3)=O(n-1)
T(step3.2.4)=O(1)+T(step3.2.4.1)+ T(step3.2.4.2)+T(step3.2.4.3)
T(step3.2.4.1)=O(1)
T(step3.2.4.2)=O(1)
T(step3.2.4.3)=O(1)
T(step3.2.4)=O(1)+O(1)+O(1)+O(1)
T(step3.2.4)=O(4)
T(step3.2)=O(1)+O(1)+O(m-1)+ O(n-1)+O(4)=O(m+n+4)
T(step3)=m*(O(1)+O(m+n+4))
T(step3)=m*O(m+n+5)
T(construct DFA)=O(mn*(m+n+5)) **// where n is no. of state , m is no. of productions in current state**

- **The complexity of construct SLR-table algorithm: this algorithm takes computational complexity as follows:**

T(construct SLR-table)=n* T(step1.2)
T(step1.2)=m * (T(step1.2.1)+ max(T(step1.2.2), T(step1.2.3), T(step1.2.4), T(step1.2.5),T(step1.2.6)))
T(step1.2.1)=O(1)
T(step1.2.)=O(2)
T(step1.2.3)=O(2)
T(step1.2.4)=O(1)+ T(step1.2.4.1)
T(step1.2.4.1)=f * T(step1.2.4.1.1)
T(step1.2.4.1.1)=O(1)
T(step1.2.4.1)=f
T(step1.2.4)=O(1)+O(f)=O(f)
T(step1.2.5)=O(2)
T(step1.2.6)=O(2)
T(step1.2)=m* O(1+f)=m
T(construct SLR-table)=O(mn)

// where n is no. of state , m is no. of productions in current state

- **The complexity of new algorithm (construct DFA-SLR table):This algorithm takes computational complexity as follows:**

T(construct DFA-SLR-table)=n * T(step3)
T(step3)=m * (T(step3.1)+ max(T(step3.2), T(step3.3), T(step3.4), T(step3.5), T(step3.6)))
T(step3.1)=O(1)
T(step3.2)=O(1)+T(step3.2.1)+ T(step3.2.2)+T(step3.2.3)+ T(step3.2.4)+ T(step3.2.5)
T(step3.2.1)=O(1)
T(step3.2.2)=O(m-1)
T(step3.2.3)=O(n-1)
T(step3.2.4)=O(1)+ T(step3.2.4.1)+ T(step3.2.4.2)
T(step3.2.4.1)=O(1)
T(step3.2.4.2)=O(1)
T(step3.2.4)=O(3)
T(step3.2.5)=O(1)
T(step3.2)=O(1)+O(1)+O(m-1)+ O(n-1)+O(3)+O(1)=O(m+n+4)
T(step3.3)=O(1)+ T(step3.3.1) + T(step3.3.2) + T(step3.3.3) + T(step3.3.4) + T(step3.3.5)
T(step3.3.1)=O(1)
T(step3.3.2)=O(m-1)
T(step3.3.3)=O(n-1)
T(step3.3.4)=O(1)+ T(step3.3.4.1)+T(step3.3.4.2)
T(step3.3.4.1)=O(1)
T(step3.3.4.2)=O(1)
T(step3.3.4)=O(3)
T(step3.3.5)=O(1)
T(step3.3)=O(1)+O(1)+O(m-1)+ O(n-1)+O(3)+O(1)=O(m+n+4)
T(step3.4)=O(1)+ T(step3.4.1)
T(step3.4.1)=f * T(step3.4.1.1)
T(step3.4.1.1)=O(1)
T(step3.4.1)=O(f)
T(step3.4)=O(1)+O(f)=O(f)
T(step3.5)=O(2)
T(step3.6)=O(2)
T(step3)=m * (O(1)+O(m+n+4))
T(step3)=O(m(m+n+5)

T(construct DFA-SLR-table)=
O(mn(m+n+5)) **// where n is no. of state , m is no. of productions in current state**

The new algorithm perform (construction DFA, construction SLR-table) in *O(mn(m+n+5))* operation, while implement construction DFA algorithm and construction SLR-table algorithm separately take the complexity that equal the summation of complexity for each one:

O(mn*(m+n+5)) + O(mn) = O(mn*(m+n+6))

Therefore, the new algorithm achieve the best use of parsing time.

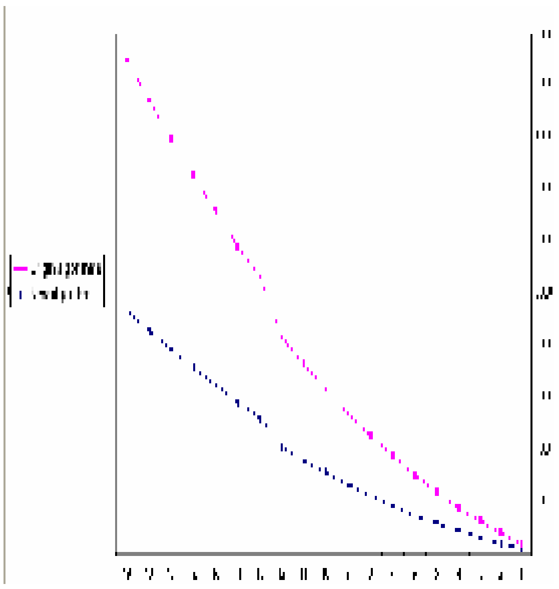
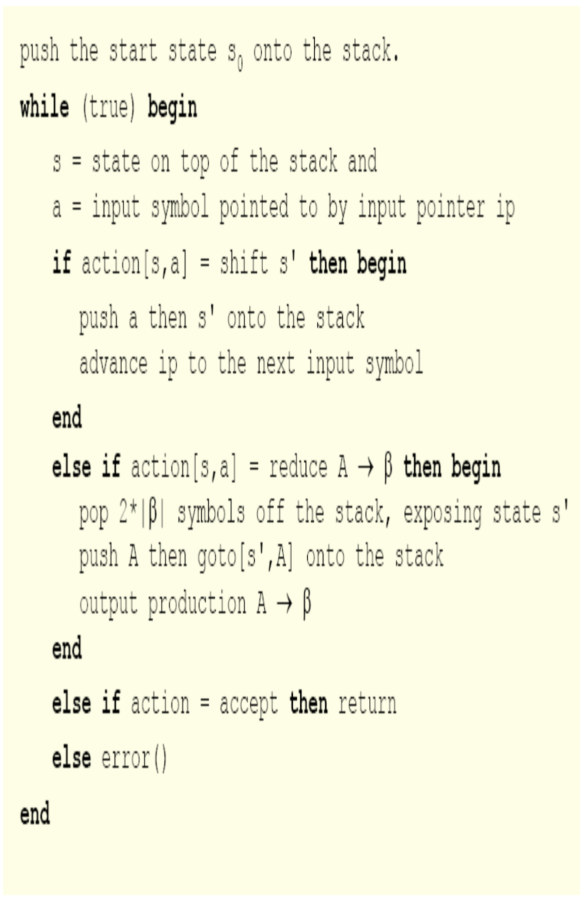


Fig-10- complexity of New Algorithm and Origin Algorithms

Implement SLR-algorithm stage:

In this stage, we implement the SLR-program by using the following algorithm[8] :



```
push the start state $s_0$ onto the stack.
while (true) begin
    s = state on top of the stack and
    a = input symbol pointed to by input pointer ip
    if action[s,a] = shift s' then begin
        push a then s' onto the stack
        advance ip to the next input symbol
    end
    else if action[s,a] = reduce A → β then begin
        pop 2*|β| symbols off the stack, exposing state s'
        push A then goto[s',A] onto the stack
        output production A → β
    end
    else if action = accept then return
    else error()
end
```

Conclusions

Through the studying of the compiler design concept and the compiler phases and implementing the original the SLR parser we are suggested a new model of SLR parser technique and implemented it in Delphi programming language and we conclude the following:

1.The separation of construct DFA stage and construct SLR-table stage is not perfect approach for parsing, therefore, we suggest a new algorithm for constructing DFA a SLR-table in one stage. The suggestion algorithm involves an interaction between construct of DFA and of SLR –table. The suggestion algorithm achieve the best used of parsing time and storage area because the two construction works in the same time.

2.We can use numerical representation (integer) for symbols of grammar instead of symbols name (string), the advantage of using this method is that all

representation numbers are of a fixed size (the best use of storage area). Furthermore, it more efficient for the parser to work with integer values representing the symbols rather than the actual variable-length strings.

3.SLR-parser attempts to match multiple productions at the same time and postpones making a decision until sufficient input has been seen. In contrast, an LL(k) parser must make decisions about which production to match.

4.SLR(k) recognizers are stronger than LL(k) recognizers because the LR strategy uses more context information.

5.SLR-parser treats with many kinds of grammars, because it can over take on the problems that other parsers couldn't made it, such as "*Backtracking*".

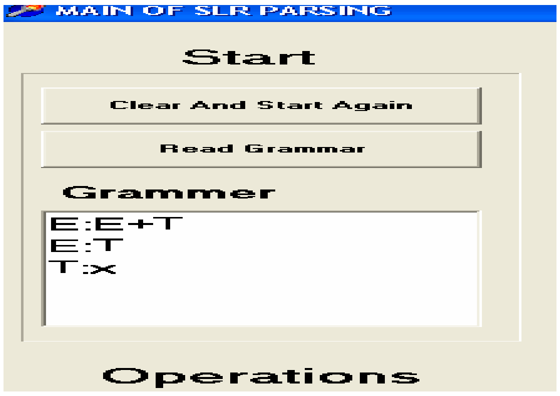
6.SLR-parser directly makes parsing process without getting rid of *Left-Recursion* and *Left-Factoring*.

Appendix : Implement of new model :

For implement the new model, Consider the following grammar

$$E \longrightarrow E+T$$
$$E \longrightarrow T$$
$$T \longrightarrow x$$

First stage: The function of this stage is read the grammar and detect the grammar symbols:


MAIN OF SLR PARSING

Start

Clear And Start Again

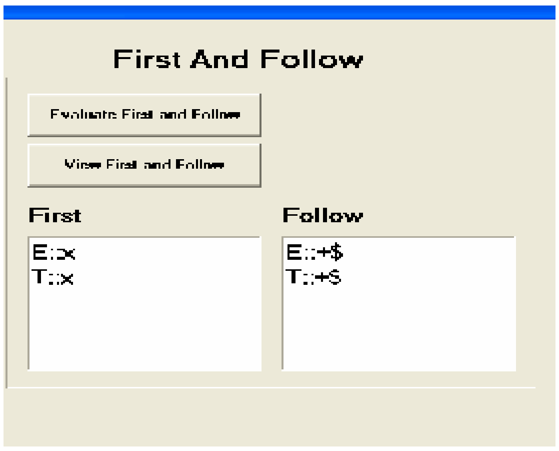
Read Grammar

Grammer

```
E:E+T
E:T
T:x
```

Operations

Second stage: During this stage First and Follow are detected :

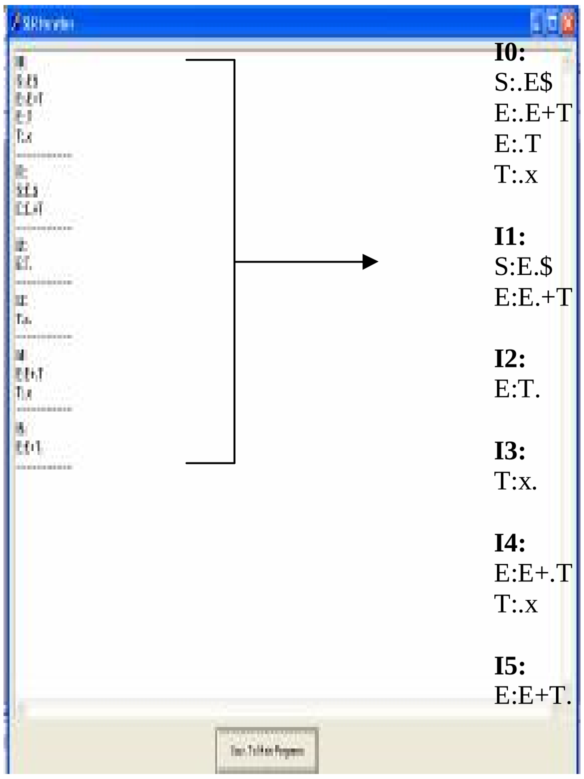

First And Follow

Evaluate First and Follow

View First and Follow

First	Follow
E::x	E::+$
T::x	T::+$

Third stage: The output of this stage is DFA and SLR-table:


I0:
S:.E$
E:.E+T
E:.T
T:.x

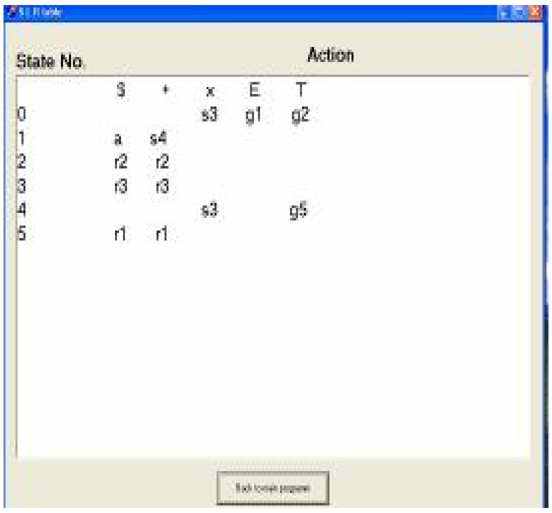
I1:
S:E.$
E:E.+T

I2:
E:T.

I3:
T:x.

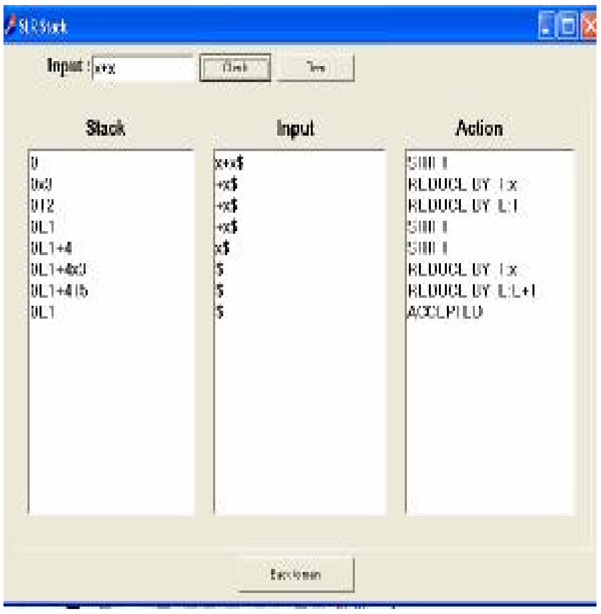
I4:
E:E+.T
T:.x

I5:
E:E+T.



State No.	$	+	x	E	T
0			s3	g1	g2
1	a	s4			
2	r2	r2			
3	r3	r3			
4			s3		g5
5	r1	r1			

Forth stage : suppose input string is *x+x*.After insert input string the SLR-program is executed, as follows:



Stack	Input	Action
0	x+x$	SHIFT
0x3	+x$	REDUCE BY T:x
0T2	+x$	REDUCE BY E:T
0E1	+x$	SHIFT
0E1+4	x$	SHIFT
0E1+4x3	$	REDUCE BY T:x
0E1+4T5	$	REDUCE BY E:E+T
0E1	$	ACCEPTED

REFERENCES

1. **A.Aho,R.Sethi,J.D.Ullman," Compilers-Principles, Techniques and Tools" Addison-Weseley,1986**
2. **E.Taha,"Evaluation and Improvement Code Optimization Methods in Compiler Design",M.Sc.,University of Anbar,2005**
3. **J.Tremblay,P.G.Sorenson ," The Theory and Practice of Compiler Writing ", McGRAW-HILL,1985**
4. **E.Tolman," Language Compiler", university of Chicago, Newton BBS,2002**
5. **A.W.Appel,"Modern Compiler Implementation in ML" ,CambridgeUniversity Press,1998**
6. **http://www.cs.usfca.edu/~parrt/course/652/lectures/LR.parsing.html**
7. **W.M.Waite,L.R.Carter,"An Introduction to Compiler Construction",Harper Collins,New york,1993**
8. **Keith D. Cooper, Ken Kennedy & Linda Torczon," Parsing IV,Bottom-up Parsing", Rice University, 2003**
9. **http://www.inf.ed.ac.uk/teaching/courses/ct/slides/Lecture7.pdf**
10. **http://www.jambe.co.nz/UNI/FirstAndFollowSets.html**
11. **Curt Clifton,"Building Bottom-Up Parsing Tables", Rose-Hulman Institute of Technology**
12. **http://en.wikipedia.org/wiki/SLR_parser**
13. **R.Sedgewick,"Algorithms in ",Addison-wesley,1998**

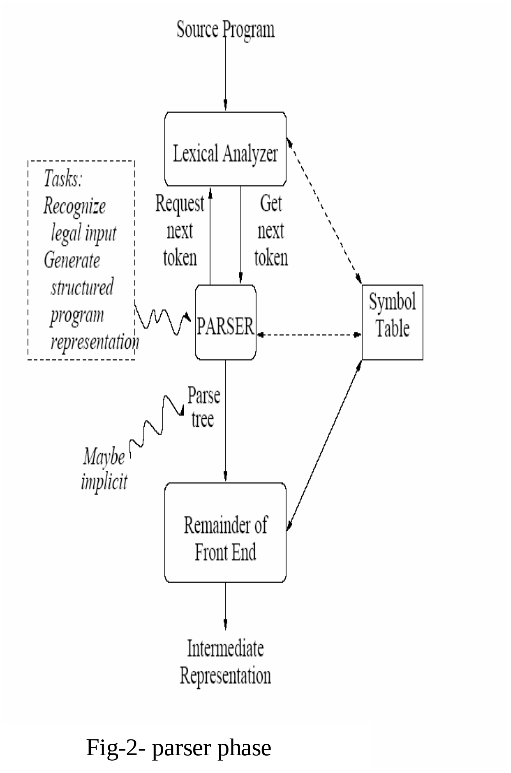


Fig-2- parser phase

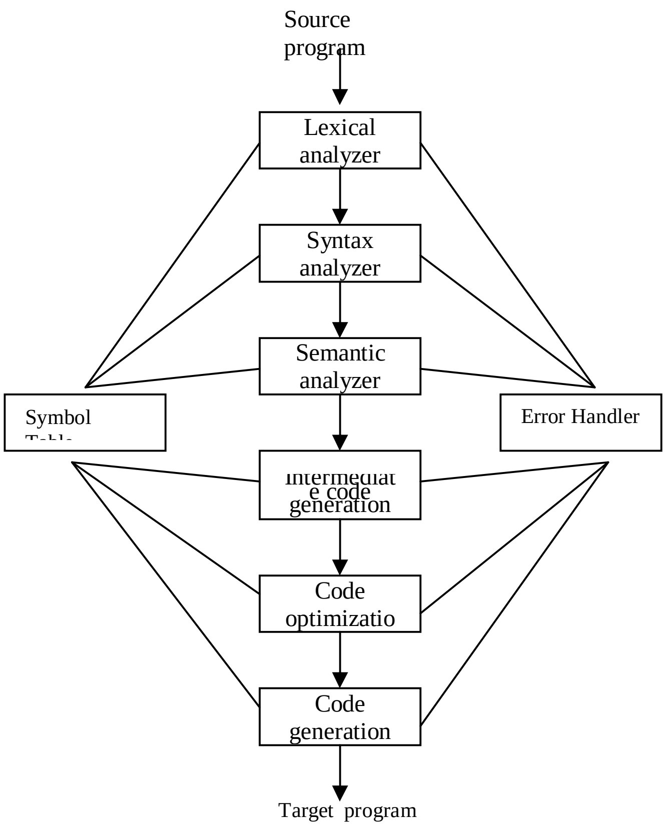


Fig-1- Compiler phases

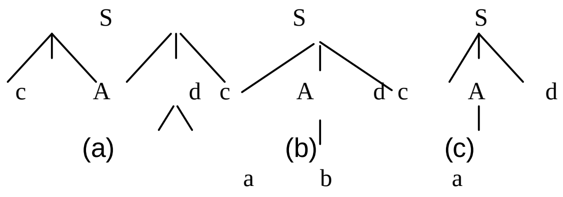


Fig-4- steps in Recursive-

تصميم نموذج جديد للـ SLR-Parser

عصام طه ياسين

E.mail: E_T_972@yahoo.com

الخلاصة:

في العقود الأخيرة ازدادت استخدامات تطبيقات أنظمة الحاسوب في مختلف المجالات مثل أنظمة المراقبة وغيرها من التطبيقات،هذه التطبيقات تحتاج إلى استجابات مباشرة وسريعة من الأنظمة البرمجية.المترجم يمثل قلب أي لغة برمجية لذلك حينما نعزز كفاءة المترجمات سنحصل بالمقابل على تنفيذ أكثر كفاءة للغات البرمجة.

هذا البحث يقدم نموذج جديد للـ SLR-Parser الذي يمثل مرحلة أساسية من مراحل المترجم ،لأنها مسؤولة عن صحة البناء القواعدي للبرنامج المصدر التي تحتاج إلى وقت أكثر من باقي المراحل.النموذج المقترح يبدو أسرع من النموذج الأصلي ويكون اقل تعقيدا منه لذلك يعتبر أكثر كفاءة للاستخدام .