SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – II - Compiler Design – SCSA1604

II. PARSER

Role of Parser-Context-free Grammar – Derivations and Parse Tree - Types of Parser – Bottom Up: Shift Reduce Parsing - Operator Precedence Parsing, SLR parser- Top Down: Recursive Decent Parser - Non-Recursive Decent Parser-Error handling and Recovery in Syntax Analyzer-YACC.

SYNTAX ANALYSIS:

Every programming language has rules that prescribe the syntactic structure of well-formed programs. In Pascal, for example, a program is made out of blocks, a block out of statements, a statement out of expressions, an expression out of tokens, and so on. The syntax of programming language constructs can be described by context-free grammars or BNF (Backus-Naur Form) notation. Grammars offer significant advantages to both language designers and compiler writers.

- A grammar gives a precise, yet easy-to-understand. Syntactic specification of a programming language.
- From certain classes of grammars we can automatically construct an efficient parser that determines if a source program is syntactically well formed. As an additional benefit, the parser construction process can reveal syntactic ambiguities and other difficult-to-parse constructs that might otherwise go undetected in the initial design phase of a language and its compiler.
- A properly designed grammar imparts a structure to a programming language that is useful for the translation of source programs into correct object code and for the detection of errors. Tools are available for converting grammar-based descriptions of translations into working pro-grams.

Languages evolve over a period of time, acquiring new constructs and performing additional tasks. These new constructs can be added to a language more easily when there is an existing implementation based on a grammatical description of the language.

ROLE OF THE PARSER:

Parser for any grammar is program that takes as input string w (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for w , if w is a valid sentences of grammar or error message indicating that w is not a valid sentences of given grammar.

The goal of the parser is to determine the syntactic validity of a source string is valid, a tree is built for use by the subsequent phases of the computer. The tree reflects the sequence of derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue diagnostic message identifying the nature and cause of the errors in string. Every elementary subtree in the parse tree corresponds to a production of the grammar.

There are two ways of identifying an elementary subtree:

1. By deriving a string from a non-terminal or
2. By reducing a string of symbol to a non-terminal.

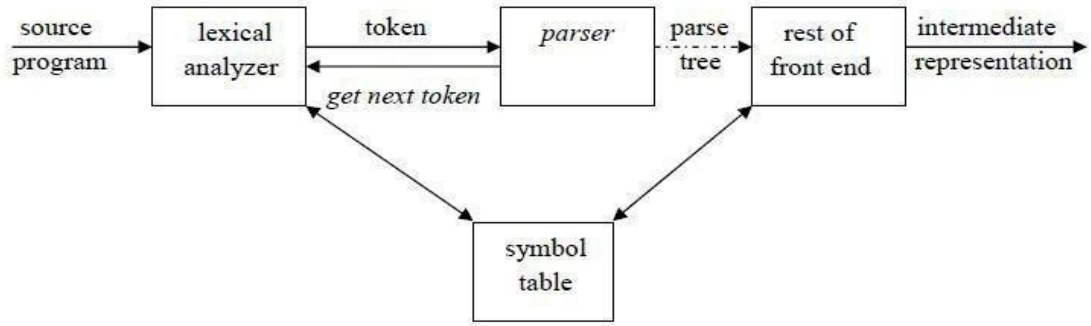


Fig 2.1 Role of Parser

CONTEXT FREE GRAMMARS

A context-free grammar (grammar for short) consists of terminals, non-terminals, a start symbol, and productions.

1. Terminals are the basic symbols from which strings are formed. The word "token" is a synonym for "terminal" when we are talking about grammars for programming languages.
2. Non terminals are syntactic variables that denote sets of strings. They also impose a hierarchical structure on the language that is useful for both syntax analysis and translation.
3. In a grammar, one non terminal is distinguished as the start symbol, and the set of strings it denotes is the language defined by the grammar.
4. The productions of a grammar specify the manner in which the terminals and non terminals can be combined to form strings. Each production consists of a non terminal, followed by an arrow, followed by a string of non terminals and terminals.

Inherently recursive structures of a programming language are defined by a context-free Grammar. In a context-free grammar, we have four triples G(V,T,P,S). Here , V is finite set of terminals (in our case, this will be the set of tokens) T is a finite set of non-terminals (syntactic- variables).P is a finite set of productions rules in the following form A → α where

A is a non- terminal and α is a string of terminals and non-terminals (including the empty string).S is a start symbol (one of the non-terminal symbol).
L(G) is the language of G (the language generated by G) which is a set of sentences.

A sentence of L(G) is a string of terminal symbols of G. If S is the start symbol of G then ω is a sentence of L(G) iff S ω where ω is a string of terminals of G. If G is a context-free grammar (G) is a context-free language. Two grammars G_1 and G_2 are equivalent, if they produce same grammar.
Consider the production of the form S α, If α contains non-terminals, it is called as a sentential form of G. If α does not contain non-terminals, it is called as a sentence of G.
Example: Consider the grammar for simple arithmetic expressions:

expr → expr op expr
expr → (expr)
expr → - expr
expr → **id**
op → +
op → -
op → *
op → /
op → ^

Terminals : id + - * / ^ ( )
Non-terminals : expr , op
Start symbol : expr

Notational Conventions:

1. These symbols are terminals:
   i. Lower-case letters early in the alphabet such as a, b, c.
   ii. Operator symbols such as +, -, etc.
   iii. Punctuation symbols such as parentheses, comma etc.
   iv. Digits 0,1,…,9.
   v. Boldface strings such as id or if (keywords)
2. These symbols are non-terminals:
   i. Upper-case letters early in the alphabet such as A, B, C..
   ii. The letter *S*, when it appears is usually the start symbol.

iii. Lower-case italic names such as *expr or stmt*.

3. Upper-case letters late in the alphabet, such as X,Y,Z, represent grammar symbols, that is either terminals or non-terminals.
4. Greek letters α , β , γ represent strings of grammar symbols.
   e.g a generic production could be written as A → α.
5. If A → α_1 , A → α_2 , ... , A → α_n are all productions with A , then we can write A → α_1 | α_2 |. . . . | α_n , (alternatives for A).
6. Unless otherwise stated, the left side of the first production is the start symbol.

Using the shorthand, the grammar can be written as:

E → E A E | (E) | - E | **id**

A → + | - | * | / | ^

Derivations:

A *derivation* of a string for a grammar is a sequence of grammar rule applications that transform the start symbol into the string. A derivation proves that the string belongs to the grammar's language.

$\Rightarrow$ derives in one step

$\overset{+}{\Rightarrow}$ derives in ≥ one step

$\overset{*}{\underset{G}{\Rightarrow}}$ indicates that the derivation utilizes the rules of grammar G

To create a string from a context-free grammar:

- Begin the string with a start symbol.
- Apply one of the production rules to the start symbol on the left-hand side by replacing the start symbol with the right-hand side of the production.
- Repeat the process of selecting non-terminal symbols in the string, and replacing them with the right-hand side of some corresponding production, until all non-terminals have been replaced by terminal symbols.

In general a derivation step is αAβ αγβ is sentential form and if there is a production rule A→γ in our grammar. where α and β are arbitrary strings of terminal and non-terminal symbols α1 α2... αn (αn derives from α1 or α1 derives αn). There are two types of derivation:

1. Leftmost Derivation (LMD):

- If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation.

- The sentential form derived by the left-most derivation is called the left-sentential form.

2. Rightmost Derivation (RMD):

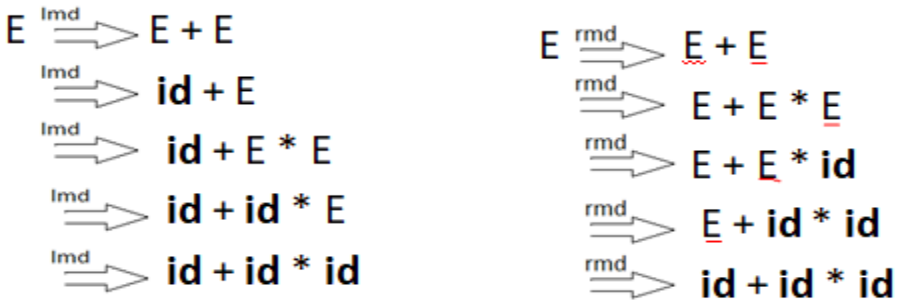
- If we scan and replace the input with production rules, from right to left, it is known as right-most derivation.
- The sentential form derived from the right-most derivation is called the right-sentential form.

Example:

Consider the G,

E → E + E | E * E | (E) | - E | **id**

Derive the string **id + id * id** using leftmost derivation and rightmost derivation.



(a) **(b)**

Fig 2.2 a) Leftmost derivation b) Rightmost derivation

Strings that appear in leftmost derivation are called left sentential forms. Strings that appear in rightmost derivation are called right sentential forms.

Sentential Forms:

Given a grammar G with start symbol S, if S => α , where α may contain non-terminals or terminals, then α is called the sentential form of G.

Parse Tree:

A parse tree is a graphical representation of a derivation sequence of a sentential form.

In a parse tree:

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

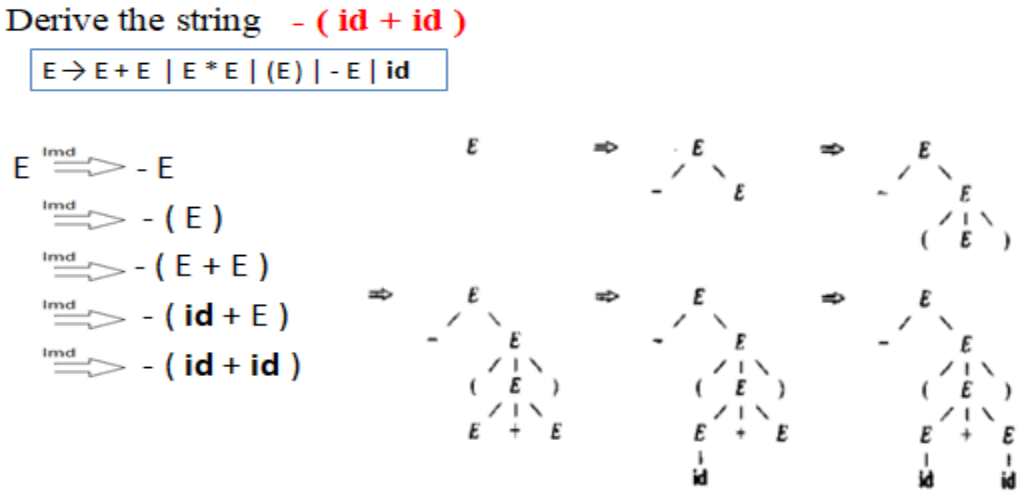


Fig 2.3 Build the parse tree for string **–(id+id)** from the derivation

Yield or frontier of tree:

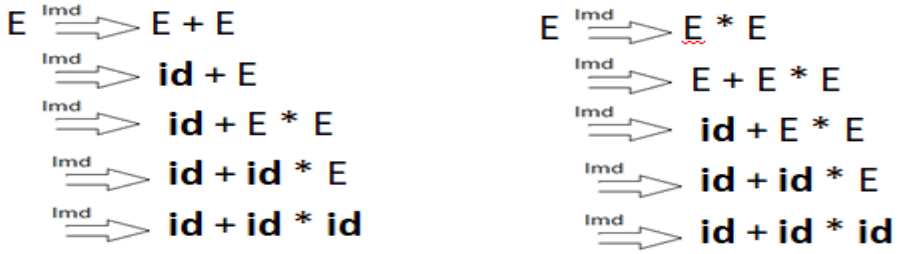
Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentential forms that are read from left to right. The sentential form in the parse tree is called yield or frontier of the tree.

Ambiguity:

A grammar that produces more than one parse tree for some sentence is said to be ambiguous grammar. i.e. An ambiguous grammar is one that produce more than one leftmost or more than one rightmost derivation for the same sentence.

Example : Given grammar G : E → E+E | E*E | (E) | - E | id

The sentence id+id*id has the following two distinct leftmost derivations:



The two corresponding parse trees are:

Fig 2.4 Two Parse tree for id+id*id

Consider another example,

stmt → if expr then stmt | if expr then stmt else stmt | other

This grammar is ambiguous since the string if E1 then if E2 then S1 else S2 has the following Two parse trees for leftmost derivation :

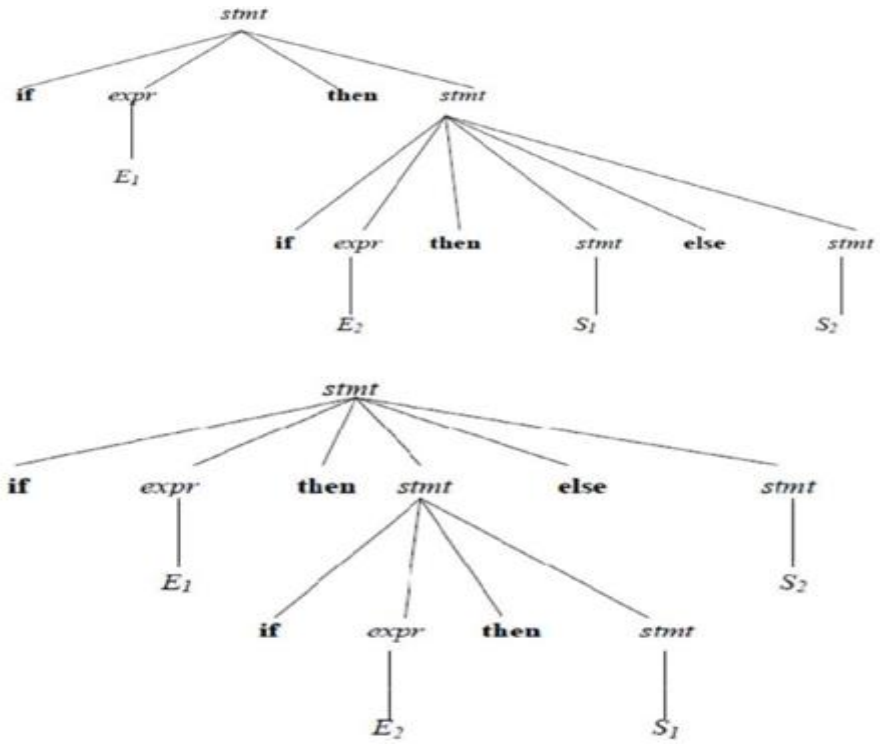


Fig 2.5 Two Parse tree for **if E1 then if E2 then S1 else S2**

Eliminating Ambiguity:

An ambiguous grammar can be rewritten to eliminate the ambiguity. e.g. Eliminate the ambiguity from **"dangling-else"** grammar,

stmt → **if** expr **then** stmt

| **if** expr **then** stmt **else** stmt

| **other**

Match each else with the closest previous unmatched then. This disambiguity rule can be incorporated into the grammar.

stmt → matched_stmt | unmatched_stmt

matched_stmt →if expr then matched_stmt else matched_stmt

| other

unmatched_stmt → if expr then stmt

| if expr then matched_stmt else unmatched_stmt

This grammar generates the same set of strings, but allows only one parsing for string.

Table 2.1 Ambiguous grammar vs. Unambiguous grammar

Ambiguous Grammar	Unambiguous Grammar
A grammar is said to be ambiguous if for at least one string generated by it, it produces more than one- • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation	A grammar is said to be unambiguous if for all the strings generated by it, it produces exactly one- • parse tree • or derivation tree • or syntax tree • or leftmost derivation • or rightmost derivation
For ambiguous grammar, leftmost derivation and rightmost derivation represents different parse trees.	For unambiguous grammar, leftmost derivation and rightmost derivation represents the same parse tree.
Ambiguous grammar contains less number of non-terminals.	Unambiguous grammar contains more number of non-terminals.
For ambiguous grammar, length of parse tree is less.	For unambiguous grammar, length of parse tree is large.
Ambiguous grammar is faster than unambiguous grammar in the derivation of a tree. (Reason is above 2 points)	Unambiguous grammar is slower than ambiguous grammar in the derivation of a tree.

Removing Ambiguity by Precedence & Associativity Rules:

An ambiguous grammar may be converted into an unambiguous grammar by implementing:

- Precedence Constraints
- Associativity Constraints

These constraints are implemented using the following rules:

Rule-1:

- The level at which the production is present defines the priority of the operator contained in it.
  - The higher the level of the production, the lower the priority of operator.
  - The lower the level of the production, the higher the priority of operator.

Rule-2:

- If the operator is left associative, induce left recursion in its production.
- If the operator is right associative, induce right recursion in its production.

Example: Consider the ambiguous Grammar:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \textbf{id}$$

Introduce new variable / non-terminals at each level of precedence,

- an expression **E** for our example is a sum of one or more terms. (+,-)
- a term **T** is a product of one or more factors. (*, /)
- a factor **F** is an identifier or parenthesised expression.

The resultant unambiguous grammar is:

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid \mathbf{id}$

Trying to derive the string **id+id*id** using the above grammar will yield one unique derivation.

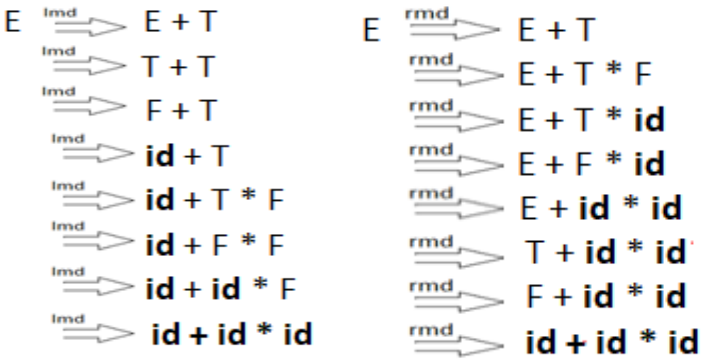


Fig 2.6 Distinct Leftmost and Rightmost derivation

Regular Expression vs. Context Free Grammar:

- Every construct that can be described by a regular expression can be described by a grammar.
- NFA can be converted to a grammar that generates the same language as recognized by the NFA.
- Rules:
  - For each state *i* of the NFA, create a non-terminal symbol A_i .
  - If state *i* has a transition to state *j* on symbol ***a***, introduce the production $A_i \rightarrow$ a A_j
  - If state *i* goes to state *j* on symbol ε, introduce the production $A_i \rightarrow A_j$
  - If *i* is an accepting state, introduce $A_i \rightarrow \varepsilon$
  - If *i* is the start state make A_i the start symbol of the grammar.

Example: The regular expression (a|b)*abb, consider the NFA

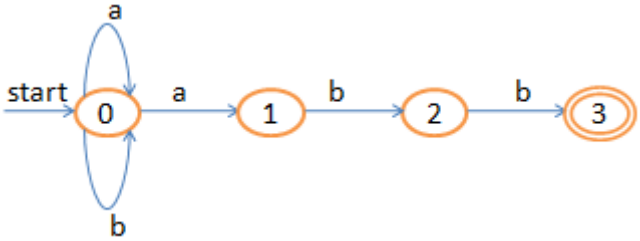


Fig 2.7 NFA for (a|b)*abb

Equivalent grammar is given by:

$A_0 \rightarrow a\ A_0 \mid b\ A_0 \mid a\ A_1$

$A_1 \rightarrow b\ A_2$

$A_2 \rightarrow b\ A_3$

$A_3 \rightarrow \varepsilon$

Types of Parser:

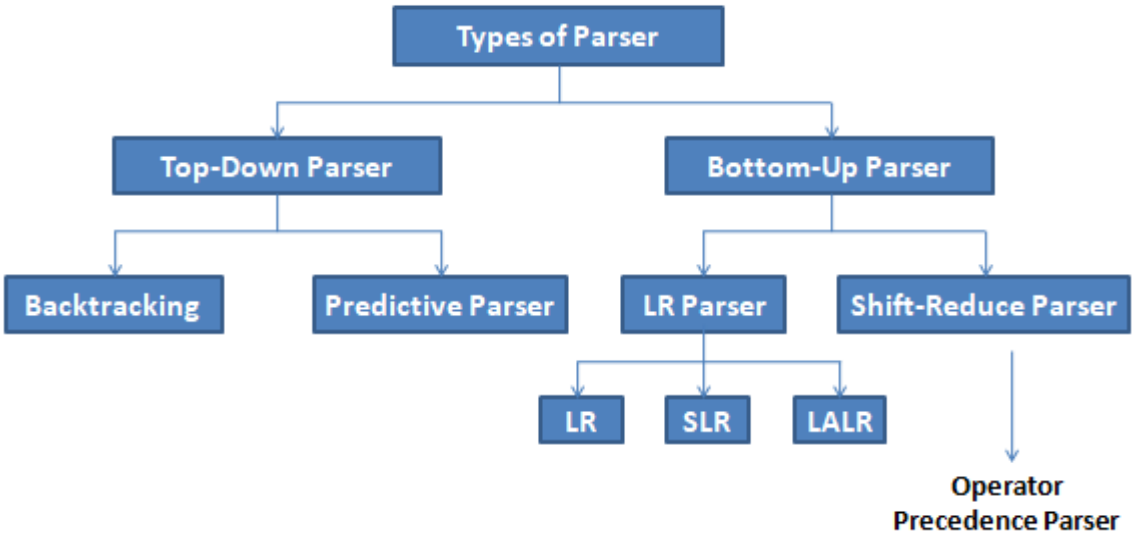


Fig 2.8 Types of Parser

LR Parsing:

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.

Why LR parsing:

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
- The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
- The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.
- 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 disadvantage is that it takes too much work to construct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

Bottom-Up Parsing:

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing. A general type of bottom-up parser is a shift-reduce parser.

Shift-Reduce Parsing:

Shift-reduce parsing is a type of bottom -up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example:

Consider the grammar:

S → aABe

A → Abc | b

B → d

The string to be recognized is abbcde. We want to reduce the string to S.

Steps of reduction:

abbcde (b,d can be reduced)

aAbcde (leftmost b is reduced)

aAde (now Abc,b,d qualified for reduction)

aABe (d can be reduced)

S

Each replacement of the right side of a production by the left side in the above example is called reduction, which is equivalent to rightmost derivation in reverse.

$$S \overset{rmd}{\Rightarrow} aABe \overset{rmd}{\Rightarrow} aAde \overset{rmd}{\Rightarrow} aAbcde \overset{rmd}{\Rightarrow} abbcde$$

Handle:

A substring which is the right side of a production such that replacement of that substring by the production left side leads eventually to a reduction to the start symbol, by the reverse of a rightmost derivation is called a handle.

Stack Implementation of Shift-Reduce Parsing:

There are two problems that must be solved if we are to parse by handle pruning. The first is to locate the substring to be reduced in a right-sentential form, and the second is to determine what production to choose in case there is more than one production with that substring on the right side.

A convenient way to implement a shift-reduce parser is to use a stack to hold grammar symbols and an input buffer to hold the string w to be parsed. We use **$** to mark the bottom of the stack and also the right end of the input. Initially, the stack is empty, and the string ***w*** is on the input, as follows:

STACK	INPUT
$	w$

The parser operates by shifting zero or more input symbols onto the stack until a handle is on top of the stack. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	INPUT
$ S	$

Example: The actions a shift-reduce parser in parsing the input string $id_1+id_2*id_3$, according to the ambiguous grammar for arithmetic expression.

	STACK	INPUT	ACTION
(1)	$	$id_1 + id_2 * id_3$	shift
(2)	$id_1	$+ id_2 * id_3$	reduce by $E \rightarrow id$
(3)	$E	$+ id_2 * id_3$	shift
(4)	$E +	$id_2 * id_3$	shift
(5)	$E + id_2	$* id_3$	reduce by $E \rightarrow id$
(6)	$E + E	$* id_3$	shift
(7)	$E + E *	id_3	shift
(8)	$E + E * id_3	$	reduce by $E \rightarrow id$
(9)	$E + E * E	$	reduce by $E \rightarrow E * E$
(10)	$E + E	$	reduce by $E \rightarrow E + E$
(11)	$E	$	accept

Fig 2.9 Configuration of Shift Reduce Parser on input $id_1+id_2*id_3$

RIGHT-SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Fig 2.10 Reductions made by Shift Reduce Parser

While the primary operations of the parser are shift and reduce, there are actually four possible actions a shift-reduce parser can make:

(1) shift, (2) reduce,(3) accept, and (4) error.

- In a ***shift*** action, the next input symbol is shifted unto the top of the stack.
- In a ***reduce*** action, the parser knows the right end of the handle is at the top of the stack. It must then locate the left end of the handle within the stack and decide with what non-terminal to replace the handle.
- In an ***accept*** action, the parser announces successful completion of parsing.
- In an ***error*** action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

Figure 2.11 represents the stack implementation of shift reduce parser using unambiguous grammar.

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce by F → id
$F	+id*id$	reduce by T → F
$T	+id*id$	reduce by E → T
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by F → id
$E+F	*id$	reduce by T → F
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	reduce by F → id
$E+T*F	$	reduce by T → T*F
$E+T	$	reduce by E → E+T
$E	$	accept

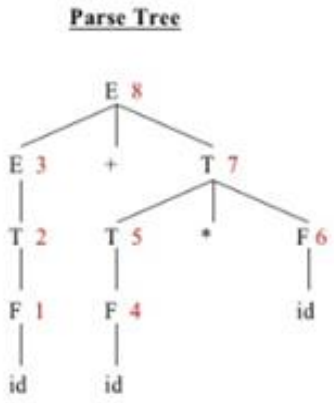


Fig 2.11 A stack implementation of a Shift-Reduce parser

Operator Precedence Parsing:

Operator grammars have the property that no production right side is ε (empty) or has two

adjacent non terminals. This property enables the implementation of efficient operator-precedence parsers.

Example: The following grammar for expressions:

E→E A E | (E) | -E | id

A→ + | - | * | / | ^

This is not an operator grammar, because the right side EAE has two consecutive non-terminals. However, if we substitute for A each of its alternate, we obtain the following operator grammar:

E→E + E |E – E |E * E | E / E | (E) | E ^ E | - E | id

In operator-precedence parsing, we define three disjoint precedence relations between pair of terminals. This parser relies on the following three precedence relations.

Relation	Meaning
$a <\cdot b$	*a* yields precedence to *b*
$a \doteq b$	*a* has the same precedence as *b*
$a \cdot> b$	*a* takes precedence over *b*

Fig 2.12 Precedence Relations

These precedence relations guide the selection of handles. These operator precedence relations allow delimiting the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end.

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

Fig 2.13 Operator Precedence Relation Table

Example: The input string: $id_1 + id_2 * id_3$

After inserting precedence relations the string becomes:

$\$ <\cdot\ id_1 \cdot> + <\cdot\ id_2 \cdot> * <\cdot\ id_3 \cdot> \$$

Having precedence relations allows identifying handles as follows:

1. Scan the string from left end until the leftmost ·> is encountered.
2. Then scan backwards over any =’s until a <· is encountered.
3. Everything between the two relations <· and ·> forms the handle.

Stack	Rule	Input	Comments
$ < id > + < id > * < id > $	E → id	$ id + id * id $	Here the first "id" is looked as the handle and since we were able to reduce, we reduce it in the input
$ < + < id > * < id > $	E → id	$ E + id * id $	The second handle is also "id" since that is available between a pair of lesser than and greater than precedences
$ < + < * < id > $	E → id	$ E + E * id $	The third handle is also "id".
$ < + < * > $	E → E*E	$ E + E * E $	The fourth handle is E *E, and is popped in the stack and we push the greater than symbol.
$ < + > $	E → E+E	$ E + E $	The last handle is E+E and that is also reduced.
$$			The stack is empty and has only the $ symbol, we say the string is accepted.

Defining Precedence Relations:

The precedence relations are defined using the following rules:

Rule-01:

- If precedence of b is higher than precedence of a, then we define a < b
- If precedence of b is same as precedence of a, then we define a = b
- If precedence of b is lower than precedence of a, then we define a > b

Rule-02:

- An identifier is always given the higher precedence than any other symbol.
- $ symbol is always given the lowest precedence.

Rule-03:

- If two operators have the same precedence, then we go by checking their associativity.

1. ↑ is of highest precedence and right-associative,
2. * and / are of next highest precedence and left-associative, and
3. + and − are of lowest precedence and left-associative,

	+	−	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Fig 2.14 Operator Precedence Relation Table

STACK	INPUT	COMMENT
$	<· id+id*id $	shift id
$ id	·> +id*id $	pop the top of the stack id
$	<· +id*id $	shift +
$ +	<· id*id $	shift id
$ +id	·> *id $	pop id
$ +	<· *id $	shift *
$ + *	<· id $	shift id
$ + * id	·> $	pop id
$ + *	·> $	pop *
$ +	·> $	pop +
$	$	accept

Fig 2.15 Stack Implementation

Implementation of Operator-Precedence Parser:

- An operator-precedence parser is a simple shift-reduce parser that is capable of parsing a subset of LR(1) grammars.
- More precisely, the operator-precedence parser can parse all LR(1) grammars where two consecutive non-terminals and epsilon never appear in the right-hand side of any rule.

Steps involved in Parsing:

1. Ensure the grammar satisfies the pre-requisite.
2. Computation of the function LEADING()
3. Computation of the function TRAILING()
4. Using the computed leading and trailing ,construct the operator Precedence Table
5. Parse the given input string based on the algorithm
6. Compute Precedence Function and graph.

Computation of LEADING:

- Leading is defined for every non-terminal.
- Terminals that can be the first terminal in a string derived from that non-terminal.
- LEADING(A)={ a| $A =>^{+} \gamma a \delta$ },where γ is ε or any non-terminal, $=>^{+}$ indicates derivation in one or more steps, A is a non-terminal.

Algorithm for LEADING(A):

{

1. ‘a’ is in LEADING(A) is A→ γaδ where γ is ε or any non-terminal.

2.If ‘a’ is in LEADING(B) and A→B, then ‘a’ is in LEADING(A).

}

Computation of TRAILING:

- Trailing is defined for every non-terminal.
- Terminals that can be the last terminal in a string derived from that non-terminal.
- TRAILING(A)={ a| A=>$^{+}$ γaδ },where δ is ε or any non-terminal, =>$^{+}$ indicates derivation in one or more steps, A is a non-terminal.

Algorithm for TRAILING(A):

{

1. ‘a’ is in TRAILING(A) is A→ γaδ where δ is ε or any non-terminal.

2.If ‘a’ is in TRAILING(B) and A→B, then ‘a’ is in TRAILING(A).

}

Example 1: Consider the unambiguous grammar,

E→E + T

E→T

T→T * F

T→F

F→(E)

F→**id**

Step 1: Compute LEADING and TRAILING:

LEADING(E)= { +,LEADING(T)} ={+ , * , (, id}

LEADING(T)= { *,LEADING(F)} ={* , (, id}

LEADING(F)= { (, id}

TRAILING(E)= { +, TRAILING(T)} ={+ , * ,) , id}

TRAILING(T)= { *, TRAILING(F)} ={* ,) , id}

TRAILING(F)= {) , id}

Step 2: After computing LEADING and TRAILING, the table is constructed between all the terminals in the grammar including the ‘$’ symbol.

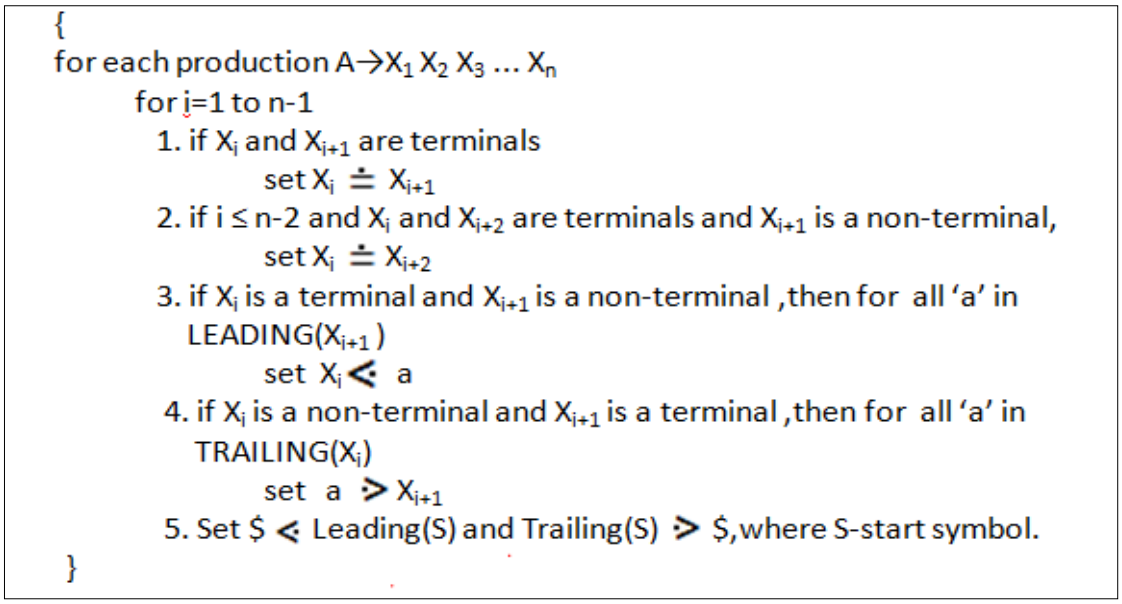


Fig 2.16 Algorithm for constructing Precedence Relation Table

	+	*	id	(	)	$
+	>	<	<	<	>	>
*	>	>	<	<	>	>
id	>	>	e	e	>	>
(	<	<	<	<	=	e
)	>	>	e	e	>	>
$	<	<	<	<	e	Accept

Fig 2.17 Precedence Relation Table * All undefined entries are error (e).

Rough work:

LEADING(E)= {+ , * , (, id}

LEADING(T)= {* , (, id}

LEADING(F)= { (, id}

TRAILING(E)= {+ , * ,) , id}

TRAILING(T)= {* ,) , id}

TRAILING(F)= {) , id}

Terminal followed by NonTerminal

Rule-1.

+ T => + < leading(T)

Rule-3.

* F => * < leading(F)

Rule-4.

(E => (< leading(E)

Non-Terminal followed by Terminal

Rule-1. E + => Trailing(E) > +

Rule-3. T * => Trailing(T) > *

Rule-4. E) => Trailing(E) >)

Step 3: Parse the given input string **(id+id)*id$**

```
Set ip to point to the first symbol of w$
Repeat forever
    if $ is on the top of the stack and ip points to $ then return
    else begin
        Let a be the top terminal on the stack, and b the symbol pointed to by ip
            if a <· b or a =· b then
                push b onto the stack
                advance ip to the next input symbol
            end
            else if a ·> b then
                repeat
                    pop the stack
                until the top stack terminal is related by <·
                    to the terminal most recently popped
            else error()
    end
```

Fig 2.18 Parsing Algorithm

STACK	REL.	INPUT	ACTION
$	$ < (	(id+id)*id$	Shift (
$(	(< id	id+id)*id$	Shift id
$(id	id > +	+id)*id$	Pop id
$(	(< +	+id)*id$	Shift +

$(+	+ < id	id)*id$	Shift id
$(+id	id >)	)*id$	Pop id
$(+	+ >)	)*id$	Pop +
$(	(=)	)*id$	Shift)
$() $(	) > *	*id $	Pop) Pop (
$	$ < *	*id $	Shift *
$*	* < id	id$	Shift id
$*id	id > $	$	Pop id
$*	* > $	$	Pop *
$		$	Accept

Fig 2.19 Parse the input string (id+id)*id$

Precedence Functions:

Compilers using operator-precedence parsers need not store the table of precedence relations. In most cases, the table can be encoded by two precedence functions f and g that map terminal symbols to integers. We attempt to select f and g so that, for symbols a and b.

1. f (a) < g(b) whenever a<·b.
2. f (a) = g(b) whenever a = b. and
3. f(a) > g(b) whenever a ·> b.

Algorithm for Constructing Precedence Functions:

1. Create functions f_a for each grammar terminal *a* and for the end of string symbol.
2. Partition the symbols in groups so that f_a and g_b are in the same group if $a = b$ (there can be symbols in the same group even if they are not connected by this relation).
3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of g_b to the group of f_a if $a <\cdot b$, otherwise if $a \cdot> b$ place an edge from the group of f_a to that of g_b.
4. If the constructed graph has a cycle then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of f_a and g_b respectively.

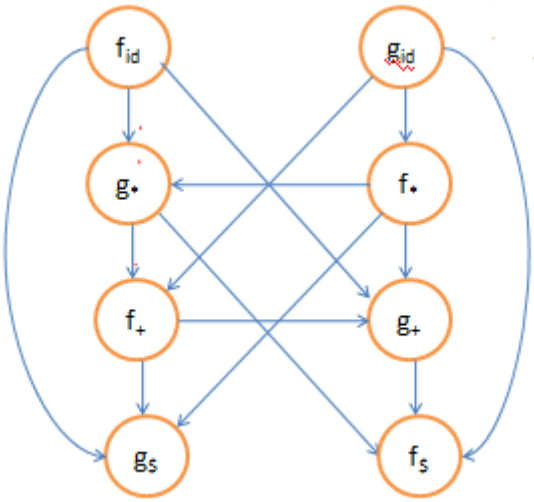


Fig 2.20 Precedence Graph

There are no cycles,so precedence function exist. As f$ and g$ have no out edges,f($)=g($)=0.The longest path from g+ has length 1,so g(+)=1.There is a path from g_{id} to f* to g* to f+ to g+ to f$,so g(id)=5.The resulting precedence functions are:

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Example 2:

Consider the following grammar, and construct the operator precedence parsing table and check whether the input string (i) *id=id (ii)id*id=id are successfully parsed or not?

S→L=R

S→R

L→*R

L→id

R→L

Solution:

1. **Computation of LEADING:**

LEADING(S) = {=, * , id}

LEADING(L) = {* , id}

LEADING(R) = {* , id}

2. **Computation of TRAILING:**

TRAILING(S) = {= , * , id}

TRAILING(L)= {* , id}

TRAILING(R)= {* , id}

3.Precedence Table:

	=	*****	**id**	**$**
=	e	<·	<·	·>
*****	·>	<·	<·	·>
id	·>	e	e	·>
$	<·	<·	<·	accept

* All undefined entries are error (e).

4.Parsing the given input string:

1. ***id = id**

STACK	**INPUT STRING**	**ACTION**
$	*id=id$	$<·* Push
$*	id=id$	*<·id Push
$*id	=id$	id·>= Pop
$*	=id$	*·>= Pop
$	=id$	$<·= Push
$=	id$	=<·id Push
$=id	$	id·>$ Pop
$=	$	=·>$ Pop
$	$	Accept

2. **id*id=id**

STACK	**INPUT STRING**	**ACTION**
$	id*id=id$	$<·idPush
$id	*id=id$	Error

Example 3: Check whether the following Grammar is an operator precedence grammar or not.

E→E+E

E→E*E

E→id

Solution:

1. **Computation of LEADING:**

LEADING(E) = {+, * , id}

2. **Computation of TRAILING:**

TRAILING(E) = {+, * , id}

3. **Precedence Table:**

	+	*****	**id**	**$**
+	<·/·>	<·/·>	<·	·>
*****	<·/·>	<·/·>	<·	·>
id	·>	·>		·>
$	<·	<·	<·	

All undefined entries are error. Since the precedence table has multiple defined entries, the grammar is not an operator precedence grammar.

LR PARSERS:

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(k) parsing. The "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the "k" for the number of input symbols of lookahead that are used in making parsing decisions.. When (k) is omitted, it is assumed to be 1. Table 2.2 shows the comparison between LL and LR parsers.

Table 2.2 LL vs. LR

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

Types of LR parsing method:

1. SLR- Simple LR
    - Easiest to implement, least powerful.
2. CLR- Canonical LR
    - Most powerful, most expensive.
3. LALR- Look -Ahead LR
    - Intermediate in size and cost between the other two methods

The LR Parsing Algorithm:

The schematic form of an LR parser is shown in Fig 2.25. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (*action* and *goto*).The driver program is the same for all LR parser. The parsing table alone changes from one parser to another. The parsing program reads characters from an input buffer one at a time. The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots\ldots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a symbol called a state.

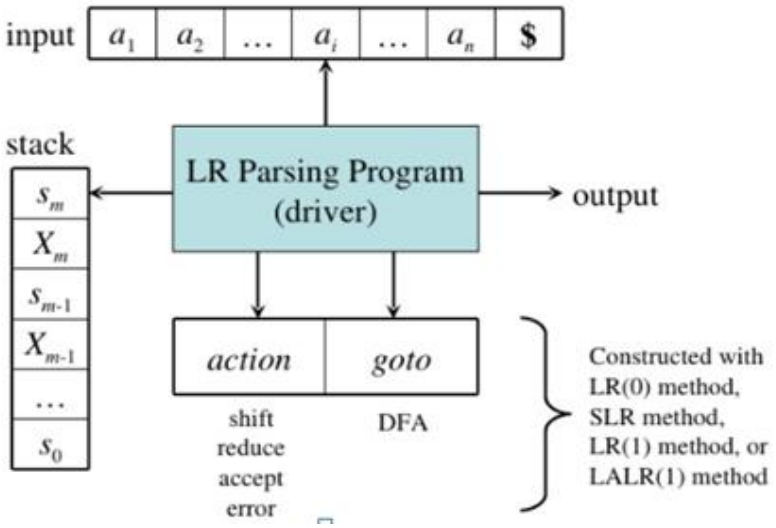


Fig 2.25 Model of an LR Parser

The parsing table consists of two parts : *action* and *goto* functions.

Action : The parsing program determines sm, the state currently on top of stack, and ai, the current input symbol. It then consults *action*[s_m,a_i] in the action table which can have one of four values :

1. shift s, where s is a state,
2. reduce by a grammar production A → β,
3. accept, and
4. error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

CONSTRUCTING SLR PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute *goto*(I,X), where, I is set of items and X is grammar symbol.

LR(0) items:

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

A → •XYZ

A → X•YZ

A → XY•Z

A → XYZ•

Closure operation:

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

1. Initially, every item in I is added to closure(I).
2. If $A \rightarrow \alpha . B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow . \gamma$ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items $[A \rightarrow \alpha X \bullet \beta]$ such that $[A \rightarrow \alpha \bullet X\beta]$ is in I.Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G`
2. Construct the canonical collection of set of items C for G"
3. Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G"

Output : The SLR parsing table functions *action* and *goto* for G'

Method :

1. Construct $C = \{I_0, I_1, \ldots I_n\}$, the collection of sets of LR(0) items for G'.
2. State *i* is constructed from I_i. The parsing functions for state *i* are determined as follows:
   (a) If $[A \rightarrow \alpha \bullet a\beta]$ is in I_i and $goto(I_i,a) = I_j$, then set *action*[*i*,*a*] to "shift j". Here *a* must be terminal.
   (b) If$[A \rightarrow \alpha \bullet]$ is in I_i , then set *action*[*i*,*a*] to "reduce $A \rightarrow \alpha$" for all *a* in FOLLOW(A).
   (c) If [S"→S•] is in I_i, then set *action*[*i*,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The *goto* transitions for state *i* are constructed for all non-terminals A using the rule: If *goto*$(I_i,A) = I_j$, then *goto*[i,A] = *j*.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing [S'→•S].

SLR Parsing algorithm:

Input: An input string *w* and an LR parsing table with functions *action* and *goto* for grammar G.

Output: If *w* is in L(G), a bottom-up-parse for *w*; otherwise, an error indication.

Method: Initially, the parser has s0 on its stack, where s0 is the initial state, and *w*$ in the input buffer. The parser then executes the following program :

set *ip* to point to the first input symbol of *w*$;

repeat forever begin

let *s* be the state on top of the stack and *a* the symbol pointed to by *ip*;

Example: Implement SLR Parser for the given grammar:

1.E→E + T

2.E→T

3.T→T * F

4.T→F

5.F→(E)

6.F→**id**

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar:

E'→E

E→E + T

E→T

T→T * F

T→F

F→(E)

F→id

Step 2 : Find LR (0) items.

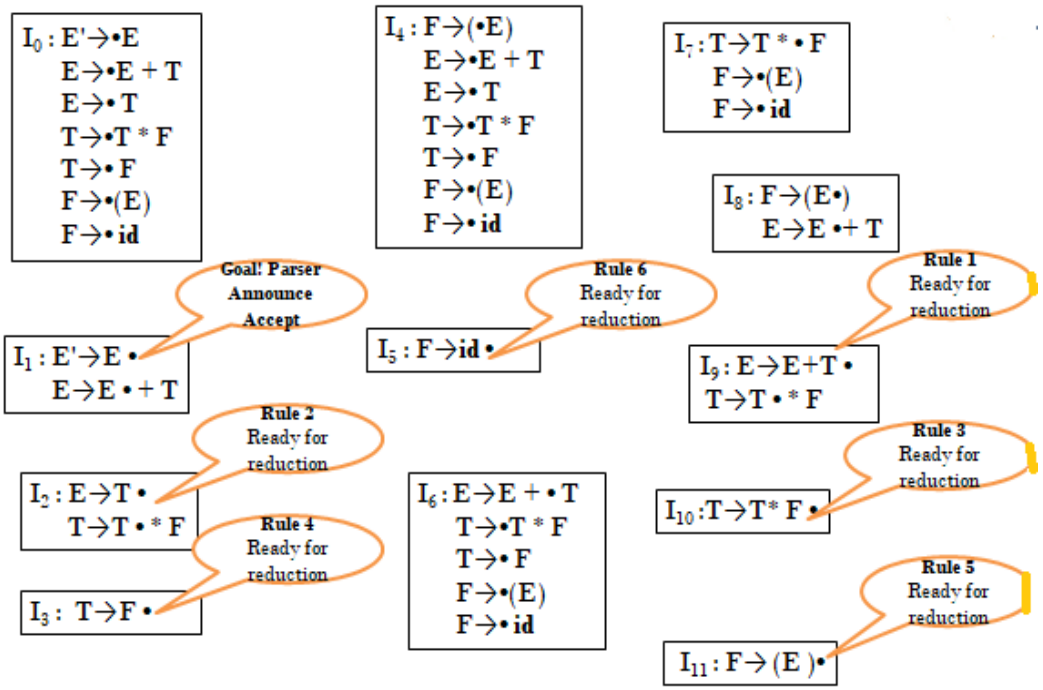


Fig 2.26 Canonical LR(0) collections

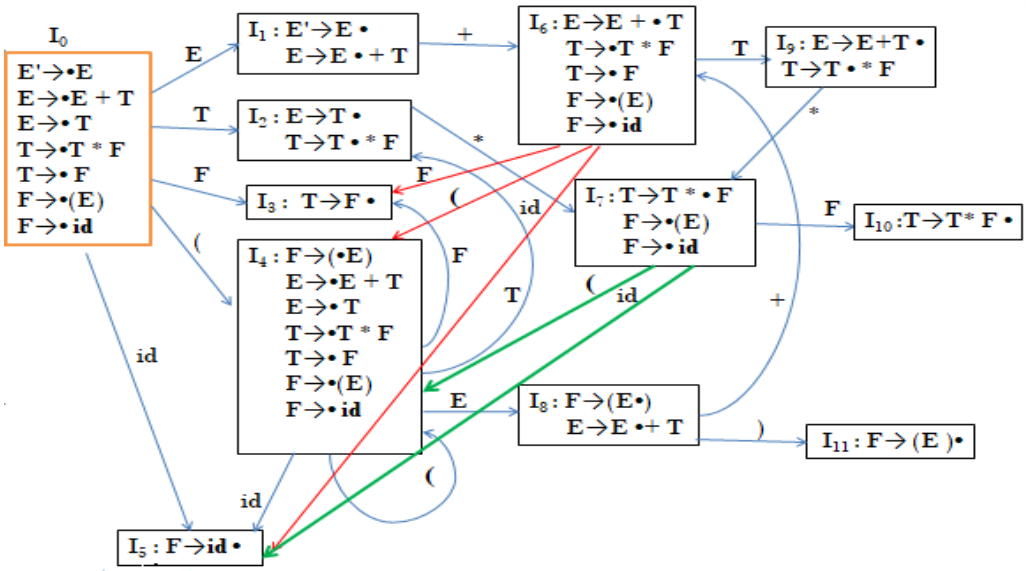


Fig 2.27 DFA representing the GOTO on symbols

Step 3 : Construction of Parsing table.

1. Computation of FOLLOW is required to fill the reduction action in the ACTION part of the table.

   FOLLOW(E) = {+,),$ }

   FOLLOW(T) ={*,+,) ,$}

   FOLLOW(F) ={*,+,) ,$}

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig 2.28 Parsing Table for the expression grammar

1. s_i means shift and stack state i.
2. r_j means reduce by production numbered j.
3. acc means accept.
4. Blank means error.

Step 4: Parse the given input. The Fig 2.29 shows the parsing the string id*id+id using stack implementation.

Stack	Input	Action
0	id*id+id$	s5 Shift 5
0 id 5	*id+id$	r6 Reduce by F→**id**
0 **F** 3	*id+id$	r4 Reduce by T→**F**
0 **T** 2	*id+id$	s7 Shift 7
0 **T** 2 * 7	id+id$	s5 Shift 5
0 **T** 2 * 7 id 5	+id$	r6 Reduce by F→**id**
0 **T** 2 * 7 **F** 10	+id$	r3 Reduce by T→**T*F**
0 **T** 2	+id$	r2 Reduce by E→**T**
0 **E** 1	+id$	s6 Shift 6
0 **E** 1+ 6	id$	s5 Shift 5
0 **E** 1+ 6 id 5	$	r6 Reduce by F→**id**
0 **E** 1+ 6 **F** 3	$	r4 Reduce by T→**F**
0 **E** 1+ 6 **T** 9	$	r1 Reduce by E→**E+T**
0 **E** 1	$	Accept

Fig 2.29 Moves of LR parser on **id*id+id**

Top-Down Parsing- Recursive Descent Parsing:

Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string. Equivalently it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

A general form top-down parsing called recursive descent parsing, involves backtracking, that is making repeated scans of the input. A special case of recursive descent parsing called predictive parsing, where no backtracking is required.

Consider the grammar

S → cAd

A → ab | a

and the input string w=cad. Construction of parse is shown in fig 2.21.

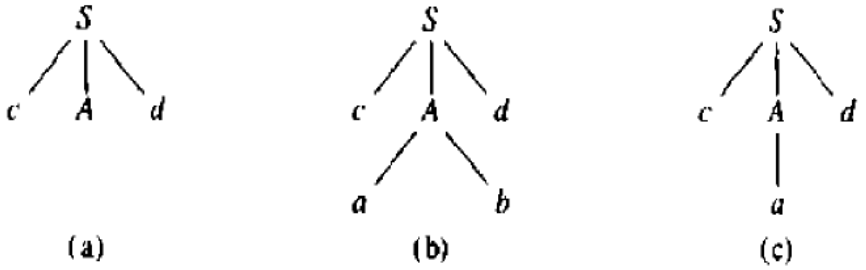


Fig 2.21 Steps in Top-down Parse

The leftmost leaf, labeled c, matches the first symbol of w, hence advance the input pointer to a, the second symbol of w. Fig 2.21(b) and (c) shows the backtracking required to match the input string.

Predictive Parser:

A grammar after eliminating left recursion and left factoring can be parsed by a recursive descent parser that needs no backtracking is a called a predictive parser. Let us understand how to eliminate left recursion and left factoring.

Eliminating Left Recursion:

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production A → Aα | β it can be replaced with a sequence of two productions

A → βA'

A' → αA' | ε

Without changing the set of strings derivable from A.

Example : Consider the following grammar for arithmetic expressions:

E → E+T | T

T → T*F | F

F → (E) | id

First eliminate the left recursion for E as

E → TE'

E' → +TE' | ε

Then eliminate for T as

T → FT '

T'→ *FT ' | ε

Thus the obtained grammar after eliminating left recursion is

E → TE'

E' → +TE' | ε

T → FT '

T'→ *FT ' | ε

F → (E) | id

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order $A_1, A_2 \ldots A_n$.
2. **for** i := 1 to n do begin

   for j := 1 to i-1 do begin

   replace each production of the form Ai → Aj γ

   by the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k \gamma$.

   where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current Aj-productions;

   end

   eliminate the immediate left recursion among the A_i- productions

   end

Left factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$, it can be rewritten as

A → αA'

A' → $\alpha\beta_1 \mid \alpha\beta_2$

Consider the grammar,

S → iEtS | iEtSeS | a

E → b

Here,***i,t,e*** stand for **if ,the**,and **else** and **E** and **S** for "expression" and "statement".

After Left factored, the grammar becomes

S → iEtSS' | a

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

Non-recursive Predictive Parsing:

It is possible to build a non-recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that of determining the production to be applied for a non-terminal. The non-recursive parser in Fig 2.22 looks up the production to be applied in a parsing table.

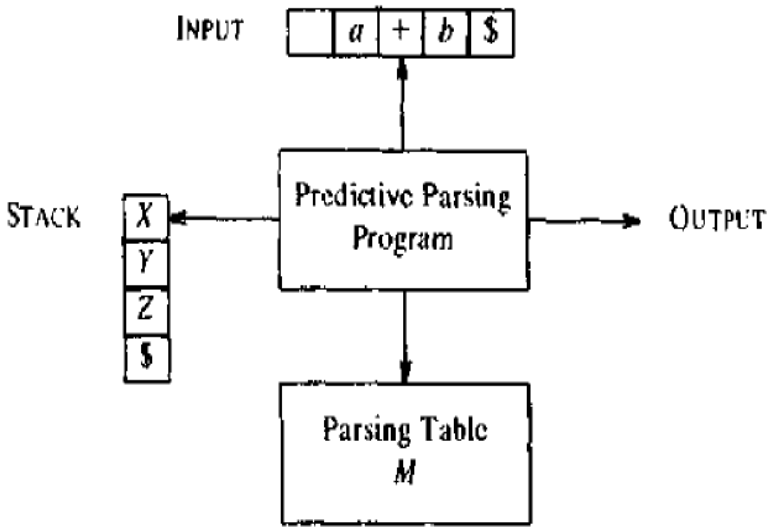


Fig 2.22 Model of a Non-recursive predictive parser

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of S. The parsing table is a two-dimensional array *M[A,a]*,where *A* is a non-terminal, and *a* is a terminal or the symbol $.

The program considers X, the symbol on top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1. If X = a =$,the parser halts and announces successful completion of parsing.
2. If X =a ≠$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a nonterminal, the program consults entry M[X,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example,

M[X,a] = {X→UVW}, the parser replaces X on top of the stack by WVU (with U on top). If M[X, a] = error, the parser calls an error recovery routine.

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G .These functions are FIRST and FOLLOW.

Rules for FIRST():

1. If X is terminal, then FIRST(X) is {X}.
2. If X → ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X → aα is a production then add a to FIRST(X).
4. If X is non-terminal and X → $Y_1 Y_2 \ldots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y_1),…,FIRST(Y_{i-1}); that is, $Y_1,\ldots Y_{i-1}$ => ε. If ε is in FIRST(Y_j) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for FOLLOW():

1. If S is a start symbol, then FOLLOW(S) contains $.
2. If there is a production A → αBβ, then everything in FIRST(β) except ε is placed in follow(B).
3. If there is a production A → αB, or a production A → αBβ where FIRST(β) contains ε,then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm for construction of predictive parsing table:

Input : Grammar G

Output : Parsing table M

Method :

1. For each production A → α of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add A → α to M[A, a].
3. If ε is in FIRST(α), add A → α to M[A, b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A) , add A → α to M[A, $].
4. Make each undefined entry of M be error.

Algorithm : Non-recursive predictive parsing.

Input: A string w and a parsing table M for grammar G.

Output: If w is in L(G), a leftmost derivation of w; otherwise, an error .

Method: Initially, the parser is in a configuration in which it has $$ on the stack with S, the start symbol of G on top, and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input.

set *ip* to point to the first symbol of *w$;*
repeat
let X be the top stack symbol and **a** the symbol pointed to by i_p;
if X is a terminal or $ ***then***
If X=a ***then***

Example:

Consider the following grammar:

E → E+T | T

T → T*F | F

F → (E) | id

Step 1: After eliminating left recursion the grammar is

E → TE'

E' → +TE' | ε

T → FT '

T'→ *FT ' | ε

F → (E) | id

Step 2: Computation of FIRST( ) :

FIRST(E) = { (, id}

FIRST(E') ={+ , ε }

FIRST(T) = { (, id}

FIRST(T') = {*, ε }

FIRST(F) = { (, id }

Step 3: Computation of FOLLOW( ):

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

FOLLOW(T) = { +, $,) }

FOLLOW(T') = { +, $,) }

FOLLOW(F) = {+, * , $,) }

Step 4: Construction of Predictive parsing table

M[X,a]	id	+	*	(	)	$
E	E →TE′			E →TE′		
E′		E′ →+TE′			E′ →ε	E′ →ε
T	T →FT′			T →FT′		
T′		T′ →ε	T′ →*FT′		T′ →ε	T′ →ε
F	F →id			F → (E)		

Fig 2.23 Parsing table

Step 5: Parsing the given string

With input **id+id*id** the predictive parser makes the sequence of moves shown in Fig 2,24.

STACK	INPUT	OUTPUT
$E	id+id*id$	E →TE’
$E’T	id+id*id$	T →FT’
$E’T’F	id+id*id$	F →id
$E’T’id	id+id*id$	pop
$E’T’	+id*id$	T’ →ε
$E’	+id*id$	E’ →+TE’
$E’T+	+id*id$	pop
$E’T	id*id$	T →FT’
$E’T’F	id*id$	F →id
$E’T’id	id*id$	Pop
$E’T’	*id$	T’ →*FT’
$E’T’F*	*id$	Pop
$E’T’F	id$	F →id
$E’T’id	id$	Pop
$E’T’	$	T’ →ε
$E’	$	E’ →ε
$	$	Accept

Fig 2.24 Moves made by predictive parser on input **id+id*id**

LL(1) Grammars:

For some grammars the parsing table may have some entries that are multiply-defined. For example, if G is left recursive or ambiguous , then the table will have at least one multiply-defined entry. A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

Example: Consider this following grammar:

S→ iEtS | iEtSeS | a

E → b

After eliminating left factoring, we have

S→ iEtSS’ | a S’→ eS | ε

E → b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

FIRST(S) ={ i, a }

FIRST(S’) = {e, ε }

FIRST(E) = { b}

FOLLOW(S) = { $,e }

FOLLOW(S’) = { $,e }

FOLLOW(E) = {t}

Parsing Table for the grammar:

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS S' → ε			S' → ε
E		E → b				

Since there are more than one production for an entry in the table, the grammar is not LL(1) grammar.

Error detection and Recovery in Syntax Analyzer:

In this phase of compilation, all possible errors made by the user are detected and reported to the user in form of error messages. This process of locating errors and reporting them to users is called the **Error Handling process**.

Functions of an Error handler.

- Detection
- Reporting
- Recovery

Classification of Errors

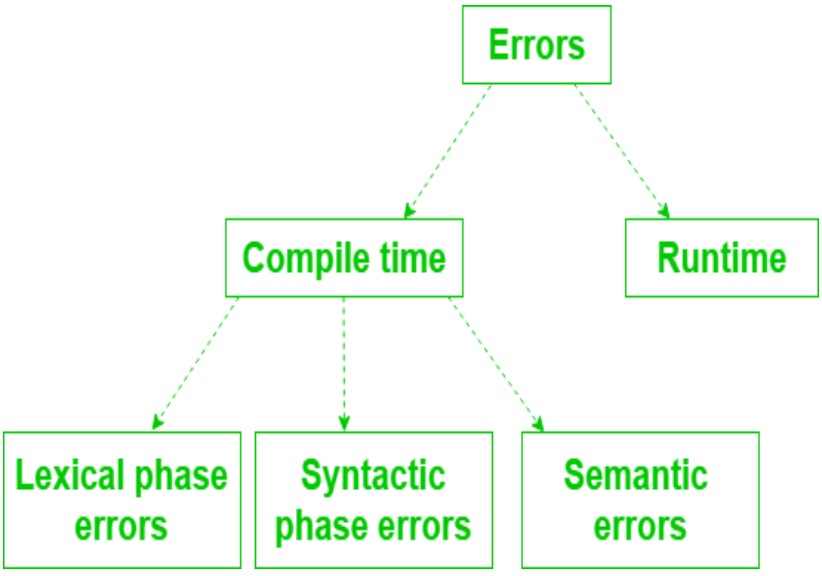


Fig 2.25 Classification of Errors

Compile-time errors:

Compile-time errors are of three types:-

1.Lexical phase errors

These errors are detected during the lexical analysis phase. Typical lexical errors are:

- Exceeding length of identifier or numeric constants.
- The appearance of illegal characters
- Unmatched string

2.Syntactic phase errors:

These errors are detected during the syntax analysis phase. Typical syntax errors are:

- Errors in structure
- Missing operator
- Misspelled keywords
- Unbalanced parenthesis

Error recovery for syntactic phase recovery:

1. Panic Mode Recovery

- In this method, successive characters from the input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are delimeters such as ; or }
- The advantage is that it's easy to implement and guarantees not to go into an infinite loop
- The disadvantage is that a considerable amount of input is skipped without checking it for additional errors

2. Statement Mode recovery

- In this method, when a parser encounters an error, it performs the necessary correction on the remaining input so that the rest of the input statement allows the parser to parse ahead.
- The correction can be deletion of extra semicolons, replacing the comma with semicolons, or inserting a missing semicolon.
- While performing correction, utmost care should be taken for not going in an infinite loop.
- A disadvantage is that it finds it difficult to handle situations where the actual error occurred before pointing of detection.

3. Error production

- If a user has knowledge of common errors that can be encountered then, these errors can be incorporated by augmenting the grammar with error productions that generate erroneous constructs.

- If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.
- The disadvantage is that it's difficult to maintain.

4. Global Correction

- The parser examines the whole program and tries to find out the closest match for it which is error-free.
- The closest match program has less number of insertions, deletions, and changes of tokens to recover from erroneous input.
- Due to high time and space complexity, this method is not implemented practically.

3.Semantic errors

These errors are detected during the semantic analysis phase. Typical semantic errors are

- Incompatible type of operands
- Undeclared variables
- Not matching of actual arguments with a formal one

Error recovery for Semantic errors

- If the error **"Undeclared Identifier"** is encountered then, to recover from this a symbol table entry for the corresponding identifier is made.
- If data types of two operands are incompatible then, automatic type conversion is done by the compiler.

YACC-Yet Another Compiler Compiler

Before 1975 writing a compiler was a very time-consuming process. Then Lesk [1975] and Johnson [1975] published papers on lex and yacc. These utilities greatly simplify compiler writing.

- YACC stands for **Yet Another Compiler Compiler**.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

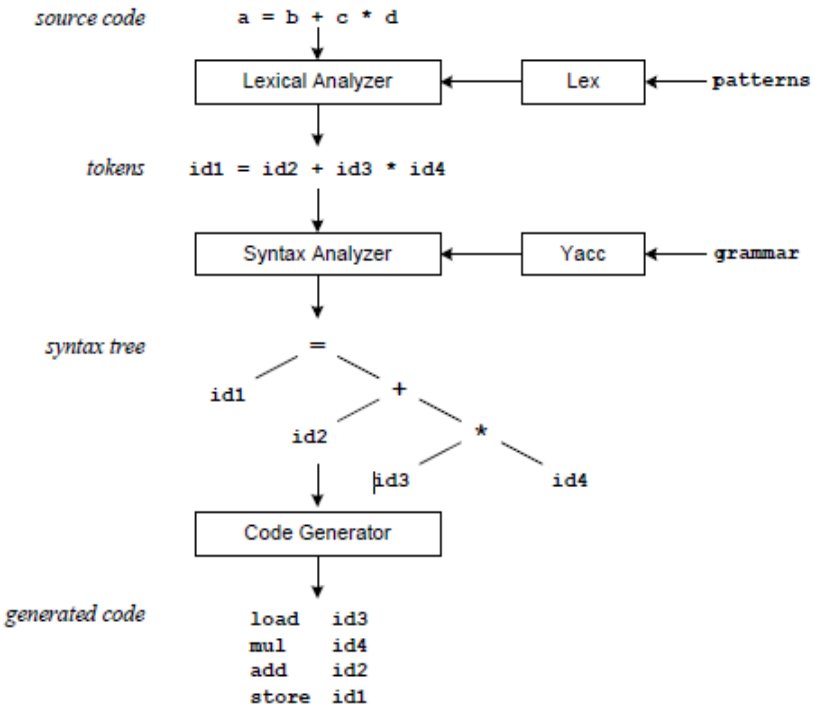


Fig 2.26 Compilation Sequence

The **patterns** in the above diagram is a file you create with a text editor. Lex will read your patterns and generate C code for a lexical analyzer or scanner. The lexical analyzer matches strings in the input, based on your patterns, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing.

When the lexical analyzer finds identifiers in the input stream it enters them in a symbol table. The symbol table may also contain other information such as data type (integer or real) and location of each variable in memory. All subsequent references to identifiers refer to the appropriate symbol table index.

The **grammar** in the above diagram is a text file you create with a text edtior. Yacc will read your grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree. The syntax tree imposes a hierarchical structure the tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depth-first walk of the syntax tree to generate code. Some compilers produce machine code, while others, as shown above, output assembly language.

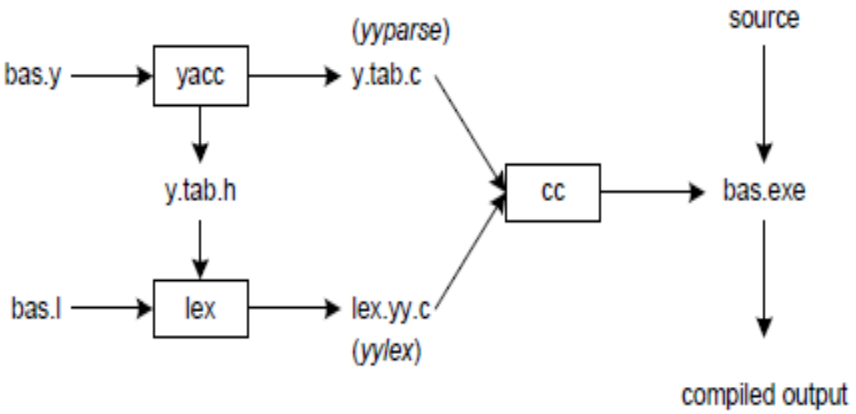


Fig. 2.27 Building a Compiler with Lex/Yacc

```
yacc -d bas.y                    # create y.tab.h, y.tab.c
lex bas.l                        # create lex.yy.c
cc lex.yy.c y.tab.c -obas.exe    # compile/link
```

Yacc reads the grammar descriptions in bas.y and generates a syntax analyzer (parser) that includes function yyparse, in file y.tab.c. Included in file bas.y are token declarations. The –d option causes yacc to generate definitions for tokens and place them in file y.tab.h.

Lex reads the pattern descriptions in bas.l, includes file y.tab.h, and generates a lexical analyzer, that includes function yylex, in file lex.yy.c.

Finally, the lexer and parser are compiled and linked together to create executable bas.exe. From main we call yyparse to run the compiler. Function yyparse automatically calls yylex to obtain each token.

Input File:

YACC input file is divided into three parts.

```
/* definitions */
  ....

%%
/* rules */
....
%%

/* auxiliary routines */
....
```

Definition Part:

The definition part includes information about the tokens used in the syntax definition:

```
%token NUMBER
%token ID
```

The definition part can include C code external to the definition of the parser and variable declarations, within **%{** and **%}** in the first column.

Rules Part:

- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Example Program:

Evaluation of Arithmetic expression using Unmbiguous Grammar(Use Lex and Yacc Tool)

E-> E+T | E-T|T

T->T*F | T/F|F

F-> (E) | id

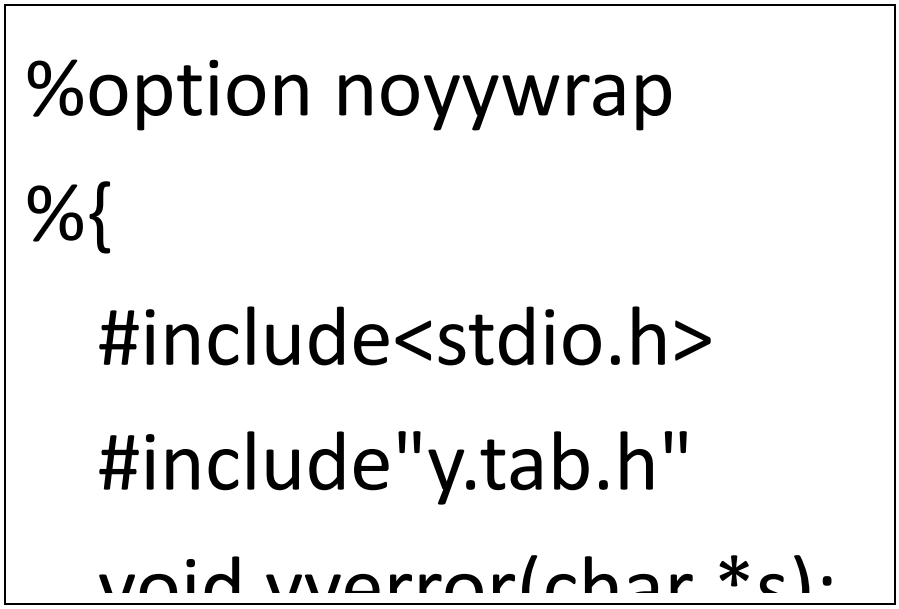


Fig 2.28 Lex Program

```
%{
        #include<stdio.h>
        void yyerror(char*);
        extern int yylex(void);
%}
%token NUM
%%
S:
S E '\n'        {printf("%d\n",$2);}
|
;
E:
E '+' T  {$$=$1+$3;}
|E '-' T {$$=$1-$3;}
|T       {$$=$1;}
T:
T '*' F  {$$=$1*$3;}
|T '/' F {$$=$1/$3;}
|F       {$$=$1;}
F:
'(' E ')'  {$$=$2;}
|NUM  {$$=$1;}
```

Fig 2.29 YACC Program