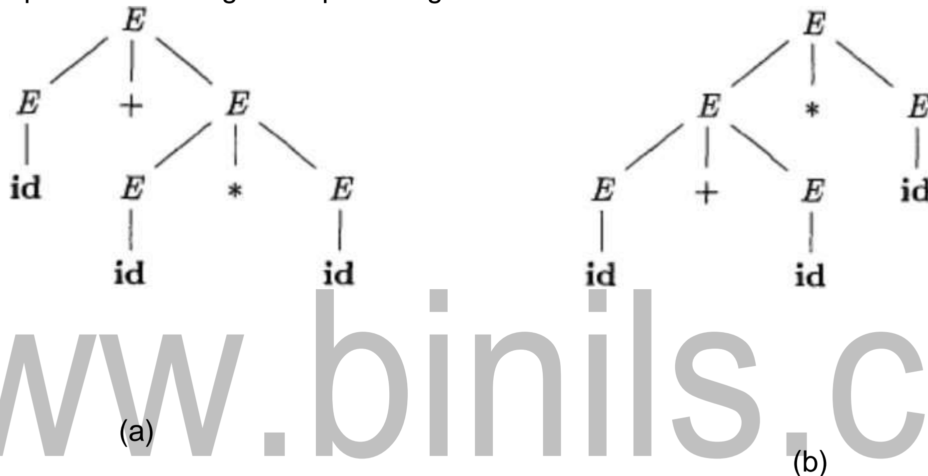
AMBIGUITY in CFGs

Definition: A grammar that produces more than one parse tree for some sentence (input string) is said to be ambiguous.

In other words, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

Or If the right hand production of the grammar is having two non terminals which are exactly same as left hand side production Non terminal then it is said to an ambiguous grammar. Example: If the Grammar is **E-> E+E | E*E | -E| (E) | id** and the Input String is **id + id* id**

Two parse trees for given input string are



www.binils.com

(a) (b)

Two Leftmost Derivations for given input String are :

E => E+E	E => E * E
=> id + E	=> E+E*E
=> id + E * E	=> id + E * E
=> id + id * E	=> id+ id*E
=> id + id * id	=> id + id * id
(a)	(b)

The above Grammar is giving two parse trees or two derivations for the given input string so, it is an ambiguous Grammar

Note: LL (1) parser will not accept the ambiguous grammars or We cannot construct an LL(1) parser for the ambiguous grammars. Because such grammars may cause the Top Down parser to go into infinite loop or make it consume more time for parsing. If necessary we must remove all types of ambiguity from it and then construct.

ELIMINATING AMBIGUITY: Since Ambiguous grammars may cause the top down Parser go into infinite loop, consume more time during parsing.

Therefore, **s**ometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. The general form of ambiguous productions that cause ambiguity in grammars is

$$A \rightarrow A\alpha \mid \beta$$

This can be written as (introduce one new non terminal in the place of second non terminal)

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Example : Let the grammar is **E** → **E+E | E*E | -E| (E) | id .** It is shown that it is ambiguous that can be written as

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

In the above grammar the 1st and 2nd productions are having ambiguity. So, they can be written as

E-> E+E | E*E this production again can be written as
E-> E+E | β , where β is E*E
The above production is same as the general form. so, that can be written as E->E+T|T
T->β

www.binils.com

The value of β is E*E so, above grammar can be written as

1) **E->E+T|T**
2) **T->E*E** **The first production is free from ambiguity** and substitute E->T in the 2nd production then it can be written as
   T-> T*T | -E| (E) | **id** this production again can be written as
   T->T*T | β where β is -E| (E) | id, introduce new non terminal in the Right hand side production then it becomes
   T->T*F | F
   F->-E|(E)|id now the entire grammar turned in to it equivalent unambiguous,

The Unambiguous grammar equivalent to the given ambiguous one is

1) **E → E+T|T**
2) **T → T*F|F**
3) **F → -E|(E)|id**

LEFT RECURSION:

Another feature of the CFGs which is not desirable to be used in top down parsers is left recursion. A grammar is left recursive if it has a non terminal A such that there is a derivation A=>Aα for some string α in (TUV)*. LL(1) or Top Down Parsers cannot handle the Left Recursive grammars, so we need to remove the left recursion from the grammars before being used in Top Down Parsing.

Download Binils Android App in Playstore Download Photoplex App

The General form of Left Recursion is

$$A \rightarrow A\alpha \mid \beta$$

The above left recursive production can be written as the non left recursive equivalent :

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid €$$

Example : - Is the following grammar left recursive? If so, find a non left recursive grammar equivalent to it.

E → E + T | T

T → T * F | F

F → E | (E) | id

Yes ,the grammar is left recursive due to the first two productions which are satisfying the general form of Left recursion, so they can be rewritten after removing left recursion from **E → E + T**, and **T → T * F** is

E → TE′
E′ → +T E′ | €
T → F T′
T′ → *F T′ | €
F → (E) | id

www.binils.com

LEFT FACTORING:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing. A grammar in which more than one production has common prefix is to be rewritten by factoring out the prefixes.

For example, in the following grammar there are n A productions have the common prefix **α**, which should be removed or factored out without changing the language defined for A.

$$A \rightarrow \alpha A1 \mid \alpha A2 \mid \alpha A3 \mid \alpha A4 \mid \ldots \mid \alpha An$$

We can factor out the **α** from all n productions by adding a new A production **A → αA′** and rewriting the A′ productions grammar as

$$A \rightarrow \alpha A'$$
$$A' \rightarrow A1 \mid A2 \mid A3 \mid A4 \ldots \mid An$$

BOTTOM-UP PARSING

Bottom-up parsing corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom nodes) and working up towards the root (the top node). It nvolves -reducing an input string _w' to the Start Symbol of the grammar. in each reduction step, a particular substring matching the right side of the production is replaced by symbol on the left of that production and it is the Right most derivation. For example consider the following Grammar:

E → E+T|T

T → T*F

F → (E)|id

Bottom up parsing of the input string **"id * id "**is as follows:

INPUT STRING	SUB STRING	REDUCING PRODUCTION
id*id	Id	F->id
F*id	T	F->T
T*id	Id	F->id
T*F	*	T->T*F
T	T*F	E->T
E		Start symbol. Hence, the input String is accepted

www.binils.com

Parse Tree representation is as follows:

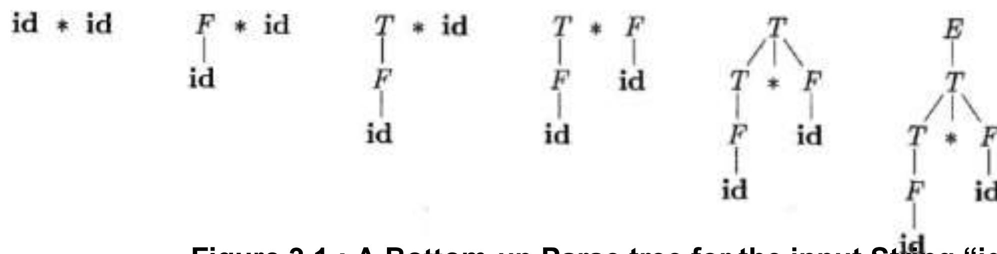


Figure 3.1 : A Bottom-up Parse tree for the input String "id*id"

Bottom up parsing is classified in to 1. Shift-Reduce Parsing, 2. Operator Precedence parsing , and 3. [Table Driven] L R Parsing

i. SLR(1)

ii. CALR(1)

iii.LALR(1)

Download Binils Android App in Playstore Download Photoplex App

SHIFT-REDUCE PARSING:

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed, We use $ to mark the bottom of the stack and also the right end of the input. And it makes use of the process of shift and reduce actions to accept the input string. Here, the parse tree is Constructed bottom up from the leaf nodes towards the root node

When we are parsing the given input string, if the match occurs the parser takes the reduce action otherwise it will go for shift action. And it can accept ambiguous grammars also.

For example, consider the below grammar to accept the input string -id * id–, using S-R parser

E → E+T|T
T → T*F | F
F → (E)|id

Actions of the Shift-reduce parser using Stack implementation

STACK	INPUT	ACTION
$	Id*id$	Shift
$id	*id$	Reduce with F → d
$F	*id$	Reduce with T → F
$T	*id$	Shift
$T*	id$	Shift.
$T*id	$	Reduce with F □id
$T*F	$	Reduce with T□T*F
$T	$	Reduce with E □T
$E	$	Accept

www.binils.com

Consider the following grammar:

S → aAcBe
A → Ab|b
B → d

Let the input string is —abbcde‖. The series of shift and reductions to the start symbol are as follows.

abbcde ⇒ aAbcde ⇒ aAcde ⇒ aAcBe ⇒ S

Note: in the above example there are two actions possible in the second Step, these are as follows :

1. Shift action going to 3rd Step
2. Reduce action, that is A->b

If the parser is taking the 1st action then it can successfully accepts the given input string, if it is going for second action then it can't accept given input string. This is called shift reduce conflict. Where, S-R parser is not able take proper decision, so it not recommended for parsing.

LALR (1) Parsing

LALR refers to the lookahead LR. To construct the LALR (1) parsing table, we use the canonical collection of LR (1) items.

In the LALR (1) parsing, the LR (1) items which have same productions but different look ahead are combined to form a single set of items

LALR (1) parsing is same as the CLR (1) parsing, only difference in the parsing table.
Example

LALR (1) Grammar

1. S → AA
2. A → aA
3. A → b

www.binils.com

Add Augment Production, insert '•' symbol at the first position for every production in G and also add the look ahead.

1. S` → •S, $
2. S → •AA, $
3. A → •aA, a/b
4. A → •b, a/b

I0 State:
Add Augment production to the I0 State and Compute the ClosureL

I0 = Closure (S` → •S)

Add all productions starting with S in to I0 State because "•" is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •S, $
S → •AA, $
Add all productions starting with A in modified I0 State because "•" is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •S, $
S → •AA, $

A → •aA, a/b
A → •b, a/b

I1= Go to (I0, S) = closure (S` → S•, $) = S` → S•, $
I2= Go to (I0, A) = closure (S → A•A, $)

Add all productions starting with A in I2 State because "•" is followed by the non-terminal. So, the I2 State becomes

I2= S → A•A, $
A → •aA, $
A → •b, $

I3= Go to (I0, a) = Closure (A → a•A, a/b)

Add all productions starting with A in I3 State because "•" is followed by the non-terminal. So, the I3 State becomes

I3= A → a•A, a/b
A → •aA, a/b
A → •b, a/b

Go to (I3, a) = Closure (A → a•A, a/b) = (same as I3)
Go to (I3, b) = Closure (A → b•, a/b) = (same as I4)

I4= Go to (I0, b) = closure (A → b•, a/b) = A → b•, a/b
I5= Go to (I2, A) = Closure (S → AA•, $) =S → AA•, $
I6= Go to (I2, a) = Closure (A → a•A, $)

Add all productions starting with A in I6 State because "•" is followed by the non-terminal. So, the I6 State becomes

I6 = A → a•A, $
A → •aA, $
A → •b, $

Go to (I6, a) = Closure (A → a•A, $) = (same as I6)
Go to (I6, b) = Closure (A → b•, $) = (same as I7)
I7= Go to (I2, b) = Closure (A → b•, $) = A → b•, $
I8= Go to (I3, A) = Closure (A → aA•, a/b) = A → aA•, a/b
I9= Go to (I6, A) = Closure (A → aA•, $) A → aA•, $

If we analyze then LR (0) items of I3 and I6 are same but they differ only in their lookahead.

I3 = { A → a•A, a/b
A → •aA, a/b

www.binils.com

A → •b, a/b
}

I6= { A → a•A, $
A → •aA, $
A → •b, $
}

Clearly I3 and I6 are same in their LR (0) items but differ in their lookahead, so we can combine them and called as I36.

I36 = { A → a•A, a/b/$
A → •aA, a/b/$
A → •b, a/b/$
}

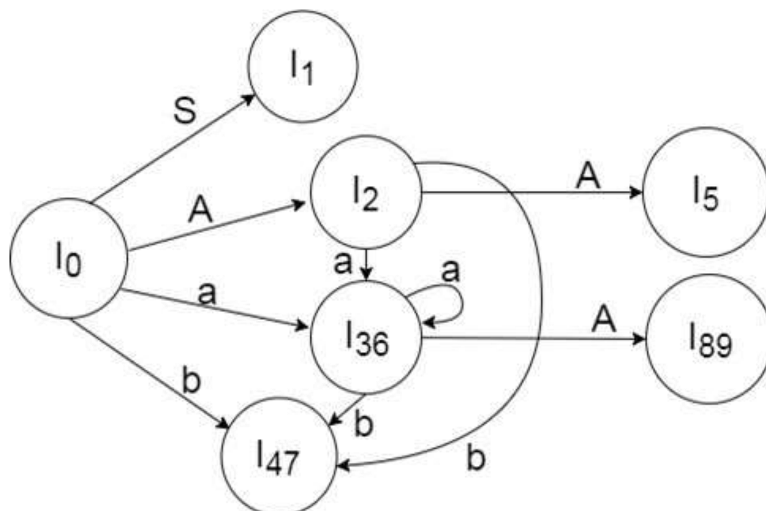
The I4 and I7 are same but they differ only in their look ahead, so we can combine them and called as I47.

I47 = {A → b•, a/b/$}

The I8 and I9 are same but they differ only in their look ahead, so we can combine them and called as I89.

I89 = {A → aA•, a/b/$}

Drawing DFA:



www.binils.com

LALR (1) Parsing table:

States	a	b	$	S	A
I_0	S_{36}	S_{47}		12	
I_1		accept			
I_2	S_{36}	S_{47}			5
I_{36}	$S_{36}S_{47}$				89
I_{47}	R_3R_3	R_3			
I_5			R_1		
I_{89}	R_2	R_2	R_2		

www.binils.com

LRPARSERS

An efficient bottom-up syntax analysis technique that can be used 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. When 'k' is omitted, it is assumed to be 1.

Advantages of LR parsing:

1. It recognizes virtually all programming language constructs for which CFG can be written.
2. It is an efficient non-backtracking shift-reduce parsing method.
3. A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
4. It detects a syntactic error as soon as possible.

Drawbacks of LR method:

It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

www.binils.com

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 as follows:

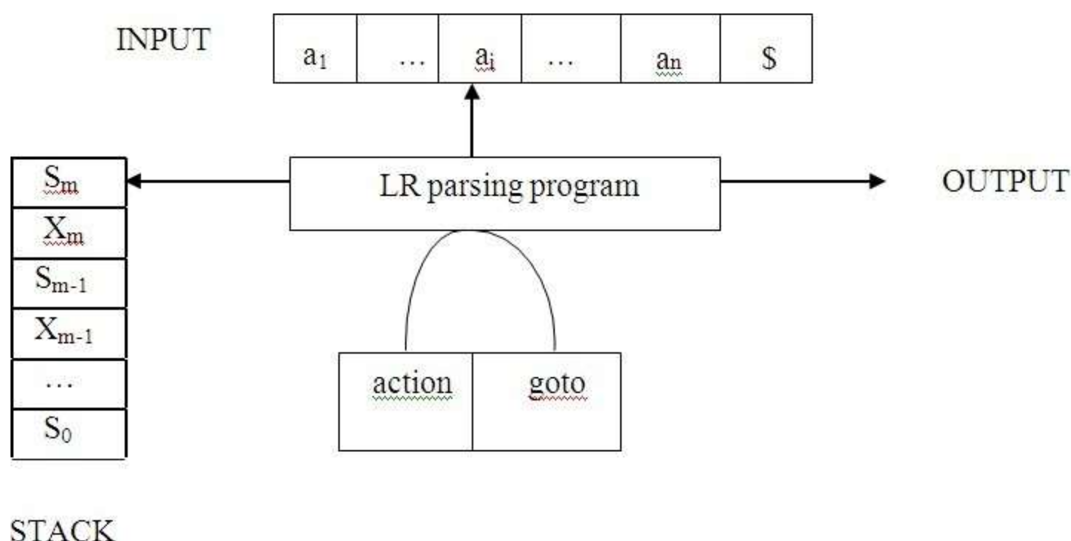


Fig. 2.5 Model of an LR parser

It consists of an input, an output, a stack, a driver program, and a pa parts (action and goto).

- The driver program is the same for all LR parser.
- The parsing program reads characters from an input buffer one at a time.
- The program usesastacktostoreastringoftheforms0X1s1X2s2…Xmsm, where sm is on top. Each Xi is a grammar symbol and each si is a state.
- The parsing table consists of two parts :action and goto functions.

Action: Theparsingprogramdeterminessm,thestatecurrentlyontopofstack,and ai,thecurrentinputsymbol.Itthenconsultsaction[sm,ai]intheactiontablewhichcan have one of four values:

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

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

LR Parsing algorithm:

Input: AninputstringwandanLRparsingtablewithfunctionsactionandgotofor grammarG. Output: IfwisinL(G),abottom-up-parseforw;otherwise,anerror

www.binils.com

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;
if action[s, a] = shift s' then begin
        push a then s' on top of 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;
let s' be the state now on top of the stack; push A then goto[s', A] on top of the stack; output
the production A→β
end
else if action[s, a] =
        accept then
        return
```

CONSTRUCTING SLR(1) PARSINGTABLE:

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

www.binils.com

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→α.Bβ is in closure(I) and B→γ is a production, then add the item B→.γ 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→ αX . β] such that [A→ α . Xβ] 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 fo rG'
3. Constructtheparsingactionfunctionactionandgotousingthefollowingalgorit hmthat requires FOLLOW(A) for each non-terminal ofgrammar.

www.binils.com

Algorithm for construction of SLR parsing table:

Input : An augmentedgrammarG'

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

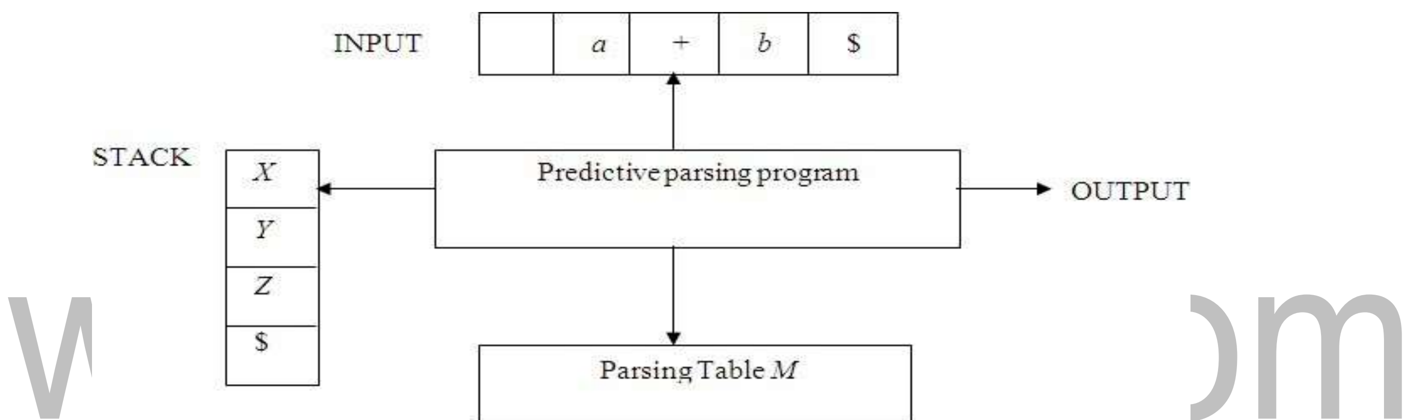
Method :

1. Construct C={I0, I1,….In},the collection of sets ofLR(0) items for G'.
2. State I is constructed from Ii..The parsing functions for state I are determined as follows:
   (a) If [A→α·aβ] is in Ii and goto(Ii,a) = Ij, then set action[i,a] to "shift j". Here a must be terminal.
   (b) If [A→α·] is in Ii , then set action[i,a] to "reduce A→α" for all a inFOLLOW(A).
   (c) If [S'→S.] is in Ii, 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-term If goto(Ii,A) = Ij, 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[S'→.S].

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 figure looks up the production to be applied in parsing table. In what follows, we shall see how the table can be constructed directly from certain grammars.



Model of a nonrecursive 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 endmarker 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 $. 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 parser is controlled by a program that behaves as follows. The program considers X, the symbol on the 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 non terminal, 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). As output, we shall assume that the parser just prints the production used; any other code could be executed here. If M[X,a]=error, the parser calls an error recovery routine.

www.binils.com

Download Binils Android App in Playstore Download Photoplex App

Algorithm for 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 indication.

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

```
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 ip. if X is
   a terminal of $ then
     if X=athen

        pop X from the stack and advance ip
     elseerror()
   else
     if M[X,a]=X->Y1Y2...Yk then begin
       pop X from the stack;
       push Yk,Yk-1...Y1 onto the stack, with Y1 on top;
       output the production X-> Y1Y2...Yk
     end
     else error()
untilX=$        /* stack is empty*/
```

www.binils.com

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G :

1. FIRST
2. FOLLOW

Rules for first( ):

1. If X is terminal, then FIRST(X) is{X}.
2. If X → ε is a production, then add ε toFIRST(X).
3. If X is non-terminal and X → aα is a production then add a toFIRST(X).
4. If X is non-terminal and X → Y1 Y2…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);that is, Y1,….Yi-1=> ε. If ε is in FIRST(Yj) 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 every thing 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).

www.binils.com

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 and3.
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 εis in FIRST(α) and $ is in FOLLOW(A) , add A → α to M[A,$].
4. Make each undefined entry of M beerror.

Example:

Consider the following grammar :

E→E+T|T

T→T*F|F

F→(E)|id

www.binils.com

After eliminating left-recursion the grammar is E →TE'

E' → +TE' | ε

T →FT'

T' → *FT' | ε

F → (E)|id

First( ) :

FIRST(E) ={ (,id}

FIRST(E')={+ , ε}

FIRST(T) = { (,id}

FIRST(T') = {*, ε}

FIRST(F) = { (, id }

Follow( ): FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

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

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

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

Predictive parsing Table

www.binils.com

Download Binils Android App in Playstore Download Photoplex App

NON-TERMINAL	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)		

www.binils.com

stack	Input	Output
$E	id+id*id $	
$E'T	id+id*id $	E→TE'
$E'T'F	id+id*id $	T→FT'
$E'T'id	id+id*id $	F→id
$E'T'	+id*id $	
$E'	+id*id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id*id $	
$E'T'F	id*id $	T→FT'
$E'T'id	id*id $	F→id
$E'T'	*id $	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F→id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

om

Stack Implementation

LL(1) grammar:

Download Binils Android App in Playstore Download Photoplex App

Theparsingtableentriesaresingleentries.Soeachlocationhasnotmorethanoneentry.Thisty pe of grammar is called LL(1)grammar.

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}

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

Parsing table:

Since there are more than one production, the grammar is not LL(1) grammar. Actions performed in predictive parsing:

www.binils.com

1. Shift
2. Reduce
3. Accept
4. Error

Implementation of predictive parser:

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsingtable.
4. Parse the given input string using stack and parsing table.

www.binils.com

SYNTAX ANALYSIS

Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.

Advantages of grammar for syntactic specification :

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.
2. Anefficientparsercanbeconstructedautomaticallyfromaproperlydesignedgrammar.
3. A grammar imparts a structure to a source program that is useful for its translation into object code and for the detection of errors.
4. New constructs can be added to a language more easily when there is a grammatical description of the language.

2.1 THE ROLE OFPARSER

www.binils.com

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

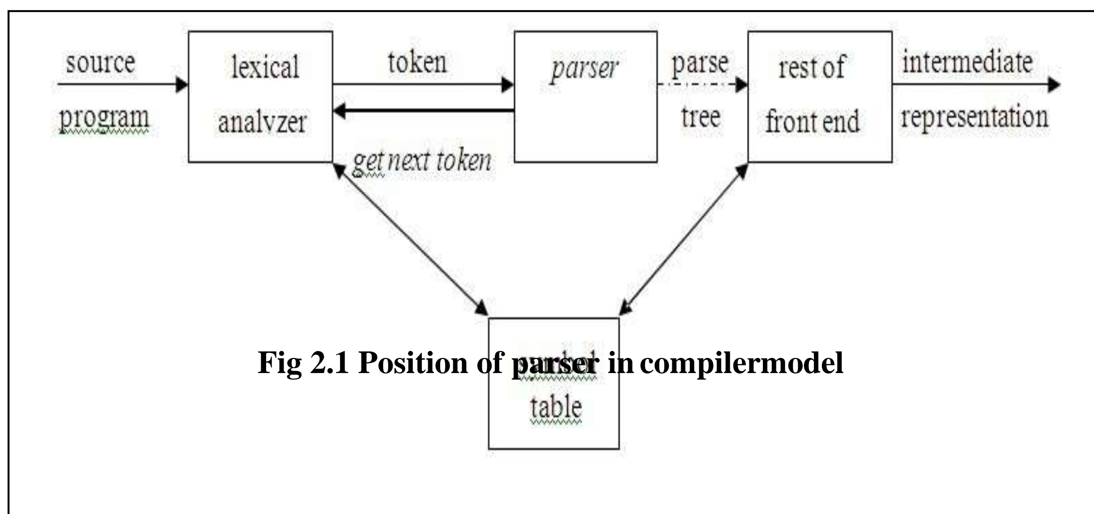


Fig 2.1 Position of parser in compilermodel

Position of parser in compiler model

Functions of the parser :

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs errorrecovery.

Issues :

Parser cannot detect errors such as:

1. Variable re-declaration
2. Variable initialization before use.
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

Syntax error handling :

www.binils.com

Programs can contain errors at many different levels. For example :

1. Lexical, such as misspelling an identifier, keyword oroperator.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.

Functions of error handler :

1. It should report the presence of errors clearly and accurately.
2. Itshouldrecoverfromeacherrorquicklyenoughtobeabletodetectsubsequenterrors.
3. It should not significantly slow down the processing of correct programs.

Error recovery strategies :

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

Download Binils Android App in Playstore Download Photoplex App

Panic mode recovery:

Ondiscoveringanerror,theparserdiscardsinputsymbolsoneatatimeuntil semicolon or end. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

Phrase level recovery:

On discovering an error, the parser performs local correction on the remaining input that allows it to continue. Example: Insert a missing semicolon or delete an extraneous semicolon etc.

Error productions:

www.binils.com

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

Global correction:

Given an incorrect input string x and grammar G, certain algorithms can be used to find a parse tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in generaltoo costly in terms of time and space.