SYNTAX ANALYSIS

Definition of Context – free – Grammar:- [CFG]

A CFG has four components.
1. a set of Tokens known as Terminal symbols.
2. a set of non-terminals
3. start symbol
4. production.

Notational Conventions:-

a) These symbols are terminals. (Ts)
- (i) Lower case letters early in the alphabet such as a,b,c
- (ii) Operator symbols such as +, -, etc.
- (iii) Punctuation symbols such as parenthesis, comma etc.
- (iv) The digits 0, 1, 2, 3, …, 9
- (v) Bold face Strings.

b) These symbols are Non-Terminals (NTs)
- (i) Upper case letters early in the alphabet such as A, B, C
- (ii) The letter S, which is the start symbol.
- (iii) Lower case italic names such as expr, stmt.

c) Uppercase letters such as X, Y, Z represent grammar symbols either NTs or Ts.

PARSER:

A parser for grammar G is a program that takes a string W as input and produces either a parse tree for W, if W is a sentence of G or an error message indicating that W is not a sentence of G as output.

There are two basic types of parsers for CFG.
1. Bottom – up Parser
2. Top – down Parser

1. **Bottom up Parser**:-

The bottom up parser build parse trees from bottom (leaves) to the top (root). The input to the parser is being scanned from left to right, one symbol at a time. This is also called as "Shift *Reduce Parsing*" because it consisting of shifting input symbols onto a stack until the right side of a production appears on top of the stack.

There are two kinds of shift reduce parser (Bottom up Parser)
1. Operator Precedence Parser
2. LR Parser (move general type)

Designing of Shift Reduce Parser(Bottom up Parser) :-

Here let us "reduce" a string w to the start symbol of a grammar. At each step a string matching the right side of a production is replaced by the symbol on the left.

For ex. consider the grammar,

S → aAcBe
A → Ab/b
B → d

and the string abbcde.

We want to reduce the string to S. We scan abbcde, looking for substrings that match the right side of some production. The substrings b and d qualify.

Let us choose the left most b and replace it by A. That is A→Ab/b

So, S →abbcde
aAbcde (A → b)

We now that Ab,b and d each match the right side of some production.

Suppose this time we choose to replace the substring Ab by A, in the left side of the production.

A → Ab

We now obtain,

aAcde (A →Ab)

Then replacing d by B

aAcBe (B → d)

Now we can replace the entire string by S.

W = abbcde	position	production
abbcde	2	A→Ab/b (that is, A→b)
aAbcde	2	A→Ab
aAcde	4	B→d
aAcBe		S→aAcBe

Thus we will be reached the starting symbol S.

Each replacement of the right side of a production by the left side in the process above is called a reduction.

In the above example abbcde is a right sentential form whose handle is,

A→b at position 2

A→Ab at position 2

B→d at position 4.

Example:- Consider the following grammar

E→E+E

E→E*E

E→(E)

E→id   and the input string $id_1+id_2*id_3$. Reduce to the start symbol E.

Solution:-

Right sentential form	handle	Reducing Production
$id_1 + id_2 * id_3$	id_1	E→id
$E + id_2 * id_3$	id_2	E→id
$E + E * id_3$	id_3	E→id
E + E * E	E*E	E→E*E
E+E	E+E	E→E+E
E		

Stack implementation of shift reduce parsing:

Initialize the stack with $ at the bottom of the stack. Use a $ the right end of the input string.

Stack	Input String
$	w$

The parser operates by shifting one or more input symbols onto the stack until a handle β is on the top of a stack.

Example:- Reduce the input string $id_1+id_2*id_3$ according to the following grammar.

1. E→E*E
2. E→E+E
3. E→(E)
4. E→id

Solution:-

Stack	Input String	Action
$	$id_1+id_2*id_3$$	shift
$$id_1$	$+id_2*id_3$$	E→id(reduce)
$E	$+id_2*id_3$$	shift
$E+	$id_2*id_3$$	shift

$E+id$_2$	*id$_3$$	E→id(reduce)

Stack	Input String	Action
$E+E	*id$_3$$	shift
$E+E*id$_3$	$	E→id(reduce)
$E+E*E	$	E→E*E(reduce)
$E+E	$	E→E+E(reduce)
$E	$	Accept

The Actions of shift reduce parser are,

1. shift → Shifts next input symbol to the top of the stack
2. Reduce → The parser knows the right end of the handle which is at the top of the stack.
3. Accept → It informs the successful completion of parsing
4. Error → It detects syntax error then calls error recovery routine.

Operator Precedence Parsing;-

In operator precedence parsing we use three disjoint relations.

< if a < b means a "yields precedence to" b

= if a = b means a "has same precedence as" b

> if a > b means a "takes precedence over" b

There are two common ways of determining *precedence relation* hold between a pair of terminals.

1. Based on associativity and precedence of operators
2. Using operator precedence relation.

For Ex, * have higher precedence than +. We make + < * and * > +

Problem 1:- Create an operator precedence relation for id+id*id$

	id	+	*	$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	-

Problem 2: <u>Tabulate the operator precedence relation for the grammar</u>

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

<u>Solution:-</u>

Assuming 1. ↑ has highest precedence and right associative

2. * and / have next higher precedence and left associative

3. + and – have lowest precedence and left associative

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>	-	-	>	>
(	<	<	<	<	<	<	<	=	-
)	>	>	>	>	>	-	-	>	>
$	<	<	<	<	<	<	<	-	-

Derivations:-

The central idea is that a production is treated as a rewriting rule in which the non-terminal in the left side is replaced by the string on the right side of the production.

For Ex, consider the following grammar for arithmetic expression,

E→E+E | E*E | (E) | -E |id

That is we can replace a single E by –E. we describe this action by writing

E => -E , which is read "E derives –E"

E→(E) tells us that we could also replace by (E).

So, E*E => (E) * E or E*E => E* (E)

We can take a single E and repeatedly apply production in any order to obtain sequence of replacements.

E => -E

E => -(E)

E => -(id)

We call such sequence of replacements is called *derivation*.

Suppose $\alpha\ A\ \beta \Rightarrow \alpha\ \gamma\ \beta$ then

A→ γ is a production and α and β are arbitrary strings of grammar symbols.

If $\alpha_1 \Rightarrow \alpha_2 \ldots\ldots \Rightarrow \alpha_n$ we say $\alpha 1$ derives α_n

The symbol,

=> means " derives in one step"

=> means "derives zero or more steps"

=> means "derives in one or more steps"

Example:- E→E+E | E*E | (E) | -E |id. The string –(id+id) is a sentence of above grammar.

E => -E
=> -(E)
=> -(E+E)
=> - (id+E)
=> -(id+id)

The above derivation is called left most derivation and it can be re written as,

E => -E

=> -(E)

=> -(E+E)

=> - (id+E)

=> -(id+id)

we can write this as E => -(id+id)

Example for Right most Derivation:-

Right most derivation is otherwise called as *canonical derivations*.

E => -E

=> -(E)

=> -(E+E)

=> - (id+E)

=> -(id+id)

References

1. J. Tremblay, P.G. Sorenson, “The Theory and Practice of Compiler Writing “, McGRAW-HILL,1985.
2. W.M. Waite, L.R. Carter, “An Introduction to Compiler Construction”, Harper Collins, New york,1993
3. A.W. Appel,"Modern Compiler Implementation in, CambridgeUniversity Press,1998
4. Internet Papers
5. Aho, R. Sethi, J.D. Ullman," Compilers- Principles, Techniques and Tools"Addison-Weseley, 2007