Assignment - IV

1. What is a parser and explain detail about types of parsers?

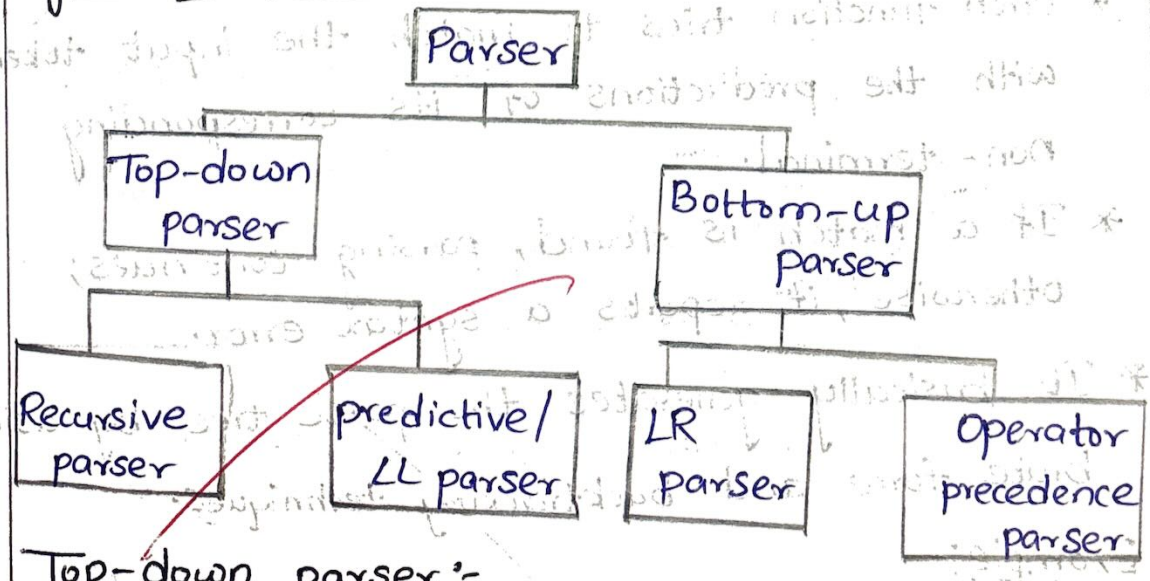
parser:-

The parser is one of the phases of the compiler which takes a token of string as input and converts it into the corresponding Intermediate Representation with the help of an existing grammer. The parser is also known as syntax analyzer.

Functions of parser:-

1) syntax analysis
2) parse tree construction
3) Error detection and Recovery

Types of parser:-



Top-down parser:-

* Top-down parser is the parser that generates parse tree for the given input string

with the help of grammer productions by expanding the non-terminals.

* It starts from the start symbol and ends down on the terminals.

* It uses left most derivation.

* The top-down parser is again classified into two types:

  1) Recursive parser

  2) non-recursive / LL parser

1) Recursive parser:-

A recursive descent parser is a type of top-down parser built from a collection of recursive procedures, one for each non-terminal in the grammer.

* It starts with the start symbol and calls functions recursively to expand non-terminals.
* Each function tries to match the input tokens with the productions of its corresponding non-terminal.
* If a match is found, parsing continues; otherwise, it reports a syntax error.

* It basically generates the parse tree by using brute force and backtracking techniques.

Example:-

$E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$

$T \rightarrow id$

The input string is id+id

```
E()
{
 T()
 Eprime()
}

Eprime()
{
 if (input == '+')
  {
   iput++;
   T()
   Eprime()
  }
else
  return
}

T()
{
 if (input == 'id')
 {
  input++;
 }
}
```

$E \rightarrow TE'$

$E \rightarrow idE'$

$E \rightarrow id+TE'$

$E \rightarrow id+idE'$

$E \rightarrow id+id\varepsilon$

$E \rightarrow id+id$

Characteristics:-

1) Simple to implement
2) works well for small and simple grammers
3) can suffer from backtracking if the grammer is ambigous or not well-designed.
4) cannot handle left recursion.

2) predictive / LL parser:-

* A predictive parser is a type of a top-down parser used in compiler design. It predicts which production rule to use by looking at the next input symbol without backtracking.
* It uses first and follow sets, it is help in deciding which production to choose.
* predictive parsing generally works for LL grammers (Left-to-right scanning, Leftmost derivation).
* The predictive parser have 5 steps:
  i) Eliminate left recursion
  ii) Left factoring
  iii) find first and follow
  iv) construct parser table
  v) parse the input string

Example:- To find follow:-

* If the non-terminal comes before end of the non-terminal then we have to consider first of the next non-terminal and follow of the left terminal.
* If the non-terminal comes end then follow the grammer.

Example:-

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

Step-1:- Eliminate Left recursion

$E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \epsilon$

$F \rightarrow (E) / id$

Step-2:- Left factoring

No need

Step-3:- find first and follow

First (E) - { (, id }	Follow (E) - { $,) }
First (E') - { +, ϵ }	Follow (E') - { $,) }
First (T) - { (, id }	Follow (T) - { $, +,) }
First (T') - { *, ϵ }	Follow (T') - { $, +,) }
First (F) - { (, id }	Follow (F) - { $, *, +,) }

Step-4:- construct parser table

	+	*	(	)	id	$
E			$E \rightarrow TE'$		$E \rightarrow TE'$	
E'	$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$		$E' \rightarrow \epsilon$
T			$T \rightarrow FT'$		$T \rightarrow FT'$	
T'	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$		$T' \rightarrow \epsilon$
F			$F \rightarrow (E)$		$F \rightarrow id$	

Step-5

The input string will be

I/p - id + id * id

Stack	Input string	output
$E	id+id*id$	E→TE'
$E'T	id+id*id$	T→FT'
$E'T'F	id+id*id$	F→id
$E'T'id	id+id*id$	
$E'T'	+id*id$	T'→ϵ
$E'	+id*id$	E'→+TE'
$E'T+	+id*id$	
$E'T	id*id$	T→FT'
$E'T'F	id*id$	F→id
$E'T'id	id*id$	
$E'T'	*id$	T'→*FT'
$E'T'F*	*id$	
$E'T'F	id$	F→id
$E'T'id	id$	
$E'T'	$	T'→ϵ
$E'	$	E'→ϵ
$	$	

2) Bottom-up parser:-

* Bottom-up parser is the parser that generates the parse tree for the given input string with the help of grammer productions by compressing the terminals. It starts from terminals and ends upon the start symbol.
* It uses the rightmost derivation in reverse order.

* Bottom-up parser is classified into two types:
  1) precedence
  2) LR

1) operating precedence:-

* Operator Precedence Parser generates the parse tree from the given grammer and string but it has the condition that two consecutive non-terminals and epsilon will never appear on the right-hand side of any production.
* The operator precedence parsing technique is applied to operator grammers.
* An operator precedence grammer is a context-free grammer that has the property that no-production has either:
  1) An empty on the right-hand side
  2) Two adjacent non-terminals in its right-hand side.
* The steps in precedence parser are:
  1) check the grammer is it in OP or not.
  2) construct op relation table
  3) parse tree I/P string
  4) generate parse tree

Example:-

E→E+E/E*E/id

The input string is id+id*id.

Step-1:-

The grammer is in operation operator precedence.

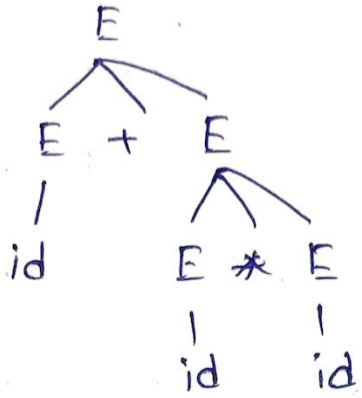
Step-2:- construct op relation table

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

Step-3:-

Stack	relation	input string	operation
$	<	id+id*id$	shift id
$ id	>	+id*id$	reduce E→id
$ E	<	+id*id$	shift +
$ E+	<	id*id$	shift id
$E+id	>	*id$	reduce E→id
$E+E	<	*id$	shift *
$ E+E*	<	id$	shift id
$ E+E*id	>	$	shift id
$ E+E*E	>	$	reduce E→id
$ E+E*E	>	$	reduce E→E*E
$ E+E	>	$	reduce E→E+E
$ E		$	accepted

Step-4:- Parse tree



2) LR technique:-

* This is a bottom-up parser that generates the parse tree for the given string by using unambigous grammer.

*It follows the reverse of the rightmost derivation.

*LR parser classified into two types:

1) LR
2) SLR
3) LALR
4) CLR

Example:-

E→E+T

E→T

T→T*F

T→F

F→(E)

F→id

The input string is id*id+id

Augmented grammer:-

$E' \to E$

$E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to (E)$

$F \to id$

Cannonical LR grammer:-

$E' \to \cdot E$

$E \to \cdot E+T$ — (1)

$E \to \cdot T$ — (2)

$T \to \cdot T*F$ — (3)

$T \to \cdot F$ — (4)

$F \to \cdot (E)$ — (5)

$F \to \cdot id$ — (6)

goto (I_0, E)

$I_1 = E' \to E\cdot$

$E \to E\cdot + T$

goto (I_0, T)

$I_2 = E \to T\cdot$

$T \to T\cdot * F$

goto (I_0, F)

$I_3 = T \to F\cdot$

goto $(I_0, ()$

$I_4 = F \to (\cdot E)$

$E \to \cdot E+T$

$E \to \cdot T$

$T \to \cdot T*F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

goto (I_0, id)

$I_5 = F \to id\cdot$

goto $(I_1, +)$

$I_6 = E \to E+\cdot T$

$T \to \cdot T*F$

T → .F

F → .(E)

F → .id

goto (I_2, *)

I_7 = T → T*.F

F → .(E)

F → id

goto (I_4, E)

I_8 = F → (E.)

E → E.+T

goto (I_6, +)

I_9 = E → E+T.

T → T.*F

goto (I_7, F)

I_{10} = T → T*F.

goto (I_8,))

I_{11} = F → (E).

SLR :-

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

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

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

State	action						goto		
	+	*	(	)	id	$	E	T	F
0			s_4		s_5		1	2	3
1	s_6					accept			
2	r_2	s_7		r_2		r_2			
3	r_4	r_4		r_4		r_4			
4			s_4		s_5		8	2	3
5	r_6	r_6		r_6		r_6			
6			s_4		s_5			9	3
7			s_4		s_5				10
8	s_6			s_{11}					
9	r_1	s_7		r_1		r_1			
10	r_3	r_3		r_3		r_3			
11	r_5	r_5		r_5		r_5			

LALR:-

Stack	Input string	Action
$0	id * id + id $	shift id5
$id5	* id + id $	reduce F→id |X2=2
$0F3	* id + id $	reduce T→F |X2=2
$0T2	* id + id $	shift *7
$0T2*7	id + id $	shift id5
$0T2*7id5	+ id $	reduce F→id |X2=2

$0T2*7F10	+id$	reduce F→T*F 3*2=6
$0T2	+id$	reduce E→T 1×2=2
$0E1	+id$	shift +6
$0E1+6	id$	shift id5
$0E1+6id5	$	reduce F→id 1×2=2
$0E1+6F3	$	reduce T→F 1×2=2
$0E1+6T9	$	reduce E→E+T 3*2=6
$0E1	$	Accept

2. Explain detail about intermediate code generation and 3 address codes.

Intermediate code generation:-

*Intermediate code generation is a stage in the process of compiling a program, where the compiler translates the source code into an intermediate representation.

* This representation is not machine code but is simpler than the original high-level code.

* How it works:

1) Translation: The compiler takes the high-level code and converts it into an intermediate form, which can be easier to analyze and manipute.

2) Portability: This intermediate code can often run on different types of machines without

needing major changes, making it more versatile.

* optimization - Before turning it into machine code, the compiler can optimize this intermediate code to make the final program run faster or use less memory.

* The front end of the compiler translates a source program into an independent intermediate code, then the back end of the compiler uses intermediate code to generate the target node.

* If we generate machine code directly from source code then for n target machine we will have optimizers and n code generator but if we will have a machine independent intermediate code, then we need only one optimizer.

* There are three notations used in Intermediate code generation:

1) Infix notation
2) prefix notation
3) postfix notation

1) Infix notation:-

* Infix notation is the traditional way of writing arithmetic and logical expressions, where operands are placed between operands.

* In the context of, intermediate code generation in compilers, infix notation is primarily a

source-level representation and is often contrasted with postfix and prefix forms, which are better suited for machine processing and code optimization.

Example:-

(a+b)*c

problems:-

a) Ambiguity arises without parantheses

b) Harder for compilers to evaluate directly.

c) Requires operators precedence and associativity rules.

prefix notation:-

* prefix notation is a way of writing arithmetic and logical expressions where the operator comes before its operands.

* No parantheses are needed.

* In prefix notation, each operator is written first, followed by its operands.

* The main benefit is that every expression can be parsed in a single, left-to-right precedence or associativity.

Example:-

A+(B*C)

Prefix notation- +A*BC

3. Postfix Notation:-

* It is also known as reverse polish notation or suffix notation.
* In postfix notation the operator are at the right end.
* postfix notation eliminates the need for paranthesis, as the operator's position and arity allow unambigous expression decoding.
* In postfix notation, the operator consistently follows the operand.

Example:-

(a-b)*(c+d)+(a-b) is

ab-cd+*ab-+

Three address code generation:-

* A three address statement involves a maximum of three references, consisting of two for operands and one for the result.
* A sequence of three address statements collectively forms a three address code.
* The typical form of a three address statement is expressed as x=y op z, where x, y and z represent memory address.
* Each variable (x, y, z) in a three address statement is associated with a specific memory location.

* There are 3 ways to represent a three-address code in compiler design:

1) Quadruples
2) Triples
3) Indirect triples

1) Quadruples:-

It is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage:-

1) Easy to rearrange code for global optimization.
2) one can quickly access value of temporary variables using symbol table.

Disadvantage:-

1) Contains lot of temporaries
2) Temporary variable creation increases time and space complexity.

Ex:-

$(x+y) * (y+2) + (x+y+2)$

The three address code is

$t_1 = x+y$

$t_2 = y+2$

3) $t_3 = t_1 * t_2$

4) $t_4 = t_1 + 2$

5) $t_5 = t_3 + t_4$

Quadruple representation:-

operator	src1	src2	Result
+	x	y	t_1
+	y	z	t_2
=	t_1	t_2	t_3
+	t_1	2	t_4
+	t_3	t_4	t_5

2) Triples:-

This representation doesn't make use of extra temporary variable to represent a single operation insted when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage:-

1) Temporaries are implicit and difficult to rearrange code.

2) It is difficult to optimize involves moving intermediate code. when a triple is moved, any other triple reffering to it must be updated also with help of pointer one can directly

access symbol table entry.

Example:-

$(x+y) * (y+2) + (x+y+z)$

The three address code:

1) $t_1 = x+y$ — 1

2) $t_2 = y+2$ — 2

3) $t_3 = t_1 * t_2$ — 3

4) $t_4 = t_1+2$ — 4

5) $t_5 = t_3+t_4$ — 5

Triples representation:-

Operator	Src 1	Src 2
+	x	y
+	y	2
*	(1)	(2)
+	y)	2
+	(3)	(4)

3) Indirect tuples:-

This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Advantages Of three address code:-

* This makes it easy to translate into assembly /machine code.
* Simplifies code generation
* Easier for humans to understand compared to raw machine code.
* Each step of computation is clearly shown.
* Easier register allocation.