"Bottom up parser"

Bottom up parsing is a type of syntax analysis method where the parser starts from input symbols and attempts to reduce them to the start symbol of the grammar.

It can be implement by —

- Start with token :- The parsers begin with terminal symbols(input) which are leaves of the parse tree. For this usually a stack is used. stack keep track of what's been processed and a input buffer hold the remaining input.
- Shift and Reduce(Action): The parser repeatedly applies two actions shift and reduce. shift means the next token is pushed onto a stack and reduce means a sequence of symbols on the stack is replaced by a non-terminal according to production rule of the grammer. For this usually using parse table (LR, SLR, CLR), guides the shift reduce decision.
- Repeat until root: The process of shifting and reducing continues until the entire input is reduced to the start symbol.

It is also known as shift-Reduce Parser. Because it uses two main actions :—

Shift: Read the next input symbol and put in on a stack

Reduce: Replace a set of symbols on the stack with the non terminal on the left hand side of that rule.

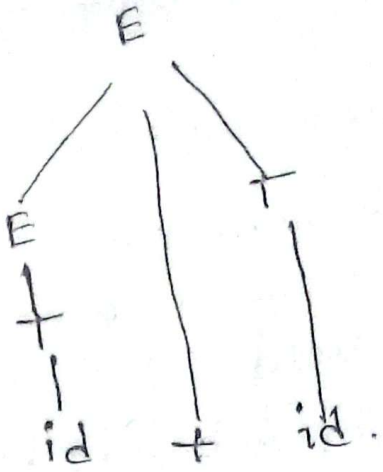
Thats why it called shift-Reduce - it keeps shifting symbols and reduce them to grammer rule.

For example :-

$E \to E + T$

$E \to T$

$T \to id$



stack	input	Action.
	id + id	shift id.
id	+ id	Reduce $T \to id$
T	+ id	Reduce $E \to T$
E	+ id	shift +
E +	id	shift id
E + id		Reduce $T \to id$
E + T		Reduce $E \to E + T$
E		accept

$S \to T$ ①

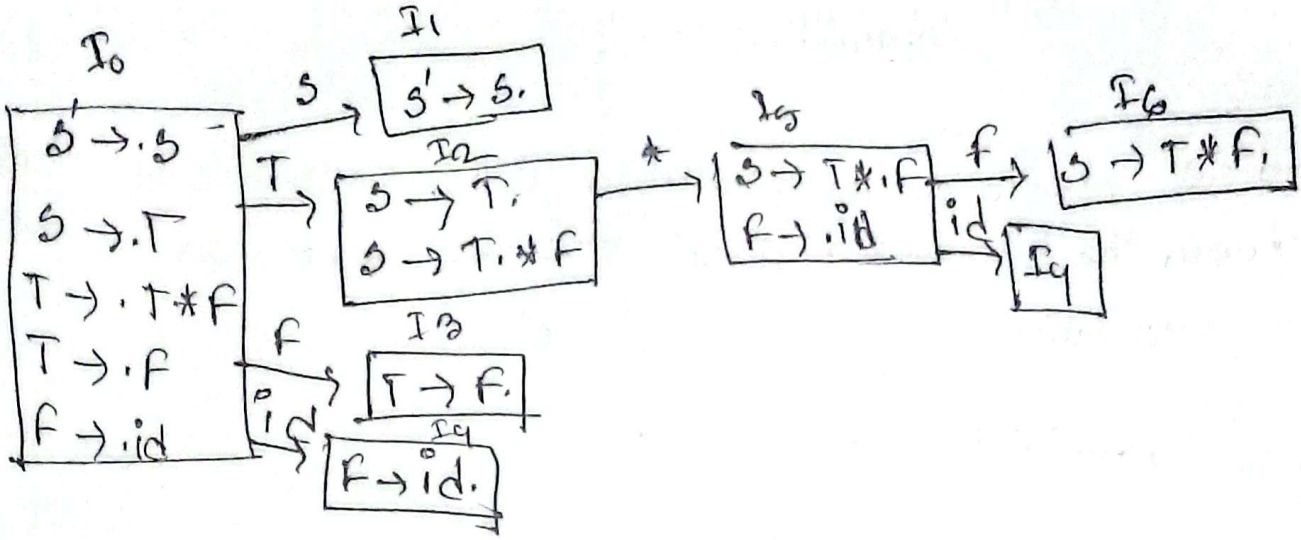
$T \to T * f$ ②

$T \to f$ ③

$f \to id$ ④

SLR(1) or not?

From SLR(1) (Simple LR we can avoid the problem of shift-reduce conflict in a parse table.



parse table:

	Action			go to		
	*	id	$	S	T	F
0		s4		1	2	3
1			Acc			
2	s5		r_1			
3	r_3	r_3	r_3			
4	r_4	r_4	r_4			
5		s4				6
6	r_2	r_2	r_2			

we can see shift-reduce conflict in I_2.So —

Follow(S) = {$}
Follow(T) = {*, $}
Follow(F) = {$, *}

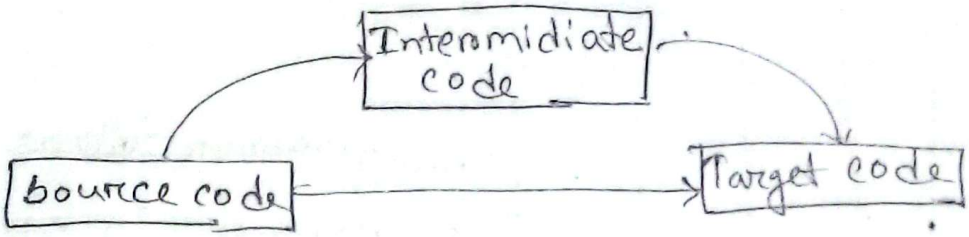
∴ The ~~grammer follows~~ is not
The grammer is SLR(1).

"Intermidiate code"

Intermidiate code is a representation of the source code that sits between the high-level language and the target machine code during compilation.

Why is intermidiate code is used?

If a compiler directly translates source code to machine code, then for each new machine a separate full compiler is needed. Using intermidiate code solves this problem by separating the compiler into two parts – Analysis phase (front-end) and synthesis phase (Back-end)



For example –

we hire one translator for each taget language.

English → Beg Bengali, English → Hindi, English → Chinese.

It take lot of works

Use intermidiate language –

English → one translator, one translator → Bengali, Hindi etc.

In this way only second half changes, first remains same. Intermidiate code works as the same way –

Type used in intermidiate code:—

1 **Postfix Notation:** operators come after operands. Example—

(a + b) * c

ab+c* (Postfix) No need for brackets.

2 **Three-Address code:** Each instruction has at most 3 parts. like: result = operand 1 op operand 2.

Example: a+b*c

$T_1 = b * c$ [T_1, T_2 temporary variable]

$T_2 = a + T_1$

~~Two three ways~~

There are two ways to represent Three-Address code:—

Quadruple (Use temporary variable)

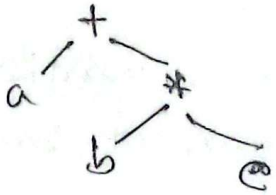
	op	arg 1	arg 2	result
(1)	*	b	c	T_1
(2)	+	a	T_2	T_2

Triples (No use temporary variable)

	op	arg 1	arg 2
0	*	b	c
1	+	a	(0)

3. **Syntax tree:** This is tree diagram for an expression.

a+b*c



Useful:

1. No need to build a full compiler for every machine.
2. Easier to improve performance.
3. Reduce duplication of work.
4. One source code can run on many machines

SDT (Syntax Directed Translation)

SDT require some information to convert the parse tree into a code. This information cannot be represented by by efg, hence attribute are attached to the variables.
We can represent it as:-

Grammer + semantic rule = SDT

Example:

1. $E \rightarrow E+T$ { E.val := E.val + T.val }
2. $E \rightarrow T$ { E.val := T.val }

Semantic analysis:- It is the third phase of compiler. Semantic analysis make sure that declarations and statements of program are semantically correct.

For example:

```
int a;
a = "text";
```

Syntax is correct but semantically wrong. Because of assigning a string to an int. Semantic analysis catches this errors.

Errors recognized by semantic analyzer are as follows:-

- Type mismatch.
- Undeclared variables.
- Reserved identifier misuse

Type checking: It is an important part of semantic analysis. make sure operators and operands match.

Example: int a = "text" error

Undeclared variable:- Make sure all variables are declared before use. Example x = 5; error; x not declared.

Reserved identifier Misuse: Don't use keyword as variable names.
Example: int if = 10; error.

"Symbol table".

Compiler use a symbol table to track all variables, functions and identifiers in a program. It stores information such as, the name type, memory location of each identifier. for example:-

```
int x = 5;
float y = 3.14
x = x+2
```

Name	type	Value	memory location
x	int	5	1001
y	float	3.14	1002

Operations on symbol table:—

4 operation perform —

Insert: Add a new symbol (when variable is declared)

Lookup: search for a symbol (when used in expression)

Update: change information (new value or type)

Delete: Remove a symbol (after a function ends)

Use Perpose:

1. store information about variable, function, Location.
2. check correctness (semantic errors)
3. Help in code generation.

"Code Optimization"

Code optimization is the phase in a compiler where the intermediate code is improved to make the final program faster, use less memory and more efficient.

Example:

Before optimization

```
int a = 5;
int b = 5;
int c = a+b
```

after optimization

```
int c = 10;
```

Local vs Global optimizations

Local: Done inside a single basic block, it is fast and simple

```
int c = 2
int b = 2
int c = a+b  => c = 4 (constant folding)
```

Global:- Done accross multiple blocks. more complex but more powerful.

```
int x = a+b    // used in one block
...
int y = a+b    // used in another block.
```

Source of Optimization:-

These are main ideas behind code optimization.

1. Eliminate repeated calculation. Ex- x = a+b; y = a+b (Reuse result)
2. Remove calculations not used later.
3. Avoid too many new variables. (Reuse temporary variable)
4. Use faster operation (Replace x = x*2 with x = x<<1)
5. Remove code that never run. [Ex:- if (false) { }]

Optimization two types:

- Machine independent:- Improve the intermediate code without caring about the target machines architechture like CPU, memory, register.

- Machine Dependent:- Optimizes target code specifically for the machines architechture (registers, memory)

"Previous solve"

⑤ Postfix notation for the infix

if a then if c-d then a+c else a*c else a+b

Solve:

If a then

if c-d then a+c
else a*c

} Inner if.

else a+b.

For inner if:—

if c-d then a+c.
else a*c.

Postfix: cd-ac+ac*?:

For outher if.

If a then [inner if postfix] else a+b

Postfix: a [inner if postfix] ab+?:

Now. for the given ~~no~~ infix notation postfix will be:

acd-ac+ac+?:ab+?:

(*) $S \rightarrow id = E$

$E \rightarrow E - T$

$E \rightarrow T$.

$T \rightarrow T/F$

$T \rightarrow F$

$F \rightarrow id$.

Using SDT produce three address code for the statement "x = a-b/c"

Solve:

$S \rightarrow id = E$ {gen(id.name = E.place)}

$E \rightarrow E - T$ { E.place = new temp(); gen(E.place = Eplace - Tplace) }

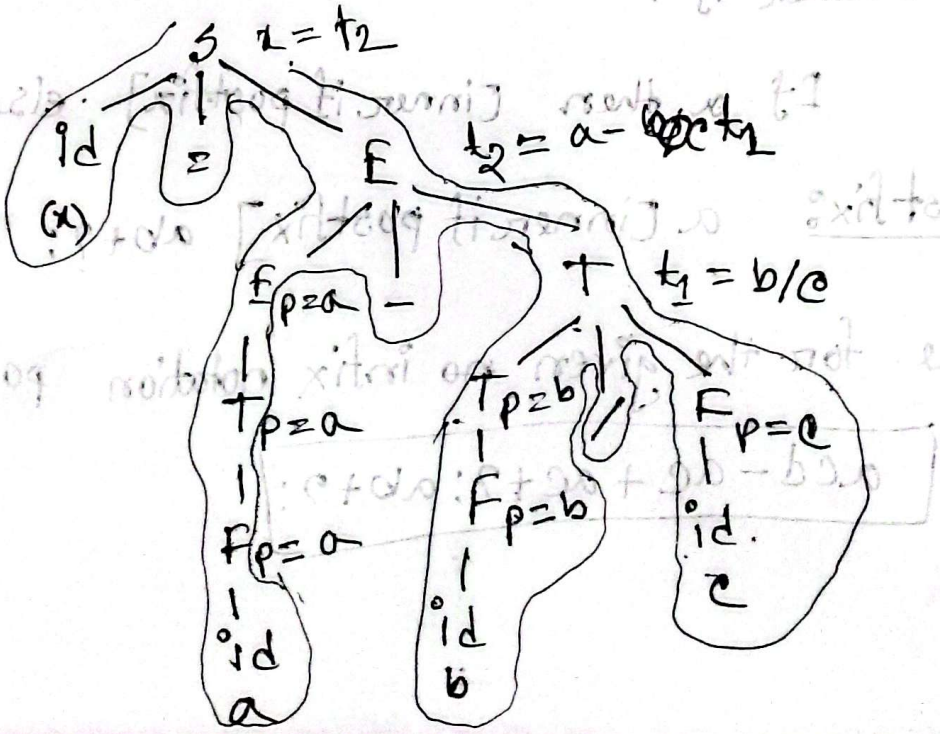
$E \rightarrow T$ { gen(E.place = T.place) }

$T \rightarrow T/F$ { Tplace = new temp(), gen(Tplace = Tplace / Fplace) }

$T \rightarrow F$ { T.place = F.place }

$F \rightarrow id$. { F.place = id.name } :

"x = a-b/c"



(*) E → E*T$.

E → T

T → T+F

T → F

F → id.

using SDT produce postfix notation for the expression "a+b*c"

Solve:

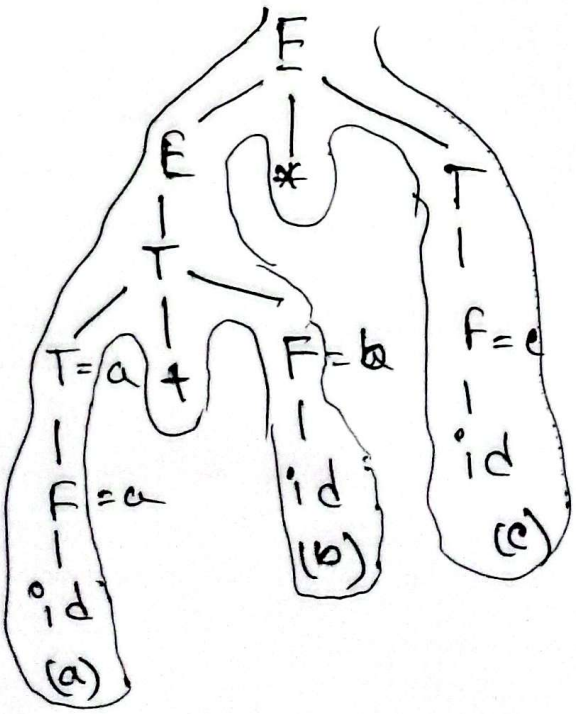
E → E*T { printf("*"); }

E → T { }

T → T+F { printf("+"); }

T → F { }

F → id { printf(num.lval); }



ab+c*

[postfix notation for given grammer]