

Unit-III Syntax Analysis

Need and Role of the Parser-Context Free Grammars -Top Down Parsing -General Strategies-Recursive Descent Parser Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0)Item-Construction of SLR Parsing Table -Introduction to LALR Parser - Error Handling and Recovery in Syntax Analyzer-YACC-Design of a syntax Analyzer for a Sample Language .

SYNTAX ANALYSIS

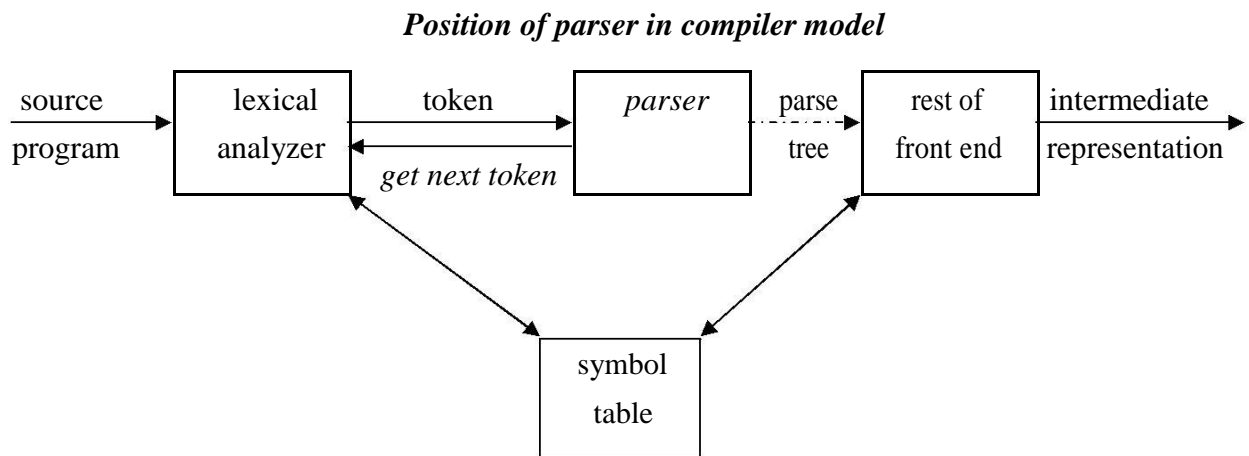
Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.

Advantages of grammar for syntactic specification :

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.
2. An efficient parser can be constructed automatically from a properly designed grammar.
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.

THE ROLE OF PARSER

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.



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 error recovery.

Issues handled by Semantic Analysis phase are

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

Syntax error handling :

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

1. Lexical, such as misspelling a keyword.
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. It should recover from each error quickly enough to be able to detect subsequent errors.
3. It should not significantly slow down the processing of correct programs.

Error recovery strategies :

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

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

Panic mode recovery:

On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as 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:

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 general too costly in terms of time and space.

CONTEXT-FREE GRAMMARS

A Context-Free Grammar is a quadruple that consists of **terminals**, **non-terminals**, **startsymbol** and **productions**.

Terminals : These are the basic symbols from which strings are formed.

Non-Terminals : These are the syntactic variables that denote a set of strings. These help to define the language generated by the grammar.

Start Symbol : One non-terminal in the grammar is denoted as the “Start-symbol” and the set of strings it denotes is the language defined by the grammar.

Productions : It specifies the manner in which 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.

Example of context-free grammar: The following grammar defines simple arithmetic expressions:

$$expr \rightarrow expr\ op\ expr$$
$$expr \rightarrow (expr)$$
$$expr \rightarrow -\ expr$$
$$expr \rightarrow id\ op$$
$$\rightarrow +\ op \rightarrow -$$
$$op \rightarrow *$$
$$op \rightarrow /$$
$$op \rightarrow \uparrow$$

In this grammar,

- **id** + - * / \uparrow () are terminals.
- *expr* , *op* are non-terminals.
- *expr* is the start symbol.
- Each line is a production.

Derivations:

Two basic requirements for a grammar are :

1. To generate a valid string.
2. To recognize a valid string.

Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

Example : Consider the following grammar for arithmetic expressions :

$$E \rightarrow E+E \mid E * E \mid (E) \mid - E \mid id$$

To generate a valid string - (id+id) from the grammar the steps are

1. $E \rightarrow - E$
2. $E \rightarrow - (E)$
3. $E \rightarrow - (E+E)$
4. $E \rightarrow - (id+E)$
5. $E \rightarrow - (id+id)$

In the above derivation,

- E is the start symbol.
- - (id+id) is the required sentence (only terminals).
- Strings such as E, -E, -(E), . . . are called sentinel forms.

Types of derivations:

The two types of derivation are:

1. Left most derivation
 2. Right most derivation.
- In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.
 - In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

Example:

Given grammar $G : E \rightarrow E+E \mid E * E \mid (E) \mid - E \mid id$

Sentence to be derived : - (id+id)

LEFTMOST DERIVATION RIGHTMOST DERIVATION

$E \rightarrow - E$

$E \rightarrow - E$

$E \rightarrow - (E)$

$E \rightarrow - (E)$

$E \rightarrow - (E+E)$

$E \rightarrow - (E+E)$

$E \rightarrow - (id+E)$

$E \rightarrow - (E+id)$

$E \rightarrow - (id+id)$

$E \rightarrow - (id+id)$

- String that appear in leftmost derivation are called **left sentinel forms**.
- String that appear in rightmost derivation are called **right sentinel forms**.

Sentinels:

Given a grammar G with start symbol S, if $S \rightarrow \alpha$, where α may contain non-terminals or terminals, then α is called the sentinel form of G.

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 sentinel forms that are read from left to right. The sentinel form in the parse tree is called **yield** or **frontier** of the tree.

Ambiguity:

A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**.

Example : Given grammar $G : E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid id$

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

$$E \rightarrow E + E$$

$$E \rightarrow id + E$$

$$E \rightarrow id + E * E$$

$$E \rightarrow id + id * E$$

$$E \rightarrow id + id * id$$

$$E \rightarrow E * E$$

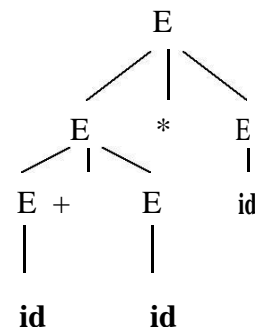
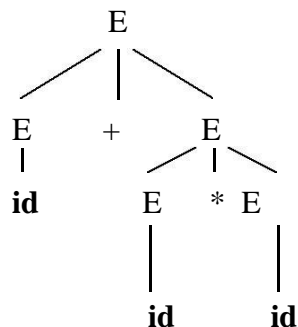
$$E \rightarrow E + E * E$$

$$E \rightarrow id + E * E$$

$$E \rightarrow id + id * E$$

$$E \rightarrow id + id * id$$

The two corresponding parse trees are :



Converting ambiguous grammar to Unambiguous grammar.

1. Eliminating left-recursion
2. Left-factoring.

Each parsing method can handle grammars only of a certain form hence, the initial grammar may have to be rewritten to make it parsable.

- The lexical rules of a language are simple and RE is used to describe them.
- Regular expressions provide a more concise and easier to understand notation for tokens than grammars.
- Efficient lexical analyzers can be constructed automatically from RE than from grammars.
- Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end into two manageable-sized components.

Regular Expressions vs. Context-Free Grammars:

REGULAR EXPRESSION	CONTEXT-FREE GRAMMAR
It is used to describe the tokens of programming languages.	It consists of a quadruple where $S \rightarrow$ start symbol, $P \rightarrow$ production, $T \rightarrow$ terminal, $V \rightarrow$ variable or non-terminal.
It is used to check whether the given input is valid or not using transition diagram .	It is used to check whether the given input is valid or not using derivation .
The transition diagram has set of states and edges.	The context-free grammar has set of productions.
It has no start symbol.	It has start symbol.
It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.	It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on.

Eliminating ambiguity:

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

Consider this example, $G: stmt \rightarrow \text{if } expr \text{ then } stmt \mid \text{if } expr \text{ then } stmt \text{ else } stmt \mid \text{other}$

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 :

W = “if E₁ then if E₂ then S₁ else S₂”

W = ibtibtaea

\Rightarrow ibt(ibtaea)

OR

\Rightarrow ibt(ibta)ea

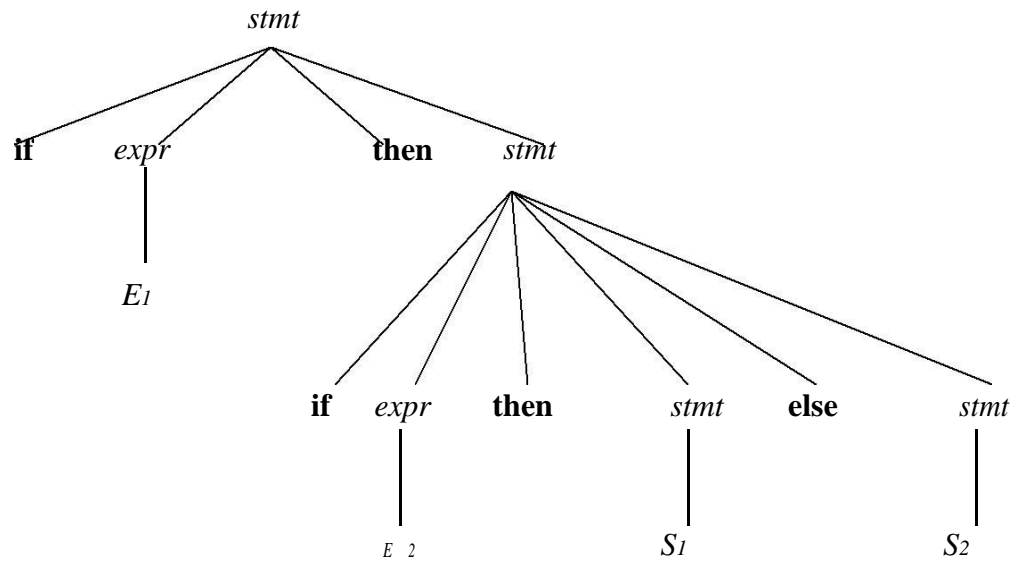
To eliminate ambiguity, the following grammar may be used:

$stmt \rightarrow matched_stmt \mid unmatched_stmt$

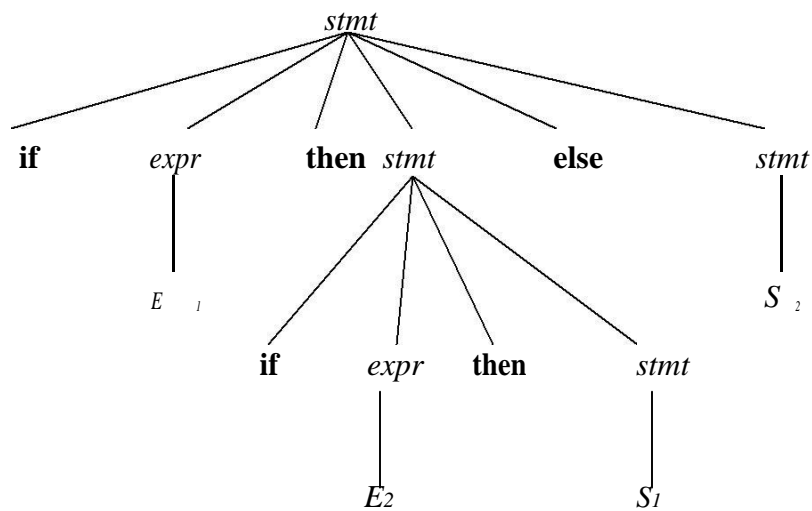
$matched_stmt \rightarrow \text{if } expr \text{ then } matched_stmt \text{ else } matched_stmt \mid \text{other}$

$unmatched_stmt \rightarrow \text{if } expr \text{ then } stmt \mid \text{if } expr \text{ then } matched_stmt \text{ else } unmatched_stmt$

1.



2.



Eliminating Left Recursion:

A grammar is said to be *left recursive* if it has a non-terminal A such that there is a derivation $A \Rightarrow A\alpha$ 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 $A \rightarrow A\alpha \mid \beta$ it can be replaced with a sequence of two productions

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon \text{ without}$$

changing the set of strings derivable from A .

Example : Consider the following grammar for arithmetic expressions:E

$\rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

First eliminate the left recursion for E as E

$\rightarrow TE'$

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

Then eliminate for T as T

$\rightarrow FT'$

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

Thus the obtained grammar after eliminating left recursion is E

$\rightarrow TE'$

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

$T \rightarrow FT'$

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

$\rightarrow (E) \mid id$

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order $A_1, A_2 \dots A_n$.

2. **for** $i:= 1$ **to** n **do begin**

for $j:= 1$ **to** $i-1$ **do begin**

 replace each production of the form $A_i \rightarrow A_j \gamma$ by the

 productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$

 where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all the current A_j -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

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

Consider the grammar, $G : S \rightarrow iEtS \mid iEtSeS \mid a$

$$E \rightarrow b$$

Left factored, this grammar becomes

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

PARSING

It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Parse tree:

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

Types of parsing:

1. Top down parsing
2. Bottom up parsing

➤ Top-down parsing : A parser can start with the start symbol and try to transform it to the input string.

Example : LL Parsers. (Recursive descent parsing , Predictive parsing)

➤ Bottom-up parsing : A parser can start with input and attempt to rewrite it into the start symbol.

Example : LR Parsers. (Shift Reduce Parser, Operator Precedence parser, SLR , CLR, LALR)

TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of top-down parsing :

1. Recursive descent parsing
2. Predictive parsing

1. RECURSIVE DESCENT PARSING

- Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input.
- This parsing method may involve **backtracking**, that is, making repeated scans of the input.

Example for backtracking :

Consider the grammar $G : S \rightarrow cAd$

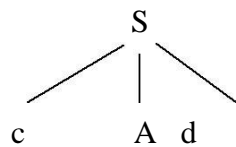
$A \rightarrow ab \mid a$

and the input string $w=cad$.

The parse tree can be constructed using the following top-down approach :

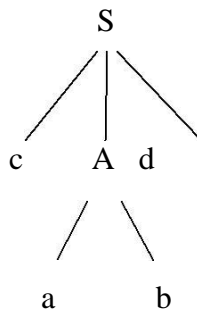
Step1:

Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.



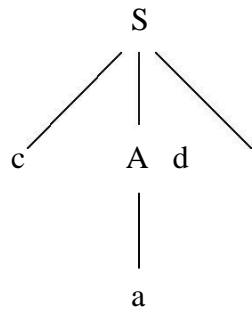
Step3:

The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'. But the third leaf of tree is b which does not match with the input symbol **d**.

Hence discard the chosen production and reset the pointer to second position. This is called **backtracking**.

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

Example for recursive decent parsing:

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop. Hence, **elimination of left-recursion** must be done before parsing.

Consider the grammar for arithmetic expressions

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

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

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

After eliminating the left-recursion the grammar becomes,

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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

Now we can write the procedure for grammar as follows:

Recursive procedure:

Procedure E()

begin

 T();

 EPRIME();

end

```

Procedure EPRIME() begin
    If input_symbol='+' then ADVANCE();
    T(); EPRIME();
end

Procedure T() begin
    F(); TPRIME();
end

Procedure TPRIME() begin
    If input_symbol='*' then ADVANCE(
    );

    F(); TPRIME( );
end

Procedure F() begin
    If (input_symbol='id' ) then
    ADVANCE( );
    else
    {
    if (input_symbol='(' )then ADVANCE( );
    E();
    else if input_symbol=')' then ADVANCE(
    );
    }
end

else ERROR( );

```

Stack implementation:

To recognize input **id+id*id\$** :

PROCEDURE	INPUT STRING
E()	<u>id</u> +id*id\$
T()	<u>id</u> +id*id\$
F()	<u>id</u> +id*id\$
ADVANCE()	id+ <u>id</u> *id\$
TPRIME()	id+ <u>id</u> *id\$
EPRIME()	id+ <u>id</u> *id\$
ADVANCE()	id+ <u>id</u> *id\$
T()	id+ <u>id</u> *id\$
F()	id+ <u>id</u> *id\$
ADVANCE()	id+id* <u>id</u> \$
TPRIME()	id+id* <u>id</u> \$

ADVANCE()	id+id* <u>id</u> \$
F()	id+id* <u>id</u> \$
ADVANCE()	id+id* <u>id</u> \$
TPRIME()	id+id*id <u>\$</u>

2. PREDICTIVE PARSING

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.
- At each step, the choice of rule to be expanded is made upon the made upon the next terminal symbol.

Suppose $A \rightarrow A1 \mid A2 \mid \dots \mid An$

If the non-terminal to be expanded next is 'A', then the choice of rule is made on the basis of the current input symbol 'a' only.

- Recursive predictive parser**
- Non-recursive predictive parser**

1. Recursive predictive parser

Procedure for making an Predictive Parser:

1. Make a transition diagram (like dfa/nfa) for every rule of the grammar.
2. Optimize the dfa by reducing the number of states, yielding the final transition diagram.
3. To parse a string, simulate the string on the transition diagram.
4. If after consuming the input the transition diagram reaches an accept state, it is parsed.

Each of these steps are expanded below:

Consider the following grammar:

- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid id$

After removing left-recursion , left-factoring

$E \rightarrow TE'$

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

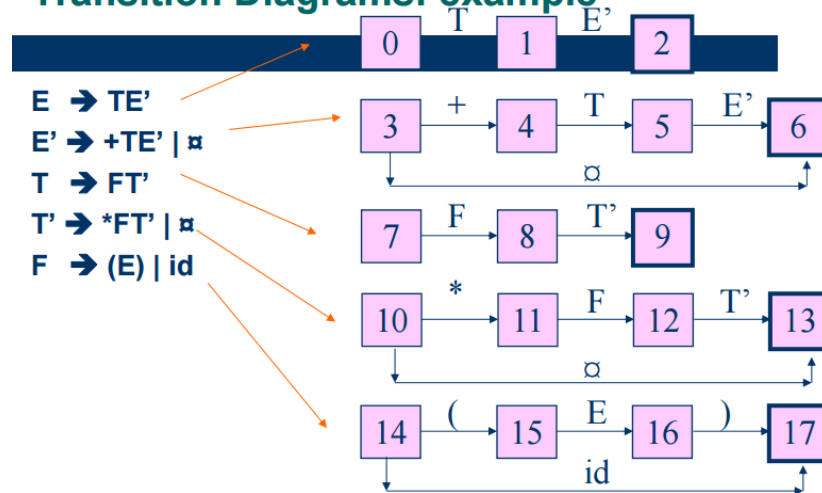
$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Step 1: make dfa-like transition diagrams for each rule:

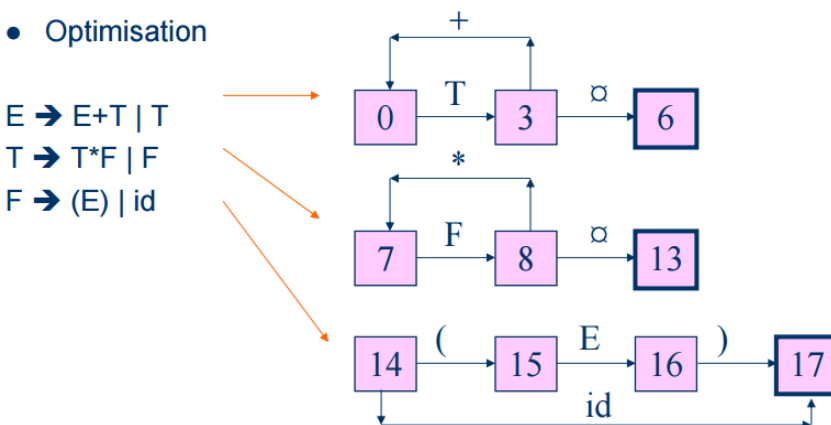
Transition Diagrams: example



Step 2 : Optimization

The purpose of optimization is to reduce the number of states

• Optimisation



After optimization is over we start simulation over these “dfa’s”.

Step 3: Simulation on the input string:

Following steps list out the simulation procedure briefly.

- Start from the start state
- If a terminal comes consume it, move to next state
- If a non – terminal comes go to the state of the “dfa” of the non-term and return on reaching the final state
- Return to the original “dfa” and continue parsing
- If on completion(reading input string completely), you reach a final state, string is successfully parsed.

Disadvantages of Predictive Parser:

- It is inherently a **recursive parser**, so it consumes a lot of memory as the stack grows.
- Doing **optimization** may not be as simple as the complexity of grammar grows
- To remove this recursion, we use **LL-parser**, which uses a table for lookup.

2. Non-recursive predictive parser

The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer: It consists of strings to be parsed, followed by \$ to indicate the end of the input string.

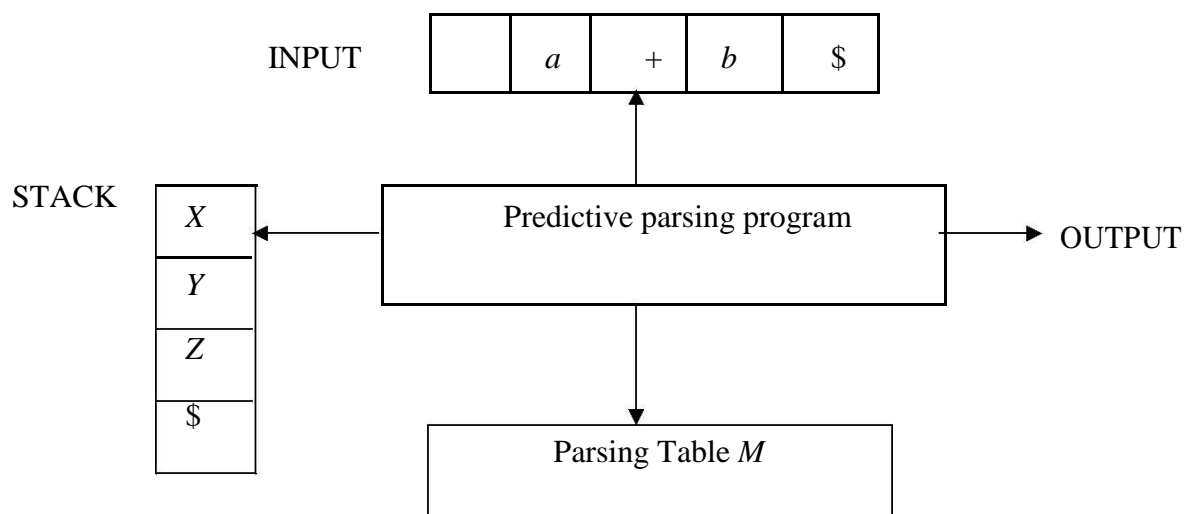
Stack: It contains a sequence of grammar symbols preceded by \$ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of \$.

Parsing table: It is a two-dimensional array $M[A, a]$, where 'A' is anon-terminal and 'a' is a terminal.

Predictive parsing program:

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

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, 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 either be an X -production of the grammar or an error entry.
If $M[X, a] = \{X \rightarrow UVW\}$, the parser replaces X on top of the stack by WVU . If $M[X, a] = \text{error}$, the parser calls an error recovery routine.



Algorithm for nonrecursive 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 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 as follows:

```
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 or  $\$$  then  
        if  $X=a$  then  
            pop  $X$  from the stack and advance  
        else  $error()$   
    else /*  $X$  is a non-terminal */  
        if  $M[X,a] = X \rightarrow Y_1 Y_2 \dots Y_k$  then begin  
            pop  $X$  from the stack;  
            push  $Y_k, Y_{k-1}, \dots, Y_1$  onto the stack, with  $Y_1$  on top;  
            output the production  $X \rightarrow Y_1 Y_2 \dots Y_k$   
        end  
    else  $error()$   
until  $X = \$$  /* stack is empty */
```


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

Rules for follow():

1. If S is a start symbol, then $\text{FOLLOW}(S)$ contains $\$$.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in $\text{FIRST}(\beta)$ except ϵ is placed in $\text{follow}(B)$.
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where $\text{FIRST}(\beta)$ contains ϵ , then everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$.

Algorithm for construction of predictive parsing table:

Input : Grammar G

Output : Parsing table M

Method :

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

Example:

Consider the following grammar :

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

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

$$F \rightarrow (E) \mid id$$

After eliminating left-recursion the grammar is

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

$$F \rightarrow (E) \mid id$$

First() :

$$\text{FIRST}(E) = \{ (, id \}$$

$$\text{FIRST}(E') = \{ +, \epsilon \}$$

$$\text{FIRST}(T) = \{ (, id \}$$

$$\text{FIRST}(T') = \{ *, \epsilon \}$$

$$\text{FIRST}(F) = \{ (, id \}$$

Follow() :

$$\text{FOLLOW}(E) = \{ \$,) \}$$

$$\text{FOLLOW}(E') = \{ \$,) \}$$

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

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

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

Predictive parsing table :

NON- TERMINAL	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	$F \rightarrow id$			$F \rightarrow (E)$		

Stack implementation:

stack	Input	Output
\$E	id+id*id \$	
\$E'T	id+id*id \$	$E \rightarrow TE'$
\$E'T'F	id+id*id \$	$T \rightarrow FT'$
\$E'T'id	id+id*id \$	$F \rightarrow id$
\$E'T'	+id*id \$	
\$E'	+id*id \$	$T' \rightarrow \epsilon$
\$E'T+	+id*id \$	$E' \rightarrow +TE'$
\$E'T	id*id \$	
\$E'T'F	id*id \$	$T \rightarrow FT'$
\$E'T'id	id*id \$	$F \rightarrow id$
\$E'T'	*id \$	
\$E'T'F*	*id \$	$T' \rightarrow *FT'$
\$E'T'F	id \$	
\$E'T'id	id \$	$F \rightarrow id$
\$E'T'	\$	
\$E'	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$

LL(1) grammar:

The parsing table entries are single entries. So each location has not more than one entry. This type of grammar is called LL(1) grammar.

Consider this following grammar:

$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$

After eliminating left factoring, we have

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

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

$$\text{FIRST}(S) = \{ i, a \}$$

$$\text{FIRST}(S') = \{ e, \epsilon \}$$

$$\text{FIRST}(E) = \{ b \}$$

$$\text{FOLLOW}(S) = \{ \$, e \}$$

$\text{FOLLOW}(S') = \{ \$, e \}$

$\text{FOLLOW}(E) = \{ t \}$

Parsing table:

NON- TERMINAL	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Since there are more than one production, the grammar is not LL(1) grammar.

Actions performed in predictive parsing:

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

Implementation of predictive parser:

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

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:

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

The sentence to be recognized is **abbcde**.

REDUCTION (LEFTMOST)

abbcde ($A \rightarrow b$)
a**A**bcde ($A \rightarrow Abc$)
aA**d**e ($B \rightarrow d$)
aA**B**e ($S \rightarrow aABe$)
S

RIGHTMOST DERIVATION

$S \rightarrow aABe$
 $\rightarrow aAde$
 $\rightarrow aAbcde$
 $\rightarrow abbcde$

The reductions trace out the right-most derivation in reverse.

Handles:

A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Example:

Consider the grammar:

$E \rightarrow E+E$
 $E \rightarrow E * E$
 $E \rightarrow (E)$
 $E \rightarrow id$

And the input string $id_1 + id_2 * id_3$

The rightmost derivation is :

$E \rightarrow \underline{E+E}$
 $\rightarrow E + \underline{E * E}$
 $\rightarrow E + E * \underline{id_3}$
 $\rightarrow E + \underline{id_2} * id_3$
 $\rightarrow \underline{id_1} + id_2 * id_3$

In the above derivation the underlined substrings are called **handles**.

Handle pruning:

A rightmost derivation in reverse can be obtained by “**handle pruning**”.

(i.e.) if w is a sentence or string of the grammar at hand, then $w = \gamma_n$, where γ_n is the n^{th} right-sentinel form of some rightmost derivation.

Stack implementation of shift-reduce parsing :

Stack	Input	Action
\$	id ₁ +id ₂ *id ₃ \$	shift
\$ id ₁	+id ₂ *id ₃ \$	reduce by E→id
\$ E	+id ₂ *id ₃ \$	shift
\$ E+	id ₂ *id ₃ \$	shift
\$ E+id ₂	*id ₃ \$	reduce by E→id
\$ E+E	*id ₃ \$	shift
\$ E+E*	id ₃ \$	shift
\$ E+E*id ₃	\$	reduce by E→id
\$ E+E*E	\$	reduce by E→ E *E
\$ E+E	\$	reduce by E→ E+E
\$ E	\$	accept

Actions in shift -reduce parser:

- shift – The next input symbol is shifted onto the top of the stack.
- reduce – The parser replaces the handle within a stack with a non-terminal.
- accept – The parser announces successful completion of parsing.
- error – The parser discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift shift-reduce parsing:

1. **Shift-reduce conflict:** The parser cannot decide whether to shift or to reduce.
2. **Reduce-reduce conflict:** The parser cannot decide which of several reductions to make.

1. Shift-reduce conflict:

Example:

Consider the grammar:

$E \rightarrow E+E \mid E * E \mid id$ and input $id+id*id$

Stack	Input	Action	Stack	Input	Action
\$ E+E	*id \$	Reduce by $E \rightarrow E+E$	\$E+E	*id \$	Shift
\$ E	*id \$	Shift	\$E+E*	id \$	Shift
\$ E*	id \$	Shift	\$E+E*id	\$	Reduce by $E \rightarrow id$
\$ E*id	\$	Reduce by $E \rightarrow id$	\$E+E*E	\$	Reduce by $E \rightarrow E*E$
\$ E*E	\$	Reduce by $E \rightarrow E*E$	\$E+E	\$	Reduce by $E \rightarrow E*E$
\$ E		Accept	\$E		Accept

2. Reduce-reduce conflict:

Consider the grammar:

$M \rightarrow R+R \mid R+c \mid R$

$R \rightarrow c$

and input $c+c$

Stack	Input	Action	Stack	Input	Action
\$	c+c \$	Shift	\$	c+c \$	Shift
\$ c	+c \$	Reduce by $R \rightarrow c$	\$ c	+c \$	Reduce by $R \rightarrow c$
\$ R	+c \$	Shift	\$ R	+c \$	Shift
\$ R+	c \$	Shift	\$ R+	c \$	Shift
\$ R+c	\$	Reduce by $R \rightarrow c$	\$ R+c	\$	Reduce by $M \rightarrow R+c$
\$ R+R	\$	Reduce by $M \rightarrow R+R$	\$ M	\$	Accept
\$ M	\$	Accept			

Viable prefixes:

- α is a viable prefix of the grammar if there is w such that αw is a right sentinel form.
- The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- The set of viable prefixes is a regular language.

OPERATOR-PRECEDENCE PARSING

An efficient way of constructing shift-reduce parser is called operator-precedence parsing.

Operator precedence parser can be constructed from a grammar called Operator-grammar. These grammars have the property that no production on right side is ϵ or has two adjacent non-terminals.

Example:

Consider the grammar:

$E \rightarrow EAE \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

Since the right side EAE has three consecutive non-terminals, the grammar can be written as follows:

$E \rightarrow E+E \mid E-E \mid E * E \mid E/E \mid E \uparrow E \mid -E \mid id$

Operator precedence relations:

There are three disjoint precedence relations

namely $<'$ - less than

$=$ - equal to

$'>$ - greater than

The relations give the following meaning:

$a <' b$ - a yields precedence to b

$a = b$ - a has the same precedence as b

$a '> b$ - a takes precedence over b

Rules for binary operations:

1. If operator θ_1 has higher precedence than operator θ_2 , then

make $\theta_1 '> \theta_2$ and $\theta_2 <' \theta_1$

2. If operators θ_1 and θ_2 , are of equal precedence, then make

$\theta_1 '> \theta_2$ and $\theta_2 '> \theta_1$ if operators are left associative

$\theta_1 <' \theta_2$ and $\theta_2 <' \theta_1$ if right associative

3. Make the following for all operators θ :

$\theta <' id, id '> \theta$

$\theta <' (, (<' \theta$

$) '> \theta, \theta '>)$

$'> \$, \$ <' \theta$

Also make

$(=), (<^{\cdot} (,)^{\cdot} >), (<^{\cdot} \text{id}, \text{id}^{\cdot} >), \$ <^{\cdot} \text{id}, \text{id}^{\cdot} > \$, \$ <^{\cdot} (,)^{\cdot} > \$$

Example:

Operator-precedence relations for the grammar

$E \rightarrow E+E \mid E-E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid -E \mid \text{id}$ is given in the following table assuming

1. \uparrow is of highest precedence and right-associative
2. $*$ and $/$ are of next higher precedence and left-associative, and
3. $+$ and $-$ are of lowest precedence and left-associative

Note that the **blanks** in the table denote error entries.

TABLE : Operator-precedence relations

	+	-	*	/	\uparrow	id	()	\$
+	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
-	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
*	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
/	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
\uparrow	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot >$	$\cdot >$
id	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	---	---	$\cdot >$	$\cdot >$
($\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	=	---
)	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	$\cdot >$	---	---	$\cdot >$	$\cdot >$
\$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	$\cdot <$	---	Accept

Operator precedence parsing algorithm:

Input : An input string w and a table of precedence relations.

Output : If w is well formed, a skeletal parse tree, with a placeholder non-terminal E labeling all interior nodes; otherwise, an error indication.

Method : Initially the stack contains $\$$ and the input buffer the string $w\$$. To parse, we execute the following program :

- (1) Set ip to point to the first symbol of $w\$$;
- (2) **repeat forever**
- (3) **if** $\$$ is on top of the stack and ip points to $\$$ **then**
- (4) **return**
- else begin**
- (5) let a be the topmost terminal symbol on the stack
 and let b be the symbol pointed to by ip ;
- (6) **if** $a <^{\cdot} b$ or $a = b$ **then begin**
- (7) push b onto the stack;
- (8) advance ip to the next input symbol;
- end;**

```

(9)    else if  $a' > b$  then    /*reduce*/
(10)   repeat
(11)       pop the stack
(12)   until the top stack terminal is related by
        <' to the terminal most recently popped
(13)   else error( )
        end

```

Stack implementation of operator precedence parsing:

Operator precedence parsing uses a stack and precedence relation table for its implementation of above algorithm. It is a shift-reduce parsing containing all four actions shift, reduce, accept and error.

The initial configuration of an operator precedence parsing

is STACK
\$

INPUT
w\$

where w is the input string to be parsed.

Example:

Consider the grammar $E \rightarrow E+E \mid E-E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid id$. Input string is **id+id*id**. The implementation is as follows:

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

Advantages of operator precedence parsing:

1. It is easy to implement.
2. Once an operator precedence relation is made between all pairs of terminals of a grammar , the grammar can be ignored. The grammar is not referred anymore during implementation.

Disadvantages of operator precedence parsing:

1. It is hard to handle tokens like the minus sign (-) which has two different precedence.
2. Only a small class of grammar can be parsed using operator-precedence parser.

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. When ' k ' is omitted, it is assumed to be 1.

Advantages of LR parsing:

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

Drawbacks of LR method:

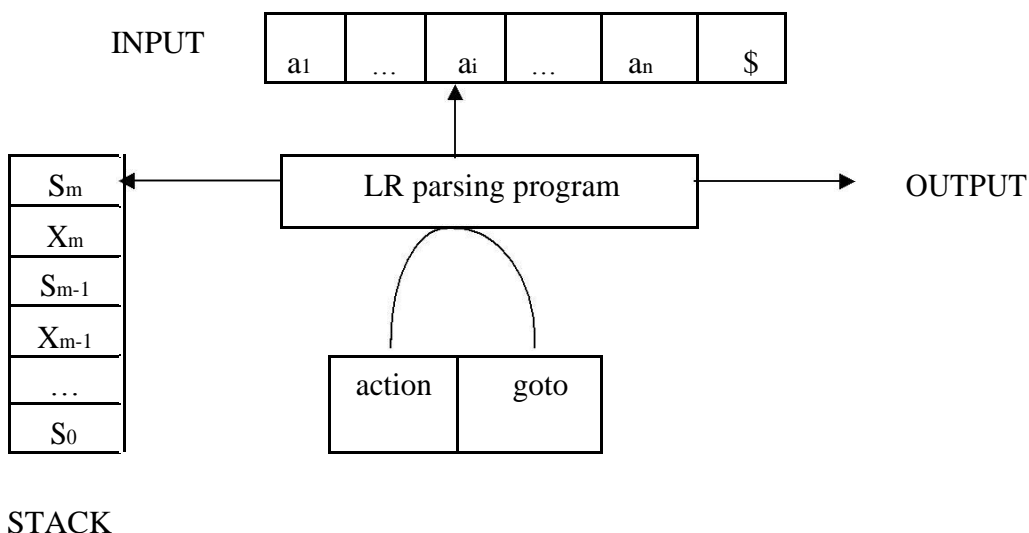
It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

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:



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 program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\dots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a state.
- The parsing table consists of two parts : *action* and *goto* functions.

Action : The parsing program determines s_m , the state currently on top of stack, and a_i , 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 $A \rightarrow \beta$,
3. accept, and
4. error.

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

LR 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 s_0 on its stack, where s_0 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  $action[s, a] = \text{'shifts'}$  then begin push  $a$ 
        then  $s'$  on top of the stack; advance
        ip to the next input symbol
    end
    else if  $action[s, a] = \text{'reduce } A \rightarrow \beta$  then begin
        pop  $2 * |\beta|$  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  $A \rightarrow \beta$ 
    end
    else if  $action[s, a] = \text{'accept'}$  then
        return
    else  $error()$ 
end
```

CONSTRUCTING SLR(1) 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 $A \rightarrow XYZ$ yields the four items :

$A \rightarrow \cdot XYZ$

$A \rightarrow X \cdot YZ$

$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$

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 $A \rightarrow \alpha \cdot B\beta$ is in $closure(I)$ and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot \gamma$ 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 $[A \rightarrow \alpha X \cdot \beta]$ such that $[A \rightarrow \alpha \cdot X\beta]$ 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 $C = \{I_0, I_1, \dots, I_n\}$, 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 $[A \rightarrow \alpha \cdot a\beta]$ 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 $[A \rightarrow \alpha \cdot]$ is in I_i , then set $action[i, a]$ to "reduce $A \rightarrow \alpha$ " for all a in FOLLOW(A).
 - (c) If $[S' \rightarrow S \cdot]$ 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 $[S' \rightarrow .S]$.

Example for SLR parsing:

Construct SLR parsing for the following grammar :

$G : E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

The given grammar is :

$G : E \rightarrow E + T \quad \text{----- (1)}$

$E \rightarrow T \quad \text{----- (2)}$

$T \rightarrow T * F \quad \text{----- (3)}$

$T \rightarrow F \quad \text{----- (4)}$

$F \rightarrow (E) \quad \text{----- (5)}$

$F \rightarrow id \quad \text{----- (6)}$

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar :

$E' \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Step 2 : Find LR (0) items.

$I_0 : E' \rightarrow . E$

$E \rightarrow . E + T$

$E \rightarrow . T$

$T \rightarrow . T * F$

$T \rightarrow . F$

$F \rightarrow . (E)$

$F \rightarrow . id$

$GOTO(I_0, E)$

$I_1 : E' \rightarrow E .$

$E \rightarrow E . + T$

$GOTO(I_4, id)$

$I_5 : F \rightarrow id .$

GOTO (I₀ , T)

I₂ : E → T .

T → T . * F

GOTO (I₀ , F)

I₃ : T → F .

GOTO (I₀ , ()

I₄ : F → (. E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → . id

GOTO (I₀ , id)

I₅ : F → id .

GOTO (I₁ , +)

I₆ : E → E + . T

T → . T * F

T → . F

F → . (E)

F → . id

GOTO (I₂ , *)

I₇ : T → T * . F

F → . (E)

F → . id

GOTO (I₄ , E)

I₈ : F → (E .) E

→ E . + T

GOTO (I₄ , T)

I₂ : E → T .

T → T . * F

GOTO (I₄ , F)

I₃ : T → F .

GOTO (I₆ , T)

I₉ : E → E + T .

T → T . * F

GOTO (I₆ , F)

I₃ : T → F .

GOTO (I₆ , ()

I₄ : F → (. E)

GOTO (I₆ , id)

I₅ : F → id .

GOTO (I₇ , F)

I₁₀ : T → T * F .

GOTO (I₇ , ()

I₄ : F → (. E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → . id

GOTO (I₇ , id)

I₅ : F → id .

GOTO (I₈ ,))

I₁₁ : F → (E) .

GOTO (I₈ , +)

I₆ : E → E + . T

T → . T * F

T → . F

F → . (E)

F → . id

GOTO (I₉ , *)

I₇ : T → T * . F

F → . (E)

F → . id

GOTO (I₄,)

I₄ : F → (. E)

E → . E + T

E → . T

T → . T * F

T → . F

F → . (E)

F → id

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

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

FOOLOW (F) = { * , + ,) , \$ }

SLR parsing table:

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

Blank entries are error entries.

Stack implementation:

Check whether the input **id + id * id** is valid or not.

STACK	INPUT	ACTION
0	id + id * id \$	GOTO (I ₀ , id) = s5 ; shift
0 id 5	+ id * id \$	GOTO (I ₅ , +) = r6 ; reduce by F→id
0 F 3	+ id * id \$	GOTO (I ₀ , F) = 3 GOTO (I ₃ , +) = r4 ; reduce by T → F
0 T 2	+ id * id \$	GOTO (I ₀ , T) = 2 GOTO (I ₂ , +) = r2 ; reduce by E → T
0 E 1	+ id * id \$	GOTO (I ₀ , E) = 1 GOTO (I ₁ , +) = s6 ; shift
0 E 1 + 6	id * id \$	GOTO (I ₆ , id) = s5 ; shift
0 E 1 + 6 id 5	* id \$	GOTO (I ₅ , *) = r6 ; reduce by F→ id
0 E 1 + 6 F 3	* id \$	GOTO (I ₆ , F) = 3 GOTO (I ₃ , *) = r4 ; reduce by T → F
0 E 1 + 6 T 9	* id \$	GOTO (I ₆ , T) = 9 GOTO (I ₉ , *) = s7 ; shift
0 E 1 + 6 T 9 * 7	id \$	GOTO (I ₇ , id) = s5 ; shift
0 E 1 + 6 T 9 * 7 id 5	\$	GOTO (I ₅ , \$) = r6 ; reduce by F→ id
0 E 1 + 6 T 9 * 7 F 10	\$	GOTO (I ₇ , F) = 10 GOTO (I ₁₀ , \$) = r3 ; reduce by T → T * F
0 E 1 + 6 T 9	\$	GOTO (I ₆ , T) = 9 GOTO (I ₉ , \$) = r1 ; reduce by E → E + T
0 E 1	\$	GOTO (I ₀ , E) = 1 GOTO (I ₁ , \$) = accept

CANONICAL LR PARSING

Algorithm For Construction Of The Canonical Lr Parsing Table

Input: grammar G'

Output: canonical LR parsing table functions action and goto

Method :

1. Construct $C = \{I_0, I_1, \dots, I_n\}$ the collection of sets of LR(1) items for G'. State i is constructed from I_i.
2. if $[A \rightarrow a.ab, b >]$ is in I_i and $\text{goto}(I_i, a) = I_j$, then set $\text{action}[i, a]$ to "shift j". Here a must be a terminal.

3. if $[A \rightarrow a., a]$ is in I_i , then set $\text{action}[i, a]$ to "reduce $A \rightarrow a$ " for all a in $\text{FOLLOW}(A)$. Here A may *not* be S' .
4. if $[S' \rightarrow .S]$ is in I_i , then set $\text{action}[i, \$]$ to "accept"
5. If any conflicting actions are generated by these rules, the grammar is not LR(1) and the algorithm fails to produce a parser.
6. The goto transitions for state i are constructed for all *nonterminals* A using the rule: If $\text{goto}(I_i, A) = I_j$, then $\text{goto}[i, A] = j$.
7. All entries not defined by rules 2 and 3 are made "error".
8. The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow .S, \$]$.

For example, let us Consider the following grammer,

$S' \rightarrow S$

$S \rightarrow CC \quad \text{---1}$

$C \rightarrow cC \quad \text{----2}$

$C \rightarrow d \quad \text{-----3}$

Sets of LR(1) items

I0: $S' \rightarrow .S, \$$

$S \rightarrow .CC, \$$

$C \rightarrow .cC, c/d$

$C \rightarrow .d, c/d$

Go to (I0,S)

I1: $S' \rightarrow S., \$$

Go to (I0,C)

I2: $S \rightarrow C.C, \$$

$C \rightarrow .cC, \$$

$C \rightarrow .d, \$$

Go to (I0,c)

I3: $C \rightarrow c.C, c/d$

$C \rightarrow .cC, c/d$

$C \rightarrow .d, c/d$

Go to (I0,d)

I4: $C \rightarrow d., c/d$

Go to (I2,C)

I5: $S \rightarrow CC., \$$

Go to (I2,c)

I6: $C \rightarrow c.C, \$$

$C \rightarrow .cC, \$$

$C \rightarrow .d, \$$

Go to (I2,d)

I7: C->d.,\$

Go to (I3,C)

I8: C->cC.,c/d

Go to (I3,c)

I3 : C->c.C, c/d

C->.cC, c/d

C->.d, c/d

Go to (I3,d)

I4: C->d., c/d

Go to (I6,C)

I9: C->cC.,\$

Go to (I6,c)

I6: C->c.C,\$

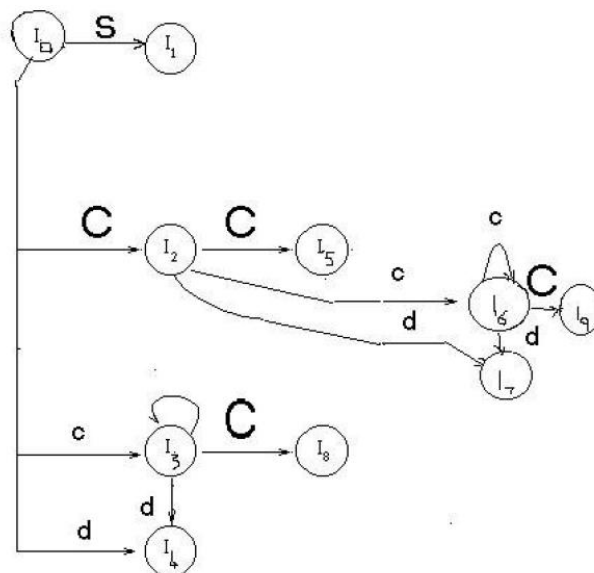
C->.cC, \$

C->.d, \$

Go to (I6,d)

I7: C->d.,\$

The corresponding DFA



Reduce - extension

Parsing Table

state	c	d	\$	S	C
0	S3	S4		1	2
1			ACCEPT		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

LALR PARSER

Algorithm for construction of LALR parsing table:

Input : An augmented grammar G'

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

Method :

1. Construct $C = \{I_0, I_1, \dots, I_n\}$ the collection of sets of LR(1) items for G' .
2. For each core present among the set of LR(1) items, find all sets having that core and replace these sets by the union.
3. Let $C' = \{J_0, J_1, \dots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in the construction of the canonical LR parsing table.
4. If there is a conflict, the grammar is not LALR(1) and the algorithm fails.
5. The goto table is constructed as follows: If J is the union of one or more sets of LR(1) items, that is, $J = I_0 \cup I_1 \cup \dots \cup I_k$, then the cores of $\text{goto}(I_0, X)$, $\text{goto}(I_1, X)$, ..., $\text{goto}(I_k, X)$ are the same, since I_0, I_1, \dots, I_k all have the same core. Let K be the union of all sets of items having the same core as $\text{goto}(I_1, X)$.
6. Then $\text{goto}(J, X) = K$.

Consider the above example,

I3 & I6 can be replaced by their union

I36: $C \rightarrow c.C, c/d/\$$

$C \rightarrow .cC, c/d/\$$

$C \rightarrow .d, c/d/\$$

I47: $C \rightarrow d., c/d/\$$

Parsing Table

state	c	d	\$	S	C
0	S36	S47		1	2
1			ACCEPT		
2	S36	S47			5
36	S36	S47			89
47	R3	R3			
5			R1		
89	R2	R2	R2		

LR ERROR RECOVERY

An LR parser will detect an error when it consults the parsing action table and find a blank or error entry. Errors are never detected by consulting the goto table. An LR parser will detect an error as soon as there is no valid continuation for the portion of the input thus far scanned. A canonical LR parser will not make even a single reduction before announcing the error. SLR and LALR parsers may make several reductions before detecting an error, but they will never shift an erroneous input symbol onto the stack.

GENERATE YACC SPECIFICATION TO VALIDATE A VARIABLE

AIM:

To recognize a valid variable which starts with a letter followed by any number of letters or digits using a Yacc program.

ALGORITHM:

- Step 1: Start the program.
- Step 2: Include the necessary header files.
- Step 3: Use corresponding parse functions to parse the given input.
- Step 4: Use the if alpha and if digit functions to check the variable syntax.
- Step 5: If the variable starts with a letter and has digits then print Accepted.
- Step 6: Else if the variable starts with a number print rejected.
- Step 7: Terminate the program.

PROGRAM:

```
%{
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>

%}

%token letdig

%%
```

```

TERM:VAR'n'{printf("\nAccepted\n");exit(0);}
|error{yyerror("Rejected\n");exit(0);}
;
VAR:VARlet
|VARdig
|let
;
%%
main()
{
printf("Enteravariabel:");
yyparse();
}
yylex()
{
charch;
while((ch=getchar())!="");
if(isalpha(ch))
returnlet;
if(isdigit(ch))
returndig;
returnch;
}
yyerror(char*s)
{
printf("%s",s);
}

```

To compile : yacc-d4bnew.y and ccylab.c-ll

Enteravariabel: a1

Accepted

Enteravariabel:6a

syntaxerror

Rejected

GENERATE YACC SPECIFICATION TO VALIDATE ARITHMETIC EXPRESSIONS

AIM:

To create LEX and YACC programs to recognize the arithmetic expressions.

ALGORITHM:

Step 1: Start the program.

Step 2: Create a lex program to read and include the output of the yacc program.

Step 3: Define the rule for the operands for operations to be only numbers.

Step 4: Define the Yacc rules for the expressions and conditions to be validated.

Step 5: Terminate the programs.

Program:

4a.y

```
% {
#include<stdio.h>
% }
%token NUM
%left '+' '-'
%left '*' '/'
%left '(' ')'
%%
expr:e{
printf("result:%d\n",$$);
return 0;
}
e:e+'e'{$$=$1+$3;}
|e-'e'{$$=$1-$3;}
|e'*e'{$$=$1*$3;}
|e'/e'{$$=$1/$3;}
|'('e')'{$$=$2;}
|NUM{$$=$1;}
;
%%
main()
{
printf("\nEnter the arithmetic expression:\n");
yyparse();
printf("\nInvalid expression\n");
}
yyerror()
{
printf("\nInvalid expression\n");
exit(0);
}
```

4a.l

```
% {
#include<stdio.h>
#include"y.tab.h"
extern int yylval;
% }
%%
[0-9]+{
yylval=atoi(yytext);
return NUM;
}
[\t];
\nreturn 0;
.\return yytext[0];
%%
```

OUTPUT:

To compile : yacc-d4a.y, lex4a.l, cc-clex.yy.cy.tab.c, cc-oa.outlex.yy.oy.tab.o-lfl

```

enterthearithmeticepression:
10+10
result:20
validexpression
enterthearithmeticepression:
10-10
result:0
validexpression

```

CALCULATOR USING LEX AND YACC

AIM:

To create a calculator using Lex and Yacc programs.

ALGORITHM:

- Step 1: Start the program.
- Step 2: Include the necessary header files.
- Step 3: Use a function for printing the error message.
- Step 4: Get the input from the user and parse it.
- Step 5: Check the input is a valid expression or not.
- Step 6: Write separate operations for addition, subtraction, multiplication and division using the expr and matching it with the operators in the in the input.
- Step 7: Print the error messages for the invalid operators.
- Step 8: Print the output of the expression.
- Step 9: Terminate the program.

PROGRAM:

4d.y

```

% {
#include <stdlib.h>
#include <stdio.h>
int yylex(void);
#include "y.tab.h"
% }
%token INTEGER
%%

```

Program :

```

line program
| line
line:
expr '\n' { printf("%d\n", $1); }
| '\n'
expr:
expr '+' mulex { $$ = $1 + $3; }
| expr '-' mulex { $$ = $1 - $3; }
| mulex { $$ = $1; }
mulex:
mulex '*' term { $$ = $1 * $3; }
| mulex '/' term { $$ = $1 / $3; }
| term { $$ = $1; }
term:
 '(' expr ')' { $$ = $2; }
| INTEGER { $$ = $1; }
%%
void yyerror(char *s)
{
fprintf(stderr, "%s\n", s);
}

```



```

return;
}
yywrap()
{
return(1);
}
int main(void)
{
/*yydebug=1;*/
yyparse();
return 0;
}

```

4d.l

```

% {
#include <stdlib.h>
#include <stdio.h>
#include "y.tab.h"
void yyerror(char*);
extern int yylval;
% }
%%
[ \t]+ ;
[0-9]+ {yylval = atoi(yytext);
return INTEGER;}
[-+*/] {return *yytext;}
"(" {return *yytext;}
")" {return *yytext;}
\n {return *yytext;}
. {char msg[25];
sprintf(msg,"%s <%s>","invalid character",yytext);
yyerror(msg);
}

```

OUTPUT:

To compile : `yacc -d 4d.y, lex 4d.l, gcc y.tab.c lex.yy.c`

```

10+20-5
25
10*10
100
10/10
1
10+10
20
10-10
0

```