Compiler Designs

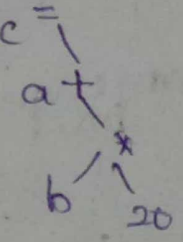
Compiler — Converts source code to byte code
(High level) (Low level)
— Reports error.

High level language consists of preprocessing statements. but it does not compile by compiler.

High level ⟶ Assembly level ⟶ Relocatable language
⇓
No fixed location in memory.

* Linker links all subroutines, loaders loads at fixed location.
* Compiler has 6 phases for converting purely high level - target.

1) Lexical Analysis ⟶ scans, splits as tokens } related to language.
2) Syntax ⟶ checks the syntax
3) semantics
4) Intermediate Code } Dependent on Machine, independent of language.
5) Code optimization.
6) Code Generation



Intermediate Code:- (Almost 3 intermediateries)
Address code

$T_1 = id_3 * 20$
$T_2 = id_2 + T_1$
$id_1 = T_2$

Code optimization:-

$T_1 = id_3 * 20$
$id_1 = id_2 + T_1$

Code Generation:-

Instruction	source	destination
movf	id_3	R_1
mulf	20	R_1
movf	id_2	R_2
addf	R_2	R_1

movf R_1 id_1

These 6 phases are connecting to error handler.

Explain about phases

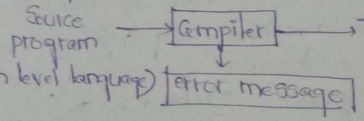
lexical Analysis is scanner / Linear Analysis

syntax " " Hierarchial Analysis

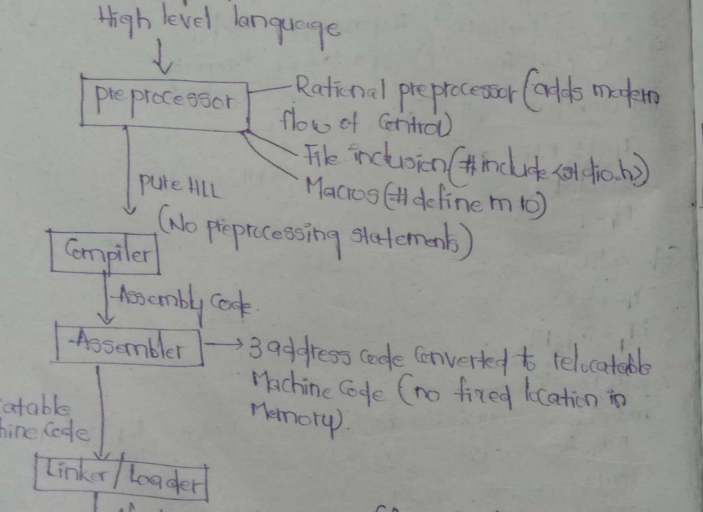
position = initial + final * 20

Introduction

Compiler:— Converts high level language into low level language & also reports the error.



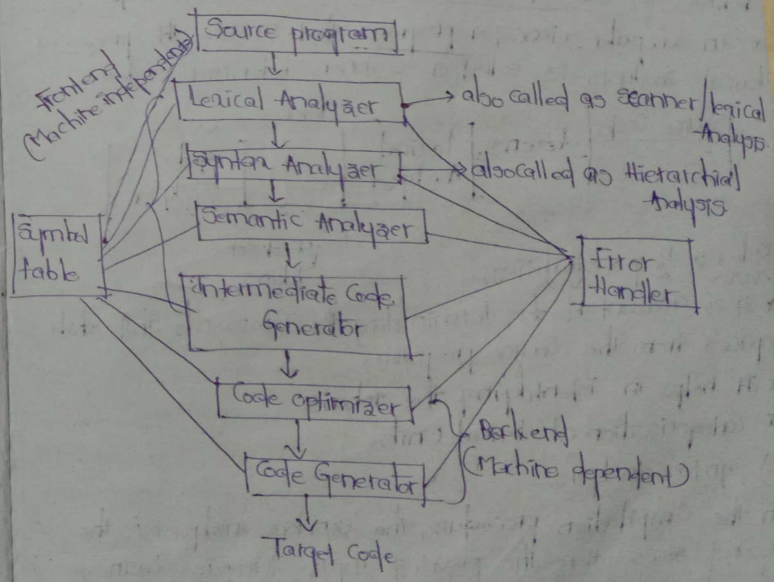
Language processing system:—

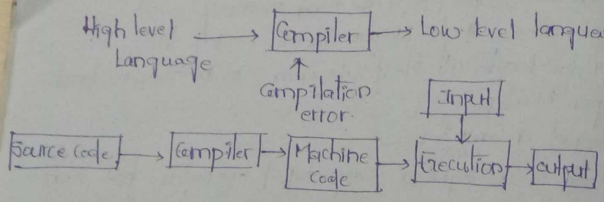


* all subroutines are placed at fixed memory.
→ Linker → allows single program from several files.
→ Loader :— locates all files / subroutines at particular location or proper location.

* phases of a Compiler:—
* A phase is a logically interrelated operation that takes program in one representation & produce output in another representation (Target program) → HLL
* there are 2 parts of Compiler 1) Analysis (Machine Independent, language dependent)
2) Synthesis (Machine dependant, Language independent).

phases (6)



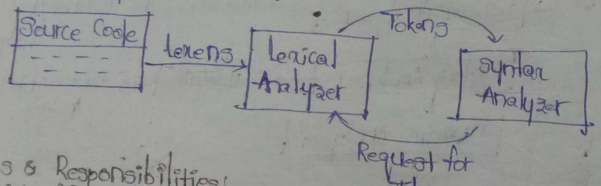


* Lexical Analysis:-

Lexical Analysis (or) Lexical Analyzer is the initial stage or phase of the compiler. This phase scans the source code and transforms the input program into a series of a token.

* A token is basically the arrangements of character that defines a unit of information in the source code.

* In computer science, a program that executes the process of lexical analysis is called a scanner, tokenizer, or lexer.



Roles & Responsibilities:-

* It is accountable for terminating the comments and white spaces from the source program.
* It helps in identifying the tokens.
* Categorization of lexical units.

2) Syntax Analysis:-

In the compilation procedure, the syntax analysis is the second stage. Here the provided input string is scanned for the validation of the structure of the standard grammar. Basically, in the second phase, it analyses the syntactical structure and inspects if the given input is correct or not in terms of programming syntax.

* It is also known as parsing in a compiler.

* Roles and Responsibilities:-

- Note syntax errors.
- Helps in building a parse tree
- Acquire tokens from the lexical Analyzer.
- Scan the syntax errors, if any.

3) Semantic Analysis:-

In the process of compilation, semantic analysis is the third phase. It scans whether the parse tree follows the guidelines of language. It also helps in keeping track of identifiers and expressions. In simple words, we can say that a semantic analyzer defines the validity of the parse tree, and the annotated syntax tree comes to an output.

Roles and Responsibilities:-

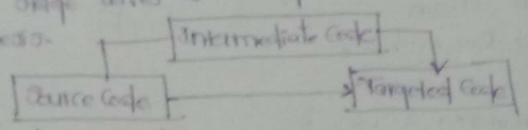
* Saving collected data to symbol tables or syntax trees.
* It notifies semantic errors.
* Scanning for semantic errors.

4) Intermediate Code Generation:-

The parse tree is semantically confirmed; now, an intermediate code generator develops three address codes. A middle level language code generated by a compiler at the time of the translation of a source program into the object code is known as intermediate code (or) text.

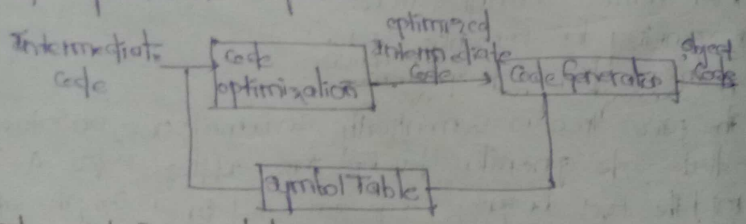
* A code that is neither high-level nor machine code, but a middle-level code is an intermediate code.

Lexical Analysis
* It can translate this code to machine code later.
* This stage serves as a bridge or way from analysis to synthesis.



Roles and Responsibilities—
* Helps in maintaining the priority ordering of the source language
* Translate the intermediate code into the machine code.
* Having operands of instructions

5) Code optimizer:—
Now coming to a phase that is totally optional, and it is code optimization. It is used to enhance the intermediate code. this way, the output of the program is able to run fast and consume less space. To improve the speed of the program, it eliminates the unnecessary strings of the code, and organizes the sequence of statements



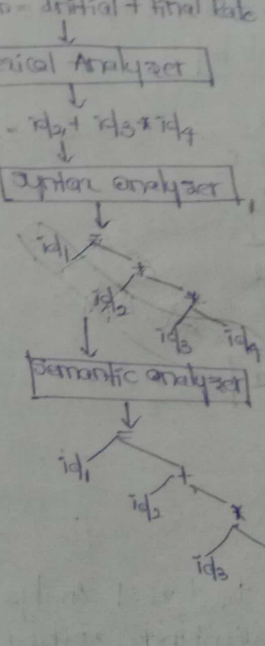
Roles and Responsibilities—
* Remove the unused variables and unreached code.
* Enhance runtime and execution of the program.
* produce streamlined code from the intermediate expression.

6) Code Generator:—
The final stage of the compilation process is the code generation process. In this phase, it tries to acquire the intermediate code as input which is fully optimized and map it to the machine code or language. Later, the code generator helps in translating the intermediate code into the machine code.

Roles and Responsibilities—
* Translate the intermediate code to target machine code.
* select the allocate memory, slots and registers

Ex:

position = Initial + Final Rate * 60



↓

Intermediate code generator

↓

temp1 := inttoreal (60)
temp2 := id3 * temp1
temp3 := id2 * temp2
id1 := temp3

↓

Code optimization

↓

temp1 := id3 * 60.0
id1 := id2 + temp1

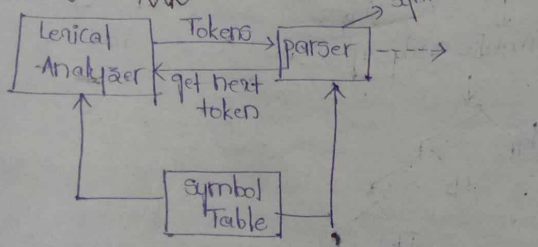
↓

Code generation

↓

MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

* Lexical Analysis:-



parser sends requests for tokens to Lexical Analyser then sends the tokens both are combinely working but both of them are seperated for simplification & attached to symbol Table.

* lexical Analyser forms tokens based on Transition state.
* Transition diagrams are stored in lexical Analyser.
* a|b|c

based on pattern it identifies the token whether it is identifier keyword etc...

* It is identifies as lexim.
* every part is identifies by matching with existing pattern

Token:-

Token is a sequence of characters that can be treated as a single logical entity. they are:-

identifiers, keywords, operators, special symbols, constants

pattern:-

A set of strings in input for which the same token is produced as output.

* this set of strings is described by a rule called pattern associated with Token.

Lexeme:-

A lexeme is a sequence of characters in source program that is matched by the pattern for a Token.

Ex: int a

int → Data type | a → identifier ⇒ existed patterns } Lexeme

a → b

a → identifier ⇒ Not Matched

Token	lexemes	pattern
Constant	Constant	Constant
if	if	if
relation	=, <, >, <=, >=, <>, a	<or <= or <> or == or ≥ or >=

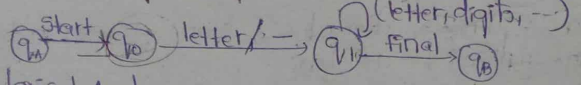
id letter followed by tokens and digits.
pi
num 3.14 any numeric constants

* E=|M|*|C|**|2 = 7 tokens
* C="Hello" ⇒ 3 tokens
* Error Recovery:- Eg: if → fi
* If is panic mode recovery.

Simple strategy.

* In panic mode error recovery we delete successive characters from the remaining input until the lexical Analyser can find a well formed token / suitable pattern Token.
* other possible error recovery actions :—
i) Inserting a Missing characters. (int → it)
ii) Replacing an incorrect character by correct character. (inf → int)
iii) Transposing Two adjacent characters. (if → fi)
iv) Delete number of characters (Eg: int → int)

Rule:*
(letter + underscore) (letter + digit + alphanumeric characters)*
(This is Regular expression for identifiers)

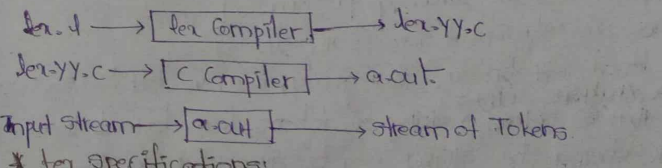


* lexical-Analyser removes spaces.

Lex Tool

* Lex Tool is a lexical Compiler
No. of Lex Tools → flex depends on Regular expressions.
Initially we write lex program that is Compiled by lex tool and gets output as C program, c file.
Extension → .l , produces → lex.yy.c
C file is Compiled by C Compiled and gets object file.
object file produces stream of Tokens.
* there is a wide range of Tools for Construction of lexical analysers the majority of these tools are based on regular expressions.
* one of the traditional tool of that kind is lex and another tool is flex.

steps
i) A specification of the lexical analyser is prepared by creating a program lex.l in the lex language.
ii) the lex.l is run through the lex Compiler to produce a C program lex.yy.c
iii) The program lex.yy.c Consists of a Tabular representation of a Transition diagram Constructed from the regular expression of lex.l
iv) Finally lex.yy.c is run through the C Compiler to produce an object program a.out which is the lexical Analyser that transforms an input stream into a sequence of Tokens.



* lex specifications:—
* A lex program Consists of three parts.
i) Declaration
%% → used to seperate the parts.
ii) Translation Rules
%%
iii) Auxiliary procedures

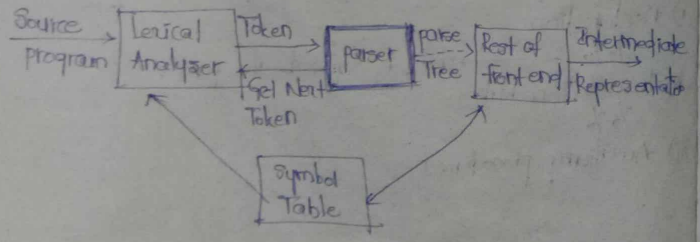
i) Declaration :—
the declaration section includes declarations of variable, constants, and regular definitions.

ii) Translation Rules :—
the Translation Rules of a lex program (or) statements of the form $P_1/P_2/P_3/\cdots/P_n$

P_1 (Caction 1)
P_2 (Caction 2)
P_3 (action3)
!
P_n (action n)

(Based on regular expressions we write action in C language)

iii) Auxiliary procedures :—
Whatever Auxiliary procedures are needed by the actions and these procedures can be compiled seperately and loaded with lexical Analyzer.
* These procedures are already precompiled.
* Here we use only Context free Grammar.
* Right linear, left linear, Ambiguity, Remove left Recursion.
* Syntax Analysis (ii)
* the parser obtain a string of tokens from lexical Analyser and check whether the string can be generated by the grammar for source language.



In parser it scans i/p from left → right only one value.
* Top-Down parsing Techniques :- Root → yield (leaf Nodes)
* Down-Top parsing Techniques :- Yield → Root (leaf Node)
(Bottom-up)
* A parser for programming language always make a single left to right scan over the i/p. Everytime it take one at a time. A parser can be constructed for any grammar. the methods commonly used in Compilers are classified as
i) Top - Down parsing
ii) Bottom-up parsing.
* syntax analyser also have error handling.
Assignment → Error handling Techniques in syntax analysis or in a parser.

Context free Grammar :—
Definition.

* Derivations.

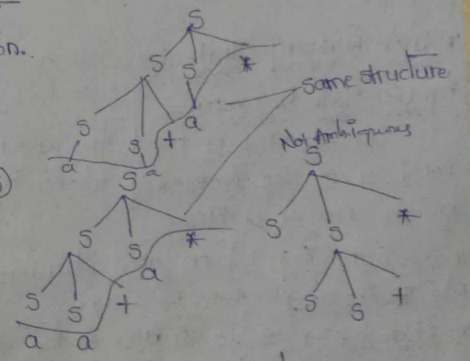
G:
$S \rightarrow SS+ / SS* / a$

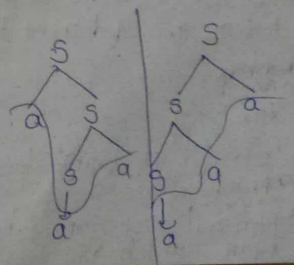
$G(\{S\}, \{+, *, a\}, P, S)$

$w = aa+a*$

$S \rightarrow SS*$
$\rightarrow SS+S*$
$\rightarrow SS+a*$
$\rightarrow aa+a*$

$S = aS / Sa / a.$
$w = aS$
aSa
$w = aaa$
(Different structures)
(Ambiguous)





closure properties for all:—
Assignment

* Syntax Analyzer or parser:—
Analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not.

* Syntax Error Handling:—
The error handler in a parser has goals that are simple to state but challenging to realize:—
* Report the presence of errors clearly and accurately.
* Recover from each error quickly enough to detect subsequent errors.
* Add minimal overhead to the processing of correct programs.

* Error Recovery Strategies:—

i) panic-Mode Recovery:—
Once an error is found, the parser intends to find designated set of synchronizing tokens by discarding input symbols one at a time.
* Synchronizing tokens are delimiters, semicolon or } whose role in source program is clear.
* When parser finds an error in the statement, it ignores the rest of the statement by not processing the i/p.

Advantage:—
* Simplicity
* Never get into infinite loop.

Disadvantage:—
* Additional errors cannot be checked as some of the input symbols will be skipped.

ii) phrase level Recovery:—
* When a parser finds an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. the corrections may be
* Replacing a prefix by some string.
* Replacing comma by semicolon.
* Deleting extraneous semicolon.
* Inserting missing semicolon.

Advantage:—
* It can correct any input string.

Disadvantage:—
* It is difficult to cope up with actual error if it has occured before the point of detection.

iii) Error productions:—
* The use of the error production method can be incorporated if the user is aware of common mistakes that are encountered in grammar in conjuction with errors that produce erroneous constructs.
Ex:- Write 5r instead of 5*r

Advantage:—
* If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.

Disadvantage:—
* The disadvantage is its difficult to maintain.

iv) Global Correction:—
* The parse considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is

error-free.

* When an erroneous i/p statement X is fed, it creates a parse tree for some closest error-free statement Y.

Advantage:-

* This may allow the parser to make minimal changes in the source code.

Disadvantage:-

* Due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

* Top-Down parsing Techniques:-

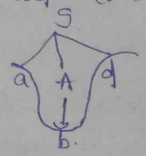
* Back-tracking is also called Brute Force Method.

i) Brute-Force Method (Back-Tracking):-

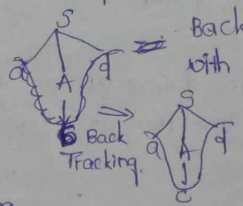
G:- S → aAd/aB
A → b/c
B → cd/dd c

We fail to produce from first production again we start Back-tracking.

w = abd (Scans input from left to right)

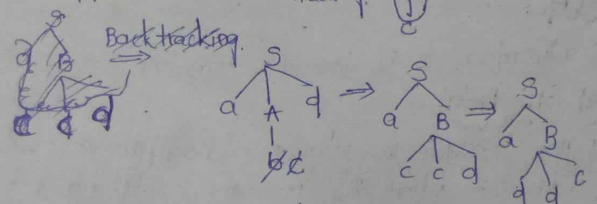


w = acd.



Backtracking is possible with recently used one

w = add/c

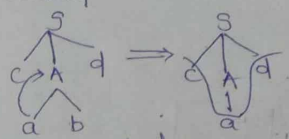


Error occurs when we fail to produce given string.

* Back tracking is possible based on recently used

G:- S → cAd
A → ab/a.

w = cad



(Not Match with the i/p string so we choose alternative production Rule)

ii) Recursive descent parser:- (Repeat same function)

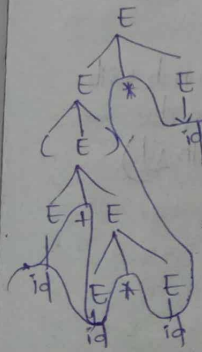
A parser that use a set of recursive procedures to recognize its input with no backtracking it is called a Recursive descent parser.

G:- E → E+E
E → E*E
E → (E)/id

[∵ We get left Recursion, when the same Non-terminal is present at left side & right side]

E+ w = (2+3*4)*5

w = (id + id * id) * id.



Here we have different structures for same i/p & same string ⇒ Ambiguous.

$A \to \beta/\beta z$

$z \to \alpha/\alpha z$

Removing left recursion:—

G_1: $E \to E+T/T$

$T \to T*F/F$

$F \to (E)/id$

Rule:—

$A \to A\alpha/\beta$ (i)

$A \to \beta A'$ (ii)

$A' \to \alpha A'/\epsilon$ (iii)

$E \to TE'$

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

$T \to FT'$

$T' \to *FT'/\epsilon$ → (i)

$F \to (E)/id$

G_2: $S \to as/sa/a$ Rules:—

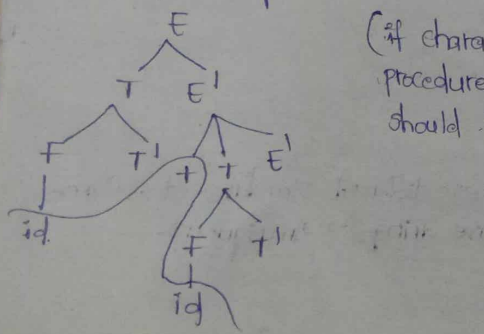
$S \to \underset{\alpha}{sa}/\underset{\beta_1}{as}/\underset{\beta_2}{a}$ (i) $A \to A\alpha/\beta$

$S \to as\,s'\,as'$ (ii) $A \to \beta A'$

$s' \to as'/\epsilon$ (iii) $A' \to \alpha A'/\epsilon$

First we write procedures for Non-terminals which are at left side. (from G_1)

$w = id+id$ string is stored in Buffer.



(if character is match with procedure we take, else we should replace with id)

Terminal → i/p buffer symbol

Non-terminal → procedure.

Code for eg (i)

Advance → pointer moves to next position

```
procedure E();
begin
  T();
  EPRIME();
end;
procedure EPRIME()
if input_symbol = '+' then
begin
    ADVANCE();
    T();
    EPRIME();
end;
procedure T()
begin
   F();
   TPRIME();
end;
procedure TPRIME()
if input_symbol = '*' then
begin
    ADVANCE();
    F();
    TPRIME()
end;
procedure F()
if input_symbol = 'id' then
  ADVANCE();
else if input_symbol = '(' then
begin
    ADVANCE();
```

```
    E();
    if input_symbol = ')' then
    ADVANCE();
    else ERROR();
end;
else ERROR();
```

* To avoid the necessity of a Recursive language we shall also consider a Tabular implementation of recursive descent called predictive parsing. It is also called Non-recursive parsing.

* predictive parsing :— (Non-recursive predictive parser)
(or) (Table driven predictive parser)
We use stacks some operations, Table.

Rules :—

i) the predictive parser has an input, stack and a parsing Table & o/p.

ii) the i/p contains the string to be parsed, followed by i/p dollar-(end marker $)

iii) the stack contains a sequence of grammar symbols. (Variables & Terminals) preceeded by dollar '$' bottom of stack start marker.

iv) Initially the stack contains the start symbol of the grammar preceeded by dollar.

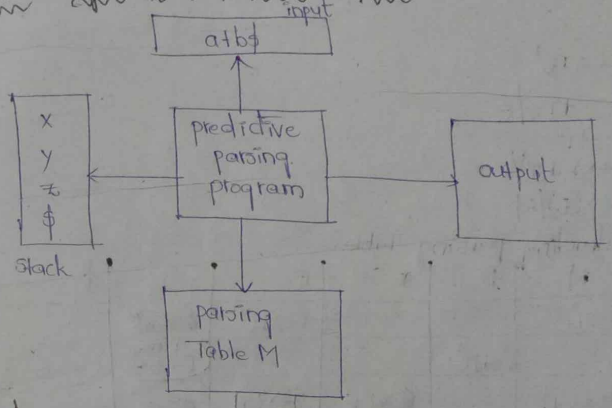
v) the parsing Table is a two dimensional array M[A, a] where A is a Non-terminal & a is a terminal (or) the symbol '$'.

S → SS/A
A → a

	a	$
S		
A		

rows ⇒ Non-terminals
columns → Terminals

Block Diagram of a predictive parser:—



Rules :—

* 'X' the symbol on the top of the stack and 'a' is current i/p symbol. these two symbols determine the action of the parser.

i) if X = a = $ (stack & i/p is end). the parser halts and announce successful completion of parsing.

ii) if X = a ≠ $, (Top of the stack & i/p symbol are same). the parser pops 'X' of the stack and advances the i/p pointer to the next i/p symbol.

iii) if 'X' is a Non-terminal, the program consult every M[X, a], where i/p symbol is terminal & both are different of the parsing Table M. this entry will be either an X production of the grammar (or) an error entry.

→ if M[X, a] = {X → UVW} the parser replaces X on top of the stack by UVW

→ if M[X,a] = error, the parser calls an error recovery routine.

G: E → TE'
E' → +TE' / ε
T → FT'
T' → * FT' / ε
F → (E) / id

1) Constructing parsing Table.

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε (follow)	E'→ε (follow)
T	T→FT'			T→FT'		
T'		T'→ε (follow)	T'→*FT'		T'→ε (follow)	T'→ε (follow)
F	F→id			F→(E)		

E→TE'
T→FT'
F→(E)/id

i.e E→TE'
→FT'E'
→(E)T'E'/idT'E'

First of E → E→TE' (only left side) ⇒ E→TE'
→FT'E'
→(E)T'E'/idT'E'
First(E) = {(, id}

first of E

Follows of E → i.e only on right side
F→(E)/id

First(E) = {(, id}
Follow(E) = {), $}

⇒ E→TE'
→FT'E'
→(E)T'E'/idT'E'
Follows(E) = {), $, id}

When there is no followed Non-Terminal at particular Non-terminal i.e

E → TE' (not follows any Non terminal)

So we should take follows of E

i.e E' → has follows of E

So we should replace Follow(E') = {), id} as Follow(E)

if E → TE' (follows any terminal we should replace same thing as follows)

---

* The Construction of a predictive parser is aided by two functions associated with a grammar G. These functions, FIRST and FOLLOW allow us to fill in the entries in the predictive parsing table for G, whenever possible.

1) FIRST :—

If α is any string of grammar symbols, the First(α) be the set of terminals that begins strings derived from α. If α → ε, then ε is also in FIRST(α)

To Compute FIRST(X) for all grammar symbols 'X', apply the following rules until no more terminals or ε can be added to any FIRST SET

i) If X is terminal, then FIRST(X) is {X}

ii) If X is non-terminal and X → aα is a production, then add 'a' to FIRST(X).

If X → ε is a production, then add ε to FIRST(X)

iii) If X → Y1 Y2 ... Yk is a production, then for all i such that of all

Y_i --- Y_{i-1} are non-terminals and FIRST(Y_j) contains ϵ for $j=1, 2 \dots i-1$, (i.e $Y_1, Y_2 \dots Y_{i-1} \to \epsilon$) add every non-$\epsilon$ symbol in FIRST(Y_i) to FIRST(X).

FOLLOW:—

For non terminal A, FOLLOW(A) to be the set of terminal a, that can appear immediately to the right of A in some sentential form i.e $S \to \alpha A a \beta$ for some α and β to compute FOLLOW(A) for all non-terminals A, Apply the following rules until nothing can be added to any FOLLOW set.

i) $ is in FOLLOW(S), where S is the start symbol.

ii) if there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except for ϵ, is in FOLLOW(B).

iii) if there is a production $A \to \alpha B$ or a production $A \to \alpha B \beta$ where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

* Construction of a predictive parsing Table:—

Input:— Grammar G.

output:— parsing Table M

Method:—

1) For each production $A \to \alpha$ of the grammar do the following.
   i) For each terminal a in FIRST(α) add $A \to \alpha$ to M[A, a]
   ii) If ϵ is in FIRST(α), Add $A \to \alpha$ to M[A, b] for each terminal b in FOLLOW(A).
   iii) if ϵ is FIRST(α) and $ is in FOLLOW(A), add $A \to \alpha$ to M[A, $]

2) Make each undefined entry of M be an error.

G:

(E) $\to$ TE'

E' $\to$ +TE'/ϵ

T $\to$ FT'

T' $\to$ *FT'/ϵ

F $\to$ (E)/id

i) First(a) = {a}

FIRST(E) = {+, (, id}

FIRST(E') = {+, ϵ}

FIRST(T) = {(, id}

FIRST(T') = {*, ϵ}

FIRST(F) = {(, id}

* Based on these values we write production rules.

* FOLLOW:—

1) FOLLOW(E) = {$,)}

FOLLOW(E') = {$,)} $\Rightarrow$ Rule(iii)

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

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

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

Rule ii

Rule(i) E $\to$ TE'

F $\to$ (E)/id & Rule(i)

follow

if we get ϵ again we write Follow(E') $\Leftrightarrow$ Follow(T)

* Stack Implementation :—

* Stack Implementation:

	Stack	Input	Action
1	$E (initial)	id+id*id$	E→TE'
2	$E'T	id+id*id$	T→FT'
3	$E'T'F	id+id*id$	F→id
4	$E'T'id	id+id*id$	pop → remove from stack & pointer move to next
5	$E'T'	+id*id$	pop
6	$E'T'+	+id*id$	T'→ε
7	$E'	+id*id$	E'→+TE'
8	$E'T+	+id*id$	pop
9	$E'T	id*id$	T→FT'
10	$E'T'F	id*id$	F→id
11	$E'T'id	id*id$	(pop)
12	$E'T'	*id$	T'→*FT'
13	$E'T'F*	*id$	(pop)
14	$E'T'F	id$	F→id
15	$E'T'id	id$	(pop)
16	$E'T'	$	T'→ε
17	$E'	$	E'→ε
18	$	$	ε'→ε ∴ Accepted.

terminals (pop)

* Construct a predictive parser for the following Grammar

G: S → (L)/a ⟹ Contains left recursion (remove it first)

L → L,S/S

G': S → (L)/a

L → SL'

L' → ,SL'/ε

FIRST(L) → {(, a}

FIRST(S) → {(, a}

FOLLOW(L) → {)}

Homework

* LL(1) Grammar parsing:

↑ grammar whose parsing table has No multiple defined entries is said to be LL(1)

	a	,	)	$
S	S→a			
L	L→SL'			
L'		L'→,SL'	L'→ε	

↑ Identify whether LL' (or) not. (Not LL(1)) (Left factoring present i.e starting with 'a')

* The first L in LL(1) → left most derivation.

↑ Imp scans i/p from left to right

It is top-down parsing.

* In Bottom-up parsing. LR(1) → right most derivation in reverse

↑ Imp scans i/p from left to right

(1) → No. of look ahead symbols are one

* Ambiguous Grammar are not LL(1) Grammars. bcz we get multiple entries in grammar.

* Before solving we should check whether it is ambiguous or not.

* suppose there are 8 production rules,

* If there are errors, we place synchronization to skip the error only for follows which not having 'ε' production rules.

y= FOLLOW(E) = {$,)}

j=

in

Fc

r

Fc

Cor