Static array - array where size is fixed during declaration

Semi-dynamic array - size during runtime.

Dynamic array - array whose size is decided during runtime. or data type

```
int * p = new int [size];
```

After dynamic memory allocation size cannot change.
Once array is created it cannot change.

During execution if type of array can be changed then dynamic array. Type → (data type or size).

During a function call when control is passed to a function then the called function uses the same registers as the calling function since number of registers is fixed. During transfer of control saving of data in registers is done. Before transferring control to called function the calling function again puts back the data in registers. The called function stores the contains of registers in main memory, not the calling function. Along with register contents, some local variables are also stored.

When function A calls B, the starting address of B is stored and return address is stored in one of the registers of A. There are total 16 general purpose registers numbered from 0-15.

→ address of instruction → general purpose register

1004 BALR 14, 15 → branching address / address of called unit

↓ return address

[register 14 will store return address of called unit]

BR 14

[branch to content of register 14] last instruction of called unit.

The responsibility of registers is given to called unit.

A → B → C

During return C restores contents of reg B and B restores contents of A.

Register 13 is used to save address of register save area.

When called unit is returning a single value that value is stored in register 0.

When more than one value the address of these values is stored in register.

FORTRAN, COBOL does not support recursion → does not have separate code segment and data segment

Scope rules of a language - how a language accesses local variables.

Block structured language -

variable - entity

rval — entity.

Each entity will have certain attributes.

Variable's attributes - type, scope, lifetime, lval, rval

↓

decides size

name of a variable is not a property, its an identifier.

lval → address allocated to that name.

If the name appears on the left side side of an assignment then its lval is used and if on right side of assignment then its rval is used.

$$a = a + b.$$

$$lval(a) = rval(a) + rval(b).$$

lval is associated only with identifiers, not with expressions. So,

$$x + y = 3 ;$$

is not a valid statement because LHS is an expression.

---

lex → Unix

Flex → Linux (open source)

Specifications to lex :-

$ cc test.c → for compiling a C program.
a.out

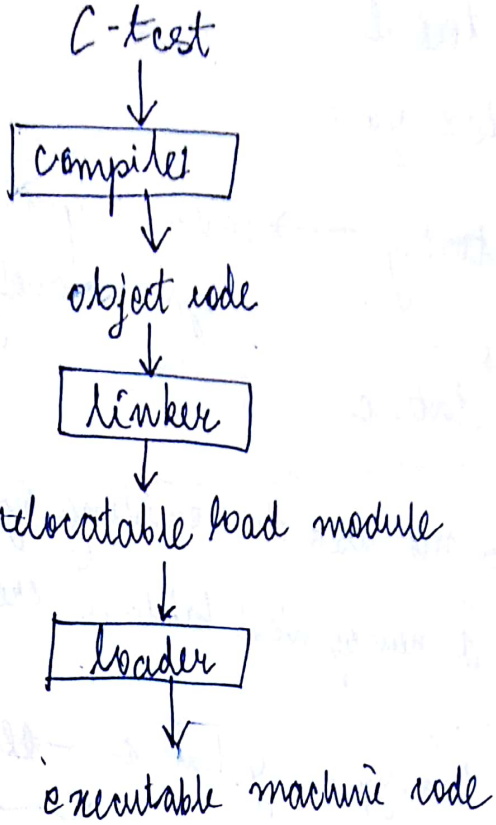
$ file test.c → for finding type of file

$ file a.out → pure executable statements

$ cat a.out → displays junk character.

Difference b/w join and merge is join is vertical and merge is horizontal.

In $ cc test.c, cc is not just C compiler.
Along with compiling cc also performs linking and loading.



.l is for lex specification

The command for checking lex specification is,

$ lex test.l

Declaration section
Rules section → grammer rules.
codes of rule section

} sections of lexical analyzer.

Every rule will contain two parts. The left part is a regular expression (specification) and right part contains the action to be taken.

The lex generates another C-program

$ lex test.l

lex.yy.c

$ yacc -d test.y → contains specification for parser or syntax analyzer.

generates

y.tab.c

If -d is not used the existing symbol table is modified otherwise a new symbol table is created.

$ cc lex.yy.c y.tab.c -ll -ly

for including library during linking.

The library routines are precompiled and stored in specific locations.

#include<math.h>

.h files contain function prototypes and not the actual code.

#include "math.h"

↳ we can give location using double quotes.

The default directory of include statement is

"/usr/include".

The name of the directory itself is "include".

While checking for presence of a.out file, it is checked in default directories along with current directory.

./a.out → a.out of current directory.

The environment variable PATH has value

PATH = $PATH::

To this variable we need to add another directory.

A particular user's profile is specified in .profile

Whenever a profile is created its default .profile is created.

---

lex/flex -

- lex is a scanner generator

  input — set of regular expression and associated actions (written in C)

  output — table-driven scanner (lex.yy.c)

- flex — an open source implementation of the original UNIX lex utility

lex input example -

filename : ex1.l

%%
→ first part empty

"hello world" → Regular expression → action → printf("GOOD BYE \n");

second part

;

prints GOODBYE whenever string "hello world" is encountered.

%%
→ third part empty

. → does nothing for any other character

Using lex -

```
% lex ex1.l          → process lex file to generate a scanner (gets saved as lex.yy.c)
% cc lex.yy.c -ll    → compile the scanner and grab main() from the lex library (-ll option)
% ./a.out            → run scanner taking input from std input
hello world          → standard input
GOODBYE              → standard output
%
```


- lower alpha
- upper alpha
- nums
  - ints
  - float
- digits
- words
- characters
- whitespaces
- lines.
- special character

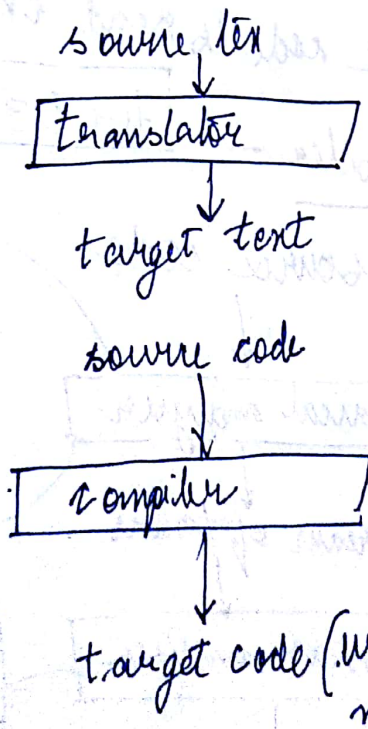
Anything other that digits, alphabet and whitespace is a special character.

Symbol table - array or LL

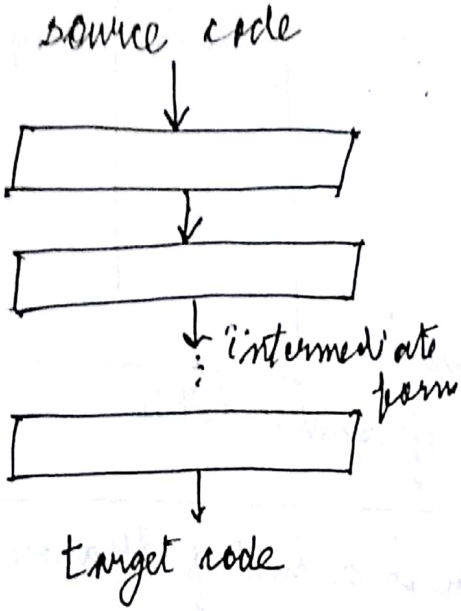
word	frequency	line number
		(a word in one line is ~~entir~~ considered only once).

generation of symbol table

Any compiler is a kind of translator



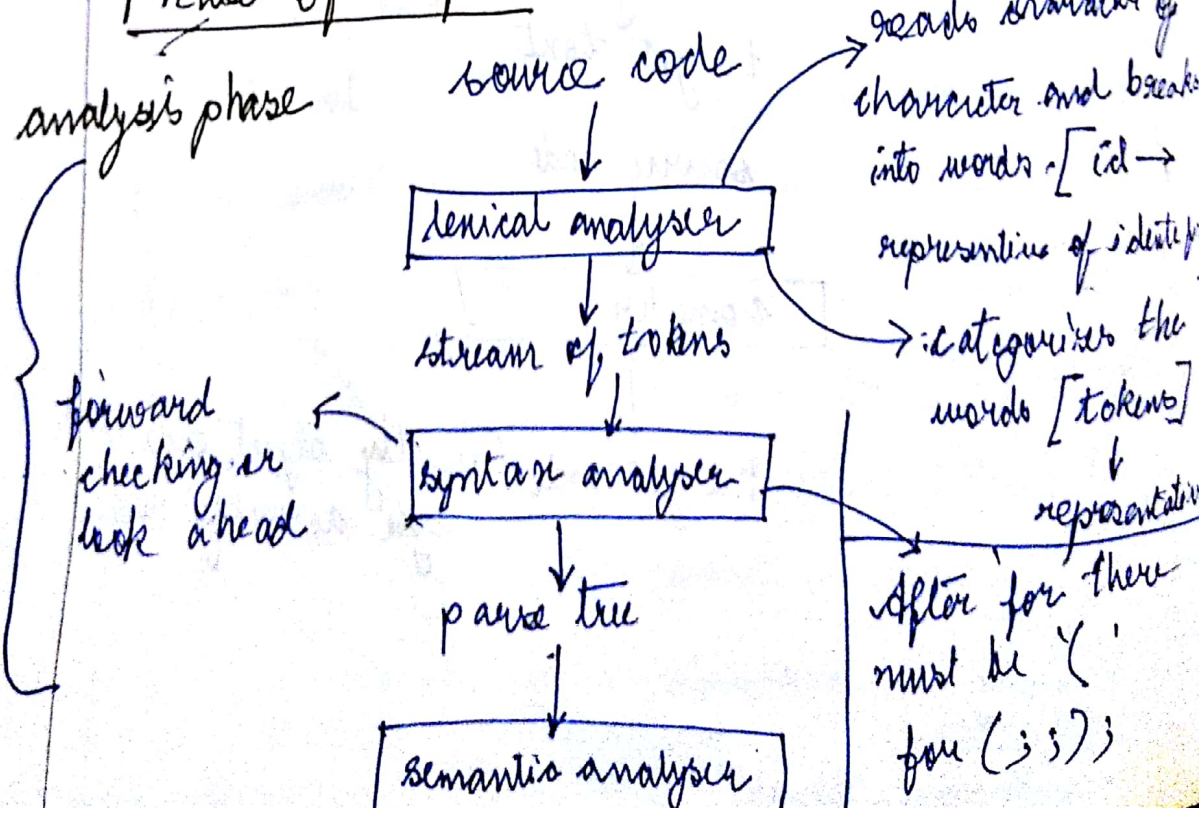
converting source code to target code is done in steps by converting into some intermediate form. The intermediate form is converted into target form in phases.



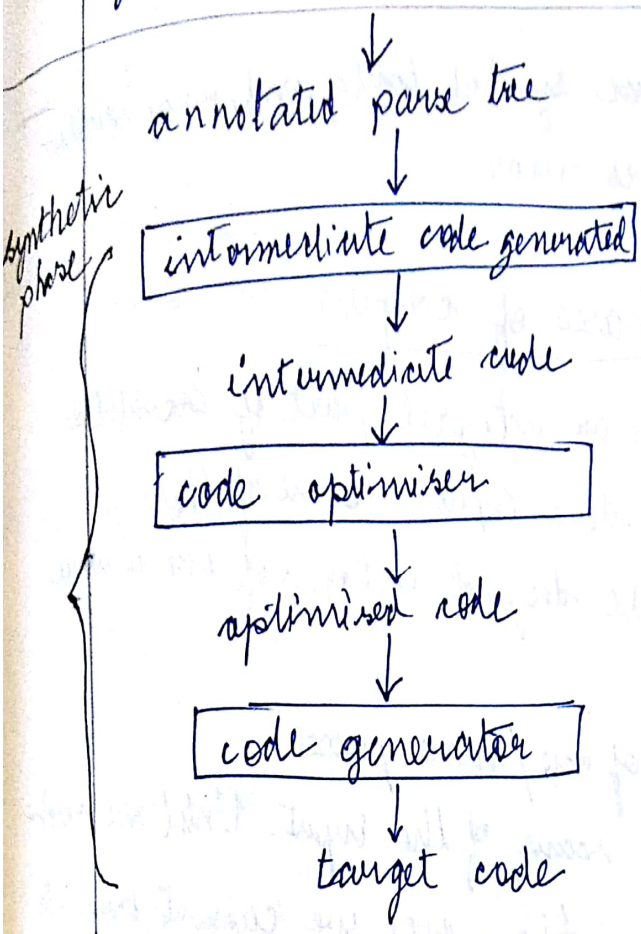
The input source code is read character by character.

for i := 1 to n do

Phases of compiler –



syntax analysis generates parse tree → so also known as parser.

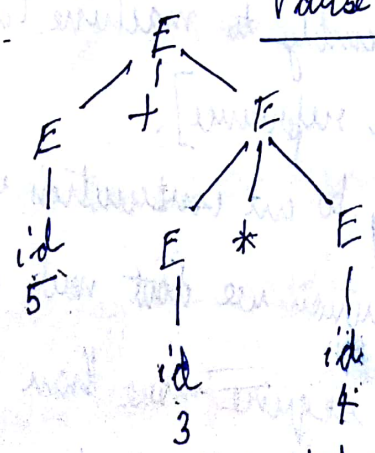


syntax analyzer checks grammar rules and checks whether the string is an yield of a parse tree.
- (membership problem)
- (order of derivation)
- (if the string is not a member then check for error)

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

$E \rightarrow id \mid num$

Annotated Parse tree



Parse tree is travelled in bottom up in level order.
We get a dependency graph.
Topographical ordering gives sequence of evaluation.

semantic analysis -

Type checking is done

i = x * 10

↓ ↓

float int

semantic → means 'logical meaning' of a statement.

(i) operational semantics (imperative lang.)

(ii) denotational semantics

(iii) axiomatic semantics

We have tools to describe lexical and syntax analyzer, not for semantic analyzer

Scan Scan

These 6 phases use symbol table and error handler finds and rectifies error.

Phases of compiler

Phase - a tool or an integral part of translation

Pass - scan the entire input. Some of these translations can be designed in terms of one or more passes.

A phase consists of one/more passes.

We need multiple scans of the input. Until we read the input more than once we cannot translate properly.

some instructions require two passes for proper translation from assembly to machine language. [Eg - macros, forward reference].

If we are branching to an instruction which is behind current instruction we don't need two passes.

In case of macros we require more than two passes. The preprocessing is done even before lexical analysis.

In compiler design we look at phases, not passes.

The lexical analysis is the only phase where input is read character by character. After this no other phase reads input. It uses output of previous phase.

In code optimisation phase we reduce size of code by modifying intermediate code. Thus we remove unnecessary instructions and reduce overall execution time.

lexical analyzer - (scan input)

Find longest word that matches a particular rule for tokenization.

Eg: id: alpha (alpha / digit)* . [rule for identifier]

That is lexical analyzer matches longest applicable rule classifying the input string and returns a representation for classified set (token).

a
ab
abc
ab2

} all these strings are identifiers (lexical analyzer returns a representative of the "identifier set".

→ a = b + c [5 tokens]
= = = =
id aop id bop id [identifiers and operators]

→ The type of variables is to be defined before using it.

int a, b, c;

When this definition is read the information about symbols used in the program is stored in symbol table.
All attributes associated with an identifier are used in symbol table. [read, local, lifetime, scope, type]
The symbol table consists of pointers, variables.
A pointer of the symbol table points to an array of characters containing the name of the variable.

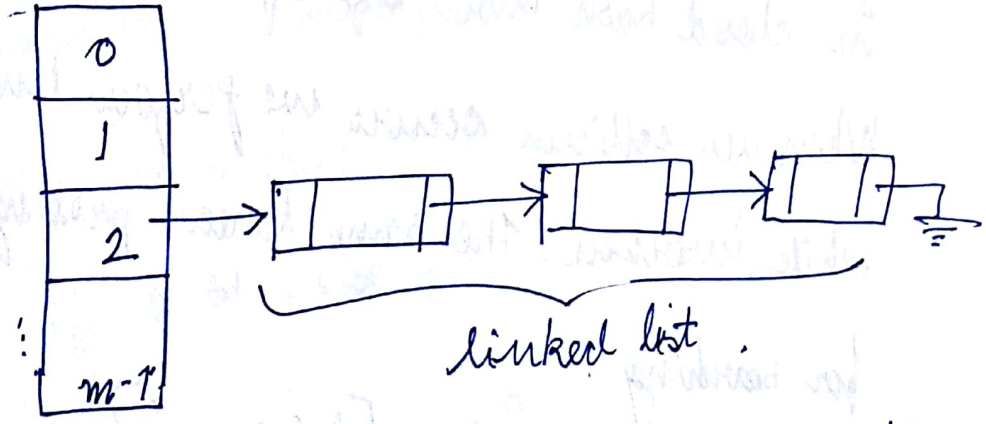
Operations on symbol table-

(i) accessing the variable.
(ii) checking whether variable is already present.
(iii) Inserting a variable in symbol table.
(iv) deleting a variable from symbol table when its scope finishes.

In symbol table for each node we store more than one fields. [AVL tree → height balanced tree
B tree → balanced tree (leaf nodes are at same level].

Hash table can be used to store symbol ~~table~~ table.
[Two methods → open hash table
closed hash table].

In open hash table we convert a string into an integer x which is an index of table.



If there are n variables and table size is m then each linked list is assumed to be of size n/m on an average. A particular rule is followed for hashing -

"a" → 6
"abc" → 6
} both strings hashed to same location

x → k → i
k%m

For finding k,

$3 * 32^0 + 2 * 32^1 + 1 * 32^2$

The base is taken as a power of 2, so that we dont need to multiply, we can just perform left shift.

But in open hash table the size of linked list is not fixed. So we use closed hash table.

In closed hash table size of table is fixed.

Whenever collision occurs we perform linear probing while insertion. The same linear probing is used for searching.

Linear probing $\longrightarrow [h(x) + i]$

The problem of linear probing is formation of clusters. In order to avoid that we go for quadratic probing.

$[h(x) + i^2] \longrightarrow$ we may not be able to reach all locations in table.

If load factor < 0.5, then free spaces are occupied using quadratic probing.

$$\text{load factor} = \frac{\text{occupied space}}{\text{total space}}.$$

---

consider an arithmetic expression,

$x = a * b + 10;$ → incomplete in C without semi-colon

The ; in Pascal is a separator of statements and a terminator in C-language.

Pascal – if e then s1 else s2

C – if e s1 ; else s2 ;

alpha (alpha | digit)* → rule applied to recognise identifiers.

The text must match one of the rules.

$x \longrightarrow$ lexeme	$x = a * b + 10$
id $\longrightarrow$ token	id = id * id + num
	[stored in symbol table].

lex analyser returns token and actual text through yytext

Symbol table -

	Token	Type	Scope → attributes of variable		
x	id	float			
a		float			
b		float			
10	num	int			

The information related to each lexical entry is stored into symbol table. The symbol table is used by subsequent phases.

Stream → delimited

string → consequent; no delimiter.

The next phase checks a derivation sequence for id = id * id + num.

We can construct a derivation tree for this string. Derivation tree is sometimes called parse tree. In syntax analysis, again we are checking membership problem. In lexical analysis also we are checking membership of a string in a language of some grammar rules.

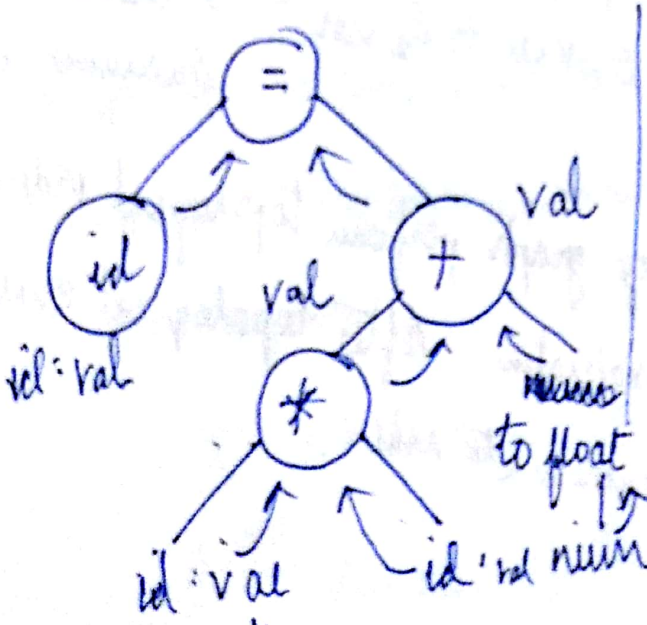
lexical analyser uses recursive constructs and syntax analyser uses recursive construct.

$E \rightarrow E + E$ (recursive)

$E \rightarrow AB$ (non-recursive)

lexical and syntax analysis is done simultaneously but for ease of understanding we take them as separate phases. lexical analyser is an integral part of syntax analyser.

Derivation Tree. (id = id * id + num)



$S \rightarrow id = E$

$E \rightarrow E + E / T$

$T \rightarrow T * F \mid F$

$F \rightarrow id \mid num$

(syntactic analysis)

↓

does type checking

↓

Annotated parse tree

(decorated)

(created from parse tree)

[Dependency graph]

↓

created from annotated parse tree

The grammer rules must be in Backus-Normal Form. or in EBNF.

↓

some special characters represent other characters → meta-characters

$ run * → represent "all"

$ run -i *

[enquires before deleting].

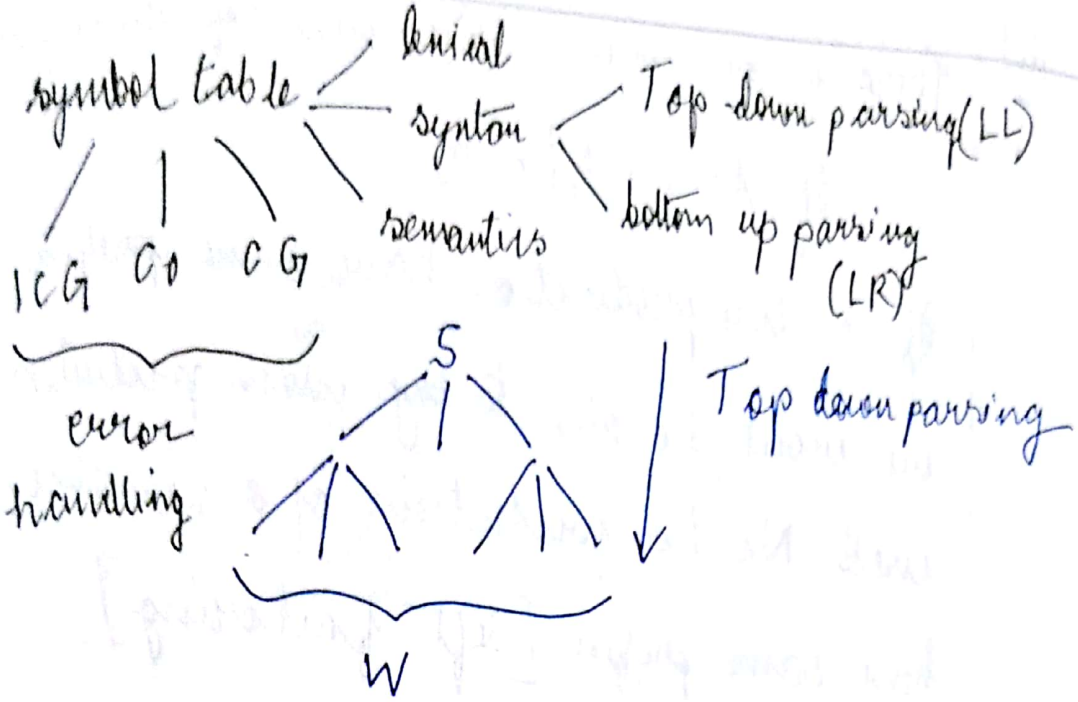
. is a meta-character in yacc.

'|' is a meta-character in yacc.

E = E + E

E.val = E_1.val + E_2.val → attributed grammer rule

In dependency graph perform topological ordering or topological evaluation. After topological evaluation we get intermediate code.



Most compilers cant perform top down parsing.
Expression grammars can be parsed using top down parsing.
Most general parsing technique is bottom-up.

LL Parser is a topdown parser using leftmost derivation sequence.

$$A \rightarrow A\alpha \text{ (immediate left recursion)}$$

$$A \xrightarrow{*} A\alpha \text{ (left recursion)}$$

① To apply ll parsing technique the grammar should not be left recursive.

(ii) There is one more issue with top down parsing

If $A \rightarrow \alpha_1 \beta_1 | \alpha_1 \beta_2$

i.e. two productions have same prefix α_1

we wont be able to say which production to be used. No two productions of a variable should have same prefix [left factoring].

To perform ll parsing - remove left recursion and perform left factoring.

LR parsing - rightmost derivation sequence in reverse order.

↓

i/p read from left to right.

[first we get rightmost sentential form and finally we get S].

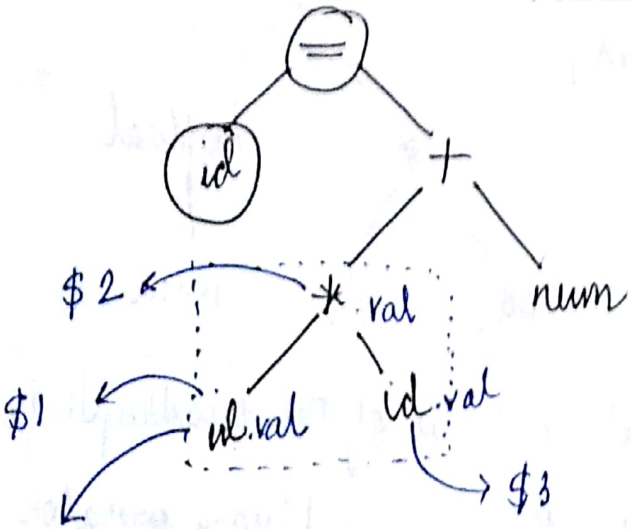
A lookup table is created using lr parsing (3 methods)

LR parser

- SLR (simple LR)
- canonical LR
- LALR [look ahead LR parser] eg: bison/yacc

Illustration

$id = id * id + num$



suppose production rule used is $E \rightarrow E * E$.

[yacc directly generates intermediate code from syntax analysis]

$$\$\$ = \$1 * \$3$$

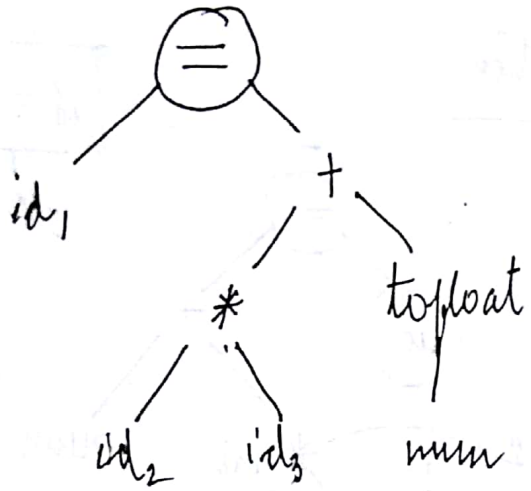
- $\$\$$ → value of attribute on left side
- $\$1$ → value associated with 1st token on right side
- $\$3$ → value associated with 3rd token on right side
- $*$ → intermediate code

$E.val = E_1.val * E_2.val$

(Syntax directed translation)

[goto instructions are essential for an intermediate code generation].

#



Intermediate code is of the quadruple form.

Ex- $a = b$

$a = b \text{ op } c$ → binary operator

'?:' is a ternary operator which is right associat

Ternary operators are converted into binary operator during ICG. → temporary identifier names.

$t_1 = id_2.val * id_3.val$

$t_2 = tofloat(num.lexval)$

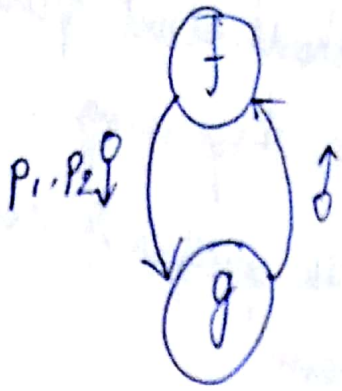
$t_3 = t_1 + t_2$

$id_1 = t_3$

Most of the temporary identifier names generated by system begin with '_' (underscore).

We have to avoid memory references and number of instructions during code optimisation.

Eg- $id_1 = t_1 + t_2$



static nesting of function

lex specification - (structure)

%{
Declaration section
%}

%%

Rules (Regular Exp. Action) → functions can be called here which are written in auxiliary function section

%%

Auxiliary functions

%{
C - definitions (copied verbatim to lex.yy.c by lex)
%}

regular definitions → symbolic naming for a class of character

%%

Example of regular definition —

digit [0-9]

digits digit+

- Every lex statement should start from column 1. Do not leave a space at the beginning.
- yylex() → invokes lex specifier to convert the rules into a C-program.
- lex and yacc do not do lexical analysis.
- yylex → generates a program in C (lex.yy.c) which is the lexer and that C program creates tokens.
- yacc → generates ~~lexer~~ parser program.
- lex matches longest input string.

Suppose input string is "abc" and rule is [a-z] then only "a" is matched.

[a-z]+ → also called a pattern.

"abc" → lexeme (the actual text which is matched against a rule)

yytext() returns the lexeme

[a-z]+ "ab"

[a-z]+

Every rule returns a token

[a-z]+ —action→ returns tokens

return (ID) → manifest constants

↓

value is returned to yylex() & the system again and again calls yylex().

- Only the first applicable rule is considered.
- '.' dot is a metacharacter → matches every character other than newline characters.
- space [\t\n]
- alpha [a-zA-Z]
- ws [space]+.

[ ] → set of characters

[^ab] → all characters except "a" or "b".

[A] → checks for "A".
↓
blank

- lower [a-z]
- upper [A-Z]
- alpha [~~abc~~ lower|upper] {lower}|{upper}

%

{alpha}+ {action1 ; action2 ; ... }

```
{alpha}+    {word++; ch+=yyleng;
                lower+=ent_lower(yylex
            }

%
int ent_lower(char* s){


}
```

Assignment -

① Postfix evaluation

② Postfix to infix

AWK → unix tool

compiler for language lisp - (lots of irrelevant and silly programming) → functional programming

fortran → formula translation

- There are only two data types in lisp →

  ① atom ② list

lisp is not an imperative language.

In an imperative language we have assignment operation and they follow Context Free grammar

- In lisp we give input to a function and the output to this function is given to another function as input. Lisp uses prefix notation for denoting expressions. Prefix is unambiguous with respect to precedence of operators.

- Functional languages do not have a compiler. lisp has a ~~translator~~ interpreter
  ↓
  translates line by line.

- The lisp was developed using 5 basic functions and these functions are used to make other functions. Lisp is inbuilt in linux.

- C-language is a low level language in a way that it has low level of abstraction. Using C program we can invoke linux commands.

```
system(command);
```

↓

control goes to OS. OS executes command and control goes back to program. C language is close to kernel.

Writing a compiler for a Language –

find CFG for the language.

Nobody writes compiler from scratch. The tools existing compilers are used to build new compilers. Can a construct for language Y be written using construct of language X? If yes then tools of X are used.

Bootstrapping – the technique used is called bootstrapping. It is used to reuse compiler X to get compiler Y, cross compilers.

Source language

Implementation language → language in which compiler is written

Target language

S_I^T

language → L_M^N → machine code

L_M^N → language of compiler

The compiler for language L written in language

M used for machine N.

$$L_S N + S_M M \longrightarrow L_M N.$$

concept of cross compiler → intermediary cross compiler

$$L_L N + L_M M \longrightarrow L_M N.$$
$$L_L N + L_M N \longrightarrow L_N N.$$

} Porting a compiler.

The program $L_L N$ is again given as input to intermediary cross compiler $L_M N$ and then we are getting $L_N N$.

M → dirty compiler

N → improved or optimised compiler

The process bootstrapping is "not reinventing the wheel".

Sem. Assignment – [operators in 2nd line have higher precedence than 1st].

Data types: int, unsigned int, boolean.

Assignment: =, +=, -=, *=, /=

Binary operators: +, -, *, /, @ (exponentiation)

left association; right association

Bitwise: |, &, ^, ~

Logical : || , &&

Relational op : == , != , < , <= , > , >=

Assignment stmt : S → id AOP Expression

iterative

conditional : if , if else , switch case

Repetitive : while

id : simple ids (no special characters)

Expressions in infix notation :-

Input : x = a + b * c ;

Output : t = b * c ; } intermediate code

x = a + t ; }

If you have ,

E → E + T

write semantic actions associated with it in yacc Semantic action : E = E.val + T.val

Traverse parse tree in bottom up manner using yacc.

Generally semantic action will be,

$$\$\$ = \$1 + \$3.$$

printf("$$" "=" "$1" "+" "$3");

So next when you compile it as C program, you can generate $$ = $1 + $3.

∴ You have to break an expression in simple binary expression → 1st part.

2nd part → using specification given write a program through it and then use lex and yacc to get the C program.

Contents of documentation :-

- Problem description
- Decomposition
- Approach
- Algorithm proposed
- Description of functional units
- Relationship among functions - i/o.

- static nesting tree
- Implementation details of major / functional units
- doc. of each fn through comments for all non-trivial statements ~~trivial~~.
- sample i/o for testing
- test results
- major limitations / assumptions.

Lexical Analyser

PL/I:

IF THEN → THEN identifier → identifier

THEN = ELSE ELSE ELSE = THEN

↙ boolean var. ↘ valid statement in PL/I.

Keywords in the programming language are not reserved. [THEN → keyword as well as identifier]

- To make keywords reserved -

store them in symbol table and mark it as keyword.

PL/I language was a failure.

The lexical analyser works as a subordinate of yacc due to simplicity of design.

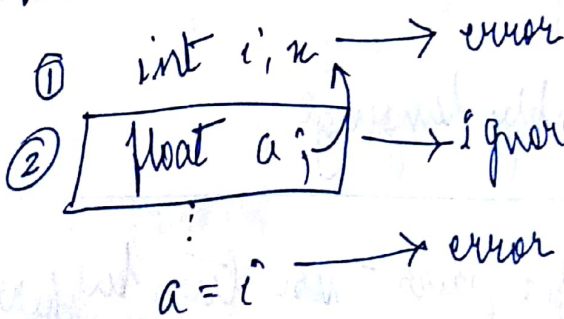
If lexical analyser works as an independent entity then error detection is easy.

For statement,

```
fi(a==x).
```

fi is not a keyword, so its interpreted as an identifier. On using opening and closing parenthesis its interpreted as a function.

Error Recovery - Try to find as many errors as possible.



One of the methods of error recovery is panic mode error recovery.

In statement ① compiler expecting , or ; so it ignores all subsequent characters until it gets , or ;.

Here statement (1) is ignored in pass 1 mode.

Designing a lexical analyzer - (3 ways)

(1) Use a tool lex → lexical analyzer generator

(2) Develop lexical analyzer using a system level program or any low level language like C which allows reading or writing from a buffer character by character.

getc() → buffered input
ungetc() → push back into buffer.
getch() → not buffered input. One can not be pushed back into buffer.

(3) Use assembly language.

Input buffer pairs - use two buffers. Fill one buffer and process it. While processing fill other input buffer. There are two pointers for identifying keyword. One stays at beginning

and other one moves forward. Example when lex reads "for" it may be a possible keyword.

optional [^] set of chrs other than specified

a? → ∈/a

\ → escape characters

[ ] → set of characters

{ }

- range

- Number representation in scientific notation.

atof("1.23E+2") → 1.23 * 10^2

atoi("5678")

n = n * 10 + (ch - '0')

- digits → [0-9]+

opt_fraction → digits | ∈

opt_exp → (E(+/-/∈) digits) / ∈

num → digits opt_fraction opt_exp.

calc.

calc.

Q- Lex program to simulate hex calc. [Assignment]

0-9 a-f +, -, *, /

|-5A| → 5A
or f 5A → 5A
} absolute (own choice)

% → both integers (final remainder)
(preferably +ve integer)

logical: OR, AND, ^, XOR 45 | 76

bitwise OR

Both input and output → hexadecimal.

Q- English Text encryption. [case sensitive]

- special words — different approach
- other words — normal approach
- spot — left rotate by 2 positions (cyclic)
- [print in reverse order]
- other words — right rotate by 3 positions (print rev order)

- Number → divide into two halves and swap.
  [1945 → 4519]
- other characters - retain
- Words like 'can't' should also be accepted.

7- Sub C program — stream of tokens.

---

8. Word frequency [read text file].
   [Print word frequency and line numbers].
   [form a symbol table].

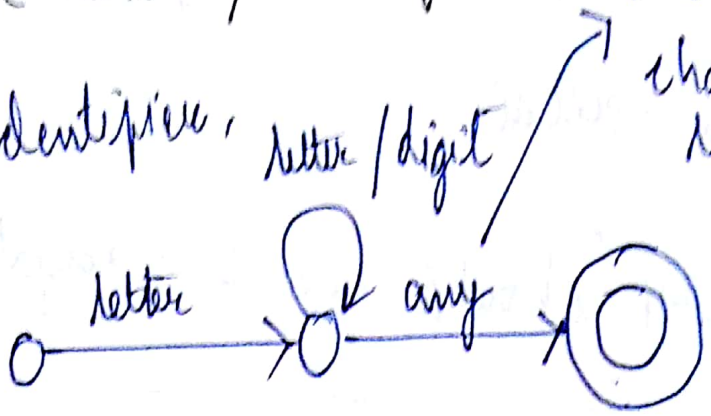
---

9. Infix expression evaluation.
   Output — evaluation order.
   [~~use~~ precedence, associativity to be specified].

$$3 + 4 + 5$$
$$\equiv 7 + 5 \equiv 12.$$

Q Draw a transition diagram for recognizing a number / identifier.

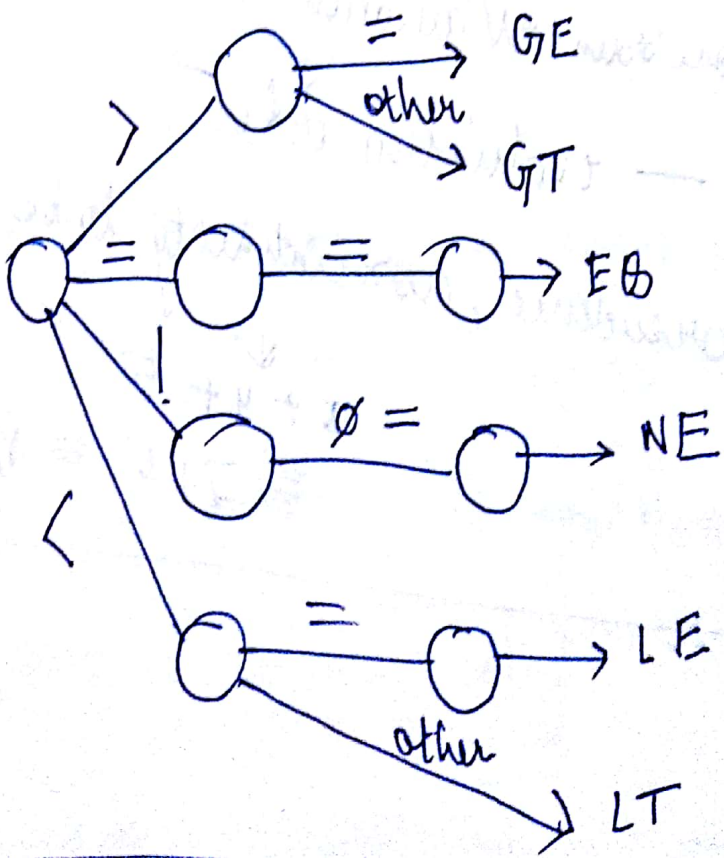
→ identifier, letter / digit



↓

install in symbol table along with attributes.

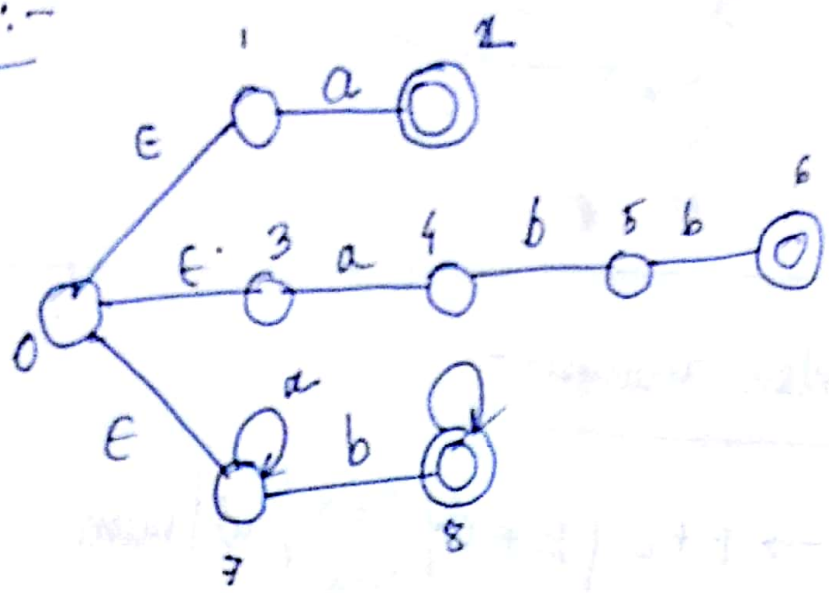
Q- Transition diagram to recognize relational operators



Consider three patterns,

a {action$_1$}
abb {action$_2$}
a^*b^+ {action$_3$}

NFA:-

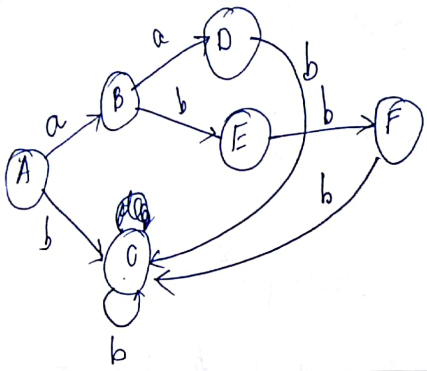


converting into tabular form,

	a	b	token recognized
[0137] A (start state)	[247] B	[8] C	none
[247] B	[7] D	[58] E	(1) 'a'
[8] C	∅	[8] C	(3) a^+b^+
[7] D	[7] D	[8] C	None
[58] E	∅	[68] F	(3) a^*b^+
[68] F	∅	[8] C	(2) abb

[0137] → [247] → [58] not an acceptable word

DFA :-



Syntax analyser -

$E \rightarrow E+E \mid E*E \mid (E) \mid id \mid num.$

we give precedence order of operators to remove ambiguity. The least precedent rule is written first

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

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

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

$F \rightarrow F@G \mid G$ (should be G@F)

In yacc specification :

%token + -

%token * /

%left + - * / → left associative

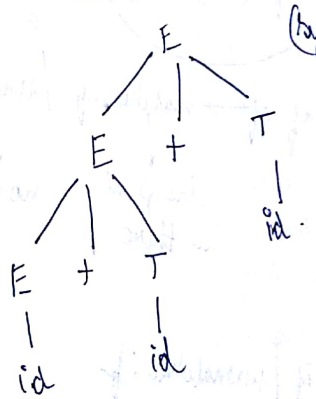
%right @ → right associative.

Derivation of a+b+c :-

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

$\rightarrow T+T+T \rightarrow id+id+id.$

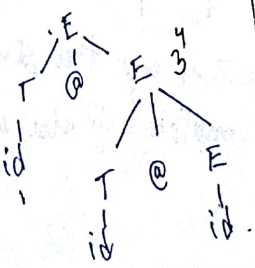
a+b+c



(Syntax tree)

~~E~~ $T \rightarrow F \rightarrow id$ is written as $T \rightarrow id$.

2@3@4 $E \rightarrow T@E$



$E \rightarrow E \, op \, T$ (op is left)

$E \rightarrow T \, op \, E$ (op is right)

Only fortran language uses exponentiation op as $a ** b = a \hat{} b$.

In C we write as function, pow(a,b).

- $S \rightarrow iEtSeS \mid$ (if then else statement).
  $iEtS$.

if E_1 then if E_2 then S_1 else S_2 (ambiguous)

Two kinds of if → matched if | unmatched if
- matched if → else part is there
- unmatched if → no else part

$S \rightarrow$ matched if | unmatched if

matched if → if expr then statement else stmt.

unmatched if → if expr then stmt | if expr then matched if else unmatched if

If error is not detected in lexical analysis phase then it WILL be caught in syntax analysis.

If caught in lex then rest is less.

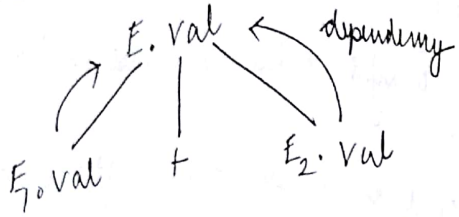
- Top down parser → not left recursive but left factored.

Top Down Parsing

Parsing —
- checking membership
- identify derivation sequence
- construct parse tree
- Attributed parse tree
- dependency graph
- evaluation order

- In expression grammar there is a type associated with every non-terminal.
- Attributes associated with each non-terminal. Not same for every non-terminal. Identifying dependency of parent-child node using attributes.

These attributes are synthesized attributes or inherited attributes.



The value associated with an attribute of a child is inherited from parent.

We need to traverse dependency graph to get evaluation order (intermediate code generation).

Traversal – Topological ordering

- For every grammar production we associate an action

$E \to E + E \quad \{E.val = E_1.val + E_2.val\}$

[syntax directed translation].

Leftmost Derivation Sequence (for top-down parsing)

$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

} left recursive

For a production like,

$A \to A\alpha$

leftmost symbol of RHS is same as LHS: then immediate left recursion.

$A \xrightarrow{*} A\alpha$ (left recursive)

I. Elimination of (immediate) left recursion:-

$A \to A\alpha \mid \beta$.

① separate all productions of A having immediate left recursion

$A \to A\alpha_1 \mid A\alpha_2 \cdots \mid A\alpha_m$
$\to \beta_1 \mid \beta_2 \cdots \mid \beta_n$

We have to modify our grammar rules and show that strings in both grammar rules are same

- $A \to A\alpha_1 \mid \beta \to A\alpha_2\alpha_1 \cdots \to \beta_j\alpha_2\alpha_1$.

$A \to \beta_i B$ $B \to \alpha_j B \mid \epsilon$	$A \to \beta_j B \to \beta_j\alpha_2 B \to \cdots$ $\to \beta_j\alpha_2\alpha_1 B \to \beta_j\alpha_2\alpha_1$.

Example - removing immediate left recursion for expression grammar.

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

II. Left Factoring - (removing productions having same prefix)

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_k$.

same prefix

(helps in checking unmatched i/p)

$A \rightarrow \alpha A'$

$A \rightarrow \beta_1 \mid \beta_2 \dots \mid \beta_k$.

- Left recursion removed and left factored.

Q. Removal of left recursion and left factoring given a CFG. [you may use ln].

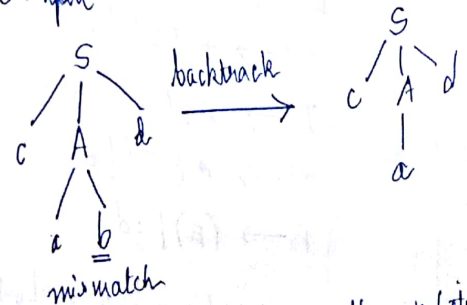
Top Down Parsing
[recursive descent parsing]

- requires backtracking [inefficient algorithm]

Example: $S \rightarrow cAd$

$A \rightarrow ab \mid a$

consider input: cad.



Whenever you get mismatch, check for other productions of A. Keep two pointers i, j. Increase j and pointer as long as you are getting match.

Predictive parser – (a top down parser which does not require backtracking but might use recursion).

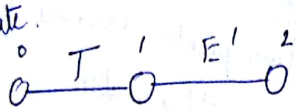
Non-recursive predictive parser :– L to R leftmost derivation → (LL(1) parser)

If predictive parsing table does not have multiple entries then LL(1) parsing – one character at a time → (will have conflicts).

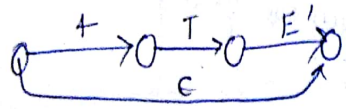
$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow (E) \mid id$$

For every variable we have a starting state and a final state.

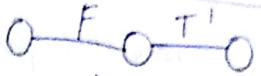
E :



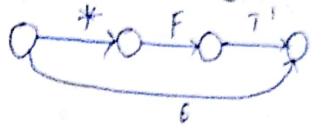
E' :



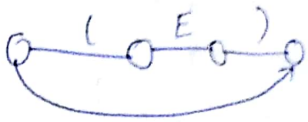
T :



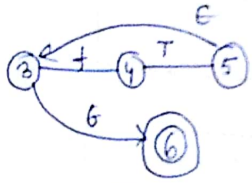
T' :



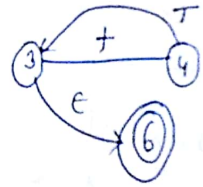
F :



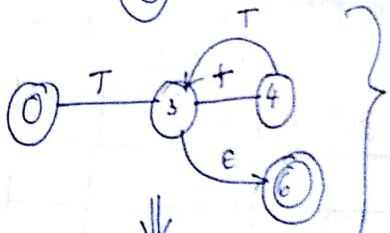
E' :



E' :

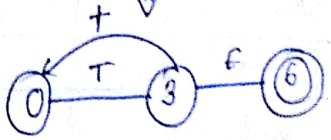


E :



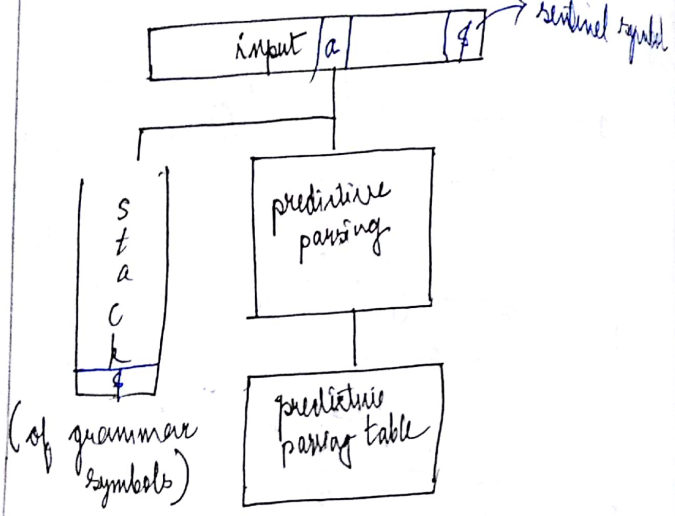
containing E, E' and T.

E :



A predictive parser can completely parse any binary expression.

Non recursive predictive parser



Entries of predictive parsing table

$M[A,a] = A \rightarrow \alpha$.

	a	b	c			
A		$A \rightarrow \alpha$				

check current input symbol and current top of stack.

If x is a terminal, and matches s.top(),

$x = a$, pop(x), advance i/p.

If x is a non terminal, A

look for entry in cell $[A,a]$

If $M[A,a] : A \rightarrow \alpha$

pop A, push α such that leftmost symbol α is at top.

A	$A \rightarrow \alpha$			$A \rightarrow \alpha_2$	

If entry corresponding $M[A,a]$ is empty then error condition. That particular symbol is not expected at that point.

Algorithm ends when entire input is exhausted i.e we are using a sentinel symbol '$'.

$ is also pushed as bottom of stack.

If input symbol = $ and s.top = '$' then stop.

- not left recursive - operators left associative

Parsing Table M :-

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

Example :- id + id * id $

$\$E \Rightarrow \$E'T \Rightarrow \$E'TF \Rightarrow \$E'Tid$

match (pop)

$\$E' \leftarrow \$E'\epsilon \leftarrow \$E'T'$

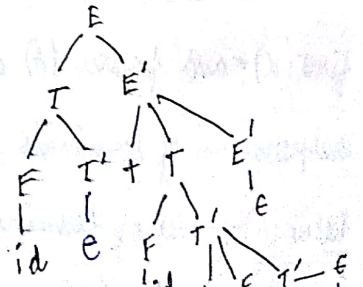
$\$E'T+ \xrightarrow{\text{match (pop)}} \$E'T \longrightarrow$

Parsing action :-

Stack	Input	Action
$E	id + id * id $	$E \to TE'$
$E'T	id + id * id $	$T \to FT'$
$E'T'F	id + id * id $	$F \to id$
$E'T'id	id + id * id $	match and pop.
$E'T'	+ id * id $	$T' \to \epsilon$
$E'	+ id * id $	$E' \to +TE'$
$E'T+	+ id * id $	match and pop.
$E'T	id * id $	$T \to FT'$
$E'T'F	id * id $	$F \to id$
$E'T'id	id * id $	match and pop.
$E'T'	* id $	$T' \to *FT'$
$E'T'F*	* id $	match and pop.
$E'T'F	id $	$F \to id$

production to be applied

Parse Tree



[No change in parse tree during match and pop]

- In CYK algorithm we cannot find the derivation sequence but in top down parser we get parse tree.
- On including another operator (eg- exponentiation) there will not be any change in method.
- C-language does not have exponentiation due to left right associativity. Also exponentiation is repeated multiplication by shift operation. User has to write his/her own exponentiation function.

Construction of parsing table:

first(α) and follow(A) are two functions outputting set of terminals.

- follow(A) → set of terminals which can appear immediately after A in any sentential form.

In follow(A) we will not include ϵ.

- first(α): consider any string derived from α ($\alpha \in (V_N \cup T)^*$).

$\alpha \xrightarrow{*} a\beta$. so $a \in$ first(α).

first(α) is the set of strings which appear as the leftmost symbol in any string derived from α.

[α can be null as well]

- Consider $A \rightarrow Y_1 Y_2 \dots Y_k$

so first(A) = first(Y_1) if first(Y_1) does not contain ϵ.

If first(Y_1) contains ϵ,

$A \rightarrow Y_2 Y_3 \dots Y_k$

so first(A) = first(Y_2).

- first(α) can be ϵ as well unlike follow(A) which does not contain ϵ.

first(X):

If X is a terminal, it is in first(X).

If $X \to \epsilon$ is in P, then add ϵ to first(X)

If $X \to Y_1 Y_2 .. Y_n$ and X is non-terminal, place a in first(X), if a is in first(Y_i) and ϵ is in first($Y_1 Y_2 .. Y_{i-1}$) each

If ϵ is in first(Y_i) for all $i = 1,2,..k$ then add ϵ to first(X).

$ is in follow(S).

If $A \to \alpha B \beta$ is in P, the symbols in first(β) except ϵ are in follow(B).

If $A \to \alpha B$ or $A \to \alpha B \beta$ with $\beta \Rightarrow \epsilon$ then every symbol in follow(A) is in follow(B)

$E \to TE'$

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

$T \to FT'$

$T' \to *FT' | \epsilon$

$F \to (E) | id$

first(E) = {(, id}	follow(E) = {), $}
first(T) = {(, id}	follow(T) = {+,), $, }
first(F) = {(, id}	follow(T') = {+,), $, }
first(E') = {+, ϵ}	follow(E') = {), $}
first(T') = {*, ϵ}	follow(F) = {*, +,), $}.

For a production, $A \to \alpha B \beta$, first(β) is in follow

$A \to \alpha B$ or $A \to \alpha B \beta$, ϵ is in first(β) then everything in follow(A) is in follow(B).

For $E \to TE'$, $E' \Rightarrow \epsilon$, so everything in follow(E) will be in follow(T').

construction of parsing table :-

For each production $A \to \alpha$ in G_1,

- for each terminal a in first(α)
  add $A \to \alpha$ to $M[A, a]$
- if ϵ is in first(α)
  add $A \to \alpha$ to $M[A, b]$ where b is in follow(A) for each b.
- If ϵ is in first(α) add $A \to \alpha$ to $M[A, \$]$ if $\$$ is in follow(A).

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

All the blank entries correspond to error conditions.

construct predictive parsing table and parse a given input. [lab assignment] → C++

(given in books for +, *) [+ - * / ^]

No unary minus

Consider integral division.

Use exponentiation as well.

the input, each operator should be used twice

[remove left recursion and left factor].

S → i E t S e S | i E t S E → b.

first(S) = {i} follow(S) = {$, e}

first(E) = {b} follow(E) = {t}

first(S') = {e, ϵ}

	i	t	e	b	$
S	S→iEtSeS S→iEtS				
E					

$S \rightarrow iEtSeS \mid iEtS \mid a \quad E \rightarrow b.$

$S \rightarrow iEtSX \mid a$

$X \rightarrow eS \mid \epsilon$

$E \rightarrow b.$

first(S) = {i, a} follow(S) = {e, $}

first(X) = {e, ε} follow(X) = {e, $}

first(E) = {b} follow(E) = {t}

	a	b	i	t	e	$
S	S→a		S→iEtSX			
X				X→eS X→ε		X→ε
E		E→b.				

We get multiple entries due to ambiguity.

G without multiple parsers in PPT is called LL(1).

On removing $X \rightarrow \epsilon$ from PPT, the grammar becomes LL(1).

Error recovery in Top/Down LL parsing -

Panic mode error correction. (Element at top of stack is not expected).

- identify → through tables
- recover
- proceed for finding further error

} parser.

→ in case of panic mode recovery ignore those symbols which were not expected.

Example. If T is in top and input symbol is id then ignore.

- For every non-terminal symbol, find a set of terminals which can be present (synchronizing set).

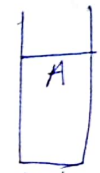
  $M[A, a] =$ error synch.

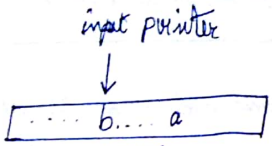
  If $M[A, a] = E \rightarrow TE'$ (valid)

- The symbols in follow(A) will be definitely in synchronizing set of A.

Illustration :

first(A) : { a .... }

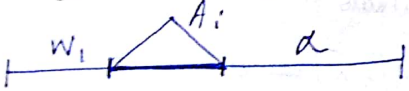




ignored by parser as long as a symbol belonging to first(A) is not obtained.

- Another way of recovery is to insert symbols in input but that leads to many other errors and confuses programmer. So many commercial compilers use panic mode recovery.

Bottom Up Parsing.



terminal string.

(1) Identify a substring of this terminal string which is right side of a production of a grammar rule.

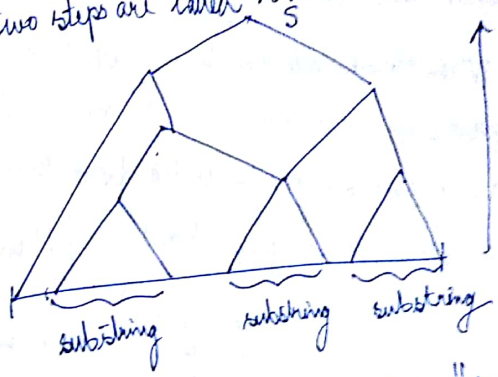
$a_1 a_2 \ldots a_i \boxed{a_{i+1} \ldots a_j} \ldots a_n$

↳ reduced to A_i

$a_1 a_2 \ldots a_i A_i \ldots a_n$

(2) Replace that substring with a variable in LHS of this production.

These two steps are called reduction.



- In any sentential form we can find more than one substring which matches RHS of a production.
- The derivation sequence should be rightmost derivation sequence in reverse order.

Example :

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

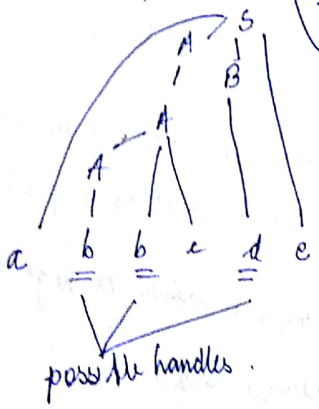
$B \rightarrow d$

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

- In Bottom-up parsing we preform rightmost derivation sequence.
- A handle is a substring in a given sentential form which matches RHS of a production rule.

When this handle can be reduced to a LHS of a production rule such that the new sentential form is previous right sentential form then that handle is called proper handle. [Handle pruning]

→ choosing proper handle is toughest job.

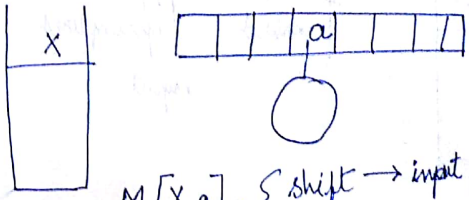


pass the handles.

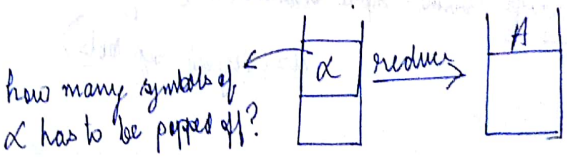
abbcde
aAbcde
aABe
S

Shift-Reduce Parsing - (bottom up parsing).

→ has a stack of grammar symbols.
(terminals & non-terminals).



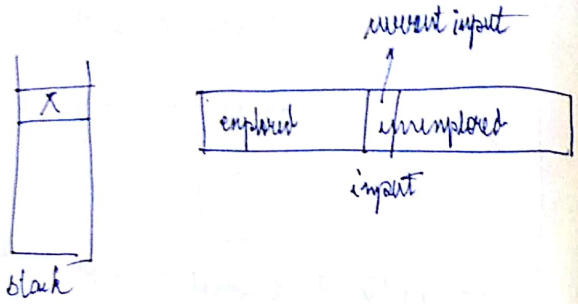
$M[X,a] = \begin{cases} \text{shift} \rightarrow \text{input symbol onto stack} \\ \text{Reduce } A \rightarrow \alpha. \end{cases}$



$$\frac{\beta \quad \alpha \quad w'}{\beta A \, w'} \quad \text{(unexplored input)}.$$

- In shift reduce parsing we assume that entire α is on top of stack.
- A viable prefix is that prefix which does not cross the stack.
- If $\beta A w'$ is previous rightmost sentential form then α is a viable prefix.

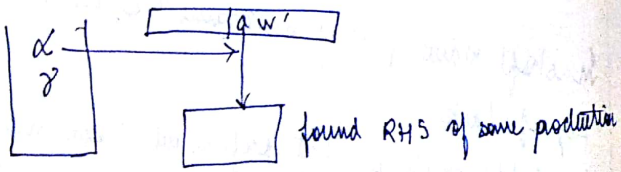
Operator Precedence Parsing - (makes sure handle is on top of stack).



shift current input onto stack (along with few user defined symbols).

$S \Rightarrow \ldots\ldots \rightarrow \gamma A a w' \rightarrow \gamma \alpha a w' \xrightarrow{*} w.$

γA : on stack; a : unexplored input; α : handle, $A \rightarrow \alpha$.



- If a reduce operation is performed on a right sentential form we must get previous right sentential form.

- **Viable prefix** - prefix of a right sentential form. Not every prefix of a right sentential form is viable. A viable prefix is one that appears entirely on stack.

A prefix of a right sentential form that does not move past the right end of rightmost handle.

- **Identifying sequence of viable prefixes** - (or identifying right end of handle).

Only after finding left end we can pop off α.

There is one language for identifying left end of handle. (limited language)

→ Operator Precedence grammar.
(parses expressions containing operators)

$E \rightarrow EAE \mid id$
$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$
} not OPG

Any grammar whose right hand side does not have two or more consecutive non-terminals.

$E \to E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid id \mid (E)$

Operator precedence relationships

$\theta_1 \cdot> \theta_2$

$\theta_1 <\cdot \theta_2$

$\theta_1 \doteq \theta_2$

If $\theta_1 > \theta_2$ and $\theta_2 > \theta_3$ it may happen that θ_1 and θ_3 are not related at all. Transitivity, reflexivity does not apply to operator precedence.

If $\theta_1 \cdot> \theta_2$ it may happen that $\theta_1 <\cdot \theta_2$

Example: $a - b + c$ $\quad a + b - c$

$-$: more precedence; $+$: higher precedence.

$\theta_1 \cdot> \theta_2$: θ_1 takes precedence over θ_2.

$\theta_1 <\cdot \theta_2$: θ_1 yields precedence over θ_2

$\theta_1 \doteq \theta_2$ θ_1 and θ_2 have same precedence.

equal. equal.

If θ_1 has higher precedence over θ_2 make
$\theta_1 \cdot> \theta_2$ and $\theta_2 <\cdot \theta_1$.

If θ_1 and θ_2 have equal precedence,

if they are left associative
$\theta_1 \cdot> \theta_2$ and $\theta_2 \cdot> \theta_1$

if they are right associative,
$\theta_1 <\cdot \theta_2$ and $\theta_2 <\cdot \theta_1$.

[between binary operators]

From operator precedence relationships we make operator precedence functions f and g using which we identify precedence relationships.

[Pg - 219, 220 Nisk]

(11)

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Operater precedence relationship.

θ <· id
id ·> θ
θ <· (
(<· θ
) ·> θ
θ ·>)
θ ·> $
$ <· θ

(·=)
(<· (
(<· id
$ <· (
id ·> $
id ·>)
$ <· id
) ·> $
) ·>)

Q. Create operater precedence table from operator precedence relationships.

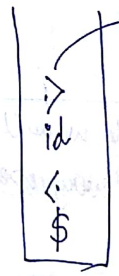
- Between every pair of non-terminal symbols give an operater symbol. relationship

Suppose string is "id + id * id $"
to indicate beginning of input.

$ <· id ·> + <· id ·> * <· id ·> $ "

- Note: Nowhere in the parsing we ~~will use~~ have used non-terminal symbol E till now.

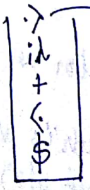
- Push all symbols onto stack including $



If anywhere on stack we get "takes" symbol we know that a handle is on top of stack. The string between <· ⟷ ·> is handle

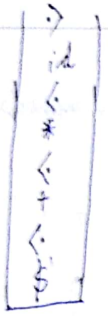
E
|
id + id * id

- [Between every pair of consecutive operators there is a non-terminal. Non-terminals are just placeholder]



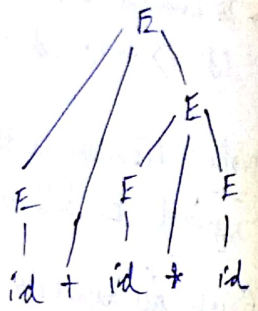
another handle is found.

E E
| |
id + id * id

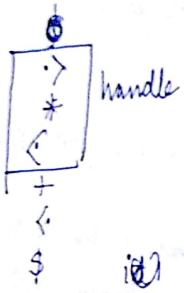


E E E
| | |
id + id * id

id + id + id → E + E * E



The stack contents are,



- Instead of operator precedence table we will have functions $f(a)$ and $g(a)$ for every operator a.

$f(a)$ $g(a)$.

used when operator a is on left side — used when operator a is on right side.

If $f(\theta_1) < g(\theta_2)$ then $\theta_1 < \theta_2$

- Using $f(a)$ and $g(a)$ for each operator we draw a directed graph. The length of longest path is value of $f(a)$ and $g(a)$.

- If there are n operators, we have 2n nodes. Some of these nodes will be grouped into one.
- if $a \cdot = b$, group f_a and g_b.
- if $c \cdot = b$, group f_c and g_b.

In same group we have here f_a, f_c and g_b.

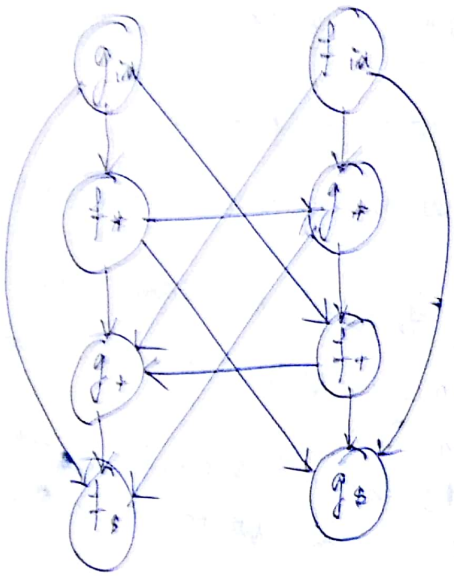
- Even if f_a and f_c are in same group then they might have no relationship between them.

$a \cdot = b$
$c \cdot = b$ } f_a and g_b are in same group but have no relationship.
$c \cdot = d$

- Number of nodes becomes $< 2n$.
- If there is a relationship $a \lessdot b$ (a yields to b), add an edge from g_b to f_a.

If $a \gtrdot b$, add edge from f_a to g_b.

Graph :-



- Define f_a = length of longest path from group containing f_a to any leaf node.

	id	+	*	$
f(a)	4	2	4	0
g(b)	5	1	3	0

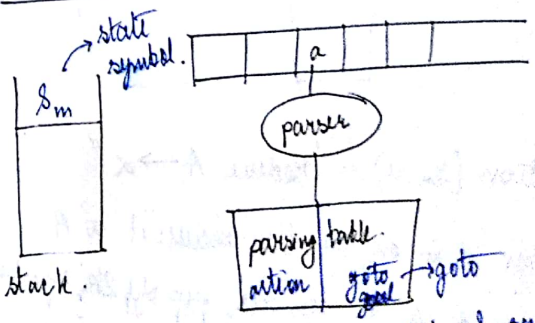
g_a :..... path from group containg g_a to any leaf node.

(iii) LR Parsing :- (bottom up parsing)
↳ rightmost derivation
left to right

- LR parsers are also table driven.
- shift-reduce is also done here.
- there are 3 different approaches for constructing LR parsing table-

(i) SLR
(ii) Canonical LR
(iii) LALR (yacc or bison is similar to LALR). uses LALR table.

Parsing action -

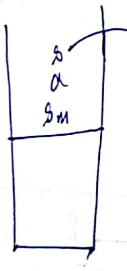


The stack contains grammar symbol and some state symbols. The t-o-s always contain state symbol. State symbol is representation of the configuration or the contents of stack.

The parsing table contains two parts, action and goal.

- action (s_m, a) = "shift s" → state symbol.
  - ↓ input
  - shift current symbol on top of stack
  - [a is pushed on stack]

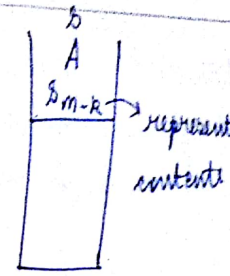
When a is pushed, the stack configuration changes. So, s is new state configuration which is pushed in stack.



→ new state symbol.

- action (s_m, a) = 'Reduce $A \rightarrow \alpha$"

When α is on tos, then reduce it to A. Suppose α has k symbols, pop off 2k symbols from stack and push A, [k symbols of α and k ~~stack~~ state symbols] and push state symbols.



→ representation of stack contents below it

$s = goto\ (s_{m-k}, A)$ → defined for state symbols and non-terminals.

- action (s_m, a) = "accept"

If none of the action can be done i.e. we have blank then,

- action (s_m, a) = "error".

Illustration :-

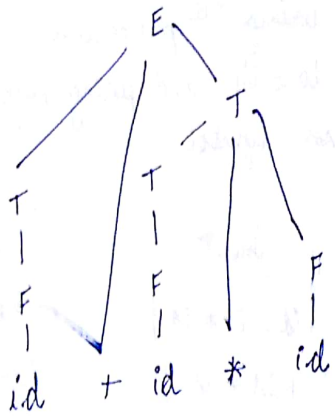
1. $E \rightarrow E+T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

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

r_i → reduce using i^{th} production

s_i → shift to state i and push corresponding column symbol.

Stack	Input	Action
0	id + id * id $	"shift 5"
0 id 5	+ id * id $	"reduce F → id"
0 F 3	+ id * id $	"reduce T → F"
0 T 2	+ id * id $	"reduce E → T"
0 E 1	+ id * id $	"shift 6"
0 E 1 + 6	id * id $	"shift 5"
0 E 1 + 6 id 5	* id $	"reduce 6" F → id
0 E 1 + 6 F 3	* id $	"reduce T → F"
0 E 1 + 6 T 9	* id $	"shift 7"
0 E 1 + 6 T 9 * 7	id $	"shift 5"
0 E 1 + 6 T 9 * 7 id 5	$	"reduce F → id"
0 E 1 + 6 T 9 * 7 F 10	$	"reduce T → T * F"
0 E 1 + 6 T 9	$	"reduce E → E + T"
0 E 1	$	accepted



Rightmost derivation :-

id + id * id
↑
F + id * id
↑
T + id * id
↑
E + id * id
↑
E + F * id
↑
E + T * id
↑
E + T * F
↑
E + T
↑
E

Construction of SLR Parsing Table :-

LR(0) Item

Item is a production by placing a '.' between, before or after any symbol.

Itemset
- $A \rightarrow .XYZ$
- $A \rightarrow X.YZ$
- $A \rightarrow XY.Z$
- $A \rightarrow XYZ.$

} a production with n symbols gives (n+1) items.

* If stack is
$s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m$

(i) and $M[s_m, a] =$ push s.
then stack becomes,
$s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m a s$.

(ii) and for $M[s_m, a] = pop(A \rightarrow \alpha)$ and $\alpha = \underbrace{X_{k+1} \ldots X_m}_{(m-k) \text{ symbols}}$.
stack becomes,
$s_0 X_1 s_1 X_2 s_2 \ldots s_k A s$
where $goto[s_k, A] = s$.

LR(0) Item :-

It is a production in G with . placed in any position on the right side.

$A \to XYZ$ yields

$A \to .XYZ$
$A \to X.YZ$
$A \to XY.Z$
$A \to XYZ.$

} LR(0) items depends number of symbols in production.

Sets of LR(0) items :- Each set of LR(0) item represents a state in SLR table.

The common property of items in a set comes from position of . in production

$A \to X.YZ$ means we have seen a string derivable from X and we are expecting a string derivable from YZ to reduce XYZ to A.

Therefore $A \to \alpha.\beta$ means string derivable from α is seen and expecting to see a string derivable from β so that $\alpha\beta$ can be reduced to A.

Closure of a set of items :-

Suppose itemset $I = \{A \to \alpha.B\}$,

closure (I) = { every element in I is in closure (I).
Suppose $A \to \alpha.B\beta$ is in I and $B \to \gamma$ is in P then $B \to .\gamma$ is in closure (I).
⇓ we are expecting string deriving from $B\beta$.

- If a substring is reduced to start symbol then we can't say that parsing is complete.
That's why we add a production (augmenting) an extra production $E' \to E$ to grammar. Such a grammar is augmented grammar. When E is reduced to E' we say parsing is complete.

- closure $(E' \to .E)$ = { $E' \to .E$,
(I_0) ↓ item
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .id$
$F \to .(E)$ }

Goto operation -

$g(I, x)$ $\quad I \rightarrow$ item, $x \rightarrow$ any grammar symbol.

If I contains $A \rightarrow \alpha . X \beta$ then $g(I, x)$ contains closure $(A \rightarrow \alpha X . \beta)$.

We can define for any symbol after '.'

$g(I_0, E) =$ closure $\{E' \rightarrow E.,\ E \rightarrow E.+T\}$

$\Downarrow$

we do not have a non-terminal after '.' so no more items can be added to g this set

$g(I_0, ( ) = \{ F \rightarrow (.E) \quad F \rightarrow .(E)$
$E \rightarrow .E+T \quad F \rightarrow .id \}$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$

Canonical collection of LR(0) item set -

We start from I_0 (closure of augmented grammar)
We define all possible goto from I_0.

$I_1 = g(I_0, E) = \{ E' \rightarrow E.$
$E \rightarrow E.+T \}$

$I_2 = g(I_0, T) = \{ E \rightarrow T.$
$T \rightarrow T.*F \}$

$I_3 = g(I_0, F) = \{ T \rightarrow F. \}$ [I_0 goto can be defined on I_3].

$I_4 = g(I_0, ( ) = \{ F \rightarrow (.E)$
$E \rightarrow .E+T$
$E \rightarrow .T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id \}$

$I_5 = g(I_0, id) = F \rightarrow id.$

$I_6 = g(I_1, +) = \{ E \rightarrow E+.T$
$T \rightarrow .T*F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id \}$

$I_7 = g(I_2, *) = \{ T \rightarrow T*.F$
$F \rightarrow .(E)$
$F \rightarrow .id \}$.