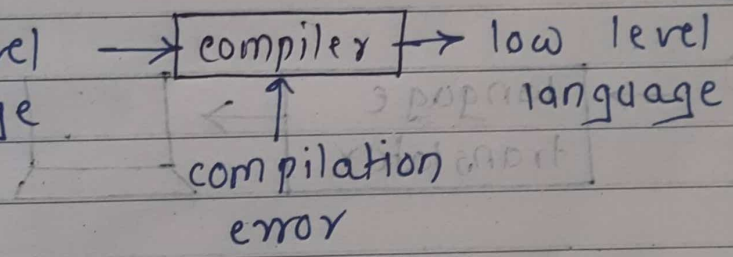
PAGE No.
DATE / /

Unit 1:

Compilers : Introduction to compiler phases, introduction to cross compiler, features of m/c dependent & independent compilers, types of compilers, Interpreters: compilers vs interpreter, phases, working, Preprocessor: header file and macro expansion.

Compiler :

A compiler is a slw that converts a program written in a high level language (source lang.) to a low level language (object/target m/c lang. 1's, 0's)

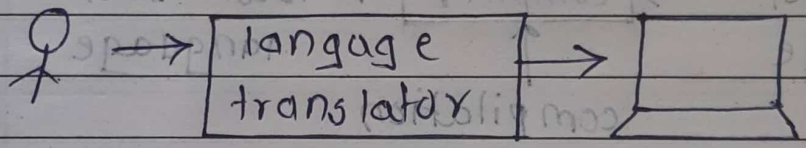


It performs following operations:

1. Performs a pre-processing of source code. Gather all files required for the source code to compile.
2. Parses the entire source code. checks for any syntax errors in the source code.
3. Performs a through syntax analysis of the source code. To understand the structure & semantic of the source code.
4. Optionally translates the source code in an intermediate code known as object code to enhance the performance.
5. Translates the object code to binary language known as executable code.

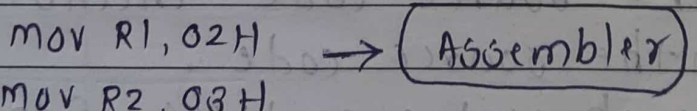
Introduction

- computers are electronic devices so it understand the language of 0's & 1's only.
- In early days of computers, instruction to a computer is given with the help of paper tape & punched cards, so in these the presentation of hole represents 0 else it is 1.
- so writing of 0's & 1's is difficult
- so with punched cards we are facing a communication barrier to make the communication with the computers, so will require a language translator.



Language Translators

1. Assembler



Assembly lang. — m/c code

so to write the code we require the complete knowledge of computer architecture so its difficult to remember this. so the Interpreter & compiler came into picture.

PAGE No.
DATE / /

Interpreter eg.-
php, ruby, Javascript

compiler eg.- c, c++, erlang.

- compiler is more faster & efficient than interpreter

High level language

```
#include <stdio.h>
int main()
{
    int x, a=2, b=3, c=5;
    x = a+b*c
    printf("The value of x is %d", x);
    return 0;
}
```

source code/HLL code

⇩

language Translator

⇩

```
00110110111
01101101011
01011011100
```

mlc code.

Language Translator - Internal architecture

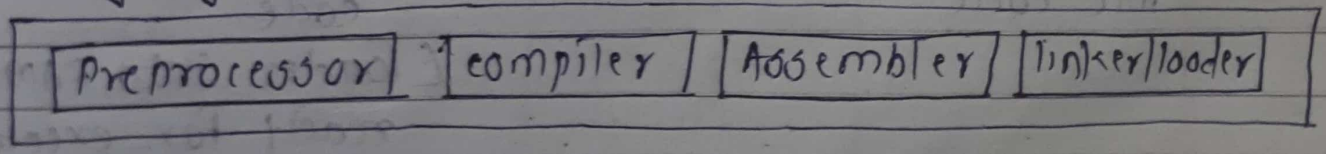


Fig: clang. translator Arch.

PAGE No.
DATE / /

Preprocessor -
Reserve all preprocessor dir & remove all comments

```
#include<stdio.h>                    stdio.h
// code      ⇒ Preprocessor ⇒      // code
  HLL code                         pure HLL
```

Compiler -
Generate equivalent assembly lang. code

```
stdio.h                            mov a, 02h
// code      ⇒ compiler ⇒          mov B, 04h
pure HLL                           assembly code
```

Assembler -
Generate relocatable mlc code.

```
assembly      ⇒ Assembler ⇒  i+0 : 0101110
code                         i+1 : 001101
                             relocatable
                             mlc code.
```

Linker/ loader -

```
i+0 : 0100110
i+1 : 0110110   ⇒ linker/  ⇒
relocatable       loader     absolute mlc
mlc code                     code
                             [loader into RAM &
                             ready for execution]
```

PAGE No.
DATE / /

compiler

It has 6 different phases these are,

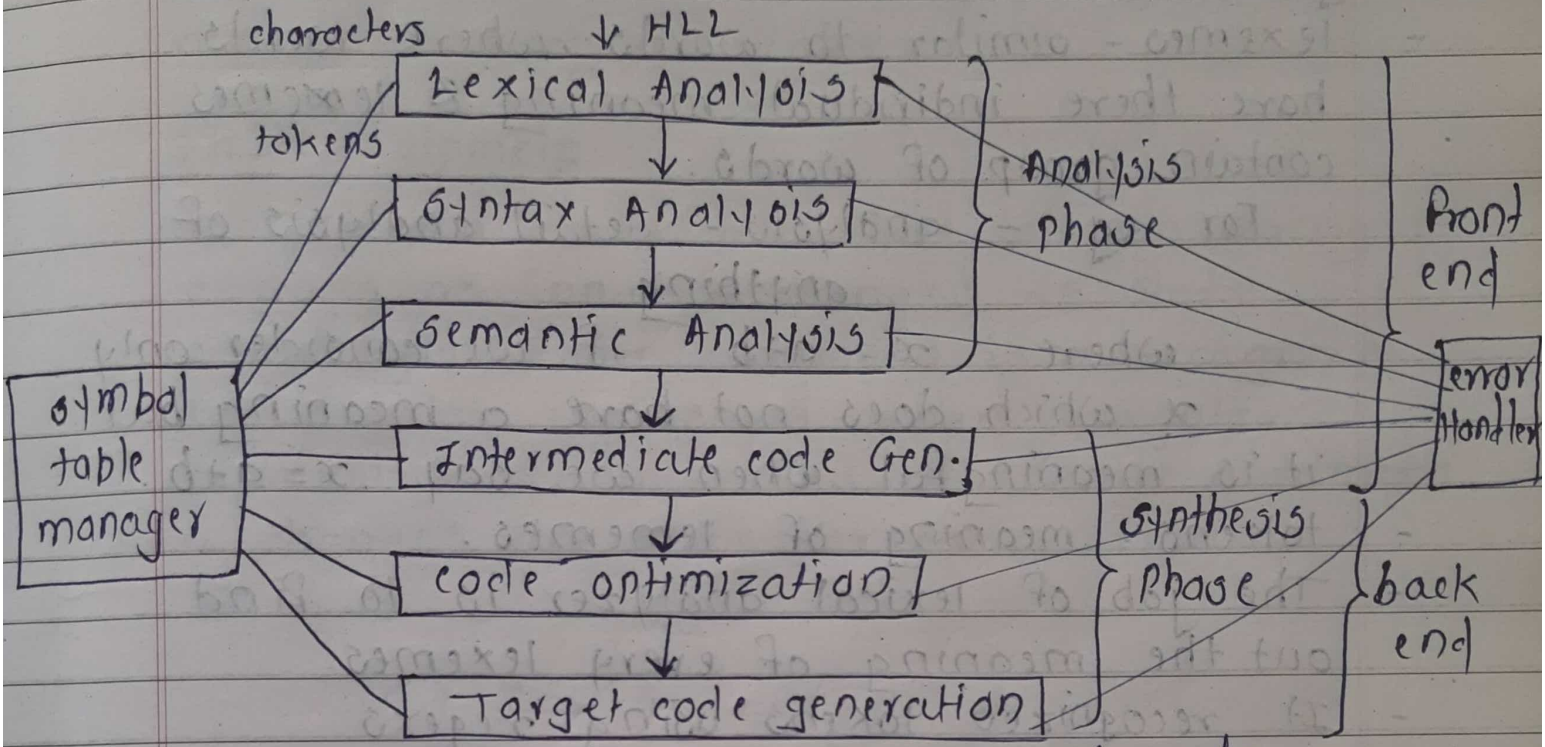


Fig: compiler Design

1) Lexical Analyzer :

$x = a + b * c$

⇓

Lexical Analysis

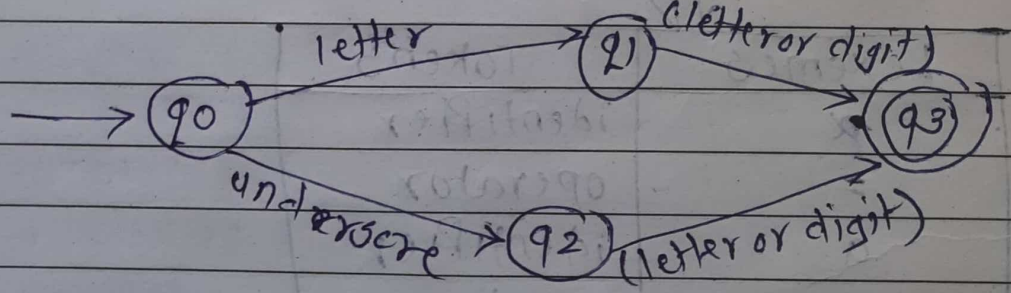
⇓

Lexemes	Tokens
x	identifier
=	operator
a	identifier
+	operator
b	identifier
*	operator
c	identifier

- Lexical Analyzer takes lexemes as input & generate tokens
- lexemes - similar to words, where words have there individual meaning & lexemes contain group of words.
  For eg. - analysis - detail analysis of anything
  where $x = a+b$ if we consider only x which does not have a meaning but it is meaningful when we say $x = a+b$.
- tokens: meaning of lememes.
- The job of lexical analyzer is to find out the meaning of every lexemes.
- It recognizes tokens using regexs
  Eg. - Regex for identifier

Regular expression → $l(l+d)^* | \_ (l+d)^*$ where,
l - letter
d - digit
_ - underscore.

illustrate this using finite autometa.



- So for examing the lexemes the lexical analyzer uses the regular grammers

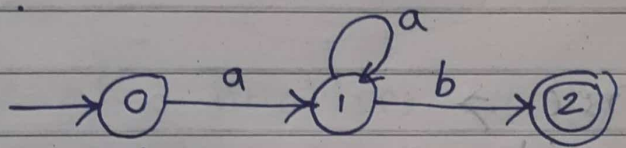
Regular expression :

it is composed of simple characters such as /abc/ or combination of simple & special characters such as /ab*c/

Finite autometa :

It is a state mlc that takes a string of symbols as input & changes its state accordingly. Finite autometa is a recognizer for regular expression. When a regular expression string is fed into finite autometa, it changes its state for each literal.

Eg :



Initial state — Final state

I/P / states	a	b
0	1	–
1	1	2
2	–	–

2) Syntax Analyzer (Parser) :

Stream of tokens passed to syntax analysis phase. Syntax analysis depends on types grammer or context free grammer

For eg. - $x = a + b * c$

⇓

$S \rightarrow id = E;$

$E \rightarrow E + T | T$

$T \rightarrow T * F | F$

$F \rightarrow id$

these are CFG rule to form a parse tree

where,
E ⇒ exp.
T ⇒ Term
F ⇒ Factor

Parse Tree

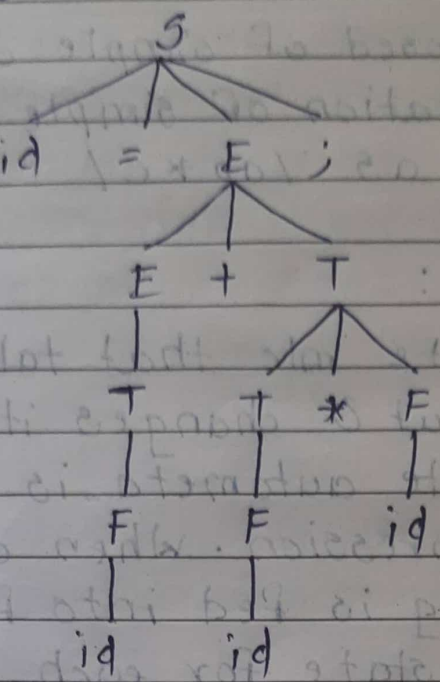


Fig: Parse Tree

yield of parse Tree ⇒
id = id + id * id ;

if yield of parse & expression is same then there is no error.

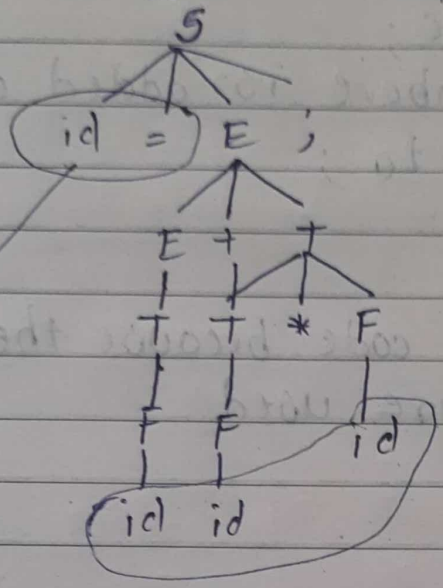
Context Free Grammer (CFG):

is a set of recursive rules used to generate pattern of strings. A CFG can describe all regular languages.

3) Semantic Analyzer:

parse Tree ⇒ Semantic Analyzer ⇒ semantically verified parse Tree

- Semantic Analyzer is responsible for type checking, error bound checking & correctness of scope resolution
- It does the logical analysis of parse tree



this is not constant beoz it is followed by = symbol

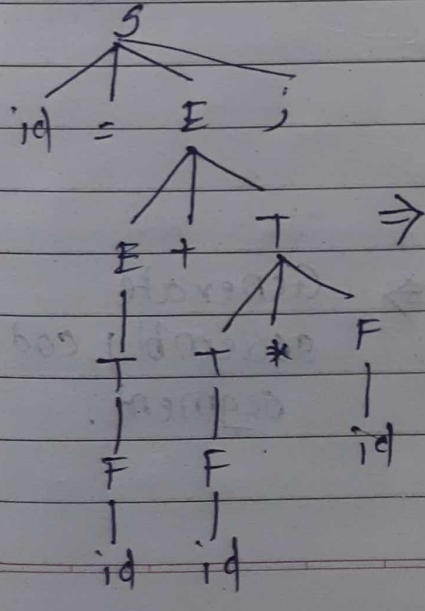
identifies its a variable

- this is constant

- Semantic analyzer looks for the meaningfulness of parse tree

4) Intermediate code Generator

Semantically verified parse tree ⇒ Intermediate code generation ⇒ Intermediate code



⇒

id = id + id * id;

↓

x = a + b * c;

Three Address code (TAC)

1. $t_0 = b * c$;
2. $t_1 = a + t_0$
3. $x = t_1$;

- TAC - popular intermediate code
- if we look the parse tree from bottom to up, so at lowest level * operator is there, so * performs first

$\therefore t_0 = b * c;$

the result of above is added with a so,

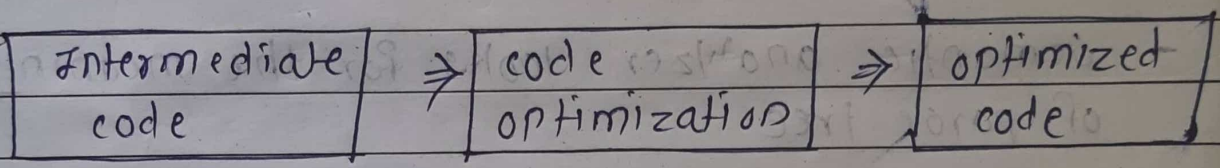
$t_1 = a + t_0;$

so at last,

$x = t_1;$

its three address code because there is max. three addresses are used

5) Code optimizer :



- code optimization can be mlc dependent or mlc independent.

TAC

1. $t_0 = b + c;$
2. $t_1 = a + t_0;$
3. $x = t_1;$

1. $t_0 = b + c;$
2. $x = a + t_0;$

6) Target code Generation :

optimized code ⇒ Target code generation ⇒ Generate assembly code segment.

$t_0 = b + c$

$x = a + t_0$

PAGE No.
DATE / /

```
mov eax , DWORDPTR [rbp-8]
```

- eax — extended version of ax; accumulator variable (32 bits stored)
- DWORD PTR — data word pointer
- [rbp-8] — scaling factor 8, pointing to b → value of b move to eax reg

signed multiplication

```
imul  eax , DWORT PTR[rbp -12]
mov   edx , eax
mov   eax , DWORD PTR [rbp-4]
add   eax , edx
mov   DWORD PTR [rbp-16], eax
```

- imul — value pointing to variable c; eax * c = eax
- mov edx, eax — result = b*c in accumulator
- mov eax, [rbp-4] ⇒ a in accumulator
- add — eax & edx
- [rbp-16] — add^r of x.

7) Symbol Table:

- Symbol table is basically a data structure which stores the occurances of entities
  - variable & function names
  - objects
  - classes
  - Interfaces

Symbol Table : Usage by phases

Lexical analysis -
creates entries for identifies.

syntax analysis -
Adds information regarding attributes

semantic analysis -
using the available info. stored in symbol table checks the semantics of identifier created by lexical analysis phase & update the symbol table.

PAGE No.
DATE / /

Intermediate code generation -
available info. in symbol table helps in adding temporary variable info.

code optimization -
available info. in symbol table is used in mlc dependent optimization

Target code generation -
Generates the target code using address info. of identifiers.

Symbol table: entries

Name	Type	size	dimension	line of declaration	line of usage	Address
count	int	2	0	-	-	-
x	char	3	1	-	-	-

line no. | line no. in symtist if identifier used multiple times

```
int count;
char x[] = "xyz";
```

- for premitive data type the dimension is 0 & for 1D, 2D its 1 or 2 respectively.

Symbol table: operations

- operations perform on symbol table depends on the language
- Early days of langs we used
  - Non-block structured language
    - contains single instance of the variable declarations
- so on symbol table we will perform

i) insert() ii) lookup() - access variable.
Eg. For non structured block lang. is FORTRAH lang.

- Block structured lang:
  - variable declaration may happen multiple times
  operations
    i) insert() ii) lookup()
    iii) set() iv) Reset()

* Features of machine dependent & independent compilers

- A compiler operates in phases. A phase is a logically interrelated operation that takes source pgm in one representation & produces output in another representation.
- There are two phases of compilation
  1) Analysis (mlc independent / lang. dependent)
  2) Synthesis (mlc dependent / lang. independent)

Machine dependent

- The allocation of the registers as well as their machine instructions rearrangement for the execution of the efficiency improvement process.
- It is the form which is considered for the code optimization technique of the pgm which has to be compiled.

PAGE No.
DATE / /

Machine Independent

- attempts to improve the intermediate code to get a better target code.
- The process of intermediate code generation introduces much inefficiency like: using variable instead of constants, extra copies of variable, repeated evaluation of expression. Through code optimization you can remove such efficiencies & improves code.

code optimization can perform in the following different ways:

1) compile time evaluation -

```
x = 5.7
y = x/3.6
```

evaluate x/3.6 as 5.7/3.6 at compile time

2) variable propogation -

Before optimization

```
c = a*b
x = a
till
d = x*b + 4
```

After optimization

```
c = a*b
x = a
till
d = a*b + 4
```

Here, after variable propogation a*b & x*b identified as common sub expression

3) Dead code elimination -

Before elimination	after elimination
c = a*b	c = a*b
x = b	Hii
Hii	d = a*b+4
d = a*b+4	

Here, x=b is a dead state bcoz it will never subsequently used in the pgm. so, we can eliminate this state.

4) code motion -
- It reduces the evaluation freq. of exp.r
- It brings loop invariant stmts out of the loop.

```
do                          item=10;
{                           do
  item=10                   {
  value2 = value + item     value2 = value+item;
} while (value<100);        } while(value<100);
```

5) Induction variable & strength Reduction -
- strength reduction is used to replace the high strength operator by the low strength.
- An induction variable is used in loop for the following kind of assignment like i=i+ constant.

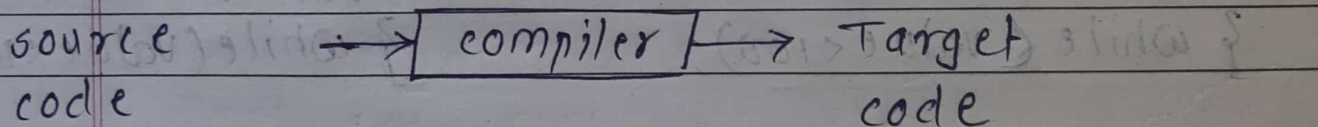
Before reduction	after reduction
i=1;	i=1
while(i<10)	t=4
{	{
y=i*4;	while(t<40)
}	y=t;
	t=t+4;
	}

* Types of compiler

1) Single Pass compiler
2) Two Pass compiler
3) Multipass compiler

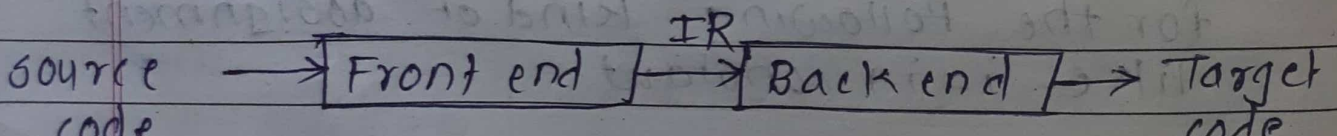
1) Single Pass compiler ⇒

When we merge all the phases of compiler design in a single module, then it is called a single pass compiler. In a case of single pass compiler, the source code converts into mlc code.



2) Two pass compiler ⇒

A processor that runs through the pgm to be translated twice is considered two-pass compiler

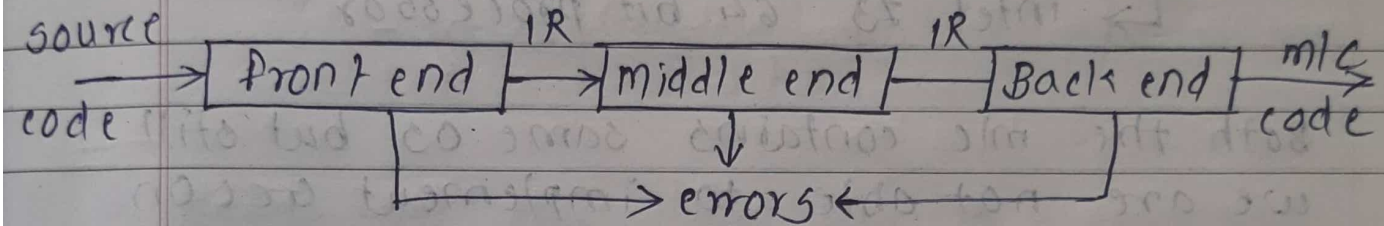


Incremental — execute the recompilation of only a changed source instead of compiling the complete code.

PAGE NO.
DATE / /

3) Multipass compiler ⇒

A pgm source code or syntax tree is processed many times by the multipass compiler. It breaks down a huge pgm into numerous smaller pgms & runs them all at the same time. It creates no. of intermediate codes. All of these multipasses use the previous phases output as an input. As a result, it necessitates less memory.
'wide compiler' is another name for it.

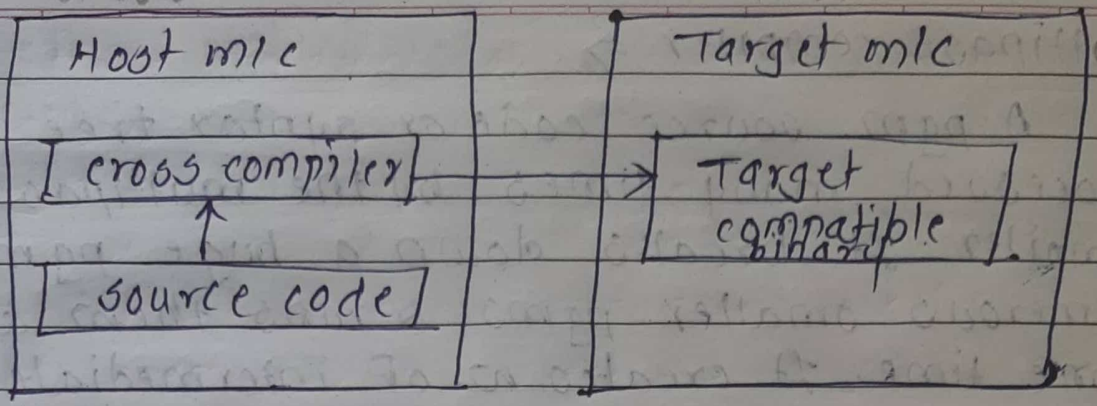


* Cross Compiler

- A cross compiler is a type of compiler that generates m/c code targeted to run on a system different than one generating it
For eg.- A compiler that runs on windows platform also generates a code that runs on linux platform is a cross compiler.
- The process of creating executable code in different machines is called "retargeting".
- The cross compiler is also known as retargetable compiler.
- Eg. of cross compiler ⇒ GHU GCC

windows

linux



Gcc → old mlc with 8 bit processor

→ intel i3 64 bit processor

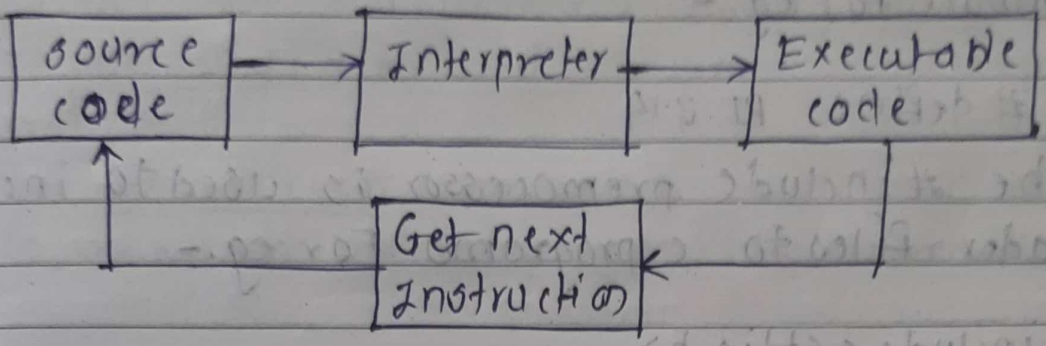
Both the mlc contains same os but still we are not able to implement Gcc on second mlc,
so, Gcc will get upgrade to implement on intel i3, so this is known as cross compiler.

*. Interpreter

- The translation of a single statement of the source program into machine code is done by a language processor & executes immediately before moving on to the next line is called an interpreter.
- If there is an error in the statement, the interpreter terminates its translating process at that stmt & displays an error message.
- The interpreter moves on to the next line for

execution only after the removal of the error.

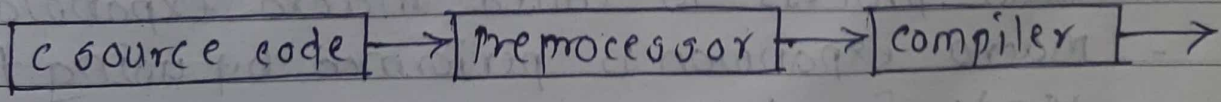
- An interpreter directly executes instructions instructions written in a programming or scripting language without previously converting them to an object code or m/c code.
eg. - perl, python, matlab



* Interpreter Phases & working

- Interpreter works like the front end of a compiler. That is, an interpreter does lexing, parsing & semantic analysis.
- The process of interpretation can be carried out in following phases
  - lexical analysis
  - syntax analysis
  - semantic analysis
  - Direct execution

* Preprocessor : header file & macro expansion



working of c preprocessor

- The c preprocessor is a macro preprocessor (allows you to define macros) that transforms pgm before it is compiled.
- These transformation can be the inclusion of header files, macro expansion etc.
- All preprocessing directives begin with a # symbol. for eg.-

```
#define PI 3.14
```

- The #include preprocessor is used to include header files to c programs. For eg.-

```
#include <stdio.h>
```

- Here, stdio.h is a header file. The #include preprocessor directive replaces the above line with the contents of stdio.h header file.
- Macro provide the feature for defining new operations & data in programming lang.
- A macro definition in a pgm defines either new operation or new method of declaring data.
- A macro call in the pgm is an innovation of the new operation or the new method of declaring data defined in the macro.
- It leads to a pgm generation activity during which the macro call is replaced by a sequence of stmts in the programming lang. this process is called macro expansion.
- Macro expansion is performed by using two kinds of actions -
  1) lexical substitution
  2) Semantic expansion

- A macro assembler performs expansion of each macro call into a sequence of assembly stmts & also assembles the resulting assembly pgm.
- A macro preprocessor merely performs expansion of macro calls & produces an assembly program.

Macros are useful for following purposes
1) To simplify & reduce the amount of repetitive coding
2) To reduce errors caused by repetitive coding
3) To make an assembly pgm more readable.

A macro consist of name, set of formal parameters & body of code. The use of macro name with set of actual parameters is replaced by some code generated by its body. This is called macro expansion.

Eg. -
```
#define max(a,b) a>b ? a:b
main(){
 int x,y;
 x=4, y=6
 z=max(x,y);}
```
After preprocessing becomes `z=x>y ? x:y;`

After macro expansion, the whole code would look like:
```
#define max(a,b) a>b ? a:b
main()
{
 int x,y;
```

PAGE No.
DATE / /

x=4; 4=6; z = x>4 ? x:y ;}

* Two pass macro processor

General Design steps

1. Specification of problem
2. Specification of databases
3. Specification of database formats
4. Algorithm

There are four basic tasks that a macro processor must perform

1) Recognize macro definition -
A macro inst. processor must recognize macro def. identified by MACRO & MEND pseudo operations.

2) Save Definition -
The processor must store macro definition which will be needed at the time of expansion of calls.

3) Recognize macro call -
The processor must recognize macro calls that appear as operation mnemonics.

4) Expand calls & substitute arguments -
The macro call is replaced by the macro definition. The dummy arguments are replaced by the actual data.

PAGE No.
DATE / /

Specification of databases

Pass 1 data bases -

- The input macro source pgm.
- The output macro source pgm to be used by Pass 2.
- Macro definition Table (MDT), to store the body of macro
- Macro definition table counter (MDTC), to mark next available entry MDT.
- Macro Name Table (MNT), used to store names of macros.
- Macro Name Table counter, used to indicate the next available entry in MNT.
- Argument List Array (ALA), used to substitute index,

Pass 2 Data bases -

- The copy of the input from Pass1 (intermediate file)
- The output expanded source to be given to assembler (Target file)
- MDT, created by pass 1
- MNT, created by pass 1
- Macro definition table pointer (MDTP), used to indicate the next line of text to be used during macro expansion.
- Argument list array (ALA), used to substitute macro call arguments for the index markers in the stored macro.

PAGE No.
DATE / /

Pass 1 : Algorithm

1. Pass 1 of macro processor makes a line by line scan over its input.
2. Set MDTC = 1 as well as MNTC = 1
3. Read next line from input program
4. If it is a MACRO pseudo-op the entire macro definition except this (MACRO) line is stored in MDT.
5. The name is entered into Macro Name Table along with a pointer to the first location of MDT entry of the definition.
6. When the END pseudo-op is encountered all the macro have been processed, so control is transferred to pass 2.

Pass 2 : Algorithm

1. This algo. reads one line of i/p pgm at a tim
2. For each line it checks if op-code of that line matches any of the MNT entry.
3. When match is found (ie when call is pointer called MDTP to corresponding macro stored in M
4. The initial value of MDTP is obtained from m index field of MNT entry.
5. The macro expander prepares the ALA consisting of a table of dummy argument indice & corresponding arguments to the call.
6. Reading proceeds from the MDT, as each successive line is read, the values from the argument list are submitted for the dummy arguments. indices in the macro.
7. Reading MEND line in MDT terminates expans

of macro & scanning continues from i/p file.

8. When END pseudo-op encountered, the expanded source pgm is given to the assembler.

Databases:

Macro Name Table (MNT)		
Index	MacroName	MDT index

Macro defn Table (MDT)	
Index	card

Argument list array (ALA)	
Index	Argument

card - means inst (body of macro)

↳ positional or actual argument

Example (2 pass macro processor)

Source pgm	Intermediate code
MACRO	
&LAB INCR &ARG1, &ARG2, &ARG3	
&LAB ADD AREG, &ARG1	
ADD AREG, &ARG2	
ADD AREG, &ARG3	
MEND	
START	START.
LOOP INCR A, B, C	LOOP INCR A, B, C
LABEL INCR DATA1, DATA2, DATA3	LABEL INCR DATA1 DATA2 DATA3
A DC 2	A DC 2
B DC 2	B DC 2
C DS 2	C DS 2
DATA1 DC3	DATA1 DC3
DATA2 DS2	DATA2 DS2
DATA3 DC4	DATA3 DC4
END	END

MDT (Macro Def Table)

Index	card
01	&LAB INCR &ARG1,&ARG2,&ARG3
02	#1 ADD AREG #2
03	ADD AREG #3
04	ADD AREG #4
05	MEND

MDTC → 01

#1, #2 ... ⇒ defines index pos. from ALA

MNT (Macro Name Table)

Index	Macro Name	MDT index
01	INCR	01

MDTC →

ALA (Argument list array)

Index	Argument
01	&LAB
02	&ARG1
03	&ARG2
04	&ARG3

So, the output of pass 1 of 2 pass Macro processor is :
Intermediate code, MNT, MDT, ALA Table.

PAGE No.
DATE / /

Pass2 of 2Pass Macro Processor [Target code]

Intermediate code

```
START
LOOP INCR A,B,C
LABEL INCR DATA1,DATA2,DATA3
A DC 2
B DC 2
C DC 2
DATA1 DC 3
DATA2 DS 2
DATA3 DC 4
END
```

Expanded code

```
START
LOOP ADD AREG,A
     ADD AREG,B
     ADD AREG,C
LABEL ADD AREG,DATA1
      ADD AREG,DATA2
      ADD AREG,DATA3
A DC 2
B DC 2
C DC 2
DATA1 DC 3
DATA2 DS 2
DATA3 DC 4
END
```

Prepare new ALA

new ALA

ALA	
Index	Argument
01	LOOP
02	A
03	B
04	C

ALA	
Index	Argu.
01	LABEL
02	DATA
03	DATA2
04	DATA3

- LOOP INCR A,B,C → macro call.
- Here macro is call, so we can prepare a new ALA with actual arguments
- LOOP ADD AREG,A

PAGE No.
DATE / /

* Assembler

- Assembler is a pgm for converting insts written in low level assembly code into relocatable mic code & generating along information for the loader.

Assembly code → [Assembler] → mic code

- It generates instructions by evaluating the mnemonics (symbols) in operation field & find the value of symbol and literals to produce mic code.
- We can create an assembly lang. code using a compiler or a programmer can write it directly.
- Mostly programmers use high level langs but, when more specific code is required, assembly lang. is used.
- It uses opcode for the instructions.
- An opcode basically gives info. about the perticular instruction.
- The symbolic representation of opcode (mic level inst.") is called mnemonics.

For example ADD A, B

Here, ADD is the mnemonic that tells the processor that it has to perform addition of function. A & B are the operands.

PAGE No.
DATE / /

Types of Assembler

Assemblers generate instruction, on the basis of a no. of phases used to convert to mlc code, assemblers have two types:

1) one-pass Assembler :-
These assemblers perform the whole conversion of assembly code to mlc code in one go.

2) multi pass / Two-pass Assembler
These assemblers first process the assembly code & store values in the opcode table & symbol table. And then in the second step, they generate the mlc code using these tables.

a) Pass 1 -
- symbol table & opcode tables are defined.
- Keep the record of the location counter
- Also, processes the pseudo instructions.

b) Pass 2 -
- Finally, converts the opcode into the corresponding numeric opcode.
- Generates mlc code according to values of literals & symbols.

PAGE No.
DATE / /

Opcode table -

They store the value of mnemonics & their corresponding numeric values.

Symbol table -

They store the value of programming lang. symbols used by the programmer, & their corresponding numeric values.

Location counter -

It stores the address of the location where the current inst. will be stored.

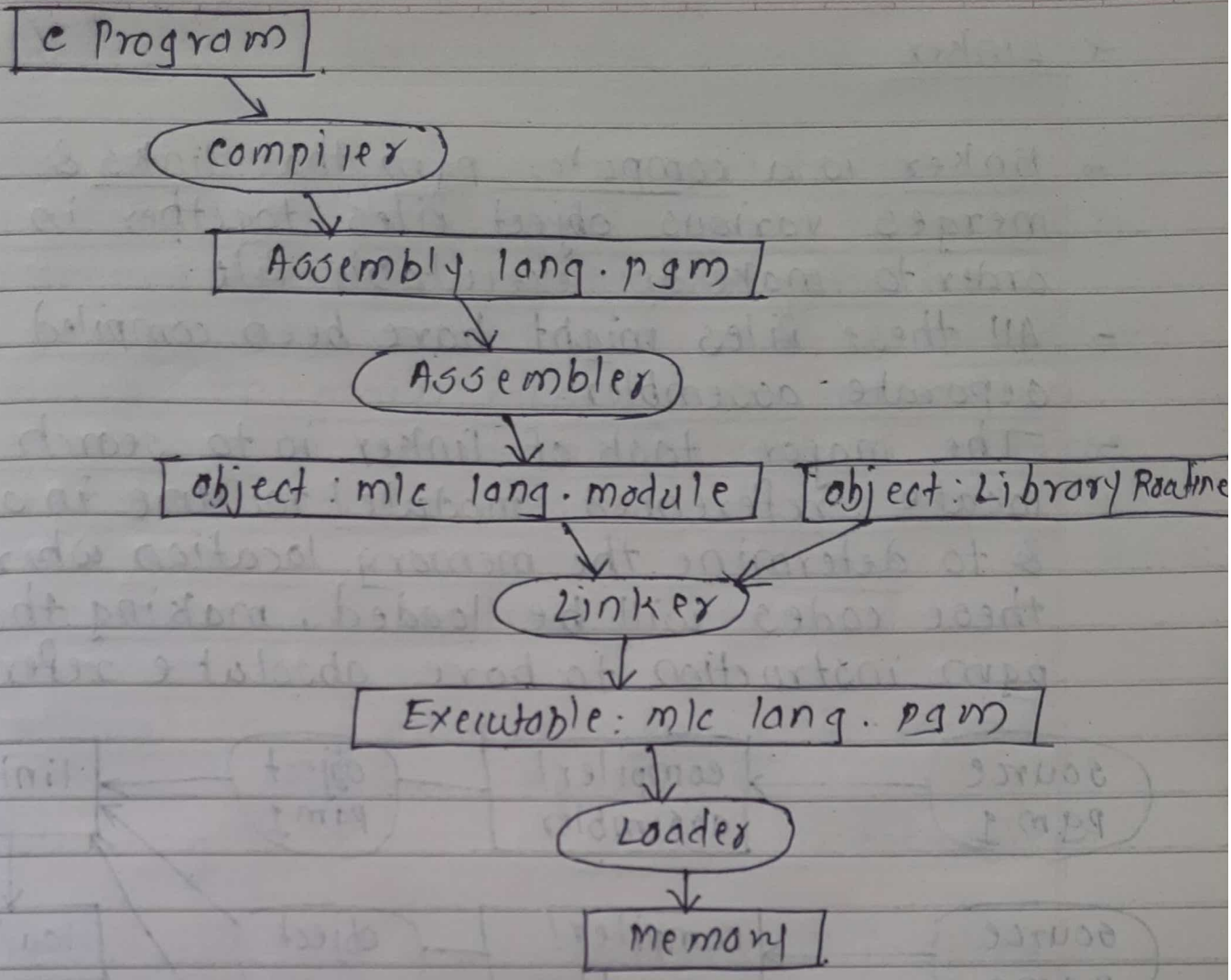
* Linkers & loaders

Translation Hierarchy

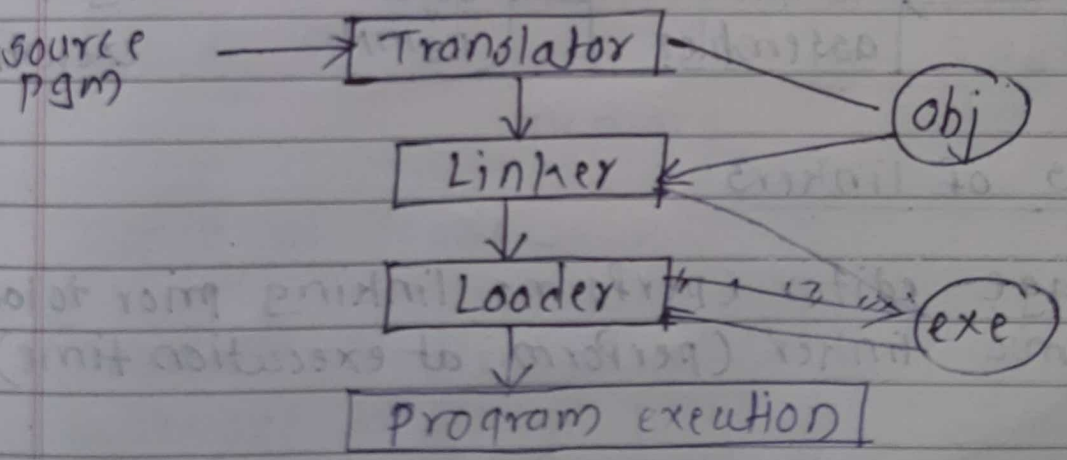
1) compiler -
Translates high level language pgm into assembly language.

2) Assembler -
converts assembly language pgm into object files.

- object file contains a combination of mlc instructions, data & information needed to place instructions properly in memory.



Flow of Execution

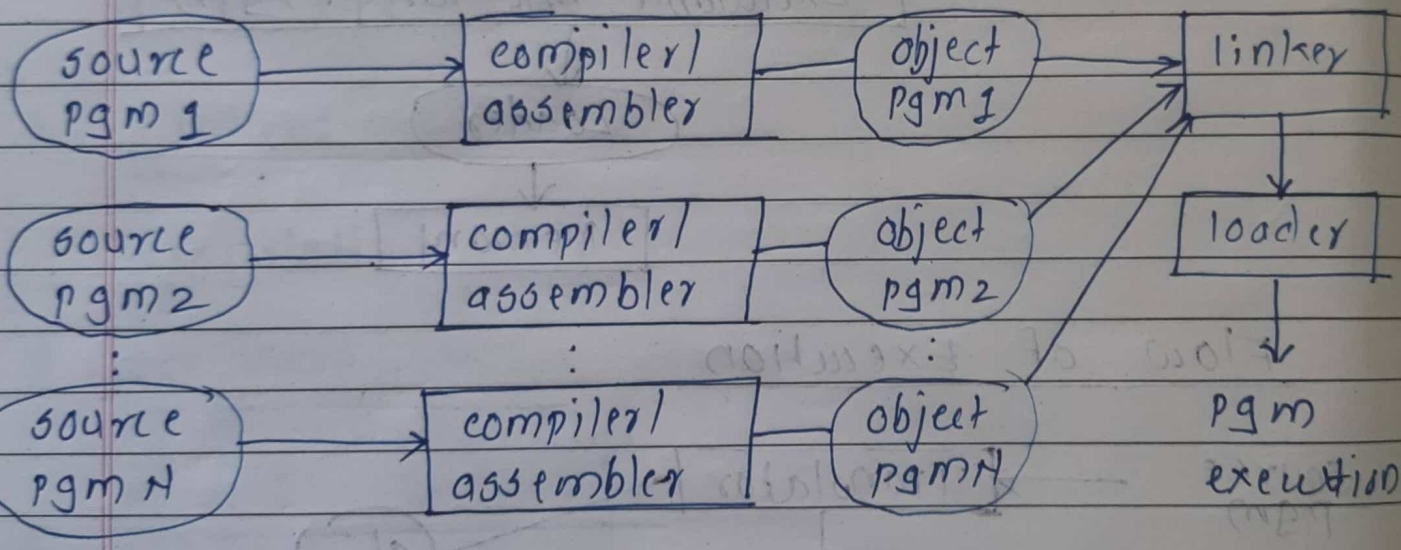


printf is a fn. that is not in your code but in an object library. printf calls some subroutine to all your messages. The linker extract object module from libraries & add them to your executable.

* Linker

- linker is a computer pgm that links & merges various object files together in order to make an executable file.
- All these files might have been compiled by separate assembler.
- The major task of linker is to search & locate referenced module/routine in a pgm & to determine the memory location where these codes will be loaded, making the pgm instruction to have absolute references.

eg. printf



Types of linkers

1) Linkage editor (performs linking prior to load ti
2) Dynamic linker (perform at execution time)

linking is a process of collecting & maintainin piece of code & data into a single file.

PAGE No.
DATE / /

* Loader

- loader is a pgm which accepts the i/p as linked modules & loads them into main memory for execution by the computer.
- Loaders load or copies pgm from secondary storage to main memory for execution.
- Loaders can also replace virtual address with physical addresses.

Functions of loaders

Allocation - allocate space in memory for the pgms.

Linking - symbol resolution bet. object modules.

Relocation - Adjust address dependent locations of address constants i.e assign load addresses to different parts of a pgm.

loading - store mlc inst. & data into memory.

Types of loaders

1) Absolute loader
2) Bootstrap loader
3) Relocating loader (Relative loader)
4) Linking loader

PAGE No.
DATE / /

Absolute loader -

It loads the object code to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program.

Bootstrap loader -

When a computer is first turned on or restarted a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first pgm to be run by the computer - usually an operating system. The bootstrap itself begins at address 0.

Relocating loader -

- loaders that allow for pgm relocation are called relocating loaders or relative loaders.
- The execution of the object pgm is done using any part of the available & sufficient memory.
- The object pgm is loaded into memory wherever there is a room for it.
- The actual starting address of the object pgm is not known until load time.
- Relocation provides the efficient sharing of the m/c with larger memory & when several independent programs are to be run together.
- It also supports the use of subroutine libraries efficienty.

PAGE No.
DATE / /

Linking loaders -

- Perform all linking & relocation at load time.

The other alternatives:

linkage editors -

- Produces a linked version of the pgm - called a load module or an executable image.
- This load module is written to a file or library for later execution.
- The linked pgm produced is suitable for processing by a relocating loader.
- Using linkage editor, an absolute object program can be created, if starting address is already known.

Dynamic linking -

- This scheme postpones the linking functions until execution.
- A subroutine is loaded & linked to the rest of the program when it is first called.
- It is usually called dynamic linking, dynamic loading or load on call.

Macro Example (Tut)

```
MACRO
INCR &M, &I=, &REG=AREG
MOVER &REG, &M
ADD &REG, &I
MOVEM &REG, &M
MEND
MACRO
CALC &X, &Y=B, &OP=MULT
MOVER AREG &X
```

```
&OP AREG, &Y
MOVEM AREG, &X
MEND
START 100
READ N1
CALC A, OP=ADD
ADD AREG, 21
LDA CREG, 100
SUB CREG, A
INCR B, REG=BREG, I=A
A DS 1
B DS 1
END
```

* Single and Two pass assembler

Assembly lang. -

Assembly lang. is a kind of low level programming lang., which uses symbolic codes or mnemonics as instruction.

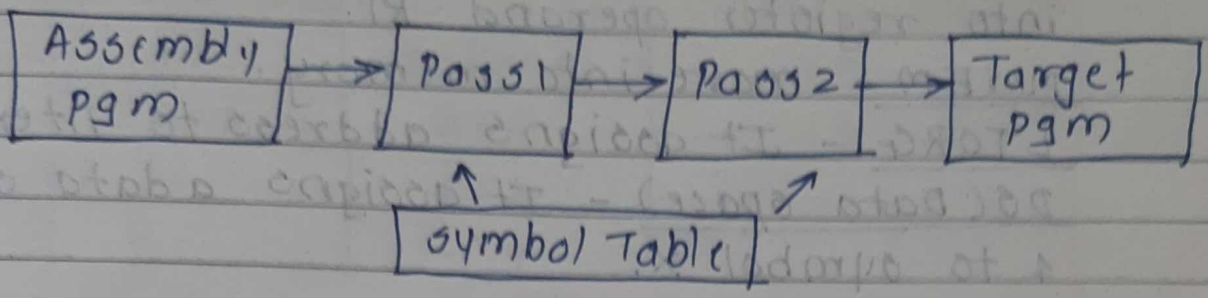
Eg. - ADD, SUB, LDA, STA

Elements of Assembly lang.

1) mnemonic operation code - (mlc inst.)
   ADD, SUB, MOVE
2) symbolic operands - (variables, symbolic name)
   ADD R1, R2, R3
3) Data Declaration -
   NUM1 03
   NUM1 0011H

Assembly process

convert .asm file into .obj file



Pass 1 :
complete scan .asm file. Fine all labels, instructions & calculating corresponding address.

Pass 2 :
convert all the instructions into mlc lang. format

Symbol Table :
stored all the info. of assembly lang. data, variables, instructions, address etc.

Example of Assembly lang :

Assembly Label	Program opcode	operand (value)	cvalue (add?)
JOHN	START	200	
	MOVER	R1, ="3"	200
	MOVEM	R2, X	201
L1	MOVER	R2, ="2"	202
	LTORG		203
X	DS	1	204
	END		205

START - This inst. starts the execution of pgm.
MOVER - It moves the content of literal (='3') into register operand R1.
MOVEM - move into memory.
LTORG - It assigns address to literals.
DS (Data space) - It assigns a data space of 1 to symbol X.

Design of working of Assembler

1) Analysis Phase (Pass 1 of Assembler) -
- To build symbol table for synthesis phase to processed.
- Determine address of each symbols called as memory allocation.
- location counter used to hold address of next inst.
- Isolated label, mnemonic opcode, operands, constants etc.
- validate meaning of address of each inst.

2) Synthesis Phase (Pass 2 Assembler) -
- Use data structure generated by analysis phase.
- To build mlc inst. for every assembly stmt as per mnemonic code & three address allocation.
- Synthesis mlc inst. as per source code.

PAGE No.
DATE / /

Pass 1 Assembler :

1) Symbol Table (ST or SYMTAB) : store value or address assign to the label.

label	address
JOHN	200
L1	202
X	204

2) Literal Table (LT or LITAB) : store each literal or constants (='3') with its location.

Index	literal	Address
0	='3'	200
1	='2'	202

3) Operation code Table (optab) : store mnemonic operation code with there opcodes & length.

on which element action is performed

Inst.	opcode	length (bytes)	
MOVER	3	2	- int
MOVEM	X	1	- char
MOVER	2	2	- int

Pass 2 Assembler :

- Assembled using symbol table.
- generates mic code for inst.
- convert mnemonic operation code, symbol table operands with there equivalent mic code.
- convert data constants to internal mic representations.
- convert complete pgm into obj file.

PAGE No.
DATE / /

Example:

```
    START 200
    MOVER AREG = "5"
    MOVEM AREG, X
L1  MOVR  BREG, ='2'
    ORIGIN L1+3
    LTORG
X   DS    1
    END
```

PAGE No.
DATE / /

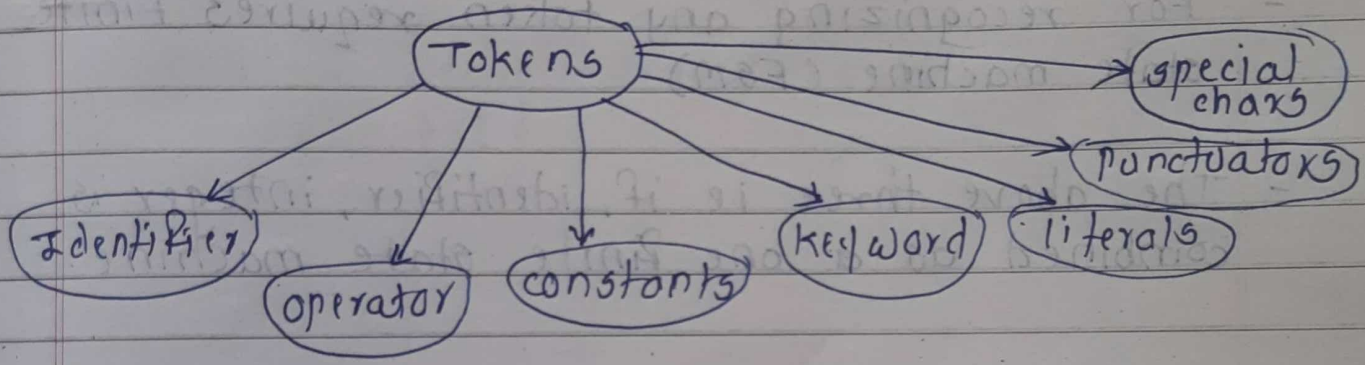
Unit 2: Lexical Analysis & Introduction of Syntax Analysis

Introduction to compiler, Phases & Passes, Bootstraping, Role of lexical analyzer, specification & Recognition of tokens, LEX/FLEX, Expressing syntax, Top-down Parsing, Predictive Parsers, Implementing scanners, operator precedence parsers.

* Lexical Analyzer

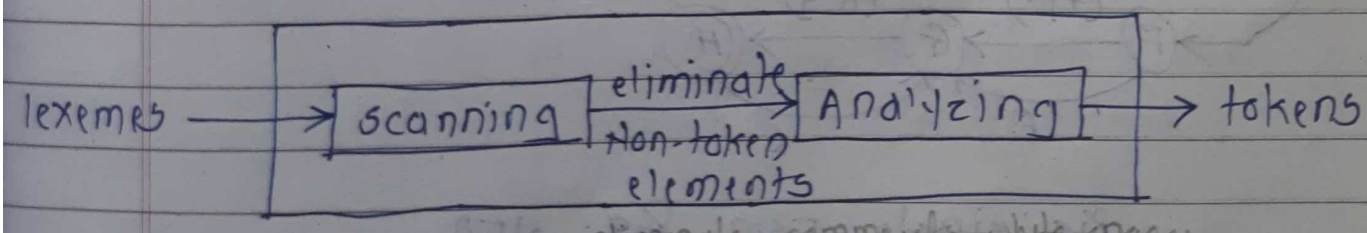
Features

1) Scans the pure HLL code line by line.
2) Tokens lexemes as i/p & produces Tokens.



- Tokens are classified into above 7 categories.

- In lexical analyzer two functions take place.
  - Scanning
  - Analyzing

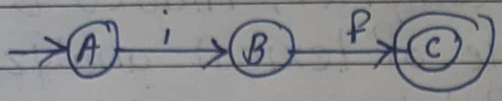


eliminate - comments, white spaces

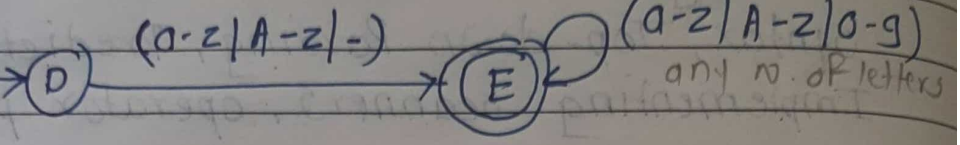
Let see how Analyzing works:

c - Tokens

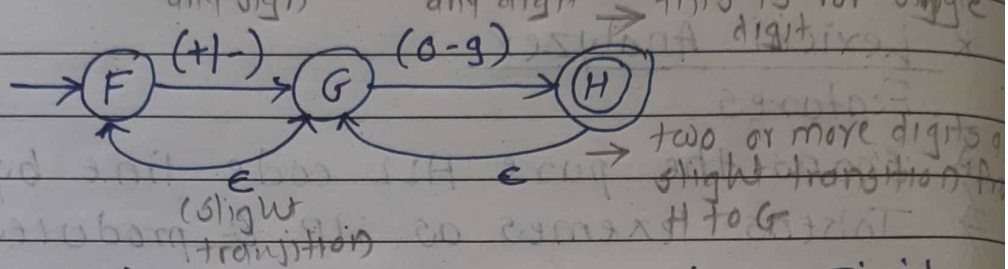
- if :



- Identifier :

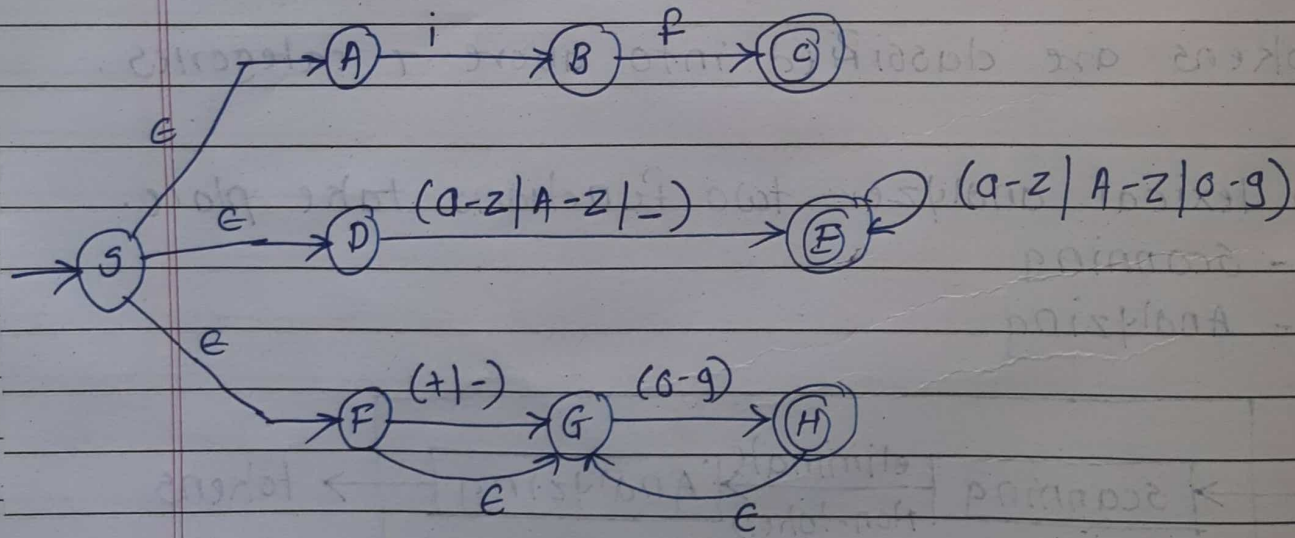


- Integer :



- For recognizing any token requires Finite state machine (FSM)

- The above three ie if, identifier, integer is combined as a one Finite state machine.



~~Fig. single FSM~~ Fig: NFA.

- How the above will combine into a single FSM.

		First
4)	$E \to TE'$	id, (
	$E' \to *TE \mid \epsilon$	*, ϵ
	$T \to FT'$	id, (
	$T' \to \epsilon \mid +FT'$	ϵ, +
	$F \to id \mid (E$	id, (

FOLLOW ()

Follow(A) contains set of all terminals present immediately in right of A.

Rules:

Follow of start symbol is $

Eg.

1) $FO(A) = \{\$\}$

2) $S \to ACD\epsilon$
$C \to a \mid b$

$Follow(A) = First(C) = \{a, b\}$

at right side of A there is C which is non terminal so calculate First of that variable.

$Follow(D) = Follow(S) = \{\$\}$

after D there is nothing so write ϵ, so it will point to start symbol.

3) $S \rightarrow aSbS \mid bSaS \mid \epsilon$

Follow(S) = { $, b, a }

Follow never contain ϵ

4) $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

Follow(A) = { a, b }

Follow(B) = { b, a }

5) $S \rightarrow ABC$

$A \rightarrow DEF$

$B \rightarrow \epsilon$

$C \rightarrow \epsilon$

$D \rightarrow \epsilon$

$E \rightarrow \epsilon$

$F \rightarrow \epsilon$

Follow(A) = First(B) = First(C) = { $ } =
Follow(S) = { $ }

6) $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

after eliminating left recursion the grammer is :

PAGE No.
DATE / /

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

Follow
1) $ (Follow of start)
2) if $A \to \alpha B\beta$ - Follow of B is First of β except ϵ
3) $A \to \alpha B$ / Follow(B) is Follow(A)
First of B contains ϵ then also Follow A

First (E) = First (T) = First (F) = { (, id }
First (E') = { +, ϵ }
First (T) = First (F) = { (, id }
First (T') = { *, ϵ }
First (F) = { (, id }

E is start symbol so $
Follow (E) = { $,) }
Follow (E') = { $,) } after E' is nothing so Follow(E') is Follow(E)
Follow (T) = { +, $,) } First of (E') = + E → TE → First(E') → S, ϵ) ⇒ E → T
Follow (T') = { +, $,) } Follow(E)
Follow (F) = { +, *, $,) } First (T') + Follow (T)

For Follow(E) check on right side where the E appear & taken the right of E
ie (E) =)
Follow (E') = Follow(E)

7) S → iEtS | iEtSeS | a
E → b
after eliminating left factoring,
S → iEtSS' | a
S' → eS | ϵ
E → b
To construct parsing table, we need FIRST() & FOLLOW() for all the non-terminals

PAGE No.

parsing topic: rule.

$A \to \alpha$

1) $m[A, a] = A \to \alpha$ = α is first of α.

2) $m[A, b] = A \to \alpha$ - if ϵ is in first of (α) b is in follow(A)

First(S) = { i, a }
First(S') = { e, ϵ }
First(E) = { b }
Follow(S) = { $, e }
Follow(S') = { $, e }
Follow(E) = { t }

* Parsing Table (For eg. 6) - substitute all production rules in table.
Predictive Parsing Table : - $A \to \alpha$ | find first of right hand side of production

N.T.	+	*	(	)	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 → (E)		F → (id)	

1) E → TE' ⇒ First(T) ⇒ { (, id } so in (& id we write E → TE'

continue -

2) E' → +TE' / ϵ ⇒ First(+) = + ⇒ Follow(E') → $,)
3) T → FT' ⇒ First(F) ⇒ (, id
4) T' → *FT' / ϵ ⇒ First T' ⇒ * (ϵ) ⇒ Follow(T') ⇒ +, $,)
5) F → (E) / id ⇒ First(F) = (, id

* LL(1) : Grammer parsing table

① ②
S → (L) / a
L → SL' ③
L' → ϵ / ,SL' ④ ⑤

- First find out first & follow for all non-terminals.

- Introduce new initial state s & have slight transitions to all other states.
- This one is NFA, so this is an epslient NFA
- NFA is conceptual means we cann't implement it.
- For implementation we will derive an equivalent Deterministic finite autometa (DFA)
- The below is equivalent DFA

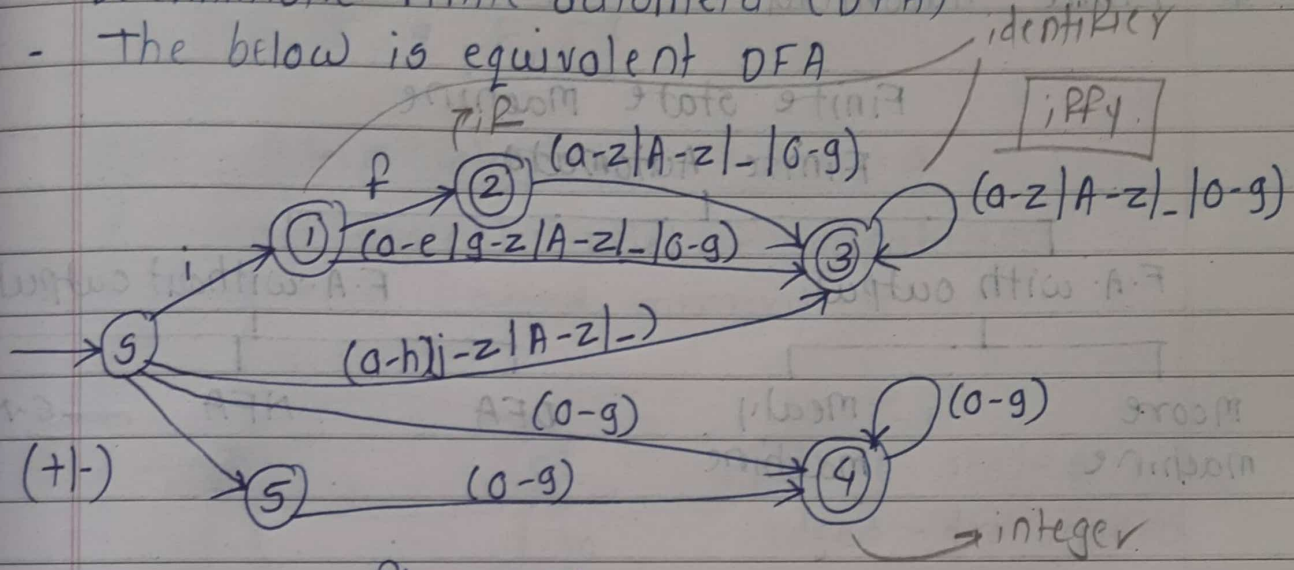
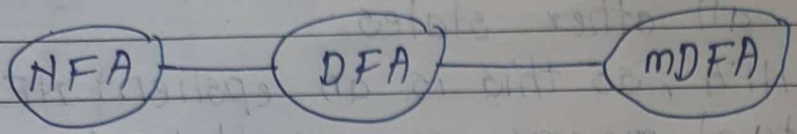


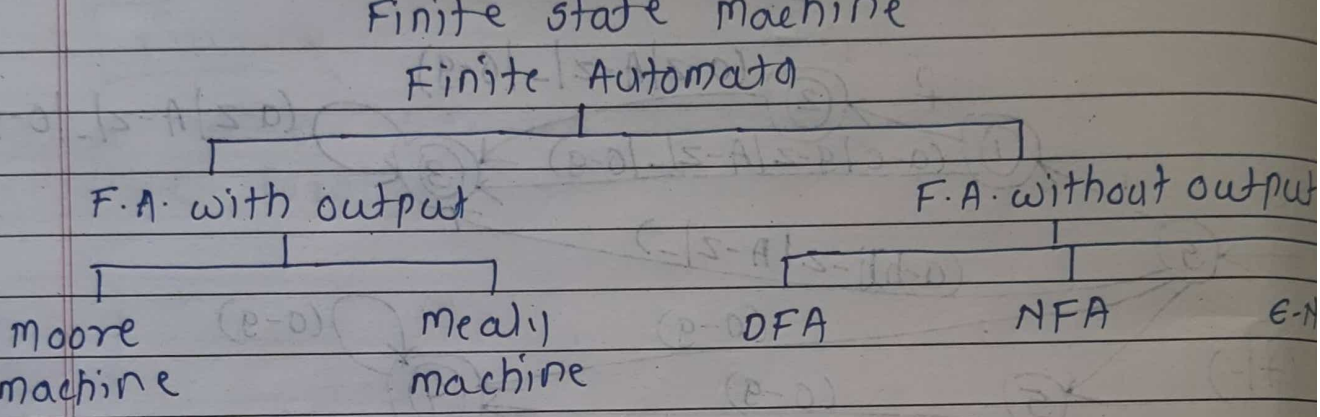
Fig: DFA

- s-1-2 ⇒ if
- s-1-2 (if) — 3 (iFf) — 3 (iFFy) } identifier
- s-1-3 (except i)
- s-3 (except i)
- s-4 (any single digit)
- s-4-4 (self loop) (two or more digits) } integer

- lexical analyzer takes lexemes as i/p & produces tokens - which uses DFA for pattern matching.



* Finite Automata



- Finite automata (FA) is the simplest mlc to recognize patterns.
- It has set of states and rules for moving from one state to another but it depends upon the applied input symbol.

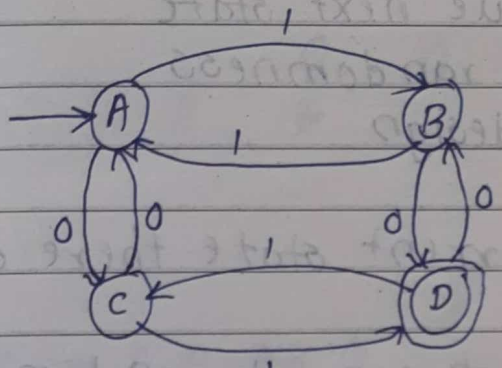
* Features of automata:

1. Input
2. Output
3. states of automata
4. state relation
5. output relation

It consist of following
Q : finite set of states
Σ : set of i/p symbols
q : initial states
F : set of final states
δ : transition function
{ Q, Σ, q, F, δ }

DFA - Deterministic Finite Automata

- It is the simplest model of computation.
- It has very limited memory



where,
circle - states
edges - transitions
0,1 - inputs

Every DFA can be defined using 5 tuples
(Q, Σ, q, F, δ)
Q - set of all states
Σ - inputs
q - start state / initial state
F - set of final states
δ - transition function from $Q \times \Sigma \rightarrow Q$

Q - { A, B, C, D }
Σ - { 0, 1 }
q - A
F - { D }
δ -

For δ

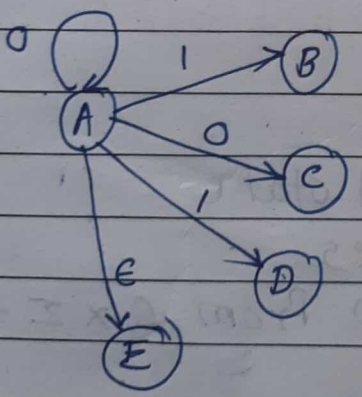
	0	1
A	C	B
B	D	A
C	A	D
D	B	C

when we are in state A if my input is 0 then we will go to state C & if my input is 1 we will go to state B. likewise.

NFA - Non-deterministic Finite automata

- In DFA, given the current state we know what the next state will be
- It has only one unique next state
- It has no choices or randomness.
- It is simple & easy to design

- In NFA, given the current state there could be multiple next states.
- The next state may be chosen at random.
- All the next states may be chosen in parallel



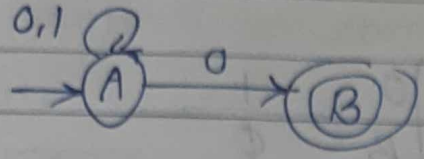
ϵ - empty string

- NFA is also define with 5 tuples
  $(Q, \Sigma, q, F, \delta)$

Q - set of all states
Σ - input
q - start or initial state

F - set of final states
δ - transition function that maps to

$$Q \times \Sigma = 2^Q$$



$Q = \{A, B\}$
$\Sigma = \{0, 1\}$
$q_0 = A$
$F = B$
$\delta = ?$

	0	1	
	A	A	$A \times 0 \to A$ state A on getting i/p 0
A	A		$A \times 0 \to B$ state A on getting i/p 0 goes to B
			$A \times 1 \to A$
			$B \times 0 \to \phi$ does not go anywhere so represented by ϕ
			$B \times 1 \to \phi$

$A \to A, B, AB, \phi$ For state A getting i/p either 0 or 1 it goes to either A, B, AB, ϕ
if there are 2 state there are 4 possibilities
For 3 it is 8 possibilities

* Conversion of NFA to DFA

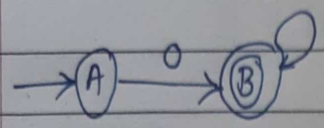
- Every DFA is an NFA, but not vice versa, but there is an equivalent DFA for every NFA.

DFA	NFA
$\delta = Q \times E \to Q$	$\delta = Q \times \Sigma \to 2^Q$

$NFA \cong DFA$

$L = \{$ set of all strings over $(0,1)$ that starts with '0' $\}$

$\Sigma = \{0, 1\}$



NFA

	0	1
A	B	ϕ
B	B	B

Now convert this state transition diagram to equivalent DFA

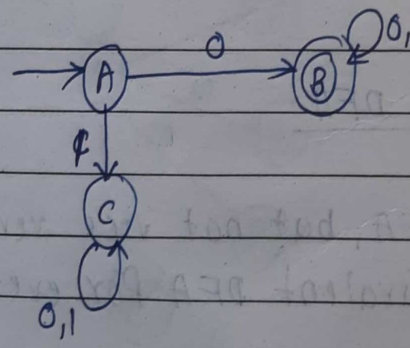
DFA -

	0	1
A	B	C
B	B	B
C	C	C

C - dead state / trap state

we cant indicate ϕ in DFA means a dead state cant indicate so we introduce a new state ie C.

state diagram for DFA :



Feature :

3) Removes comments & whitespaces from the Pure HLL code.

PAGE No.
DATE / /

Types of comments in C

// single line comment

/* multi line comment */

- while scanning lexical analyzer detects comments
- If its a single line comment till enter encounters it ignores that line & if its multiline till the next */ it will ignore all the text.
- The lexical analyzer classify the comments as non token elements & eliminates those.

For eg.

```
int VIT /* it is a comment */ Pune;
                //
        int VIT Pune;
```

- after scanning by lexical analyzer the comment will be removed, the blank white space is put between the comment.
- During symantic analysis an error is generated, there pune is reported as undeclared variable.
- The lexical analyzer predicts the 2 different tokens (ie VIT Pune).

Whitespaces

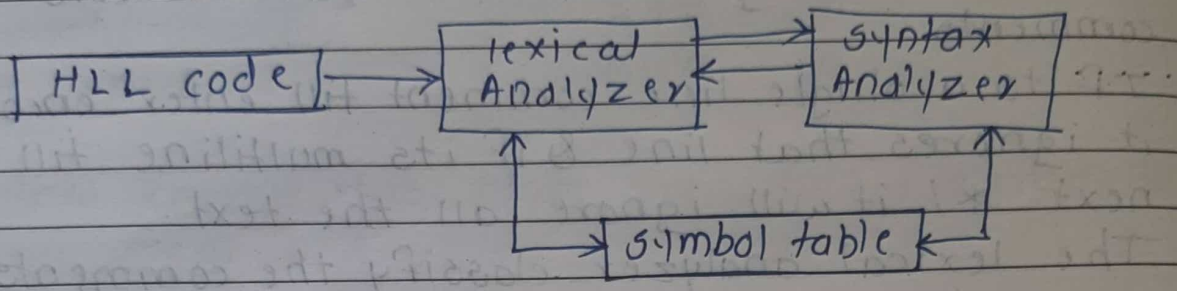
types

' '	space
'\t'	horizontal tab
'\n'	newline
'\v'	vertical tab
'\f'	form feed
'\r'	carriage return

- all whitespaces are non-token elements.

feature

4) Helps in macro expansion in the pure HLL code.



* Specification & Recognition of Tokens

- Recognition of tokens by lexical analyzer.
- It is done to separate out different tokens.
- To understand the concept, let us consider that we have certain regular expression such as:
  - if → if
  - then → then (condition operator)
  - else → else
  - relop → < | <= | > | >= | = | <> - relation operator
  - id → letter (letter | digit)* (identifier (* - closure))
  - num → digit+ (digit)* ? (∈ (+|-)) ? digit+ ?
  - delim → blank | tab | newline

* Specification of Token

There are 3 specifications of tokens:

1) strings
2) languages
3) Regular expression

PAGE No.
DATE / /

String & languages

- An alphabet or character class is a finite set of symbols.
- A string over an alphabet is a finite sequence of symbols drawn from that alphabet.
- A language is any countable set of strings over some fixed alphabet.
- In language theory, the terms 'sentence' & 'word' are often used as synonyms for string.
- The length of string s, usually written |s|, is the no. of occurrences of symbols in s.
- For example, banana is a string of length six. The empty string, denoted ϵ, is the string of length zero.

Operations on string

1. A prefix of string is any string obtained by removing zero or more symbols from the end of string s. For eg., ban is a prefix of banana.
2. A suffix of string s is any string obtained by removing zero or more symbols from the beginning of s. For eg., nana is a suffix of banana.
3. A substring of s is obtained by deleting any prefix & any suffix from s. For eg., nan is a substring of banana.
4. The proper prefixes, suffixes, substrings of a string s are those prefixes, suffixes & substrings, respectively of s that are not ϵ or not equal to s itself.
5. A subsequence of s is any string formed by deleting zero or more not necessarily consecutive

positions of s.

6. For eg., baan is a subsequence of banana.

Operations on languages

The following are the operations that can be applied to languages:

1) Union
2) concatnation
3) Kleene closure
4) Positive closure

Let $L = \{0, 1\}$ & $S = \{a, b, c\}$

1) Union : $L \cup S = \{0, 1, a, b, c\}$

2) concatnation : $L \cdot S = \{0a, 1a, 0b, 1b, 0c, 1c\}$

3) Kleene closure : $L^* = \{\epsilon, 0, 1, 00 \ldots\}$

4) Positive closure : $L^+ = \{0, 1, 00 \ldots\}$

Regular Expression

- Each regular expression r denotes a language $L(r)$.
- Here are the rules that define the regular expressions over some alphabet Σ & the languages that those expressions denote.

1) ϵ is regular expression, & $L(\epsilon)$ is $\{\epsilon\}$, ie, the lang. whose sole member is the empty string.

2) If 'a' is a symbol in Σ, then 'a' is regular expression & $L(a) = \{a\}$, ie, the lang. with one string, of length one, with 'a' in its one position

3) Suppose r & s are regular expressions denoting the langs $L(r)$ & $L(s)$. Then
   a) $(r)|(s)$ is a regular expression denoting the lang $L(r) \cup L(s)$.
   b) $(r)(s)$ is a regular expression denoting the lang. $L(r)L(s)$.
   c) $(r)^*$ is a regular expression denoting $(L(r))^*$.
   d) (r) is a regular expression denoting $L(r)$.

4) The unary operator * has highest precedence & is left associative.

5) concatnation has second highest preceddence & its left associative.

6) | has lowest precedence & is left associative.

Regular set

- A lang. that can be defined by a regular expression is called a regular set. If two regular expressions r & s denote the same regular set, we say that they are equivalent & write $r = s$.
- There are a no. of algebric laws for regular expressions that can be used to manipulate into equivalent forms.
For instance, $r|s = s|r$ is commutative; $r(s|t) = (r|s)|t$ is associative.

Regular definitions

- Giving names to regular expressions is referred to as a regular definitions. If Σ is an alphabet

of basic symbols, then a regular definition is a sequence of definitions of the form

$d_1 \rightarrow r_1$,

$d_2 \rightarrow r_2$

.....

.....

$d_n \rightarrow r_n$

1. Each d_i is a distinct name.
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

letter → A|B|...|z|a|b|...|z|
id → letter (letter | digit)*
digit → 0|1|...|9

How to write regular expression ?

1) *, +, { } ⇒
These symbols act as repeaters & tell the computer that the preceding character is to be used for more than just one time.

2) \ - escape character
3) . - any character
4) \d - digit
5) \D - not a digit
6) \w - word character
7) \W - not a word character
8) \s - whitespace
9) \S - not whitespace
10) \b - word boundary
11) \B - not a word boundary
12) ^ - beginning of string
13) $ - end of string

PAGE No.	
DATE	/ /

[ ]	- matches chars in brackets
[^]	- matches chars not in brackets
|	- either or
( )	- capturing group
*	- 0 or more
+	- 1 or more
?	- 0 or 1
{ }	- exact no. of characters
[min, max]	- range of no. of chars.

* ~~Specification of Tokens~~

* Recognition of Tokens

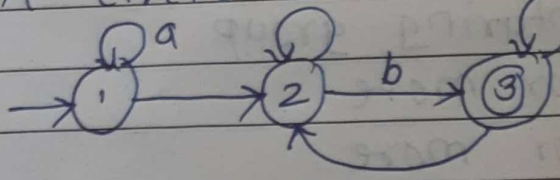
- Tokens can be recognized by finite automata.
- A finite automata is a simple idealized machine used to recognize patterns within input taken from some character set.
- The job of FA is to accept or reject an i/p depending on whether the pattern defined by the FA occurs in the i/p.
- There are two notations for representing finite automata. they are:

    1) Transition Diagram
    2) Transition Table

- Transition diagram is a directed labeled graph in which it contains nodes & edges.
- Nodes represents the states & edges represents the transition of a state.

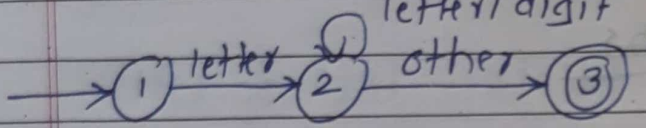
- Every transition diagram is only one initial state represented by an arrow mark (→) & zero or more final states are represented by double circle.

eg. -



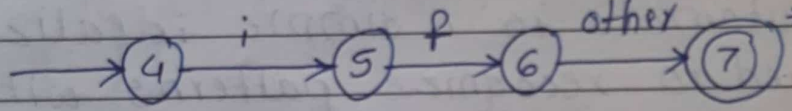
where state '1' is initial state & state 3 is final state.

* • Finite Automata for recognizing identifiers



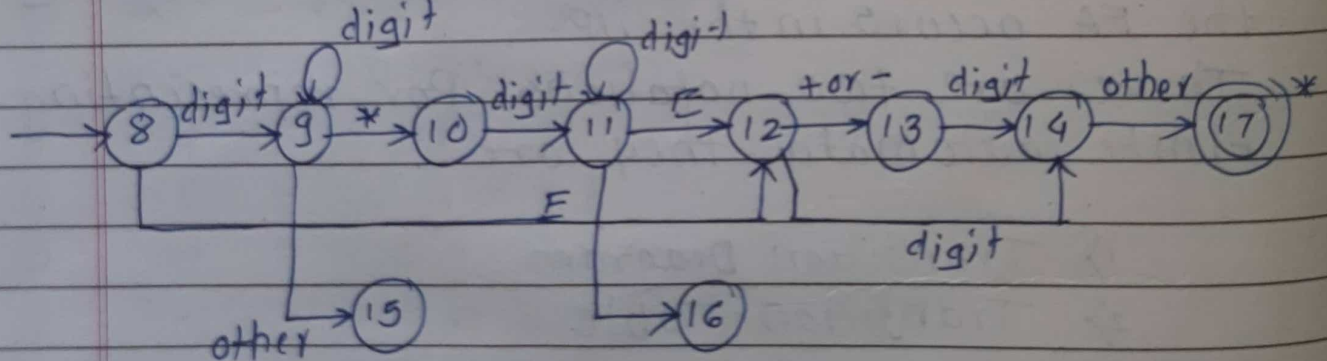
Return (getToken(), installID())

• Finite Automata for recognizing keywords

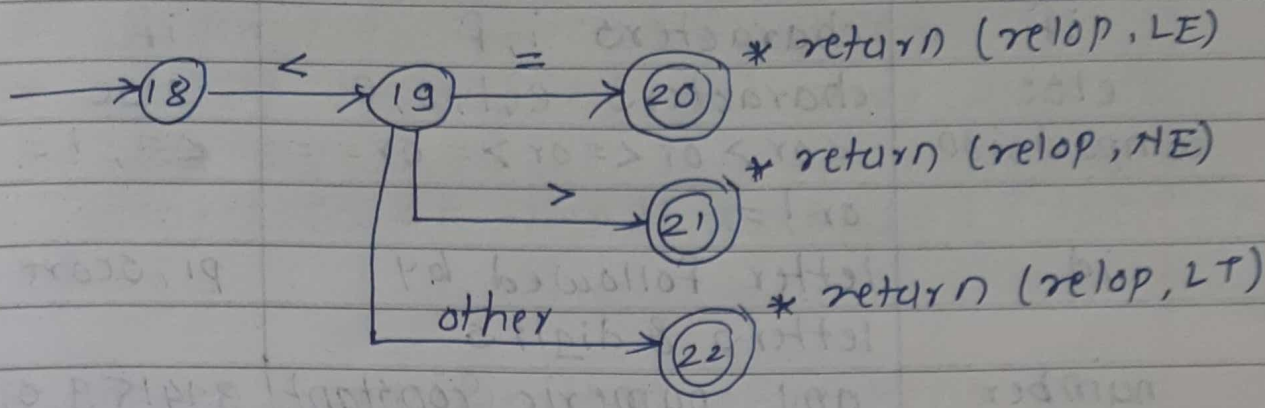


accept if as keyword

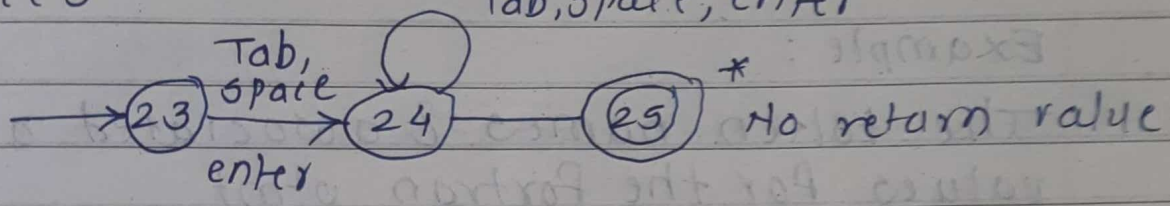
* • Finite automata for recognizing numbers



- Finite automata for relational operators



- Finite automata for recognizing white spaces



* Tokens, Pattern, Lexemes

- A token is a pair consisting of a token name & an optional attribute value.
- A pattern is a description of the form that the lexemes of a token may take.
- A lememe is a sequence of characters in the source program that matches the pattern for a token & is identified by the lexical analyzer as an instance of that token.

For eg:

printf("Total = %d\n", score);

both printf & score are ~~term~~ lexemes matching the pattern for token id & "Total = %d\n" is a lexeme matching literal.

Token	informal description	sample lexeme
if	characters i, f	if
else	characters e, l, s, e	else
comparision	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters & digits	pi, score, D2
number	any numeric constant	3.1415 9, 0, 6.02e
literal	anything but ", surrounded by "s	"core dumped"

Example :

The token names & associated attribute values for the fortran stmt

E = M * C ** 2

<id, pointer to symbol table entry for E>
<assign_op>
<id, pointer to symbol table entry for M>
<mult_op>
<id, pointer to symbol table entry for C>
<exp_op>
<number, integer value 2>

Example :

Divide the following c++ pgm into appropriate lememes, which lexemes should get associated lexical values? What should those values be?

** ⇒ Fortran exponentiation operator

PAGE No. / DATE / /

```
float limitedSquare(x) float x {
/* returns x-squared, but never more than
   100 */
   return (x<=-10.0 || x>=10.0) ? 100 : x*x;
}
```

* Examples: Specification of tokens

Describe the languages denoted by the following regular expressions:

1) a(a|b)*a
- string of a's & b's that start & end with a.

2) ((ε|a)b*)*
- string of a's & b's.

3) (a|b)* a (a|b) (a|b)
- string of a's & b's that the character third form the last is a.

4) a*ba*ba*ba*
- string of a's and b's that only contains three b.

5) !! (aa|bb)* ((ab|ba) (aa|bb)* (ab|ba) (aa|bb)*)*
- string of a's and b's that has a even number of a and b.

Write a RE for the set of strings of a's & b's that begin with either aa or bb & have an optional c at the end.

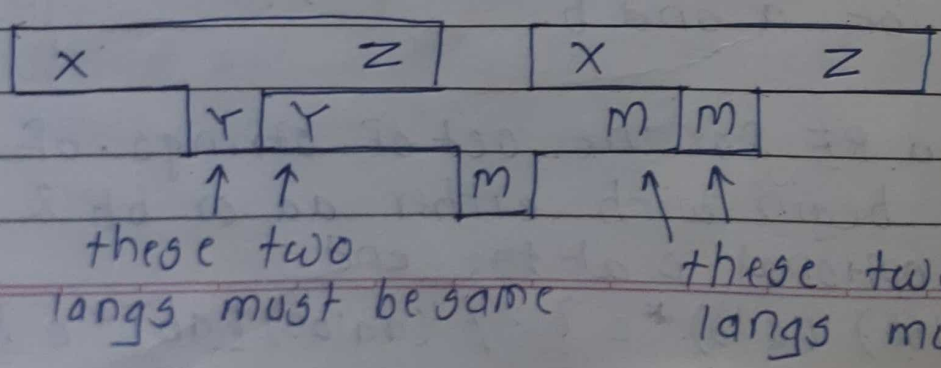
⇒ (aa|bb) (a|b)*c? / (aa|bb)[ab]*c?

PAGE No.
DATE / /

* Bootstrapping in compiler design

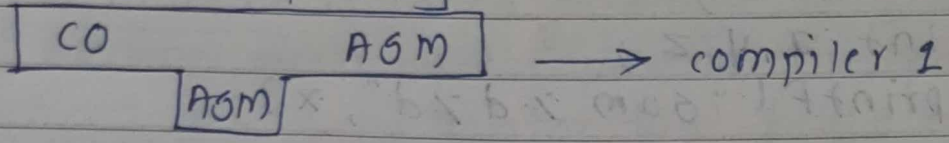
- Bootstrapping is a process in which simple lang. is used to translate more complicated program which in turn may handle for more complicated pgm.
- This complicated pgm can further handle even more complicated pgm & so on.
- Writing a compiler for any high level lang. is complicated process. It takes lot of time to write a compiler from scratch.
- Hence simple lang. is used to generate target code in some stages.
- Suppose we want to write a cross compiler for new language x. The implementation lang. of this compiler is say Y. and the target code being generated is in language Z.
- That is, we create XYZ.
- Now if existing compiler Y runs on mc M & generates code for M then it is denoted as YMM.
- Now if we run XYZ using YMM then we get a compiler XMZ; that means a compiler for source lang. x that generates a target code in lang. z & which runs on machine M.

Example:

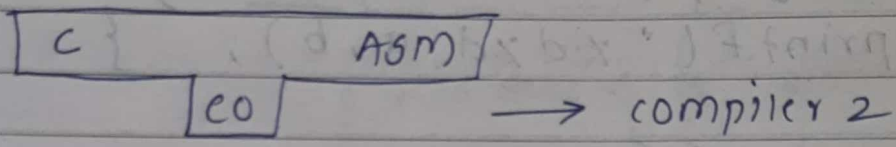


compiler which takes c lang. & generates an assembly lang. as an o/p with the availability of a m/c of assembly lang.

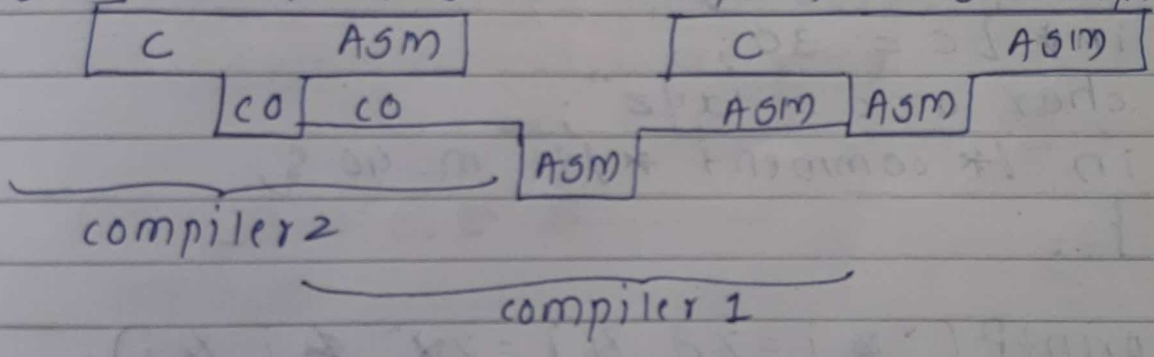
step 1: First we write a compiler for a small of c in assembly lang.



step 2: Then using with small subset of c ie co, for the source lang. c the compiler is written



step 3: Finally we compile the second compiler, using compiler 1 the compiler 2 is compiled.



step 4: Thus we get a compiler written in ASM which compiles c & generates code in ASM.

Find the no. of tokens in the given pgm

1) int main)(
{
x = 4 + z ;
int x, 4, z ;
printf("sum %d %d", x);
{ 26 } 26

2) main() {
a = b + + + - - - - + + += = ;
printf("%d %d, a, b); }
⇒ stop here only

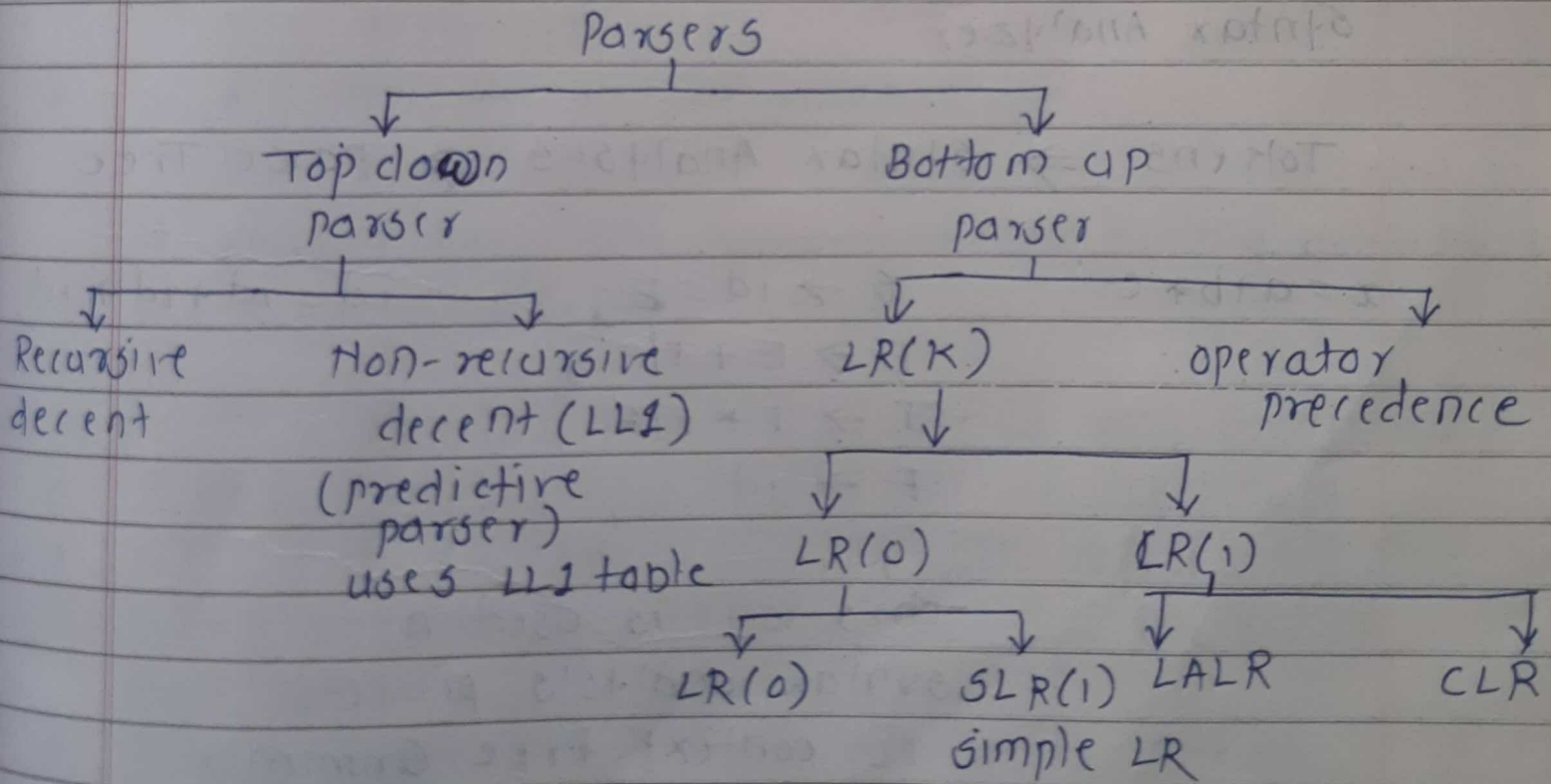
3) main() {
int a = 10 ;
char b = 'abc' ;
int *c = 30 ;
char ard = "xyz" ;
in /* comment */ t m = 40.5 ;
}

4) printf("// i = %d, & i = %x", i, & i);
count ⇒ 10

* Parsers

Parsing :

- The process of transforming the data from one format to another is called parsing. This process can be accomplished by the parser. The parser is a component of the translator that helps to organise linear text structure following the set of defined rules which is known as grammer.
- It is a process of deriving string from a given grammer.
- Stream of tokens given as a input to parser (ie to syntax analyzer).
- whether the token belong to proper grammer or not.
- It creates parse tree (syntax tree, derivation tree).
- It uses CFG (context free grammer).



LL(1) - (L) left to right
(L) left most derivation
(1) look at one symbol

LR(0), LR(1) - find the canonical item

LR(0) - (L) - left to right scan
(R) - right most derivation

LALR - look ahead LR
CLR - canonical LR

operator precedence - operates on unambiguous grammer & ambiguous grammer.

* Formal Grammer

Syntax Analyzer

Tokens ⇒ Syntax Analysis ⇒ Parse Tree

x = a+b*c

S → id = E;
E → E+T | T;
T → T*F | F
F → id

id = id + id *

↑
This one is used in syntax analysis phase
i.e. context free Grammar
(CFG)

- CFG is a grammer of formal language
- For the grammer we have set of non-terminals (variables) and terminals (constants)

non-terminals - noun, verb, adjective, verb, phrase.
Terminals - constants, words in sentences.

<S> → <noun> <verb phrase>
<verb phrase> → <verb> <adjective>
<noun> → noun_word
<verb> → <verb_word>
<adjective> → <adjective_word>

So, the formal grammer can be describe by using 4 tuples <N, T, P, S>

N - non-terminals
T - terminals
P - specific set of rules
S - sentence

A phrase structure grammer (or simply grammer) is (N, T, P, S) where

1) N is a finite, non-empty set of non-terminals
2) T is a finite, non empty set of terminals
3) $N \cap T = \phi$ (set of non-terminals & terminals dont have anything in common)
4) S is a special non-terminal (ie $S \in N$), called start symbol
5) P is a finite set whose elements are of the form, $\alpha \rightarrow \beta$ [NUT]

production rules

PAGE No.
DATE : / /

α - must have at least non-terminal.

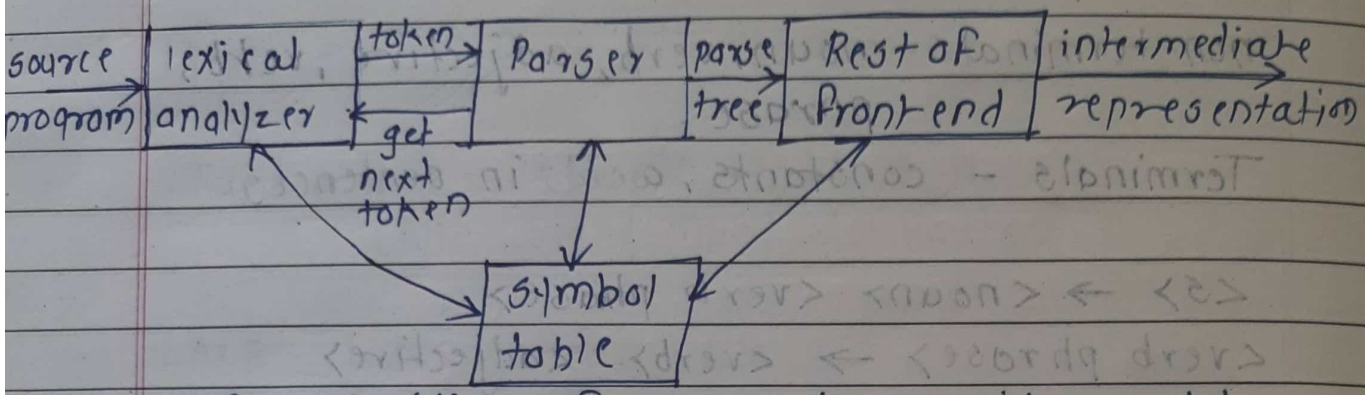


Fig : Position of parser in compiler model

Context free Grammer

1. $E \rightarrow E+E$
2. $E \rightarrow E*E$
3. $E \rightarrow id$

Grammer can be defined as 4 tuples

(N, T, P, S)

$N = E$

$T = +, *, id$

$P =$ all sentences

$S = E$ (start symbol)

CFG (type 2) can be written as

Type 2 : $A \rightarrow \alpha$

$A \in N$
$\alpha \in V^*$

V^* - any combination of terminals & nonterminals including ϵ

suppose from the above we want to derive id + id x id

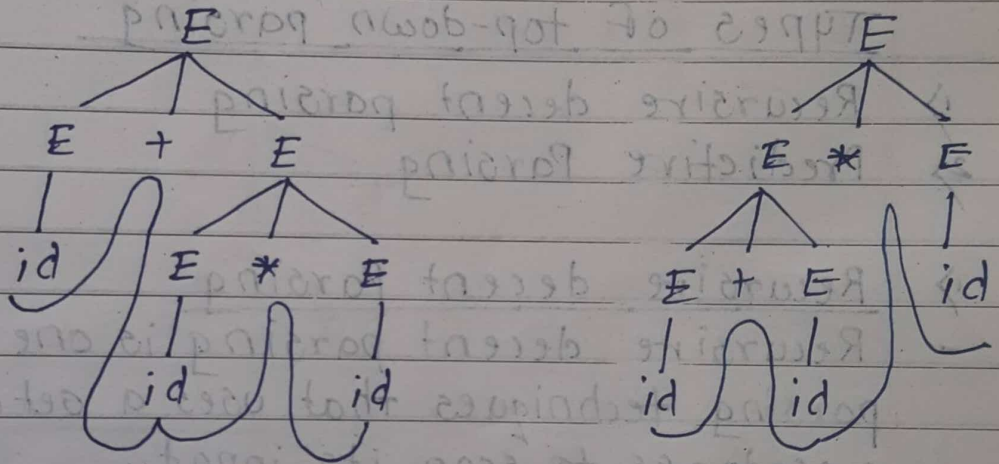
left most derivation

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

right most derivation

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

Parse tree derivation:



- In the above example, if we obtain more than one left most derivation, more than one right most derivation, more than one parse tree derivation then the grammer is ambiguous.

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

$id + id * id$

PAGE No.
DATE / /

Top-Down Parser

- A parser can start with the start symbol & try to transform it to input string.
  Example: LL Parsers

- 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 decent parsing
2) Predictive Parsing

1) Recursive decent Parsing

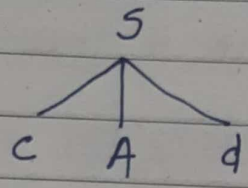
- Recursive decent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input.
- This parsing method may involve backtracking i making repeated scans of the input.

Backtracking example:
consider the grammer G: $S \rightarrow cAd$
$A \rightarrow ab \mid a$
& the input string $w = cad$.

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

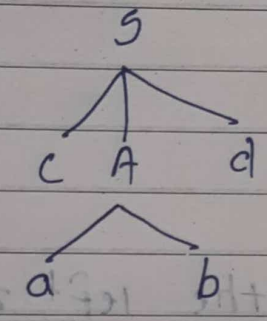
step1:

Initially create a single node labeled S. An i/p pointer points to 'c', the first symbol of ω. Expand the tree with the production of S.



step 2:

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



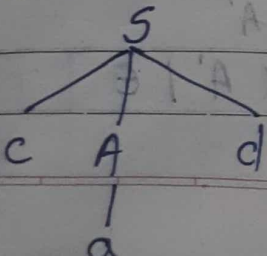
step 3:

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

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

step 4:

Now try the second alternative for A.



A left recursive grammer can cause a recursive decent parser to go into an infinite loop. Hence elimination of left recursion must be done before parsing.

PAGE NO.
DATE / /

Example of recursive decent parsing:

consider the grammer for arithmetic expressions

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

After eliminating the left recursion the grammer becomes

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

How to eliminate the left recursion from the Grammer?

A Grammer $G(V, T, P, S)$ is left recursive if it has a production in the form

$$A \rightarrow A\alpha \mid B$$

The above grammer is left recursive because the left of production is occurring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with

$$A \rightarrow BA'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

left recursion - A production of grammer is said to be left recursion if the leftmost variable of its RHS is same as variable of its LHS.

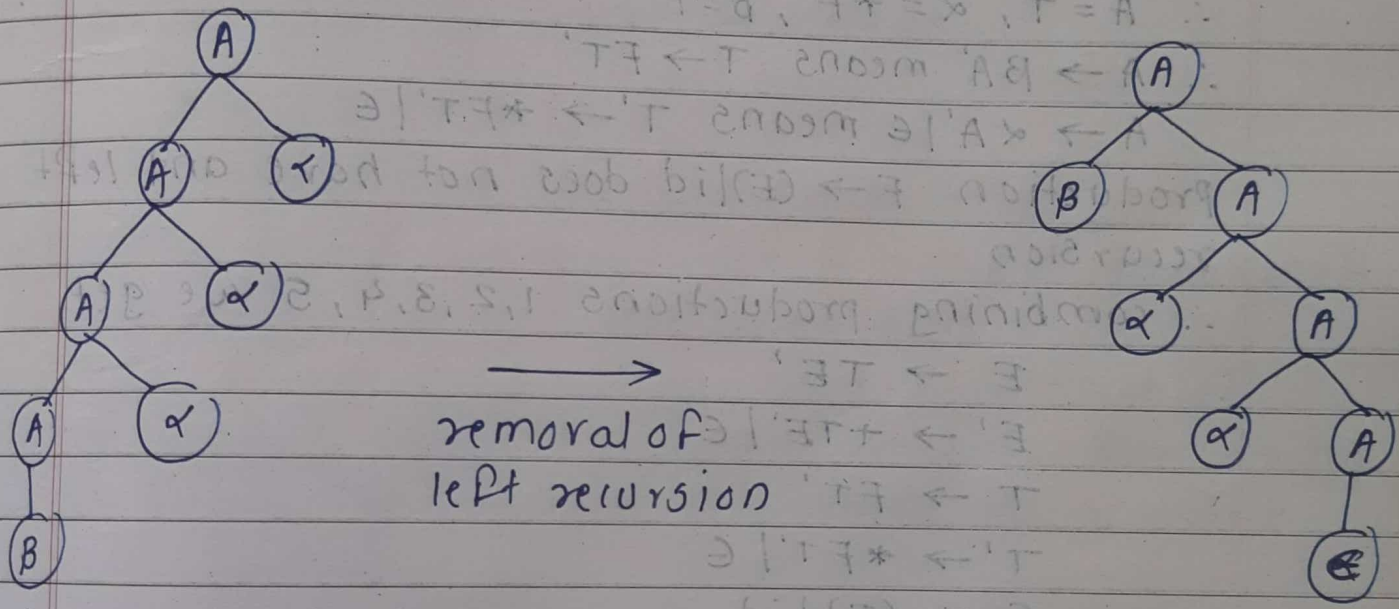
PAGE No.
DATE / /

Elimination of left recursion

left recursion can be eliminated by introducing new non-terminal A such that

$A \to A\alpha | B$ $\xrightarrow{\text{removal of left recursion}}$ $A \to BA'$, $A' \to \alpha A' | \epsilon$

~~Example~~ In left recursive grammer, expansion of A will generate Aa, Aaa, Aaaa at each step, causing it to enter into an infinite loop



Example:

consider the left recursion from the grammer

$E \to E+T | T$

$T \to T*F | F$

$F \to (E) | id$

solution:

comparing $E \to E+T | T$ with $A \to A\alpha | B$

E	→	E	+T	|	T
A	→	A	α	|	B

∴ $A = E$, $\alpha = +T$, $B = T$

∴ $A \to A\alpha | B$ is change to

$A \to BA'$ &

$A' \to \alpha A' | \epsilon$

∴ $A \to BA'$ means $E \to TE'$

$A' \to \alpha A' | \alpha$ means $E' \to +TE' | \epsilon$

comparing $T \to T*F | F$ with $A \to A\alpha | B$

T	→	T	* F		F
A	→	A	α		B

∴ $A = T$, $\alpha = *F$, $B = F$

∴ $A \to BA'$ means $T \to FT'$

$A \to \alpha A' | \epsilon$ means $T' \to *FT' | \epsilon$

Production $F \to (E) | id$ does not have any left recursion

∴ combining productions 1, 2, 3, 4, 5 we get

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) | id$

Example:

$S \to a | \wedge | (T)$

$T \to T, S | S$

solution :

$S \to a | \wedge (T)$

$T \to ST'$

$T' \to ,ST' | \epsilon$

Example :

$E \rightarrow E(T) | T$

$T \rightarrow T(F) | F$

$F \rightarrow id$

solution :

$E \rightarrow TE'$

$E \rightarrow (T)E' | \epsilon$

$T \rightarrow FT'$

$T' \rightarrow (F)T' | \epsilon$

$F \rightarrow id$

* Left Factoring (~~Left Recursion~~)

$A \rightarrow \alpha B_1 | \alpha B_2 | \alpha B_3$

(grammer with common prefixes)

- This kind of grammer creates a problematic situation for top-down parsers.
- Top down parsers can not decide which production must be chosen to parse the string in hand. To remove this confusion, we use left factoring.
- left factoring is a process by which the grammer with common prefixes is transformed to make it useful for top down parsers.

In left factoring -

1) we make one production for each common prefixes.
2) The common prefix may be a terminal or a non-terminal or combination of both.
3) Rest of deriration is added by new productions.

Example :

$P \rightarrow P+Q \mid Q$

Here P is similar to generate non terminal

So,

$P \rightarrow P+Q \mid Q$

$A \rightarrow A\alpha \mid B$ — (1)

* So if we will observe the eq. (1) its left recursive, so we can eliminate it

If there is any production

$A \rightarrow \alpha B_1 \mid \alpha B_2$, it can be rewritten as

$A \rightarrow \alpha A'$

$A' \rightarrow B_1 \mid B_2$

Example :

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

$E \rightarrow b$

* left factored, this grammer becomes

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

PAGE No.
DATE / /

Example :

$A \rightarrow aAB / aBC / aAC$

left Factored - (step 1)

$A \rightarrow aA'$

$A' \rightarrow AB / Bc / AC$

step 2 :

$A \rightarrow aA'$

$A' \rightarrow AD / BC$

$D \rightarrow B / C$

Example :

$S \rightarrow bSSaaS / bSSaSb / bSb / a$

step 1 :

$S \rightarrow bSS' / a$

$S' \rightarrow SaaS / SaSb / b$

step 2 :

$S \rightarrow bSS' / a$

$S' \rightarrow SaA / b$

$A \rightarrow aS / Sb$

Example :

$S \rightarrow a / ab / abc / abcd$

step 1 :

$S \rightarrow aS'$

$S' \rightarrow b / bc / bcd / \epsilon$

step 2 :

$S \rightarrow aS'$

$S' \rightarrow bA / \epsilon$

$A \rightarrow c / cd / \epsilon$

capital letter denotes variables
small letter denotes terminals

PAGE NO.
DATE / /

step 3:

$S \rightarrow aS'$

$S' \rightarrow bA / \epsilon$

$A \rightarrow cB / \epsilon$

$B \rightarrow d / \epsilon$

* Predictive Parser (LL1)

steps:

1. Remove left recursion, left factoring, ambiguity
2. Use first() & follow() method for all non-terminals
3. construct predictive parsing table
4. stack implementation
5. Parse ~~Tree generation~~ the given input string using stack & parsing table.

* FIRST()

First & follow sets are needed so that the parser can properly apply the needed production rule at the correct position.

Example:

$A \rightarrow abc / def / ghi$

Then we have,

$FIRST(A) = \{a, d, g\}$

Rules:

1. For a production rule $X \rightarrow \epsilon$,

$FIRST(X) = \{\epsilon\}$

PAGE NO.
DATE / /

2. For any terminal symbol 'a',

$$FIRST(a) = \{a\}$$

3. For production rule

$$X \rightarrow Y_1 Y_2 Y_3$$

Then,

calculating FIRST(X)

- If $\epsilon \notin First(Y_1)$, then $First(X) = First(Y_1)$
- If $\epsilon \in First(Y_1)$, then $First(X) = \{First(Y_1) - \epsilon\} \cup First(Y_2 Y_3)$

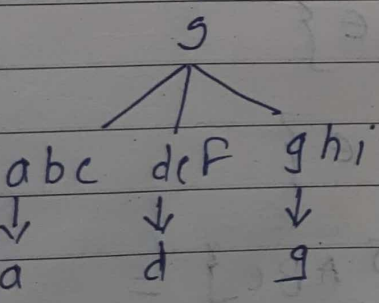
calculating $First(Y_2 Y_3)$

- If $\epsilon \notin first(Y_2)$, then $first(Y_2 Y_3) = First(Y_2)$
- If $\epsilon \in First(Y_2)$, then $First(Y_2 Y_3) = \{First(Y_2) - \epsilon\} \cup First(Y_3)$

- First(A) contains all terminals present in first place of every string derived by A.

Examples:

1) $S \rightarrow abc \mid deF \mid ghi$



$First(S) = \{a, d, g\}$

First (terminal) = terminal
First (ϵ) = ϵ

2) S → ABC | ghi | jkl
A → a | b | c
B → b
D → d

First (D) = { d }

First (B) = { b }

First (A) = { a, b, c }

First (S) = First { ABC } → First { A } = a, b, c

∴ First (S) = { a, b, c, g, j }

3) S → ABC
A → a | b | ϵ
B → c | d | ϵ
C → e | f | ϵ

First (C) = { e, f, ϵ }

First (B) = { c, d, ϵ }

First (A) = { a, b, ϵ }

ϵ ϵ ϵ

First (S) = First { ABC } =
First { a, b, c, d, e, f, ϵ }

First (S) = { (, a }
First (L) = First (S) = { (, a }
First (L') = { ϵ, , }

Follow (S) = { $, , ,) }
Follow (L) = {) }
Follow (L') = Follow (L) = {) }

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

Fill the productions in parse table.

First production

S → (L) — calculate the First of right side. (First is ()

S → a — First is a

L → SL' — First (S) = ((, a)

L' → ϵ — L' → ϵ - follow (L') = {) }

L' → ,SL' → ,

In the parse table if a column contains more than one entry then it is not a LL(1) grammer.

The above is LL(1) grammer.

$ - represents the bottom of stack
E - represents start symbol

PAGE No.
DATE / /

continue -

w = id * id + id we can pars this string by using above predictive parsing table

stack implementation :

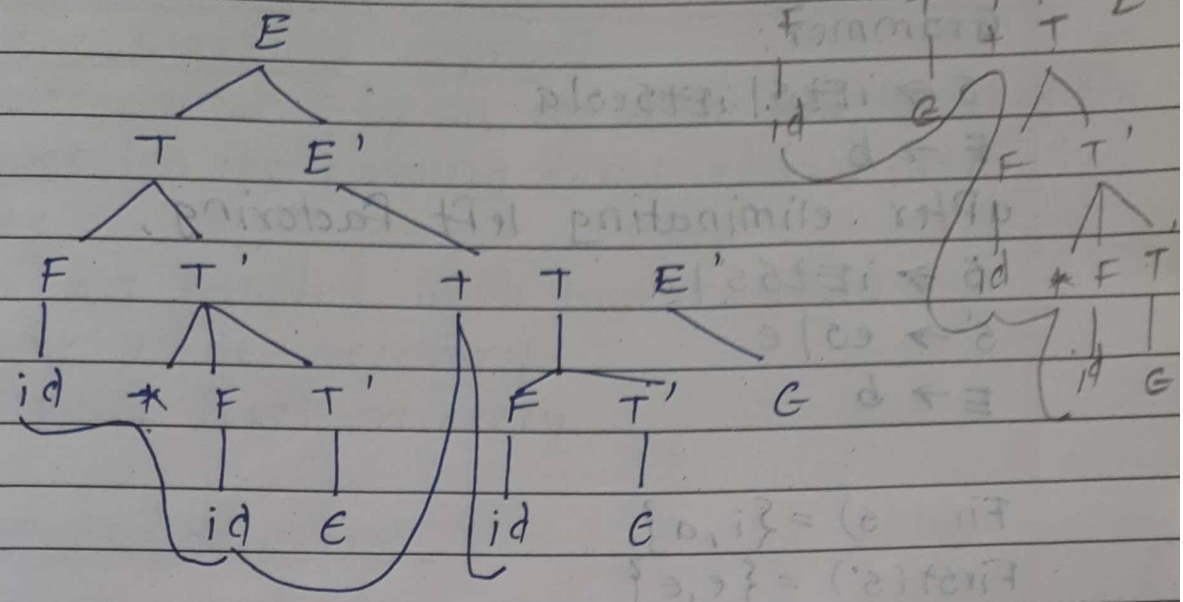
	stack	Input	output
read E from stack & id from i/p & from parsing table so we get E→TE'	$E	id * id + id $	
*	$E'T	id * id + id $	E→TE'
	$E'T'F	id * id + id $	T→FT'
	$E'T'id	id * id + id $	F→id
	$E'T'	* id + id $	
	$E'T'F*	* id + id $	T'→*FT'
	$E'T'F	id + id $	
	$E'T'id	id + id $	F→id
	$E'T'	+ id $	
	$E'	+ id $	T'→ϵ
	$E'T+		E'→+TE'
	$E'T'F	id $	T→FT'
	$ E T' id	id $	F→id
*	$ E'T'	$	T'→ϵ
	$ E'	$	
	$		E'→ϵ

- $ represents the bottom of stack & E is the start symbol
- w (ie input string) followed by $
- if stack contains nonterminal (capital) element & input contains terminal symbol then refer predictive parsing table
- so E→TE' that production is return in o/p & in stack we can replace the E by its production ie E→TE' in reverse order ie E'T in stack.

id + id * id

PAGE No. | DATE

Parse Tree:



From parse Tree : id * id + id

- Non terminal on top of stack & input contains terminal symbol then we have to refere to predictive parsing table.
- E & id gives some production from table that will be written to output.
- How stack is replace with output from ~~repre~~ reverse order.
- if terminal symbol on top of stack & that match with top of input then, we pop the terminal symbol from top of stack & remove the terminal from input buffer.

→ How Tid → so in output T → FT' & in stack E'T'F

- How Fid → so F → id → so in o/p F → id & in stack pop F & put id
- How stack & input both containing terminal symbol ie id so remove from both

PAGE No.
DATE / /

* LL(1) grammer Example

Grammer:

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

$E \rightarrow b$

after eliminating left factoring,

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

First(S) = {i, a}

First(S') = {e, ϵ}

First(E) = {b}

Follow(S) = {$, e}

Follow(S') = {$, e}

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 grammer is not LL(1) grammer.

* Error recovery in predictive Parsing

(PPT)

PAGE No.
DATE / /

- An error is detected during predictive parsing when
  - The terminal on the top of the stack does not match the next input symbol.
  - When nonterminal A is on top of the stack, a is the next symbol & the parsing table entry m(A,a) is empty.
- Errors are handled with two methods
  - Panic mode recovery
  - Phrase level recovery

Panic Mode Recovery

- Skip symbols on the input until a token in a selected set of synchronizing tokens appear
  - Use FOLLOW symbols as synchronizing tokens
  - Use synch in predictive parsing table to indicate synchronizing tokens obtains from the Follow set of non-terminals.

instead of follow we can place synch word in parsing table

Rules :

1. If parse looks up entry M[A,a] & finds it blank then the input symbol a is skipped.
2. If the entry is synch then the nonterminal on top of stack is popped in an attempt to resume parsing.
3. If a token on the top of the stack does not match the input symbol, then we pop the token from the stack.

PAGE No.
DATE / /

Example :

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

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

Non Terminal	Input symbol					
	id	+	*	(	)	$
E	E → TE'			E → TE'	synch	synch
E'		E' → +TE'			E' → ϵ	E' → ϵ
T	T → FT'	synch		T → FT'	synch	synch
T'		T' → ϵ	T' → *FT'		T' → ϵ	T' → ϵ
F	F → id	synch	synch	F → (E)	synch	synch

- Follow (E) is ( ), $) so put synch in parsing table.

	Stack	Input	Action
	$E	)id * +id $	error, skip)
id E → TE' from parsing table	$E	id * + id $	id is in FIRST(E)
	$TE'	id * +id $	
	FT'E'	id * + id $	
id & id match so pop id	idT'E'	id * +id $	
	T'E'	* +id $	
* & * match so pop	*FT'E'	* +id $	
	FT'E'	+id $	Error, M(F,+) = synch
	T'E'	+id $	F has been poped
T → ∈	+TE'	+id $	
	TE'	id $	

like wise it goes continue on checking error

- Panic mode recovery does not address the important issue of error messages.

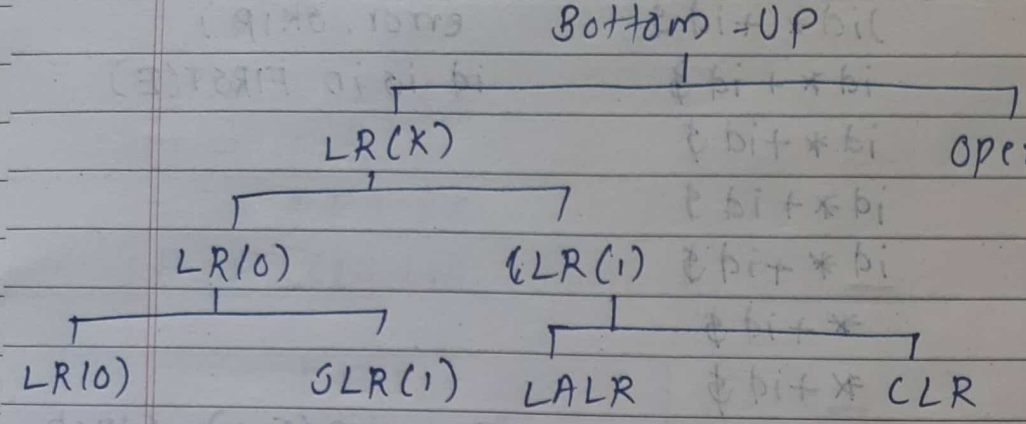
Phrase Level Recovery

- This involves, defining the blank entries in the table with pointers to some error routines which may
  - change, delete or insert symbols in the i/p
  - may also pop symbols from the stack.

Whatever may be the action, care should be taken that it should not lead the parser into infinite loop.

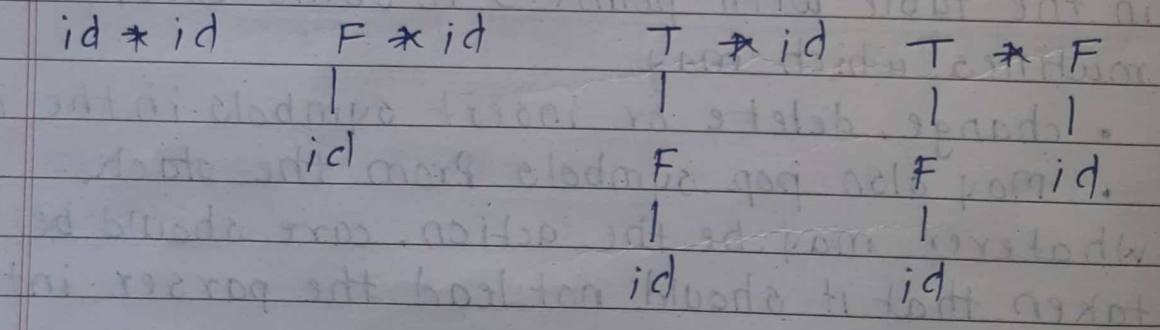
PAGE No.
DATE / /

* Bottom-Up Parsing



LR(0), LR(1) - Find the canonical item
LR(0) - (L) - left to right scan
(R) - right most derivation
LALR - look ahead LR
CLR - canonical LR

- Bottom up parse corresponds to the construction of a parse tree for an input string beginning at the leaves & working up towards the root (the top).



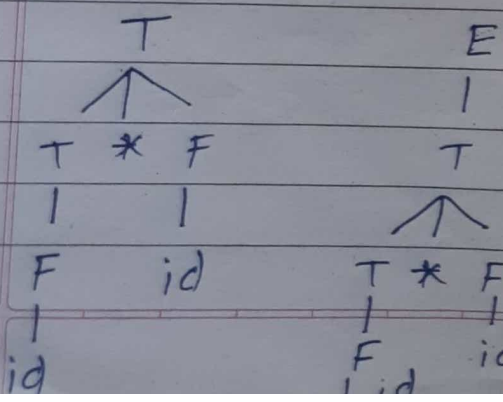


Fig. A bottom-up parse for id * id

- bottom-up parsing as the process of reducing a string w to the start symbol of the grammer. At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.
- The key decisions during bottom-up parsing are about when to reduce & about what production to apply, as the parse proceeds.

id * id, F * id, T * id, T * F, T, E

- The goal of bottom up parsing is therefore to construct a derivation in reverse. The following derivation corresponds to the parse in

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$

- This derivation is in fact a rightmost derivation.

* Handle Pruning

- Bottom up parsing during a left to right scan of the input constructs a right most derivation in reverse.
- A handle is a substring that matches the body of a production & whose reduction represents one step along the reverse of rightmost derivation.

Right sentential Form	Handle	Reducing production
$id_1 * id_2$	id_1	$F \to id$
$F * id_2$	F	$T \to F$
$T * id_2$	id_2	$F \to id$
$T * F$	$T * F$	$E \to T * F$

Examples:

1. For the grammer $S \to 0S1 \mid 01$ indicate the handle in each of the following right sentential forms:
   a) 000111
   b) 00S11

2) For the grammer $S \to SS+ \mid SS* \mid a$ indicate the handle in each of the following right sentential forms.
   a) SSS+a*+
   b) SS+a*a+
   c) aaa*a++

3) Give bottom up parses for the following input strings & grammers
   a) The input 000111 for grammer $S \to 0S1 \mid 01$
   b) The input aaa*a++ for grammer $S \to SS+ \mid SS* \mid a$

PAGE No.
DATE / /

* Shift Reduce Parsing
- It is a type of bottom up parsing

Stack	Input
$	w $
$ S	$

Actions:
1) Shift : Parser shifts 0 or more i/p symbols until handle β appears on top of stack.
2) Reduce : β is reduced to left hand side of the production
3) Accept : announces successful compilation.
4) Error : calls an error recovery routines.

$A \to \beta$ $A \to \underbrace{abc}_{\beta}$ handle abc

when handle on top of stack it must be reduced

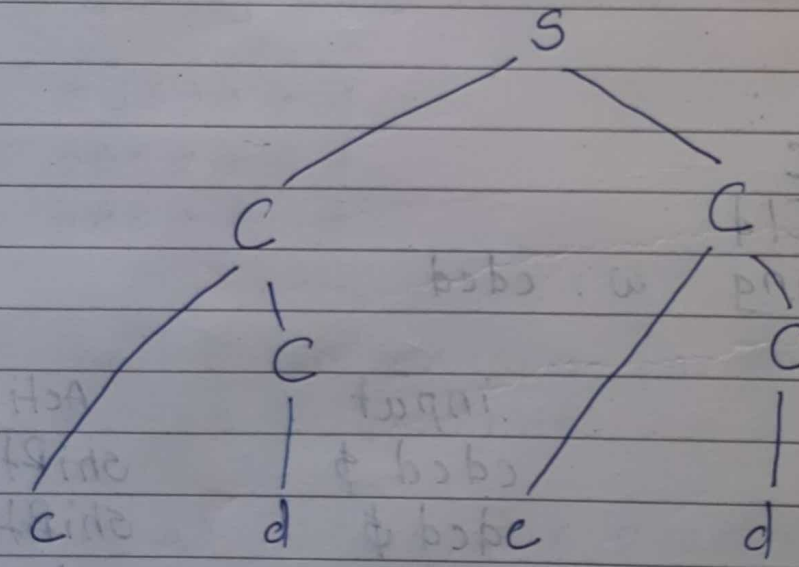
Example :
$S \to CC$
$C \to cCld$
i/p string w : cdcd

stack	input	Action
$	cdcd $	shift
$ c	dcd $	shift
$ cd	cd $	reduce by $C \to d$
$ cC	cd $	reduce by $C \to cC$
$ C	cd $	shift
$ Cc	d $	shift
$ Ccd	$	reduce by $C \to d$
$ CcC	$	reduce $C \to cC$

- c shifted on stack
- check whether c is handle or not (ie to check whethe c is the right side of any production)
- we can reduce if we will found the handle in production rule.

$ CC	$	reduce by S → CC
$ S	$	Accept

Now,
ilp is cdcd, Now we can construct the parse tree in bottom-up manner.



Examples:

Consider the grammer & perform the shift reduce parsing for the given i/p string

1) $S \rightarrow S+S$
$S \rightarrow S*S$
$S \rightarrow id$
i/p $\Rightarrow$ id + id + id

2) $E \rightarrow 2E2$
$E \rightarrow 3E3$
$E \rightarrow 4$
i/p $\rightarrow$ 32423

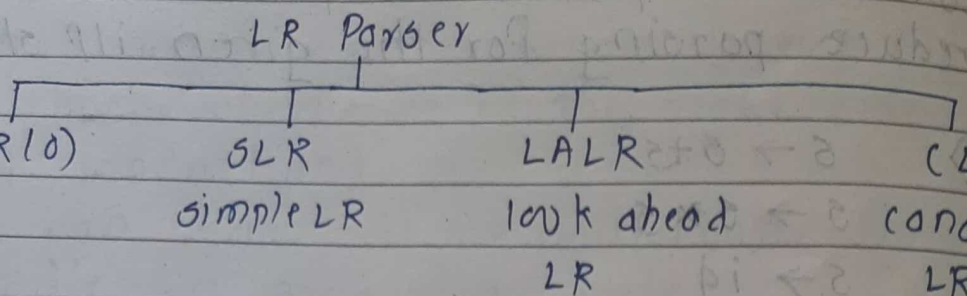
3) $S \rightarrow (L) \mid a$
$L \rightarrow L,S \mid S$
i/p $\rightarrow$ (a, (a, a))

4) $S \rightarrow TL$
$T \rightarrow int \mid Float$
$L \rightarrow L, id \mid id$
i/p $\rightarrow$ int id, id ;

5) $S \rightarrow 0S0 \mid 1S1 \mid 2$
i/p $\rightarrow$ 10201

PAGE No.
DATE / /

* LR parsers



- An efficient bottom up syntax analysis techniq
* that can be used to parse a large class of CFG is called LR(k) parsing.
L - left to right scanning
R - constructing rightmost derivation in reverse
k - no. of input symbols.
when k is omitted it is assumed to be 1.

Advantages:

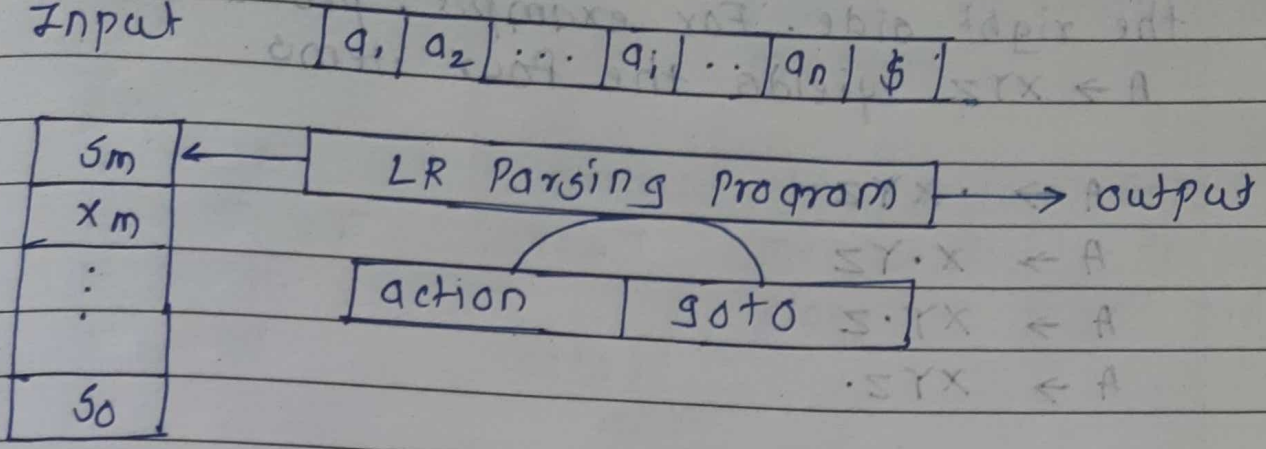
1. It recognizes virtually all programming lang constructs for which CFG can be written.
2. It is an efficient non-backtracking shift reduce parsing method.
* 3. A grammer that can be parsed using LR method is a proper superset of a grammer that can be parsed with predictive parser.
4. It detects a syntatic error as soon as possible.

Dis Advantages:

It is too much of work to construct a LR parser by hand for a programming language grammer. A specialized tool, called a LR

parser generator, is needed. Example: YACC.

* LR Parsing Algorithm



- It consist of: input, output, stack, driver program & a parsing table that has two parts (action & goto).

Action: The parsing pgm determines s_m, the state currently on top of stack & a_i, the current i/p symbol. It then constructs 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 grammer production $A \rightarrow B$,
3. accept &
4. error

Goto: The function goto takes a state & grammer symbol as arguments & produces a state.

PAGE No.
DATE / /

LR(0) items:

An LR(0) item of a grammer G is a production of G with a dot at some position of the right side. For example, production $A \to XYZ$ yields the four items.

$A \to \cdot XYZ$

$A \to X \cdot YZ$

$A \to XY \cdot Z$

$A \to XYZ \cdot$

* LR(0) Parsing Table

$E \to T + E \mid T$ (1) (2)

$T \to id$ (3)

- To make LR(0) parsing table we use LR(0) canonical items.
- How to make canonical item:

For a given grammer make it augmented grammer.

For eg.

$E' \to \cdot E$

Meaning of . ⇒ basically we are using bottom-up technic so . represents what we have seen & what we have not seen

For eg.- id + . id

means parse had check with id + & remain to parse id which is on right hand side.

Item$_0$ I_0
$E' \to \cdot E$
$E \to \cdot T+E$
$E \to \cdot T$
$T \to \cdot id$

→ now we will write the production of E from grammer & before that put a · symbol (closure property)

accept $E' \to E\cdot$ goto for E

reduce term 3 $E \to T\cdot + E$ goto T

$E \to T\cdot$

$T \to id\cdot$ goto id

+ → $E' \to E\cdot$

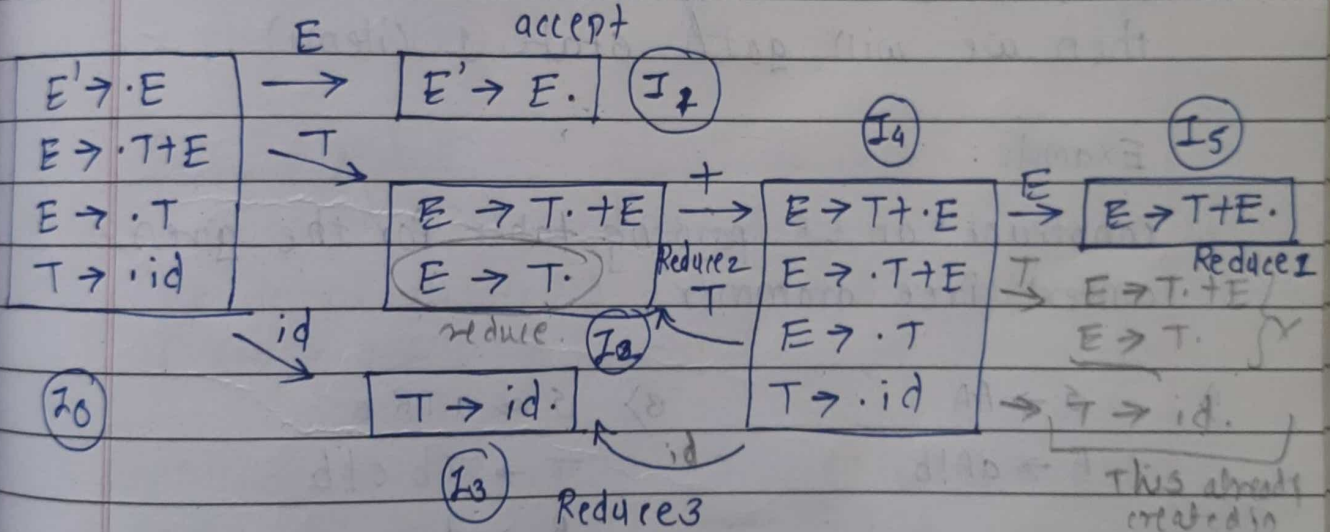
$E \to T+\cdot E$ →

$E \to \cdot T+E$

$E \to \cdot T$

$T \to \cdot id$

How perform Goto means shift · to one place on right side



where, $I_0, I_1, I_2, I_3, I_4, I_5$ ⇒ are item0, item1, item2, item3, item4, item5 to create again

Total = 6

Fig: Canonical LR(0) collection for a given grammer.

state	Action			GoTo	
	id	+	$	E	T
0	S_3			1	2
1	-	-	Accept		
2	R_2	S_4/R_2	R_2		
3	R_3	R_3	R_3		
4	S_3			5	2
5	R_1	R_1	R_1		

If in the parsing table if any of the column contains more than 1 value then it is not a LR(0) parser. (in action part only)

Two conflicts :

1) Shift - Reduce
2) Reduce - Reduce

In state 0 from item 0 if we consider more E then we will goto state 1. (item1)

Example :

Construct an LR parsing table for the given context free grammer.

1) $S \to AA$
   $A \to aA|b$

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

3) $S \to aTRe$
   $T \to Tbc|b$
   $R \to d$

4) $S \to L=R|R$
   $L \to *R|id$
   $R \to L$

PAGE No.
DATE / /

* SLR(1)

1. Find LR(0) items.
2. completing the closure.
3. compute goto (I, x), where, I is set of items & x is grammar symbol.

closure operation:

If I is a set of items for a grammer 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 $A \to \alpha \cdot B\beta$ is in closure (I) & $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ 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 $[A \to \alpha X \cdot \beta]$ such that $[A \to \alpha \cdot X\beta]$ is in I.

continue with previous example of LR(0). →
- same till the items are found.

state	id	+	$	Goto E	Goto T
0	S_3			1	2
1			Accept		
2		S_4	R_2		
3		R_3	R_3		
4	S_3			5	2
5			R_1		

- Goto for terminals is as it is from the LR(0) parsing table.
- shift & accept is as it is.
- for reduce -, r_2 written in complete eol row. In SLR we can't write reduction in complete row, we can find the follow of left hand side.

For r_2 reduction we will go to production

$E \rightarrow T+E \mid T$ (2)

$E \rightarrow .T$

so now calculate follow (E) & write r_2 reduction in follow (E).

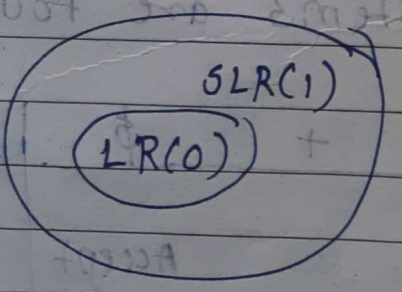
so,

follow (E) $\Rightarrow$ $

follow (T) $\Rightarrow$ +, $

so, at place of $ we can write r_2 R_3 for follow (T) $\Rightarrow$ +, $

so, each column value contains either shift or reduce so, SLR reduces the conflicts of shift Reduce. so this grammer is SLR(1).



- The G which is accepted by LR(0) is also accepted by SLR but the G which is not

accepted by LR(0) is also accepted by SLR, so the SLR is more powerful than LR.

* construct SLR parsing for the following grammer:

G : E → E+T | T
T → T*F | F
F → (E) | id

also shows the action performed for id+id*id

* show that the following grammer

S → AaAb | BbBa
A → ∈
B → ∈

is LL(1) but not SLR(1).

* G : S → SA | A
A → a

is SLR(1) but not LL(1).

* The following is an ambiguous grammer:

S → AS | b
A → SA | a

construct for this grammer its collection of sets of LR(0) items. If we try to build an LR-parsing table for the grammer, there are certain conflicting actions. what are they? suppose we tried to use the parsing table by nondeterministically choosing a possible action whenever there is a conflict. show all the possible sequences of actions on input abab.

LALR
CLR > LR(1) (add lookahead symbol)

* Canonical LR Parsing table

- CLR uses LR1 canonical item
- makes full use of the lookahead symbols.

1. First Augment the start symbol, for start symbol the lookahead symbol is added as $
2. Open the variable ie nonterminal symbol.
3. Then add first of that variable to remaining opening production so in below eg. first of S is $ so add $ as a lookhead symbol to remaining productions.

G:

$S \to aAd \,(1) \mid bBd \,(2) \mid aBe \,(3) \mid bAe \,(4)$

$A \to c \,(5)$

$B \to c \,(6)$

$S' \to .S, \$$
$S \to .aAd$
$S \to .bBd$
$S \to .aBe$
$S \to .bAe$

after .S calculate the First of .S ie $ so add $ to remaining

So, Goto (I_1, S)

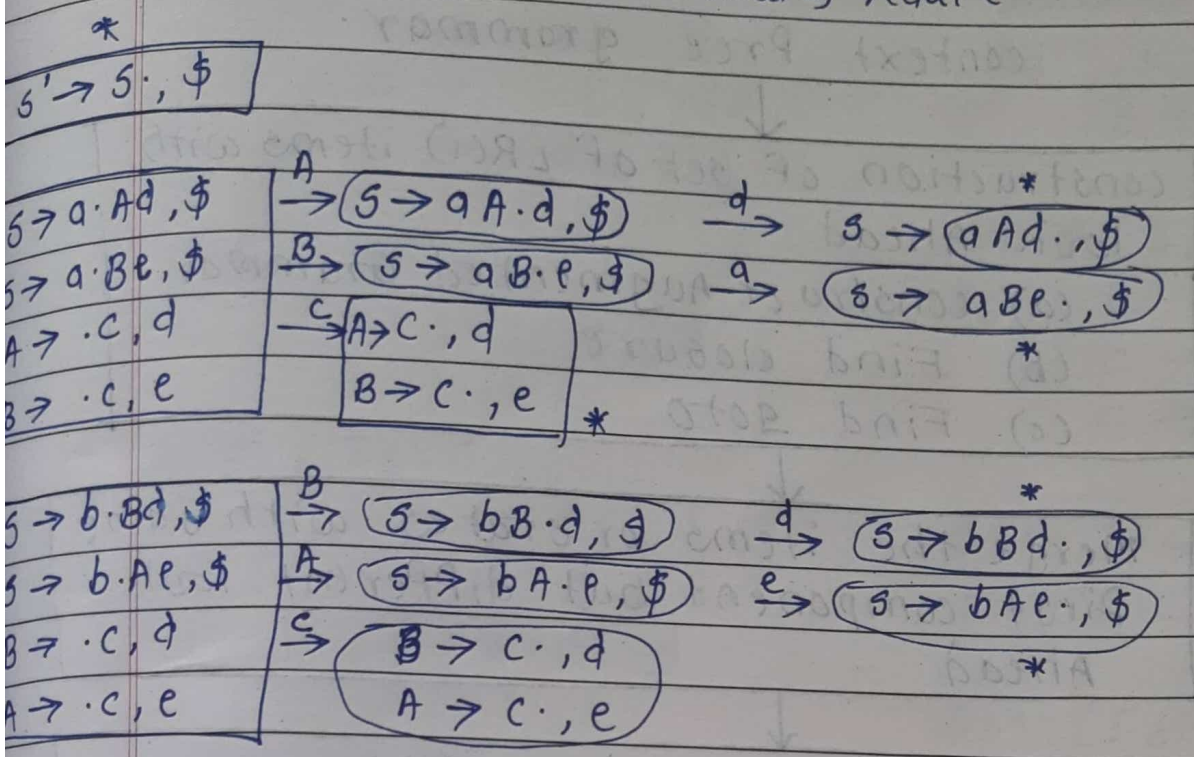
$S' \to .S, \$$	S →	$S' \to S., \$$ * Accept
$S \to .aAd, \$$	a →	$S \to a.Ad, \$$
$S \to .bBd, \$$		$S \to a.Be, \$$
$S \to .aBe, \$$		$A \to .c, d$ (First of $A \Rightarrow d$)
$S \to .bAe, \$$		$B \to .c, e$ (First of $B \Rightarrow e$)

I_1

I_1 —b→

$S \to b.Bd, \$$
$S \to b.Ae, \$$
$B \to .c, d$
$A \to .c, e$

Goto{ , } * - indicates reduce



Total ⇒ 14 items.

Parsing table of CLR ⇒ same as the construction of SLR

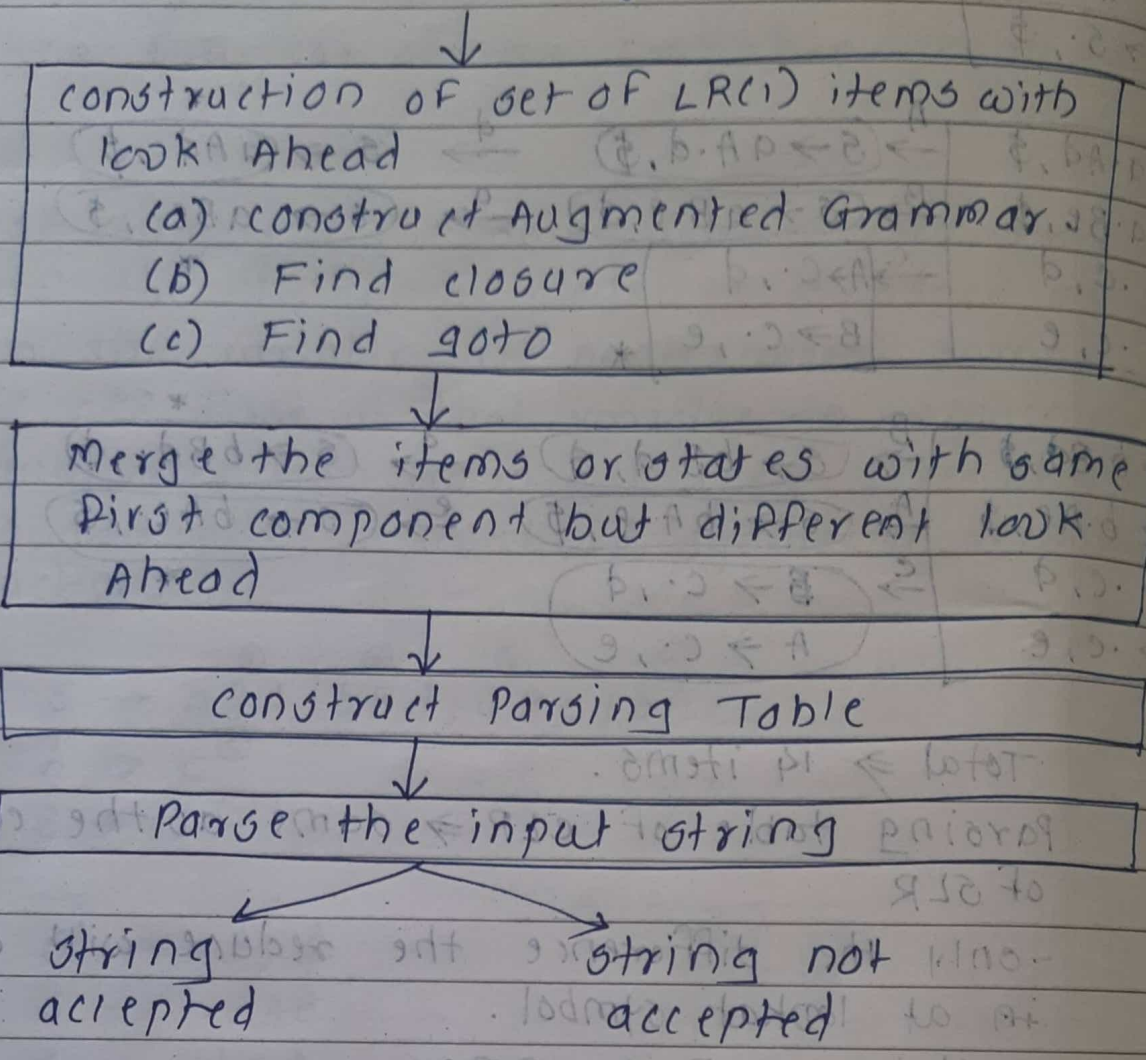
- only the difference the reduce will written at lookat symbol.
- It is CLR grammar

* LALR (lookahead LR)

- Follows LR(1) canonical items.
- LALR is less powerful than CLR.
- A grammer where the LALR parsing table has no multiply represented entries are said to be LALR(1) or LALR grammar.

PAGE No.
DATE / /

working of LALR parser



Example :

1) $S \rightarrow CC$
2) $C \rightarrow cC$
3) $C \rightarrow d$

Augmented Grammar

$S' \rightarrow .S$
$S \rightarrow .CC$
$C \rightarrow .cC$
$C \rightarrow .d$

LR(1) Items

I_0 : $S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$

Goto (I_0, S)
I_1 : $S' \to S \cdot, \$$

Goto (I_0, C)
I_2 : $S \to C \cdot C, \$$
$C \to \cdot CC, \$$
$C \to \cdot d, \$$

Goto (I_0, c)
I_3 : $C \to c \cdot C, c/d$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$

Goto (I_0, d)
I_4 : $C \to d \cdot, c/d$

Goto (I_2, C)
I_5 : ~~$C \to d \cdot, c/d$~~
$S \to CC \cdot, \$$

Goto (I_2, c)
I_6 : $C \to c \cdot C, \$$
$C \to \cdot CC, \$$
$C \to \cdot d, \$$

Goto (I_4, d)
I_7 : $C \to d \cdot, \$$

Goto (I_3, c)
I_8 : $C \to cC \cdot, c/d$

Goto (I_6, C)
I_9 : $C \to cC \cdot, \$$

- In this we check the code items are common & look ahead is different, if code items are common then we can combine the code items & merge the look ahead ~~for~~ perform union op^n with look ahead symbol).

- As we have observe the above grammer the I_3 & I_6 are common so we combine I_3 & I_6

I_{36} : $C \to c \cdot C, c|d|\$$
$C \to \cdot cC, c|d|\$$
$C \to \cdot d, c|d|\$$

I_{47} : $C \to d \cdot, c|d|\$$

I_{89} : $C \to cC \cdot, c|d|\$$

LALR parsing table :

- Instead of I_3 we can write I_{36} similarly instead of I_4 write I_{47} & I_8 write I_{89}
- From CLR parsing table we can construct LALR parsing table.

CLR Parsing table :

	c	d	$	goto S	goto C
0	S_3	S_4		1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	R_3	R_3			
5			R_1		
6	S_6	S_7			9
7			R_3		
8	R_2	R_2			
9			R_2		

- From CLR we can construct LALR parsing table
- From CLR 0 - c ⇒ S_3 so in LALR write S_{36}

LALR Parsing Table

	action			goto non terminal	
	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

All entries are single entries so its LALR grammer.

Examples:

1. Show that the following grammer
   S → Aa | bAc | dc | bda
   A → d
   is LALR(1) but not SLR(1).

2. Show that the following grammer
   S → Aa | bAc | Bc | bBa
   A → d
   B → d
   is LR(1) but not LALR(1).

PAGE No.
DATE / /

* Operator Precedance Parser

- An efficient way of constructing shift reduce parser is called operator precedance parsing
- Operator precedence parser can be constructed from a grammar called operator grammer.
- There are three disjoint precedence relations namely

  $<\cdot$ - less than
  $=$ - equal to
  $\cdot>$ - greater than

- The relations give the following meaning:

  $a <\cdot b$ - a yields precedence to b
  $a = b$ - a has same precedence as b
  $a \cdot> b$ - a takes precedence over b.

1) stack Input
   $ w$

2) Terminal on top of the stack is a symbol pointed to by input pointer is b.

   (i) $a <\cdot b$ or $a \doteq b$
   shift

   (ii) if $a \cdot> b$
   pop terminal symbol from top of stack
   pop until top stack terminal is $<\cdot$ to the terminal most recently popped.
   Reduce

- if blank entries in relation table then syntax error is occur in grammar

PAGE No. / DATE / /

S → a | ↑ | (T)
T → T, S | S

operator precedence relation table

	a	↑	(	)	,	$
a				>	>	>
↑				>	>	>
(	<	<	<	=	<	
)				>	>	>
,	<	<	<	>	>	
$	<	<	<			Accept

Stack	Input	Action
$	(a, a) $	$ < (, shift
$ (	a, a) $	(< a, shift
$ (a	, a) $	a > , , pop a once pop a now terminal symbol on top of stack is (so we check the relation bet^n top of stack & most recently pop symbol ie (< a so we stop poping until relation is < so a is handle now a is right hand side of production so, S → a so its is reduce
$ (S		reduce by S → a reduce T → S
$ (T	, a) $	(< , shift
$ (T,	a) $	, < a shift
$ (T, a	) $	a >), pop a
$ (T, S	) $	reduce S → a
$		, >) , POP T, S

- if we pop the terminal symbol & if there is non terminal symbol also then pop both ie terminal & nonterminal symbols.

$ (T	) $	(=) shift
$ (T)	$	) > $, POP T) now top of stack is (& recent 4 pop is (T) so relation put (=) - so
$		
$ S	$	reduce S → (T)

reduce by T → T,S

* Example:

Table: operator-precedence relations

	+	-	*	/	↑	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
↑	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			>	>
(	<	<	<	<	<	<	<	=	
)	>	>	>	>	>			>	>
$	<	<	<	<	<	<	<		

G:

E → E+E | E-E | E*E | E/E | E↑E | (E) | id

i/p ⇒ id + id * id

Rules:

Rule 1:

- If precedence of b is higher than precedence of a, then we define a < b.
- If precedence of b is same as precedence of a, then we define a = b
- If precedence of b is lower than precedence of a then we define a > b.

* If operators θ_1 & θ_2 are of equal precedence then make
  - $\theta_1 > \theta_2$ & $\theta_2 > \theta_1$ if operators ~~are~~ left associative
  - $\theta_1 < \theta_2$ & $\theta_2 < \theta_1$ if right associative

Rule 2:

- An identifier is always given the higher precedence than any other symbol.
- $ symbol is always given the lowest precedence

Rule 3:

- If two operators have the same precedence, then we go by checking their associativity.

Example:

1) $E \rightarrow EAE \mid id$
   $A \rightarrow + \mid *$

equivalent operator precedence grammar is

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

i/p ⇒ id + id * id

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

2) $S \rightarrow (L) \mid a$
   $L \rightarrow L, S \mid S$

i/p ⇒ (a, (a, a))

	a	(	)	,	$
a		>	>	>	>
(	<	>	>	>	>
)	<	>	>	>	>
,	<	<	>	>	>
$	<	<	<	<	

* Semantic Analysis:
Type checking
symbol table structure

Unit 4:
Syntax directed Translation & Intermediate code Generation:

Syntax directed definition, bottom-up evaluation, Intermediate representation, & intermediate code generation: Quadruples, Error detection & recovery: lexical phase errors, syntatic phase errors, semantic errors. More about translation: Array references in arithmetic expressions:

* Syntax directed Definition:

- SDD is a kind of abstract specification. It is generalization of CFG in which each grammar production $x \rightarrow a$ is associated with it a set of production rules of the form $s = f(b_1, b_2 \dots b_k)$ where s is attribute obtained from function f.
- The attribute can be a string, number, type or a memory location.
- Semantic rules are fragments of code which are embedded usually at the end of production & enclosed in curly braces. ({})

Example:

$E \rightarrow E_1 + T$ { E.val = E1, .val + T.val }

PAGE No.
DATE / /

Annotated Parse Tree -

- The Parse Tree containing the values of attributes at each node for given i/p string is called annotated or decorated parse tree.

- SDD specifies the values of attributes by associating semantic rules with grammar production.
- For example, an infix to postfix translator might have a production & rule

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E.code = E_1.code \,\|\|\, T.code \,\|\, '+'$

- Both E & T have a string valued attribute code.
- A syntax directed translation scheme embeds pgm fragments called semantic actions within production bodies, as in

$E \rightarrow E_1 + T \{ print '+' \}$

- By convention, semantic actions are enclosed within curly braces. (if curly braces occur as grammar symbols, we enclose them within single quotes.)

- A SDD is a CFG together with attributes & rules. Attributes are associated with grammar ~~rules~~ symbols & rules are associated with productions.
- If x is symbol & a is one of its attributes, then we write x.a to denote the value of a at a particular parse tree by records or objects.
- Attributes may be of any kind: numbers, types,

table references, string for instance.

* Types of Attribute

1) Inherited attribute
2) Synthesized attribute

* Synthesized attribute:

- A synthesized attribute for a nonterminal A at a parse tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined is defined only in terms of attribute values at the children of N & at N itself.
- These attributes get values from the attribute values of their child nodes.

eg.- $S \rightarrow ABC$

if S is taking values from its child nodes (A,B,C) then it is said to be synthesized attribute, as the values of ABC are synthesized to S.

* Inherited Attribute:

- An inherited attribute for a nonterminal B at a parse tree node N is defined by a semantic rule associated with the production at the parent of N.
- Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of attribute values of N's parent, N itself & N's siblings.

- Inherited attributes can take values from parent & ~~sidb~~ siblings.

eg. -

$S \rightarrow ABC$

A can get values from S, B & C. B can take values from S, A & C. likewise, C can take values from S, A & B.

Example :

	Production	Semantic Rules
1)	$L \rightarrow En$	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ digit	$F.val =$ digit.lexval

fig ① Syntax directed definition of a simple desk calculator

- In SDD, each of the nonterminals has a single synthesized attribute, called val.
- Terminal digit has a synthesized attribute lexval, which is an integer value returned by the lexical analyzer.
- An SDD that involves only synthesized attributes is called S-attributed. The above fig. has this property.
- In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head

of a production from attributes taken from the body of the production.

- An S-attributed SDD can be implemented in conjunction with an LR parser.

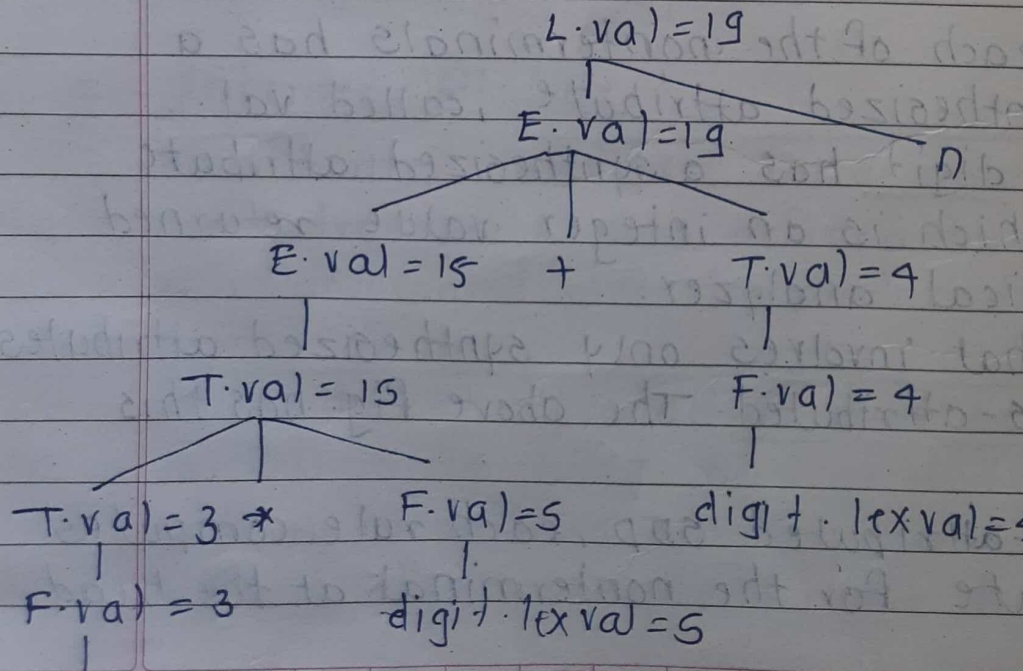
Annotated Parse Tree

- A parse tree showing the values of its attributes is called an annotated parse tree.

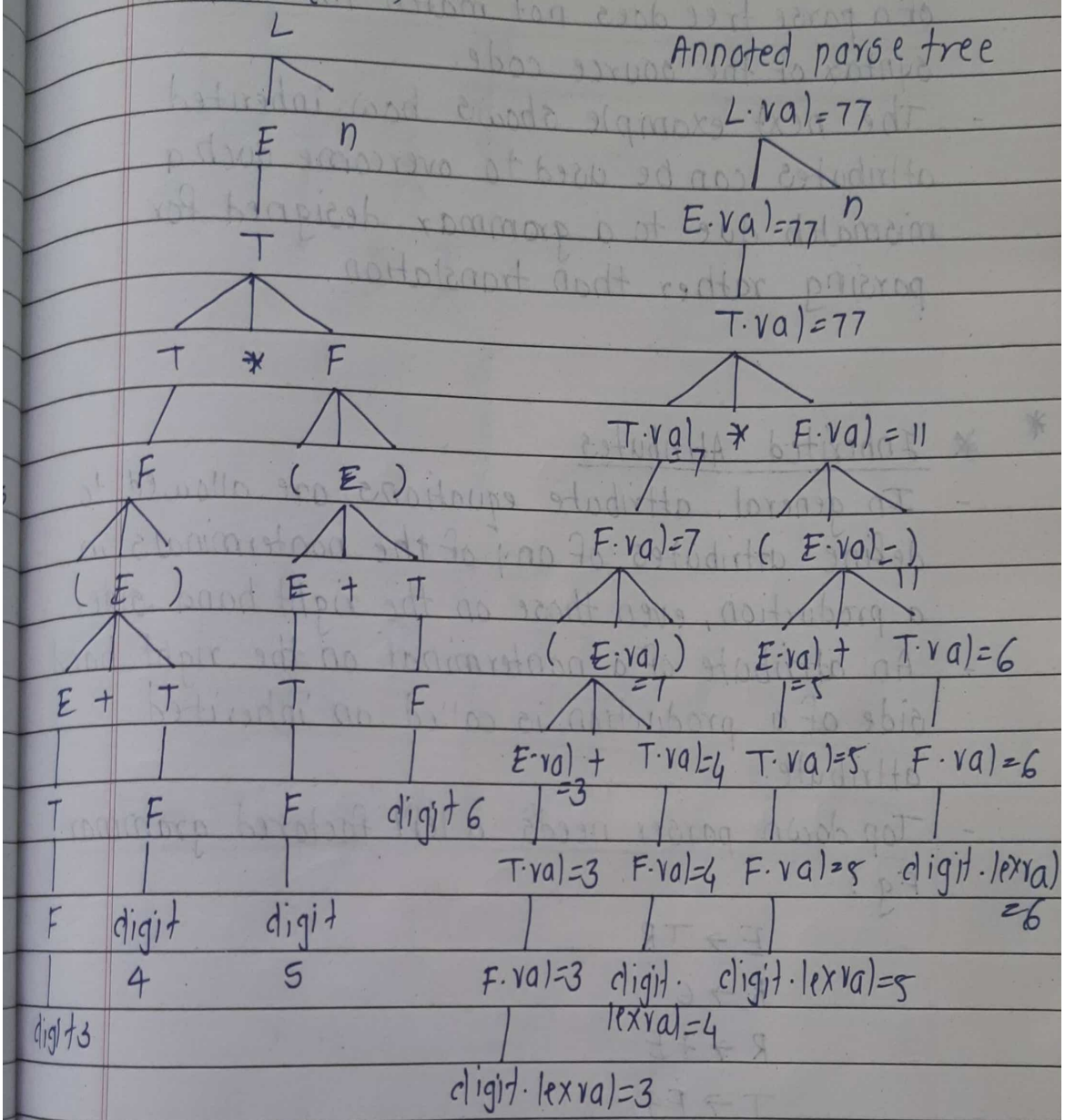
construction of annotated parse tree:

- Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends.
- For example: if all attributes are synthesized, then we must evaluate the val attributes at all of the children of a node before we can evaluate the val attribute at the node itself.
- With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of postorder traversal of parse tree.

Example: 3 * 5 + 4 n



* (3+4)*(5+6)n



Annotated parse tree

* 1*2*3*(4+5)n

* (9+8*(7+6)+5)*4n

- Inherited attributes are useful when the structure of a parse tree does not match the abstract syntax of the source code.
- The next example shows how inherited attributes can be used to overcome such a mismatch due to a grammar designed for parsing rather than translation.

* * * Inherited Attributes

- In general, attribute equations are allowed to define attributes of any of the nonterminals in a production, even those on the right hand side.
- An attribute of a nonterminal on the right hand side of a production is called an inherited attribute.
- Top down parser needs a left factored grammar.

Eg:

$E \rightarrow TR$

$R \rightarrow \epsilon$

$R \rightarrow +E$

$T \rightarrow FS$

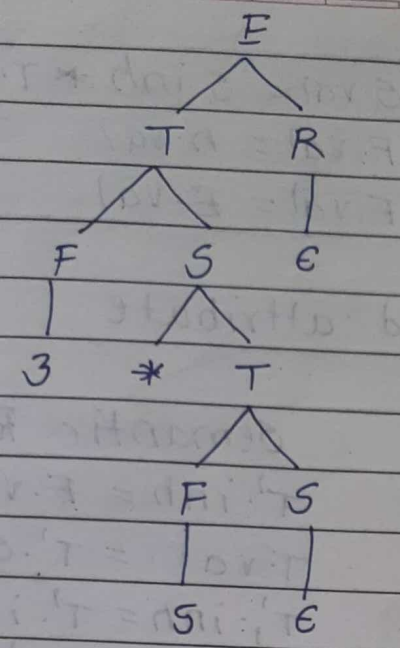
$S \rightarrow \epsilon$

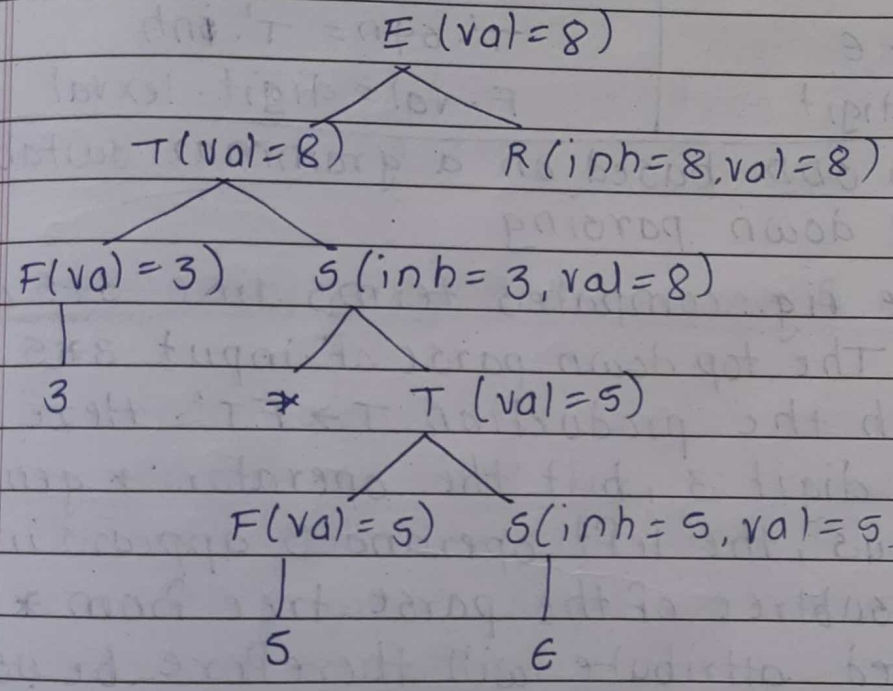
$S \rightarrow *T$

$F \rightarrow n$

$F \rightarrow (E)$

suppose that our goal is to associate an attribute of each E, T & F node that is the value of the expression.





Production	Semantic Rule
$E \rightarrow TR$	$E.val = R.val$
	$R.inh = T.val$
$R \rightarrow \epsilon$	$R.val = R.inh$
$R \rightarrow +E$	$R.val = R.inh + E.val$
$T \rightarrow FS$	$T.val = S.val$
	$S.inh = F.val$
$S \rightarrow \epsilon$	$S.val = S.inh$

$S \rightarrow *h$	$S.val = S.inh * T.val$
$F \rightarrow n$	$F.val = n.val$
$F \rightarrow (E)$	$F.val = E.val$

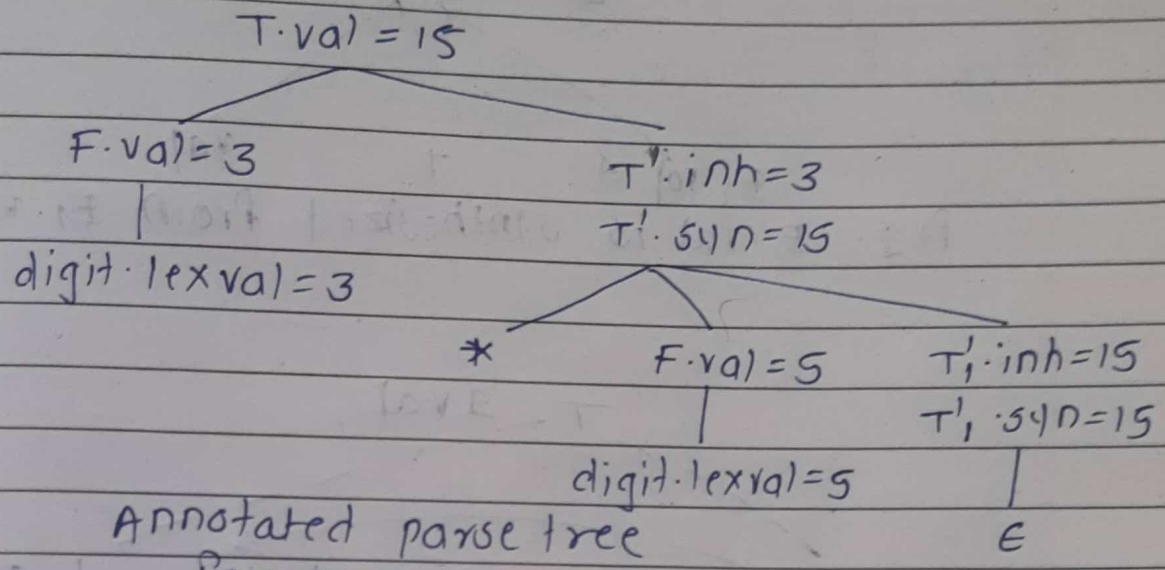
* Example : Inherited attribute

Production	Semantic Rule
1) $T \rightarrow FT'$	$T'.inh = F.val$
	$T.val = T'.syn$
2) $T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh \times F.val$
	$T'.syn = T_1'.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow digit$	$F.val = digit.lexval$

Fig: An SDD based on a grammar suitable for top down parsing

- The above fig. computes terms like 3*5 & 3*5*7. The top down parse of input 3*5 begins with the production $T \rightarrow FT'$. Here F generates digit 3, but the operator * generated by T'. Thus, the left operand 3 appears in a different subtree of the parse tree from *. An inherited attribute will therefore be used to pass the operand to the operator.
- The semantic rule are based on the idea that the left operand of the operator * is inherited. More precisely, the head T' of the production $T' \rightarrow *FT_1'$ inherites the left operand of * in the production body.
- Given a term x*y*z, the root of the subtree for *y*z inherites x. Then root of subtree

for *z inherites the value of x*y & so on. once all the factors have been accumulated, the result is passed back up the tree using synthesized attributes.



Annotated parse tree for 3*5

* Dependency Graphs (Evaluation order of SDD).

- Dependency graph are useful tool for determining an evaluation order for the attribute instances in given parse tree.
- Dependency graph helps us determine how those values can be computed.
- Dependency graph depicts the flow of information among the attribute instances in a particular parse tree.

Example:

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

- As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid.

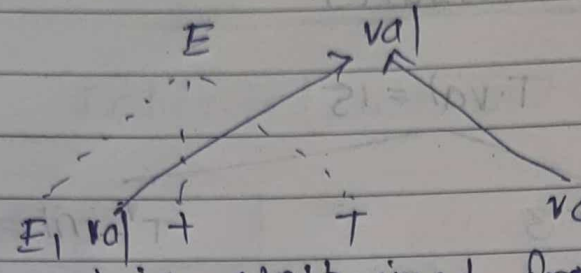


Fig. E.val is synthesized from E_1.val & T.val

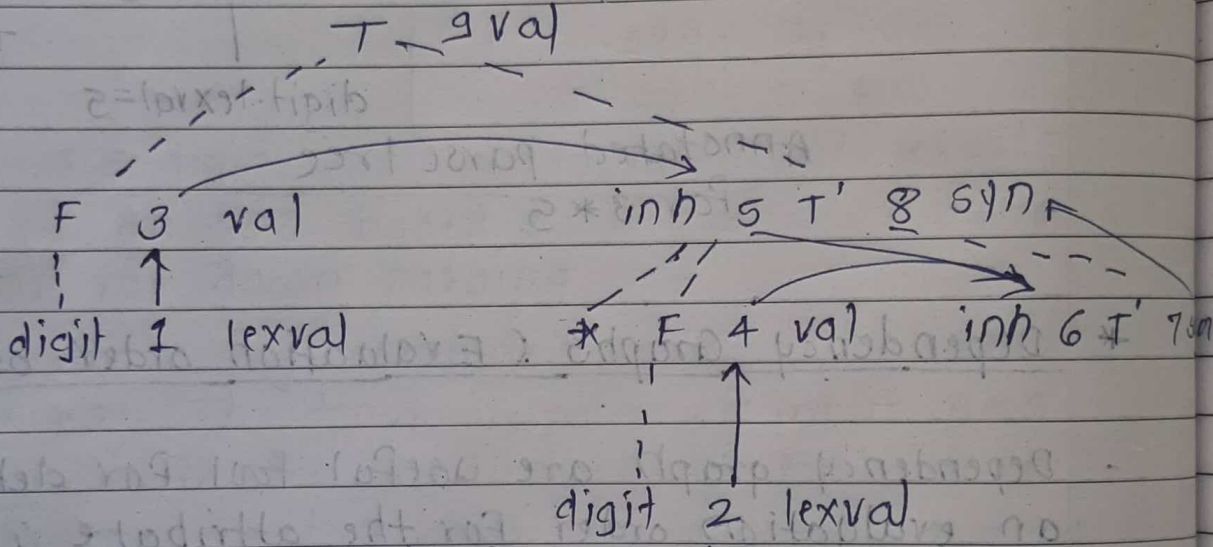


Fig: Dependency graph for annotated parse tree for 3*5

* S-Attributed SDD
- If every attribute is synthesized, then an SDD is called S-attribute SDD.
- If the value of parent nodes depends upon the value of child nodes, the S-attributed SDD is evaluated in bottom-up parsing.
- The right most place of RHS holds the semantic action.
- Fig ① is S-attributed SDD.

PAGE NO.
DATE / /

* L-attributed SDD

- If an attribute of an SDD is synthesized or inherited with some restriction on inherited attributes, it can inherit values from left siblings only. It is known as L-attributed SDD
- Attributes of this SDD are evaluated by depth first & left to right parsing methods.

Eg. -

$X \Rightarrow ABC$

$\{ B.P = X.P, B.P = A.P, B.P = C.P \}$

This is not L attributed SDD.

Eg:

Production	Semantic rule
$T \rightarrow FT'$	$T'.inh = F.val$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh \times F.val$

Example:

Any SDD containing the following production & rules cannot be L-attributed.

Production	Semantic rule
$A \rightarrow BC$	$A.s = B.b;$
	$B.i = f(C.c, A.s)$

* SDD with simple type declarations:

	Production	Semantic Rule
1)	$D \rightarrow TL$	$L.inh = T.type$
2)	$T \rightarrow int$	$T.type = integer$
3)	$T \rightarrow float$	$T.type = float$
4)	$L \rightarrow L_1, id$	$L_1.inh = L.inh$
		addType (id.entry, L.inh)
5)	$L \rightarrow id$	addType (id.entry, L.inh)

Fig. SDD for simple type declarations

- Fig. takes a simple declarations D consisting of a basic type T followed by a list L of identifiers.
- The purpose of L.inh is to pass the declared type down the list of identifiers, so that it can be added to the appropriate symbol table entries.
- Production 4 & 5 also have a rule in which a function addType is called with two arguments:

1) id.entry, a lexical value that points to a symbol table object.
2) L.inh, the type being assigned to every identifier on the list.

Example:

input string : float id1, id2, id3

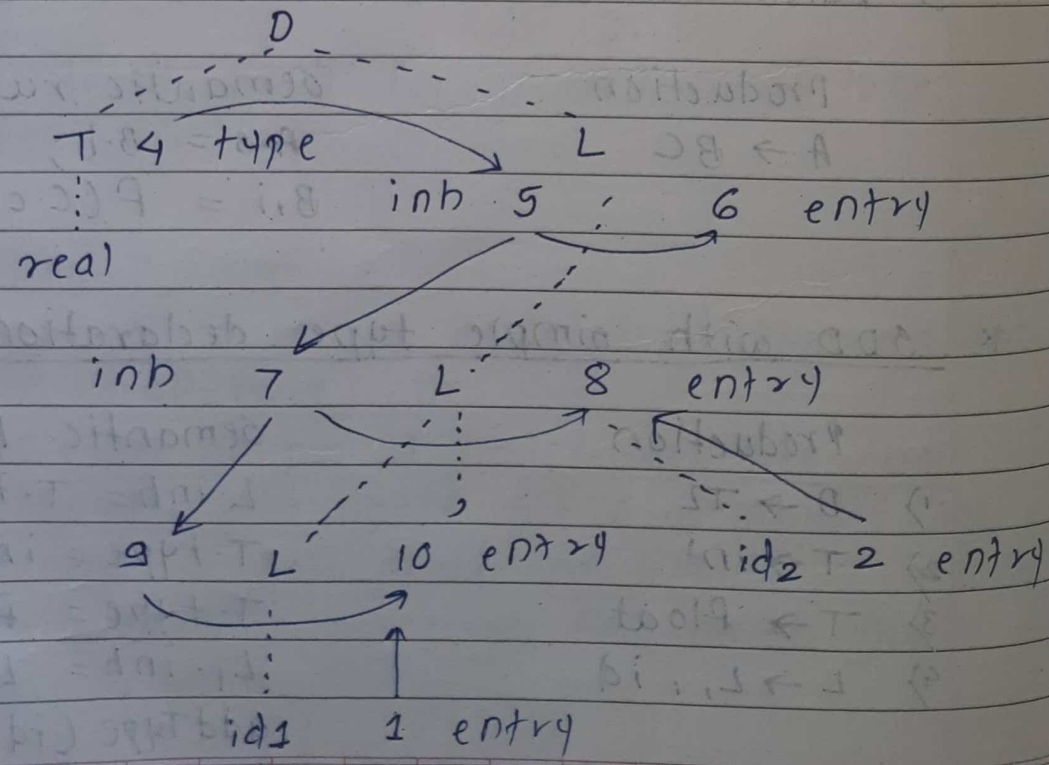


Fig. dependency graph for declaration float id1, id2, id3

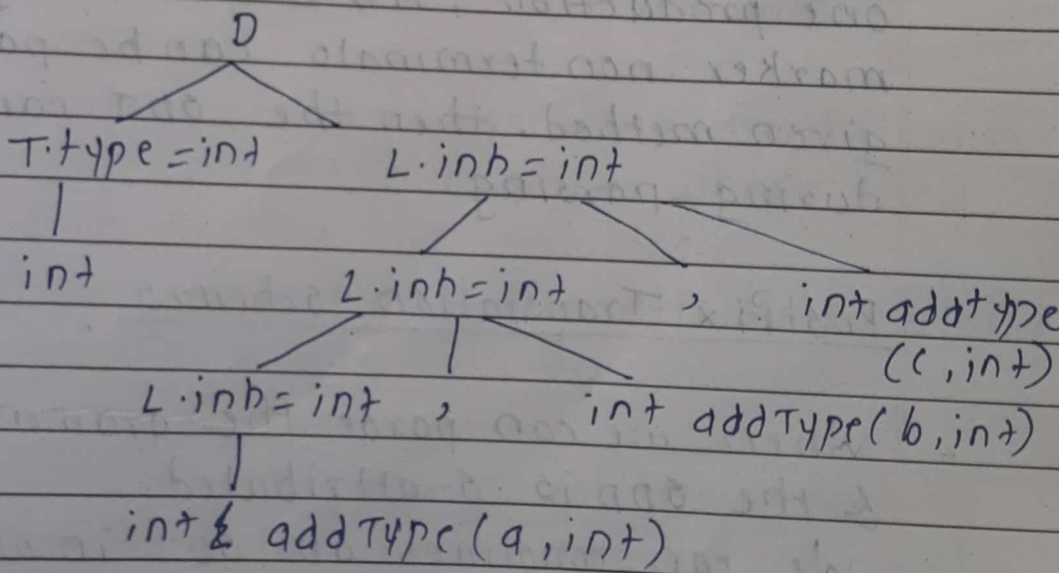
PAGE No.
DATE / /

* For the given grammar give annotated parse tree for following expression

a) int a, b, c
b) float w, x, y, z

	Production	Semantic rule
1)	D → TL	L.inh = T.type
2)	T → int	T.type = integer
3)	T → float	T.type = float
4)	L → L1 , id	L1.inh = L.inh
		addType(id.entry, L.inh)
5)	L → id	addType(id.entry, L.inh)

a)



PAGE No.
DATE / /

* Syntax Directed Translation Schemes:

- The SDT's are implemented during parsing, without building a parse tree.
- classes of SDD.
1) The underlying grammar is LR parsable & the SDD is S-attributed.
2) The underlying grammar is LL parsable & the SDD is L-attributed.

*

- SDT's that can be implemented during parsing can be characterized by introducing distinct marker nonterminals in place of each embedded action, each marker M has only one production, $M \rightarrow \epsilon$. If the grammar with marker non terminals can be parsed by a given method, then the SDT can be implemented during parsing.

Postfix Translation Schemes

*

- When we can parse the grammar bottom-up & the SDD is S-attributed.
- We can construct an SDT in which each action is placed at the end of the production & is executed along with the reduction of the body to the head of that production.
- SDT with all actions at the right ends of the production bodies are called postfix SDTs

Example:

$L \rightarrow En$	{ print (E.val); }
$E \rightarrow E_1 + T$	{ E.val = E_1.val + T.val; }
$E \rightarrow T$	{ E.val = T.val; }
$T \rightarrow T_1 * F$	{ T.val = T_1.val * F.val; }
$T \rightarrow F$	{ T.val = F.val; }
$F \rightarrow (E)$	{ F.val = E.val; }
$F \rightarrow$ digit	{ F.val = digit.lexval; }

Fig: Postfix SDT implementing the desk calculator

* Parser-stack Implementation of Postfix SDT.

- Postfix SDTs can be implemented during LR parsing by executing the actions when reductions occur.
- The attributes of each grammar symbol can be put on the stack in a place where they can be found during the reduction.
- The best plan is to place the attributes along with the grammar symbols in records on the stack itself.
- Fig shows the parser stack contains records with a field of a grammar symbol (or parser state) & below it a field for an attribute.

	X	Y	Z	state / grammar symbol
	X.x	Y.y	Z.z	synthesized attributes.

Fig: Parser stack with a field for synthesized attributes

- Stack manipulation is usually done automatically by the parser.

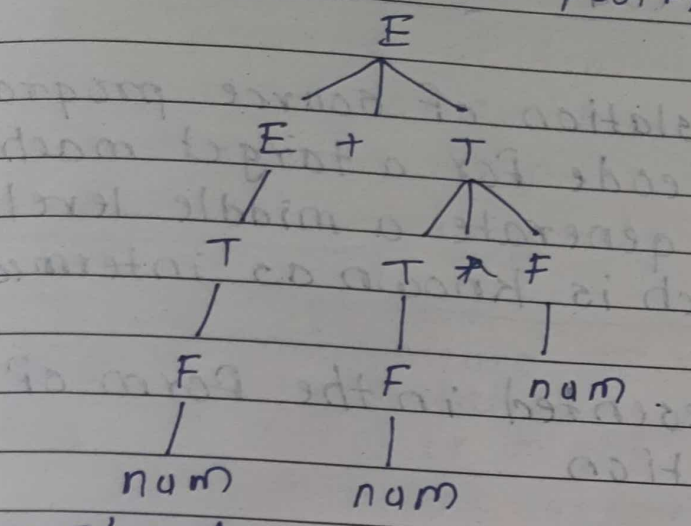
Production	Actions
$L \to En$	{ print (stack [top-1] . val); top = top - 1
$E \to E_1 + T$	{ stack [top-2] . val = stack [top-2] . val + stack [top] . val; top = top - 2; }
$E \to T$	
$T \to T_1 * F$	{ stack [top-2] . val = stack [top-2] . val x stack [top] . val; top = top - 2 ; }
$T \to F$	
$F \to (E)$	{ stack [top-2] . val = stack [top-1] . val ; top = top - 2 ; }
$F \to$ digit ;	

Fig. Implementing the desk calculator on a bottom-up parsing stack.

* SDD for infix to postfix translation

Grammar	Symantic Action
$E \to E + T$	{ print ("+") }
$E \to T$	{ }
$T \to T * F$	{ print ("*") }
$T \to F$	{ }
$F \to$ num	{ print (num . val) }

convert 2+3*4 to postfix notation



234*+

* Remaining from PPT unit: 3

Intermediate code Generation

- During the translation of source program into the object code for a target machine, a compiler may generate a middle level lang. code which is known as intermediate code.
- It can be represented in the form of:
  - postfix notation
  - syntax tree
  - directed acyclic graph (DAG)
  - three address code
  - quadruples
  - triples

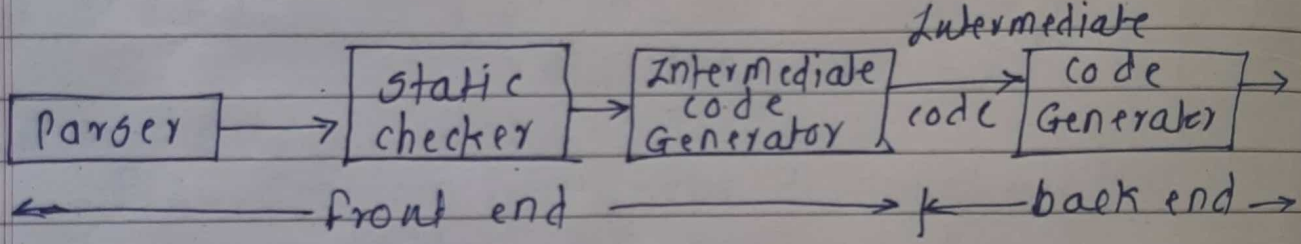


Fig. logical structure of compiler front end.

- static checking includes type checking, which ensures that operators are applied to compatible operands.
- It also includes any syntatic checks that remain after parsing.
- For example, static checking assures that a break stmt in c is enclosed within a while, for, switch stmt; an error is reported if such an enclosing statement does not exist.
- Intermediate representations includes

syntax trees & three address code.
- In the process of translating a program in a given source lang. into code for a given target m/c, a compiler may construct a sequence of intermediate representations.

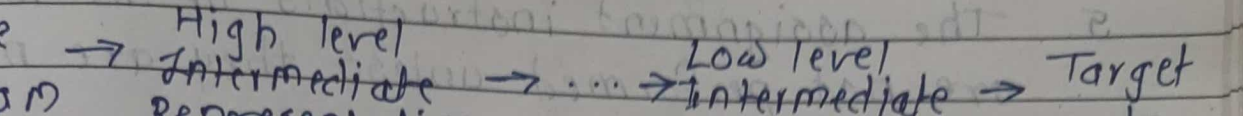


fig: A compiler might use a sequence of intermediate representations

* Directed Acyclic Graphs

- like a syntax tree for expression, a DAG has leaves corresponding to atomic operands & interior codes corresponding to operators.
- The difference is that a node N in a DAG has more than one parent if N represents a common subexpression
- Thus a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expression.
- A DAG is constructed for optimizing the basic block.
- A DAG is usually constructed using Three address code.

Construction of DAG (Rules)

1. Interior nodes always represent the operators.
2. Exterior nodes (leaf nodes) always represent

the names, identifiers or constants.

3. A check is made to find if there exists any node with the same value.
4. A new node is created only when there does not exist any node with the same value.
5. The assignment instructions of the form $x := y$ are not performed unless they are necessary.

Example:

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

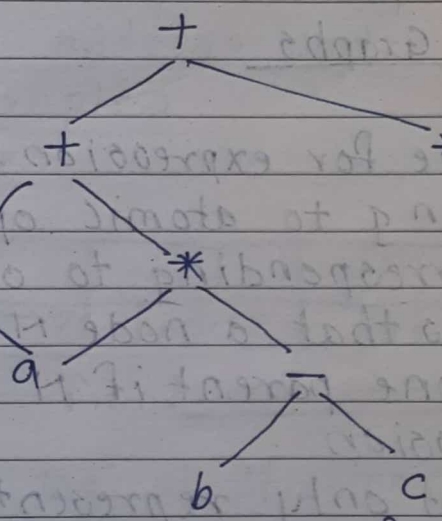


fig. DAG representation for the expression $a + a * (b - c) + (b - c) * d$

- The leaf for a has two parents, because a appears twice in the exp. The two occurrences of the common subexpression $b - c$ are represented by one node the node labeled −.
- That node has two parents, representing its two uses in the subexpression $a * (b - c)$ & $(b - c) * d$.
- Even though b & c appear twice in the complete expression, their nodes each have one parent.

since both uses are in the common subexpression b-c.

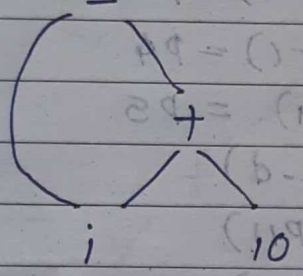
```
P1  = Leaf (id, entry-a)
P2  = Leaf (id, entry-a) = P1
P3  = Leaf (id, entry-b)
P4  = Leaf (id, entry-c)
P5  = Node ('-', P3, P4)
P6  = Node ('*', P1, P5)
P7  = Node ('+', P1, P6)
P8  = Leaf (id, entry-b) = P3
P9  = Leaf (id, entry-c) = P4
P10 = Node ('-', P3, P4) = P5
P11 = Leaf (id, entry-d)
P12 = Node ('*', P5, P11)
P13 = Node ('+', P7, P12)
```

fig. steps for constructing the DAG for above fig.

- Here node & leaf return an existing node. We assume that entry-a points to the symbol table entry for a & similarly for the other identifiers.
- When a call to leaf (id, entry-a) is repeated at step 2, the node created by the previous call to is returned, so $p_2 = p_1$. Similarly, the nodes returned at steps 8 & 9 are the same as those returned at step 3 & 4 (ie $P_8 = P_3$ & $P_9 = P_4$). Hence the node returned at step 10 must be the same at that returned at step 5 ie $P_{10} = P_5$.

PAGE No.
DATE / /

* Value Number method for constructing DAG
- The nodes of syntax tree or DAG are stored in an array of records.
- Each row of the array represents one record & therefore one node. In Fig. (b) leaves have one additional field, which holds lexical value (either symbol table pointer or a constant) & interior nodes have two additional fields indicating the left & right children



1	id	89	→ to entry for i
2	num	10	
3	+	1	2
4	=	1	3
5			

(a) DAG (b) Array

Fig. nodes of DAG for i = i + 10 allocated in an array.

Example:

Construct DAG for following expression

1) ((x+y) - ((x+y) * (x-y))) + ((x+y) * (x-y))

construct DAG & identify the value numbers for the subexpressions of the following exp., assuming + associates from left.

1) a+b+(a+b)
2) a+b+a+b
3) a+a+((a+a+a+(a+a+a+a)).

PAGE No.
DATE / /

* Three Address codes

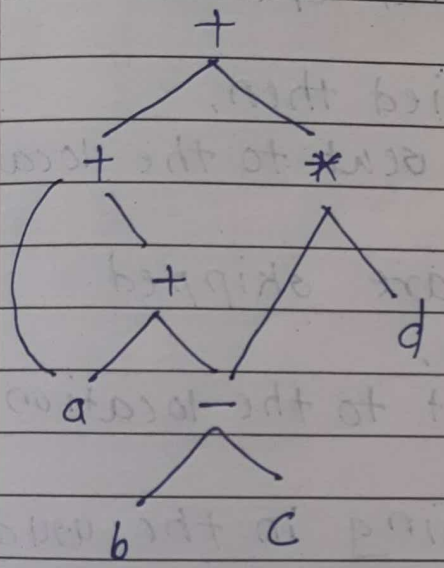
- In TAC, there is at most one operator on the right side of an instruction.

Eg: $x + y * z$

so, sequence of three address instructions

$$t_1 = y * z$$
$$t_2 = x + t_1$$

where, t_1 & t_2 are compiler generated temporary names.



$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

(a) DAG (b) Three address code

fig. a DAG & its corresponding three address code.

Instructions to Follow for construction of TAC

1. Assignment statement:

$x = y$ op z and $x =$ op y

x, y, z are operands

op is operator.

2. copy statement

$x = y$

3. conditional Jump

If x relop y goto X

Here, x & y are operands

X is the tag or label of the target statement.

relop is a relational operator.

if x relop y gets satisfied then,

- The control is directly sent to the location specified by label x.
- All stmts inbetween are skipped.

if x relop y fails then,

- The control is not sent to the location specified by label x.
- The next stmt appearing in the usual sequence is executed.

Examples:

1) $a = b + c + d$

$t_1 = b + c$

$t_2 = t_1 + d$

$a = t_2$

2) $(a \times b) + (c + d) - (a + b + c + d)$

3) If A < B then 1 else 0

⇒ 1) If (A<B) goto (4)
2) $t_1 = 0$
3) goto (5)
4) $t_1 = 1$
5) exit

4) If A<B and C<D then t=1 else t=0

⇒ 1) if(A<B) goto (3)
2) goto (4)
3) if (C< D) goto (6)
4) t = 0
5) goto (7)
6) t = 1
7) exit

* Quadruples

- Three address instructions specifies the components of each type of instruction, but it does not specify the representation of these inst. in DS.
- In compiler these instructions can be implemented as objects or as records with fields for the operator & operands.
- These such representations are called quadruples, triples & indirect triples.
- A quadruple (quad) has four fields ie op, arg1, arg2, result. op field contains an internal code for the operator.

Rules:

TAC — Quadruples
TAC — Triples
TAC — Indirect Triples

PAGE No.
DATE / /

1. Instructions with unary operators like $x = \text{minus } y$ or $x = y$ do not use arg2. Note that for a copy stmt like $x = y$, op is =, while for most other operations, the assignment operator is implied.
2. Operators like param use neither arg2 nor result.
3. conditional and unconditional jumps put the target label in result.

Eg:
$a = b * -c + b * -c$;

(-c) unary operator

$t_1 = \text{minus } c$
$t_2 = b * t_1$
$t_3 = \text{minus } c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

(a) TAC

	op	arg1	arg2	result
0	minus	c		
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a

(b) Quadruples

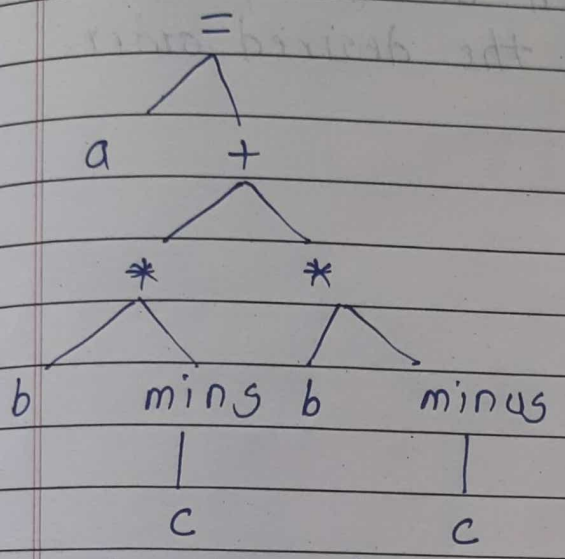
Triples

- A triple has only three fields, which we call op, arg1, & arg2. Note that the result field in above Fig is used primarily for temporary names.
- Using triples, we refer to the result of an operation x op y by its position, rather than

by an explicit temporary name.

Example:

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



	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	. . .		

(a) Syntax tree (b) Triples

Fig. representation of a + a * (b - c) + (b - c) * d

- Thus instead of the temporary t, in above fig. (Quadruples), a triple representation refers to position (0).
- Parenthesized numbers represent pointers into the triple structure itself.
- DAG & triple representations of expressions are equivalent.

In triple representation,

- References to the instructions are made.
- Temporary variables are not used.

PAGE No.
DATE / /

* Indirect triples

- It consist of listing of pointers to triples, rather than a listing of triples themselves. For example, let us used an array inst. to list pointers to triples in the desired order.

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	. .

Fig. Indirect triples representation of TAC

Examples :

1) $a + b \times c / e \uparrow f + b \times c$

TAC :

$t_1 = e \uparrow f$
$t_2 = b \times c$
$t_3 = t_2 / t_1$
$t_4 = b \times a$
$t_5 = a + t_3$
$t_6 = T_5 + t_4$

Quadruple :

	op	arg1	arg2	result
(0)	↑	e	f	t_1
(1)	x	b	c	t_2
(2)	/	t_2	t_1	t_3
(3)	x	b	a	t_4
(4)	+	a	t_3	t_5
(5)	+	t_5	t_4	t_6

Triple :

	op	arg1	arg2
(0)	↑	e	f
(1)	x	b	c
(2)	/	(1)	(0)
(3)	x	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

Indirect Triple :

	statement
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

	op	arg1	arg2
(0)	↑	e	f
(1)	×	b	e
(2)	/	(1)	(0)
(3)	×	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

Examples:

Translate the arithmetic exp. into
a) A syntax tree b) Quadruples
c) Triples d) Indirect triples

1) a + - (b + c)
2) a = b * - c + b * - c

* Types & Declarations:

Type checking : uses logical rules to reason about the behavior of a pgm at run time.
Translation Applications: From the type of a name a compiler can determine the storage that will be needed for that name at run time.

* Type Expression:

Example:

The array type int [2] [3] can be read as "array of 2 arrays of 3 integers each" & written as a type expression array (2, array (3, integer)).

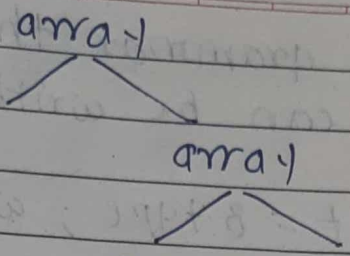


fig. type expression for int [2] [3]

* Declarations :

- The type & declarations uses a simplified grammar that declares just one name at a time.

The grammar is :

$D \rightarrow T\ id\ ;\ D \mid \epsilon$

$T \rightarrow B\ C \mid record\ '\{'\ D\ '\}'$

$B \rightarrow int \mid float$

$C \rightarrow \epsilon \mid [num]\ C$

Here, we consider storage layout as well as types. Here,

D - generates the sequence of declarations.

T - generates basic, array or record types.

B - generates basic type int & float.

C - generate string of zero or more integers, each integer surrounded by brackets.

A record - type is a sequence of declarations for the fields of the record, all surrounded by curly braces.

For the above grammar the translation scheme (SDT) can be written as,

$T \rightarrow B \; C$ { t = B.type ; w = B.width; }

B → int { B.type = integer; B.width = 4; }
B → float { B.type = float ; B.width = 8; }
C → ϵ { C.type = t ; C.width = w; }
C → [num] C_1 { array(num.value, C_1.type);
C.width = num.value × C_1.width; }

- The SDT computes types & their widths for basic & array types.
- Here type & width are synthesized attributes for each non terminal & two variables t & w to pass type & width info. From B node in a parse tree to the node for production C → ϵ
- The body of T production consist of nonterminal B, an action & nonterminal C, which appears on next line.
- The action between B & C sets t to B.type & w to B.width.
- The production for C determine whether T generates a basic type or array type.
- If C → ϵ then t becomes C.type & w becomes C.width. otherwise, C specifies an array component.
- The action for C → [num] C_1 forms C.type by applying the type constructor array to the operands num.value & C_1.type.

PAGE No.
DATE / /

* Translation of Expression

slide 24

Expression : $a = b + -c$

Three address code :

$t_1 = \text{minus } c$

$t_2 = b + t_1$

$a = t_2$

Semantic Rule for above exp. with TAC

Production	Semantic Rule
$S \rightarrow id = E ;$	S.code = E.code || gen(top.get(id.lexeme) '=' E.addr)
$E \rightarrow E1 + E2$	E.addr = new Temp()
	E.code = E1.code || E2.code || gen(E.addr '=' E1.addr '+' E2.addr)
$\| - E1$	E.addr = new Temp()
	E.code = E1.code || gen(E.addr '=' 'minus' E1.addr)
$\| (E1)$	E.addr = E1.addr
	E.code = E1.code
~~$\| id$~~ $\| id$	E.addr = top.get(id.lexeme)
	E.code = ""

Fig ①

- Attributes S.code & E.code denote the TAC for S & E respectively.
- Attribute E.addr denotes the address that will hold the value of E.
- consider the last production ie $E \rightarrow id$, when an exp. is a single identifier say x, then x itself holds the value of expression. The semantic rule for this production define E.addr to point to symbol table entry. Let top denote the current symbol table. Function

top.get retrieves the entry when it is applied to string representation id.lexeme of this instance of id. E.code is set to empty string.

- When E → (E1) the translation of E is same as that of subexpression E1. Hence, E.addr equals E1.addr & E.code equals E1.code.
- The rule E → E1+E2, generate code to compute the value of E from the values of E1 & E2. values are computed into newly generated temporary names. ie t = E1.addr + E2.addr, where t is new temporary name. E.addr is set to t.
- Here we used notation gen(x '=' y '+' z) to represent three address inst. x=y+z. expressions appearing in place of variables like x, y & z are evaluated when passed to gen, & quoted string like '=' are taken literally.

* In SDD, gen builds an inst. & returns it. In translation scheme, gen builds an inst. & incrementally emits it by putting it into the stream.

PAGE No.
DATE / /

* Increamental Translation

slide 22

- Instead of building E.code, we arrange to generate only the new three address instructions.
- In the increamental approach, gen not only constructs a three address inst., it appends the inst. to the sequence of inst.ns generated so far.
- The sequence may either be retained in memory for further processing or it may be output increamentally.
- With the increamental approach, the code attribute is not used, since there is a single sequence of instructions that is created by successive calls to gen.

```
S → id = E;   { gen(top.get(id.lexeme) '=' E.addr;) }
E → E1 + E2   { E.addr = new Temp();
                gen (E.addr '=' E1.addr '+'
                                E2.addr); }
  | - E1      { E.addr = new Temp();
                gen( E.addr '=' 'minus' E1.addr); }
  | (E1)      { E.addr = E1.addr ; }
  | id        { E.addr = top.get (id.lexeme); }
```

Fig: Generating three address code for expressions increamentally

Pg ②

* Translation of Array References

- Let nonterminal L generate an array name followed by a sequence of index expressions:

$L \rightarrow L(E) \mid id\ [E]$

$S \rightarrow id = E$; { gen (top.get(id.lexeme) '=' E.addr); }
| L = E ; { gen(L.addr.base '[' L.addr ']' '=' E.addr); }

$E \rightarrow E1 + E2$ { E.addr = new Temp();
gen (E.addr '=' E1.addr '+' E2.addr); }
| id { E.addr = top.get (id.lexmeme); }
| L { E.addr = new Temp();
gen (E.addr '=' L.array.base '[' L.addr ']'); }

$L \rightarrow id[E]$ { L.array = top.get (id.lexeme);
L.type = L.array.type.elem;
L.addr = new Temp();
gen(L.addr '=' E.addr '*' L.type.width); }

| $L_1 (E)$ { L.array = L_1.array;
L.type = L_1.type.elem;
t = new Temp();
L.addr = new Temp();
gen(t '=' E.addr '*' L.type.width);
gen(L.addr '=' L_1.addr '+' t); }

Fig : Semantic actions for array references

Non terminal L has three synthesized attributes:

1. L.addr - denotes a temporary that is used while computing the offset for the array reference by summing the terms ij * wj.
2. L.array - is a pointer to symbol table entry for the array name. The base address of the array, L.array.base is used to determine the actual l-value of an array reference after all index expressions are analyzed.
3. L.type is the type of the subarray generated by L.

- The semantic action for S → L = E; generates an indexed copy inst. to assign the value denoted by expression E to the location denoted by the array reference L. The attribute L.array gives the symbol table entry for the array.
- The arrays base address - the address of its 0th element - is given by L.array.base.
- Attribute L.addr denotes the temporary that holds the offset for the array ref. generated by L.
- The location for the array reference is therefore L.array.base [L.addr].
- The generated inst. copies the r-value from address E.addr into the location for L.

Example:

1. Add the to the translation of Fig ① rules for the following productions:

   a) $E \rightarrow E1 * E2$
   b) $E \rightarrow + E1$ (unary plus)

2. Repeat above eg. for the increamental translation of fig. ②

* Flow of control statements:

34,

Grammar:

$S \rightarrow if(B)\ S_1$
$S \rightarrow if(B)\ S_1\ else\ S_2$
$S \rightarrow while\ (B)\ S_1$

Nonterminal B represents a boolean expression & terminal S represents a statement.

Production	Semantic Rule
$P \rightarrow S$	S.next = newlabel() P.code = S.code || label(S.next)
$S \rightarrow assign$	S.code = assign.code
$S \rightarrow if(B)S1$	B.true = newlabel() B.false = S1.next = S.next S.code = B.code || label(B.true) || S1.code

cont⇒

Fig: code for if, if-else & while statements

The translation of if(B) S1 consist of B.code followed by S1.code. Within B.code are jumps based on the value of B. If B is true, control flows to the first inst. of S1.code, & if B is false, control flows to the inst. immediately following S1.code.

- The SDD for above grammar produces three address code for boolean expressions in the context of if, if-else & while statements.

contd ⇒ $S \rightarrow if(B)\ S_1\ else\ S_2$

B.true = newlabel()
B.false = newlabel()
S1.next = S2.next = S.next
S.code = B.code
|| label(B.true) || S1.code
|| gen('goto' S.next)
|| label(B.false) || S2.code

$S \rightarrow while(B)\ S_1$

begin = newlabel()
B.true = newlabel()
B.false = S.next
S1.next = begin
S.code = label(begin) || B.code
|| label(B.true) || S1.code
|| gen('goto' begin)

$S \rightarrow S_1 S_2$

S1.next = newlabel()
S2.next = S.next
S.code = S1.code || label
(S1.next) || S2.code

- We assume that newlabel() creates a new label each time it is called, & that label(L) attaches label L to the next three address inst. to be generated.

P → S ⇒

The semantic rule associated with this production initialize S.next to a new label. P.code consist of S.code followed by the new label S.next.

assign ⇒

Token assign in the production S → assign is a placeholder for assignment statements. S.code is simply assign.code.

S → if(B) S1 ⇒

create a new label B.true & attach it to the first three address inst. generated for stmt S1, Thus jumps to B.true within the code for B will go to the code for S1. By setting B.false to S.next we ensure that control will skip the code for S1 if B evaluates to false.

S → if(B) S1 else S2 ⇒

the code for the boolean expression B has jumps out of it to the first inst. of the code for S1 if B is true, & to the first inst of the code for S2 if B is false. Further, control flows from both S1 & S2 to the three address inst. immediately following the code for S - its label is given by the inherited attribute S.next. An explicit goto S.next appears after the code for S1 to skip over the code for S2. No goto is

needed after S_2, since $S_2.next$ is the same as $S.next$. ⇒

$S \rightarrow$ while (B) S_1 ⇒

is formed from $B.code$ & $S_1.code$. We use a local variable begin to hold a new label attach to the first inst. for this while stmt, which is also the first inst. for B. The inherited label $S.next$ marks the inst. that control must flow to if B is false, hence, $B.false$ is set to be $S.next$. $B.true$ is attached to the first inst. For S_1, the code for B generates a jump to this label if B is true. After the code for S_1 we place the inst. goto begin, which causes a jump back to the beginning of the code for the boolean expression. $S.next$ is set to this label begin, so jumps from within $S_1.code$ can go directly to begin

$S \rightarrow S_1 S_2$ ⇒

consist of the code for S_1 followed by the code for S_2. The semantic rule manage the labels. the first inst. after the code for S_1 is the beginning of the code for S_2, & the inst after the code for S_2 is also the inst after the code for S.

Example:

1) Repeat S while B

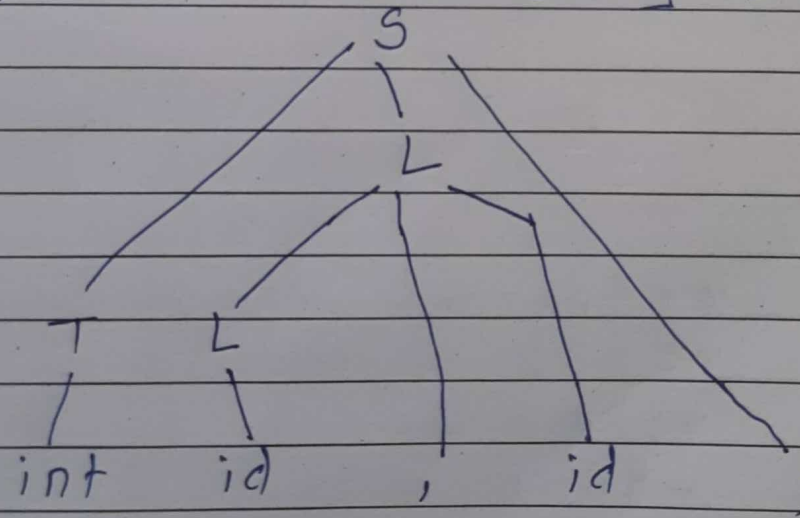
PAGE No.
DATE / /

S → TL
T → int | float
L → L, id | id

int id, id ;

stack	input	Action
$	int id, id ; $	shift
$ int	id, id ; $	Reduce int → T
$ T	id, id ; $	shift
$ T id	, id ; $	Reduce id → L
$ TL	, id ; $	shift
$ TL,	id ; $	shift
$ TL, id	; $	Reduce L, id → L
$ TL	; $	Reduce TL → S
$ S	; $	shift
$ S;	$	Accept

As ; is the end of string It is accepted.



PAGE No.
DATE / /

Ambiguity in CFG:

* Recursive decent Parsing

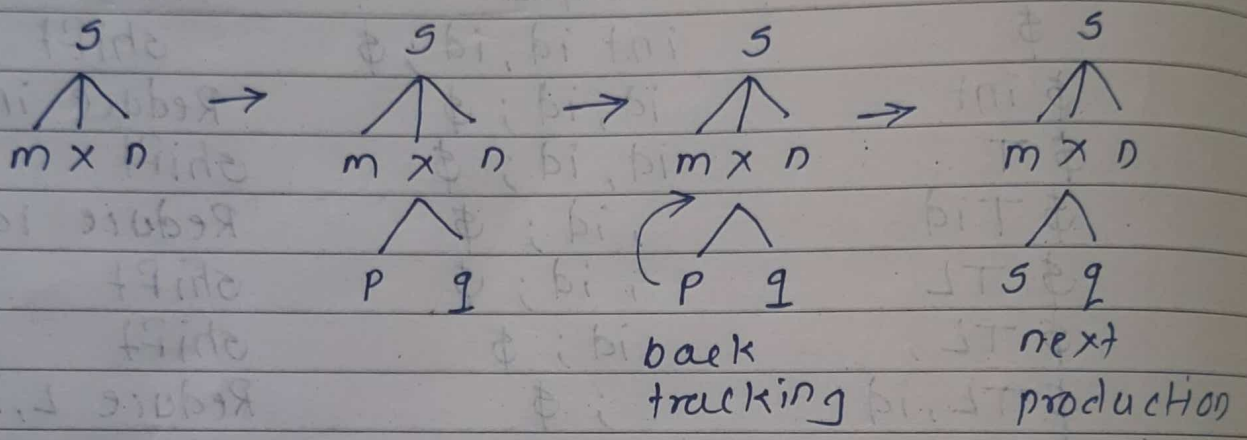
Example:

$S \rightarrow mxn \mid mzn$

$x \rightarrow pq \mid sq$

$z \rightarrow qr$

input $\Rightarrow$ msqn



PAGE No.
DATE / /

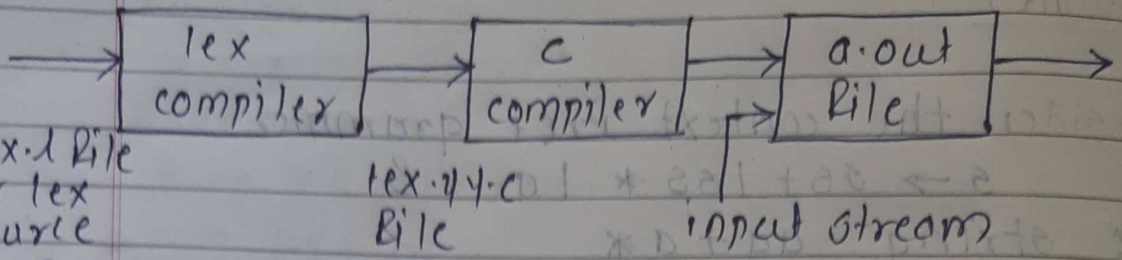
Examples

1. Consider the context free grammer
$S \rightarrow SS+ \mid SS* \mid a$ &
the string $aa+a*$
a) Give the left most derivation for the string
b) Give the rightmost derivation for the string
c) Give the parse tree for the string
d) Is the grammer ambiguous or unambiguous
Justify your answer.
e) Describe the lang. generated by this grammer.

2. Repeat above for following grammer
a) $S \rightarrow 0S1 \mid 01$ with string 000111
b) $S \rightarrow +SS \mid *SS \mid a$ +*aaa
c) $S \rightarrow S(S)S \mid \epsilon$ (()()).
d) $S \rightarrow S+S \mid SS \mid (S) \mid S* \mid a$ (a+a)*a
e) $S \rightarrow (L) \mid a$ & $L \rightarrow L,S \mid S$ ((a,a),a,(a))
f) $\rightarrow aSbS \mid bSaS \mid \epsilon$ aabbab.

1) Solution
a) $aa+a*$
$S \rightarrow SS*$
$\rightarrow$ ~~$Sa*$~~ $\rightarrow SS+S*$
$\rightarrow$ ~~$SS+a*$~~ $\rightarrow aS+S*$
$\rightarrow$ $\rightarrow aa+S*$
$\rightarrow aa+a*$

b) $S \rightarrow SS*$
$\rightarrow Sa* \rightarrow SS+a* \rightarrow Sa+a* \rightarrow aa+a*$
a) unambiguous
e) The set of all postfix exp.r consist of additions multiplicat



Program structure

1) Definition section -

```
%{
```

PAGE No.
DATE / /

Flex (Fast Lexical Analyzer Generator)

- Lex/Flex to recognize patterns in text, the pattern must be described by a regular expression
- The input to lex/flex is a machine readable set of regular expressions.
- The input is in the form of pairs of regular expressions & c code, called rules.
- It generates as output a c source file, lex.yy.c, which defines a routine yylex().
- This file is compiled & linked with the -lfl library to produce an executable.
- Whenever the executable is run, it analyzes its input for occurences of the regular expression. Whenever it finds one, it executes the corresponding c code.

- It is a tool/computer pgm for generating lexical analyzers (scanners)
- It is used together with yacc or Bison parser generator.
- Bison produces parser from the input file provided by the user.
- The function yylex() is automatically generated by the flex when it is provided with a .l file & this yylex() function is expected by parser to call to retrieve tokens from current token stream.