SYSTEM PROGRAMMING COMPILER CONSTRUCTION

(SPCC) OR COMPILER

LAST MOMENT TUITIONS

For Full Course

www.Lastmomenttuitions.com

INDEX

- Introduction to System Programming and Compiler Construction - 3
- Assembler - 13
  - → Introduction to Assembler -13
  - → Forward Reference Problem - 20
  - → Multipass Assembler - 24
- Macros and Macro Processors - 39
- Loaders and Linkers- -52
- Editor- 78
- Compilers- 81
- Lexical Analysis- 93
- Parser (Syntax Analysis) - 108
  - → Introduction- 108
  - → Types of Parser - 110
  - → Ambiguous Grammar - 115
  - → Recursive descent parser - 117
  - → Left Recursion and Left factoring - 121
  - → First and follow- 125
  - → Predictive parser (LL1) - 138
  - → Operator Precedence Parser - 154
  - → LR parser (LR (0) and SLR(1)) - 164
  - → CLR (1) and LALR (1) - 178
- Syntax Directed Translation - 185
- Intermediate Code Generation - 199
- Code Generation - 207
- Code Optimization - 222
- Run Time Storage - 238
- Compiler-Compilers - 259
  - → LEX- 264
  - → YACC - 267
- Important University Question for Practise - 269

LAST MOMENT TUITIONS

SPCC (System Programming Compiler Construction) Basics:

Languages: Languages are used for communication with the system. There are also some human languages like English, Punjabi and Hindi those are used for communicating with the other humans. So there are so many languages developed those are used for communication with the humans. The languages those are human's own language are also known as Natural Languages because they are used for humans only. Like this way when a user wants to perform any operation then a user must have to use some program or some request. And the system will the respond to the request of the user. For making a request to the system, a user can use any programs, and the programs are the collection of statements and these statements are written into some Programming Languages. All the programming languages also developed in the form of some generations those are as explained below:

Machine Language or 1st GL: This is also called as the first generation computer languages. The machine language programs contain all the instructions in the binary form which are easily understandable to the computer system. So that it is very Last Moment Tuitions difficult to understand for the user because all the instructions are written into the form of 0 and 1. Basically this language was used for writing the machine instructions means how the registers will work and how they will be work for performing the requests of the users. So that we can also say that these languages are used for internal structure of the computer system. The main advantage is that this is very easy for the computer and this will also increase the speed of processing of the computer system. And the main limitation is that this will consume lots of time of user, if a user wants to understand any program and this is very difficult to learn. But the programs those are written into the machine languages are never need to translate into the other languages.

(Computer ko sirf machine language samjhti hai aur kuch nahi jo ki binary form me hoti hai 0's and 1's me ab yeh computer ke liye bahut

achi hai par users ke liye bahut mushkil isilye picture me aya second Generation)

Assembly Language 2 GL: The program which is written into the machine language are very difficult to understand for the user so that there is the development of the new programming language which is also known as the second generation language which is also known as the assembly language. In the assembly language there are many mnemonics those are also called as the reserve words those are easy to understand to the users and those have some specific meaning. But the words are not machine language oriented means the words are just the English language forms so that they are not easily understood by the machine and it is needed to convert the words of the assembly language into the machine language. For converting the assembly language into the machine language, the assembler is used. This converts the whole program which is written into the assembly language into the machine language. For example, if a user wants to add two numbers then he has to use some mnemonics like this ADD A, B in this ADD is the mnemonics which is used for performing the addition operation. This makes very easy for the user to understand. The main limitation is that the program which is written into the assembly language must be converted into the machine language because the program will contain some words those are in the human language form and they are not machine language form.

(apne jaise bando ko binary language bahut hard padh rahi thi so ek lazy bande ne bola bhai meko yeah nahi jamta 0 and 1 so usne nikali binary language but hame Last MomentTuitions assembly language ko machine language me convert karna padhta hai aur uske liye assembler cahiye)

assembly language ko machine language me convert karna padhta hai aur uske liye assembler cahiye)

3. 3 GL or some high level languages: Due to the development of the assembly language and the mnemonics there is also the new development of some high level languages those are used for making the programs. There are many high level programming languages has developed at that time which contains the Java, C, C++, etc. All the high level programming languages are user friendly means the syntax of theses languages is quite simple because they contains all the words those are in the form of English language but the main problem is that they are not machine oriented means the program which is written into the high level languages is not understand by the computer so that the program must be converted into the machine language and the program which is written into the human language will cause the computer in slow of processing. Typically, a Compiler is needed to convert HLL into machine Language.

(ab ek banda bolta hai yaar meko na assembly languagr bhi hard padhti hai so usne nikali high level lnguage jo ki thori bahut natural language ki tarah hai(eg:C,C++, java,etc)

MACROS:(ab ek banda bolta hai meko na high level language bhi hard padhi hai usme kabhi kabhi agar ek cheez 500 times likhni padhe toh obvious hume hard padengi so usne nikale macros matlab voh same sentence ko uper program ke. ek baar likhdo aur ek word se replace kardo ab jaha bhi voh sentence aata hai vaha sirf voh word likhne ka)

Eg:

.

.

.

A data1

A data2

A data3

.

.

A data1

A data2

A data3

.

.

.

A data1

A data2

A data3

.

.

.

(jaise app dekh sakte ho ek hi cheez 3 baar likhni pad rahi hai program me so hum macros use karenge)

Macros

INCR <- (woh word jisse hum sab replace karenge pure sentence ko)

A data1

A data2

A data3

Last Moment Tuitions

MEND

.

.

.

INCR

.

.

INCR

.

.

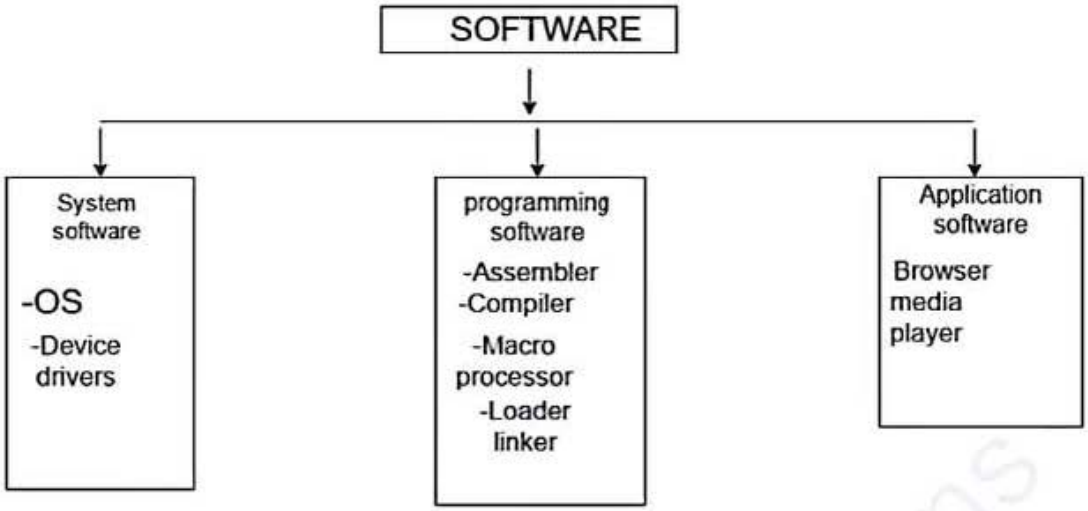
INCR

.

(jaise app dekh sakte hai abhi baar baar same sentence likhne ki jaroorat nahi sirf ek word likhna hai aur voh pura sentence usse replace ho jayenga)

Software and Its types

Last MomentTuitions



System software

Computer system divides the software systems into three major categories:

1. System Software: System software coordinates the activities and functions of hardware and software, and it controls the operations of computer hardware. It consists of variety of programs that supports the operation of a computer. System software is computer software that provides the infrastructure over which programs can operate. This makes it possible for the users to focus on an application or other problems to be solved, without needing to know the details of how machine works internally.

2. System Program Software/ Programming Software: System programs are the set of programs that are required for the effective execution of the application programs. It usually provides tools to assist programmer in writing computer programs, and software using different programming languages. The programming software tools include: ⏲ Compiler: Compiler is program that takes as input a program written in its source language (usually high level language like C++) and produces an equivalent program written in its target language (object code or machine code)

⏲ Assembler: An assembler is a language translator for the assembly language of a particular computer. Input to an assembler is an assembly language program. Output is an object program plus information that enables the loader to prepare the object program for execution.

⏲ Linkers: Both the compiler and assemblers rely on linker which a program that collects code separately compiled or assembled into a file that is directly executable. Linker also connects an object program to the code for standard library functions and to resources supplied by the OS of the computer, such as memory allocators and I/O devices.

⏲ Loaders: Compiler, assembler or linker will produce a code that is not yet completely fixed or ready to execute, but whose principal memory references are all made relative to an undetermined starting location that can be anywhere in the memory. Such code is said to be relocatable and a loader will resolve all relocatable addresses relative to a give base or starting address. In short, a loader is a program that places programs into memory and prepares them for execution. There are various loading schemes: absolute, relocating, and direct-linking. In general, the loader must load, relocate and link the object program.

⏲ Macro Processor: A macro call is an abbreviation (or name) for some code. A macro definition is a sequence of code that has a name (macro call). A macro processor is a program that substitutes and specializes macro definitions for macro calls.

⏲ Preprocessors: A pre-processor is a separate program that is called by the compiler before actual translation begins. Such pre-processor can delete comments, include other files and perform macro substitutions.

Module 01: System Software

⏲ Editors: Compilers usually accept source programs written using an editor that will produce a standard file such as ASCII file. Compilers normally are bundled together with the editors and other programs into an Interactive Development Environment or IDE.

⏲ Debuggers: Debuggers are program that can be used to determine execution errors in a compiled program. It is also packaged with a compiler in an IDE. Running a program with debugger differs from straight execution in that the debugger keeps track of most or all source code information, such as line numbers names of variable and procedures. It can also halt execution at pre-specified locations called breakpoints as well as provide information on what functions have been called and what the current values of the variables are.

⏲ Profilers: Profiler is a program that collects statistics on the behaviour of an object program during execution. Statistics that may be of interest to the programmer are the number of times each procedure is called and the percentage of execution time spent in each procedure.

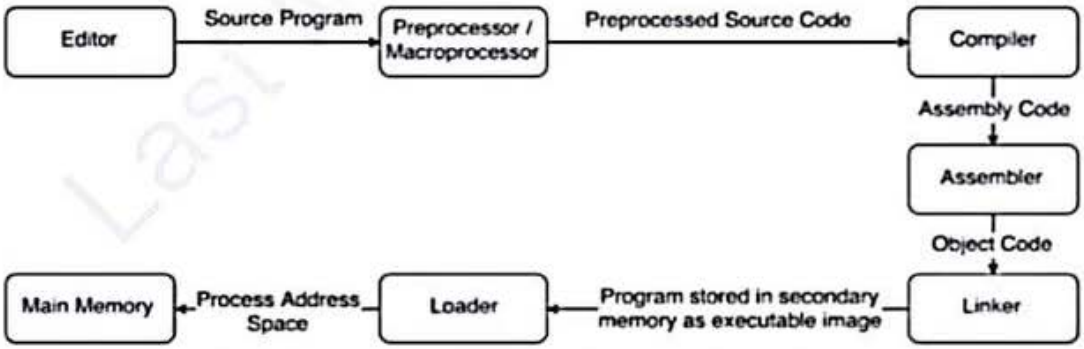
⏲ Project Managers: Modern software projects are usually so large that they are undertaken by groups of programmers rather than a single programmer. In such cases, it is important that the files being worked on by different people are coordinated and this is the job of a project of a project manager program. Project manager should coordinates the merging of separate version of the same file being produced by different programmers.

⏲ Decompiler: Decompiler is the name given to a computer program that performs the reverse operation of that of compiler i.e. it translates a file containing computer readable information to human readable. It can be used for the recovery of lost source code, computer security, interoperability and error correction. The success of decompilation depends on the amount of information present in the code being decompiled. The byte code formats used by many virtual machines often include extensive metadata and highlevel features that make decompilation.

⏲ Disassembler: Disassembler is a computer program that translates machine language into assembly language.

⏲ Interpreter: An interpreter translates high level instructions into an intermediate form. An interpreter may be program that either ▪ Executes the source code directly ▪ Translates source code into some efficient intermediate representation and executes them. E.g.: Interpreter of LISP, BASIC, etc

Lets see how the flow the works



Relative Locations of System Programming Software

3. Application Software: Application software is any tool that functions and is operated by means of a computer and allow end users to accomplish one or more specific tasks. Application Software are the most common programs that run in the foreground of the computer. They tend to perform useful tasks which are not associated with computer maintenance, system boot-up, or hardware communication. Application software is directly reliant on the Systems Software to communicate to the physical components of the computer and cannot operate without it. Application Software are the most familiar forms of software and come in a variety of types. Most often they can be accessed through the graphical user interface of the operating system being used by double-clicking on an icon. Typically, application software includes: ⏲ Industrial Automation ⏲ Office Suites ⏲ Databases ⏲ Media Players ⏲ Computer Games ⏲ Photo-editing programs ⏲ Accounting programs

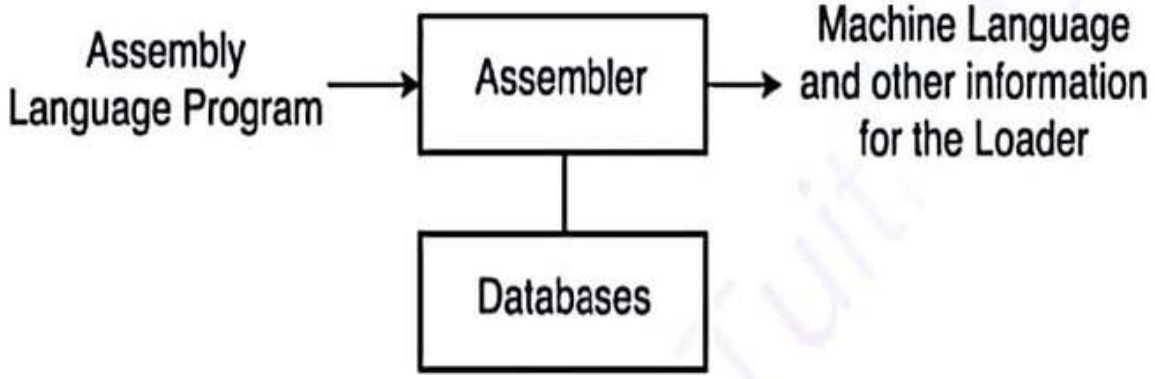
thank you so much to refer the notes and see the video on Last moment tuitions

ASSEMBLER

Introduction: An assembler is a program that accepts as input an assembly language program and produces its machine language equivalent along with information for the loader. The Assembler must also produce other information for the loader to use. For example, externally defined symbols must be noted and passed on to the loader; the assembler does not know the address of these symbols and it is up to the loader to find the programs containing them, load them into core, and place the values of these symbols in the calling program.

(assembler ek translator hai jo assembly language ko machine language me translate karta hai with some information for the loader

Loader ke liye information dena jaroori hai bcoz assembler file translate karke secondary memory me alag alag jagah store karta hai agar ek page me function use hua jisko bulaya durse page me so loader ko pata hona cahiye isliye information for loader is important)



Function of an Assembler

Advantages:

1. Assembly language allows getting the better performance out of the processor as possible (because assembly language program run faster than their higher language counterparts.)

2. It helps us to gain access to specific characteristics of the hardware that might not be possible from a higher level language.

3. It is mnemonic; e.g., we write ST instead of the big configuration 01010000 for the store instruction.

4. Addresses are symbolic, not absolute.

5. Reading is easier. 6. Introduction of data to program is easier.

Disadvantages:

1. It always requires the use of an assembler to translate a source program into an object code.

2. Assembly language are specific to a given micro-processor /computer and hence are not portable.

Elements of assembly language programming:

Structure of an Assembly Level Program:

[label:] mnemonic [operands] [; comments]

1. **Label:** is optional; it is used to assign a symbolic name for a memory location. While performing jumps within the program, we use a label for the destination location instead of the memory location for the jump.
   (label matlab symbolic name kisi memory location ka par yeh optional hota hai no compulsory dena hai toh do warna mat do **Eg: Jump,L1,L2**)

2. **Mnemonic**: is a human-readable keyword for the binary opcode of an instruction. It eliminates the need to memorize numeric operation codes. It also enables the assembler to provide helpful diagnostics, e.g. indication of misspelt operation codes.
   (mnemonic ka matlab hota hai operation to be perform Eg: ADD , SUB , MULTIPLY , DIVIDE ,etc)

3. **Operands**: are the data that the instruction operates upon. Operands may be direct values (literals) or symbolic reference to stored data (symbols). Some instructions do not have operands, as the opcode implicitly specifies the operands. The assembler performs memory binding to operands; the programmer need not know any details of the memory bindings performed by the assembler. This leads to important practical advantage during program modification.
   (operand data hai jispe hum operation perform karenge eg 2 +3 isme two and three are my operand)

4. **Comments**: are for the programmer to understand the meaning of an ALP(assembly language programming) statement in context with the remaining code. The assembler must ignore them as they are not for actual processing.
(yeh programmer ke liye hota hai ke 10 saal baad bhi code kholo toh fir bhi samjhe comment padh ke kiye kya tha)

Assembly Level Programming Terms:

Location Counter (LC), Program Counter (PC), Instruction Counter (IC): LC is a hardware memory device which denotes the location of the current instruction being executed

Literals: Literals are plain values that are not associated with any variable name. They are often used by programmers as ad-hoc values that are not needed throughout the program, but only for few statements. These literals are handled by the Assembler by using a separate data structure called Literal Table (LT).

For example, an integer variable abc is to be increased by 4 units, we write the statement as abc = abc + 4; Here, it is assumed that 4 is not needed again in the program. So, the value '4' is used as an integer literal here

(Literals are constant value jo sirf code me kahi ek do jagah use hoti hai aur unke handle karne ke liye we have literal table)

Symbols: Symbols are the variable names that hold values needed throughout the program. The assembler stores these symbols and their definition addresses in a data structure called as the Symbol Table (ST). Symbols can be used for labels, operands and procedure names. The function of assembler is to search the symbol table when a symbol is found in the program and replace it with its definition addresses (where the value is stored).

(Symbol kuch variable hote hai jo value store karte hai aur pure program me kafi baar use hote hai aur unko handle karne ke liye we have symbol table)

Procedures: Procedures are nothing but functions in ALP; they're used when a relatively large number of statements are to be used repeatedly in the program. When a procedure is called, current execution state of the program is pushed onto stack. When it returns, the execution state is popped out of the stack and the calling program is resumed. Procedure names are also handled by Assembler as symbols.

(jaise C program me function hote hai waise hi assembly language me Procedure hote hai kisi repeated kaam ko baar baar karne ke liye)

Assembly Language Statements:

An Assembly program contains three kinds of statements:

1. Imperative statements: An imperative statement indicates an action to be performed during the execution at the assembled program. Each imperative statement typically translates into a machine instruction. For example, ADD A1, B1 is an imperative statement

2. Declarative statements :The syntax of declaration statement is as follows:

[label] DS <constant>

[label] DC '<value>'

The DS (declare storage) statement reserves areas of memory and associates names with them. Consider the following DS statement:

```
A  DS     1
G  DS     200
```

The statements reserve a memory area of 1 word and associates the name A with it. The second statement reserves a block of 200 memory words.

The DC (Declare Constant) statements constructs memory words containing constants.

```
ONE  DC   '1'
```

The above statement associates the name ONE with a memory word containing the value '1'. The programmer can declare constants in different forms decimal, binary, hexadecimal, etc

3. Assembler directives : Assembler directives instruct the assembler to perform certain action during the assembly of a program. Some assembler directives are described in the following manner. They are not a part of executable instructions. So, they are not included in the final object file generated by Assembler as output.

a. label START <addr> This directive indicates that the first word of the target program generated by the assembler should be placed in the memory word with address <addr>

Example:

PG1 START 2000; Start storing the program PG1 from address 2000

b. END [<addr>]

This directive indicates the End of the source program. The optional <addr> indicates the address of the instruction where the execution of the program should begin

c. ORIGIN <addr>

This directive indicates that LC should be set to the address given by <addr> . The ORIGIN statement is useful when the target program does not consist of consecutive memory words. The ability to use an <addr> in the ORIGIN statement provides the ability to perform LC processing in a relative rather than absolute manner. It has a similar purpose as START, but since it does not include the program name, it is to be used inside the program just before the start of code segment or any other segment.

Example :

ORG 3000; Start storing current program from address 3000

d. <symbol>EQU <addr> The EQU statement defines the symbol to represent <addr> . This differs from the DC/DS statement as no LC processing is implied. Thus EQU simply associates the name <symbol> with <addr>

The EQU statement is as assembler directive used to equate a symbolic name to a value and make program more readable. Whenever a symbol is defined using EQU directive, no memory is allocated to it; only an entry is made in symbol table.

Example.

SUNDAY EQU 1

Now, whenever in the program we use the symbol SUNDAY, assembler will replace it by 1, i.e. its equated value.

e. USING base_addr, base_reg:

The USING statement is an assembler directive to indicate which register is to be used as base register and what will be its value for the base.

USING 1500, 15; Use 15 as a base register and 1500 as its value for the base.

f. DROP base_reg :

The DROP statement is an assembler directive used to drop a base register that has been allocated by a USING statement.

Example :

DROP 15; Drop register 15 as the base register.

Forward Reference Problem:

- When a symbol is referred before it is defined, it is called a forward reference.
- The function of the assembler is to replace each symbol by its machine address and if we refer to that symbol before it is defined its address is not known by the assembler such a problem is called forward reference problem.
- Forward reference is solved by making different passes (i.e. One Pass, Two Pass, and Multi-Pass) over the assembly code.
- Solution: In multi-pass approach, the assembler uses 1st pass to construct symbol table. In the 2nd pass, input file is read again along with the symbol table and an assembled object code file is generated in the output. This is a 2-Pass approach, but some complex programs may need more than two passes for completing the assembly of the program. Such an approach is called Multi pass approach.
- In single-pass approach, the assembler makes use of a special data structure to keep track of only the forward-referenced symbols. At the end of pass, this new data structure is used to first update definition addresses of forward-referenced symbols in ST and then in the program.

Forward reference Problem

(Defination sab age padenge pehle ek example lete hai aur samajte hai forward reference problem kya hota hai)

Assembler ka kaam hota hai Assembly language ko machine language me translate karne ka

Lets take an example

```
R1=3
mul R1, X ;
:
:
X   DC '2';
M   DC '7';
```

Assembler translate kar raha hai

mul :- 110100011

R_1 : 0011 → becoz R_1 pehle se 3 declare ki hai

X :→ ?

ab yeha assembler attack gaya isko nahi pata X ko kisme translate karna becoz X ko ~~[illegible]~~ age declare kiya hai par abhi toh uski value nahi pata na this is called forward refference Problem

Ab iska solution 2 Pass or Multipass hota hai usme kya karte hai hum first Pass me jitne bhi pura code hai usko scan karte hai translate nahi karte sirf scan karte hai sab symbol, literal, etc, sabko alag tables hota hai usme dalte hai aur aage jaha bhi badme voh symbol ki value declare hui hai vaha table ke baju me uski value aur jagah likte hai.

Lets take Previous example

```
R1 = 3
mul R1, X;
  :
  :
X   DC '2';
M   DC '7';
```

Pass 1 (sirf scan karke symbols ki value leni hai)

Symbol table

Srno.	Symbol	value
1)	X	2
2)	M	7

Jab maine scan shuru kiya mujhe X mila so maine 1st me dala par uski value nahi thi fir mai aage bada tab. mujhe X ki value 2 mili tab maine value ki jagah 2 dala same meko M aur uski value mili

Pass 2

ab hum translate karenge

Srno	Symbol	valu
1)	X	2
2)	m	7

```
R1 = 3
mul R1, X;
  .
  .
  .
X  DC '2';
m  DC '7';
```

chalo Translate bhuro karte hai with symbol table help

mul :- 110100011
R_1 : 0011
X : 0010 → ab yeh X me assembler atka Par usne table me dekha vaha mujhe X ki value mili jo ki 2 hai so maine translate kardiya hence Problem solve.

(Two pass or Multipass Assembler ke liye unke table yaad karna bahut jaroori hai bcoz sirf symbol table nahi hota table bahut type ke hote hai chalo dekhte hai)

NOTE: Table ki heading yaad jaroor rakhna andar kya value dalni hai utna matter nahi karta par upper ki heading jaroor yaad rakhna and ho sake toh with its size Eg: MOT me mnemonics op-code (4-bytes)

1. Machine Opcode Table(MOT):

(yeh first table hai jo pehle se pre define hota hai jitney bhi mneomonics hai voh is table me aate hai like add ,subtract, multiplication,etc hame sirf apne code me jo bhi mneomonics hai unko iss table se binary leke translate karna hai)

MOT is fixed length table (i.e. the contents of this table are not filled in or altered during the assembly process) defined by programmer as per assembly language of the underlying machine. While translation, the assembler searches for the mnemonic in MOT and replaces it with the associated binary opcode given in MOT.

← 6-bytes per entry →

Mnemonic op-code (4-bytes) (characters)	Binary op-code (1-byte) (hexadecimal)	Instruction length (2-bits) (binary)	Instruction format (3-bits) (binary)	Not used in this design (3-bits)
"A*bbb*"	5A	10	001	
"AH*bb*"	4A	10	001	
"AL*bb*"	5E	10	001	
"ALR*b*"	1E	01	000	
"AR*bb*"	1A	01	000	
...	...	...	...	
"MVC*b*"	D2	11	100	
...	...	...	...	

b~represents the character "blank"

Codes:

Instruction length

01 = 1 half-words = 2 bytes
10 = 2 half-words = 4 bytes
11 = 3 half-words = 6 bytes

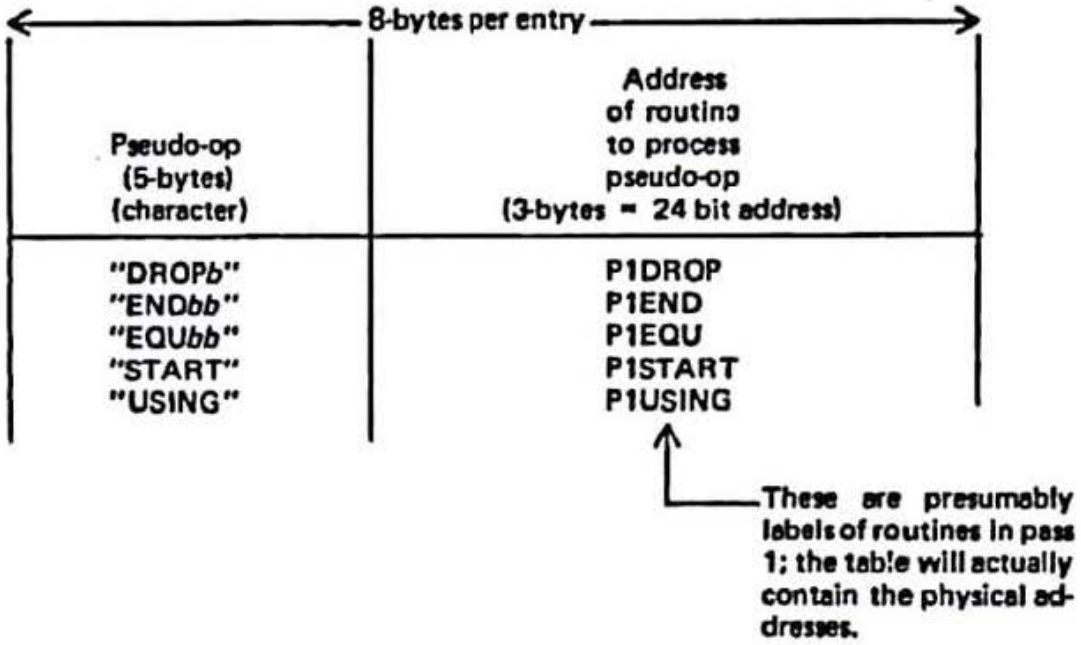
Instruction format

000 = RR
001 = RX
010 = RS
011 = SI
100 = SS

2. Pseudo-Opcode Table (POT):

(Isme saare assembler directives aate hai jo assembler ko batata hai ki konsa action lene hai konse command pe EG : End to end karo)

POT is fixed length table (i.e. the contents of this table are not filled in or altered during the assembly process) defined by programmer as per assembly language of the underlying machine. While translation, the assembler searches for the pseudo-opcode in POT and takes necessary action associated with it.



Pseudo-Op Table (POT) for pass 1(similar table for pass 2)

3. Symbol Table:

(abhi idhar hame khud table me value daalni padti hai jaise pichle example me hamne X ki value daali thi toh yeah bahut important hai table because second pass me isko check karte hai jab koi unknown ya undeclared variable milta hai)

When a symbol gets defined, assembler makes its entry into the symbol table along with its definition address. When a symbol is used in an instruction, the assembler first verifies validity of the symbol using symbol table and if validation is successful, definition addresses of symbol is read from ST and written into the output object file.

← 14-bytes per entry →

Symbol (8-bytes) (characters)	Value (4-bytes) (hexadecimal)	Length (1-byte) (hexadecimal)	Relocation (1-byte) (character)
"JOHNbbbb"	0000	01	"R"
"FOURbbbb"	000C	04	"R"
"FIVEbbbb"	0010	04	"R"
"TEMPbbbb"	0014	04	"R"

Symbol Table (ST) for pass 1 and pass 2

Symbol Table include for each entry

1. Name of the symbol
2. Symbol assembly time value
3. Length (in bytes)
4. Relative Location indicator (R/A)

Operands can be either immediate or the data located elsewhere. (Relative/ Absolute value)

4. Literal Table:

(Symbols ke alawa Literals bhi hote hai jo contant value hoti hai code me sirf ek do baar use hui hoti hai toh unka bhi table hota hai jisko literal table kehte hai)

Literal is equivalent to define a constant explicitly and assign an address label for it literal table is often organized as a hash table, using literal name or value as the key. Literal Table include for each entry

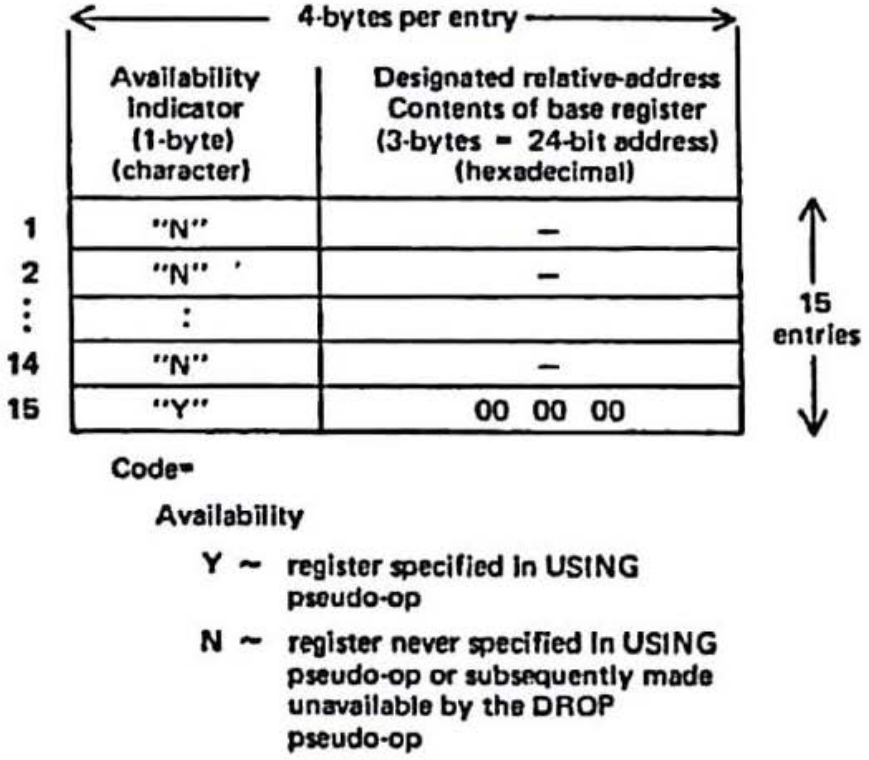
a. Name of the literal
b. Literals assembly time value
c. Length (in bytes)
d. Relative Location indicator (R/A)

Name of the Literal (1-byte)	Value of the Literal (4-byte)	Length of Literal (1-byte)	Relative Location Indicator (1-byte)

5. Base Table:

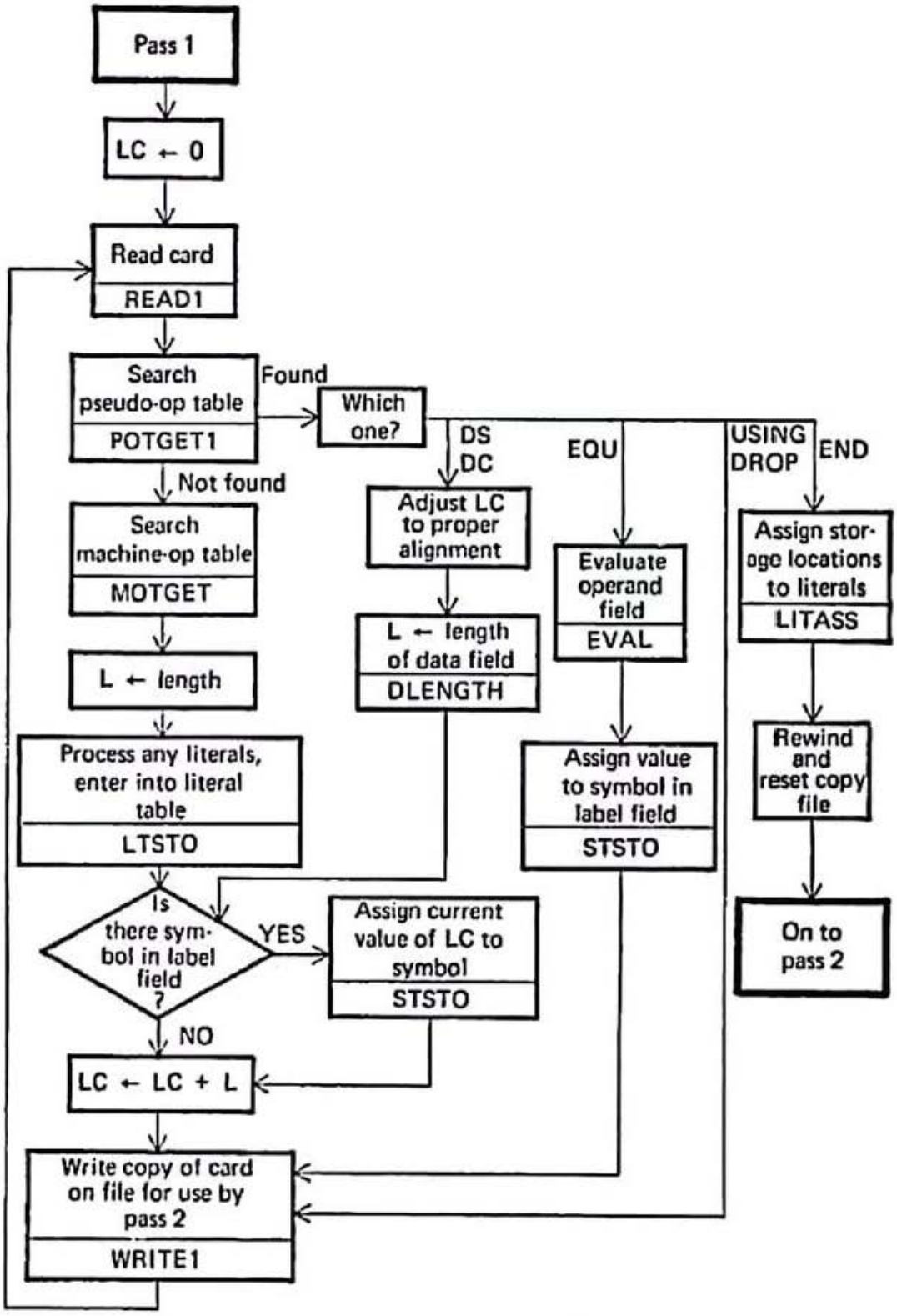
(Iss table me yeh pata chalta hai apna base register konsa hai 15register hote hai usme se sirf ek hi base register hota hai)

It is used by the assembler to generate the proper register reference in machine instructions and its correct offsets.

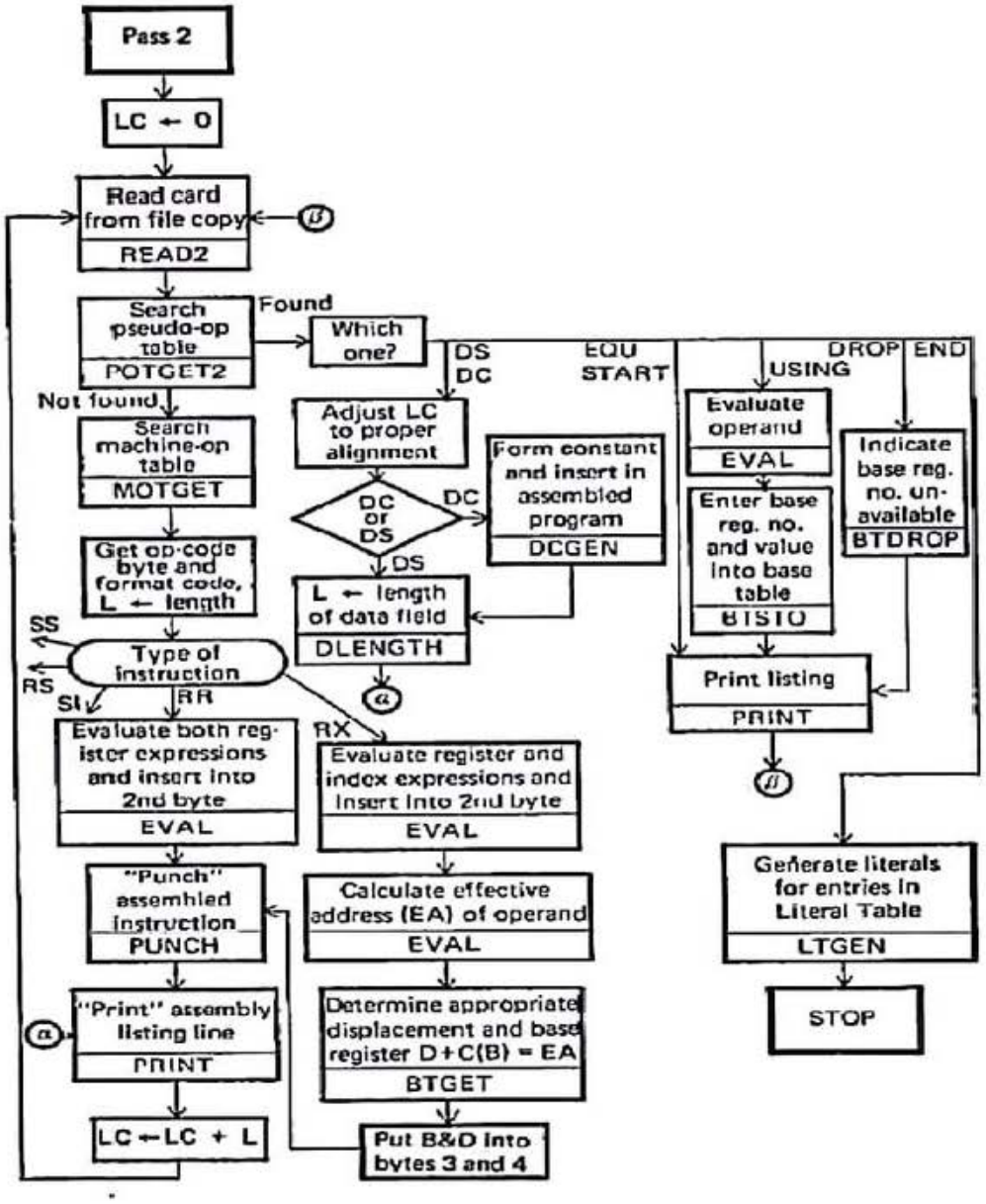


Pass 1

Note : Flowchart dekhne ke pehle video jaroor dekhlena LMT pe tabhi achese samjhenga



Pass 2



(Dekh bhai fat fat table bana lena pass 1 bhi easy hai pass 2 me ek extra part hai voh tu hata sakta hai u will get full marks jitna aye utna likhna baki kuch na kuch toh thok ke aa jana bhai best of luck for your exams)

Explanation apne hisaab se de dena jo tune pass one and pass 2 me kiya hai teacher tera fl0wchart dekhengi agar voh correct hai to explanation koi padta nahi hai par likhna jaroor

For Referrence tu yeh padh ke jaa sakta hai

Pass 1: Define Symbols:

The purpose of the first pass is to assign a location to each instruction and data defining pseudo-instruction, and thus to define values for symbols appearing in the label fields of the source program. Initially, the Location Counter (LC) is set to the first location in the program (relative address 0). Then a source statement is read.

The operation-code field is examined to determine if it is a pseudo-op; if it is not, the table of machine op-codes (MOT)is searched to find a match for the source statement's op-code field. The matched MOT entry specifies the length (2,4 or 6 bytes) of the instruction. The operand field is scanned for the presence of a literal. If anew literal is found, it is entered into the Literal Table(LT) for later processing. The label field of the source statement is then examined for the presence of asymbol. If there is a label, the symbol is saved in the Symbol Table (ST) along with the current value of the location counter.

Finally, the current value of the location counter is incremented by the length of the instruction and a copy of the source card is saved for use by pass 2. The above sequence is then repeated for the next instruction.

We now consider what must be done to process a pseudo- p. The simplest procedure occurs for USING and DROP. Pas s 1 is only concerned with pseudo-ops that define symbols (labels) or affect the location counter; USING and DROP do ne her. The assembler need only save the USING and DROP cards for pass 2.

In the case of the EQU pseudo-op during pass 1, we are concerned only with defining the symbol in the label field. This requires evaluating he expression in the operand field.

The DS and DC pseudo-ops can affect both he location counter and the definition of symbols in pass 1. The operand field must be examined to determine the number of bytes of storage required. Due to requirements for certain alignment conditions (e.g., full words must start on a byte whose address is a multiple of four) it may be necessary to adjust the location counter before defining the symbol.

When the END pseudo- p is encountered, pass 1 is terminated. Before transferring control to pass 2, there are various "housekeeping" operations that must be performed. These include assigning locations to literals that have been collected during pass I, a procedure that is very similar to that for the DC pseudo-op. Finally, conditions are reinitialized for processing by pass 2.

Pass 2: Generate Code:

After all the symbols have been defined by pass 1, it is possible to finish the assembly by processing each card and determining values for its operation code and its operand field. In addition, pass 2 must structure the generated code into the appropriate format for later processing by the loader, and print an assembly listing containing the original source and the hexadecimal equivalent of the bytes generated. The Location Counter Is initialized as in pass 1 and the processing continues as follows.

A card is read from the source file left by pass 1. As in pass 1, the operation code field is examined to determine if it is a pseudo-op; if it is not, the table of machine op-codes (MOT) is searched to find a match for the card's op-code field. The matching MOT entry specifies the length, binary op-code, and the format type of the instruction. The operand fields of the different instruction format types require somewhat different processing.

The END pseudo-op indicates the end of the source program and terminates the assembly. Various "housekeeping" tasks must now be performed. For example, code must be generated for any literals remaining in the Literal Table(LT).

For the RR-format instructions, each of the two register specification fields is evaluated. The two fields are inserted into their respective four-bit fields in the second byte of the RRinstruction.

For RX-format instructions, the register and index fields are evaluated and processed in the same way as the register specifications for RR-format instructions. The storage address operand is evaluated to generate an Effective Address(EA). After the instruction has been assembled, it is put into the necessary format for later processing by the loader. Typically, several instructions are placed on a single card. A listing line containing a copy of the source card, its assigned storage location, and its hexadecimal representation is then printed. Finally, the location counter is incremented and processing is continued with the next card. As in pass 1, each of the pseudo-ops calls for special processing. The EQU pseudo-op requires very little processing in pass 2, because symbol definition was completed in pass 1. It is necessary only to print the EQU card as part of the printed listing. The USING and DROP pseudo-ops, which were largely ignored in pass 1, require additional processing in pass 2. The operand fields of the pseudo-ops are evaluated; then the corresponding BT entry is either marked as available, if USING, or unavailable, if DROP. The base table is used extensively in pass 2 to compute the base and displacement fields for machine instructions with storage operands. The DS and DC pseudo-ops are processed essentially as in pass I. In pass 2, however, actual code must be generated for the DC pseudo-op. Depending upon the data types specified, this involves various conversions (e.g., floating point character to binary representation) and symbol evaluations (e.g., address constants). The END pseudo-op indicates the end of the source program and terminates the assembly. Various "housekeeping" tasks must now be performed. For example, code must be generated for any literals remaining in the Literal Table(LT).

Sample Program Assembly Using Two-Pass Assembler:

Sample Program			First Pass		Second Pass	
			Relative Address	Mnemonic Instruction	Relative Address	Mnemonic Instruction
PRG	START	0				
	USING	*, 15				
		1, FOUR	0	L 1, -(0, 15)	0	L 1, 16(0,15)
	A	1, FIVE	4	A 1, -(0, 15)	4	A 1, 20(0,15)
	A	1, =F'2'	8	A 1, -(0, 15)	8	A 1, 28(0, 15)
	ST	1, TEMP	12	ST 1, -(0, 15)	12	ST 1, 24(0,15)
FOUR	DC	F'4'	16	4	16	4
FIVE	DC	F'5'	20	5	20	5
TEMP	DS	1F	24		24	
	END					

Machine-Op Table:

Mnemonic	Binary Op-code	Length	Format
L	01011000	04	RX
A	01011010	04	RX
ST	01010000	04	RX

Pseudo-Op Table:

Pseudo-Op	Address
START	PA_START
USING	PA_USING
DC	PA_DC
DS	PA_DS
END	PA_END

Symbol Table:

Symbol (1-byte)	Value (4-byte)	Length (1-byte)	Relocation (1-byte)
PRG	0	01	R
FOUR	16	04	R
FIVE	20	04	R
TEMP	24	04	R

Literal Table:

Name (1-byte)	Value (4-byte)	Length (1-byte)	Relocation (1-byte)
=F'2'	28	04	R

Base Table:

	Availability Indicator	Contents of the Base Register
1	N	
. . .	. . .	
15	Y	00

2.3 Design of Single-Pass Assembler for Intel x86:

In a single pass assembler, the assembler passes over the assembly over the source exactly once and converts equivalent machine code of the whole program at one time. Due to this it requires less time but more memory space as it has a single chance to convert the program.

Two types of One Pass Assembler:

1. First type produces object code directly in memory for immediate execution e.g. **LOAD AND GO ASSEMBLER**. In this type of assembler, no object program is written out, no loader is needed. It is useful in a system with frequent program development and testing.
2. The second type produces the usual kind of object program for later execution e.g. 8088 Assembler, MASM Assembler.

Assembler Directive Statements for 8088 Assembler:

1. SEGMENT:

The SEGMENT directive is used to indicate the start of a logical segme t. Preceding the SEGMENT directive is the name you want to give the segment. F r example, the statement CODE SEGMENT indicates to the assembler the start of a log cal segment called CODE. The SEGMENT and ENDS directive are used to "bracket" a log cal segment containing code of data.

Additional terms are often added to a SEGMENT directive statement to indicate some special way in which we want the assembler o reat the segment. The statement CODE SEGMENT WORD tells the assembler that we wa he content of this segment located on the next available word (even address) when segme ts ate combined and given absolute addresses. Without this WORD addition, the segm nt will be located on the next available paragraph (16-byte) address, which might waste as much as 15 bytes of memory. The statement CODE SEGMENT PUBLIC tells the asse bler that the segment may be put together with other segments named CODE from ther asse bly modules when the modules are linked together.

2. ENDS (END SEGMENT):

This directive is used with the name of a segment to indicate the end of that logi cal segment.

3. END (END PROCEDURE):

The END directive is put after the last statement of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statements after an END directive, so you should make sure to use only one END directive at the very end of your program module. A carriage return is required after the END directive.

4. ASSUME:

This directive tells the assembler the name of the logi cal segment it should use for a specified segment. For example, ASSUME CS: CODE, tells the assembler that the instructions for a program are in a logical segment named CODE. The 8086 works directly with only 4 physical segments: a Code segment, a data segment, a stack segment, and an extra segment.

5. DB (Define Byte):

This directive is used to declare a byte type variable or to store a byte in memory location. For example,

```
NAME DB 'THOMAS';
VALUES DB '0','1','2','3';
```

6. EQU:

This directive is used to give a name to some value or to a symbol. Each time the assembler fnds the name in the program, it will replace the name with the value or symbol you given to that name. For example, FACTOR EQU 03H; this equates factor to 03H can be used as ADD AL, FACTOR; When it codes this instruction the assembler will code it as ADD AL, 03H;

The advantage of using EQU in this manner is, if FACTOR is used many no of times in a program and you want to change the value, all you had to do is change the EQU statement at beginning, it will change the rest of all.

7. ORG (ORIGIN):

As an assembler assembles a section of a data declarations or instruction statements, it uses a location counter to keep track of how many bytes it is from the start of a segment at any time. The location counter is automatically set to 0000 when assembler starts reading a egment. The ORG directive allows you to set the location counter to a desired value at any point in the program. The statement ORG 2000H tells the assembler to set the location counter to 2000H, for example.

A "$" it often used to symbolically represent the current value f he l cation counter, the $ actually represents the next available byte location where he assembler can put a data or code byte. The $ is often used in ORG statements to tell the assembler to make some change in the location counter relative to its current value. The statement ORG $ + 100 tells the assembler increment the value of the location counter by 100 from its current value.

Design:

- LC processing in this algorithm differs from LC processing in the fr st pass of a two-pass assembler in one signifc ant r sp ct.
- In Intel 8088, the unit for memory allocation is a byte; however, certain entities require their fr st byte to be aligned on specifc boundaries in the address space.
- While processing declarati ns and i perative statements, the assembler fr st aligns the address contained in the LC n the appropriate boundary. We call this action LC alignment.
- Allocation of memory for a statement is performed after LC alignment.

Format of Data base:

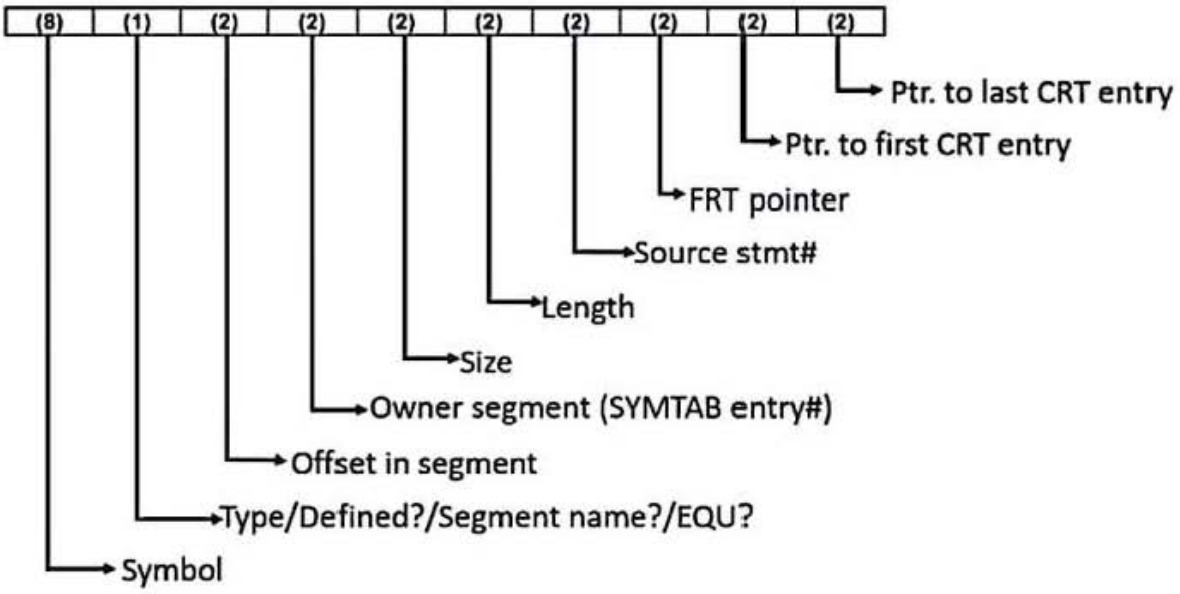
1. Machine Operation Table (MOT):

The MOT is hash organized and contains the following fields: mnemonic opcode, machine opcode, alignment/format info and routine id. The routine id field of an entry specifies the routine which bandies l.bat opcode. Alignment/format info is specific to a given routine. For example, in the table given below, the code '00H' in the Alignment/format info field in the entry for the opcode JNE implies that routine R2 should use the instruction format with self-relative displacement. If the code 'FFH' were to be used, it would imply that all instruction formats are to be supported, so the routine must decide which machine opcode to use.

Mnemonic opcode (6)	Machine opcode (2)	Alignment/format info (1)	Routine id (4)
JNE	75H	00H	R2

2. Symbol Table (SYMTAB):

It is a hash organized table and contains information about symbols defn ed and used in the source program. The contents of some important fe lds are as follows: The owner segment fe ld indicates the id of the segment in which the symbol is defne d. It contains the SYMTAB entry # of the segment name. For a non-EQU symbol the type fe ld indicates the alignment information. For an EQU symbol, the type field indicates whether the symbol is to be given a numeric value or a textual value, and the value itself is accommodated in the owner segmen1 and offset fe lds of the entry.



Symbol Table

3. Segment Register Table (SRTAB):

The segment register table (SRTAB) contains four entries, one for each segment register. Each entry shows the SYMTAB entry # of the segment whose address is contained in the segment register. Whenever an assembler encounters ASSUME statement it stores the previous SRTAB on the stack and new SRTAB is created.

SEGMENT REGISTER (1)	SEGMENT NAME (2)
00 (ES)	Extra Segment

Segment Register Table

4. Stored Segment Register Table (STSRT):

Stored Segment Register Table also known as SRTAB_ARRAY is an array of SRTABs.

Segment Register (1)	SYMTAB entry# (2)	
00 ES (Extra segment)	23	STSRT #1
..		
..		
..		
..		STSRT #2
..		
..		
..		

5. Forward Reference Table (FRT):

- It contains the address of the instruction making the forward reference.
- Kind of reference made.
- Segment registers to be used for the references.
- Statement number at which the forward reference occurred.
- Usage code is used to indicate the type of forward reference that was made to the symbol.

Pointer to Next Entry (2)	Entry # in STSRT (1)	Instruction Address (2)	Usage Code (1)	Source Statement (2)

6. Cross Reference Table (CRT):

CRT is used for producing a cross reference Directory. This directory li ts all references to a symbol in the ascending statement numbers.

Pointer (2)	Source Stmt # (2)

Being linked lists, the sizes of FRT and CRT change as a program is processed. To avoid making a fix ed commitment of storage to hese ables, both of them are organized in a single memory area that grows from the high end of memory to its low end. The assembler places the generated target code from the low e d of memory to its high end. This way, no size restrictions need to be placed on individual tabl s. Assembly of a source program would fail only if the target code overlaps with its table.

Forward references:

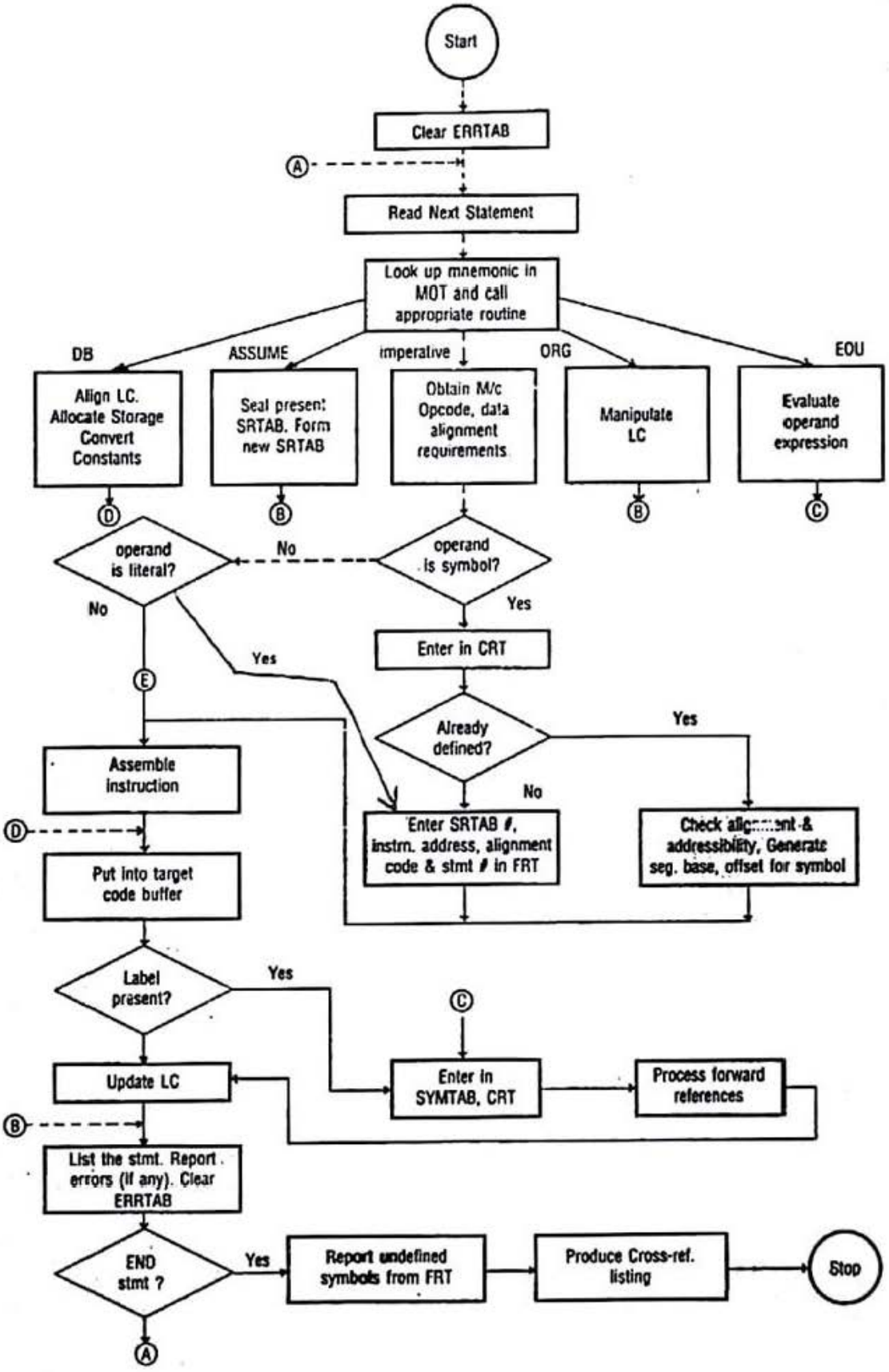
Information concerning forward references to a symbol is organized as a linked list. Thus, the forward reference table (FRT) contains a set of linked lists, one for each forward referenced symbol. The FRT pointer field of a SYMTAB entry points to the head of a symbol's list Since the ordering of FRT entries in a list is not important, for efficiency reasons new entries are added at the beginning of the list. Each FRT entry contains SRTAB # of the SRTAB that is to be used for embling the forward reference. It also contains an instruction address and a usage code th t indicates where and how the forward reference is to be assembled. When the definition of symbol is encountered, the FRT entries in its list of forward references (if any) are processed and the forward references list is discarded. To minimize the size of FRT, entries in the discarded list are reused for storing information about other forward reference.

Cross references:

A cross reference directory is a report produced by the assembler which lists all references to a symbol sorted in the ascending order by statement numbers. The assembler uses the cross reference table (CRT) to collect the relevant information.

Each entry in the CRT describes one reference to one symbol. Entries pertaining to the same symbol are linked together. New entries are added at the end of the list so that entries would be in ascending order by statement number. The SYMTAB entry of each symbol points to the head and tail of its linked list in the CRT.

Flowchart of a Single Pass Assembler:



Flowchart of Single Pass Assembler

Module 03: Macros and Macroprocessors

Contents:
Macro instructions, Features of Macro facility, Design of 2 Pass Macroprocessor.

Introduction:

Macro instructions are single-line abbreviation for groups of instructions. In employing a macro, the programmer essentially defines a single "instruction" to represent a block of code. For every occurrence of this one-line macro instruction in his program, the macro processing assembler will substitute the entire block.

By defining the appropriate macro instructions, an assembly language programmer can tailor his own higher level facility in a convenient manner, at no cost in control over the structure of his program. He can achieve the conciseness and ease in coding of high level languages without losing the basic advantage of assembly language programming. Integral macro operations simplify debugging and program modification and they facilitate standardization.

Macro instructions are usually considered an extension of the basic assembler language, and the macro processor is viewed as an extension of the basic assembler algorithm. The purpose of the macros is to either automate frequently used sequences or enable a more powerful abstraction

3.1 Macro Instructions:

In its simplest form, a macro is an abbreviation for a sequence of operations.
Consider the following program:

```
        .
        .
        .
A       1, DATA
A       2, DATA
A       3, DATA
        .
        .
        .
A       1, DATA
A       2, DATA
A       3, DATA
        .
        .
        .
```

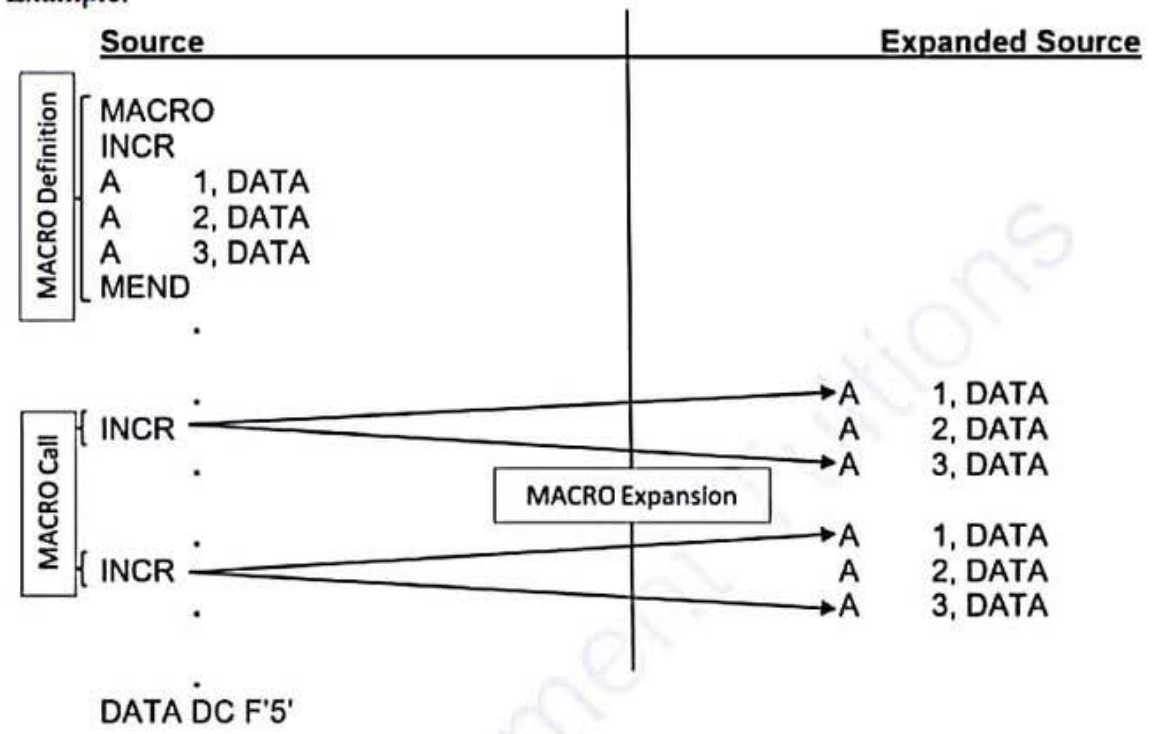
General Format of Macro Definition:

Start of definition	→	**MACRO**
MACRO Name	→	[ ]
Sequence to be abbreviated	→	{ ____ ____ ____ }
End of definition	→	**MEND**

Module 03: Macros and Macroprocessors

- The MACRO pseudo-op is the first line of the definition and identifies the following line as the macro instruction name.
- Following the name line is the sequence of instruction being abbreviated – instruction comprising the 'macro' instruction.
- The definition is terminated by a line with the MEND (macro end) pseudo-op.

Example:



Macro v/s Subroutine:

MACRO	Subroutine (Functions/Procedure)
MACRO name in the mnemonic field leads to expansion only.	Subroutine name is a call statement in the program leads to execution.
MACROs are completely handled by the Assembler (Preprocessor) during assembly time.	Subroutine are completely handled by the hardware at runtime.
Macro definition and macro expansion are executed by the assembler. So, the assembler has to know all the features, options, and exceptions associated with them.	Hardware executes the subroutine call instruction. So, it has to know how to save the return address and how to branch to the subroutine.
The hardware knows nothing about macros.	The assembler knows nothing about subroutines.
The macro processor generates a new copy of the macro and places it in the program.	The subroutine call instruction is assembled in the usual way and treated by the assembler as any other instruction.
Macro processing increases the size of the resulting code but results in faster execution of program for expanded programs.	Use of subroutines does not result into bulk object codes but has substantial overheads of control transfer during execution.
Macros are faster because it avoids function call overhead.	Involves call overhead, hence are slower.

Macros check the number of arguments; they do not check argument types.	Function checks both number of arguments and argument type.
The macro replaces the shorthand text, with the larger definition. If a macro is used twenty times in code in all instances, the longer definition will be substituted in and thus code grows.	A function definition occurs only once.
Debugging in macro is difficult as compared to function as size of the code grows.	Debugging is comparatively easy.
No type restriction is placed in arguments so that one macro may serve for several data types.	Has a specific data type for the arguments.
Macro should be used for small tasks.	Functions should be used for the complex tasks.
Macro does not alter the flow of execution.	Function does alters the flow of execution.
Macro calls a process at the translation time.	Function calls a process at the execution time.

3.2 Features of Macro Facility:

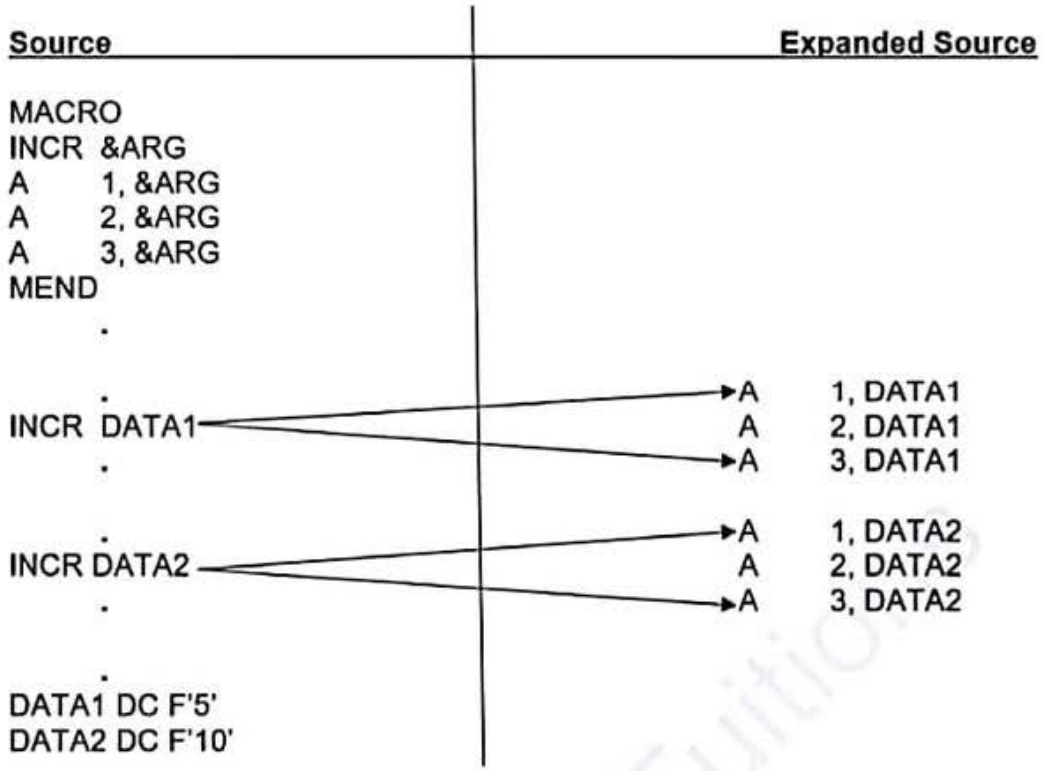
1. Macro Instruction Arguments/ Parameterized Macro/ Positional Macro:

The macro facility presented thus far is capable of inserting blocks of instruction in place of macro calls. All of the calls to any given macro will be replaced by identical blocks. This macro facility lacks flexibility: there is no way for a specific macro call to modify the coding that replaces it. An important extension of this facility consists of providing for arguments, or parameters, in macro calls. Corresponding macro dummy arguments (preceded by '&' symbol) will appear in macro definitions.

Consider the following program:

```
                .
                .
                .
        A       1, DATA1
        A       2, DATA1
        A       3, DATA1
                .
                .
                .
        A       1, DATA2
        A       2, DATA2
        A       3, DATA2
                .
                .
                .
DATA1   DC      F'5'
DATA2   DC      F'10'
```

In this case the instruction sequences are very similar but not identical. The first sequence performs an operation using DATA1 as operand; the second, using DATA2. They can be considered to perform the same operation with a variable, or argument. Such parameter is called as macro instruction argument or dummy argument.



There are generally two ways of specifying arguments to a macro call:

a. **Positional arguments:**
   In this arguments are matched with the dummy arguments according to the order in which they appear with the positional parameter the programmer must be careful enough to specify the arguments in the proper order.
   E.g. INCR &ARG1 &ARG2 &ARG3

b. **Keyword arguments:**
   They allow reference to the dummy arguments by name as well as by the position. This is useful when a macro has arguments that are not always used.
   E.g. INCR &ARG1=A, &ARG3=C, &ARG2=B
   Or
   &ARG1=A, &ARG2=&ARG3=C

2. **Conditional MACRO Expansion:**

Two important macro processor pseudo-ops AIF and AGO, permit conditional reordering of the sequence of macro expansion. This allows conditional selection of the machine instruction that appears in expansions of macro call.

AIF: Arithmetic operations are performed by AIF and if nested is true then only are branched. It is conditional opcode.

AGO: It is similar to 'go-to' statement in C/C++. It is an unconditional pseudo-opcode. It specifies a label appearing on some other statements in the macro definition.

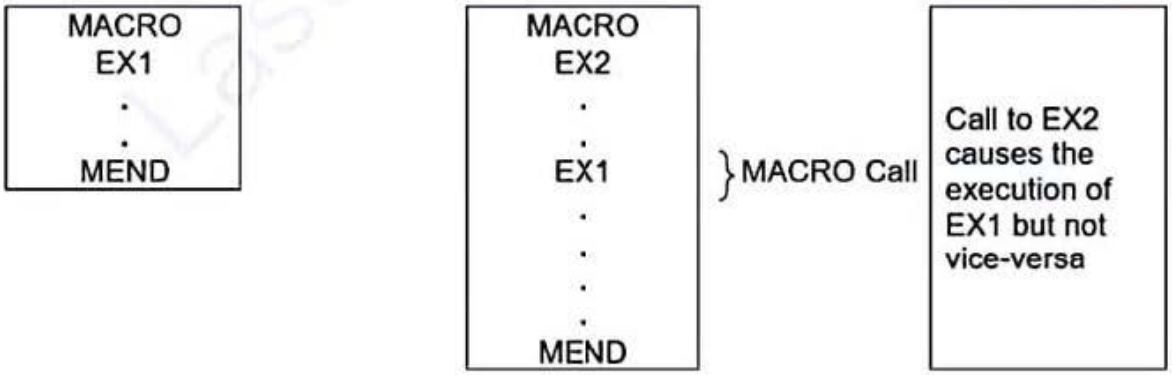
Consider the following program:

Source			Expanded Source		
		.			.
		.			.
	MACRO				.
&AG	DATA	&C, &A2, &A3			.
&AG	D	2, &A2			.
	AIF	(&C EQ 1) .FINI			.
	D	3, &A3			.
	AIF	(&C EQ 2) .FINI			.
.FINI	MEND				.
		.			.
		.			.
		.			.
S1	DATA	2, VALUE2, VALUE3	S1	D	2, VALUE2
				D	3, VALUE3
		.			.
		.			.
		.			.
S2	DATA	1, VALUE2	S2	D	2, VALUE2
		.			.
		.			.

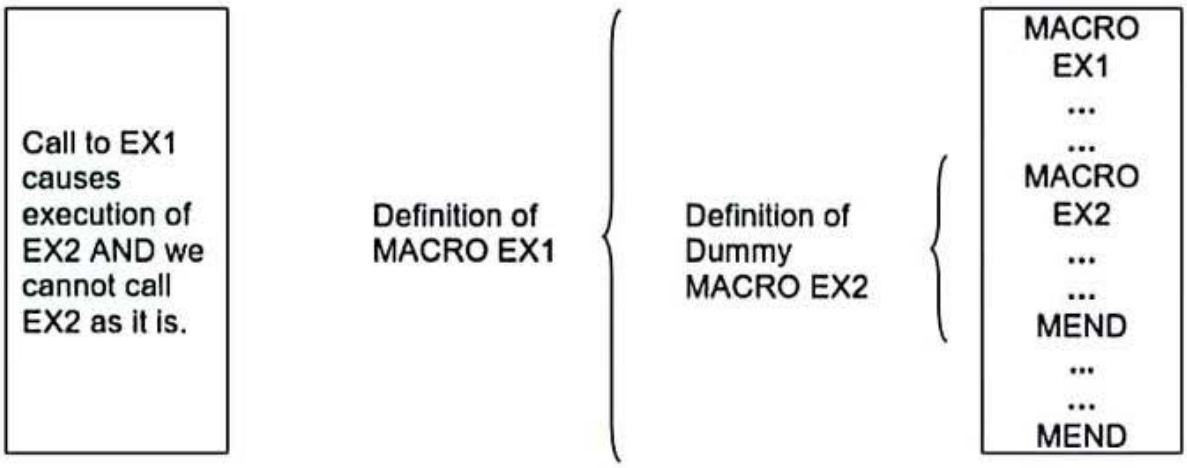
Note: .FINI is a Macro Label meant to be used only by the Macroprocessor. Macro Labels are preceded by the '.' Dot Symbol.

3. MACRO Calls within MACROS/ Nested MACROS:

In this feature of macro facility when one MACRO expansion is taking place we can encounter another macro call or during one macro call another macro call is invoked. In this macro bodies contain macro calls and processes macro definition during expansion time.



MACRO Calls within MACRO



Nested MACRO

4. Recursive MACROS:

It allows the macro invocation statements appearing within the body of a macro instruction. MACROs that invoke themselves are called recursive MACROs. If MACROs are calling themselves, one speaks of recursive macro calls. In this case, there must be some stop criterion, to prevent the macro of calling itself over and over until the assembler runs out of memory. To write a recursive macro we need to have,

- Recursive function
- Termination function

5. MACRO Instructions Defining MACROs:

A single macro instruction might be used to simplify the process of defining a group of similar macros. To call the inner macro it is necessary to define the outer macro first.

Module 03: Macros and Macroprocessors

Functions of Macro Processor (Statement of Problem):

There are four basic tasks that any macro instruction processor must perform:

1. **Recognize macro definition:** A macro instruction processor must recognize macro definitions identified by the MACRO and MEND pseudo-ops. This task can be complicated when macro definitions appear within macros. When MACROs and MENDs are nested, the macro processor must recognize the nesting and correctly match the last or outer MEND with the first MACRO. All the intervening text, including nested MACROs and MENDs, defines a single macro instruction.

2. **Save the definitions:** The processor must store the macro instruction definitions, which it will need for expanding macro calls.

3. **Recognize calls:** The processor must recognize macro calls that appear as operation mnemonics. This suggests that macro names be handled as a type of op-code.

4. **Expand calls and substitute arguments:** The processor must substitute for dummy or macro definition arguments the corresponding arguments from a macro call; the resulting symbolic text in then substituted for the macro call.

3.3 Design of Two-Pass Macro-Processor:

Specification of Databases:

The following data bases are used by the two passes of the macro processor.

Pass 1 Databases:

1. The input macro source deck
2. The output macro source deck copy for use by pass 2
3. The Macro Definition Table (MDT), used to store the body of macro definitions
4. The Macro Name Table (MNT), used to store the names of defined macros
5. The Macro Definition Table Counter (MDTC), used to indicate the next available entry in the MDT
6. The Macro Name Table Counter (MNTC), used to indicate the next available entry in the MNT
7. The Argument List Array (ALA), used to substitute index markers for dummy arguments before storing a macro definition

Pass 2 Databases:

1. The copy of the input macro source deck
2. The output expanded source deck to be used as input to the assembler
3. The Macro Definition Table (MDT), created by pass 1
4. The Macro Name Table (MNT), created by pass 1
5. The Macro Definition Table Pointer (MDTP), used to indicate the next line of text to be used during macro expansion
6. The Argument List Array (ALA), used to substitute macro call arguments for the index markers in the stored macro definition

Module 03: Macros and Macroprocessors

Specification of Database Format:

The only data bases with nontrivial format are the Macro Definition Table (MDT), the Macro Name Table (MNT), and the Argument List Array (ALA). The others we will not present in detail.

1. Argument List Array (ALA):

The Argument List Array (ALA) is used during both pass 1 and pass 2 but for somewhat reverse functions. During pass 1, in order to simplify later argument replacement during macro expansion, dummy arguments in the macro definition are replaced with positional indicators when the definition is stored. The 1th dummy argument on the macro name card is represented in the body of the macro by the index marker symbol # where #i is a symbol reserved for the use of the macro processor (i.e., not available to the programmers). These symbols are used in conjunction with the argument list prepared before expansion of a macro call. The symbolic dummy arguments are retained on the macro name card to enable the macro processor to handle argument replacement by name rather than by position.

As an example, consider the macro INCR used in above example. The stored macro definition would be:

Macro Definition Table

80 bytes per entry

Index		*Card*	
⋮		⋮	
15	&LAB	INCR	&ARG1,&ARG2,&ARG3
16	#0	A	1, #1
17		A	2, #2
18		A	3, #3
19		MEND	
⋮		⋮	

During pass 2 it is necessary to substitute macro call arguments for the index markers stored in the macro definition. Thus upon encountering the call

LOOP INCR DATA1, DATA2, DATA3

The macro call expander would prepare an argument list array:

Argument List Array

8 bytes per entry

Index	*Argument*
0	"LOOP1*bbb*"
1	"DATA1*bbb*"
2	"DATA2*bbb*"
3	"DATA3*bbb*"

(*b* denotes the blank character)

2. Macro Definition Table (MDT):

The Macro Definition Table (MDT) is a table of text lines; if input is from 80-column cards, the MDT can be a table with 80-byte string as entries. Every line of each macro definition, except the MACRO line, is stored in the MDT. (The MACRO line is useless during macro expansion.) The MEND is kept to indicate the end of the definition; and the macro name line is retained to facilitate keyword argument replacement. Thus, for example, the INCR macro discussed might be stored as follows:

Macro Definition Table			
		80 bytes per entry	
Index		*Card*	
⋮		⋮	
15	&LAB	INCR	&ARG1,&ARG2,&ARG3
16	#0	A	1, #1
17		A	2, #2
18		A	3, #3
19		MEND	
⋮		⋮	

3. Macro Name Table (MNT):

The Macro Name Table (MNT) serves a function very similar to that of the assembler's Machine-Op Table (MOT) and Pseudo-Op Table (POT). Each MNT entry consists of a character string (the macro name) and a pointer (index) to the entry in the MDT that corresponds to the beginning of the macro definition. The MNT entry for the INCR macro discussed might be:

	8 bytes	4 bytes
Index	*Name*	*MDT Index*
⋮	⋮	⋮
3	"INCR*bbbb*"	15
⋮	⋮	⋮

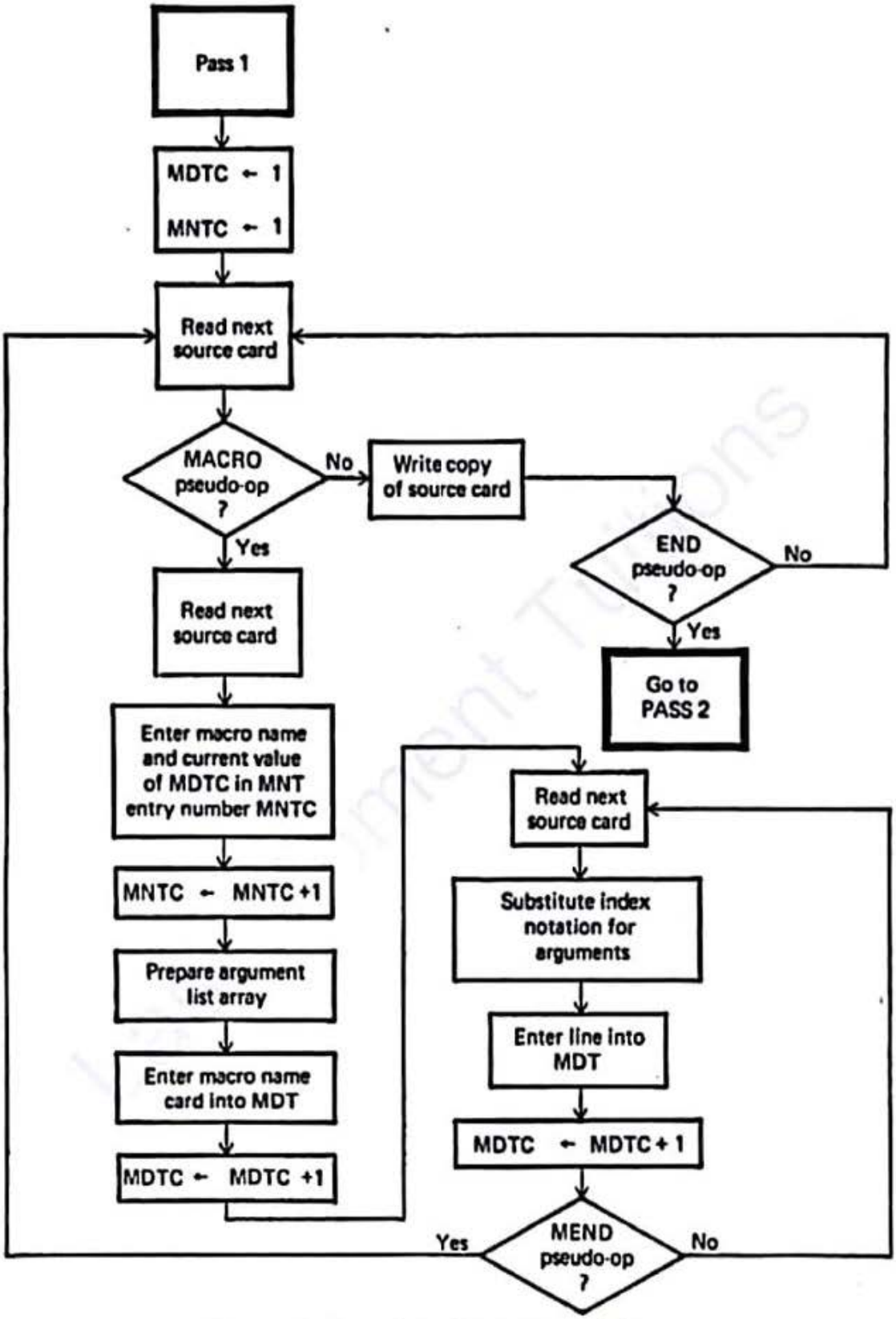
Macro Name Table

Algorithm:

Each of the algorithms makes a line-by-line scan over its input. The 'READ' boxes refer to the fetching of successive input lines from secondary storage into a workspace.

PASS 1- MACRO DEFINITION:

The algorithm for pass 1 tests each input line. If it is a MACRO pseudo-op, the entire macro definition that follows is saved in the next available locations in the Macro Definition Table (MDT). The first line of the definition is the macro name line. The name is entered into the Macro Name Table (MNT), along with a pointer to the first location of the MDT entry of the definition. When the END pseudo-op is encountered, all of the macro definitions have been processed so control transfers to pass 2 in order to process macro calls.



Pass 1 - Processing of Macro definitions

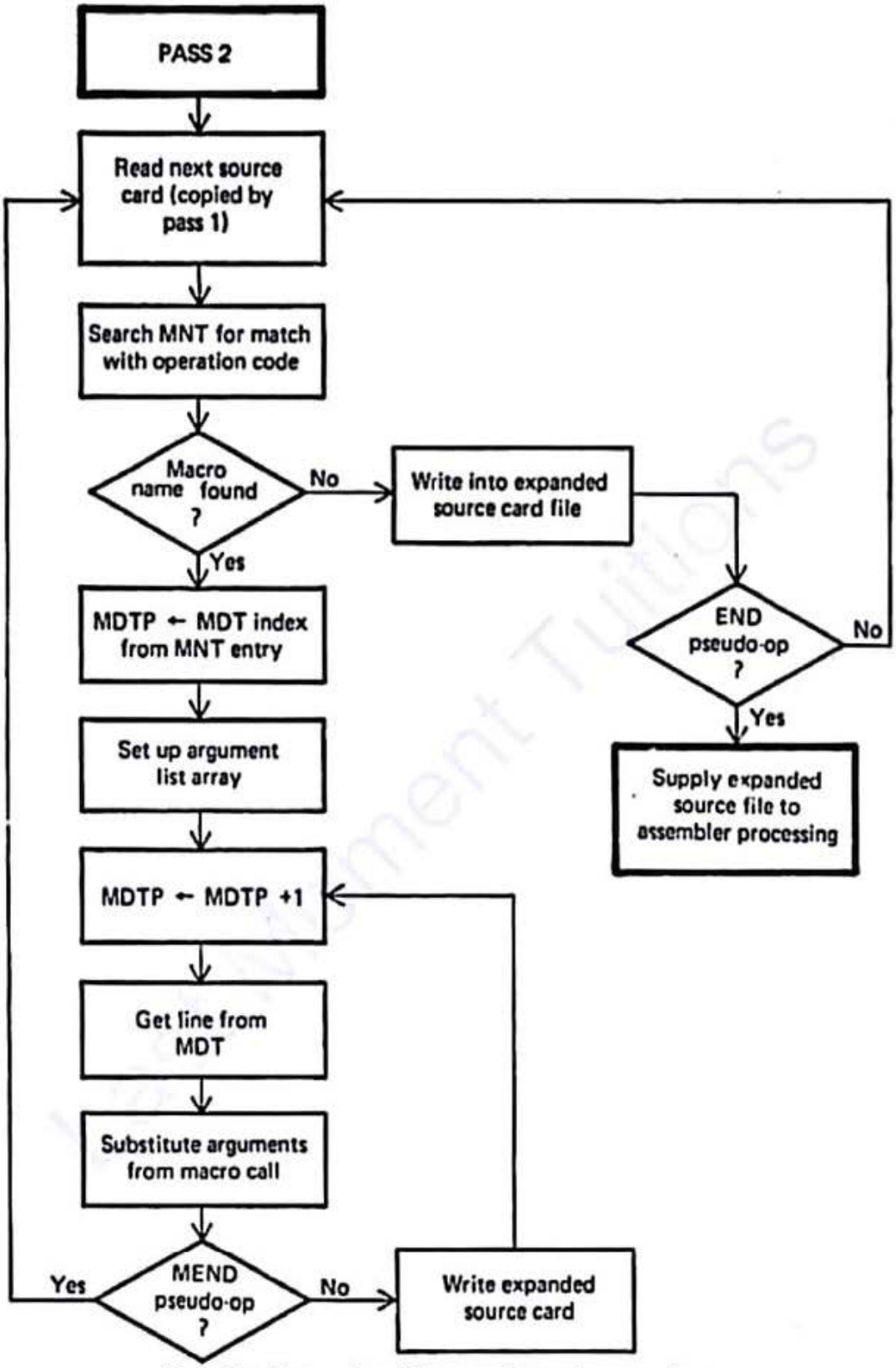
PASS 2-MACRO CALLS AND EXPANSION:

The algorithm for pass 2 tests the operation mnemonic of each input line to see if it is a name in the MNT. When a call is found, the call processor sets a pointer, the Macro Definition Table Pointer (MDTP), to the corresponding macro definition stored in the MDT. The initial value of the MDTP is obtained from the "MDT index" field of the MNT entry. The macro expander prepares the Argument List Array (ALA) consisting of a table of dummy argument indices and corresponding arguments to the call. This list is simply a succession of symbols ordered to match the dummy arguments on the name card (the first is the label argument, which is considered to have an index or zero).

Arguments not represented in a call are considered blank, and superfluous arguments are ignored. In the case of argument reference by position, this scheme is completely straightforward. For references by name, the macro processer locates the dummy argument on the macro name line (which is available at the beginning of the definition in the MDT) in order to determine the proper index.

Reading proceeds from the MDT; as each successive line is read, the values from the argument list are substituted for dummy argument indices in the macro definition. Reading of the MEND line in the MDT terminates expansion of the macro, and scanning routines from the input file. When the END pseudo-op is encountered, the expanded source deck is transferred to the assembler for further processing.

Last Moment Tuti



Pass 2 – Processing Macro calls and expansion

Module 03: Macros and Macroprocessors

Example:

Sample Program	Pass I	Pass II
PRG START 0	PRG START 0	PRG START 0
.	.	.
.	.	.
MACRO	.	.
INCR	.	.
A 1 DATA	INCR	A 1 DATA
A 2 DATA	.	A 2 DATA
A 3 DATA	.	A 3 DATA
MEND	INCR	.
.	.	.
.	.	A 1 DATA
INCR	END	A 2 DATA
.		A 3 DATA
.		.
INCR		.
.		END
.		
END		

MNT		
1	INCR	1

MDT	
1	INCR
2	A 1 DATA
3	A 2 DATA
4	A 3 DATA
5	MEND

Last Moment Tutorials

Module 04: Loaders and Linkers

Contents:
Loader schemes, Design of Absolute loader, Design of Direct linking loader

Loader:

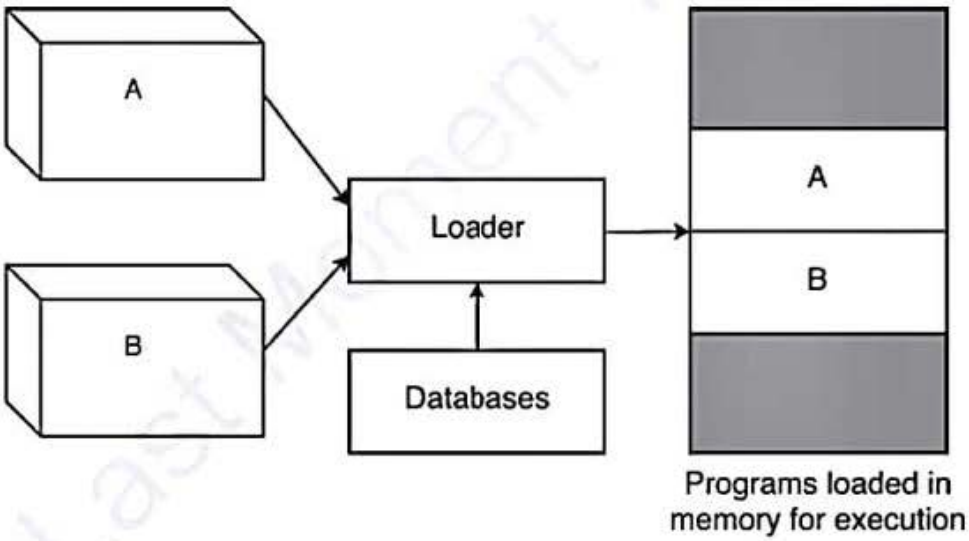
The user's source program decks are usually converted to object program decks (machine language) by assemblers and compilers. The loader is a program which accepts the object program decks, prepares these programs for execution by the computer, and initiates the execution

Functions of a Loader:

1. **Allocation:** Allocate space in memory for the programs
2. **Linking:** Resolve symbolic references between object decks.
3. **Relocation:** Adjust all address dependent locations, such as address constants, to correspond to the allocated space.
4. **Loading:** Physically place the machine instructions and data into memory.

General Loader:

In General loader, the source program is converted to object program by some translator (assembler). The loader accepts these object modules and puts machine instruction and data in an executable form at their assigned memory. The loader occupies some portion of main memory.



Loader

Advantages:

- The program need not be retranslated each time while running it. This is because initially when source program gets executed an object program gets generated. If program is not modified, and then loader can make use of this object program to convert it to executable form.
- There is no wastage of memory, because assembler is not placed in the memory, instead of it, loader occupies some portion of the memory. And size of loader is smaller than assembler, so more memory is available to the user.
- It is possible to write source program with multiple programs and multiple languages, because the source programs are first converted to object programs always, and loader accepts these object modules to convert it to executable form.

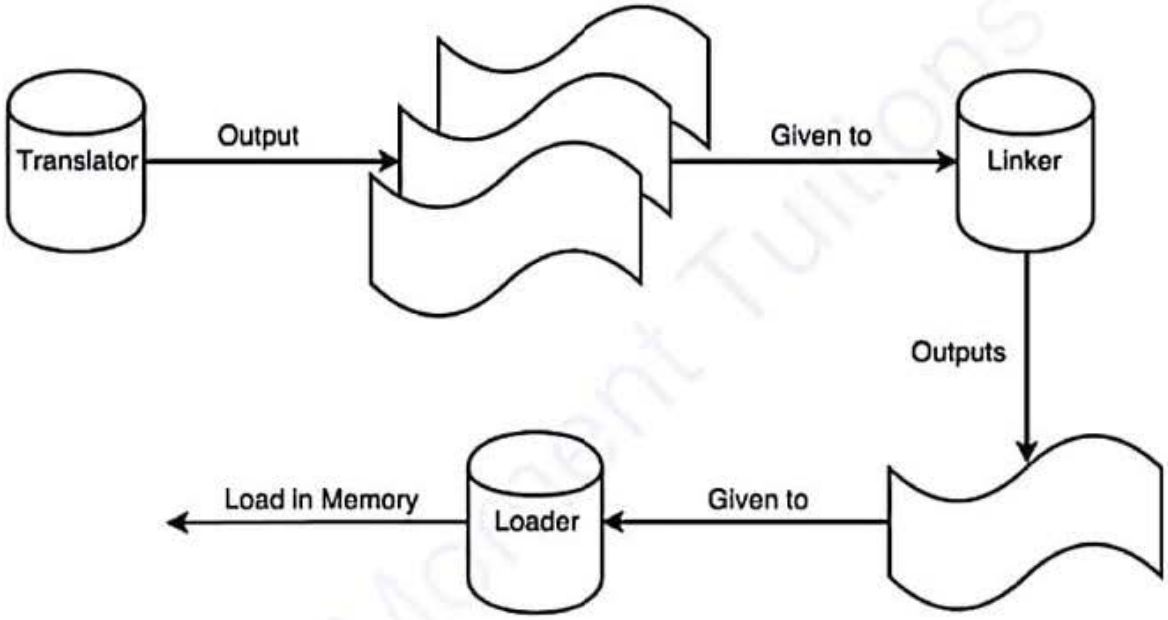
Module 04: Loaders and Linkers

Concept of Linking:

As we have discussed earlier, the execution of program can be done with the help of following steps:

1. **Translation** of the program (done by assembler or compiler)
2. **Linking** of the program with all other programs which are needed for execution. This also involves preparation of a program called load module.
3. **Loading** of the load module prepared by linker to some specified memory location.

The output of translator is a program called object module. The linker processes these object modules binds with necessary library routines and prepares a ready to execute program. Such a program is called binary program. The "binary program also contains some necessary information about allocation and relocation. The loader then loads this program into memory for execution purpose.



Process of linking a program

Various tasks of linker are:

1. Prepare a single load module and adjust all the addresses and subroutine references with respect to the offset location.
2. To prepare a load module concatenate all the object modules and adjust all the operand address references as well as external references to the offset location.
3. At correct locations in the load module, copy the binary machine instructions and constant data in order to prepare ready to execute module.

The linking process is performed in two passes. Two passes are necessary because the linker may encounter a forward reference before knowing its address.

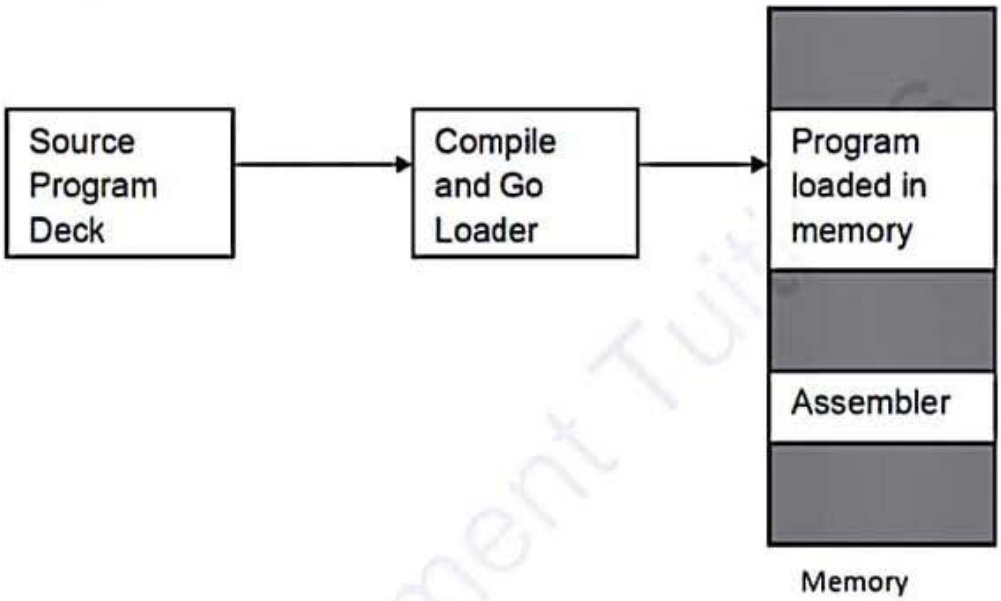
Module 04: Loaders and Linkers

4.1 Loader Schemes:

Based on the various functionalities of loader, there are six types of loaders:

1. **"Assemble (or Compile) and Go" Loader:**

In this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address. That means the assembler runs in one part of memory and the assembled machine instructions and data is directly put into their assigned memory locations. After completion of assembly process, assign starting address of the program to the location counter. The typical example is WATFOR-77; it's a FORTRAN compiler which uses such "load and go" scheme. This loading scheme is also called as "assemble and go".



Compile and Go Loader

Advantages:

- This scheme is simple to implement. Because assembler is placed at one part of the memory and loader simply loads assembled machine instructions into the memory.
- No extra procedures are involved.

Disadvantages:

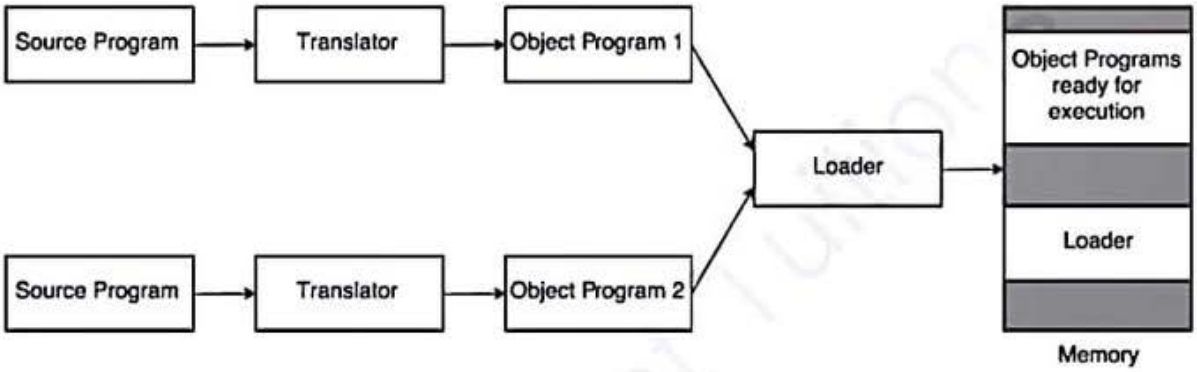
- In this scheme some portion of memory is occupied by assembler which is simply wastage of memory. As this scheme is combination of assembler and loader activities, this combination program occupies large block of memory.
- There is no production of .obj file; the source code is directly converted to executable form. Hence even though there is no modification in the source program it needs to be assembled and executed each time, which then becomes a time consuming activity.
- It cannot handle multiple source programs or multiple programs written in different languages. This is because assembler can translate one source language to other target language.
- For a programmer it is very difficult to make an orderly modulator program and also it becomes difficult to maintain such program, and the "compile and go" loader cannot handle such programs.
- The execution time will be more in this scheme as every time program is assembled and then executed.

Module 04: Loaders and Linkers

2. General Loader Scheme:

Outputting the instructions and data as they are assembled circumvents the problem of wasting core for the assembler. Such an output could be saved and loaded whenever the code was to be executed. The assembled programs could be loaded into the same area in core that the assembler occupied (since the translation will have been completed). This output form, which may be on cards containing a coded form of the instructions, is called an object deck.

The use of an object deck as intermediate data to avoid one disadvantage of the preceding "compile-and-go" scheme requires the addition of a new program to the system, a loader. The loader accepts the assembled machine instructions, data and other information present in the object format and places machine instructions and data in core in an executable computer form. The loader is assumed to be smaller than the assembler, so that more memory is available to the user. A further advantage is that reassembly is no longer necessary to run the program at a later date.



General Loading Scheme

Finally, if all the source program translators (assemblers and compilers) produce compatible object, program deck formats and use compatible linkage conventions, it is possible to write subroutines in several different languages since the object decks to be processed by the loader will all be in the same "language" (machine language).

3. Absolute Loader:

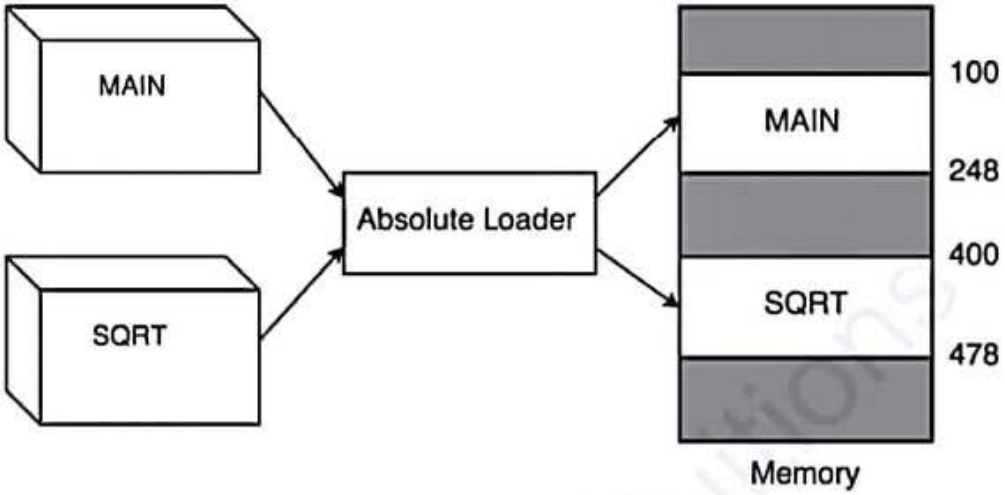
Absolute loader is a kind of loader in which relocated object files are created. Loader accepts these files and places them at specified locations in the memory. This type of loader is called absolute because no relocation information is needed; rather it is obtained from the programmer or assembler.

The starting address of every module is known to the programmer, this corresponding starting address is stored in the object file, then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file. In this scheme, the programmer or assembler should have knowledge of memory management. The resolution of external references or linking of different subroutines is the issues which need to be handled by the programmer.

The programmer should take care of two things: First thing is the specification of starting address of each module to be used. If some modification is done in some module, then the length of that module may vary. This causes a change in the starting address of immediate next modules, its then the programmer's duty to make necessary changes in the starting addresses of respective modules. Second thing is, while branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction.

Module 04: Loaders and Linkers

The resolution of mutual references and linking is done by the programmer. The task of assembler is to create the object codes for the above segments and along with the information such as starting address of the memory where actually the object code can be placed at the time of execution. The absolute loader accepts these object modules from assembler and by reading the information about their starting addresses, it will actually place (load) them in the memory at specified addresses.



Absolute Loader

Thus the absolute loader is simple to implement in this scheme

- Allocation is done by either programmer or assembler
- Linking is done by the programmer or assembler
- Resolution is done by assembler
- Simply loading is done by the loader
- No relocation information is needed, if at all it is required then that task can be done by either a programmer or assembler

Advantages:

- It is simple to implement and the process of execution is efficient.
- This scheme allows multiple programs or the source programs written different languages. If there are multiple programs written in different languages, then the respective language assembler will convert it to the language and a common object file can be prepared with all the ad resolution.
- The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code in the main memory.

Disadvantages:

- In this scheme it is the programmer's duty to adjust all the inter segment addresses and manually do the linking activity.
- For that, it is necessary for a programmer to know the memory management.
- If at all any modification is done some segments, the starting addresses of immediate next segments may get changed, the programmer has to take care of this issue and he needs to update the corresponding starting addresses on any modification in the source.

4. Subroutine Linkages:

It deals with the special mechanism for calling another subroutine in an assembly language program. The problem of subroutine linkage is this: a main program A wishes to transfer to subprogram B. The programmer, in program A, could write a transfer instruction (e.g., BAL 14, B) (Branch and Link) to subprogram B. However, the assembler does not know the value of this Symbol reference and will declare it as an error (undefined Symbol) unless a special mechanism has been provided.

This mechanism is typically implemented with a relocating or a direct-linking loader. The assembler pseudo-op EXTRN followed by a list of Symbols indicates that these Symbols are defined in other programs but referenced in the present program. Correspondingly, if a Symbol is defined in one program and referenced in others, we insert it into a Symbol list following the pseudo-op ENTRY. In turn, the assembler will inform the loader that these Symbols may be referenced by other programs. For example, the following sequence of instructions may be a simple calling sequence to another program:

```
MAIN     START
         EXTRN     SUBROUT
         ....
         ....
         L         15, A(SUBROUT)    } CALL
         BALR      14, 15            } SUBROUT
         ....
         ....
         END
```

The above sequence of instructions first declares SUBROUT as an external variable, that is, a variable reference but not defined in this program. The load instruction loads the address of that variable into register IS. The BALR instruction branches to the contents of register 15, which is the address of SUBROUT, and leaves the value of the next Instruction in register 14. In most assemblies we may simply use a CALL SUBROUT macro, which is translated by the assembler into a calling sequence as Shown above.

Now we may see the reason for programming conventions, since it is necessary for both the caller and called subprograms to cooperate. On a 360 we observe the convention that register 15 is used as a linkage and base register. Note that the caller in the above program has loaded register 15 with the beginning address of the subroutine being called. Thus the called subroutine does not have to load a base register. Register 14 contains the· return address of the caller. The programmer must not use register 14 within a subprogram unless he saves its contents and restores them before he returns. A typical sequence for subroutines is:

```
SUBROUT     START
            USING *, 15
            ....
            ....
            BR 14
            END
```

No BALR(Branch and Link Register) instruction is necessary at the beginning since register 15 was already loaded with the address of the start of this program. The BR 14 instruction is an unconditional branch to the address contained in register 14, which is the return address of the calling program. The following discussion introduces the basic mechanism used.

Module 04: Loaders and Linkers

Assembler Linkage Pseudo-Ops

Subroutine and Entry
Naming (START and ENTRY Pseudo-ops)

```
A     START                    Defines Subroutine A
      ENTRY     B1, B2, B3...  Defines locations of B1, … Bn as entry points

B1    ....
B2    ....
```

The uses of multiple entry points are:

- Common coding: Example, SIN and COS involve basically the same computations and could employ different entry points of the same routine.
- Collecting together related routines for convenience.
- Better or convenient access to common data base.

Subroutine Reference (EXTRN PSEUDO-OP)

Assembler symbols are either internal or external. External means that their value is not known to the assembler but will be provided by the loader – the action of the loader will be discussed in the following section.

EXTRN E1, E2, etc.

defines E1, E2, etc. as external symbols to be used in address constants.

Example,

CALL BETA becomes:

```
      EXTRN     BETA
      ....
      ....
      L         15, ABETA
      BALR      14, 15
      ....
      ....
ABETA DC        A(BETA)
```

5. **Relocating Loader:**

Relocation is the process of updating the addresses used in the address sensitive instructions of a program. It is necessary that such a modification should help to execute the program from designated area of the memory. The assembler generates the object code. This object code gets executed after loading at storage locations. The addresses of such object code will get specified only after the assembly process is over. Therefore, after loading,

Address of object code = Mere address of object code + Relocation constant

There are two types of addresses being generated: Absolute address and relative address. The absolute address can be directly used to map the object code in the main memory. Whereas the relative address is only after the addition of relocation constant to the object code address. This kind of adjustment needs to be done in case of relative address before actual execution of the code. The typical example of relative reference is: addresses of the symbols defined in the Label field, addresses of the data which is defined by the assembler directive, literals and re-definable symbols.

Similarly, the typical example of absolute address is the constants which are generated by assembler are absolute. The assembler calculates which addresses are absolute and which addresses are relative during the assembly process. During the assembly process the assembler calculates the address with the help of simple expressions.

To avoid possible reassembling of all subroutines when a single subroutine is changed, and to perform the tasks of allocation and linking for the programmer, the general class of relocating loaders was introduced.

An example of a relocating loader scheme is that of the **Binary Symbolic Subroutine (BSS)** loader such as was used in the IBM 7094, IBM 1130, GE 635, and UNIVAC 1108. The BSS loader allows many procedure segments, yet only one data segment (common segment). The assembler assembles each procedure segment independently and passes on to the loader the text and information as to relocation and inter-segment references.

The output of a relocating assembler using a BSS scheme is the object program and information about all other programs it references. In addition, there is information (relocation information) as to locations in this program that need to be changed if it is to be loaded in an arbitrary place in core, i.e. the locations which are dependent on the core allocation.

For each source program the assembler outputs a text (machine translation of the program) prefixed by a transfer vector that consists of addresses containing names of the subroutines referenced by the source program. For example, if a Square Root Routine (SQRT) was referenced and was the first subroutine called, the first location in the transfer vector would contain the symbolic name SQRT. The statement calling SQRT would be translated into a transfer instruction indicating a branch to the location of the transfer vector associated with SQRT.

Program length = 48 bytes
Transfer vector = 8 bytes

Source program				*Rel. addr.*	*Relocation*	*Object code*	
MAIN	START						
	EXTRN	SQRT		0	00	'SQRT'	
	EXTRN	ERR		4	00	'ERRb'	
	ST	14,SAVE	Save return address	8	01	ST	14,36
	L	1,=F'9'	Load test value	12	01	L	1,40
	BAL	14,SQRT	Call SQRT	16	01	BAL	14,0
	C	1,=F'3'	Compare answer	20	01	C	1,44
	BNE	ERR	Transfer to ERR	24	01	BC	7,4
	L	14,SAVE	Get return address	28	01	L	14,36
	BR	14	Return to caller	32	0	BCR	15,14
				34	0	(Skipped for alignment)	
SAVE	DS	F	Temp. loc.	36	00	(Temp location)	
	END			40	00	9	
				44	00	3	

Assembly of program for "direct-address" 360

Module 04: Loaders and Linkers

The assembler would also provide the loader with additional information, such as the length of the entire program and the length of the transfer vector portion. After loading the text and the transfer vector into core, the loader would load each subroutine identified in the transfer vector. It would then place a transfer instruction to the corresponding subroutine in each entry in the transfer vector. Thus, the execution of the call SQRT statement would result in a branch to the first location in the transfer vector, which would contain a transfer instruction to the location of SQRT.

Since it is necessary to relocate the address portion of every instruction, computers with a direct-address instruction format have a much more severe relocation problem than the 360. This problem can be solved by the use of "relocation bits". The assembler associates a bit with each instruction or address field. If this bit equals one, then the corresponding address field must be relocated; otherwise the field is not relocated.

BSS performs all the four functions of the loader.

- **Allocation:** It is done with the help of the program length and the information
- **Linking:** It is done by Transfer Vector.
- **Relocation:** With the help of the relocation bits.
- **Loading:** It is done by the loader.

Advantages:

- Overcomes the drawbacks of Absolute Loaders.

Disadvantages:

- The Transfer Vector is used only for transfers and not for loading or storing of external data.
- The size of the object program in the memory is increased due to transfer vector.
- The BSS Loader processes procedure segments but does not facilitate access to the data segments that can be shared.

6. Direct-Linking Loader:

A direct-linking loader is a general relocatable loader and is perhaps the most popular loading scheme presently used. It has the advantage of allowing the programmer multiple procedure segments and multiple data segments and of giving him complete freedom in referencing data or instructions contained in other segments. This provides flexible inter segment referencing and accessing ability, while at the same time allowing independent translations of programs.

The assembler (translator) must give the loader the following information with each procedure or data segment:

- The length of segment.
- A list of all the symbols in the segment that may be referenced by other segments and their relative location within the segment.
- A list of all symbols not defined in the segment but referenced in the segment.
- Information as to where address constants are located in the segment and a description of how to revise their values.
- The machine code translation of the source program and the relative addresses assigned.

Module 04: Loaders and Linkers

The design of the direct-linking loading scheme is similar to the standard IBM 370 scheme. The assembler produces four types of cards in the object deck: ESD, TXT, RLD, and END. External Symbol Dictionary (ESD) cards contain information about all symbols that are defined in this program but that may be referenced elsewhere and all symbols referenced in this program but defined elsewhere. The text (TXT) cards contain the actual object code translated version of the source program. The Relocation and Linkage Directory (RLD) cards contain information about those locations in the program whose contents depend on the address at which the program is placed. For such locations, the assembler must supply information enabling the loader to correct their contents. The END card indicates the end of the object deck and specifies the starting address for execution if the assembled routine is the "main" program.

7. Linkage Editors (Binders):

Each time for execution of a program, the allocation, relocation linking and loading needs to be done. Now doing these activities each time increases the time and space complexity. The idea is to separate out these activities in separate groups. The program which performs allocation, relocation and linking is called binder.

A **Binder** is a program that performs the same functions as the Direct Linking Loader in *"binding"* subroutines together, but rather than placing the relocated and linked text directly into memory, it outputs the text as a file or card deck. This output file is in a format ready to be loaded and is typically called a *load module*. The *module loader* merely has to physically load the module into core. The binder essentially performs the functions of allocation, relocation, and linking; the module loader merely performs the function of loading.

There are two major classes of binders. The simplest type produces a load module that looks very much like a single absolute loader deck. This means that the specific core allocation of the program is performed at the time that the subroutines are bound together. Since this kind of module looks like an actual "snapshot" or "image" of a section of core, it is called a *core image module* and the corresponding binder is called a **core image builder**.

A more sophisticated binder, called a **Linkage Editor**, can keep track of the relocation information so that the resulting load module, as an ensemble, can be further relocated and thereby loaded anywhere in core. In this case the module loader must perform additional allocation and relocation as well as loading, but it does not have to worry about the complex problems of linking. It performs all linking and some relocation prior to load time and writes the linked program into an *"executable image"*. This approach is suitable when the program has to be re-executed many times without being re-assembled.

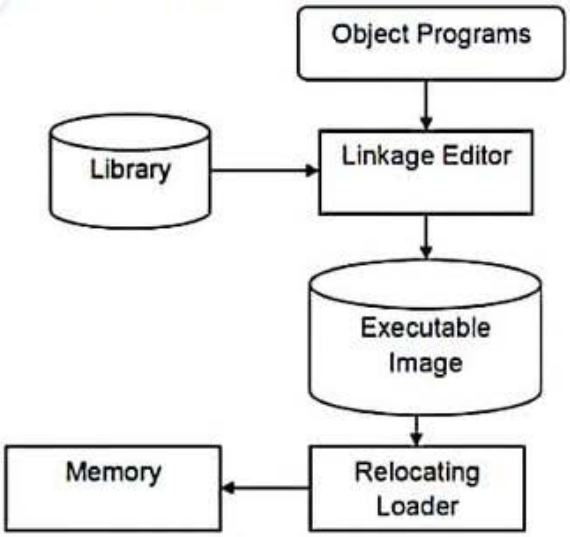
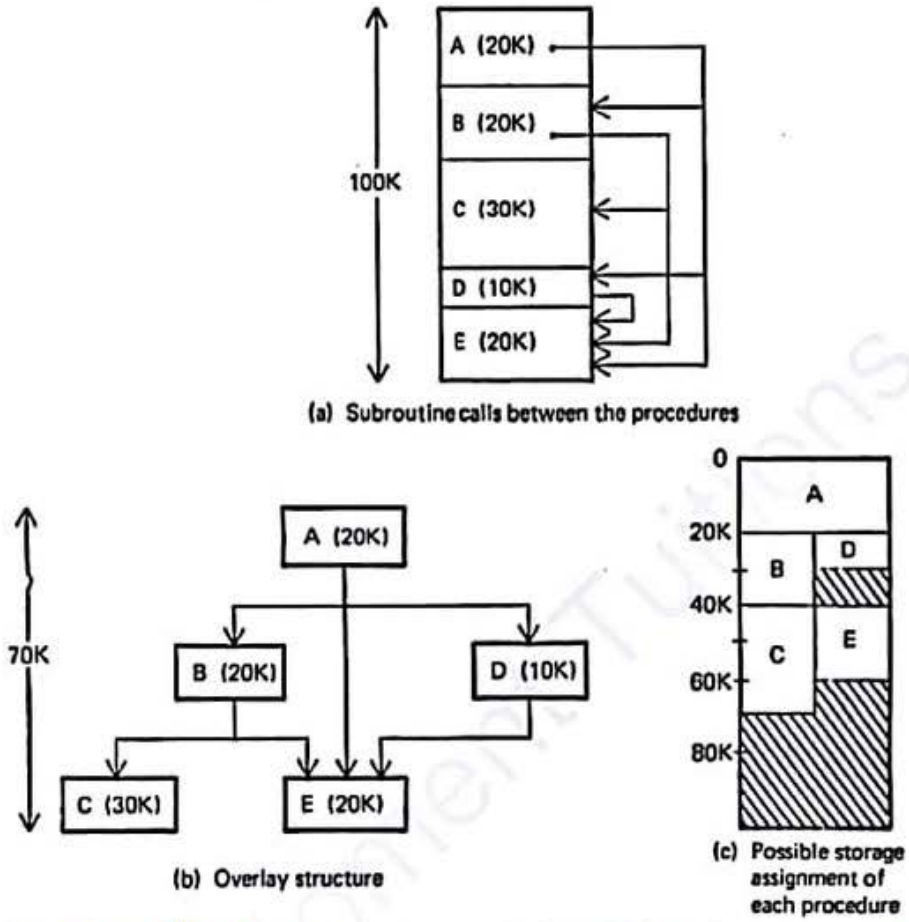


Figure: Linkage Editor

8. Dynamic Loading Loader

Usually the subroutines of a program are needed at different times. By explicitly recognizing which subroutines call other subroutines it is possible to produce an overlay structure that identifies mutually exclusive subroutines.



The above figure illustrates a program consisting of five subprograms (A, B, C, D and E) that require 100K bytes of core. The arrows indicate that subprogram A only calls B, D and E; subprogram B only calls C and E; subprogram D only calls E; and subprograms C and E do not call any other routines. The Figure (b) highlights the interdependencies between the procedures. If we load only those procedures that are actually to be used at any particular time, the amount of core needed is equal to the longest path of the overlay structure. This happens to be 70K for procedures A, B, and C. The Figure (c) illustrates a storage assignment for each procedure consistent with the overlay structure.

In order for the overlay structure to work it is necessary for the module loader to load the various procedures as they are needed. That means it is not necessary to resident all the subroutines in the memory for all the time. The portion of the loader that actually intercepts the "calls" and loads the necessary procedure is called the overlay supervisor or simply the flipper. This process of linking and loading of external reference is called dynamic loading or load-on-call (LOCAL).

Advantages:

- No wastage of memory.
- A system can be dynamically loaded whenever it is required.

9. Dynamic Linking Loader

In Dynamic Linking mechanism, loading and linking of external references are postponed until execution time. That is, the assembler produces text, binding, and relocation information from a source language deck. The loader loads only the main program. If the main program should execute a transfer instruction to an external address, or should reference an external variable (that is, a variable that has not been defined in this procedure segment), the loader is called. Only then is the segment containing the external reference loaded. It performs linking and relocation at run time and loads the linked program directly into memory for execution; no "executable image" is generated.

It is more suitable in a program development and testing environment (where size of programs being executed is comparatively small), as it avoids steps of reading and writing the linked program into an executable image.

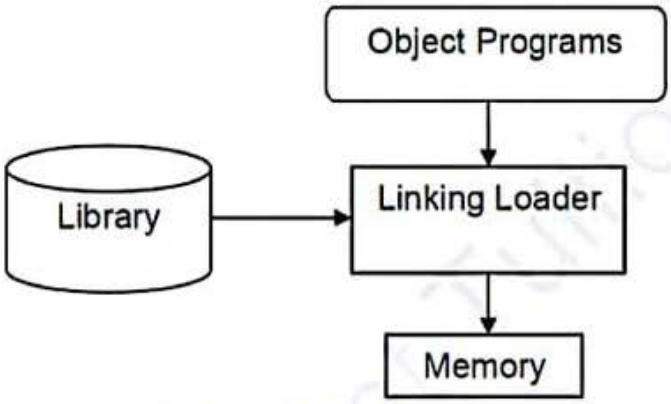


Figure: Linking Loader

Linking function is performed at runtime and facilities of the OS are used to load and link subprograms at the time when they are first called. Dynamic Linking is often used to allow several executing programs to share one copy of a subroutine through a library; all supporting run-time routines common to a set of executing programs can be packed into a single "library" of subroutines, which is shared by all programs.

Advantages:

- The overhead on the loader is reduced. The required subroutine will be load in the main memory only at the time of execution.
- The system can be dynamically reconfigured.

Disadvantages:

- The linking and loading need to be postponed until the execution.
- During the execution if at all any subroutine is needed then the process of execution needs to be suspended until the required subroutine gets loaded in the main memory.

Module 04: Loaders and Linkers

Comparison between Dynamic Linking Loader v/s Linkage Editor (Binder)

	Dynamic Linking Loader	Linkage Editor (Binder)
1	A Linking Loader performs linking and relocation at run time and directly loads the linked program into the memory.	A Linkage Editor performs linking and some relocation operations prior to load time and write the linked program to an executable for later execution.
2	It resolves external references every time the program is executed.	External references are resolved and library searching is performed only once, when the program is "link-edited".
3	Executable image is not generated; the linked program is directly loaded into memory.	The linked programs is written into an executable image, which is later given to a relocating loader for execution.
4	It is more suitable for development and testing environment where the size of program is comparatively smaller.	It is suitable when the program has to be re-executed many times without being reassembled.
5	Produces output into memory.	Produces output on disc.
	E.g. Direct-Linking Loader	E.g. MS-DOS Linker

4.2 Design of an Absolute Loader:

With an absolute loading scheme, the programmer and the assembler perform the tasks of allocation, relocation, and linking. Therefore, it is only necessary for the loader to read cards of the object deck and move the text on the cards into the absolute locations specified by the assembler.

There are two types of information that the object deck must communicate from the assembler to the loader. First, it must convey the machine instructions that the assembler has created along with the assigned core locations. Second, it must convey the entry point of the program, which is where the loader is to transfer control when all instructions are loaded. Assuming that this information is transmitted on cards, a possible format is shown below.

Card Formats for an Absolute Loader:

Text Cards (for instructions and data)

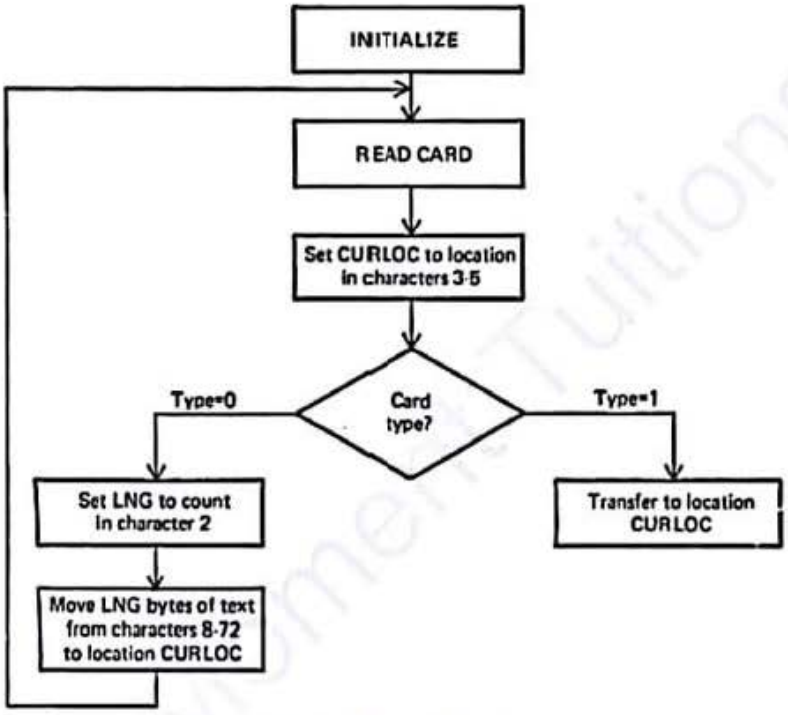
Card Column	Contents
1	Card type = 0 (for text card identifier)
2	Count of number of bytes (1 byte per column) of information on card
3-5	Addresses at which data on card is to be put
6-7	Empty (could be used for validity checking)
8-72	Instructions and data to be loaded
73-80	Card sequence number

Transfer Cards (to hold entry point to program)

Card Column	Contents
1	Card type = 1 (transfer card identifier)
2	Count = 0
3-5	Addresses of entry point
6-72	Empty
73-80	Card sequence number

Note that in the card format shown above, the instructions are stored on the card as one core byte per column. For each of the 256 possible contents of an eight-bit byte there is a corresponding punched card code (e.g., hexadecimal 00 is a column punched with five holes, 12-0-1-8-9, whereas a hexadecimal F1 is a column with a single punch in row 1). Thus, when a card is read, it is stored in core as 80 contiguous bytes.

The algorithm for an absolute loader is quite simple. The object deck for this loader consists of a series of text cards terminated by a transfer card. Therefore, the loader should read one card at a time, moving the text to the location specified on the card, until the transfer card is reached. At this point, the assembled instructions are in core, and it is only necessary to transfer to the entry point specified on the transfer card. A flowchart for this process is given below.



Flowchart of Absolute Loader

Bootstrap Loader (Type of Absolute Loader):

Alternatively referred to as **bootstrapping**, **bootloader**, or **boot program**, a **bootstrap loader** is a program that resides in the computer's EPROM, ROM, or other non-volatile memory. It is automatically executed by the processor when turning on the computer. The bootstrap loader reads the hard drives boot sector to continue the process of loading the computer's operating system. The term bootstrap comes from the old phrase "Pull yourself up by your bootstraps".

The bootstrap loader is stored in the master boot record (MBR) on the computer's hard drive. When the computer is turned on or restarted, it first performs the power-on self-test, also known as POST. If the POST is successful and no issues are found, the bootstrap loader will load the operating system for the computer into memory. The computer will then be able to quickly access, load, and run the operating system.

Function of Bootstrap Loader:

1. Enable the user to select the OS to start
2. Loading the OS file from the boot partition
3. Controls the OS selection process and hardware detection prior to kernel initialization.

Module 04: Loaders and Linkers

4.3 Design of a Direct Linking Loader (DLL):

The direct linking loader is the most common type of loader. This type of loader is a re-locatable loader. The loader cannot have the direct access to the source code. And to place the object code in the memory there are two situations: either the address of the object code could be absolute which then can be directly placed at the specified location or the address can be relative. If at all the address is relative, then it is the assembler who informs the loader about the relative addresses.

Specification of Problem:

The assembler should give the following information to the loader

- The length of the object code segment
- The list of all the symbols which are not defined in the current segment but can be used in the current segment.
- The list of all the symbols which are defined in the current segment but can be referred by the other segments.

Format of Data Bases (Object Deck, GEST, LESA):

OBJECT DECK (Assembled Code):

Direct Linking Loader uses four types of records in the object decks (files) which contains all information needed for relocation and linking. These four sections are:

1. **External Symbol Dictionary cards (ESD):**

The ESD cards contain the information necessary to build the external symbol dictionary or symbol table.

ESD cards

Reference no.	*Symbol*	*Type*	*Relative location*	*Length*
1	JOHN	SD	0	64
2	RESULT	LD	52	—
3	SUM	ER	—	—

There are three types of external symbols:

a. **Segment Definition (SD):** name on START or CSECT card.
b. **Local Definition (LD):** specified on ENTRY card. There must be a label in same program with same name.
c. **External Reference (ER):** specified on EXTRN card. There must be a corresponding ENTRY, START, or CSECT card. In another program with same name.

Each SD and ER symbol is assigned a unique number (e.g., 1, 2, 3 ...) by the assembler. This number is called the symbol's identifier, or ID, and is used in conjunction with the RLD cards.

2. Instructions and data cards, called "text" of program (TXT):

The TXT cards contain blocks of data and the relative address at which the data is to be placed. Once the loader has decided where to load the program, it merely adds the Program Load Address (PLA) to the relative address and moves the data into the resulting location. The data on the TXT card may be instructions, non-relocated data, or initial values of address constants.

TXT cards

Reference no.	*Relative location*	*Object code*	
4	0	BALR	12,0
6	2	ST	14,54(0,12)
7	6	L	1,46(0,12)
8	10	L	15,58(0,12)
9	14	BALR	14,15
10	16	ST	1,50(0,12)
11	20	L	14,54(0,12)
12	24	BCR	15,14
13	28	1	

3. Relocation and Linkage Directory cards (RLD):

The RLD cards contain the following information:

a. The location and length of each address constant that needs to be changed for relocation or linking

b. The external symbol by which the address constant should be modified (Added or subtracted)

c. The operation to be performed (add or subtract)

ID	*Flag*	*Length*	*Rel. loc.*
01	+	4	52
02	+	4	56

RLD Card

Rather than using the actual external symbol's name on the RLD card, the external symbol's identifier, or ID, is used. There are various reasons for this, the major one probably being that the ID is only a single byte long, compared to the eight bytes occupied by the symbol name, so that a considerable amount of space is saved on the RLD cards.

4. End card (END):

The END card specifies the end of the object deck. If the assembler END card has a symbol in the operand field, it specifies a start of execution point for the entire program (all subroutines). This address is recorded on the END card.

5. End of File (EOF)/ Loader Terminate(LDT):

There is a final card required to specify the end of a collection of object decks. The 360 loaders usually use either a loader terminate (LDT) or End of File (EOF) card.

Subroutine A	ESD TXT RLD END
Subroutine B	ESD TXT RLD END
Subroutine C	ESD TXT RLD END
	EOF or LDT

GLOBAL EXTERNAL SYMBOL TABLE (GEST)

The Global External Symbol Table (GEST) is used to store the external symbols defined by means of a Segment Definition (SD) or Local Definition (LD) entry on an External Symbol Dictionary (ESD) card. When these symbols are encountered during pass 1, they are assigned an absolute core address; this address is stored, along with the symbol, in the GEST.

← 12 bytes per entry →

External symbol (8-bytes) (characters)	Assigned core address (4-bytes) (decimal)
"PG1*bbbbb*"	104
"PG1ENT1*b*"	124
"PG1ENT2*b*"	134
"PG2*bbbbb*"	168
"PG2ENT1*b*"	184

Global External Symbol Table (GEST) format

LOCAL EXTERNAL SYMBOL ARRAY (LESA)

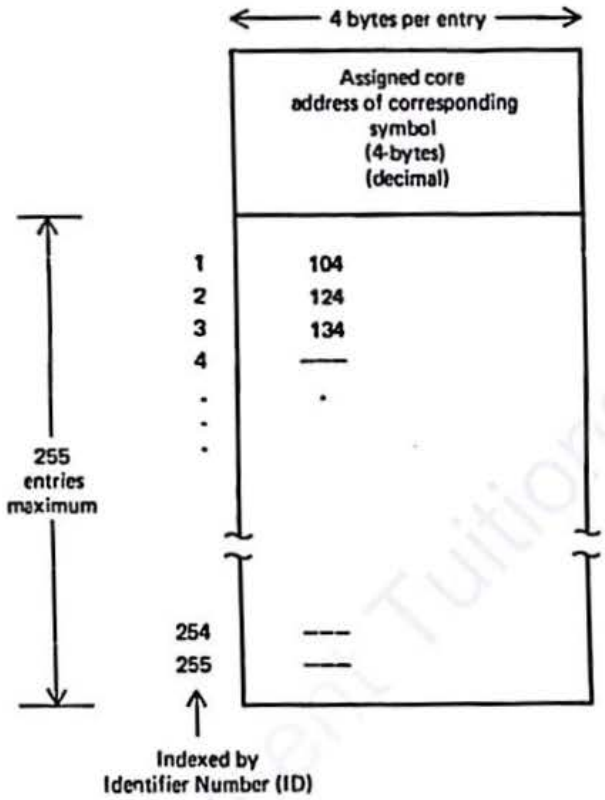
The external symbol to be used for relocation or linking is identified on the RLD cards by means of an ID number rather than the symbol's name. The ID number must match an SD or ER entry on the ESD cards. This technique both saves space on the RLD cards and speeds processing by eliminating many searches of the Global External Symbol Table. It is necessary to establish a correspondence between an ID number on an RLD card and the absolute core address value. The ESD cards contain the ID numbers and the symbols they correspond to, while the information relating these symbols to absolute core address values may be found in the GEST.

In pass 2 of the loader the GEST and ESD information for each individual object deck is merged to produce the local external symbol array that directly relates ID number and value. In principle it is necessary to create a separate LESA for each segment, but since the LESAs are only produced one at a time, the same array can be reused for each segment. It is not necessary to search the LESA; given an ID number, the corresponding value is written as LESA (ID) and can be immediately obtained.

... relating these symbols to absolute core address values may be found in the GEST.

In pass 2 of the loader the GEST and ESD information for each individual object deck is merged to produce the local external symbol array that directly relates ID number and value. In principle it is necessary to create a separate LESA for each segment, but since the LESAs are only produced one at a time, the same array can be reused for each segment. It is not necessary to search the LESA; given an ID number, the corresponding value is written as LESA (ID) and can be immediately obtained.

Module 04: Loaders and Linkers



Local External Symbol Array (LESA) format

Specification of Data Structures:

Pass 1 data bases:

1. Input object decks.
2. A parameter, the Initial Program Load Address (IPLA) supplied by the programmer or the operating system that specifies the address to load the first segment.
3. A Program Load Address (PLA) counter, used to keep track of each segment's assigned location.
4. A table, the Global External Symbol Table (GEST), that is used to store each external symbol and its corresponding assigned core address.
5. A copy of the input to be used later by pass 2. This may be stored on an auxiliary storage device, such as magnetic tape, disk, or drum, or the original object decks may be reread by the loader a second time for pass 2.
6. A printed listing, the load map that specifies each external symbol and its assigned value

Module 04: Loaders and Linkers

Pass 2 data bases:

1. Copy of object programs inputted to pass 1.
2. The Initial Program Load Address parameter (IPLA).
3. The Program Load Address counter (PLA).
4. The Global External Symbol Table (GEST), prepared by pass 1, containing each external symbol and its corresponding absolute address value.
5. An array, the Local External Symbol Array (L.ESA), which is used to establish a correspondence between the ESD ID numbers, used on ESD and RLD cards, and the corresponding external symbol's absolute address value.

Algorithm:

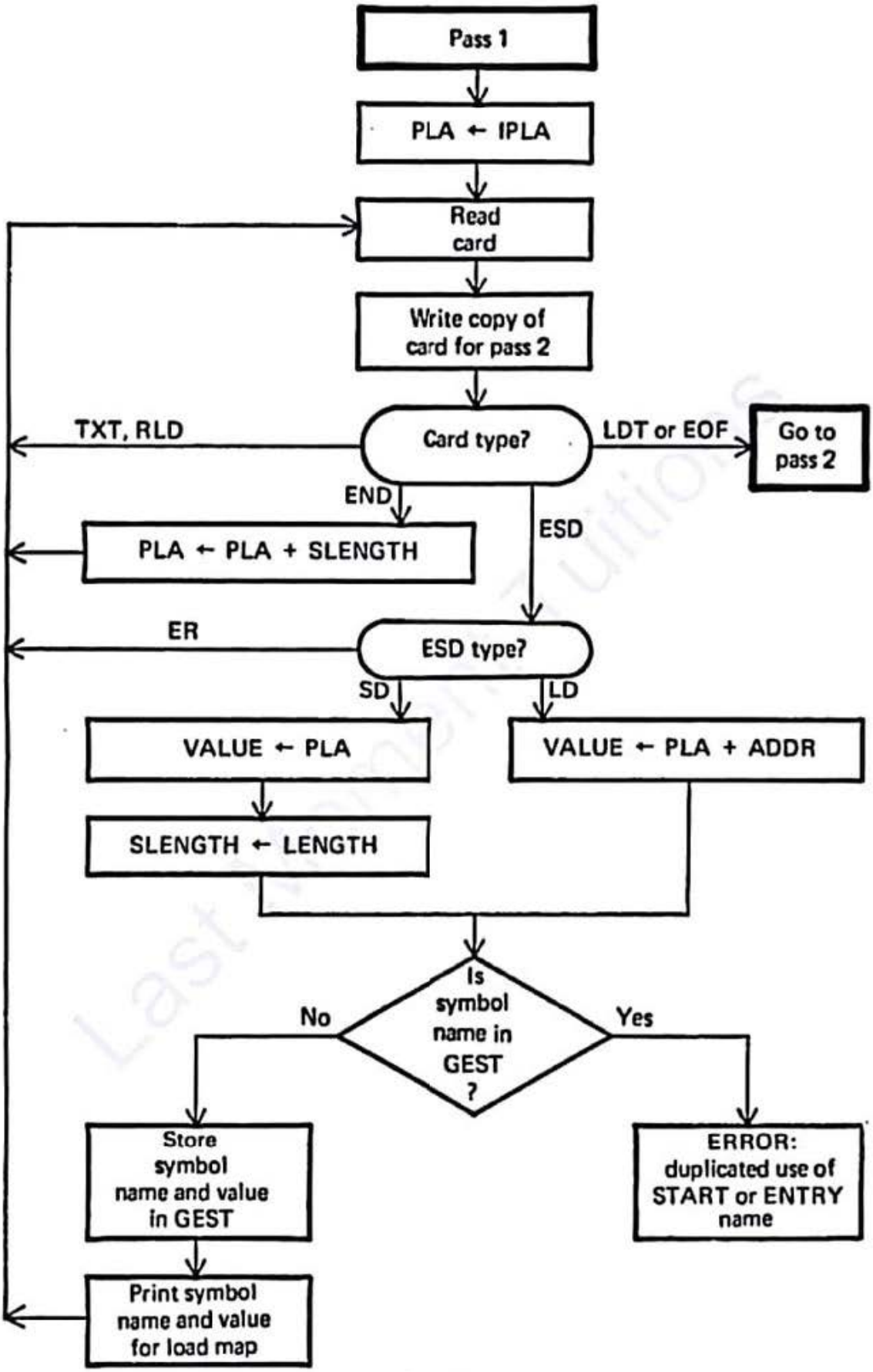
Pass 1 - Allocate segments and define symbols

The purpose of the first pass is to assign a location to each segment, and thus to define the values of all external symbols. It is necessary for the loader to know where it can load the first segment. This address, the Initial Program Load Address (IPLA), is normally determined by the operating system.

Initially, the Program Load Address (PLA) is set to the Initial Program Load Address (IPLA). An object card is then read and a copy written for use by pass 2. The card can be one of five types, ESD, TXT, RLD, END, or LDT/EOF. If it is a TXT or RLD card, then no processing required during pass 1, so the next card is read.

An ESD card is processed in different ways depending upon the type of external symbol, SD, LD or ER. If a segment definition ESD card is read, the length field, LENGTH, from the card is temporarily saved in the variable, SLENGTH. The value, VALUE, to be assigned to this symbol is set to the current value of the PLA. The symbol and its assigned value are then stored in the GEST; if the symbol already existed in the GEST, there must have been a previous SD or LD, ESD with the same name – this is an error. The symbol and its value are printed as part of the load map. A similar process is used for LD symbols; the value to be assigned is set to the current PLA plus the relative address, ADDR, indicated on the ESD card. The ER symbols do not require any processing during pass 1.

When an END card is encountered, the program load address is incremented by the length of the segment and saved in SLENGTH, becoming the PLA for the next segment. When the LDT or EOF card is finally read, pass 1 is completed and control transfers to pass 2.


Pass 1
PLA ← IPLA
Read card
Write copy of card for pass 2
Card type?
TXT, RLD
LDT or EOF
Go to pass 2
END
ESD
PLA ← PLA + SLENGTH
ESD type?
ER
SD
LD
VALUE ← PLA
VALUE ← PLA + ADDR
SLENGTH ← LENGTH
Is symbol name in GEST ?
No
Yes
Store symbol name and value in GEST
ERROR: duplicated use of START or ENTRY name
Print symbol name and value for load map

Pass 1

Pass 2 - load text and relocate/link address constants

After all the segments have been assigned locations and the external symbols have been defined by pass 1, it is possible to complete the loading by loading the text and adjusting (relocation or linking) address constants.

At the end of pass 2, the loader will transfer control to the loaded program. The following simple rule is often used to determine where to commence execution:

1. If an address is specified on the END card, that address is used as the execution start address:
2. Otherwise, execution will commence at the beginning of the first segment.

At the beginning of pass 2 the program load address is initialized as in pass 1, and the execution start address (EXADDR) is set to IPLA. The cards are read one by one from the object deck file left by pass 1. Each of the five types of cards is processed differently, as follows:

ESD CARD

Each of the ESD card types is processed differently.

- **SD-type ESD:** The LENGTH of the segment is temporarily saved in the variable SLENGTH. The appropriate entry in the local external symbol array, LESA (ID), is set to the current value of the Program Load Address.
- **LD-type ESD:** The LD does not require any processing during pass 2.
- **ER-type ESD:** The Global External Symbol Table (GEST is searched for a match with the ER symbol. If it is not found, the corresponding segment or entry must be missing - this is an error. If the symbol is found in the GEST, its value is extracted and the corresponding Local External Symbol Array entry, LESA (ID), is set equal to it.

TXT CARD:

When a TXT card is read, the text is copied from the card to the appropriate relocated core location (PLA + ADDR).

RLD CARD:

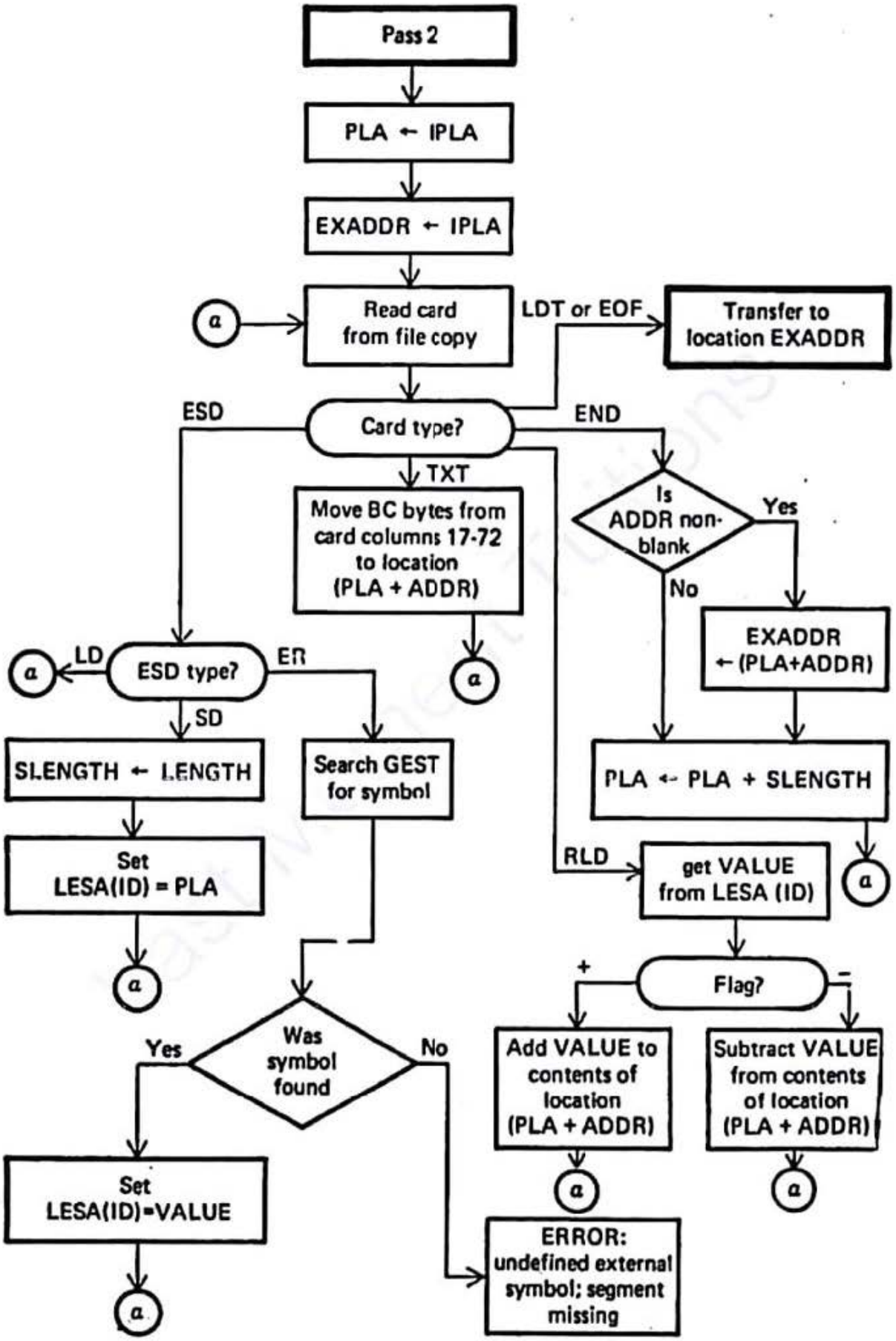
The value to be used for relocation and linking is extracted from the local external symbol array as specified by the ID field, i.e., LESA (ID). Depending upon the flag setting (plus or minus) the value is either added to or subtracted from the address constant. The actual relocated address of the address constant is computed as the sum of the PLA and the ADDR field specified on the RLD card.

END CARD:

If an execution start address is specified on the END card, it is saved in the variable EXADDR after being relocated by the PLA. The Program Load Address is incremented by the length of the segment and saved in SLENGTH, becoming the PLA for the next segment.

LDT/EOF CARD:

The loader transfers control to the loaded program at the address specified by current contents of the execution address variable (EXADDR).


Pass 2
PLA ← IPLA
EXADDR ← IPLA
a
Read card from file copy
LDT or EOF
Transfer to location EXADDR
ESD
Card type?
END
TXT
Move BC bytes from card columns 17-72 to location (PLA + ADDR)
Is ADDR non-blank
Yes
No
EXADDR ← (PLA+ADDR)
a
LD
ESD type?
ER
a
SD
SLENGTH ← LENGTH
Search GEST for symbol
PLA ← PLA + SLENGTH
Set LESA(ID) = PLA
RLD
get VALUE from LESA (ID)
a
a
+
Flag?
−
Yes
Was symbol found
No
Add VALUE to contents of location (PLA + ADDR)
Subtract VALUE from contents of location (PLA + ADDR)
Set LESA(ID)=VALUE
a
a
ERROR: undefined external symbol; segment missing
a

Pass 2

Module 04: Loaders and Linkers

Example of Direct Linking Loader:

Source card reference	Relative address		Sample program (source deck)		
1	0	PG1	START		(a) Procedure PG1
2			ENTRY	PG1ENT1,PG1ENT2	
3			EXTRN	PG2ENT1,PG2	
4	20	PG1ENT1	≡		
5	30	PG1ENT2	≡		
6	40		DC	A(PG1ENT1)	
7	44		DC	A(PG1ENT2+15)	
8	48		DC	A(PG1ENT2-PG1ENT1-3)	
9	52		DC	A(PG2)	
10	56		DC	A(PG2ENT1+PG2-PG1ENT1+4)	
11			END		
12	0	PG2	START		(b) Procedure PG2
13			ENTRY	PG2ENT1	
14			EXTRN	PG1ENT1,PG1ENT2	
15	16	PG2ENT1	≡		
16	24		DC	A(PG1ENT1)	
17	28		DC	A(PG1ENT2+15)	
18	32		DC	A(PG1ENT2-PG1ENT1-3)	
19			END		

Sample procedures PG1 and PG2

ESD cards

Source card reference	Name	Type	ID	Relative address	Length
1	PG1	SD	01	0	60
2	PG1ENT1	LD	—	20	—
2	PG1ENT2	LD	—	30	—
3	PG2	ER	02	—	—
3	PG2ENT1	ER	03	—	—

TXT cards
(only the interesting ones, i.e. those involving address constants)

Source card reference	Relative address	Contents	Comments
6	40-43	20	
7	44-47	45	= 30 + 15
8	48-51	7	= 30-20-3
9	52-55	0	unknown to PG1
10	56-59	-16	= -20 + 4

RLD cards

Source card reference	ESD ID	Length (bytes)	Flag + or −	Relative address
6	01	4	+	40
7	01	4	+	44
9	02	4	+	52
10	03	4	+	56
10	02	4	+	56
10	01	4	−	56

Object deck program PG1

ESD cards

Source card reference	*Name*	*Type*	*ID*	*ADDR*	*Length*
12	PG2	SD	01	0	36
13	PG2ENT1	LD	—	16	—
14	PG1ENT1	ER	02	—	—
14	PG1ENT2	ER	03	—	—

TXT cards
(only the interesting ones)

Source card reference	*Relative address*	*Contents*
16	24-27	0
17	28-31	15
18	32-35	-3

RLD cards

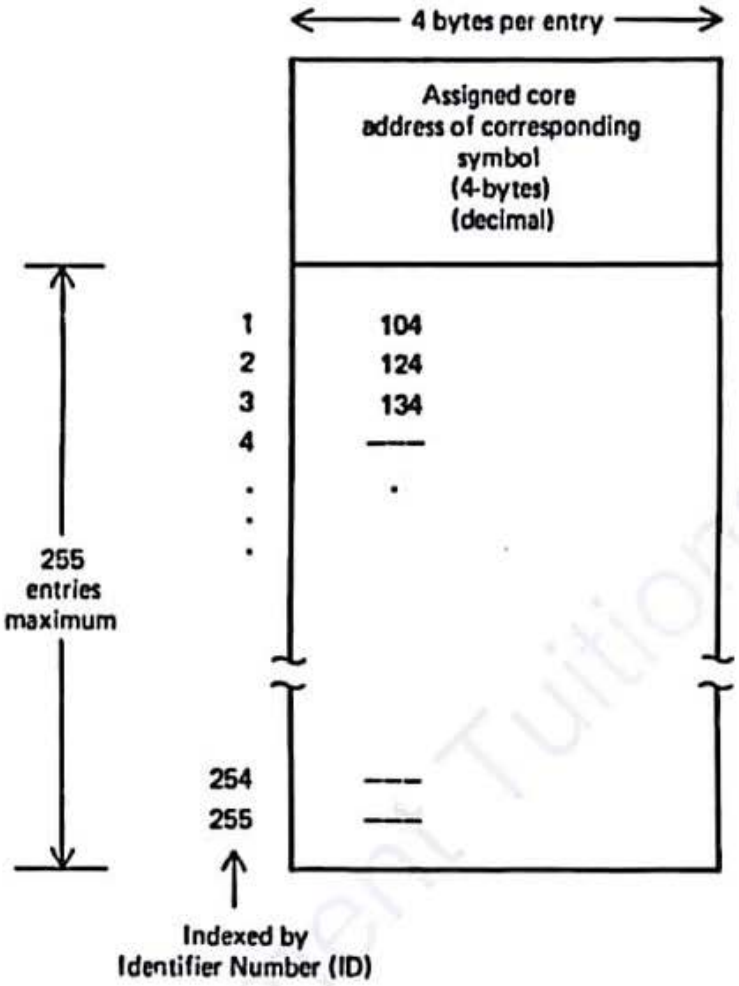
Source card reference	*ESD ID*	*Length flag (bytes)*	*Flag + or –*	*Relative address*
16	02	4	+	24
17	03	4	+	28
18	03	4	+	32
18	02	4	–	32

Object deck program PG2

← 12 bytes per entry →

External symbol (8-bytes) (characters)	Assigned core address (4-bytes) (decimal)
"PG1*bbbbb*"	104
"PG1ENT1*b*"	124
"PG1ENT2*b*"	134
"PG2*bbbbb*"	168
"PG2ENT1*b*"	184

Global External Symbol Table (GEST) format



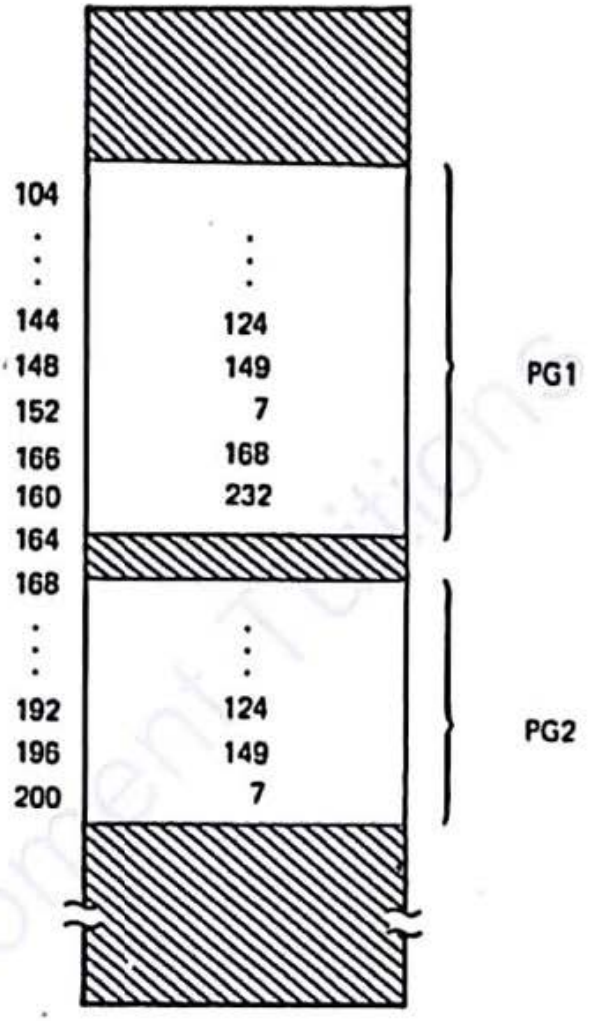
Local External Symbol Array (LESA) format

Last Moment Tuitions

Module 04: Loaders and Linkers

Assume:

PG1 loaded at location 104
PG2 loaded at location 168



Main storage after loading programs PG1 and PG2

Last Moment Tuitions

Editor

(Hindi)

Content

- What is Editor
- Types of Editor
- Architecture of Editor

Editor

- Editor, sometimes called as text editor, is a program that enables user to create and edit text files.
- The term editor usually refers to source code editors that include many special features for writing and editing source code

Editor (continue)

- A text editor allows you to edit a text file (create, modify etc.). For example, the Interactive text editors on Windows OS - Notepad, WordPad, Microsoft Word, and text editors on UNIX OS - vi, emacs, jed, pico
- Normally, the common editing features associated with text editors are, Moving the cursor, Deleting, Replacing, Pasting, Searching, Searching and replacing, Saving and, Miscellaneous

Types of Editor

1. **Line Editors**: In this you can type a single line or an integral number of lines. E.g. tele printer, edlin, TECO, QED.

2. **Stream Editors**: In this file is treated as continuous flow or sequence of characters instead of line number. E.g. Sed Editor in Unix.

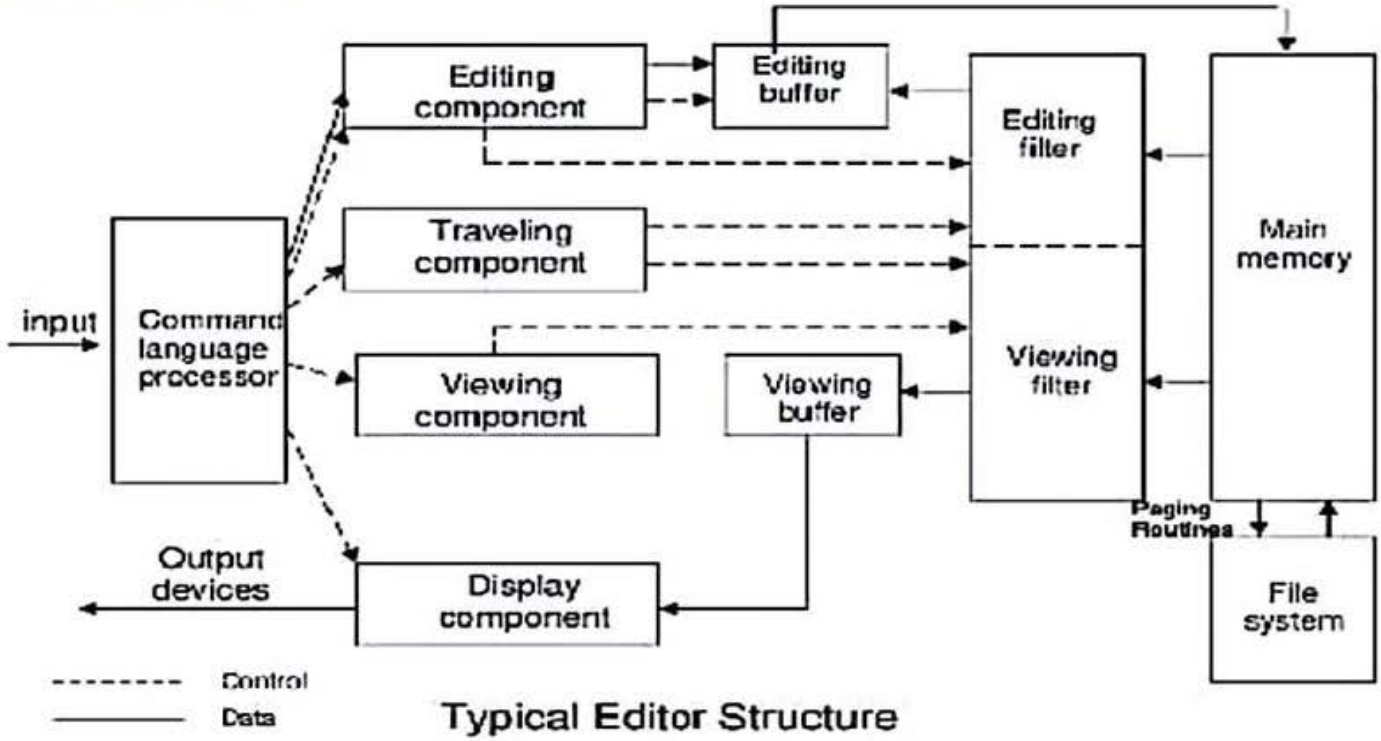
3. **Screen Editors:** In this user able to see cursor on screen and made copy, cut, paste operation easy, easy to use mouse pointer. E.g. Vi, emac ,Notepad

Types of Editor (continue)

4. **Word Processor**: Overcome limitation of screen editors uses of some format like including images in document, text support with various format, font, style, size etc. work on natural languages.

5. **Structure Editor**: focus on programming languages. E.g. gEdit, Net beans IDE.

Architecture



Typical Editor Structure

Compiler :

High level language → [compiler] → machine code

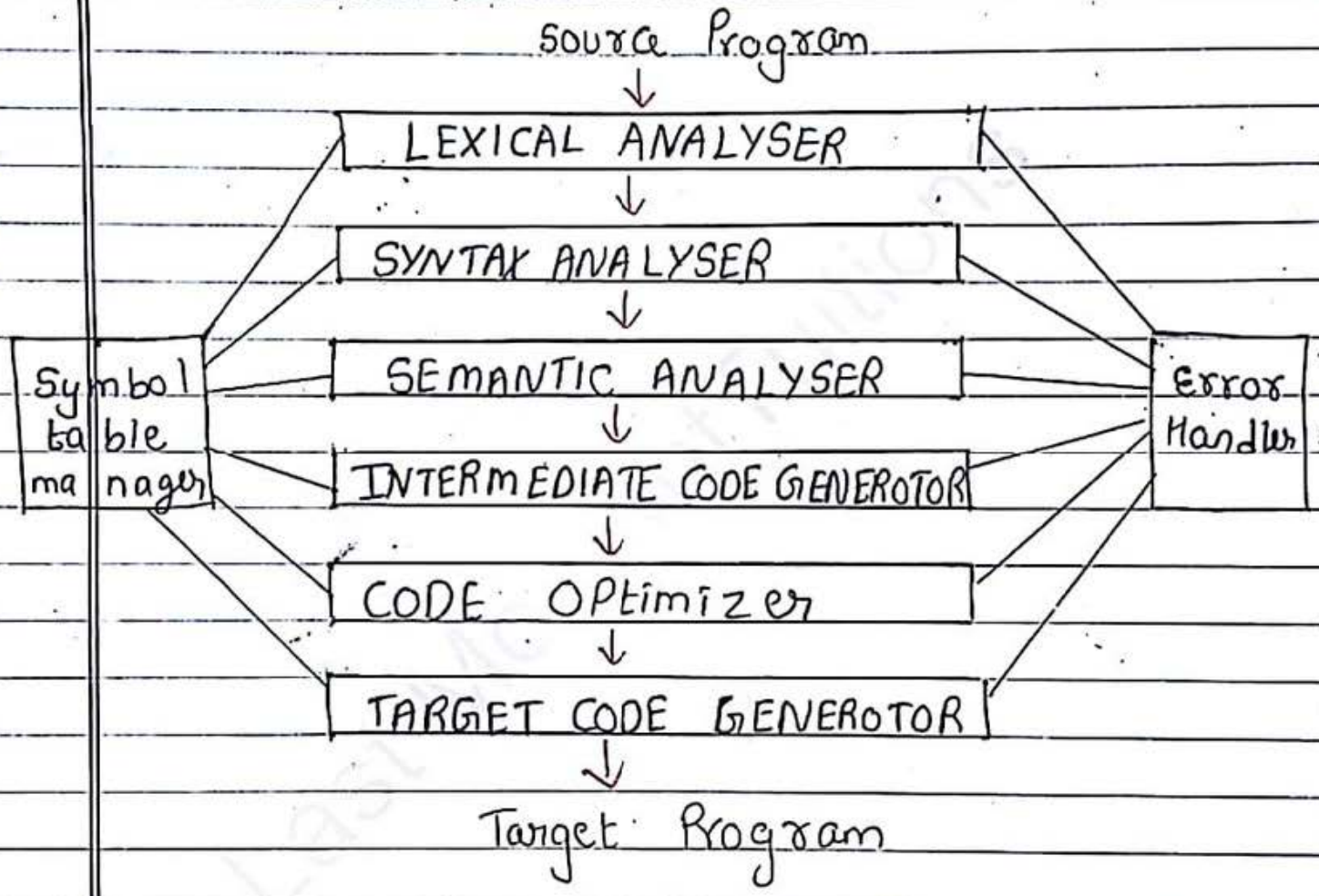
or

High level language → [compiler] → Assembly language

<u>Compiler</u> :- Toh compiler ek Translator hai jo high level language ko machine language me translate karta hai
english me exams ke liye

Compiler is a program that can read a program in one language and translate it into an equivalent Program in another language. The important role of the compiler is to report any errors in the source program that it detect during translation.

Note: Compiler ke andar 6 Phases hote hai jo translation ka kaam karte hai milke.



ab yeh 6 Phases kaise kaam karte hai chalo USKO detail me dekhte hai.

lexical analyser: yeh first Phase hai compiler ka jisme yeh input leta hai pattern check karta hai fir token output deta hai

Example:- $a = b + c * d$

$\Downarrow$

lexical analyser

$\Downarrow$

$id_1 = id_2 + id_3 * id_4$

Note:- i) Koi bhi naam ya variable hai toh usko token id hota hai

ii) reserved Keyword ka token reserved keyword hi hota hai.

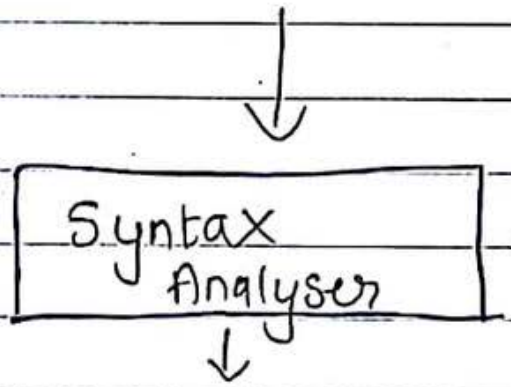
eg: if ka token if, else ka token else.

iii) saare operator ke token bhi same hi hote hai

$(+, -, *, /) \rightarrow$ Tokens $\rightarrow (+, -, *, /)$

Syntax Analyser: yeh token input leta hai grammer ke saath check karta hai fir usko agar sahi hai toh aage bhejta hai ya fir error thokta hai

Eg: $id_1 = id_2 + id_3 * id_4$



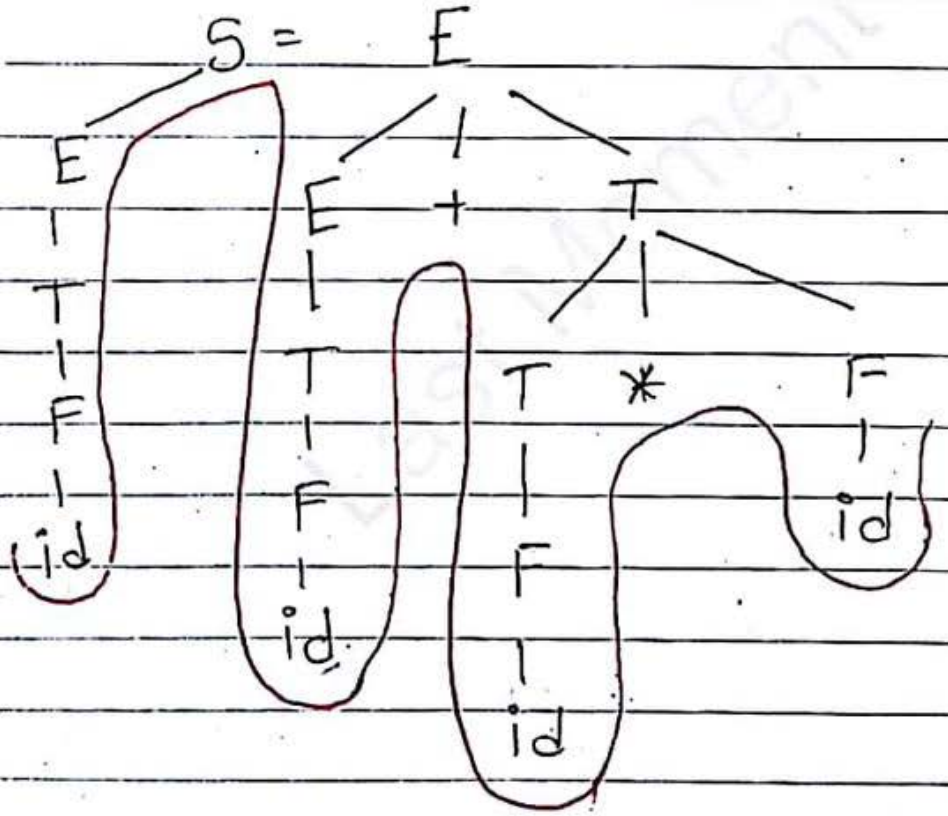
Grammer

S = E

E = E + T / T

T = T * F / F

F = id



Semantic analyser = yeh input me meaning dhundta hai ki koi matlab nikalta hai ke nahi code ka.

Eg :- $a = b + c$ ← yeh sahi it has some meaning

but

$c = a/0$ ab iska koi meaning nahi hai infinity me jayenga

intermediate code generator : bahut bade code ko chote chote tukdo me divide karta hai

code optimization : yeh bade code me se faltu ki cheez eliminate karta hai and code optimize karta hai

target code generator : actual machine code generate karta hai.

EXAMPLE

$$x = a + b * c$$

⇓

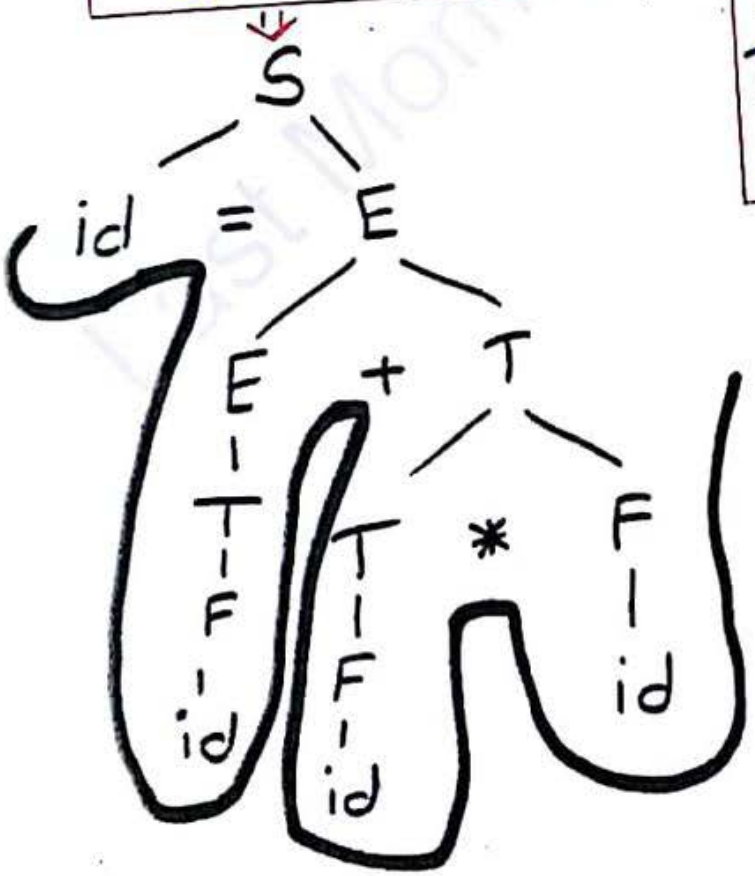
lexical analyser

⇓

$$id_1 = id_2 + id_3 * id_4$$

Syntax analyser ← $S \Rightarrow id = E$, $E = E + T / T$, $T = T * F / F$, $F = id$

⇓



Scanned by CamScanner

⇓

Semantic analyser

⇓

Parse tree semantically verified

⇓

intermediate code generator

⇓

$t_1 = b \times c$

$t_2 = a + t_1$

$x = t_2$

⇓

Code optimization

⇓

$t_1 = b \times c$

$x = a + t_1$

Scanned by CamScanner

$\Downarrow$

Target code Generator

$\Downarrow$

MUL	R_3, R_4	$id_1 = R_1$
Add	R_2, R_3	$id_2 = R_2$
EQU	R_1, R_2	$id_3 = R_3$
		$id_4 = R_4$

=

Last Moment Tuitions

Scanned by

Module 06: Compilers

Features of Compiler:

- The compiler should be bug free.
- The compiler should run fast.
- The compiler should generate correct machine code.
- The generated machine code should run fast.
- The compiler should have consistent and predictable optimizations.
- The compiled target code should work well with the existing debuggers.
- Compiler should be efficient enough to print error messages.
- A good compiler should be portable.
- The compiler should be easy and simple to implement.

Compiler Requirements:

a. **Correctness:** It is absolutely paramount. A buggy compiler is next to useless. The correctness of the compiler means that the semantics of the source language and the target language should be well defined and understood by the users and the programmers.
b. **Efficiency:** The compiler should be efficient enough to generate the target code and must be able to respond to changes in the source language and target language.
c. **Interoperability:** Compiler should be able to link all the files with the library code before they are executed.
d. **Usability:** A compiler interacts with the programmer primarily when there are errors in the program. Compiler should be able enough to generate messages and it should also help users to debug runtime errors during program execution.
e. **Retargetability:** Compilers should be able to generate more than one target from a given source code.

Types of Compiler:

1. **One Pass Compiler:** A one pass compiler is a type of compiler that passes through the source code of each compilation unit only once. Some programming languages like PASCAL have been designed to be compiled with the one pass compilers. Normally PASCAL requires that the procedures should be fully defined before used.

2. **Multi-pass Compiler:** A multi-pass compiler is a type of compiler that processes the source code of program several times. In this the entire program is compiled instead of just small portion of a program. The typical multi-pass compiler outputs machine code from its final pass. The multi-pass will forward its output to the next pass as it will act as the input for further processing for other passes.

3. **Load and Go compiler:** A load and go compiler compiles directly into memory and immediately runs the code.

4. **Optimizing Compiler:** They optimize the compiled code the translated code, using during the code generation phase. They improve the performance by:
   - Reducing the number of instructions.
   - Replacing the number of instructions.
   - Reduce latency of cache/ memory access.
   - Other than these there are also various types of compilers.

5. **Source to Source Compiler:** It is a type of compiler which takes a high level language as its input and outputs a high level language.

6. **Native Compiler:** It is also known as hosted compiler whose output is intended directly to run on the same type of computer and operating system as the compiler itself runs on.

7. **Cross Compiler:** When a compiler runs on one machine and produces target code for another machine. This kind of compilers is known as cross compiler. These are widely used for mini and microcomputers.

Compiler Construction Tools:

Some general tools have been created for the automatic design of specific compiler components. Below mentioned are some compiler construction tools.

1. **Parser Generators:** These generators produce syntax analysers, normally form input that is based on a context-tree grammar.
2. **Scanner Generators:** These Scanner Generators automatically generate lexical analysers, normally from a specification based on regular expressions.
3. **Syntax Directed Translation Engines:** These engines produce collection of routines that walk the parse tree generating intermediate code.
4. **Automatic Code Generators:** Such tools take a collection of rules that define the translation of each operation of the intermediate languages into the machine language for the target machine.
5. **Data-flow Engines:** Much of the information needed to perform good code optimization involves "data-flow analysis" i.e. the gathering of information about how values are transmitted form one part of a program to each other.

Interpreter:

An interpreter is a kind of translator which produces the result directly when the source language and data is given to it as input. The process of interpretation can be carried out in the following phases:

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Direct Execution

Examples of Interpreted languages are: BASIC, SNOBO and LISP.

Advantages:

- Modification of user program can be done easily made and implemented as execution proceeds.
- Type of object that denotes a variable may change dynamically.
- Debugging a program and finding errors is simplified task for a program used for interpretation.

Disadvantages:

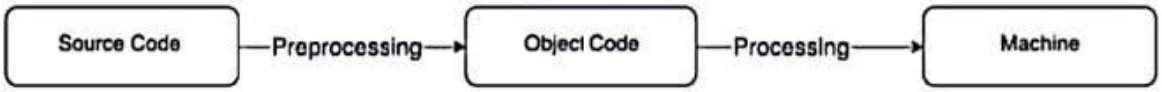
- The execution of the program is slower.
- Memory consumption is more.

Module 06: Compilers

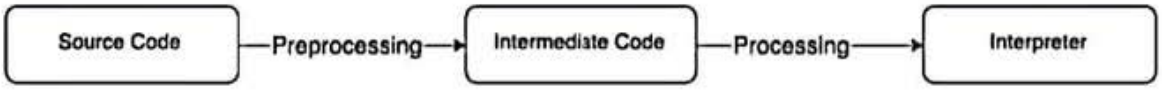
6.3 Comparison of Compilers and Interpreters:

	Compilers	Interpreters
Definition	A compiler converts the source high level language program into a target program, which may be assembly code or machine code	An interpreter converts the source high level language program into machine code line-by-line and executes it if there are no errors.
Translation Approach	The entire program is translated at once.	Translation proceeds line-by-line.
Object File/Code	It creates an object file/code.	It does not create an object file/code.
Error Reporting	All compile-time errors are notified collectively to the programmer.	If there is any error during interpretation, it is reported immediately to the programmer and remaining source code lines are not processed.
Avoiding Chain of Errors	Chain of errors cannot be avoided.	Since an error is immediately reported to the programmer, the programmer can correct the error and avoid chain of errors.
Translation for each Run	The program is translated once and can be executed repeatedly.	Even if no changes are, the program need to be translated every time before executing it.
Speed of Execution	Speed of execution is faster than an Interpreter.	Since it goes through all phases of compilation separately for each line, the speed of execution is much slower.
Conditional Statements	Compiler executed conditional control statement (like if-else and switch-case) and logical constructs faster than interpreter.	Interpreter execute conditional control statement at a much slower speed.
Memory Usage	Compiled programs take more memory because the entire object code has to reside in memory.	Interpreter does not generate intermediate object code. As a result, interpreted programs are more memory efficient.
Debugging	A compiled language is more difficult to debug.	Debugging is easy because interpreter stops and reports errors as it encounters them.
Examples	Examples of programming languages that uses compilers: C, C++	Examples of programming languages that use interpreters: BASIC, Python, Matlab.

Module 06: Compilers



Compiler



Interpreter

Question:
Java - Compiled or interpreted?
Answer:

Java is both compiled as well as interpreted language.

Working of JAVA:

When user runs "javac HelloWorld.java", java compiler is invoked which converts human readable code (Contents of .java file) to java byte codes (intermediate form). These bytecodes are stored in a special file (called Class file) with .class extension.

Finally, when user runs "java HelloWorld", java interpreter is invoked which reads these bytecodes line by line, convert it into machine language and execute it. This is why Java is called as both compiled as well as interpreted language.

Module 07: Lexical Analysis

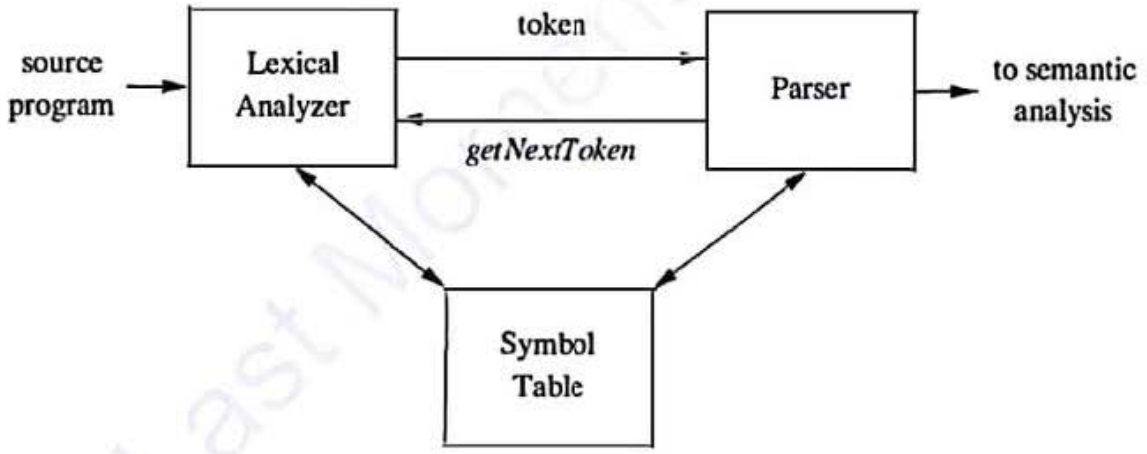
Contents:
Role of a Lexical analyser, input buffering, specification and recognition of tokens, designing a lexical analyser generator, Pattern matching based on NFA's.

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyser or scanner. A Lexical Analyser often exists as a single function which is called by a parser or another function.

7.1 Role of a Lexical Analyser:

The lexical analyser is the first phase of a compiler. The main task of the lexical analyser is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyser to interact with the symbol table as well. When the lexical analyser discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. In some cases, information regarding the kind of identifier may be read from the symbol table by the lexical analyser to assist it in determining the proper token it must pass to the parser.

Commonly, the interaction is implemented by having the parser call the lexical analyser. The call, suggested by the *getNextToken* command, causes the lexical analyser to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.



Interactions between the lexical analyzer and the parser

Since the lexical analyser is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyser may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyser makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyser.

Module 07: Lexical Analysis

Sometimes, lexical analysers are divided into a cascade of two processes:

a. **Scanning** consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
b. **Lexical analysis** proper is the more complex portion, where the scanner produces the sequence of tokens as output.

Issues of Lexical Analyser

There are three issues in lexical analysis:

- To make the design simpler.
- To improve the efficiency of the compiler.
- To enhance the computer portability.

Lexical Analysis v/s Parsing:

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

1. Simplicity of design is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be, considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyser. If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.
2. Compiler efficiency is improved. A separate lexical analyser allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
3. Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyser.

Tokens, Patterns, and Lexemes:

- A **token** is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

- A **pattern** is a description of the form that the lexemes of a token may take. In the case of a keyword as a token; the pattern is just the sequence of characters that form the keyword.

- A **lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyser as an instance of that token.

Module 07: Lexical Analysis

Token	Informal Description	Sample Lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letters and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but ", surrounded by "s	"core dumped"

Examples of Tokens

In many programming languages, the following classes cover most or all of the tokens:

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token comparison.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal strings.
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Attributes for Tokens:

When more than one lexeme can match a pattern, the lexical analyser must provide the subsequent compiler phases with additional information about the particular lexeme that matched.

For example, the pattern for token **number** matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyser returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse.

The most important example is the token **id**, where we need to associate with the token a great deal of information. Normally, information about an identifier- e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) - is kept in the symbol table. Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

Example: The token names and associated attribute values for the FORTRAN statement

E = M * C ** 2

are written below as a sequence of pairs.

<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>

Module 07: Lexical Analysis

Error Recovery Strategies in Lexical Analysis:
The following are the error-recovery actions in lexical analysis:

- Deleting an extraneous character.
- Inserting a missing character.
- Replacing an incorrect character by a correct character.
- Transforming two adjacent characters.
- Panic mode recovery: Deletion of successive characters from the token until error is resolved.

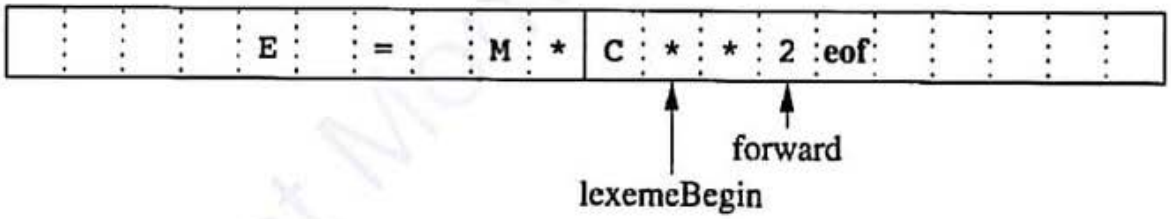
7.2 Input Buffering:

Reading the source program is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. There are many situations where we need to look at least one additional character ahead.

For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id. In C, single-character operators like - , =, or < could also be the beginning of a two-character operator like - >, ==, or <=. Thus, a two-buffer scheme is introduced to handle large lookaheads safely.

Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded.



Using a pair of input buffers

Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters into a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file and is different from any possible character of the source program.

Two pointers to the input are maintained:

a. Pointer **lexemeBegin** marks the beginning of the current lexeme, whose extent we are attempting to determine.
b. Pointer **forward** scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, *lexemeBegin* is set to the character immediately after the lexeme just found. In the above figure, we see forward has passed the end of the next lexeme, * * (the FORTRAN exponentiation operator), and must be retracted one position to its left.

Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

Sentinels:

An improvement in the above method involves "**sentinels**" that saves time checking for the ends of buffers. We must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multi way branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**.

Note that **eof** retains its use as a marker for the end of the entire input. Any **eof** that appears other than at the end of a buffer means that the input is at an end.

7.3 Specification of Tokens:

Regular expressions are an important notation for specifying lexeme patterns. They are very effective in specifying those types of patterns that we actually need for tokens.

Alphabets, Strings and Languages:

An **alphabet** is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set {0, 1} is the binary alphabet. ASCII is an important example of an alphabet. Unicode, which includes approximately 100,000 characters from alphabets around the world, is another important example of an alphabet.

A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string s, usually written $|s|$, is the number 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.

A **language** is any countable set of strings over some fixed alphabet. Abstract languages like ø, the empty set, or $\{\varepsilon\}$, the set containing only the empty string, are languages under this definition. Note that the definition of "language" does not require that any "meaning" be ascribed to the strings in the language.

A **prefix** of string S is any string obtained by removing zero or more symbols from the end of s. For example, *ban, banana,* and ε are prefixes of *banana*.

A **suffix** of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, *nana, banana* and ε are suffixes of *banana*.

Module 07: Lexical Analysis

A **substring** of s is obtained by deleting any prefix and any suffix from s. For instance, *banana*, *nan* and ε are substrings of *banana*.

The **proper** prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not ε or not equal to s itself.

A **subsequence** of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, *baan* is a subsequence of *banana*.

Operations on Languages:

In lexical analysis, the most important operations on languages are union, concatenation, and closure.

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Definitions of operations on languages

Regular Expressions:

We are able to describe identifiers by giving names to sets of letters and digits and using the language operators: union, concatenation and closure. This process is so useful that a notation called regular expressions has come into common use for describing all the languages that can be built from these operators applied to the symbols of some alphabet. In this notation, if *letter_* is established to stand for any letter or the underscore, and *digit* is established to stand for any digit, then we could describe the language of identifiers by:

*letter_*(letter_|*digit*)*

The vertical bar above means union, the parentheses are used to group sub expressions, the star means "zero or more occurrences of," and the juxtaposition of *letter_* with the remainder of the expression signifies concatenation. The regular expressions are built recursively out of smaller regular expressions. Each regular expression r denotes a language L(r), which is also defined recursively from the languages, denoted by r's sub expressions.

Regular Definitions:

For notational convenience, we may wish to give names of certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If $\sum$ is an alphabet of basic symbols, then the regular definition is a sequence of definitions of the form:

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\dots$$
$$d_n \rightarrow r_n$$

where:

1. Each d_i is a new symbol, not in $\sum$ and not the same as any other of the d's, and
2. Each r_i is a regular expression over the alphabet $\sum \cup \{d_1, d_2, \dots, d_{i-1}\}$.

By restricting r_i to $\sum$ and the previously defined d's, we avoid recursive definitions, and we can construct a regular expression over $\sum$ alone, for each r_i. We do so by first replacing uses of d_1 in r_2 (which cannot use any of the d's except for d_1), then replacing uses of d_1 and d_2 in r_3 by r_1 and (the substituted) r_2, and so on. Finally, in rn we replace each d_i of i = 1, 2, …, n-1, by the substituted version of r_i, each of which has only symbols of $\sum$.

Shorthands:

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

One or more instances (+):

- The unary postfix operator + means "one or more instances of".
- If r is a regular expression that denotes the language L(r), then $(r)^+$ is a regular expression that denotes the language $(L(r))^+$.
- Thus the regular expression a^+ denotes the set of all strings of one or more a's.
- The operator + has the same precedence and associativity as the operator *.

Zero or one instance (?):

- The unary postfix operator ? means "zero or one instance of".
- The notation r? is a shorthand for r | ε.
- If 'r' is a regular expression, then (r)? is a regular expression that denotes the language L(r) U {ε}.

Character Classes:

- The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.
- Character class such as [a – z] denotes the regular expression a | b | c | d | …. |z.
- We can describe identifiers as being strings generated by the regular expression, [A–Za–z][A–Za–z0–9]*

7.4 Recognition of Tokens:

In the previous section we learned how to express patterns using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

Consider the given grammar below:

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \epsilon \\
expr & \rightarrow & term \textbf{ relop } term \\
 & | & term \\
term & \rightarrow & \textbf{id} \\
 & | & \textbf{number}
\end{array}
$$

A grammar for branching statements

The grammar fragment in above figure describes a simple form of branching statements and conditional expressions. This syntax is similar to that of the language Pascal, in that then appears explicitly after conditions.

For **relop**, we use the comparison operators of languages like Pascal or SQL, where = is "equals" and <> is "not equals," because it presents an interesting structure of lexemes.

The terminals of the grammar, which are **if, then, else, relop, id,** and **number** are the names of tokens as far as the lexical analyser is concerned. The patterns for these tokens are described using regular definitions.

digit	→	[0-9]
digits	→	*digit*$^+$
number	→	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	→	[A-Za-z]
id	→	*letter* (*letter* | *digit*)*
if	→	if
then	→	then
else	→	else
relop	→	< | > | <= | >= | = | <>

Patterns for tokens

Transition Diagrams:

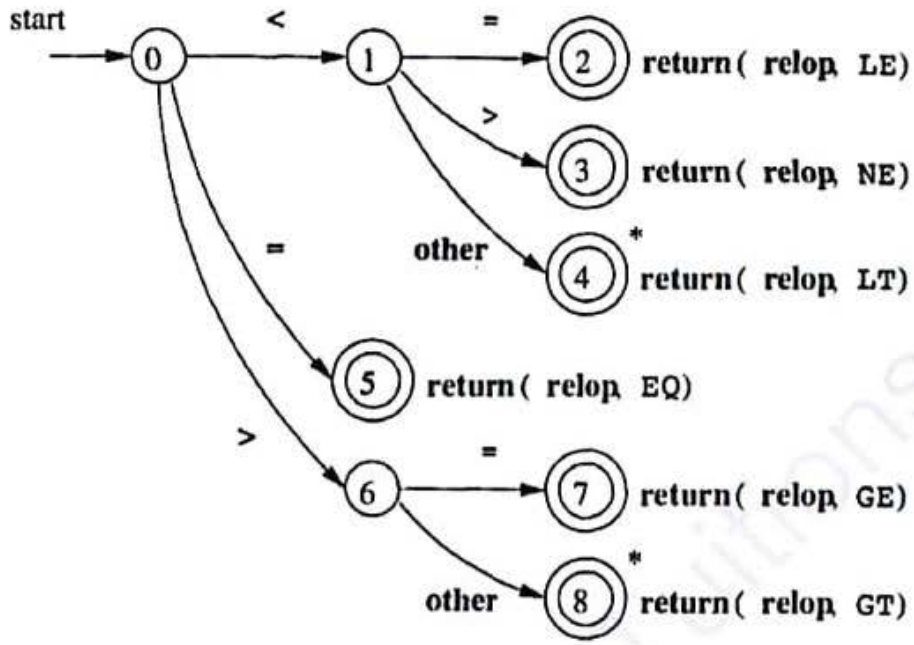
As an intermediate step in the construction of a lexical analyser, we first convert patterns into stylized flowcharts, called "transition diagrams". Transition diagrams have a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. We may think of a state as summarizing all we need to know about what characters we have seen between the *lexemeBegin* pointer and the forward pointer.

Edges are directed from one state of the transition diagram to another. Each edge is labelled by a symbol or set of symbols. If we are in some state s, and the next input symbol is a, we look for an edge out of state s labelled by a. If we find such an edge, we advance the forward pointer and enter the state of the transition diagram to which that edge leads. All the transition diagrams are deterministic, meaning that there is never more than one edge out of a given state with a given symbol among its labels.

Some important conventions about transition diagrams are:

1. Certain states are said to be *accepting*, or *final*. These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the lexemeBegin and forward pointers. We always indicate an accepting state by a double circle, and if there is an action to be taken - typically returning a token and an attribute value to the parser - we shall attach that action to the accepting state.
2. In addition, if it is necessary to retract the forward pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state. In our example, it is never necessary to retract forward by more than one position, but if it were, we could attach any number of *'s to the accepting state.

3. One state is designated the start state, or initial state; it is indicated by an edge, labelled "start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.



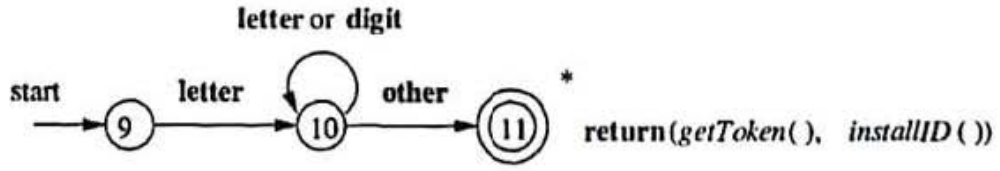
Transition diagram for **relop**

Above transition diagram recognizes the lexemes matching the token relop. We begin in state 0, the start state. If we see < as the first input symbol, then among the lexemes that match the pattern for relop we can only be looking at <, < >, or <=. We therefore go to state 1, and look at the next character. If it is =, then we recognize lexeme <=, enter state 2, and return the token relop with attribute LE, the symbolic constant representing this particular comparison operator. If in state 1 the next character is >, then instead we have lexeme <>, and enter state 3 to return an indication that the not-equals operator has been found. On any other character, the lexeme is <, and we enter state 4 to return that information. Note, however, that state 4 has a * to indicate that we must retract the input one position.

On the other hand, if in state 0 the first character we see is =, then this one character must be the lexeme. We immediately return that fact from state 5. The remaining possibility is that the first character is >. Then, we must enter state 6 and decide, on the basis of the next character, whether the lexeme is >= (if we next see the = sign), or just > (on any other character). Note that if, in state 0, we see any character besides <, =, or >, we cannot possibly be seeing a relop lexeme, so this transition diagram will not be used.

Recognition of Reserved Words and Identifiers:

Recognizing keywords and identifiers presents a problem. Usually, keywords like *if* or *then* are reserved, so they are not identifiers even though they look like identifiers.

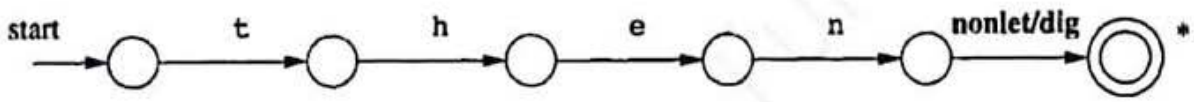


A transition diagram for **id**'s and keywords

Module 07: Lexical Analysis

There are two ways that we can handle reserved words that look like identifiers:

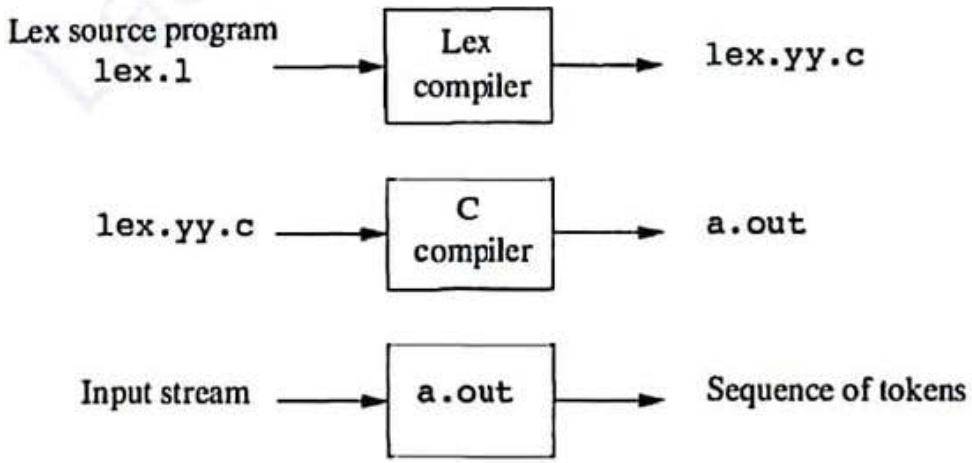
1. Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. When we find an identifier, a call to *installID* places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is id. The function get Token examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents - either id or one of the keyword tokens that was initially installed in the table.
2. Create separate transition diagrams for each keyword. Note that such a transition diagram consists of states representing the situation after each successive letter of the keyword is seen, followed by a test for a "non-letter-or-digit," i.e., any character that cannot be the continuation of an identifier. It is necessary to check that the identifier has ended, or else we would return token then in situations where the correct token was id, with a lexeme like *thenextvalue* that has then as a proper prefix. If we adopt this approach, then we must prioritize the tokens so that the reserved-word tokens are recognized in preference to id, when the lexeme matches both patterns.



Hypothetical transition diagram for the keyword `then`

7.5 Designing a lexical analyser generator (Lex):

A tool called Lex, or in a more recent implementation Flex, allows one to specify a lexical analyser by specifying regular expressions to describe patterns for tokens. The input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler. The Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called *lex.yy.c* that simulates this transition diagram.



Creating a lexical analyzer with Lex

Module 07: Lexical Analysis

Use of Lex:

Above figure suggests how Lex is used. An input file, which we call *lex.l*, is written in the Lex language and describes the lexical analyser to be generated. The Lex compiler transforms *lex.l* to a C program, in a file that is always named *lex.yy.c*. The latter file is compiled by the C compiler into a file called *a.out*, as always. The C-compiler output is a working lexical analyser that can take a stream of input characters and produce a stream of tokens.

The normal use of the compiled C program, referred to as *a.out* in above figure, is as a subroutine of the parser. It is a C function that returns an integer, which is a code for one of the possible token names. The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable *yylval*, which is shared between the lexical analyser and parser, thereby making it simple to return both the name and an attribute value of a token.

Structure of Lex Programs:

A Lex program has the following form:

{declarations}
%%
{translation rules}
%%
{auxiliary functions}

The **declarations** section includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions.

The **translation rules** each have the form:

Pattern {Action}

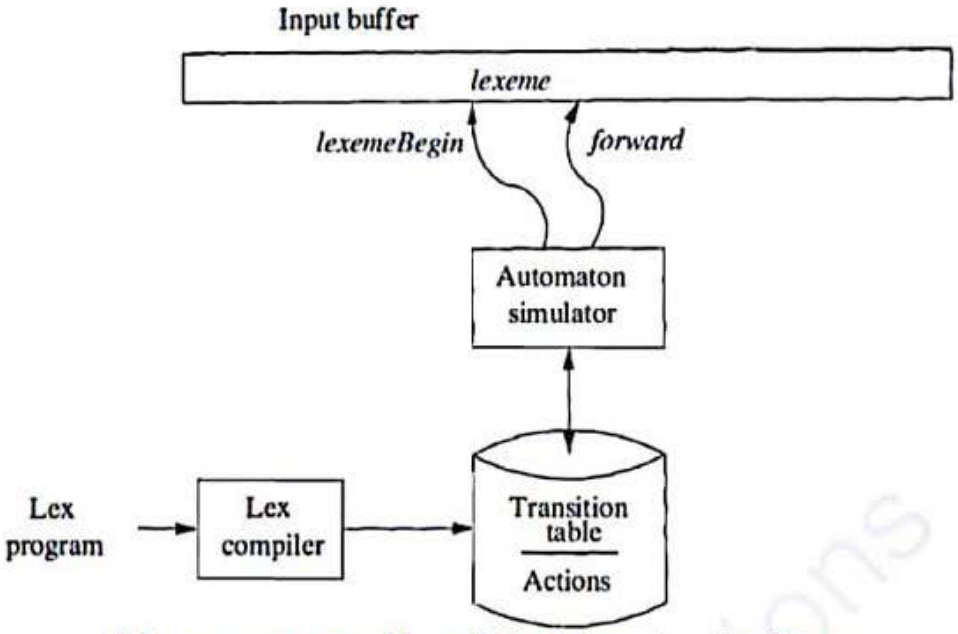
Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C, although many variants of Lex using other languages have been created.

The third section holds whatever **auxiliary functions** are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyser.

The lexical analyser created by Lex behaves in concert with the parser as follows. When called by the parser, the lexical analyser begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns. It then executes the associated actions. Typically, Actions will return to the parser, but if it does not (e.g., because Prefix describes whitespace or comments), then the lexical analyser proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser. The lexical analyser returns a single value, the token name, to the parser, but uses the shared, integer variable *yylval* to pass additional information about the lexeme found, if needed.

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}       {/* no action and no return */}
if         {return(IF);}
then       {return(THEN);}
else       {return(ELSE);}
{id}       {yylval = (int) installID(); return(ID);}
{number}   {yylval = (int) installNum(); return(NUMBER);}
"<"        {yylval = LT; return(RELOP);}
"<="       {yylval = LE; return(RELOP);}
"="        {yylval = EQ; return(RELOP);}
"<>"       {yylval = NE; return(RELOP);}
">"        {yylval = GT; return(RELOP);}
">="       {yylval = GE; return(RELOP);}

%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Example of Lex Program



A Lex program used by a finite-automaton simulator

The Structure of the Generated Analyser:

Above figure overviews the architecture of a lexical analyser generated by Lex. The program that serves as the lexical analyser includes a fixed program that simulates an automaton; at this point we leave open whether that automaton is deterministic or non-deterministic. The rest of the lexical analyser consists of components that are created from the Lex program by Lex itself.

These components are:

a. A transition table for the automaton.

b. Those functions that are passed directly through Lex to the output.

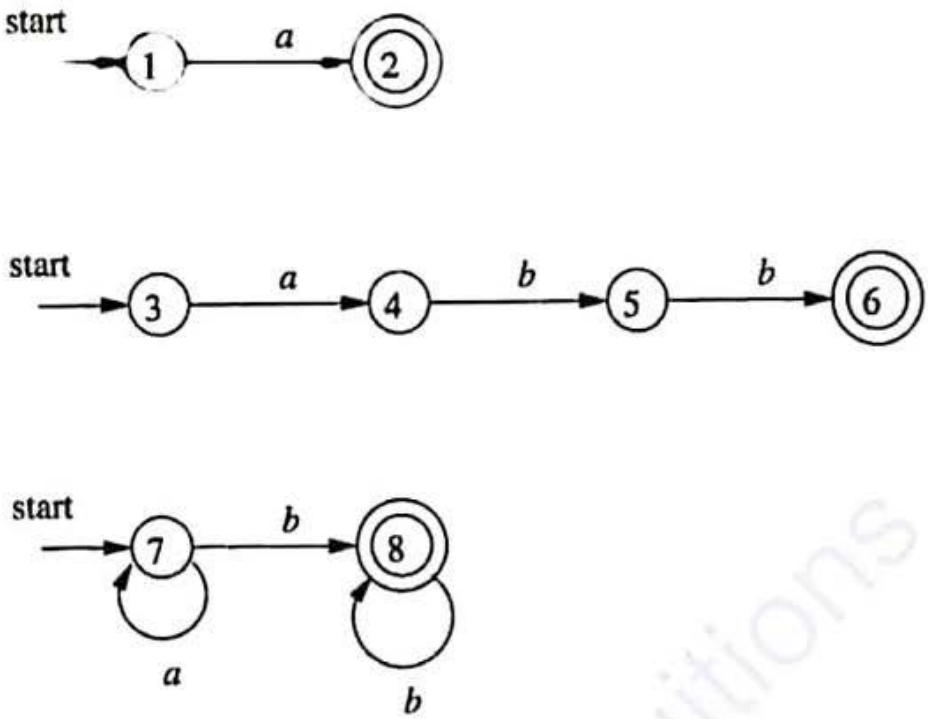
c. The actions from the input program, which appear as fragments of code td be invoked at the appropriate time by the automaton simulator.

To construct the automaton, we begin by taking each regular-expression pattern in the Lex program and converting it to an NFA. We need a single automaton that will recognize lexemes matching any of the patterns in the program, so we combine all the NFA's into one by introducing a new start state with E-transitions to each of the start states of the NFA's.

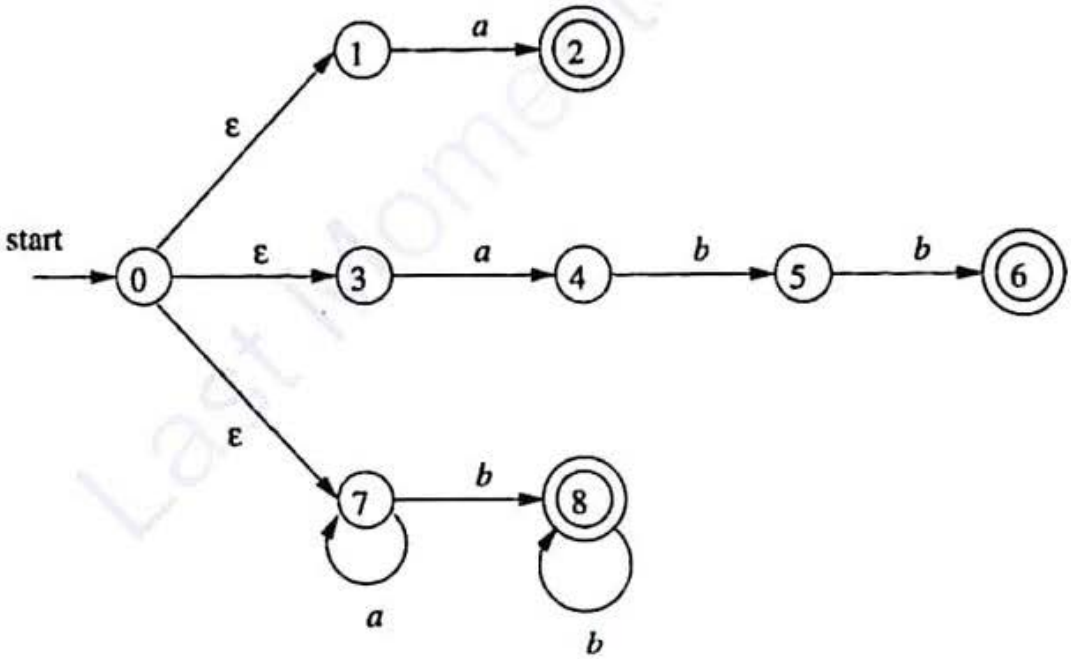
7.6 Pattern matching based on NFA's:

If the lexical analyser simulates an NFA, then it must read input beginning at the point on its input which we have referred to as *lexemeBegin*. As it moves the pointer called forward ahead in the input, it calculates the set of states it is in at each point.

Eventually, the NFA simulation reaches a point on the input where there are no next states. At that point, there is no hope that any longer prefix of the input would ever get the NFA to an accepting state; rather, the set of states will always be empty. Thus, we are ready to decide on the longest prefix that is a lexeme matching some pattern.

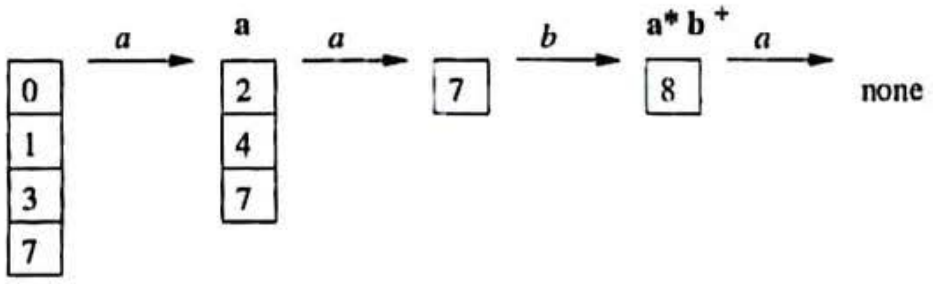


NFA's for **a**, **abb**, and **a*b**$^+$



Combined NFA

Last Moment Tuitions



Sequence of sets of states entered when processing input *aaba*

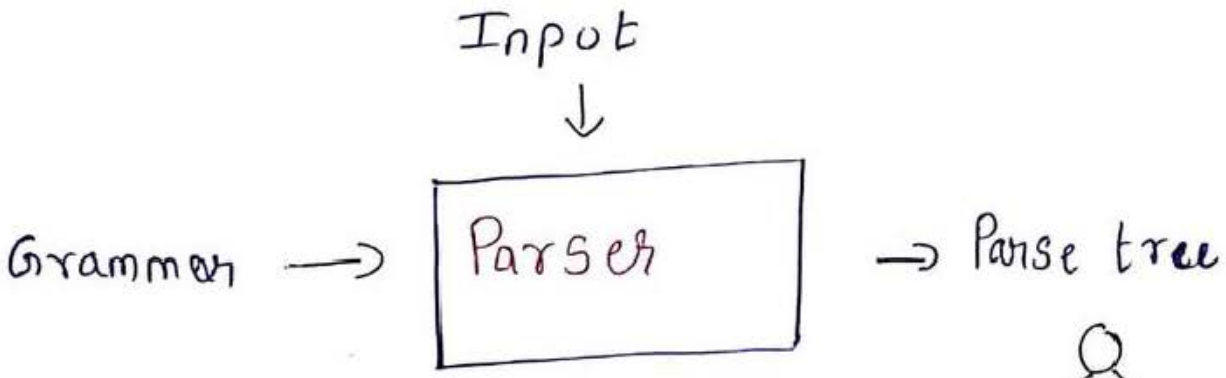
The sequence of sets of states are looked in backwards, until a set that includes one or more accepting states is found. If there are several accepting states in that set, pick the one associated with the earliest pattern P_i in the list from the Lex program. Move the forward pointer back to the end of the lexeme, and perform the action A_i associated with pattern P_i.

Last Moment Tuitions

Syntax Analysis
(Parser)

- Parser : this is second stage of compiler

Parser ka main kaam hota hai Parse tree generate karna.



(Parser ko ek input string + ek Grammar diya hong a isko check karna hai ki uss grammer se input Parse hoke Parse tree banta hai ya nahi bus)

The Role of Parser: Syntax analysis is the second Phase of the compile

- It gets the input from the tokens and generates a syntax tree or parse tree
- The parser or syntax analyser obtains a string of token from the lexical analyser and verifies that the string can be generated by the grammer for the source language
- It also report if there is any syntax error in the program
- It also recover from commonly occuring errors so that it can continue processing its input (agar koi chote mote error ho toh khud hi handle kar leta hai)

functions of the parser:

1) It verifies the structure generated by the token based on the grammer
2) It construct the parse tree
3) It reports error
4) It perform error recovery

Types of Parser

- Parser
  - Top down
    - Recursive descent Parser
    - Non Recursive descent Parser (LL(1)) also called as predictive Parser
  - Bottom UP
    - Operator Precedence Parser
    - LR Parser
      - LR(0)
      - SLR(1) Simple LR
      - LALR(1) Look AHead LR
      - CLR (1) conical LR

Baki jagah apko aur bhi dikhenge Lekin Kaam anne wale sirf itne hi hai.

Top down parsing: Top down method Build Parse tree from the top (root) to the bottom (leaves). A Parser can start with the start symbol and try to transform it to the input string.

Example: LL(1)

Example: Given Grammer is

$$\boxed{\begin{array}{l} S \rightarrow aABe \\ A \rightarrow Abc \mid b \\ B \rightarrow d \end{array}}$$

Grammer

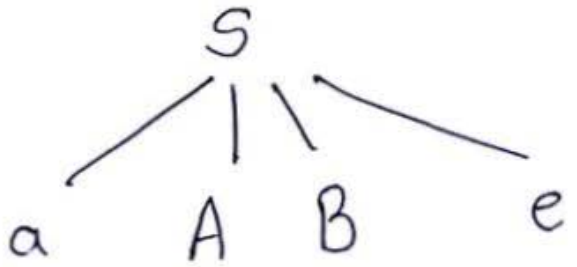
input : a bbcde

input string

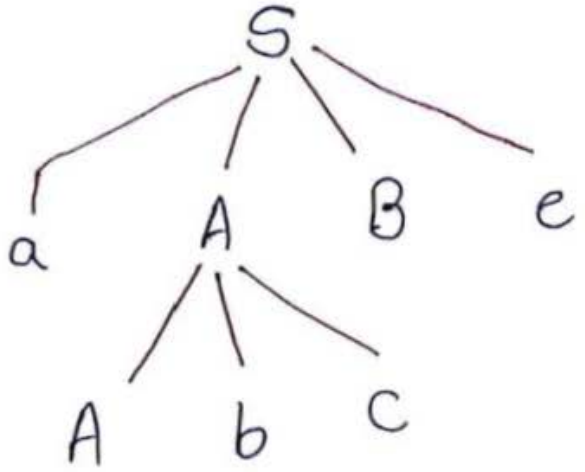
Hame given grammer ki help se input ka Parse tree banana hai with top down parser

Also Always start with start symbol

Step 1)



step 2)



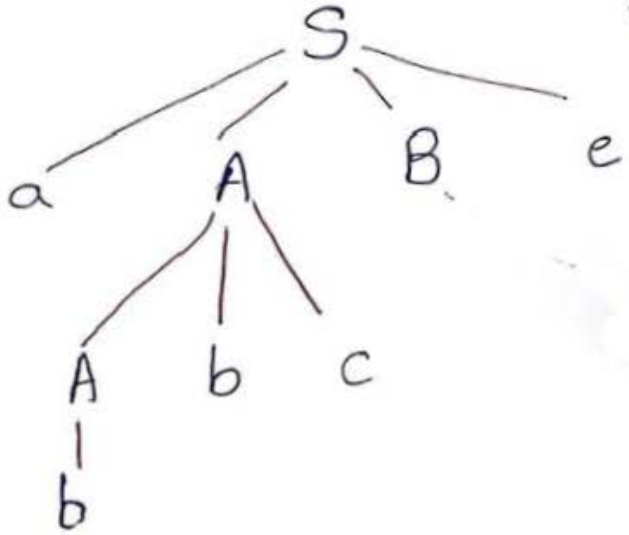
Note

$A \to Abc$

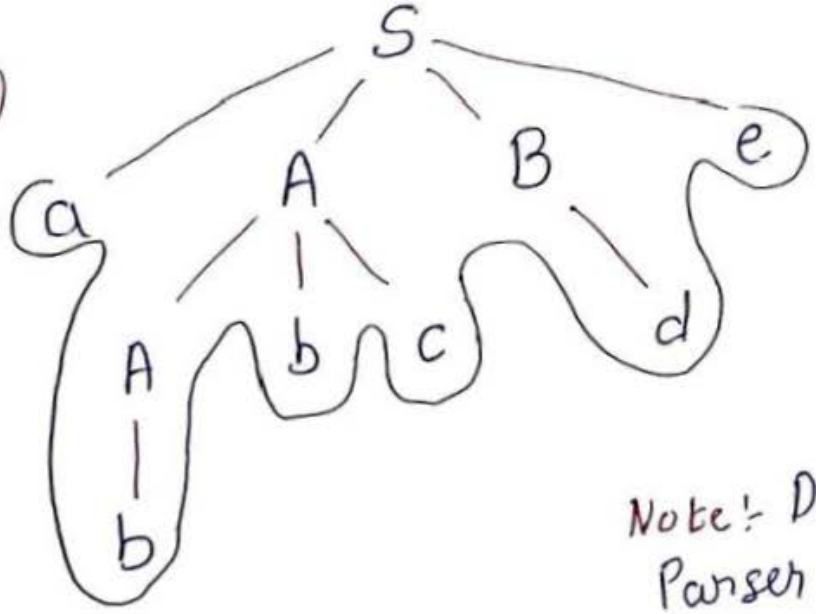
$A \to b$

jaise u can see A ke 2 production hai Toh konsa choose karna Proper Parse tree banne ko voh main kaam hai Top down Parser ka.

step 3)



step 4)



Parse tree generated

Note:- Dekh bhai top-down Parser Left recursive and Left factoring grammer ko accept nahi karta ab voh kya hota hai video me bataya hai.

Bottom up parsing:- Bottom-up methods starts from the leaves and work their way up to the root

- yeh input string se start karta hai aur start symbol pe pahuchta hai.

Example: LR Parser

Example: Lets take the same Grammer

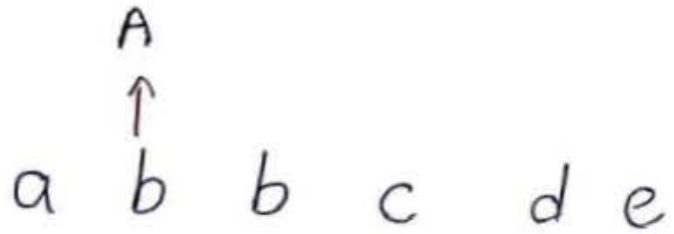
$S \rightarrow aABe$ input: abbcde

$A \rightarrow Abc \mid b$

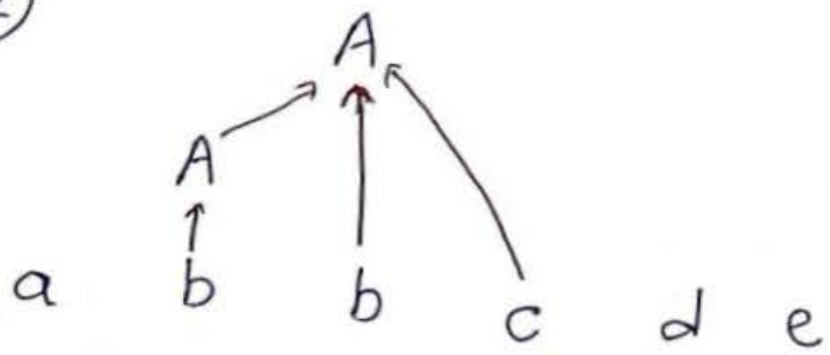
$B \rightarrow d$

Hame given grammer ki help se input ka Parse tree banana hai with Bottom up parser

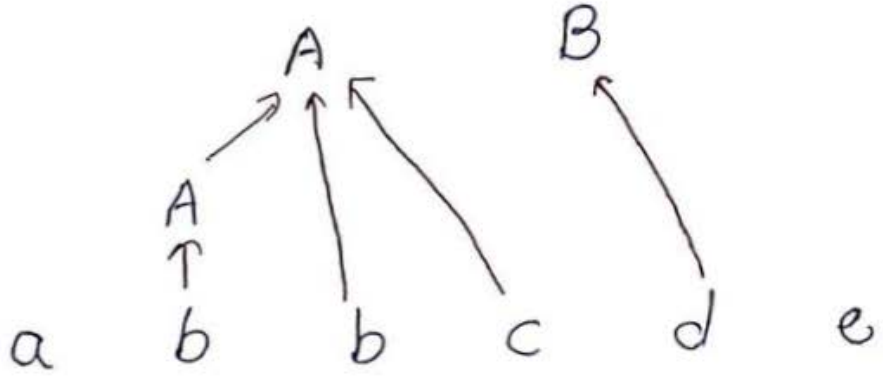
Step 1)



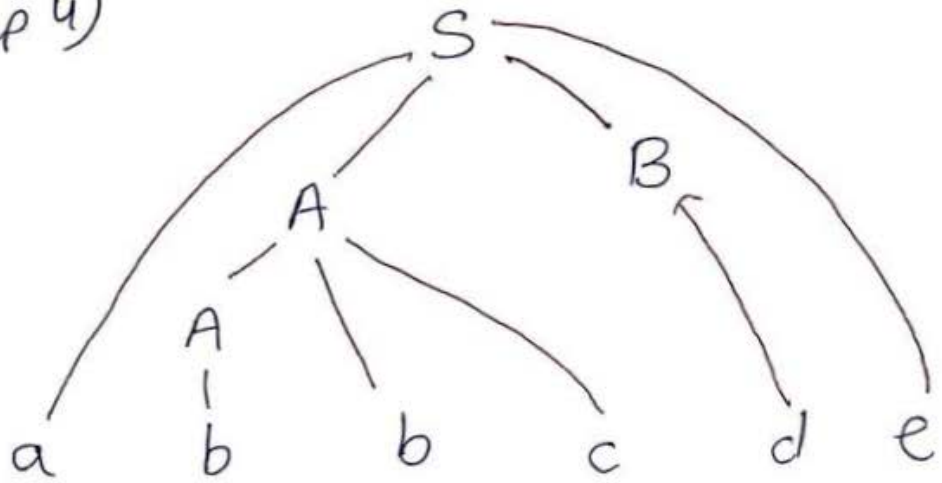
Step 2)



Step 3)



Step 4)



Hence we got the start symbol with given input string.

- Top down me main kaam tha what to use
  $A \to Abc$
  $A \to b$
- Bottom up ka main kaam hota hai what to reduce

Example: $b \leftarrow A$
$d \leftarrow B$

line me dono de
Par Pehla kisiko reduce
karna hai voh Bottom up
Parser ko dhyan rakhna hai.

Ambigous Grammer

- Parser dont use ambigous Grammer (yeh apne aksar suna honga toh chalo dekhte hai ambigous grammer hota kya hai aur Parser usko use kyo nahi karte)

Given Grammer is

$E \rightarrow E+E$
$\quad / E*E$
$\quad / id$

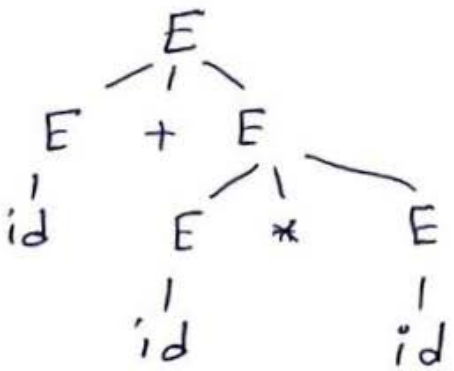
Ambiguity :- A grammer that Produce more than one parse tree for same sentence is called ambigious grammer

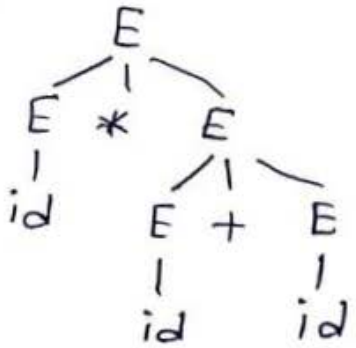
input :- id+id*id

(1)
$E \rightarrow E+E$
$E \rightarrow id+E$
$E \rightarrow id+E*E$
$E \rightarrow id+id*E$
$E \rightarrow id+id*id$

(2)
$E \rightarrow E*E$
$E \rightarrow E+E*E$
$E \rightarrow id+E*E$
$E \rightarrow id+id*E$
$E \rightarrow id+id*id$

The two corresponding Parse tree



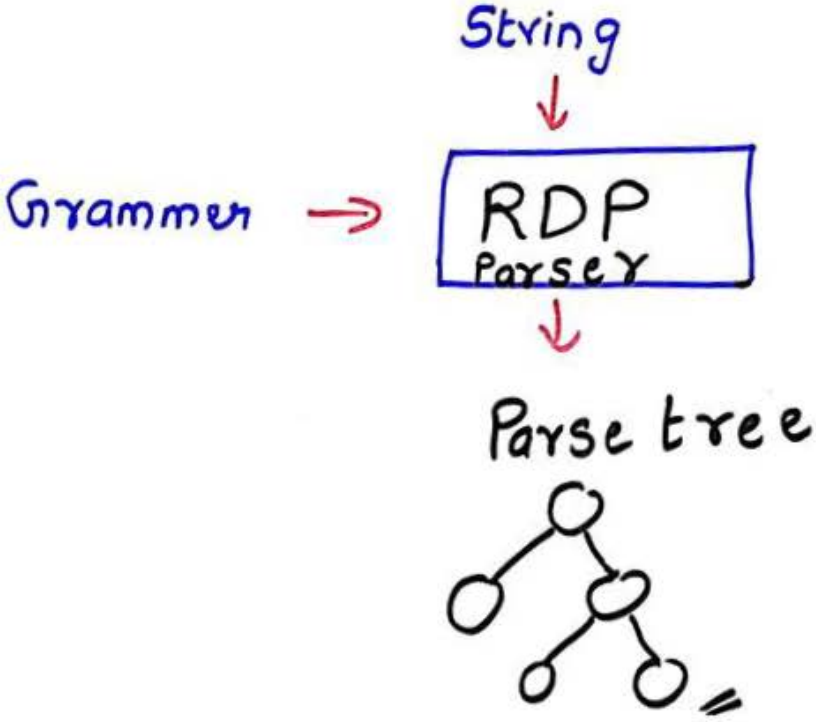


(jaise tumne dekha ek Grammer and input ki 2 Parse Tree ban rahi hai aur Parser machine usko yeh nahi samajhta ki konsa Parse tree banana hai isliye voh ambigious grammer hath me hi nahi leta wata deta hai)

- For a single input there are two parse tree so parser is a machine he is not able to decide to choose which one hence Parser dont take ambigious grammer.

except: Operator precendence Parser
this is a special case which accept ambigious grammer

Recursive Descent Parser



Defination: Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input

The parsing method may involve backtracking: that is making repeated scan of its input

Example for backtracking:

Consider the grammer G:

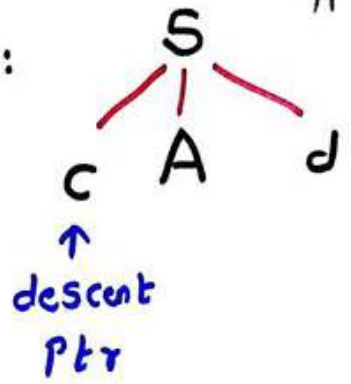
$S \rightarrow cAd$

$A \rightarrow ab \mid a$

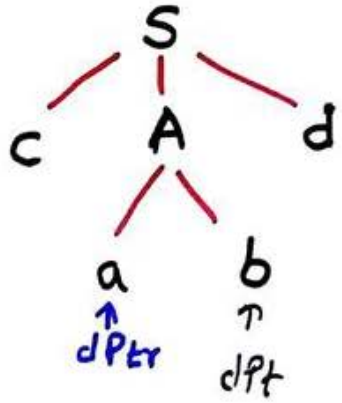
and input string $w = c\,a\,d$

i/p i/p i/p

Step 1:



Step 2:

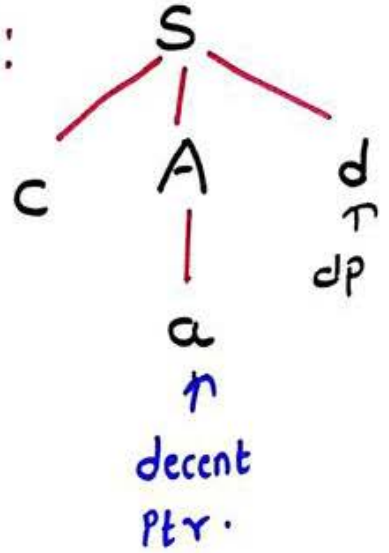


Step 3: Backtracking

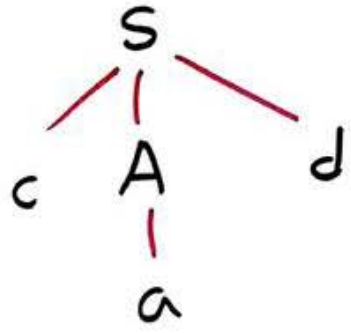
Input string $w = c\,a\,d$

i/p i/p

Step 4:

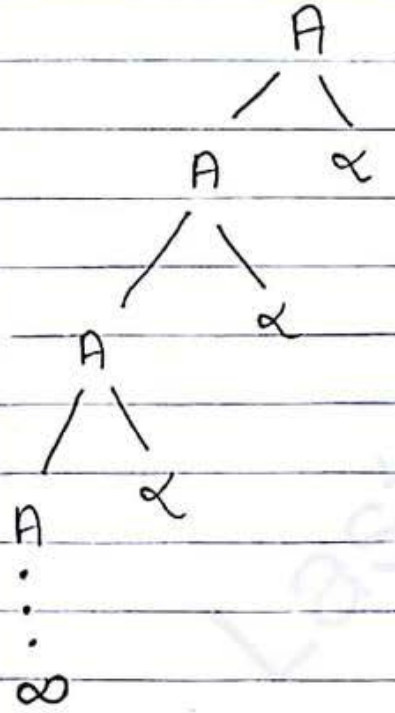


Yes the given string can be Parse and the parse tree is shown below



Elimination of left recursion

if $A \rightarrow A\alpha \mid \beta$ then it contains left recursion



matlab A na khud ko hi baar baar call karta hai aur infinite loop me jata hai

↓

```
A()
{
A()
⋮
}
```

isiliye hum left recursion eliminate karte hai

$\therefore \quad A \rightarrow A\alpha \mid \beta$

After eliminating left recursion we get

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Example: 1) $E \rightarrow E+T \mid T$
2) $T \rightarrow T*F \mid F$

1) $\underset{A}{E} \rightarrow \underset{A}{E}\underset{\alpha}{+T} \mid \underset{\beta}{T}$

jaisa app dekh sakte ho it contain left recursion

After eliminating left recursion we get

$$E \Rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$

2) $T \rightarrow T*F \mid F$ is homework for you.

Lab Monk Tuitions

Multiple left recursion

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \beta_1 \mid \beta_2 \mid \beta_3$

After eliminating left recursion we gets

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \varepsilon$

Example:

$A \rightarrow Aa \mid Ab \mid Ac \mid e \mid f \mid g$

After eliminating left recursion we gets

$A \rightarrow eA' \mid fA' \mid gA'$

$A' \rightarrow aA' \mid bA' \mid cA' \mid \varepsilon$

Elimination of left factoring

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$

$\downarrow$

After eliminating lefts factoring

$\downarrow$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$

$\boxed{\cancel{A \rightarrow Sa \mid Sb \mid Sc}}$

Example:

$A \rightarrow Sa \mid Sb \mid Sc$

After eliminating left factoring

$A \rightarrow SA'$

$A' \rightarrow a \mid b \mid c$

First and Follow

Notes

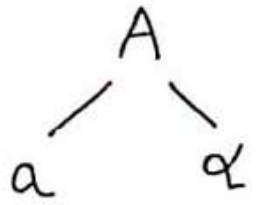
Note: Example Pe direct jane se Pehle hum first and follow ka basic concept samajhte hai.

Imp note: Kisi bhi start symbol ka follow $ (dollar) Pehle se hota hai by default.

First me 3 type ke cases hote hai

case 1

$A \rightarrow a\alpha$



$\therefore$ first(A) = {a}

ab α kuch bhi ho hame usse koi matlab nahi bcoz hume first direct mil gaya jo hai 'a'.

Scanned by CamScanner

Example: $A \to aBcde$

$first(A) = \{a\}$

I hope case 1 app samajh gaye

Case 2

$A \to \epsilon$

$first(A) = \{\epsilon\}$

(Agar koi non-terminal (A) direct ϵ (epsilon) produce kar raha hai toh first of that will be epsilon)

Case 3

$A \to B$

$B \to C$

$C \to id$

Ab jaise app dekh sakte ho A ka first B hai jo ek non-terminal so mai usko first me nahi likh sakta toh hum B ka first ~~che~~ check karte hai jo C hai non-terminal toh ap C ka first which is id yeh terminal hai isko hum first me likh sakte hai

Hence $first(A) = \{id\}$

Scanned by CamScanner

Let solve a problem:

Find First of given grammer

$A \rightarrow a$

$B \rightarrow (c) \mid id$

$C \rightarrow * T \mid \epsilon$

first $(A) = \{a\}$

first $(B) = \{ (, id\}$

first $(C) = \{ *, \epsilon \}$

This was the basic example and basic concept of first :)

Scanned by CamScanner

Follow

Note: follow me kabhi ϵ (epsilon) nahi aata

∴ follow dhunde ne ke liye hamesha grammer ke right side me dekho

Follow me bhi 3 cases hote hai.

Case 1

$A \rightarrow \alpha B \beta$

$\beta \rightarrow a$

follow (B) = {a}

mujhe B ka follow nikalna hai sabse pehle grammer ke right side me B ko dhundo ab uske piche kon hai β so β ka jo bhi first honga voh mera follow of B honga.

Example: find follow T.

$A \rightarrow TE'$

$E' \rightarrow +F$

follow (T) = {+}

Scanned by CamScanner

Case 2:

$A \rightarrow \alpha B \beta$

$\beta \rightarrow \epsilon$

Note:- jab apke none terminal ke piche kuch bhi naho ya epsilon ho toh voh apne parents ke follow le lenga.

i.e $A \rightarrow \alpha B$

then

[Note: follow me kabhi epsilon mat likhna.]

follow (B) $\rightarrow$ { All follow of A }

Let take some example

Example 1: find follow of E'

$E \rightarrow TE'$

follow (E) = { \$,) } ← yeh apne pass hai pehle se

follow (E') = { \$,) } ab E' ke piche kuch hai hi nahi so voh apne parents ke saare follow lenge hence E ke follow hi E' follow ban gaye.

Scanned by CamScanner

Example 2: find follow of E'

$E \rightarrow TE'B$

$B \rightarrow \epsilon$

ab jaise app dekh sakte ho E' ke piche B hai par B toh ϵ (epsilon) produce kar raha hai aur follow pe hum kabhi epsilon nahi likhte so jab hum B ko epsilon replace kar dete hai we get

$E \rightarrow TE'$

→ ab E' ke piche kuch nahi hai so voh apne parents ke follow lenga ☺

Scanned by CamScanner

Case 3

Note: case 3 hybrid hai case 1 and case 2 ka bus jisme dono concept use hua hai.

$A \to \alpha B \beta$

$\beta \to a \mid \epsilon$

then

follow(B) = { a, All follow of A }

Example: Find follow of B

$A \to aBD$

$D \to d \mid \epsilon$

follow(A) = { \$, b, c } ← given

follow(B) = { d, \$, b, c }

Scanned by CamScanner

Exam level example of first and follow.

Find first and follow of given grammer

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Note: $ (dollar) Pure grammer me apko kahi nahi dikhenga but E mera start symbol hai so uske follow me by default ayenga.

	first	follow
E	(, id	$,)
E'	+, ε	$,)
T	(, id	+, $,)
T'	*, ε	+, $,)
F	(, id	*, +, $,)

WWW.Lastmomenttuitions.com

Scanned by CamScanner

Problems for you to Practise
find first and follow for the given
Grammer

1) $S \rightarrow AS \mid [S]S \mid \varepsilon$

$A \rightarrow (B) \mid \varepsilon$

$B \rightarrow AB \mid [B]B \mid \varepsilon$

2) $S \rightarrow F$

$S \rightarrow (S-F)$

$F \rightarrow a$

Answers

1)

	first	follow
S	(, [, ε	$,]
A	(, ε	(, [, $,],)
B	(, [, ε	),]

2) yaar second wala chindi tha ab utna toh khud karo :)

Best of Luck

Machao

① find first and follow (Twisted Problem)

$S \rightarrow aBDh$
$B \rightarrow cC$
$C \rightarrow bC \mid \varepsilon$
$D \rightarrow EF$
$E \rightarrow g \mid \varepsilon$
$F \rightarrow f \mid \varepsilon$

NT	first	follow
S	a	$
B	c	g, f, h
C	b, ε	g, f, h
D	g, f, ε	h
E	g, ε	f, h
F	f, ε	h

② find first and follow.

$E \to TQ$

$T \to FR$

$Q \to +TQ \mid -TQ \mid E$

$R \to *FR \mid /FR \mid E$

$F \to (E) \mid id$

first (E) = { (, id }

first (T) = { (, id }

first (Q) = {+, −, (, id }

first (R) = { *, /, (, id }

first (F) = { (, id }

follow(E) = { $, follow(Q), follow(R) (nothing), {+, −, (, id }

follow(T) = {+, −, (, id }

follow(Q) = {follow (E)}

follow(R) = {+, −, (, id }

follow(F) = {*, /, (, id }

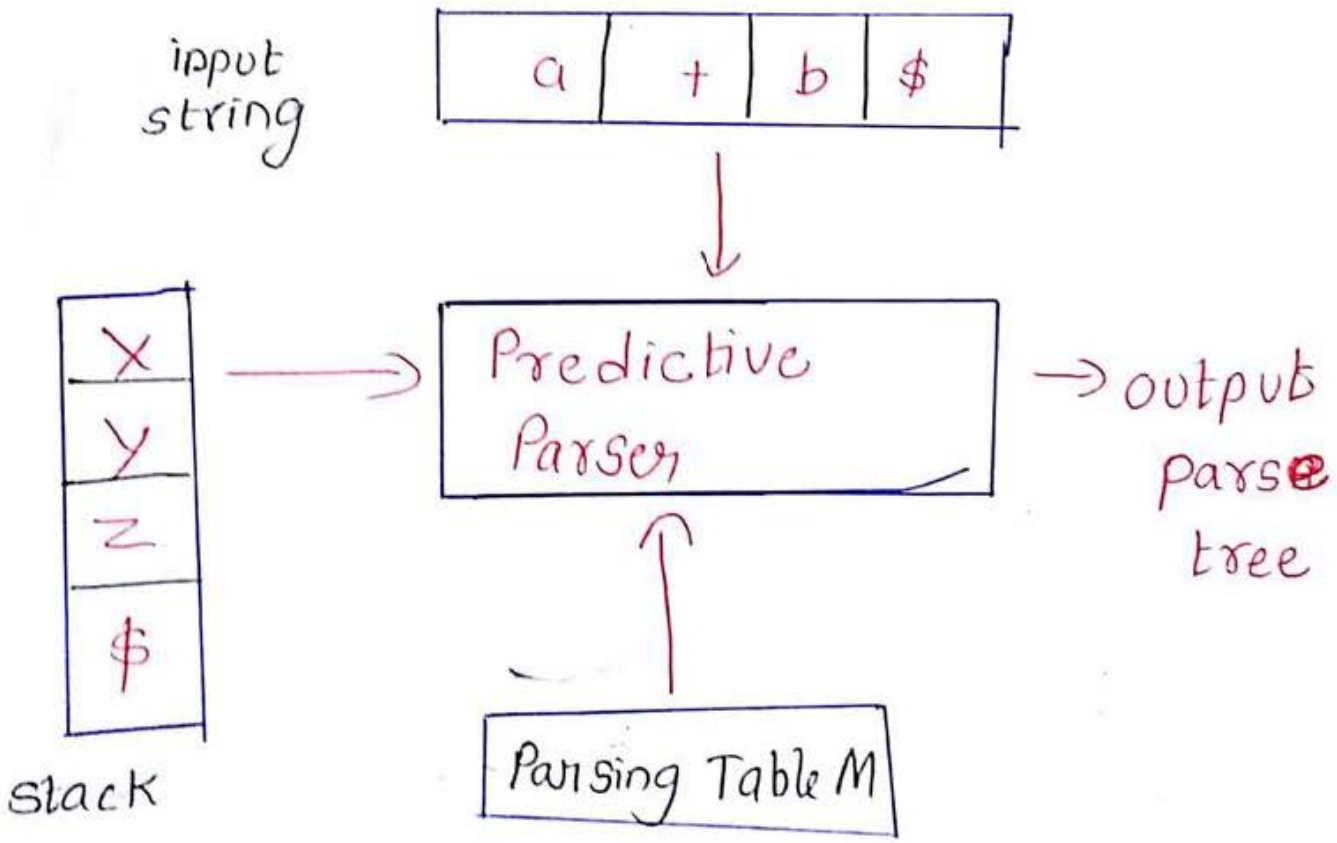
Hence Rewriting again all follow

follow (E) = { $, +, -, (, id }
follow (T) = { +, -, (, id }
follow (Q) = { $, +, -, (, id }
follow (R) = { +, -, (, id }
follow (F) = { *, /, (, id }

Toh kabhi kabhi recursive first and follow bhi hota hai so uska last tak solve karna hai

Predictive parser

Predictive Parser is a special case of recursive descent parser where no back tracking is required.



Predictive Parser 3 input leta hai • input string • stack • Parsing table and then give output a parse tree.

Note: Predictive parser me left recursion and Left factoring ka grammer nahi chalta so agar Left recursion and Left factoring hai toh usko remove karo pehle

Steps to solve predictive parser or LL(1) sums

Step 0) remove left recursion and Left factoring if it is there

1) find first and follow
2) Parsing table
3) stack implementation
4) Generate parse tree

Example: Construct LL(1) parsing table for the following grammer:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

check whether the string "id+id*id$" is valid or not

Ans) Step 0) The given grammer contains left recursion so after eliminating left recursion we get

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Step 1) find first and follow (isse 2 videos hai IMT Pe dekh lena)

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

Step 2) Generate Parsing table:

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

Agar apne video dekhi hai toh app aaram se first and follow ki help se parsing table bana donge

Most Important Point

- Question aksar aata hai check wheather it is LL(1) or not toh kaise pata kare given grammer LL(1) hai ke nahi
- Its very easy apko sirf parsing table banana hai agar parsing table ke kisi bhi ek box ke andar 2 entry ek saath hai matlab it is not LL(1)
- Aur agar sirf ek entry toh it is LL(1)

aage hum ek example not LL(1) ka solve karenga toh aur ache se samjhega.

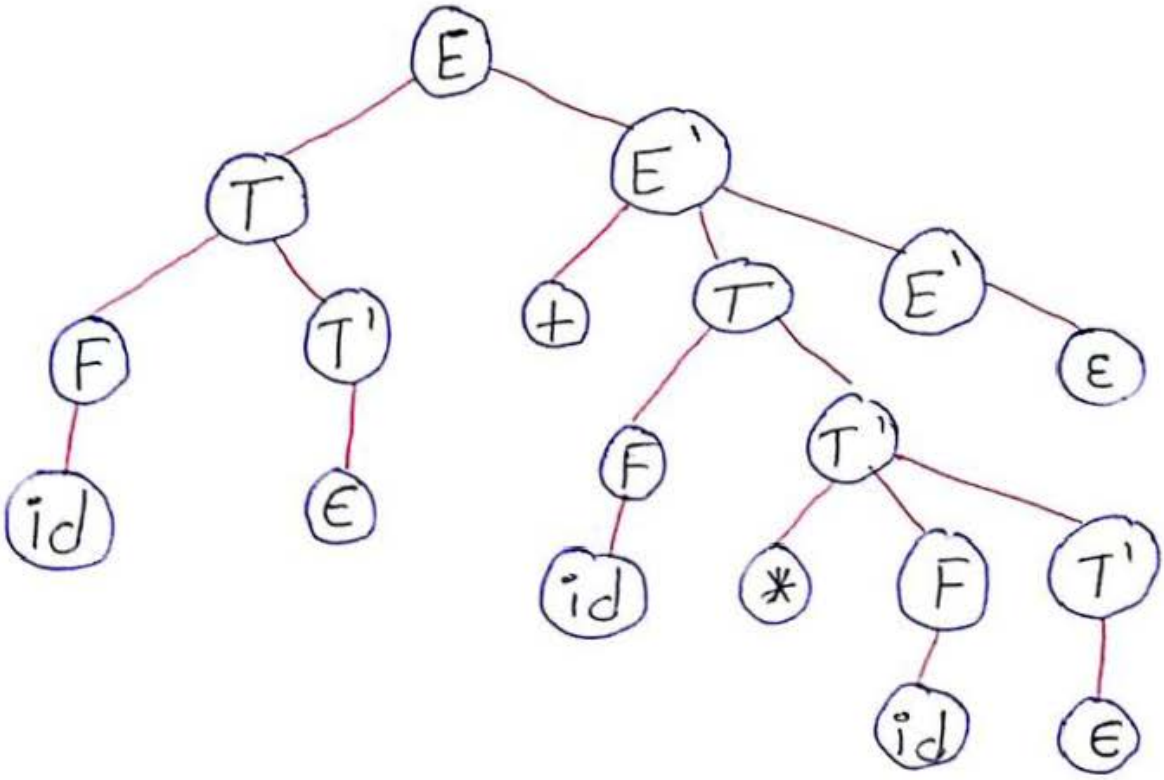
3) Stack implementation

Stack	input	output
E $	id+id*id $	
TE' $	id+id*id $	E→TE'
FT'E' $	id+id*id $	T→FT'
id T'E' $	id+id*id $	F→id
T'E' $	+id*id $	
E' $	+id*id $	T'→ϵ
+TE' $	+id*id $	E'→+TE'
TE' $	id*id $	
FT'E' $	id*id $	T→FT'
idT'E' $	id*id $	F→id
T'E' $	*id $	
*FT'E' $	*id $	T'→*FT'
FT'E' $	id $	
id T'E' $	id $	F→id
T'E' $	$	

E' $	$	T' → ε
$	$	E' → ε

Step 4 : Draw Parse tree

(stack implementation ka jo output column hai usko dekh ke aaramse parse tree banega)



Parse tree for input: id+id*id

www.Lastmomentbitions.com

2) Check wheather given grammar is LL(1) or NoT

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

$E \rightarrow b$

After eliminating Left factoring

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

Eliminating LF

$A \rightarrow \alpha A'$

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

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

NT	first	follow
S	i, a	e, $
S'	e, ϵ	e, $
E	b	t

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 1 production

The Grammer is NOT LL(1)

Problems for you to practise

Test wheather the grammer is LL(1) or not and construct a predictive parsing lables for it

1) $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

2) $S \rightarrow A$

$A \rightarrow aB \mid Ad$

$B \rightarrow bBC \mid f$

$C \rightarrow g$

Best of luck

Question:
Test whether the grammar is LL(1) or not and construct a predictive parsing table for it.

$S \rightarrow AaAb \mid BbBa$
$A \rightarrow \varepsilon$
$B \rightarrow \varepsilon$

Solution:
The given grammar neither contains left recursion nor left factoring.

FIRST(S) = {a, b}
FIRST(A) = {ε}
FIRST(B) = {ε}

FOLLOW(S) = {$}
FOLLOW(A) = {a, b}
FOLLOW(B) = {a, b}

Parsing Table:

Non-Terminal	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	
B	$B \rightarrow \varepsilon$	$B \rightarrow \varepsilon$	

As evident from the table above, none of the cells contain more than one production, so the grammar is LL(1).

Question:
Test whether the grammar is LL(1) or not and construct a predictive parsing table for it.

$S \rightarrow A$
$A \rightarrow aB \mid Ad$
$B \rightarrow bBC \mid f$
$C \rightarrow g$

Solution:
Given contains Left Recursion in the production: $A \rightarrow aB \mid Ad$
Therefore, after eliminating left recursion, the grammar becomes,

$S \rightarrow A$
$A \rightarrow aBA'$
$A' \rightarrow dA' \mid \varepsilon$
$B \rightarrow bBC$
$B \rightarrow f$
$C \rightarrow g$

FIRST(S) = {a}
FIRST(A) = {a}
FIRST(A') = {d, ε}
FIRST(B) = {b, f}
FIRST(C) = {g}

FOLLOW(S) = {$}
FOLLOW(A) = {$}
FOLLOW(A') = {$}
FOLLOW(B) = {$, d, g}
FOLLOW(C) = {$, d, g}

Parsing Table:

Non-Terminal	a	b	d	f	g	$
S	S → A					
A	A → aBA'					
A'			A' → dA'			A' → ε
B		B → bBC		B → f		
C					C → g	

As evident from the table above, none of the cells contain more than one production, so the grammar is LL(1).

Question:
Test whether following grammar is LL (1) or not. Construct LL (1) parsing table.

S → AB | g Da
A → ab | c
B → dC
C → gC | g
D → f D | g

Solution:
Given grammar contains left factoring in the production: C → gC | g
After removing left factoring, grammar becomes:

S → AB
S → gDa
A → ab
A → c
B → dC
C → gC'
C' → C
C' → ε
D → f D
D → g

FIRST(S) = {a, c, g}
FIRST(A) = {a, c}
FIRST(B) = {d}
FIRST(C) = {g}
FIRST(C') = {g, ε}
FIRST(D) = {f, g}

FOLLOW(S) = {$}
FOLLOW(A) = {d}
FOLLOW(B) = {$}
FOLLOW(C) = {$}
FOLLOW(C') = {$}
FOLLOW(D) = {a}

Parsing Table:

Non-Terminal	a	b	c	d	f	g	$
S	S → AB		S → AB			S → gDa	
A	A → ab		A → c				
B				B → dC			
C						C → gC'	
C'						C' → C	C' → $
D					D → fD	D → g	

As evident from the table above, none of the cells contain more than one production, so the grammar is LL(1).

Question:
Consider the following grammar with terminals (, [,) and].

S → TS | [S]S |)S | ε
T → (X)
X → TX | [X]X | ε

1. Construct first and follow set for the non-terminals.
2. Construct its LL(1) parsing table.
3. Is this LL(1) grammar.

Solution:
The given grammar neither contains left recursion nor left factoring.

FIRST(S) = { (, [,), ε }
FIRST(T) = { (}
FIRST(X) = { (, [, ε }

FOLLOW(S) = { $,] }
FOLLOW(T) = { (, [,), $,] }
FOLLOW(X) = {),] }

Parsing Table:

Non-Terminal	(	)	[	]	$
S	S → TS	S →)S	S → [S]S	S → ε	S → ε
T	T → (X)				
X	X → TX	X → ε	X → [X]X	X → ε	

As evident from the table above, none of the cells contain more than one production, so the grammar is LL(1).

Question:

For the following grammar construct LL(1) parser table and parse the string (a – a). Show contents of stack and input buffer and action taken after each step,

$S \rightarrow F$
$S \rightarrow (S - F)$
$F \rightarrow a$

Solution:

The given grammar neither contains left recursion nor left factoring.

FIRST(S) = { (, a }
FIRST(F) = { a }

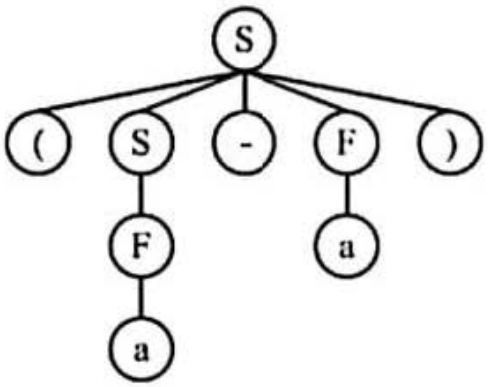
FOLLOW(S) = { $, - }
FOLLOW(F) = { $, -,) }

Parsing Table:

Non-Terminal	(	)	-	a	$
S	S → (S-F)			S → F	
T				F → a	

Stack implementation:

Stack	Input	Output
S $	(a - a) $	S → (S-F)
(S - F) $	(a - a) $	
S - F) $	a - a) $	S → F
F - F) $	a - a) $	F → a
a - F) $	a - a) $	
F) $	a) $	F → a
a) $	a) $	
$	$	



Parse Tree for input: (a-a)

Question:
Modify the given grammar and construct a predictive parser table explaining each step

$E \rightarrow TQ$
$T \rightarrow FR$
$Q \rightarrow +TQ \mid -TQ \mid E$
$R \rightarrow *FR \mid /FR \mid E$
$F \rightarrow (E) \mid id$

Solution:
Given grammar neither contains left recursion nor left factoring.

$E \rightarrow TQ$
$T \rightarrow FR$
$Q \rightarrow +TQ$
$Q \rightarrow -TQ$
$Q \rightarrow E$
$R \rightarrow *FR$
$R \rightarrow /FR$
$R \rightarrow E$
$F \rightarrow (E)$
$F \rightarrow id$

FIRST(E) = { (, id }
FIRST(T) = { (, id }
FIRST(Q) = { +, -, (, id }
FIRST(R) = { *, /, (, id }
FIRST(F) = { (, id }

FOLLOW(E) = { $,), +, -, (, id }
FOLLOW(T) = { +, -, (, id }
FOLLOW(Q) = { $,), +, -, (, id }
FOLLOW(R) = { +, -, (, id }
FOLLOW(F) = { *, /, (, id }

Parsing Table:

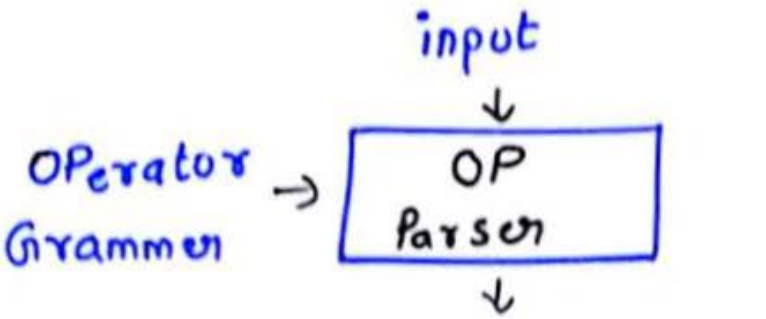
Non-Terminal	(	)	+	-	*	/	id	$
E	E→TQ						E→TQ	
T	T→FR						T→FR	
Q	Q→E		Q→+TQ	Q→-TQ			Q→E	
R	R→E				R→*FR	R→/FR	R→E	
F	F→(E)						F→id	

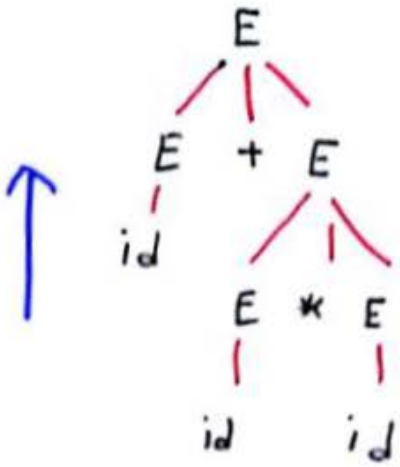
As evident from the table above, none of the cells contain more than one production, so the grammar is LL(1).

Moment Tuitions

Operator Precedence Grammer

Operator-Precedence Parser





Bottom up Parser

Operator Grammer: 1) No Epsilon in right side
2) No adjacent variable

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

- $E \rightarrow EAE \mid id$
  $A \rightarrow + \mid *$

not an operator Grammer

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

Example

AB ✗
$A+B$ ✓
$A*B$ ✓
A/B ✓

$S \rightarrow AB$

what do you mean by operator precedence grammer

with the help of following grammer parse the input string: "id + id * id" $

$T \rightarrow T+T \,/\, T*T \,/\, id$

Steps to solve

1) Check OPG or NOT (Operator Precedence grammer)
2) Operator Precedence Relation Table
3) Parse the given string
4) Generate Parse tree.

$T \rightarrow T+T \;/\; T*T \;/\; id$

Operator Precedence Relation Table

	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	—	>
$	<	<	<	A

Basics

id, a, b, c → High

$ → low

+ > +

* > *

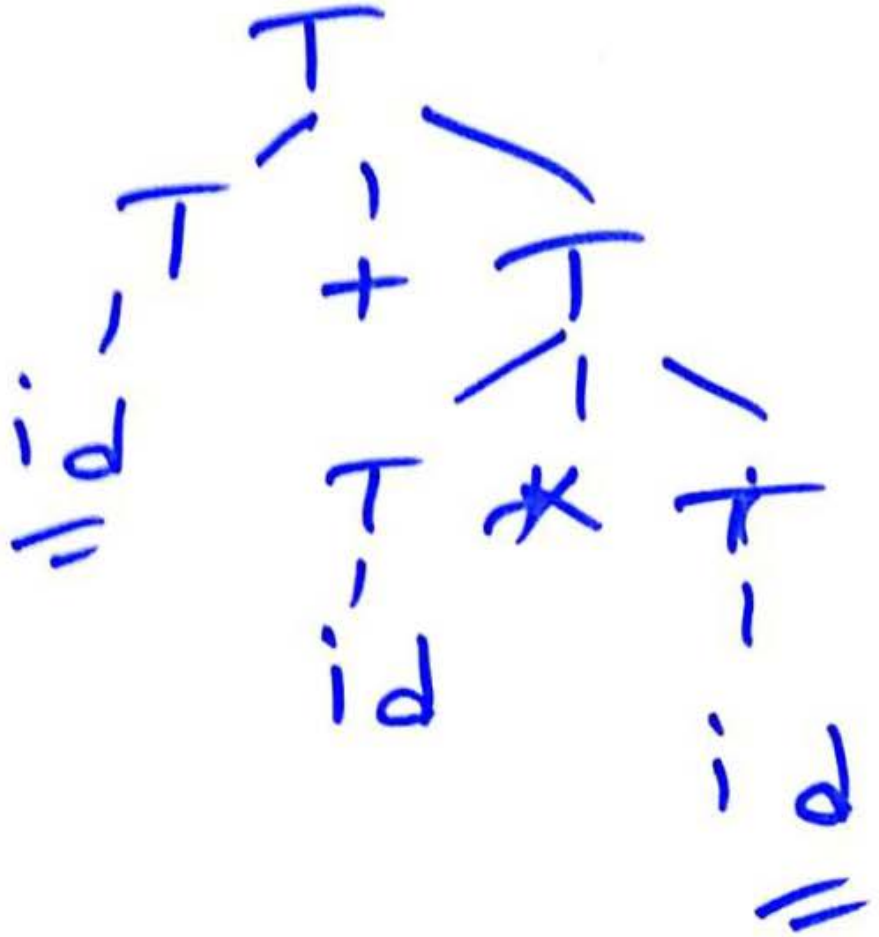
id ≠ id

$ A $

S R I C

Stack	Relation	Input	Comment
$	<	id+id*id$	shift id
$id	>	+id*id$	Reduce T→id
$T	<	+id*id$	shift +
$T+	<	id*id$	shift id
$T+id	>	*id$	Reduce T→id
$T+T	<	*id$	Shift *
$T+T*	<	id$	shift id
$T+T*id	>	$	Reduce T→id
$T+T*T	>	$	T→T*T
$T+T	>	$	T→T+T
$T	A	$	
	=		

Parse tree



With the help of following grammer Parse the input string: " a+b*c*d "

$E \rightarrow E+T / T$

$T \rightarrow T*V / V$

$V \rightarrow a / b / c / d$

Operator Precedence Relation Table

	+	*	a	b	c	d	$
+	>	<	<	<	<	<	>
*	>	>	<	<	<	<	>
a	>	>					>
b	>	>					>
c	>	>					>
d	>	>					>
$	<	<	<	<	<	<	A

Grammar

$E \rightarrow E + T \,/\, T$

$T \rightarrow T * V \,/\, V$

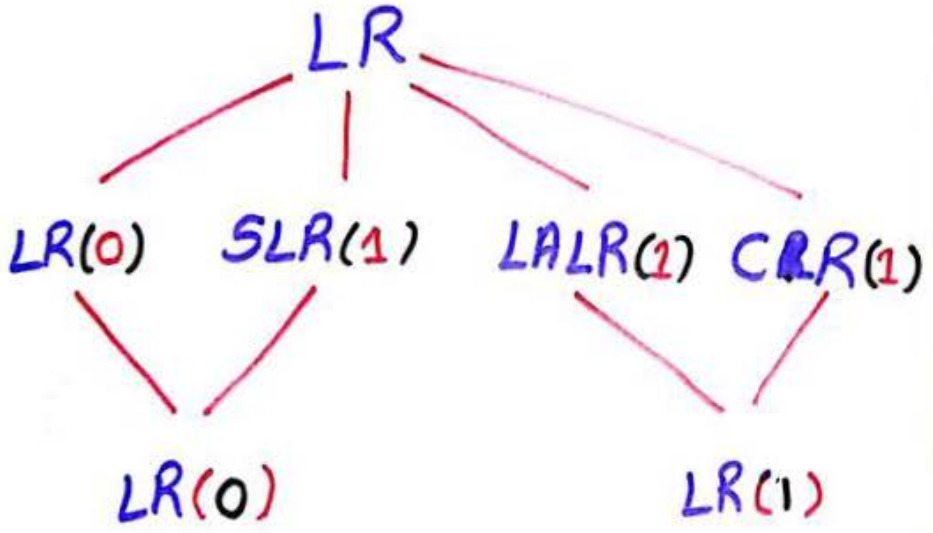
$V \rightarrow a \,/\, b \,/\, c \,/\, d$

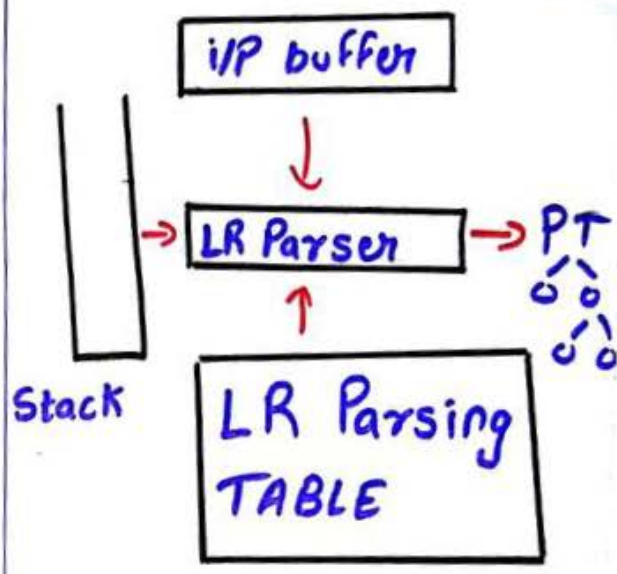
SRIC

Stack	R	Input	Comments
$	<	a+b*c*d$	shift a
$a	>	+b*c*d $	Reduce V→a
$V	<	+b*c*d $	shift +
$V+	<	b*c*d $	shift b
$V+b	>	*c*d $	Reduce V→b
$ V+V	<	*c*d $	shift *
$V+V*	<	c*d $	shift c
$V+V*c	>	*d $	Reduce V→c
$V+V*V	>	*d $	Reduce T→V
$V+V*T	>	*d $	Reduce E→T
$V+V*E	>	*d $	

- No Production found for reduction operation
- Since, parsing process failed to complete, the given input cannot be parsed by the given grammer using OPP method :)

LR Parsers





Bottom UP Parser

Steps to Solve LR(0) Sums

1. Augment the given grammer
2. Draw Conical Collection of LR(0) item/DFD
3. Number the Production
4. Create the parsing table
5. Stack implementation
6. Draw parse tree

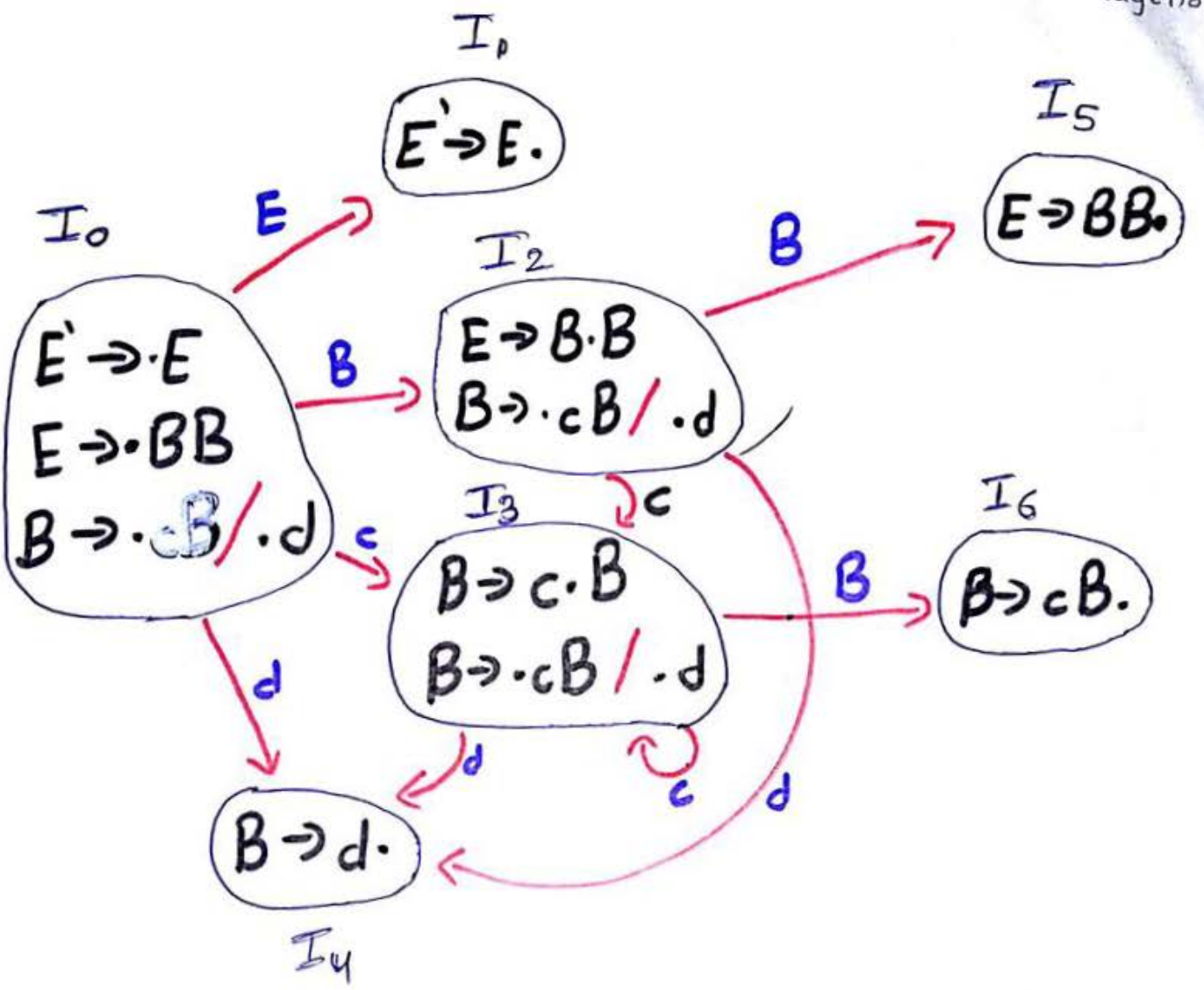
$E \rightarrow BB$
$B \rightarrow cB / d$

cc dd $

Step 1) Augment the given Grammer

$E' \rightarrow E$ (nakli grammer)
$E \rightarrow BB$
$B \rightarrow cB / d$

Step 2) Draw conical collection of LR(0) item / DFD



Step 3) [illegible] Parsing table

Number the Production

$E' \to E$

$E \to BB$ (1)

$B \to cB$ (2) / d (3)

4) Parsing table

State	Action			Goto	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

Step 5) Stack implementation

$E' \rightarrow E$

$E \rightarrow BB$ (1)

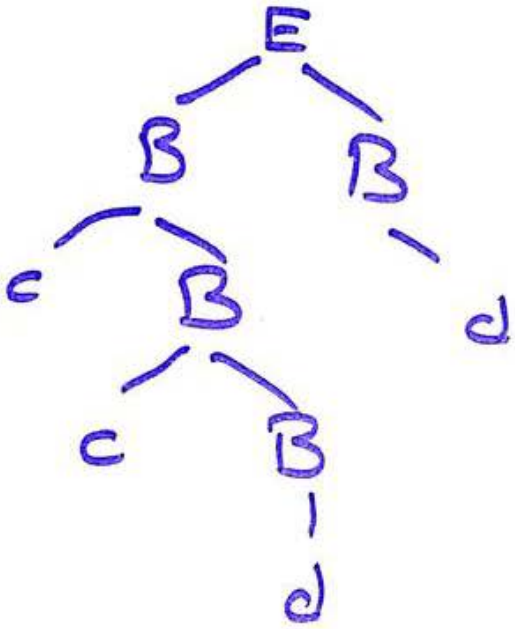
$B \rightarrow cB$ (2) / d (3)

c c d d $

Stack	Input	Action
$0	ccdd$	Shift c Goto S3
$0c3	cdd$	shift c Goto S3
$0c3c3	dd$	Shift d Goto S4
$0c3c3d4	d$	reduce r_3 B→d
$0c3c3B6	d$	reduce to r_2 B→cB
$0c3B6	d$	reduce to r_2 B→cB
$0B2	d$	Shift d Goto S4
$0B2d4	$	reduce to r_3 B→d
$0B2B5	$	reduce to r_1 E→BB
$0E1	$	Accepted ☺

www.Lastmomenttuitions.com

Step 6) Draw the Parse tree



SLR(1)

State	Action			Goto	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_3	S_4			5
I_3	S_3	S_4			6
I_4	r_3	r_3	r_3		
I_5			r_1		
I_6	r_2	r_2	r_2		

$B \rightarrow d$

$E \rightarrow BB$

$B \rightarrow cB$

Conflict

RR

$A \to a$ — ①
$B \to b$ — ②
$C \to aB$ — ③

I_1

$A \to a.$
$B \to b.$

	Action			Goto
	a	b	$	
I_1	r_1/r_2	r_1/r_2	r_1/r_2	

SR

$C \to .Ba$ — ①
$B \to d$ — ②

I_1

$C \to B.a$
$B \to d.$

$\xrightarrow{a}$

I_2

$C \to Ba.$

	Action			Goto
	a	d	$	
I_1	s_2/r_2	r_2	r_2	
I_2				

$E \rightarrow T+E \,/\, T$

$T \rightarrow i$

i) check Grammer is LR(0) or not

ii) check Grammer is SLR(1) or not

Step 1) Augment the given grammer

$E' \rightarrow E$

$E \rightarrow T+E \,/\, T$

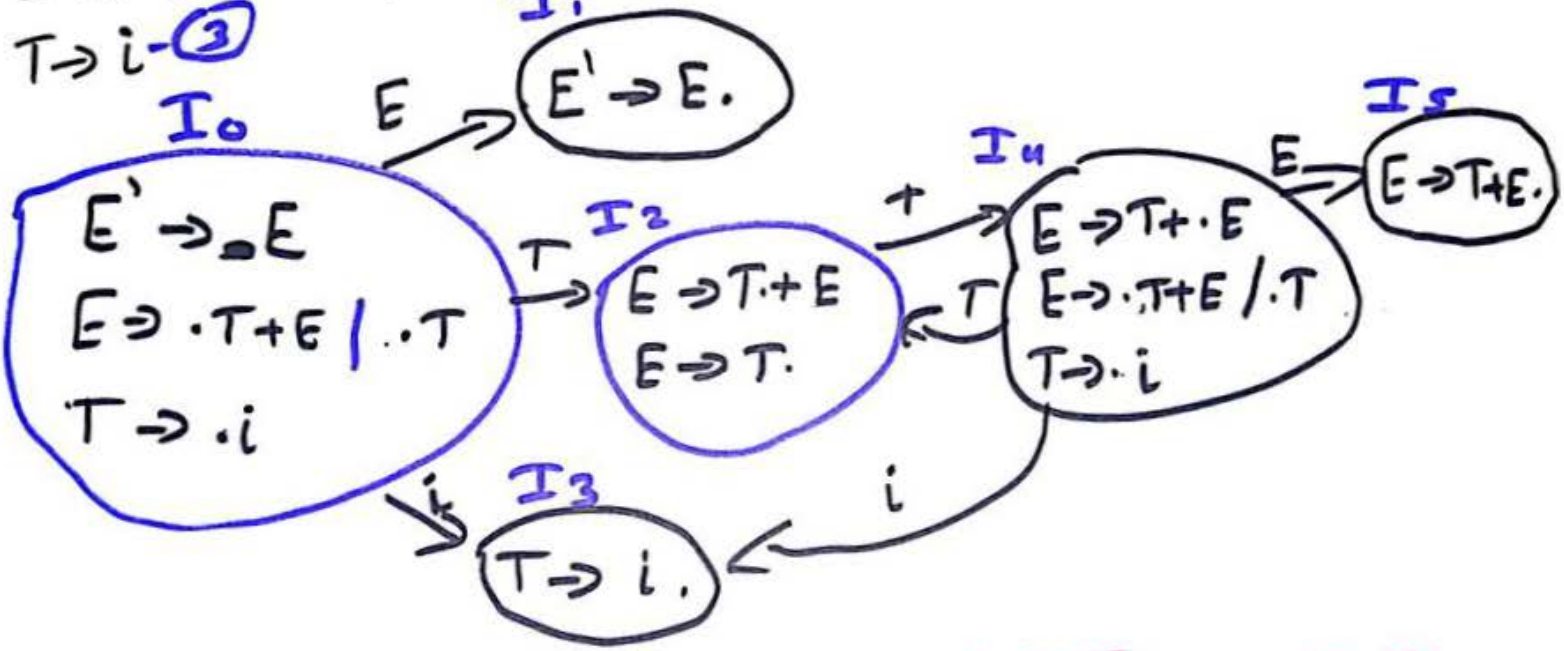
$T \rightarrow i$

step 2) Draw conical collection of LR(0) item

$E' \to E$ — (1)

$E \to T+E \;/\; T$ — (2)

$T \to i$ — (3)



$E \to T$ — r_2

$T \to i$.

$E \to T+E$

State	Actions			Goto	
	+	i	$	E	T
I_0		S3		1	2
I_1			Accept		
I_2	S4/r_2	r_2	r_2		
I_3	r_3	r_3	r_3		
I_4		S3		5	2
I_5	r_1	r_1	r_1		

$E' \rightarrow E$

$E \rightarrow T + E$ (1) / T (2)

$T \rightarrow i$ (3)

NOT AN LR(0)

State	Actions			Goto	
	+	i	$	E	T
I_0		S3		1	2
I_1			Accept		
I_2	S4		r_2		
I_3	r_3		r_3		
I_4		S3		5	2
I_5			r_1		

follow (T) = { +, $ }

follow (E) = { $ }

∴ It is SLR(1)

CLR(1) & LALR(1)

LR(1) item = LR(0) item + Look ahead

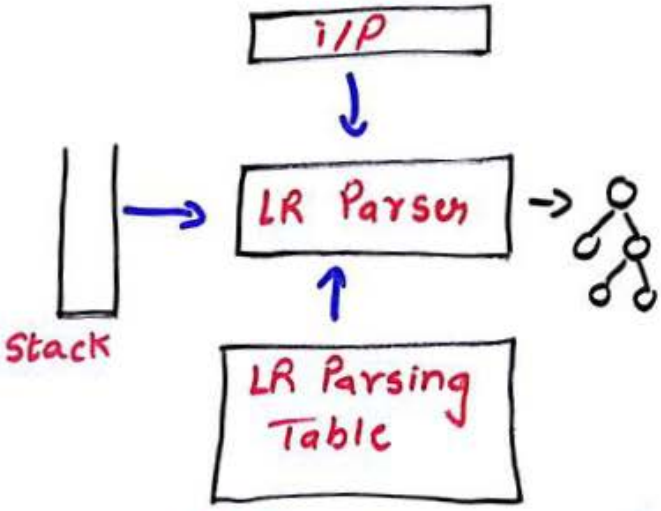
LR(0) → Put reduce in full row

SLR(1) → Put reduce in follow of P

CLR(1) & LALR(1) → Put reduce only on Lookahead

E → .eB , c/d

E → eB. , c/d

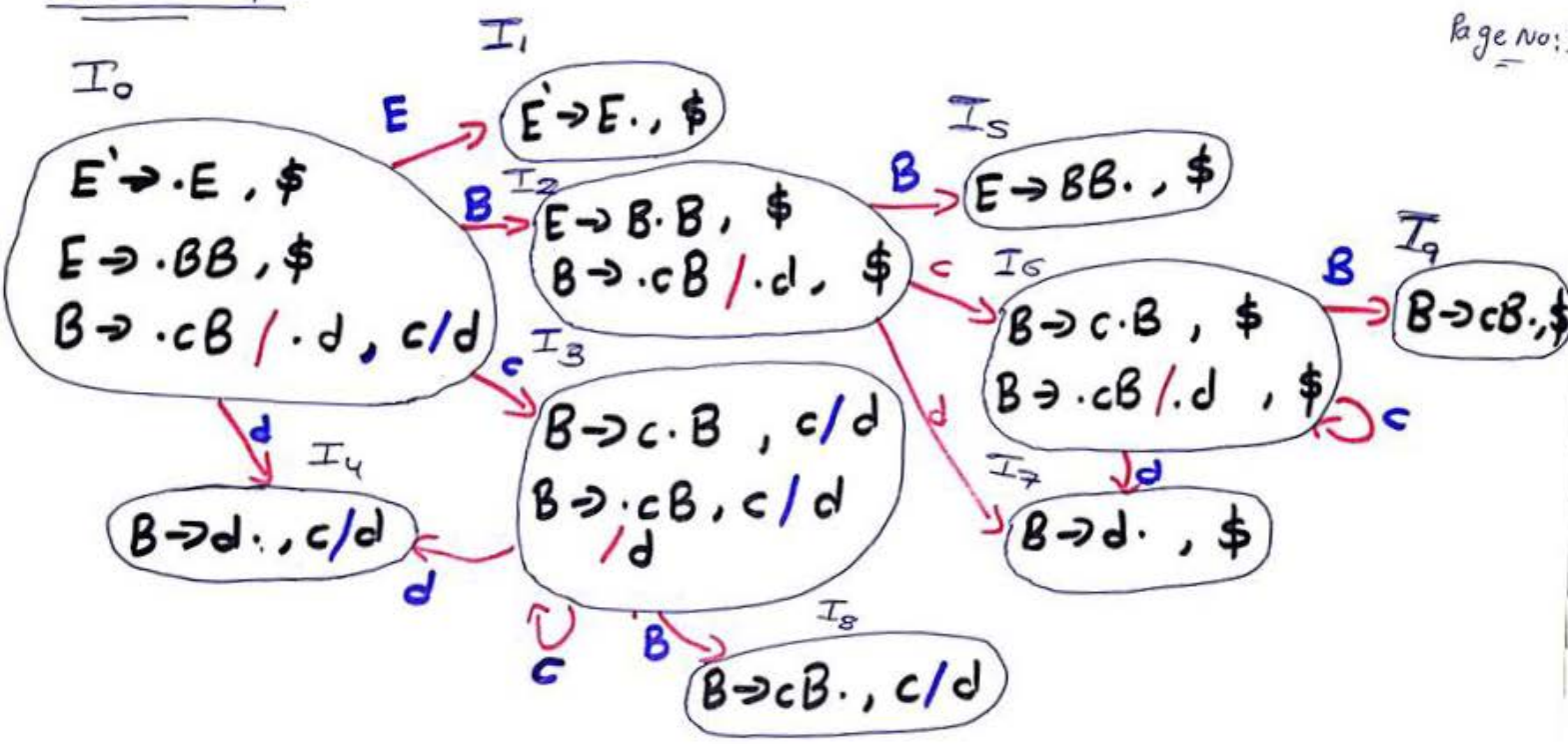


using LR(1) item

Steps to Solve CLR(1) Sums

1. Augment the given grammer
2. Draw Conical Collection of LR(1) item/DFD
3. Number the Production
4. Create the parsing table
5. Stack implementation
6. Draw parse tree

Step 1 and step 2



Step 3

Number the Production

$E' \to E$

$E \to BB$ (1)

$B \to cB$ (2) / d (3)

I_3, I_6

I_4, I_7

I_8, I_9

Step 4) Parsing table

State	Action			Goto	
	c	d	$	E	B
I_0	S_3	S_4		1	2
I_1			Accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

Step 5) and step 6

Baki stack implementation and parse tree is same as LR(0)

LALR(1)

www.Lastmomenttuitions.com

Parsing table for LALR(1)

State	Action			Goto	
	c	d	$	E	B
I_0	S_{36}	S_{47}		1	2
I_1			Accept		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
I_{89}	r_2	r_2	r_2		

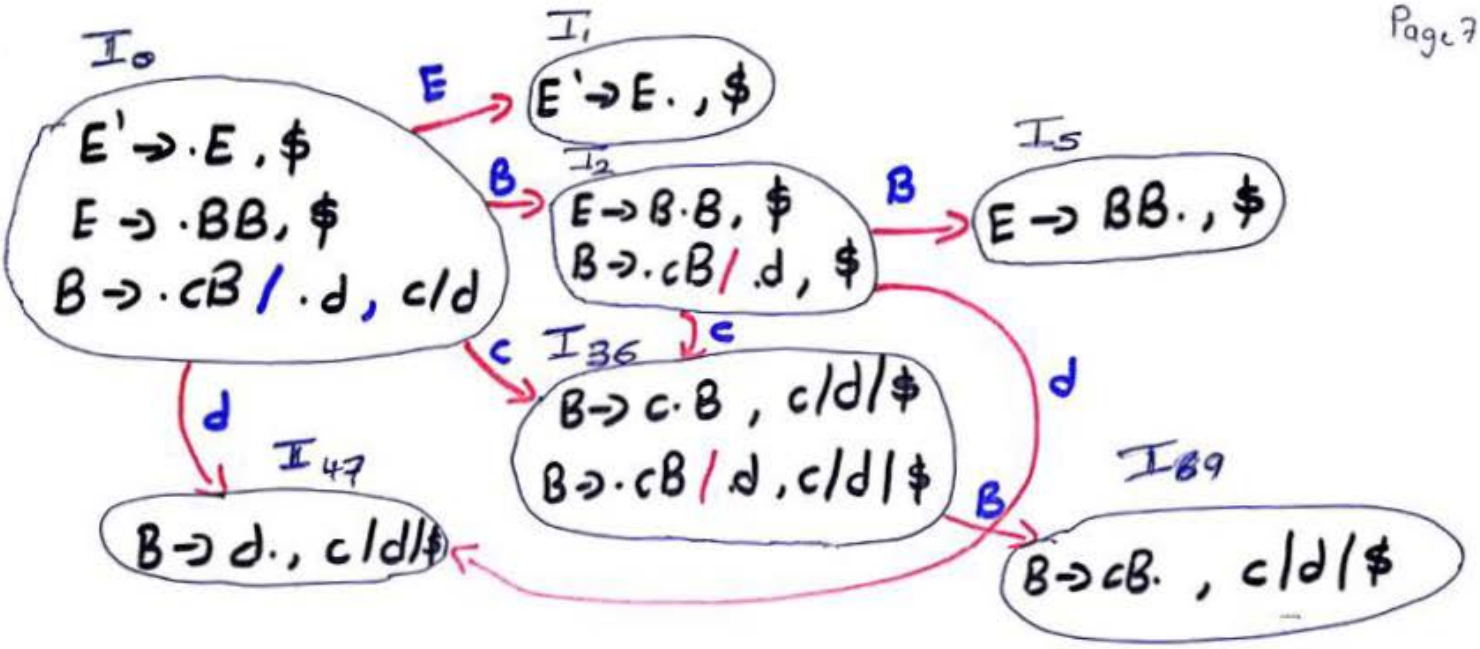
www.Lastmomentkuitions.com

After merging

$I_3 + I_6 = I_{36}$

$I_4 + I_7 = I_{47}$

$I_8 + I_9 = I_{89}$


I0
E'→·E, $
E→·BB, $
B→·cB / ·d, c/d
E
I1
E'→E·, $
B
I2
E→B·B, $
B→·cB / ·d, $
B
I5
E→BB·, $
c
I36
B→c·B, c/d/$
B→·cB / ·d, c/d/$
c
d
I47
B→d·, c/d/$
d
B
I89
B→cB·, c/d/$

LALR(1)

Module 09: Syntax Directed Translation

Contents:
Syntax directed definitions, Inherited and Synthesized attributes, Evaluation order for SDDs, S attributed Definitions, L attributed Definitions.

Syntax-Directed Translation:

It refers to a method of compiler implementation where the source language translation is completely driven by the parser, i.e., based on the syntax of the language. The parsing process and parse trees are used to direct semantic analysis and the translation of the source program. Almost all modern compilers are syntax-directed.

Syntax-Directed Translation can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation. Such grammars are called attribute grammars.

We augment a grammar by associating **attributes** with each grammar symbol that describes its properties. With each production in a grammar, we give **semantic rules/ actions**, which describe how to compute the attribute values associated with each grammar symbol in a production.

There are two ways to represent the semantic rules/ actions associated with grammar symbols.

- Syntax-Directed Definitions (SDD)
- Syntax-Directed Translation Schemes (SDT)

Between the two notations, syntax-directed definitions can be more readable and hence more useful for specifications. However, translation schemes can be more efficient and hence more useful for implementations. The most general approach to syntax-directed translation is to construct a parse tree or a syntax tree and then to compute the values of attributes at the nodes of the tree by visiting the nodes of the tree.

In many cases, translation can be done during parsing, without building an explicit tree. We shall therefore study a class of syntax-directed translations called "**L-attributed translations**" (*L for left-to-right*), which encompass virtually all translations that can be performed during parsing. We also study a smaller class, called "**S-attributed translations**" (*S for synthesized*), which can be performed easily in connection with a bottom-up parse.

Syntax-Directed Definitions (SDD):

A Syntax-Directed Definition is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.

An **attribute** has a name and an associated value: a string, a number, a type, a memory location, an assigned register, strings. The strings may even be long sequences of code, say code in the intermediate language used by a compiler. If X is a symbol and a is one of its attributes, then we write **X.a** to denote the value of **a** at a particular parse-tree node labelled **X**. If we implement the nodes of the parse tree by records or objects, then the attributes of **X** can be implemented by data fields in the records that represent the nodes for **X**. The attributes are evaluated by the semantic rules attached to the productions.

Example:	PRODUCTION	SEMANTIC RULE
	$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel$ '+'

SDDs are highly readable and give high-level specifications for translations. But they hide many implementation details. For example, they do not specify order of evaluation of semantic actions.

Module 09: Syntax Directed Translation

Syntax-Directed Translation Schemes (SDT):

Syntax-directed translation schemes are a complementary notation to syntax directed Definitions. A Syntax- directed translation scheme (SDT) is a Context Free Grammar with program fragments called *semantic actions* embedded within production bodies. The position of semantic action in a production body determines the order in which the action is executed.

Example: $E \rightarrow E_1 + T$ {*print* '+'}

In the above rule, the action is positioned after the body of the production. Translation Schemes are more efficient than SDDs as they indicate the order of evaluation of semantic actions associated with a production rule. This also gives some information about implementation details.

Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal. Typically, SDT's are implemented during parsing, without building a parse tree.

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

Postfix SDT implementing the Desk Calculator

Inherited and Synthesized Attributes:

A nonterminal A can have both inherited and synthesized attributes. The difference is how they are computed by rules associated with a production at a node N of the parse tree.

- 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 only in terms of attribute values at the children of N and at N itself.

- 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 at N's parent, N itself and N's siblings.

An inherited attribute at node N cannot be defined in terms of attribute values at the children of node N. However, a synthesized attribute at node N can be defined in terms of inherited attribute values at node N itself. Terminals can have synthesized attributes, which are given to it by the lexical analyser (not the parser). There are no rules in an SDD giving values to attributes for terminals. Terminals do not have inherited attributes.

Module 09: Syntax Directed Translation

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
2)	$E \rightarrow E_1\ +\ T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1\ *\ F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (\ E\ )$	$F.val = E.val$
7)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.\text{lexval}$

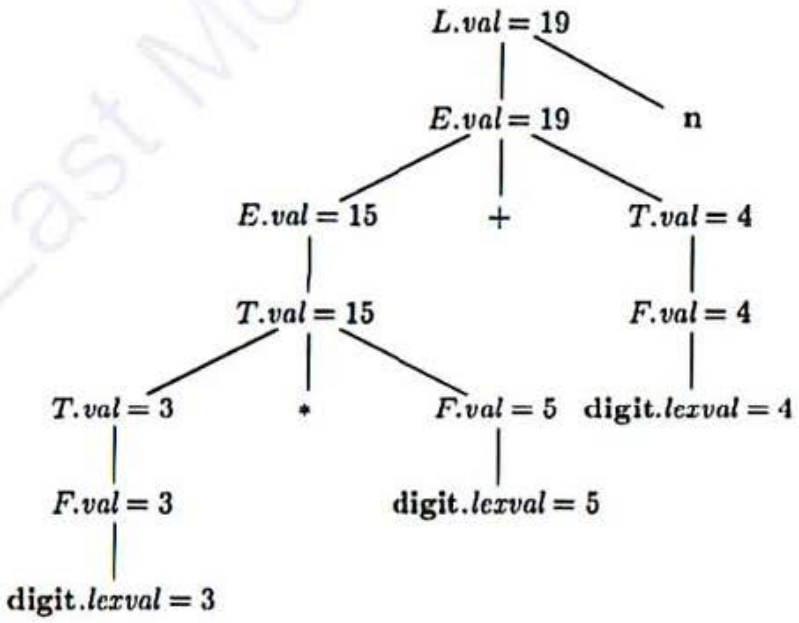
Syntax-directed definition of a simple desk calculator

The SDD in above figure is based on grammar for arithmetic expressions with operators + and *. It evaluates expressions terminated by an endmarker *n*. In the SDD, each of the nonterminals has a single synthesized attribute, called *val*. We also suppose that the terminal digit has a synthesized attribute *lexval*, which is an integer value returned by the lexical analyser.

Evaluating an SDD at the Nodes of a Parse Tree:

Parse tree helps us to visualize the translation specified by SDD. The rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree. A parse tree, showing the value(s) of its attribute(s) is called an **annotated parse tree.**

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree.



Annotated Parse Tree for 3 * 5 + 4 n

Module 09: Syntax Directed Translation

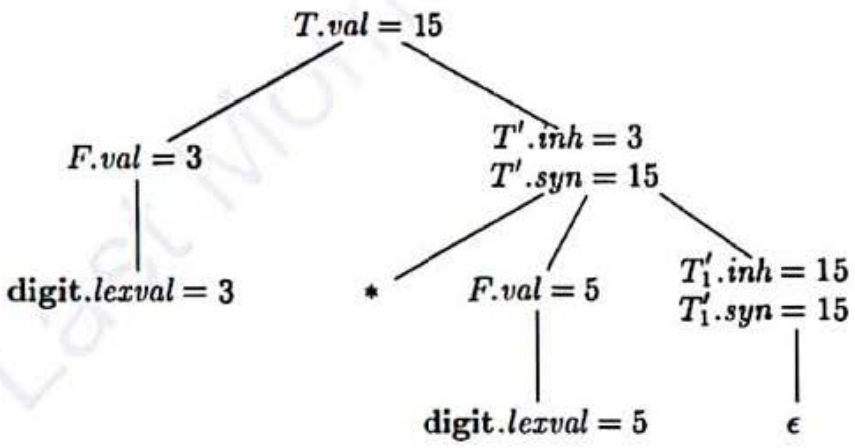
Above figure shows an annotated parse tree for the input string 3 * 5 + 4 **n**, constructed using the grammar and rules. The values of *lexval* are presumed supplied by the lexical analyser. Each of the nodes for the nonterminals has attribute *val* computed in a bottom-up order and we see the resulting values associated with each node. For instance, at the node with a child labelled *, after computing *T.val* = 3 and *F.val* = 5 at its first and third children, we apply the rule that says *T.val* is the product of these two values or 15.

Inherited attributes are useful when the structure of a parse tree does not match the abstract syntax of the source code. They can be used to overcome the mismatch due to grammar designed for parsing rather than translation.

In the SDD below, the nonterminal T′ has an inherited attribute *inh* as well as a synthesized attribute *val*. T′ inherits *F.val* from its left sibling F in the production T → FT′.

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$

SDD for expression grammar with inherited attributes

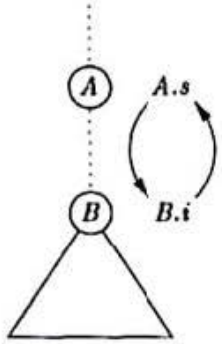


Annotated Parse Tree for 3*5

An SDD with both inherited and synthesized attributes does not ensure any guaranteed order; even it may not have an order at all. For example, consider nonterminals A and B, with synthesized and inherited attributes A.*s* and B.*i*, respectively, along with the production and rules. These rules are circular; it is impossible to evaluate either A.*s* at a node N or B.*i* at the child of N without first evaluating the other. The circular dependency of A.*s* and B.*i* at some pair of nodes in a parse tree below.

Module 09: Syntax Directed Translation

PRODUCTION	SEMANTIC RULES
$A \rightarrow B$	$A.s = B.i;$ $B.i = A.s + 1$



The circular dependency of $A.s$ and $B.i$ on one another

Evaluation Orders for SDD's:

"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

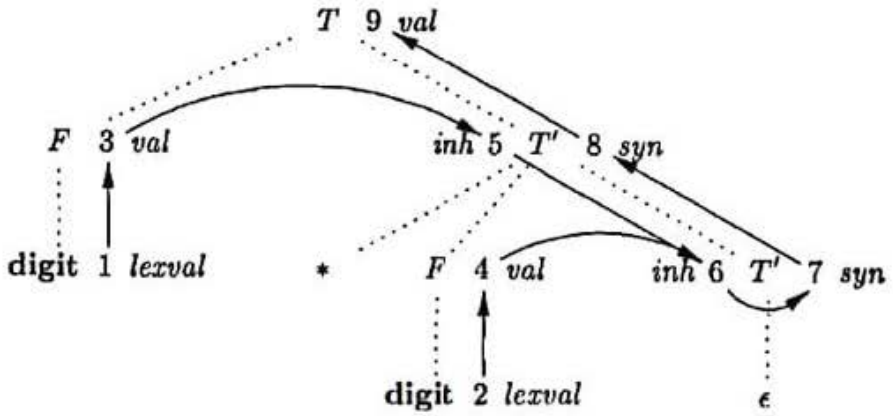
In addition to dependency graphs, we define two important classes of SDD's: the "*S-attributed*" and the more general "*L-attributed*" SDD's. The translations specified by these two classes fit well with the parsing methods we have studied, and most translations encountered in practice can be written to conform to the requirements of at least one of these classes.

Dependency Graphs:

A dependency graph shows the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules.

In more detail:

- For each parse-tree node, say a node labelled by grammar symbol X, the dependency graph has a node for each attribute associated with X.
- If a semantic rule associated with a production p defines the value of synthesized attribute A.b in terms of the value of X.c, then graph has an edge from X.c to A.b
- If a semantic rule associated with a production p defines the value of inherited attribute B.c in terms of value of X.a, then graph has an edge from X.a to B.c



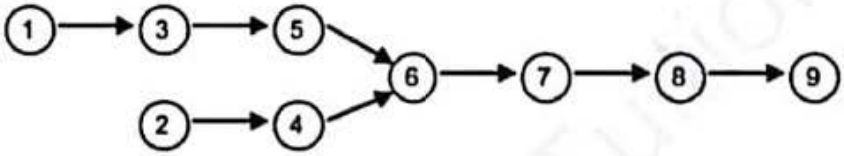
Module 09: Syntax Directed Translation

Topological Sort of the Dependency Graph:

A dependency graph characterizes the possible order in which we can evaluate the attributes at various nodes of a parse tree. If there is an edge from node M to N, then attribute corresponding to M first be evaluated before evaluating N. Thus the only allowable orders of evaluation are $N_1, N_2, ..., N_k$ such that if there is an edge from N_i to N_j then $i < j$. Such an ordering embeds a directed graph into a linear order, and is called a *topological sort* of the graph.

If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least one *topological sort*.

Example: Topological sorts for the above the dependency graph is shown in a simplified fashion with the nodes indicating the nodes of the parse tree and the edges indicating precedence of evaluation. It is apparent that the last section of the dependence graph denoted by the nodes { 6 7 8 9 } needs to be evaluated in sequence. The only flexibility of the order of evaluation is on the two sub-sequences { 1 3 5 } and { 2 4 } which can with any interleaving provided the relative order of their nodes is preserved are as below.



Base sequences are {1 3 5} and {2 4} with the suffix {6 7 8 9} being constant as the last nodes of the topological sorting need to remain fixed.

1 3 5 2 4 6 7 8 9, 1 3 2 5 4 6 7 8 9, 1 2 3 5 4 6 7 8 9, 1 3 2 4 5 6 7 8 9, 1 2 3 4 5 6 7 8 9,
1 2 4 3 5 6 7 8 9, 2 1 3 5 4 6 7 8 9, 2 1 3 4 5 6 7 8 9, 2 1 4 3 5 6 7 8 9, 2 4 1 3 5 6 7 8 9

S-Attributed Definitions:

An SDD is *S-attributed* if every attribute is synthesized. Attributes of an S-attributed SDD can be evaluated in bottom-up order of the nodes of parse tree. Evaluation is simple using post-order traversal of the parse tree and evaluating the attributes at a node N when the traversal leaves N for the last time. That is, we apply the function postorder, defined below, to the root of the parse tree.

```
postorder(N) {
        for (each child C of N, from the left) postorder(C);
        evaluate attributes associated with node N;
}
```

S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a postorder traversal. Specifically, postorder corresponds exactly to the order in which an LR parser reduces a production body to its head.

Module 09: Syntax Directed Translation

L-Attributed Definitions:

The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left (hence "L-attributed"). Each attribute must be either

1. Synthesized, or
2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X_1, X_2, \ldots, X_n$, and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:
   a. Inherited attributes associated with the head A.
   b. Either inherited or synthesized attributes associated with the occurrences of symbols $X_1, X_2, \ldots, X_{i-1}$ located to the left of X_i.
   c. Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i.

Example 1: The following definition is L-attributed. Here the inherited attribute of T′ gets its values from its left sibling F. Similarly, T_1' gets its value from its parent T′ and left sibling F.

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

Example 2: the definitions below are not L-attributed as B.*i* depends on its right sibling C′s attribute.

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

Last Moment Tutors

Module 09: Syntax Directed Translation

SDD for Simple Type Declarations:

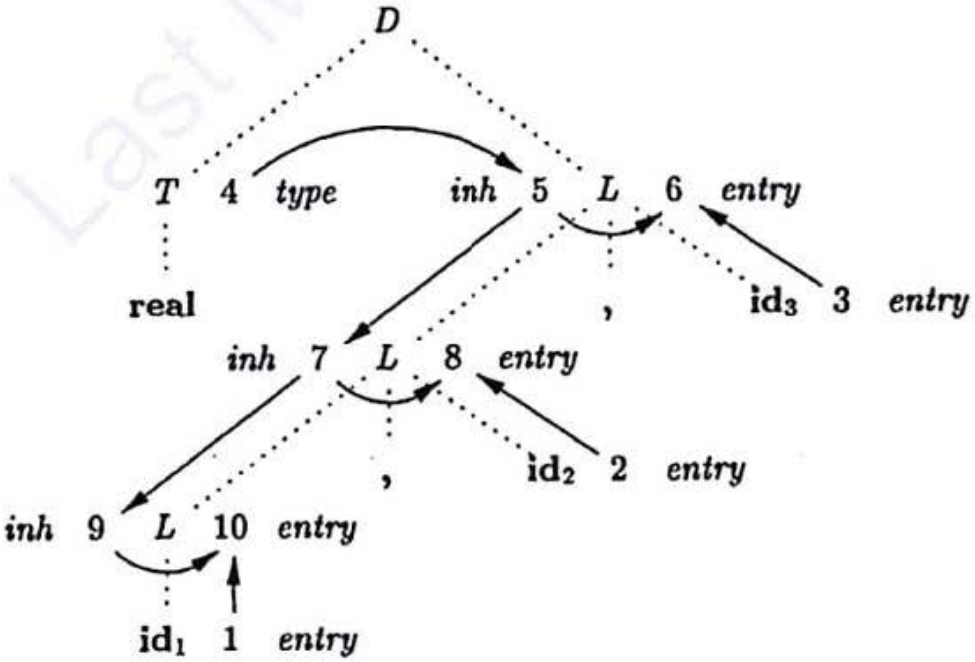
Nonterminal D represents a declaration, which, from production 1, consists of a type T followed by a list L of identifiers. T has one attribute, T.*type*, which is the type in the declaration D. Nonterminal L also has one attribute, which we call ink to emphasize that it is an inherited attribute. The purpose of L.*inh* is to pass the declared type down the list of identifiers, so that it can be the appropriate symbol-table entries. Productions 2 and 3 each evaluate the synthesized attribute T.*type*, giving it the appropriate value, integer or float. This type is passed to the attribute L.*inh* in the rule for production 1. Production 4 passes L.*inh* down the parse tree. That is, the value L_1.*inh* is computed at a parse-tree node by copying the value of L.*inh* from the parent of that node; the parent corresponds to the head of the production. Productions 4 and 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.

	PRODUCTION	SEMANTIC RULES
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow \textbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \textbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \textbf{id}$	$L_1.inh = L.inh$
		$addType(\textbf{id}.entry, L.inh)$
5)	$L \rightarrow \textbf{id}$	$addType(\textbf{id}.entry, L.inh)$

The function *addType* properly installs the type L.*inh* as the type of the represented identifier. Note that the side effect, adding the type info to the table, does not affect the evaluation order.

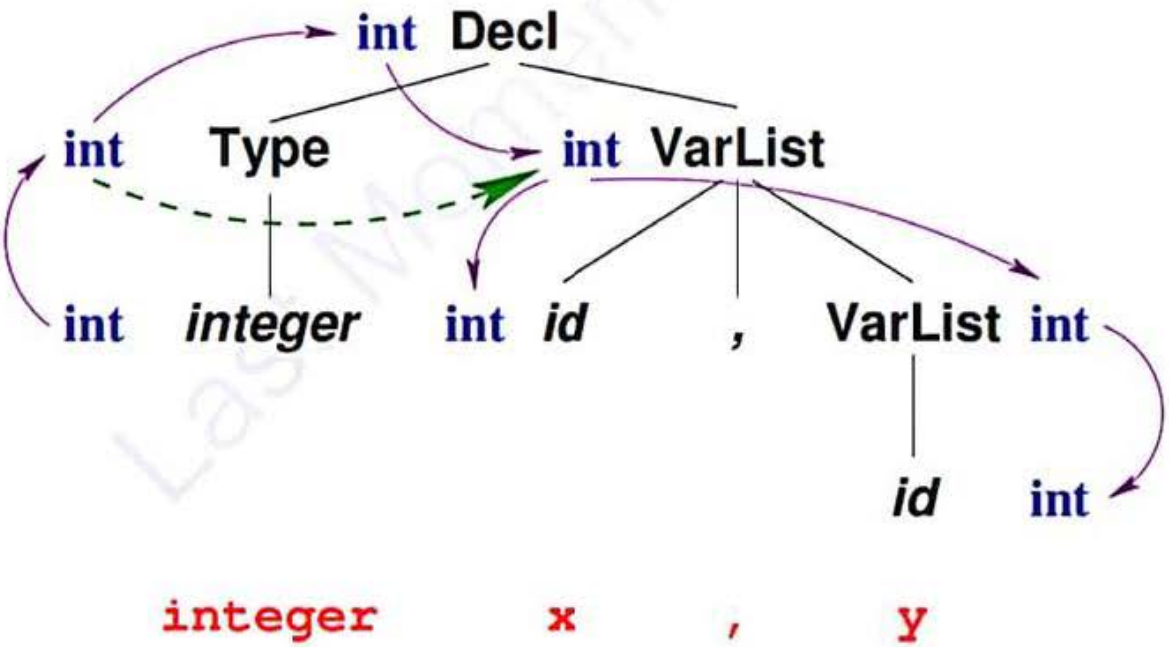
A dependency graph for the input string **float id$_1$, id$_2$, id$_3$** is shown below.



Decl	$\longrightarrow$	*Type VarList*
Type	$\longrightarrow$	`integer`
Type	$\longrightarrow$	`float`
VarList	$\longrightarrow$	`id` , *VarList*
VarList	$\longrightarrow$	`id`

Decl	$\longrightarrow$	*Type VarList*	$\{VarList.type := Type.type\}$
Type	$\longrightarrow$	`integer`	$\{Type.type := int\}$
Type	$\longrightarrow$	`float`	$\{Type.type := float\}$
VarList	$\longrightarrow$	`id` , $VarList_1$	$\{VarList_1.type := VarList.type;$ $id.type := VarList.type\}$
VarList	$\longrightarrow$	`id`	$\{id.type := VarList.type\}$

Information Flow for "Type" Example



Intermediate Code Generation

Content

Basics of Intermediate code

Way of representation

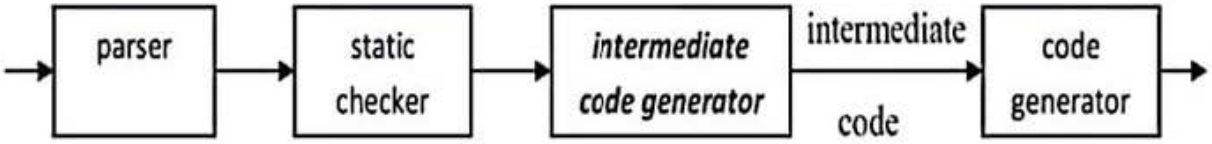
- Graphical
  - Syntax tree
  - DAG
- Postfix
- **Three address code**
  -Types of three address code
  -Implementation of Three address code
  Quadruple
  Triple
  Indirect Triple
- Sums of three Address code
- Backpatching

Intermediate Code Generation

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine.

This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

(bahut bade code ko chote tukdo me divide karta hai jisko hum intermediate kehte hai)



Intermediate Language

Three ways of intermediate representation:

- Graphical Representation (Syntax tree & DAG)
- Postfix notation
- Three address code

Graphical Representations:

Syntax Tree & DAG:

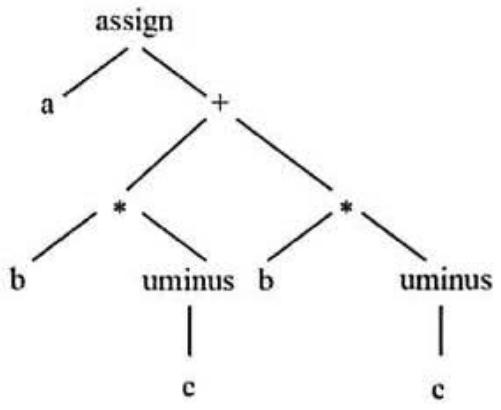
A **Syntax Tree** represents the natural hierarchical structure of a source program. A **DAG (Directed Acyclic Graph)** gives the same information but in a more compact way because common subexpressions are identified.

(yeh dono simply code ko graphical form me represent karte hai)

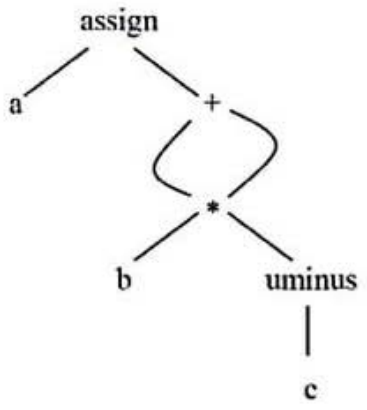
A syntax tree and dag for the assignment statement **a: = b * - c + b * - c** are as follows:

(uminus maltab minus hota hai)

Note : DAG ka videos last moment tuition channel ki spcc series me hai pehle voh dekh lena)



(a) Syntax Tree



(b) DAG

Postfix notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is

a b c uminus * b c uminus * + assign

Three-Address Code (TAC, 3AC):

Three-address code is a sequence of statements of
the general form x: = y *op* z
Where x, y and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed-point or floating-point arithmetic operator, or a logical operator on Boolean valued data. Thus, a source language expression like x+ y*z might be translated into a sequence

t1: = y * z

t2: = x + t1

Where t1 and t2 are compiler-generated temporary names.

Advantages of three-address code:

- The unravelling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.

- The use of names for the intermediate values computed by a program allows three address code to be easily rearranged – unlike postfix notation.

Three-address code is a linearized representation of a Syntax Tree or a DAG in which explicit names correspond to the interior nodes of the graph. The Syntax Tree and DAG are represented by the three-address code sequences. Variable names can appear directly in three address statements.

Three-address code corresponding to the Syntax Tree and DAG given above.

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

code for syntax Tree

$t_1 := -c$
$t_2 := b * t_1$
$t_5 := t_2 + t_2$
$a := t_5$

Code for the DAG

The reason for the term "*Three Address Code*" is that each statement usually contains three addresses, two for the operands and one for the result.

Types of Three-Address Statements:

1. **Assignment statements** of the form ***x: = y op z***, where op is a binary arithmetic or logical operation
   (example : a:=b+c).

2. **Assignment instructions** of the form ***x: = op y***, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
   (example : a: = - c)

3. **Copy statements** of the form ***x: = y*** where the value of *y* is assigned to *x*.
   (example : x=y)

4. The **unconditional jump *goto L***. The three-address statement with label L is the next to be executed.
   (isme koi condition nahi hoti if x aya toh L pe jao nahi yeh bolta hai direct L pe jao)

5. **Conditional jumps** of the form ***if x goto L*** and ***if false x goto L***. These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise,
   the following three-address instruction in sequence is executed next, as usual.
   (isme condition hoti hai if X aya toh L pe udi maro ya fir if false X aya toh L pe udi maro)

6. **Conditional jumps** such as ***if x relop y goto L***. These instruction applies a relational operator (<, =, >=, etc.) to *x* and *y*, and executes the statement with label L next if *x*
   stands in relation *relop to y*. If not, the three – address statement following if *x relop y* goto L is executed next, as in the usual sequence.

   (yeh thoda alag hai upper wale se isme relational operator hote hai example If (x < y) then go to L)

7. ***param x* and *call p, n* for procedure calls** and ***return y***, where y representing a returned

value is optional. For example, *param* x_1

param

x_2

. . .

Param

x_n

call p, n

generated as part of a call of the procedure p (x_1, x_2, ..., x_n).

(yeh procedure ya function call me use hota hai jaha hum har parameter ko alag se likhte hai)

8. **Indexed assignments** of the form x: = y[i] and x[i]: = y.

(yeh simple array ki position ko point karta hai)

9. **Address and pointer assignments** of the form x: = &y, x: = *y, and *x: =y

(yeh pointer ki value of assign karta hai)

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are:

- Quadruples
- Triples
- Indirect triples

Quadruples:

• A quadruple is a record structure with four fields, which are op, arg1, arg2 and result. • The op field contains an internal code for the operator. The three-address statement x: = y op z is represented by placing y in arg1, z in arg2 and x in result. • The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

(isme 4 cheeze hoti hai 3 argument (arg matlab hota hai koi bhi variable x,y,z) and ek operator(+, - , *)

Example : x:= a + b (where x,a,b are argument and +(plus) is operator)

Triples:

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.

- If we do so, three-address statements can be represented by records with only three fields: op, arg1 and arg2.

- The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure (for temporary values).

- Since three fields are used, this intermediate code format is known as triples

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

(a) Quadruples

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

(explaination : simple hai yaar Quadraple me t2 dekh t2 : =t1*b ab t1 dekhte hai t1= - c)

(Triple me temporary variable ki jagah seedha address daal diya hai aur result ka column hata diya hai triple me second line dekh t1 ki jagah maine direct (0) dala hai bcoz quadruple ki first line me t1 ki value (0) me store ho rahi hai .)

Indirect Triples:

• Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.

• For example, let us use an array statement to list pointers to triples in the desired order. Then the triples shown above might be represented as follows .

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg1*	*arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

(same to same triple jaise hota hai bus yeh alag se ek table maintain karta jisme hum triple ke pointer store karte hai toh agar ek bhi statement idhar udhar hua toh atleast pointer toh hai)

Module 10: Intermediate Code Generation

Question:

Generate three address code for given expression

```
while (a < b) do
        if (c < d) then
                x = y + z
        else
                x = y – z
```

Solution:

```
(1)     if (a < b) goto 3
(2)     goto 11
(3)     if (c < d) goto 5
(4)     goto 8
(5)     t1 = y + z
(6)     x = t1
(7)     goto 10
(8)     t2 = y – z
(9)     x = t2
(10)    goto 1
(11)
```

Question:

Convert the following code into corresponding TAC

```
{
        int i;   int j;   float a[100];   float v;        float x;
        while (true) {
                do i = i + 1;    while (a[i] < v);
                do j = j - 1;    while (a[j] > v);
                if (i >= j)      break;
                x = a[i];        a[i] = a[j];      a[j] = x;
        }
}
```

Solution:

```
(1)     i = i + 1
(2)     t1 = a[i]
(3)     if (t1 < v) goto 1
(4)     j = j – 1
(5)     t2 = a[j]
(6)     if (t2 > v) goto 4
(7)     if !(i >= j) goto 8
(8)     goto 14
(9)     x = a[i]
(10)    t3 = a[j]
(11)    a[i] = t3
(12)    a[j] = x
(13)    goto 1
(14)
```

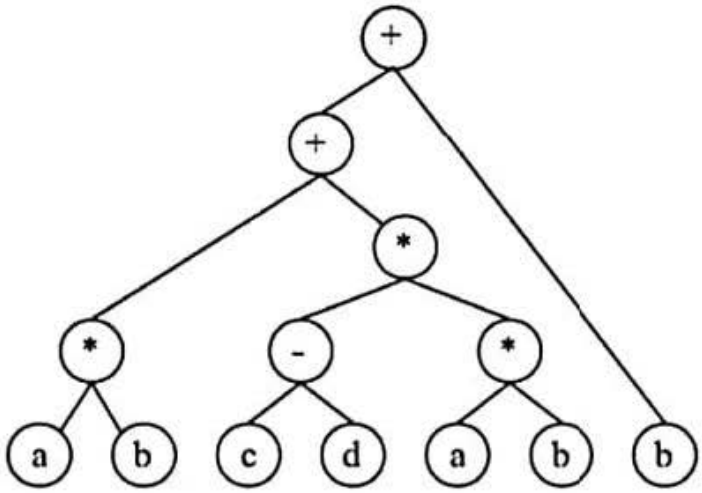
Last Moment Tuitions

Module 10: Intermediate Code Generation

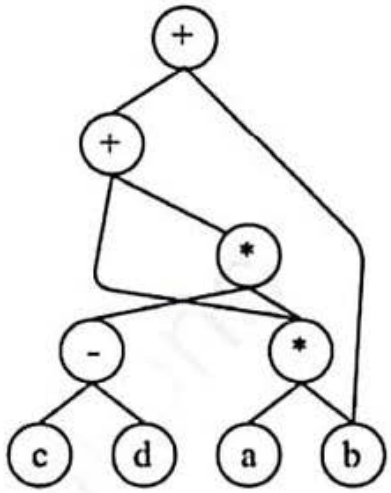
Question:

Draw Syntax tree and Directed Acyclic Graph (DAG) for expression
(a * b) + (c – d) * (a * b) + b

Solution:



Syntax Tree



DAG

Last Moment Tuition

Backpatching: The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes.

- First, construct a syntax tree for the input, and then walk the tree in depth first order, computing the translations.
- The main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated.
- Hence, a series of branching statements with the targets of the jumps left unspecified is generated.
- Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. This subsequent filling in of labels is called as backpatching.
- To manipulate lists of labels, we use three functions:

1. makelist (i) creates a new list containing only i, an index into the array of quadruples; makelist returns a pointer to the list it has made.

2. merge (p1, p2) concatenates the lists pointed to by p1 and p2, and returns a pointer to the concatenated list.

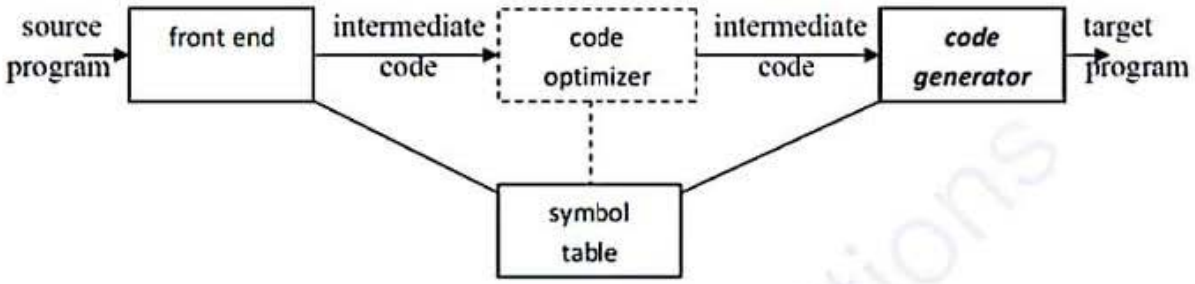
3. backpatch (p, i) inserts i as the target label for each of the statements on the list pointed to by p

Module 11: Code Generation

Content:
Issues in the design of Code Generator, Basic Blocks and Flow graphs, Code Generation algorithm, DAG representation of Basic Block.

Introduction:

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.



Position of code generator

11.1 Issues in the Design of a Code Generator:

The following issues arise during the code generation phase:

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

The input to the code generation consists of the intermediate representation of the source program produced by front end, together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

Intermediate representation can be:

- Linear representation such as postfix notation
- Three address representation such as quadruples
- Virtual machine representation such as stack machine code
- Graphical representations such as syntax trees and DAGs.

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

The output of the code generator is the target program. The output may be:

a. **Absolute machine language**: It can be placed in a fixed memory location and can be executed immediately.
b. **Relocatable machine language**: It allows subprograms to be compiled separately.
c. **Assembly language**: Code generation is made easier.

Module 11: Code Generation

3. Memory management:

Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator. It makes use of symbol table, i.e., a name in a three-address statement refers to a symbol-table entry for the name. Labels in three-address statements have to be converted to addresses of instructions.

For example, *j:* **goto** *i* generates jump instruction as follows:

- if $i < j$, a backward jump instruction with target address equal to location of code for quadruple *i* is generated.
- if $i > j$, the jump is forward. We must store on a list for quadruple *i* the location of the first machine instruction generated for quadruple *j*. When *i* is processed, the machine locations for all instructions that forward jumps to *i* are filled.

4. Instruction selection:

The instructions of target machine should be complete and uniform. Instruction speed and machine idioms are important factors when efficiency of target program is considered. The quality of the generated code is determined by its speed and size. The former statement can be translated into the latter statement as shown below:

```
a := b+c
d := a+e
```

Can be translated into

```
MOV  b, R0
ADD  c, R0
MOV  R0, a
MOV  a, R0
ADD  e, R0
MOV  R0, d
```

→ This can be eliminated

5. Register allocation

Instructions involving register operands are shorter and faster than those involving operands in memory.

The use of registers is subdivided into two sub problems:

- ***Register allocation*** – the set of variables that will reside in registers at a point in the program is selected.
- ***Register assignment*** – the specific register that a variable will reside in is picked.

Certain machines (IBM System/370) requires even-odd register pairs for some operands and results. For example, consider the division instruction of the form:

D x, y

where,

x – dividend occupies an even register in even/odd register pair

y – divisor

After division, even register holds the remainder; odd register holds the quotient

6. Evaluation order

The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

Module 11: Code Generation

11.2 Basic Blocks and Flow Graphs:

Basic Blocks:

A ***basic block*** is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end. The following sequence of three-address statements forms a basic block:

$t_1 := a * a$
$t_2 := a * b$
$t_3 := 2 * t_2$
$t_4 := t_1 + t_3$

Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

Method:

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are of the following:
   a. The first statement is a leader.
   b. Any statement that is the target of a conditional or unconditional goto is a leader.
   c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example 1: Consider the following code for dot product of two vectors *a* and *b* of length 20

```
begin
        prod := 0;
        i := 1;
        do begin
                prod := prod + a[i] * b[i];
                i := i + 1;
        end
        while i <= 20
end
```

Module 11: Code Generation

The three-address code for the above source program is given as:

```
1)   prod := 0
2)   i := 1
3)   t1 := 4* i
4)   t2 := a[t1]          /*compute a[i] */
5)   t3 := 4* i
6)   t4 := b[t3]          /*compute b[i] */
7)   t5 := t2*t4
8)   t6 := prod+t5
9)   prod := t6
10)  t7 := i+1
11)  i := t7
12)  if i<=20 goto (3)
```

Basic block 1: Statement (1) to (2)
Basic block 2: Statement (3) to (12)

Example 2: Consider the following source code which turns a 10 x 10 matrix *a* into an identity matrix.

```
for i from 1 to 10 do
    for j from 1 to 10 do
        a[i, j] = 0.0;
for i from 1 to 10 do
    a[i, i] = 1.0;
```

The three-address code for the above source program is given as:

```
1)   i = 1
2)   j = 1
3)   t1 = 10 * i
4)   t2 = t1 + j
5)   t3 = 8 * t2
6)   t4 = t3 – 88
7)   a[t4] = 0.0
8)   j = j + 1
9)   if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i – 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```

Moment Tuitions

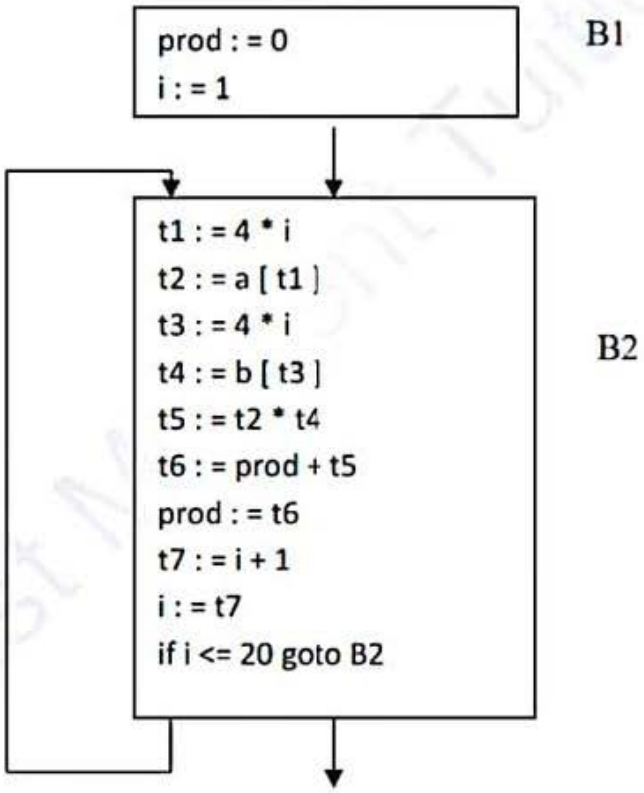
First, instruction 1 is a leader by rule (1) of Algorithm. To find the other leaders, we first need to find the jumps. In this example, there are three jumps, all conditional, at instructions 9, 11 and 17. By rule (2), the targets of these jumps are leaders; they are instructions 3, 2 and 13, respectively. Then, by rule (3), each instruction following a jump is a leader; those are instructions 10 and 12. Note that no instruction follows 17 in this code, but if there were code following, the 18th instruction would also be a leader.

We conclude that the leaders are instructions 1, 2, 3, 10, 12, and 13. The basic block of each leader contains all the instructions from itself until just before the next leader. Thus, the basic block of 1 is just 1, for leader 2 the block is just 2. Leader 3, however, has a basic block consisting of instructions 3 through 9, inclusive. Instruction 10's block is 10 and 11; instruction 12's block is just 12, and instruction 13's block is 13 through 17.

Flow Graphs

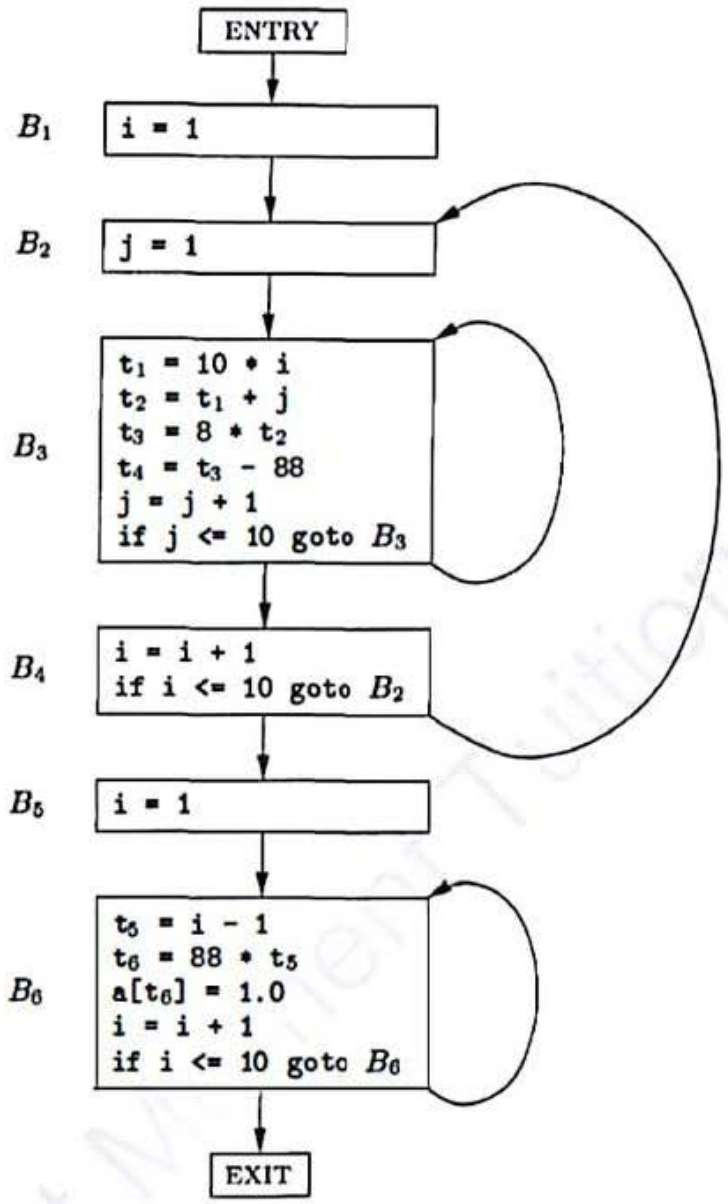
Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program. The nodes of the flow graph are basic blocks. It has a distinguished initial node.

E.g.: Flow graph for the vector dot product is given as follows:



B1 is the *initial* node. B2 immediately follows B1, so there is an edge from B1 to B2. The target of jump from last statement of B1 is the first statement B2, so there is an edge from B1 (last statement) to B2 (first statement). B1 is the *predecessor* of B2, and B2 is a *successor* of B1.

Module 11: Code Generation



The set of basic blocks constructed in Example 2 yields the flow graph shown above. The entry points to basic block B1, since Bl contains the first instruction of the program. The only successor of Bl is B2, because Bl does not end in an unconditional jump and the leader of B2 immediately follows the end of B1.

Block B3 has two successors. One is itself, because the leader of B3, instruction 3, is the target of the conditional jump at the end of B3, instruction 9. The other successor is B4, because control can fall through the conditional jump at the end of B3 and next enter the leader of B4.

Only B6 points to the exit of the flow graph, since the only way to get to code that follows the program from which we constructed the flow graph is to fall through the conditional jump that ends B6.

Module 11: Code Generation

Next-Use Information:

If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.

Input: Basic block B of three-address statements

Output: At each statement i: $x = y$ **op** z, we attach to i the liveliness and next-uses of x, y and z.

Method: We start at the last statement of B and scan backwards.

1. Attach to statement i the information currently found in the symbol table regarding the next-use and liveliness of x, y and z.
2. In the symbol table, set x to "not live" and "no next use".
3. In the symbol table, set y and z to "live", and next-uses of y and z to i.

Symbol Table:

Names	Liveliness	Next-use
x	Not live	No next-use
y	Live	i
z	Live	i

A Simple Code Generator:

A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements. For example: consider the three-address statement **a := b+c.** It can have the following sequence of codes:

ADD R_j, R_i	Cost = 1	// if R_i contains b and R_j contains c
	(or)	
ADD c, R_i	Cost = 2	// if c is in a memory location
	(or)	
MOV c, R_j ADD R_j, R_i	Cost = 3	// move c from memory to R_j and add

Register and Address Descriptors:

- A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- An address descriptor stores the location where the current value of the name can be found at run time.

Module 11: Code Generation

11.3 A Code-Generation Algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block.

For each statement $x := y$ **op** z

1. Set location $L = getReg(y, z)$
2. If $y \notin L$ then generate
   MOV y', L
   where y' denotes one of the locations where the value of y is available
3. Generate
   OP z', L
   where z' is one of the locations of z; Update register/address descriptor of x to include L.
4. If y and/or z has no next use and is stored in register, update register descriptors to remove y and/or z

1. Invoke a function *getReg* to determine the location L where the result of the computation **y op z** should be stored.
2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction **MOV y', L** to place a copy of y in L.
3. Generate the instruction **OP z', L** where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.
4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x: = y op z, those registers will no longer contain y or z.

Generating Code for Assignment Statements:

The assignment d: = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

t: = a – b
u: = a – c
v: = t + u
d: = v + u

with d live at the end.

Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t: = a - b	MOV a, R_0 SUB b, R_0	R_0 contains t	t in R_0
u: = a - c	MOV a, R_1 SUB c, R1	R_0 contains t R_1 contains u	t in R_0 u in R_1
v: = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d: = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

Module 11: Code Generation

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements **a: = b[i] and a[i]: = b**

Statements	Code Generated	Cost
a: = b[i]	MOV b(R_i), R	2
a[i]: = b	MOV b, a(R_i)	3

Generating Code for Pointer Assignments: a: = *p and *p: = a

Statements	Code Generated	Cost
a: = *p	MOV *R_p, a	2
*p: = a	MOV a, *R_p	2

Generating Code for Conditional Statements

Statement	Code
if x < y goto z	CMP x, y CJ < z /* jump to z if condition code is negative */

x: = y +z
if x < 0 goto z
	MOV y, R_0
	ADD z, R_0
	MOV R_0, x
	CJ < z

11.4 The DAG Representation for Basic Blocks:

A DAG for a basic block is a **Directed Acyclic Graph** with the following labels on nodes:

1. Leaves are labelled by unique identifiers, either variable names or constants.
2. Interior nodes are labelled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

DAGs are useful data structures for implementing transformations on basic blocks. It gives a picture of how the value computed by a statement is used in subsequent statements. It provides a good way of determining common sub-expressions.

Module 11: Code Generation

Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i) x: = y OP z
Case (ii) x: = OP y
Case (iii) x: = y

Method:

Step 1: If y is undefined then create node(y).
If z is undefined, create node(z) for case(i).

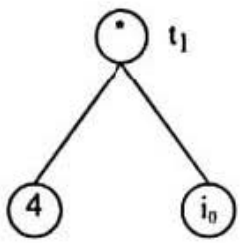
Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is node(z). (Checking for common sub expression). Let n be this node.
For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.
For case(iii), node n will be node(y).

Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached identifiers for the node n found in step 2 and set node(x) to n.

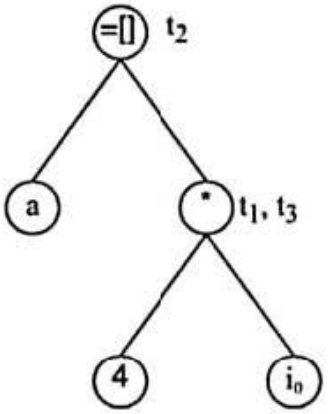
Example: Consider the block of three- address statements:

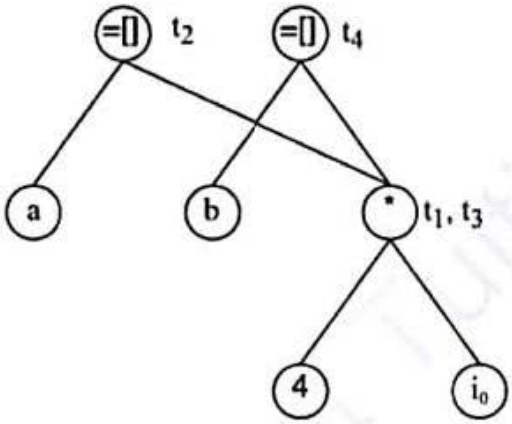
1) $t_1 := 4 * i$
2) $t_2 := a [t_1]$
3) $t_3 := 4 * i$
4) $t_4 := b [t_3]$
5) $t_5 := t_2 * t_4$
6) $t_6 := p + t_5$
7) $p := t_6$
8) $t_7 := i + 1$
9) $i := t_7$
10) if i <= 20 goto (1)

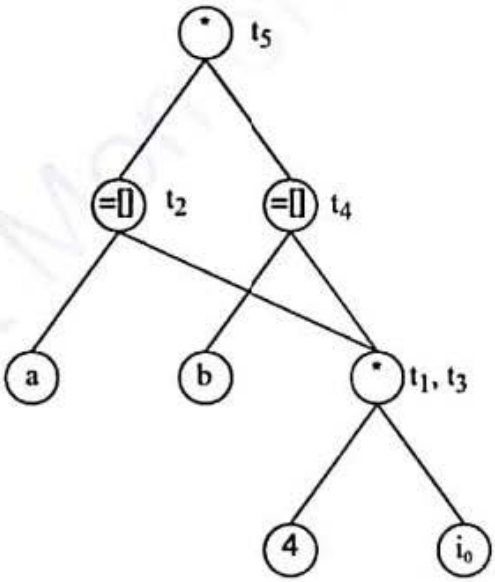
Stages in DAG Construction



Module 11: Code Generation

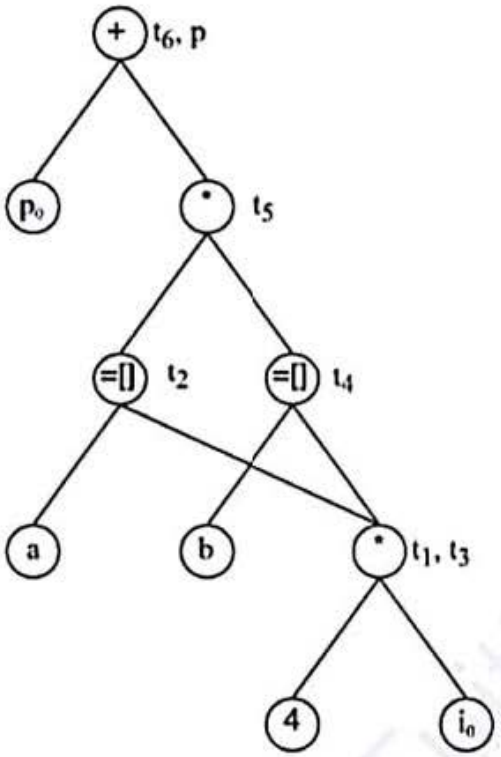


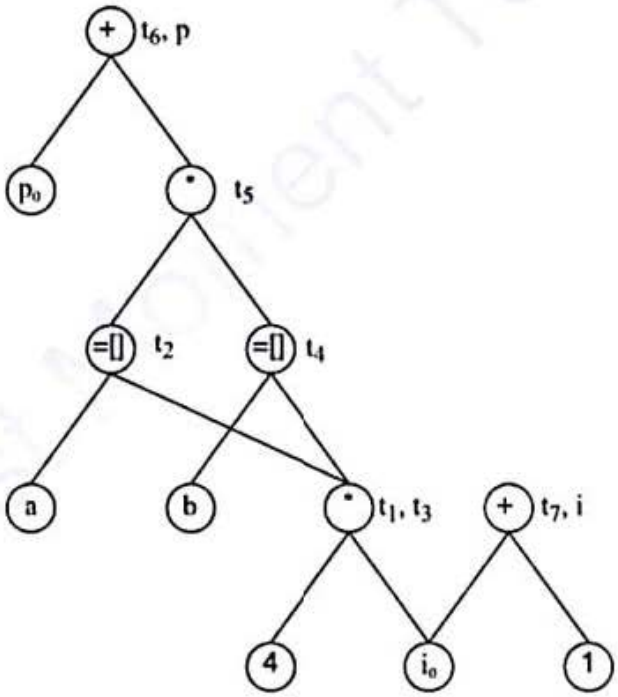




Last Moment Tuitions

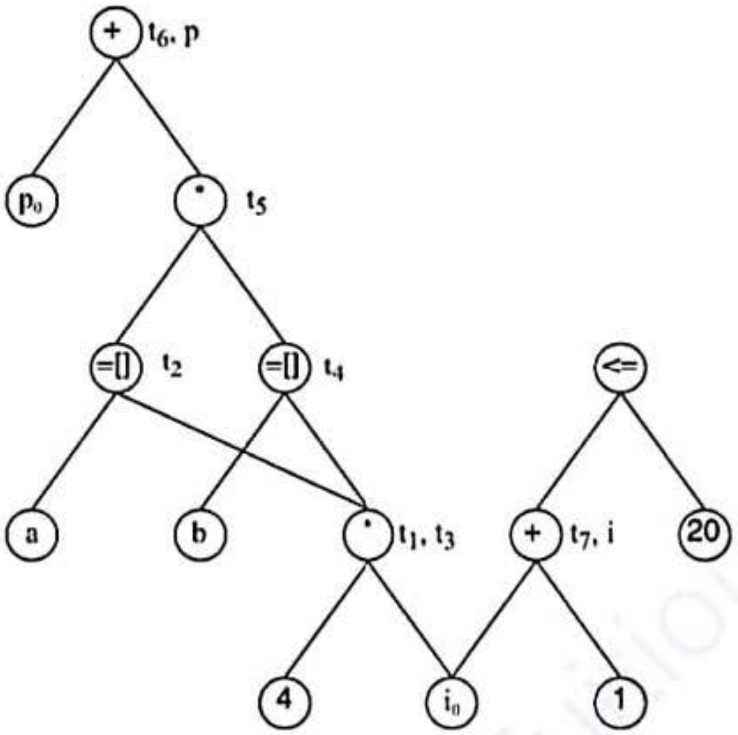
Module 11: Code Generation





Last Moment Tuitions

Module 11: Code Generation



Application of DAGs:

1. It can automatically detect common sub expressions.
2. It can determine which identifiers have their values used in the block.
3. It can determine which statements compute values that could be used outside the block.

Generating Code from DAGs

The advantage of generating code for a basic block from its DAG representation is that, from a DAG, we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code. For example, consider the following basic block:

$t_1 := a + b$
$t_2 := c + d$
$t_3 := e - t_2$
$t_4 := t_1 - t_3$

Generated code sequence for basic block:

```
MOV  a, R0
ADD  b, R0
MOV  R0, t1
MOV  c, R0
ADD  d, R0
MOV  e, R1
SUB  R0, R1
MOV  t1, R0
SUB  R1, R0
MOV  R0, t4
```

Module 11: Code Generation

Rearranged basic block:

Now t_1 occurs immediately before t_4.

$t_2 := c + d$
$t_3 := e - t_2$
$t_1 := a + b$
$t_4 := t_1 - t_3$

Revised code sequence:

```
MOV  c, R0
ADD  d, R0
MOV  e, R1
SUB  R0, R1
MOV  a, R0
ADD  b, R0
SUB  R1, R0
MOV  R0, t4
```

In this order, two instructions **MOV R_0, t_1** and **MOV t_1, R_0** have been saved.

A Heuristic ordering for DAGs:

The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument. The algorithm shown below produces the ordering in reverse.

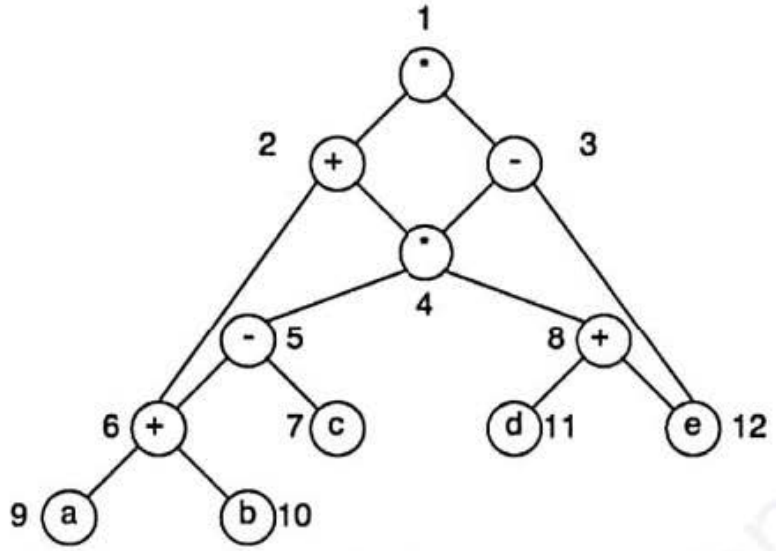
Algorithm:

```
while unlisted interior nodes remain do
    begin
        select an unlisted node n, all of whose parents have been listed;
        list n;
        while the leftmost child m of n has no unlisted parents and is not a leaf do
            begin
                list m;
                n: = m
            end
    end
```

Last Moment Tuitions

Module 11: Code Generation

Example: Consider the DAG shown below:



Initially, the only node with no unlisted parents is 1 so set n=1 and list 1. Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2.

Now, we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus, we select a new node 3, which is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that.

The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

$t_8 := d + e$
$t_6 := a + b$
$t_5 := t_6 - c$
$t_4 := t_5 * t_8$
$t_3 := t_4 - e$
$t_2 := t_6 + t_4$
$t_1 := t_2 * t_3$

This will yield an optimal code for the DAG on machine whatever be the number of registers.

Last Moment

Module 12: Code Optimization

Contents:
Principal sources of Optimization, Optimization of Basic Blocks, Loops in Flow graph, Peephole Optimization.

Introduction:

High-level language constructs can introduce substantial run-time overhead if we naively translate each construct independently into machine code. Elimination of unnecessary instructions in object code, or the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing is called "code improvement" or "code optimization". Compilers that apply code-improving transformations are called optimizing compilers.

Local code optimization is a code improvement within a basic block. **Global code optimization** deals with improvements take into account what happens across basic blocks. Most global optimizations are based on data flow analyses, which are algorithms to gather information about a program.

Optimizations are classified into two categories:

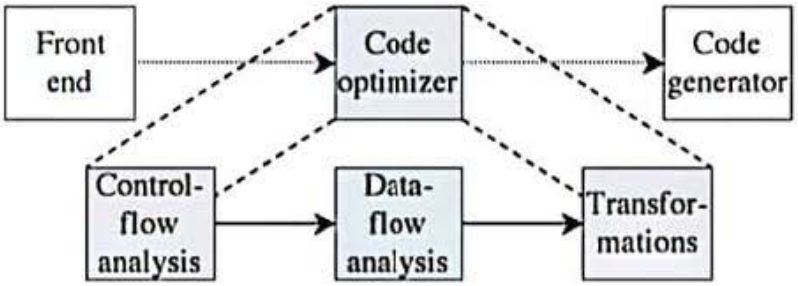
1. **Machine independent optimizations:** Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.
2. **Machine dependant optimizations:** Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations:

- The best program transformations are those that yield the most benefit for the least effort.
- The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input or cause an error such as division by zero, that was not present in the original source program. At all times we take the "*safe*" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.
- A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an "*optimization*" may slow down a program slightly.
- The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed.

Module 12: Code Optimization

Organization for an Optimizing Compiler:



Flow analysis is a fundamental prerequisite for many important types of code improvement. Generally, control flow analysis precedes data flow analysis. Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as

- Control flow graph
- Call graph

Data flow analysis (DFA) is the process of asserting and collecting information prior to program execution about the possible modification, preservation and use of certain entities (such as values or attributes of variables) in a computer program.

12.1 Principal Sources of Optimization:

Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

A compiler optimization must preserve the semantics of the original program. Except in very special circumstances, once a programmer chooses and implements a particular algorithm, the compiler cannot understand enough about the program to replace it with a substantially different and more efficient algorithm.

A compiler knows only how to apply relatively low-level semantic transformations, using general facts such as algebraic identities like $i + 0 = i$ or program semantics such as the fact that performing the same operation on the same values yields the same result.

Causes of Redundancy:

There are many redundant operations in a typical program. Sometimes the redundancy is available at the source level. For instance, a programmer may find it more direct and convenient to recalculate some result, leaving it to the compiler to recognize that only one such calculation is necessary. But more often, the redundancy is a side effect of having written the program in a high-level language. In most languages (other than C or C++, where pointer arithmetic is allowed), programmers have no choice but to refer to elements of an array or fields in a structure through accesses like A[i] [j] or $X \rightarrow f$.

As a program is compiled, each of these high-level data-structure accesses expands into a number of low-level arithmetic operations, such as the computation of the location of the (i, j) element of a matrix A. Accesses to the same data structure often share many common low-level operations. Programmers are not aware of these low-level operations and cannot eliminate the redundancies themselves. It is, in fact, preferable from a software-engineering perspective that programmers only access data elements by their high-level names; the programs are easier to write and, more importantly, easier to understand and evolve. By having a compiler eliminate the redundancies, we get the best of both worlds: the programs are both efficient and easy to maintain.

Module 12: Code Optimization

Semantics-Preserving (Function-Preserving) Transformations:

There are a number of ways in which a compiler can improve a program without changing the function it computes. The common examples of such function-preserving transformations are:

- Global Common Subexpression Elimination
- Global Copy propagation
- Dead-code elimination
- Global Constant folding

The above transformations come up primarily when global optimizations are performed. Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

Global Common Subexpressions Elimination:

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

For example,

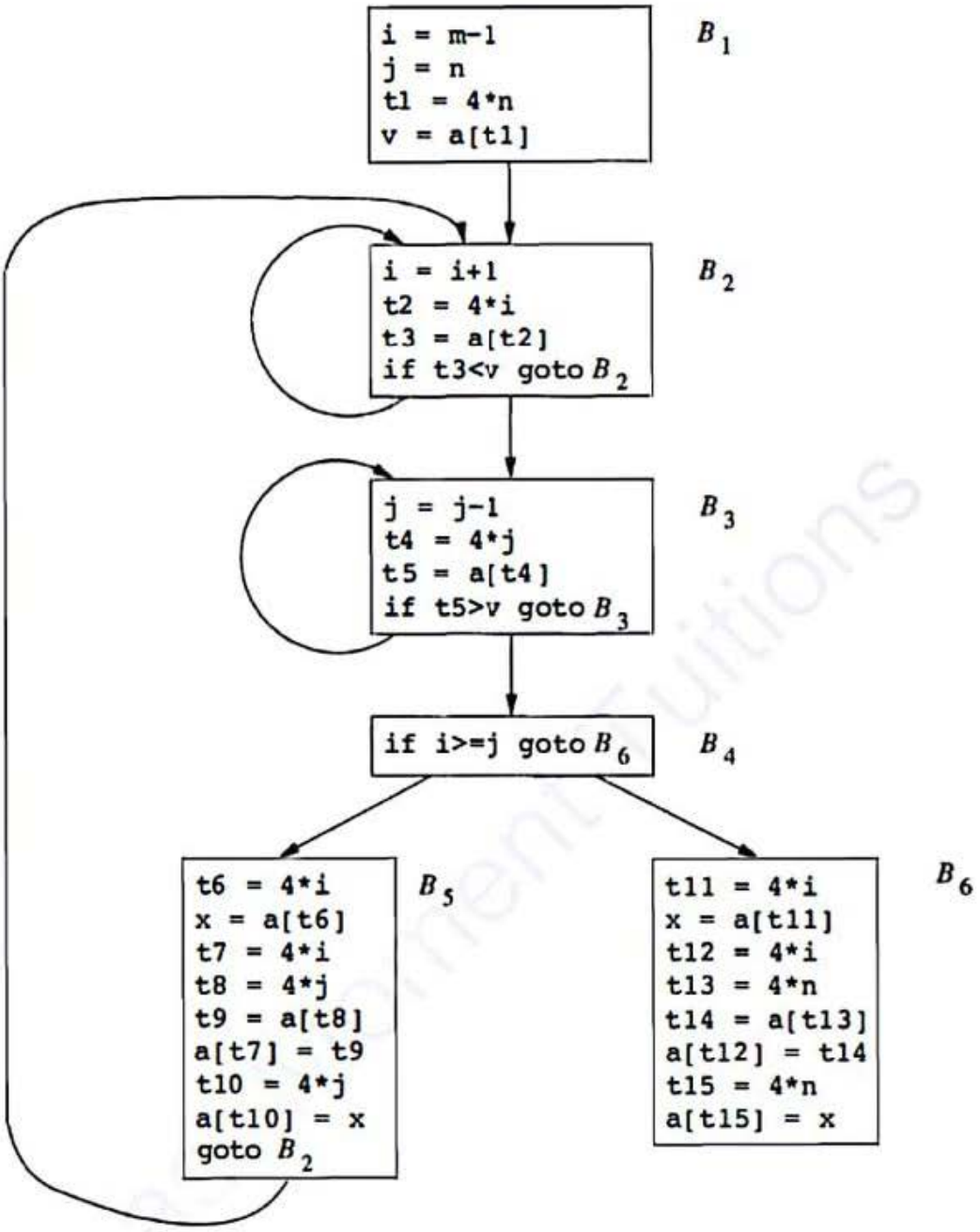
t_1: = 4*i
t_2: = a [t_1]
t_3: = 4*j
t_4: = 4*i
t_5: = n
t_6: = b [t_4] +t_5

The above code can be optimized using the common sub-expression elimination as

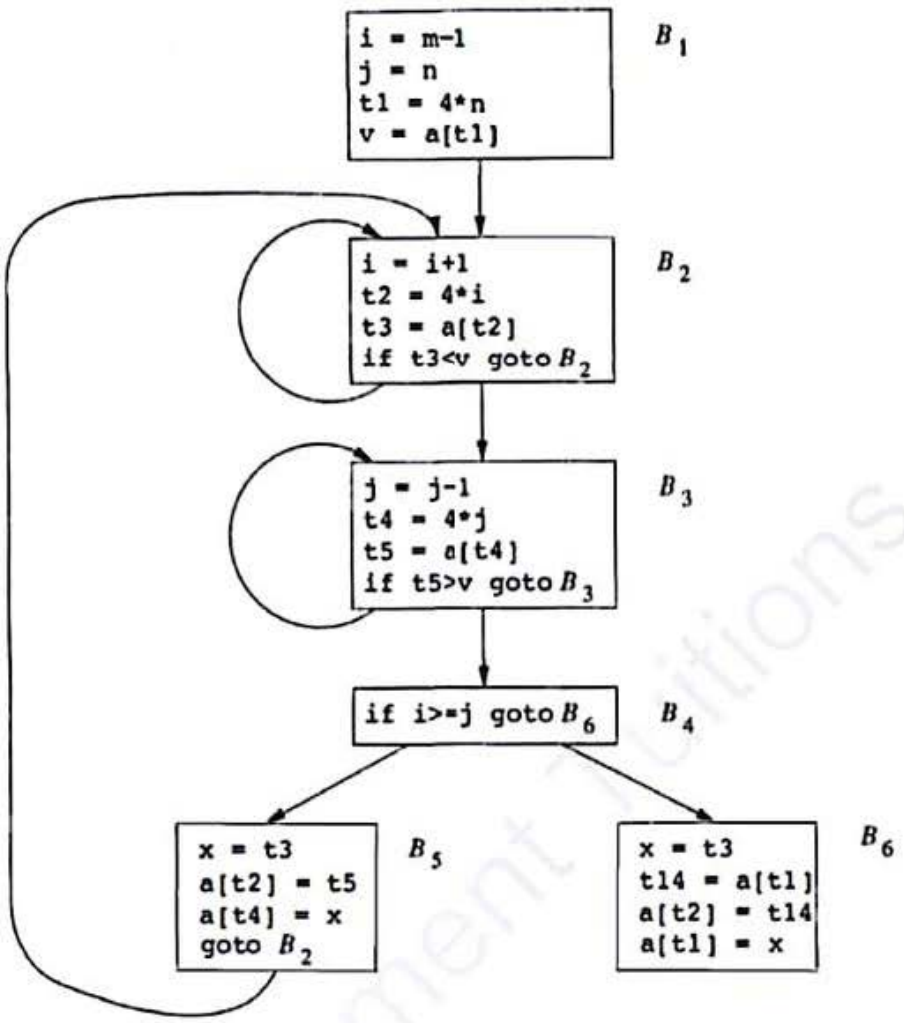
t_1: = 4*i
t_2: = a [t_1]
t_3: = 4*j
t_5: = n
t_6: = b [t_1] +t_5

The common sub expression t_4: =4*i is eliminated as its computation is already in t_1. And value of i is not been changed from definition to use.

Last Moment Tuitions



Flow graph for the quicksort fragment



Flow graph after removing common subexpression in B_5 and B_6

Global Copy Propagation:

Assignments of the form $f := g$ called copy statements or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement $f := g$.

Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x. One advantage of copy propagation is that it often turns the copy statement into dead code.

For example:

x = Pi;
A = x * r * r;

The optimization using copy propagation can be done as follows:
A = Pi * r * r;

Here the variable x is eliminated.

Module 12: Code Optimization

Dead-Code Eliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.

Example:

```
i = 0;
if(i==1) {
        a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

Constant folding:

More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

For example,

a = 3.14/2	can be replaced by
a = 1.57	thereby eliminating a division operation.

Loop Optimizations:

One of the very important places for optimizations are loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if there is a decrease in the number of instructions in an inner loop, even if there is increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- **Code Motion**, which moves code outside a loop;
- **Induction Variable Elimination**, which we apply to replace variables from inner loop.
- **Reduction in Strength**, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

Code Motion:

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (*a loop-invariant computation*) and places the expression before the loop.

For example, evaluation of *limit* - 2 is a loop-invariant computation in the following while-statement:

```
while (i < (limit – 2))        /* statement does not change limit*/
```

Code motion will result in the equivalent of

```
t = limit - 2;
while (i < t)                  /* statement does not change limit or t */
```

Module 12: Code Optimization

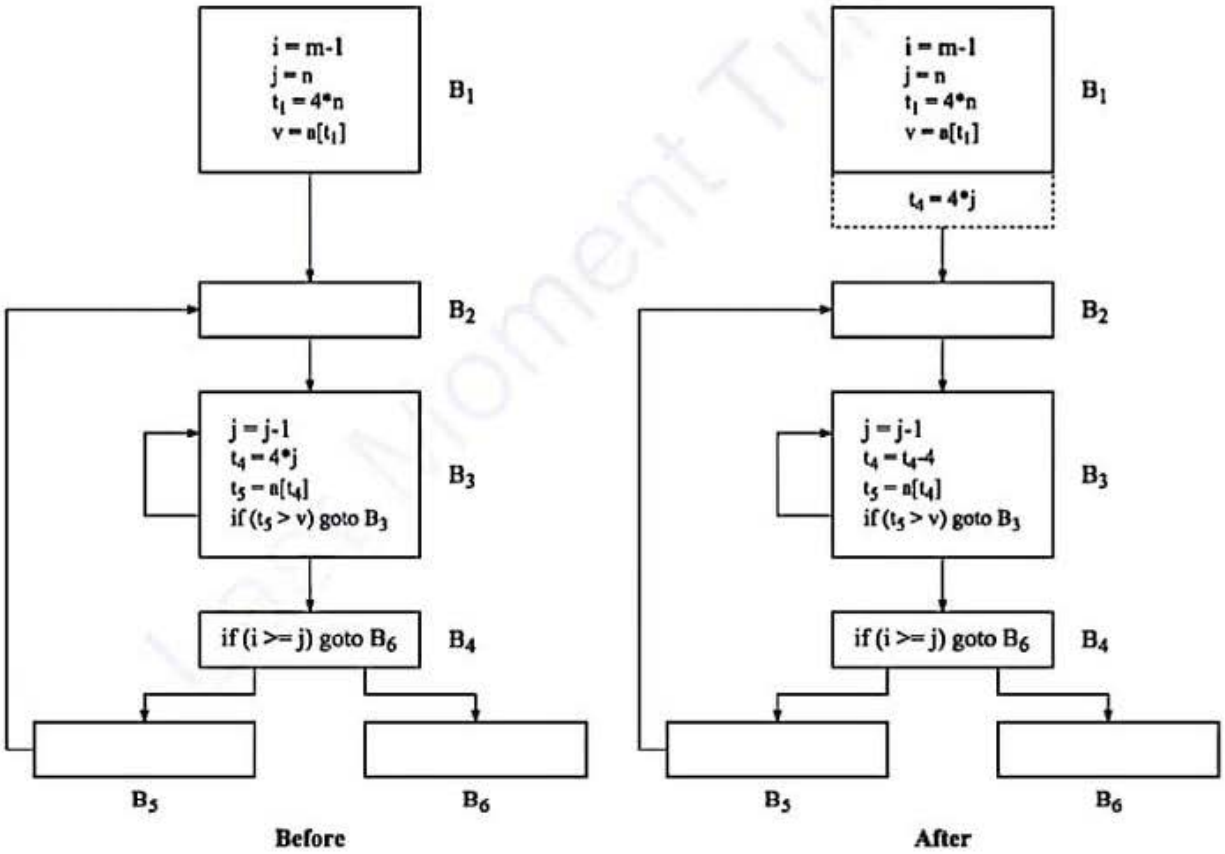
Induction Variables:

Loops are usually processed inside out. For example, consider the loop around B_3. Note that the values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because 4*j is assigned to t_4. Such identifiers are called induction variables.

When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B_3 in figure below, we cannot get rid of either j or t_4 completely; t_4 is used in B_3 and j in B_4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B_2 - B_5 is considered.

Example:

As the relationship t_4: = 4*j surely holds after such an assignment to t_4 in figure and t_4 is not changed elsewhere in the inner loop around B_3, it follows that just after the statement j: = j-1 the relationship t_4: = 4*j-4 must hold. We may therefore replace the assignment t_4: = 4*j by t_4: = t_4-4. The only problem is that t_4 does not have a value when we enter block B_3 for the first time. Since we must maintain the relationship t_4: = 4*j on entry to the block B_3, we place an initialization of t_4 at the end of the block where j itself is initialized, shown by the dashed addition to block B_1 in second figure.



The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

Module 12: Code Optimization

Reduction in Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

12.2 Optimization of Basic Blocks:

A substantial improvement in the running time of code can be obtained merely by performing local optimization within each basic block by itself. There are two types of basic block optimizations. They are:

1. **Structure-Preserving Transformations**
2. **Algebraic Transformations**

1. Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- **Common sub-expression elimination**
- **Dead code elimination**
- **Renaming of temporary variables**
- **Interchange of two independent adjacent statements.**

Common sub-expression elimination:

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again of course providing the variable values in the expression still remain constant.

Example:

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Basic block can be transformed to

```
a := b + c
b := a - d
c := b + c
d := b
```

Dead code elimination:

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

```
x := y + z
```

If x never used again in block, we can eliminate this statement.

Module 12: Code Optimization

Renaming of temporary variables:

t := b + c	u := b + c
a := t + a	a := u + a
t := d + e	v := d + e
f := t	f := v

A statement t: = b + c where t is a temporary name can be changed to u: = b + c where u is another temporary name and change all uses of t to u. In this we can transform a basic block to its equivalent block called normal-form block.

It makes some optimizations at Basic Block level a lot simpler (e.g. common subexpression elimination, copy propagation, etc.)

Interchange of two independent adjacent statements:

t_1 := b + c	t_2 := x + y
t_2 := x + y	t_1 := b + c

Two statements can be interchanged or reordered in its computation in the basic block when value of t_1 does not affect the value of t_2.

2. **Algebraic Transformations:**

Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.

1. Addition of zero:

   x := x + 0 can be removed

2. Multiplication by one:

   x := x * 1 can be removed

3. Integer exponent

   $y := x^3$

   can be replaced by

   t_1 := x * x

   y := t_1 * x

The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics.

Last Moment Tuitions

Module 12: Code Optimization

12.3 Loops in Flow Graph:

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations and the edges represent the flow of control.

Programming-language constructs like while-statements, do-while-statements and for-statements naturally give rise to loops in programs. Since virtually every program spends most of its time in executing its loops, it is especially important for a compiler to generate good code for loops. Many code transformations depend upon the identification of "loops" in a flow graph. We say that a set of nodes L in a flow graph is a loop if

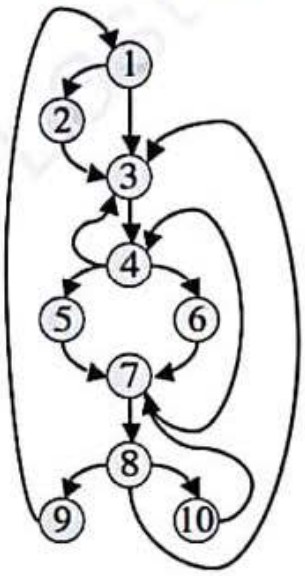
1. There is a node in L called the loop entry with the property that no other node in L has a predecessor outside L. That is, every path from the entry of the entire flow graph to any node in L goes through the loop entry.
2. Every node in L has a nonempty path, completely within L, to the entry of L.

Dominators:

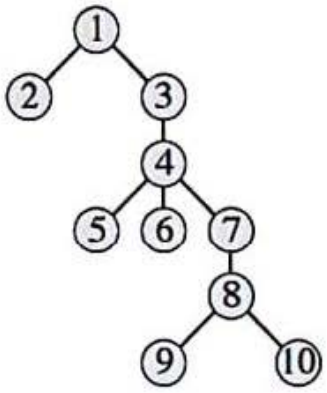
In a flow graph, a node *d* dominates node *n*, if every path from initial node of the flow graph to *n* goes through *d*. This will be denoted by *d* dom *n*. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly, every node dominates itself.

Example:
In the flow graph below,

- Initial node, node1 dominates every node.
- node 2 dominates itself
- node 3 dominates all but 1 and 2.
- node 4 dominates all but 1, 2 and 3.
- node 5 and 6 dominates only themselves, since flow of control can skip around either by going through the other.
- node 7 dominates 7, 8, 9 and 10.
- node 8 dominates 8, 9 and 10.
- node 9 and 10 dominates only themselves.



CFG



Dominator tree

Module 12: Code Optimization

The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root. The parent of each other node is its immediate dominator. Each node d dominates only its descendants in the tree. The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n. In terms of the dom relation, the immediate dominator m has the property is $d = !n$ and d dom n, then d dom m.

Natural Loop:

One application of dominator information is in determining the loops of a flow graph suitable for improvement.

The properties of loops are

- A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
- There must be at least one way to iterate the loop i.e. at least one path back to the header.

One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If $a \rightarrow b$ is an edge, b is the head and a is the tail. These types of edges are called as back edges.

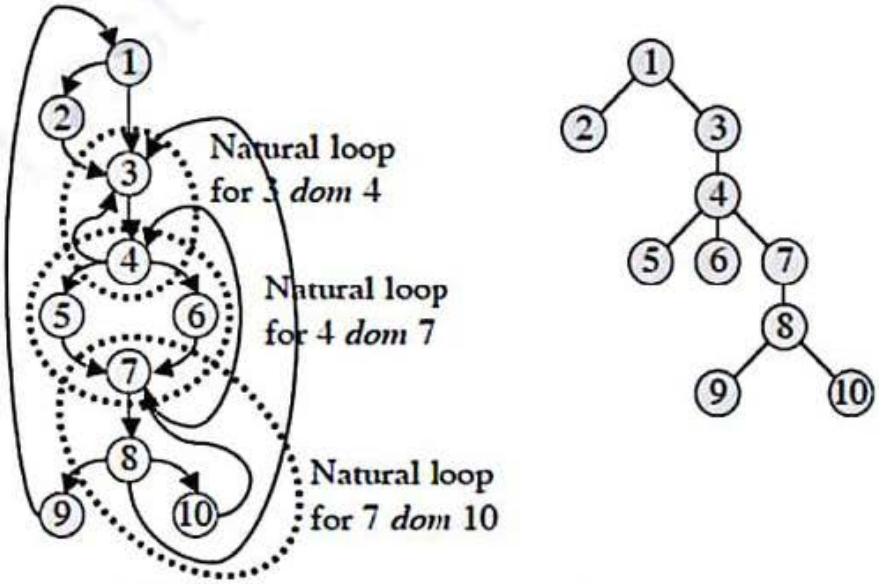
Example:

In the above graph,

1. Backedge $10 \rightarrow 7$: {7, 8, 10} (The inner loop)
2. Backedge $7 \rightarrow 4$: {4, 5, 6, 7, 8, 10}
3. Backedge $4 \rightarrow 3$: {3, 4}
4. Backedge $8 \rightarrow 3$: {3, 4, 5, 6, 7, 8, 10}
5. Backedge $9 \rightarrow 1$: The entire flow graph

The above edges will form loop in flow graph. Given a back edge $n \rightarrow d$, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.

Inner loop:



CFG Dominator tree

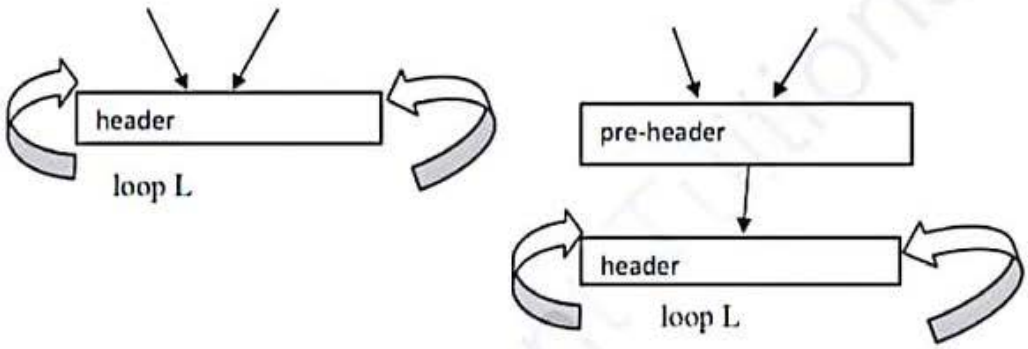
Module 12: Code Optimization

If we use the natural loops as "the loops", then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop. When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

Pre-Headers:

Several transformations require us to move statements "before the header". Therefore, begin treatment of a loop L by creating a new block, called the preheader. The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header.

Edges from inside loop L to the header are not changed. Initially the pre-header is empty, but transformations on L may place statements in it.



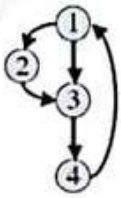
(a) Before **(b) After**

Reducible flow graphs:

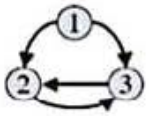
Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.

Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue and break statements produces programs whose flow graphs are always reducible.

The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.



Example of a reducible CFG



Example of a nonreducible CFG

Definition:

A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, forward edges and back edges, with the following properties.

- The forward edges from an acyclic graph in which every node can be reached from initial node of G.
- The back edges consist only of edges where heads dominate theirs tails.

Example:
The above flow graph is reducible.

- If we know the relation *dom* for a flow graph, we can find and remove all the backedges.
- The remaining edges are forward edges.
- If the forward edges form an acyclic graph, then we can say the flow graph reducible.
- In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic.
- The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a back edge.

12.4 Peephole Optimization:

A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying "optimizing" transformations to the target program.

A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

We shall give the following examples of program transformations that are characteristic of peephole optimizations:

- **Redundant-instructions elimination**
- **Flow-of-control optimizations**
- **Algebraic simplifications**
- **Use of machine idioms**
- **Unreachable Code**
- **Reduction in Strength**

Redundant Loads and Stores:

If we see the instructions sequence

1) LD a, R_0
2) ST R_0, a

in a target program, we can delete the store instruction because whenever it is executed, the first instruction will ensure that the value of a has already been loaded into register R_0. Note that if the store instruction had a label, we could not be sure that the first instruction is always executed before the second, so we could not remove the store instruction. Put another way, the two instructions have to be in the same basic block for this transformation to be safe. Redundant loads and stores of this nature would not be generated by the simple code generation algorithm of the previous section.

Module 12: Code Optimization

Flows-Of-Control Optimizations:

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations.

We can replace the jump sequence
goto L_1
....
L_1: goto L_2

by the sequence
goto L_2
....
L_1: goto L_2

If there are now no jumps to L_1, then it may be possible to eliminate the statement L_1: goto L_2 provided it is preceded by an unconditional jump. Similarly, the sequence
if a < b goto L_1
....
L_1: goto L_2

can be replaced by
If a < b goto L_2
....
L_1: goto L_2

Finally, suppose there is only one jump to L_1 and L_1 is preceded by an unconditional goto. Then the sequence
goto L_1
...
L_1: if a < b goto L_2
L_3: ..... (1)

May be replaced by
If a < b goto L_2
goto L_3
...
L_3: ..... (2)

While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1). Thus (2) is superior to (1) in execution time.

Last Moment Tuitions

Module 12: Code Optimization

Algebraic Simplification:

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

x := x+0 **OR** x := x*1

Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Use of Machine Idioms:

The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.

The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like i := i+1.

i := i+1 → i++
i := i-1 → i- -

Unreachable Code:

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabelled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1. In C, the source code might look like:

```
#define debug 0
...
if (debug) {
        print debugging information
}
```

In the intermediate representations the if-statement may be translated as:

```
if (debug ==1) goto L2
        goto L2
L1: print debugging information
L2:                                    ..............................(a)
```

One obvious peephole optimization is to eliminate jumps over jumps. Thus no matter what the value of debug; (a) can be replaced by:

```
if (debug ≠1) goto L2
        print debugging information
L2:                                    ..............................(b)
```

As the argument of the statement of (b) evaluates to a constant true it can be replaced by:

```
if (debug ≠0) goto L2
        print debugging information
L2:                                    ..............................(c)
```

As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L_2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Reduction in Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$y := x^2$	→	y := x * x
b := y/8	→	b := y>>3
x = x *2	→	x := x << 1

Last Moment Tuitions

Module 13: Run Time Storage

Content:
Storage Organization, Storage allocation strategies, Parameter passing, Symbol table, Introduction to garbage collection and compaction.

Introduction:

A compiler must accurately implement the abstractions embodied in the source language definition. These abstractions typically include the concepts such as names, scopes, bindings, data types, operators, procedures, parameters and flow-of-control constructs. The compiler must cooperate with the operating system and other systems software to support these abstractions on the target machine.

To do so, the compiler creates and manages a Run-time environment in which it assumes its target programs are being executed. This environment deals with a variety of issues such as the layout and allocation of storage locations for the objects named in the source program, the mechanisms used by the target program to access variables, the linkages between procedures, the mechanisms for passing parameters and the interfaces to the operating system, input/output devices and other programs.

Source Languages Issues:

Procedures:

A *procedure definition* is a declaration that associates an identifier with a set of statements. The identifier is the *procedure name*, and the statement is the *procedure body*. For example, the following is the definition of procedure named *readArray*:

```
procedure readArray
        var i : integer;
        begin
                for i := 1 to 9 do
                        read(a[i])
        end;
```

When a procedure name appears within an executable statement, the procedure is said to be *called* at that point.

The Scope of a Declaration:

A declaration is a syntactic construct that associates information with a name. Declarations may be explicit, such as:

var i : integer ;

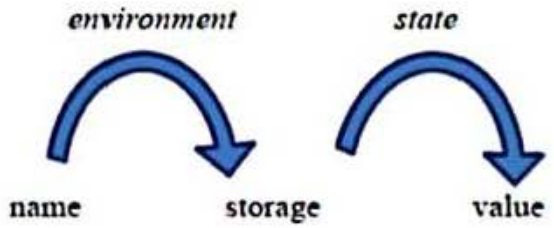
or they may be implicit. Example, any variable name starting with I is assumed to denote an integer. The portion of the program to which a declaration applies is called the *scope* of that declaration.

Binding of names:

Binding refers to association of a program entity to its value. This means a variable name is bound to the location where its value is stored and a procedure name is bound to the location where its activation record is stored.

Even if each name is declared once in a program, the same name may denote different data objects at run time. "Data object" corresponds to a storage location that holds values. The term *environment* refers to a function that maps a name to a storage location. The term *state* refers to a function that maps a storage location to the value held there.

Module 13: Run Time Storage



When an *environment* associates storage location *s* with a name *x*, we say that *x* is *bound* to *s*. This association is referred to as a *binding* of *x*.

Binding can be achieved in two ways:

- Static Binding
- Dynamic Binding

Static Binding:

In Static Binding, the storage once allocated remains static. Binding process takes place before the execution of a program. Allocated space is not freed, even if its defining program unit is not active. An example of programming language that uses the static binding model is FORTRAN.

Dynamic Binding:

In Dynamic Binding, the storage once allocated can be changed. Binding process takes place during program execution. Allocated space can be de-allocated dynamically. An example of programming language that uses dynamic binding is PASCAL.

There are two approaches to dynamic binding:

- **Automatic Dynamic Binding:** When a program unit starts execution, the memory is allocated to its variable and procedures. When the program unit ends, the memory is de-allocated. This allows the same memory area to be used for other allocations as well. Automatic Dynamic Binding is implemented through Stack Allocation.
- **Program-controlled Dynamic Binding:** The program explicitly controls allocation/de-allocation of memory space, which may cause some processing overhead. It is implemented using Heap Allocation method.

Activation trees:

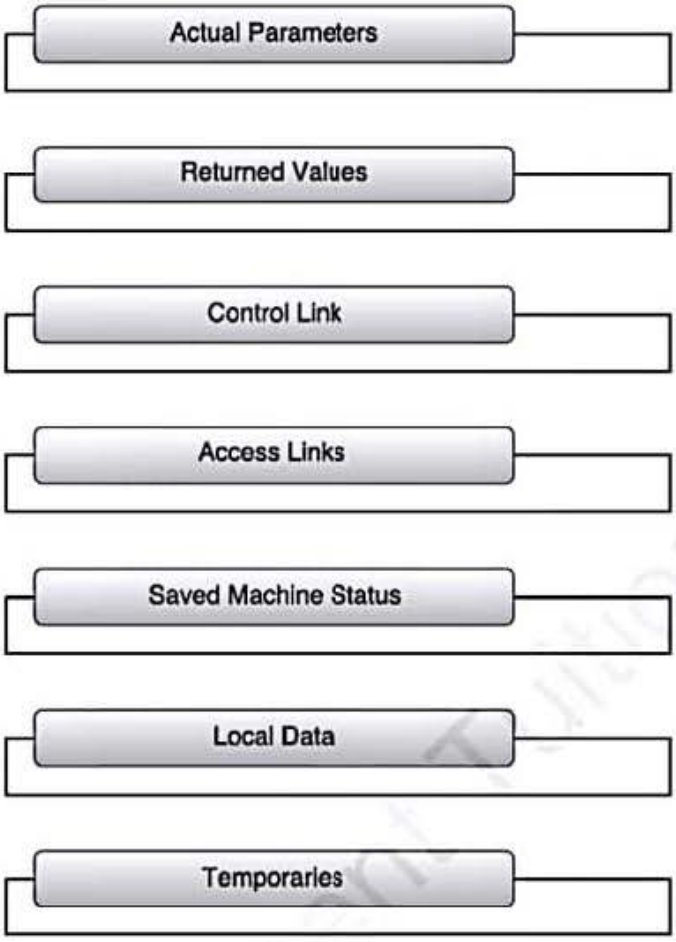
An *activation tree* is used to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for *a* is the parent of the node for *b* if and only if control flows from activation *a* to *b*.
4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before the lifetime of *b*.

Activation records:

- Procedure calls and returns are usually managed by a run time stack called the *control stack*.
- Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.

Module 13: Run Time Storage



- Temporary values such as those arising from the evaluation of expressions.
- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.
- Space for the return value of the called functions, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

Control stack:

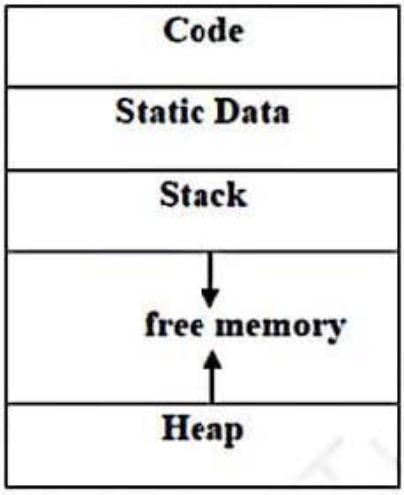
- A *control stack* is used to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.
- The contents of the control stack are related to paths to the root of the activation tree. When node *n* is at the top of control stack, the stack contains the nodes along the path from *n* to the root.

Module 13: Run Time Storage

13.1 Storage Organisation:

The executing target program runs in its own logical address space in which each program value has a location. The management and organization of this logical address space is shared between the complier, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

Typical subdivision of run-time memory:



- Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multi-byte objects are stored in consecutive bytes and given the address of first byte.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

13.2 Storage Allocation Strategies

Static vs Dynamic Allocation:

The layout and allocation of data to memory locations in the run-time environment are key issues in storage management. The two adjectives static and dynamic distinguish between compile time and run time, respectively. A storage-allocation decision is said to be static, if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes. Conversely, a decision is dynamic if it can be decided only while the program is running. Many compilers use some combination of the following two strategies for dynamic storage allocation:

1. Stack storage: Names local to a procedure are allocated space on a stack. The stack supports the normal call/return policy for procedures.
2. Heap storage: Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage.

Module 13: Run Time Storage

The different storage allocation strategies are:

1. **Static Allocation:** lays out storage for all data objects at compile time
2. **Stack Allocation:** manages the run-time storage as a stack.
3. **Heap Allocation:** allocates and deallocates storage as needed at run time from a data area known as heap.

Static Allocation:

- In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.
- Since the bindings do not change at run-time, every time a procedure is activated, its names are bound to the same storage locations.
- Therefore, values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, the addresses can be filled at locations which the target code can find the data it operates on.

Stack Allocation of Space:

All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack. Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

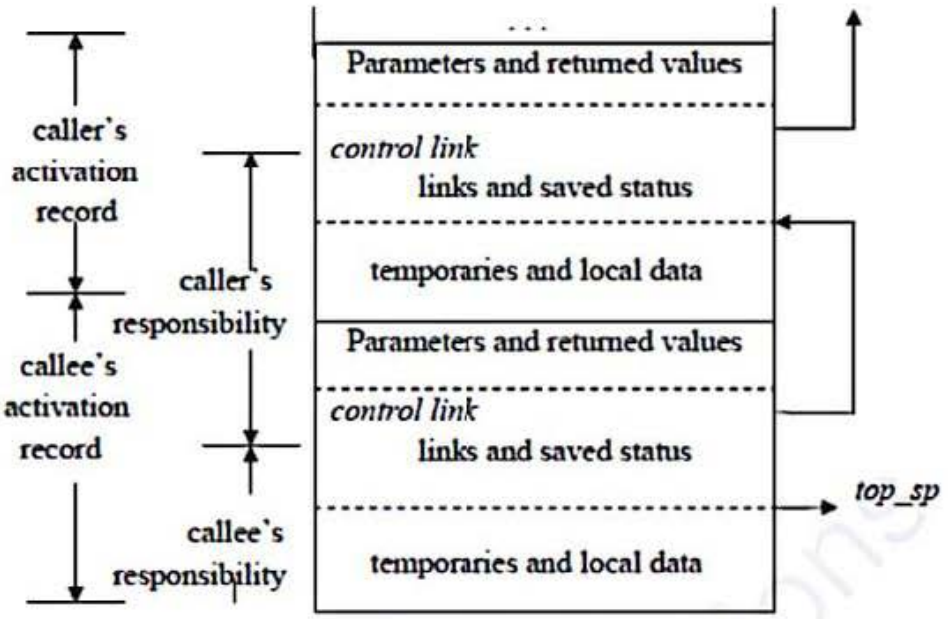
Calling Sequences:

Procedures called are implemented in what is called as *calling sequence*, which consists of code that allocates an activation record on the stack and enters information into its fields. A *return sequence* is similar to code to restore the state of machine so the calling procedure can continue its execution after the call. The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).

When designing calling sequences and the layout of activation records, the following principles are helpful:

- Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.
- Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status fields.
- Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the value of one of the callee's parameters determines the length of the array.
- We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.

Module 13: Run Time Storage



Division of tasks between caller and callee

The calling sequence and its division between caller and callee are as follows:

- The caller evaluates the actual parameters.
- The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments the *top_sp* to the respective positions.
- The callee saves the register values and other status information.
- The callee initializes its local data and begins execution.

A suitable, corresponding return sequence is:

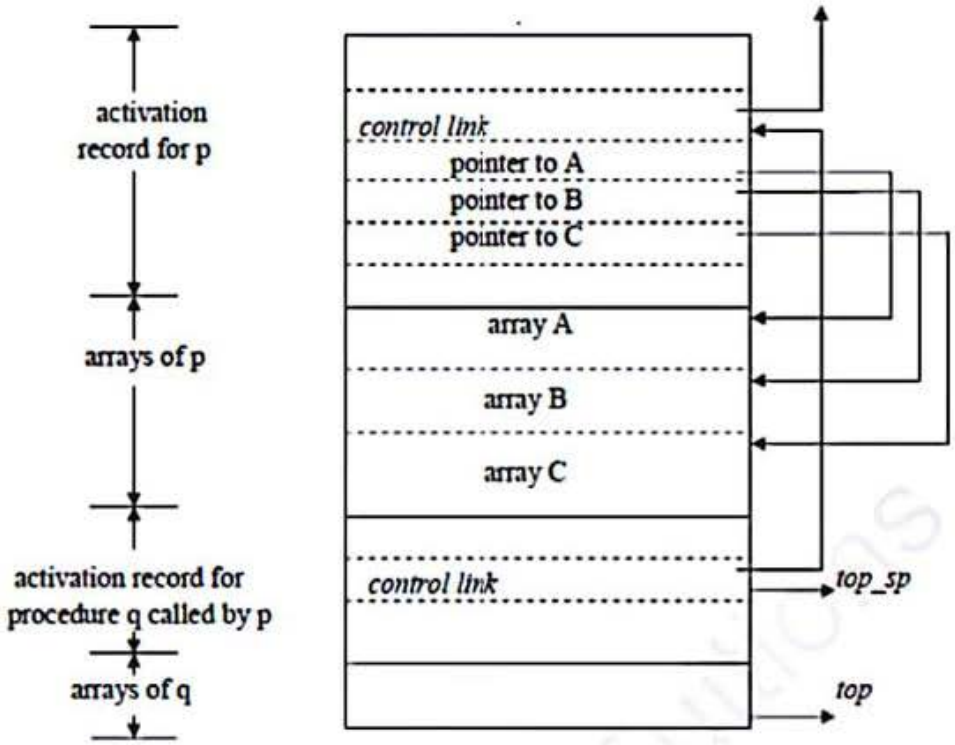
- The callee places the return value next to the parameters.
- Using the information in the machine-status field, the callee restores *top_sp* and other registers, and then branches to the return address that the caller placed in the status field.
- Although *top_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top_sp*; the caller therefore may use that value.

Variable length data on stack:

The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack. The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.

The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.

Module 13: Run Time Storage



Access to dynamically allocated arrays

- Procedure *p* has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for *p*.
- Access to the data is through two pointers, *top* and *top_sp*. Here the *top* marks the actual top of stack; it points the position at which the next activation record will begin.
- The second *top_sp* is used to find local, fixed-length fields of the top activation record.
- The code to reposition *top* and *top_sp* can be generated at compile time, in terms of sizes that will become known at run time.

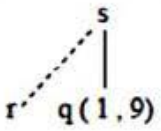
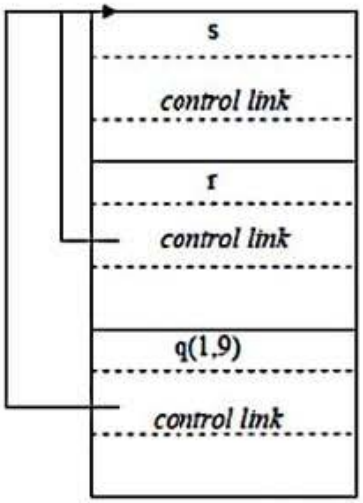
Heap Allocation:

Stack allocation strategy cannot be used if either of the following is possible:

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects. Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Module 13: Run Time Storage

Position in the activation tree	Activation records in the heap	Remarks
		Retained activation record for r

- The record for an activation of procedure *r* is retained when the activationends.
- Therefore, the record for the new activation *q*(1, 9) cannot follow that for *s* physically.
- If the retained activation record for *r* is deallocated, there will be free space in theheap between the activation records for *s* and *q*.

13.3 Parameter Passing:

All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments. The great majority of languages use either "Call-by-value" or "Call-by-reference" or both. There is also "Copy-restore" and "Call-by-name" used in few programming languages.

Call-by-Value:

In call-by-value, the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure. This method is used in C and Java and is a common option in C++, as well as in most other languages. Call-by-value has the effect that all computation involving the formal parameters done by the called procedure is local to that procedure and the actual parameters themselves cannot be changed.

Note, however, that in C we can pass a pointer to a variable to allow that variable to be changed by the callee. Likewise, array names passed as parameters in C, C++ or Java give the called procedure what is in effect a pointer or reference to the array itself. Thus, if *a* is the name of an array of the calling procedure and it is passed by value to corresponding formal parameter *x*, then an assignment such as $x[i] = 2$ really changes the array element $a[2]$. The reason is that, although *x* gets a copy of the value of *a*, that value is really a pointer to the beginning of the area of the store where the array named a is located.

Similarly, in Java, many variables are really references or pointers, to the things they stand for. This observation applies to arrays, strings and objects of all classes. Even though Java uses call-by-value exclusively, whenever we pass the name of an object to a called procedure, the value received by that procedure is in effect a pointer to the object. Thus, the called procedure is able to affect the value of the object itself.

Module 13: Run Time Storage

Call-by-Reference:

In call-by-reference, the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter. Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller. Changes to the formal parameter thus appear as changes to the actual parameter.

If the actual parameter is an expression, however, then the expression is evaluated before the call and its value stored in a location of its own. Changes to the formal parameter change this location, but can have no effect on the data of the caller.

Call-by-reference is used for "*ref*" parameters in C++ and is an option in many other languages. It is almost essential when the formal parameter is a large object, array or structure. The reason is that strict call-by-value requires that the caller copy the entire actual parameter into the space belonging to the corresponding formal parameter. This copying gets expensive when the parameter is large. As discussed, call-by-value languages such as Java solve the problem of passing arrays, strings or other objects by copying only a reference to those objects. The effect is that Java behaves as if it used call-by-reference for anything other than a basic type such as an integer or real.

Copy restore:

This method is a hybrid between call by value and call by reference. This method is also known as copy-in-copy-out or values result. The calling procedure calculates the value of actual parameter and it then copied to activation record for the called procedure. During execution of called procedure, the actual parameters value is not affected. If the actual parameter has L-value, then at return the value of formal parameter is copied to actual parameter.

Call-by-Name:

Another mechanism - call-by-name - was used in the early programming language Algol 60. It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct). When the actual parameter is an expression rather than a variable, some unintuitive behaviours occur, which is one reason this mechanism is not favoured today.

13.4 Symbol Table:

After syntax tree have been constructed, the compiler must check whether the input program is type-correct (called type checking and part of the semantic analysis). During type checking, a compiler checks whether the use of names (such as variables, functions, type names) is consistent with their definition in the program. Consequently, it is necessary to remember declarations so that we can detect inconsistencies and misuses during type checking. This is the task of a symbol table.

A symbol table is a compile-time data structure. It's not used during run time by statically typed languages. Formally, a symbol table maps names into declarations (called attributes), such as mapping the variable name *x* to its type *int*.

Symbol Tables are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phases of a compiler and used by the synthesis phases to generate the target code. Symbol tables typically need to support multiple declarations of the same identifier within a program.

Module 13: Run Time Storage

More specifically, a symbol table stores:

- For each type name, its type definition.
- For each variable name, its type. If the variable is an array, it also stores dimension information. It may also store storage class, offset in activation record etc.
- For each constant name, its type and value.
- For each function and procedure, its formal parameter list and its output type. Each formal parameter must have name, type, type of passing (by-reference or by-value), etc.

Operation on Symbol Table:

Following operations are implement on a symbol table:

1. **insert** (*s*, *t*)**:** return index of new entry for string '*s*' and token '*t*'
2. **lookup** (*s*)**:** return index of new entry for string '*s*' or 0 if '*s*' is not found.
3. **begin_scope( )** and **end_scope( ):** When we have a new block (i.e., when we encounter the token {), we begin a new scope. When we exit a block (i.e. when we encounter the token}) we remove the scope (this is the end_scope). When we remove a scope, we remove all declarations inside this scope. So basically, scopes behave like stacks. One way to implement these functions is to use a stack. When we begin a new scope we push a special marker to the stack (e.g., 1). When we insert a new declaration in the hash table using insert, we also push the bucket number to the stack. When we end a scope, we pop the stack until and including the first -1 marker.
4. **Handling Reserve Keywords:** Symbol table also handle reserve keywords like 'PLUS', MINUS', 'MUL' etc. This can be done in following manner.

   insert ("PLUS", PLUS);
   insert ("MINUS", MINUS);

   In this case first 'PLUS' and 'MINUS' indicate lexeme and other one indicate token.

Ordered and Unordered Symbol Table:

1. **Ordered Symbol Table:** In ordered symbol table the entries of variables are made in alphabetical manner. The searching of ordered symbol table can be done using linear and binary search.

Advantages:

- Searching of particular variable is efficient.
- Relationship of particular variable with another can be established very easily.

Disadvantages:

- Insertion of element in ordered list is costly if there are large number of entries.

2. **Unordered Symbol Table:** As variables are encountered, its entry is made in symbol table. These entries are unsorted. Each time, before inserting any variable in the symbol table, a lookup is made to check whether it is already present in the symbol table or not.

Advantages:

- Insertion of variable in symbol table is very efficient.

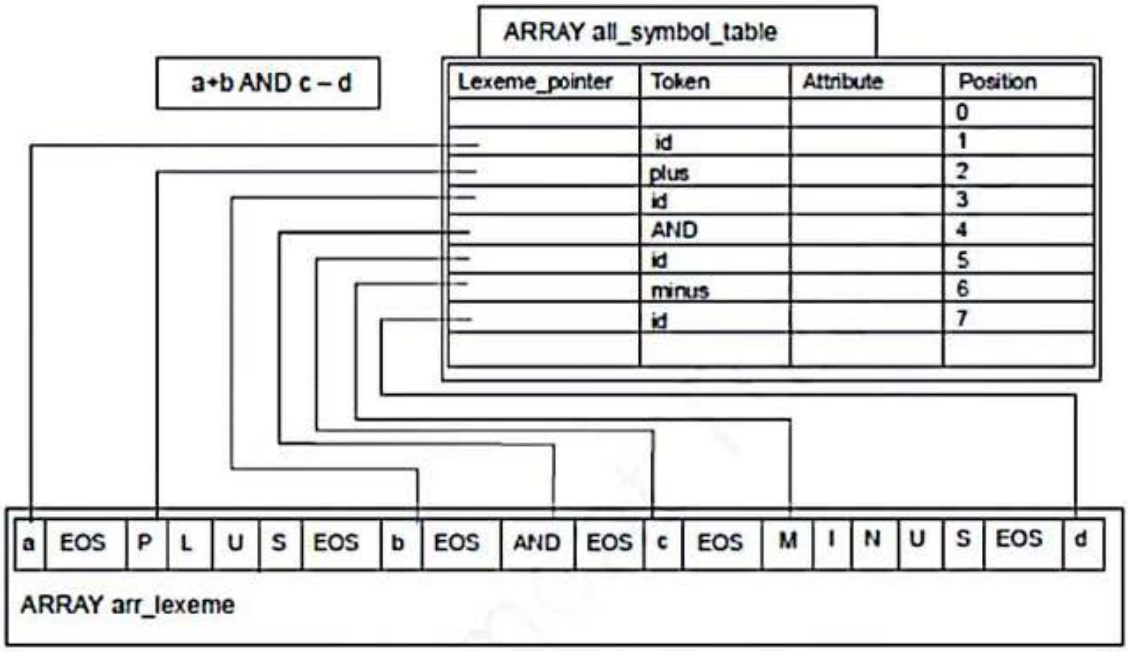
Disadvantages:

- Searching must be done using linear search.
- For large table size, this method turns to be inefficient, because before insertion a lookup is made.

Module 13: Run Time Storage

Symbol Table Implementation:

The data structure for a particular implementation of a symbol table is sketched in figure below. In the given figure, a separate array '*arr_lexemes*' holds the character string forming an identifier. The string is terminated by an end-of-string character, denoted by EOS, that may not appear in identifiers. Each entry in symbol-table array '*arr_symbol_table*' is a record consisting of two fields, as "*lexeme_pointer*", pointing to the beginning of a lexeme and token. Additional fields can hold attribute values. The 0th entry is left empty, because lookup return 0 to indicate that there is no entry for a string. The 1st, 2nd, 3rd, 4th, 5th, 6th and 7th entries are for the '*a*', '*plus*' '*b*' '*AND*', '*c*', '*minus*' and '*d*' where 2nd, 4th and 6th entries are for reserve keyword.



Symbol Table Implementation

When lexical analyser reads a letter, it starts saving letters, digits in a buffer '*lex_bufffer*'. The string collected in *lex_bufffer* is then looked in the symbol table, using the lookup operation. Since the symbol table initialized with entries for the keywords *plus*, *minus*, *AND* operator and some identifiers as shown in the above figure, the lookup operation will find these entries if *lex_buffer* contains either *div* or *mod*. If there is no entry for the string in *lex_buffer*, i.e., lookup return 0, then *lex_buffer* contains a lexeme for a new identifier. An entry for the identifier is created using insert( ). After the insertion is made; '*n*' is the index of the symbol table entry for the string in *lex_buffer*. This index is communicated to the parser by setting *tokenval* to *n* and the token in the token field of the entry is returned.

Data Structures for Symbol Table:

If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only. A symbol table can be implemented in one of the following ways:

- Linear (sorted or unsorted) list
- Binary Search Tree
- Hash table

Module 13: Run Time Storage

Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

1. List

- Linear list is a simplest kind of mechanism to implement the symbol table.
- In this method an array is used to store names and associated information.
- New names can be added in the order as they arrive.
- The pointer 'available' is maintained at the end of all stored records. The list data structure using arrays is as given below.

Name 1	Info 1
Name 2	Info 2
Name 3	Info 3
Name n	Info n

- To retrieve the information about some name we start from beginning of array and go on searching up to available pointer. If we reach at pointer available without finding a name, we get an error "use of undeclared name".
- While inserting a new name we should ensure that it should not be already there. If it is there another error occurs i.e. "Multiple defined Name".
- The advantage of list organization is that it takes minimum amount of space.

2. Self-Organizing List:

- This symbol table implementation is using linked list. A link field is added to each record.
- Records are searched in the order pointed by the link of link field.
- A pointer "First" is maintained to point to first record of the symbol table.

	Name 1	Info 1
	Name 2	Info 2
First →	Name 3	Info 3
	Name n	Info n

- The reference to these names can be Name 3, Name 1, Name 4, Name 2.
- When the name is referenced or created it is moved to the front of the list.
- The most frequently referred names will tend to be front of the list. Hence time to most frequently referred names will be the least.

3. **Binary tree:**

- When the organization symbol table is by means of binary tree, the node structure will be as follows:

Left Child	Symbols	Information	Right Child

- The left child field stores the address of previous symbol.
- Right child field stores the address of next symbol. The symbol field is used to store the name of the symbol.
- Information field is used to give information about the symbol.
- Binary tree structure is basically a binary search tree in which the value of left is always less than the value of parent node. Similarly, the value of right node is always more or greater than the parent node.

Advantages

- Insertion of any symbol is efficient.
- Any symbol can be searched efficiently using binary search method.

Disadvantages

- This structure consumes lot of space in storing left pointer, right pointer and null pointers.

4. **Hash Table:**

- Hashing is an important technique used to search the records of symbol table. This method is superior to list organization.
- In hashing scheme two tables are maintained-a hash table and symbol table.
- The hash table consists of k entries from 0,1 to k-1. These entries are basically pointers to symbol table pointing to the names of symbol table.
- To determine whether the 'Name' is in symbol table, we use a hash function 'h' such that h(name) will result any integer between 0 to k-1. We can search any name by *position = h(name)*.
- Using this position, the exact locations of name can be obtained in symbol table.
- Hash function should result in uniform distribution of names in symbol
- Hash function should be such that there will be minimum number of collision. Collision is such a situation where hash function results in same location for storing the names.
- Collision resolution techniques are open addressing, chaining, rehashing.
- Advantage of hashing is quick search is possible.
- Disadvantage of hashing is that hashing is complicated to implement. Some extra space is required. Obtaining scope of variables is very difficult.

13.5 Introduction to Garbage Collection and Compaction:

Data that cannot be referenced is generally known as garbage. Many high-level programming languages remove the burden of manual memory management from the programmer by offering automatic garbage collection, which de-allocates unreachable data. Languages that offer garbage collection include Java, Perl, ML, Modula-3, PROLOG, and Smalltalk.

Module 13: Run Time Storage

Design Goals for Garbage Collectors

Garbage collection is the reclamation of chunks of storage holding objects that can no longer be accessed by a program. We need to assume that objects have a type that can be determined by the garbage collector at run time. From the type information, we can tell how large the object is and which components of the object contain references (pointers) to other objects. We also assume that references to objects are always to the address of the beginning of the object, never pointers to places within the object. Thus, all references to an object have the same value and can be identified easily.

A user program, which we shall refer to as the mutator, modifies the collection of objects in the heap. The mutator creates objects by acquiring space from the memory manager, and the mutator may introduce and drop references to existing objects. Objects become garbage when the mutator program cannot "reach" them. The garbage collector finds these unreachable objects and reclaims their space by handing them to the memory manager, which keeps track of the free space.

A Basic Requirement: Type Safety

Not all languages are good candidates for automatic garbage collection. For a garbage collector to work, it must be able to tell whether any given data element or component of a data element is, or could be used as, a pointer to a chunk of allocated memory space. A language in which the type of any data component can be determined is said to be *type safe*. There are type-safe languages like ML, for which we can determine types at compile time. There are other type safe languages, ' like Java, whose types cannot be determined at compile time, but can be determined at run time. The latter are called *dynamically typed* languages. If a language is neither statically nor dynamically type safe, then it is said to be unsafe.

Unsafe languages, which unfortunately include some of the most important languages such as C and C++, are bad candidates for automatic garbage collection. In unsafe languages, memory addresses can be manipulated arbitrarily: arbitrary arithmetic operations can be applied to pointers to create new pointers, and arbitrary integers can be cast as pointers: Thus a program no memory location can be considered to be inaccessible, and no storage can ever be reclaimed safely.

In practice, most C and C++ programs do not generate pointers arbitrarily, and a theoretically unsound garbage collector that works well empirically has been developed and used.

Performance Metrics

Garbage collection is often so expensive that, although it was invented decades ago and absolutely prevents memory leaks, it has yet to be adopted by many mainstream programming languages. The performance metrics that must be considered when designing a garbage collector.

- **Overall Execution Time:** Garbage collection can be very slow. It is important that it not significantly increase the total run time of an application. Since the garbage collector necessarily must touch a lot of data, its performance is determined greatly by how it leverages the memory subsystem.
- **Space Usage:** It is important that garbage collection avoid fragmentation and make the best use of the available memory.

- **Pause Time:** Simple garbage collectors are notorious for causing programs - the mutators - to pause suddenly for an extremely long time, as garbage collection kicks in without warning. Thus, besides minimizing the overall execution time, it is desirable that the maximum pause time be minimized. As an important special case, real-time applications require certain computations to be completed within a time limit. We must either suppress garbage collection while performing real-time tasks, or restrict maximum pause time. Thus, garbage collection is seldom used in real-time applications.
- **Program Locality:** We cannot evaluate the speed of a garbage collector solely by its running time. The garbage collector controls the placement of data and thus influences the data locality of the mutator program. It can improve a mutator's temporal locality by freeing up space and reusing it; it can improve the mutator's spatial locality by relocating data used together in the same cache or pages.

Some of these design goals conflict with one another, and trade-offs must be made carefully by considering how programs typically behave. Also objects of different characteristics may favour different treatments, requiring a collector to use different techniques for different kinds of objects.

Reachability

Most garbage collectors are based on the idea of reclaiming whole blocks that are no longer reachable from a set of roots, which are pointers into the heap that are assumed to always be accessible. The roots of a given computation consist of pointers that appear in the environment, plus the pointer to the currently computed result. A block of memory is reachable from the roots if there is a direct pointer to that block among the roots, or if there is a pointer to that block in another block that is reachable from the roots.

Looking at memory more abstractly, we see that the memory heap is simply a directed graph in which the nodes are blocks of memory and the edges are the pointers between these blocks. So reachability can be computed as a graph traversal.

Manual Storage Management vs. Automatic Garbage Collection

There are two basic strategies for dealing with garbage: manual storage management by the programmer and automatic garbage collection built into the language run-time system. Language constructs for manual storage management are provided by languages like C and C++. There is a way for the programmer to explicitly allocate blocks of memory when needed and to deallocate (or "free") them when they become garbage. Languages like Java provide automatic garbage collection: the system automatically identifies blocks of memory that can never be used again by the program and reclaims their space for later use.

Automatic garbage collection offers the advantage that the programmer does not have to worry about when to deallocate a given block of memory. In languages like C, the need to manage memory explicitly complicates any code that allocates data on the heap and is a significant burden on the programmer, not to mention a major source of bugs:

- If the programmer neglects to deallocate garbage, it creates a memory leak in which some allocated memory can never again be reused. This is a problem for long-running programs, such as operating systems. Over time, the unreclaimable garbage grows in size until it consumes all of memory, at which point the system crashes.
- If the programmer accidentally deallocates a block of memory that is still in use, this creates a dangling pointer that may be followed later, even though it now points to unallocated memory or to a new allocated value that may be of a different type.

- If a block of memory is accidentally deallocated twice, this typically corrupts the memory heap data structure even if the block was initially garbage. Corruption of the memory heap is likely to cause unpredictable effects later during execution and is extremely difficult to debug.

In practice, programmers manage explicit allocation and deallocation by keeping track of what piece of code "owns" each pointer in the system. That piece of code is responsible for deallocating the pointer later. The tracking of pointer ownership shows up in the specifications of code that manipulates pointers, complicating specification, use, and implementation of the abstraction.

Automatic garbage collection helps modular programming, because two modules can share a value without having to agree on which module is responsible for deallocating it. The details of how boxed values will be managed does not pollute the interfaces in the system.

Automatic Garbage Collection

We classify automatic memory reclamation algorithms into two classes:

- **Reference Counters:** Each allocated region knows how many references there are to it. When this number becomes zero, the region is freed.
- **Sweep Algorithms:** Starting from a set of *roots*, the collection of all accessible values is traversed in a way similar to the traversal of a directed graph.

Sweep algorithms are more commonly used in programming languages. In effect, reference counting garbage collectors increase the processing costs (through counter updating) even when there is no need to reclaim anything.

Reference Counting Garbage Collectors:

We now consider a simple, although imperfect, garbage collector, based on reference counting, which identifies garbage as an object changes from being reachable to unreachable; the object can be deleted when its count drops to zero. With a reference-counting garbage collector, every object must have a field for the reference count. Reference counts can be maintained as follows:

1. **Object Allocation:** The reference count of the new object is set to 1.
2. **Parameter Passing:** The reference count of each object passed into a procedure is incremented.
3. **Reference Assignments:** For statement u = v, where u and v are references, the reference count of the object referred to by v goes up by one, and the count for the old object referred to by u goes down by one.
4. **Procedure Returns:** As a procedure exits, all the references held by the local variables of that procedure activation record must also be decremented. If several local variables hold references to the same object, that object's count must be decremented once for each such reference.
5. **Transitive Loss of Reachability:** Whenever the reference count of an object becomes zero, we must also decrement the count of each object pointed to by a reference within the object.

There are a few problems with this conceptually simple solution:

- It imposes a lot of run-time overhead, because each time a pointer is updated, the reference counts of two blocks of memory must be updated (one incremented, one decremented). This cost can be reduced by doing compile-time analysis to determine which increments and decrements are really needed.

- It can take a long time, because deallocating one object can cause a cascade of other objects to be deallocated at the same time. The solution to this problem is to put objects to be deallocated onto a queue. When an allocation for n bytes is performed, objects taking up space totalling at least n bytes are dequeued and deallocated, possibly causing more objects to lose all their references and be enqueued.
- Worst, reference counting cannot collect garbage that lies on a cycle in the heap graph, because the reference counts will never go down to zero. Cyclical data structures are common, for instance with many representations of directed graphs.

Applications for the Apple iPhone are written in Objective C, but there is no garbage collector available for the iPhone at present. Memory is managed manually by the programmer using a built-in reference counting scheme.

Sweep Algorithms:

Sweep algorithms allow us to explore the graph of accessible values on the heap. This exploration uses a set of roots indicating the beginning of the traversal. These roots are exterior to the heap, stored most often in a stack.

The two most commonly used algorithms are *Mark & Sweep*, which constructs the list of the free cells in the heap and *Stop & Copy*, which copies cells that are still alive to a second memory region.

Mark & Sweep Garbage Collectors:

Mark & sweep proceeds in two phases:

1. A mark phase in which all reachable memory is marked as reachable.
2. A sweep phase in which all memory that has not been marked is deallocated.

This algorithm requires that every block of memory have a bit reserved in it to indicate whether it has been marked.

Marking for reachability is essentially a graph traversal; it can be implemented as either a depth-first or a breadth-first traversal. One problem with a straightforward implementation of marking is that graph traversal takes $O(n)$ space where n is the number of nodes. However, this is not as bad as the graph traversal we considered earlier, one needs only a single bit per node in the graph if we modify the nodes to explicitly mark them as having been visited in the search. Nonetheless, if garbage collection is being performed because the system is low on memory, there may not be enough added space to do the marking traversal itself.

A simple solution is to always make sure there is enough space to do the traversal. A cleverer solution is based on the observation that there is $O(n)$ space available already in the objects being traversed. It is possible to record the extra state needed during a depth-first traversal on top of the pointers being traversed. This trick is known as pointer reversal. It works because when returning from a recursive call to the marking routine, the code knows what object it came from. Therefore, the predecessor object that pointed to it does not need the word of storage that it was using to store the pointer; it can be restored on return. That word of storage is used during the recursive call to store the pointer to the predecessor's predecessor, and so on.

In the sweep phase, all unmarked blocks are deallocated. This phase requires the ability to find all the allocated blocks in the memory heap, which is possible with a little more bookkeeping information per each block.

Stop & Copy Garbage Collectors:

The principal idea of this garbage collector is to use a secondary memory in order to copy and compact the memory regions to be saved. The heap is divided into two parts: the useful part (called from-space), and the part being re-written (called to-space).

The algorithm is the following. Beginning from a set of roots, each useful part of the from-space is copied to the to-space; the new address of a relocated value is saved (most often in its old location) in order to update all of the other values that point to this value.

The contents of the rewritten cells give new roots. As long as there are unprocessed roots the algorithm continues.

In the case of sharing, in other words, when attempting to relocate a value that has already been relocated, it suffices to use the new address.

At the end of garbage collection, all of the roots are updated to point to their new addresses. Finally, the roles of the two parts are reversed for the next garbage collection.

The principal characteristics of this garbage collector are the following:

- it depends solely on the size of the objects to be kept;
- only half of the memory is available;
- it compacts memory;
- it may localize values (using breadth-first traversal);
- it does not use extra memory (only from-space + to-space);
- the algorithm is not recursive;
- it relocates values into the new part of memory;

Other Garbage Collectors:

Many other techniques, often derived from the two preceding, have been used: either in particular applications, e.g., the manipulation of large matrices in symbolic calculations, or in a general way linked to compilation techniques. Generational garbage collectors allow optimizations based on the age of the values. Conservative garbage collectors are used where there is not an explicit differentiation between immediate values and pointers (for example, when one translates into C). Finally, incremental garbage collectors allow us to avoid a noticeable slow-down at the time of garbage collection activation.

Generational Garbage Collection

Generational garbage collection separates the memory heap into two or more generations that are collected separately. In the basic scheme, there are tenured and new (untenured) generations. Garbage collection is mostly run on the new generation (minor collections), with less frequent scans of older generations (major collections). The reason this works well is that most allocated objects have a very short life span; in many programs, the longer an object has lasted, the longer it is likely to continue to last. Minor collections are much faster because they run on a smaller heap. The garbage collector doesn't waste time trying to collect long-lived objects.

After an allocated object survives some number of minor garbage collection cycles, it is promoted to the tenured generation so that minor collections stop spending time trying to collect it.

Generational collectors introduce one new source of overhead. Suppose a program mutates a tenured object to point to an untenured object. Then the untenured object is reachable from the tenured set and should not be collected. The pointers from the tenured to the new generation are called the remembered set, and the garbage collector must treat these pointers as roots. The language run-time system needs to detect the creation of such pointers. Such pointers can only be created by imperative update; that is the only way to make an old object point to a newer one. Therefore, imperative pointer updates are often more expensive than one might expect.

Module 13: Run Time Storage

Conservative Garbage Collectors:

To this point, all of the garbage collection techniques presume knowing how to tell a pointer from an immediate value. Note that in functional languages with parametric polymorphism values are uniformly represented, and in general occupy one word of memory. This is what allows having generic code for polymorphic functions.

However, this restriction on the range for integers may not be acceptable. In this case, conservative garbage collectors make it possible to avoid marking immediate values such as integers. In this case, every value uses an entire memory word without any tag bits. In order to avoid traversing a memory region starting from a root actually containing an integer, we use an algorithm for discriminating between immediate values and pointers that relies on the following observations:

- the addresses of the beginning and end of the heap are known so any value outside of these bounds is an immediate value;
- allocated objects are aligned on a word address. Every value that does not correspond to such an alignment must also be an immediate value.

Thus each heap value that is valid from the point of view of being an address into the heap is considered to be a pointer and the garbage collector tries to keep this region, including those cases where the value is in fact an immediate value. These cases may become very rare by using specific memory pages according to the size of the objects. It is not possible to guarantee that the entire unused heap is collected. This is the principal defect of this technique. However, we remain certain that only unused regions are reclaimed.

In general, conservative garbage collectors are conservative, i.e., they do not relocate objects. Indeed, as the garbage collector considers some immediate values as pointers, it would be harmful to change their value. Nevertheless, some refinements can be introduced for building the sets of roots, which allow to relocate corresponding to clearly known roots.

Garbage collection techniques for ambiguous roots are often used when compiling a functional language into C, seen here as a portable assembler. They allow the use of immediate C values coded in a memory word.

Incremental Garbage Collection:

One of the criticisms frequently made of garbage collection is that it stops the execution of a running program for a time that is perceptible to the user and is unbounded. The first is embarrassing in certain applications, for instance, rapid-action games where the halting of the game for a few seconds is too often prejudicial to the player, as the execution restarts without warning. The latter is a source of loss of control for applications which must process a certain number of events in a limited time. This is typically the case for embedded programs which control a physical device such as a vehicle or a machine tool. These applications, which are real-time in the sense that they must respond in a bounded time, most often avoid using garbage collectors.

Incremental garbage collectors must be able to be interrupted during any one of their processing phases and be able to restart while assuring the safety of memory reclamation. They give a sufficiently satisfactory method for dealing with the former case, and can be used in the latter case by enforcing a programming discipline that clearly isolates the software components that use garbage collection from those that do not.

Let us reconsider the Mark & Sweep example and see what adaptations are necessary in order to make it incremental. There are essentially two:

1. how to be sure of having marked everything during the marking phase?
2. how to allocate during either the marking phase or the reclamation phase?

Module 13: Run Time Storage

If Mark & Sweep is interrupted in the Mark phase, it is necessary to assure that cells allocated between the interruption of marking and its restart are not unduly reclaimed by the Sweep that follows. For this, we mark cells allocated during the interruption in black or grey in anticipation.

If the Mark & Sweep is interrupted during the Sweep phase, it can continue as usual in re-colouring the allocated cells white. Indeed, as the Sweep phase sequentially traverses the heap, the cells allocated during the interruption are localized before the point where the sweep restarts, and they will not be re-examined before the next garbage collection cycle.

Compacting (Copying) Garbage Collection

Collecting garbage is nice, but the space that it creates may be scattered among many small blocks of memory. This external fragmentation may prevent the space from being used effectively. A compacting (or copying) collector is one that tries to move the blocks of allocated memory together, compacting them so that there is no unused space between them. Compacting collectors tend to cause caches to become more effective, improving run-time performance after collection.

Compacting collectors are difficult to implement because they change the locations of the objects in the heap. This means that all pointers to moved objects must also be updated. This extra work can be expensive in time and storage.

Some compacting collectors work by using an object table containing pointers to all allocated objects. Objects themselves only contain pointers into (or indices of) the object table. This solution makes it possible to move all allocated objects around because there is only one pointer to each object. However, it doubles the cost of following a pointer.

At an abstract level, a copying collector starts from a set of roots and traverse all of the reachable memory-allocated objects, copying them from one half of memory into the other half. The area of memory that we copy from is called old space (or from-space) and the area of memory that we copy to is called new space (or to-space). When we copy the reachable data, we compact it so that it is in a contiguous chunk. So, in effect, we squeeze out the holes in memory that the garbage data occupied. After the copy and compaction, we end up with a compacted copy of the data in new space data and a (hopefully) large, contiguous area of memory in new space in which we can quickly and easily allocate new objects. The next time we do garbage collection, the roles of old space and new space will be reversed.

For example, suppose memory looks like this, where the shaded boxes represent different objects and the black box in the middle represents the half-way point in memory.

At this point, we've filled up half of memory, so we initiate a collection. Old space is on the left and new space on the right. Suppose further that only objects 2 and 4 are reachable from the stack. After copying and compacting, we would have a picture like this:

Notice that we copied the live data into new space, but left the unreachable data in the first half. Now we can "throw away" the first half of memory (this doesn't really require any work).

After copying the data into new space, we restart the computation where it left off. The computation continues allocating objects, but this time allocates them in the other half of memory (i.e., new space). The fact that we compacted the data makes it easy for the interpreter to allocate objects, because it has a large, contiguous hunk of free memory. So, for instance, we might allocate a few more objects:

When the new space fills up and we are ready to do another collection, we flip our notions of new and old. Now old space is on the right and new space on the left. Suppose now that Obj 4 and Obj 6 boxes are the reachable live objects. We copy them into the other half of memory and compact them, throwing away the old data:

What happens if we do a copy but there's no extra space left over? Typically, the garbage collector will ask the operating system for more memory. If the OS says that there's no more available (virtual) memory, then the collector throws up its hands and terminates the whole program.

Last Moment Tutorials

Module 14: Compiler-Compilers

Content:
JAVA Compiler Environment, YACC.

14.1 JAVA Compiler Environment:

JavaCC:

JavaCC (Java compiler compiler) is an open source parser generator for the JAVA programming language. JavaCC is similar to YACC in that it generates a parser for the grammar provided in EBNF (Extended Backus Naur Form) notation, except the output in java source code. Unlike YACC, however, JavaCC generates top-down parsers, which limits it to the LL(k) class f grammar. The tree builder that accompanies LL(k). JJTree in which constructs trees from the bottom to up. Java Compiler is the most popular parser generator for use with Java application. A parser generator is the tool that reads the grammar specification and converts it to a java program that can recognize match to the grammar. In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generator such as tree building, action debugging. JavaCC works with any Java VM version 1.2 or greater. It has been certified to be 100% pure Java. A java compiler is a compiler for the java programming language. The most common form of output from a java Compiler are java class files containing platform-neutral java bytecode. There are also certain compilers which generates optimized native machine code for a particular hardware/operating system combinations.

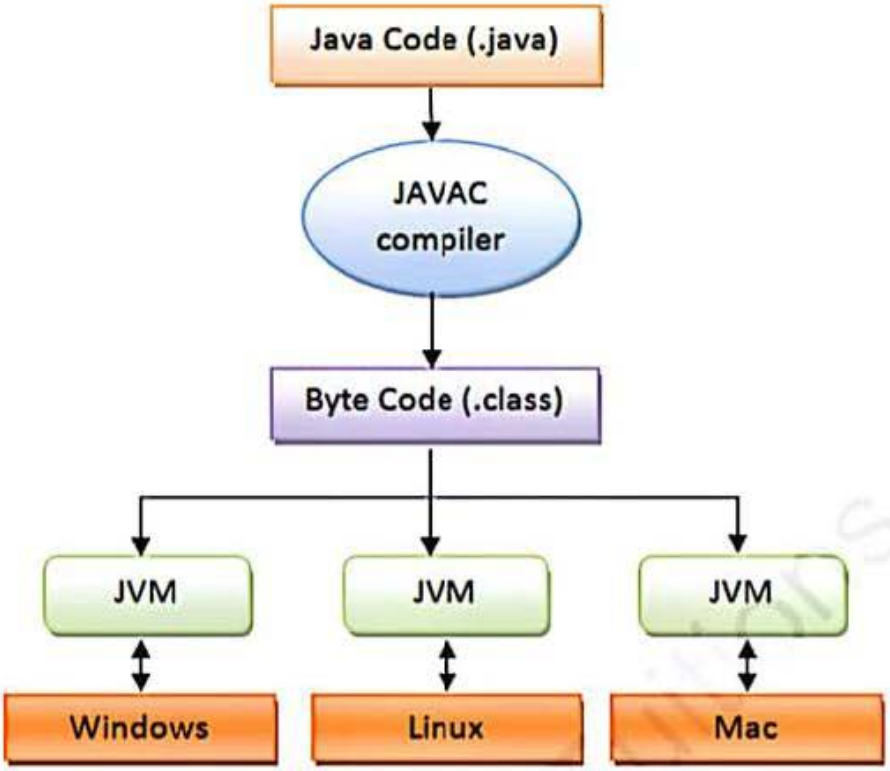
Java Virtual Machine:

Java Virtual Machine (JVM) is the execution program that executes the compiled file of java program called bytecode and produce specified results. JVM provides a Java runtime system that support exception handling which provides detail information of debugging code. JVM is compatible with almost all operating system and can also be implemented independently on any hardware. Java bytecode is the intermediate language that is compiled form of a Java program text file or a java source code file and is compiled mostly through Java Compiler or another compiling program. Ada source can also be converted to Java bytecode and run through JVM. Java runtime Environment is the framework that comprises Standard Java Class Library, Compiler to create bytecode and JVM. When Java Virtual Machine comes with most of the hardware and Software that make Java program write once and execute anywhere.

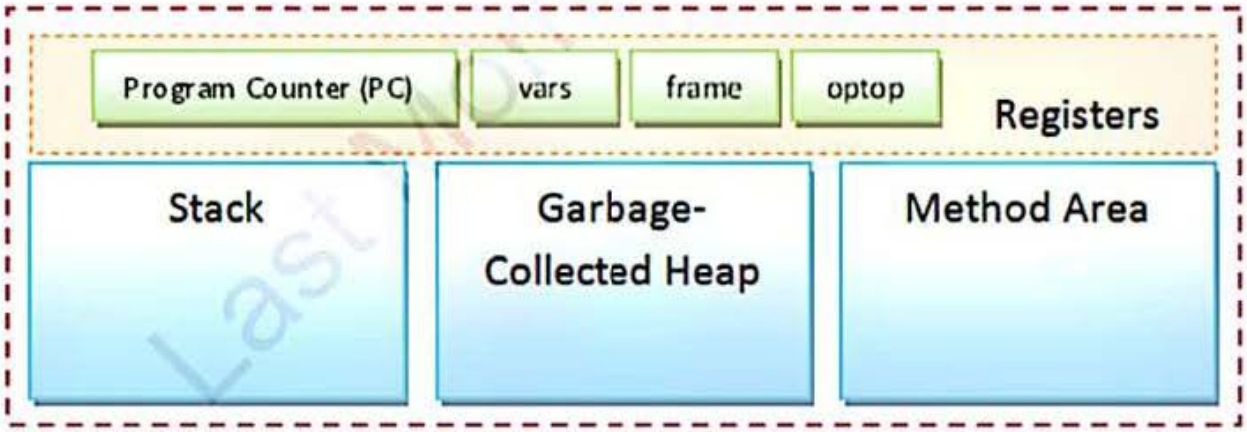
Execution Architecture of Java Virtual Machine

At first Java Program is created in simple text format. Then this file is compiled to be converted to bytecode. The bytecode is passed through the Java Virtual Machine for execution. Then Java Virtual Machine converts the bytecode into machine language that can be understood by a computing device and the Output is generated. Suppose there is a program for adding two numbers, named as "add.java". When it is compiled then it is converted to "add.class". This "add. Class" is passed through the JVM and output/ machine language is generated. Then JVM interact with the operating system to present the output.

Module 14: Compiler-Compilers



Java Virtual Machine like its real counterpart, executes the program and generate output. To execute any code, JVM utilizes different components. JVM is divided into several components like the stack, the garbage-collected heap, the registers and the method area. Let us see diagram representation of JVM.



The Stack:

Stack in Java virtual machine stores various method arguments as well as the local variables of any method. Stack also keep track of each and every method invocation. This is called **Stack Frame**. There are three registers that helps in stack manipulation. They are vars, frame, optop. This registers points to different parts of current Stack.

There are three sections in Java stack frame:

- **Local Variables:** The local variables section contains all the local variables being used by the current method invocation. It is pointed to by the **vars** register.
- **Execution Environment:** The execution environment section is used to maintain the operations of the stack itself. It is pointed to by the **frame** register.

- **Operand Stack:** The operand stack is used as a work space by bytecode instructions. It is here that the parameters for bytecode instructions are placed, and results of bytecode instructions are found. The top of the operand stack is pointed to by the **optop** register.

Method Area

This is the area where bytecodes reside. The program counter points to some byte in the method area. It always keeps tracks of the current instruction which is being executed (interpreted). After execution of an instruction, the JVM sets the PC to next instruction. Method area is shared among all the threads of a process. Hence if more than one threads are accessing any specific method or any instructions, synchronization is needed. Synchronization in JVM is achieved through Monitors.

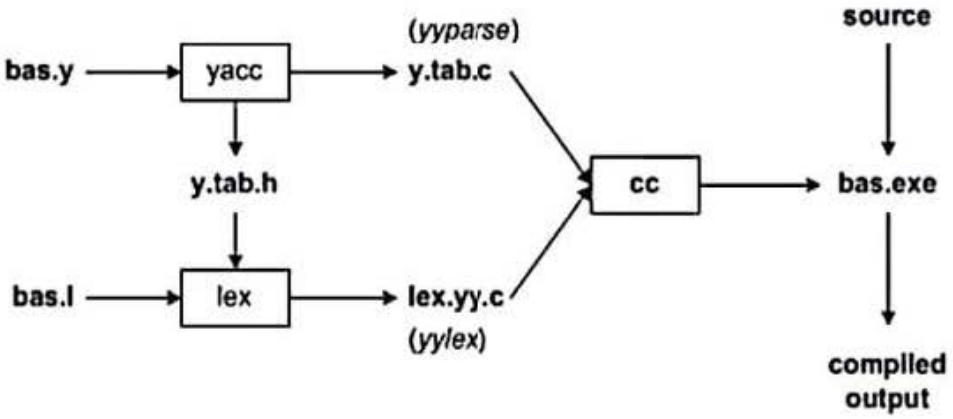
Garbage-collected Heap

The Garbage-collected Heap is where the objects in Java programs are stored. Whenever we allocate an object using new operator, the heap comes into picture and memory is allocated from there. Unlike C++, Java does not have free operator to free any previously allocated memory. Java does this automatically using Garbage collection mechanism. Till Java 6.0, **mark and sweep** algorithm is used as a garbage collection logic. Remember that the local object reference resides on Stack but the actual object resides in Heap only. Also, arrays in Java are objects, hence they also reside in Garbage-collected Heap.

14.2 YACC – Yet Another Compiler-Compiler:

YACC is a computer program for the Unix operating system. The name is an acronym for "Yet Another Compiler-Compiler". It is a LALR parser generator. Parser is the part of a compiler that tries to make syntactic sense of the source code. A LALR parser is based on an analytic grammar written in a notation similar to BNF. It was developed in 1970 by Stephen C. Johnson at AT&T Corporation and originally written in the B programming language. YACC and similar programs (largely reimplementation) have been very popular. YACC itself used to be available as the default parser generator on most Unix systems, though it has since been supplanted as the default by more recent, largely compatible, programs such as Berkeley YACC, GNU Bison, MKS YACC and Abraxas PCYACC. An updated version of the original AT&T version is included as part of Sun's Open Solaris project. YACC produces only a parser (phrase analyser); for full syntactic analysis this requires an external lexical analyser to perform the first tokenization stage (word analysis), which is then followed by the parsing stage.

Basic Structure of YACC Compiler:

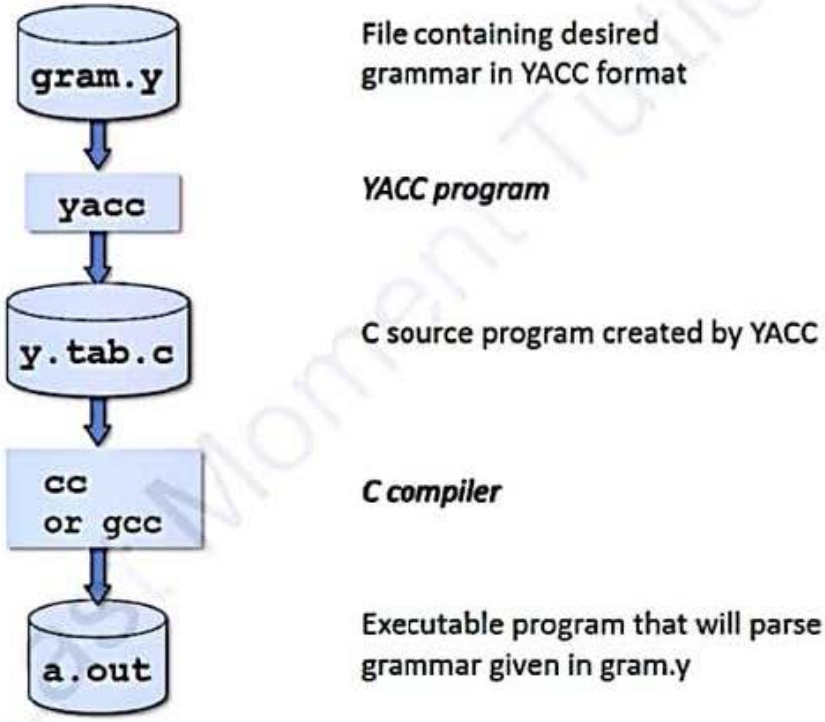


Module 14: Compiler-Compilers

Working of YACC:

YACC is designed for use with C code and generates a parser written in C. The parser is configured for use in conjunction with a lex-genrated scanner and relies on standard shared features and calls the function yylex as a scanner co-routine. A grammar specification file is provided, which is traditionally named using a .y extension. YACC is invoked on this .y file and y.tab.h is created. y.tab.c files containing a thousand or so lines of intense C code that implements an efficient LALR (1) parser out of grammar, including the code for the actions were specified. The file provides an extern function yyparse.y that will attempt to successfully parse a valid sentence. The C file is compiled normally and linked with the rest of the code and the parser is generated. By default, the parser reads from stdin and writes to stdout, just like a lex generated scanner does.

%yacc myFile.y	creates y.tab.c of C code for parser
%gcc -c y.tab.c	compiles parser code
%gcc -o parser y.tab.o lex.yy.o –ll-ly	link parser, scanner, lib.
%parser	invokes parser, reads form stdin



File Format of YACC:

Definitions

%%

Rules

%%

Supplementary Code

Module 14: Compiler-Compilers

- **Definition Section:** All code between % and % is copied to the C file. The definitions section is where we configure various parser features such as defining token codes, establishing operator precedence and associativity and setting up variables used to communicate between the scanner and the parser.
- **Rules Section:** The required productions section is where we specify the grammar rule.
- **Supplementary Code Section:** It is used for ordinary C code that we want copied verbatim to the generated C file, declarations are copied to the top of the file, user subroutines to the bottom.

Error Handling in YACC:

When the parser reads an input stream, that input stream might not match the rules in the grammar file. The parser detects the problem as early as possible. If there is an error-handling subroutine in the grammar file, the parser can allow for entering the data again, ignoring the bad data, or initiating a clean-up and recovery action. When the parser finds an error, for example, it may need to reclaim parse tree storage, delete or alter symbol table entries, and set switches to avoid generating further output.

When an error occurs, the parser stops unless you provide error-handling subroutines. To continue processing the input to find more errors, restart the parser at a point in the input stream where the parser can try to recognize more input. One way to restart the parser when an error occurs is to discard some of the tokens following the error. Then try to restart the parser at that point in the input stream. The YACC command uses a special token name, error, for error handling. Put this token in the rules file at places that an input error might occur so that you can provide a recovery subroutine. If an input error occurs in this position, the parser executes the action for the error token, rather than the normal action.

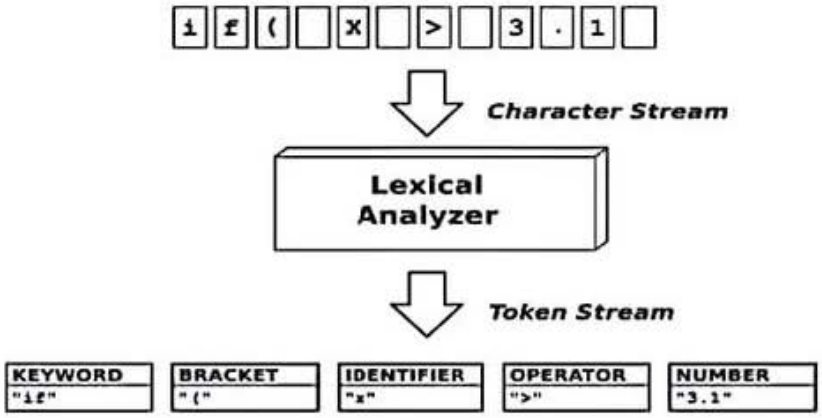
Last Moment

LEX

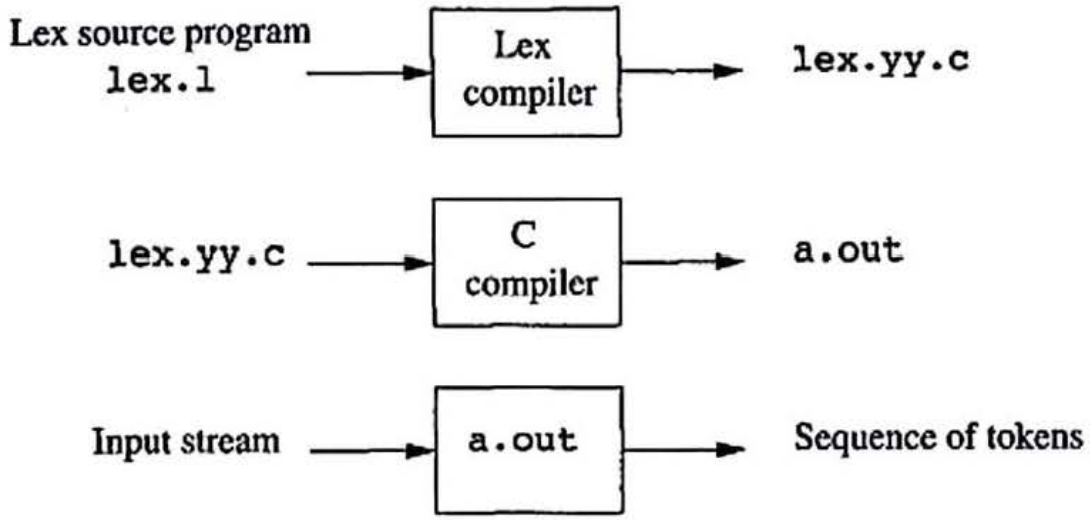
Full Concept in Hindi

LEX

- It is tool which Generate Lexical analyser
- Lexical analyser is first phase of compiler which take input as source code and generate output as tokens



- The input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler.
- The Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex.yy.c



Working

- An input file, which we call lex.l, is written in the Lex language and describes the lexical analyser to be generated.
- The Lex compiler transforms lex.l to a C program, in a file that is always named lex.yy.c.
- The later file is compiled by the C compiler into a file called a.out, as always.
- The C-compiler output is a working lexical analyser that can take a stream of input characters and produce a stream of tokens

Structure of Lex Programs:

A Lex program has the following form:

{declarations}

%%

{translation rules}

%%

{auxiliary functions}

- The declarations section includes declarations of variables
- The translation rules have the form:

Pattern {Action}

- The third section holds whatever auxiliary functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyser

Sample program

```
declaration
letter [A-Z a-z]
digit [0-9]
id (letter)((letter|digit ))*
number  {digit}({digit}|{.digit})
%% (translation )
if              {return (IF);}
else            {return(ELSE);}
{id}            {yylval = (int) installID(); return(ID);}
{number}        {yylval = (int) installID(); return(ID);}
%% (auxiliary function)
int installID()
Int installNum()
```

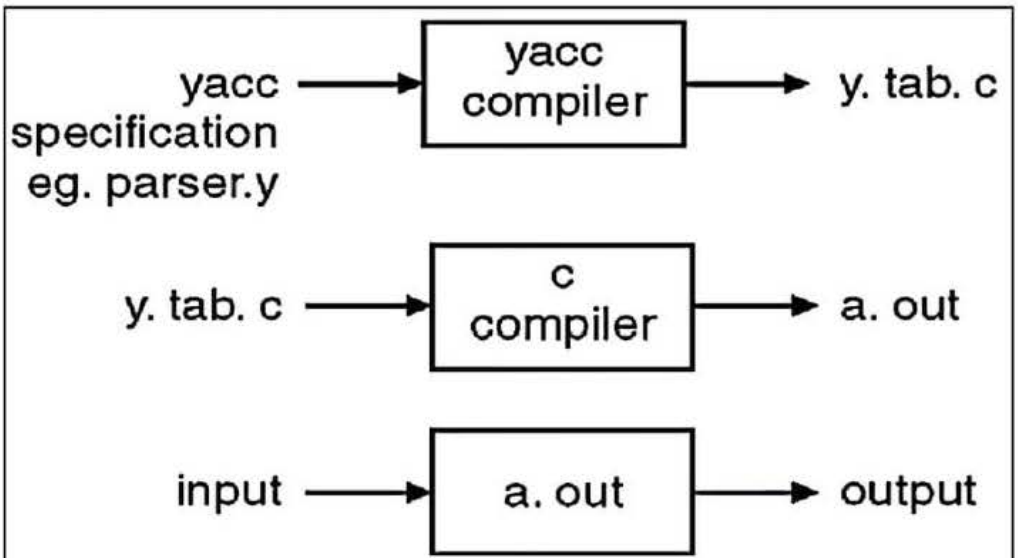
YACC

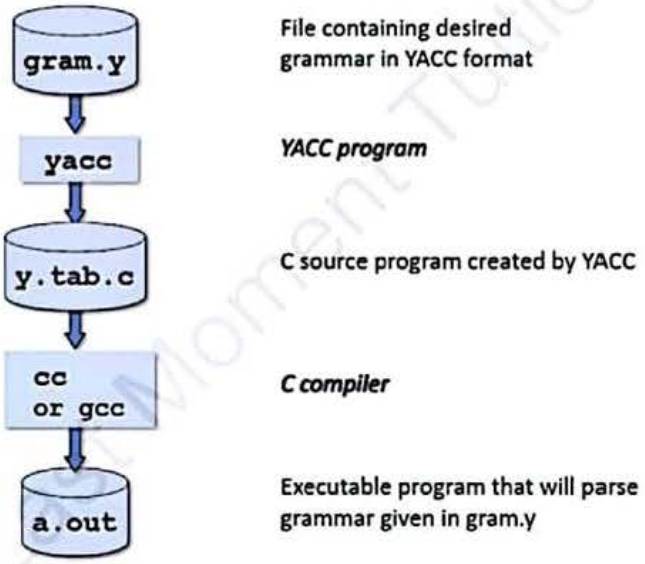
Full Concept in Hindi

YACC

- YACC stands for Yet Another Compiler Compiler.
- It is a tool which Generate LALR Parser
- Syntax analyser (parser) is second phase of compiler which take input as token and generate syntax tree

Working





Syntax

Definitions

%%

Rules

%%

Supplementary Code

- **Definition Section**: All code between % and % is copied to the C file. The definitions section is where we configure various parser features such as defining token codes, establishing operator precedence and associativity and setting up variables used to communicate between the scanner and the parser.
- **Rules Section:** The required productions section is where we specify the grammar rule.
- **Supplementary Code Section:** It is used for ordinary C code that we want copied verbatim to the generated C file, declarations are copied to the top of the file, user subroutines to the bottom.

Reference Questions for SPCC

Highlighted parts are very critical. Always remember them.
Question given in black font are University Questions.
Question given in green font are tentative question that can be asked in exams.
In case of any doubt refer Ebooks uploaded on Moodle.
Get all the updated copies of notes from Moodle.
First section consists of questions out of 5 marks and second section consists of questions for 10 marks.

Module No. 01

What is System Programming? List some System Programs and write their functions.
Hint: Refer **Section 1.2** from notes. Answer should include small description about each system programming software with their use and diagram explaining the relative location of system programming software.

What is System Software and application software? Differentiate between them. Indicate the order in which following system programs are used from developing program up to its execution.: Assembler, Loader, Linker, Macroprocessor, Compiler, Editor.
Hint: Refer **Section 1.2** from notes. Answer should include small description about each system programming software with their use and diagram explaining the relative location of system programming software. At least ten points of comparison are expected.

Questions can be asked to various language processing activities.
Concepts like Semantic Gap, PL Domain, other types of gaps.

Module No. 02

What is forward reference problem? How is it handled in two-pass and one a pass assembler?
OR State the reason for the assembler to be a multi pass program. How many passes should be there in a typical compiler?
Hint: Explain what is an assembler with diagram, give a small example forward reference problem if possible. Then explain both single pass and multi pass approach to solve forward reference problem.

What are assembler directives? Explain with example. What is the use of ORG (ORIGIN)?
Hint: Explain types of assembly language statements, with detailed explanation of Assembler directives giving proper example. Then explain ORG giving proper example.

Questions can also be asked on types of assembly language statements.
Explain various pseudo instructions like ORG, EQU, USING, DROP, ASSUME, etc. from Single pass or Multi pass assembler.
Explain what is assembler, with its advantages and disadvantages.
General design procedure of a software.

Module No. 03

Define Macro. Explain macro calls within macro giving example.
Hint: Answer must include definition, format of macro definition used of macro with some examples. Macro calls within macros is slightly **DIFFERENT** from Nested Macro. Refer notes for the answer and example.

Explain the concept of macro definition within a macro.

Hint: Explain macro. Give various features of macro. Give detailed explanation of Macro instructions defining macro (Refer **Page 6** in the notes) i.e. feature number 5 giving relevant example.

What is positional parameter in macro? **OR** Explain different ways of parameter passing in macros.
Hint: Refer feature number 1 of the Macroprocessor.

Explain difference between Procedure calls and macro calls. **OR** Explain when will a macro be used in program? How is macro different from subroutine?
Hint: Give at least ten points of comparison. Give correct definition and proper examples.

What are the features of macro to be included in the preprocessor?
Hint: Explain all the features of the Macroprocessor. Include examples wherever necessary.

Explain functions of Macroprocessor.
A program can be given and it can be asked to show the output for pass 1 and pass 2. Refer Page 13 of the notes for the answer.

Module No. 04
Explain difference between Linker and Loaders.
Hint: At least ten points, proper definition and diagram.

Explain Loading and linking scheme. **OR** Define Loader. Explain the functions of a loader in brief.
Hint: Definition, functions, block diagram is expected for both loader as well as linker.

Write short note on Dynamic Loading and Linking.
Hint: Refer notes (**Page 11** and **Page 12**) for correct answer.

Short note on any one the loading scheme.
Short note on Bootstrap Loader

Module No. 05
Write a brief note on Design of an Editor.
Hint: Include Definition, very short description overview of Editing Process. Design should be explained in detail.

Write short note on Debug monitor.
Hint: Refer notes. No need to draw the screenshots given in the notes.

Short notes on Software tools used Program development.
Short note on Editor and its types with example.

Module No. 06
Compare the performance of compilers and interpreters.
Hint: Include the definition, examples and diagram given the notes.

What is function of interpreter?
Hint: Refer notes.

Explain different phases of compiler.
Hint: Describe each phase in detail. Draw the organisation of the phases of the compiler. Also include an example how code is processed through various phases of the compiler. **Strictly refer the example given in the notes.**

Types of Compilers.

Module No. 07
Important: LEX is not same Lexical Analyser.

Explain role of lexical analyser.
Hint: Include Block Diagram, its position in the compiler, need for having a separate lexical analyser and issues in Lexical Analyser.

Write short note on LEX.
Hint: Explain LEX Compiler with all the necessary diagrams. It is not same as Lexical Analyser.

Distinguish between DFA and NFA.
Hint: Related to old syllabus, can be asked in the exam, refer TCS Notes.

Explain the role of finite regular state automata in compiler design. **OR** Role of finite automata in compiler theory.
Hint: Related to old syllabus, can be asked in the exam, refer TCS Notes.

Short Note on LEX
Hint: Refer notes. Include all the diagrams and simple example of LEX Program.

Short note on Tokens, Patterns and Lexemes.
Input Buffering.
Specification and recognition of Tokens.
Transition Diagram.
Pattern Matching based on NFA.

Module No. 08 Syntax Analysis
Explain operator precedence parsing.
Hint: Explain with example. Take very simple grammar like E → E+E | id and input like "id+id". The answer must include Algorithm, Precedence Table, Advantages & Disadvantages and conditions necessary for OPP.

Distinguish between Syntax tree and Parse trees

both)

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \varepsilon$

Hint: Show the definition of direct and indirect left recursion along with the formula for multiple left recursions.

Explain Error recovery techniques used by compiler.
Hint: Include definition, need and examples of error recovery. Refer Pearson Ebook for more content.

Find the FIRST and FOLLOW set for given grammar below:

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

Hint: Refer solved examples in the notes. Mention the **Rules for First and Follow.**

Find the FIRST and FOLLOW set for given grammar below:

$A \rightarrow Ba \mid \varepsilon$
$B \rightarrow Ab \mid \varepsilon$

Hint: FIRST(A) = FIRST(B) = {a, b, ε}. FOLLOW(A) = (b, $) and FOLLOW(B) ={a}. Also mention the **Rules for First and Follow.**

Module No. 09 Syntax Directed Translation

Difference between SDD and SDT.
Short note on SDD.
Short note on Translation Scheme (SDT).
Evaluation order for SDD.
L and S Attributed Definitions.

Module No. 10 Intermediate Code Generation

Explain DAG.
Hint: Refer Ebook and notes. Answer must include explanation of Syntax Tree & DAG, advantage of DAG over Syntax Tree and simple example showing DAG for a given statement. Explain the process (Algorithm) of generating DAG.

Convert the following code into corresponding TAC.

```
{
        int i;   int j;   float a[100];  float v;        float x;
          while (true) {
                do i = i + 1;   while (a[i] < v);
                do j = j - 1;   while (a[j] > v);
                if (i >= j)     break;
                x = a[i];       a[i] = a[j];      a[j] = x;
          }
}
```

Hint: Refer solved examples in the updated copy of notes from Moodle.

Generate three address code for given expression

```
while (a < b) do
```

```
if (c < d) then
        x = y + z
else
        x = y − z
```

Hint: Refer solved examples in the updated copy of notes from Moodle.

Module No. 11 Code Generation

What is flow graph? State its significance in code generation.
Hint: Refer Ebook and notes.

Question can be asked to identify the basic blocks in the given program. Refer solved example given in the notes.
Short note on Code Generation Algorithm can asked with example.
Problem on Heuristic Ordering.
Problem on Generation of DAG.

Module No. 12 Code Optimization

What is source of optimization?
Hint: Refer Ebook and notes.

Explain local code optimization in brief.
Hint: Explain optimization of the basic blocks. Also include the definition of code optimization, local code optimization. Refer **Section 12.2** from notes.
Important Note: Do not mix up different optimisation techniques.

Differentiate between machine dependent and machine independent code optimization.
Hint: Refer notes, Ebook and other resources. At least ten points of comparison must be present. Include simple examples.

Write short note on Code Optimization Techniques.
Hint: Refer notes, ebook and other resources. Include simple examples. Answer must contain Global as well as Local code optimization techniques. Include the organisation of the optimizing compiler.

Write short note on Code motion.
Hint: Refer notes, ebook and other resources. Include simple examples.

Module No. 13 Run Time Storage

Write a short a note on Heap allocation.
Hint: Refer notes. Include the storage organisation diagram. Define heap allocation method, its need and how it is done using an example.

Explain the dangling references in run time storage allocation with example.
Hint: Refer this link: https://en.wikipedia.org/wiki/Dangling_pointer. Include Heap Allocation Strategy in the answer.

What is compacting?
Short note on Symbol Table in Runtime Storage.
Short note on Parameter Passing.

Module No. 14

Explain Java compiler environment.

Important: Single pass assembler is different from Multi pass (Two Pass) Assembler.
Pass 1 of Multi pass assembler is **NOT** Single pass assembler.
Also note that the Multi pass assembler is **NOT** same as Multi pass (Two Pass) Macro processor.

Explain with suitable flow chart working of single pass assembler. Show the structure of its databases used.
Hint: Explain all the tables in short. Tables should contain some entries present in them. Neat Flow graph is expected along with the algorithm explaining the working of the single pass Assembler. Example entries for the various tables is given at the end of the notes. Algorithm can be explained in explained in your own words unless you don't change the algorithm.

Explain with the help of neat flowchart the working of a two-pass assembler. Clearly showing the organization and uses of the databases.
Hint: Explain all the tables in short. Tables should contain some entries present in them. Neat Flow graph is expected along with the algorithm explaining the working of the Multi pass Assembler. Algorithm can be explained in explained in your own words unless you don't change the algorithm.

With reference to assembler explain the following tables with suitable example.
POT, MOT, ST, LT, BT
Hint: Explain all the tables in short. Tables should contain some entries present in them. Brief introduction of assembler is expected along with the description of the objectives of pass 1 and pass 2. Solved example for the various tables is given in the notes. Include diagram given on **Page 10** in the notes.

Explain the databases used by each pass of the 2-pass assembler. Explain how these databases are used by the 2-pass assembler when it processes the source program with an example. Clearly show all the entries in the databases built by the 2-pass assembler.

For following code what will be output generated by Pass-I and Pass-II for two pass assembler. Explain with databases.

```
ABC     START     0
        USING*, 15
        L         1, FIVE
        A         1, FOUR
        ST        1, TEMP
FOUR    DC        F'4'
FIVE    DC        F'5'
TEMP    DS        1F
        END
```

Important: Multi pass assembler is **NOT** same as Multi pass (Two Pass) Macro processor.

How is macro defined, called and expanded?
Hint: Answer must include definition, format of macro definition and an example given on **Page 2** of the notes with proper explanation.

Explain different pseudo-ops used for conditional macro expansion along with example.
Hint: Basically explanation of conditional macro expansion here. Give example for **BOTH AIF** as well as **AGO** with proper explanation. Also explain Macro labels.

Explain when will a macro be used in program? How is macro different from subroutine?

Explain with neat and labelled flowchart the working of Multi pass (Two pass) Macro processor.
Hint: Include all the three tables with some entries present in them. Draw the flow graph. Algorithm is **MUST**.

Explain anyone of the feature of the macro.
Explain anyone of the database of the Multi pass Macroprocessor.

Module No. 04

A program can be given and you are expected to find the result for pass 1 and pass 2 of Direct Linking Loader. Refer example given in the notes.
Similar examples can be given for the absolute Loader. Refer example given in the Practical.

Module No. 06
Important: Every phase **SHOULD BE EXPLAINED** in short.
EXAMPLE given in the notes should be included/referred for the answer.

Define cross compiler. Explain in brief what activities are performed in various phases of the compiler.

What are the different phases of compiler? Illustrate compilers internal presentation of source program for the following statement after each phase.

```
position = initial + rate * 60
```

OR

```
while(a>b)
    c = b + 2.0 * a – 50;
```

Hint: Refer notes.

Module No. 07 Lexical Analysis
For a regular expression (0 + 1) * 01, construct an NFA for this expression and convert this NFA into DFA and minimize the DFA.
Hint: Part of the old syllabus, but still can be asked in the paper. Refer TCS Notes. In the answer, final minimized DFA should be shown.

For regular expression (a | b) * abb construct NFA and construct it into DFA and minimize the DFA.
Hint: Part of the old syllabus, but still can be asked in the paper. Refer TCS Notes. In the answer, final minimized DFA should be shown.

Module No. 08 Syntax Analysis

Let L be the language consisting of strings of a's and b's having same number of a's and b's. Construct LL (1) grammar of L. Construct a predictive parsing table for the grammar obtained above.

Explain the working of Recursive Descent Parser and Operator Precedence Parser with examples.

Explain the LR parser. Write an algorithm for it. Show the working of this algorithm with an example.

For the following grammar construct LL(1) parser table and parse the string (a – a). Show contents of stack and input buffer and action taken after each step,

$$S \rightarrow F$$
$$S \rightarrow (S - F)$$
$$F \rightarrow a$$

Consider the following grammar with terminals (, [,) and].

$$S \rightarrow TS \mid [S]S \mid)S \mid \varepsilon$$
$$T \rightarrow (X)$$
$$X \rightarrow TX \mid [X]X \mid \varepsilon$$

(i) Construct first and follow set for the non-terminals.
(ii) Construct its LL(1) parsing table.
(iii) Is this LL(1) grammar.

Consider the following grammar:

$$S \rightarrow aSbS \mid bSaS \mid \varepsilon$$

E → TQ
T → FR
Q → +TQ | -TQ | E
R → *FR | /FR | E
F → (E) | id

OR

S → A
A → aB | Ad
B → bBC | f
C → g

Test whether following grammar is LL (1) or not. Construct LL (1) Parse table.

S → AB | g Da
A → ab | c
B → dC
C → gC | g
D → f D | g

For the given grammar below, construct operator precedence relation matrix, assuming *, + are binary operators and id as terminal symbol and E as non-terminal symbol.

E → E + E
E → E * E
E → id

Apply operator precedence parsing algorithm to obtain skeletal syntax tree for the statement

id + id * id

Eliminate left recursion present in following grammar (Remove Direct and Indirect recursion both)

S → Aa | b
A → Ac | Sd | ε

Module No. 09 Syntax Directed Translation

Syntax Directed Translation is **NOT** same as Syntax Directed Translation Scheme (SDT).

Explain Syntax Directed Translation. Give syntax directed definition to translate infix expression to postfix expression.

Hint: Refer Ebook for infix to postfix translation. Rest everything is covered in notes.

Explain synthesized and Inherited attributes used in syntax directed definitions.

Hint: Refer notes. Include examples of both types of attributes. Trees can be used to explain the cocept.

Module No. 10 Intermediate Code Generation

Explain three-address code, its types and also implementation of three address statements.
Hint: Define intermediate code. Explain what is TAC, its types and implementations in detail giving examples. Refer notes.

Explain Backpatching with an example.
Hint: Refer notes.

Draw Syntax tree and Directed Acyclic Graph (DAG) for expression

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

Hint: Refer last pages of the updated notes from Moodle. Include definition of syntax tree and parse tree. Explain the logic of developing the trees.

Explain the handling of control structures and procedures calls in code generation phase of compilers.
Hint: Explain how procedure calls and various control structure like if-else statements, switch-case statements, etc. are handled during intermediate code generation phase of the compiler. Refer notes for this answer.

Short note on Boolean Expressions.
Short note on Assignment Statements.
Short note on Declarations.

Module No. 11 Code Generation
What are different issues in Code Generation, explain in detail.
Hint: Refer **Section 11.1** from the notes. Explain all the issues in detail.

Module No. 12 Code Optimization
Explain loop optimization with example.
Hint: Define a loop. Explain various optimisation methods giving proper reasoning that how actually it will be useful and example.

Explain role of code optimization in compiler designing with suitable example. **OR** Explain different code optimization techniques in compiler design. **OR** What is code optimization? What are various strategies for code optimization? Explain Peephole optimization along with an example. **OR** Explain role of code optimization in compiler designing with suitable example.
Hint: Refer notes. Everything is explained clearly. Include all the necessary diagrams and give simple examples wherever possible.

Module No. 13 Run Time Storage
Explain run time storage organization in detail. **OR** Explain management of variable length block and storage allocation strategies. **OR** With reference to stack allocation and heap allocation explain runtime storage organization.
Hint: Refer **Section 13.1** and **Section 13.2** from the notes. All the storage allocation strategies must be covered in the answer. Include all the necessary Diagrams.

Explain the different types or Garbage collection and compaction in compilers.
Hint: Refer **Section 13.5** from the notes. Include Introduction, Automatic Garbage Collection methods and other garbage collection methods.

What is activation record? Draw the diagram of General Activation record and explain the

What is activation record? Draw the diagram of General Activation record and explain the

purpose of different fields of an activation record.
Hint: Define Activation Trees, Activation Records and Control Stacks from **Page 2** and **Page 3** from notes.

What is binding? Explain the static and dynamic binding.
Hint: Explain Binding of names and its types from **Page 1** and **Page 2** from notes.

Module No. 14
Explain difference between JAVA compiler and YACC compiler.
Hint: Refer the notes and other resources. Comparison must be to the point and should contain at least ten differences.

Write short note on Compiler-compiler.
Hint: Answer must explain what is compiler-compiler i.e. a compiler generator. Refer this link for greater understanding https://en.wikipedia.org/wiki/Compiler-compiler. Example can be given of YACC for this answer.

Strongly suggest to see

Sandeep maheshwari

videos

To change your life