| | | |
|---|---|---|
| **16CSU601B** | **SYSTEM PROGRAMMING** | **Semester – VI**<br>**4H – 4C** |

**Instruction Hours / week: L: 4 T: 0 P: 0** **Marks:** Int : **40** Ext : **60** Total: **100**

**SCOPE**

This course enables for good understanding of the role of system programming and the scope of duties and tasks of a system programmer. This course enables to learn the concepts and principles of developing system-level software (e.g., compiler, and networking software)

**OBJECTIVES**

- To introduce students the concepts and principles of system programming
- To provide students the knowledge about both theoretical and practical aspects of system programming, teaching them the methods and techniques for designing and implementing system-level programs.
- To train students in developing skills for writing system software with the aid of sophisticated OS services, programming languages and utility tools.

**UNIT-I**

**Assemblers & Loaders, Linkers:** One pass and two pass assembler design of an assembler, Absolute loader, relocation and linking concepts, relocating loader and Dynamic Linking., overview of compilation, Phases of a compiler.

**UNIT-II**

**Lexical Analysis:**

Role of a Lexical analyzer, Specification and recognition of tokens, Symbol table, lexical

**UNIT-III**

**Parsing:**

Bottom up parsing- LR parser, yaCSU. **Intermediate representations: Three** address code generation, syntax directed translation, translation of types, control Statements.

**UNIT-IV**

**Storage organization:** Activation records stack allocation.

**UNIT-V**

**Code Generation:** Object code generation

**Suggested Readings**


1. Santanu Chattopadhyaya. (2011). Systems Programming. New Delhi: PHI.
2. Alfred, V. Aho., Monica, S. Lam., Ravi Sethi.,& Jeffrey, D. Ullman. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). New Delhi: Prentice Hall.
3. Dhamdhere, D. M. (2011). Systems Programming. New Delhi: Tata McGraw Hill.
4. Leland Beck., & Manjula, D. (2008). System Software: An Introduction to System Programming (3rd ed.). New Delhi: Pearson Education.
5. Grune, D., Van Reeuwijk, K., Bal, H. E., Jacobs, C. J. H., & Langendoen, K.(2012). Modern Compiler Design (2nd ed.). Springer.


**Websites**

1. cs.lmu.edu/~ray/notes/sysprog/

2. https://www.tutorialspoint.com/assembly_programming

**ESE MARKS ALLOCATION**

| | | |
|---|---|---|
| **1.** | **Section A**<br>20 X1 = 20<br>(Online Examination) | 20 |
| **2.** | **Section C**<br>5X8 = 40<br>(Either 'A' or 'B' Choice) | 40 |
| **3.** | **Total** | **60** |

**KARPAGAM ACADEMY OF HIGHER EDUCATION**

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021.

**LECTURE PLAN**
**DEPARTMENT OF COMPUTER SCIENCE**

**STAFF NAME: Dr. T. GENISH**
**SUBJECT NAME: SYSTEM PROGRAMMING** **SUB.CODE: 16CSU601B**
**SEMESTER: VI** **CLASS: III B. Sc - CS**

| Sl.No | Lecture Duration (Periods) | Topics to be covered | Support Materials |
|---|---|---|---|
| **UNIT- I** | | | |
| 1 | 1 | **Assemblers & Loaders, Linkers:** One pass and two pass assembler design of an assembler | T2: 71-118 |
| 2 | 1 | Absolute loader | T2: 161-162 |
| 3 | 1 | Relocation and linking concepts | T2: 162-1169 |
| 4 | 1 | Relocating loader and Dynamic Linking | T2: 170-183 |
| 5 | 1 | Overview of compilation | T2: 183-185 |
| 6 | 1 | Phases of a compiler | T2: 185-192,W1 |
| 7 | 1 | Recapitulation and Discussion of Possible Questions | |
| | | **Total No. Of Hours Planned for unit I** | **07** |
| **TEXT BOOK:** | | 1. **T1:** Alfred, V. Aho., Monica, S. Lam., Ravi Sethi.,& Jeffrey, D. Ullman. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). New Delhi: Prentice Hall. | |

| | | | |
|---|---|---|---|
| **WEB SITES** | | W1: cs.lmu.edu/~ray/notes/sysprog/ | |
| **Sl.No** | **Lecture Duration (Periods)** | **Topics to be covered** | **Support Materials** |
| | | **UNIT- II** | |
| 1 | 1 | Role of a Lexical analyzer | T1: 109-113 |
| 2 | 1 | Contd... Role of a Lexical analyzer | T1: 109-113 |
| 3 | 1 | Contd… Specification and recognition of tokens | T1: 116-135,W1 |
| 4 | 1 | Contd…Specification and recognition of tokens | T1: 116-135,W1 |
| 5 | 1 | Contd…Specification and recognition of tokens | T1: 116-135,W1 |
| 6 | 1 | Contd…Symbol table | T1: 85-99,W2 |
| 7 | 1 | Contd…Symbol table | T1: 85-99,W2 |
| 8 | 1 | Contd…Symbol table | T1: 85-99,W2 |
| 9 | 1 | Recapitulation and Discussion of Possible Questions | |
| | | **Total No. Of Hours Planned for unit II:** | **09** |
| **TEXT BOOK:** | | **T1** | |
| **WEB SITES** | | W2: https://www.tutorialspoint.com/assembly_programming | |
| **Sl.No** | **Lecture Duration (Periods)** | **Topics to be covered** | **Support Materials** |

| | | | |
|---|---|---|---|
| **UNIT- III** | | | |
| 1 | 1 | Bottom up parsing- LR parser | T1: 233-256 |
| 2 | 1 | yaCSU | W1 |
| 3 | 1 | Intermediate representations: Three address code generation | T1: 363-369,W1 |
| 4 | 1 | syntax directed translation | T1: 303-306 |
| 5 | 1 | translation of types | T1: 370-378 |
| 6 | 1 | control Statements. | T1: 399-408 |
| 7 | 1 | Recapitulation and Discussion of Possible Questions | |
| | | **Total No. Of Hours Planned for unit III:** | **07** |
| **TEXT BOOK:** | | Alfred, V. Aho., Monica, S. Lam., Ravi Sethi.,& Jeffrey, D. Ullman. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). New Delhi: Prentice Hall. | |
| **WEB SITES** | | **W1** | |
| **Sl.No** | **Lecture Duration (Periods)** | **Topics to be covered** | **Support Materials** |
| **UNIT- IV** | | | |
| 1 | 1 | **Storage organization** | T2: 435-459, W2 |
| 2 | 1 | Contd… **Storage organization** | |
| 3 | 1 | Contd… **Storage organization** | |
| 4 | 1 | Contd…Activation records stack allocation | T2: 463-481 |
| 5 | 1 | Contd…Activation records stack allocation | |
| 6 | 1 | Contd…Activation records stack allocation | |
| 7 | 1 | Recapitulation and Discussion of Possible Questions | |

| | | | |
|---|---|---|---|
| 8 | 1 | Recapitulation and Discussion of Possible Questions | |
| | | **Total No. Of Hours Planned for unit IV:** | **08** |
| **TEXT BOOKS:** | | T2: Dhamdhere, D. M. (2011). Systems Programming. New Delhi: Tata McGraw Hill. | |
| **WEBSITES** | | **W2** | |
| **Sl.No** | **Lecture Duration (Periods)** | **Topics to be covered** | **Support Materials** |
| | | **UNIT- V** | |
| 1 | 1 | **Code Generation** | T1: 505-520,W2 |
| 2 | 1 | **Contd… Code Generation** | |
| 3 | 1 | Object code generation | T1: 520-530 |
| 4 | 1 | **Contd…** Object code generation | |
| 5 | 1 | Recapitulation and Discussion of Possible Questions | |
| 6 | 1 | Recapitulation and Discussion of Possible Questions | |
| 7 | 1 | Discussion of Previous ESE Question Paper | |
| 8 | 1 | Discussion of Previous ESE Question Paper | |
| | | **Total No. Of Hours Planned for unit V:** | **08** |
| | | **Overall Planned Hours : 40** | |
| **TEXT BOOKS:** | | **T1** | |
| **WEBSITES** | | - | |

**Suggested Readings**


1. Santanu Chattopadhyaya. (2011). Systems Programming. New Delhi: PHI.
2. Alfred, V. Aho., Monica, S. Lam., Ravi Sethi.,& Jeffrey, D. Ullman. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). New Delhi: Prentice Hall.
3. Dhamdhere, D. M. (2011). Systems Programming. New Delhi: Tata McGraw Hill.
4. Leland Beck., & Manjula, D. (2008). System Software: An Introduction to System Programming ($3^{rd}$ ed.). New Delhi: Pearson Education.
5. Grune, D., Van Reeuwijk, K., Bal, H. E., Jacobs, C. J. H., & Langendoen, K.(2012). Modern Compiler Design (2nd ed.). Springer.


**Websites**

1. cs.lmu.edu/~ray/notes/sysprog/
2. https://www.tutorialspoint.com/assembly_programming

# UNIT I

## SYLLABUS

**Assemblers & Loaders, Linkers:** One pass and two pass assembler design of an assembler, Absolute loader, relocation and linking concepts, relocating loader and Dynamic Linking., overview of compilation, Phases of a compiler.

**Assemblers & Loaders, Linkers:**

Assembly language is a low-level programming language for a computer or other programmable device specific to particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM, etc.

Linker and Loader are the utility programs that plays a major role in the execution of a program. The Source code of a program passes through compiler, assembler, linker, loader in the respective order, before execution. On the one hand, where the **linker** intakes the object codes generated by the assembler and combine them to generate the executable module. On the other hands, the **loader** loads this executable module to the main memory for execution.

**Linker**

- Tool that merges the object files produced by *separate compilation* or assembly and creates an executable file
- Three tasks
  - Searches the program to find library routines used by program, e.g. printf(), math routines,…
  - Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references
  - Resolves references among files

**Translation Hierarchy**

- Compiler
  - Translates high-level language program into assembly language (CS 440)

- Assembler
  - Converts assembly language programs into *object* files
    - Object files contain a combination of machine instructions, data, and information needed to place instructions properly in memory

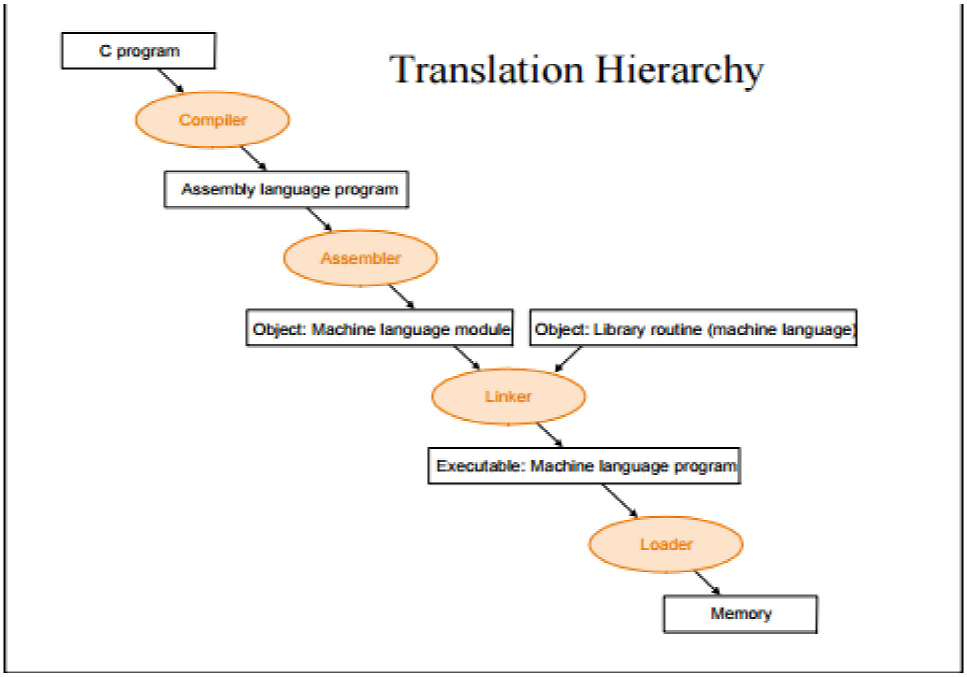
**Assemblers**

- Assemblers need to
  - translate assembly instructions and pseudo-instructions into machine instructions
  - Convert decimal numbers, etc. specified by programmer into binary
- Typically, assemblers make two passes over the assembly file
  - First pass: reads each line and records *labels* in a *symbol table*
  - Second pass: use info in symbol table to produce actual machine code for each line

**Differences between Linkers and Loaders**

| BASIS FOR COMPARISON | LINKER | LOADER |
|---|---|---|
| Basic | It generates the executable module of a source program. | It loads the executable module to the main memory. |

| BASIS FOR COMPARISON | LINKER | LOADER |
|---|---|---|
| Input | It takes as input, the object code generated by an assembler. | It takes executable module generated by a linker. |
| Function | It combines all the object modules of a source code to generate an executable module. | It allocates the addresses to an executable module in main memory for execution. |
| Type/Approach | Linkage Editor, Dynamic linker. | Absolute loading, Relocatable loading and Dynamic Run-time loading. |




**Object file format**

| | | | | | |
|---|---|---|---|---|---|
| Object file header | Text segment | Data segment | Relocation information | Symbol table | Debugging information |

- Object file header describes the size and position of the other pieces of the file
- Text segment contains the machine instructions
- Data segment contains binary representation of data in assembly file
- Relocation info identifies instructions and data that depend on absolute addresses
- Symbol table associates addresses with external labels and lists unresolved references
- Debugging info

**One pass and two pass assembler design of an assembler**

One pass assemblers perform single scan over the source code. If it encounters any undefined label, it puts it into symbol table along with the address so that the label can be replaced later when its value is encountered. On the other hand two pass assembler performs two sequential scans over the source code.

An assembler program creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents. This representation typically includes an *operation code* ("opcode") as well as other control bits and data. The assembler also calculates constant expressions and resolves symbolic names for memory locations and other entities. The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Most assemblers also include macro facilities for performing textual substitution – e.g., to generate common short sequences of instructions as inline, instead of *called* subroutines.

Some assemblers may also be able to perform some simple types of instruction set-specific optimizations. One concrete example of this may be the ubiquitous x86 assemblers from various vendors. Most of them are able to perform jump-instruction replacements (long jumps replaced by short or relative jumps) in any number of passes, on request. Others may even do simple rearrangement or insertion of instructions, such as some assemblers for RISC architectures that can help optimize a sensible instruction scheduling to exploit the CPU pipeline as efficiently as possible.

Like early programming languages such as Fortran, Algol, Cobol and Lisp, assemblers have been available since the 1950s and the first generations of text based computer interfaces. However, assemblers came first as they are far simpler to write than compilers for high-level languages. This is because each mnemonic along with the addressing modes and operands of an instruction translates rather directly into the numeric representations of that particular instruction, without much context or analysis. There have also been several classes of translators and semi automatic code generators with properties similar to both assembly and high level languages, with speedcode as perhaps one of the better known examples.

There may be several assemblers with different syntax for a particular CPU or instruction set architecture. For instance, an instruction to add memory data to a register in a x86-family processor might be `add eax,[ebx]`, in original *Intel syntax*, whereas this would be written `addl (%ebx),%eax` in the *AT&T syntax* used by the GNU Assembler. Despite different appearances, different syntactic forms generally generate the same numeric machine code, see further below. A single assembler may also have different modes in order to support variations in syntactic forms as well as their exact semantic interpretations (such as FASM-syntax, TASM-syntax, ideal mode etc., in the special case of x86 assembly programming).

**Number of passes**

There are two types of assemblers based on how many passes through the source are needed (how many times the assembler reads the source) to produce the object file.

- **One-pass assemblers** go through the source code once. Any symbol used before it is defined will require "errata" at the end of the object code (or, at least, no earlier than the point where the symbol is defined) telling the linker or the loader to "go back" and overwrite a placeholder which had been left where the as yet undefined symbol was used.
- **Multi-pass assemblers** create a table with all symbols and their values in the first passes, then use the table in later passes to generate code.

In both cases, the assembler must be able to determine the size of each instruction on the initial passes in order to calculate the addresses of subsequent symbols. This means that if the size of an operation referring to an operand defined later depends on the type or distance of the operand, the assembler will make a pessimistic estimate when first encountering the operation, and if necessary pad it with one or more "no-operation" instructions in a later pass or the errata. In an assembler with peephole optimization, addresses may be recalculated between passes to allow replacing pessimistic code with code tailored to the exact distance from the target.

The original reason for the use of one-pass assemblers was speed of assembly – often a second pass would require rewinding and rereading the program source on tape or rereading a deck of cards or punched paper tape. Later computers with much larger memories (especially disc storage), had the space to perform all necessary processing without such re-reading. The advantage of the multi-pass assembler is that the absence of errata makes the linking process (or the program load if the assembler directly produces executable code) faster.[10]

**Example:** in the following code snippet a one-pass assembler would be able to determine the address of the backward reference *BKWD* when assembling statement *S2*, but would not be able to determine the address of the forward reference*FWD* when assembling the branch statement *S1*; indeed *FWD* may be undefined. A two-pass assembler would determine both addresses in pass 1, so they would be known when generating code in pass 2,

```
S1   B    FWD
  ...
FWD   EQU *
  ...
```

```
BKWD  EQU *
  ...
S2    B   BKWD
```

**High-level assemblers**

More sophisticated high-level assemblers provide language abstractions such as:

- High-level procedure/function declarations and invocations
- Advanced control structures (IF/THEN/ELSE, SWITCH)
- High-level abstract data types, including structures/records, unions, classes, and sets
- Sophisticated macro processing (although available on ordinary assemblers since the late 1950s for IBM 700 series and since the 1960s for IBM/360, amongst other machines)
- Object-oriented programming features such as classes, objects, abstraction, polymorphism, and inheritance

A program written in assembly language consists of a series of mnemonic processor instructions and meta-statements (known variously as directives, pseudo-instructions and pseudo-ops), comments and data. Assembly language instructions usually consist of an opcode mnemonic followed by a list of data, arguments or parameters.[12] These are translated by anassembler into machine language instructions that can be loaded into memory and executed.

For example, the instruction below tells an x86/IA-32 processor to move an immediate 8-bit value into a register. The binary code for this instruction is 10110 followed by a 3-bit identifier for which register to use. The identifier for the *AL* register is 000, so the following machine code loads the *AL* register with the data 01100001.[13]

```
10110000 01100001
```

This binary computer code can be made more human-readable by expressing it in hexadecimal as follows.

```
B0 61
```

Here, `B0` means 'Move a copy of the following value into *AL*', and `61` is a hexadecimal representation of the value 01100001, which is 97 in decimal. Assembly language for the

8086 family provides the mnemonic MOV (an abbreviation of*move*) for instructions such as this, so the machine code above can be written as follows in assembly language, complete with an explanatory comment if required, after the semicolon. This is much easier to read and to remember.

```
MOV AL, 61h       ; Load AL with 97 decimal (61 hex)
```

In some assembly languages the same mnemonic such as MOV may be used for a family of related instructions for loading, copying and moving data, whether these are immediate values, values in registers, or memory locations pointed to by values in registers. Other assemblers may use separate opcode mnemonics such as L for "move memory to register", ST for "move register to memory", LR for "move register to register", MVI for "move immediate operand to memory", etc.

The x86 opcode 10110000 (B0) copies an 8-bit value into the *AL* register, while 10110001 (B1) moves it into *CL* and 10110010 (B2) does so into *DL*. Assembly language examples for these follow.[13]

```
MOV AL, 1h        ; Load AL with immediate value 1
MOV CL, 2h        ; Load CL with immediate value 2
MOV DL, 3h        ; Load DL with immediate value 3
```

The syntax of MOV can also be more complex as the following examples show.[14]

```
MOV EAX, [EBX]          ; Move the 4 bytes in memory at the address contained in EBX into EAX
MOV [ESI+EAX], CL ; Move the contents of CL into the byte at address ESI+EAX
```

In each case, the MOV mnemonic is translated directly into an opcode in the ranges 88-8E, A0-A3, B0-B8, C6 or C7 by an assembler

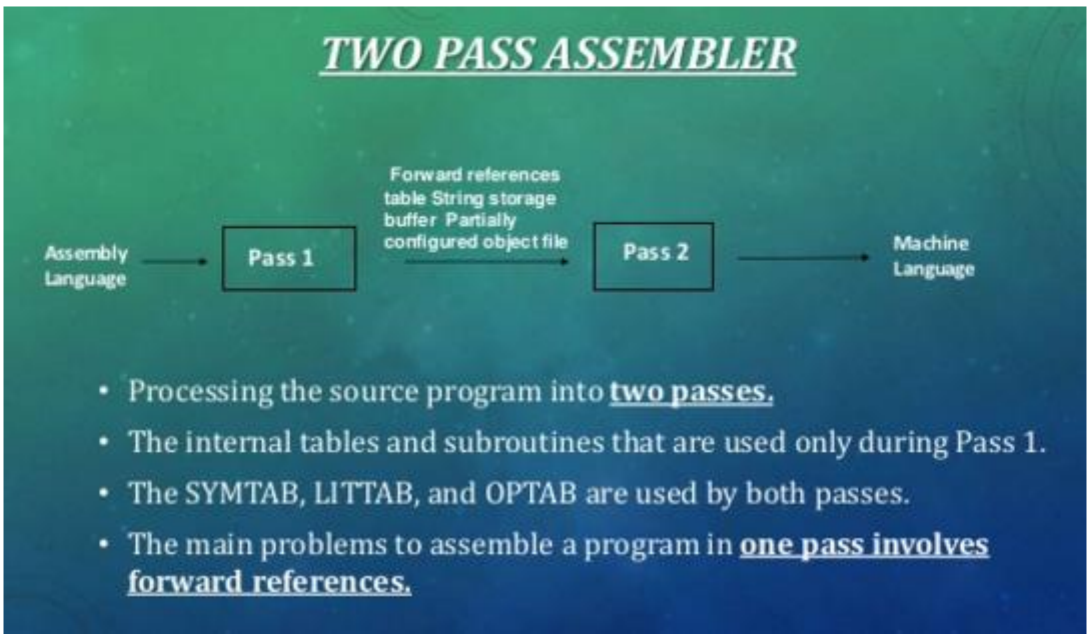
**Algorithm for Pass-1 Assembler**

```
read first input line
if OPCODE='START' then
    begin
        save #[OPERAND] as starting address
        initialize LOCCTR to starting address
        write line to intermediate file
        read next input line
    end
else
    initialize LOCCTR to 0
while OPCODE≠'END' do
    begin
        if this is not a comment line then
         begin
            if there is a symbol in the LABEL field then
begin
            search SYMTAB for LABEL
            if found then
              set error flag (duplicate symbol)
            else
              insert (LABEL, LOCCTR) into SYMTAB
         end {if symbol}
          search OPTAB for OPCODE
          if found then
         add 3 {instruction length} to LOCCTR
          else if OPCODE='WORD' then
         add 3 to LOCCTR
          else if OPCODE='RESW' then
         add 3 * #[OPERAND] to LOCCTR
   else if OPCODE='RESB' then
            add #[OPERAND] to LOCCTR
             else if OPCODE='BYTE' then
            begin
                find length of constant in bytes
                add length to LOCCTR
            end {if BYTE}
             else
            set error flag (invalid operation code)
        end  {if not a comment}
       write line to intermediate file
       read next input line
    end {while not END}
Write last line to intermediate file
Save (LOCCTR-starting address) as program length
```

**Two-pass assembler**

Uses of two-pass assembler

- A two-pass assembler reads through the source code twice. Each read-through is called a pass.
  On pass one the assembler doesn't write any code. It builds up a table of symbolic names against values or addresses.
  On pass two, the assembler generates the output code, using the table to resolve symbolic names, enabling it to enter the correct values.
  The advantage of a two-pass assember is that it allows forward referencing in the source code because when the assembler is generating code it has already found all references.



**Algorithm for Pass-2 Assembler**

```
read first input line (from intermediate file)
If OPCODE='START' then
  begin
     write listing line
     read next input line
  end {if START}
Write Header record to object program
Initialize first Text record
While OPCODE≠ 'END' do
  begin
     if this is not a comment line then
       begin
          search OPTAB for OPCODE
          if found then
            begin
if there is a symbol in OPERAND field then
                begin
             search SYMTAB for OPERAND
             if found then
               store symbol value as operand address
             else
               begin
                   store 0 as operand address
                   set error flag (undefined symbol)
               end
                end {if symbol}
              else
                store 0 as operand address
              assemble the object code instruction
          end {if opcode found}
       else if OPCODE='BYTE' or 'WORD' then
         convert constant to object code
       if object code will not fit into the current Text record then
         begin
            write Text record to object program
            initialize new Text record
         end
       add object code to Text record
     end {if not comment}
   write listing line
   read next input line
 end {while not END}
write last Text record to object program
Write End record to object program
Write last listing line
```

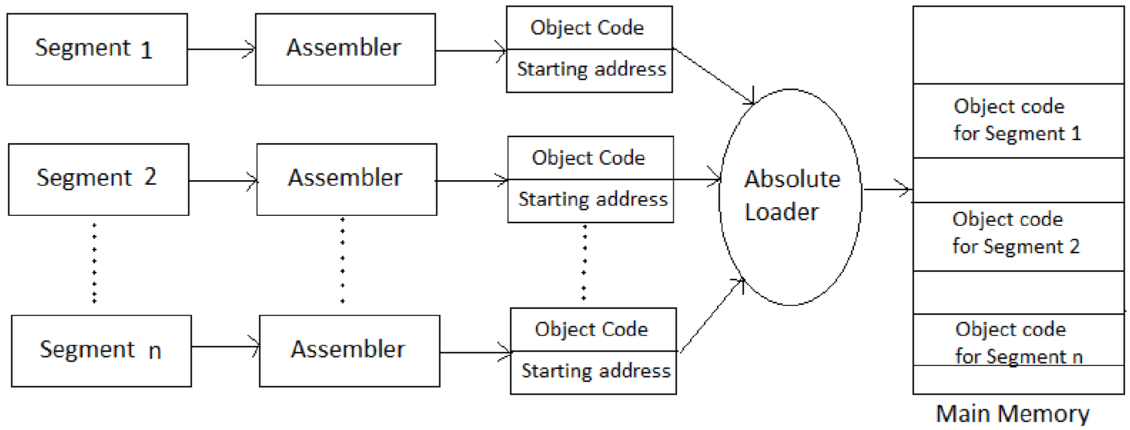
**Absolute loader**

In computer **systems** a **loader** is the part of an operating **system** that is responsible for loading programs and libraries. It is one of the essential stages in the process of starting a **program**, as it places programs into memory and prepares them for execution.

An **absolute** loader is the simplest type of **loading scheme** that **loads** the file into memory at the location specified by the beginning portion (header) of the file, then it passes control to the program.

There are two types of loaders, relocating and absolute. The absolute loader is the simplest and quickest of the two. The loader loads the file into memory at the location specified by the beginning portion (header) of the file, then passes control to the program. If the memory space specified by the header is currently in use, execution cannot proceed, and the user must wait until the requested memory becomes free.

- The absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at a specified location in the memory.
- This type of loader is called absolute loader because no relocating 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 the task of loader becomes very simple 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 programmer should take care of two things:
    - 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, it's then the programmer's duty to make necessary changes in the starting address of respective modules.
    - 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.




**Advantages:**

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

**Disadvantages:**

1. In this scheme, it's 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.
2. If at all any modification is done to some segment the starting address of immediate next segments may get changed the programmer has to take care of this issue and he/she needs to update the corresponding starting address on any modification in the source.

**The relocating loader**

The relocating loader will load the program anywhere in memory, altering the various addresses as required to ensure correct referencing. The decision as to where in memory the program is placed is done by the Operating System, not the programs header file. This is obviously more efficient, but introduces a slight overhead in terms of a small delay whilst all the relative offsets are calculated. The relocating loader can only relocate code that has been produced by a linker capable of producing relative code.

**Types of Loaders:**

Absolute Loader.
Bootstrap Loader.
Relocating Loader (Relative Loader)
Linking Loader.

**Two methods for specifying relocation as part of the object program:**

**The first method:**

- A Modification is used to describe each part of the object code that must be changed when the program is relocated.

**Consider the program**

| Line | Loc | Source statement | | | Object code |
|---|---|---|---|---|---|
| 5 | 0000 | COPY | START | 0 | |
| 10 | 0000 | FIRST | STL | RETADR | 17202D |
| 12 | 0003 | | LDB | #LENGTH | 69202D |
| 13 | | | BASE | LENGTH | |
| 15 | 0006 | CLOOP | +JSUB | RDREC | 4B101036 |
| 20 | 000A | | LDA | LENGTH | 032026 |
| 25 | 000D | | COMP | #0 | 290000 |
| 30 | 0010 | | JEQ | ENDFIL | 332007 |
| 35 | 0013 | | +JSUB | WRREC | 4B10105D |
| 40 | 0017 | | J | CLOOP | 3F2FEC |
| 45 | 001A | ENDFIL | LDA | EOF | 032010 |
| 50 | 001D | | STA | BUFFER | 0F2016 |
| 55 | 0020 | | LDA | #3 | 010003 |
| 60 | 0023 | | STA | LENGTH | 0F200D |
| 65 | 0026 | | +JSUB | WRREC | 4B10105D |
| 70 | 002A | | J | @RETADR | 3E2003 |
| 80 | 002D | EOF | BYTE | C'EOF' | 454F46 |
| 95 | 0030 | RETADR | RESW | 1 | |
| 100 | 0033 | LENGTH | RESW | 1 | |
| 105 | 0036 | BUFFER | RESB | 4096 | |
| 110 | | . | | | |
| 115 | | . | SUBROUTINE TO READ RECORD INTO BUFFER | | |
| 120 | | . | | | |
| 125 | 1036 | RDREC | CLEAR | X | B410 |
| 130 | 1038 | | CLEAR | A | B400 |
| 132 | 103A | | CLEAR | S | B440 |
| 133 | 103C | | +LDT | #4096 | 75101000 |
| 135 | 1040 | RLOOP | TD | INPUT | E32019 |
| 140 | 1043 | | JEQ | RLOOP | 332FFA |
| 145 | 1046 | | RD | INPUT | DB2013 |
| 150 | 1049 | | COMPR | A,S | A004 |
| 155 | 104B | | JEQ | EXIT | 332008 |
| 160 | 104E | | STCH | BUFFER,X | 57C003 |
| 165 | 1051 | | TIXR | T | B850 |
| 170 | 1053 | | JLT | RLOOP | 3B2FEA |
| 175 | 1056 | EXIT | STX | LENGTH | 134000 |
| 180 | 1059 | | RSUB | | 4F0000 |
| 185 | 105C | INPUT | BYTE | X'F1' | F1 |
| 195 | | . | | | |
| 200 | | . | SUBROUTINE TO WRITE RECORD FROM BUFFER | | |
| 205 | | . | | | |
| 210 | 105D | WRREC | CLEAR | X | B410 |
| 212 | 105F | | LDT | LENGTH | 774000 |
| 215 | 1062 | WLOOP | TD | OUTPUT | E32011 |
| 220 | 1065 | | JEQ | WLOOP | 332FFA |
| 225 | 1068 | | LDCH | BUFFER,X | 53C003 |
| 230 | 106B | | WD | OUTPUT | DF2008 |
| 235 | 106E | | TIXR | T | B850 |
| 240 | 1070 | | JLT | WLOOP | 3B2FEF |
| 245 | 1073 | | RSUB | | 4F0000 |
| 250 | 1076 | OUTPUT | BYTE | X'05' | 05 |
| 255 | | | END | FIRST | |

- Most of the instructions in this program use relative or immediate addressing.
- The only portions of the assembled program that contain actual addresses are the extended format instructions on lines 15, 35, and 65. Thus these are the only items whose values are affected by relocation.

**Object program**

```
HCOPY  000000001077
T0000001D17202D69202D4B1010360320262900003320074B10105D3F2FEC032010
T00001D130F20160100030F200D4B10105D3E2003454F46
T0010361DB410B400B44075101000E32019332FFADB2013A00433200857C003B850
T0010531D3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T001070073B2FEF4F000005
M00000705+COPY
M00001405+COPY
M00002705+COPY
E000000
```

- Each Modification record specifies the starting address and length of the field whose value is to be altered.
- It then describes the modification to be performed.
- In this example, all modifications add the value of the symbol COPY, which represents the starting address of the program.

**The second method:**

- There are no Modification records.
- The Text records are the same as before except that there is a *relocation bit* associated with each word of object code.
- Since all SIC instructions occupy one word, this means that there is one relocation bit for each possible instruction.

**Object program with relocation by bit mask**

```
HCOPY  00000000107A
T0000001EFFC1400334810390000362800303000154810613C000300002A0C003900002D
T00001E15E000C00364810610800334C0000454F46000003000000
T0010391EFFC040030000030E0105D30103FD8105D2800303010575480392C105E38103F
T0010570A8001000364C0000F1001000
T0010611 9FE0040030E01079301064508039DC10792C00363810644C000005
E000000
```

**Dynamic Linking**

An application that depends on **dynamic linking** calls the external files as needed during execution. The subroutines are typically part of the operating system, but may be auxiliary files that came with the application.

Dynamic linking has the following advantages: Saves **memory** and reduces swapping. Many processes can use a single DLL simultaneously, sharing a single copy of the DLL in **memory**. In contrast, Windows must load a copy of the library code into **memory** for each application that is built with a static link library.

A dynamic link library (DLL) is a collection of small programs that can be loaded when needed by larger programs and used at the same time. The small program lets the larger program communicate with a specific device, such as a printer or scanner. It is often packaged as a DLL program, which is usually referred to as a DLL file. DLL files that support specific device operation are known as device drivers.

Link editors are commonly known as linkers. The compiler automatically invokes the linker as the last step in compiling a program. The linker inserts code (or maps in shared libraries) to resolve program library references, and/or combines object modules into an executable image suitable for loading into memory. On Unix-like systems, the linker is typically invoked with the ld command.

Static linking is the result of the linker copying all library routines used in the program into the executable image. This may require more disk space and memory than dynamic linking, but is both faster and more portable, since it does not require the presence of the library on the system where it is run.

Dynamic linking is accomplished by placing the name of a sharable library in the executable image. Actual linking with the library routines does not occur until the image is run, when both the executable and the library are placed in memory. An advantage of dynamic linking is that multiple programs can share a single copy of the library.

Linking is often referred to as a process that is performed when the executable is compiled, while a dynamic linker is a special part of an operating system that loads external shared libraries into a running process and then binds those shared libraries dynamically to the running process. This approach is also called dynamic linking or late linking.

**Overview of compilation:**

The process of compiling a set of source files into a corresponding set of class files is not a simple one, but can be generally divided into three stages. Different parts of source files may proceed through the process at different rates, on an "as needed" basis.


This process is handled by the JavaCompiler class.

1. All the source files specified on the command line are read, parsed into syntax trees, and then all externally visible definitions are entered into the compiler's symbol tables.
2. All appropriate annotation processors are called. If any annotation processors generate any new source or class files, the compilation is restarted, until no new files are created.
3. Finally, the syntax trees created by the parser are analyzed and translated into class files. During the course of the analysis, references to additional classes may be found. The compiler will check the source and class path for these classes; if they are found on the source path, those files will be compiled as well, although they will not be subject to annotation processing.

***Parse and Enter***

Source files are processed for Unicode escapes and converted into a stream of tokens by the Scanner.

The token stream is read by the Parser, to create syntax trees, using a TreeMaker. Syntax trees are built from subtypes of JCTree which implementcom.sun.source.Tree and its subtypes.

Each tree is passed to Enter, which enters symbols for all the definitions encountered into the symbols. This has to done before analysis of trees which might reference those symbols. The output from this phase is a *To Do* list, containing trees that need to be analyzed and have class files generated.

Enter consists of phases; classes migrate from one phase to the next via queues.

| | | | | |
|---|---|---|---|---|
| class enter | → | Enter.uncompleted | → | MemberEnter (1) |
| | → | MemberEnter.halfcompleted | → | MemberEnter (2) |
| | → | To Do | → | (Attribute and Generate) |

1. In the first phase, all class symbols are entered into their enclosing scope, descending recursively down the tree for classes which are members of other classes. The class symbols are given a MemberEnter object as completer.

   In addition, if any package-info.java files are found, containing package annotations, then the top level tree node for the file is put on the *To Do* list as well.

2. In the second phase, classes are completed using MemberEnter.complete(). Completion might occur on demand, but any classes that are not completed that way will be eventually completed by processing the*uncompleted* queue. Completion entails

   - (1) determination of a class's parameters, supertype and interfaces.
   - (2) entering all symbols defined in the class into its scope, with the exception of class symbols which have been entered in phase

3. After all symbols have been entered, any annotations that were encountered on those symbols will be analyzed and validated.

Whereas the first phase is organized as a sweep through all compiled syntax trees, the second phase is on demand. Members of a class are entered when the contents of a class are first accessed. This is accomplished by installing completer objects in class symbols for compiled classes which invoke the MemberEnter phase for the corresponding class tree.

***Annotation Processing***

This part of the process is handled by JavacProcessingEnvironment.

Conceptually, annotation processing is a preliminary step before compilation. This preliminary step consists of a series of rounds, each to parse and enter source files, and then to determine and invoke any appropriate annotation processors. After an initial round, subsequent rounds will be performed if any of the annotation processors that are called generate any new source files or class files that need to be part of the eventual compilation. Finally, when all necessary rounds have been completed, the actual compilation is performed.

***Analyse and Generate***

Once all the files specified on the command line have been parsed and entered into the compiler's symbol tables, and after any annotation processing has occurred,JavaCompiler can proceed to analyse the syntax trees that were parsed with a view to generating the corresponding class files.

**Attr**

The top level classes are "attributed", using Attr, meaning that names, expressions and other elements within the syntax tree are resolved and associated with the corresponding types and symbols. Many semantic errors may be detected here, either by Attr, or by Check.

**Flow**

If there are no errors so far, flow analysis will be done for the class, using Flow. Flow analysis is used to check for definite assignment to variables, and unreachable statements, which may result in additional errors.

**TransTypes**

Code involving generic types is translated to code without generic types, usingTransTypes.

**Phases of a compiler:**

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.

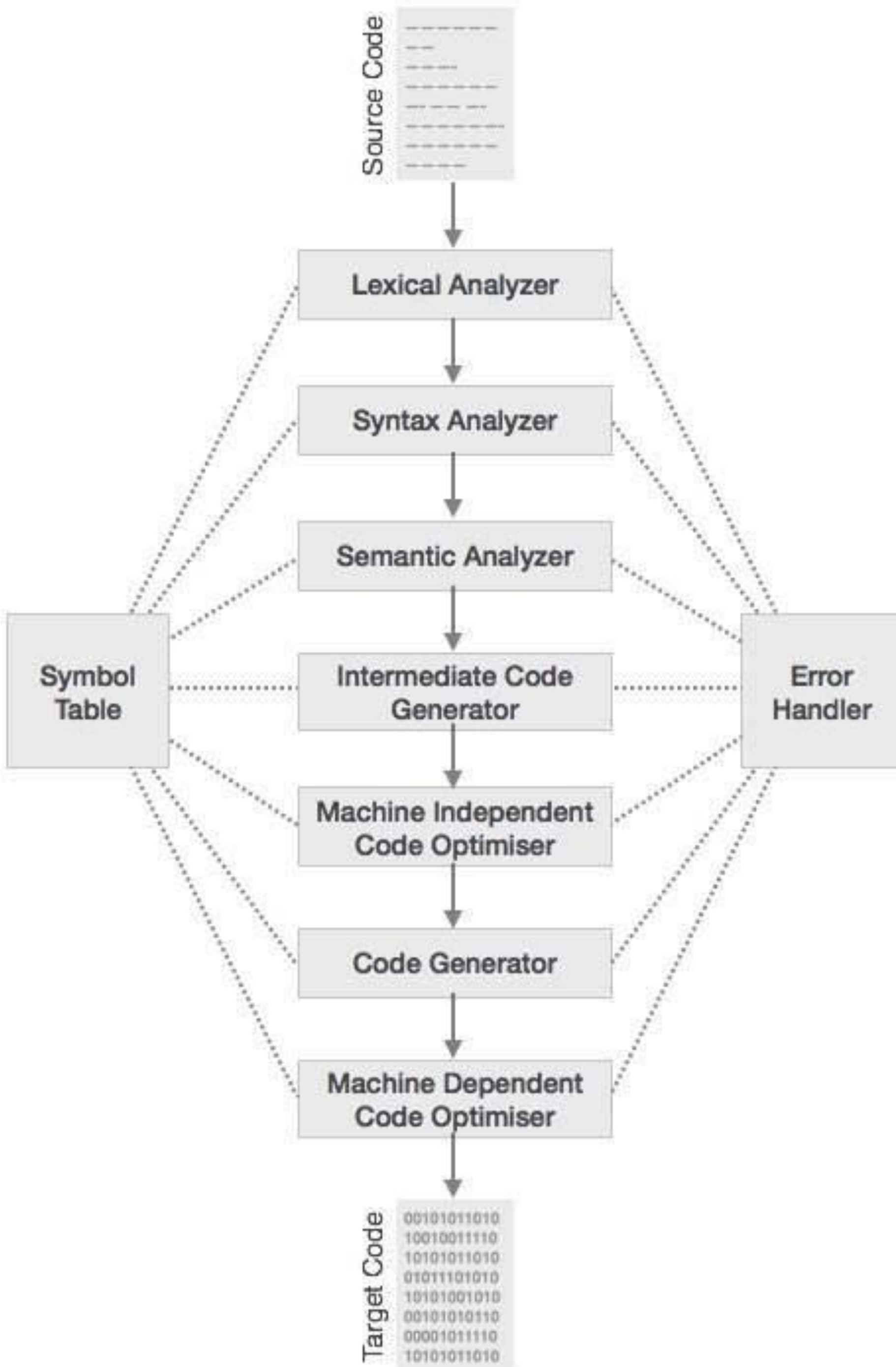
**Lexical Analysis**

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

```
<token-name, attribute-value>
```

**Syntax Analysis**

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.


Source Code
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Symbol Table
Intermediate Code Generator
Error Handler
Machine Independent Code Optimiser
Code Generator
Machine Dependent Code Optimiser
Target Code

### Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

### Intermediate Code Generation

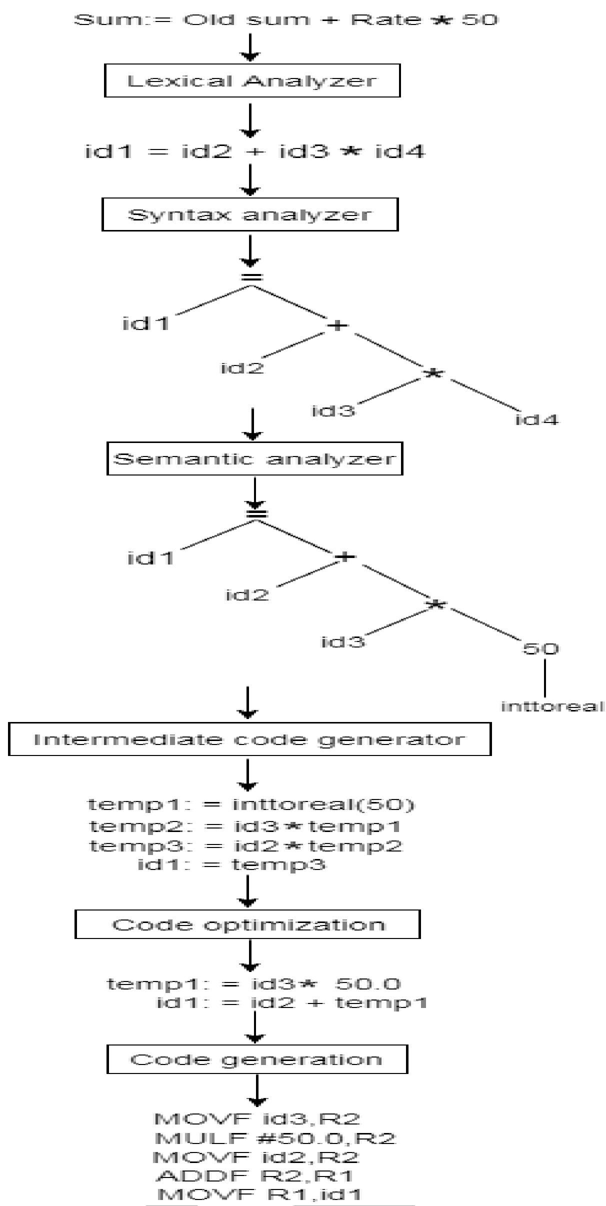
After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

### Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

### Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.




**Symbol Table**

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**DEPARTMENT OF CS, CA & IT**
**SUBJECT: SYSTEM PROGRAMMING**
**SUBJECT CODE: 16CSU601B**
**MULTIPLE CHOICE QUESTIONS**
**UNIT-1**

| sno | Questions | opt1 | opt2 |
|---|---|---|---|
| 1 | In a two pass assembler the object code generation is done during the ? | Second pass | First pass |
| 2 | Which of the following is not a type of assembler ? | one pass | two pass |
| 3 | In a two pass assembler, adding literals to literal table and address resolution of local symbols are done using ? | First pass and second respectively | Both second pass |
| 4 | In a two pass assembler the pseudo code EQU is to be evaluated during ? | Pass 1 | Pass 2 |
| 5 | Which of the following system program foregoes the production of object code to generate absolute machine code and load it into the physical main storage location from which it will be executed immediately upon completion of the assembly ? | Macro processor | Load and go assembler |
| 6 | Translator for low level programming language were termed as | Assembler | Compiler |
| 7 | An assembler is | programming language dependent | syntax dependant |
| 8 | An imperative statement | Reserves areas of memory and associates names with them | Indicates an action to be performed during execution of assembled program |
| 9 | In a two-pass assembler, the task of the Pass II is to | separate the symbol, mnemonic opcode and operand fields. | build the symbol table. |
| 10 | TII stands for | Table of incomplete instructions | Table of information instructions |
| 11 | Which of the following system software resides in main memory always ? | Text editor | Assembler |
| 12 | Daisy chain is a device for ? | Interconnecting a number of devices to number of controllers | Connecting a number of devices to a controller |

| | | | |
|---|---|---|---|
| 13 | Which of the following type of software should be used if you need to create,edit and print document ? | Word processing | Spreadsheet |
| 14 | Producer consumer problem can be solved using ? | semaphores | event counters |
| 15 | What is bootstraping? | A language interpreting other language program | A language compiling other language program |
| 16 | Shell is the exclusive feature of | UNIX | DOS |
| 17 | A program in execution is called | Process | Instruction |
| 18 | A UNIX device driver is | Structured into two halves called top half and bottom half | Three equal partitions |
| 19 | Memory | is an device that performs a sequence of operations specified by instructions in memory | is the device where information is stored |
| 20 | In which addressing mode, the operand is given explicitly in the instruction itself | absolute mode | immediate mode |
| 21 | In which addressing mode the effective address of the operand is generated by adding a constant value to the context of register | absolute mode | immediate mode |
| 22 | A garbage is | un-allocated storage | allocated storage with all across path to it destroyed |
| 23 | Which of the following program is not a utility? | Debugger | Editor |
| 24 | A development stategy whereby the executive control modules of a system are coded and tested first, is known as | Bottom-up development | Top-down development |
| 25 | Which of the following systems software does the job of merging the records from two flies into one? | Documentation system | Utility program |
| 26 | A computer can not boot if it does not have the | compiler | loader |

| | | | |
|---|---|---|---|
| 27 | The Process Manager has to keep track of: | the status of each program | the priority of each program |
| 28 | A sequence of instructions, in a computer language, to get the desired result, is known as | Algorithm | Decision Table |
| 29 | Action implementing instruction’s meaning are a actually carried out by | Instruction fetch | Instruction decode |
| 30 | A bottom up parser generates | Right most derivation | Right most derivation in reverse |
| 31 | Object program is a | Program written in machine language | Program to be translated into machine language |
| 32 | Software that allows your computer to interact with the user, applications, and hardware is called | application software | word processor |
| 33 | Programs that coordinate computer resources, provide an interface between users and the computer, | utilities | operating systems |
| 34 | Specialized programs that allow particular input or output devices to communicate with the rest of the computer system are called | operating systems | utilities |
| 35 | Also known as a service program, this type of program performs specific tasks related to managing computer resources. | utility | operating system |
| 36 | In order for a computer to understand a program, it must be converted into machine language by | operating system | utility |
| 37 | Which of the following is not a function of the operating system? | Manage resources | Internet access |
| 38 | The items that a computer can use in its functioning are collectively called its | resources | stuff |
| 39 | Programs that coordinate all of the computer’s resources including memory, processing, storage, and devices such as printers are collectively referred to as | language translators | resources |
| 40 | A compiler is a software tool that translates ___________that the computer can understand. | Algorithm into data | Source code into data |
| 41 | The object code is then passed through a program called a ___________ which turns it into an executable program. | Integer | Source code |

| | | | |
|---|---|---|---|
| 42 | When a computer is first turned on or restarted, a special type of absolute loader is executed, called a | Compile and Go loader | Boot loader |
| 43 | What is memory in Computer ? | is a sequence of instructions | is the device where information is stored |
| 44 | A program - | is a sequence of instructions | is the device where information is stored |
| 45 | The __ of a system includes the program s or instructions. | icon | software |
| 46 | Various applications and documents are represented on the Windows desktop by __. | icons | labels |
| 47 | The coordination of processor operation in CPU is controled by | CU | ALU |
| 48 | The name of the first microprocessor chip was | Intel1004 | Intel2004 |
| 49 | ntel introduced first 32 bit processor in | 1985 | 1987 |
| 50 | In a microprocessor there are 120 instructions, how many bits needed to implement this | 5 | 6 |
| 51 | Which device can understand the difference data and programs? | ALU | Registers |
| 52 | A memory bus is used for communication between | ALU and Register | Processor and Memory |
| 53 | The fourth generation computer was made up of | chips | transistor |
| 54 | The number of clock cycles necessary to complete 1 fetch cycle in 8085 is | 3 or 4 | 4 or 5 |

| opt3 | opt4 | | Answer |
|---|---|---|---|
| Zeroeth pass | Not done by assembler | | Second pass |
| three pass | load and go | | three pass |
| Second pass and first respectively | Both first pass | | Both first pass |
| not evaluated by the assembler | None of above | | Pass 1 |
| Two pass assembler | Compiler | | Load and go assembler |
| Linker | Loader | | Assembler |
| machine dependant | data dependant | | machine dependant |
| Indicates an action to be performed during optimization | None of the above | | Indicates an action to be performed during execution of assembled program |
| construct intermediate code. | synthesize the target program. | | synthesize the target program. |
| Translation of instructions information | Translation of information instruction | | Table of incomplete instructions |
| Linker | Loader | | Loader |
| Connecting a number of controller to devices | All of above | | Connecting a number of devices to a controller |

| | | | |
|---|---|---|---|
| Desktop publishing | UNIX | | Word processing |
| monitors | all of above | | all of above |
| A language compile itself | All of above | | A language compile itself |
| System software | Application software | | UNIX |
| Procedure | Function | | Process |
| Unstructured | None of the above | | Structured into two halves called top half and bottom half |
| is a sequence of instructions | is a computational unit to perform specific functions | | is the device where information is stored |
| indirect mode | index mode | | immediate mode |
| indirect mode | index mode | | indirect mode |
| allocated storage | uninitialized storage | | allocated storage with all across path to it destroyed |
| Spooler | All of the above | | Spooler |
| Left-Right development | All of the above | | Top-down development |
| Networking software | Security software | | Utility program |
| operating system | assembler | | loader |

| | | | |
|---|---|---|---|
| the information management support to a programmer using the system | both a and b | | both a and b |
| Program | All of the above | | Program |
| Instruction execution | Instruction program | | Instruction execution |
| Left most derivation | Left most derivation in reverse | | Right most derivation in reverse |
| Translation of high-level language into machine language | None of the mentioned | | Translation of high-level language into machine language |
| system software | database software | | system software |
| device drivers | language translators | | operating systems |
| device drivers | language translators | | device drivers |
| language translator | device driver | | utility |
| device driver | language translator | | language translator |
| Provide a user interface | Load and run applications | | Internet access |
| capital | properties | | resources |
| applications | interfaces | | resources |
| Computer language into data | None of the above | | Source code into data |
| Linker | None of the above | | Linker |

| | | | |
|---|---|---|---|
| Bootstrap loader | Relating loader | | Bootstrap loader |
| is an device that performs a sequence of operations specified by instructions in memory | none of these | | is the device where information is stored |
| is a device that performs a sequence of operations specified by instructions in memory | none of these | | is a sequence of instructions |
| hardware | information | | software |
| graphs | symbols | | icons |
| Registers | All of the above | | CU |
| Intel3004 | Intel4004 | | Intel4004 |
| 1989 | 1993 | | 1985 |
| 7 | 8 | | 7 |
| Motherboard | Microprocessor | | Microprocessor |
| Input and Output devices | All of the above | | Processor and Memory |
| vaccume tubes | microprocessor chips | | microprocessor chips |
| 4 or 6 | 6 or 7 | | 4 or 6 |

## UNIT II

## SYLLABUS

**Lexical Analysis:**

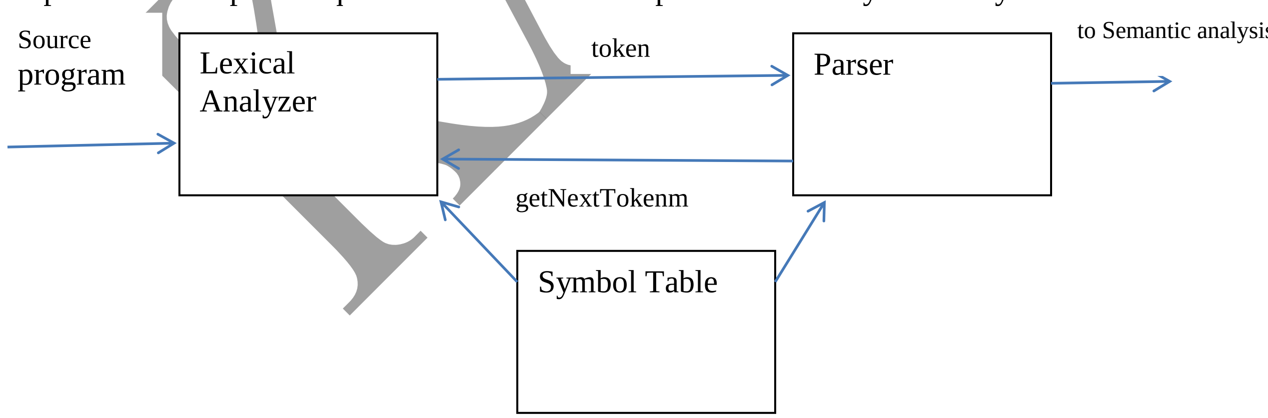
Role of a Lexical analyzer, Specification and recognition of tokens, Symbol table

## OVER VIEW OF LEXICAL ANALYSIS

To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
Secondly, having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

ROLE OF LEXICAL ANALYZER

The LA is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.




Upon receiving a 'get next token' command form the parser, the lexical analyzer reads the input character until it can identify the next token. The LA return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.
LA may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands and white spaces in the form of

blank, tab and new line characters. Another is correlating error message from the compiler with the source program.

**LEXICAL ANALYSIS VS PARSING:**

| Lexical analysis | Parsing |
|---|---|
| A Scanner simply turns an input String (say a file) into a list of tokens. These tokens represent things like identifiers, parentheses, operators etc.<br><br>The lexical analyzer (the "lexer") parses individual symbols from the source code file into tokens. From there, the "parser" proper turns those whole tokens into sentences of your grammar | A parser converts this list of tokens into a Tree-like object to represent how the tokens fit together to form a cohesive whole (sometimes referred to as a sentence).<br><br>A parser does not give the nodes any meaning beyond structural cohesion. The next thing to do is extract meaning from this structure (sometimes called contextual analysis). |

**TOKEN, LEXEME, PATTERN:**

**Token**: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are, 1) Identifiers 2) keywords 3) operators 4) special symbols 5)constants
**Pattern**: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.
**Lexeme**: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token. **Example**:

| Token | lexeme | pattern |
|---|---|---|
| const | const | const |
| if | if | if |
| relation | <,<=,= ,<>,>=,> | < or <= or = or <> or >= or letter followed by letters & digit |
| i | pi | any numeric constant |
| nun | 3.14 | any character b/w “and “except" |
| literal | "core" | pattern |

A pattern is a rule describing the set of lexemes that can represent a particular token in source program.

**LEXICAL ERRORS:**

Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there's no way to recognise a lexeme as a valid token for you lexer. Syntax errors, on the other side, will be thrown by your scanner when a given set of already recognised valid tokens don't match any of the right sides of your grammar rules. simple panic-mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected.

- Error-recovery actions are:
- Delete one character from the remaining input.
- Insert a missing character in to the remaining input.
- Replace a character by another character.
- Transpose two adjacent characters.

**DIFFERENCE BETWEEN COMPILER AND INTERPRETER**

- ✓ A compiler converts the high level instruction into machine language while an interpreter converts the high level instruction into an intermediate form.
- ✓ Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it and so on.
- ✓ List of errors is created by the compiler after the compilation process while aninterpreter stops translating after the first error.
- ✓ An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.
- ✓ The compiler produce object code whereas interpreter does not produce object code.
- ✓ In the process of compilation the program is analyzed only once and then the code is generated whereas source program is interpreted every time it is to be executed and every time the source program is analyzed. hence interpreter is less efficient than compiler.
- ✓ Examples of interpreter: A UPS Debugger is basically a graphical source level debugger but it contains built in C interpreter which can handle multiple source files. example of compiler: Borland c compiler or Turbo C compiler compiles the programs written in C or C++.

**Specification and recognition of tokens:**

There are 3 specifications of tokens:
1)Strings
2) Language
3)Regular expression

**Strings and Languages**

An **alphabet** or character class is a finite set of symbols.

A **string** over an alphabet is a finite sequence of symbols drawn from that alphabet.

A **language** is any countable set of strings over some fixed alphabet.

In language theory, the terms "sentence" 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.

**Operations on strings**

The following string-related terms are commonly used:

1. A **prefix** of string s is any string obtained by removing zero or more symbols from the end of string s. For example, ban is a prefix of banana.

2. A suffix of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, nana is a suffix of banana.

3. A substring of s is obtained by deleting any prefix and any suffix from s. For example, nan is a substring of banana.

4. 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.
5. A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of s
6. For example, baan is a subsequence of banana.

**Operations on languages:**
The following are the operations that can be applied to languages:
1. Union
2. Concatenation
3. Kleene closure
4. Positive closure

The following example shows the operations on strings: Let L={0,1} and S={a,b,c}

**Regular Expressions**

· Each regular expression r denotes a language L(r).

· Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

1.ε is a regular expression, and L(ε) is { ε }, that is, the language whose sole member is the empty string.
2. If ‘a’ is a symbol in Σ, then ‘a’ is a regular expression, and L(a) = {a}, that is, the language with one string, of length one, with ‘a’ in its one position.
3.Suppose r and s are regular expressions denoting the languages L(r) and L(s). Then, a) (r)|(s) is a regular expression denoting the language L(r) U L(s).

b) (r)(s) is a regular expression denoting the language L(r)L(s). c) (r)* is a regular expression denoting (L(r))*.
d) (r) is a regular expression denoting L(r).
4.The unary operator * has highest precedence and is left associative.
5.Concatenation has second highest precedence and is left associative.
6. | has lowest precedence and is left associative.

**Regular set**

A language that can be defined by a regular expression is called a regular set. If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s.

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.
For instance, r|s = s|r is commutative; r|(s|t)=(r|s)|t is associative.

**Regular Definitions**

Giving names to regular expressions is referred to as a Regular definition. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$d_l \rightarrow r_1$
$d_2 \rightarrow r_2$

………
$d_n \rightarrow r_n$

1.Each $d_i$ is a distinct name.
2.Each $r_i$ is a regular expression over the alphabet Σ U {$d_l, d_2, \ldots, d_{i-l}$}.

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular
definition for this set:

letter → A | B | …. | Z | a | b | …. | z | digit → 0 | 1 | …. | 9

id → letter ( letter | digit ) *

**Shorthands**

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

**1.** ***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 $^*$.

**2.** ***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

**3.** ***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]*

**Non-regular Set**

A language which cannot be described by any regular expression is a non-regular set. Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

## Symbol table

**Symbol table** is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. **Symbol table** is used by both the analysis and the synthesis parts of a compiler.

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

```
<symbol name,  type,  attribute>
```

For example, if a symbol table has to store information about the following variable declaration:

```
static int interest;
```

then it should store the entry such as:

```
<interest, int, static>
```

The attribute clause contains the entries related to the name.

*Implementation*

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

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.

*Operations*

A symbol table, either linear or hash, should provide the following operations.

insert()

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example:

```
int a;
```

should be processed by the compiler as:

```
insert(a, int);
```

lookup()

lookup() operation is used to search a name in the symbol table to determine:

- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol declared multiple times.

The format of lookup() function varies according to the programming language. The basic format should match the following:

```
lookup(symbol)
```

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

*Scope Management*

A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and**scope symbol tables** that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
. . .
int value=10;

void pro_one()
  {
  int one_1;
  int one_2;

    {             \
    int one_3;      |_  inner scope 1
    int one_4;      |
    }             /

  int one_5;

    {             \
```

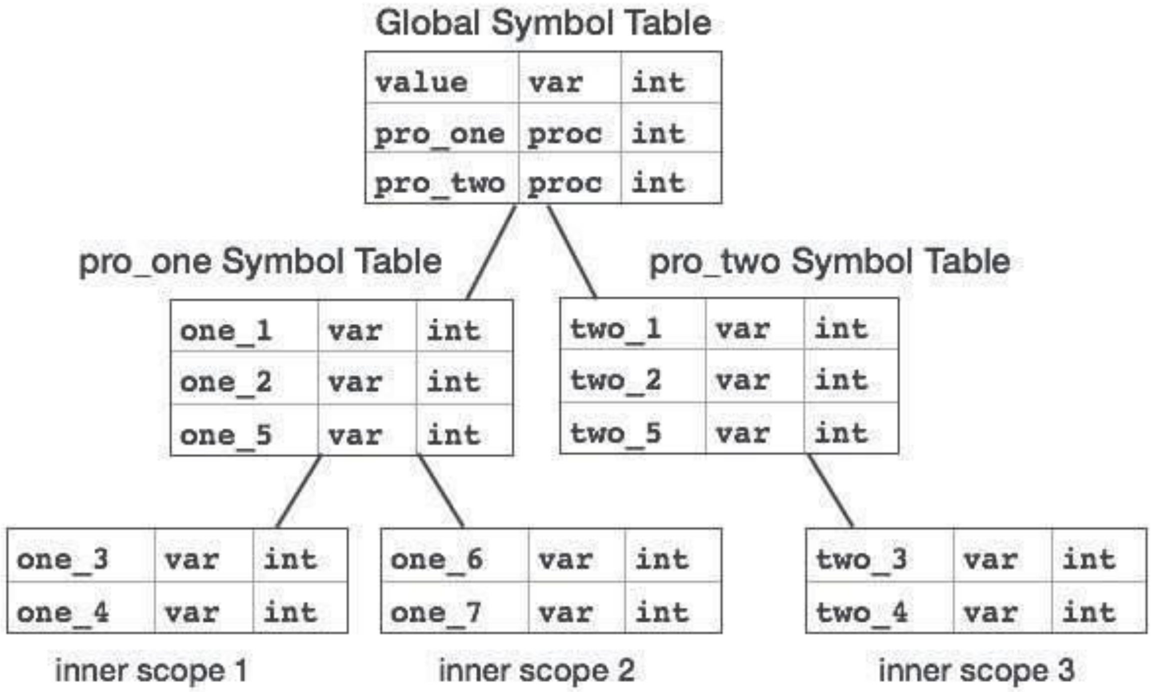
```
    int one_6;      |_  inner scope 2
    int one_7;      |
    }               /
  }

void pro_two()
  {
  int two_1;
  int two_2;

    {               \
    int two_3;      |_  inner scope 3
    int two_4;      |
    }               /

  int two_5;
  }
. . .
```

The above program can be represented in a hierarchical structure of symbol tables:




The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

- first a symbol will be searched in the current scope, i.e. current symbol table.
- if a name is found, then search is completed, else it will be searched in the parent symbol table until,
- either the name is found or global symbol table has been searched for the name.

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**DEPARTMENT OF CS, CA & IT**
**SUBJECT: SYSTEM PROGRAMMING**
**SUBJECT CODE: 16CSU601B**
**MULTIPLE CHOICE QUESTIONS**
**UNIT-2**

| sno | Questions | opt1 | opt2 |
|---|---|---|---|
| 1 | Which of the following electronic component are not found in ordinary ICs? | Diodes | Resistors |
| 2 | Intel 486 is___ bit microprocessor. | 8 Bit | 16 Bit |
| 3 | The graph that shows basic blocks and their successor relationship is called | DAG | Flow graph |
| 4 | When a computer is first turned on or resrarted, a special type of absolute loader is executed called | Boot loader | Relating loader |
| 5 | Sotware that measures, monitors, analyzes and controls real world events is called | System software | Business software |
| 6 | The root directory of a disk should be placed | at a fixed address in main memory | at a fixed location on the disk |
| 7 | Linker and Loader are the _______. | Utility programs | Sub-Task |
| 8 | Dividing a project into segments and smaller units in order to simplify analysis, design and programming efforts is called | Left right approach | Modular approach |
| 9 | System generation | is always quite simple | is always very difficult |
| 10 | While running DOS on a PC, command which can be used to duplicate the entire diskette is | COPY | DISKCOPY |
| 11 | A system program which sets up an executable program in main memory ready for execution, is | assembler | linker |
| 12 | Operating system for the laptop computer called MacLite is | windows | DOS |
| 13 | Computer general-purpose software is basically a | system software | data base software |
| 14 | Special purpose software are | application softwares | system softwares |
| 15 | In computers, operating system and utility programs are examples of | system software | device drivers |

| | | | |
|---|---|---|---|
| 16 | Control, usage and allocation of different hardware components of computer is done by | address bus | system software |
| 17 | Computer software which is designed only for the use of particular customer or organization is called | program | application |
| 18 | Computer software designed for the use of sale to general public is called | package software | application software |
| 19 | The linker ? | is same as the loader | is required to create a load module |
| 20 | A system program that combines the separately compiled modules of a program into a form suitable for execution ? | Assembler | Linking loader |
| 21 | Loading process can be divided into two separate programs, to solve some problems. The first is binder the other is ? | Linkage editor | Module Loader |
| 22 | Load address for the first word of the program is called | Linker address origin | Load address origin |
| 23 | A linker program | places the program in the memory for the purpose of execution. | relocates the program to execute from the specific memory area allocated to it. |
| 24 | Resolution of externally defined symbols is performed by | Linker | Loader |
| 25 | Relocatable programs | cannot be used with fixed partitions | can be loaded almost anywhere in memory |
| 26 | Static memory allocation is typically performed during _____________. | compilation | execution |
| 27 | Dynamic memory allocation is typically performed during _______________. | loading of the program | compilation of the program |
| 28 | Dynamic memory allocation is implementing using ________________. | queue and stacks | trees |
| 29 | ________________ are used for reduce the main memory requirements of program. | Heaps | Overlays |

| | | | |
|---|---|---|---|
| 30 | _________________is used for reducing relocation requirements. | Relocation register | Track register |
| 31 | Linking is process of binding | Internal part of a program | external functional call |
| 32 | If load origin is not equal to linked origin then relocation is performed by | Loader | Linker |
| 33 | If linked origin is not equal to translated address then relocation is performed by_____________. | Absolute Loader | Loader |
| 34 | Which is not a funciton of a loader | allocation | translation |
| 35 | A system program that set up an executable program in main memory ready for execution is | assembler | linker |
| 36 | Linker and Loader are the | Utility programs | Sub-Task |
| 37 | _______ converts assembly language programs into object files | Compiler | Assembler |
| 38 | ______ loads the executable module to the main memory. | Interpreter | Linker |
| 39 | _______ is the type of linker. | Informal | Linkage Editor |
| 40 | _______ header describes the size and position of the other pieces of the file. | Object file | Donald Knuth |
| 41 | _______ assemblers perform single scan over the source code. | Two pass | One pass |
| 42 | "opcode" is otherwise called as ______. | operation code | operable code |
| 43 | The mnemonic used to move data from "register to register" is ______. | L | R |
| 44 | Relocating Loader is otherwise called as _______ loader. | Relative | Relational |
| 45 | DLL files that support specific device operation are known as _______. | Bootstrap | device drivers |
| 46 | Source files are converted into a stream of tokens by ______. | Scanner | Memory |
| 47 | _______ is a sequence of characters that can be treated as a single logical entity. | Function | Method |
| 48 | A ________ is a sequence of characters in the source program that is matched by the Pattern. | lexagon | lexeme |
| 49 | A _______ converts the high level instruction into machine language. | Loader | Assembler |

| | | | |
|---|---|---|---|
| 50 | NASM, MASM are the examples for _________. | Analyser | Assembler |
| 51 | Resolving references among files is done by __________. | Linker | Unicode |
| 52 | ________ files contain a combination of machine instructions, data, and information. | Source | Register |
| 53 | ________ pass reads each line and records labels in a symbol table. | First | Second |
| 54 | ________ takes executable module generated by a linker. | Assembler | Linker editor |
| 55 | _______ segment contains binary representation of data in assembly file. | Data | Text |

| opt3 | opt4 | | Answer |
|---|---|---|---|
| Inductors | Transistors | | Inductors |
| 32 Bit | 64 Bit | | 32 Bit |
| Control graph | Hamiltonion graph | | Flow graph |
| Boot strap loader | " Compile and GO " loader | | Boot strap loader |
| Scientific software | Real time software | | Real time software |
| anywhere on the disk | none of these | | at a fixed location on the disk |
| Sub-problems | Process | | Utility programs |
| Top down approach | Bottom up approach | | Modular approach |
| varies in difficulty between systems | requires extensive tools to be understandable | | requires extensive tools to be understandable |
| CHKDSK | TYPE | | DISKCOPY |
| loader | compiler | | loader |
| MS-DOS | OZ | | OZ |
| package software | application software | | system software |
| utility softwares | Bespoke softwares | | application softwares |
| application software | customized software | | system software |

| | | | |
|---|---|---|---|
| application software | data bus | | system software |
| customized software | system software | | customized software |
| system software | customized software | | package software |
| is always used before programs are executed | None of above | | is required to create a load module |
| Cross compiler | Load and Go | | Linking loader |
| Relocator | None of these | | Module Loader |
| Phase library | Absolute library | | Load address origin |
| links the program with other programs needed for its execution. | interfaces the program with the entities generating its input data. | | links the program with other programs needed for its execution. |
| Compiler | Editor | | Linker |
| do not need a linker | can be loaded only at one specific location | | can be loaded almost anywhere in memory |
| loading | linking | | compilation |
| execution of the program | None of the above | | execution of the program |
| stack and heaps | graphs | | stack and heaps |
| Graphs | None of the above | | Overlays |

| | | | |
|---|---|---|---|
| Binding register | Segment Register | | Segment Register |
| External reference to the correct link time address | None of the above | | External reference to the correct link time address |
| By program itself | Relocation not performed | | Loader |
| Linker | None of the above | | Linker |
| relocation | loading | | translation |
| loader | text-editor | | loader |
| Sub-problems | Process | | Utility programs |
| Linker | Loader | | Assembler |
| Compiler | Loader | | Loader |
| Assembler | Loader | | Linkage Editor |
| Source file | Obj file | | Object file |
| Three pass | All of these | | One pass |
| ope code | Obj code | | operation code |
| LRU | LR | | LR |
| Redo | Recursive | | Relative |
| Parsing | Scheduling | | device drivers |
| Register | Unicode | | Scanner |
| Definition | Token | | Token |
| Analyser | combine | | lexeme |
| Interpreter | compiler | | compiler |

| | | | |
|---|---|---|---|
| Compiler | Linker | | Assembler |
| Source file | All of these | | Linker |
| Object | Obj code | | Object |
| Third | First & Second | | First |
| Loader | Compiler | | Loader |
| Object file | Header | | Data |

UNIT III

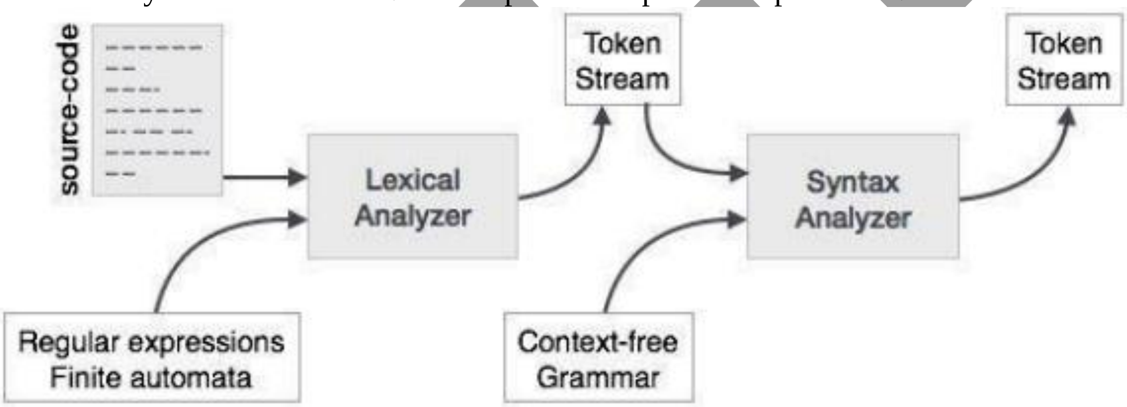
SYLLABUS

**Parsing:**

Bottom up parsing- LR parser, yaCSU. **Intermediate representations: Three** address code generation, syntax directed translation, translation of types, control Statements.

**Parsing**

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.




Working principle Syntax Analyzer

In this way, the parser accomplishes two tasks, i.e., parsing the code and looking forerrors. Finally a parse tree is generated as the output of this phase. Parsers are expected to parse the whole code even if some errors exist in the program.Parsers use error recovering strategies.

**Limitations of Syntax Analyzers**

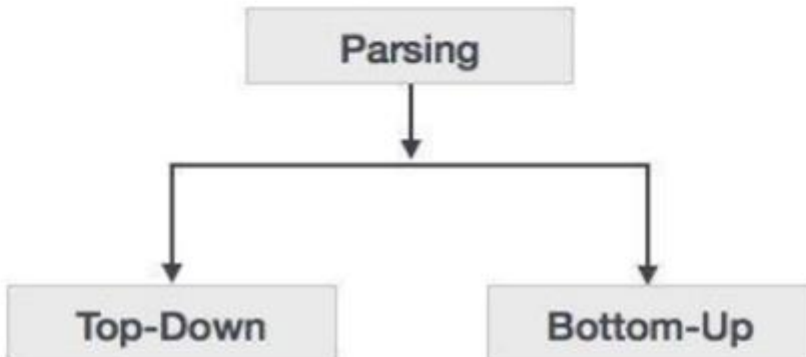
Syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers.Lexical analyzers are responsible for the validity of a token supplied by the syntaxanalyzer. Syntax analyzers have the following drawbacks:

- it cannot determine if a token is valid,
- it cannot determine if a token is declared before it is being used,
- it cannot determine if a token is initialized before it is being used,
- It cannot determine if an operation performed on a token type is valid or not.

These tasks are accomplished by the semantic analyzer, which we shall study inSemantic Analysis.

**Types of Parsing**

Syntax analyzers follow production rules defined by means of context-free grammar.The way the production rules are implemented (derivation) divides parsing into two types:top-down parsing and bottom-up parsing.




**Top-down Parsing**

When the parser starts constructing the parse tree from the start symbol and thentries to transform the start symbol to the input, it is called top-down parsing.

*Recursive descent parsing:* It is a common form of top-down parsing. It is calledrecursive, as it uses recursive procedures to process the input. Recursive descentparsing suffers from backtracking.

*Backtracking:* It means, if one derivation of a production fails, the syntax analyser restarts the process using different rules of same production. This technique mayprocess the input string more than once to determine the right production.

**Bottom-up Parsing**

As the name suggests, bottom-up parsing starts with the input symbols and tries toconstruct the parse tree up to the start symbol.

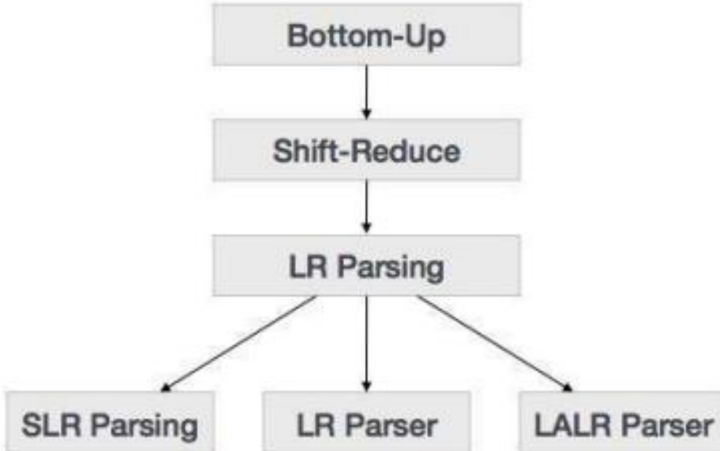
Note:

In both the cases the input to the parser is being scanned from left to right, onesymbol at a time.

The bottom-up parsing method is called “Shift Reduce” parsing. The top-down parsingis called “Recursive Decent” parsing.

Bottom-up parsing starts from the leaf nodes of a tree and works in upward directiontill it reaches the root node. Here, we start from a sentence and then apply productionrules in reverse manner in order to reach the start symbol. The image given below depictsthe bottom-up parsers available

An operator-precedence parser is one kind of shift reduce parser and predictive parseris one kind of recursive descent parser.




**Shift reduce parsing methods**

It is called as bottom up style of parsing. Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

**Shift step**

The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.

**Reduce step**

When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

*Reducing a string W to the start symbol S of a grammar.*

At each step a string matching the right side of a production is replaced by the symbol on the left.

Example:

$S \rightarrow aAcBe$; $A \rightarrow Ab$; $A \rightarrow b$; $B \rightarrow d$ and the string is abbcde, we have to reduce it to S.

Abbcde $\rightarrow$ abbcBe

$\rightarrow$ aAbcBe

$\rightarrow$ aAcBe

$\rightarrow$ S

Each replacement of the right side of the production the left side in the process above is called reduction .by reverse of a right most derivation is called Handle

$S^{*} \rightarrow \alpha A w \rightarrow \alpha\beta w$, then $A \rightarrow \beta$ in partition following is a handle of $\alpha\beta w$. The string w to the right of the handle contains only terminal symbol.

A rightmost derivation in reverse often called a canonical reduction sequence, is obtained by "Handle Pruning".

Example:

E→ E+E

E→ E*E

E→(E)

E→id

Input: $id_1+id_2*id_3$→E

| Right Sentential Form | Handle | Reducing production |
|---|---|---|
| $id_1+id_2*id_3$ | $id_1$ | E→id |
| $E+id_2*id_3$ | $id_2$ | E→id |
| $E+E*id_3$ | $id_3$ | E→id |
| E+E*E | E*E | E→E*E |
| E+E | E+E | E→E+E |
| E | | |

**LR Parser**

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide classof context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of look ahead symbols to make decisions.

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms, namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).

| LL | LR |
|---|---|
| Does a leftmost derivation. | Does a rightmost derivation in reverse. |
| Starts with the root nonterminal on the stack | Ends with the root nonterminal on the stack. |
| Ends when the stack is empty | Starts with an empty stack. |
| Uses thestack for designating what is still to be expected. | Uses the stack for designating what is already seen |
| Builds the parse tree top-down. | Builds the parse tree bottom-up. |
| Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side. | Tries to recognize a right hand side on thestack, pops it, and pushes the corresponding nonterminal. |
| Expands the non-terminals | Reduces the non-terminals. |
| Reads the terminals when it pops one off the stack. | Reads the terminalswhile itpushes them on the stack. |
| Pre-order traversal of the parse tree. | Post-order traversal of the parse tree |

**INTERMEDIATE CODE GENERATION**

Intermediate code forms:

An intermediate code form of source program is an internal form of a program created by the compiler while translating the program created by the compiler while translating the program from a high –level language to assembly code(or)object code(machine code).an intermediate source form represents a more attractive form of target code than does assembly. An optimizing Compiler performs optimizations on the intermediate source form and produces an object module.

Analysis + syntheses=translation

Creates an generate target code

Intermediate code

In the analysis –synthesis model of a compiler, the front-end translates a source program into anintermediate representation from which the back-end generates target code, in many compilers thesource code is translated into a language which is intermediate in complexity between a HLL andmachine code .the usual intermediate code introduces symbols to stand for various temporaryquantities.

Intermediate representations span the gap between the source and target languages.

• High Level Representations

- closer to the source language
- easy to generate from an input program
- code optimizations may not be straightforward

• Low Level Representations

- closer to the target machine
- Suitable for register allocation and instruction selection
- easier for optimizations, final code generation

There are several options for intermediate code. They can be either Specific to the language being implemented

- P-code for Pascal
- Byte code for Java

We assume that the source program has already been parsed and statically checked.. the variousintermediate code forms are:

a) Polish notation
b) Abstract syntax trees(or)syntax trees
c) Quadruples
d) Triples three address code
e) Indirect triples
f) Abstract machine code(or)pseudocopde

**Postfix**

The ordinary (infix) way of writing the sum of a and b is with the operator in the middle: a+b. thepostfix (or postfix polish)notation for the same expression places the operator at the right end, as ab+.In general, if e1 and e2 are any postfix expressions, and Ø to the values denoted by e1 and e2 isindicated in postfix notation nby e1e2Ø.no parentheses are needed in postfix notation because theposition and priority (number of arguments) of the operators permits only one way to decode apostfix expression.

Example:

1. (a+b)*c in postfix notation is ab+c*,sinceab+ represents the infix expression(a+b).
2. a*(b+c)is abc+* in postfix.
3. (a+b)*(c+d) is ab+cd+* in postfix.

Postfix notation can be generalized to k-ary operators for any k>=1.if k-ary operator Ø is applied topostfix expression e1,e2,……….ek, then the result is denoted by e1e2…….ek Ø. if we know thepriority of each operator then we can uniquely decipher any postfix expression by scanning it fromeither end.

Example:

Consider the postfix string ab+c*.

The right hand * says that there are two arguments to its left. since the next –to-rightmost symbol isc, simple operand, we know c must be the second operand of *.continuing to the left, we encounterthe operator +.we know the sub expression ending in + makes up the first operand of

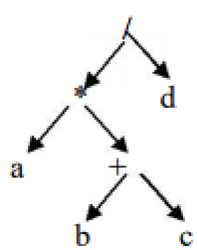
*.continuing in this way ,we deduce that ab+c* is "parsed" as (((a,b)+),c)*.

b**. syntax tree**:

The parse tree itself is a useful intermediate-language representation for a source program,especially in optimizing compilers where the intermediate code needs to extensively restructure.

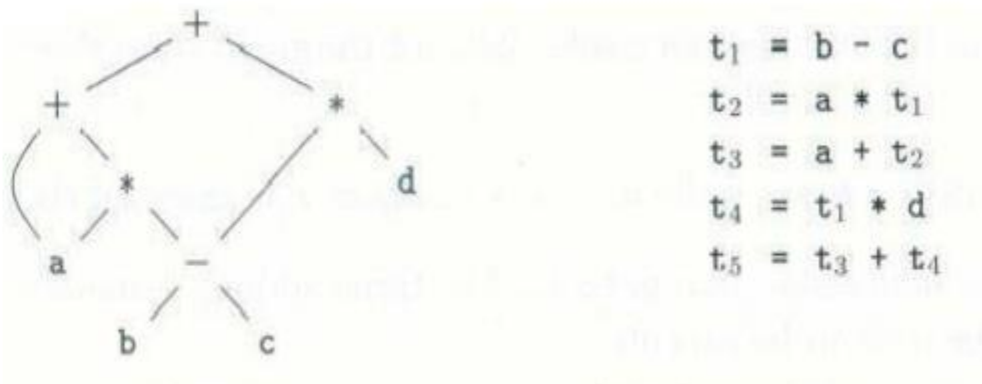
A parse tree, however, often contains redundant information which can be eliminated, Thusproducing a more economical representation of the source program. One such variant of a parse treeis what is called an (abstract) syntax tree, a tree in which each leaf represents an operand and eachinterior node an operator.

Exmples: 1) Syntax tree for the expression a*(b+c)/d




**c.Three-Address Code**: • In three-address code, there is at most one operator on the right side of aninstruction; that is, no built-up arithmetic expressions are permitted.

x+y*z t1 = y * z t2 = x + t1 • Example




(a) DAG (b) Three-address code

**LANGUAGE INDEPENDENT 3-ADDRESS CODE**

IR can be either an actual language or a group of internal data structures that are shared by the phases of the compiler. C used as intermediate language as it is flexible, compiles into efficient machine code and its compilers are widely available.In all cases, the intermediate code is a linearization of the syntax tree produced during syntax and semantic analysis. It is formed by breaking down the tree structure into sequential instructions, each of which is equivalent to a single, or small number of machine instructions.

Machine code can then be generated (access might be required to symbol tables etc). TAC can range from high- to low-level, depending on the choice of operators. In general, it is a statement containing at most 3 addresses or operands. The general form is x := y op z, where "op" is an operator, x is the result, and y and z are operands. x, y, z are variables, constants, or "temporaries". A three-address instruction consists of at most 3 addresses for each statement.

It is a linear zed representation of a binary syntax tree. Explicit names correspond to interiornodes of the graph. E.g. for a looping statement , syntax tree represents components of thestatement, whereas three-address code contains labels and jump instructions to represent theflow-of-control as in machine language. A TAC instruction has at most one operator on theRHS of an instruction; no built-up arithmetic expressions are permitted.

e.g. x + y * z can be translated as

t1 = y * z

t2 = x + t1

Where t1 & t2 are compiler–generated temporary names.

Since it unravels multi-operator arithmetic expressions and nested control-flow statements,it is useful for target code generation and optimization.

## Addresses and Instructions

• TAC consists of a sequence of instructions, each instruction may have up to threeaddresses, prototypically t1 = t2 op t3

• Addresses may be one of:

A name. Each name is a symbol table index. For convenience, we write the names as the identifier.

- A constant.

- A compiler-generated temporary. Each time a temporary address is needed, thecompiler generates another name from the stream t1, t2, t3, etc.

 Temporary names allow for code optimization to easily move Instructions

 At target-code generation time, these names will be allocated to registers or to memory.

 TAC Instructions

- Symbolic labels will be used by instructions that alter the flow of control.

The instruction addresses of labels will be filled in later.

L: t1 = t2 op t3

- Assignment instructions: x = y op z
  Includes binary arithmetic and logical operations
- Unary assignments: x = op y
  Includes unary arithmetic op (-) and logical op (!) and typeconversion
- Copy instructions: x = y
- Unconditional jump: goto L
  • L is a symbolic label of an instruction
- Conditional jumps:

if x goto L If x is true, execute instruction L next

ifFalse x goto L If x is false, execute instruction L next

- Conditional jumps:

if x relop y goto L

– **Procedure** calls. For a procedure call p(x1, …,xn)

param x1

…

paramxn

call p, n

– **Function**calls : y= p(x1, …, xn) y = call p,n , return y

**Types of three address code**

There are different types of statements in source program to which three address code has to

be generated. Along with operands and operators, three address code also use labels to provide flow of control for statements like if-then-else, for and while. The different types of

three address code statements are:

**Assignment statement**

a = b op c

In the above case b and c are operands, while op is binary or logical operator. The result of

applying op on b and c is stored in a.

**Unary operation**

a = op b This is used for unary minus or logical negation.

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

Three address code for the above example will be

t1 = -c

t2 = t1 * b
t3 = t2 + d
a = t3

**Copy Statement**

a=b

The value of b is stored in variable a.

**Unconditional jump**

goto L

Creates label L and generates three-address code ‘goto L’

v. Creates label L, generate code for expression exp, If the exp returns value true then go to
the statement labelled L. exp returns a value false go to the statement immediately following
the if statement.

**Function call**

For a function fun with n arguments a1,a2,a3….an ie.,
fun(a1, a2, a3,…an),
the three address code will be
Param a1
Param a2
…
Param an
Call fun, n
Where param defines the arguments to function.

**Array indexing**

In order to access the elements of array either single dimension ormultidimension, three address code requires base address and offset value. Base addressconsists of the address of first element in an array. Other elements of the array can beaccessed using the base address and offset value.

Example: x = y[i]
Memory location m = Base address of y + Displacement i
x = contents of memory location m
similarly x[i] = y
Memory location m = Base address of x + Displacement i
The value of y is stored in memory location m
Pointer assignment
x = &y x stores the address of memory location y
x = *y y is a pointer whose r-value is location
*x = y sets r-value of the object pointed by x to the r-value of y

Intermediate representation should have an operator set which is rich to implement. Theoperations of source language. It should also help in mapping to restricted instruction set oftarget machine.

**QUADRUPLES**-

Quadruples consists of four fields in the record structure. One field to store operator op, twofields to store operands or arguments arg1and arg2 and one field to store result res. res = arg1op arg2

Example: a = b + c

b is represented as arg1, c is represented as arg2, + as op and a as res.

Unary operators like ‘-‘do not use agr2. Operators like param do not use agr2 nor result. Forconditional and unconditional statements res is label. Arg1, arg2 and res are pointers tosymbol table or literal table for the names.

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

Three address code for the above statement is as follows

t1 = - b

t2 = t1 * d

t3 = t2 + c

t4 = - b

t5 = t4 * d

t6 = t3 + t5

a = t6

Quadruples for the above example is as follows

| Op | Arg1 | Arg2 | Res |
|---|---|---|---|
| - | B | | t1 |
| * | t1 | d | t2 |
| + | t2 | c | t3 |
| - | B | | t4 |
| * | t4 | d | t5 |
| + | t3 | t5 | t6 |
| = | t6 | | a |

**TRIPLES**

Triples uses only three fields in the record structure. One field for operator, two fields foroperands named as arg1 and arg2. Value of temporary variable can be accessed by theposition of the statement the computes it and not by location as in quadruples.

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

Triples for the above example is as follows

| Stmt no | Op | Arg1 | Arg2 |
|---|---|---|---|
| (0) | - | b | |
| (1) | * | d | (0) |
| (2) | + | c | (1) |
| (3) | - | b | |
| (4) | * | d | (3) |
| (5) | + | (2) | (4) |
| (6) | = | a | (5) |

Arg1 and arg2 may be pointers to symbol table for program variables or literal table forconstant or pointers into triple structure for intermediate results.Example: Triples for statement x[i] = y which generates two records is as follows

| Stmt no | Op | Arg1 | Arg2 |
|---|---|---|---|
| (0) | []= | x | i |
| (1) | = | (0) | y |

**INDIRECT TRIPLES**

Indirect triples are used to achieve indirection in listing of pointers. That is, it uses pointers totriples than listing of triples themselves.

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

| | Stmt no |
|---|---|
| (0) | (10) |
| (1) | (11) |
| (2) | (12) |
| (3) | (13) |
| (4) | (14) |
| (5) | (15) |
| (6) | (16) |

| Stmt no | Op | Arg1 | Arg2 |
|---|---|---|---|
| (10) | - | b | |
| (11) | * | d | (0) |
| (12) | + | c | (1) |
| (13) | - | b | |
| (14) | * | d | (3) |
| (15) | + | (2) | (4) |
| (16) | = | a | (5) |

Conditional operator and operands. Representations include quadruples, triples and indirect triples

**SYNTAX DIRECTED TRANSLATION**

- The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
- By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.
  - We associate Attributes to the grammar symbols representing the language constructs.
  - Values for attributes are computed by Semantic Rules associated with grammar productions.
- Evaluation of Semantic Rules may:
  - Generate Code;
  - Insert information into the Symbol Table;
  - Perform Semantic Check;
  - Issue error messages;
  - etc.

There are two notations for attaching semantic rules:

1. *Syntax Directed Definitions.* High-level specification hiding many implementationdetails (also called Attribute Grammars).
2. *Translation Schemes*. More implementation oriented: Indicate the order in whichsemantic rules are to be evaluated.

**Syntax Directed Definitions**

• Syntax Directed Definitions are a generalization of context-free grammars in which:

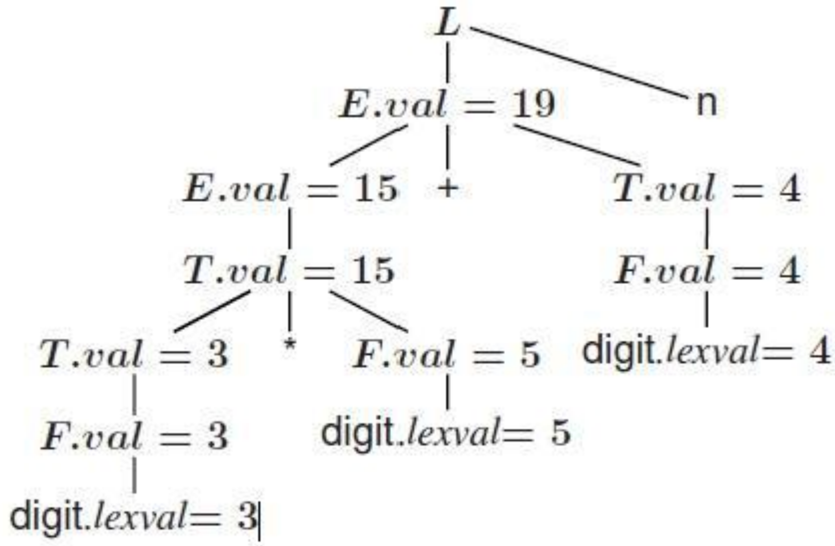
1. Grammar symbols have an associated set of Attributes;
2. Productions are associated with Semantic Rules for computing the values of attributes.

- Such formalism generates Annotated Parse-Trees where each node of the tree is a record with a field for each attribute (e.g.,X.a indicates the attribute a of the grammarsymbol X).
- The value of an attribute of a grammar symbol at a given parse-tree node is defined bya semantic rule associated with the production used at that node.

We distinguish between two kinds of attributes:
1. Synthesized Attributes. They are computed from the values of the attributes of thechildren nodes.
2. Inherited Attributes. They are computed from the values of the attributes of both thesiblings and the parent nodes
Syntax Directed Definitions: An Example
• Example. Let us consider the Grammar for arithmetic expressions. TheSyntax Directed Definition associates to each non terminal a synthesizedattribute called val.

| PRODUCTION | SEMANTIC RULE |
|---|---|
| $L \to E\mathrm{n}$ | $print(E.val)$ |
| $E \to E_1 + T$ | $E.val := E_1.val + T.val$ |
| $E \to T$ | $E.val := T.val$ |
| $T \to T_1 * F$ | $T.val := T_1.val * F.val$ |
| $T \to F$ | $T.val := F.val$ |
| $F \to (E)$ | $F.val := E.val$ |
| $F \to \mathrm{digit}$ | $F.val := \mathrm{digit}.lexval$ |

**S-ATTRIBUTED DEFINITIONS**
Definition. An S-Attributed Definition is a Syntax Directed Definition that usesonly synthesized attributes.
• Evaluation Order. Semantic rules in a S-Attributed Definition can beevaluated by a bottom-up, or PostOrder, traversal of the parse-tree.
• Example. The above arithmetic grammar is an example of an S-AttributedDefinition. The annotated parse-tree for the input 3*5+4n is:




**L-attributed definition**

A SDD its L-attributed if each inherited attribute of Xi in the RHS of A !X1 : :Xn depends only on

1. attributes of X1;X2; : : : ;Xi□1 (symbols to the left of Xi in the RHS)
2. inherited attributes of A.

Restrictions for translation schemes:

1. Inherited attribute of Xi must be computed by an action before Xi.
2. An action must not refer to synthesized attribute of any symbol to the right of that action.
3. Synthesized attribute for A can only be computed after all attributes it references have been completed (usually at end of RHS).

**Applications of Syntax-Directed Translation**

• Construction of syntax Trees

– The nodes of the syntax tree are represented by objects with a suitable number of fields.

– Each object will have an op field that is the label of the node.

– The objects will have additional fields as follows

• If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function Leaf (op, val) creates a leaf object.

• If nodes are viewed as records, the Leaf returns a pointer to a new record for a leaf.

• If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function

Node takes two or more arguments:

Node (op , c1,c2,…..ck) creates an object with first field op and k additional fields for the k children c1,c2,…..ck

**Syntax-Directed Translation Schemes**

A SDT scheme is a context-free grammar with program fragments embedded within production bodies .The program fragments are called semantic actions and can appear at any position within the production body.

Any SDT can be implemented by first building a parse tree and then pre-forming the actions in a left-to-right depth first order. i.e during preorder traversal.

The use of SDT's to implement two important classes of SDD's

1. If the grammar is LR parsable, then SDD is S-attributed.
2. If the grammar is LL parsable, then SDD is L-attributed.

Postfix Translation Schemes

The postfix SDT implements the desk calculator SDD with one change: the action for the first production prints the value. As the grammar is LR, and the SDD is S-attributed.

L →E n {print(E.val);}
E → E1 + T { E.val = E1.val + T.val }
E → E1 - T { E.val = E1.val - T.val }
E → T { E.val = T.val }
T → T1 * F { T.val = T1.val * F.val } T → F { T.val = F.val }
F → ( E ) { F.val = E.val }
F → digit { F.val = digit.lexval }

| | PRODUCTION | SEMANTIC RULES |
|---|---|---|
| 1) | $E \rightarrow E_1 + T$ | $E.node = \textbf{new}\ Node('+', E_1.node, T.node)$ |
| 2) | $E \rightarrow E_1 - T$ | $E.node = \textbf{new}\ Node('-', E_1.node, T.node)$ |
| 3) | $E \rightarrow T$ | $E.node = T.node$ |
| 4) | $T \rightarrow (\ E\ )$ | $T.node = E.node$ |
| 5) | $T \rightarrow \textbf{id}$ | $T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$ |
| 6) | $T \rightarrow \textbf{num}$ | $T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$ |

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**DEPARTMENT OF CS, CA & IT**
**SUBJECT: SYSTEM PROGRAMMING**
**SUBJECT CODE: 16CSU601B**
**MULTIPLE CHOICE QUESTIONS**
**UNIT-3**

| sno | Questions | opt1 | opt2 |
|---|---|---|---|
| 1 | _______ is considered as a sequence of characters in a token. | Texeme | Pattern |
| 2 | What is the name of the process that determining whether a string of tokens can be generated by a grammar? | Analysing | Recognizing |
| 3 | A _________ is a software utility that translates code written in higher language into a low level language. | Converter | Compiler |
| 4 | Which of the following derivations does a top-down parser use while parsing an input string? | Leftmost derivation | Leftmost derivation in reverse |
| 5 | | | |
| 6 | The process of assigning load addresses to the various parts of the program and adjusting the code and data in the program to reflect the assigned addresses is called _______________ | Assembly | Parsing |
| 7 | . Which of the following statements is false? | Left as well as right most derivations can be in Unambiguous grammar | An LL (1) parser is a top-down parser |
| 8 | Given the following expression grammar:E -> E * F \| F+E \| F<br>F -> F-F \| id which of the following is true? | * has higher precedence than + | – has higher precedence than * |
| 9 | Which of the following grammers are not phase-structured? | regular | context free grammer |
| 10 | LR stands for ____________ | left to right | left to right reduction |
| 11 | Which of thefoloowing parsersare more powefull | linear list | search tree |
| 12 | Which of the following cannot be used as intermediate form? | Postfix notation | Three address code |

| | | | |
|---|---|---|---|
| 13 | Which of the following symbol table implementation is based on ptoperty of locality of reference | | search tree |
| 14 | Synthenized attribute can be easily simulated by __________ | LL grammer | ambiguious grammer |
| 15 | A pictorial representation of value computed by each statement in basic block is ___________ | tree | DAG |
| 16 | Three address code involves ___________ | excatly 3 address | at the most 3 address |
| 17 | When is type checking is done ? | during syntax directed translation | during lexical analysis |
| 18 | Which of the following is/are grouped together into sematic structures? | Syntax analyzer | Semantic analyzer |
| 19 | Which of the following describes a handle (as applicable to LR-parsing) appropriately? | It is the position in a sentential form where the next shift or reduce operation will occur | It is non-terminal whose production will be used for reduction in the next step |
| 20 | Which one of the following is a top-down parser? | Recursive descent parser | Operator precedence parse |
| 21 | Which of the following suffices to convert an arbitrary CFG to an LL(1) grammar? | Removing left recursion alone | Factoring the grammar alone |
| 22 | In a bottom-up evaluation of a syntax directed definition, inherited attributes can | always be evaluated | be evaluated only if the definition is L--attributed |
| 23 | Consider the grammar shown below. S → C C C → c C \| d The grammar is | LL(1) | SLR(1) but not LL(1) |
| 24 | Which of the following statements is false? | An unambiguous grammar has same leftmost and rightmost derivation | An LL(1) parser is a top-down parser |
| 25 | Which one of the following is True at any valid state in shift-reduce parsing? | Viable prefixes appear only at the bottom of the stack and not inside | Viable prefixes appear only at the top of the stack and not inside |

| | | | |
|---|---|---|---|
| 26 | In the context of abstract-syntax-tree (AST) and control-flow-graph (CFG), which one of the following is True? | In both AST and CFG, let node N2 be the successor of node N1. In the input program, the code corresponding to N2 is present after the code corresponding to N1 | For any input program, neither AST nor CFG will contain a cycle |
| 27 | Some code optimizations are carried out on the intermediate code because ________ | they enhance the portability of the compiler to other target processors | program analysis is more accurate on intermediate code than on machine code |
| 28 | One of the purposes of using intermediate code in compilers is to ________ | make parsing and semantic analysis simpler | improve error recovery and error reporting |
| 29 | What is the maximum number of reduce moves that can be taken by a bottom-up parser for a grammar with no epsilon- and unit-production (i.e., of type A -> ϵ and A -> a) to parse a string with n tokens? | n/2 | n-1 |
| 30 | The grammar $S \rightarrow aSa \mid bS \mid c$ is | LL(1) but not LR(1) | LR(1)but not LL(1) |
| 31 | For predictive parsing the grammar A->AA I (A) I ε is not suitable because | The grammar is right recursive | The grammar is left recursive |
| 32 | How many tokens are there in the following C statement?<br>printf (“j=%d, &j=%x”, j&j) | 4 | 5 |
| 33 | In a compiler, the data structure responsible for the management of information about variables and their attributes is | Semantic stack | Parser table |
| 34 | One of the purposes of using intermediate code in compilers is to | make parsing and semantic analysis simpler. | improve error recovery and error reporting |

| | | | |
|---|---|---|---|
| 35 | Syntax directed translation scheme is desirable because | It is based on the syntax | Its description is independent of any implementation |
| 36 | A top down parser generates | Right most derivation | Right most derivation in reverse |
| 37 | Intermediate code generation phase gets input from | Lexical analyzer | Syntax analyzer |
| 38 | An intermediate code form is | Postfix notation | Syntax trees |
| 39 | Input to code generator | Source code | Intermediate code |
| 40 | A grammar is meaningless | If terminal set and non terminal set are not disjoint | If left hand side of a production is a single terminal |
| 41 | Pee hole optimization | Loop optimization | Local optimization |
| 42 | Which is not true about syntax and semantic parts of a computer language | syntax is generally checked by the programmer | semantics is the responsibility of the programmer |
| 43 | Which of the following grammers are not phase structured ? | regular | context free gramm |
| 44 | Any syntactic constrct that can be described by a regular expression can also be described by a _________ | context sensitive grammar | non-context free grammar |
| 45 | In which addressing mode, the operand is given explicitly in the instruction itself? | absolute mode | immediate mode |
| 46 | YACC stands for __________________ | yet accept compiler constructs | .yet accept compiler compiler |
| 47 | An ideal computer should a) be small in size b) produce object code that is smaller in size and executes into tokens in a compiler | parser | cose optimize |
| 48 | A lex program consists of ___________ | declarations | .auxillary procedure |
| 49 | Which of the following pairs is the most powerful? | SLR, LALR | Canonical LR ,LALR |
| 50 | Which phase of compiler is Syntax Analysis? | First | Second |
| 51 | What is Syntax Analyser also known as ? | Hierarchical Analysis | Hierarchical Parsing |
| 52 | Syntax Analyser takes Groups Tokens of source Program into Grammatical Production | TRUE | FALSE |
| 53 | Parsers are expected to parse the whole code | TRUE | FALSE |

| | | | |
|---|---|---|---|
| 54 | A grammar for a programming language is a formal description of __________ | Syntax | Semantics |
| 55 | An LR-parser can detect a syntactic error as soon as __________ | The parsing starts | It is possible to do so a left-to-right scan of the input. |
| 56 | Which of the following is incorrect for the actions of A LR-Parser I) shift s ii) reduce A->ß iii) Accept iv) reject? | Only I) | I) and ii) |
| 57 | If a state does not know whether it will make a shift operation or reduction for a terminal is called | Shift/reduce conflict | Reduce /shift conflict |
| 58 | When there is a reduce/reduce conflict? | If a state does not know whether it will make a shift operation using the production rule i or j for a terminal. | If a state does not know whether it will make a shift or reduction operation using the production rule i or j for a terminal. |
| 59 | Which of these is also known as look-head LR parser? | SLR | LR |
| 60 | What is the similarity between LR, LALR and SLR? | Use same algorithm, but different parsing table | Same parsing table, but different algorithm. |

| opt3 | opt4 | | Answer |
|---|---|---|---|
| Lexeme | Mexeme | | Lexeme |
| Translating | Parsing | | Parsing |
| Text editor | Code optimizer | | Compiler |
| Rightmost derivation | Rightmost derivation in reverse | | Leftmost derivation |
| | | | |
| Relocation | Symbol resolute | | Relocation |
| LALR is more powerful than SLR | Ambiguous grammar can't be LR (k) | | Left as well as right most derivations can be in Unambiguous grammar |
| + and — have same precedence | + has higher precedence than * | | – has higher precedence than * |
| context senstive | none | | none |
| right to left | left to right and right most derivation in reverse | | left to right and right most derivation in reverse |
| hash table | self-organizing list | | self-organizing list |
| Syntax trees | qudraples | | qudraples |

| | | | |
|---|---|---|---|
| hash table | self-organizing list | | self-organizing list |
| LR grammer | RR grammer | | LR grammer |
| Graph | none | | DAG |
| no unary operators | none | | at the most 3 address |
| during syntax analysis | during code optimization | | during syntax directed translation |
| Lexical analyzer | Intermediate code generation | | Lexical analyzer |
| It is a production that may be used for reduction in a future step along with a position in the sentential form where the next shift or reduce operation will occur | It is the production p that will be used for reduction in the next step along with a position in the sentential form where the right hand side of the production may be found | | It is the production p that will be used for reduction in the next step along with a position in the sentential form where the right hand side of the production may be found |
| An LR(k) parser | An LALR(k) parser | | Recursive descent parser |
| Removing left recursion and factoring the grammar | None of these | | None of these |
| be evaluated only if the definition has synthesized attributes | never be evaluated | | be evaluated only if the definition is L--attributed |
| LALR(1) but not SLR(1) | LR(1) but not LALR(1) | | LL(1) |
| LALR is more powerful than SLR | An ambiguous grammar can never be LR(k) for any k | | An unambiguous grammar has same leftmost and rightmost derivation |
| The stack contains only a set of viable prefixes | The stack never contains viable prefixes | | The stack contains only a set of viable prefixes |

| | | | |
|---|---|---|---|
| The maximum number of successors of a node in an AST and a CFG depends on the input program | Each node in AST and CFG corresponds to at most one statement in the input program | | The maximum number of successors of a node in an AST and a CFG depends on the input program |
| the information from dataflow analysis cannot otherwise be used for optimization | the information from the front end cannot otherwise be used for optimization | | they enhance the portability of the compiler to other target processors |
| increase the chances of reusing the machine-independent code optimizer in other compilers | improve the register allocation | | increase the chances of reusing the machine-independent code optimizer in other compilers |
| 2n-1 | 2n | | n-1 |
| Both LL(1)and LR(1) | Neither LL(1)nor LR(1 | | Both LL(1)and LR(1) |
| The grammar is ambiguous | The grammar is an operator grammar | | The grammar is left recursive |
| 9 | 10 | | 10 |
| Symbol table | Abstract syntax-tree | | Symbol table |
| increase the chances of reusing the machine-independent code optimizer in other compilers. | improve the register allocation. | | increase the chances of reusing the machine-independent code optimizer in other compilers. |

| | | | | |
|---|---|---|---|---|
| It is easy to modify | All of these | | It is easy to modify | |
| Left most derivation | Left most derivation in reverse | | Left most derivation | |
| Semantic analyzer | Error handling | | Semantic analyzer | |
| Three address code | All of these | | All of these | |
| Target code | All of the above | | Intermediate code | |
| If left hand side of a production has no non terminal | All of these | | If terminal set and non terminal set are not disjoint | |
| Constant folding | Data flow analysis | | Constant folding | |
| semantics is checeked mechanically by a computer | both (b) and (c) | | both (b) and (c) | |
| context sensitive | none of these | | none of these | |
| context free grammar | none of these | | context free grammar | |
| .indirect mode | index mode | | immediate mode | |
| yet another compiler constructs | yet another compiler compiler | | yet another compiler compiler | |
| .code generator | .scanner | | .scanner | |
| .translation rules | .all of these | | .all of these | |
| SLR canonical LR | LALR canonical LR | | SLR canonical LR | |
| Third | Fourth | | Second | |
| None of the mentioned | Hierarchical Analysis & Parsing | | Hierarchical Analysis & Parsing | |
| NULL | None | | TRUE | |
| NULL | None | | TRUE | |

| | | | |
|---|---|---|---|
| Structure | Library | | Structure |
| It is possible to do so a right-to-left scan of the input. | Parsing ends | | It is possible to do so a left-to-right scan of the input. |
| I), ii) and iii) | I), ii) , iii) and iv) | | I), ii) and iii) |
| Shift conflict | Reduce conflict | | Shift/reduce conflict |
| If a state does not know whether it will make a reduction operation using the production rule i or j for a terminal. | None of the mentioned | | If a state does not know whether it will make a reduction operation using the production rule i or j for a terminal. |
| LLR | None | | LLR |
| Their Parsing tables and algorithm are similar but uses top down approach. | Both Parsing tables and algorithm are different. | | Use same algorithm, but different parsing table |

# UNIT -4

# SYLLABUS

**Storage organization:** Activation records stack allocation.

## RUNTIME ENVIRONMENT

A program as a source code is merely a collection of text (code, statements etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions. A program contains names for procedures, identifiers etc., that require mapping with the actual memory location at runtime.

By runtime, we mean a program in execution. Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.

Runtime support system is a package, mostly generated with the executable program itself and facilitates the process communication between the process and the runtime environment. It takes care of memory allocation and de-allocation while the program is being executed.

- Runtime organization of different storage locations
- Representation of scopes and extents during program execution.
- Components of executing program reside in blocks of memory (supplied by OS).
- Three kinds of entities that need to be managed at runtime (code, variables and procedures)

  Generated code for various procedures and programs. forms text or code segment of your program: size known at compile time.

  Data objects:

  - Global variables/constants: size known at compile time
  - Variables declared within procedures/blocks: size known
  - Variables created dynamically: size unknown.

Stack to keep track of procedure activations.

Subdivide memory conceptually into code (program) and data areas

**STATIC VERSUS DYNAMIC STORAGE ALLOCATION**

- Much (often most) data cannot be statically allocated. Either its size is not known at compile time or its lifetime is only a subset of the program's execution.
- Early versions of Fortranused only statically allocated data. This required that each array had a constant size specified in the program. Another consequence of supporting only static allocation was that recursion was forbidden (otherwise the compiler could not tell how many versions of a variable would be needed).
- Modern languages, including newer versions of Fortran, support both static and dynamic allocation of memory.
- The advantage supporting dynamic storage allocation is the increased flexibility and storage efficiency possible (instead of declaring an array to have a size adequate for the largest data set; just allocate what is needed). The advantage of static storage allocation is that it avoids the runtime costs for allocation/deallocation and may permit faster code sequences for referencing the data.
- An (unfortunately, all too common) error is a so-called memory leak where a long running program repeated allocates memory that it fails to delete, even after it can no longer be referenced. To avoid memory leaks and ease programming, several programming language systems employ automatic garbage collection. That means the runtime system itself can determine if data can no longer be referenced and if so automatically deallocates it.

**STORAGE ALLOCATION:**

• Compiler must do the storage allocation and provide access to variables and data

• Memory management

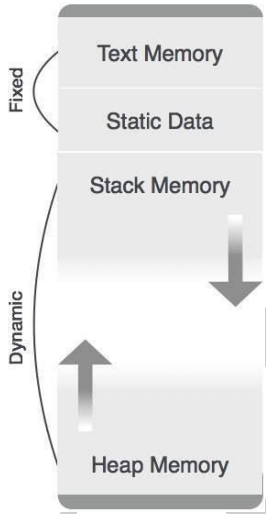
- ✓ Stack allocation
- ✓ Heap management
- ✓ Garbage collection

**Storage Allocation Strategies**

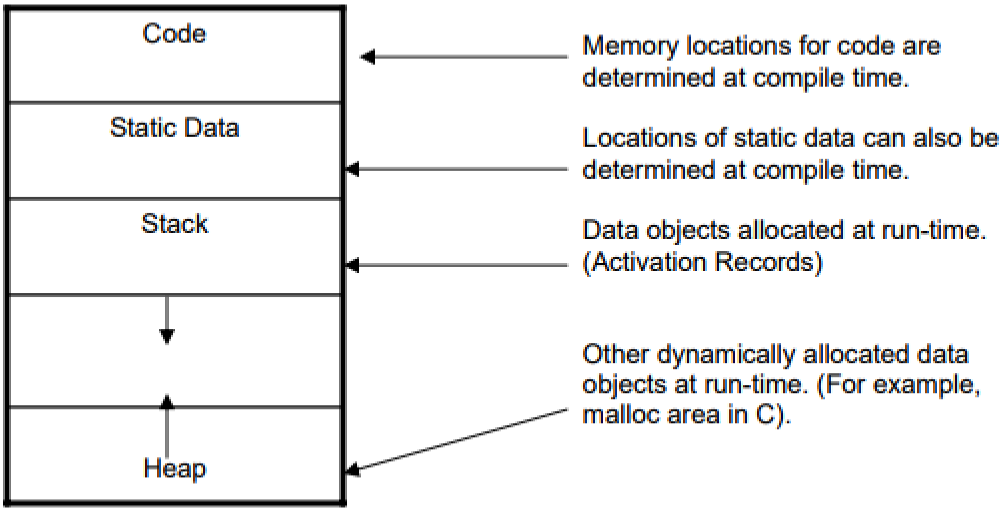
• Static allocation (Code): lays out storage at compile time for all data objects

• Stack allocation(Procedures): manages the runtime storage as a stack

• Heap allocation (Variables): allocates and deallocates storage as needed at runtime from heap




## STORAGE ORGANIZATION



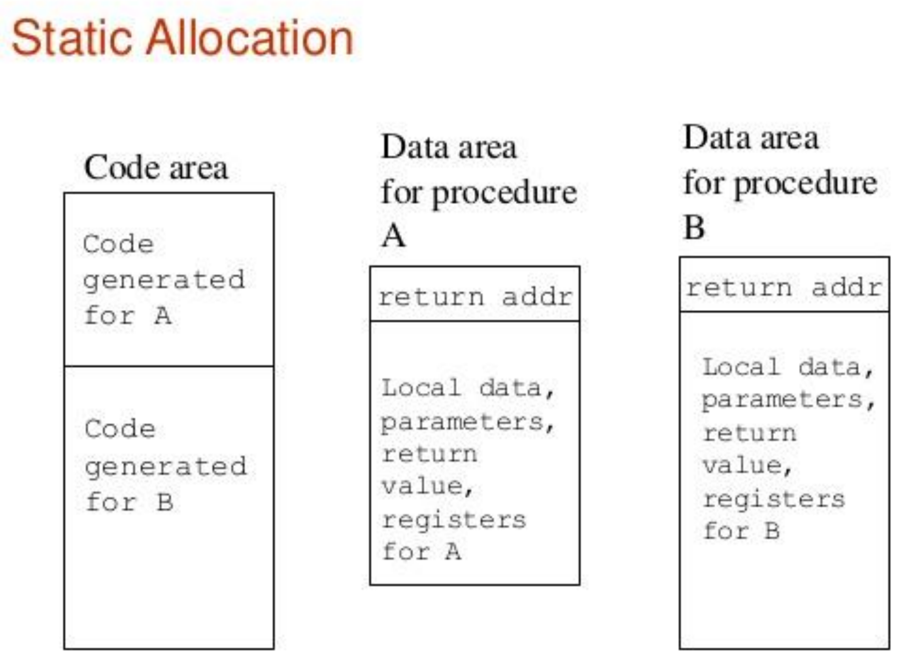

- Assumes a logical address space
  - Operating system will later map it to physical addresses, decide how touse cache memory, etc.
- Memory typically divided into areas for
  - Program code
  - Other static data storage, including global constants and compiler generated data
  - Stack to support call/return policy for procedures
  - Heap to store data that can outlive a call to a procedure

**STATIC ALLOCATION**

Statically allocated names are bound to storage at compile time. Storage bindings of statically allocated names never change, so even if a name is local to a procedure, its name is always bound to the same storage. The compiler uses the type of a name (retrieved from the symbol table) to determine storage size required. The required number of bytes (possibly aligned) is set aside for the name.The address of the storage is fixed at compile time.

*Limitations:*

- ✓ The size required must be known at compile time.
- ✓ Recursive procedures cannot be implemented as all locals are statically allocated.
- ✓ No data structure can be created dynamically as all data is static.




**Stack-dynamic allocation**

- ✓ Storage is organized as a stack.
- ✓ Activation records are pushed and popped.
- ✓ Locals and parameters are contained in the activation records for the call.
- ✓ This means locals are bound to fresh storage on every call.

- ✓ If we have a stack growing downwards, we just need a stack_top pointer.
- ✓ To allocate a new activation record, we just increase stack_top.
- ✓ To deallocate an existing activation record, we just decrease stack_top.

**RUN-TIME STACK AND HEAP**

The STACK is used to store:

- Procedure activations.
- The status of the machine just before calling a procedure, so that the status can be restored when the called procedure returns.
- The HEAP stores data allocated under program control (e.g. by malloc() in C).

**ACTIVATION RECORDS**

Any information needed for a single activation of a procedure is stored in the ACTIVATION RECORD (sometimes called the STACK FRAME). Today, we'll assume the stack grows DOWNWARD, as on, e.g., the Intel architecture. The activation record gets pushed for each procedure call and popped for each procedure return.

A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure**.**

Each time the flow of control enters a function or procedure, we update its procedure activation record. This maintains the values of the function arguments and all local variables defined inside the function, and pointers to the start of the code segment, the current location in the code segment, and the segment of code to which we return on exit.

Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.

We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the **activation tree**.

**Example:**

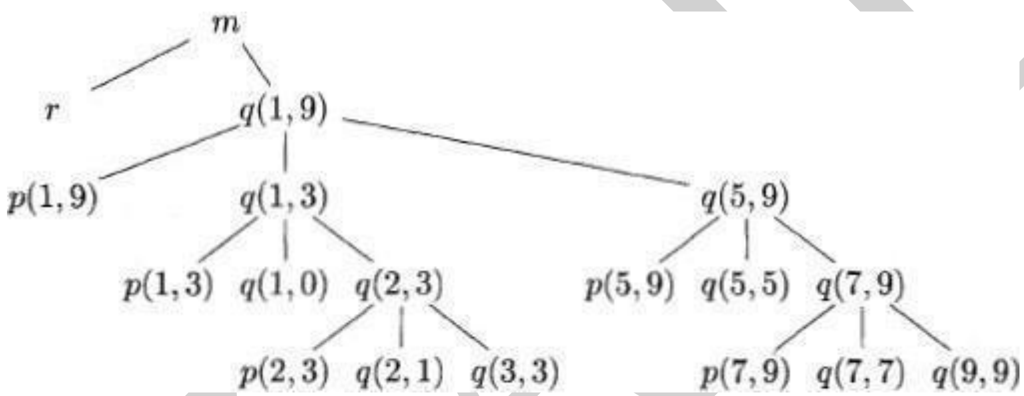
Consider the quick sort program

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

**Activation for Quicksort:**

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

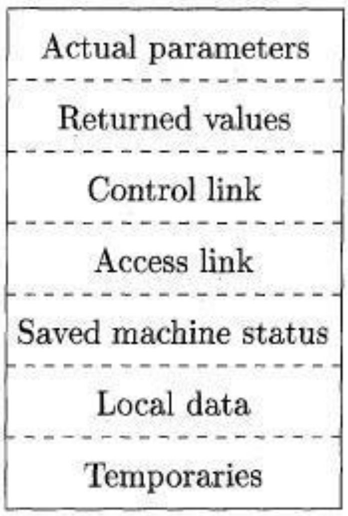
**Activation tree representing calls during an execution of quicksort:**




**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 (sometimes called a frame)
- ✓ The root of activation tree is at the bottom of the stack
- ✓ The current execution path specifies the content of the stack with the last
- ✓ Activation has record in the top of the stack.

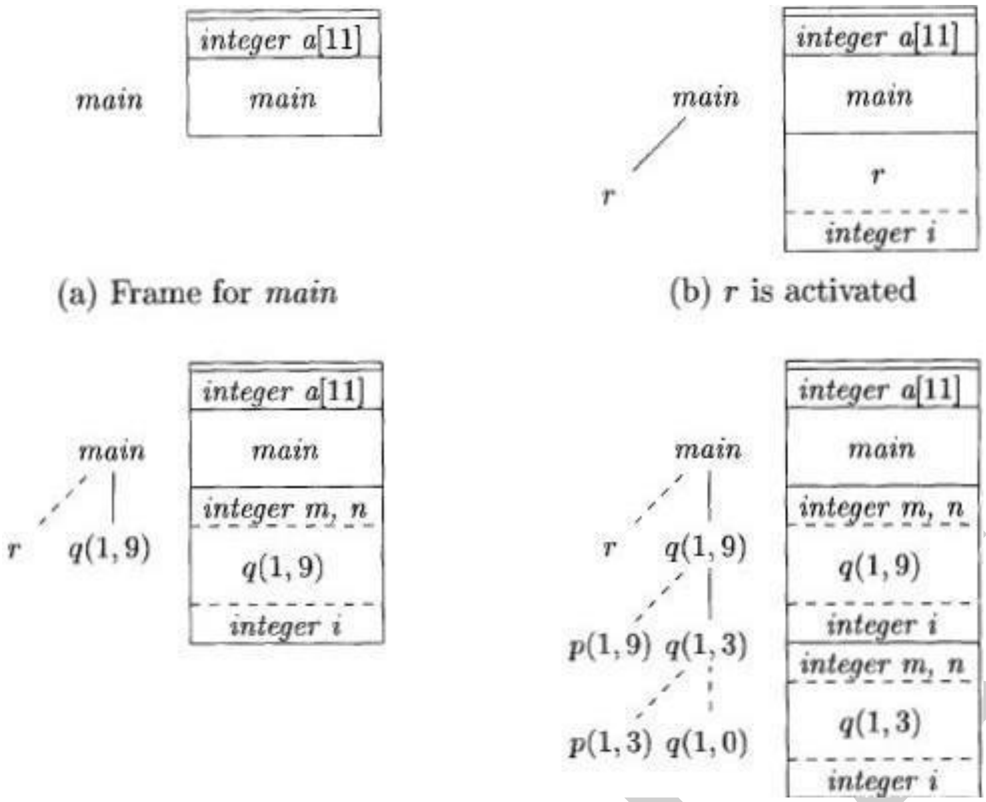
**A General Activation Record**




**Activation Record**

| **Fields** | **Elements** |
|---|---|
| Temporaries | Stores temporary and intermediate values of an expression. |
| Local Data | Stores local data of the called procedure. |
| Machine Status | Stores machine status such as Registers, Program Counter etc., before the procedure is called. |
| Control Link | Stores the address of activation record of the caller procedure. |
| Access Link | Stores the information of data which is outside the local scope. |
| Actual Parameters | Stores actual parameters, i.e., parameters which are used to send input to the called procedure. |
| Return Value | Stores return values. |

**Downward-growing stack of activation records:**




(a) Frame for *main*

(b) *r* is activated

**Address generation in stack allocation**

The position of the activation record on the stack cannot be determined statically. Therefore the compiler must generate addresses RELATIVE to the activation record. If we have a downward-growing stack and a stack_top pointer, we generate addresses of the form stack_top + offset

**HEAP ALLOCATION**

Some languages do not have tree-structured allocations. In these cases, activations have to be allocated on the heap. This allows strange situations, like callee activations that live longer than their callers' activations. This is not common Heap is used for allocating space for objects created at run timeFor example: nodes of dynamic data structures such as linked lists and trees

- Dynamic memory allocation and deallocation based on the requirements of the programmalloc() and free() in C programs
    - new()and delete()in C++ programs
    - new()and garbage collection in Java programs

- Allocation and deallocation may be *completely manual* (C/C++), *semi-automatic*(Java), or *fully automatic* (Lisp)

**PARAMETERS PASSING**

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

r-value

The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

l-value

The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears at the left hand side of an assignment operator.

A language has first-class functions if functions can be declared within any scope passed as arguments to other functions returned as results of functions.

- In a language with first-class functions and static scope, a function value is generally represented by a closure.
- A pair consisting of a pointer to function code a pointer to an activation record.
- Passing functions as arguments is very useful in structuring of systems using upcalls

*Formal Parameters*

Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

*Actual Parameters*

Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address.

*Pass by Value*

In pass by value mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record. Formal parameters then hold the values passed by the calling procedure. If the values held by the formal parameters are changed, it should have no impact on the actual parameters.

*Pass by Reference*

In pass by reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location. Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value.

*Pass by Copy-restore*

This parameter passing mechanism works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters.

*Pass by Name*

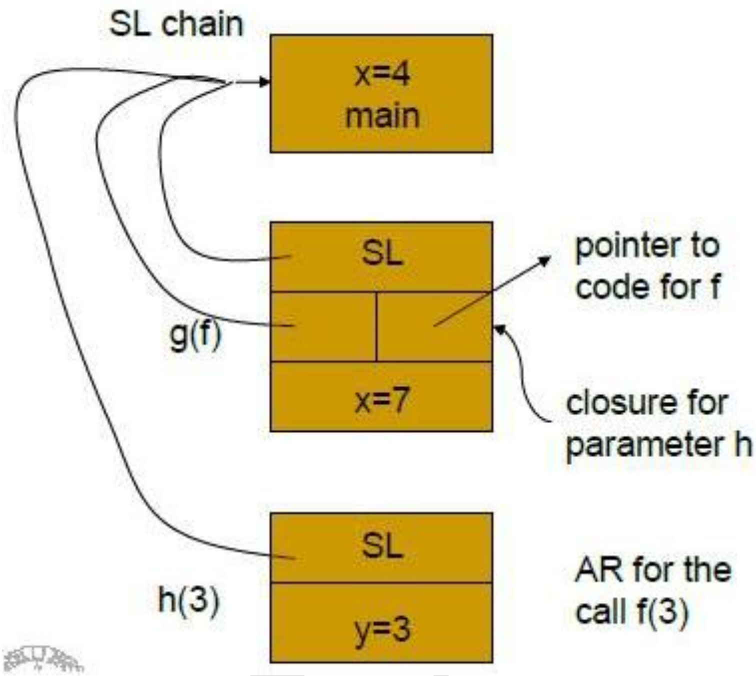
Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference.

An example:
```
main()
{ int x = 4;
int f (int y) {
return x*y;
}
int g (int →int h){
int x = 7;
return h(3) + x;
}

g(f);//returns 12

}
```

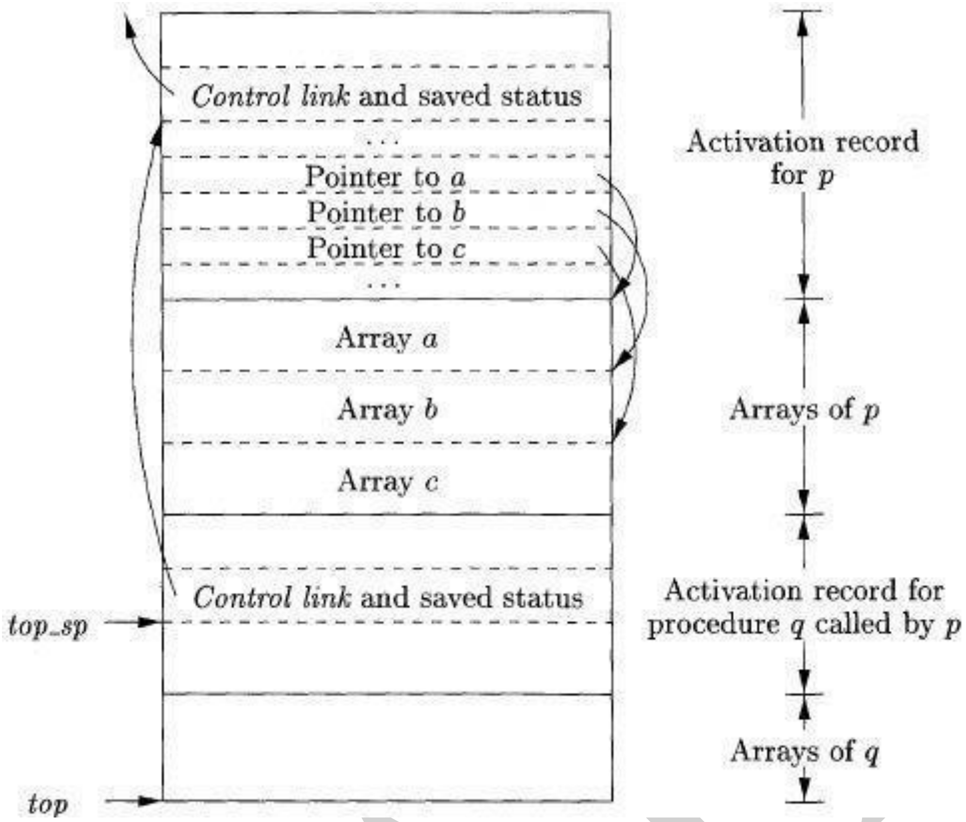
Passing Functions as Parameters – Implementation with Static Scope




**Designing Calling Sequences:**

- ✓ Values communicated between caller and callee are generally placed at the beginning of callee's activation record
- ✓ Fixed-length items: are generally placed at the middle
- ✓ Items whose size may not be known early enough: are placed at the end of activation record  We must locate the top-of-stack pointer judiciously: a common approach is to have it point to the end of fixed length fields

Access to dynamically allocated arrays:




Memory Manager:

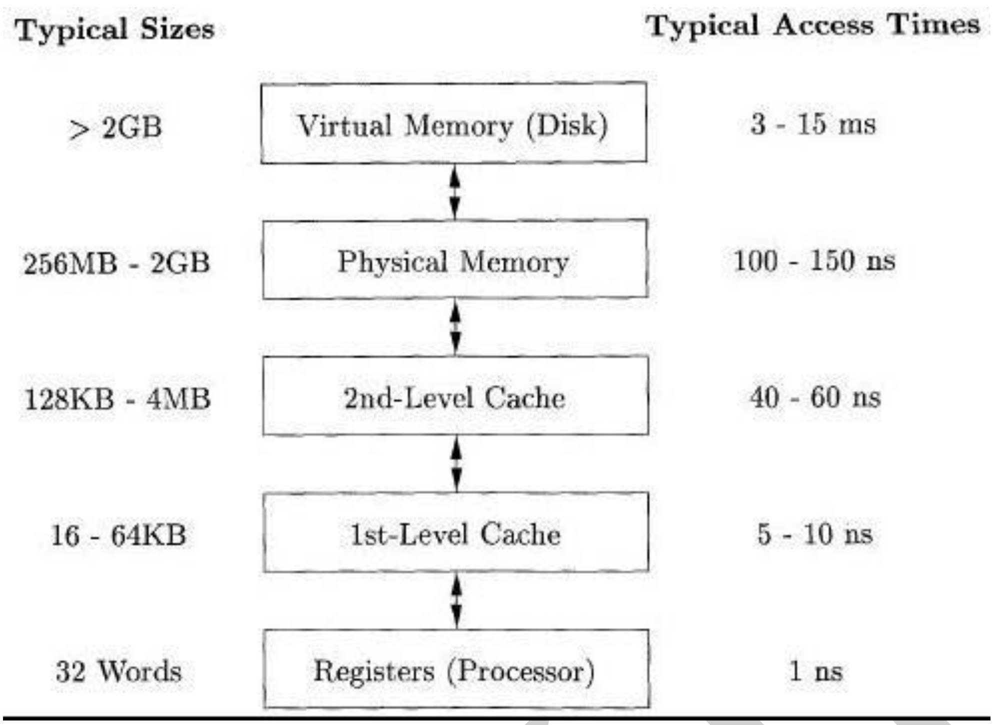
Two basic functions:

- ✓ Allocation
- ✓ Deallocation

Properties of memory managers:

- ✓ Space efficiency
- ✓ Program efficiency
- ✓ Low overhead

**Typical Memory Hierarchy Configurations**




**Locality in Programs:**

The conventional wisdom is that programs spend 90% of their time executing 10% of the code:

- ✓ Programs often contain many instructions that are never executed.
- ✓ Only a small fraction of the code that could be invoked is actually executed in typical run of the program.
- ✓ The typical program spends most of its time executing innermost loops and tight recursive cycles in a program.

**KARPAGAM ACADEMY OF HIGHER EDUCATION**
**DEPARTMENT OF CS, CA & IT**
**SUBJECT: SYSTEM PROGRAMMING**
**SUBJECT CODE: 16CSU601B**
**MULTIPLE CHOICE QUESTIONS**
**UNIT-4**

| sno | Questions | opt1 |
|---|---|---|
| 1 | The average time required to reach a storage location in memory and obtain its contents is called the _____ | Seek time |
| 2 | What characteristic of RAM memory makes it not suitable for permanent storage? | Too slow |
| 3 | Assembly language ________ | Uses alphabetic codes in place of binary numbers used in machine language |
| 4 | Select a Machine Independent phase of the compiler | Syntax Analysis |
| 5 | Which of the following system software resides in the main memory always | Text Editor |
| 6 | Which of these features of assembler are Machine-Dependent | Instruction formats |
| 7 | Which of these is not true about Symbol Table | All the labels of the instructions are symbols |
| 8 | In Reverse Polish notation, expression A*B+C*D is written as | AB*CD*+ |
| 9 | The circuit converting binary data in to decimal is_____ | Encoder |
| 10 | In computers, subtraction is carried out generally by____ | 1's complement method |
| 11 | The identification of common sub-expression and replacement of run-time computations by compile-time computations is | local optimization |
| 12 | The graph that shows basic blocks and their successor relationship is called | DAG |

| | | |
|---|---|---|
| 13 | The specific task storage manager performs | allocation/ deallocation of storage to programs |
| 14 | When a computer is first turned on or resrarted, a special type of absolute loader is executed called | " Compile and GO " loader |
| 15 | Function of the storage assignment is | assign storage to all variables referenced in the source program |
| 16 | Relocation bits used by relocating loader are specified by | relocating loader itself |
| 17 | Running time of a program depends on | the way the registers and addressing modes are use |
| 18 | Advantage of panic mode of error recovery is that | it is simple to implement |
| 19 | Which of the following can be accessed by transfer vector approach of linking ? | external data segments |
| 20 | Generation of intermediate code based on a abstract machine model is useful in compilers because | it makes implementation of lexical analysis and syntax analysis easier |
| 21 | Which of the following module does not incorporate initialization of values changed by the module ? | non reusable module |
| 22 | A self-relocating program is one which | cannot be made to execute in any area of storage other than the one designated for it at the time of its coding or translation |
| 23 | The string (a)\|((b)*(c)) is equivalent to | Empty |
| 24 | Which one of the following statements is FALSE ? | Context-free grammar can be used to specify both lexical and syntax rules. |
| 25 | Some code optimizations are carried out on the intermediate code because | they enhance the portability of the compiler to other target processors |

| | | |
|---|---|---|
| 26 | A non relocatable program is the one which | cannot be made to execute in any area of storage other than the one designated for it at the time of its coding or translation |
| 27 | A relocatable program form is one which | cannot be made to execute in any area of storage other than the one designated for it at the time of its coding or translation |
| 28 | A self-relocating program is one which | cannot be made to execute in any area of storage other than the one designated for it at the time of its coding or translation |
| 29 | In which storage allocation stragey size is required at compile time | static allocation |
| 30 | Which field is not present in activation record | saved machine status |
| 31 | Which of the following are activation records? | return value |
| 32 | which of the following are storage allocation stratagey | stack allocation |
| 33 | ____________ tree is used to depict the way control enters and leaves activation | Activation tree |
| 34 | In activation tree each node represent | activation of main program |
| 35 | _________ can be used to keep track of live procedures activations | control stack |
| 36 | if the occurance of the name in the procedure is in the scope of declaration within the procedure then it is said to be | local |
| 37 | subdivision of runtime memory consists of | code |
| 38 | In activation record, optional control link points to | activation record of caller |
| 39 | The field of actual parameter in activation record is used by which procedure? | calling procedure |
| 40 | Allocation of activation record and entering information into fields is done by | return sequence |
| 41 | call by reference is also called as | call-by-address |

| | | |
|---|---|---|
| 42 | In which allocation, names are bound to storage as program is compiled? | static |
| 43 | Flow of control in a program corresponds to which traversal of activation tree ? | Depth first traversal |
| 44 | Which is the correct sequence of compilation process? | Assembler → Compiler → Preprocessor → Linking |
| 45 | Why is calloc() function used for? | allocates the specified number of bytes |
| 46 | The instruction 'ORG O' is a______ | Machine Instruction |
| 47 | Memory unit accessed by content is called______ | Read only memory |
| 48 | ________ register keeps tracks of the instructions stored in program stored in memory. | AR (Address Register) |
| 49 | The circuit converting binary data in to decimal is_____ | Encoder |
| 50 | In computers, subtraction is generally done by _________ | 1's complement method |
| 51 | PSW is saved in stack when there is a _____ | Interrupt recognized |
| 52 | Memory unit accessed by content is called______ | Read only memory |
| 53 | 'Aging registers' are _______ | Counters which indicate how long ago their associated pages have been Referenced. |
| 54 | The size of the activation record can be determined at ____________ | Run time |
| 55 | Which of the following are parameter passing method | call by value |
| 56 | Which one of the following statement is false for the SLR (1) and LALR (1) parsing tables for a context free grammar? | The reduce entries in both the tables may be different |

| opt2 | opt3 | opt4 |
|---|---|---|
| Turn around time | transfer time | access time |
| Unreliable | It is volatile | Too bulky |
| Is the easiest language to write programs | Need not be translated into machine language | None of the mentioned |
| Intermediate Code generation | Lexical analysis | all the above |
| Assembler | Linker | Loader |
| Addressing modes | Program relocation | All of the mentioned |
| Table has entry for symbol name address value | Perform the processing of the assembler directives | Created during pass 1 |
| A*BCD*+ | AB*CD+* | A*B*CD+ |
| Multiplexer | Decoder | Code converter |
| 2’s complement method | signed magnitude method | BCD subtraction method |
| loop optimization | constant folding | data flow analysis |
| Hamiltonion graph | Flow graph | control graph |

| | | |
|---|---|---|
| protection of storage area allocated to a program from illegal access by othere programs in the system | the status of each program | both ( a ) and ( b ) |
| Boot loader | Boot strap loader | Relating loader |
| assign storage to all temporary locations that are necessary for intermediate results | assign storage to literals, and to ensure that the storage is allocated and appropriate locations are initialized | all of these |
| linker | assembler | macro processor |
| the order in which computations are performed | the usage of machine idioms | all of these |
| it never gets into an infinite loop | both (a) and (b) | none of these |
| external sub-routines | data located in other procedure | all of these |
| syntax directed translations can be written for intermediate code generation | it enhances the portability of the front end of the compiler | it is not possible to generate code for real machines directly from high level language programs |
| serially reusable module | re-enterable module | all of these |
| consists of a program and relevant information for its relocation | can itself perform the relocation of its address sensitive portions | all of these |
| abcabc | b*c\|a | None of the mentioned |
| Type checking is done before parsing. | High-level language programs can be translated to different Intermediate Representations. | Arguments to a function can be passed using the program stack. |
| program analysis is more accurate on intermediate code than on machine code | the information from dataflow analysis cannot otherwise be used for optimization | the information from the front end cannot otherwise be used for optimization |

| | | |
|---|---|---|
| consists of a program and relevant information for its relocation | can itself perform the relocation of its address sensitive portions | all of these |
| consists of a program and relevant information for its relocation | can be processed to relocate it to a desired area of memory | all of these |
| consists of a program and relevant information for its relocation | can itself perform the relocation of its address sensitive portions | all of these |
| dynamic allocation | stack allocation | all |
| register allocation | optional control link | temporaries |
| local data | temporaries | all |
| static allocation | heap allocation | all |
| tree | parse tree | none |
| activaiton of procedure | both (a) and (b) | none |
| activation tree | activation node | none |
| nonlocal | global | none |
| static data | stack | all |
| activation record of callee | both (a) and (b) | none |
| called procedure | both (a) and (b) | none |
| call sequence | both (a) and (b) | none |
| call-by-location | both (a) and (b) | none |

| | | |
|---|---|---|
| heap | stack | none |
| Breadth first traversal | both (a) and (b) | none |
| Compiler → Assenbler → Preprocessor → Linking | Preprocessor → Compiler → Assembler → Linking | Assembler → Compiler → Linking → Preprocessor |
| | increases or decreases the size of the specified block of memory and reallocates it if needed | calls the specified block of memory for execution. |
| Pseudo instruction | High level instruction | Memory instruction |
| Programmable Memory | Virtual Memory | Associative Memory |
| XR (Index Register) | PC (Program Counter) | AC (Accumulator) |
| Multiplexer | Decoder | Code converter |
| 2's complement method | BCD subtraction method | signed magnitude method |
| Execution of RST instruction | Execution of CALL instruction | All of these |
| Programmable Memory | Virtual Memory | Associative Memory |
| Registers which keep track of when the program was last accessed | Counters to keep track of last accessed instruction | Counters to keep track of the latest data structures referred |
| Compile time | both (a) and (b) | none of these |
| call by reference | call by restore | all |
| The error entries in both the tables may be different | The go to part of both tables may be different | The shift entries in both the tables may be identical |

| | Answer |
|---|---|
| | access time |
| | It is volatile |
| | Uses alphabetic codes in place of binary numbers used in machine language |
| | all the above |
| | Loader |
| | All of the mentioned |
| | Perform the processing of the assembler directives |
| | AB*CD*+ |
| | Code converter |
| | 2's complement method |
| | constant folding |
| | Flow graph |

| | |
|---|---|
| | both ( a ) and ( b ) |
| | Boot strap loader |
| | all of these |
| | linker |
| | all of these |
| | both (a) and (b) |
| | external sub-routines |
| | it makes implementation of lexical analysis and syntax analysis easier |
| | non reusable module |
| | can itself perform the relocation of its address sensitive portions |
| | b*c\|a |
| | Type checking is done before parsing. |
| | they enhance the portability of the compiler to other target processors |

| | |
|---|---|
| | cannot be made to execute in any area of storage other than the one designated for it at the time of its coding or translation |
| | can be processed to relocate it to a desired area of memory |
| | can itself perform the relocation of its address sensitive portions |
| | static allocation |
| | register allocation |
| | all |
| | all |
| | Activation tree |
| | |
| | |
| | local |
| | stack |
| | activation record of caller |
| | calling procedure |
| | call sequence |
| | both (a) and (b) |

| | |
|---|---|
| | static |
| | Depth first traversal |
| | Preprocessor → Compiler → Assembler → Linking |
| | allocates the specified number of bytes and initializes them to zero |
| | Pseudo instruction |
| | Associative Memory |
| | PC (Program Counter) |
| | Code converter |
| | 2's complement method |
| | Interrupt recognized |
| | Associative Memory |
| | Counters which indicate how long ago their associated pages have been Referenced. |
| | both (a) and (b) |
| | all |
| | The go to part of both tables may be different |

**UNIT -5**

**SYLLABUS**

**Code Generation:** Object code generation

# CODE GENERATION

## Introduction

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

1. The term "code optimization" refers to techniques, a compiler can employ in an attempt to produce a better object language program than the most obvious for a given source program.
2. The quality of the object program is generally measured by its size (for small computation) or its running time (for large computation).
3. It is theoretically impassible for a compiler to produce the best possible object program for every source program under any reasonable cast function.
4. The accurate term for "code optimization" is "code improvement".
5. There are many aspects to code optimization.
   a. Cast
   b. Quick & straight forward translation (time).

**The Principal Sources Of Optimization**
It consists of detecting patterns in the program and replacing these patterns by equivalent and more efficient construct.

The optimization can be done by
1) Programmer- Write source program (user can write)
2) Compiler -e.g.: array references are made by indexing, rather than by pointer or address calculation prevents the programmer from dealing with offset calculations in arrays.

Example:
A [i+1]:=B [i+1] is easier.

J: =i+1
A[j]:=B[j]

I. Local optimization-performed within a straight line and no jump.
II. Loop optimization
III. Data flow analysis-the transmission of useful information from one part of the program to another.

**Optimization**

Optimization is a program transformation technique, which tries to improve the code by making it consume less resource (i.e. CPU, Memory) and deliver high speed. In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types : machine independent and machine dependent.

## What Is a Loop?

Before we discuss loop optimizations, we should discuss what we identify as a loop. In our source language, this is rather straightforward, since loops are formed with while or for, where it is convenient to just elaborate a for loop into its corresponding while form.

The key to a loop is a back edge in the control-flow graph from a node l to a node h that dominates l. We call h the header node of the loop. The loop itself then consists of the nodes on a path from h to l. It is convenient to organize the code so that a loop can be identified with its header node. We then write loop(h, l) if line l is in the loop with header h.

When loops are nested, we generally optimize the inner loops before the outer loops. For one, inner loops are likely to be executed more often. For another, it could move computation to an outer loop from which it is hoisted further when the outer loop is optimized and so on

# Machine-dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

For example:

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

This code involves repeated assignment of the identifier item, which if we put this way:

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

should not only save the CPU cycles, but can be used on any processor.

# Basic Blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

***Basic block identification***

We may use the following algorithm to find the basic blocks in a program:

- Search header statements of all the basic blocks from where a basic block starts:
    - First statement of a program.
    - Statements that are target of any branch (conditional/unconditional).

   - Statements that follow any branch statement.

- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

Basic blocks are important concepts from both code generation and optimization point of view.

```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
   y = x;
   x++;
  }
else
  {
   y = z;
   z++;
  }
w = x + z;
```

Source Code

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

```
w = x + z;
```

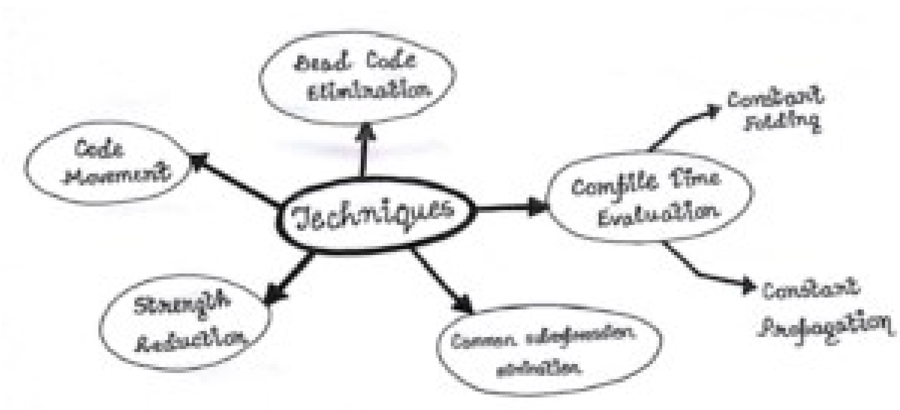
Basic Blocks

Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

**Advantages of Code Optimization-**

- Optimized code has faster execution speed
- Optimized code utilizes the memory efficiently
- Optimized code gives better performance

**Techniques for Code Optimization-**




1. Compile Time Evaluation
2. Common sub-expression elimination
3. Dead Code Elimination
4. Code Movement
5. Strength Reduction

**1.Compile Time Evaluation-**

Two techniques that falls under compile time evaluation are-

**A) Constant folding-**

As the name suggests, this technique involves folding the constants by evaluating the expressions that involves the operands having constant values at the compile time.

- **Example-**

Circumference of circle = (22/7) x Diameter

Here, this technique will evaluate the expression 22/7 and will replace it with its result 3.14 at the compile time which will save the time during the program execution.

**B) Constant Propagation-**

In this technique, if some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program wherever it has been used during compilation, provided that its value does not get alter in between.

- **Example-**

pi = 3.14

radius = 10

Area of circle = pi x radius x radius

Here, this technique will substitute the value of the variables ‘pi’ and ‘radius’ at the compile time and then it will evaluate the expression 3.14 x 10 x 10 at the compile time which will save the time during the program execution.

**2. Common sub-expression elimination-**

The expression which has been already computed before and appears again and again in the code for computation is known as a common sub-expression.

As the name suggests, this technique involves eliminating the redundant expressions to avoid their computation again and again. The already computed result is used in the further program wherever its required.

**Example-**

| Code before Optimization | Code after Optimization |
|---|---|
| S1 = 4 x i<br>S2 = a[S1]<br>S3 = 4 x j<br>S4 = 4 x i // **Redundant Expression**<br>S5 = n<br>S6 = b[S4] + S5 | S1 = 4 x i<br>S2 = a[S1]<br>S3 = 4 x j<br>S5 = n<br>S6 = b[S1] + S5 |

**3. Code Movement-**

As the name suggests, this technique involves the movement of the code where the code is moved out of the loop if it does not matter whether it is present inside the loop or it is present outside the loop.

Such a code unnecessarily gets executed again and again with each iteration of the loop, thus wasting the time during the program execution.

**Example-**

| Code before Optimization | Code after Optimization |
|---|---|

| | |
|---|---|
| for ( int j = 0 ; j < n ; j ++)<br>{<br>x = y + z ;<br>a[j] = 6 x j;<br>} | x = y + z ;<br>for ( int j = 0 ; j < n ; j ++)<br>{<br>a[j] = 6 x j;<br>} |

**4. Dead code elimination-**

As the name suggests, this technique involves eliminating the dead code where those statements from the code are eliminated which either never executes or are not reachable or even if they get execute, their output is never utilized.

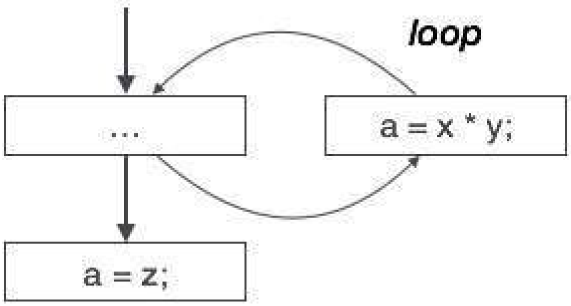
**Example-**

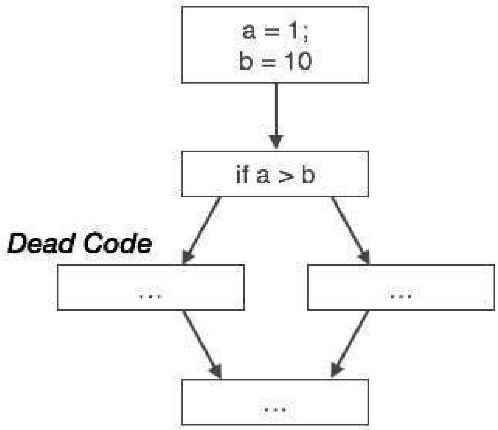
| Code before Optimization | Code after Optimization |
|---|---|
| i = 0 ;<br>if (i == 1)<br>{<br>a = x + 5 ;<br>} | i = 0 ; |

Dead code is one or more than one code statements, which are:

- Either never executed or unreachable,
- Or if executed, their output is never used.

Thus, dead code plays no role in any program operation and therefore it can simply be eliminated. **First** fig. depicts partial dead code, **second** fig. depicts complete dead code.




**5. Strength reduction-**

As the name suggests, this technique involves reducing the strength of the expressions by replacing the expensive and costly operators with the simple and cheaper ones.

**Example-**

| Code before Optimization | Code after Optimization |
|---|---|
| B = A x 2 | B = A + A |

Here, the expression "A x 2" has been replaced with the expression "A + A" because the cost of multiplication operator is higher than the cost of addition operator.

## Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

- **Invariant code**: A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.
- **Induction analysis**: A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.
- **Strength reduction**: There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

**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.

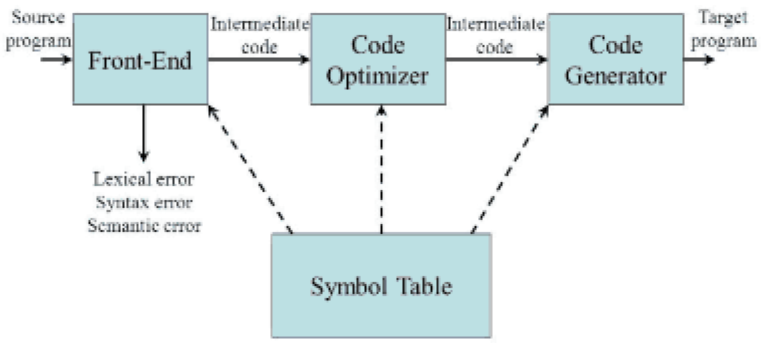
The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

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

## Code Generator

A code generator is expected to have an understanding of the target machine's runtime environment and its instruction set.




The code generator should take the following things into consideration to generate the code:

- **Target language** : The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.

- **IR Type**: Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.
- **Selection of instruction**: The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.
- **Register allocation**: A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.
- **Ordering of instructions**: At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

## Descriptors

The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code. For both of them, the following two descriptors are used:

- **Register descriptor**: Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.
- **Address descriptor**: Values of the names (identifiers) used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.

Code generator keeps both the descriptor updated in real-time. For a load statement, LD R1, x, the code generator:

- updates the Register Descriptor R1 that has value of x and
- updates the Address Descriptor (x) to show that one instance of x is in R1.

## Code Generation

Basic blocks comprise of a sequence of three-address instructions. Code generator takes these sequence of instructions as input.

**Note** : If the value of a name is found at more than one place (register, cache, or memory), the register's value will be preferred over the cache and main memory. Likewise cache's value will be preferred over the main memory. Main memory is barely given any preference.

**getReg** : Code generator uses *getReg* function to determine the status of available registers and the location of name values. *getReg* works as follows:

- If variable Y is already in register R, it uses that register.
- Else if some register R is available, it uses that register.
- Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions.

For an instruction x = y OP z, the code generator may perform the following actions. Let us assume that L is the location (preferably register) where the output of y OP z is to be saved:

- Call function getReg, to decide the location of L.
- Determine the present location (register or memory) of **y**by consulting the Address Descriptor of **y**. If **y** is not presently in register **L**, then generate the following instruction to copy the value of **y** to **L**:

  MOV y', L

  where **y'** represents the copied value of **y**.
- Determine the present location of **z** using the same method used in step 2 for **y** and generate the following instruction:

  OP z', L

  where **z'** represents the copied value of **z**.
- Now L contains the value of y OP z that is intended to be assigned to **x**. So, if L is a register, update its descriptor to indicate that it contains the value of **x**. Update the descriptor of **x** to indicate that it is stored at location **L**.
- If y and z has no further use, they can be given back to the system.

Other code constructs like loops and conditional statements are transformed into assembly language in general assembly way.

KARPAGAM ACADEMY OF HIGHER EDUCATION
DEPARTMENT OF CS, CA & IT
SUBJECT: SYSTEM PROGRAMMING
SUBJECT CODE: 16CSU601B
MULTIPLE CHOICE QUESTIONS
UNIT-5

| sno | Questions | opt1 | opt2 |
|---|---|---|---|
| 1 | Which of the following comment about peep-hole optimization is true? | they enhance the portability of the compiler to other target processors | program analysis is more accurate on intermediate code than on machine code |
| 2 | The method which merges the bodies of two loops is | Loop rolling | Loop jamming |
| 3 | Running time of a program depends on | Addressing mode | Order of computations |
| 4 | Which of the following is the fastest logic | TTL | ECL |
| 5 | he optimization which avoids test at every iteration is | Loop unrolling | Loop jamming |
| 6 | Scissoring enables | A part of data to be displayed | Entire data to be displayed |
| 7 | Compiler should report the presence of ________in source program, in translation process. | data | object |
| 8 | Compiler can check _______ error | syntax | logical |
| 9 | Reduction in strength means | Replacing run time computation by compile time computation | emoving loop invariant computation |
| 10 | A optimizing compiler | Is optimized to occupy less space | Is optimized to take less time for execution |
| 11 | Code can be optimized at | Source from user | Target code |
| 12 | In which way(s) a macroprocessor for assembly language can be implemented ? | Independent two-pass processor | Independent one-pass processor |

| | | | |
|---|---|---|---|
| 13 | A compiler for a high level language that runs on one machine and produce code for different machine is called | Optimizing compiler | One pass compiler |
| 14 | Local and loop optimization in turn provide motivation for | Data flow analysis | Constant folding |
| 15 | What is responsible for generation of final machine code tailored to target system? | Interpreter | Semantic analyzer |
| 16 | Which programming language use compiler as well as interpreter to produce output? | C language | C++ |
| 17 | Optimization of program that works within a single basic block is called | A. Local optimization | A. Global optimizatior |
| 18 | Variable that can be accessed through out program is known as | A. Local variable | A. Global Variable |
| 19 | Gcc level of procedure integration, can be calculated as | 1 | 2 |
| 20 | Which languages necessarily need heap allocation in the runtime environment? | Those that support recursion | Those that use dynamic scoping |
| 21 | Some code optimizations are carried out on the intermediate code because | They enhance the portability | Program analysis is mo |
| 22 | Input buffer is | symbol table | divided into two halves |
| 23 | Daisy chain is a device for ? | Interconnecting a number of devices to number of controllers | Connecting a number of devices to a controller |
| 24 | Which of the following type of software should be used if you need to create,edit and print document ? | Word processing | Spreadsheet |
| 25 | What is bootstraping? | A language interpreting other language program | A language compiling other language program |
| 26 | Shell is the exclusive feature of | UNIX | DOS |

| | | | |
|---|---|---|---|
| 27 | A program in execution is called | Process | Instruction |
| 28 | A UNIX device driver is | Structured into two halves called top half and bottom half | Three equal partitions |
| 29 | Memory | is an device that performs a sequence of operations specified by instructions in memory | is the device where information is stored |
| 30 | In what module multiple instances of execution will yield the same result even if one instance has not terminated before the next one has begun ? | Serially usable | Re-entrable module |
| 31 | The segment base is specified using the register named is | ORG instructions | TITLE instruction |
| 32 | If special forms are needed for printing the output, the programmer specifies these forms through | IPL | JCL |
| 33 | Register or main memory location which contains the effective address of the operand is known as | pointer | indexed register |
| 34 | Name given to the organized collection of software that controls the overall operation of a computer is | working system | peripheral system |
| 35 | Which of the following is not a function of the operating system? | Manage resources | Internet access |
| 36 | The items that a computer can use in its functioning are collectively called its | resources | stuff |
| 37 | Programs that coordinate all of the computer’s resources including memory, processing, storage, and devices such as printers are collectively referred to as | language translators | resources |

| | | | |
|---|---|---|---|
| 38 | A compiler is a software tool that translates ____________that the computer can understand. | Algorithm into data | Source code into data |
| 39 | The object code is then passed through a program called a ____________ which turns it into an executable program. | Integer | Source code |
| 40 | What is memory in Computer ? | is a sequence of instructions | is the device where information is stored |
| 41 | A program - | is a sequence of instructions | is the device where information is stored |
| 42 | The __ of a system includes the program s or instructions. | icon | software |
| 43 | System generation | is always quite simple | is always very difficult |
| 44 | While running DOS on a PC, command which can be used to duplicate the entire diskette is | COPY | DISKCOPY |
| 45 | Operating system for the laptop computer called MacLite is | windows | DOS |
| 46 | In computers, application software executes | all the time | when required |
| 47 | To perform specific tasks or calculations in the computer we use | system software | application software |
| 48 | Computer can run without | application software | system software |
| 49 | Computer software which is also known as Off-the-shelf software is | customized software | package software |
| 50 | The number of clock cycles necessary to complete 1 fetch cycle in 8085 is | 3 or 4 | 4 or 5 |

| | | | |
|---|---|---|---|
| 51 | Which of the following electronic component are not found in ordinary ICs? | Diodes | Resistors |
| 52 | The root directory of a disk should be placed | at a fixed address in main memory | at a fixed location on the disk |
| 53 | In a two pass assembler the object code generation is done during the ? | Second pass | First pass |
| 54 | Which of the following is not a type of assembler ? | one pass | two pass |
| 55 | In a two pass assembler, adding literals to literal table and address resolution of local symbols are done using ? | First pass and second respectively | Both second pass |
| 56 | In a two pass assembler the pseudo code EQU is to be evaluated during ? | Pass 1 | Pass 2 |

| opt3 | opt4 | | Answer |
|---|---|---|---|
| the information from dataflow analysis cannot otherwise be used for optimization | the information from the front end cannot otherwise be used for optimization | | they enhance the portability of the compiler to other target processors |
| Constant folding | None of the mentioned | | Loop jamming |
| The usage of machine idioms | All of the mentioned | | All of the mentioned |
| CMOS | LSI | | ECL |
| Constant folding | None of the mentioned | | Loop unrolling |
| None of the mentioned | No data to be displayed | | A part of data to be displayed |
| errors | text | | errors |
| content | both a and b | | syntax |
| Removing common sub expression | Replacing a costly operation by a relatively cheaper one | | Replacing run time computation by compile time computation |
| Optimized the code | None of the above. | | Optimized the code |
| Intermediate code | All of the above | | Source from user |
| Expand macrocalls and substitute arguments | All of the above | | All of the above |

| | | | |
|---|---|---|---|
| Cross compiler | Multipass compiler | | Cross compiler |
| Pee hole optimization | DFA and constant folding | | Data flow analysis |
| Code generator | Code optimizer | | Code generator |
| Java | Cobol | | Java |
| A. Loop un-controlling | A. Loop controlling | | A. Local optimization |
| A. Integer | A. Constant | | A. Global Variable |
| 3 | 4 | | 3 |
| hose that allow dynamic data structures | Those that use global variables | | hose that allow dynamic data structures |
| The information from | The information from the front end cannot otherwise be used for optimization | | They enhance the portability of the compiler to other target processors |
| divided into Three halves | not divided | | divided into two halves |
| Connecting a number of controller to devices | all of these | | Connecting a number of devices to a controller |
| Desktop publishing | UNIX | | Word processing |
| A language compile itself | All of above | | A language compile itself |
| System software | Application software | | UNIX |

| | | | |
|---|---|---|---|
| Procedure | Function | | Process |
| Unstructured | None of the above | | Structured into two halves called top half and bottom half |
| is a sequence of instructions | is a computational unit to perform specific functions | | is the device where information is stored |
| Non reusable module | None of these | | Re-entrable module |
| ASSUME instruction | SEGMENT instruction | | ORG instructions |
| Load modules | Utility programs | | JCL |
| special location | scratch pad | | pointer |
| operating system | controlling system | | operating system |
| Provide a user interface | Load and run applications | | Internet access |
| capital | properties | | |
| applications | interfaces | | resources |

| | | | |
|---|---|---|---|
| Computer language into data | None of the above | | Source code into data |
| Linker | None of the above | | Linker |
| is an device that performs a sequence of operations specified by instructions in memory | none of these | | is the device where information is stored |
| is a device that performs a sequence of operations specified by instructions in memory | none of these | | is a sequence of instructions |
| hardware | information | | software |
| varies in difficulty between systems | requires extensive tools to be understandable | | requires extensive tools to be understandable |
| CHKDSK | TYPE | | DISKCOPY |
| MS-DOS | OZ | | OZ |
| any time | for few hours | | when required |
| customized software | package software | | application software |
| operating system | windows | | application software |
| system software | utility program | | package software |
| 4 or 6 | 6 or 7 | | 4 or 6 |

| | | | |
|---|---|---|---|
| Inductors | Transistors | | Inductors |
| anywhere on the disk | none of these | | at a fixed location on the disk |
| Zeroeth pass | Not done by assembler | | Second pass |
| three pass | load and go | | three pass |
| Second pass and first respectively | Both first pass | | Both first pass |
| not evaluated by the assembler | None of above | | Pass 1 |