FABER-CASTELL
Date
Page No.

Tutorial No. 1

Q-1 What are Language Processors? Write different types of Language processors. Explain language processing activities.

→ • Language processors are software tools that facilitate the development & execution of program written in different programming languages (High-level languages).
• They transform human-readable code written in high-level programming languages into machine-readable instructions that can be executed by a computer.
• Language processors plays an important role in software development by enabling programmers to write code efficiently & effetively.
• Designer expresses ideas in terms related to application domain of software, to implement these ideas, their description has to be interpreted in terms related to execution domain.
• Language processor bridges a specification or execution gap.

• Types of language processors:-

1] Compilers:- A compiler translates entire source code written in high-level language into machine code or an intermediate representation. The resulting executable file can then be executed independently of the compiler.

2] Interpreter:- An interpreter translates source code into machine code or an intermediate representation line by line, executing each line immediately. Interpreters are commonly used in scripting languages & are useful for rapid development and debugging.

3] Assembler:- An assembler translates assembly language code, which is low-level symbolic representation of machine code instructions, into machine code directly executed by computer's hardware.

Language processing activities involve several stages:-

1] Lexical Analysis:-

- This stage involves breaking down the source code into tokens or lexemes, which are the smallest meaningful units of language.
- This process identifies keywords, identifiers, constants, operators and other language elements.
- For each lexeme, the lexical analyzer produces as output a token of form
  <token_name, attribute_value>
- The lexical analyzer produces a symbol table

e.g. position = initial + rate * 60

<id, 1> <=> <id, 2> <+> <id, 3> <*> 60

2] Syntax Analyzer:-

- The second phase is syntax analysis or parsing.
- The parser uses the first components of tokens produced by lexical analyzer to create a tree-like intermediate representation that depicts grammatical structure of token stream.
- A typical representation is a syntax tree in which each interior node represents an operation & children of the node represents argument of operation.

3] Semantic Analysis:-

- The semantic analyzer uses syntax tree & the information in symbol table to check the source program for semantic consistency with the language definition.
- An important part of semantic analysis is type checking, where compiler checks whether each operator has matching operands.

4] Intermediate Code Generation:-

- This step involves generating an intermediate representation of source code that is closer to machine language but still independent of target machine architecture.
- A compiler may construct one or more intermediate representations which can have variety of forms.
- This intermediate representation should be:
  i] Easy to produce
  ii] Easy to translate into target machine.

5] Machine - independent code optimization :-

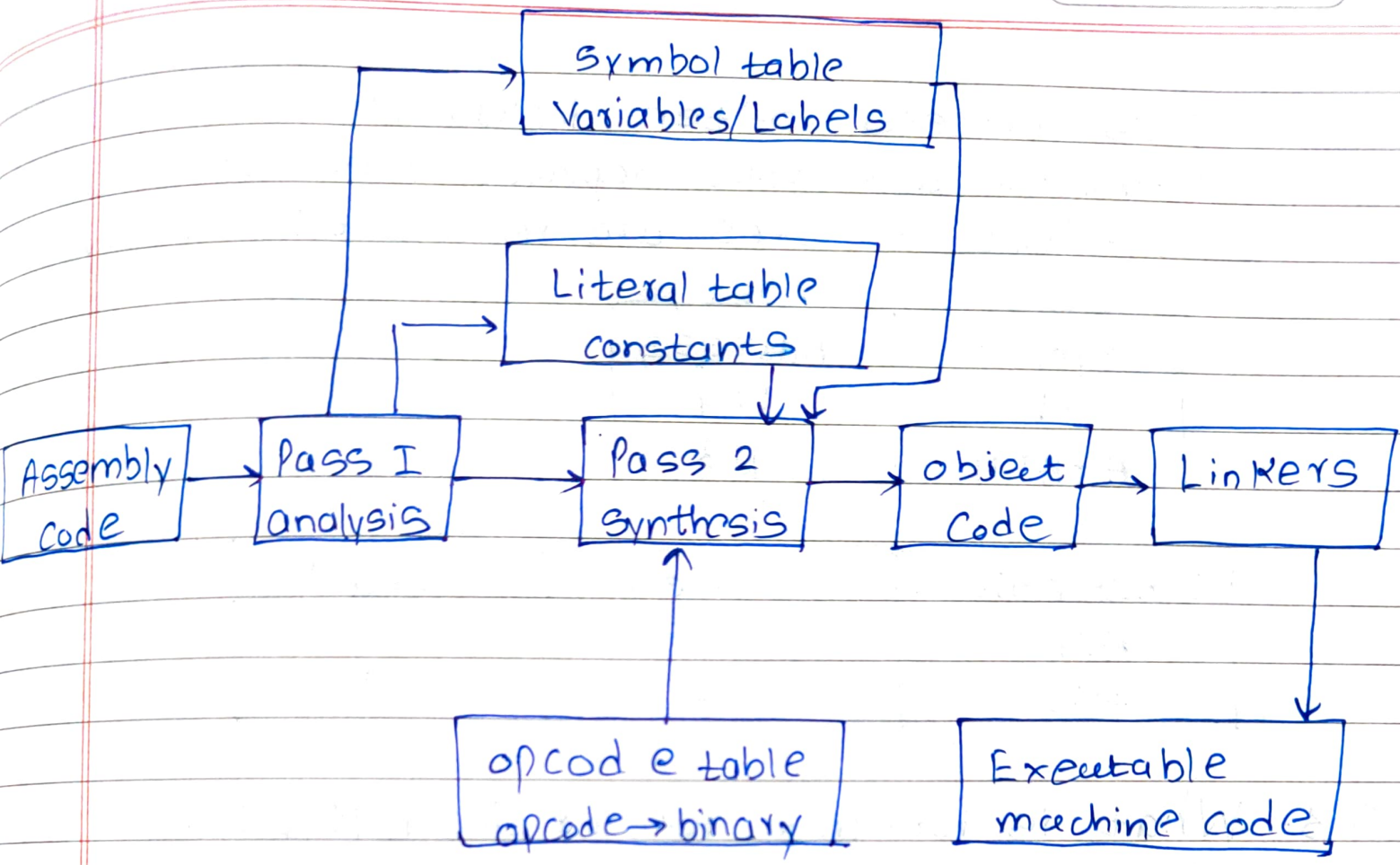
- This phase attempts to improve the intermediate code so that better target code will result.
- Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

6] Code Generation :-

- The code generator takes intermediate representation as an input and maps it into target language.
- If target language is machine code, registers or memory locations are selected for each of the variables used by the program.
- Then intermediate instructions are translated into sequences of machine instructions that perform the same task.

Q.2. Explain pass structure of assembler.

→ - Assembler can be defined as a program that translates an assembly language program into a machine language program.
- The design of assembler can be of:
  i] Scanning (tokenizing)
  ii] Parsing (validating the instructions)
  iii] Creating the symbol table.
  iv] Resolving the forward references.
  v] Converting into machine language.



At run time : Executable → loader

Pass I :- Analysis

- This part scans program looking for symbols, label Variables, etc and organises them in tables.
- Passes through instruction in sequence, looking for symbol addresses.
- Create a symbol and literal table.
- Keep track of the location counter.
- Process pseudo operations
- Error checking

Pass 2:- Synthesis

- If no errors are found in pass one then the second pass assembles code into object code.

Date
Page No

· This process often includes:
- Symbolic addresses replaced with absolute addresses
- Symbolic opcodes are replace with binary opcodes

Symbol	Address
N	104
AGAIN	113

Mnemonic	Opcodes	length
ADD	01	1
SUB	02	1

Q.3. What is forward referencing? How it is resolved.

→ · Forward referencing occurs when a symbol or label is used in a program before it is defined.
· In the context of assembly, this typically happens when a label is referenced before it appears in the source code.
· Resolving forward references is a crucial task for assemblers to generate correct M/c.

Here's how forward referencing is typically resolved in two-pass assemblers:

1) First pass:-
· The assembler scans entire source code to identify & record symbols & labels.
· When a label is encountered, assembler notes its address or assigns a temporary value to it.
· If symbol is used before its definition, assembler makes note of this forward reference.

2] Second pass :-

- The assemblers utilizes information collected in the first pass.
- It resolves forward references by updating addresses or values of symbols that were initially left as temporary in first pass.
- The assembler replaces temporary values with actual addresses or values calculated during second pass.
- This process ensures that all symbols & labels used in program are correctly resolved & final M/C is generated without any undefined references.

Q.4. Describe design specification of two pass assemblers :-

→ Two pass assembelers :

- Does the work in two pass
- Resolves the forward references.

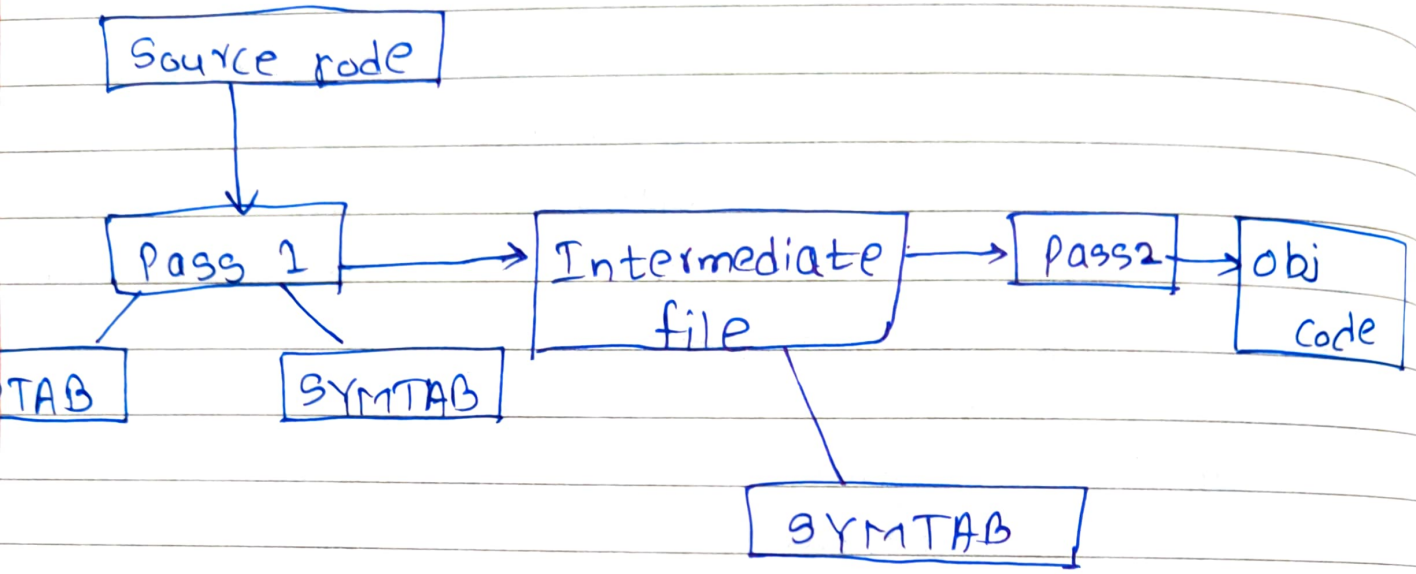
Assembler Design :-

First pass :-

- Scan code by separating symbol, mnemonic opcode & operand fields.
- Build symbol table.
- Perform LC (Location Counter) processing
- Construct intermediate representation.

Second pass:-

- Solves forward references
- Converts the code to machine code.



Read from input file
LABEL, OPCODE, OPERAND

Data structures in pass I

OPTAB :- A table of mnemonic opcodes.

- contains mnemonic opcode, class & mnemonic info.
- class field ~~contains~~ indicate whether opcode corresponds to
  - an imperative statement (IS)
  - a declarative statement (DL)
  - an assembler directive (AD)

Mnemonic opcode	Class	Mnemonic Info
MOVER	IS	(04, 1)
DS	DL	R #7
START	AD	R #11

- for IS, mnemonic info field contains pair (machine code, instruction length)
- else, it contain id of routine to handle declaration or directive statement.
- The routine processes the operand field of the statement to determine the amount of memory required & updates LC & SYMTAB entry of symbol defined.

SYMTAB - Symbol Table
Contains Addr & length

LOCCTR - location counter
LITTAB - table of literals used in program

- Contains literal & address
- Literals are allocated addresses starting with current value in LC & LC is incremented.

Intermediate Code (IC) forms.

- Two criteria for choice Intermediate code
  - Processing efficiency.
  - Memory efficiency
- * Design of consider two variants of IC & ~~and~~ comparing them on above criteria.
- Intermediate code is sequence of IC units.
- Each IC unit contains three parts:-
  - Address
  - Representation of mnemonic opcode.
  - Representation of operands.

Address	Mnemonic opcode	operands

- Mnemonic opcode field contains pair of form (statement class, code)
- statement class can be IS, DL & AD.
- for imperative statement, code is instruction opcode in machine language.
- For declarations & Assembler directives, code is ordinal number within class.
- Code for DL & AD

Declaration Statement	
DC	01
DS	02

DC → Declare Constants
DS → Declare storage

Assembler	Directives
START	01
END	20
ORIGIN	03
EQU	04
LTORG	05