Assignment

11/7/24

Q:1 Explain the role of assembler, compiler, loader & linker in the language processing system.

A: • ASSEMBLER:-

- The role of the assembler is to convert the assembly language code produced by compiler into the object code. that the computer's processor can understand & execute.

- It takes each assembly level instruction & translates it into the corresponding machine code. & also handles directives & macros defined in the assembly code.

• Compiler :-

- It translates high level programming languages (like C, C++, C#, etc) into machine code or intermediate code that can be directly executed by the computer.

- It analyzes the entire code before converting into machine code.

- It has several Phases like lexical, syntax, semantic, intermediate code generation, code optimization & target code generation.

• LOADER :-

- The role of the loader is to the executable file into memory for execution by resolving external references & allocating memory space.

• LINKER :-

- The task of the linker is to link together multiple object files & libraries into a single executable file.

Q: Explain the role of Lexical Analysis.

A: • LEXICAL ANALYSIS :-

- The first phase of a compiler that converts the sequence of characters in the source code into a sequence of tokens (lexical units)

- It scans the entire source code, identifies the tokens (like keywords, identifiers, constants, etc) & output tokens.

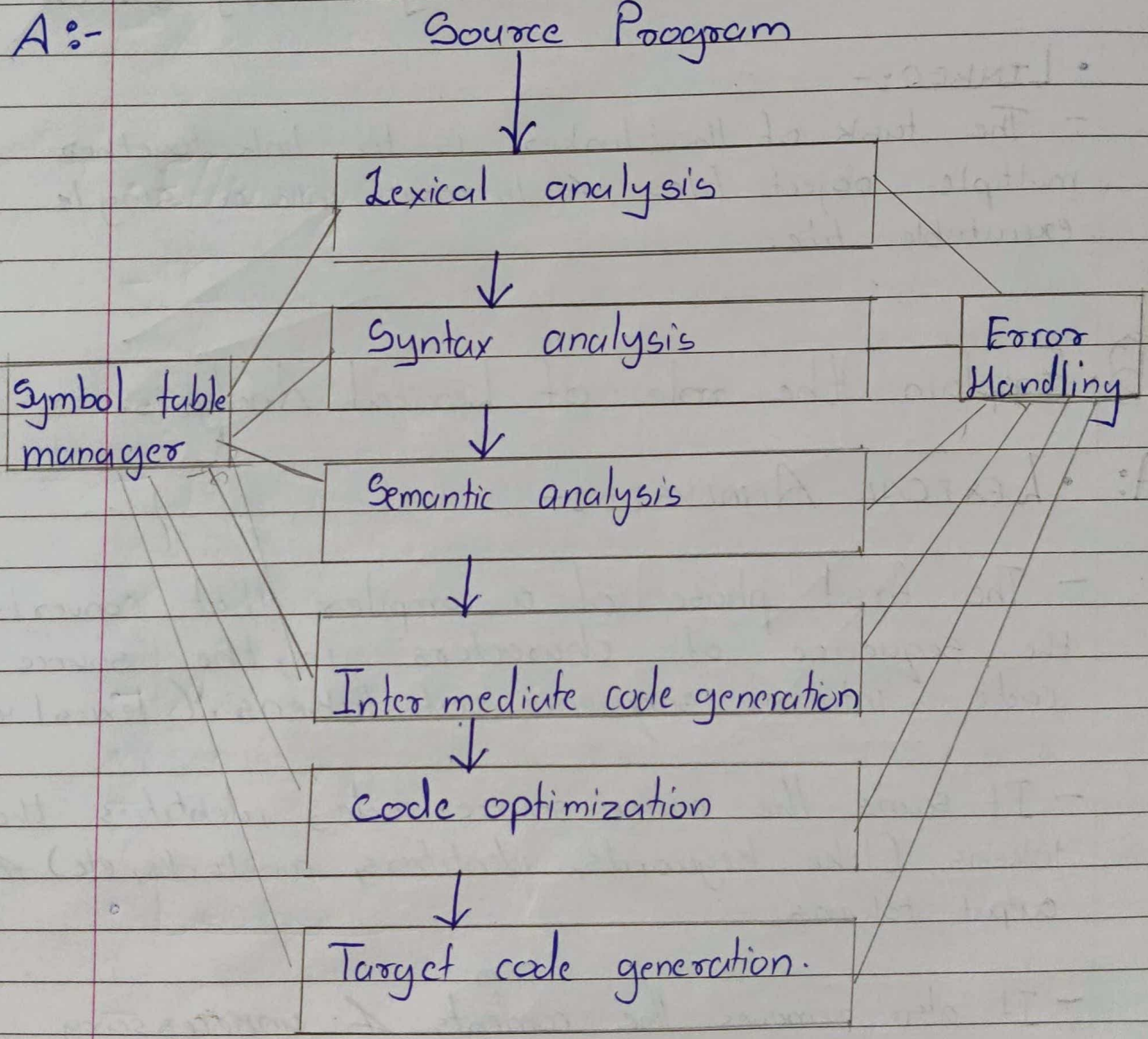
- It also removes the comments & unnecessary white space.

- The output tokens are used by the subsequent phases like syntax analysis

- This is the role of lexical Analysis.

Q3: Draw a block diagram of Phases in and list the main functions of each Phase.

A:-



* Main functions :-

- lexical anaylsis :- Converts characters to tokens.
- Syntax analysis :- checks for the sytax & builds a parse tree.
- Semantic analysis :- Verifies the meaning of statements.

Page No.
Date

- Intermediate Code generation :- Produces intermediate representation for easier generation.
- Code optimization :- Enhances intermediate code for efficiency
- Code Generation :- Translates optimized code into target machine code.
- Symbol table manager :- stores info. about identifiers & symbols for efficient lookup during compilation.
- Error handling :- Handles the error generated at any of the phases.

Q4 Compare and contrast compilers, interpreters & assemblers.

A:
- Input :-

- compilers & interpreter both convert high level lang into machine code but assemblers take Assembly language code (mneomics) as input.

- Execution Process :-

- Compilers Perform a complete translation of the source code to produce an executable file. by different Phases like lexical, syntax, etc...

- Interpreters directly execute the source code by interpreting each line or statement at runtime.

- Assemblers convert each assembly language inst into its corresponding machine code inst.

• **Efficiency:-**

- Compilers generally produce highly optimized machine code since they can perform extensive analysis and optimization across the entire program.

- Interpreters can be slower compared to compiled programs because each line or statement is interpreted at runtime.

- Assemblers are generally faster than compilers in generating executable code since they perform a more direct translation.

• **Debugging:-**

- Compilers & Assemblers both have challenging time in debugging the code because errors are detected after the entire code is compiled or transalted from assembly to machine code.

- Debugging in interpreters is easier as errors are identified during run time.

- Examples:-

- Compilers :- GCC, Clang, java compiler.
- Interpreter :- Python interpreter, javascript interpreter Ruby interpreter.
- Assemblers :- Netwide, Microsoft Macro & GNU.

Q6 Explain about the structure of a compiler.

A: A compiler typically consists of the following major components :-

- Front End :-
  - It handles input & initial processing phases.
  - the lexical phase generates tokens
  - Syntax Analysis (Parser) checks the syntax and generates the parse tree.
  - Semantic Analysis checks the semantics & meaning of the statements beyond syntax.

- Middle End :- Intermediate
  - It consists of, code generation (IR) & code optimization
  - The source code is translated to Intermediate representation (IR).

- Back End :-
  - Target code generation & target-specific optimizations are done
  - The source code is translated to the machine code.

Page No.
Date

- **Symbol table :-**
- Information about identifiers & symbols used in the source codes are stored here
- All the phases make use of symbol table.

Q7: Explain about input buffering in detail.

A: Input buffering involves reading characters from input (like a file or keyboard) into a buffer before processing.

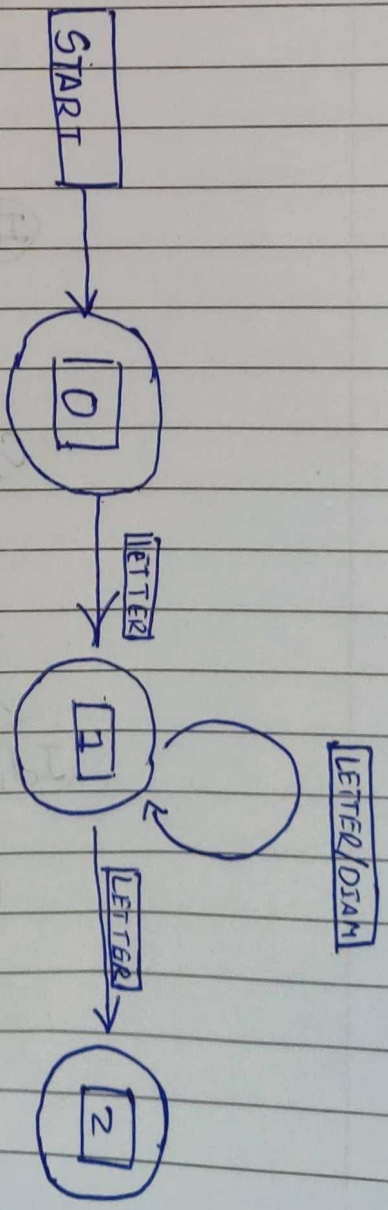
=> The purpose is to enhance the efficiency by reducing the numbers of I/O operation & allow the compiler to handle data in larger chunks rather than character by character

- It refers to the way in which the compiler reads input from the source code
- The size of the buffer can vary depending on the specific needs of the compiler & the characteristics of the source code being compiler.
- One of the main advantages of input buffering is that it can reduce the number of the system calls required to read input from the source code.

- Since each system call carries some over-head reducing the number of calls can improve performance.

- This is how input buffering helps with increasing the efficiency.

Q8 Design a transition diagram for identifier & the keyword "else".



Page No.
Date

Q9 Show the O/P of each phase in compiler for the following statement.

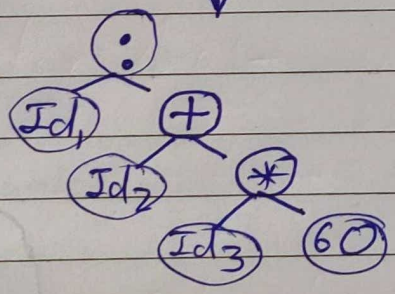
Position = initical + rate * 60.

A:

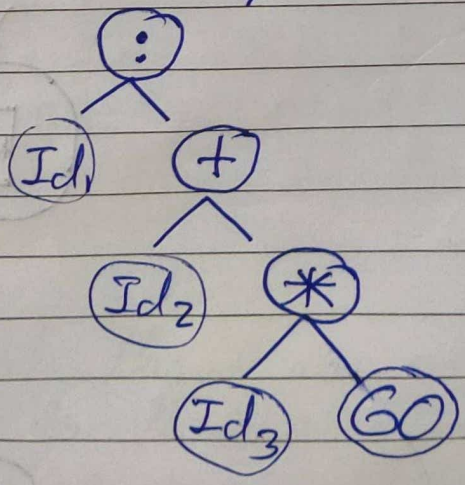
lexical analysis

$id_1 : id_2 + id_3 * 60$

Syntax analysis



Semantic analysis



Page No.
Date

Intermediate Code generation

↓

$temp_1 := integer\ to\ real\ (60)$
$temp_2 := id_3 * temp1$
$temp_3 := id_2 + temp2$
$id_1 = temp3$

Code Optimization

↓

$temp1 := id_3 * 60.0$
$id_1 = id_2 + temp1$

Code generation

↓

```
movf  id3 , R2
mulf  60.0 , R2
movf  id2 , R1
addf  R1 , R2
movf  R2 , id1
```