Chapter 1 Phases of compiler……

Page :
Date : / /

* Compiler
Defination - A Computer program which reads source code and outputs Assembly code or Executable code is called as Compiler
Example - 'C' Compiler, 'CPP' - Compiler, VB - Compiler

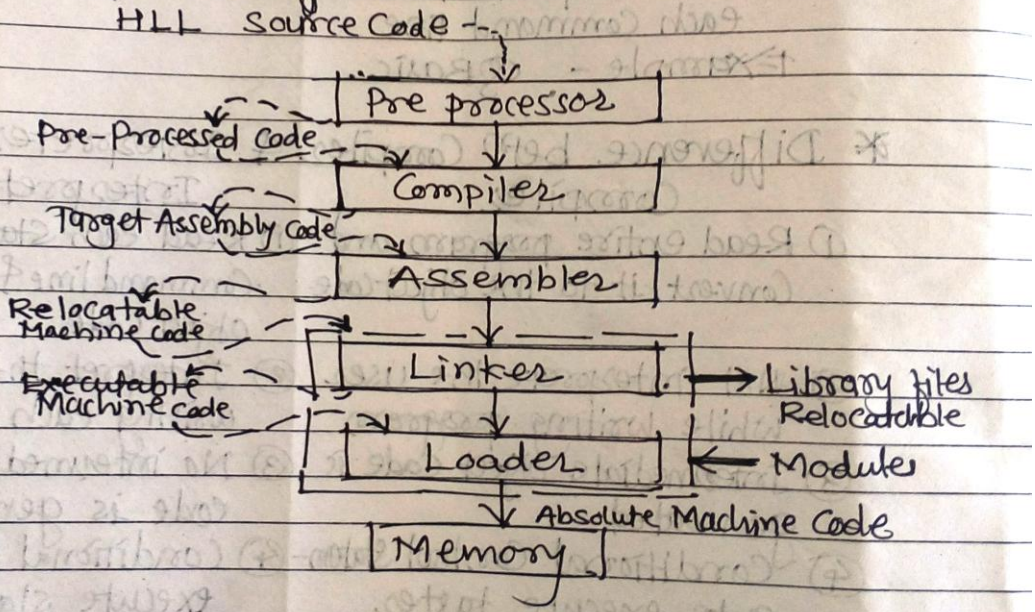
* Interpreter
Defination - Interpreter is a program which reads each and every line of the source program and converts it to the object code.
- It interprets the source line while writing each command line.
Example - QBasic..

* Difference bet? Compiler & Interpreter

Compiler	Interpreter
① Read entire program and Convert it to the object code	① Read each statement or Command line & Convert it to object code.
② Not interprete the user while writing program	② Interpret the uses while writing each Command line
③ Intermediate object code is generated	③ No intermediate object code is generated.
④ Conditional Control statements execute faster	④ Conditional Control statements execute slower
⑤ Create Stand alone executable file which run without loading language	⑤ It is necessary to load interpreter every time to run. Source program does not create stand alone executable file.
⑥ Program need not be Compiled every time	⑥ Every time higher level Prog is converted into Lower Level Program

Compiler	Interpreter
⑦ Memory requirement is more	⑦ Memory requirement is less
⑧ Error are displayed after entire program is checked	⑧ Error are displayed for every instruction interpreted
⑨ Due to executable file source program is secured i.e. it is difficult to modify executable file.	⑨ Any body can modify the program written in interpreter environment
⑩ Example - C Compiler	⑩ Example - Basic

* ~~Compiler Design~~ —



① User write a program i.e. High Level Language (HLL) Program
② Compiler compiles High level Language Program & ~~Convert~~ translate it to Assembly Program i.e. in Low level language
③ Assembler translate it into Machine Code i.e. Object Code
④ Linker tool is used to link all parts of Programs (Modules) together for execution (Executable Machine Code)
⑤ A Loader loads all of the executable code into Memory & then Program is executed.

* Preprocessor
A preprocessor produce input to Compilers. They may perform the following functions...
1) Macro processing — A preprocessor may allow a user to define macros that are short hands for longer constants.
2) File inclusions — A preprocessor may include header files into the program text.
3) Rational preprocessor — These preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
4) Language Extensions — These preprocessor attempts to add capabilities to the language by certain amounts, to build-in macro.

* Compiler
Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a Compiler is error showing to the programmer.

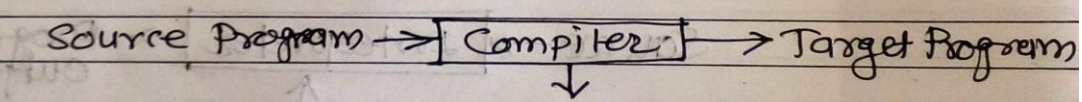


Fig:- Structure of Compiler

Executing a program written in HLL programming language is basically of two parts...
① The source program must first be Compiled translated into a Object Program.
② Then the results object program is loaded into memory executed.

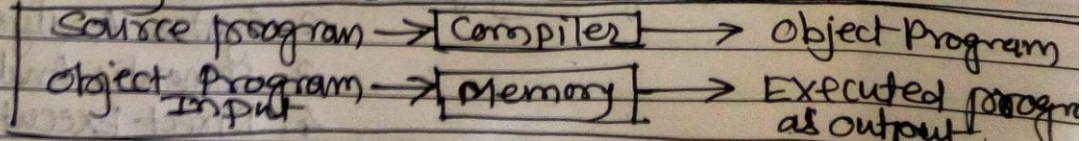


Fig- Execution proceis of Source Program in Compiler

Date: / /

* Assembler
- Programmers found it difficult to write or read programs in machine language.
- They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language.
Such a mnemonic machine language is now called an assembly language.
- Program known as assemblers were written to automate the translation of assembly language into machine language.
- The input to an assembler program is called source program, the output is a machine language translation (object Program)

* Interpreter
- An interpreter is a program that appears to execute a source program as if it were machine language.

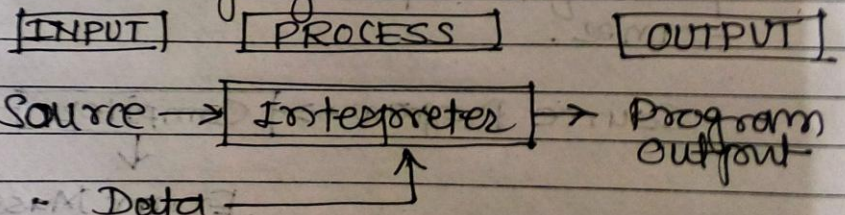


Fig: Execution in interpreter

- Language such as BASIC, SNOBOL, LISP can be translated using interpreters.
- Java also uses interpreter.
- The process of interpretation can be carried out in following phases 1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Direct Execution

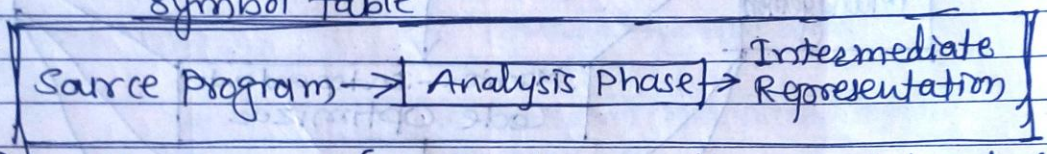
1) Phases of Compiler Design
   i) Lexical Analysis
   ii) Syntax Analysis
   iii) Semantic Analysis
   iv) Intermediate code generation
   v) Code optimization
   vi) Code generation
2) Lexical Analysis
   i) Grammar
   ii) Leftmost derivation
   iii) Rightmost derivation
   iv) Ambiguous Grammar
   v) How to mak Ambiguous Grammar disambiguous
   vi) Left Recursion & Right Recursion
   vii) Elimination of Left Recursion - Left factoring
   viii) Deterministic Grammar &
      Non Deterministic Grammar
3) Parsers
   i) Top Down Parsers
      @ Top Down Parser with full Back tracking
      (b) Top Down Parser without - Back tracking
         • Recursive Descent
         • Non-Recursive Descent Parsing (LL(
   ii) Bottom-up Parsers (BUP)
      @ Operator-Precedence Parser
      (b) LR Parser
         i) LR(0)
         ii) SLR(1)
         iii) LALR(1)
         iv) CLR
4) Syntax Directed Translation
5) Intermediate Code Generation
6) Code optimization

Page :
Date : / /

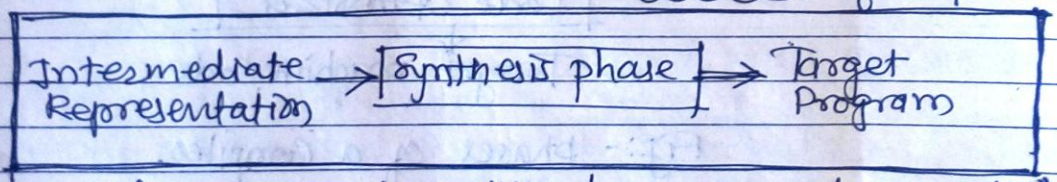
* Phases of Compilation

- The structure of compiler consists of two parts...

① **Analysis part** (Machine Independent/Language Dependent)
- → Analysis part breaks the source program into constituent pieces and imposes a grammatical structure on them which further uses this structure to create an intermediate representation of the source program.
- → It is also termed as front end of compiler.
- → Information about the source program is collected & stored in a data structure called Symbol table



② **Synthesis part** (Machine dependent/Language independent)
- → Synthesis part takes the intermediate representation as input and transforms it to the target program.
- → It is also termed as back end of compiler.



- Compilation process is partitioned into no.of sub process called Phases
- The design of compiler can be decomposed into several phases, each of which converts one form of source program into another.
- The different phases of compiler are as follows...
  1. Lexical Analysis
  2. Syntax Analysis
  3. Semantic Analysis
  4. Intermediate Code generation
  5. Code optimization
  6. Code generation

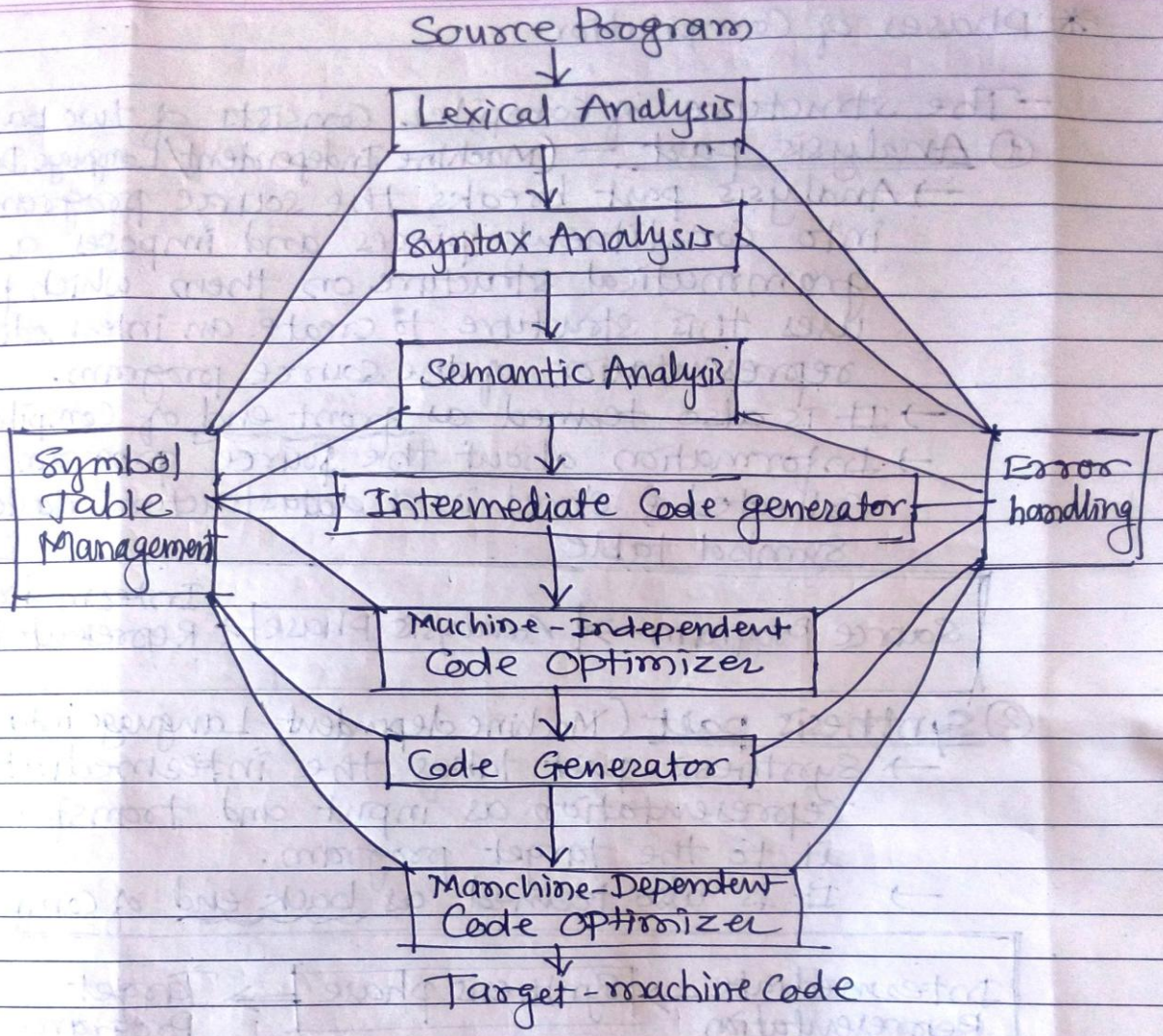


Fig:- phases of a Compiler

① Lexical Analysis

→ Lexical Analysis is the first phase of Compiler which is also termed as scanning

→ Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes which produces token as output.

• Token :- It is a sequence of characters that represent lexical unit, which matches with the pattern, such as keywords, operators, identifiers etc.

• Lexeme :- Lexeme is instance of a token i.e. group of characters forming a token.

Example $c = a + b * 5$;

Lexemes	Tokens
c	identifier
=	assignment symbol
a	identifier
+	+ (operator) (Addition symbol)
b	identifier
*	* (operator) (Multiplication symbol)
5	5 (number)

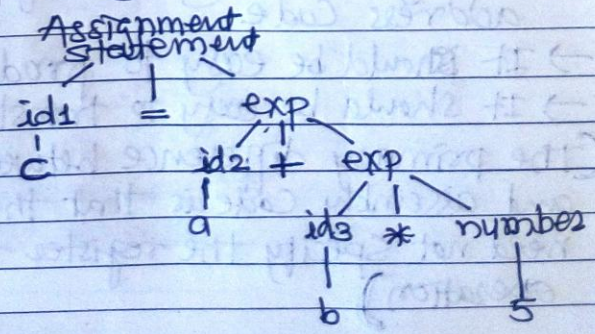
→ Output of lexical analysis is a string of tokens which is passed through next phase.

② Syntax Analysis

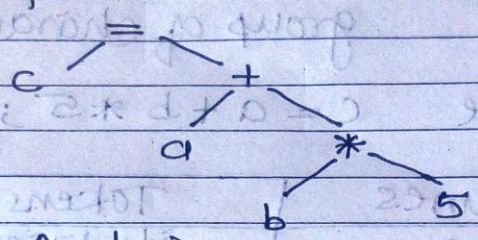
→ Syntax analysis is the second phase of compiler which is also called as parsing.

→ Parser converts the tokens produced by lexical analyzer into a tree like representation called parse tree.

→ A parse tree describes the syntactic structure of the input (Example $c = a + b * 5$)

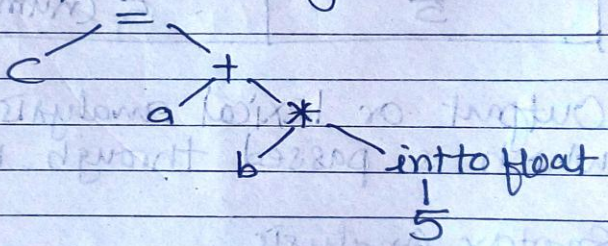


→ Syntax tree is a compressed representation of the parse tree in which the operator appear as interior nodes and the operands of the operator are the children of the node for that operator.



③ Semantic Analysis

→ Semantic analysis is the third phase of compiler.
→ It checks for the semantic consistency.
→ Type information is gathered and stored in symbol table or in the syntax tree.
→ Peeforms type checking



④ Intermediate Code generation

→ It produces intermediate representation for the source program which are the following forms:
  ⓐ Postfix notation
  ⓑ Three address code (x=y+z)
  ⓒ Syntax tree
→ Mostly commonly used form is the three address code.
→ It should be easy to produce
→ It should be easy to translate into target program.

(The primary difference between intermediate code and assembly code is that the intermediate code need not specify the register to be used each operation)

⑤ Code optimization

→ Code optimization phase designed to improve/optimize the intermediate code so that ultimate object program runs faster and/or takes less space.

→ This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code (object code) will result

→ To improve the code generation, the code optimization involves...

ⓐ Deduction of dead code (unreachable code)
ⓑ Calculation of constants in expressions & terms.
ⓒ Loop unrolling.
ⓓ Moving code outside the loop.
ⓔ Removal of unwanted temporary variables

for Ex- $t1 = y * z$, $t2 = x + t1$, $a = t2$ } $t1 = y * z$, $a = x + t1$

⑥ Code Generation

→ Code generation gets the input from code optimization phase and produces the target code or object code as result.

→ The final phase, code generation produces the object code, by deciding the memory-location for data, selecting code to accept & selecting the register in which each calculation is to be done.

→ The code generation is the most difficult part of compiler designing in the both way's practically and theoretically.

→ The code generation involves...

ⓐ Allocation of register and memory.
ⓑ Generation of correct references
ⓒ Generation of correct data types.
ⓓ Generation of missing code.

```
LDF   R2, id3
MULF  R2, #5.0
LDF   R1, id2
ADDF  R1, R2
STF   id1, R1
```

*Symbol Table Management

→ Symbol table is used to store all the information about identifiers used in the program.

→ It is a data structure containing a record for each identifier, with fields for the attribute of the identifier.

→ It allows finding the record for each identifier quickly and to store or retrieve data from that record.

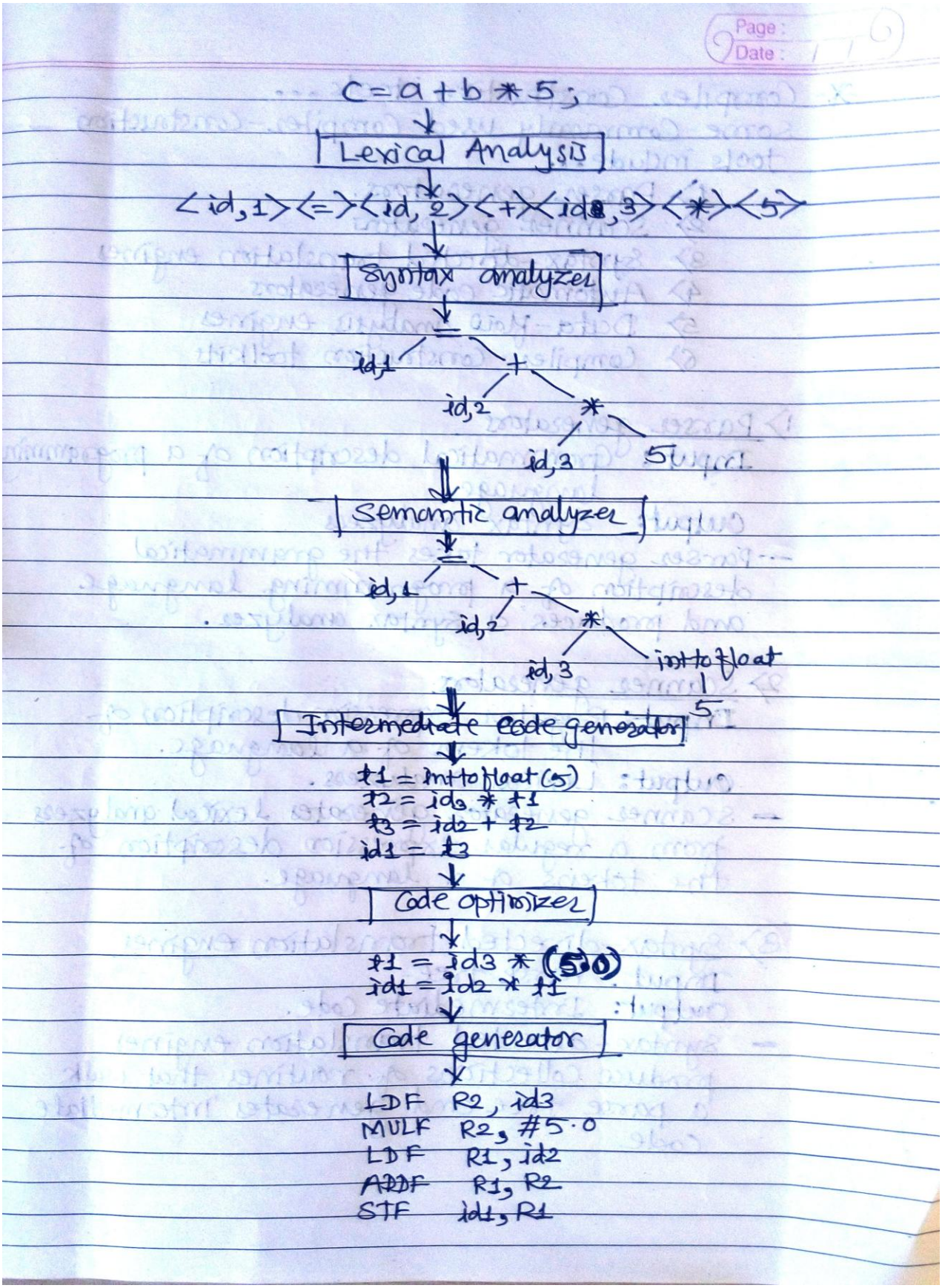
→ Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

*Error handling

→ The Error handling is invoked when an Error in source program is detected.

→ Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.

- In Lexical analysis, errors occur during seperation of tokens.
- In syntax analysis, errors occurs during construction of syntax tree.
- etc


c=a+b*5;
Lexical Analysis
<id,1><=><id,2><+><id,3><*><5>
Syntax analyzer
id,1
=
+
id,2
*
id,3
5
Semantic analyzer
id,1
=
+
id,2
*
id,3
inttofloat
5
Intermediate code generator
t1 = inttofloat(5)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code optimizer
t1 = id3 * (5.0)
id1 = id2 * t1
Code generator
LDF R2, id3
MULF R2, #5.0
LDF R1, id2
ADDF R1, R2
STF id1, R1

* Compiler Construction tools...
Some Commonly used Compiler-Construction tools include...

1) Parser generators.
2) Scanner generators
3) Syntax-directed translation engines
4) Automatic code generators
5) Data-flow analysis engines
6) Compiler-Construction toolkits

1) Parser generators
Input: Grammatical description of a programming language.
Output: Syntax analyzers
— Parser generator takes the grammatical description of a programming language and produces a syntax analyzer.

2) Scanner generators
Input: Regular expression description of the tokens of a language.
Output: Lexical analyzers.
— Scanner generator generates lexical analyzers from a regular expression description of the tokens of a language.

3) Syntax-directed translation engines
Input: Parse tree.
Output: Intermediate code.
— Syntax-directed translation engines produce collections of routines that walk a parse tree and generates Intermediate code

4) Automatic code generators
Input: Intermediate language
Output: Machine language.
- Code-generator takes a collection of rules that defines the translation of each operation of the intermediate language into the machine language for a target machine.

5) Data-flow Analysis Engines
- Input: Data-flow analysis engine gathers the information, that is, the values transmitted from one part of a program to each of the other parts.
- Data-flow analysis is a key part of Code optimization.

6) Compiler Construction Toolkits
- The toolkits provide integrated set of routines for various phases of Compilers
- Compiler construction toolkits provide an integrated set of routines for construction of phases of Compiler.

a = b * c + d * e

↓

[Lexical Analysis]

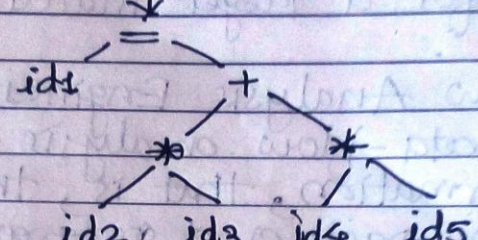
↓

id1 = id2 * id3 + id4 * id5

↓

[Syntax Analysis]

↓



↓

[Intermediate Code generator]

↓

temp1 = id2 * id3
temp2 = id4 * id5
id1 = temp1 + temp2

↓

[Code generator]

↓

```
LDF  R1, id2
LDF  R2, id3
MUL  R1, R2
LDF  R3, id4
LDF  R4, id5
MUL  R3, R4
ADD  R2, R4
MOV  id1, R4
HLT
```