Module - I Introduction to Compiler

What is a compiler?

A compiler is a program that reads a program written in one language and translates it into an equivalent program in another language.

A compiler also reports error present in the source program as a part of its translation process.

Why should we study compiler design?

- Compilers are everywhere!
- Many applications for compiler technology
  - Parsers for HTML in web browser.
  - Interpretes for javascript / flash.
  - Machine code generation for high level languages.
  - Software testing.
  - Program optimization.
  - Design of new computer architecture.
    - Compiler in-the-loop hardware development.
  - Compiled simulation
    - Used to simulate designs written in VHDL.
    - No interpretation of design, hence faster.

Evaluation of compiler

Computers are electronic machines or in other words they understand only zeros and ones. Therefore in early days of computation instructions to a computer were given using perforated paper tapes and punch cards. In these the presence of a ~~whole~~ hole is representation of zero and absence of them means one. Now writting down instructions in the form of ones and zeros is pretty hectic.

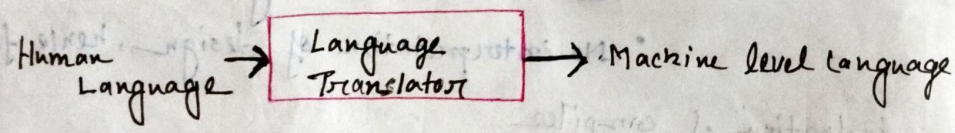
P.T.O

Punched card if converted to binary using ASCII standard

P - 1010000
V - 1110110
W - 1101110
c - 1100011
h - 1101000
e - 1100101
d - 1100100

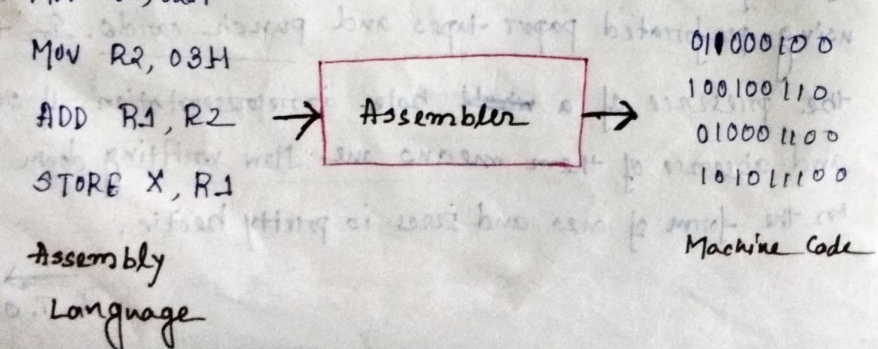
c - 1000011
a - 1100001
r - 1110010
d - 1100011

This is the format which computers can process directly. But for us it is almost impossible. Even if it becomes possible for any human to encode instructions like this, it is undoubtedly time consuming. For this we need language translator, in simpler terms we will say punch cards, and the middleman that is the language translator will convert that into the strings of ones and zeros.



(High level language)

1. Assembler :



Now assembly language has a very strong correspondence b/w the instructions of it and the architectural specification of the computer. That means in order to write instructions for the computer in assembly language the coder needs to have a proper knowledge about the computer's architecture first.

In the 50's two new language translators came into picture

II. Interpreter

~~Some~~ PHP
RUBY
PYTHON
Javascript

* one line at a time

III. Compiler

C
C++
C#
ERLANG

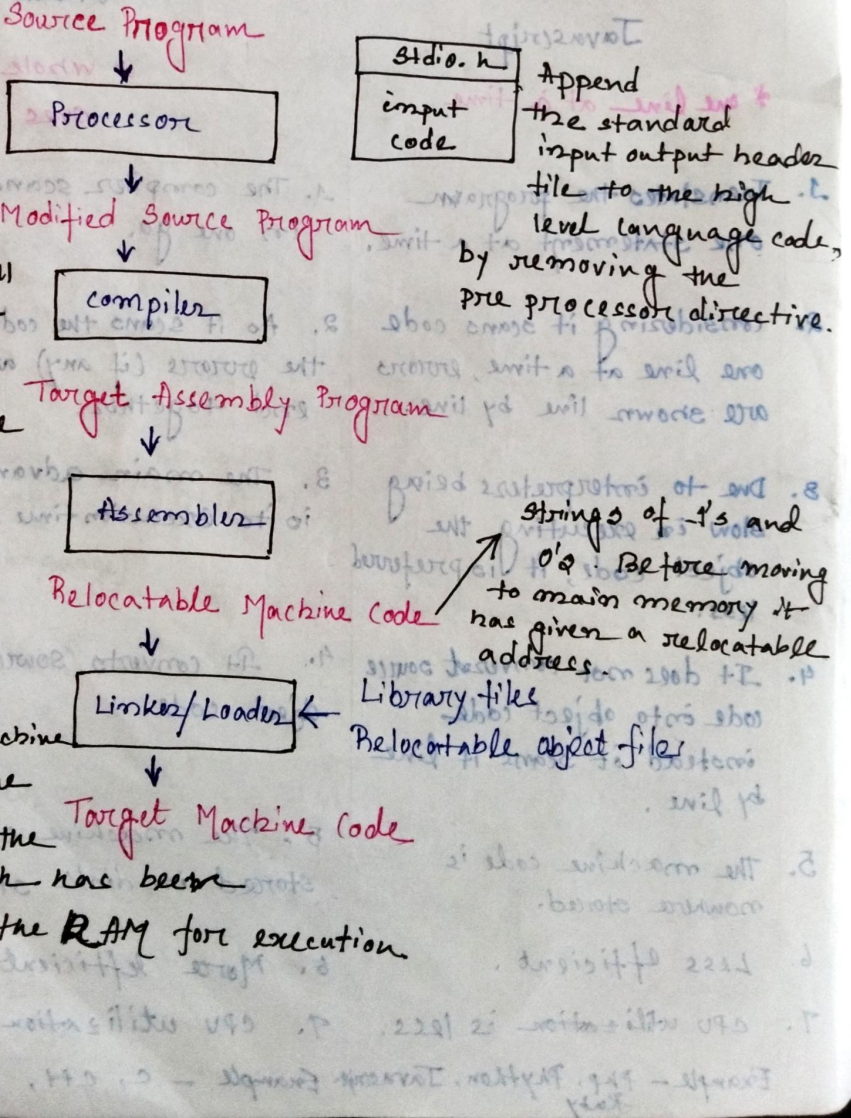
* whole program at ones

1. Translates the program one statement at a time.	1. The compiler scans the whole program in one go.
2. considering it scans code one line at a time, errors are shown line by line.	2. As it scans the code in one go, the errors (if any) are shown at the end together.
3. Due to interpreters being slow in executing the object code, it is preferred less.	3. The main advantage of compilers is its execution time.
4. It does not convert source code into object code instead it scans it line by line.	4. It converts source code into object code.
5. The machine code is nowhere stored.	5. The machine code is ~~nowhere~~ stored in disk storage.
6. Less efficient.	6. More efficient.
7. CPU utilization is less.	7. CPU utilization is ~~less~~ more.
Example – Php, Python, Javascript, Ruby	Example – C, C++, C#

C – High level language

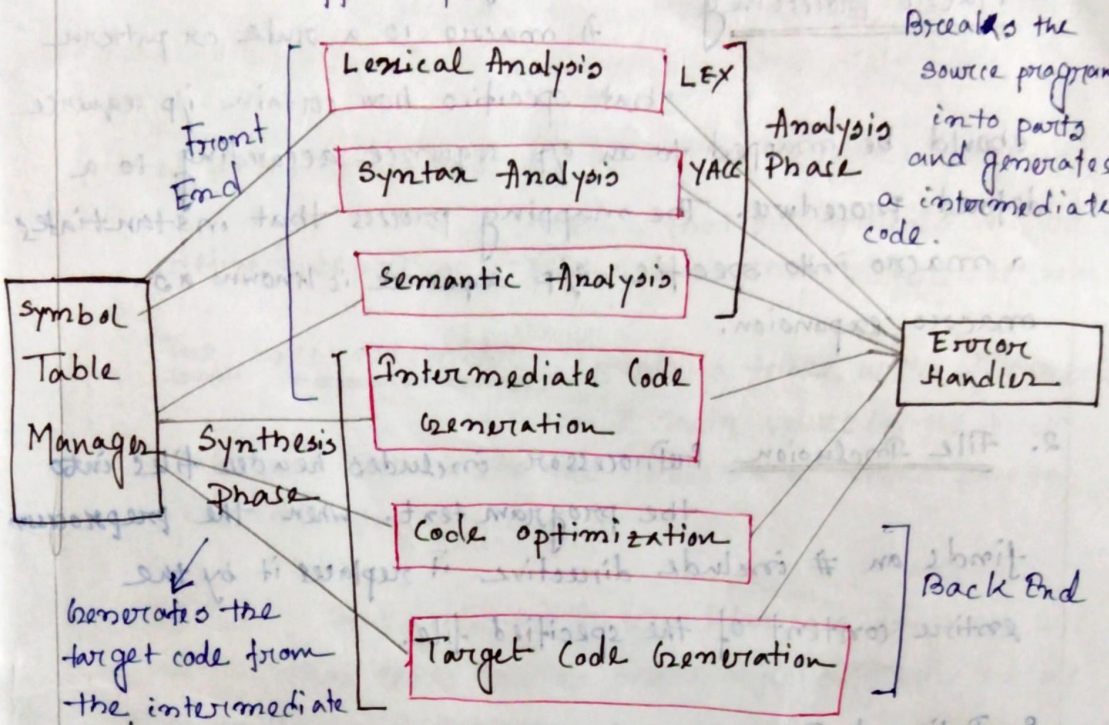
```
#include <stdio.h>          → preprocessor directive
int main()                  → main function
{
    int x, a=2, b=3, c=5;
    x = a+b*c;
    printf("The value of x is %d", x);
    return 0;
}
```

Language Processing System / Language Translator



Internal Architecture of Compiler

It has 6 different phases.



Cousins of a compiler

1. Preprocessor
2. Assembler
3. Loader
4. Linker

1. Preprocessor — A preprocessor is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent program like compilers.

They may perform the following functions:

1. Macro processing — A macro is a rule or pattern that specifies how certain i/p sequence should be mapped to an o/p sequence according to a defined procedure. The mapping process that instantiates a macro into specific o/p sequence is known as macro expansion.

2. File Inclusion — PreProcessor includes header files into the program text. When the preprocessor finds an # include directive it replaces it by the entire content of the specified file.

3. Rational Preprocessor — These processors change older language with more ~~modern~~ modern flow-of-control and data-structuring facilities.

4. Language Extension — These processor attempt to add capabilities to the language by what amounts to built-in macros. For example, the language Equel is a database query language embedded in C.

Assemblers Assembler creates object code by translating assembly instruction mnemonics into machine code. There are two types of assemblers:

Relocatable

One-pass assembler go through the souce code and assume that all symbols will be defined before any instruction that references them.

Two-pass assembler create a table with all symbols and their values in the first pass and then uses the table in a second pass to generate code.

Linker A linker or link editor is a program that takes one or more objects generated by a compiler, and combines them into a single executable program. The three main tasks of the linker are;

1. Searches the program to find library routines used by program, e.g printf(), math routines.
2. Determines the memory locations that code from each module will occupy and relocates the instruction by adjusting absolute references.
3. Resolves references among files.

Loader A loader is the part of an operating system that is responsible for loading programs in memory, one of the essential stages in the process of starting a program.

Different Phases of Compiler

x = a + b * a ; 20;

Lexical Analysis Phase generates tokens, by scanning the whole program.

x = a + b * a 20

Lexems	Tokens
x	identifier 1
=	operator 1
a	identifier 2
+	operator 2
b	identifier 3
*	operator 3
~~a~~	~~identifier 2~~
20	number

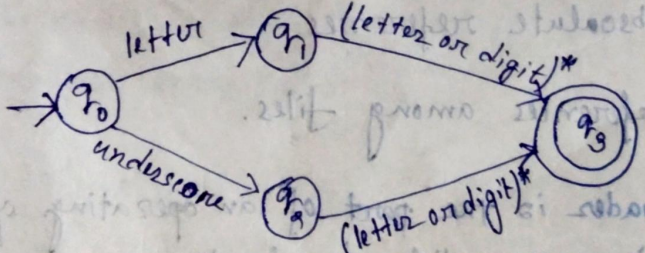
Regular expression for identifier:

$$l(l+d)^* \mid \_(l+d)^*$$

l : letter

d : digit

_ : underscore



So, for examining the lexemes the lexical analyzer makes use of the type 3 grammar.

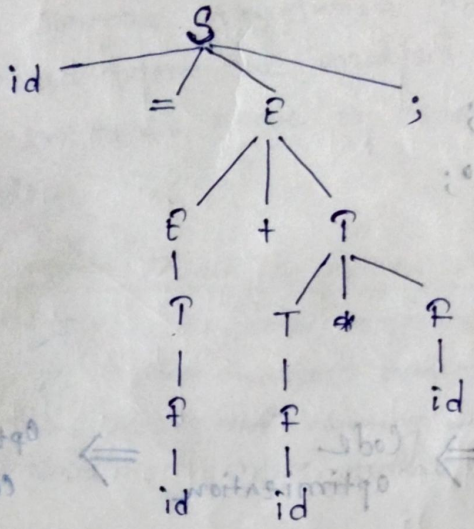
Syntax Analyzer - The stream of tokens is passed to the syntax analysis phase. The syntax analyzer depends on the type 2 or context-free grammar.

$S \rightarrow id = E;$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

CFG Production Rules of the expression.



Parse Tree

Output → id = id + id * id ;

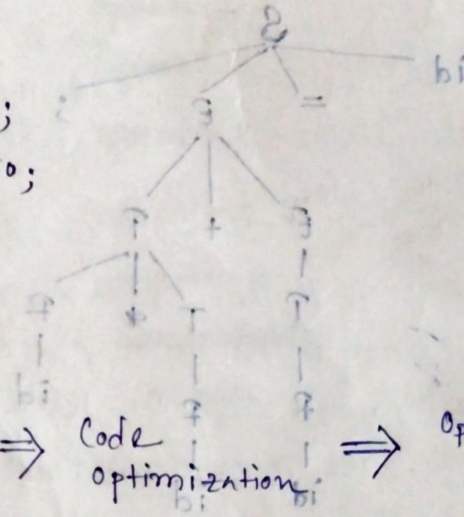
If the parse tree and the provided expression is not same, then syntax analyzer phase generates error message, or syntax error.

Semantic Analyzer — is responsible for type checking, array bound checking and the correctness of scope resolution. Basically it does the logical analysis of the parse tree. Semantic analyzer detects type mismatch errors, undeclared variables, misuse of reserved words, multiple declaration of a variable, ~~missing~~ out of scope variable, etc.
Semantic analyzer looks for the meaningfulness of the parse tree and verifies that.

Intermidiate Code Generator

Semantically verified Parse Tree ⇒ Intermediate code generation ⇒ Intermediate code

For generating Intermediate code, Three Address code is used.



1. $t_0 = b * c;$
2. $t_1 = a + t_0;$
3. $a = t_1;$

Code optimizer

Intermediate code ⇒ Code optimization ⇒ Optimized code

Code optimization can be machine dependent or machine ~~identi~~ independent.

1. $t_0 = b * c;$
2. $a = a + t_0;$

Target code Generator

Optimized code ⇒ Target code generator ⇒ Assembly code segment

```
mov eax, DWORD PTR [rbp-8]
```

Symbol Table It gathers information from the analysis phase and those gathered information are used by the synthesis phase. Symbol table is basically a data structure which is created and maintained by compilers in order to store information about the occurences of various entities such as variable and function name, objects, classes, interfaces extra.

Now the information inside the symbol table is gathered from the analysis phase which in turn used by synthesis phase in order to generate the target code.

Usages of Symbol Table by various Phases

Lexical Analysis - It creates entries for identifiers. lexical analyzer scan the entire source code line by line and during scanning whenever it encounters any identifier it creates the entry for that inside the symbol table.

Syntax Analysis - It adds information regarding attributes, like type, scope, dimension, line of reference, line of usage etc.

Semantic Analysis - using available info checks semantics of the identifier created, and updates the symbol table accordingly.
↓
during lexical analysis phase

Intermediate Code Generation - The available information inside the symbol table, helps the intermediate code generator in adding the temporary variables information.

Code Optimization - The available information stored inside the symbol table is used specifically in machine dependent optimization.

Target Code Generator - Generates the target code using address infor for identifiers, stored inside the symbol table.

Symbol Table - Entries

```
int count;
char x[] = "Compiler Design";
```

Name	Type	Size	Dimension	Line of Declaration	Line of usage	Address
count	int	2	0	-	-	-
x	char	15	1	-	-	-

Symbol Table - operations

- Non Block Structure Language (Fortran)
  - Contains single instance of the variable declaration.
  - Operations:
    i) insert() ii) lookup()
- Block Structure Language (C, C++)
  - Variable Declaration may happen multiple times.
  - Operations:
    i) insert() ii) lookup() iii) Set() iv) Reset()