

COMPILER CONSTRUCTION

Muhammad Islam tanoli

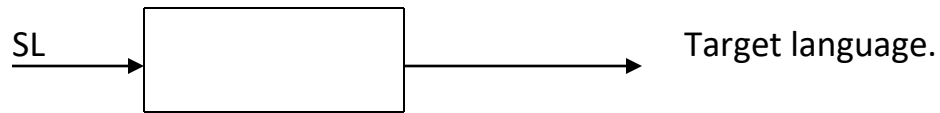
MCS

Session (2012-14)

Hazara University Mansehra.

Compiler:

Generally it is called “Translator”.



Definition:

“compiler is a system software that translate a code from one language to another”.

OR

Compiler is a system software that translate a program written in HLL into some other “Machine level language”.

Compiler have two steps.

1. Analysis.
2. Synthesis.

DECOMPOSITION OF COMPILER:

Four scenarios are here....

1. Analysis synthesis model.
2. Logical phases.
3. Front-end and Back-end.
4. Pass.

*** Analysis synthesis model:**

In the analysis synthesis model, the compiler is decomposed into two parts:

1. Analysis Part:

2. Synthesis Part:

Analysis Part::

In the analysis part, the source code is taken as input and analysed.

The following analysis are performed at this stage:

- * **Syntax analysis:**

In this analysis, we check the code either it is gramatically correct or Not?

- * **Lexicon analysis:**

In this analysis , we check the lexicon of the code.

- * **Sementic analysis:**

In this analysis, we check the code sementically, either meaning of this code is valid or not?

The analysis part translates the source code into some intermediate code.

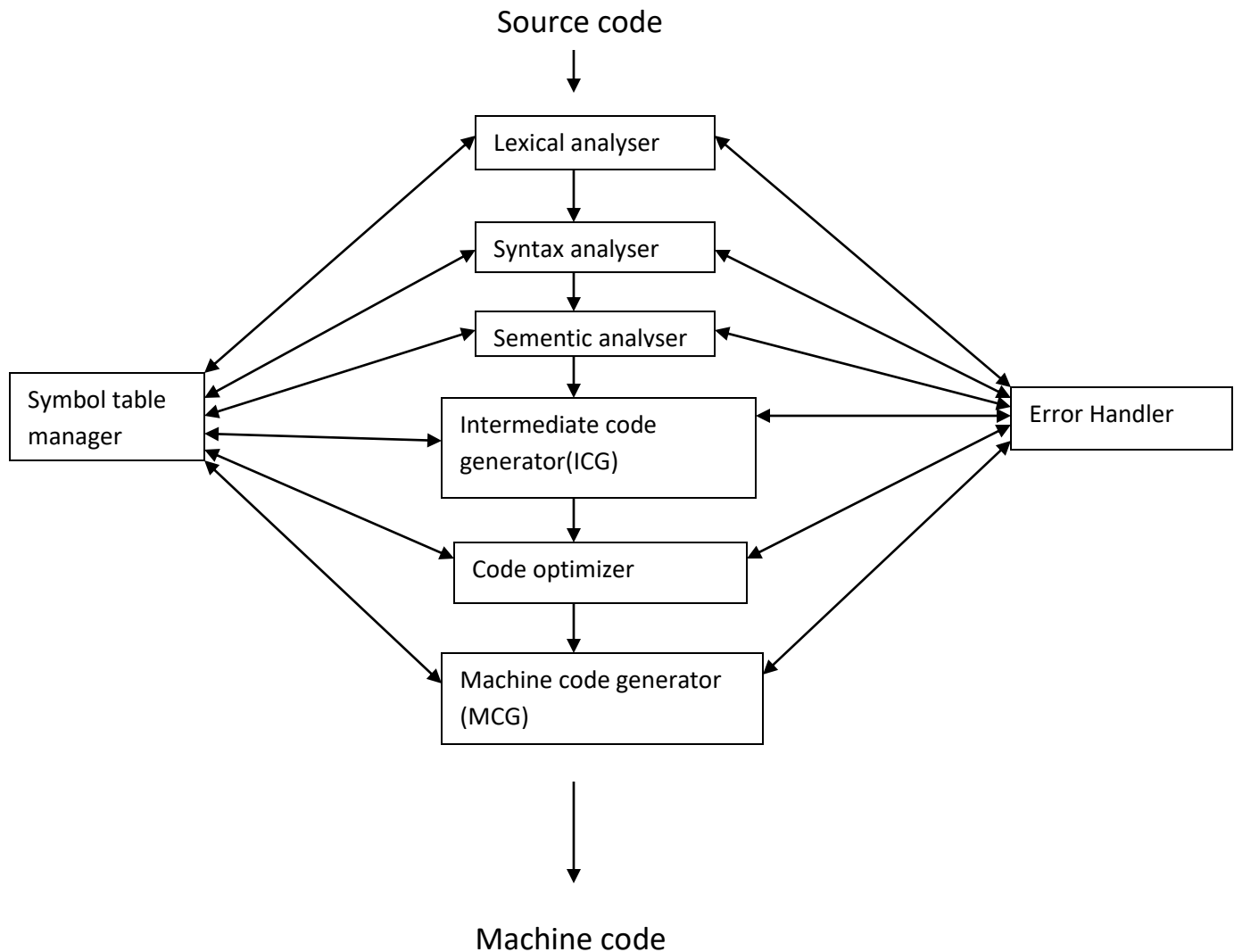
Synthesis Part::

Synthesis Part takes intermediate code as input, process it and translate it into machine code.

- * **Logical Phases:**

A compiler is decomposed into 8 different logical Phases:

PTO



1. Lexical analyser/scanner:

Scanner is responsible for that “it takes the source code, reads it, make tokenization and decompose the source code and construct the symbol table”. At the last scanner identify the lexical errors(occurs because of punctuation sign) in the source code.

Symbol table consists of informations such as : Name, datatype, space etc.

Here decompose means : what is the group of a given word?

Example:

We have an input as source code:

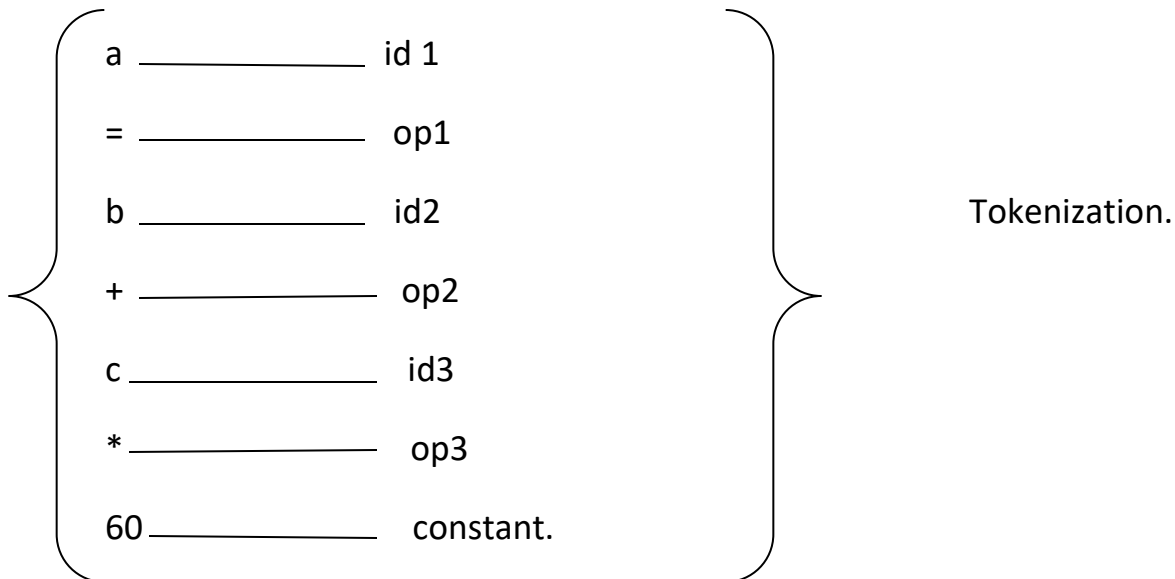
a=b+c*60;

[source code as input]

decomposition:

Here decomposition means which are identifiers, operators and which are constants in the above source code.

Now:



Now the structure will become:

id1=id2+id3*constant.

2. Syntax Analyser/ Parser:

Parser is the backbone of any compiler.

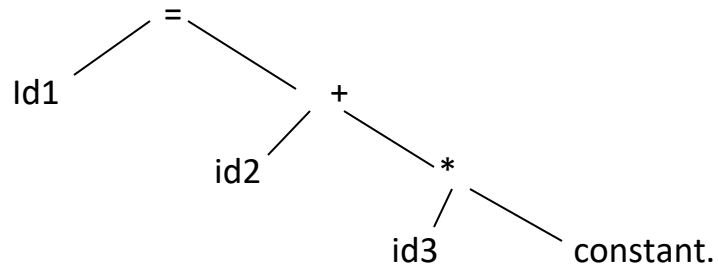
Parser detects the syntax errors from the code.. Much back-traking is used in parsing.

Parser should construct the **parse tree**.

“the graphical representation of derivation process is called Parse-tree”.

Now

id1=id2+id3*constant.



CFG for the above parse tree....

E-----> EOE

E----->(E)

E----->-E

O----->-/'/'/*/%

E----->id/constant.

Check either the following word is valid or not?

X=Y+/Z.

E----->EOE

E----->idOE

E----->id=E

E----->id=EOE

E----->id=idOE

E----->id=id+E.

Hence the given word is invalid.

3. Sementic analyser:

Sementic analyzer takes parse-tree as its input....

It performs the following tasks:

- * Type relevant information collection and store it in symbol table.
- * Type conversion.
- * Error detection.(sementic errors).

4. Intermediate code generator:

ICG is used to generate intermediate code from the parse-tree.

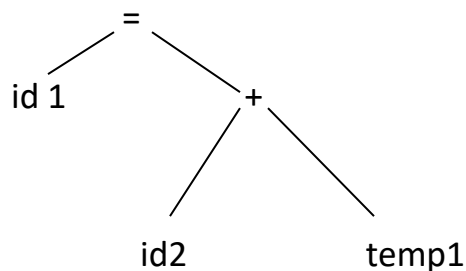
It performs its process in two steps.

Step1:

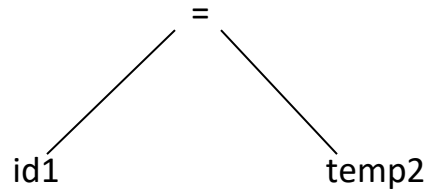
It performs the “war-through process” on parse-tree. Means that it starts from the highest level.

e.g.

Temp1= id3* const.



temp2= id2+temp1



id1=temp2

Above process is called “War through” process. And resultant code is called “intermediate code”.

id1=temp2

Step 2:

If there occurs an error, it detects the error and Handover that error to the Error Handler.

5. Code optimization:

This phase optimizes the intermediate code, means that it makes the code efficient.

Basic purpose of code optimization is to input that intermediate code.

Now

Temp2= id2+temp1

id1=temp2

so

id1= id2+temp1

it is called code optimization.

OR

```

for(int i=1; i<x+y; i++)
{
    }
  
```


Now optimize this code.

```
a=x+Y;  
for( int i=1; i<=a; i++)  
{  
  
}
```

Hence the above code is efficient.

6. Machine code generator (MCG):

Intermediate code is called in different languages:

In assembly language-----> intermediate code.

C language-----> object code.

In Java-----> Byte code.

Function of this phase is to convert data into machine code and make links and if there occurs an error, it deliver it to Error Handler.

There is pneumatic version of Machine code in which there is a Dictionary of acronyms.

MCG makes links with system function through addresses. It takes the addresses from vector table.

Now

Intermediate code is in three address code.

Temp1=id3*const.

id1= id2+temp1.

Let

Move Ax, const.

Multi Ax, id3

Move temp1, Ax

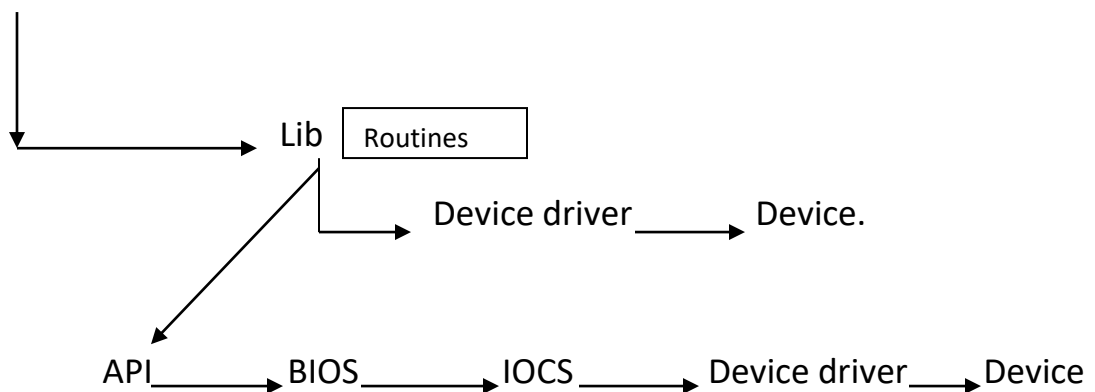
Move Bx, temp1

Add Bx, id2

Move id1, Bx

This is the complete machine structure.

Printf/cout



7. Symbol table manager:

STM is a two dimensional array used to hold the relevant information of all programming constructs.

Two phases enter data into the symbol table while other uses the data stored in symbol table.

(lexical, semantic analyser etc)

STM is a software module used to Handle the symbol table.

8. Error Handler:

During the code transformation, whenever there occurs an error at any stage, it will give into the custody of Error Handler.

Error Handler finds the nature and location of error and generate a proper Error report and displays it to the user.

FRONT-END AND BACK-END:

Front-end of compiler consists of all those phases that directly concerned with source language.

Front-end consist of lexical analyser, semantic analyzer and some part of ICG.

Backend of compiler consist of those phases that are concerned with machine language.

Backend consist of remaining part of ICG and code optimizer and MCG.

NOTE: Symbol table and error handler are included in both ends.

WHY DECOMPOSITION:

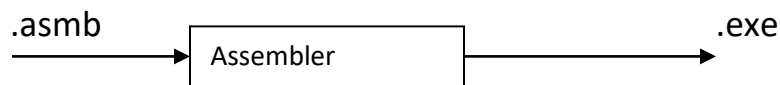
Decompose compiler into F.E & B.E increases the portability of compiler and reduces development burdon of compiler.

PASS::

Symbolic representation of machine language is called "Assembly language/pattern matching".

"Pass" means 'Go/turn'.

The word pass is used for Assembler.



Definition of Pass:

Pass we means "writing the output file once.

OR

Pass we means to perform a task in how many Go's..

Conventionally Assembler works in Passes.

There are two types of Assembler.

1. One pass assembler.

2. Two pass assembler.

- * **One pass assembler:**

it takes the .asm file as an input and translate it into the relevant .exe file in one go.

- * **Two pass assembler:**

It works in two passes.

In the first pass, the simple opcodes in the direct values are translated.

In second pass, the complex opcodes in the indirect values (variables) are translated.

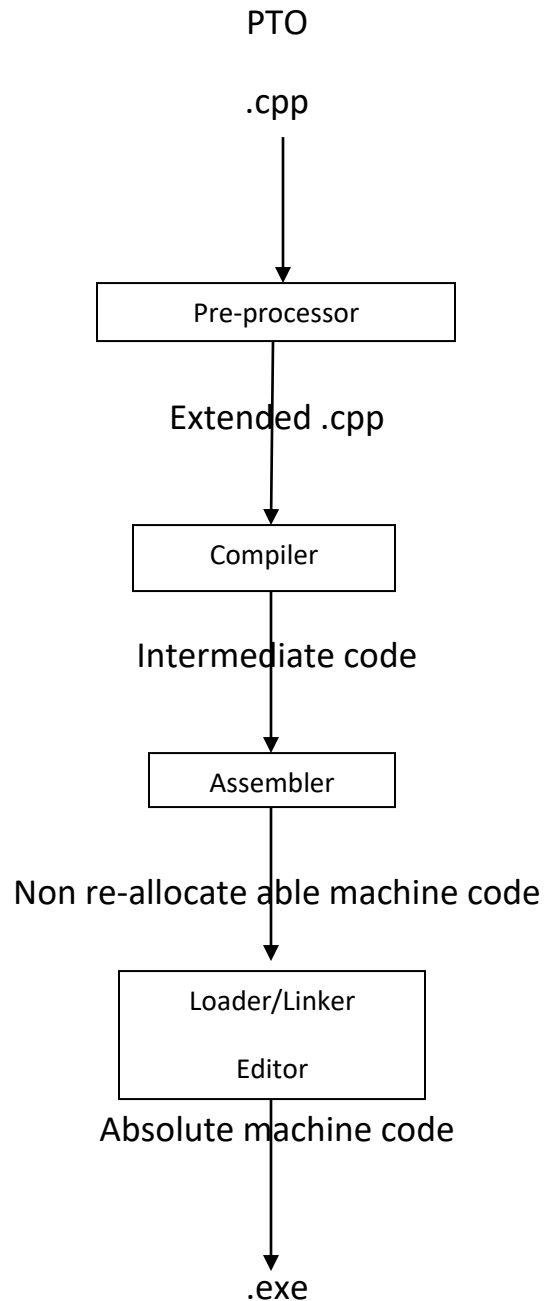
Note: Generally, all assemblers works in two passes.

.....

COUSIONS OF COMPILER/NEXT TO COMPILER

Compiler itself didn't do any task, it takes help of other softwares, called cousions of compiler.

When .cpp code has to transform to .exe code. It will passes through the following steps.



1. PRE-PROCESSOR::

Pre-processor is a software that takes the source file as input and process the pre-processor directives/commands before the compilation starts.

Pre-processor directives are identified by “#” symbol.

Important functions of pre-processor:-

- ✚ File inclusion.
- ✚ Macros.
- ✚ Language Extensions.
- ✚ Rational pre-processing.

➤ **File inclusion:**

File inclusion command is used to include a file from outside the world into our source code.

“# include” command is used for file inclusion.

➤ **Macros:**

Header file for macro is “# defines”.

define, is used for three different tasks.

1. Short Hand Notation:

Let

float pi=3.14159

if we need to use the “pi” repeatedly in a program, and if we write the value of “pi” instead of “pi” ..it generates an error.

So we write it as:

```
# define pi 3.14159
```

This is called “short Hand Notation”.

2. Renaming of reserve words:

Example:

```
# define      ab      cout
```

```
void main()
{
    ab<<"Rename"<<endl;
}
```

3. Inline one line function:

Functions that are replaced but not called are called "inline functions".

OR

Function in sequential line of execution called "inline function".

Example:

```
void main()
```

```
{
    -----
```

```
    -----
```

```
    -----
```

```
f1();
```

```
    -----
```

```
    -----
```

```
    ----- }
```

```
f1()
```

```
{
```

```
    ----
```

```
    ----
```

```
}
```

Now if we put the word "inline" with f1(), then f1() will become inline function.

Now

```
# define cube (a) a*a*a
```

```
{
```

```
    int x;
```

```
cout<<cube (x);           /* inline one line function */

}
```

Language extension by which we means to increase the capabilities of a compiler.

- (i) To add additional data structures into our language.
- (ii) To embed an outside compiler into our compiler.

Rational pre-processing is used to decide that what part of the program will be compiled and what part will be left un-compiled.

Loader is an operating system module which is responsible for loading a user program from secondary memory to main memory.

If we click the above file, on the first click it will be highlighted, means that it has loaded into the RAM.

Linker is also an operating system module and is responsible for making links between system routines and their relative calls.

If we type “printf” for the display,, Now here the compiler links this the folder of lib.

Then from lib, it will go to the IOCS and link it, then call the device driver and then device..

PARSING

“The process of constructing parse tree is called “parsing”.

Let a CFG is

A-----> aA

A-----> bB

B-----> bB

B-----> Ba

B-----> \emptyset

And a word is given for justification.....

aabbbba

Now

A----->aA

A-----> aaA

A-----> aabB

A----->aabbB

A-----> aabbbB

A-----> aabbbbB

A-----> aabbbbBa


A-----> aabbbb \emptyset a

A-----> aabbbba

Which is the required word.

TYPES OF PARSING:

 Top down parsing.

 Bottom up parsing.

*** TOP DOWN PARSING::**

In top-down parsing, the parsing process is started from the starting Non-terminal and the required word is justified.

Top down parsing are of two types.

1. Recursive Decent parsing.

2. Non-Recursive predictive parsing.

* **BOTTOM UP PARSING:**

In this approach, the process is started from the given sentence and it is constructed until we reach the starting Non-terminal.

TOP DOWN PARSING:::

1. **Recursive decent parsing:**

The normal top down parsing process is also called recursive decent parsing.

This process suffers from heavy recursion.

ELEMINATION OF RECURSION:

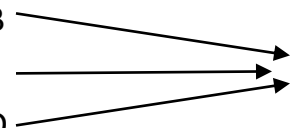
1. **Left factoring:**

Left factoring is the first step to reduce recursion in the parsing process.

In the left factoring, the common prefixes to the left are written once.

e.g.

A-----> αB
A-----> αC
A-----> αD



Left common prefixes.

So

A-----> $\alpha A'$

A'-----> B/C/D.

It is known as Left factoring.

2. **Left Recursive:**

A grammar is said to be left Recursive if it is of the form.

A-----> $A\alpha$

A-----> β

A-----> $A\alpha$

$A \rightarrow A\alpha\alpha$

$A \rightarrow A\alpha\alpha\alpha$

.

.

.

The Net Recursion is avoided by making a transformation of the form.

$A \rightarrow \alpha A'$

$A' \rightarrow \beta/\epsilon$

=====

Now if

$A \rightarrow A\alpha$

$A \rightarrow \beta$

From the above productions, we can remove recursion if we make a transformation of the form:

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A'/\epsilon$

Let we have

$E \rightarrow E+T$

$E \rightarrow T$

Now check if the left most “E” repeats itself then there is left recursion.

Here

$A=E$

$$A' = E'$$

$$\alpha = +T$$

$$\beta = T$$

so

$$A \longrightarrow \beta A'$$

$$A' \longrightarrow \alpha A' / E$$

$$E \longrightarrow TE'$$

$$E' \longrightarrow +TE'$$

Let we have another example:

$$X \longrightarrow Xby$$

$$X \longrightarrow a$$

Eliminate Left Recursion:

$$A = X$$

$$A' = X'$$