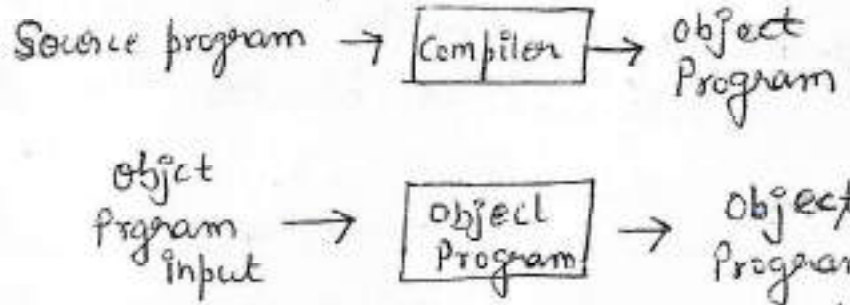
Unit- I

Translator: → A translator is a kind of program that takes one form of program as input and converts it into another form. the input program is called source language and the output program is called target language.

If source language is a high level language such as FORTRAN, PL/I, COBOL, C and C++, and the object laguage is low-level language or m/c language, such a translator is called a Compiler.

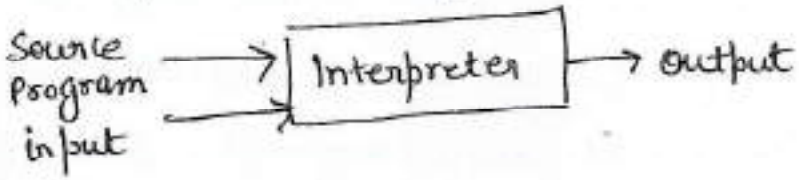
Executing a program written in HLL is basically a two-step process. in first, the source program must first be compiled, that is translated into object program. then the resulting object program is loaded into memory and executed.



Compilation and execution

Note: Compiler and Assemblers are two translators Compiler converts HLL to m/c language while assembler converts assembly language to m/c language.

An Interpreter: is another kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on input supplied by the user.



An Interpreter

Note: →

(1) A m/c language target program produced by a compiler is much faster than an interpreter.

(2) An interpreter can usually give better error diagnostics than a compiler because it executes the source program statement by statement.

Imp Note: An interpreter, like a compiler, translates high level language into low level m/c language. the diff^n lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, check semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a stmt from the input, convert it to an intermediate code, execute it, then takes the next stmt in sequence. if an error occurs, an interpreter stops execution & reports it; whereas a compiler reads the whole program even if it encounters several Errors.

Compiler: Analysis - Synthesis Model:→

- The compilation can be done in two parts: 1. Analysis Phase (dividing source code into small chunks called tokens)
  2. Synthesis Phase (we group all the intermediate code module into corresponding object/target code)

A compiler can broadly be divided into two phases based on the way they compile.

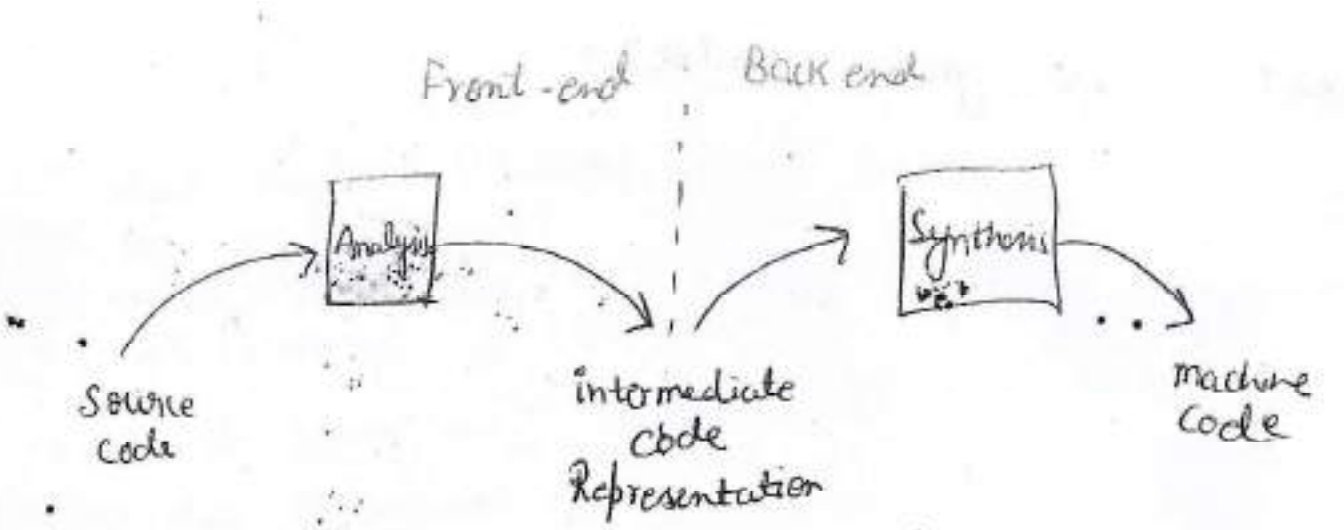
The Analysis phase known as front-end of the compiler. the Analysis part is carried out in three sub parts:→

Analysis of Source Program

- Lexical Analysis (In this source program is read & then it is broken into stream of strings. Such string are called tokens)
- Syntax Analysis (In this step the tokens are arranged in hierarchical structure that helps in finding the syntax of the source string)
- Semantic Analysis (In this step the meaning of the source string is determined)

After carrying out the synthesis phase the program gets executed.

The Analysis phase generates an intermediate representation of the source program and symbol table, which ~~soul~~ should be fed to the Synthesis phase as input.

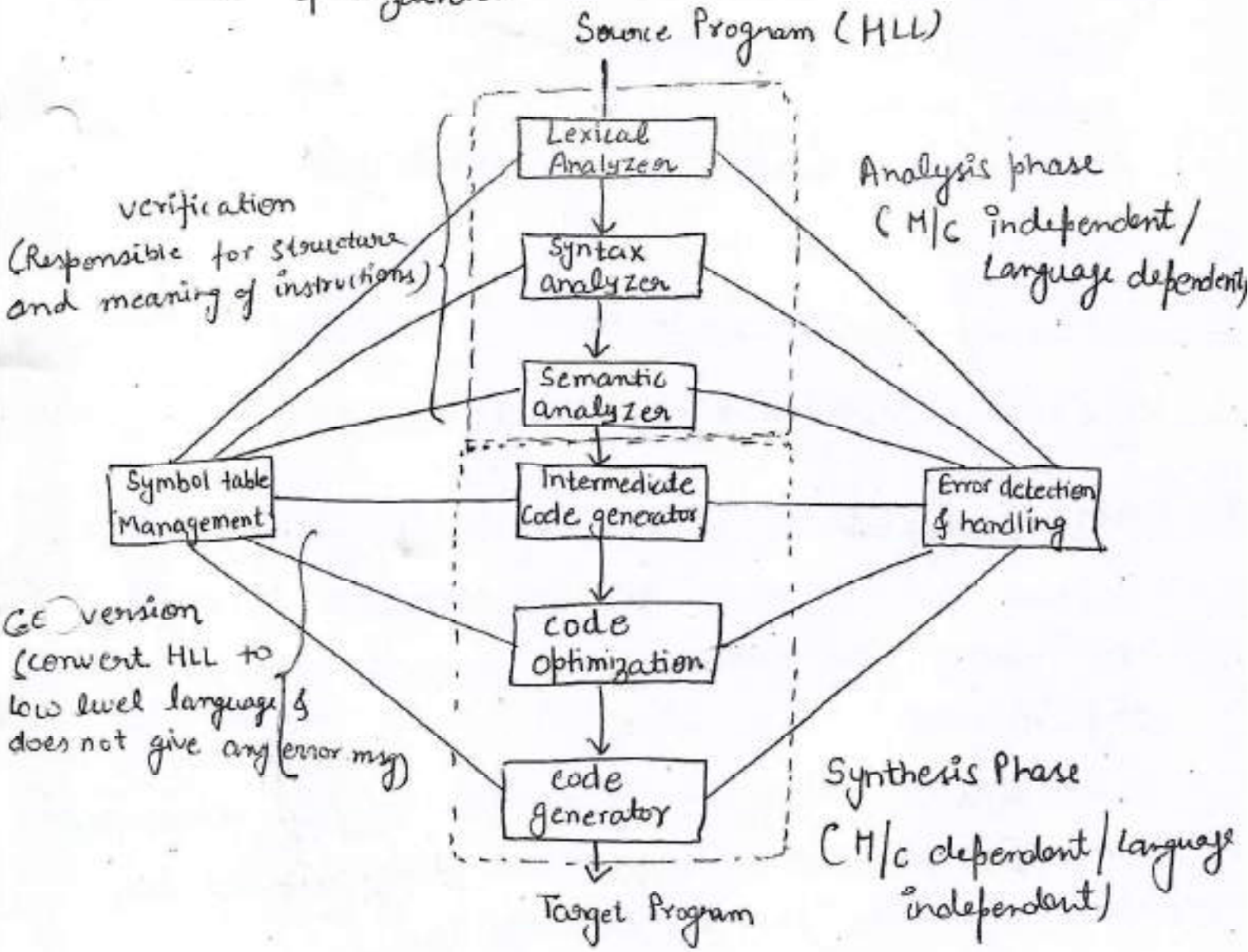


[working principle of compiler]

Synthesis phase also known as back-end of the compiler, the synthesis phase generates the target program with the help of intermediate source code representation and symbol table.

Phases of Compiler (Structure of Compiler)

As we have discussed earlier the process of Compilation is carried out in two parts: Analysis and Synthesis. Analysis is carried out in three phases: lexical Analysis, Syntax analysis and semantic Analysis. And the synthesis is carried out with the help of intermediate code generation, code generation and code optimization.



1. Lexical Analysis it is also called Scanning.

it reads the program and converts it into tokens. it converts a stream of lexems into a stream of tokens. tokens are defined by regular expression which are understood by the lexical analyzer. it also removes white spaces and comments.

- Token :→ are keywords, identifiers, operators and punctuation symbols such as parantheses or commas.

input: Stream of characters

Output: token

token template: <token_name, attribute_value>

ex: C = a + b * 5

↓ ↓ ↓ ↓ ↓ ↓ ↓

id op id op id op id

Hence, <id,1><=><id,2><+><id,3><*><5>

2. Syntax Analyzer: it is called parser. it constructs the parse tree. it takes all the tokens one by one and uses context free grammar to construct parse tree.

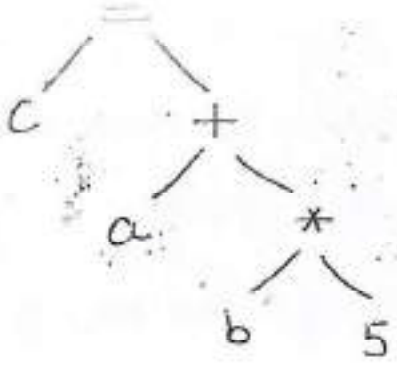
Why Grammar?

The rules of programming can be entirely represented in some few productions. Using these productions we can represent what the program actually is.

Syntax error can be detected at this level if the input is not in accordance with the grammar.

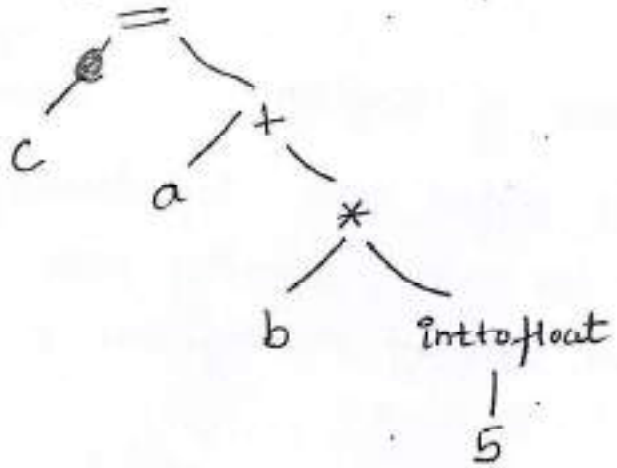
input: tokens

output: Syntax tree



3. Semantic Analysis

Semantic Analysis checks whether the parse tree constructed follows the rules of language. whether it is meaning or not. it also performs type conversion of all the data types into real data types



4. Intermediate Code Generator: it generates intermediate code, that is a form which can be readily executed by m/c we have many popular intermediate codes. ex:- Three address code etc. Intermediate code is converted to m/c language using at least two phases which are platform dependent.

$t_1 = \text{int to float}(5)$
$t_2 = id_3 * t_1$
$t_3 = id_2 + t_2$
$id_1 = t_3$

Note: The primary difference b/w intermediate code and assembly code is that the intermediate code need not specify the registers to be used for each operation.

5. Code Optimization: * is an optional phase designed to improve the intermediate code so that the object program runs faster and/or takes less space.

* it can be done by reducing the no: of lines of code for a program
* During code optimization, the result of the program is not affected.

$$t_1 = id_3 * 5.0$$
$$id_1 = id_2 + t_1$$

```
MOVF  id3  R2
MULF  #60.0 R2
MOVF  id2  R1
Add   R2   R1
MOVF  R1   id1
```

6. Code Generation:

* it is final phase of compiler.
* it produces the object code by deciding on the memory location for data, selecting code to access each datum and selecting the registers in which each computation is to be done

Symbol Table Management: (Book-keeping)

* Symbol table is used to store all the info about identifiers used in the program.
* It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
* whenever an identifier is detected in any of the phase, it is stored in the symbol table.

```
Int a,b;
float c;
char z;
```

Symbol	type	address
a	int	1000
b	int	1002
z	char	1004
c	float	1008

Symbol handling:

* the attributes of identifiers are usually its type, its scope info^n about the storage allocated for it.

```
extern double test(double x);
   double sample(int count)
{
   double sum = 0.0
   for(int i=1; i<= count; i++)
      sum = sum + test(double(i));
      return sum;
}
```

Symbol Name	Type	scope
test	function, double	extern
x	double	function parameter
sample	double, func	global
count	int	function parameter
sum	double	block local
i	int	for loop statement

Note:

Error Handling:

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

* In lexical Analysis, errors occur in sepration of tokens (identifier typed incorrectly).

* In Syntax Analysis: it includes missing semicolon or unbalanced parenthesis, So parser should be able to detect and report those errors in the program.

* In Semantic Analysis: errors may occur in the following cases:
  * type mismatch
  * Undeclared variable
  * Multiple declaration of variable in a scope
  * Accessing an out of scope variable.
  * Actual and formal parameter mismatch.

* In code optimization: error occur when the result is affected by the optimization.

* In code Generation: when code is missing etc.

Difference Between Compiler And Interpreter

* A compiler converts the High level instruction into m/c language while interpreter converts the high level instruction into an intermediate form.
* Before execution, entire programs is executed by the compiler whereas after translating the first line, an Interpreter then execute it and so on.
* List of errors is created by the compiler after the compilation process while an interpreter stops translating after the first error.
* An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.
* The Compiler produce object code whereas interpreter does not produce object code.
* In the process of compilation the program is analyzed only once and then the code is generated whereas source program is interpreted every time it is to be executed and every time the source progeram is analyzed. Hence interpreter is less efficient than compiler.
* Example of compiler : Borland C compiler or Turbo C compiler, gcc

  Example of interpreter: A UPS debugger, Ruby, Python, Php, LISP, MATLAB

* JAVA programs are first compiled to an intermediate form, then interpreted by the interpreter.

Bootstrapping:

In CS, bootstrapping is the technique for producing a self compiling compiler.

Compiler is characterized by 3 languages
- Source language
- Object language
- Language in which compiler is written.

Bootstrapping is concept of obtaining compiler for a language by using compilers for less powerful subset of the same language.

*

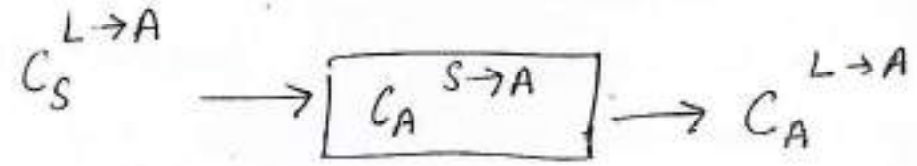


fig-1

L → Source language

A → Object language

S → Language in which compiler is written, S is subset of L.

* We write a compiler $C_S^{L\to A}$ in the simple language S. this program, when run through $C_A^{S\to A}$, becomes $C_A^{L\to A}$, the compiler for the complete language L, running on m/c A. and producing object code for A. process shown in fig 1.

Cross Compiler: It is a compiler which runs on one m/c and generate the code for another m/c.

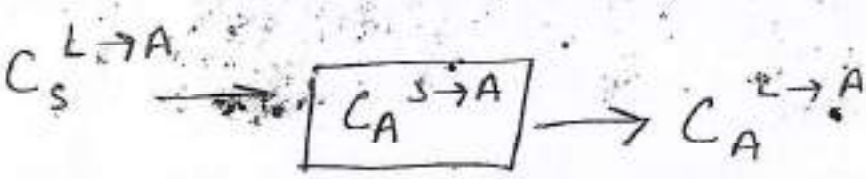
$C_X^{L\to Y}$

Cross compiler [A compiler that runs on platform and is capable of generating executable code for platform is called a cross-compiler.]

Here X and Y are diff^m machines. X is generally a larger m/c in comparision to Y m/c.

This permits use of high level language for small m/c.

Transporting Compiler from One Computer A to another Computer B

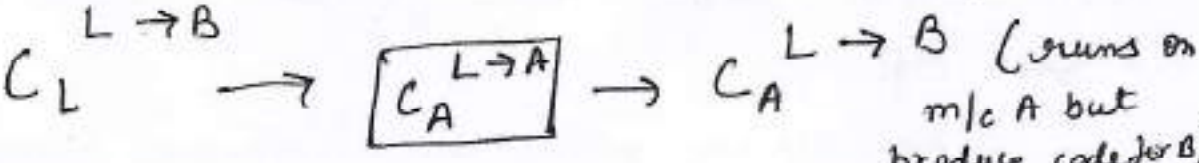


L → High level language
S → Subset of L
A → m/c language of computer A.

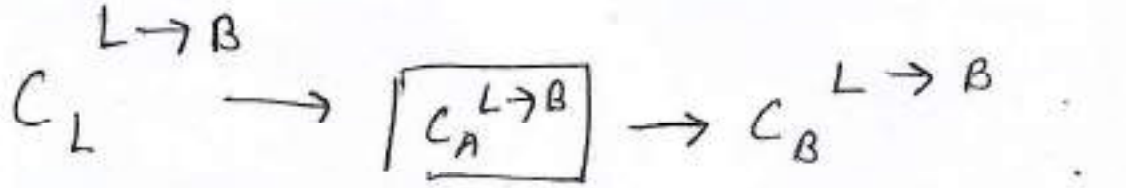
Now suppose we want to produce another ~~Language~~ compiler for L to run on m/c B and to produce code for B.

Now with very little efforts $C_S^{L \to A}$ can be converted to $C_L^{L \to B}$ (compiler written in lang L to convert program of lang. L into m/c lang. of B)

Now using compiler of lang. L on m/c A we can obtain cross compiler $C_A^{L \to B}$



Now using this cross compiler we can obtain a compiler for language L on m/c B



Compiler Construction Tools:

writing a compiler is tedious and time consuming task, there are some specialized tools for helping in implementation of various phases of compilers. these tools are called Compiler Construction tools. These tools are also called as compiler-compiler, compiler-generators, or translator writing systems, which produce a compiler from some form of specification of a source language and target m/c.

the input specification for these systems may contain:

* a description of the lexical and syntactic structure of the source language.
* a description of what o/p is to be generated for each source language construct.
* a description of target m/c.

Various compiler construction tools are given as below.

1. Scanner Generator – These generators generate lexical analyzers. the specification given to these generators are in the form of regular expressions.

   the UNIX has utility for a scanner generator called LEX. The specification given to the LEX consists of regular expressions for representing various tokens.

2. Parser generators: These produce the syntax analyzer. The specification given to these generators is given in the form of CFG.

   UNIX has a tool called YACC which is a parser generator.

3. Syntax-directed translation engines:

In this tool the parse tree is scanned completely to generate an intermediate code. the translation is done for each node of the tree.

4. Automatic code Generator:

These generators take an intermediate code as input and converts each rule of intermediate language into equivalent m/c language.

5. Data flow engines: The data flow analysis is required to perform good code optimization, the data flow engines are basically useful in code optimization.