Compiler. Construction tools

Compiler Construction tools can use Morden s/w dvmt., environments Containing tools such as

- language editors
- debuggers
- Version Managers.
- profilers
- text harnesses.

and so on.

general s/w dvmt., tools.

* Along with the above tools, other more specialized tools have been created to help implement Various phases of a Compiler.

* These tools use ~~Specifili~~ Specialized PLs for implementing Speific Components using many sophisticated and Complex algorithms.

* Most of the tools provide high degree of abstraction Means the details of implementation is hedden from outside world.

Compiler Construction tools (or) Writing tools

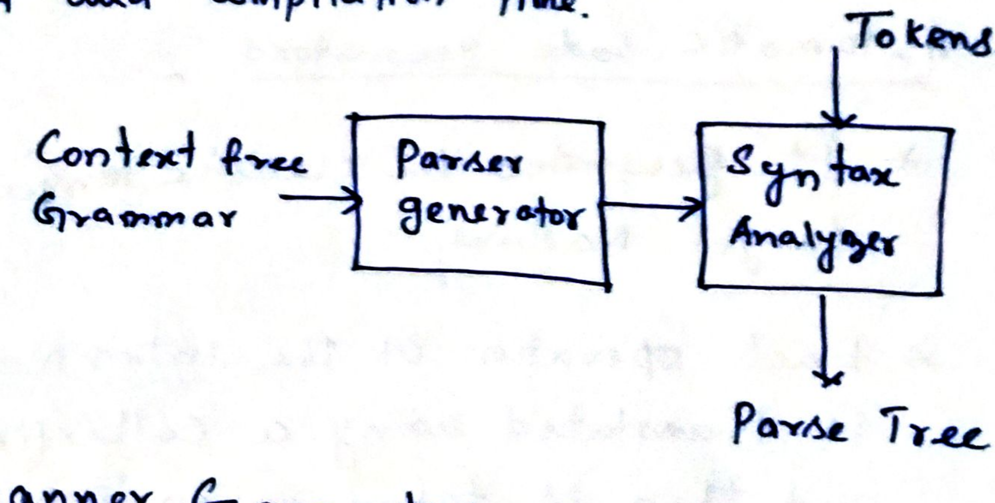
* The Compiler writer can use some specialized tools that help in implementing various phases of a Compiler.

* These tools assist in the creation of on entire Compiler or its parts.

* Some Commonly used Compiler Construction tools include.:

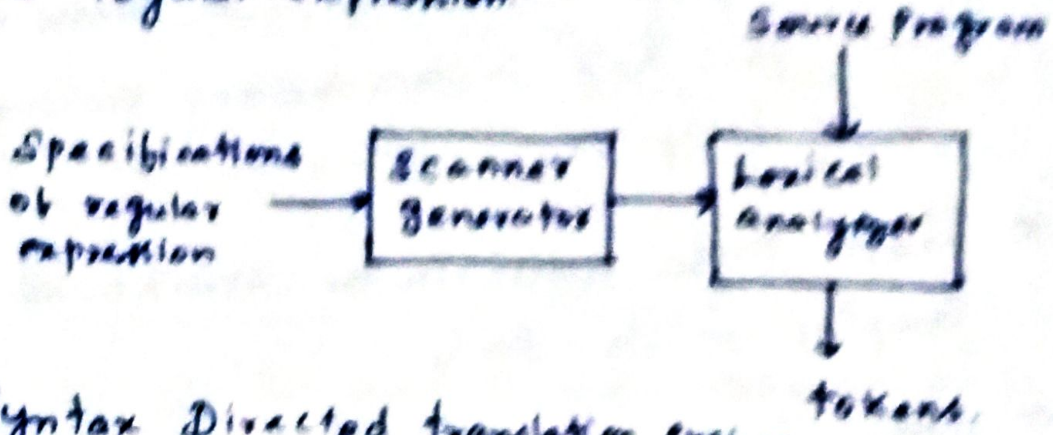
1. Parser Generator

It produces Syntax analyzers (Parsers) from the input that is based on a grammatical description of PL or on a Context free grammar. It is useful as the Syntax analysis phase is highly Complex and Consumes more Manual and and Compilation time.



2. Scanner Generator

It generates lexical analyzers from the input that Consists of regular expression description based on tokens of a language. It generates a finite automaton to recognize the regular expression.



3. Syntax Directed translation engines

- It generates intermediate code with three address format from the input that consists of a parser tree.

- These engines have routines to traverse the parse tree and then produces the intermediate code.

- In this each node of the parse tree is associated with one or more translations.

4. Automatic Code generators

- It generates the Machine language for a target Machine.

- Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator.

- * A template matching process is used.

- * An intermediate Language Statement is replaced by its equivalent Machine language Statement using templates.

5. **Data flow analysis engines.**

- * It is used in Code optimization.
- * Data flow analysis is a Key part of the code optimization that gathers information, that is the values that flow from one part of a program to another.

6. **Compiler Construction toolkits.**

- * It provides an integrated set of routines that aids in building Compiler Components in the Construction of various phases of Compiler.

Ex_2

$x = a + b * c;$

⇓

$l(l+d)^* \rightarrow$ Lexical Analyser.

Pattern.

→ once the LA receives this i/p., the i/p is Converted into a Strea of tokens. Not only this, it removes the white Spaces and skipping the Comments.

⇓

id. = id + id * id

⇓

Syntax Analyser ← Contex free Grammar.

S → id = E ;

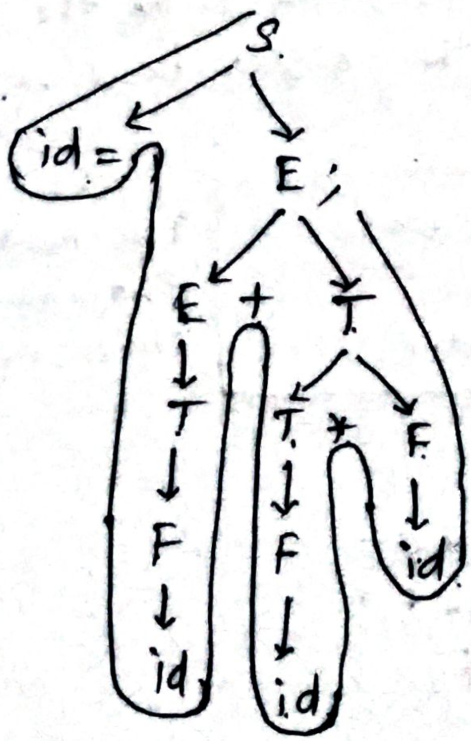
E → E + T / T

T → T * F / F

F → id.

rules.

⇓



Parse Tree.

The Rules the PL can be entirely represented in some Productions rules.

S → A statement can be identifier = expression followed by ;

E → An expression can be Expression + Term or Term.

T → A Term can be Term * Factor or Factor

F → can be an identifier.

The Syntax error is detected by Syntax Analyser if the input is not according to the Grammar given.

⇓

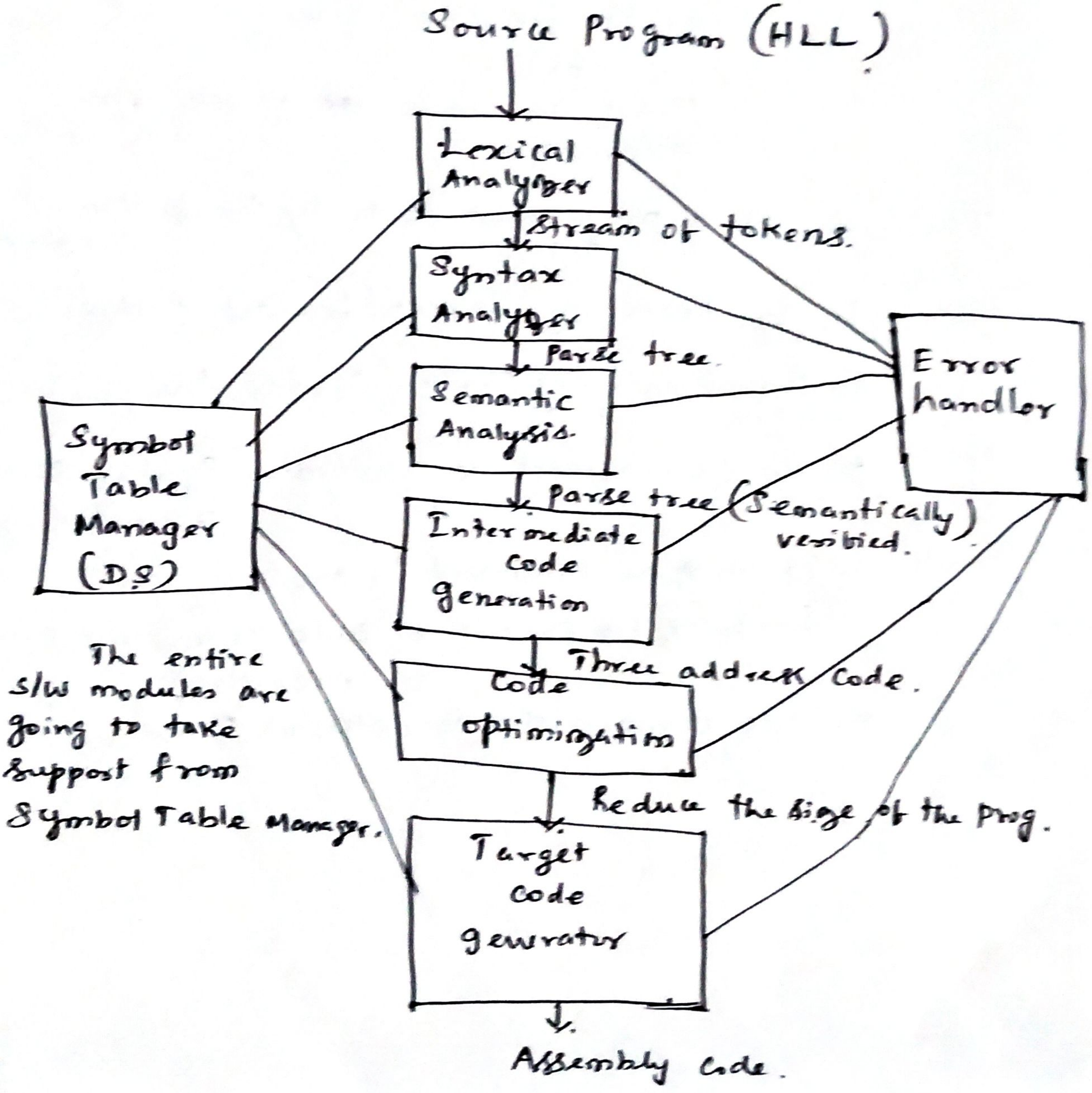
Semantic Analyser

⇓

Phases of ~~Computer~~. Compiler

Phase ?

is a logically interrelated operation that takes source prog. in one representation and produce ~~of~~ output in another representation.



The entire s/w modules are going to take support from Symbol Table Manager.

Explanation with example

Ex.

Position:= initial + rate * 60 ; → Source Prog.

⇓

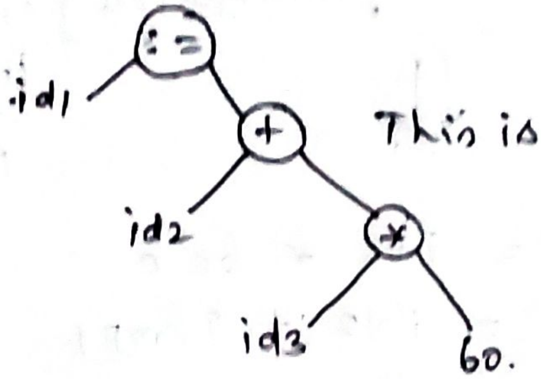
Lexical Analysis — divide the SC into no. of tokens.

id1 = id2 + id3 * 60

⇓

Syntax Analysis

⇓



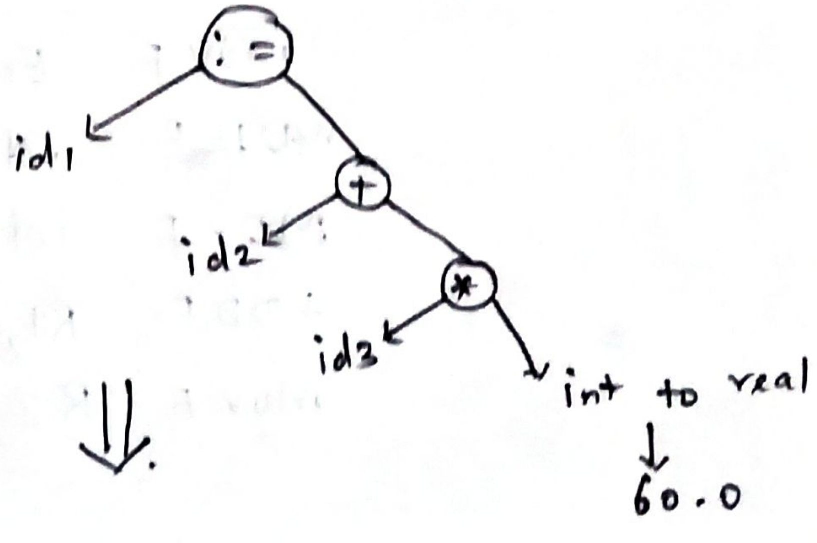
This is a Parse Tree. generated by Syntax Analyser

⇓

Semantic Analysis — checks the Meaning or rules

⇓

Suppose <u>rate</u> is float type but 60 is int. So, convert 60 into float.



⇓

⇓

Intermediate Code Genatation. — Generates a three address code.

⇓

```
Temp1 = int to real(60)
Temp2 = id3 * Temp1
Temp3 = id2 + Temp2
id1   = temp3
```

} Three address Code

⇓

Code optimization — removes unnecessary information to increase the speed of the Program.

⇓

```
temp1 = id3 * 60.0
 id1  = id2 + Temp1
```

⇓

Target code generator

⇓

```
MOV F   60.0, R1
MUL F   id3, R1
MOV F   id2, R2
ADD F   R1, R2
MOV F   R2, id1.
```

} Assembly Language Prog.

Ex-1

x = a + b * c ;

⇓

l(l+d)* → Lexical Analyser

Pattern

Once the L.A receives the I/P, the I/P is converted into a stream of tokens. Not only this, it removes the white spaces and skipping the comments.

⇓

id = id + id * id

⇓

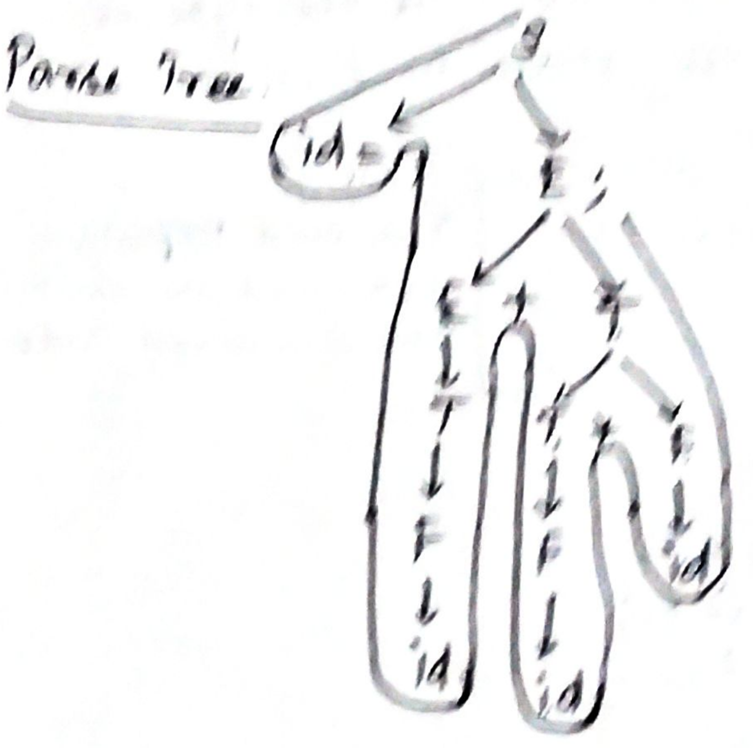
Syntax Analyser ← Context free Grammar

S → id = E ;
E → E + T / T
T → T * F / F
F → id.

rules.

⇓

Parse Tree



The Rules the CFG can be entirely represented in some Productions rules.

S → A statement can be identifier = expression followed by ;

E → An expression can be Expression + Term or Term.

T → A Term can be Term * Factor or Factor

F → can be an identifier.

The Syntax error is detected by Syntax Analyser if the input is not according to the Grammar given.

⇓

Semantic Analyser

⇓

Semantic Analyser is going to veriby the parse Tree whether the Parse tree is semantically correct or not.

eg

id = id + id * id.

↓

left hand side should be a variable can't be a constant or can't be a fun name or can't be an array name.

So, ~~left~~ left hand side has to be a variable which is compatible with the type of variable in the right hand side.

⇓

Intermediate Code Generator

The most popular Intermediate code is three address code.

⇓

$t_1 = b * c;$

$t_2 = a + t_1;$

$x = t_2;$

⇓

Code optimization

→ reduce the ~~lin~~ no. of lin

⇓

$t_1 = b * c;$

$x = a + t_1;$

⇓

TCG

⇓

MUL	R1, R2	a → R0
add	R0, R2	b → R1
MOV	R2, X	c → R2

Compiler Design.

Importance of Subject

* Compilers provide you with the theoretical and practical knowledge that is needed to implement a Prog. Language.

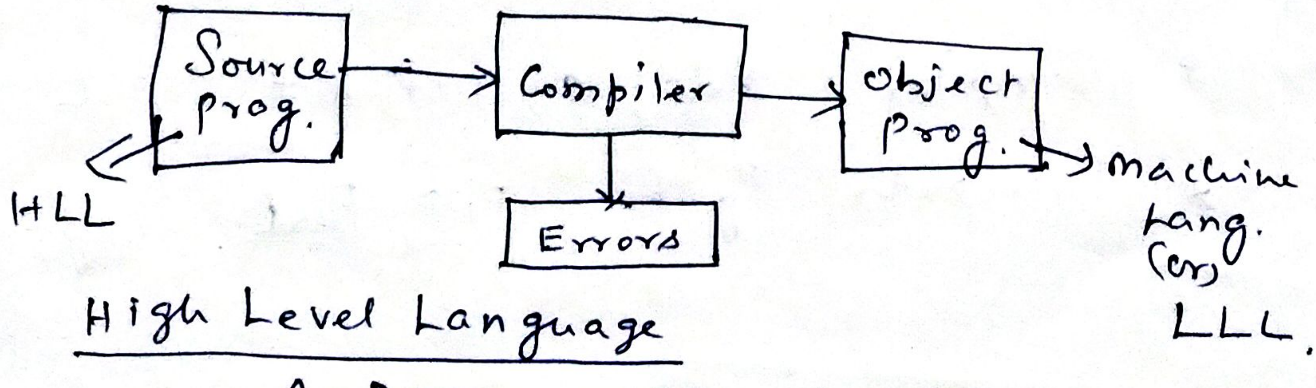
Contents

* Structure of Compiler
* Interpreter
* Compiler Vs Interpreter
* Language Processing System.

Introduction to Compiler.

A Compiler is a translator / s/w that Converts the program written in High-level language (Source Language) to Low Level Language (Object / Target / Machine Language).

Block Diagram.



High Level Language

A Program written in English Language English Language is familer with us to understand easily.

We can easily understand the Syntax, Simantic, and Structure.

* Machines do not understand HLL. They can understand LLL.
So Compiler Converts HLL to LLL.

Compiler and Interpreter.

* To Convert Source Code (SC) into machine code, we use either a Compiler or an interpreter.

* Compiler transforms Code written in a high-level Prog. Lang. into the Machine Code at once before the Prog. runs.

* where-as an interpreter Converts each high level program statement, one by one into the Machine Code, during execution of prog.

* Compiler : Ada, C, C++, C#, COBOL.

* Interpreter : Python, PHP, Perl, Ruby.

Compiler Vs Interpreter.

Compiler	Interpreter.
* Scans the entire Prog. and translates it as a whole into Machin code.	* Translate prog., one St., at a time.
* Take a large amt., of time to analyze the SC.	* Take less amt., of time to analyze the SC.
* The overall exe., time is Comparatively faster than interpreters.	* The overall exe., time is Comparatively slower than Compilers.
* Requires More Memory (generates intermediate code, so it requires more amts of memory)	* Memory efficient. (No need to generate intermediate code.)

Have you heard any PL name which uses both Compiler and Interpreter?

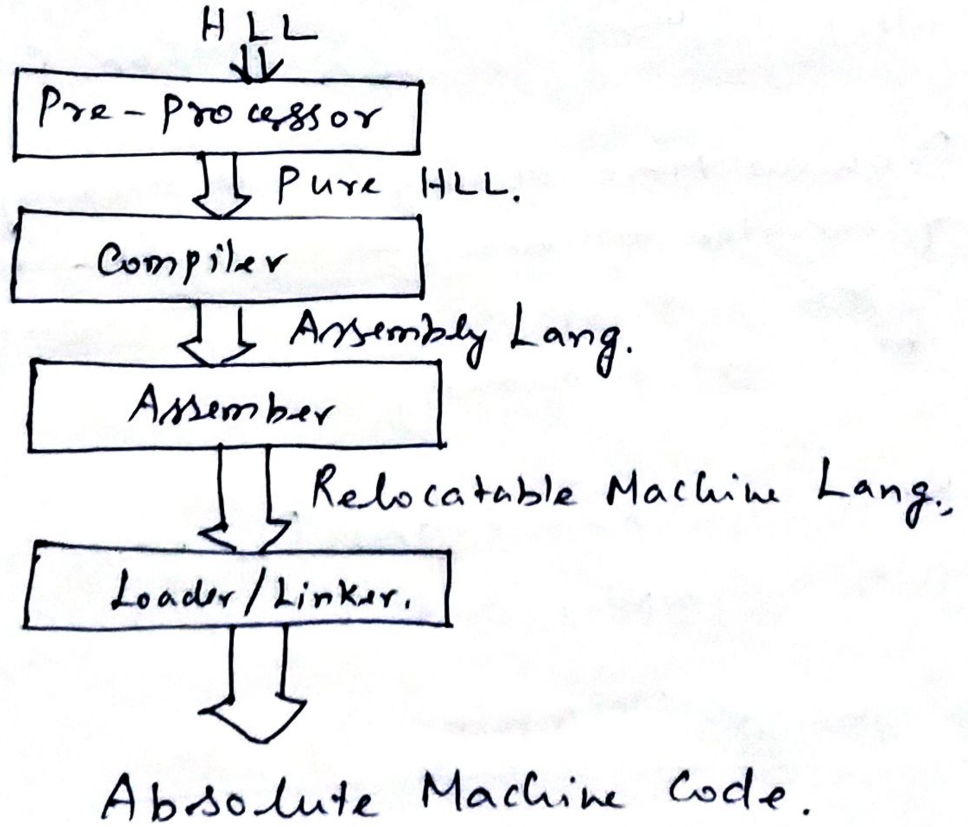
Language Processing System.

The task of compilation will not be done by Compiler along. There are other s/w modules which will accompany compiler in this process.

So, in order to know what are the s/w modules, we ~~need~~ need to know what is Lang., Processing System. is.

Language Processing System Consists of 4 s/w modules.

1. Pre-Processor
2. Compiler
3. Assembler
4. Loader / Linker.



Let us have a detailed descriptions about each s/w Modules.

1. Pre-Processor

The first s/w module is preprocessor whose input is HLL and output is pure HLL.

Now what is this pure HLL?

Consider a C program. Basically your C prog should start with a line # include <stdio.h> and # include <math.h>.

These header files are also called as pre processor Directives

Removes :-

Directives like

include

define

Performs :-

Macro expansion

File Inclusion

Function calling is over head but

Macro calling is simple

The Pre Processor is going to remove the above # include, # define by including files related to them. This is called as File inclusion

ie whatever the files you want to include, preprocessor is going to substitute that the entire files in your source Prg.

Pre processer will also do Macro expansion

2. Compiler.

The input to the Compiler is Pure HLL which means the prog. will not contain any # lines (or) tags.

Compiler is going to Convert Pure HLL into assembly Language.

This Assembly Lang. is not entirely 0s and 1s and not entirely HLL.

It is somewhat intermediate.

3. Assembler

Assemblers of one platform will not work with any other Platform

The o/p of the Assembler will be machine Codes.

Basically Machine codes of two types.

1. Relocatable Machine Code
2. Absolute Machine code.

Relocatable Machine Code is loaded at any Point of your Computer. and You can run it.

This ~~leto~~ Relocatable Machine Code is given as input to the next phase Loader/linker.

4. <u>Loader/Linker.</u>

The linker will link variet of object files into one file and the loader will loade this ~~into~~ files into the memory.

By this the execution of prog will be Completed.

Now we will focus on Compiler and not on all s/w modules.

Before discussing the phases of a Compiler, Let us know some thing about

- word
- Sentences
- Languages

* **Word.** can be defined as a set of chars. which gives a meaning.

eg "boy" → word.

→ defined from the set of chars from the alphabets available in English Lang.

→ It denotes Masculine gender child.

* **Sentences**

is a set of words which gives a meaning.

eg "The boy is running"

- The sentence should follow the grammar
- Here the Grammar defines the way in which the sentence can be formed.
- If a sentence does not follow any Grammar, then it can be called as a grammatically incorrect sentence.

* **Languages**.

- The Lang., is a tool used to Communicate with others
- It is defined over the sentences in turn words, which in turn defined over the chars.

The following example will help us in understanding Concepts of Compiler Construction.

Consider the following Sentence in English.

I AM GOING TO MARKET.

I → Subject

AM → Auxiliary verb.

Goting → Verb + ing

To MARKET → object.

This simple sentence is Syntactically, Grammatically Correct.

eg The boy is going to hastel

↳ This Sentence is not Syntactically correct.

⇓ Syntax error.

eg The boy is go to hostel.

⇓ Grammatically incorrect.

we have a Sentence which is Syntactically and Grammatically Correct.

"The boy is going to hostel"

Now what is to be done?

The Answer is, The boy must do some action.

ie The boy must go to the hostel.

The action is attached to the Sentence

∴ It is clear that an action is associated with a Sentence which is Syntactically, Grammatically Correct.

The above simple analogies will explain how the ~~computer~~ Compiler works in very simpler terms.

Language	→	Any HLL.
word	→	String
Sentence	→	Statement in a HLL.
set of Sentences	→	Set of Statements ie a Prog. in a HLL.

To carryout certain task a sentence must be written, which should satisfy the foll:

1. Syntactically Correct
2. Grammatically Correct
3. An Action must be associated
4. It must be understood by the Executor for execution.
   (or)
   "must be made ~~for~~ ready for execution.".

The St, of a prog. must satisfy the foll: in a Compiler.

1. Syntactically Correct (Lexical Analyser)
2. Grammatically Correct (Syntax Analyser)
3. An action must be associated (Syntax Direct Translation)
4. It must be understood by the Executor for execution (Code generation and execution)

* Code optimization is optional phase.

Symbolic Assembly Language

* The Most immediate step away from machine Language is Symbolic Assembly Language.

* In this Language, a programmer uses Mnemonic names for both operation Code and data addresses.

* Thus a programmer Could write

ADD X, Y. → Assembly Language,
⇓ ⇓ ⇓
instead of 0110 001110 010101 → Machine Language.

* A Computer can't execute a program written in assembly Language.

* That program has to be first translated to Machine Language, which the Computer can understand. The program that performs this translation is the ~~assembly~~. Assembler.

*

Macros

* A Macro facility is a text replacement capability.

* There are two aspects to Macros.
  - definition
  - Use.

To illustrate the utility of Macros, Consider the foll. situation.

Macro definition with two-address ADD inst.

```
MACRO        ADD2   X, Y
             LOAD   Y  }
             ADD    X  }  These three Sts
                       }  define the Macro.
             STORE  Y  }  ie they give its
ENDMACRO.                 translation.
```

ADD2 → name of the MACRO.

X, Y → dummy arguments (formar Parameter)

Having Defined ADD2 in this way, we can then use it as an ordinary assembly Lang., op-code.

For example

If the St., ADD2 A, B is encountered somewhere after the definition of ADD2, we have a Macro Use.

Here the Macro processor substitutes for ADD2 A, B the three St., which form the definition of ADD2, but with the actual parameters A and B replacing the formal parameters x and y respectively.

~~Thus~~ That is ADD2 A, B is translated to

LOAD B → Moves a datum from memory to a register.

ADD A → ADDs the contents of a memory address. to that of a register.

STORE B → moves data from a register to Memory.

Bootstrapping

- a process by which Simple language is used to translate more complicated Program, which in turn may handle for ~~an even~~ More Complicated program. This Complicated program can further handle even more complicated prog. and so on.

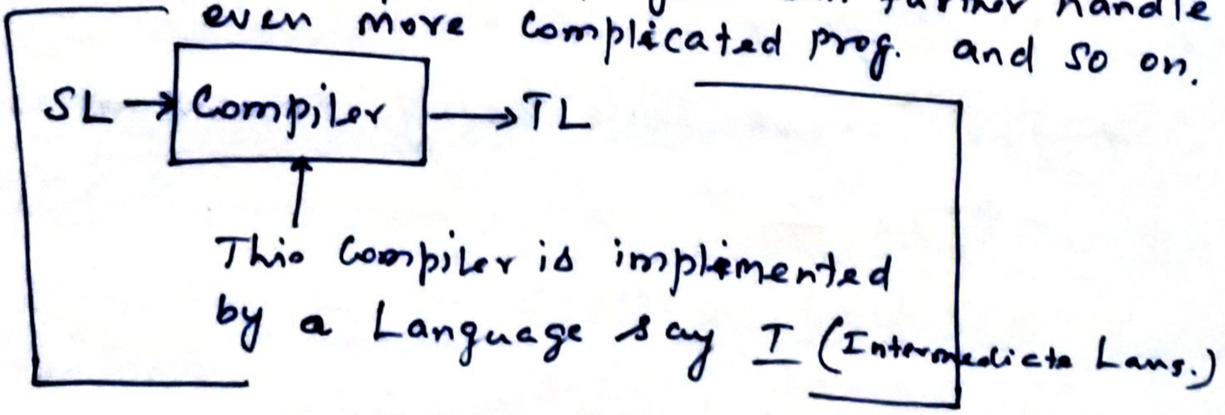

SL
Compiler
TL
This Compiler is implemented by a Language say I (Intermedicte Lang.)

fig-1

* Writing a compiler for any high level Language is a Complicated process.

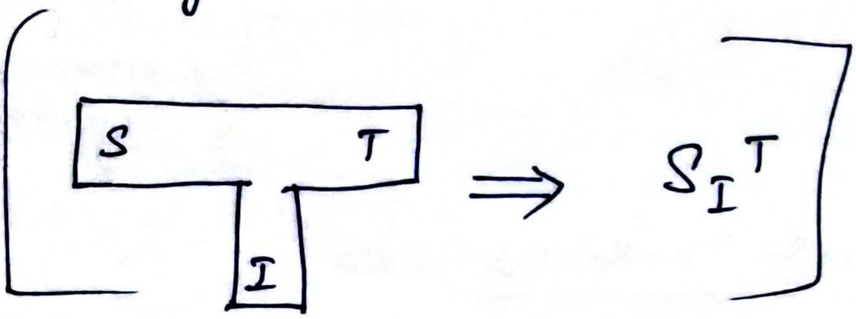
* It takes lot of time to write a Compiler from scratch.

* Hence Simple Lang., is used to generate target code in some stages.

- To Clearly understand the Bootstrapping tech., Consider the foll., Scenario.

- The above fig-1 is represented as T diagram.

fig-2.



- Suppose we want to write a cross Compiler for new language X.

- The implementation Lang., of this Compiler is say Y.

- The target Code being generated is in Lang., Z.

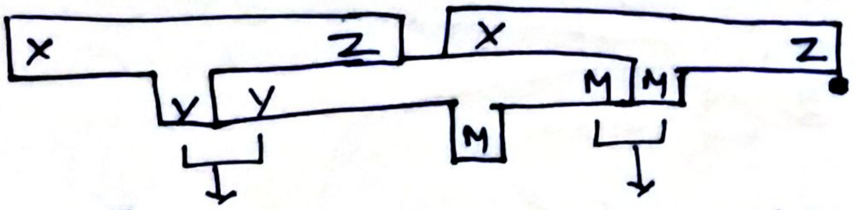
(ie) We create XYZ.

- Now if existing Compiler Y runs on Machine M. and generates Code for M.
Then it is denoted as YMM.

- Now if we run XYZ using YMM then we get a Compiler XMZ

- That Means a compiler for Source Lang., X that generates a target Code in lang., Z. and which runs on Machine M.

The foll., diagram illustrates the above scenario.



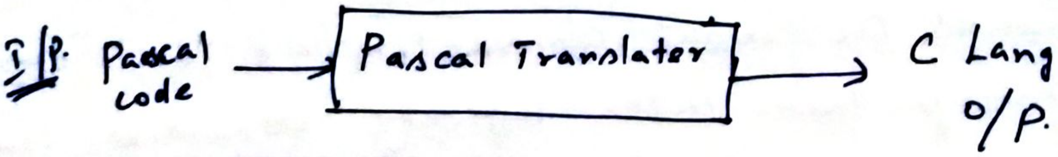
These two Languages must be same

These two Lang., must be same.

eg

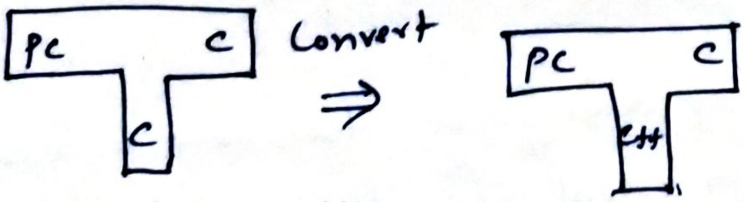
We have a Pascal Translator which is written in C Lang.,

Pascal Translatr – C Lang.,



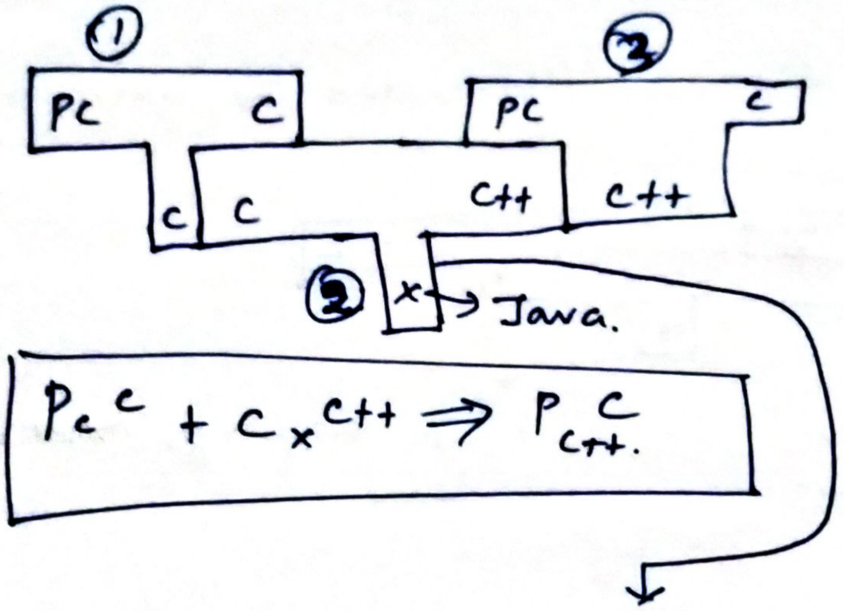
Create a Pascal Translatr in C++

How will we do It?



Pascal translatr written in C Lang and takes Pascal code as input and produce C Lang., as output.

Pascal translatr written in C++ and takes pascal code as input and Produce C Lang., as output.



$$P_c^{c} + C_x^{C++} \Rightarrow P_{C++}^{C}.$$

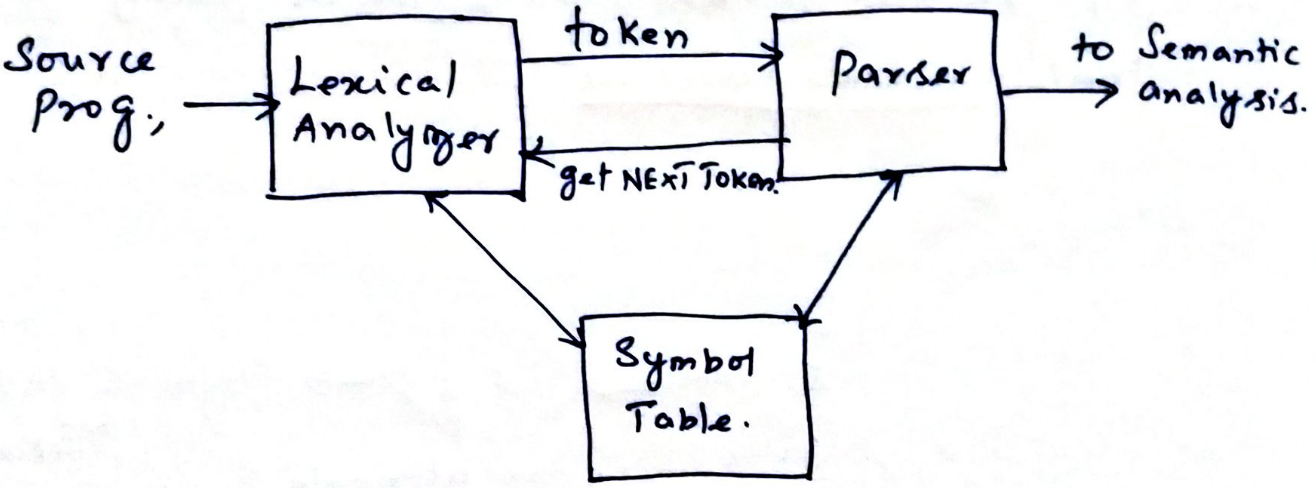
To Convert First T diagram to Third T diagram, we should Process Second T diagram.

Role of Lexical Analyzer

* Reads the input characters of the Source Prog., group them into logically cohesive elements called Lexemes.

* Produces a Seq., of tokens for each lexeme in the source program as output.

* when the lexical analyzer discovers a lexeme Constituting an identifier, it enters that lexeme into the Symbol table.



① Source Prog., is the input to Lexical Analyzer.

②. Lexical Analyzer reads Source Prog., char by char until a meaningful token is found.

③ The output of the Lexical Analyser is token.

④. The Lexical Analyzer is accompanyd a Symbol Table.

⑤ whenever the LA identifies an identifier in the Source Prog., it has to Check that the identifier is already ~~as~~ existed in the Symbol table or not.

when it reads an identifier SUM, ~~we~~ we have to check ~~that~~ whether the SUM is already encounted in the Prog.