Compilers

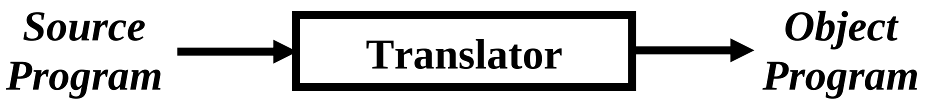
Principles, Techniques, & Tools

References

"Compilers Principles, techniques, & tools", by Alfred Aho, second edition, Pearson Addison Wesley, 2007.

Compiler and Translators

Translator is a program that takes as input a program written in one programming language (the source language) and produces as output a program in another language (the object or target language).

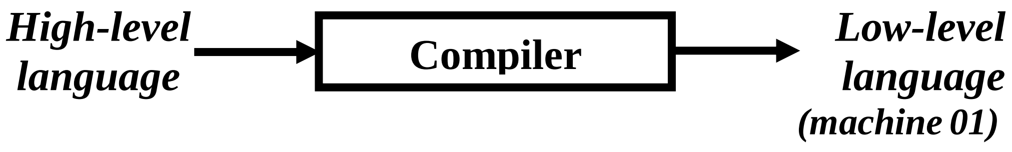


Types of translator

There are four types of translator according to the source language as well as target language:

Compiler is a translator that translates a high-level language program such as FORTRAN, PASCAL, C^{++}, to low-level language program such as an assembly language or machine language.

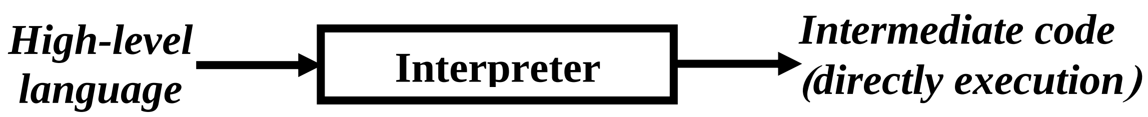
Note: The translation process should also report the presence of errors in the source program.



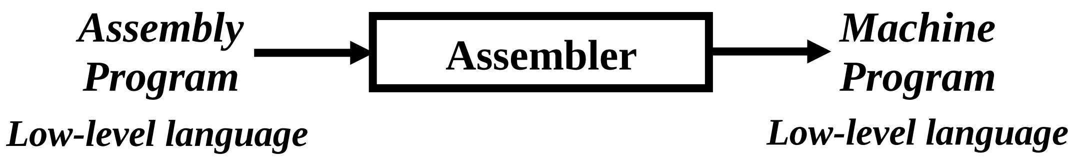
Others Translators:

Interpreters are another kind of translators. An ***Interpreter*** is a translator that effectively accepts a source program and executes it directly, without, producing any object code first. It does this by fetching the source program instructions one by one, analyzing them one by one, and then "executing" them one by one.

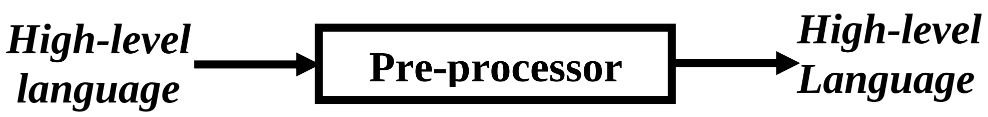
1- Smaller (advantage)
2- Slower (disadvantage)



An ***Assembler*** is a translator that translates the assembly language program (mnemonic program) to machine language program.



Pre-processor: is a Translator Translate program written by H.L.L Into an equivalent program in H.L.L.



Why do we Need Translations?

If there are no translators then we must programming in machine language. With machine language we must be communicate directly with a computer in terms of bits, registers, and very primitive machine operations. Since a machine language program is nothing more than a sequence of 0's and 1's, programming a complex algorithm in such a language is terribly tedious and fraught with mistakes.

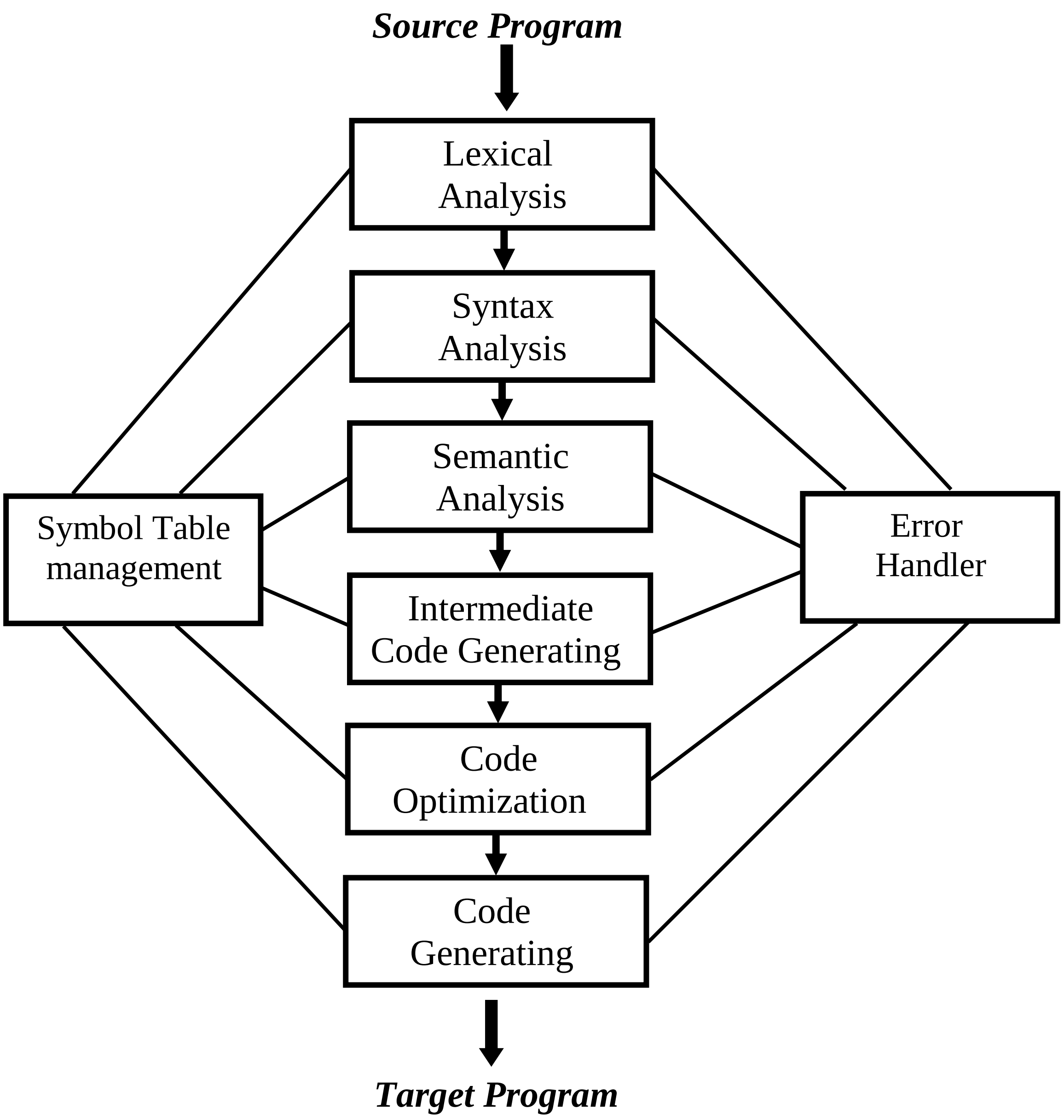
Why we Write Programs in High-Level Language?

We write programs in high-level language (advantages of high-level language) because it is:

1) **Readability**: high-level language will allow programs to be written in the same ways that used in description of the algorithms.
2) **Portability**: High-level languages can be run without changing on a variety of different computers.
3) **Generality**: Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become expert in many languages.
4) **Brevity**: Programs expressed in high-level languages are often considerably shorter (in terms of their number of source lines) than their low-level equivalents.
5) It's easy in the **Error checking** process.

The Structure of a Compiler

The process of compilation is so complex to be achieved in one single step, either from a logical point of view or from an implementation point of view. For this reason it is partition into a series of subprocesses called Phases. The typical compiler consists of several phases each of which passes its output to the next phase plus symbol table manager and an error handler as shown below:



Phases of a Compiler.

<u>Lexical Analyzer (Scanner):</u> It is the first phase of the complier and also called scanner, which represent the interface between source program and the compiler. where the Lexical will read the source program by one letter at a time and decomposed it into group of letters called TOKEN that is logically interrelated between them.

This token represent string of characters that can be processed together as a logical input.

In others words, lexical Separates characters of the source language program into groups (sets) that logically belong together. These groups are called Tokens.

Tokens are Keywords, Identifiers, Operator Symbols, and Punctuation Symbols. The output of the analyzer is stream of tokens, which is passed to the syntax analyzer.

Tokens: groups of characters of source language program logically belong together. For example:

Tokens =

For, If, Do	{Keyword}
X, A1, Num	{Identifiers}
3, 44.2,-53	{Constants}
>, +, =	{Operator Symbols}
; , . ,'	{Punctuation Symbols}

Syntax Analyzer (Parser): Groups tokens together into syntactic structures called ***Expressions***. Expressions might further be combined to form Statements. Often the syntactic structure can be regarded as a ***Tree (parse tree)*** whose leaves are the tokens. The interior nodes of the tree represent strings of tokens that logically belong together.

Every programming language has rules that prescribe the syntactic structure of well –formed programs. Syntax analysis takes an out of lexical analyzer and produce a large tree call parse tree

Semantics:. is a term used to describe "meaning", and so the constraint analyzer is often called the static semantic analyzer, or simply the semantic analyzer. Semantic analyzer takes the output of syntax analyzer and produces another tree.The output of the syntax analyzer and semantic analyzer phases is sometimes expressed in the form of a **Abstract Syntax Tree (AST).** This is a very useful representation, as it can be used in clever ways to optimize code generation at a later phase.

It is Recently added a stage to analyze in terms of sentence meaning, in other words, will reject the sentence in the case of the discovery of the meaning of error (Note: The sentence can be true in terms of the syntax but have no meaning).

For example the English sentence : "The man eat the house " This is in the natural language, put in programming languages can either write inter correct the hand, but wrong in the sense rules .

Intermediate Code Generator: Create a stream of simple instructions. Many style of intermediate code are possible. One common style uses instructions with one operator and a small number of operands.

This phase is used structures built in the previous stages and convert them into intermediate codes by using a most famous meta language such as:

- Postfix notation
- Polish notation

In some versions be enforceable as is the case in the interpreter for the BASIC language and commands interpreter of the DOS operating system.

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

Code Generation: In a real compiler this phase takes the output from the previous phase and produces the object code, by deciding on the memory locations for data, generating code to access such locations, selecting registers for intermediate calculations and indexing, and so on.

Table Management: Portion of the compiler keeps tracks of the name used by the program and records essential information about each, such as its type integer, real, … etc. the data structure used to record this information is called a ***Symbolic Table***.

Error Handler: One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error messages should allow the programmer to determine exactly where the errors have occurred. Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message.

Passes of Compiler

The previous phases showing the main parts of the compiler theoretically, either in the case of the application must merge combine more than one phase in round one and this is called one pass. Each pass will read the output of the previous pass (except for the first pass) that reads the source program.

In the case of integrating more than one stage in one round will lead to interference in the work of these stages through the control structure between these stages.

The influential factor in the number of stages per pass depends on the structure of the source language , the translator , who owns multi-pass uses less memory space of a translator who is in round one single pass. On the other hand , the multi-pass type slower than the another because each stage reads and writes in the intermediate file leads to the slow work of the translator . Must be a balance between the number of stages in a single pass and between the speed of the compiler. And on this basis can be lexical in one pass with syntax or separated .

When a lexical phase are separate from syntax pass there are four pros :

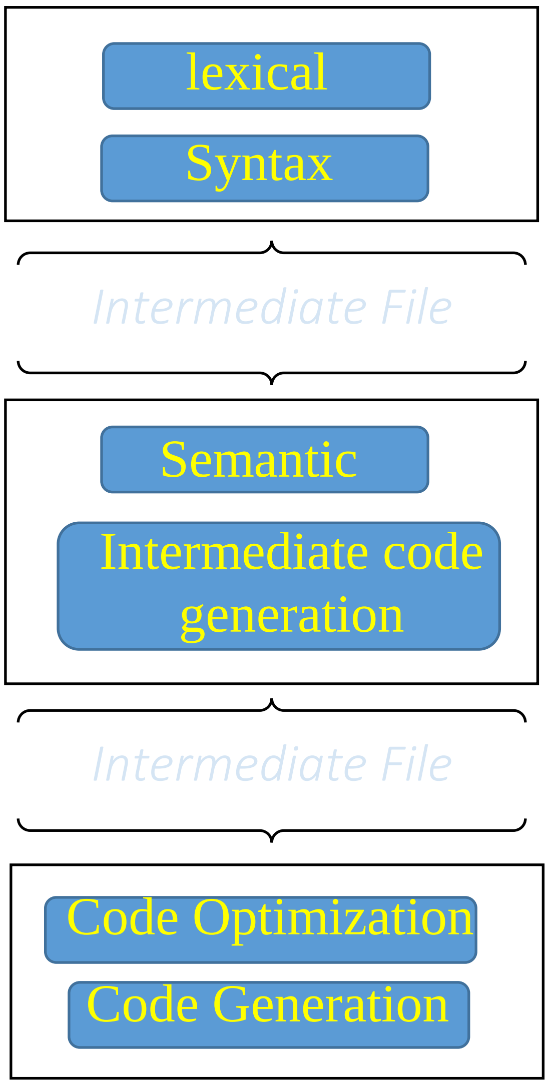
1 - the design of each model separately and thus achieved simplicity.

2 - be efficiency translator largest so that you can manipulate each part in order to achieve the desired efficiency .

3 - including that the lexical is the interface for that is change when you change the language and vocabulary development process.

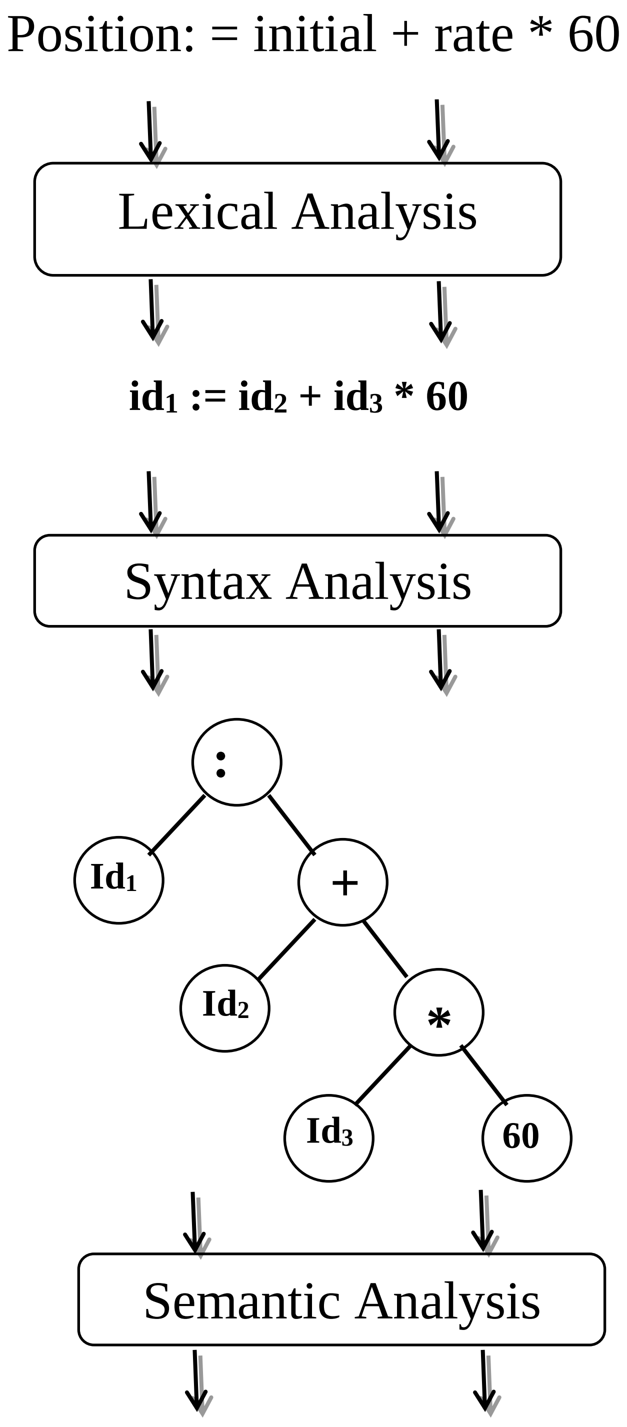
4 – can make symbol table more structure in which we can use it as input and output in the same time.

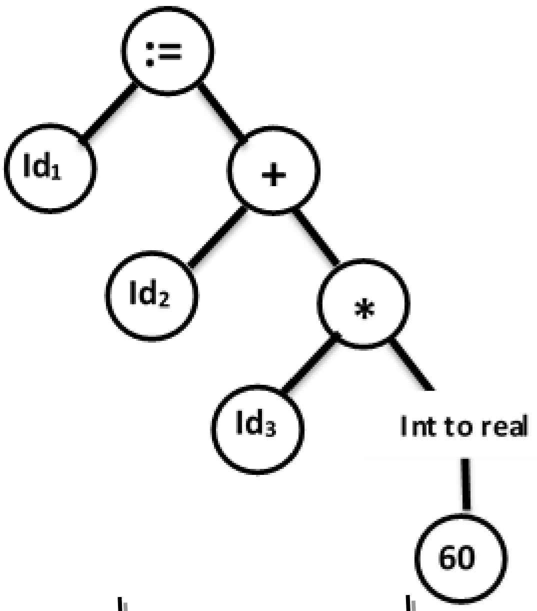
Notice: If phase lexical built with syntax In this case, the Lexical in the form of a subroutine assistant called from the phase of syntax that followed , and if their separation will store the results of intermediate file, this file is called symbol table.


lexical
Syntax
Intermediate File
Semantic
Intermediate code generation
Intermediate File
Code Optimization
Code Generation

Example of the Compilation Process

Consider the following sentence is segment from source program:





intermediate Analysis

$temp_1$:=integer to real (60)
$temp_2$:=id_3 * $temp_1$
$temp_3$:=id_2 + $temp_2$
id_1:= $temp_3$

Code optimization

$temp_1$:= id_3 * 60.0
id_1:= id_2+ $temp_1$

Code generation

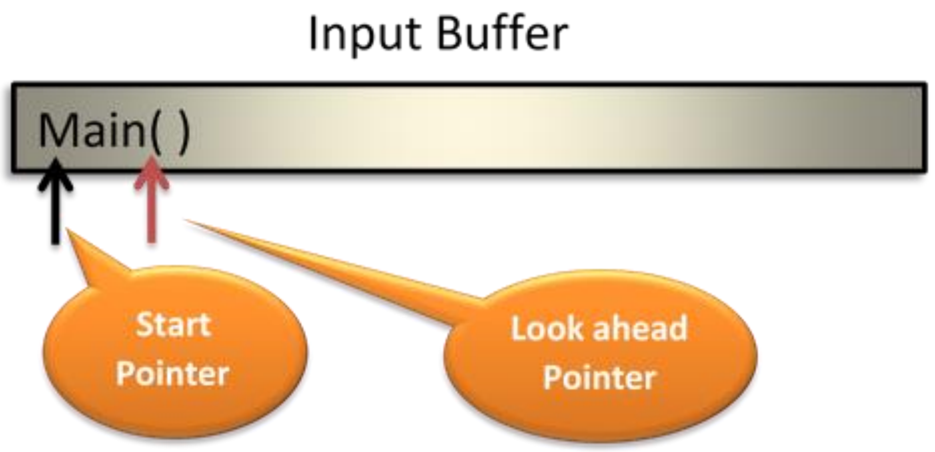
movf id_3 ,R_2
mulf #60.0 ,R_2
movf id_2 ,R_1
addf R_1,R_2
movf R_2,id_1

How Lexical Work

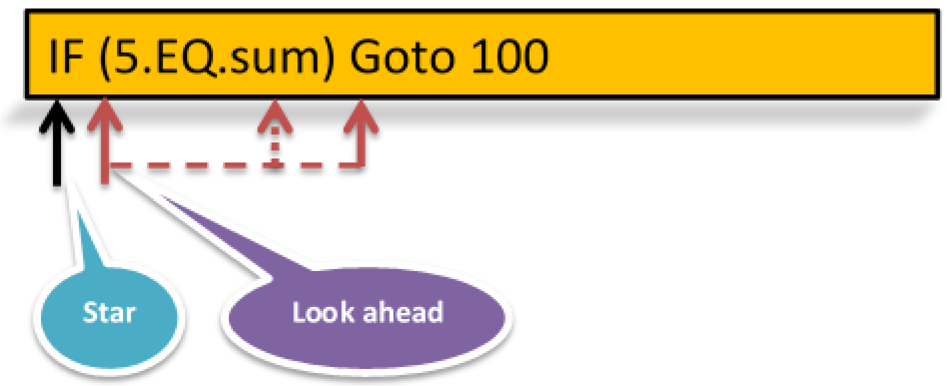
Input Buffering:

The lexical analyzer scans the characters of the source program one at a time to discover tokens; it is desirable for the lexical analyzer to read its input from an **input buffer**.

We have two pointers one marks to the beginning of the token begin discovered. A look-ahead pointer scans a head of the beginning point, until the token is discovered



For example, if we have the following statement written by HLL program



token	type
IF	keyword
(	punctuation
5	Constant
.EQ.	Logical Operator
Sum	Identifier
)	punctuation
Goto	keyword
1000	Constant