Compiler Design

Compiler Design

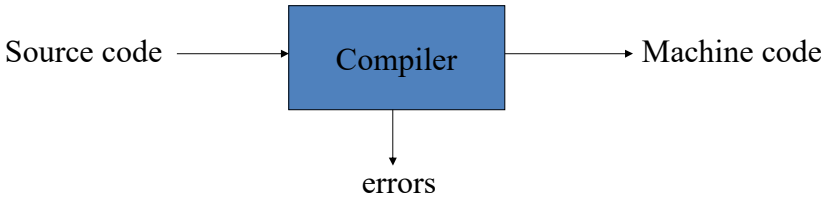
BOOKS AND REFERENCES

1. A.V. Aho, R. Sethi, J.D. Ullman, “Compilers Principles, Techniques and Tools”, Addison-Wesley, 1986.

2. Santanu Chattopadhyay, “Compiler Design”, PHI Learning Pvt. Ltd., 2015.

Compiler

- A compiler is a program that reads a program written in one language - the *source* Language - and translates it into an equivalent program in another language - the *target* language.
- Compiler translates high level language into assembly language or machine language.
- As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.



Cross Compiler: A compiler may run on one machine and produce object code for another machine is called Cross Compiler.

Interpreter vs. Compiler

Interpreter: Interpreter translates just one statement of the program at a time into machine code.

Interpreter	Compiler
➢ Translates one statement of the program at a time.	➢ Translates it as a whole into machine code.
➢ Interpreters usually take less amount of time to analyze the source code.	➢ Compilers usually take a large amount of time to analyze the source code.
➢ The overall execution time is comparatively more than compilers.	➢ The overall execution time is comparatively lesser than interpreters.
➢ No intermediate object code is generated, hence it is memory efficient.	➢ Generates intermediate object code which further requires linking, hence it requires more memory.
➢ JavaScript, Python, Ruby use interpreters.	➢ C, C++, Java use compilers.

The Analysis-Synthesis Model of Compilation

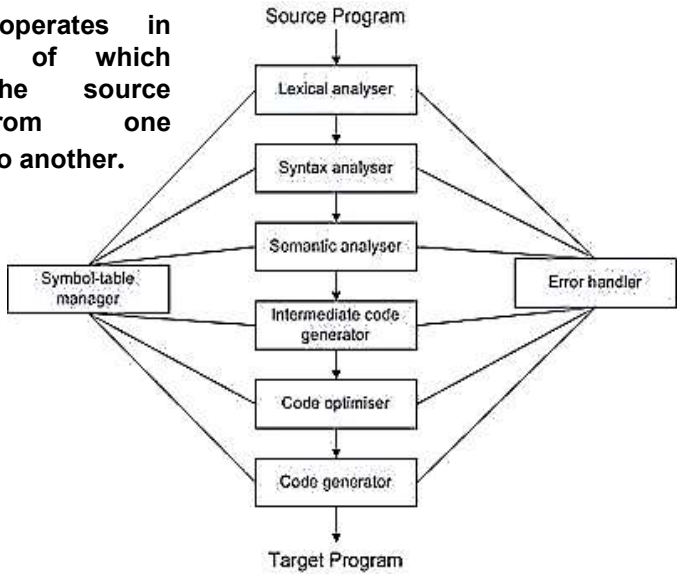
There are two parts to compilation: analysis and synthesis.

- The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.
- The synthesis part constructs the desired target program from the intermediate representation.

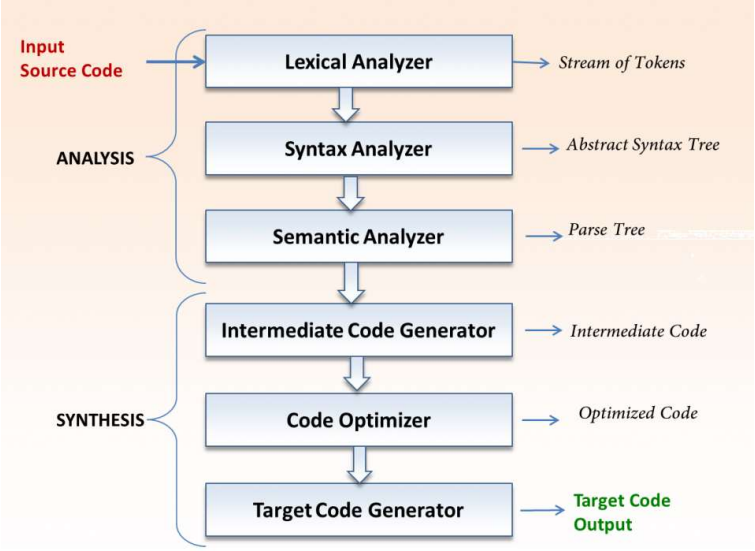
The synthesis requires the most specialized techniques.

Phases of Compiler

- **A compiler operates in *phases,* each of which transforms the source program from one representation to another.**



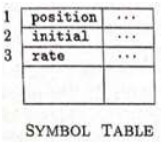
Phases of Compiler



Phases of Compiler

Symbol-Table Management

- To **record the identifiers** used in the source program and collect information about various attributes of each identifier.
- These attributes may provide information about the **storage allocated for an identifier, its type, its scope**, and, in the case of ***procedure*** names, the *number* and *types of its arguments*, the method of passing each argument, and the *type returned*, if any.
- A **symbol table** is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to store or retrieve data from that record quickly.
- When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table.
- Example: **position := initial + rate * 60**
  The lexical value associated with this occurrence of id points to the **symbol-table** entry.
  id1, id2, and id3 for position, initial, and rate respectively.



Phases of Compiler

Error Detection

- Each phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
- The ***lexical phase*** can detect errors where the characters in the input do not form any token of the language.
- Errors where the token stream violates the structure rules (syntax) of the language are determined by the ***syntax analysis phase***.
- During **semantic analysis** the compiler tries to detect constructs those have the right syntactic structures but no meaning to the operation involved, e . g , if we try to *add two identifiers*, one of which is the name of *an array*, and the other the name of *a procedure*.
- The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler.

Phases of Compiler

Lexical analysis

- Reads the characters in the source program and **groups them into a stream of tokens** in which each token represents a logically interrelated sequence of characters, such as an identifier, a keyword, a punctuation character, or a multi-character operator like >=.
- The character sequence forming a token is called the **lexeme** for the token.
- It also does the additional job by removing extra white spaces, comments from the program, expanding the user defined macros like #include and #define in C.
- Example: **position := initial + rate * 60**
  The tokens are **position , := , initial , + , rate , * , 60**

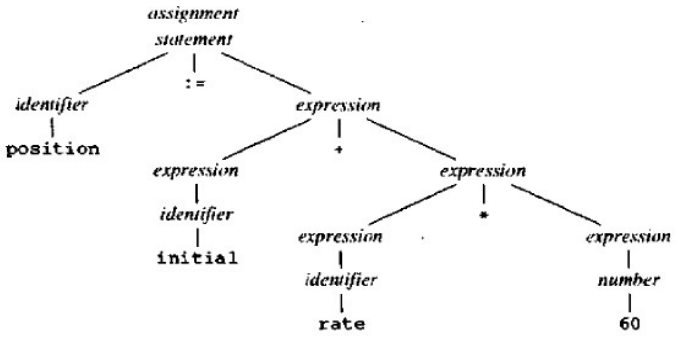
  id1, id2, and id3 for position, initial, and rate respectively,

  id1 := id2 + id3 * 60

Phases of Compiler

Syntax analysis

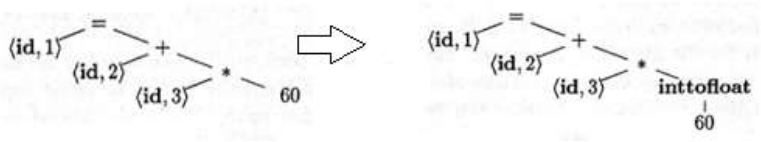
- Hierarchical analysis is called parsing *or syntax* analysis.
- It involves grouping the tokens of the source program into grammatical phrases to synthesize output.
- The grammatical phrases are represented by a **parse tree**.
- Example: **position := initial + rate * 60**



Phases of Compiler

Semantic Analysis

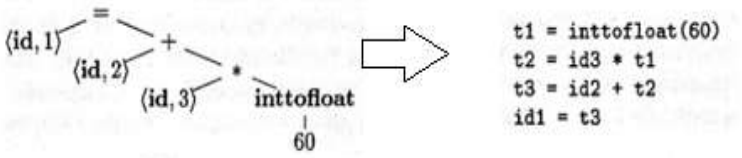
- Checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.
- It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.
- An important component of semantic analysis is ***type checking***.
- The compiler checks that each operator has operands that are permitted by the source language specification.
- For example, many programming language definitions require a compiler to report an error every time a real number is used to index an array.



Phases of Compiler

Intermediate Code Generation

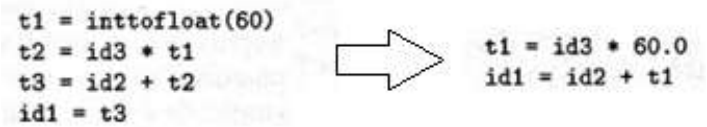
- After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program.
- This intermediate representation should have two important properties; it should be easy to produce, and easy to translate into the target program.



Phases of Compiler

Code Optimization

- The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result.



- The compiler can deduce the conversion of 60 from integer to real. This representation can be done once at compile time, so the inttoreal operation can be eliminated.
- **t3** is used only once to transmit its value to **id1**. It then becomes safe to substitute **id1** for **t3**.

Phases of Compiler

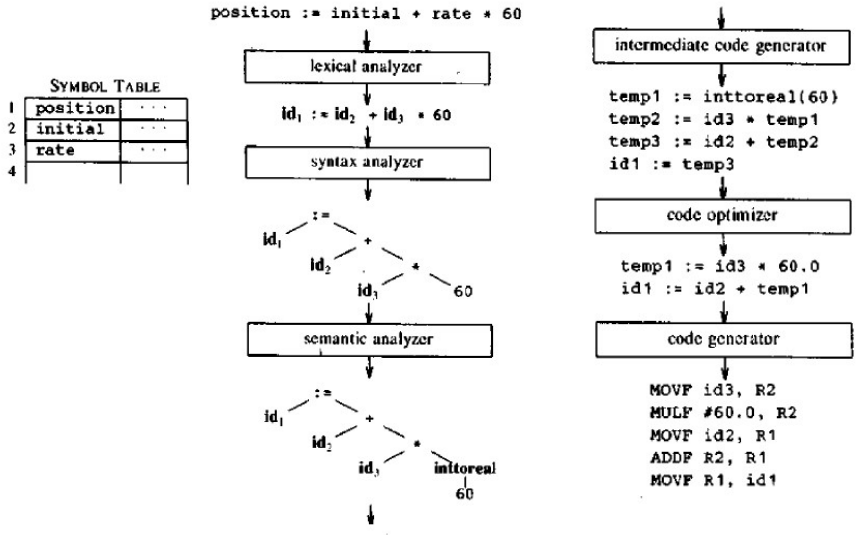
Code Generation

- The final phase of the compiler is the generation of target code, consisting of relocatable machine code or assembly code.
- Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- A crucial aspect is the assignment of variables to registers.

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```


All Phases of Compiler



Cousins of the Compiler

Pre-processor: Pre-processor produces input to compiler. It may perform the following functions:

- *Macro* processing: A preprocessor may allow a user to define macros that is shorthand for longer constructs.
- File *inclusion:* A preprocessor may include header files into the program text.

Assemblers: Some compilers produce assembly code, that is passed to an assembler for further processing. ***Assembly* code** is a mnemonic version of machine code, in which names are used instead of binary codes for operations, and names are also given to memory addresses.

```
MOV  a,  R1
ADD  #2, R1
MOV  R1, b
```

This code moves the content of the address ***a*** into register 1, then adds the constant ***2*** to it, and finally stores the result in the location named by ***b***. Thus, it computes ***b : = a + 2*** .

Cousins of the Compiler

Two-Pass Assembly:

- The simplest form of assembler makes two passes over the input, where a *pass* consists of reading an input file once.
- In the **first pass**, all the identifiers are assigned into storage locations as they are encountered for the first time.
- In the **second pass**, the assembler scans the input again. This time, it translates each operation code into the sequence of bits representing that operation in machine language, and it translates each identifier representing a location into the address given for that identifier in the symbol table.
- The output of the second pass is usually ***relocatable*** machine code, meaning that it can be loaded starting at any location *L* in memory; i.e., if *L* is added to all addresses in the code, then all references will be correct.

Cousins of the Compiler

Loaders and Link-Editors:

- A program is called a loader, which performs the two functions of **loading** and **link-editing**.
- The process of **loading** consists of taking relocatable machine code, altering the reloatable addresses and placing the altered instructions and data in memory at the proper locations.
- The **link-editor** allows us to make a single program from several files of relocatable machine code. These files may have been the result of several different compilations, and one or more may be library files of routines provided by the system and available to any program that needs them.
- If the files are to be used together in a useful way, there may be some ***external*** references, in which the code of one file refers to a location in another file. This reference may be to a data location defined in one file and used in another.

THANK YOU