INTRODUCTION TO COMPILERS

Vasuki P
<VasukiP@ssn.edu.in>

Introduction

Unit Objectives

- To understand the need for compilers
- To understand various phases of compilers
- To understand the structure and usage of symbol table

Unit Outcomes

- UO1: To segregate the compiler process into different phases
- UO2: Understanding of every segment
- UO3: To generate and use the symbol table

Mapping of Unit Outcome to Course Outcome

Table: Add caption

UO \ CO	CO1	CO2	CO3
UO1			
UO2			
UO3			

Language Processer

The high-level language is converted into machine excutable binary language in various phases.

- Compiler -A compiler is a program that converts high-level language to assembly language.
- Assembler - It is program that converts the assembly language to machine-level language.
- Interpreter - It converts a statement into executable code.

High level to machine level translators

- Compiler is a translator - Compilation is translating a source language program to target language program

  $$Sourceprogram- > Compiler- > Targetprogram$$

  Where source is any high level language program
  Target can be
  - Machine dependent code
  - Another language program
  - Intermediate Code [ex. Byte Code]
- Interpretation - is the conceptual process of running high-level code by an interpreter
  Execution of every statement on the input supplied by the user.

Interpreter does the operation line by line, where the compiler generates translation for the entire code then executes whenever it is needed. Thus compiler is faster, where as debugging is eaier in interpretter.

Compiler - Phases

- **Analysis Phase** -Analyzes correctness of the program
- **Synthesis Phase** - Generation of target code

$$\Bigg\}$$

Phases of Compiler

- Analysis
  - Syntatic Analysis
  - Structural Analysis
- Synthesis
  - Intermediate code generation
  - Code optimization
  - Target code generation
  - Machine dependent code optimization

Phases of Compiler

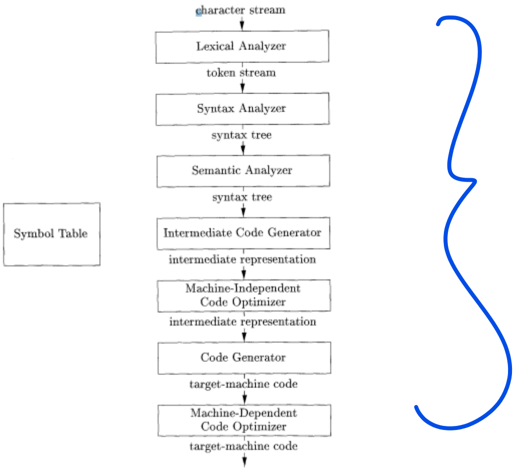

Figure: Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, Compilers - Principles, Techniques and Tools, 2nd Edition, Pearson Education, 2007

Phases of Compiler

- **Lexical Analysis** Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.
- **Syntax Analysis** A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

Phases of Compiler

- **Semantic Analysis** Semantic analysis, also context sensitive analysis, is a process in compiler construction, usually after parsing, to gather necessary semantic information from the source code.
- **Intermediate Code Generation** Code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code.

Phases of Compiler

- **Code Optimization** An optimizing compiler is a compiler that tries to minimize or maximize some attributes of an executable computer program. The most common requirement is to minimize the time taken to execute a program; a less common one is to minimize the amount of memory occupied
- **Code Generation** Code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine.
- **Symbol Table** Symbol table is a data structure, which is used in all the phases of compilers. It is created and maintained by compilers to store the details of occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. It also stores the type, size and scope of the variables. The data stucture should be designed in a way for easy access of any entry. Symbol table gets entry at every scope.

Structure of Symbol Table

Sample Program Snippet

```
int main()
{
int x,y;
int myfunction(float x, float a)
{
float innerfun(int x, int y, float z)
{
...
}   ....
} }
```

Symbol Tables

Table: Symbol Table for innerfun

x	int
y	int
z	float

Table: Symbol Table for mfunction

x	flaot
a	flaot
innerfun	float

Table: Symbol Table for main

x	int
y	int
myfunction	int

Symbol Table -Uses

- To verify whether all used variables are defined
- To verify the scope of the variable
- To verify sementic correctness of asignment of two variables -(Type checking)
- To generate intermediate code.

Error Handler

Compiler Design - Error Recovery. A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input.

Error Handler

Error in program is checked at varios phases of compilation.

- Lexical : Not a valid lexeme; not a valid variable ex 45a;
- Syntactical : missing semicolon or unbalanced parenthesis - structural error(ex int a)
- Semantical : assignment of imcompatible variables.
- Logical : code not reachable, infinite loop

Modes of Error recovery

- **Panic mode**
  When parser encounters an error, it stop processing from that point to the end. This is the easiest way as the parser doesn't involve in infinite loop.
- **Statement mode**
  Parser identifies the simple mistakes in the statment and corrects those errors and allows the parser to move ahead of the next line. For example the parser has to attach a semicolon at the end replacing dot with semicolan.
  Other possible error-recovery actions are:
  - Delete one character from the remaining input
  - Insert a missing character into the remaining input
  - Replace a character by another character
  - Transpose two adjacent characters
- **Global correction**
  The parser checks the syntatical and structural correctness of the statement by constructing parse tree.If the statment is erroroneous, it

Reference Books

TextBooks

- Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, Compilers - Principles, Techniques and Tools, 2nd Edition, Pearson Education, 2007.

Reference Book

1. Alfred V Aho, Monica S. Lam, Ravi Sethi and Jeffrey D Ullman, Compilers - Principles, Techniques and Tools, 2nd Edition, Pearson Education, 2007.
2. Scott, M. L., Programming Language Pragmatics, second edition, Morgan- Kaufmann, San Francisco, CA, 2006.
3. 1. Randy Allen, Ken Kennedy, "Optimizing Compilers for Modern Architectures: A Dependence-based Approach", Morgan Kaufmann Publishers, 2002.
4. V. Raghavan, "Principles of Compiler Design", Tata McGraw Hill Education Publishers, 2010.
5. Allen I. Holub, "Compiler Design in C", Prentice-Hall Software Series, 1993.