Chapter 1

Introduction

- **Def:** Compiler --
  - a program that translates a program written in a language like Pascal, C, PL/I, FORTRAN, or COBOL into machine language.

1. Machine Language, Assembly Language, High-level Languages

- **Machine Language** -- the native language of the computer on which the program is run.
  - It consists of bit strings which are interpreted by the mechanism inside the computer. These strings looked like:
    0001100000110101
  - In the early days people programmed in this, and wrote out these bit strings.

- The first translators were **assemblers**.
  - They translated from
    LR 3,5
  - to the bit string on the previous slide.
  - It did this by looking up the mnemonic (LR in this case) in a table and pulled out its corresponding opcode, found the binary representations of 3 and 5 and assembled them into the instruction
  - This representation is known as assembly language

- Languages like Pascal, C, PL/I, FORTRAN and COBOL are known as **high-level languages**.
  - They have the property that a single statement such as
    X := Y + Z;

    corresponds to more than one machine language instruction.

- The previous statement could be translated to:

```
L    3,Y
A    3,Z
ST   3,X
```

- The main virtue of high-level languages is productivity.
  - It has been estimated that the average programmer can produce 10 lines of ***debugged*** code in a working day
  - and that number is independent of the language.

2. Terminology

- **source language** -- the high level language that the compiler accepts as its input.
- **source code** -- the source language program that is fed to the compiler.
- **object language** -- the particular machine language that the compiler generates.
- **object code** -- the output of the compiler

- **object file** -- the file to which object code is normally written to (in external storage). This is sometimes called and **object module**
- **target machine** -- the computer on which the program is to be run.
- **cross-compiler** -- a compiler that generates code for a machine that is different from the machine on which the compiler runs.

3. Compilers and Interpreters

- A compiler translates; an interpreter executes.
- the main advantage of interpreters is the immediacy of the response.
- the main disadvantage is the slow speed of execution.

4. The Environment of the Compiler

- The object file produced by the compiler is normally not ready to run.
- It is not practical for a compiler to have at hand all the various methods for computing things like square roots, logs, and other functions

- **def:** run-time library -- a collection of object modules for computing basic functions.
  - math functions.
  - character and string i/o
- Another step is needed…the linker
  - all the required run-time library services are identified and put with the user's object with the linker
  - The linker normally generates an executable program

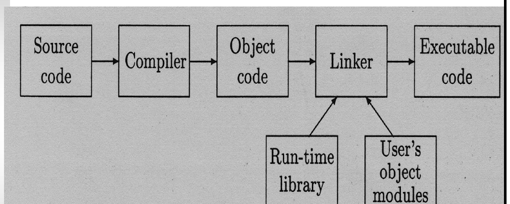

Figure 1.1

- Another step is a loader,
  - Places the executable into memory and executes it.
  - if the system uses shared libraries, also does linking on the fly.
- The study of linkers and loaders is beyond the scope of this class.

5. Phases of a Compiler

- **Lexical Analysis** -- breaking up the source code into meaningful units (tokens)
  - This is covered in Chapter 2
- **Syntactic Analysis** -- determines the structure of the program and of all the individual statements.
  - This is covered in Chapters 3 and 4

- **Intermediate Code Generation** -- An internal representation of the program that reflects the information uncovered by the parser. 3-address code, or 4-tuples
  - This is covered in Chapter 5
- **Optimization** -- Code Enhancement
  - This is covered in Chapter 6

- **Object Code Generation** -- translate the optimized intermediate code into the target language.
  - This is covered in Chapter 7

6. Passes, Front End, Back End

- **Pass** -- A pass consists of reading a version of the program from a file, and writing a new version of it to an output file.
  - A Pass normally comprises more than one phase, but the number of passes and phases varies.
  - Single pass compilers tend to be fastest, but there are reasons for more than one pass (memory and language issues)

- **Front End** -- the phases of the compiler that are heavily dependent upon the source language and have little or no concern with the target machine.
  - (Lexical Analysis, Parsing, Intermediate Code Generation, and some Optimizations)
- **Back End** -- those phases that are machine dependent.
  - (some Optimization, and Code Generation)

7. System Support

- Symbol Table
  - the central repository of information about the names (identifiers) created by the programmer.
- Error Handling
  - this implements the compiler's response to errors in the code it is compiling.
  - The error handler must tell the user as much about the error as possible.

8. Writing a Compiler

- When you start with a brand new piece of hardware, you write a compiler in assembler
  - At first you have no choice.
- Once an adequate high level language is available, there are more attractive options available
  - like writing the compiler for the language you want in the language you have.

- **Boot Strapping** -- writing a minimal compiler, then writing the full compiler in that minimal language.
  - Write a minimal C compiler in assembler.
  - Write a C compiler in minimal C.
- Tools for helping with compiler writing
  - LEX (flex)
  - YACC (bison)

9. Retargetable Compiler

- One way is to take advantage of the break between the front end and the back end
  - Front end for the language
  - Back end for the machine
- Examples:
  - P-code -- UCSD P-system
  - GNU Compilers (gcc, g++, g77, …)

10. Summary

- We have seen a quick overall picture of what a compiler does, what goes into it, and how it is organized. Details on all these will appear in subsequent chapters.
- Probably the one sure way to learn about compilers in depth is to write one. So, …