Compilers

Prerequisites

- Data structures & algorithms
  - Linked lists, dictionaries, trees, hash tables
- Formal languages & automata
  - Regular expressions, finite automata, context-free grammars
- Machine organization
  - Assembly-level programming for some machine

Why we study compilers?

You may never write a commercial compiler, but that's not why we study compilers. We study compiler construction for the following reasons:

- Writing a compiler gives a student experience with large-scale applications development. Your compiler program may be the largest program you write as a student. Experience working with really big data structures and complex interactions between algorithms will help you out on your next big programming project.
- Compiler writing is one of the shining triumphs of CS theory. It demonstrates the value of theory over the impulse to just "hack up" a solution.
- Compiler writing is a basic element of programming language research. Many language researchers write compilers for the languages they design.
- Many applications have similar properties to one or more phases of a compiler, and compiler expertise and tools can help an application programmer working on other projects besides compilers.

1. Introduction to Compiling

1.1 Compilers and translators

Compilers act as translators, transforming human-oriented programming language into computer oriented machine language. To most users, a compiler can therefore be viewed as a “black box” that performs the transformation illustrated in Fig (1).

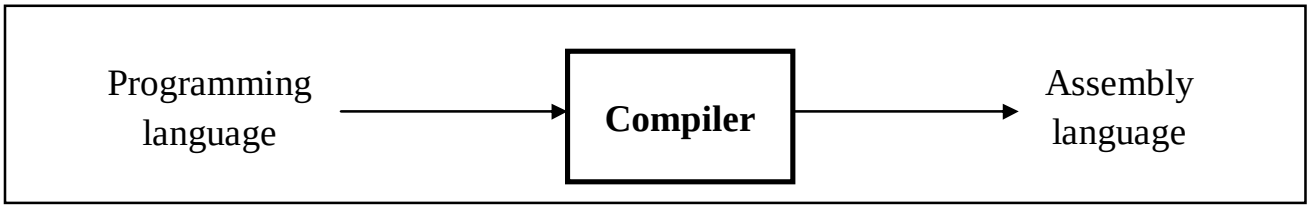

Fig (1): A user’s view of a Compiler

➢ A compiler allows most (indeed, virtually all) computer users to ignore the **machine-dependent details of machine–language**, compilers therefore allow programs and programming expertise to be machine-independent.

The term Compiler was coined in the early 1950's by "Grace Murray Hopper". Translation was then viewed as the "Compilation" of a sequence of subprograms selected from a library. Indeed, Compilation (as we now known it) was then called "automatic programming" and there was almost universal skepticism that it would ever be successful.
Today, the automatic translation of programming language is an accomplished fact, but programming translator is still termed Compilers.

➢ A **translator** is a program that takes as input a program written in one programming language (the source language) and produce as output a program written in another language (the **object or target language**). If the source language is high –level language such as Fortran ,PASCAL , … , etc , and the **object language is a low level language** such as **assembly language** or machine language , then such a translator is called a **Compiler**.

Executing a program written in **high–level programming language** is basically a **two step process** , as illustrated in Fig(2) the **source program must first be compiled** ,that is , **translated into the object program** , then the **resulting object program is loaded into memory and executed**.

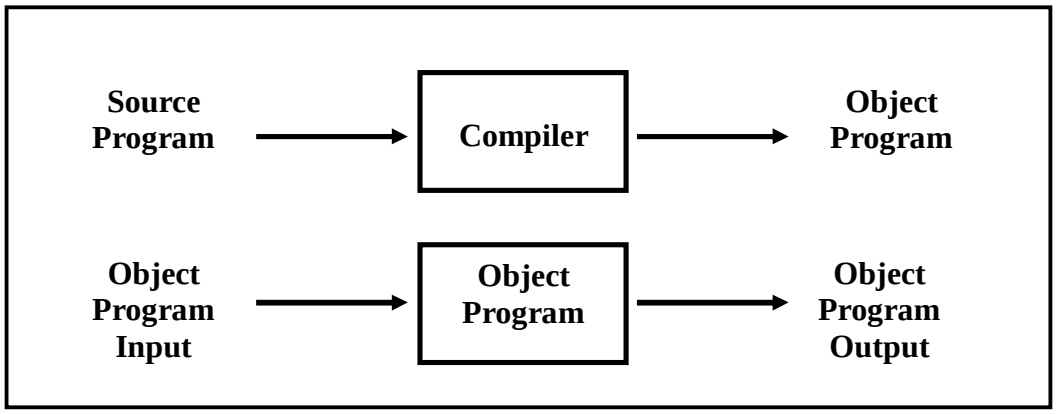

Fig (2): Compilation and execution

Certain other translators transform a programming language into a simplified language, called **intermediate code**, which can be directly executed using a

program called an **interpreter**. We may think of the **intermediate code** as the **machine language** of an abstract computer designed to execute the source code.

Interpreter are often smaller than compilers and facilitate the implementation of complex programming language constructs .However, the **main disadvantage** of interpreters is that the **execution time** of an interpreted program is usually slower than that of the corresponding compiled object program .
There are several others important types of translators, besides compilers. If the source language is assembly language and the target language is machine language then the translator is called an **assembler**

In addition to compiler, several other programs may be required to create an executable target program. A source program may be divided into modules stored into separate files. The task of collecting the source program in sometimes entrusted to a distinct program, called **preprocessor**.

- **Why we need Translators?**

The answer to this question is obvious to anyone who has programmed in machine language. With machine language we must 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 complex algorithm in such a language is terribly tedious and fraught with opportunities for mistakes.

Perhaps the most serious **disadvantage of machine language**, coding is that all operation and operands must be specified in a **numeric code**.
Because of the difficulties with machine-languages, programming a host of "higher – level" language have been invented to enable the programmer to code in a way that resembles his own thought processes rather than the elementary steps of the computer. The most **immediate step away from machine languages is symbolic assembly language**. In this language, a programmer uses numeric names for both operation codes and data addresses .Thus a programmer could write ADD X,Y in assembly languages instead of something like 0110 001110 011011 in machine language(where 0110 is hypothetical machine operation code for "ADD" and 001110 and 011011 are the address of X and Y).

A computer, however, cannot execute a program written in assembly languages. That program has to be first translated to machine language, which the computer can understand. The program that performs this translation is the **assembler.**

- **What the Differences between an Interpreter and a Compiler**

The main difference between an interpreter and a compiler

➢ An **Interpreter** reads the source code one instruction or line at a time, converts this line into machine code and executes it. The machine code is then discarded and the next line is read. The **advantage** of this is, it's simple and you can interrupt it while it is running, change the program and either continue or start again. The **disadvantage** is that every line has to be translated every time it is executed, even if it is executed many times as the program runs. Because of this interpreters tend to be slow. Examples of interpreters are **Basic** on older home computers, and script interpreters such as **JavaScript**, and languages such as **Lisp**.

➢ A **Compiler** reads the whole source code and translates it into a complete machine code program to perform the required tasks which is output as a new file. This completely separates the source code from the executable file. The biggest **advantage** of this is that the translation is done once only and as a separate process. The program that is run is already translated into machine code so is much faster in execution. The **disadvantage** is that you cannot change the program without going back to the original source code, editing that and recompiling (though for a professional software developer this is more of an advantage because it stops source code being copied). Current examples of compilers are **Visual Basic**, **C**, **C++**, **C#**, **FORTRAN**, **COBOL**, **Ada**, and **Pascal** and so on.

Java language processors combine compilation and interpretation, as shown in Fig (3). A Java source program may first be compiled into an intermediate form called *bytecode*. The *bytecodes* are then interpreted by a virtual machine. A benefit of this arrangement is that *bytecodes* compiled on one machine can be interpreted on another machine, perhaps across a network.

In order to achieve faster processing of inputs to outputs, some java compilers, called *just-in-time* compilers, translate the *bytecodes* into machine language immediately before they run the intermediate program to process the input.

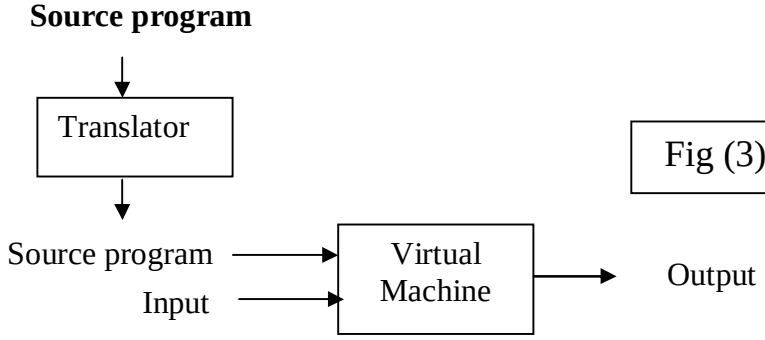

Fig (3): A hybrid Compiler

The Structure of the Compiler

A compiler takes as input a Source program and produces as output an equivalent sequence of machine instructions. This process is so complex that it is not reasonable, either from logical point of view or from an implementation point of view, to consider the compilation process as occurring in one single step. For this reason, it is customary to partition the compilation process into a series of sub process called phases, as shown in Fig (4).A phase is logically cohesive operation that takes as input one representation of the source program and produce as output another representation.

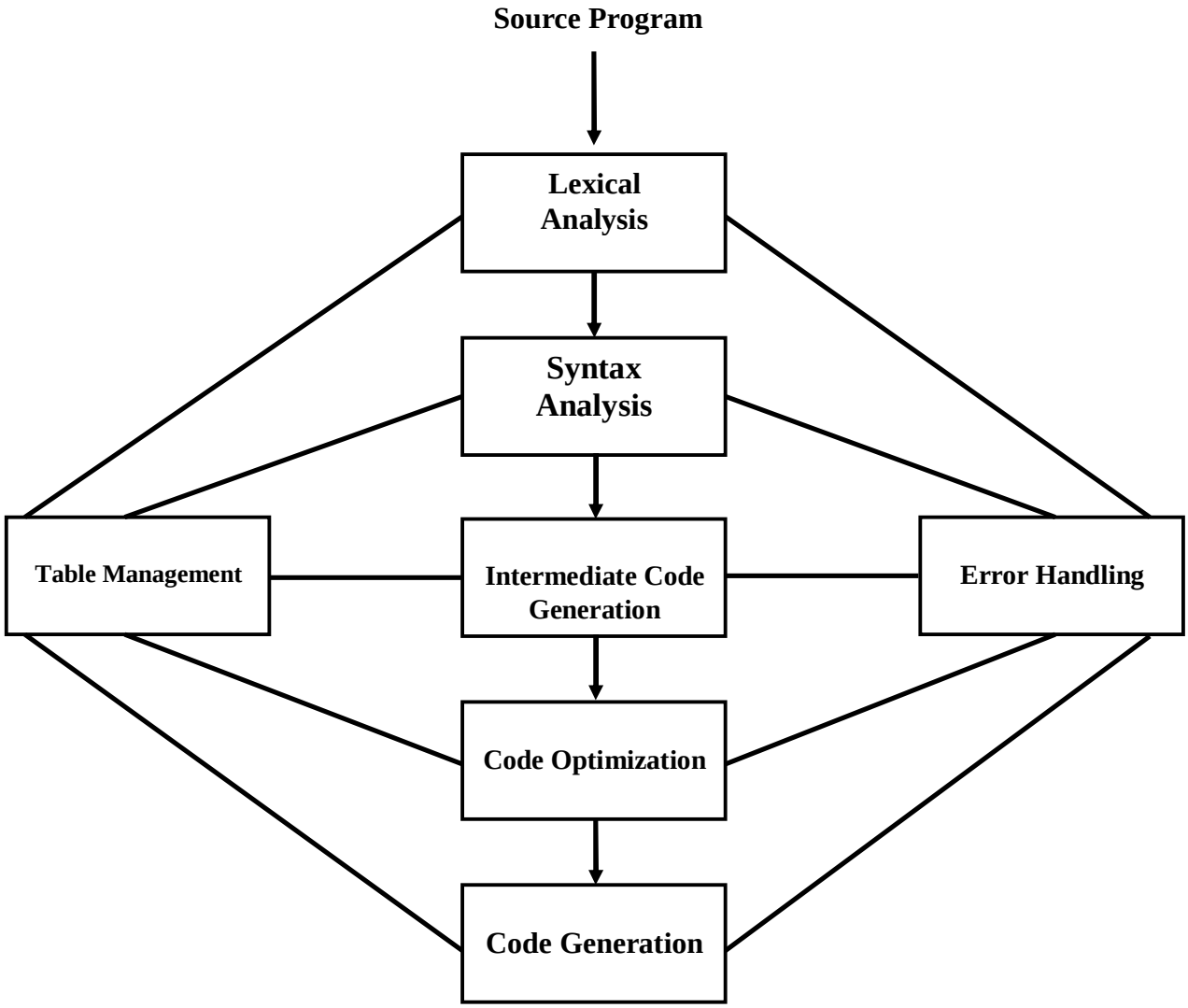

Fig (4) Phases of Compiler

The first phase, called the **lexical analysis phase** which is done by the **"Lexical Analyzer" or "Scanner"**, separates characters of the source language into groups that logically belong together; these groups are called **tokens**. The usual tokens are **keywords** (IF, ELSE, DO, …) ,**Identifiers** (X, Y, num,….) ,operator symbol (**>,>=,=,-,+,…**)**,** and **punctuation symbols** (parentheses ,commas). The **output** of the lexical analyzer is a **stream of token**, which is passed to the **next phase**, the **syntax analysis phase** which is done by the syntax analyzer or **parser**. The token in this stream can be represented by codes which we may regard as integers. Thus, "IF" might be represented by 1, "+" by 3, and "identifier" by 4. In the case of a token like "identifier" a second quantity, telling which of these identifiers used by the program is represented by this instance of token "identifier", is passed along with the integer code for "identifier".

The **"Syntax Analyzer"** groups token together into **syntactic structures**. For example, the three token representing C+D might be grouped into a syntactic structure called an **expression**. Expression might further be combined to form **Statement**. Often the syntactic structures can be regarded as a tree whose leaves are the tokens. The interior nodes of the tree represent string of tokens that logically belong together.

The **"Intermediate Code Generator"**, uses the structure produced by the Syntax Analyzer to 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. These instructions can be viewed as **simple macros**, like this:-

```
MACRO       ADD2    X, Y
            LOAD    Y
            ADD     X
            STORE   Y
ENDMACRO
```

The primary difference between intermediate code and assembly code is that the intermediate code need not specify the registers to be used for each operation.

"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**. Its output is another intermediate code program that does the same job as the original, but perhaps in a way that saves time and/or space.

The final phase **"Code Generation"** produces the **object code** by deciding on the memory locations for data, selecting code to access each data, and selecting the registers in which each computation is to be done. Designing a code generator that

produced truly efficient object programs is one of the most difficult parts of compiler design, both practically and theoretically.

The **"Table-Management"**, or **"bookkeeping"**, portion of the compiler keep track of the names 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 **Symbol Table**.

The **"Error Handler"** is invoked when a flaw in the source program is detected. It must warn the programmer issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can proceed. It is desirable that compilation be completed on flawed programs, at least through the syntax analyzer phase, so that as many errors as possible can be detected in one compilation. Both the table management and error handling routine interact with all phases of the compiler.

In an implementation of the compiler, Portion of one or more phases are combined into a module called a **pass**. A pass reads the source program or the output of the previous pass, makes the transformation specified by its phases, and writes output into an intermediate file, which may then be read by a subsequent pass. If several phases are grouped into one pass, then the operation of the phases may be interleaved, with control alternating among several phases.

The structure of the source languages has a strong effect on number of phases. Certain languages require at least two passes to generate code easily. Some languages allow the declaration of a name to occur after uses of that name. Code for expression containing such a name cannot be generated conveniently until the declaration has been seen.

The environment in which the compiler must operate can also affect the number of passes. A multi – pass compiler can be made to use less space than a single – pass compiler, since the space occupied by the compiler program for one pass can be re used by the following pass. A multi – pass compiler is slower than a single pass compiler, because each pass read and writes an intermediate file. Thus, compilers running on computers with small memory would normally use several passes while, on computers with a large random access memory, a compiler with fewer passes would be possible.

The term **"Semantic Analysis"** is applied to determination the types of intermediate results, that **cheek the arguments are of types that legal** for an application of an operator, and the determination of the operation denoted by the operator (e.g. + could denote fixed or floating add, perhaps logical "or" and possibly other operations as well). Semantic Analysis can be done during the syntax analysis phase.