Chapter 1

Introduction

Programming languages are used for expressing algorithms and data structures that transform data into additional data that have a meaning, that is, information. These algorithms express the computations carried out by an abstract computational model or physical implementation of a computational model. Expressing algorithms or programming a physical machine requires programmer-provided numeric codes for instructions or operations. These codes are decoded by circuits and activate circuits in the machine. The activated circuits execute the instructions and, when necessary,access the programmer-provided numeric codes for addresses of operands or operand values. The first electronic computers were programmed with sequences of instructions, where each instruction was a group of codes representing the operation and possibly operands. It was time consuming to write programs and deploy programs directly on these early machines. Deploying the programs were time consuming because it often took many hours or days to load the program codes into the computers since it involved wiring the circuits together. It was also difficult to read, debug, and modify the numeric codes representing the programs, thus taking more time to get programs to run. Because of these reasons, programming languages were designed to express programs that resembled the syntax of natural languages, like English, but with very precise meaning. Respective translator programs were developed to translate those programs into the machine code. Programming languages are intended to reflect higher-level abstract ideas that are represented in terms of lower-level abstraction of elementary and fundamental operations.

The first languages (called assembler languages) and translators (assemblers) were developed to overcome these challenges. They used mnemonic names (e.g., ADD) for the operations and names (e.g., X)for operands and for values (e.g., 2). They stood in almost a one-to-one correspondence with the machine code instructions, but were easier to read. We look at an example later in this chapter. Instead of using the machine code directly, mnemonic codes were used for each instruction and translator generated the machine codes. Translators of this kind were introduced in the early 1950s. Programmers still work with assemblers when certain applications that require access to specific low-level hardware or where the programmer can write more time and/or space efficient code produced by other higher-level language translators.

Higher-level programming languages were developed to capture more abstract thinking of the programmer. One of the first high-level programming languages developed originally at IBM in the mid 1950s was

ForTran (which stands for Formula Translator), which was more abstract than assembler languages. Fortran was intended for representing programs used in scientific computations. In the late 1950s and early 1960s COBOL, Algol60 and Lisp (List Processing) were designed and implemented with translators that generated assembler code or machine code. COBOL, Algol60 and Lisp were designed to "abstract away" lower-level views of hardware. COBOL is closest to English and was originally designed for business applications. Many institutions still have large amounts of legacy code in use today. Algol60 was introduced to be more general purpose in terms of use in application domains. Algol60 also introduced many programming concepts, such nested scopes, recursion, block-structures, data structure constructors, data types, abstract control structures, and several other programming language concepts relevant to many widely-used programming languages of today. Lisp is based upon the concepts of pure mathematical functions, higher-order operations on functions, and functions treated as values or objects as operands to other functions; the design was based upon a pure evaluation without regards to the underlying computer. In this overview several programming language concepts will be introduced. Throughout the book as related concepts are introduced, several examples of parts of languages will be used to illustrate the concepts from the perspective the user (programmer), designer, and implementor (translator developer). When the phrase, *concepts of programming languages*, is used, it is interpreted from at least one of these three perspectives.

The study of programming languages is woven within computer science, software engineering, and computer engineering. When studying the design and implementation of programming language one needs to utilize the theory of computation, to have a pragmatic view of programming languages for software development, and to have some knowledge of the underlying computer hardware representing the physical machines described earlier. When building translators, such as compilers, one needs to have a good working knowledge of design and implementation practices of software development, have an understanding of machine architectures related to computer engineering and has to have a working knowledge of abstract computational models, such as finite automata, push-down automata and know the relationship of these models to formal language and grammar theory. Software engineering design and implementation methodologies and processes along with new hardware technologies have influenced and continue to influence language design. In this chapter an overview of the use, design, and implementation of programming languages is introduced. Each concept is covered in detail throughout the book with some chapters devoted to one topic.

Fortran, COBOL, and Lisp were a few of the first widely-used high level languages. Now there are thousands of programming languages and many dialects of programming languages. We look at several reasons for why there are so many languages and examine several motivations for developing new languages. We also look at approaches to evaluating languages, especially the reasons for design and implementation decisions. Finally, we look a various approaches to the implementation of languages, namely the translation of languages and supporting runtime systems and libraries.

1.1 Abstraction and Models

Building software products is an engineering endeavor, therefore practices and concepts used by other engineering disciplines are needed. Similarly, software developers have to write code the simulates certain aspects of the natural world, such as in flight simulators, air-traffic control systems, and the physics in a video game.

Scientists and engineers use the concepts of abstraction and modeling in order to understand complex systems. Scientists try to interpret and understand a complex system within the natural world by forming models, an abstract representation of the chosen relevant objects and relationships between them. Deciding upon the relevant objects along with deciding upon what properties and relationships to represent is the primary issue in forming a given model. Part of the process involves deciding on the level of detail, what to and not to include in the model. In other words we exclude unnecessary details; this forms an abstraction of a system.

For example, in the study of collisions of objects, scientists develop models that consider the properties of mass, speed, and direction of the objects and arrive at the phenomena, such as conservation of momentum. In simulation software a software developer express a scientific model or design model into a model within the constraints of a computing model. Ultimately, the scientific model has to be realized on a physical computing device.

Engineers build artifacts or *artificial* systems, such as bridges, airplanes, and buildings, that need to be modeled before built in order to examine alternative designs that meet specifications within time and money constraints. Models provide engineers the ability to uncover potential problems of constructing the artifact and to help predict the money needed to construct a given artifact. Some artifacts are too big or too small to attempt to make without first studying them in an abstract models or at an affordable and practical physical size. Also, models have to be made to model the process of how the artifact will actually be built, the time-line needed, and a host of other items needed for the task of building a given artifact or artificial system.

Software engineers face challenges similar to that of other engineers. Ultimately, software engineers have to build programs as their artifacts. However, software engineers also have to use models and abstraction in: 1) uncovering requirements; 2) formally specifying the scope of the system; 3) the creating a design of the system; and 4)the implementation of the design in a given programming language. In this book, the term, *software*, includes not just the target program or programs making up the system, but all of the documents created during the construction of the program(s).

One of the general principles for understanding a system is thinking in terms of *levels of abstraction*. This principle of looking at a system in layers is used in many software systems, such as operating systems and layers of abstract/virtual machines. This will be used in extensively in building translators and support code, like runtime systems and virtual machines for the purpose fully implementing a programming language.

This book is used modeling and abstraction as they relate to the use, design and implementation of programming languages. Note that translators are themselves software artifacts and, thus, must be developed as any other software engineering endeavor. We now look at three general approaches or system models for having programs written in an abstract level that is higher than machine level executed.

Programming languages play a major role in software engineering. We also examine how software engineering principles of software designs and implementation strategies are utilized in building translators of high-level languages to either another high-level language or to lower-levels. The central constraint is that these translators must recognize all and only syntactically legal programs while preserving the semantics. In a related topic, we also look at physical or abstract machines and the nature of computation and relationship to programming languages. Software engineering principles used in designing and maintaining software also influence the design of programming languages. In doing so we need to look at computer science theory and software engineering practices. Throughout the book we see the interplay between the design of languages and the implementation challenges of building programming language translators, such as compilers and interpreters. The study of programming languages involves the foundations of computational theory, the use of principles of software engineering design, and the knowledge of thinking on all layers of abstraction, from the lowest-layer to higher abstraction layers.

1.2 Some Reasons for Studying the Use, Design and Implementation of Programming Language Concepts

Knowing how to program in more than one language helps expand occupational opportunities for software development, gives people working in a wide spectrum of disciplines, and gives options for solving problems. However, why should one be interested in the concepts surrounding programming languages? Most developers are not going to design a full language or write compilers or interpreters. Why should one be well-versed in the implementation of language or translators? There many reasons, some are given here. Now that we have a brief introduction to some fundamental terminology and programming language concepts, let's examine some motivations and rationale for study concepts of programming language theory, design, and implementation. It should be noted that some of the terminology and concepts mentioned may be new to the reader and not fully understood. The intent provide an introduction of some of the concepts to be covered in this book and to motivate further study in this subject. Let's start to answer these questions just given.

Enhancing ones ability to express and implement algorithms and data structures. Knowledge of a programming language helps one's understanding of a computational problem and the algorithm used. Like with natural languages, those fluent in a language can better express their thoughts and ideas for solving problems. Studying the design goals of a programming language and the underlying computational models or mathematical models for expressing the semantics of the language helps in not only implementing an algorithm, but may give rise to new insights into how to solve problems or may provide more efficient solutions based upon the underlying computational model. An example of this is the use of move-semantics in C++11(and following versions) for copy constructors or assignment-member functions or when overloading the assignment operator. In this case, knowledge of the language semantics and implementation of how objects and parameter passing leads to more time and space efficient translated C++ code.

Study of programming language concepts helps develop the ability to learn new languages. Change in the software and hardware industries is inevitable and ubiquitous. New hardware gives rise to new possible applications and provides room for new innovations. These new applications and innovations also results in

new approaches for software design and implementation strategies, which usually calls for new ways to develop software or to express solutions. Another situation where the ability to learn new language helps is when hired at a company that uses an internal, proprietary language. By studying how grammars formally define the syntax of languages, how grammars are used for building translators, one can have a more structured approach to learning a language and deeper understanding of the syntax of the language. Knowing how concepts apply to other languages helps with understanding the design and proper use or pragmatics of a new language.

Knowledge of constructs of one language helps simulate concepts in another language. Knowing how to program in a language that has built-in features to express and implement some abstract idea or software design, helps one simulate it in another language that does not have the corresponding features. Sometimes, a programmer does not have the luxury of the choosing or using the language of their choice. For example, suppose that a programmer has to program in C and that programmer knows C++. If one knows how to use pointers to functions, how to pass pointers to structures, and knows the meaning of static variables, the C programmer can simulate the implementation of classes and objects. Simulating iterators and generators of current languages (e.g. Python, Ruby) in older languages is another example. Later in the book we'll take a look at this in detail. This just serves as an example of how knowing programming language concepts that transcend programming languages is useful.

Viewing user interfaces, data cleaning, data manipulation, and report generation as mini-language design and implementation. Most software engineers do not directly design or build translators for full languages. However, designing a user interface is similar to designing a language. For example, the task of data validation in a front-end web page is a kind of syntax checking of legal or illegal strings. Often grammars or regular expressions are used to formally specify what is legal and illegal text for, say, a data field in a web form. The languages models (e.g., regular expressions, grammars) and translator-building tools (e.g., Antrl4, flex) can be used for identifying patterns in data, thus "mining" or extracting useful data or information. Similarly, knowledge of these tools can help develop software for identifying errors in data and for transforming data into another form. A particular pattern represents a set of legal phrases or sentences that are members of a "mini-language." Another example is creating a report according to a specified format. Strings that adhere to the format forms a language, namely the language of legal reports as specified by the requirements of a client. Having a view of these tasks as languages helps develop programs for these common tasks. There are several programming languages that have libraries or built-in functionality that can be utilized for implementing these tasks. Moreover, this approach can be used when using software applications that process text documents, spreadsheets, multi-media documents, to name a few categories of data formats. XML is a mark-up language with a strict grammatical structure to represent data that is utilized by many web applications. Knowing how parsing works in a translator helps understand code to process XML code. There are many translator-building tools (e.g., Antrl4, yacc) that can be used for building applications to process XML. XSLT is language used for easing the effort in processing XML code.

Understanding utility tools, lower-level designs, impacts of hardware architecture can be useful. While higher-level languages exist to avoid examining low-level code or considering or tying the code to specific hardware configurations. However, sometimes when debugging code, the programmer must invoke the de-

bugger utility to look at the code generated by the a compiler or to look at how the supporting runtime environment handles stack management or the memory allocation. This becomes important when programming for embedded systems or an "internet-of-things" application. Knowing how the runtime works is related to the programming language semantics and language implementation; this knowledge helps the programmer debug code and/or write more efficient code. This is also important with security issues, such as buffer overflow problems with the runtime stack. Sometimes, when using several different languages with separate compilation one has to use a linker to change the way parameters are passed (e.g., stacked or queued) between assembler or machine code files. An example of how knowing the hardware configuration and how the compiler represents data structures is that it helps the programmer write code so that cache memory hits are higher. Knowing the parallel architecture or the distributed computer network topology can help the programmer write better code for parallel or distributive programming. For example, the simple knowledge of the number of processors and whether shared or local distributed memory is configured helps the way the code is written. Studying programming language theory helps ask the right questions and helps to know how to answer them.

1.3 The Evolution of Languages

There are over 1000 programming languages according to some counts. There are dialects, much like U.S. English versus British English. The difference between a dialect and a new language is not clear. Nevertheless, there are many languages and here is an attempt to try to address why there are so many.

New approaches to software design. In early days of software design it was clear that some structure was needed in expressing the flow of an algorithms representing programs. Flow-charts were used and in the late 1960s and early 1970s saw the introduction of the discipline of writing structured flowcharts. There should be one entrance and one exit for each process, where each process is a simple task or a selection of one of two sub-processes or an iteration of one subprocess. These three constructs were sequenced. Fortran and COBOL used goto statements and had no selection statements or iteration statements. So new languages were designed with if-then block, while-loop blocks, nested blocks, and other similar structures to reflect this structured design. Thus, structured programming was adopted and new languages all included the higher-level control constructs to reflect this software design approach. It became apparent that non-local variables shared between code in the nested blocks became difficult to maintain. Object-oriented design demonstrated many advantages in development, maintenance, and reuse of code and in the 1980s and 1990s new programming languages and dialects began to appear to reflect object-oriented design concepts. Languages like Smalltalk, C++, Eiffel, and Java were introduced.

Changes to new hardware technology and related software designs. Smaller and faster processors, denser and faster main memories, faster graphics processors, new display technology, and new input technology (e.g., mice, touch-pads) led to personal computers and mobile devices with graphical user interfaces, which led to the need for new approaches to software designs, such as event-driven program, related design patterns, such as the model-view-controller design pattern. New languages were developed to help reflect the software designs in a more direct way that with say older (but widely-used) languages, like C. C could be used, but

the implementing the designs were closely modeled and it was easier to express the underlying assumptions of the designs, such as concurrent operations. For example, threading was included in the semantics of the first versions, such as Java, Ruby, Swift, Objective-C, and C#. The concept of delegation is important in event-driven program and the concept of a delegate is in the language definition of C#.

Domain-Specific Languages. Lisp was developed for A.I. applications. In these applications symbols and nested data structures are important in representing algorithms and data structures in these applications. Prolog was used for processing natural languages and A.I. applications through the use of logic for representing knowledge bases. R is good for statistical applications. There are several dialects of Lisp and Prolog and they all have been used for more general applications. Some companies develop code for a particular product and rather than use a general-purpose language, choose to develop and in-house (sometimes proprietary language) for developing code.

Legacy Languages. Programs and software systems that was written originally in early versions of Fortran, PL/I, and COBOL still are in use today. There has been a large amount of time and money invested into some large systems of programs using languages. It would be too expensive to rewrite these systems with new languages. Some sub-systems' behavior can be simulated with new tools, such as with Excel macros, but several applications are tailored for specific applications in insurance, banking, government, and health institutions. New software engineers and programmers have to learn (at least part) of these languages even to interface with the code with more modern languages.

1.4 Reflections on the Purpose of Programming Languages

All human languages are used for communicating thoughts and ideas and to share knowledge. One use of a language is to convey or express knowledge of how to complete a task or solve a problem, such as transforming some data into another form of data, how to construct an artifact, or to describe how to react to the surrounding environment given certain stimuli. Computer scientists call the steps to complete tasks, algorithms, which must be expressed in a language and they are used to solve computational problems. Human or natural languages are used for communicating solutions or algorithms directly from human to human through notations or sounds and sometimes only with sounds. Algorithms can be viewed as computations that are carried out by some machines or cooperative machines with or without some kind of memory and input/output interfaces, called a computer system or, simply, a computer. Computers or machines can be abstract (computational models) or physical. Languages used to express computations or algorithms on physical or abstract machines are called programming languages. Algorithms, computational models, computational models and computational problems are all terms that have rigorous meanings and in this chapter we begin to exam these terms and concepts for the providing the fundamentals of programming languages.

As done within the study of human-to-human communication of computations, we look at the description and use of programming languages for human-to-machine communication of computations. With all communication there is a sender and a receiver where the exchanged content is encoded in some language. In the case of human-to-machine communication, content (the message) are the instructions and are encoded with a programming language, the sender is a programmer who is the generator of the instructions in a

language and the receiver is a machine that is designed to decode the instructions and execute the necessary actions. A programming language also has to have the expressiveness for representing the objects, such as numbers, and their properties utilized within the algorithm. This is a very simplistic and birds-eye view of communication, languages, and the concept of a machine. It is intended to get the reader to think about relationships of algorithms, machines (or computational models), and languages and their roles in the concept of computation. It will provide the foundation of looking at use, design and implementation of programming languages. Before moving on, let's take a look at a simple example.

Example 1.1 *To help illustrate the use of some of the concepts and terminology introduced thus far, let's look at a simple computational problem. Suppose one person wants to communicate to another person on how to sum a finite sequence of numbers. We want to transform these positive numbers into one number, their collective sum. We need an algorithm expressed in some language, such as a natural language like English. Suppose the sender (e.g., teacher) in this case, might give the following instructions in English:*

> *"Keep a running total. Read the first number and let that be the current total. As long as there is another number, add that number to the current total. When there are no more numbers the total is your answer."*

The receiver (student) most likely will formulate some mental or physical model of a computer system to execute the semantics of the English phrases. From this example we see how human-to-human communication can take place. In natural language, many assumptions are made. In this example it assumed the total is initially zero since no numbers are read at the start of the task. There is no explicit statement about when the sequence ends; it may be assumed that there is some sort of ending signal from the list, such as no more numbers on the source document containing the numbers. When communicating with a machine or telling the machine how to carry out this algorithm, we need a more detailed and formal description of a language and machine.

When communicating an algorithm to a machine, we have to know how the machine works, use a specific syntax with rigorous semantics, and understand how the input data to the algorithm are retrieved and where and how to send the output. Shown below is one C++ program to represent the solution to the computational problem of summing a finite sequence of numbers. In contrast to the general problem and we had to pick a particular set of numbers, namely integers, which is represented by a finite set of integers on the underlying abstract machine, which is, in turn, restricted by the underlying physical machine. These are just a couple of items that have to be rigorously defined and specified.

Sum Function in C++

```
#include <iostream>
using namespace std;
int main()
{
        int sumTotal = 0;
        int num;
        while (!cin.eof()) {
                cin >> num;
                sumTotal += num;
        }
        cout << "Total = " << sumTotal << endl;
```

```
        return 0;
}
```

In this example we see how the principle of using levels of abstraction helps understand the solution. The reader or programmer can "abstract away" the details of the underlying file systems for processing input and output and the representation of integers (`int`s) and operations on the underlying abstract and physical machines. We see that we explicitly set the `sumTotal` to `0` and utilize the test for end of file by using a predicate function, `eof()`, to test whether the underlying input is at the end of the sequence. The user or program (shell script) running this program on a Windows (Linux) operating system, type a *control-z* (*control-d* to indicate the end of the sequence of integers entered through the console input.

A translator, namely a C++ compiler, translates this into C++ code according to the legal syntax into another language closer to the physical machine language (with its own syntax and semantics) so that the machine can execute the code and preserve the semantics, where the input and output relationship is preserved. This shows another use of looking at a system in terms of levels of abstraction. The default target code for the Microsoft C++ compiler is physical machine code (and optionally assembler code). In this chapter we will look at examples of Java code that translates high-level code into virtual machine code.

Example 1.1 raises several questions: how does a designer of the language specify the syntax and the meaning of syntax in a rigorous manner so that the programmer (or user of the language) understand what is legal syntax and what the meaning of the syntax phrases? Also, what are the best practices? That is, what are the best ways to use the language in ways that are efficient, readable, scalable in terms of input, and maintainable? The meaning is the semantics of a language and the best practices are related to the pragmatics of a language. In this text study the concepts related to and the relationship and interplay between *syntax*, *semantics* and *pragmatics* of several programming languages. An introduction to the meaning of the terms and an introduction of the interplay is introduced later in this chapter and studied throughout the book in detail.

1.5 Computational Problems, Computational Models, and Algorithms

The study of programming languages are interrelated with several and depends on other concepts of computer science, such as computational problems, computational models, and solutions. We look at these concepts in more depth in order to set the foundation of studying the concepts of programming languages. A *computational problem* is characterized by the relationship of input and output data, where output values are functionally dependent on the input values. What constitutes an input or output value depends on the problem. For the problem of finding the sum of two integers, the input data values are simply pairs of integers. For the problem of finding the shortest path in an edge-weighted directed graph, the input values are some structured representation of edge-weighted directed graphs and the output values are numbers that represent the total cost of the shortest path. Specific input values to computational problems are called *problem instances*.

An algorithm is viewed as a solution to a computational problem. An *algorithm* is defined as *A finite sequence of elementary, unambiguous instructions that when executed transforms input instances into correct output instances as defined the computational problem and terminates for all input values.*

A key concept of an algorithm is the notion of instructions, steps or operations that are elementary and unambiguous. What the instructions are and the meaning of each is dependent upon the *computational model*. The model can be abstract and, in some cases, may not be physically realizable. For example, Turing machines are computational models and has an infinite amount of memory. These were introduced by Alan Turning in 1936. The RAM-Machine model is another computational model; it the model most of our widely-used computers use: processor with on-board memory (registers) with a random-access memory for storing instructions and data. Another model is the rewrite system of lambda calculus designed by Alonzo Church in 1936. Lisp is based upon the lambda calculus and one approach to defining the semantics of a programming language is using denotational semantics which relies on the lambda calculus. If you used lambda functions in a high-level, the term, *lambda*, comes from the anonymous functions introduced in Church's lambda calculus.

Note that in the definition of an algorithm, there is a requirement of termination for all input instances as defined by the computational problem. Some computational problems can be proven not to have an algorithm. For computational problems where there is a partial solution in the sense that it terminates on some input values, we say that the problem is partially solved with a *procedure*. A procedure has all the requirements of algorithm except with the possibility of not terminating on some inputs. We might ask is it because the model of computation is not powerful enough? It can be proven that each of the computational models mentioned above can be modeled by the other two. So all are considered equally powerful. There are other models that have been introduced for executing or computing procedures, for example, Petri-Nets with inhibitor arcs. No one has ever come up with a model of computation that is more powerful than any of these models. When a new model is introduced one of these other models are used to compute any procedure via the other model, e.g. a Turing machine. In other words when a new model of computation can execute or simulate a Turing machine or any equally powerful model, we say that it is Turing complete. A language is said to be Turing-complete if it can simulate a Turing machine. Such a language is a ***programming*** language. C++, Python, Java, C#, and widely-used languages for representing procedures and algorithms, are Turing complete. A example of a language that is used in compute applications that is not Turing complete is HTML5. When we study the programming language concepts of control statements, we will see for languages based upon on the assignment operator and statement must have sequencing, selection (e.g., if-then statements), and iteration (e.g, do, while, for, ...), are Turing complete.

The set of elementary instructions and the input and output data along with components of the computational model (such as memory cells or registers) all require names, syntactical structure and meaning to represent algorithms and procedures. Thus, a language is needed. Often times informal notations and restricted natural languages are used to express algorithms within the framework of a computational model. When a physical computational model is used, more formal languages are used. One may think of programming languages as concrete representations and realizations of abstract algorithms.

Let us begin with a specific problem to help illustrate and these concepts.

Example 1.2 The power function, usually notated in mathematics textbooks as x^n whose meaning is a shorthand notation of the product of n copies of x, where $n > 0$. For $n = 0$, x^0 is defined to be 1. A crucial part of characterizing or distinguishing a computational problem is specifying the domain of inputs. For this problem, let us restrict x to positive integers and n to non-negative integers. A computational problem can be specified set-theoretically by listing the relationship between the input and output. In this case, one would match ordered pairs (x, n) with the corresponding result, x^n:

$$\{((1,0),1),((2,0),1),\ldots,((2,3),8),\ldots((3,3),27),\ldots,\}$$

Alternatively, one could express the meaning of x^n in terms of the definition used in mathematics by using an agreed upon primitive operation or operations, such as multiplication. One define what x^n means as follows:

$$x^n = \begin{cases} 1 & \text{for } n = 0 \\ xx^{n-1} & \text{for } n > 0 \end{cases}$$

The random-access machine (RAM) is a common computational model that we will use. It is based upon a model first proposed by John von Neumann. The architecture schematic is shown in Figure 1.2.

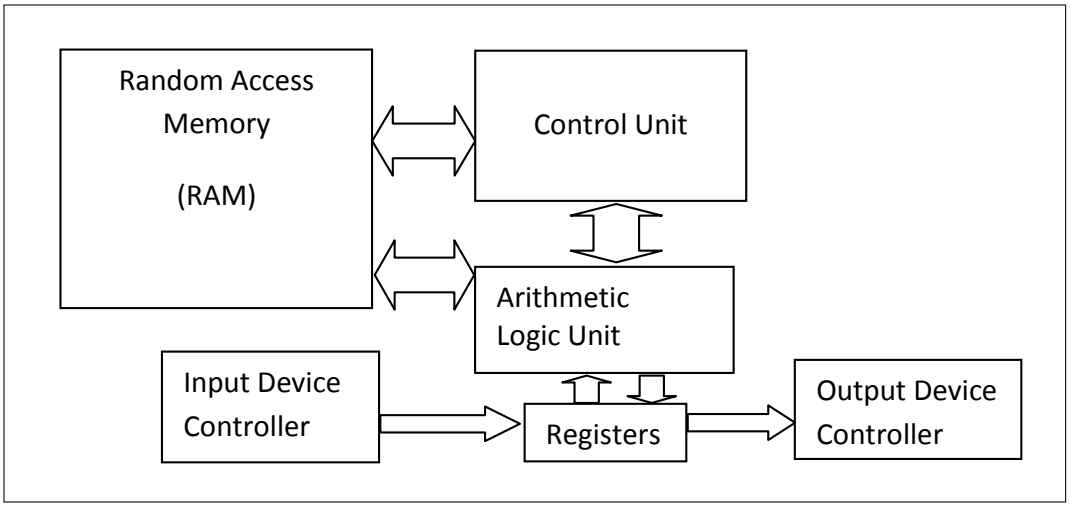

Figure 1.1: RAM Model of Computation

A pseudo-code language can be used to express a solution to the above computational problem of computing the power operation. In this pseudo-code programming language, the fundamental operations are assignment (`<-`), input, output, comparisons, looping structures (`for`), addition, and multiplication.

Power Function in Pseudo-code

```
input x
input n
product <- 1
for( i <- 1; i <= n; i <- i + 1 ) product <- product * x
output product
```

Contrast this code with the mathematical definition. The mathematical definition is time and state independent, whereas the pseudo-code executed by an abstract RAM-model is time and state dependent. The memory component keeps track of the state of the computation.

The family of Intel x86/IA64 modesl of processors are based upon the RAM model of computation. The model is defined by a finite set (vocabulary) of instructions. Essentially, the instructions are ones that carry out assignment, input, output, comparisons, jump-on-condition, addition, multiplication, and others. CPU manufacturers, like Intel and AMD, implement the realization of these instructions with different electronic circuits and variations of physical layouts. These instructions have to be encoded and these are interpreted as numbers and base-positional notation can be used; base-16 is often used. In essence this language is appropriately called a machine language. A fragment of machine code for the Intel x86 instruction set is shown below. The numbers on the left are base-16 address and to the right are the instructions encoded in base-16.

machine code

```
  00000        55
  00001        8b ec
  00003        81 ec d8 00 00 00
  00009        53
  0000a        56
  0000b        57
  0000c        8d bd 28 ff ff ff
  00012        b9 36 00 00 00
  00017        b8 cc cc cc cc
  0001c        f3 ab
  0001e        c7 45 f8 01 00 00 00
  00025        c7 45 ec 01 00 00 00
  0002c        eb 09
  0002e        8b 45 ec
  00031        83 c0 01
  00034        89 45 ec
  00037        8b 45 ec
  0003a        3b 45 0c
  0003d        7f 0c
  0003f        8b 45 f8
  00042        0f af 45 08
  00046        89 45 f8
  00049        eb e3
  0004b        8b 45 f8
  0004e        5f
  0004f        5e
  00050        5b
  00051        8b e5
  00053        5d
  00054        c3
```

Because this is tedious to code in this language, alphabetic characters were used as mnemonics for instructions, data names, and memory location names. Because the circuits are designed to recognize bit patterns and translation is needed. Such as translator is called an assembler. An example of an assembly language for the power function and could be translated into machine code such as shown above is shown below.

Power Function in an Assembly Language

```
_TEXT        SEGMENT
_i$19975 = -20                          ; size = 4
_product$ = -8                      ; size = 4
_base$ = 8                          ; size = 4
_expo$ = 12                         ; size = 4
?powerFun@@YAHHH@Z PROC             ; powerFun, COMDAT
; Line 20
  00000 55                  push ebp
  00001 8b ec               mov ebp, esp
```

```
  00003 81 ec d8 00 00 00    sub esp, 216               ; 000000d8H
  00009 53                   push ebx
  0000a 56                   push esi
  0000b 57                   push edi
  0000c 8d bd 28 ff ff ff    lea edi, DWORD PTR [ebp-216]
  00012 b9 36 00 00 00       mov ecx, 54                ; 00000036H
  00017 b8 cc cc cc cc       mov eax, -858993460        ; ccccccccH
  0001c f3 ab                rep stosd
; Line 21
  0001e c7 45 f8 01 00 00 00 mov DWORD PTR _product$[ebp], 1
; Line 22
 00025  c7 45 ec 01 00 00 00 mov  DWORD PTR _i$19975[ebp], 1
  0002c eb 09                jmp  SHORT $LN3@powerFun
$LN2@powerFun:
  0002e 8b 45 ec             mov eax, DWORD PTR _i$19975[ebp]
  00031 83 c0 01             add eax, 1
  00034 89 45 ec             mov DWORD PTR _i$19975[ebp], eax
$LN3@powerFun:
  00037 8b 45 ec             mov eax, DWORD PTR _i$19975[ebp]
  0003a 3b 45 0c             cmp eax, DWORD PTR _expo$[ebp]
  0003d 7f 0c                jg  SHORT $LN1@powerFun
; Line 23
  0003f 8b 45 f8             mov eax, DWORD PTR _product$[ebp]
  00042 0f af 45 08          imul eax, DWORD PTR _base$[ebp]
  00046 89 45 f8             mov DWORD PTR _product$[ebp], eax
  00049 eb e3                jmp SHORT $LN2@powerFun
$LN1@powerFun:
; Line 24
  0004b 8b 45 f8             mov  eax, DWORD PTR _product$[ebp]
; Line 25
  0004e 5f                   pop edi
  0004f 5e                   pop esi
  00050 5b                   pop ebx
  00051 8b e5                mov esp, ebp
  00053 5d                   pop ebp
  00054 c3                   ret 0
?powerFun@@YAHHH@Z ENDP             ; powerFun
_TEXT        ENDS
```

A C++ representation of the algorithm for computing raising x to the n^{th} power using the RAM-model as a computational model is shown below. The formal parameters of the power are `base` and `expo` for actual parameters `x` and `n`, respectively. Note that the line numbers are listed for correspondence to the assembler listing above; the starting line number of `/*19*/` is arbitrary.

Power code in C++

```
#include <iostream>
using namespace std;
int powerFun(const int, const int);

int main()
{
        int x = 5;
        int n = 3;
        cout << "Result = " << powerFun(x,n) << endl;
        return 0;
}
//...
/*19*/ int powerFun(const int base, const int expo)
/*20*/{
/*21*/        int product = 1;
/*22*/        for(int i = 1; i <= expo; i++)
/*23*/                product = product*base;
/*24*/        return product;
/*25*/}
```

Most readers have programmed in C++ or a language similar to C++ and understand the meaning of the above code. To many C++ programmers, an abstract model of a RAM-model is usually used implicitly to understand the meaning of the code. The variables each have a corresponding memory location and the control unit steps through each assignment statement and iterates through the loop by updating the variables `i` and `product`. Each new assignment and each step advanced changes the configuration of the machine. In reality, the machine code version is the code that actually is executed and is the code that updates the state or configuration of the physical machine.

One can take a closer representation of the mathematical definition of x^n by using the recursive semantics of C++ and concisely define the `powerFun` as follows.

Power code in C++ using recursion

```
int powerFun(const int base, const int expo){
         return expo == 0 ? 1 : base*powerFun(base, expo - 1);
}
```

1.6 Syntax, Semantics, and Pragmatics

These terms have been used intuitively thus far. We begin to look at these more formally and in the Section 1.7. Several chapters are devoted syntax analysis and approaches to defining semantics. The entire book is devoted to the pragmatics of programming languages.

Programming language syntax is defined by a set of legal strings of symbols. These legal strings are called sentences. A language can be finite or infinite. The symbols are strings formed from a finite alphabet; the symbols are treated as atomic by the grammar that defines which strings are sentences in the language. How Example of assignment statement in Python

The meaning of the programs (sentences) is the called the semantics of the language. There are three general approaches to semantics: operational, axiomatic, and denotational. Operational semantics involves how the sentences correspond to operations carried out on an abstract machine, such as a von Neumann machine. Axiomatic semantics are based upon first-order logical statements. Before each construct, say an assignment statement, a precondition is stated and then post-condition is given for describing what is true after the construct is executed. Denotational semantics uses mathematical domains and higher-order functions to map language elements to mathematical domains.

The pragmatics of a language is how to use the language as intended by the designer and how to use the language efficiently according to the implementation.

Let's take a look at a assignment statement found in Pascal: `x := 3*y;` .

Like many assignment-statement-based languages, the syntax has the form ¡var¿ := ¡expression¿. In Pascal, semicolons separate statements and is not really needed unless another statement follows this one. What could follow this is a reserved word, such as "end" or "END" and the semicolon is not needed. (The letter case is ignored in Pascal names). The expression on the right is a legal expression and written infix notation. Later, will begin to look at how to formally define expressions in languages.

The operational semantics might be described by stack-based instruction set for a von-Neumann machine. For example, one might write

```
ipush y //move address y contents interpreted as an integer onto the top of a stack
ipush 3 //push a integer literal 3 on top of the stack
imul //pop stack twice and store top two integers into registers r1 (3), and r2 (y), multiply and push the result.
istore z //pop the stack and store integer into a register r1 and store into z.
```

From an axiomatic semantic approach, we might write:

`{ x = 3 and y = 6} x := 3*y; {x = 18 and y = 6 }`. The braces are for surrounding the pre-condition and post-condition axioms.

Using denotational semantics all the syntactical elements are mapped to mathematical objects of two, multiplication, some function that maps memory cells to values and then maps those function a new function that shows the new mapping from memory cells to values with the x updated.

The pragmatics here is the purpose of an assignment statements. Why did the designer include this? Because the model of the language requires the programmer to move data around according assigning memory cells values through computations involving other memory cells. This is an example of an imperative language paradigm or way of solving problems or expressing algorithms. Language paradigms are discussed later.

1.7 Translation Models

What we have observed is the need for a translator and it must preserve "meaning" or semantics from one language to another. We call a language like C++ a high-level language because of the level of abstraction in contrast to the lesser abstract level of the physical machine. High-level programming languages allow the programmers to express programs that represent both natural and man-made systems in a more convenient way than in lower levels, such as assembler or machine code. Programming language translation can be

viewed as a computational problems where (source) programs written in a programming language must be translated to a semantically equivalent program expressed in a target language. The translator must synthesize a program representing the algorithm or procedure that the source code represents except using the underlying computational model of used for the target language. The target computational model could be a particular RAM-machine, like a PC, a mobile device, or processor embedded in a larger system like a microwave oven or thermostat. It turns out that the computational problem of programming language translation is a computational problem that is computable and thus has an algorithm. We need to look how to express these algorithms using programming languages. It turns out that these translator are complex and good software engineering design methodologies and software tools are necessary.

Next we look at various translation models, design architectures and how theoretical language models are utilized in the construction of translators. In this section we examine three translation models: compilers, interpreters and a hybrid of the first two approaches. For all three models we refer to the code (program) to be translated and as the *source code* (program). The programming language of the source code is appropriately called the *source language*.

1.7.1 Compilers

A compiler is a translator that translates the source code into code expressed in another language. This translated or compiled code or program is appropriately called *target* code (program). The language of the target code is appropriately called the *target language*. Each compiler is characterized by its source and target languages. Most compilers have a primary target language, but can optionally produce multiple, but related, target code, such as machine code and corresponding assembler versions.

For the most part, compilers generate very low-level code. However, there are some compilers whose target language is another high-level language. For example, in the early 1970s, RATFOR, which stands for Rational Fortran, provided programmers with higher-level constructs than found in Fortran IV. The RATFOR-compiler translated RATFOR into Fortran IV.

1.7.2 Translation: Preservation of Semantics

A compiler must be preserve semantics. How to specify and express the semantics of any programming language is a deep and on-going area of research in programming language theory. For an introductory view and sake of an intuitive understanding of what is meant by preserving semantics let us simply say that the relationship between input and output should be the same for both source and target code according to their respective semantics. For now, how each input item is transformed to its respective output value is based upon some kind of abstract operational model.

A compiler itself is a program and must be written in a language and possibly translated to a particular machine. Suppose that a compiler, C_{L_1}, whose source and target languages are L_{M_1} and L_{M_3}, respectively. Let M_1 and M_3 be the abstract or physical machines that provide the architecture for understanding the semantics of programs written in L_{M_1} and L_{M_3}, respectively. The compiler C_{L_1} could be written in language L_{M_2} and that runs on a machine M_2. Note that compilers are usually written in a high-level language or multiple high-level languages must be translated to L_{M_2}. Shown in Figure 1.2. Note that the input and

output of the program written in the source language, L_{M_1}, is usually abstract and in the mind of the programmer. The input and output of the target program are usually realized in physically in a computer system.

In most environments, the machine on which the compiler runs is usually the machine of the target language; i.e., $M_2 = M_3$. However, there are situations where the target machine is a special purpose processor or system of processors that do not have development environments for compilers. For example, some special purpose embedded systems, such as an aircraft avionics system, do not have compiler environments. Compilers that generate target code for such situations are called *cross-compilers*.

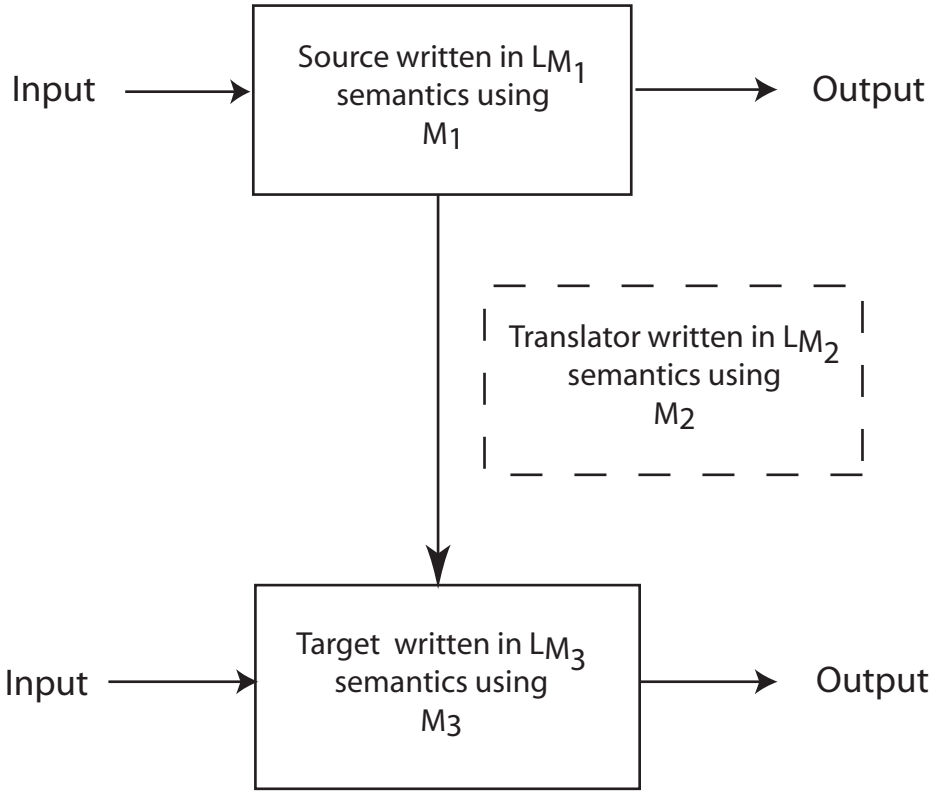

Figure 1.2: Overview of Compilation

1.8 Overview of the Translator Architectures

In this section we differentiate translators by examining the general architectures of each. Generally speaking, there are three general categories: compilers, interpreters, and a hybrids of the compilers and interpreters. We start with the compiler architecture.

1.8.1 Compiler Architectures

Because compilers are software products and because languages have different syntax and semantics, there are many different compiler designs and implementations. Shown in Figure 1.3 below is a general software

architecture used for many compilers. Some languages include preprocessors that modify the original source code before the source code is submitted to the compiler. For example, C and C++ have preprocessors the handle the `#include`s, `#define`s, etc. before the source is sent to the compiler.

Also, some compilers give programmers the option to generate assembler versions of the target code.

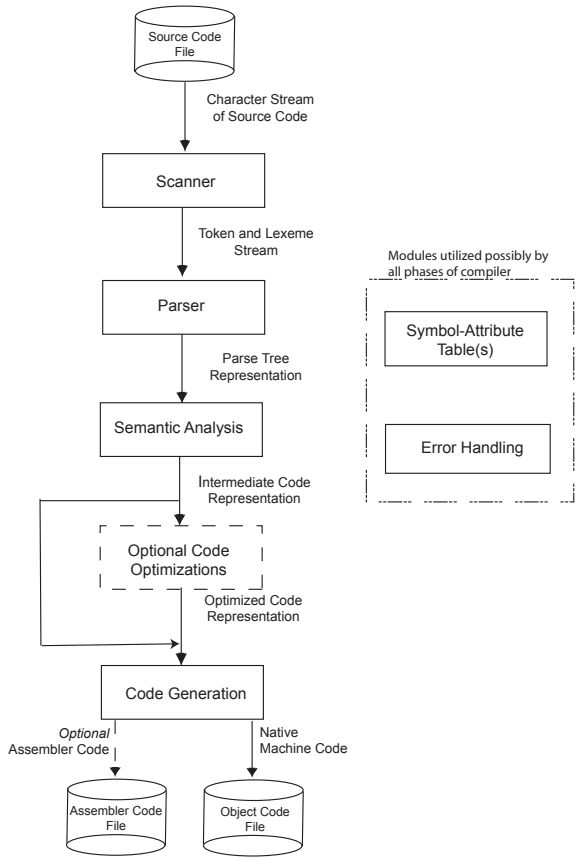

Figure 1.3: Compiler Architecture

Lexical Analysis

Lexical analysis involves reading the source code character by character, eliminating whitespace (space, tab, newline characters), removing comments and recognizing *lexemes*. Lexemes are the meaningful strings that are treated as "atomic" in the language. For example, "sum", "`<=`", "for", and '=' are all lexemes. This process is also called scanning. We will use lexical analysis and scanning interchangeably.

For each lexeme, a *token* is produced by the scanner. Each token has the form: (token_name, attribute_value). Each token_name represents a set or category of lexemes. Each token_name is a (hopefully) meaningful name for the category and is often represented as a named numeric constant internal to the compiler. Some token_names represent many lexemes. For example, a token_name, IDENT, might represent all legal identifiers. A parser may need to know the lexeme also, so a token such as (IDENT, "sum") would be returned by a scanner.

For other token categories, a single lexeme makes up the category. For example when recognizing these categories the scanner may send back the lexeme alone just the numeric code for the corresponding to-

ken_name. For example, recognizing "<=", "for", and '=', the scanner could return (LE), (FOR), and (EQ) as the named codes for the token categories. Note in some compiler designs the reserved identifiers (also called reserved words), such as "for", can be returned as an identifier and the parser is left with the task of determining whether it is a reserved identifier or a pre-declared identifier or some other keyword.

For identifiers and numbers, the lexemes are needed for other stages of compilation. For example, when an IDENT is found, the semantic analyzer will need to also know the specific lexeme associated with each IDENT, since it will have to either store or find the lexeme in the symbol table. When it comes time to either generate intermediate code or generate actual target code, the lexeme associated with each NUMBER token will be needed since the number may need to be stored as an immediate operand in an instruction or assigned to a temporary variable.

Meaningful token names are chosen by compiler designers. (Hopefully, they are meaningful.) In the compiler code, the names are name constants of some numeric codes. Shown below is a fragment of code that could come from C, C++, C#, or Java or from many other languages. It shows how the stream of tokens returned by the scanner.

for(i = 1; i <= expo; i++)

The tokens produced by the scanner and sent to the parser: FOR_TOK, LPAREN, IDENT, "i", EQ, NUMBER, "1", SEMICOLON, IDENT, "i", LE, IDENT, "expo", SEMICOLON, IDENT, "i", DBL_PLUS, RPAREN.

Depending on the semantics of the language, sometimes the lexemes for identifiers and numbers are stored in a symbol table by the scanner, and the lexeme portion of the token tuple is actually an index into a symbol table. However, if the language is syntactically structured with a declaration section and has semantics that does not allow re-declaration of variables, it is may be best to design the scanner to send back the lexeme to the syntax analyzer so that semantic analysis can detect re-declaration or other related semantic issues.

In developing scanners, compiler implementors need to know the legal lexemes and their categories as specified by the designer or committee of designers. To define the legal lexemes, regular expressions or regular grammars are used. For example in C the legal identifiers are comprised of at least an underscore character or an upper- or lower-case letter (A-Z) followed by zero or more underscores, upper- or lower-case letters, or digits (0-9). Legal C identifiers can be represented by these regular expressions:

$$\begin{aligned} letter &\rightarrow ('a'|'b'|\ldots|'z'|'A'|'B'|\ldots|'Z'|'\_') \\ digit &\rightarrow ('0'|'1'|\ldots|'9') \\ IDENT &\rightarrow letter(letter|digit)* \end{aligned}$$

For a given language, all lexemes can be defined be regular expressions. Combining all the regular expressions defines the legal lexemes for a given language. From the regular expressions, compiler writers can implement scanners. Part of implementing lexical analysis, one must watch for lexical errors. These occur when a character is encountered that does not belong to any legal lexeme.

Alternatively, instead of writing a scanner directly from the regular expressions, regular expressions can be translated into and represented by a deterministic finite automaton (DFA). A DFA is made up of a finite set of states and transitions between these states. A DFA reads an input string character by character,

making states changes. Which state is next depends on what state the machine is and the current character being read. A scanner can be implemented by simulating a DFA created from the regular expressions. We shall see this in more detail alter. There are scanner generator tools, such as `lex`, that take a collection of regular expressions and build a corresponding DFA.

Syntax Analysis

The parser analyzes the grammatical structure of the source code and, if the syntax is correct, produces a parse tree representation of syntactical structure. For each language, a grammar is used to define the set of all legal strings of tokens. The set of legal strings is what defines the language. The parser is ultimately answering the yes/no question is string $\alpha \in L$, where L is a language. All strings in L are called sentences. To accomplish this, a parser must show the steps and an ultimately the structure of the sentence (program) being translated. Since compilers are used to generate code that is semantically the same as the program being translated, a parse tree is used to represent the structure of the parsed sentence. Often the concrete parse tree is not actually built by the parser. It is a model for discussing the syntax. Usually the parser builds an virtual tree by simulating an abstract machine, such as a push-down automaton.

Example of a grammar. Shown below are the production rules.

Example 1.3 ***Simple Expression Grammar***

$$\begin{aligned} Expr &\rightarrow Expr + Term \mid Term \\ Term &\rightarrow Term * Factor \mid Factor \\ Factor &\rightarrow (E) \mid IDENT \mid NUMBER \end{aligned}$$

■

The symbols on left side of the arrow are syntactic categories and are called non-terminals in the grammar. The tokens IDENT and NUMBER are terminals. The lexemes "+", "*", "(" and ")" are used instead of token names. However we could use corresponding token names, like PLUS, MULT, LPAREN, or RPAREN.

The parser uses the production rules to try to produce the string of tokens being parsed. It attempts to do a series of rewrites by matching the left hand side symbols within the intermediate forms, called sentential forms. When we reach string or form of only terminals, we have a sentence. This series of rewrites is called a derivation.

An example of a derivation for the string `sum + 3.5*x` is shown below

$$\begin{array}{ccccccc} Expr & \Rightarrow & Expr + Term & \Rightarrow & Expr + Term * Factor & \Rightarrow \\ Expr + Term * IDENT) & \Rightarrow & Expr + Factor * IDENT & \Rightarrow & Expr + NUMBER * IDENT & \Rightarrow \\ Term + NUMBER * IDENT & \Rightarrow & Factor + NUMBER * IDENT & \Rightarrow & IDENT + NUMBER * IDENT \end{array}$$

The lexemes for NUMBER and IDENT are not shown.

Using grammar in Example 1.3 and the derivation above, a corresponding parse tree is shown in Figure 1.8.1 that corresponds to parsing the expression: `sum + 3.5*x`.

Rarely is the parse tree actually built. The parse tree is used to model parsing. The names correspond to how the parse tree is built (virtually). As a parser reads the stream of tokens from the scanner it makes

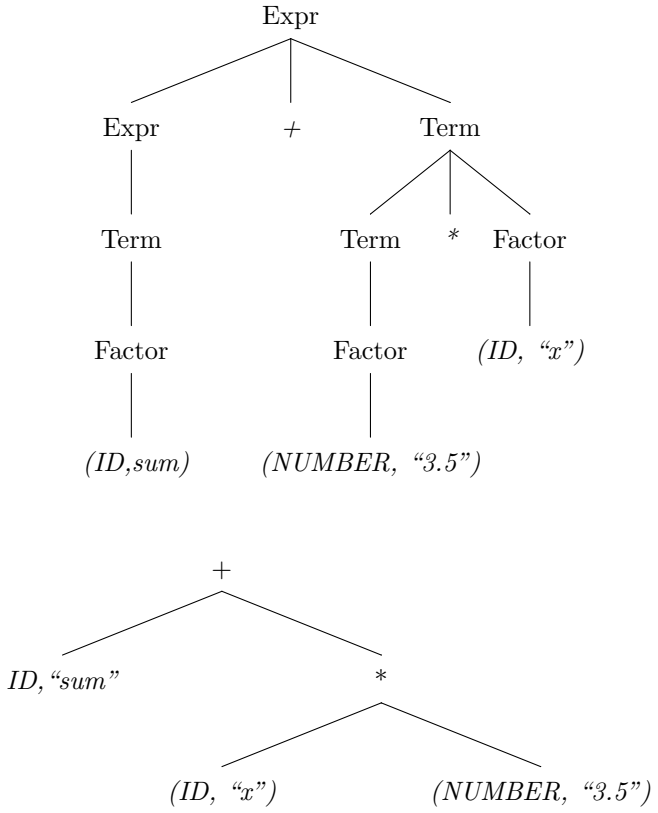

Figure 1.4: Abstract Syntax Tree

decisions about what production rules to used based upon the current token or some tokens ahead. The parser is designed around how the parse tree is built. In top-down parsing the parse tree is built from the root. Bottom-up builds the tree from the leaves upward, forming subtrees and forming subtrees from existing subtrees. We will see how these to approaches are implemented. Again the tree are often not built, but are used as models for building parse trees.

The grammars can be used to build an automaton that a parser can simulate. The class of automata that correspond to recognizing languages generated by a grammar are called push-down automata. This is analogous to building DFAs from regular expressions. Like scanner generators, there are parser generators. A parser generator that builds an automata and ultimately top-down parsers is `ANTLR`. An example of a parser generator that generates a limited class of bottom-up parsers is `yacc`.

As the parser parses, it does build a tree called an *abstract syntax tree (AST)*and represents the product of the syntax analysis phase. An AST has only the essential elements of a parse tree for the purpose of doing semantic analysis (e.g., type checking) and for generating intermediate code that will be eventually translated to machine code for a particular processor. For interpreters, an AST is traversed and actions executed as nodes are visited.

For example, an AST for `sum + 3*x` is shown in Figure 1.4.

The error handling components are responsible for issuing error messages and helping the parser recover from syntactic errors so that the remainder of the source code can be scanned and parsed.

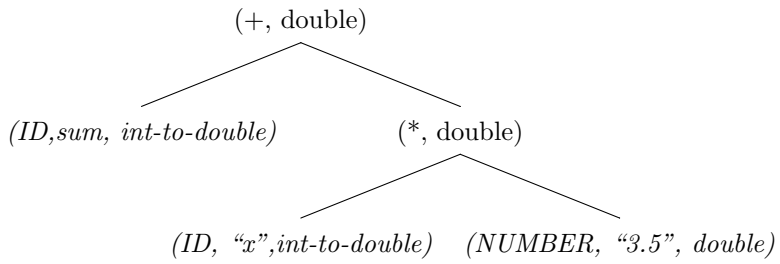

Figure 1.5: Abstract Syntax Tree with Attributes

Semantic Analysis

During semantic analyzer traverses the AST produced by the syntax analysis phase. The AST is traversed and as each node is visited or each subtree that is visited is dependent on the definition of the semantics of the language. For example, C++ compilers checks for the data type of each variable, detects variables declared with the same name, etc. The symbol table is a central abstract data type that is used for carrying out semantic analysis.

An AST general has most of the interior nodes of a parse tree removed. Also, additional information, such as data type and actions, are attached to nodes. For example, assume that x and sum are both declared to be of type integer, such as int in C++ or Java. Through consultation of the symbol table, the semantic analyzer initially sets the type attribute of nodes for "sum" and "x" to type int. The literal "3.5" is of type double and is marked accordingly. In this latter case, one could redesign the lexical analysis to distinguish between NUMBER types and introduce, for example, INT_NUM and DBL_NUM distinct tokens. As the operator nodes in the AST are visited, further type analysis is done. Since one of the operand of operator "*" is of type double and "*" needs both operands to be of the same type, type conversion from int to double is needed for the value of "x". Similarly, since the result of "*" as the right operand for "+", sum needs to be converted from int to double. This is a very simple example of doing type checking at compile time. Any kind of analysis that is done at compile time is referred to as static analysis. Shown is the result in Figure 1.5 is the annotated AST tree the semantic analyzer generates or at least uses internally.

For compiled languages, this analysis is driven by the semantics of the programming language. The parts of the semantics that can be analyzed at compile time, is called *static* semantics and the analysis is called *static* analysis. Some part of the semantics may need to be left for later phases or runtime. This is called dynamic semantics and analysis. For dynamic analysis, symbol tables (or at least parts) need to exist at runtime. For example, the semantics of polymorphism in many object-oriented languages require runtime analysis and symbol tables at runtime. For languages, such as Javascript, Python, ML, variables can change be bound to different types during their lifetimes. Thus, the type information about a variable must be kept in a symbol table at runtime.

Intermediate Code Generation and Optimization

The code generation usually occurs in two stages. Usually an intermediate language, internal to the compiler and designed by the compiler writers, is the initial target language and is used for optimizing code of the actual final target language. Again, the symbol table is used to generate code.

ASTs can have actions attached to the nodes. For example, suppose that we attached a “print” statement to each node for printing each lexeme. A postorder traversal of the AST tree above, would result int postfix expression: sum 3.5 x * +. This of course is the postfix version of the original. Instead of using “print” one could generate code into push, mult, add instructions as an intermediate language. From this intermediate language, the next phase of compiling could involve generating code for a particular CPU instruction set.

Once the intermediate code is created, optimization techniques can take place. In this phase the code optimizer transforms the intermediate code into a form that reduces the number of instructions, operations, the amount of temporary objects, and many other optimizations. Then the actual code generator takes the optimized intermediate code into code for a particular processor or collection of processors. Optimization can take advantage of special instructions for certain processors, such as vector instructions for parallel operations.

The target code of a compiler is usually not complete and other target code must be included. The linker utility program is responsible for connecting the target fragments into one complete target code.

1.8.2 Object-code to Executable

When a compiler produces target code for a particular machine is usually has some unresolved operand addresses, such as the location of a function call. This code is often called object code, which is stored in object files. In Linux and Windows environments the object files are designated as “.o” and “.obj” files.

To resolve this, a linker utility is needed. The linker is the final phase of building a program. It resolves the references between b

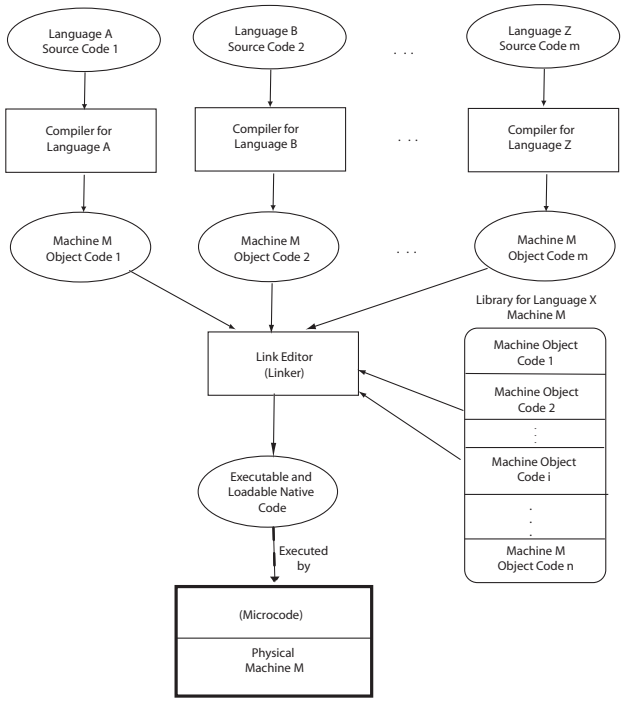

Figure 1.6: Compiler-Linker Relationship

1.8.3 Interpreter Architecture

Interpreters are similar to compilers in that their architecture includes a scanner and a parser. However, the input language is not translated to a language external to the interpreter code. Instead, an AST and/or an internal intermediate language is generated or some meaning construct is carried out by some software internal to the interpreter or passed to underlying software, such as the operating system, to emulate the semantics of the construct. Operating system shells or command interpreters are examples of such a translation architecture. Figure 1.7 captures the essence of the architecture of most interpreters.

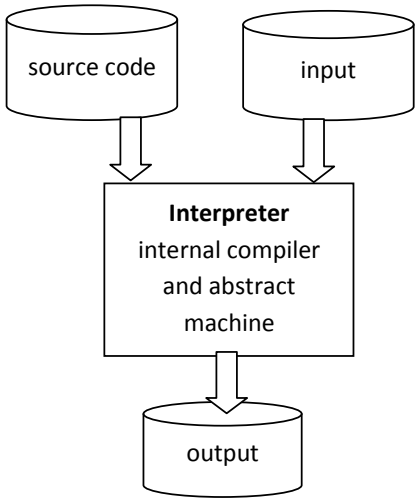

Figure 1.7: Interpreter Environment

Often, interpreters are implemented in a Read-Evaluate-Print Loop (REPL) environment. Operating system shells, such as bash and Powershell, operate in this way. Depending on the programming language, as variables are introduced through some sort of assignment statements (e.g. let x = 3) the variables are put into a symbol table and the type is often inferred. In an interpretive environment, the interpreter does scan, parse, and build ASTs. The interpreter traverses the ASTs.

One advantage of interpretation over compilation is the ability to test code without going through all phases of the compiler and then linking. Interpretation often involves traversing ASTs and, depending on the conditions of if-statements, while-statements and other control structures, the entire AST does not have to be traversed.

The advantage of compilation over interpretation is that the execution of the code can be faster. If an interpreter does not keep an AST, interpretation would be very slow since scanning, parsing and semantics analysis would have to be done for each meaning full phrase, e.g., one statement. Compiling also allows for optimization of the code for a particular processor or group of processors.

1.9 Hybrid

In this section we have a mix of compilation and interpretation. The target code generated resembles that of compilers, but instead is interpreted by an abstract, software-based machine, called a virtual machine. The most widely-used language with such a scheme is Java. The target code is called byte-code. Byte-code is interpreted by the Java virtual machine (JVM). This is shown in Figure 1.8. Python has an associated byte-code and Python Virtual Machine (PVM) and uses steps similar to Java's build steps.

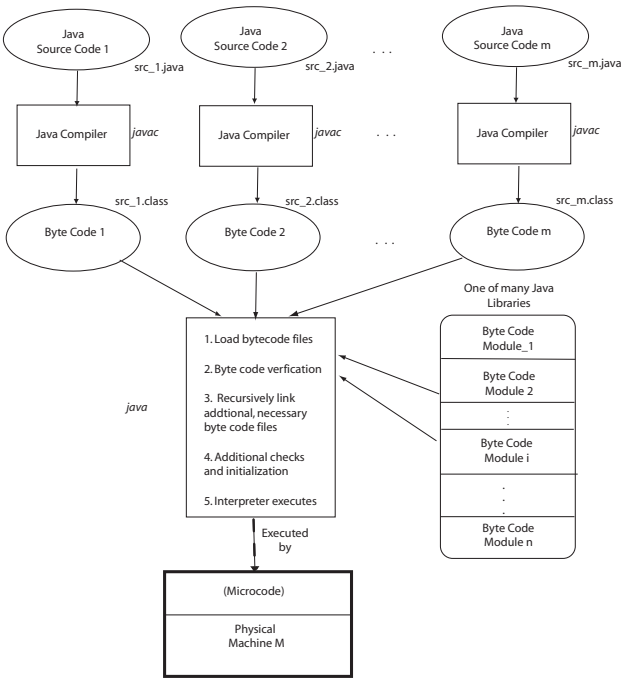

Figure 1.8: Interpreter Environment

Because interpretation of byte-code is slower than execution of native instructions by a processor's hardware, the concept of *Just-In-Time (JIT)* compilation is used by most Java implementations. Most environments now translate byte-code into machine code or at least portions of it.

1.10 Levels of Abstraction

As mentioned earlier in this chapter, one approach to software design look solve problems at different abstraction levels and have higher levels supported either by the level directly below or any of the lower levels. To implement a programming language on a particular operating system and hardware platform, more code is needed beyone the compiler or interpreter code. After the source code is translated to target code or into an AST (or other representation) that an interpreter can use, there are operations that usually need further support. For all language implementations, supporting code is needed for the runtime environments,

which include at a minimum, stack and heap management. Also, most high-level languages are defined to be independent of input, output, and network. The supporting code for all these mentioned are usually provided in a collection of static and dynamic libraries. Also, some kind of interface, adapter or facade code, written in the source language, is often provided along with the compiler. This is the case for all major languages.

Shown in Figure 1.9 is a depiction of the use of levels of abstraction for implementing many common translators along with general software applications. Operating systems are designed with the same approach of abstractions at various levels. Moreover, it is used for implementing secure operations and separating permitted operations for user and those operations of the operating system and hardware that need to be protected for the overall integrity of the entire software and hardware system. We see in Figure 1.9 that the operating system ultimately carries out the input, output and network communications. High-level translator engineering should not be concerned with writing or generating these lower-level operations for each source file compiled or interpreted. As a mentioned earlier and repeated for emphasis, this would not make the language portable if it were tied to a particular operating system or hardware platform.

We can see that even at the user-level there are levels of abstraction for the implementations. For example, we see that Microsoft has a Common Intermediate Language that has abstract operations at lower-level than the high level languages of C#, F#, and VB.Net. Linkers are utility programs that are usually closely tied to an operating system and compiler writers have to be aware of the restrictions and constraints of the underlying linker when generating relocatable code in an object file. All dialects of Ada have typing handled by both the compiler and linker, so a linker tied to Ada type semantics is needed.

When using input and output operations, the compiler translates them to calls to library routines that are linked into the executable image. Interpreters make calls to similar library routines. Some languages do not have specific input and output defined in the formal definition of the syntax and semantics. Rather there are some functions or operator symbols that are pre-defined in software interfaces and definitions in library files that are linked in. These extra interface source code and libraries are usually included with the translator. Installation of the translator requires bringing in the correct libraries for a particular operating system. Throughout the book we will look at these topics in more detail.

1.11 Porting and Bootstrapping Languages

In this section we look at how to move a compiler on one platform to another platform and how to build a new translator for a new language. We start by looking at how Pascal was developed and the process used to port Pascal to multiple platforms as designed by Niklaus Wirth, the inventor of Pascal. Then, we look at the general strategies of building new translators for new languages. Related to topic is the concept of "*bootstrapping*" a translator for a new language. Bootstrapping is derived from the idea of "pulling oneself up by one's own bootstraps." Creating a new compiler for a language and eventually using that language to write a compiler for the same language is analogous process.

The first process we cover is how a Pascal was ported when it was a new language. This could be used for many languages. Niklaus Wirth developed a virtual machine that interpreted P-Code, which is similar

<table>
<tr><td></td><td>Java Compiler/ Linker</td><td>C# Compiler</td><td>F# Compiler</td><td>VB.Net Compiler</td><td colspan="5">User script or interpreted program</td></tr>
<tr><td rowspan="2">User-level</td><td rowspan="2">Java Virtual machine</td><td colspan="3">.
Common Intermediate Language
Net
Common Language
Run Time (CLR)</td><td>User Programs (.exe)</td><td>Shell interpreter (e.g., bash, PowerShell)</td><td>C/C++ compiler /Run-time system</td><td>ML/Python and Other interpreters</td><td>Ada compiler/Linker /run-time system</td></tr>
<tr><td colspan="8"></td></tr>
<tr><td colspan="9">High-Level OS Support: Libraries & Utility Programs</td></tr>
<tr><td>OS Utilities, Libraries, and Wrappers around Kernel Calls</td><td colspan="3">Editors, Linkers, Loaders, Debuggers, OS GUIs</td><td colspan="5">File management APIs & libraries
Thread/Process Management APIs/libraries Memory management APIs/ libraries
Device management APIs/libraries
Network management APIs/libraries (e.g., TCP/IP implementation)</td></tr>
<tr><td colspan="9">OS Kernel Interface</td></tr>
<tr><td>High-level Kernel</td><td colspan="8">Process Scheduling, Job Scheduling, Memory Access Management, Thread Management (for most operating systems)</td></tr>
<tr><td>Supporting high-level code</td><td colspan="8">Process Creation, Process Destruction,
Memory Allocation/Deallocation, Paging, Segmenting, etc.
Device Management (e.g., low-level data management of disk blocks, network packets, etc.)
Interface software for Hypervisors.
Security (checking kernel-mode/user-mode, permissions)</td></tr>
<tr><td>Low-level Kernel</td><td colspan="8">Drivers
Interrupt Handlers
Timer Software
Security (e.g., checking instruction integrity)</td></tr>
<tr><td colspan="9">Hardware Interface</td></tr>
<tr><td rowspan="3">Hardware</td><td colspan="8">Physical Systems (Processors, RAM, Device Ports, Network Ports)</td></tr>
<tr><td colspan="8">Microarchitecture (microprogramming for some processors)</td></tr>
<tr><td colspan="8">Logic circuits, busses, hardware ports</td></tr>
</table>

Figure 1.9: Levels of Abstraction for Programming Languages and Operating Systems

to bytecode used by the Java Virtual Machine (JVM). He used P-code as the target language and wrote the first Pascal compiler in Fortran. He compiled a larger Pascal compiler using the p-code translation of the first Pascal compiler. The next compiler was written in Pascal and translated by the smaller version of Pascal. This was iterated until all the constructs and features were included. To distribute the Pascal compiler, he made available three machine independent files: the Pascal compiler in Pascal compiler, the P-code version of the Pascal compiler, and the P-code interpreter written in Pascal. If one wanted run Pascal on their machine, the programmer could write the P-code interpreter written in Pascal into a native language that had a compiler on their machine. Then they invoke the interpreter to process the P-code version of the Pascal compiler and feed in Pascal source, which is a Pascal application program.

Shown in Figurer 1.10, is a summary of the Pascal porting strategies. Note that the Pascal source is not necessary. However, if one wanted to generate native target machine code, one could edit the Pascal compiler to generate that native code instead of P-code.

Here is another approach to designing a new language and developing a translator for the new language. Suppose you have an idea for a new language, New. To write compiler for New, one must write the compiler using an existing compiler for another language, say L. Your compiler written in L could perform the syntax and semantic analysis and generate code for a real or virtual machine. Barjne Stroustrup did this with the first compiler for C-with-Classes, which later was renamed C++. What is described here is not exactly what he did, but it follows a general pattern. He first wrote a preprocessor in C that took C-with-Classes code and generated equivalent C code. New constructs and features were added incrementally to make C-with-Classes larger so that the preprocessor could be written in the previous version of C-with-Classes preprocessor. Then he ran the new preprocessor again. He continued until he could write a preprocessor to handle the full

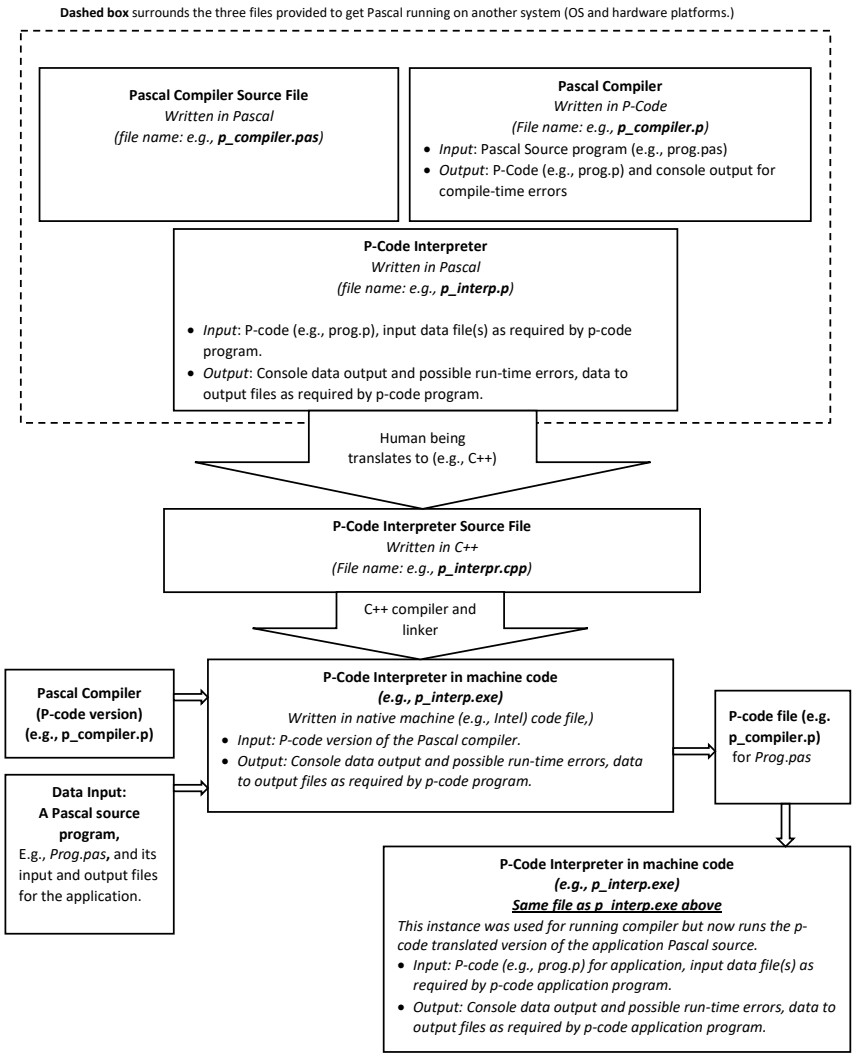

Figure 1.10: Porting Pascal

C-with-Classes. Since the target language with the full preprocessor is C, he could compile the C version into native code of the underlying platform. Ultimately, he had a *self-hosting* language: compiler or preprocessor implemented in a language that is the same as the source language.

An alternative way to generate a self-hosting language or compiler using a native C compiler to incrementally generate subsets of C++ and using the subsets as compilers for larger C++ dialects. This is another way to bootstrap a compiler for a language and another way to create a self-hosting language and translator.

As we from the processes used for generating translators for Pascal and C++, it can get complicated and confusing keeping track of the source, target, and implementation languages. An visualization of the process, we can use a T-diagram for each translator. Shown in Figure 1.11, is a generic T-diagram.

In Figure 1.12 one can see how one could use an existing language implementation to translate a new language. In this Figure 1.12 we see how a T-Diagram in the lower right is used to represent a C compiler that was translated to machine code M for execution on a machine M and for generating machine target

Figure 1.11: T-Diagram: Implementation Language of a Translator that translates a Source language to Target language

code for machine M. The C compiler in this case can be used to write a compiler for the new language and to generate target machine code M for the source language, *New Language*.

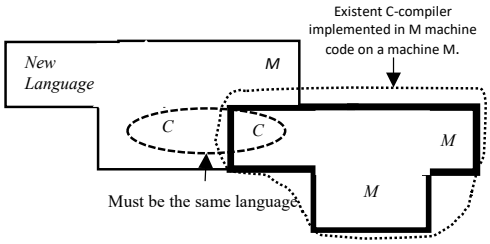

Figure 1.12: Example of a C Compiler implemented on with machine code M for a physical processor.

We can also use T-Diagrams to see how we can utilize more than one existent translator to create a new translation scheme for a new language or, at least new to the development platform. Suppose that we have a Java compiler and Java Virtual Machine and a C compiler that generates target code that is machine code for machine M. Suppose that we have a Java program that translates Python source code to C++ target code and another Java program that translates C++ source code to C code. We could put these two Java translators in a pipeline and then run the C compiler to generate native M machine code. This is shown using T-Diagrams in Figure 1.13. Note that the labels (language names) on the sides of the T's must match and do in this scheme.

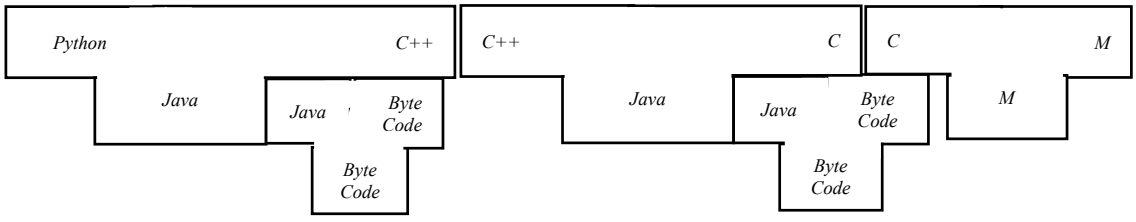

Figure 1.13: Creating a Python Compiler for Native Code using an implementation of Java and C compilers.

The next T-Diagram in Figure 1.14 represents how to build a self-hosting C++ compiler starting with a C compiler implemented on a machine with machine code M. We use two small versions of C++, C1++ and C2++. This is similar to the preprocessors that Stroustrup used for building the C with Classes (C++)

language and translators. Of course an attempt to write a full C++ compiler could be done instead of using these increments or iterations.

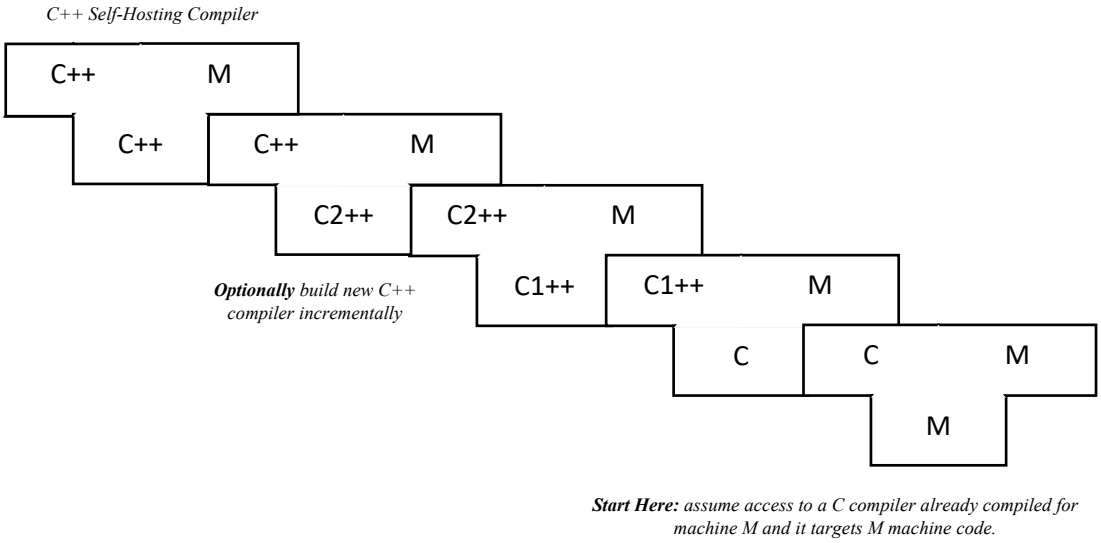

Figure 1.14: Bootstrapping to develop and Self-Hosting Language. Suppose C++ is a new language.

As we can see, it is important to know that a translator always has an implementation language (and interpreter or machine), a source language and a target language. By combining translators and doing some extra coding, we can derive new translators from existing ones. Part of reason for studying programming language theory is that it may give insights into developing new translation implementations that can solve some larger application problem. We will see this throughout the text.

1.12 Programming Computational Models Categorization

Programming languages can be classified into many different categories, such as the languages that are used for certain kinds of applications. That is discussed later. Here we look at a informal semantic model or machine used to capture how the programs compute or how the programs transform input data into output data for the procedure or algorithm. A coarse categorization can be those languages that declare *what* the underlying computer or machine should do are called *declarative* languages. Languages that describe *how* to control the machine to execute the procedure or algorithm are called *imperative* languages. We can break declarative languages into other categories based upon paradigms, which, loosely, are ways to solve problems or are ways the languages are translated and implemented.

RAM-Model-based, von Neumann languages are those that have been described extensively so far and are based upon the model shown in Figure 1.2. Program instructions and data are stored in the same memory. These are languages are based upon on series of assignments of data with control statements to alter which assignments are completed. Languages in this category are also called imperative languages in that the programs in these languages are expressed as a series of imperative statements: "Do this. Then, do that.". Control statements alter the flow around what statements executed next. The assignment statements

modify the values of variables with the contents of a memory cell (variable) or the results of computations. In mathematics, one a variable is considered to have a value for a computation, it is assumed that it never changes. Here, the variables over the entire computations can changes values. So, the semantics of an imperative language most be modeled using some representation of the state or configuration of the memory of von Neumann machine. Assembly, Fortran, Algol60 and COBOL were considered imperative languages. Many languages today, use the assignment statement extensively, such as C, C++, Java, and hundreds other. If we view the assignment statement as being an expression consisting of a function, such as =(x,2) where = is the function, we can see that x could be modified to 2. In pure mathematics, f(x, ...) should not modify any variables in the parameter list or anywhere else in definition of a mathematical expression. This leads us to another category, functional languages.

Functional languages strive to be pure declarative languages that indicate what should be computed by using only mathematical functions and higher-order functions, like composition, function application to input (domain) values, and allows functions to be passed as data to other functions. LisP was the first functional languages. Many other followed as descendants of LisP, such as ISWIM, Scheme, ML, Haskell, and F#. Many of these languages are not pure and do invoke some sort of assignment, thus making it partially imperative. Functional languages are modeled with abstract models like lambda calculus, but, because of the widely-used computers are primarily based upon on the RAM-model, the abstract semantics are translated into a imperative code to be executed on a RAM-model.

Logic languages are declarative and based upon pure first-order logic or restricted syntactic forms of first-order logic languages. Like functional languages, the computations of the abstract model are modeled by RAM-models. In many these logic languages, a restricted form of first-order logic languages, such as Horn clauses, are used. The model of computation uses goal logic clauses (first-order clauses that are to be proven) and unification of variables with predicate parameters, and some search strategies to reach the goal are used. These are often interpreted on a RAM-model by executing a search algorithm, such as depth-first search with backtracking of a graph. Prolog and XSLT are examples.

Object-Oriented languages reflect object-oriented designs. These languages allow one to look at entities in the design as objects with several internal state variables and the objects have operations that can change the state of an object or inquire the state of an object. The operations may have parameters that received messages from other objects. Objects with the same state variables and operations are considered in the same class. The first object-oriented language is considered Simula-67. Smalltalk was one of the first pure object-oriented languages. C++, Java, and Python are the widely-used object-oriented languages used today.

Concurrent and Distributed languages are that have threads with some runtime systems or rely on the underlying threading system of the operating system. These threads can be executed in an interleaved (time-shared) manner on one processor or each are assigned to different physical processors and executed at the same time. This is referred to as true concurrency or parallelism. Many languages have semantic definitions that require the translator to assign threads to processors and/or schedule them. Some processors may have local non-shared memories and are connected through network interfaces and use message passing. Distributed languages are designed to express solutions on this latter configuration. ADA83/95/2012, Concurrent Prolog, C++11/14/17/20, C#, Concurrent Prolog, Java, Parlog, Python are just a few programming languages that

have threading and parallelism within the design definition. There are many libraries and interfaces that extend programming languages to express parallelism. For example, Cuda and OpenCL are libraries and runtime environments that extend C and C++. Erlang is a widely-used language that is used on top of distributed systems. The Message Passing Interface (MPI) library and runtime systems for distributed computing works with C, C++, and Fortran.

Scripting languages are used for programs to be run in an interpretive environment and can be used in read-evaluate-print-loop (REPL) interactive session where the evaluation is completed by the respective interpreter. Programs can be put in files that "batch" the commands, statements, and expressions and viewed as a script (hence the name of the language category)file). The script file is submitted to an interpreter. JavaScript (and CoffeeScript, TypeScript), Lua, Perl, PHP, Python, Ruby, Linux/Unix shells like bash, and Windows Powershell are examples of scripting languages. JavaScript and PHP are primarily used in the context of web-pages and handed off by the web-browser to appropriate interpreters. Lua, Perl, Python, and Ruby are used in a variety of contexts and applications. Languages like bash and Powershell are used to invoke several applications or executable commands by an operating system. All the scripting languages can be used to interface other programs and "glue" application together to make a larger applications or extend and application.

1.13 Evaluating Programming Languages for Use, Design, and Implementation

When evaluating a language for use in a software project there are many properties and attributes to be considered. While designing a language there goals that must be stated and met. Design requirements must be realized by the implementation of a translator and some of the features introduced in the design may present challenges to create efficient implementations in terms the time and/or space efficient interpretation or translation of the language or efficient target code. The implementors are also faced with the underlying constraints of the operating systems, virtual, and, ultimately, the physical computer system architecture. The following evaluation are interrelated and affects one or more of the use, design and implementation perspectives of a language.

1.13.1 Writeability and Readability

Evaluating this objectively by assigning some sort of metric is difficult. These are very subjective and the several other related criteria must be taken into account. These two attributes are related in that one language might thought of as readable, but hard to write. For example, in the design of a programming language, the designer chooses a number of reserved words and keywords out of a set legal identifiers, names for objects, functions, classes, and other "things." How many should there be? COBOL85 has over 300. C++11 has around 70. Choosing meaningful identifier names might be more difficult for languages with large number of reserved workd or pre-defined identifiers than those with less.

The grammar also affects how writebility and readability. Consider a simple assignment statement in C versus COBOL. In C, we might write we write `x = y + z;` whereas in COBOL we might write either `ADD X TO Y GIVING Z` or `COMPUTE z = x * y`. COBOL is much more readable, but more time-consuming to write.

Sometimes a programmer can write concise code using few characters. Moreover, it might be very efficient code. However, such code may be very difficult to read in the sense the another programmer (or the same programmer months or year later) has to know some low-level implementation and semantics details of operator semantics and, perhaps, not-often-used operator precedence rules. Consider writing code to copy a string array contents to another string in C++ code the defines and tests the function `mycopy()`. Example 1.4 illustrates such code.

Example 1.4 *This is a concise way of writing a string copy function and cryptic to new programmers of C. The reader must know that C-strings are terminated by the null character (*`'\0'`*), that the string variables are pointers to the first address of a contiguous group of memory cells bound to character values, that the dereferencing semantics associated the star operator,* `'*'`*, that the postfix operation of updating the local parameter, that the actual parameters of the calling code are not changed because the parameters are passed by value and have local scope, and that the assignment operator* `'='` *returns the value of the evaluation of the right operand, and that the value* `0` *(or, in this case* `'\0'`*) represents false which terminates the while loop as soon as the end of the* `source` *string is reached. This is easy to write if all the semantics and implementation of strings are known. Some will argue its easy to read if very fluent in the all the semantics of string representation and operator and control structures. Some programmers may consider introducing explicit loop-control variables, array element selection, and explicit test for the end of the source string.*

Cryptic C-string Copy

```
#include<stdio.h>
void mycopy(char* src, char* target) {
        while (*target++ = *src++);
}
int main()
{
        char s1[] = "Hello";
        char s2[10];
        printf("s1 is %s\n", s1);
        mycopy(s1, s2);
        printf("s2 is also %s\n", s2);
        return 0;
}
```

■

There are many important concepts and terminology that help evaluate the readability and writeability of programming language designs. The concept of orthogonality, the use of a language with a particular programming language domain, supporting libraries, and availability of integrated development environments (IDEs) play major roles in the evaluation of readability and writeability of a language.

Orthogonality.

When writing code, many exceptions to rules are difficult to recall. Sometimes features or constructs in programming languages interact in ways unexpected. Ideally, concepts should be independent. The term orthogonality in this usage is derived from orthogonal vectors in mathematics; the (perpendicular) vectors are independent of each other. When concepts or constructs are used together ideally there should be no interactions and they appear together in a small number of ways. For example in C++ and C, all data types, including structs or class objects, can be returned from functions or passed as parameters, but arrays cannot. However, one can pass pointers to array, struct, or class objects as parameters and the same pointers can be returned from functions.

Another C example if the use of multiple attributes for variable declarations and definitions. For unsigned integers, the two declarations have the same meaning: `unsigned int x;` or `int unsigned x;`. However, `const *int p;` means p is a contains an address to memory object containing an `int` that cannot be modified; and the meaning of `int const *p;` means `p` contains address of memory object containing an `int` value and `p` cannot modified, but the memory object containing the `int` value can be modified.

Expressiveness in Appropriate Problem Domains.

This is another criterion to evaluate writability. When one has the languages that can express solutions in languages developed for particular domains, one finds it easier to write code. One aspect of designing languages the designer seeks to find language constructs to represent data, objects, operations, tasks that are part of software designs that, in turn, model some world, be it real or artificial, such as a computer game. We want to use a language that allows us to map language concepts, objects, operations, functions, and tasks back to the design components and functionality. Languages like Java, C++, and Python have the expressiveness to implement object-oriented designs that are used in many problems domains. There are many problem domains and sub-domains. In the next subsection we look at a few and a sampling of a few languages. Assemblers, ADA, and C are good languages for embedded systems.

Problem Domains.

Some of these domains can have a large intersection of applications, but they are intended to provide an appreciation that some languages can be used in multiple domains.

- **Scientific.** MatLab, Octave, R, Fortran (all versions), C, C++, have operations and data types that can express numerical methods representing discrete mathematical algorithms that approximate the continuous mathematical models. A related aspect of evaluation of a language is support for efficient compilers. The compilers for these have to generate time and/or space efficient code. MatLab, Octave aadn

- **Web Applications.** JavaScript, TypeScript, CoffeeScript, Perl, and PHP are language examples that have constructs and dynamic data type semantics along with good string handling that allows one to express functionality for web applications, especially client code. The lower-level aspects of web-servers are best written in a system language like C.

- **Finance.** Many of the applications in this area reflect the same needs as scientific programming. There are also needs to interface with data applications programming interfaces. C++ is used widely in this domain.

- **Embedded Systems.** Processors and associated memory bus interfaces to a larger system is considered and embedded computer system. The appropriate language of choice depends on what level of abstraction the application resides in the system. Lower-level may require assembler. Most low-level code can be done with C. Some higher levels can be ADA (95 and later).

- **Artificial Intelligence (A.I.).** A.I. has so many subdiscipline. Early applications and many today require manipulated large structures of symbols. LisP was originally developed for this purpose. Prolog and other logic programming languages were developed to represent automated reasoning and natural language processing. Today, the area of machine learning uses many neural network tools and systems to model applications like training classification systems. These learning tools are programmed in languages like Python, C, C++, Java, and many others. However, when using the tools interpreted languages like Python have great interfaces and libraries to act as front-ends to these tools.

- **Computer Games.** There are many sub-domains of the game design. Action-script 3.0 is used for many games. C# together with the Unity game engine or C++ with the unReal game engine platform are appropriate for developing games. The important observation is it is not just the language for game development but a software technology platform and framework is important.

Integrated Development Environments (IDEs).

The writeability of languages should also be evaluated in the context of the particular problem domain, the support from libraries, and available development environments. If the problem domain is developing a graphical user interface (GUI), a language like Visual Basic or C# within the .NET framework and integrated development environment(IDE) like Microsoft Visual Studio may make it easier to write than a language like C++. This is also more readable for developers working in this problem domain. Using Java and its libraries are appropriate too for applications requiring a GUI. On the other hand Visual Basic or C# were not designed for a low-level operating systems programming.

1.13.2 Efficient Translators and Library Availability.

The availability of compilers that have the ability to optimize code and create time and/or space efficient code. As we study code generation later in the book, we'll see that there is often an trade-off between time and space efficiency. Fortran (all versions) generates good code and there are many C/C++/C# compilers (e.g., LLVM tool-chain)that generate very efficient code. Interpreted languages are good for testing ideas and quick development, interpreters are typically slower than compiled code. When running code a processor within an embedded system, like an avionic system, the translations of source code to a section of code not translated by an interpreter may take too long for handling a time-critical event.

1.13.3 Portable Languages

Java and Python are classic examples of a current languages that is portable in the sense that write once run any where. It is because they have hybrid translation schemes of compiling to virtual machine code, Java byte-code and Python byte-code.

Bibliography

[1] "Can Programming Ever be Liberated from the von-Nuemann Style," 1978 Turing Award Lecture Communications of the ACM, Vol. 21, No. 8, August, 1978, pp. 613–641.

[2] Gallier, J. H., *Logic for Computer Science*, Harper & Row Publishers, New York, N.Y., 1986.

[3] Halmos, P. R., *Naive Set Theory*, D. Van Nostrand Co., Princeton, N.J., 1960.

[4] Kleene, S., *Introduction to Metamathematics*, D. Van Nostrand Co., Princeton, N.J., 1952.

[5] Lindholm, T., Yellin,F., Bracha, G., Buckley, A., *The Java Virtual Machine Specification: Java SE 7 Edition*,Addison Wesley, 2013.

[6] Lloyd, J. W., *Foundations of Logic Programming*, 2nd edition, Springer-Verlag, New York, N.Y., 1987.

[7] "Regular Expressions and state graphs for automata," IRE Transactions on Electronic Computers EC-9:1,(1960), pp. 38–47.

[8] Mandrioli,D. and Ghezzi, C., *Theoretical Foundations of Computer Science*, John Wiley and Sons, New York, N.Y., 1987.

[9] NAUR, Peter (ed.), "Revised Report on the Algorithmic Language ALGOL 60.," *Communications of the ACM*, Vol. 3, No.5, pp. 299-314, May, 1960.

[10] Peano, G., *Arithmetices principia, nova methodo exposita,* Bocca, Turin, 1889. (Translated to English on pp. 83–97 in [11].

[11] van Heijenoort, J., ed., *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, Harvard University Press, Cambridge, Mass., 1967.

[12] Waite, G. and Waite, W., *Compiler Construction*, Springer Verlag, New York, N.Y., 1984.