Undergraduate Topics in Computer Science

Kent D. Lee

Foundations of Programming Languages

Second Edition

Kent D. Lee

Foundations of Programming Languages

Second Edition

Springer

Kent D. Lee
Luther College
Decorah, IA
USA

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-70789-1 ISBN 978-3-319-70790-7 (eBook)
https:\\doi.org\10.1007/978-3-319-70790-7

Library of Congress Control Number: 2017958018

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Contents

Introduction

1

This text on Programming Languages is intended to introduce you to new ways of thinking about programming. Typically, computer science students start out learning to program in an imperative model of programming where variables are created and updated as a program executes. There are other ways to program. As you learn to program in these new paradigms you will begin to understand that there are different ways of thinking about problem solving. Each paradigm is useful in some contexts. This book is not meant to be a survey of lots of different languages. Rather, its purpose is to introduce you to the three styles of programming languages by using them to implement a non-trivial programming language. These three styles of programming are:

- Imperative/Object-Oriented Programming with languages like Java, C++, Python, and other languages you may have used before.
- Functional Programming with languages like Standard ML, Haskell, Lisp, Scheme, and others.
- Logic Programming with Prolog.

The book provides an in-depth look at programming in assembly language, Java, Standard ML, and Prolog. However, the programming language concepts covered in this text apply to all languages in use today. The goal of the text is to help you understand how to use the paradigms and models of computation these languages represent to solve problems. The text elaborates on when these languages may be appropriate for a problem by showing you how they can be used to implement a programming language. Many of the problems solved while implementing a programming language are similar to other problems in computer science. The text elaborates on techniques for problem solving that you may be able to apply in the future. You might be surprised by what you can do and how quickly a program can come together given the right choice of language.

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_1

To begin you should know something about the history of computing, particularly as it applies to the models of computation that have been used in implementing many of the programming languages we use today. All of what we know in Computer Science is built on the shoulders of those who came before us. To understand where we are, we really should know something about where we came from in terms of Computer Science. Many great people have been involved in the development of programming languages and to learn even a little about who these people are is really fascinating and worthy of an entire book in itself.

1.1 Historical Perspective

Much of what we attribute to Computer Science actually came from Mathematics. Many mathematicians are programmers that have written their programs, or proofs in the words of Mathematics, using mathematical notation. In the mid 1800s abstract algebra and geometry were hot topics of research among mathematicians. In the early 1800s Niels Henrik Abel, a Norwegian mathematician, was interested in solving a problem called the quintic equation. Eventually he developed a new branch of mathematics called Group Theory with which he was able to prove there was no general algebraic solution to the quintic equation. Considering the proof of this required a new branch of mathematics, much of Abel's work involved developing the mathematical notation or language to describe his work. Unfortunately, Abel died of tuberculosis at twenty six years old.

Sophus Lie (*pronounced Lee*), pictured in Fig. 1.1, was another Norwegian mathematician who lived from 1842–1899 [20]. He began where Abel's research ended and explored the connection of Abstract Algebra and Group Theory with Geometry. From this work he developed a set of group theories, eventually named Lie Groups. From this discovery he found ways of solving Ordinary Differential Equations by

Fig. 1.1 Sophus Lie [21]

exploiting properties of symmetry within the equations [8]. One Lie group, the $E8$ group was too complicated to map in Lie's time. In fact, it wasn't until 2007 that the structure of the $E8$ group could be mapped because the solution produced sixty times more data than the human genome project [1].

While the techniques Lie and Abel discovered were hard for people to learn and use at the time, today computer programs capable of symbolic manipulation use Lie's techniques to solve these and other equally complicated problems. And, the solutions of these problems are very relevant in the world today. For example, the work of Sophus Lie is used in the design of aircraft.

As mathematicians' problem solving techniques became more sophisticated and the problems they were solving became more complex, they were interested in finding automated ways of solving these problems. Charles Babbage (1791–1871) saw the need for a computer to do calculations that were too error-prone for humans to perform. He designed a *difference engine* to compute mathematical tables when he found that human *computers* weren't very accurate [27]. However, his computer was mechanical and couldn't be built using engineering techniques known at that time. In fact it wasn't completed until 1990, but it worked just as he said it would over a hundred years earlier.

Charles Babbage's difference engine was an early attempt at automating a solution to a problem, but others would follow of course. Alan Turing was a British mathematician and one of the first computer scientists. He lived from 1912–1954. In 1936 he wrote a paper entitled, "On Computable Numbers, with an Application to the Entscheidungsproblem" [23]. The Entscheidungsproblem, or decision problem, had been proposed a decade earlier by a German mathematician named David Hilbert. This problem asks: Can an algorithm be defined that decides if a statement given in first order logic can be proved from a set of axioms and known truth values? The problem was later generalized to the following question: Can we come up with a general set of steps that given any algorithm and its data, will decide if it terminates? In Alan Turing's paper, he devised an abstract machine called the Turing Machine. This Turing Machine was very general and simple. It consisted of a set of states and a tape. The set of states were decided on by a programmer. The machine starts in the start state as decided by the programmer. From that state it could read a symbol from a tape. Based on the symbol it could write a symbol to the tape and move to the left or right, while transitioning to another state. As the Turing machine ran, the action that it took was dictated by the current state and the symbol on the tape. The programmer got to decide how many states were a part of the machine, what each state should do, and how to move from one state to another. In Turing's paper he proved that such a machine could be used to solve any computable function and that the decision problem was not solvable by this machine. The more general statement of this problem was named the *Halting Problem*. This was a very important result in the field of theoretical Computer Science.

In 1939 John Atanasoff, at Iowa State University, designed what is arguably the first computer, the ABC or Atanasoff-Berry Computer [28]. Clifford Berry was one of his graduate students. The computer had no central processing unit, but it did perform logical and other mathematical operations. Eckert and Mauchly, at the University of

Pennsylvania, were interested in building a computer during the second world war. They were funded by the Department of Defense to build a machine to calculate trajectory tables for launching shells from ships. The computer, called ENIAC for Electronic Numerical Integrator and Computer, was unveiled in 1946, just after the war had ended. ENIAC was difficult to program since the program was *written* by plugging cables into a switch, similar to an old telephone switchboard.

Around that same time a new computer, called EDVAC, was being designed. In 1945 John von Neumann proposed storing the computer programs on EDVAC in memory along with the program data [26]. Alan Turing closely followed John von Neumann's paper by publishing a paper of his own in 1946 describing a more complete design for stored-program computers [24]. To this day the computers we build and use are stored-program computers. The architecture is called the von Neumann architecture because of John von Neumann's and Alan Turing's contributions. While Turing didn't get the architecture named after him, he is famous in Computer Science for other reasons like the Turing machine and the Halting problem.

In the early days of Computer Science, many programmers were interested in writing tools that made it easier to program computers. Much of the programming was based on the concept of a stored-program computer and many early programming languages were extensions of this model of computation. In the stored-program model the program and data are stored in memory. The program manipulates data based on some input. It then produces output.

Around 1958, Algol was created and the second revision of this language, called Algol 60, was the first modern, structured, imperative programming language. While the language was designed by a committee, a large part of the success of the project was due to the contributions of John Backus pictured in Fig. 1.2. He described the structure of the Algol language using a mathematical notation that would later be called Backus-Naur Format or BNF. Very little has changed with the underlying computer architecture over the years. Of course, there have been many changes in the size, speed, and cost of computers! In addition, the languages we use have become

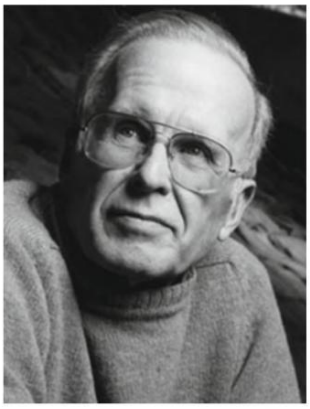

Fig. 1.2 John Backus [3]

even more structured over the years. But, the principles that Algol 60 introduced are still in use today.

Recalling that most early computer scientists were mathematicians, it shouldn't be too surprising to learn that there were others that approached the problem of programming differently. Much of the initial interest in computers was spurred by the invention of the stored-program computer and many of the early languages reflected this excitement. The imperative style was closely tied to the architecture of a stored program computer. Data was read from an input device and the program acted on that data by updating memory as the program executed. There was another approach developing at the same time. Back in 1936, Alonzo Church, a U.S. mathematician who lived from 1903–1995, was also interested in the decision problem proposed by David Hilbert. To try to solve the problem he devised a language called the lambda calculus, usually written as the λ-calculus. Using his very simple language he was able to describe computation as symbol manipulation. Alan Turing was a doctoral student of Church and while they independently came up with two ways to prove that the decision problem was not solvable, they later proved their two models of computation, Turing machines and the λ-calculus, were equivalent. Their work eventually led to a very important result called the Church-Turing Thesis. Informally, the thesis states that all computable problems can be solved by a Turing Machine or the λ-calculus. The two models are equivalent in power.

Ideas from the λ-calculus led to the development of Lisp by John McCarthy, pictured in Fig. 1.3. The λ-calculus and Lisp were not designed based on the principle of the stored-program computer. In contrast to Algol 60, the focus of these languages was on functions and what could be computed using functions. Lisp was developed around 1958, the same time that Algol 60 was being developed.

Logic is important both in Computer Science and Mathematics. Logicians were also interested in solving problems in the early days of Computer Science. Many problems in logic are expressed in the languages of propositional or predicate logic.

Fig. 1.3 John McCarthy [14]

Of course, the development of logic goes all the way back to ancient Greece. Some logicians of the 20th century were interested in understanding natural language and they were looking for a way to use computers to solve at least some of the problems related to processing natural language statements. The desire to use computers in solving problems from logic led to the development of Prolog, a powerful programming language based on predicate logic.

Practice 1.1 Find the answers to the following questions.

1. What are the origins of the three major computational models that early computer scientists developed?
2. Who were Alan Turing and Alonzo Church and what were some of their contributions to Computer Science?
3. What idea did both John von Neumann and Alan Turing contribute to?
4. What notation did John Backus develop and what was one of its first uses?
5. What year did Alan Turing first propose the Turing machine and why?
6. What year did Alonzo Church first propose the λ-calculus and why?
7. Why are Eckert and Mauchly famous?
8. Why are the history of Mathematics and Computer Science so closely tied together?

You can check your answer(s) in Section 1.8.1.

1.2 Models of Computation

While there is some controversy about who originally came up with the concept of a stored program computer, John von Neumann is generally given credit for the idea of storing a program as a string of 0's and 1's in memory along with the data used by the program. Von Neumann's architecture had very little structure to it. It consisted of several registers and memory. The Program Counter (PC) register kept track of the next instruction to execute. There were other registers that could hold a value or point to other values stored in memory. This model of computation was useful when programs were small. However, without additional structure, anything but a small program would quickly get hard to manage. This was what was driving the need for better and newer programming languages. Programmers needed tools that let them organize their code so they could focus on problem solving instead of the details of the hardware.

1.2.1 The Imperative Model

As programs grew in size it was necessary to provide the means for applying additional structure to them. In the early days a function was often called a sub-routine. Functions, procedures, and sub-routines were introduced by languages like Algol 60 so that programs could be decomposed into simpler sub-programs, providing a way for programmers to organize their code. Terms like top-down or bottom-up design were used to describe this process of subdividing programs into simpler sub-programs. This process of subdividing programs was often called *structured programming*, referring to the decomposition of programs into simpler, more manageable pieces. Most modern languages provide the means to decompose problems into simpler subproblems. We often refer to this structured approach as the imperative model of programming.

To implement functions and function calls in the von Neumann architecture, it was necessary to apply some organization to the data of a program. In the imperative model, memory is divided into regions which hold the program and the data. The data area is further subdivided into the static or global data area, the run-time stack, and the heap as pictured in Fig. 1.4.

In the late 1970s and 1980s people like Niklaus Wirth and Bjarne Stroustrup were interested in developing languages that supported an additional level of organization called Object-Oriented Programming, often abbreviated OOP. Object-oriented programming still uses the imperative model of programming. The addition of a means to describe classes of objects gives programmers another way of organizing their code into functions that are related to a particular type of object.

When a program executes it uses a special register called the stack pointer (SP) to point to the top activation record on the run-time stack. The run-time stack contains one activation record for each function or procedure invocation that is currently unfinished in the program. The top activation record corresponds to the current

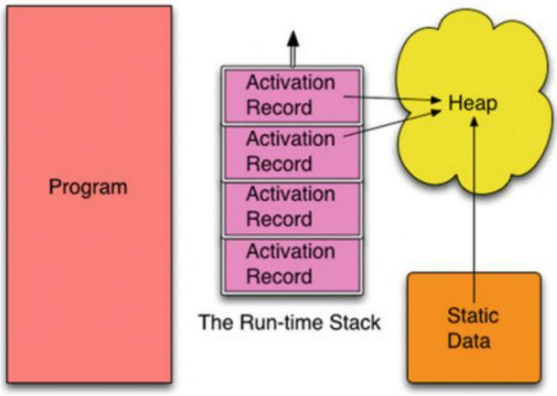

Fig. 1.4 Imperative model

function invocation. When a function call is made an activation record is pushed onto the run-time stack. When a function returns, the activation record is popped by decrementing the stack pointer to point to the previous activation record.

An activation record contains information about its associated function. The local variables of the function are stored there. The program counter's value before the function call was made is stored there. This is often called the return address. Other state information may also be stored there depending on the language and the details of the underlying von Neumann architecture. For instance, parameters passed to the function may also be stored there.

Static or global data refers to data and functions that are accessible globally in the program. Global data and functions are visible throughout the program. Where global data is stored depends on the implementation of the compiler or interpreter. It might be part of the program code in some instances. In any case, this area is where constants, global variables, and possibly built-in globally accessible functions are stored.

The heap is an area for dynamic memory allocation. The word dynamic means that it happens while the program is running. All data that is created at run-time is located in the heap. The data in the heap has no names associated with the values stored there. Instead, named variables called pointers or references point to the data in the heap. In addition, data in the heap may contain pointers that point to other data, which is also usually in the heap.

Like the original von Neumann architecture, the primary goal of the imperative model is to get data as input, transform it via updates to memory, and then produce output based on this imperatively changed data. The imperative model of computation parallels the underlying von Neumann architecture and is used by many modern languages. Some variation of this model is used by languages like Algol 60, C++, C, Java, VB.net, Python, and many other languages.

Practice 1.2 Find the answers to the following questions.

1. What are the three divisions of data memory called?
2. When does an item in the heap get created?
3. What goes in an activation record?
4. When is an activation record created?
5. When is an activation record deleted?
6. What is the primary goal of imperative, object-oriented programming?

You can check your answer(s) in Section 1.8.2.

1.2.2 The Functional Model

In the functional model the goal of a program is slightly different. This slight change in the way the model works has a big influence on how you program. In the functional model of computation the focus is on function calls. Functions and parameter passing are the primary means of accomplishing data transformation.

Data is generally not changed in the functional model. Instead, new values are constructed from old values. A pure functional model wouldn't allow any updates to existing values. However, most functional languages allow limited updates to memory in the imperative style.

The conceptual view presented in Fig. 1.4 is similar to the view in the functional world. However, the difference between program and data is eliminated. A function is data like any other data element. Integers and functions are both first-class citizens of the functional world.

The static data area is still present, but takes on a minor role in the functional model. The run-time stack becomes more important because most work is accomplished by calling functions. Functional languages are much more careful about how they allow programmers to access the heap and as a result, you really aren't aware of the heap when programming in a functional language. Data is certainly dynamically allocated, but once data is created on the heap it is not modified in a pure functional model. Impure models might allow some modification of storage but this is the influence of imperative languages creeping into the functional model as a way to deal with performance issues. The result is that you spend less time thinking about the underlying architecture when programming in a functional language.

Lisp, Scheme, Scala, Clojure, Elixir, Haskell, Caml, and Standard ML, which is covered in this text, are all examples of functional languages. Functional languages may be pure, which means they do not support variable updates like the imperative model. Scheme is a pure functional language. Most functional languages are not pure. Standard ML and Lisp are examples of impure functional languages. Scala is a recent functional language that also supports object-oriented programming.

Practice 1.3 Answer the following questions.

1. What are some examples of functional languages?
2. What is the primary difference between the functional and imperative models?
3. Immutable data is data that cannot be changed once created. The presence of immutable data simplifies the conceptual model of programming. Does the imperative or functional model emphasize immutable data?

You can check your answer(s) in Section 1.8.3.

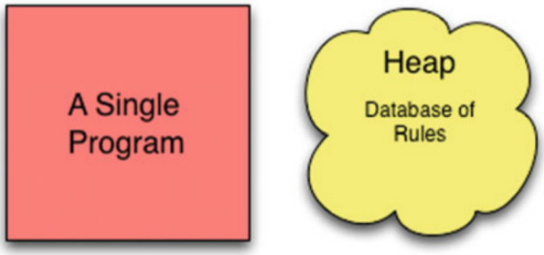

Fig. 1.5 Logic model of computation

1.2.3 The Logic Model

The logic model of computation, pictured in Fig. 1.5, is quite different from either the imperative or functional model. In the logic model the programmer doesn't actually write a program at all. Instead, the programmer provides a database of facts or rules. From this database, a single program tries to answer questions with a yes or no answer. In the case of Prolog, the program acts in a predictable manner allowing the programmer to provide the facts in an order that determines how the program will work. The actual implementation of this conceptual view is accomplished by a virtual machine, a technique for implementing languages that is covered later in this text.

There is still the concept of a heap in Prolog. One can assert new rules and retract rules as the program executes. To dynamically add rules or retract them there must be an underlying heap. In fact, the run-time stack is there too. However, the run-time stack and heap are so hidden in this view of the world that it is debatable whether they should appear in the conceptual model at all.

Practice 1.4 Answer these questions on what you just read.

1. How many programs can you write in a logic programming language like Prolog?
2. What does the programmer do when writing in Prolog?

You can check your answer(s) in Section 1.8.4.

1.3 The Origins of a Few Programming Languages

This book explores language implementation using several small languages and exercises that illustrate each of these models of computation. In addition, exercises within the text will require implementation in four different languages: assembly

language, Java (or alternatively C++), Standard ML, and Prolog. But where did these languages come from and why are we interested in learning how to use them?

1.3.1 A Brief History of C and C++

The Unix operating system was conceived of, designed, and written around 1972. Ken Thompson was working on the design of Unix with Dennis Ritchie. It was their project that encouraged Ritchie to create the C language. C was more structured than the assembly language most operating systems were written in at the time and it was portable and could be compiled to efficient machine code. Thompson and Ritchie wanted an operating system that was portable, small, and well organized.

While C was efficient, there were other languages that had either been developed or were being developed that encouraged a more structured approach to programming. For several years there had been ideas floating around about how to write code in object-oriented form. Simula, created by Ole-Johan Dahl and Kristen Nygaard around 1967, was an early example of a language that supported Object-Oriented design. Modula-2, created by Niklaus Wirth around 1978, was also taking advantage of these ideas. Smalltalk, an interpreted language, was object-oriented and was also developed in the mid 1970s and released in 1980.

In 1980 Bjarne Stroustrup, pictured in Fig. 1.6, began working on the design of C++ while working at Bell Labs. He envisioned C++ as a language that would allow C programmers to keep their old code while new code could be written using these Object-Oriented concepts. In 1983 he named this new language C++, as in the next increment of C, and with much anticipation, in 1985 the language was released. About the same time Dr. Stroustrup released a book called *The C++ Programming Language* [19], which described the language. The language is still evolving. For instance, templates, an important part of C++ were first described by Stroustrup in 1988 [17] and it wasn't until 1998 that it was standardized as ANSI C++. Today an ANSI committee oversees the continued development of C++. The latest C++ standard was released in 2014 as of this writing. The previous standard was released

Fig. 1.6 Bjarne Stroustrup [18]

in 2011. C++ is a mature language, but is still growing and evolving. The 2017 standard is currently in the works with comments presently being solicited by the standards committee.

1.3.2 A Brief History of Java

C++ is a very powerful language, but also demands that programmers be very careful when writing code. The biggest problem with C++ programs are memory leaks. When objects are created on the heap in C++, they remain on the heap until they are freed. If a programmer forgets to free an object, then that space cannot be re-used while the program is running. That space is gone until the program is stopped, even if no code has a pointer to that object anymore. This is a memory leak. And, for long-running C++ programs it is the number one problem. Destructors are a feature of C++ that help programmers prevent memory leaks. Depending on the structure of a class in your program, it may need a destructor to take care of cleaning up instances of itself (i.e. objects of the class) when they are freed.

C++ programs can create objects in the run-time stack, on the heap, or within other objects. This is another powerful feature of C++. But, with this power over the creation of objects comes more responsibility for the programmer. This control over object creation leads to the need for extra code to decide how copies of objects are made. In C++ every class may contain a copy constructor so the programmer can control how copies of objects are made.

In 1991 a team called the *Green Team*, was working for a company named *Sun Microsystems*. This group of software engineers wanted to design a programming language and run-time system that could be used in the next generation of personal devices. The group was led by a man named James Gosling. To support their vision, they designed the Java Virtual Machine (i.e. JVM), a program that would interpret byte code files. The JVM was designed as the run-time system for the Java programming language. Java programs, when compiled, are translated into bytecode files that run on the JVM.

The year 1995 brought the birth of the world wide web and with it one of the first web browsers, Netscape Navigator, which later became Mozilla Firefox. In 1995 it was announced that Netscape would include Java technology within the browser. This led to some of the initial interest in the language, but the language has grown way beyond web browsers. In fact, Java is not really a web browser technology anymore. It is used in many web backends, where Java programs wait for connections from web browsers, but it doesn't run programs within web browsers much these days. Another language, Javascript, is now the primary language of web browsers. Javascript is similar to Java in name, but not its technology. Javascript was licensed as a name from Sun Microsystems in its early days because of the popularity of Java [22].

The original intention of Java was to serve as a means for running software for personal devices. Java has become very important in that respect. It now is the basis for the Android operating system that runs on many phones and other personal devices like tablets. So, in a sense, the original goal of the Green Team has been realized, just fifteen or so years later.

When the original Green Team was designing Java they wanted to take the best of C++ while leaving behind some of its complexity. In Java objects can only be created in one location, on the heap. Sticking to one and only one memory model for objects simplifies many aspects of Java. Objects are never copied by the language. So, copy constructors are unnecessary in Java. When an object is passed to a function, a reference to an object is passed without making a copy of the object. When one object wants to contain another object, it keeps a reference to that object. Java objects are never stored inside other objects. Simplifying the memory model for objects means that in Java programs we don't have to worry about copying objects.

Objects can still be copied in Java, but making copies of objects is the responsibility of the programmer. The Java language does not make copies. Programmers make copies by calling a special method called *clone*.

Java also includes garbage collection. This means that the Java Virtual Machine takes care of deciding when the space that an object resides in can be reclaimed. It can be reclaimed when no other objects or code have a reference to it anymore. This means that programmers don't have to write destructors. The JVM manages this for them.

So, while C++ and Java share a lot of syntax, there are many differences as well. Java has a simpler memory model. Garbage collection removes the fear of memory leaks in Java programs. The Java Virtual Machine also provides other advantages to writing Java programs. This does not make C++ a bad language by any means. It's just that Java and C++ have different goals. The JVM and Java manage a lot of the complexity of writing object-oriented programs, freeing the programmer from these duties. C++ on the other hand, gives you the power to manage all the details of a program, right down to the hardware interface. Neither is better than the other, they just serve different purposes while the two languages also share a lot of the same syntax.

Fig. 1.7 Guido van Rossum [25]

1.3.3 A Brief History of Python

Python was designed and implemented by Guido van Rossum, pictured in Fig. 1.7. He started Python as a hobby project during the winter months of 1989. A more complete history of this language is available on the web at http://python-history.blogspot.com. Python is another object-oriented language like C++ and Java. Unlike C++, Python is an interpreted language. Mr. van Rossum designed Python's interpreter as a virtual machine, like the Java Virtual Machine (i.e. JVM). But Python's virtual machine is not accessible separately, unlike the JVM. The Python virtual machine is an internal implementation detail of the Python interpreter. Virtual machines have been around for some time including an operating system for IBM mainframe computers, called VM. Using a virtual machine when implementing a programming language can make the language and its programs more portable across platforms. Python runs on many different platforms like Apple's Mac OS X, Linux, and Microsoft Windows. Virtual machines can also provide services that make language implementation easier.

Programmers world-wide have embraced Python and have developed many libraries for Python and written many programs. Python has gained popularity among developers because of its portability and the ability to provide libraries to others. Guido van Rossum states in his history of Python, "A large complex system should have multiple levels of extensibility. This maximizes the opportunities for users, sophisticated or not, to help themselves." Extensibility refers to the ability to define libraries of classes to solve problems from many different application areas. Python is used in internet programming, server scripting, computer graphics, visualization, Mathematics, Computer Science education, and many, many other application areas.

Mr. van Rossum continues, saying "In many ways, the design philosophy I used when creating Python is probably one of the main reasons for its ultimate success. Rather than striving for perfection, early adopters found that Python worked "well enough" for their purposes. As the user-base grew, suggestions for improvement were gradually incorporated into the language." Growing the user-base has been key to the success of Python. As the number of programmers that know Python has increased so has interest in improving the language. Python now has two major versions, Python 2 and Python 3. Python 3 is not backward compatible with Python 2. This break in compatibility gave the Python developers an opportunity to make improvements in the language. Chapters 3 and 4 cover some of the implementation details of the Python programming language.

1.3.4 A Brief History of Standard ML

Standard ML originated in 1986, but was the follow-on of ML which originated in 1973 [16]. Like many other languages, ML was implemented for a specific purpose. The ML stands for Meta Language. Meta means above or about. So a metalanguage is a language about language. In other words, a language used to describe a language. ML was originally designed for a theorem proving system. The theorem prover was called LCF, which stands for Logic for Computable Functions. The LCF theorem

Fig. 1.8 Robin Milner [15]

prover was developed to check proofs constructed in a particular type of logic first proposed by Dana Scott in 1969 and now called Scott Logic. Robin Milner, pictured in Fig. 1.8, was the principal designer of the LCF system. Milner designed the first version of LCF while at Stanford University. In 1973, Milner moved to Edinburgh University and hired Lockwood Morris and Malcolm Newey, followed by Michael Gordon and Christopher Wadsworth, as research associates to help him build a new and better version called Edinburgh LCF [9].

For the Edinburgh version of LCF, Dr. Milner and his associates created the ML programming language to allow proof commands in the new LCF system to be extended and customized. ML was just one part of the LCF system. However, it quickly became clear that ML could be useful as a general purpose programming language. In 1990 Milner, together with Mads Tofte and Robert Harper, published the first complete formal definition of the language; joined by David MacQueen, they revised this standard to produce the Standard ML that exists today [16].

ML was influenced by Lisp, Algol, and the Pascal programming languages. In fact, ML was originally implemented in Lisp. There are now two main versions of ML: Moscow ML and Standard ML. Today, ML's main use is in academia in the research of programming languages. But, it has been used successfully in several other types of applications including the implementation of the TCP/IP protocol stack [4] and a web server as part of the Fox Project. A goal of the Fox Project was the development of system software using advanced programming languages [10].

ML is a very good language to use in learning to implement other languages. It includes tools for automatically generating parts of a language implementation including components called a scanner and a parser which are introduced in Chap. 6. These tools, along with the polymorphic strong type checking provided by Standard ML, make implementing a compiler or interpreter a much easier task. Much of the work of implementing a program in Standard ML is spent in making sure all the types in the program are correct. This strong type checking often means that once a

program is properly typed it will run the first time. This is quite a statement to make, but nonetheless it is often true.

Important Standard ML features include:

- ML is higher-order supporting functions as first-class values. This means functions may be passed as parameters to functions and returned as values from functions.
- Strong type checking (discussed later in this chapter) means it is pretty infrequent that you need to debug your code. What a great thing!
- Pattern-matching is used in the specification of functions in ML. Pattern-matching is convenient for writing recursive functions.
- The exception handling system implemented by Standard ML has been proven type safe, meaning that the type system encompasses all possible paths of execution in an ML program.

1.3.5 A Brief History of Prolog

Prolog was developed in 1972 by Alain Colmerauer, pictured in Fig. 1.9, with Philippe Roussel. Colmerauer and Roussel and their research group had been working on natural language processing for the French language and were studying logic and automated theorem proving [7] to answer simple questions in French. Their research led them to invite Robert Kowalski, pictured in Fig. 1.10, who was working in the area of logic programming and had devised an algorithm called SL-Resolution, to work with them in the summer of 1971 [11,29]. Colmerauer and Kowalski, while working together in 1971, discovered a way formal grammars could be written as clauses in predicate logic. Colmerauer soon devised a way that logic predicates could be used to express grammars that would allow automated theorem provers to parse natural language sentences efficiently. This is covered in some detail in Chap. 7.

Fig. 1.9 Alain Colmerauer [6]

Fig. 1.10 Robert Kowalski [12]

In the summer of 1972, Kowalski and Colmerauer worked together again and Kowalski was able to describe the procedural interpretation of what are known as Horn Clauses. Much of the debate at the time revolved around whether logic programming should focus on procedural representations or declarative representations. The work of Kowalski showed how logic programs could have a dual meaning, both procedural and declarative.

Colmerauer and Roussel used this idea of logic programs being both declarative and procedural to devise Prolog in the summer and fall of 1972. The first large Prolog program, which implemented a question and answering system in the French language, was written in 1972 as well.

Later, the Prolog language interpreter was rewritten at Edinburgh to compile programs into DEC-10 machine code. This led to an abstract intermediate form that is now known as the Warren Abstract Machine or WAM. WAM is a low-level intermediate representation that is well-suited for representing Prolog programs. The WAM virtual machine can be (and has been) implemented on a wide variety of hardware. This means that Prolog implementations exist for most computing platforms.

Practice 1.5 Answer the following questions.

1. Who invented C++? C? Standard ML? Prolog? Python? Java?
2. What do Standard ML and Prolog's histories have in common?
3. What do Prolog and Python have in common?
4. What language or languages is Standard ML based on?

You can check your answer(s) in Section 1.8.5.

1.4 Language Implementation

There are three ways that languages can be implemented.

- A language can be interpreted.
- A language can be compiled to a machine language.
- A language can be implemented by some combination of the first two methods.

Computers are only capable of executing machine language. Machine language is the language of the Central Processing Unit (CPU) and is very simple. For instance, typical instructions are *fetch this value into the CPU*, *store this value into memory from the CPU*, *add these two values together*, and *compare these two values and if they are equal, jump here next*. The goal of any programming language implementation is to translate a source program into this simpler machine language so it can be executed by the CPU. The overall process is pictured in Fig. 1.11.

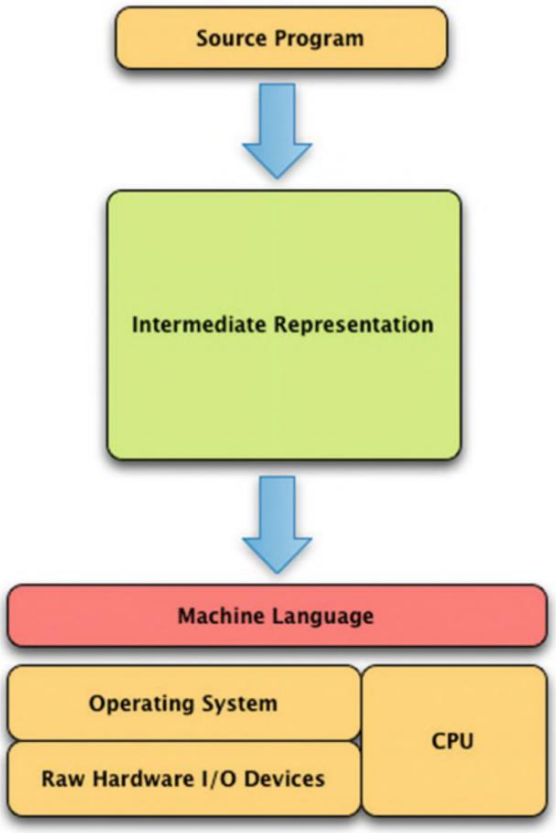

Fig. 1.11 Language implementation

All language implementations translate a source program to some intermediate representation before translating the intermediate representation to machine language. Exactly how these two translations are packaged varies significantly from one programming language to the next, but luckily most language implementations follow one of a few methodologies. The following sections will present some case studies of different languages so you can see how this translation is accomplished and packaged.

1.4.1 Compilation

The most direct method of translating a program to machine language is called compilation. The process is shown in Fig. 1.12. A compiler is a program that internally is composed of several parts. The *parser* reads a source program and translates it into an intermediate form called an abstract syntax tree (*AST*). An *AST* is a tree-like data structure that internally represents the source program. We'll read about abstract syntax trees in later chapters. The *code generator* then traverses the AST and produces another intermediate form called an assembly language program. This program is not machine language, but it is much closer. Finally, an *assembler* and *linker* translate an assembly language program to machine language making the program ready to execute.

This whole process is encapsulated by a tool called a *compiler*. In most instances, the assembler and linker are separate from the compiler, but normally the compiler runs the assembler and linker automatically when a program is compiled so as programmers we tend to think of a compiler compiling our programs and don't necessarily think about the assembly and link phases.

Programming in a compiled language is a three-step process.

- First, you write a source program.
- Then you compile the source program, producing an executable program.
- Then you run the executable program.

When you are done, you have a source program and an executable program that represent the same computation, one in the source language, the other in machine language. If you make further changes to the source program, the source program and the machine language program are not in sync. After making changes to the source program you must remember to recompile before running the executable program again.

Machine language is specific to a CPU architecture and operating system. Compiling a source program on Linux means it will run on most Linux machines with a similar CPU. However, you cannot take a Linux executable and put it on a Microsoft Windows machine and expect it to run, even if the two computers have the same CPU. The Linux and Windows operating systems each have their own format for executable machine language programs. In addition, compiled programs use operating system services for printing, reading input, and doing other Input/Output (I/O) operations. These services are invoked differently between operating systems. Lan-

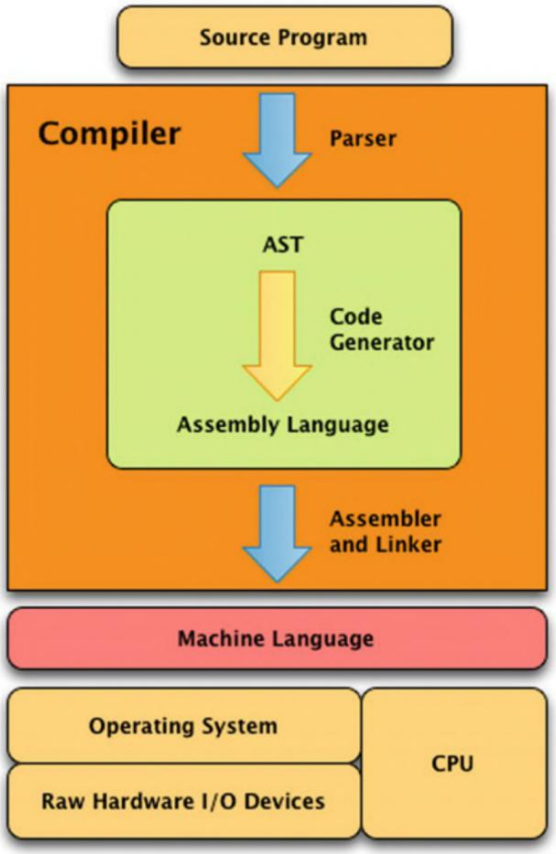

Fig. 1.12 The compilation process

guages like C++ hide these implementation details from you in the code generator, but the end result is that a program compiled for one operating system will not work on another operating system without being recompiled.

C, C++, Pascal, Fortran, COBOL and many others are typically compiled languages. On the Linux operating system the C compiler is called *gcc* and the C++ compiler is called *g++*. The *g* in both names reflects the fact that both compilers come out of the GNU project and the Free Software Foundation. Linux, gcc, and g++ are freely available to anyone who wants to download them. The best way to get these tools is to download a Linux distribution and install it on a computer. The *gcc* and *g++* compilers come standard with Linux.

There are implementations of C and C++ for many other platforms. The web site http://gcc.gnu.org contains links to source code and to prebuilt binaries for the g++ compiler. You can also download C++ compilers from Apple and Microsoft. For Mac OS X computers you can get C++ by downloading the XCode Developer Tools.

You can also install g++ and gcc for Mac OS X computers using a tool called *brew*. If you run Microsoft Windows you can install Visual C++ Express from Microsoft. It is free for educational use.

1.4.2 Interpretation

An interpreter is a program that is written in some other language and compiled into machine language. The interpreter itself is the machine language program. The interpreter itself is written to read source programs from the interpreted language and interpret them. For instance, Python is an interpreted language. The Python interpreter is written in C and is compiled for a particular platform like Linux, Mac OS X, or Microsoft Windows. To run a Python program, you must download and install the Python interpreter that goes with your operating system and CPU.

When you run an interpreted source program, as depicted in Fig. 1.13, you are actually running the interpreter. Your program is not running because your program is never translated to machine language. The interpreter is the machine language program that executes all the programs you write in the interpreted language. The source program you write controls the behavior of the interpreter program.

Programming in an interpreted language is a two step process.

- First you write a source program.
- Then you execute the source program by running the interpreter.

Each time your program is executed it is translated into an AST by a part of the interpreter called the parser. There may be an additional step that translates the AST to some lower-level representation, often called bytecode. In an interpreter, this lower-level representation is still internal to the interpreter program. Then a part of the interpreter, often called a virtual machine, executes the byte code instructions.

Not every interpreter translates the AST to bytecode. Sometimes the interpreter directly interprets the AST but it is often convenient to translate the source program's AST to some simpler representation before executing it.

Eliminating the compile step has a few implications.

- Since you have one less step in development you may be encouraged to run your code more frequently during development. This is a generally a good thing and can shorten the development cycle.
- Secondly, because you don't have an executable version of your code, you don't have to manage the two versions. You only have a source code program to keep track of.
- Finally, because the source code is not platform dependent, you can usually easily move your program between platforms. The interpreter insulates your program from platform dependencies.

Of course, source programs for compiled languages are generally platform independent too. But, they must be recompiled to move the executable program from one

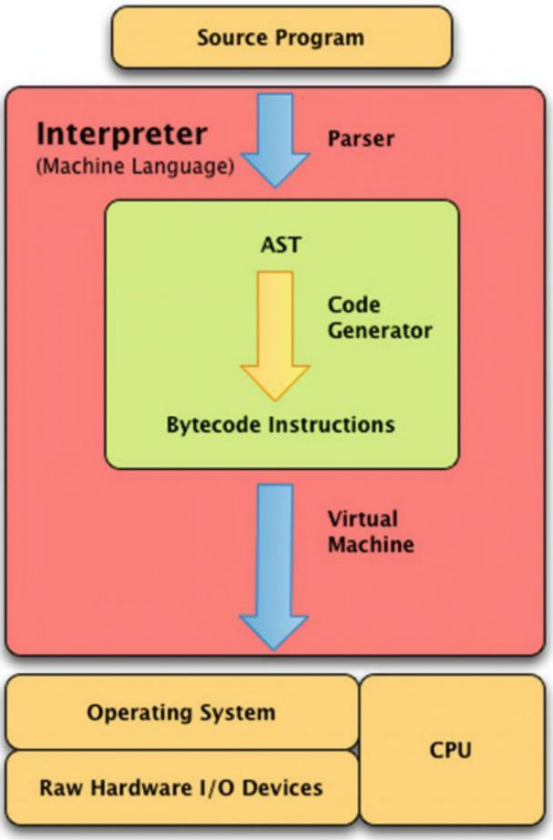

Fig. 1.13 The interpretation process

platform to another. The interpreter itself isn't platform independent. There must be a version of an interpreter for each platform/language combination. So there is a Python interpreter for Linux, another for Microsoft Windows, and yet another for Mac OS X. Thankfully, because the Python interpreter is written in C the same Python interpreter program can be compiled (with some small differences) for each platform.

There are many interpreted languages available including Python, Ruby, Standard ML, Unix scripting languages like Bash and Csh, Prolog, and Lisp. The portability of interpreted languages has made them very popular among programmers, especially when writing code that needs to run across multiple platforms.

One huge problem that has driven research into interpreted languages is that of heap memory management. Recall that the heap is the place where memory is dynamically allocated. As mentioned earlier in the chapter, C and C++ programs are notorious for having memory leaks. Every time a C++ programmer reserves some space on the heap he/she must remember to free that space. If they don't free the

space when they are done with it the space will never be available again while the program continues to execute. The heap is a big space, but if a program runs long enough and continues to allocate and not free space, eventually the heap will fill up and the program will terminate abnormally. In addition, even if the program doesn't terminate abnormally, the performance of the system will degrade as more and more time is spent managing the large heap space.

Most, if not all, interpreted languages don't require programmers to free space on the heap. Instead, there is a special task or thread that runs periodically as part of the interpreter to check the heap for space that can be freed. This task is called the *garbage collector*. Programmers can allocate space on the heap but don't have to be worried about freeing that space. For a garbage collector to work correctly, space on the heap has to be allocated and accessed in the right way. Many interpreted languages are designed to insure that a garbage collector will work correctly.

The disadvantage of an interpreted language is in speed of execution. Interpreted programs typically run slower than compiled programs. In a compiled program, parsing and code generation happen once when the program is compiled. When running an interpreted program, parsing and code generation happen each time the program is executed. In addition, if an application has real-time dependencies then having the garbage collector running at more or less random intervals may not be desirable. As you'll read in the next section some steps have been taken to reduce the difference in execution time between compiled and interpreted languages.

1.4.3 Virtual Machines

The advantages of interpretation over compilation are pretty significant. It turns out that one of the biggest advantages is the portability of programs. It's nice to know when you invest the time in writing a program that it will run the same on Linux, Microsoft Windows, Mac OS X, or some other operating system. This portability issue has driven a lot of research into making interpreted programs run as fast as compiled languages.

As discussed earlier in this chapter, the concept of a virtual machine has been around quite a while. A virtual machine is a program that provides insulation from the actual hardware and operating system of a machine while supplying a consistent implementation of a set of low-level instructions, often called bytecode. Figure 1.14 shows how a virtual machine sits on top of the operating system/CPU to act as this insulator.

There is no one specification for bytecode instructions. They are specific to the virtual machine being defined. Python has a virtual machine buried within the interpreter. Prolog is another interpreter that uses a virtual machine as part of its implementation. Some languages, like Java have taken this idea a step further. Java has a virtual machine that executes bytecode instructions as does Python. The creators of Java separated the virtual machine from the compiler. Instead of storing the bytecode instructions internally as in an interpreter, the Java compiler, called *javac*, compiles a Java source code program to a bytecode file. This file is not machine language so it cannot be executed directly on the hardware. It is a Java bytecode file which

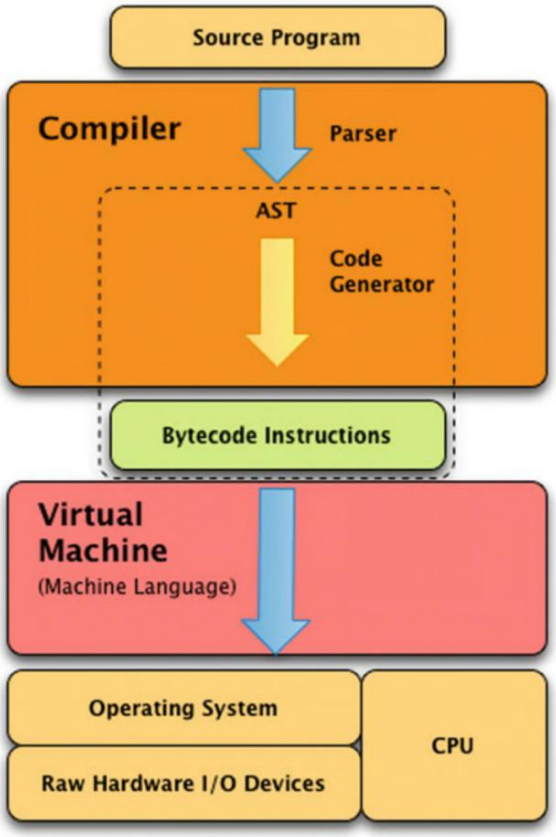

Fig. 1.14 Virtual machine implementation

is interpreted by the Java virtual machine, called *java* in the Java set of tools. Java bytecode files all end with a *.class* extension. You may have noticed these files at some point after compiling a Java program.

Programs written using a hybrid language like Java are compiled. However, the compiled bytecode program is interpreted. Source programs in the language are not interpreted directly. By adding this intermediate step the interpreter can be smaller and faster than traditional interpreters. Very little parsing needs to happen to read the program and executing the program is straightforward because each bytecode instruction usually has a simple implementation.

Languages that fall into this virtual machine category include Java, ML, Python, C#, Visual Basic .NET, JScript, and other .NET platform languages. You might notice that Standard ML and Python were included as examples of interpreted languages. Both ML and Python include interactive interpreters as well as the ability to compile and run low-level bytecode programs. Python bytecode files are named with a *.pyc* extension. Standard ML compiled files are named with a *-platform* as the last part of

the compiled file name. In the case of Python and Standard ML the virtual machine is not a separate program. Both interpreters are written to recognize a bytecode file and execute it just like a source program.

Java and the .NET programming environments do not include interactive interpreters. The only way to execute programs with these platforms is to compile the program and then run the compiled program using the virtual machine. Programs written for the .NET platform run under Microsoft Windows and in some cases Linux. Microsoft submitted some of the .NET specifications to the ISO to allow third party software companies to develop support for .NET on other platforms. In theory all .NET programs are portable like Java, but so far implementations of the .NET framework are not as generally available as Java. The Java platform has been implemented and released on all major platforms. In fact, in November 2006 Sun, the company that created Java, announced they were releasing the Java Virtual Machine and related software under the GNU Public License to encourage further development of the language and related tools. Since then the rights to Java have now been purchased by Oracle where it continues to be supported.

Java and .NET language implementations maintain backwards compatibility of their virtual machines. This means that a program compiled for an earlier version of Java or .NET will continue to run on newer implementations of the language's virtual machine. In contrast, Python's virtual machine is regarded as an internal design issue and does not maintain backwards compatibility. A *.pyc* file compiled for one version of Python will not run on a newer version of Python. This distinction makes Python more of an interpreted language, while Java and .NET languages are truly virtual machine implementations.

Maintaining backwards compatibility of the virtual machine means that programmers can distribute application for Java and .NET implementations without releasing their source code. .NET and Java applications can be distributed while maintaining privacy of the source code. Since intellectual property is an important asset of companies, the ability to distribute programs in binary form is important. The development of virtual machines made memory management and program portability much easier in languages like Java, Standard ML, and the various .NET languages while also providing a means for programmers to distribute programs in binary format so source code could be kept private.

1.5 Types and Type Checking

Every programming language defines operations that can be used to transform data. Data transformation is the fundamental operation that is performed by all programming languages. Some programming languages mutate data to new values. Other languages transform data by building new values from old values. However the transformation takes place, these data transformation operations are defined for certain *types* of data. Not every transformation operation makes sense for every type of value. For instance, addition is an operation that makes sense for numbers, but does

The next chapter provides the foundations for understanding how the syntax of a language is formally defined by a grammar. Then chapter three introduces a Python Virtual Machine implementation called JCoCo. JCoCo is an interpreter of Python bytecode instructions. Chapter three introduces assembly language programming using JCoCo, providing some insight into how programming languages are implemented.

Subsequent chapters in the book will again look at language implementation to better understand the languages you are learning, their strengths and weaknesses. While learning these languages you will also be implementing a compiler for a high level functional language called *Small* which is a robust subset of Standard ML. This will give you even more insight into language implementation and knowledge of how to use these languages to solve problems.

Finally, in the last two chapters of this text, you will learn about type checking and type inference using Prolog, a language that is well-suited to logic problems like type inference. Learning how to use Prolog and implement a type checker is a great way to cap off a text on programming languages and language implementation.

A great way to summarize the rest of this text is to see it moving from very prescriptive approaches to programming to very descriptive approaches to programming. The word *prescriptive* means that you dwell on details, thinking very carefully about the details of what you are writing. For instance, in a prescriptive approach you might ask yourself, how do you set things up to invoke a particular type of instruction? In contrast, *descriptive* programming relies on programmers describing relationships between things. Functional programming languages, to some extent, and logic programming languages employ this descriptive approach to programming. Read on to begin the journey from prescriptive to descriptive programming!

1.7 Review Questions

1. What are the three ways of thinking about programming, often called programming paradigms?
2. Name at least one language for each of the three methods of programming described in the previous question.
3. Name one person who had a great deal to do with the development of the imperative programming model. Name another who contributed to the functional model. Finally, name a person who was responsible for the development of the logic model of programming.
4. What are the primary characteristics of each of the imperative, functional, and logic models?
5. Who are recognized as the founders of each of the languages this text covers: Java, C++, Python, Standard ML, and Prolog?
6. Name a language, other than Python, C++, or Java, that is imperative object-oriented in nature.
7. Name a language besides Standard ML, that is a functional programming language.

8. What other logic programming languages are there other than Prolog? You might have to get creative on this one.
9. Why is compiling a program preferred over interpreting a program?
10. Why is interpreting a program preferred over compiling a program?
11. What benefits do virtual machine languages have over interpreted languages?
12. What is a bytecode program? Name two languages that use bytecode in their implementation.
13. Why are types important in a programming language?
14. What does it mean for a programming language to be dynamically typed?
15. What does it mean for a programming language to be statically typed?

1.8 Solutions to Practice Problems

These are solutions to the practice problems. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

1.8.1 Solution to Practice Problem 1.1

1. The origins of the three models are the Turing Machine, the λ-calculus, and propositional and predicate logic.
2. Alan Turing as a PhD student of Alonzo Church. Alan Turing developed the Turing Machine and Alonzo Church developed the λ-calculus to answer prove there were somethings that are not computable. They later proved the two approaches were equivalent in their power to express computation.
3. Both von Neumann and Turing contributed to the idea of a stored-program computer.
4. Backus developed BNF notation which was used in the development of Algol 60.
5. 1936 was a big year for Computer Science.
6. So was 1946. That was the year ENIAC was unveiled. Eckert and Mauchly designed and built ENIAC.
7. The problems in Mathematics were growing complex enough that many mathematicians were developing models and languages for expressing their algorithms. This was one of the driving factors in the development of computers and Computer Science as a discipline.

1.8.2 Solution to Practice Problem 1.2

1. The run-time stack, global memory, and the heap are the three divisions of data memory.
2. Data on the heap is created at run-time.

3. An activation record holds information like local variables, the program counter, the stack pointer, and other state information necessary for a function invocation.
4. An activation record is created each time a function is called.
5. An activation record is deleted when a function returns.
6. The primary goal of imperative, object-oriented programming is to update memory by updating variables and/or objects as the program executes. The primary operation is memory updates.

1.8.3 Solution to Practice Problem 1.3

1. Functional languages include Standard ML, Lisp, Haskell, and Scheme.
2. In the imperative model the primary operation revolves around updating memory (the assignment statement). In the functional model the primary operation is function application.
3. The functional model emphasizes immutable data. However, some imperative languages have some immutable data as well. For instance, Java strings are immutable.

1.8.4 Solution to Practice Problem 1.4

1. You never write a program in Prolog. You write a database of rules in Prolog that tell the single Prolog program (depth first search) how to proceed.
2. The programmer provides a database of facts and predicates that tell Prolog about a problem. In Prolog the programmer describes the problem instead of programming the solution.

1.8.5 Solution to Practice Problem 1.5

1. C++ was invented by Bjourne Stroustrup. C was created by Dennis Ritchie. Standard ML was primarily designed by Robin Milner. Prolog was designed by Alain Colmerauer and Philippe Roussel with the assistance of Robert Kowalski. Python was created by Guido van Rossum. Java was the work of the Green team and James Gosling.
2. Standard ML and Prolog were both designed as languages for automated theorem proving first. Then they became general purpose programming languages later.
3. Both Python and Prolog run on virtual machine implementations. Python's virtual machine is internal to the interpreter. Prolog's virtual machine is called WAM (Warren Abstract Machine).
4. Standard ML is influenced by Lisp, Pascal, and Algol.

Syntax

2

Once you've learned to program in one language, learning a similar programming language isn't all that hard. But, understanding just how to write in the new language takes looking at examples or reading documentation to learn its details. In other words, you need to know the mechanics of putting a program together in the new language. Are the semicolons in the right places? Do you use *begin...end* or do you use curly braces (i.e. { and })? Learning how a program is put together is called learning the syntax of the language. Syntax refers to the words and symbols of a language and how to write the symbols down in some meaningful order.

Semantics is the word that is used when deriving meaning from what is written. The semantics of a program refers to what the program will do when it is executed. Informally it is much easier to say what a program does than to describe the syntactic structure of the program. However, syntax is a lot easier to formally describe than semantics. In either case, if you are learning a new language, you need to learn something about both the syntax and semantics of the language.

2.1 Terminology

Once again, the *syntax* of a programming language determines the well-formed or grammatically correct programs of the language. *Semantics* describes how or whether such programs will execute.

- *Syntax* is how programs look
- *Semantics* is how programs work

Many questions we might like to ask about a program either relate to the syntax of the language or to its semantics. It is not always clear which questions pertain to

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_2

syntax and which pertain to semantics. Some questions may concern semantic issues that can be determined statically, meaning before the program is run. Other semantic issues may be dynamic issues, meaning they can only be determined at run-time. The difference between static semantic issues and syntactic issues is sometimes a difficult distinction to make.

The code

```
a=b+c;
```

is correct syntax in many languages. But is it a correct C++ statement?

1. Do *b* and *c* have values?
2. Have *b* and *c* been declared as a type that allows the + operation? Or, do the values of *b* and *c* support the + operation?
3. Is *a* assignment compatible with the result of the expression $b + c$?
4. Does the assignment statement have the proper form?

There are lots of questions that need to be answered about this assignment statement. Some questions could be answered sooner than others. When a C++ program is compiled it is translated from C++ to machine language as described in the previous chapter. Questions 2 and 3 are issues that can be answered when the C++ program is compiled. However, the answer to the first question might not be known until the C++ program executes in some cases. The answers to questions 2 and 3 can be answered at *compile-time* and are called *static* semantic issues. The answer to question 1 is a *dynamic* issue and is probably not determinable until run-time. In some circumstances, the answer to question 1 might also be a static semantic issue. Question 4 is definitely a syntactic issue.

Unlike the dynamic semantic issues, the correct syntax of a program is statically determinable. Said another way, determining a syntactically valid program can be accomplished without running the program. The syntax of a programming language is specified by a grammar. But before discussing grammars, the parts of a grammar must be defined. A *terminal* or *token* is a symbol in the language.

- C++, Java, and Python terminals: *while*, *for*, *(*, *;*, *5*, *b*
- Type names like *int* and *string*

Keywords, types, operators, numbers, identifiers, etc. are all tokens or terminals in a language.

A *syntactic category* or *nonterminal* is a set of phrases, or strings of tokens, that will be defined in terms of symbols in the language (terminal and nonterminal symbols).

- C++, Java, or Python nonterminals: <statement>, <expression>, <if-statement>, etc.
- Syntactic categories define parts of a program like statements, expressions, declarations, and so on.

2.3.1 The Infix Expression Grammar

A context-free grammar for infix expressions can be specified as $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, T, F\}$
$\mathcal{T} = \{identifier, number, +, -, *, /, (,)\}$
$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid identifier \mid number$$

2.4 Derivations

A *sentence* of a grammar is a string of tokens from the grammar. A sentence belongs to the language of a grammar if it can be derived from the grammar. This process is called constructing a derivation. A *derivation* is a sequence of sentential forms that starts with the start symbol of the grammar and ends with the sentence you are trying to derive. A *sentential form* is a string of terminals and nonterminals from the grammar. In each step in the derivation, one nonterminal of a sentential form, call it A, is replaced by a string of terminals and nonterminals, β, where $A \rightarrow \beta$ is a production in the grammar. For a grammar, G, the language of G is the set of sentences that can be derived from G and is usually written as *L(G)*.

2.4.1 A Derivation

Here we prove that the expression $(5 * x) + y$ is a member of the language defined by the grammar given in Sect. 2.3.1 by constructing a derivation for it. The derivation begins with the start symbol of the grammar and ends with the sentence.

$$\underline{E} \Rightarrow \underline{E} + T \Rightarrow \underline{T} + T \Rightarrow \underline{F} + T \Rightarrow (\underline{E}) + T \Rightarrow (\underline{T}) + T \Rightarrow (\underline{T} * F) + T$$
$$\Rightarrow (\underline{F} * F) + T \Rightarrow (5 * \underline{F}) + T \Rightarrow (5 * x) + \underline{T} \Rightarrow (5 * x) + \underline{F} \Rightarrow (5 * x) + y$$

Each step is a sentential form. The underlined nonterminal in each sentential form is replaced by the right hand side of a production for that nonterminal. The derivation proceeds from the start symbol, E, to the sentence $(5 * x) + y$. This proves that $(5 * x) + y$ is in the language *L(G)* as G is defined in Sect. 2.3.1.

Practice 2.1 Construct a derivation for the infix expression $4 + (a - b) * x$.
You can check your answer(s) in Section 2.17.1.

2.4.2 Types of Derivations

A sentence of a grammar is *valid* if there exists at least one derivation for it using the grammar. There are typically many different derivations for a particular sentence of a grammar. However, there are two derivations that are of some interest to us in understanding programming languages.

- Left-most derivation - Always replace the left-most nonterminal when going from one sentential form to the next in a derivation.
- Right-most derivation - Always replace the right-most nonterminal when going from one sentential form to the next in a derivation.

The derivation of the sentence $(5 * x) + y$ in Sect. 2.4.1 is a left-most derivation. A right-most derivation for the same sentence is:

$$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + y \Rightarrow T + y \Rightarrow F + y \Rightarrow (E) + y \Rightarrow (T) + y$$
$$\Rightarrow (T * F) + y \Rightarrow (T * x) + y \Rightarrow (F * x) + y \Rightarrow (5 * x) + y$$

Practice 2.2 Construct a right-most derivation for the expression $x * y + z$.
You can check your answer(s) in Section 2.17.2.

2.4.3 Prefix Expressions

Infix expressions are expressions where the operator appears between the operands. Another type of expression is called a prefix expression. In prefix expressions the operator appears before the operands. The infix expression $4 + (a - b) * x$ would be written $+4 * -abx$ as a prefix expression. Prefix expressions are in some sense simpler than infix expressions because we don't have to worry about the precedence of operators. The operator precedence is determined by the order of operations in the expression. Because of this, parentheses are not needed in prefix expressions.

2.4.4 The Prefix Expression Grammar

A context-free grammar for prefix expressions can be specified as $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E\}$
$\mathcal{T} = \{identifier, number, +, -, *, /\}$
$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow + \; E \; E \mid - \; E \; E \mid * \; E \; E \mid / \; E \; E \mid identifier \mid number$$

Practice 2.3 Construct a left-most derivation for the prefix expression $+4 * -abx$.
You can check your answer(s) in Section 2.17.3.

2.5 Parse Trees

A grammar, G, can be used to build a tree representing a sentence of $L(G)$, the language of the grammar G. This kind of tree is called a *parse tree*. A parse tree is another way of representing a sentence of a given language. A parse tree is constructed with the start symbol of the grammar at the root of the tree. The children of each node in the tree must appear on the right hand side of a production with the parent on the left hand side of the same production. A program is syntactically valid if there is a parse tree for it using the given grammar.

While there are typically many different derivations of a sentence in a language, there is only one parse tree. This is true as long as the grammar is not ambiguous. In fact that's the definition of ambiguity in a grammar. A grammar is *ambiguous* if and only if there is a sentence in the language of the grammar that has more than one parse tree.

The parse tree for the sentence derived in Sect. 2.4.1 is depicted in Fig. 2.1. Notice the similarities between the derivation and the parse tree.

Practice 2.4 What does the parse tree look like for the right-most derivation of $(5 * x) + y$?
You can check your answer(s) in Section 2.17.4.

Practice 2.5 Construct a parse tree for the infix expression $4 + (a - b) * x$.
HINT: What has higher precedence, "+" or "*"? The given grammar automatically makes "*" have higher precedence. Try it the other way and see why!
You can check your answer(s) in Section 2.17.5.

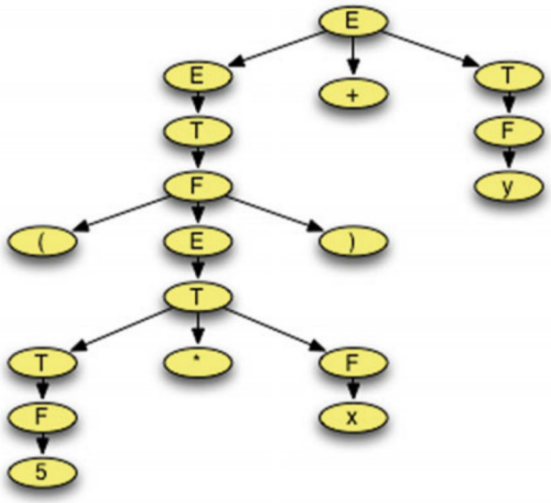

Fig. 2.1 A parse tree

Practice 2.6 Construct a parse tree for the prefix expression $+4 * -abx$.
You can check your answer(s) in Section 2.17.6.

2.6 Abstract Syntax Trees

There is a lot of information in a parse tree that isn't really needed to capture the meaning of the program that it represents. An abstract syntax tree is like a parse tree except that non-essential information is removed. More specifically,

- Nonterminal nodes in the tree are replaced by nodes that reflect the part of the sentence they represent.
- Unit productions in the tree are collapsed.

For example, the parse tree from Fig. 2.1 can be represented by the abstract syntax tree in Fig. 2.2. The abstract syntax tree eliminates all the unnecessary information and leaves just what is essential for evaluating the expression. Abstract syntax trees, often abbreviated ASTs, are used by compilers while generating code and may be used by interpreters when running your program. Abstract syntax trees throw away superfluous information and retain only what is essential to allow a compiler to generate code or an interpreter to execute the program.

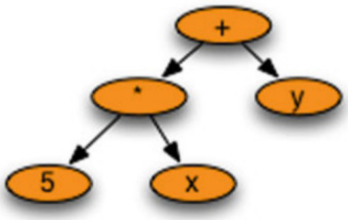

Fig. 2.2 An AST

Practice 2.7 Construct an abstract syntax tree for the expression $4+(a-b)*x$.
You can check your answer(s) in Section 2.17.7.

2.7 Lexical Analysis

The syntax of modern programming languages are defined via grammars. A grammar, because it is a well-defined mathematical structure, can be used to construct a program called a parser. A language implementation, like a compiler or an interpreter, has a parser that reads the program from the source file. The parser reads the tokens, or terminals, of a program and uses the language's grammar to check to see if the stream of tokens form a syntactically valid program.

For a parser to do its job, it must be able to get the stream of tokens from the source file. Forming tokens from the individual characters of a source file is the job of another program often called a tokenizer, or scanner, or lexer. Lex is the Latin word for *word*. The words of a program are its tokens. In programming language implementations a little liberty is taken with the definition of *word*. A *word* is any terminal or token of a language. It turns out that the tokens of a language can be described by another language called the language of regular expressions.

2.7.1 The Language of Regular Expressions

The language of regular expression is defined by a context-free grammar. The context-free grammar for regular expressions is $RE = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, T, K, F\}$
$\mathcal{T} = \{character, *, +, ., (,)\}$
$\mathcal{P}$ is defined by the set of productions

Figure 2.3 depicts a finite state machine for the language of infix expression tokens. The start state is 1. Each of states 2 through 9 are accepting states, denoted with a double circle. State 2 accepts identifier tokens. State 3 accepts number tokens. States 4 to 9 accept operators and the parenthesis tokens. The finite state machine accepts one token at a time. For each new token, the finite state machine starts over in state 1.

If, while reading a token, an unexpected character is read, then the stream of tokens is rejected by the finite state machine as invalid. Only valid strings of characters are accepted as tokens. Characters like spaces, tabs, and newline characters are not recognized by the finite state machine. The finite state machine only responds with *yes* the string of tokens is in the language accepted by the machine or *no* it is not.

2.7.3 Lexer Generators

It is relatively easy to construct a lexer by writing a regular expression, drawing a finite state machine, and then writing a program that mimics the finite state machine. However, this process is largely the same for all programming languages so there are tools that have been written to do this for us. Typically these tools are called lexer generators. To use a lexer generator you must write regular expressions for the tokens of the language and provide these to the lexer generator.

A lexer generator will generate a lexer program that internally uses a finite state machine like the one pictured in Fig. 2.3, but instead of reporting *yes* or *no*, for each token the lexer will return the string of characters, called the *lexeme* or *word* of the token, along with a classification of the token. So, identifiers are categorized as *identifier* tokens while '+' is categorized as an *add* token.

The *lex* tool is an example of a lexical generator for the C language. If you are writing an interpreter or compiler using C as the implementation language, then you would use *lex* or a similar tool to generate your lexer. *lex* was a tool included with the original *Unix* operating system. The Linux alternative is called *flex*. Java, Python, Standard ML, and most programming languages have equivalent available lexer generators.

2.8 Parsing

Parsing is the process of detecting whether a given string of tokens is a valid sentence of a grammar. Every time you compile a program or run a program in an interpreter the program is first parsed using a parser. When a parser isn't able to parse a program the programmer is told there is a *syntax error* in the program. A *parser* is a program that given a sentence, checks to see if the sentence is a member of the language of the given grammar. A parser usually does more than just answer *yes* or *no*. A parser frequently builds an abstract syntax tree representation of the source program. There are two types of parsers that are commonly constructed.

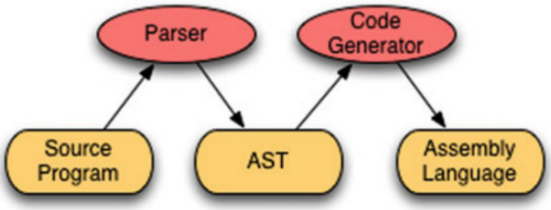

Fig. 2.4 Parser data flow

- A *top-down parser* starts with the root of the parse tree.
- A *bottom-up parser* starts with the leaves of the parse tree.

Top-down and bottom-up parsers check to see if a sentence belongs to a grammar by constructing a derivation for the sentence, using the grammar. A parser either reports success (and possibly returns an abstract syntax tree) or reports failure (hopefully with a nice error message). The flow of data is pictured in Fig. 2.4.

2.9 Top-Down Parsers

Top-down parsers are generally written by hand. They are sometimes called recursive descent parsers because they can be written as a set of mutually recursive functions. A top-down parser performs a left-most derivation of the sentence (i.e. source program).

A top-down parser operates by (possibly) looking at the next token in the source file and deciding what to do based on the token and where it is in the derivation. To operate correctly, a top-down parser must be designed using a special kind of grammar called an LL(1) grammar. An LL(1) grammar is simply a grammar where the next choice in a left-most derivation can be deterministically chosen based on the current sentential form and the next token in the input. The first L refers to scanning the input from left to right. The second L signifies that while performing a left-most derivation, there is only *1* symbol of lookahead that is needed to make the decision about which production to choose next in the derivation.

2.9.1 An LL(1) Grammar

The grammar for prefix expressions is LL(1). Examine the prefix expression grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E\}$
$\mathcal{T} = \{identifier, number, +, -, *, /\}$
$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow + \; E \; E \mid - \; E \; E \mid * \; E \; E \mid / \; E \; E \mid identifier \mid number$$

While constructing any derivation for a sentence of this language, the next production chosen in a left-most derivation is going to be obvious because the next token of the source file must match the first terminal in the chosen production.

2.9.2 A Non-LL(1) Grammar

Some grammars are not LL(1). The grammar for infix expressions is not LL(1). Examine the infix expression grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, T, F\}$
$\mathcal{T} = \{identifier, number, +, -, *, /, (,)\}$
$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow E \; + \; T \mid E \; - \; T \mid T$$
$$T \rightarrow T \; * \; F \mid T \; / \; F \mid F$$
$$F \rightarrow (\; E \;) \mid identifier \mid number$$

Consider the infix expression $5 * 4$. A left-most derivation of this expression would be

$$E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow 5 * F \Rightarrow 5 * 4$$

Consider looking at only the 5 in the expression. We have to choose whether to use the production $E \rightarrow E \; + \; T$ or $E \rightarrow T$. We are only allowed to look at the 5 (i.e. we can't look beyond the 5 to see the multiplication operator). Which production do we choose? We can't decide based on the 5. Therefore the grammar is not LL(1).

Just because this infix expression grammar is not LL(1) does not mean that infix expressions cannot be parsed using a top-down parser. There are other infix expression grammars that are LL(1). In general, it is possible to transform any context-free grammar into an LL(1) grammar. It is possible, but the resulting grammar is not always easily understandable.

The infix grammar given in Sect. 2.9.2 is left recursive. That is, it contains the production $E \rightarrow E + T$ and another similar production for terms in infix expressions. These rules are left recursive. Left recursive rules are not allowed in LL(1) grammars. A left recursive rule can be eliminated in a grammar through a straightforward transformation of its production.

Common prefixes in the right hand side of two productions for the same nonterminal are also not allowed in an LL(1) grammar. The infix grammar given in Sect. 2.9.2 does not contain any common prefixes. Common prefixes can be eliminated by introducing a new nonterminal to the grammar, replacing all common prefixes with the new nonterminal, and then defining one new production so the new nonterminal is composed of the common prefix.

2.9.3 An LL(1) Infix Expression Grammar

The following grammar is an LL(1) grammar for infix expressions. $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, RestE, T, RestT, F\}$
$\mathcal{T} = \{identifier, number, +, -, *, /, (,)\}$
$\mathcal{P}$ is defined by the set of productions

$$
\begin{aligned}
&E \rightarrow T\ RestE \\
&RestE \rightarrow +\ T\ RestE \mid -\ T\ RestE \mid \epsilon \\
&T \rightarrow F\ RestT \\
&RestT \rightarrow *\ F\ RestT \mid /\ F\ RestT \mid \epsilon \\
&F \rightarrow (\ E\ ) \mid identifier \mid number
\end{aligned}
$$

In this grammar the ϵ (pronounced epsilon) is a special symbol that denotes an empty production. An empty production is a production that does not consume any tokens. Empty productions are sometimes convenient in recursive rules.

Once common prefixes and left recursive rules are eliminated from a context-free grammar, the grammar will be LL(1). However, this transformation is not usually performed because there are more convenient ways to build a parser, even for non-LL(1) grammars.

Practice 2.9 Construct a left-most derivation for the infix expression $4 + (a - b) * x$ using the grammar in Sect. 2.9.3, proving that this infix expression is in L(G) for the given grammar.

You can check your answer(s) in Section 2.17.9.

2.10 Bottom-Up Parsers

While the original infix expression language is not *LL(1)* it is *LALR(1)*. In fact, most grammars for programming languages are LALR(1). The *LA* stands for *look ahead* with the *1* meaning just one symbol of look ahead. The *LR* refers to scanning the input from left to right while constructing a right-most derivation. A bottom-up parser constructs a right-most derivation of a source program in reverse. So, an LALR(1) parser constructs a reverse right-most derivation of a program.

Building a bottom-up parser is a somewhat complex task involving the computation of item sets, look ahead sets, a finite state machine, and a stack. The finite state machine and stack together are called a *pushdown automaton*. The construction of the pushdown automaton and the look ahead sets are calculated from the grammar. Bottom-up parsers are not usually written by hand. Instead, a parser generator is used