Main content

Chapter

10

The Tower of Babel: Programming Languages

Main content

Chapter Contents

Chapter Introduction

r studying this chapter, you will be able to:

- Explain why so many programming languages exist
- List four key procedural languages and the main purpose for the development of each
- Describe the purpose of each special-purpose language: SQL, HTML, JavaScript, and R
- Describe the alternative paradigms for programming languages: functional, logic, and parallel
- Name a functional programming language and a logic programming language
- Describe how logic programming languages work, and explain what facts, rules, and inference are
- Explain how the MIMD model of parallel processing could be used to find the largest number in a list

Main content

Chapter Contents

10.1 Why Babel?

The biblical story of the Tower of Babel takes place at a time when “the whole earth had one language and the same words.” The people began to build a city with a mighty tower when, suddenly, everyone began speaking in different tongues and could no longer communicate. They became confused, abandoned the tower, and scattered “over the face of all the earth.” A shared enterprise was impossible to pursue without the mutual understanding fostered by a common language, and (the message this story was intended to convey) the power of what people could do was thus forever limited.

Similarly, it might seem that having all computer programs written in the same programming language would have an appealing simplicity. Chapter 9 gave a brief comparison of five general-purpose programming languages: Ada, C++, C#, Java, and Python. By now, you may have also studied one or more of these languages in some depth through the online language chapters. But again, why aren’t all programs written in the best one of these languages? Or does each of these languages have some things it can’t do that some of the other languages can do? If so, then why aren’t all programs written in some “superlanguage” that overcomes these deficiencies?

There are multiple programming languages not so much because there are tasks that one language cannot do but because each language was designed to meet the specific needs of one particular area of application. Consequently, one language might be better suited than others for writing certain kinds of programs. The situation is somewhat analogous to the automobile market. The basic automotive needs of the country probably could be served by a single car model and a single truck model. So why do we have seemingly endless models from which to choose? The answer lies partly in competition: Automotive companies are all trying to claim a share of the market. More than that, though, the answer lies in the variety of ways we use our automobiles. Although a luxury car could be used for off-roading, it is not designed for that use; a four-wheel-drive vehicle does the job better, more safely, and more efficiently. Although a sports car could be used to haul Little Leaguers home from the ball game, it is not designed for that use; an SUV serves this purpose better. The diversity of tasks for which we use our automobiles has promoted a variety of automotive models, each designed to handle a particular range of tasks.

The same thing applies to programming languages. For example, we *could* use C++ to write programs for solving engineering problems (and it has indeed been so used). However, C++ was not designed with engineering applications in mind. Although C++ supports the basic arithmetic operations of addition, subtraction, multiplication, and division by providing simple operators (+, −, *, /) to do these tasks, there is no operator for exponentiation—that is, raising a value to a power. Computing in a C++ program, for example, can certainly be done but it requires some effort.* Calculations involving exponents are performed hundreds of times in many engineering and other numerical-based applications, so why not use a language that provides an operator for exponentiation, a language designed with such tasks in mind? Ada has such an exponentiation operator, as does Fortran, which we'll discuss in the next section. Similarly, suppose our program writes complicated sales reports with columns of figures and blocks of information strategically located on the page. Specifying the exact placement of output on the page is rather tedious in C++ or Java. Why not use a language that allows detailed output formatting because it was designed with such a purpose in mind? Again, we'll briefly discuss such a language—COBOL—in the next section.

What if we want a program to interact with a database, to manipulate graphics, or to act as a hyperlinked webpage? Any of these specialized tasks is probably best done with a language designed for just that purpose.

A major reason, then, for the proliferation of programming languages is the proliferation of distinct programming tasks. Another reason is that different philosophies have developed about how people should think when they are writing programs. This has resulted in several families of programming languages that take quite different approaches from the ones we've looked at so far, and we'll see some of these approaches in Section 10.4.

Main content

Chapter Contents

10.2Procedural Languages

As mentioned earlier, all the languages of Chapter 9 (Ada, C++, C#, Java, and Python) are *procedural languages*. Programs written in such languages differ in the way the statements must be arranged on a line and in how variables can be named. They differ in the details of assigning a new value to a variable, in the mechanisms the language provides for directing the flow of control through conditional and looping statements, and in the statement forms that control input and output. They also differ in the way programs can be broken down into modules to handle separate tasks and in how those modules share information. We noted some of these syntactical differences in the Feature Analysis table (Figure 9.15) of Chapter 9. But all procedural language programs tell the computer in a step-by-step fashion how to manipulate the contents of memory locations. In a general sense, then, the languages are quite similar, just as French, Spanish, and Italian are all members of the family of Romance languages. In this section we concentrate not on syntactical differences, but on the history and "intent" of some of the most important procedural languages—important in that, of the many programming languages that have come and gone over the years, these became widely used. The languages of Chapter 9 are included here, but there are additional languages as well.

Main content

Chapter Contents

10.2.1 Plankalkül

What? OK, this language never became widely used. In fact, it was never even implemented. It's a programming language designed by Konrad Zuse who, you may recall from Chapter 1, built a computer in Germany during World War II. The manuscript describing this programming language was completed in 1945 but was not published until 1972. The manuscript contained a number of complex algorithms written in Plankalkül (the name, in German, means "formal planning system"). The language itself, although burdened with obscure notation, contained a number of sophisticated concepts that, had they been known earlier, might have changed the development of programming languages. As the very first attempt to design a high-level programming language for computers, Zuse's proposal was a very important milestone in the field of computing, although it never received the attention it deserved.

Main content

Chapter Contents

10.2.2 Fortran

The name Fortran derives from *For*mula *Tran*slating System. The very name indicates its affiliation with "formulas" or engineering-type applications. Developed in the mid-1950s by a group at IBM headed by John Backus, in conjunction with some IBM computer users, the first commercial version of Fortran was released in 1957. This makes Fortran the first high-level programming language that was actually implemented. Early computer users were often engineers who were solving problems with a heavy mathematics or computational flavor. Fortran has some features ideally suited to these applications, such as the exponentiation operator we mentioned earlier, as well as the ability to carry out extended-precision arithmetic with many decimal places of accuracy, and the ability to work within the complex number system. Updated versions of Fortran have been introduced over the years, incorporating new data types, new statements to direct the flow of control, and the ability to use recursion (recursion is discussed later in this chapter). Object-oriented programming is supported in later versions of the language.

Early versions of Fortran required all variable identifiers to be uppercase. Also it did not allow the use of mathematical symbols such as < to compare two quantities; the keypunches that were used to create the punched cards on which early Fortran programs were submitted to the computer had no such symbols. Thus the condition we would usually write now as

would have been expressed as

There was no while loop mechanism. The effect of a while loop was obtained by using an IF statement together with GO TO statements. The pseudocode

would have been accomplished by

READ is the Fortran implementation of "input," so the first line inputs a value for *NUMBER*. If *NUMBER* is less than 0, the GO TO statement transfers control to statement 20. If *NUMBER* is greater than or equal to 0, something is done and then another value for *NUMBER* is obtained. Control is then redirected by the second GO TO statement back to statement 10 where the new value is tested. Directing the flow of control by GO TO statements is similar to using the various JUMP statements in the assembly language of Chapter 6, and it reflects the fact that Fortran's developers were, after all, working from assembly language.

Fortran was designed to support numerical computations. This led to concise mathematical notation (aside from the early < dilemma just mentioned) and to the availability of a number of mathematical functions within the language. Another design goal was to optimize the resulting object code, that is, to produce object code that took as little space and executed as efficiently as possible. (Remember that when Fortran was developed, machine resources were scarce and precious.) Fortran allows **external libraries** of well-written, efficient, and thoroughly tested code modules that are separately compiled and then drawn on by any program that wants to use their capabilities. Because of Fortran's extensive use as a programming language over the years, a large and well-tested Fortran library collection exists, so in many cases programmers can use existing code instead of having to write all code from scratch. This feature is sometimes highly touted for newer languages, but Fortran designers got there first.

The next revision of the ISO (International Organization for Standardization) Fortran standard is Fortran 2015, scheduled for publication in 2018.

Change font sizeMain content

Chapter Contents

10.2.3 COBOL

The name COBOL derives from *CO*mmon *B*usiness-*O*riented *L*anguage. COBOL was developed in 1959 and 1960 by a group headed by Admiral Grace Murray Hopper of the U.S. Navy. Fortran and COBOL were the dominant high-level languages of the 1960s and 1970s. COBOL was designed to serve business needs such as managing inventories and payrolls. In such applications, summary reports are important output products. Much of the processing in the business world concerns updating "master files" with changes from "transaction files." For example, a master inventory file contains the names, manufacturers, and quantities on hand of items in inventory; a transaction file has the names and quantities of items sold out of inventory or delivered to inventory over some period of time. The master file is updated from the transaction file on a daily or weekly basis to reflect the new quantities available, and a summary report is printed. The user doesn't interact directly with the COBOL program; rather, the user prepares the master file (once) and the transaction file (regularly).

Old Dog, New Tricks #1

Fortran was first introduced in 1957. In the history of computing, this is roughly the Jurassic Age, but Fortran is no extinct dinosaur. Instead, it is a chameleon, changing with the times. Fortran now runs on PCs, tablets, and even smartphones while still providing the power to help supercomputers tackle some of the most computationally intensive problems ever. Fortran programs can present a graphical user interface to help the user easily operate the program and perhaps visualize the results.

Parallelism is especially useful for speeding up the kinds of calculations on huge arrays that often occur in scientific and engineering problems, Fortran's traditional domain. An *array* is a block of numerical data, and manipulation of array values often occurs in such tough numerical problems as climate modeling, computational fluid dynamics, and computational economics. Fortran now includes the ability to use *coarrays*, in which arrays can be split between multiple processors for parallel processing, and the communication between these processors is easily managed. Problems with real-time response requirements in the areas of signal processing and image processing are also appropriate for parallelism; Fortran programs often beat more "sophisticated" languages in raw speed.

In the design of COBOL, particular attention was paid to input formatting for data being read from files and to output formatting both for writing data to a file and for generating business reports with information precisely located on the page. Therefore, much of a COBOL program may be concerned with formatting, described by "PICTURE clauses" in the program. COBOL was also designed such that programs describe what they are doing in natural language phrases. As a result, COBOL programs can be rather verbose.

Practice Problems

Write a Fortran condition to test whether the value of *ITIME* is less than or equal to 7. Use early Fortran syntax.

Answer

Using the if/then/else construct introduced in Chapter 2, rewrite the following Fortran code fragment so that it does not require any GO TO statements:

Answer

In the code fragment of Practice Problem 2, suppose that *X* has the value 2, *A* has the value 7, *B* has the value 1, and that statement 30 is

What is the final value of *A*?

Answer

Because X has the value 2 and 2 < 3, control transfers to statement 10, which sets the value of A to 2. The next statement transfers control to statement 30, at which point A has the value 2 and B still has the value 1 because statement 20 was never executed. Therefore the new value of A is .

COBOL programs are highly portable across many different COBOL compilers, are quite easy to read, and are very well suited to manipulating large data files. Because COBOL has been around for a long time, there is a huge base of existing COBOL applications. Nonetheless, the continuing importance of COBOL as a commercial programming language had perhaps been overlooked by those outside the business world until the "Year 2000 problem" came along. The Y2K problem (K stands for *kilo*, or "thousand") dealt with a lurking time bomb in **legacy code** (old computer code that is still in use), primarily COBOL code. When these programs were written, their authors never imagined their longevity. In addition, computer memory was at a premium, so efficiency was the order of the day. Why store four digits of a date (1967, say) when two digits (67)—the "19" prefix was assumed—would be sufficient and take less space? In the new millennium, "02" should mean "2002," but in these programs it would be interpreted incorrectly as "1902."

Making code Y2K-compliant was technically simple: Just change every date reference to four digits instead of two. It was the magnitude of the task that was staggering because it was necessary to locate each line of code where a date entry needed to be changed. Huge sums of government and corporate money were spent to address the problem and, despite dire predictions on the potential consequences of Y2K, it proved to be a "nonevent"—probably because of the massive effort made to address the problem.

So, does post-Y2K mark the death of COBOL? Absolutely not! All this money was not spent on fixing code that businesses planned to throw away. On the contrary, the majority of business transactions, for retail point-of-sale systems, banking, insurance, payroll, ATMs, hospital systems, and government systems are still done on COBOL code that has now been updated and is likely to continue to run for the foreseeable future. Even in 2016, the Social Security Administration relied on more than 60 million lines of COBOL code.

The current international standard for COBOL was published in 2014.

Practice Problem

Write statements in your choice of language from Chapter 9 that are equivalent to the following COBOL statements:

Answer

In Ada:

In C++, C#, or Java:

In Python:

Uncle Sam Wants *Who*?

A Y2K-like problem arose in June 2014 after the Pennsylvania Department of Motor Vehicles completed a computer-generated transfer to the U.S. Selective Service of some 400,000 records it had on file. (Males in the United States between the ages of 18 and 25 must register with the Selective Service for possible military draft.) These included records for males born 1993–1997. However, the state agency uses only a 2-digit indicator for the year of birth, so this transfer also included records for males born 1893–1897. The Selective Service duly sent letters out to those in both groups ordering them to register for the military draft or face fines and imprisonment. Many of the more than 14,000 letters to the 1893–1897 group reached bewildered relatives of these no-doubt long-dead gentlemen. Oops!

Main content

Chapter Contents

10.2.4 C/C++

C was developed in the early 1970s by Dennis Ritchie at AT&T Labs. It was originally designed for systems programming, in particular for writing the operating system UNIX. UNIX had been developed at Bell Labs a short time before and was originally written primarily in assembly language. Ritchie sought a high-level language in which to rewrite the operating system in order to gain all the advantages of high-level languages: ease of programming, portability, and so on.

Since that time, C has become a popular general-purpose language for two major reasons. One is the close relationship between C and UNIX. UNIX has been implemented on many different computers (and is the basis for the macOS operating system). UNIX provides many "tools" that support C programming. A second reason for C's popularity is its efficiency—that is, the speed with which its operations can be executed. This efficiency derives from the fact that C programs can make use of low-level information such as knowledge of where data are stored in memory. In this respect, C is closer to assembly language than are other high-level languages, yet it still has the powerful statements and portability to many machines that high-level languages offer. You can imagine C humming along as a high-level language but then, every once in a while when efficiency is really important, slipping into a low-level, machine-dependent configuration. One of the goals of a high-level language is to provide a level of abstraction that shields the programmer from any knowledge of the actual hardware/memory cells used during program execution, as depicted in Figure 10.1(a). C provides this outlook, unless the programmer wants to make use of the low-level constructs available in C that give him or her a direct view of the actual hardware, which Figure 10.1(b) depicts.

Figure 10.1 User-hardware interface and programming languages

For example, suppose *number* is a variable in a C program with the value 234. The value of *number* is stored in some specific memory location with address, say, 1000 (Figure 10.2). The notation *&number* in that same program refers to the memory address where the value of *number* is stored, in this case, 1000. Note the distinction between the content of a memory cell and the address of that cell. Here *number* refers to the value 234, but *&number* refers to the address 1000. It is possible to write a C program statement that passes *&number* as an argument to an output function so that the program actually writes out the memory address value (1000). The ability to work with an actual memory address is not available in most other high-level languages.

Figure 10.2C allows access to a memory cell address as well as to its content

C not only provides a way to see the actual memory address where a variable is stored, it also gives the programmer some control over the address where information is stored. C includes a data type called *pointer*. Variables of pointer type contain—instead of integers, real numbers, or characters—memory addresses. For example, the statement

declares *intPointer* as a pointer variable that will contain the address of a memory cell containing integer data. The assignment

assigns the memory address 800 to be the value of *intPointer*. Figure 10.3(a) illustrates this situation: The pointer variable *intPointer* is stored at some unknown memory address, but the content of *intPointer* is the memory address 800. The value stored at the address contained in *intPointer*, in this case stored at 800, is denoted by **intPointer*. In other words, **intPointer* is the value contained in the address to which *intPointer* points. We can find out what this value is by writing out **intPointer*. We can also assign an integer value, say 3, to be the content of memory address 800 by the statement

which results in Figure 10.3(b). We have controlled the content of a specific memory location, and now we know exactly what is stored in memory location 800. Similarly, if *number* is an integer variable that has been stored somewhere in memory, then the statement

Figure 10.3Storing a value in a specific memory location using C

results in the value of *number* being stored in memory cell 800.

This capability for low-level memory manipulation resembles the assembly language programming of Chapter 6. It is fraught with the same problems we sought to avoid by going to high-level languages in the first place; specifically, the programmer is assuming responsibility for what is stored where. For example, what if memory cell 800 in our example is not a memory cell allocated to this program? Perhaps something needed by another program, or even by the operating system, has been overwritten. However, the fact that it enables the programmer to reach down into the machine level is precisely why C is useful for writing system software such as operating systems, assemblers, compilers, and input/output controllers.

A program that interacts with an I/O device is called a **device driver**. Consider, for example, the problem of writing a device driver for the mouse on a PC or laptop. If the mouse is a wired mouse, it is connected to a USB port on the computer that reads changes in the mouse position by changes in voltage levels. In the case of a wireless mouse, the mouse sends radio frequency or Bluetooth signals about the mouse position to a receiver in the computer. In either case, this data is stored in fixed locations in memory that are allocated by the operating system. The job of the mouse driver is to translate this data to specific locations on the screen so that any application software that uses the mouse, such as a word processor, does not have to interact with low-level hardware information (abstraction again!). The mouse driver program has to access the specific memory locations where the mouse location data has been stored. A language like C provides such a capability.

C is the most widely used language for writing system software because it combines the power of a high-level language with the ability to circumvent that level of abstraction and work at the assembly-language-like level. But C is also used for a great deal of general-purpose computing.

The C++ language was developed in the early 1980s by Bjarne Stroustrup, also at AT&T Labs. C++ is in fact a “superset” of C, meaning that all of the C language is part of C++. Everything that can be done in C—including the ability to change the contents of specific memory locations—can be done in C++. But C++ adds many new features to C, giving it more sophistication and cleaner ways to do certain tasks. The most significant extension of C that C++ provides is the ability to do object-oriented programming.

C++ was first commercially released by AT&T in 1985. Like many other languages, C++ has evolved over time. The standardization process for the language took more than 10 years, in part because of this evolution. In November 1997, the combined C++ subcommittees of ANSI and ISO submitted their C++ standards draft, part of a document of some 800 pages, for final ISO approval. The standards were finally approved in 1998. Standardization, object orientation, and a strong collection of library code have helped to make C++ one of the most popular of the modern “industrialstrength” languages.

The newest C++ international standard is C++ 14, published in 2014, but at the time of this writing, a major revision (C++ 17) is expected to receive final approval in 2017, with work on C++ 20 to begin right after that.

Practice Problems

Suppose a C/C++ program uses a variable called *Rate*. Explain the distinction in the program between *Rate* and *&Rate*.

Answer

Rate refers to the contents of the memory cell called *Rate; &Rate* refers to the address of that cell.

Suppose that *Rate* is an integer variable in a C/C++ program with the value 10 and that *intPointer* is a pointer variable for integer data. *Rate* is stored at memory address 500. After the statement

is executed, what is the value of **intPointer*?

Answer

10

The following statement is executed after the statement in Problem 2.

What is the value of *&Rate* now? What is the value of *Rate* now?

Answer

500; 29

Main content

Chapter Contents

10.2.5 Ada

Probably more than any other language we have mentioned, Ada has a long and interesting development history. It began in the mid-1970s when the various branches of the U.S. armed services set about trying to develop a common high-level programming language for use by defense contractors. They began by specifying the requirements that the language would have to meet, including such characteristics as efficiency, reliability, readability, and maintainability.

The original set of requirements, first circulated for discussion in 1975, was known as “Strawman.” Successively tighter and more thorough requirements bore the names “Woodenman” and “Tinman.” The Tinman requirements were approved in 1976, and many existing programming languages were evaluated in the light of these requirements. All were found wanting, and it became clear that a new language would have to be developed. The “Ironman” specification, issued in 1977, became the standard against which to measure a new language. A design competition was held, and the requirements were further specified in “Steelman.”

The eventual language-design winner was chosen in 1979, and the new language was christened Ada, after Ada Augusta Byron Lovelace, daughter of the poet Lord Byron and later the wife of Lord Lovelace. Ada was trained in mathematics and science at the wish of her mother, who sought to steer Ada away from the mental instability and moral lapses she despised in Lord Byron. Lady Ada Lovelace is regarded as the world’s first programmer on the basis of her correspondence with Charles Babbage and her published notes on his work with the Analytical Engine (see the Special Interest Box on Charles Babbage and Ada Augusta Byron in Chapter 1).

Ada, like C++, is a large language, and it was adopted not only by the defense industry, where its use was mandated by the U.S. Department of Defense, but also for other technological applications and as a general-purpose language as well. Ada is known for its multiprocessing capability—the ability to allow multiple tasks to execute independently and then synchronize and communicate when directed. It is also known as a strongly object-oriented language.

The Department of Defense “Ada mandate” was terminated in 1997, but by then Ada was well established as a programming language supporting good software engineering practice, safety, and reliability. Today, Ada is still strong in the transportation industry (aircraft, helicopters, subway systems, and European high-speed train control systems) and in safety monitoring systems at nuclear reactors, as well as in financial and communication systems. Its proponents tout Ada as “the language designed for building systems that really matter.” The newest international Ada standard was adopted in 2012.

SPARK 2014 is a programming language that is a well-defined subset of Ada 2012. SPARK aims to build upon the strengths of Ada while eliminating some of its ambiguities and security concerns. SPARK goes even farther than Ada in support for critical systems development. For example, SPARK supports “contracts” that assert conditions that should be true before a module of code is executed and conditions that should be true as a result of that execution, plus a tool that can analyze the code to verify correctness of these conditions.

Practice Problem

What do you think is accomplished by the following Ada program?

Answer

The program prints the numbers from 1 through 10 on a single line with a blank space between them.

Main content

Chapter Contents

10.2.6 Java

Unlike Fortran, COBOL, C, C++, and Ada, which were carefully developed as programming languages, Java was almost an accident. In early 1991, Sun Microsystems Inc. created a team of top-notch software developers and gave them free rein to do whatever creative thing they wanted. The somewhat secret “Green Team” isolated itself and set to work mapping out a strategy. Its focus was on the consumer electronics market. Televisions, VCRs, stereo systems, laser disc players, and video game machines all operated on different CPUs. Over the next 18 months, the team worked to develop the graphical user interface (GUI), a programming language, an operating system, and a hardware architecture for a handheld, remote-control device called the *7 that would allow various electronic devices to communicate over a network. The *7 was designed to be small, inexpensive, easy to use, reliable, and equipped with software that could function over the multiple hardware platforms of the consumer electronics market at that time.

Armed with this technology, Sun went looking for a business market but found none. In 1993, Mosaic, the first graphical Internet browser, was created at the National Center for Supercomputing Applications, and the World Wide Web began to emerge. This development sent the Sun group in a new direction, where its capabilities with platform independence, reliability, security, and GUI paid off: The group wrote a web browser using the programming language (later named Java) of the *7. The web browser was released in 1995, and the first version of the Java programming language itself was released in 1996. After that, Java gained market share among programming languages at quite a phenomenal rate.

Java **applications** are complete stand-alone programs that reside and run on a self-contained computer; these are the kinds of programs we illustrated in Chapter 9, and Java applications are everywhere. As a modern example, Twitter handles over 400 million tweets per day, and much of its core service software is written in Java. Java's development went hand in hand with the development of web browsers.
Java **applets** (small applications) are programs designed to run from webpages. Applets are embedded in webpages on central servers; when the user views a webpage with a Java-enabled browser, the applet's code is temporarily transferred to the user's system (whatever that system may be) and interpreted/executed by the browser itself. Java applets bring audio, video, and real-time user interaction to webpages, making them "come alive" and become much more than static hyperlinked text. For example, a Java applet might display a streaming ticker tape of stock market quotes or a form that allows you to book an airline reservation online. Due to security issues, however, most web browsers no longer support Java applets, but instead use a technology called **Java Web Start** (or JAWS) where Java code (even for full-fledged Java applications) is accessed from the web via the user's browser, but is executed in a restricted environment outside the web browser itself. Existing Java applets can be converted to Web Start applications.

Java applications also can run on the processor of Android smartphones. (Google Android is the operating system on the majority of the world's smartphones; Android phones represented about 88% of the total number of units shipped in the third quarter of 2016.) Android apps are usually written in Java, and there are more than 2.2 million apps, by far the majority of them free. Java has certainly made its way into many hands (or many pockets).

Java is an object-oriented language based on C++, but it avoids some of the features that can make C/C++ programs error-prone. For example, in C++ we could declare an array of 12 integers by

The equivalent statement in Java is

Both Java and C++ number individual array locations beginning with 0, so there is no *hits[12]* in either case. In C++, you can write an assignment statement such as that destroys the contents of some memory location outside the array, and the program will go merrily on its way. Such an assignment in Java would cause a runtime error.

One of the main features of Java is its portability; recall that platform independence was one of the goals of the original Sun "Green Team." In most languages, source code gets compiled into the object code for a particular machine, which means that the developer who wants to distribute executable code needs to compile the source code on each target platform, using the appropriate compiler. The Java programmer, however, compiles source code just once, into low-level code called Java **bytecode**, which is then distributed to the various users. Bytecode is not itself the language of any real machine, but it can be easily translated into any specific machine language. This final translation/execution of bytecode is done by software called a **Java bytecode interpreter**, which must be present on each user machine. This approach is workable because the Java bytecode interpreter is a small piece of software.

Oracle Corporation acquired Sun Microsystems in 2010. The Java language is not defined by any international standards organization, so the de facto standard is whatever version of Java is currently supported by Oracle. OpenJDK is an open source version of Java for the Linux operating system.

Practice Problem

Output in Java is handled by requesting the predefined Java System.out object to invoke a println( ) function. Also, the + operator stands for string concatenation. What is the output after execution of the following Java statement if *number* has the value 7?

Answer

Main content

Chapter Contents

10.2.7 Python

The Python language was originally created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum in the Netherlands. Its development is now overseen by the Python Software Foundation, but Van Rossum still has the final stamp of approval on "official" features of the language. However, unlike most of the other languages mentioned here, Python is an open source language. The source code is freely available and can be used, distributed, or modified by anyone. Python's advocates claim that having a community of people interested in using and improving the language has led to a better design than would have resulted from standardized or proprietary code.

Python is an *interpreted language*, meaning it is translated from source to object code at every execution. Python was originally used for system administration tasks and as a web interface language. But with the development of an extensive library of supporting code, Python has become a powerful language for more general use. As an example, SciPy is an open-source Python library with code designed to support scientific and engineering computing needs.

Python's main distinguishing feature among the procedural family of languages is its ease of use. No doubt this accounts for its increasing use as the "first" programming language, that is, the language taught in introductory college programming courses. A survey conducted in July 2014 showed that of the top 39 U.S. universities (according to the annual *U.S. News and World Report* rankings), 69% were using Python as their introductory programming language, overcoming the dominance Java held for many years as the language taught in the first programming course.

The syntax rules for Python are relaxed and intuitive, making it easy to develop programs rapidly. However, this lack of rigidity can put the responsibility for careful usage on the programmer rather than on the compiler/interpreter.

The latest version of Python, as of this writing, was released in December 2016.

Main content

Chapter Contents

10.2.8 C# and .NET

In June 2000, Microsoft introduced the C# language (pronounced "C sharp"). This language is a successor in spirit to C++, but it is a totally new language. Therefore, it has no backward-compatibility issues, as C++ had with C. C# is designed to make some improvements over C++ in safe usage, and it shares many features with Java. As an example of potentially unsafe usage, a C++ program can dynamically grab additional memory for its use during program execution, but the programmer is responsible for releasing that memory when the program no longer needs it to reduce the possibility of running out of memory. In C#, however, this process of **garbage collection**—reclaiming memory no longer needed by the program—is handled automatically.

Practice Problem

The following are some sample programming statements (from Ada, C++, C#, Java, and Python) to output the programmer's typical first message of Hello World:

```
System.out.print("Hello World");
WITH TEXT_IO;

TEXT_IO.PUT("Hello World");

print("Hello World")
using System;

Console.Write("Hello World");

#include <iostream>

using namespace std;

cout << "Hello World";
```

Given our claims about the simplicity of Python syntax, which of these would you judge to be the Python output statement?

3. Answer
4. The Python version is answer c,

The "Popularity" Contest

There is a website that keeps track, on a monthly basis, of the most "popular" programming languages. According to this site, " popularity" is not about the best language or the one with the most lines of code that have been written. It is based mainly on data gleaned from web searches about the number of practicing programmers, courses taught, and third-party vendors. Nonetheless, it is an interesting site. Visit

www.tiobe.com/index.php/content/paperinfo/tpci/index.html

to see how your favorite language rates at the moment. For May 2017, some of the languages we have discussed (or will mention later) were rated as follows:

Language	Rating
Java	1
C	2
C++	3
Python	4
C#	5
Visual Basic .NET	6
Swift	13
R	14
Go	16
COBOL	25
Ada	29
Fortran	30
Scheme	34
Prolog	35

A different view of language popularity can be obtained from the following website http://stackoverflow.com/insights/survey/2016 that shows the results of a survey of more than 50,000 software developers. The most popular languages for 2016 were, in order, JavaScript, SQL, Java, and C#. We'll talk about JavaScript and SQL in Section 10.3.

It is impossible to discuss C# without discussing the **Microsoft .NET Framework** that supports C# and other programming languages. The .NET Framework is essentially a giant collection of tools for software development. It was designed so that traditional text-based applications, GUI applications, and web-based programs can all be built with equal ease. For example, the .NET Framework provides a whole library of classes for building GUIs with menus, buttons, text boxes, and so forth. And it is the .NET Framework (actually a part of the .NET Framework called the *Common Language Runtime* or *CLR*) that handles garbage collection for a C# program or for any other language that uses the .NET platform. All .NET programs—in whatever language—are compiled into **Microsoft Intermediate Language (MSIL)** code. Like Java bytecode, MSIL is not tied to any particular platform. The final step of compiling MSIL code into object code is done by a **just-in-time (JIT) compiler** (part of the CLR) on the user's machine. So, like Java, the developer achieves portability across multiple platforms because source code is compiled only once, into the MSIL.

There is one notable difference between the Java approach and the .NET approach. The Java bytecode translator is an interpreter, meaning that the first statement in a program is translated into object code and then executed, at which point the interpreter moves to the next statement and repeats these two operations, and so forth through the whole program. At the end of program execution, no object code is retained and the next time the program executes, the interpreter must repeat this task. The just-in-time compiler, on the other hand, senses when a particular module of MSIL code is being called, translates that module into object code, and then executes it. At the end of program execution, the object code for that module is still there, and if the program executes again with no changes, it can be run directly without invoking the JIT compiler. This difference between interpreted and compiled code leads to more efficient program execution.

Many programming languages have been adapted to fit into the .NET Framework, including Ada, Fortran, COBOL, C++, C# (of course), and Visual Basic .NET (see the Special Interest Box, "Old Dog, New Tricks #2"). That means applications written in any of these languages have access to the tools provided within the .NET Framework and, because all of these languages compile to MSIL, applications can be written that mix and match modules in various languages. Thus, the choice of which language to use becomes less an issue of language capability and more a matter of personal preference and familiarity.

In April 2003, only three years after the first release of C# and .NET, C# and the *CLI (Common Language Infrastructure)*—a significant subset of the .NET tools—were adopted as ISO standards. Edition 2 of the C# language specification was approved by ISO in 2006, but the latest version of C# is Edition 6, released in 2015. C# continues to grow in popularity as a programming language.

Old Dog, New Tricks #2

During the 1960s, programming was a rather difficult task relegated to technical professionals or, in the academic world, to advanced undergraduate engineering, math, and physics majors. Dartmouth mathematics professors John Kemeny and Thomas Kurtz wanted a programming language easy enough for anyone to learn, including high school and even elementary school students. They designed BASIC (Beginner's All-purpose Symbolic Instruction Code). Here is the first BASIC program shown in the May 1964 Instruction Manual for BASIC.*

This effort of a "programming language for the people" was very successful. BASIC was supplied with most microcomputers throughout the 1980s, and as such it introduced many people, in and out of school, to simple programming ideas. (One of the earliest BASIC compilers for microcomputers was developed by a then-tiny company called Microsoft, headed by 20-year-old Harvard University dropout Bill Gates.) But by 1990, after other much more powerful languages had appeared, BASIC had faded from view.

BASIC got a new lease on life and a whole new look when Microsoft released Visual Basic in 1991. Visual Basic supplied tools to create a sophisticated GUI application by simply dragging components such as buttons and text boxes from a Toolbox onto a form, and then writing BASIC code to allow those components to respond to events, such as the click of a button. This type of easy "drag and drop" programming made Visual Basic a very popular language for rapid prototyping of Windows applications. Subsequent versions of Visual Basic produced an ever-more-powerful language. Now Visual Basic .NET is a fully object-oriented language that, like the other .NET languages, can take advantage of all the built-in .NET Framework tools. Old languages that can evolve with the times need never die!

Practice Problem

A running Visual Basic program produces the following GUI:

The user types a name in the text box called *txtName*, then clicks the button called *btnShowName*. This "click event" is handled by the following Visual Basic module. Explain what you think happens when this module is executed.

Answer

The name entered in the text box will be displayed in the label.

Main content

Chapter Contents

10.3 Special-Purpose Languages

Although each of the procedural languages we have mentioned has its own strengths and weaknesses, all are more or less *general-purpose languages* that can address different types of problems. In this section, we visit four *special-purpose languages* that were each designed for only one particular task. These four are merely representative; many other specialized languages exist.

10.3.1 SQL

Our first specialized language is *SQL*, which stands for *Structured Query Language*. SQL is designed to be used with databases, which are collections of related facts and information. We'll do some work with databases in Chapter 14, but here is the general idea. A database stores data; the user of the database must be able to add new data and to retrieve data already stored. For example, a database contains information on vendors with which a retail store does business. For each vendor, the database contains the name, address, and phone number of the vendor, the name of the product line the vendor sells, and the amount of stock purchased from that vendor during the previous business quarter. The database user should be able to add information on a new vendor and retrieve information on a vendor already in the database.

But if this is all that a database can do, it simply acts as an electronic filing cabinet. Databases can also be queried—that is, the user can pose questions to the database. Queries can furnish information that is more than the sum of its parts because they combine the individual data items in various ways. For example, the vendor database can be queried to obtain the names of all vendors with whom the store has done more than $40,000 worth of business in the past quarter or all vendors from a certain zip code. SQL is the language used to frame database queries. Our two example queries might be expressed in SQL as

SQL was developed by IBM, and in 1986, it was adopted by the American National Standards Institute (ANSI) as the standard query language in the United States; it has since been adopted by the ISO as an international standard. Even database systems that provide users with easier—even graphical—ways to frame queries are simply using a front end that eventually

An SQL query does not give specific directions as to how to retrieve the desired result. Instead, it merely describes the desired result. This makes SQL similar in flavor to a logic programming language, which we'll see in a later section.

10.3.2 HTML

HTML stands for *H*yper*T*ext *M*arkup *L*anguage. It is used to create HTML documents that, when viewed with web browser software, become webpages. An HTML document consists of the text to be displayed on the webpage, together with a number of special characters called **tags** that achieve formatting, special effects, and references to other HTML documents. Although we speak of "HTML programming," that's a bit of a stretch.

The program is just giving the web browser instructions on how to interpret and display text; there's no computation or processing going on as we think of with programming in general.

HTML tags are enclosed in angle brackets (< >) and often come in pairs. The end tag, the second tag in the pair, looks like the begin tag, the first tag in the pair, but with an additional / in front.

The overall format for an HTML document is

Here we see the paired tags for the document as a whole (<html>, </html>), the head (<head>, </head>), the title (<title>, </title>)—framing what appears in the title bar or title tab when the web browser displays the page, and also what will be displayed in a list of search-engine results—and the body (<body>, </body>)—framing what is on the page itself.

Of course, other material needs to go between the beginning and ending "body" tags, or the page will be blank. Figure 10.4 shows an HTML document, and Figure 10.5 shows how the webpage actually looks when viewed with a web browser. By comparing the two, you can probably understand the meaning of the tags used, as explained in Figure 10.6.* Technically, href is an attribute of the "a" tag that indicates a link to another document, in this case the home page of the well-known PBS website. Similarly, src is an attribute of the "img" tag that gives the "source" for the image. In Figure 10.4, the source is just the image filename, meaning that the image file is in the same folder as the HTML code file; if the image file were elsewhere, the source would have to show the pathname where the image file can be found. The second attribute of the img tag, the *alt* attribute, is displayed if the image file cannot be located. The text in the *alt* attribute is also used by assistive technologies for visually impaired persons, who would otherwise be unable to "see" the image.

Figure 10.4 HTML code for a webpage

Figure 10.5 Body of webpage generated by the HTML code in Figure 10.4

Michael G. Schneider

Figure 10.6

Some HTML tags

HTML Tag	Purpose
`h1`	Create H1 heading (bold with largest font size)
`p`	New paragraph
`strong`	Important

HTML Tag	Purpose
`em`	Emphasized
`ul`	Unordered list (bulleted list)
`li`	List item
`a href = "…"`	Provides hyperlink address
`img src = "…"`	Gives the source (location) for the image file

Early word processors required the user to type in various codes manually to mark text for boldface, italic, and so forth. Later, more sophisticated word processors with GUI interfaces reduced these tasks to point and click. The same has come to pass with HTML code. HTML documents themselves are simply text files that can be created using any text editor by typing the appropriate tags. But web editor software makes it possible to create HTML code by, for example, highlighting text and clicking a button to insert a pair of tags that surround the highlighted text.

Main content

Chapter Contents

10.3.3 JavaScript

A scripting language is a "lightweight" language that is interpreted (translated, then executed, statement by statement). Scripting language code fragments can be embedded in webpages to make those pages active rather than static. JavaScript is such a language; keep in mind that JavaScript is not the same as the full-blown Java programming language we discussed earlier.

Consider the HTML page from the previous section. When this page is displayed by the web browser, the user has no interaction with the page except possibly to click on the hypertext link to go to another page. In particular, the image on the page is fixed. The webpage can be made a little more interesting if the image changes when the user hovers the mouse over the current image. In fact, let's switch back and forth between the original image and a second image as the user moves the mouse into and out of the image area of the page. To accomplish this, we'll use a JavaScript function called *imageSwitch()*. The JavaScript code will be placed within <script></script> tags to alert the browser that these statements are to be interpreted as JavaScript commands.

The *imageSwitch()* function must accomplish two tasks. One is to reset the src attribute of the "img" tag. But we have to say what image tag we are modifying, so in the original image tag we'll add an *id* attribute. Basically, we are giving the image a name; we'll call it *mainImage*. This name is not the name of an actual visual image (a picture); it's the name of the image element on the page that holds a picture as determined by its *src* attribute. In the original image tag, we'll also add two additional attributes that are **event handlers**, that is, they respond to the events of the user moving the mouse over the image element (a MouseOver event) or moving the mouse away from the image element (a MouseOut event). For either of those two events, we want to call the *imageSwitch()* function. Figure 10.7 shows the new image tag for the page.

Figure 10.7The new HTML <img> (image) tag

Beyond HTML

The tags in HTML are, as we have seen, specified. The tag pair <p> </p>, for example, is used for marking the enclosed text as a paragraph. The writer of the HTML document cannot invent new tags. *XML* (e*X*tensible *M*arkup *L*anguage) is a newer markup language. It is a "metalanguage," that is, a markup language for markup languages. Using XML, the writer can create his or her own tags; an XML document is not about displaying information but about how to structure and interpret information to be displayed. An XML document usually also contains or refers to a schema that describes the data, and the body of the XML document can then be checked against the schema to be sure that it is a well-formed document. All modern browsers support mechanisms that translate XML documents into HTML documents for display. XML allows for flexible document interchange across the web; for example, in May 2003, the National Library of Medicine announced a "Tagset" for journal articles to provide a single format in which journal articles that originate from many different publishers and societies can be archived. XML-based file formats now form the basis for office productivity tools such as Microsoft Office, Apple iWork, and Google Docs; for example, the file extension .docx for a Word document indicates that it is using an XML file format, and similarly with .xlsx for Excel spreadsheets and .pptx for PowerPoint slides.

Some other XML standards are shown in the following table; all of these facilitate exchange, within a community of special interests, of documents or other data that might otherwise have differing vocabulary or formats.

Name	Data to be exchanged
BeerXML	Beer brewing data
LandXML	Data of interest to land developers, civil engineers, and surveyors
LegalXML	Documents between courts and attorneys, court transcripts, electronic contracts
RailML	Railway industry data
RecipeML	Recipe ingredients (facilitates automated conversions from one type of measurement to another)

The *imageSwitch()* function itself uses a variable called *nextPicName* that contains the image filename for the new picture, the one being switched to. The image element *src* attribute will be set to the value of this variable. The code is

This statement locates the appropriate image element by its id (even though in this case there is only one image element on the page) and assigns *nextPicName* to its *src* attribute.

The second task of the *imageSwitch()* function is to make sure that *nextPicName* is updated to the appropriate value in preparation for the next image switch. This is accomplished by an if/then/else statement, one of the basic algorithmic constructs we introduced in Chapter 2. The code is straightforward: If we just changed to the flower2.jpg image, then we should be ready next time to change to the flower1.jpg image, and vice versa.

We are almost done. The only remaining thing is to initialize the *nextPicName* variable. We'll use another JavaScript function called *startValue()* to do that task. Because the page loads with the *flower1.jpg* image showing, the *nextPicName* value (the new image value) should be *flower2.jpg*. In the HTML body, we'll use the "onload" event handler to invoke the *startValue()* function to get things started.

Figure 10.8 shows the complete HTML page with the embedded JavaScript.

Figure 10.8JavaScript embedded in an HTML page

PHP

Webpages that are designed using HTML are generally static, that is, their content looks the same each time the page is opened in your browser. However, when you visit your favorite online store, the content is different with each visit, reflecting, for example, items on sale or the newest products. These are dynamic webpages (their content changes) that are stored on the web server of the online merchant. Dynamic pages are often tied to a behind-the-scenes product database. If a new product becomes available or a price changes because of a sale, the change is made in one place in the underlying database. Whenever any dynamic page tied to the database is requested from the server, the latest database information is loaded into the page before it is sent back to your web browser, and your browser then displays it. The HTML for the various pages does not have to be constantly rewritten to incorporate the new data.

PHP (which originally stood for Personal Home Pages but now stands for PHP: Hypertext Preprocessor) is a *server-side scripting language*, that is, its programs run on the web server computing system rather than on the user's own machine. Like JavaScript, PHP is embedded within HTML code in webpages hosted on a server. PHP is particularly adept at making database connections for dynamic webpages. The PHP code, when executed, sets up a connection to the database and formats HTML code on the page to include the new data values. According to a March 2017 survey, over 82% of the websites whose server-side programming languages were known were using PHP.

Practice Problems

Describe the result of executing the following SQL query on the vendor database.

Answer

It results in the names of all vendors from Chicago.

Given the following HTML statement, what does the corresponding line of text on the webpage look like?

Answer

These are the *times* that try men’s souls

Type the HTML code of Figure 10.4 into a text editor such as Notepad. Save the file with an .html extension. Find a small .jpg image (it need not be sunflowers!) and store it in the same folder as the .html file. Then double-click the .html file to bring it up in your browser. Does it look like Figure 10.5?
Change font sizeMain content

10.3.4 R

R is a specialized programming language designed for statistical computing and graphics. The name “R" comes at least in part from the first names of the two authors of the language, Ross Ihaka and Robert Gentleman. Like Python, R is an open source language. A group called the R Core Team controls modifications to the core code, but many **code libraries** have been contributed by users. The first version of R was released in 2000; the latest version, R.3.3, was released in March 2017.*

Given a set of data, you can ask R to compute the maximum value, minimum value, mean, median, standard deviation, and other more sophisticated statistical functions. Many of these computations can be done in a spreadsheet, but having an R program allows multiple data sets to be run efficiently and repeatedly. R has become more popular with the rapid development of the field of “data science” (we’ll talk more about data science in Chapter 14).

Here is an example using R on a trivially small data set. A medical research team is concerned about medical conditions X and Y, and has gathered data from surveys at six different sites. The data from each site consists of the percentage of the survey population that exhibits condition X, condition Y, are smokers, are overweight, or have low income levels. This data was entered into a spreadsheet and then saved as a .csv (comma-delimited) file. One of the advantages of R over many other languages is the ease with which tabular data in such a file can be loaded into memory and then displayed. Only two simple commands are required. Here, boldface indicates what the user types:

R responds immediately with the resulting display of data:

Another one-word command produces a lot of statistical results:

Here we see the *minimum* and *maximum* values for each of the five attributes of interest over the six sites. These results also show the *mean* (often called the average), and the *median* (roughly half of the six data values are above the median value and the other half are below if the six data values were sorted in order). The 1st Quartile value is a value such that roughly 25% of the sorted data values are below it and the 3rd Quartile value is a value such that roughly 75% of the sorted data values are below it, although these are quite meaningless on such small data samples.

The medical research team is interested in any correlation between these five attributes; *correlation* measures the extent to which two attributes increase or decrease more or less together. A correlation value can range between −1.0 and +1.0. Negative values mean that the two attributes being compared change in the opposite way—when one goes up, the other tends to go down, and vice versa; positive values mean that the two attributes change in the same way as each other—when one value goes up (or down), the other also tends to go up (or down).

The R command

produces the following result:

Obviously, X values correlate 100% with X values, and similarly for the other four attributes. Here it appears that conditions X and Y have no correlation (−0.34232660). However, condition Y has a moderate correlation (0.5264281) with being overweight, and both smoking and being overweight have a somewhat strong correlation with low income (0.67869945 and 0.66617525, respectively). Correlation does not mean cause, that is even if Y had a very high correlation with being overweight, that would not mean that condition Y is caused by being overweight or vice versa.

Finally, R makes it easy to visualize large data sets in many ways. The following two commands cause R to produce the colorful bar graph seen in Figure 10.9.

Figure 10.9R bar chart

Change font sizeMain content

Chapter Contents

10.4Alternative Programming Paradigms

Computer scientists are fond of the word *paradigm*. A **paradigm** is a model or mental framework for representing or thinking about something. The paradigm of procedural programming languages says that a sequence of detailed instructions is provided to the computer. Each instruction accesses or modifies the contents of a memory location. If the computer carries out these instructions one at a time, then the final result of all the memory cell manipulations is the solution to the problem at hand. This sounds suspiciously like our definition of an algorithm in Chapter 1 (“a well-ordered collection of unambiguous and effectively computable operations that when executed produces a result …”). In fact, programming in a procedural language consists of

- Planning the algorithm
- Capturing the “unambiguous and effectively computable operations” as program instructions

In a procedural programming language, then, we must pay attention to the details of exactly how the computer is going to accomplish the desired task in a step-by-step fashion. In object-oriented programming, the procedural paradigm still holds, but the step-by-step instructions may be split into multiple small sets that are encapsulated within classes.

In this section, we look at programming languages that use alternatives to the procedural approach—languages based on other paradigms. It is as though we have studied French, Spanish, and Italian (different but related languages) and are now about to embark on a study of Arabic, Japanese, or sign language—languages totally different in form, structure, and alphabet. Alternative paradigms for programming languages include viewing a program’s actions as:

- A combination of various transformations on items (functional programming)
- A series of logical deductions from known facts (logic programming)
- Multiple subtasks of the same problem being performed simultaneously by different processors (parallel programming)

We’ll look briefly at each of these alternative programming paradigms, focusing on the different conceptual views rather than on the details of language syntax. In short, this chapter won’t make you an expert programmer, or even a novice programmer, in any of these languages, but you’ll have a sense of some of the different approaches to programming languages that have been developed.

Main content

Chapter Contents

10.4.1 Functional Programming

Functional programming had its start with the design of the LISP (*LIS*t *P*rocessing) programming language by John McCarthy at MIT in 1958. This makes LISP second only to Fortran in longevity.

A **functional programming language** views every task in terms of (surprise!) functions. Unlike the more general usage of the word function in some procedural programming languages, *function* in this context means something like a mathematical function—a recipe for taking an argument (or possibly several arguments) and doing something with them to compute a single value. More formally, when the arguments are given values, the function transforms those values, according to some specified rule, into a corresponding resulting value. Different values for the arguments can produce different resulting values. The doubling function transforms the argument 3 into 6 because , and it transforms the argument 6 into 12 because . In the grand sense, we can think of a program as a function acting on input data (the arguments) and transforming them into the desired output.

In a functional programming language, certain functions, called *primitive functions* or just *primitives*, are defined as part of the language. Other functions can be defined and named by the programmer. We will look at examples using Scheme, a functional programming language that was derived from LISP in the late 1970s.

To define the doubling function using Scheme, we can say

The keyword *define* indicates that we are defining a new function. The function name and its list of arguments follow in parentheses. The function name is *double*, and *x* is its single argument. The definition says that when this function is invoked, it is to multiply ('*') the argument value *x* by 2. Having defined the function, we can now invoke it in a program by giving the function name, followed by a list of values for the arguments of the function. (For the *double* function, there is only one number in the list of argument values because there is only one argument.) Scheme responds immediately to a function invocation by displaying the result, so the following interaction occurs as the user invokes the *double* function with various argument values (boldface indicates what the user types).

Here's the definition of another function:

which says that the function named *square*, when invoked, is to multiply the single argument value by itself. Thus a dialog with Scheme could be

Functions, once defined, can be used in the definition of other functions. This can lead to nested tasks that must be performed. The function *polynomial*, defined by

is the function that we would write mathematically as . Using this function, the dialog could be

When the *polynomial* function is invoked with the argument 3, Scheme consults the function definition and sees that this is really

Thus, the polynomial function must invoke the *double* function, and it is to invoke that function with an argument value of (square 3). Therefore, the first thing to do is to invoke the *square* function with an argument value of 3. The result is . This 9 then gets used as the argument value for the double function, resulting in 18. The total computation is equivalent to .
Here we've defined one function (*polynomial*) in terms of another function (*double*) acting on the result of applying a third function (*square*). In functional programming languages, we can build complex combinations of functions that use the results of applying other functions, which use the results of applying still other functions, and so

on. In fact, functional programming languages are sometimes called **applicative languages** because of this property of repeatedly applying functions.

As the name LISP suggests, LISP processes lists of things and so does Scheme. The arguments to functions, then, are often lists. As a trivial case, “nothing” can be thought of as an empty list, which is called *nil*. We will use four primitive list-processing functions available in Scheme, namely,

list
car
cdr
null?

The first function is called *list*. This function can have any number of arguments, and its action is to create a list out of those arguments. Therefore,

(list 3 4 5)

evaluates to the list 3, 4, 5, which we write as

```
(3 4 5)
```

Two other list-processing functions are called *car* (pronounced as when it means an automobile) and *cdr* (pronounced “could-er”). (The names have historical significance from the distant past. Car stands for “Contents of Address Register,” and cdr stands for “Contents of Decrement Register.”

These registers were part of the architecture of the IBM 704 computer on which LISP was originally implemented.) The *car* function takes a nonempty list as its argument and produces as a result the first element in that list. Therefore, a dialog with Scheme could consist of

The *cdr* function takes a nonempty list as its argument and produces as a result the list that remains after the first element has been removed. Therefore,

evaluates to the list

As a special case, when the *cdr* function is applied to an argument consisting of a one-element list, the empty list is produced as the result. Thus,

evaluates to the list *nil*. Note that the *car* function applied to a list evaluates to a single list element, whereas the *cdr* function applied to a list evaluates to another, shorter list.

The final primitive list-processing function is *null*?, which has a single list as its argument and evaluates to true if the list is *nil* (empty) and to false if the list is nonempty. Armed with these primitives, we can at last write a little Scheme program (Figure 10.10) to add numbers.

Figure 10.10 Scheme program to add numbers

Dialog with the program in Figure 10.10 could result in

Let's see how this works. Our function *adder* was defined to have one argument, symbolically denoted in the definition by *input-list*. Now we're invoking this function where the argument has the value of (*list* 3 4 5); that is to say, the function is to operate on (3 4 5). The *cond* function (short for "conditional") is acting like an if-else statement: It is equivalent to

The condition "null? input-list" is evaluated and found to be false because *input-list* at this point is (*list* 3 4 5) or (3 4 5). The else clause is executed, and it says to add two quantities. The first of these two quantities is (*car input-list*), which is (*car* (*list* 3 4 5)), or 3. Thus, 3 is to be added to the second quantity. The second quantity is the result of invoking the *adder* function on the argument (*cdr input-list*), which is (*cdr* (*list* 3 4 5)), or (4 5). The value, as constructed so far, is therefore

Now the program invokes the *adder* function again, this time with an argument of (*list* 4 5) instead of (*list* 3 4 5). Once again we test whether this list is *nil* (it isn't), so we add together

or

The *adder* function is invoked again with an argument of (*list* 5). The list still is not *nil*, so we add together

or

A final invocation of the *adder* function, this time with the *nil* list as its argument, takes the other branch of the *cond* statement, which results in 0. Altogether, then, we've done

or

The definition of the *adder* function involves the *adder* function again, this time acting on a shorter list. Note in our example how we invoke the *adder* function repeatedly—first on (3 4 5), then on (4 5), next on (5), and finally on *nil*. Something that is defined in terms of “smaller versions” of itself is said to be **recursive**, so the *adder* function is a recursive function.

Recursion is one of the features of functional languages that makes possible short and elegant solutions to many problems. Although recursion is a dominant mode of operation in functional languages, many procedural languages also support recursion, so that’s not the major argument for using a functional language. Then what is the benefit of going to a functional language?

A functional language allows for clarity of thought; data values are transformed by flowing, as it were, through a stream of mathematical functions. The programmer has no concern about where intermediate values are stored, nor indeed about how a “list” could occupy many memory cells. Another layer of abstraction has been offered to the programmer—the rarefied layer of pure mathematics. Because functions are described in a mathematical way by what they do to an item of data rather than by how they modify memory cells in the process of doing it, the possibility of side effects is eliminated. A **side effect** occurs when a function, in the course of acting on its argument values to produce a result value, also changes other values that it has no business changing. Implementing a function in a procedural language, where the major mode of operation is modification of memory cells, opens the door to potential side effects.

It’s All in How You Look, Look, Look, … at It

We used recursion to define the function to add a list, as follows: Add the first list element to the result of adding the rest of the list elements together. The recursive way of thinking takes a bit of getting used to. For example,

Reading a book can be defined as reading the first page followed by reading the rest of the book. Climbing a ladder can be defined as climbing the first rung followed by climbing the rest of the ladder. Having learned to program in a procedural language, some people are initially uncomfortable with the recursive style of functional languages. It might seem as if mysterious things are going on in the background and suddenly there’s a result.

Consider the following scenario. You are standing in a long line at the grocery store, and you’d like to know exactly how many people are ahead of you.

Solution #1: You step out of line to get a better view and count the number of people. Your algorithm is something like:

This is an iterative (looping) solution.

Solution #2: You ask the person in line ahead of you how many people are in line ahead of him. When he responds, you add 1 to his count, and that’s the number of people ahead of you. You have to have faith that all the people in line ahead of you know how to do the same process, and that when the person ahead of you finally responds, he’ll be giving you the correct answer for how many people are ahead of him. This is a recursive solution. Here’s how it works in more detail.

You ask Jose how many people are ahead of him. He tells you to wait, and he asks Anna, “How many people are ahead of you?” Anna asks Jose to wait, and she asks Tom, “How many people are ahead of you?" Tom can answer this question, so he tells Anna, "There are 0 people ahead of me." Anna adds 1 (to account for Tom), then tells Jose, "There is 1 person ahead of me." Jose adds 1 to this count (to account for Anna), then tells you, "There are 2 people ahead of me." Finally receiving this news from Jose, you add one (to account for Jose), and then you know there are 3 people ahead of you. You just ask the correct question, wait for a bit, and then there's your answer (unless you are the special base case of Tom, who actually has to provide the first answer).

Practice Problems

To what does each of the following evaluate?

(cdr (list 1 2 3 4))

Answer

(car (cdr (list 4 5 6)))

Answer

Define a function in Scheme that adds 3 to a number.

Answer

Main content

Chapter Contents

10.4.2 Logic Programming

Functional programming gets away from explicitly instructing the computer about the details of each step to be performed; instead, it specifies various transformations of data and then allows combinations of transformations to be performed. **Logic programming languages** go a step further toward not specifying exactly how a task is to be done. In logic programming, various facts are asserted to be true, and on the basis of these facts, a logic program can infer or deduce other facts. When a *query* (a question) is posed to the program, it begins with the storehouse of facts and attempts to apply logical deductions, in as efficient a manner as possible, to answer the query. Logic programming languages are sometimes called **declarative programming languages** (as opposed to procedural languages) because their programs, instead of issuing step-by-step procedural commands, make declarations or assertions that various facts are true.

A logic program relates to a domain of interest in which the declarations make sense (such as medicine, literature, or chemistry), and the queries are related to that domain. Logic programming has been used to write expert systems. In an *expert system* about a particular domain, a human "expert" in that domain contributes facts based on his or her knowledge and experience. A logic program using these facts as its declarations can then make inferences that are close to those the human expert would make.

The best-known logic programming language is Prolog, which was developed in France at the University of Marseilles in 1972 by a group headed by A. Colmerauer. Prolog stands for *PRO*gramming in *LOG*ic. Prolog programs consist of *facts* and *rules*. A **Prolog fact** expresses a property about a single object or a relationship among several objects.

For example, let's write a Prolog program in the domain of American history. We are interested in which U.S. presidents were in office when certain events occurred and in the chronology of those presidents' terms in office. Here is a short list of facts (declarations):

The interpretation of these facts is fairly obvious. For example, the declaration

asserts or declares that Jefferson was the U.S. president during the Lewis and Clark expedition. And

asserts that Kennedy was president before Nixon. (There are a number of versions of Prolog available; the version we use requires that identifiers for specific items begin with lowercase letters and have no internal blanks or underscores.)

This list of facts constitutes a Prolog program. We interact with the program by posing queries; this is the way Prolog programs are executed. For example, the user could make the following query (boldface indicates what the user types):

which represents the question "Was Lincoln president before FDR?" Prolog responds

because "before(lincoln, fdr)" is a fact contained in the program.

Here's some further dialogue with Prolog using this same program.

The first query corresponds to a declaration in the program, and the second does not. The "false" response does not signify that the statement is indeed false, only that its truth value cannot be confirmed because it is not part of the collection of facts in the program.

More complicated queries can be phrased. A query of the form "A, B" is asking Prolog whether fact A and fact B are both in the program. Thus, a query such as

produces a "true" response because both facts are in the program. The interpretation is that Lincoln was president during the Civil War and that Lincoln was president before FDR.

So far, Prolog appears to be little more than some sort of retrieval system that does lookups on a table of facts. But Prolog can do much more. Variables can be used within queries, and this is what gives Prolog its power. Variables must begin with uppercase letters. The query is asking for a match against facts in the program of the form

In other words, X can stand for anything that is in the “president” relation with Lincoln. The responses are

because both

are facts in the program.

Let’s describe what it means for one president to precede another in office. It may appear that the *before* relation already takes care of this. Certainly if “before(X, Y)” is true, then President X precedes President Y. However, in our sample program,

are both true, but that does not tell us that Lincoln precedes Kennedy (which is also true). Of course, we could add another *before* fact to cover this case, but that is an *ad hoc* patch. Instead, let’s add further declarations to the program to define the *precedes* relation. We already know that two presidents in the *before* relation should also be in a *precedes* relation. Furthermore, from the previous example, it would appear that if X is before Z and Z is before Y, then “precedes(X, Y)” should also be true. But we can say more than that: if X is before Z and Z precedes Y, then “precedes(X, Y)” should be true. This extension means that Jefferson precedes Kennedy because

and

Using this reasoning, we have derived three new “precedes” facts that were not in the original list of facts.

Thus, we want to say that there are two ways in which X can precede Y:

We can make declarations in our Prolog program that express the *precedes* relation, but this time the declarations are stated as rules rather than as facts. A **Prolog rule** is a declaration of an “if A then B” form, which means that if A is true (A is a fact), then B is also true (B is a fact). The actual Prolog declarations follow; think of the notation B :- A as meaning “if A then B.”

The rule for *precedes* includes *precedes* as part of its definition; it is therefore a recursive rule.

Our Prolog program now consists of the facts and rules shown in Figure 10.11.

Figure 10.11A Prolog program

Here's some further dialogue, using the new program. Be sure you understand why each query receives the response or responses it does.

Let's add one final declaration to the program—a declaration that says that event X occurred earlier than event Y if X took place during President R's term in office, Y took place during President S's term in office, and President R precedes President S. (Do you agree with this definition of the *earlier* relation?) Here's the rule:

Then a final query of

produces the responses

In this simple example, it is easy to check that the responses to our queries are correct, and it is also not difficult to do the necessary comparisons with the program declarations to see how Prolog was able to arrive at its responses. The interesting thing to note, however, is that the program consists solely of declaratives (facts and rules), not instructions about what steps to take in order to produce the answers. The program provides the raw material, and in the logic programming paradigm, this raw material is inspected more or less out of our sight, and without our detailed instructions, to deduce the answers to a query. Figure 10.12 illustrates the situation. The programmer builds a **knowledge base** of facts and rules about a certain domain of interest; this knowledge base constitutes the program. Interaction with the program takes place by posing queries—sometimes rather complex queries—to an **inference engine** (also called a *query interpreter*). The inference engine is a piece of software that is supplied as part of the language itself; that is, it is part of the compiler or interpreter, not something the programmer has to write. The inference engine can access the knowledge base, and it contains its own rules of deductive reasoning based on symbolic logic. For example, a Prolog inference engine processing the program in Figure 10.11 would conclude that

Figure 10.12The logic programming paradigm

is true from the rule of the form "if before(X, Y) then precedes(X, Y)",

together with the fact

because it is a rule of deductive reasoning (known as **modus ponens**) that “if A then B” together with “A” must result in “B.” The programmer need not supply this rule or instruct the inference engine when it should be applied. Thus, the inference engine can be thought of as providing still another layer of abstraction between the programmer and the machine. The programmer supplies the fundamental facts and rules about the domain but does not direct the computer’s step-by-step processing of those facts and rules to answer a query.

Practice Problems

Using the Prolog program of Figure 10.11, what is the result of each of the following queries?

?before(jefferson, kennedy).

Answer

before(jefferson, kennedy).

false

?president(X, lewisandclark).

Answer

president(X, lewisandclark).

?precedes(jefferson, X).

Answer

precedes(jefferson, X).

This is a somewhat idealistic view of logic programming; in actuality, the idiosyncrasies of Prolog compilers mean that programmers do need to understand something about the order in which rules of logic will be applied. Yet, Prolog still gives us a good sense of the logic programming paradigm, where the intent is to concentrate on the “what” [is true] rather than on the “how” [to find it] that is the hallmark of procedural programming. You can experiment with Prolog at the following website: http://swish.swi-prolog.org.*

Main content

Chapter Contents

10.4.3 Parallel Programming

The complex scientific problems of the 21st century, such as climate change prediction, drug design and development, water sustainability, understanding of biological systems, greenhouse gas management, and many more are classified as “Grand Challenges.” These are defined as “fundamental problems in science or engineering, with broad applications, whose solution would be enabled by the application of the high performance computing resources that could become available in the near future.” * Solutions to Grand Challenge problems may require new algorithms, new data management and analysis techniques, new computational models, and increased technical communication and cooperation among diverse communities. But they will also require heavy-duty computational power. As noted in Chapter 5, problems such as these are testing the limits of the Von Neumann model of sequential processing. Parallel processing offers the promise of providing the computational speed required to solve such important large-scale problems.

Parallel processing is really a catchall term for a variety of approaches to computing architectures and algorithm design. Let's review the MIMD (*multiple instruction stream/multiple data stream*) model of parallel architecture introduced in Chapter 5: Interconnected processors independently execute their own program on their own data, communicating as needed with other processors. The MIMD model includes a number of different structures, such as **multicore computing**, in which multiple processors are packaged together on a single integrated circuit, and **cluster computing**, in which independent systems such as mainframes, desktops, or laptops are interconnected by a local area network (LAN) such as the Ethernet or a wide area network (WAN) such as the Internet.

The algorithms with which we are most familiar, like those introduced in Chapters 2 and 3, operate sequentially because they were originally designed for Von Neumann-type execution. To reap the full benefit of a parallel architecture, we need to develop totally new algorithms that exploit this collection of processing resources. After all, it does not do any good to have 100 people available to help with a project if only one is doing any useful work, while the other 99 sit idle. (In contrast to other sections of this chapter, this one does not describe a specific parallel programming language. Instead, we introduce and discuss some general principles of parallel languages and algorithms.)

Chapter 5 suggested a parallel processing solution using 1,000 processors for our old problem of locating a single telephone number in a reverse telephone directory of 350,000,000 entries. Let's look at how a parallel programming language might manage these computing resources. We now assume that we have 1,001 independent processors to assist with this task. We designate one processor, say ID number 1001, to handle input/output while the remaining ones, those with ID numbers 1 to 1000, are assigned to the search task. The job of the input/output processor is to input the 350,000,000-element reverse phone directory, partition it into 1,000 separate chunks each of size 350,000, and send these chunks to the 1,000 search processors along with the NUMBER we are looking for. After distributing this data, the input/output processor waits for one of the search processors to find the designated phone number (and the corresponding name) and send the name back. It then prints this result (or, more likely, speaks or displays it) and terminates. This MIMD data allocation strategy is diagrammed in Figure 10.13.

Figure 10.13Model of MIMD processing

Now, in parallel, the 1,000 search processors execute the sequential search algorithm (here called SEQSEARCH) on their chunk of data, called *YOURLIST*, to see if *NUMBER* is contained in this segment. However, they do not have to do this in instruction-by-instruction lockstep; instead, each processor executes independently. Here is the outline of the program that each of the 1,000 search processors will run:

Each search processor initially waits for a message from the input/output processor (ID 1001) that contains *YOURLIST* (its particular sublist), and *NUMBER* (the number for which we are searching). This is achieved via the RECEIVE instruction. When the message arrives, the processor executes the sequential search algorithm to determine if *NUMBER* is located within its 350,000-element sublist. If *NUMBER* is located within that list, then *SEQSEARCH* will exit with *FOUND* set to true, and that processor will SEND the corresponding *NAME* to processor 1001. If *NUMBER* is not found, the variable *FOUND* will remain false, and that processor will "Do nothing" and halt. The SEND/RECEIVE commands used to exchange information are called *message-passing primitives*, and they are a very important part of MIMD programming languages.

To complete this solution, we need a second program, the one executed by the input/output processor. Its job is easy to describe—distribute data to all 1,000 processors and wait for a result to arrive from whatever processor finds the answer. This program might look like the following:

Note that not every processor uses the same program. In this case, there are two programs, one for the 1,000 search processors and one for the input/ output processor. Furthermore, even though 1,000 processors are executing the same program, they are not all executing the same sequence of instructions. For example, if *NUMBER* occurs exactly one time in the reverse phone directory, then 999 processors will execute the ELSE clause and "do nothing." The one processor that does find *NUMBER* will SEND the corresponding name to processor 1001. Also note that every processor needed access to *NUMBER*, but there is no global shared memory. Instead, processor 1001 explicitly SENDs this value to every other processor, using message-passing primitives.

This reverse telephone directory search is a rather simplified example of MIMD parallelism for three reasons. First, there were only two distinct programs, and 1,000 of the 1,001 processors were executing the same one. In many MIMD algorithms, there are many more distinct programs. The situation here is equivalent to having 1,001 people building a house and having 1,000 of those 1,001 doing the exact same task. In most cases, there will be carpenters, roofers, plumbers, masons, and so forth, all performing their own specific tasks.

The second reason why this is a simplified example is that there is little communication between processors. In this algorithm, processors receive data at the start of the program and (possibly) send a result at the end. There is no communication during the computation itself. However, in most MIMD algorithms, there is message passing going on throughout the computation for such purposes as sharing intermediate computations, exchanging temporary data, and providing status information. For example, in the homebuilding analogy mentioned previously, the people putting up the walls must communicate their status (progress) to the roofers, who are waiting for wall construction to finish before they can begin. This example is simple for a third reason—it does not deal with the possibility that *NUMBER* is not in the phone book. If that occurs, all 1,000 search processors will execute the ELSE clause and "Do nothing." The input/output processor will be sitting and waiting to RECEIVE a result that never will be SENT. (We ask you to think about how to solve this problem in an upcoming Practice Exercise.)

An example of more sophisticated MIMD parallel processing is the **divide-and-conquer model**. In this approach, the problem is successively partitioned into smaller and smaller parts and sent off to other processors, until each one has only a trivial job to perform. Each processor then completes that trivial operation and returns its result to the processor that sent it the task. These processors in turn do a little work and give the results back to the processors that gave them the tasks, and so on, all the way back to the originating processor. In this model, there is far more communication between processors.

For example, the task of finding the largest number in a list can be solved in a MIMD parallel fashion using the divide-and-conquer model. (The sequential version of this algorithm was presented in Chapter 2.) The original list of numbers is assigned to the top-level processor, which partitions the list into two parts and sends each half to a different processor. Each of these two processors divides its list in half and hands it off to yet two other processors, and so on, creating the pyramid effect shown in Figure 10.14.

Figure 10.14The divide-and-conquer approach using multiple processors

At the bottom of the pyramid is a collection of processors that only have to find the largest number in a one-element list, a trivial task. They each pass this result up to their “parent” processor, which selects the larger of the two numbers it receives and passes that value up to its parent. All the way up the pyramid, each processor has only to select the larger of the two numbers it receives from its “children.” When the processor at the top of the pyramid completes this task, the problem of finding the overall largest number has been solved.

Using a single processor, finding the largest of N numbers takes $\Theta(\mathrm{N})$ time because each of the N numbers in the list must be examined exactly once. (This order of complexity was introduced and discussed in Chapter 3.) However, the parallel approach diagrammed in Figure 10.14 traverses the pyramid from top to bottom and then back to the top. Because the N numbers are divided into two halves at each step, until the lists are of length one, this down-and-up-the-pyramid process requires ($2 * \lg N$) steps, and a parallel solution to the “Find Largest” algorithm is $\Theta(\lg N)$. (Logarithmic efficiency was discussed in Section 3.4.2; recall that lg N means .) This can lead to enormous speedup in the solution time because the function $\lg N$ grows at a much slower rate than N. For example, if , then using a sequential approach to finding the largest number takes on the order of 1,000,000 steps, whereas our parallel solution needs only on the order of steps, a potential speedup of 25,000! (Of course, it also needs a whole lot of processors!)
We would expect the use of parallelism to reduce processing time because subtasks are being executed concurrently. However, one potential roadblock to achieving these higher levels of speedup is the amount of communication traffic between processors, both to distribute code and data and to share status and results. At some point, an increase in the number of processors can become more of a hindrance than a help, due to the extra data communication required. This is analogous to having too many people serve on a committee. The work involved in keeping everyone informed can slow down rather than speed up the work. In that case, it could actually be more efficient to have fewer people working on the task. One of the most important areas of research in parallel processing is the design and development of efficient parallel algorithms that

keep processors busy, minimize communications, and significantly speed up the overall execution time.

New Dogs, New Tricks

Section 9.1 described the evolution of sequential programming languages from machine language to assembly language to high-level languages like C++, Java, and Python. A similar evolution is happening with parallel programming languages. The early parallel languages required programmers to personally manage many aspects of parallelism—allocating data to local memory units, distributing programs, and sending and receiving messages. That is not unlike low-level assembly languages that required programmers to format data and manage memory, tasks that humans do not do very well. However, just as assembly languages evolved into high-level languages in which compilers perform these mundane tasks, the field of parallel programming languages is also evolving. A language called ParaSail (Parallel Specification and Implementation Language), whose design began in 2009, is targeted toward multicore chips. Instead of trying to add parallel programming features to an existing non-parallel programming language, ParaSail began as an inherently parallel language in which most of the parallel tasks are carried out automatically and are transparent to the programmer. That is, execution proceeds in parallel by default, and the programmer actually has to work hard to force sequential processing. In addition, ParaSail rules out constructs from other languages that make parallel processing potentially unsafe. The latest release of ParaSail was in November 2016.

Practice Problems

Explain how parallel processing can be used to evaluate the expression

If each addition operation takes one “time slot,” what savings can be achieved by using parallel processing instead of sequential processing?

Answer

One processor could compute A + B while another computes C + D. A third processor could then take the two quantities A + B and C + D and compute their sum. Parallel processing uses a total of two time slots: one to simultaneously do the two additions A + B and C + D, then one to do the addition (A + B) + (C + D). Sequential processing would require a total of three time slots: (A + B), then (A + B) + C, then ((A + B) + C) + D.

Explain how you could get the parallel reverse telephone directory lookup problem to work correctly even if the desired NUMBER is not in the directory.

Answer

One solution would be to have the input/output processor set a timer for the maximum possible response time from each of the search processors and if that time is exceeded, it reports that NUMBER was not found. This is not foolproof, because it could simply be that the communications link between the input/output processor and the one processor that found NUMBER is broken, so it was unable to report back the corresponding NAME. A better solution would be to have each search processor report back to the input/output processor when it cannot find NUMBER; when the input/output processor has received that message from all 1,000 search processors, it would then report that NUMBER was not found.

Main content

Chapter Contents

10.5 New Languages Keep Coming

After our discussion of many programming languages that fall into several distinct language categories, you might think that surely all programming needs have been met with existing languages. But new languages and new paradigms continue to be developed, perhaps in the hope of finding the "silver bullet" of an easy-to-use, efficient, general-purpose computing language that enforces good programming practices to allow quick production of error-free code on all computing platforms.

Here are three more programming languages that are relatively new. It remains to be seen whether they will stand the test of time and become widely used "standard" languages.

10.5.1 Go

Go, sometimes referred to as *golang*, is a programming language developed at Google. One of the people involved in its development was Ken Thompson, a recipient, along with Dennis Ritchie, of the 1983 A. M. Turing Award (see Chapter 12) for the development of UNIX.

What prompted the development of yet another programming language? Here is a quote from the FAQ (Frequently Asked Questions) page at http://golang.org/doc/faq:

Go was born out of frustration with existing languages and environments for systems programming. Programming had become too difficult and the choice of languages was partly to blame. One had to choose either efficient compilation, efficient execution, or ease of programming; all three were not available in the same mainstream language. Programmers who could were choosing ease over safety and efficiency by moving to dynamically typed languages such as Python and JavaScript rather than C++ or, to a lesser extent, Java.

Go is an attempt to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language. It also aims to be modern, with support for networked and multicore computing. Finally, it is intended to be fast: it should take at most a few seconds to build a large executable on a single computer. To meet these goals required addressing a number of linguistic issues ... a new language was called for.

Go is an open source language; the source code is available for anyone to examine and use, and changes or additions can be proposed. As noted in the quote, two of the target areas for Go applications are systems programming and programming for multicore machines. Although Go is a completely new language, it certainly borrows a lot from C/C++. As a result, you can probably understand exactly what the following Go program does:

Go was named by the TIOBE Index (see the Special Interest Box, "The 'Popularity' Contest," earlier in this chapter) as the Language of the Year in 2016, that is, the "programming language that gained the most popularity in a year." The latest version of Go at this writing is release 1.8; it was available as of February 2017.

Go is Going Places

Go supports many internal Google projects, but it has also been adopted by external users:

Dropbox is a company that offers cloud storage services. In July 2014, Dropbox announced it had migrated critical Python code to Go partly to improve speed of execution. The result was about 200,000 lines of Go code. Dropbox engineers found that they needed some new code libraries, which they ultimately released to the open source community for testing, use, and possible improvement by others.
Uber, an on-demand transportation company, operates in 580 cities worldwide. Uber has defined specific geographic areas, called "geofences." The first thing that happens when a client requests a car-and-driver via mobile phone is to locate the geofence from which the client is calling. Hundreds of thousands of geofence lookups are needed per second, and Uber requires that 99% of them be done in less than 100 milliseconds. In 2015, Uber wrote a new system to handle this task using the Go language in part because Go could provide high throughput (could handle this volume of tasks) with low latency (delay). As proof of its success, on New Years Eve in 2015, the new system handled a peak load of 170,000 queries per second with a 99% response time of less than 50 milliseconds.
Netflix, the well-known streaming video company, now serves customers in over 190 countries. Its computer systems must hold personalized data for its more than 81 million members, as well as manage member signup, browsing, and what Netflix calls the "playback experience." One of the concerns, as with Uber, is the amount of latency (delay) in processing client requests. In 2016 Netflix built a new system, written in Go, that acts as an intermediary between the client and the systems that access all the stored data. Go was chosen for lower latency than Java while handling tens of thousands of client connections, and was found to be more productive for developers than C.

Main content

Chapter Contents

10.5.2 Swift

In June 2014, Apple announced a new programming language at its annual Worldwide Developers Conference. The language is called *Swift*, and it was designed for building apps on the iOS and macOS operating systems, in others words, for the iPhone/iPad and the Mac. Apple developers had long been using Objective-C, an object-oriented version of the C language. Swift borrows much from Objective-C, and can work alongside Objective-C, but with faster performance. It also adds some modern programming constructs and attempts to make it harder to do "unsafe" things. This announcement was for the beta release only, meaning that the code was available to registered Apple developers but not yet to the general public. In July 2015, Lyft, an on-demand transportation company, announced that its mobile app for iOS platforms had been completely rewritten in Swift. In September 2015, Apple announced the first public Swift version, Swift 1.0. Originally a proprietary language, that is, the implementation of the language is controlled solely by the vendor (Apple in this case), Swift became open source software in December 2015, and now runs on Linux machines as well as all Apple products. While the rise in the use of Swift was initially driven by Apple apps, the ability to run Swift programs on low-cost Linux servers that support many web apps will increase its usage as a programming language.

By March 2017, less than two years after its first publicly released version, Swift had entered the "top 10" language group for the first time on the language popularity site www.tiobe.com/index.php/content/paperinfo/tpci/index.html. The latest version of Swift was released in September 2016.

Main content

Chapter Contents

10.5.3 Milk

In September 2016, researchers at MIT's Computer Science and Artificial Intelligence Laboratory announced a new programming language called Milk.* This language is geared to the "big data" that is of interest in many applications that attempt to detect connections and patterns between data points. Often, while the pool of data is huge, only selected points in the data are of interest at any one time; the technical term is that the data of interest is "sparse"—widely scattered throughout the large dataset.

In Chapter 5 we learned that fetching data from memory is a slow process compared to CPU actions, and that speedup is obtained by using *caching*. Caching relies on the *principle of locality* so that when an item of data is requested from main memory, that item plus items stored nearby are all read into the high-speed cache memory on the theory that such items are likely to be used again in the near future. Clearly caching based on this principle of locality is inefficient for sparse data.

Milk assumes the use of multicore processors, each with local cache memory. When a core processor is instructed to fetch a data item from main memory, it does not do so immediately. Instead, it adds the memory address of that item to a growing list in its local cache. When the lists get long enough, the cores pool their address lists, which the Milk compiler then reorganizes into new lists so that addresses "near" each other are grouped together. The new lists are redistributed to the core processors, each of which then accesses (and caches) the data on its particular new list. In that way, only needed data items are fetched from main memory. The three steps can overlap so that, for example, in a particular core a new address request list can be building while a previously distributed access list is being executed. To accomplish these effects, a few Milk directives have to be added within loops in programs that randomly request data items from a large dataset. The benefit appears to be that programs run 3-4 times faster.

As these last three example languages clearly demonstrate, even though the field of programming language design is now well over 50 years old, it is still a fertile area of creative research.

Main content

Chapter Contents

10.6 Conclusion

There is an entire spectrum of programming languages, each with its own features that make it more suitable for some types of applications than for others. A number of well-known languages (Fortran, COBOL, C, C++, Ada, Java, C#, Python) fall into the traditional, procedural paradigm. Procedural languages can be object-oriented, leading to a different program design perspective and the promise of software reuse. Some languages (such as SQL, HTML, JavaScript, and R) are designed as special-purpose tools. Still others rely on combinations of function evaluations (a functional language—Scheme), logical deductions from specified facts (a logic programming language—Prolog), or a parallel programming approach. And new languages continue to be developed. Figure 10.15 lists the languages we have discussed, along with other major languages. A few words about this figure are in order. It is hard to pinpoint a date for a programming language.

Figure 10.15

Some programming languages at a glance

Name	Date	Type
Fortran	1957	Procedural
COBOL	1960	Procedural
BASIC	1964	Procedural
Pascal	1971	Procedural
C	1974	Procedural
Ada	1979	Procedural/Parallel
Go	2009	Procedural/Concurrent
C++	1983	Object oriented
Visual Basic	1988	Object oriented
Python	1990	Object oriented
Java	1995	Object oriented
C#	2000	Object oriented
SQL	1986	Database queries
Perl	1987	Text extraction/reporting
HTML	1994	Hypertext authoring
R	2000	Statistics and graphing

Name	Date	Type
LISP	1958	Functional
Scheme	1975	Functional
Scala	2004	Functional
F#	2005	Functional
Prolog	1972	Logic
Datalog	1977	Logic
Fortran 2008	2008	Parallel
Chapel	2010	Parallel
ParaSail	2011	Parallel
Julia	2012	Parallel
Ruby	1995	Scripting language/object oriented
JavaScript	1996	Scripting language
VBScript	1996	Scripting language
PHP	1997	Server-side scripting language
JSP	1999	Server-side scripting language
ASP.NET	2002	Server-side scripting language

Should it be when the language was proposed or developed, when it was first commercially used, or when it became standardized? It is also sometimes hard to pigeonhole a language as to paradigm. Although we've tried to make clear distinctions in this chapter, many newer languages combine features drawn from several approaches, making them "multi-paradigm." Finally, your favorite language may have been omitted. (By all means, add it to the table.) At any rate, it is certain that the programming language world has been and continues to be a "Tower of Babel."

The trend in programming language design is to develop still higher levels of abstraction. This allows the human programmer to think in bigger pieces and in more novel or conceptual ways about solving the problem at hand. We would like eventually to be able to write programs that contain only the instruction "Solve the problem." Yet, we must remember that code written in any high-level programming language is still of no use to the computer trying to execute that code. No matter how abstract and powerful the language for front-end communication with the computer, the machine itself is still toiling away at the level of binary digits, absolute memory addresses, and machine language instructions. The services of an appropriate translator must be employed to take the code down into the machine language of that computer. The workings of a translator will be discussed in Chapter 11.