Curtis Roads

with John Strawn, Curtis Abbott, John Gordon, and Philip Greenspun

The Computer Music Tutorial

The MIT Press
Cambridge, Massachusetts
London, England

2 *Music Systems Programming*

Curtis Abbott

Programming is Problem-solving

Basic Elements of Programming Languages

Executing Programs
Flow Graphs and Structured Programming
Procedures
Assignment

Control Structures

Alternation
Repetition

Data Structures

Data Types
Type Declaration
Type-building Operations
Arrays
Records
Pointers and Their Discontents
Somewhat Abstract Data Types
Object-oriented Programming
Inheritance
Highly Abstract Data Types

Programming Language Themes

Conclusion

Programming is necessary in order to do anything really new in computer music. The issues raised by programming not only have practical significance in computer music but are of deep intellectual interest in their own right. Hence this chapter attempts to impart some perspective on programming—not only what it is, but why it is interesting. A single chapter on this vast topic is by necessity condensed, and all we can hope to do is to provide a glimpse of a large endeavor.

We begin by introducing the basic elements of mainstream programming languages, including control and data structures. Then we survey selected advanced themes, including functional programming, logic programming, and constraint programming—all with a view toward music systems programming.

Programming is Problem Solving

Fundamentally, programming is problem solving. Although many qualities of a program can be assessed, including practical aspects like speed and aesthetic concerns such as elegance, the most important test of a program is *correctness*—whether it solves the problem it was designed to solve. This criterion is not as simple as it may first appear. As the problems to which computers are applied become larger and more intertwined with our daily lives, it becomes increasingly difficult to state problems precisely and unambiguously. Without a precise and unambiguous statement of the problem, one cannot readily determine whether a program is correct. Also, when programs are very large or closely coupled to an unpredictable world, it becomes impossible to test them exhaustively.

Thus, programming is a problem-solving process in which the problems are often difficult to define exactly and completely. Indeed, the activities that dominate a creative programming task are often different than either nonprogrammers or programming theorists imagine—thinking about how to make vague problem statements more precise, exploring the consequences of different ways of doing this, discussing the best approach with others (such as potential users, although this happens all too rarely), and so on.

Music systems programming can have all the technical and intellectual challenges of programming generally. Composition problems are notoriously difficult to define precisely and completely, so satisfying one composer's needs may not lead to a universal solution. Sometimes it is better to provide a flexible toolkit that the user can play with than it is to attempt to

solve all aspects of a musical problem once and for all. Many musical tasks call for interactions with unpredictable components external to the computer system (such as transducers, synthesizers, or musicians), typically in real time. The demands of real-time performance put special pressure on the software to handle all situations in a timely manner.

Basic Elements of Programming Languages

The earliest attempts to program computers resembled the analog synthesizer studio of yore: users set switches and interconnected the computational units of the computer by means of patch cords (figure 2.1). Today, programs are written in programming languages. Many programming languages exist, and new ones are constantly being developed. Generally, a programming language is designed around a particular way of looking at programming. In this chapter we pay the most attention to the higher-level programming languages that are widely accepted. We do not deal with low-level, machine-oriented code such as assembly languages. Also, we limit ourselves for the most part to *sequential programming*. This means programming for computers that do only one thing at a time. *Parallel programs*, which run on computers that can perform many operations concurrently, are becoming increasingly important, but it is easiest to become familiar with programming by thinking about the sequential case.

We can divide the subject of programming into two broad areas. *Control* has to do with the actions performed by the computer and the order in which they occur, while *data* has to do with objects in the computer's memory on which these actions are performed. In structuring the discussion this way, we are acquiescing to the implicit assumption that computers consist of an active part called a *central processing unit* (often called *CPU* or *processor*) wherein control resides, and a passive part called a *memory* wherein data reside. This is a good assumption for today's computers, but as computers evolve, it may become less good.

Executing Programs

A computer obeys precise and detailed *machine instructions*. A single machine instruction accomplishes little, but when we string together hundreds or thousands of machine instructions we can accomplish useful work. In contrast to machine instructions, programs are written documents, usually

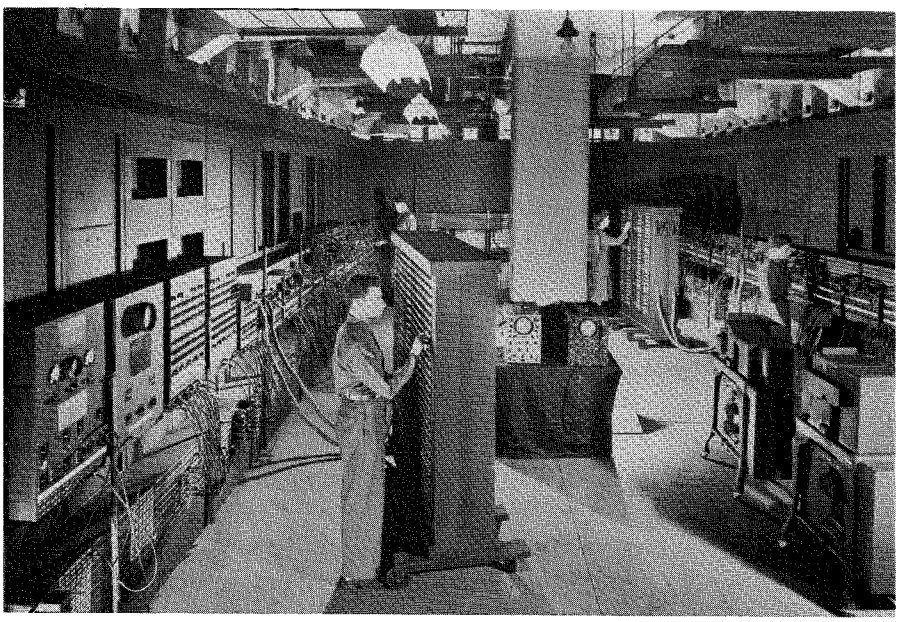

Figure 2.1 Corporal Irwin Goldstein programming the Eniac computer by setting function switches. The Eniac was operational in December 1945. This historic computer contained 18,000 vacuum tubes, 70,000 resistors, 10,000 capacitors, and 6,000 switches. It was 30 meters long, 3 meters high, and 1 meter deep. It is currently in the collection of the Smithsonian Museum, Washington. (Photograph courtesy of the Moore School of Electrical Engineering, University of Pennsylvania.)

written in high-level languages that are easier for human beings to read. Graphical programming languages, such as Max (Puckette and Zicarelli 1990), make the visual aspect even more explicit. Although the examples in this chapter are all textual, virtually every point we make could also apply to graphical languages. Before diving into the details of programming languages, we would do well to understand how high-level language programs are turned into thousands of machine instructions.

A common way to prepare a program for execution is to use a *translator*—a program that translates the program you edit into *executable form* or *object code* (an ordered group of machine instructions). This translator is called a *compiler*. Historically, another approach has been important—this is to use a program called an *interpreter*. An interpreter reads each statement in the program, performs a quick translation of the statement into machine instructions, and then hands the instructions to the computer to execute directly, before moving on to the next program statement. Such an interpreter is acting, in a way, as a different kind of computer, one that

executes a much more sophisticated program than any real, hardware computer. One important argument in favor of this approach has been that the translation process carried out by a compiler is relatively slow, since a compiler looks at the program more globally and performs many optimizations based on what it finds. When developing programs, it is important to minimize the time spent waiting to test changes. However, now that powerful computers are cheaper, this argument is less compelling, especially since programs executed by interpreters run much more slowly. (For some programming languages, there are still good arguments in favor of interpreters, but we do not have space to go into them here.)

Flow Graphs and Structured Programming

The executable form of a program is imperative: it contains the machine instructions that tell the processor what to do. However, not every instruction is executed each time the program is run because computers have *conditional instructions* that run a test to determine which instruction to execute next. *Conditional execution* of instructions is a key to the flexibility of computers.

Because instructions may or may not be executed when a program is run, it becomes important to have ways of thinking about patterns of conditional execution. Mathematicians have a convenient structure that they call *graphs*, which computer scientists have adapted as *control flow graphs*. They are drawn with boxes for the groups of instructions that are executed indivisibly, and arrows representing possible paths between them. These drawings are also called *flow charts*, and we will give some examples shortly. The pattern of instruction execution that these graphs represent is called the *flow of control* (or *control flow*) of a program. In this tutorial we introduce programming constructs that provide what is called *structured flow of control.*

The notion of structured flow of control is part of a more general area of *structured programming* (Dahl, Dijkstra, and Hoare 1972), which is an evolving set of rules and ideas for good programming style. Next we highlight various facets of structured programming and show how structured programming ideas apply to data as well as control.

Procedures

The most basic way of structuring a program is to divide it into a collection of *procedures* (also known as *functions, subroutines,* or sometimes *methods*). Procedures are ubiquitous in programming. A procedure is a program unit

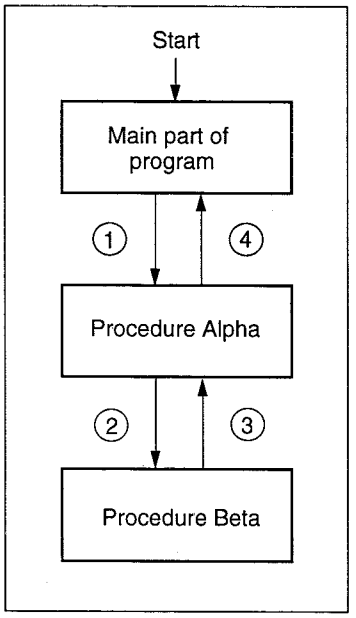

Figure 2.2 Program control flow in a nested procedure call. The main part of the program calls procedure *Alpha*, which in turn calls *Beta*. When *Beta* is done it returns control to *Alpha*, and when *Alpha* is done it returns control to the main program.

that can be *called* or *invoked*, like a mathematical function such as addition or multiplication. Procedures can be *nested*, meaning that one procedure calls another procedure before returning control to the original caller (figure 2.2).

A procedure can have *arguments* (again, like a mathematical function) that allow it to be a unit of activity with flexibility in different circumstances. This is because the arguments to a procedure are specified by the program part that calls it. Often a procedure also returns a *value*—a number or more complex data structure that the calling part can use. This is also analogous to mathematical functions—addition and multiplication generate values from their arguments.

Programming languages typically predefine a number of procedures that are the building blocks of larger programs. Some of these procedures reflect the underlying capabilities of the computer hardware, such as addition and multiplication of numbers. Others provide access to frequently used input/output facilities, such as the ability to store and retrieve files from disks or to display characters on a display. Procedures defined by programmers extend these basic capabilities.

Assignment

Another basic capability provided by most programming languages is *assignment*. Assignment is an operation that changes the value associated with a *variable*. In programs, variables are names for places in memory that hold values. These values can be interpreted in many different ways—as numbers, alphabetic characters, machine instructions, addresses of other memory locations, and so on. Hence, assignment changes the contents of the memory location corresponding to a variable.

In the C language, assignment is written with an equals sign as in

```
a=b;
```

This means the value in the memory location associated with *b* is copied to the memory location associated with *a*. This is very different from what the equals sign means to a mathematician. Some programming languages try to avoid this confusion by writing assignment with a variation of the equals sign. A common convention is to write

```
a:=b;
```

This is the syntax used in the Pascal language, for example. The control constructs in a language operate mostly on assignments and procedure calls, so we discuss them next.

Control Structures

Besides procedures and assignment, programming languages have operations that deal with flow of control. The simplest way that control can flow through a program is sequentially: each program statement is executed in order.

Alternation

Some control structures allow a program to define two or more *alternative* execution paths. When control reaches such a structure, one path is selected, based on a test, and control passes to the selected path.

Control structures based an alternation can be diagrammed in a flow graph as a structure that branches out and then comes back together again. Figure 2.3 shows several of these structures, using circles to denote tests

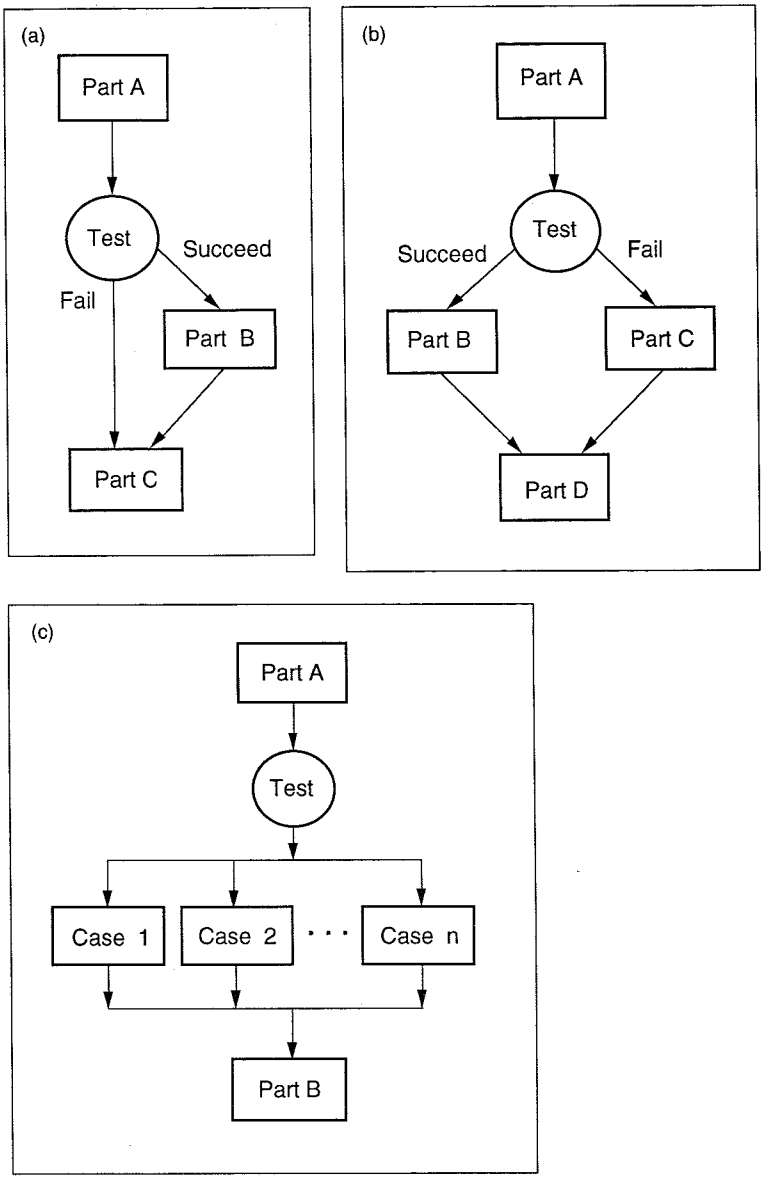

Figure 2.3 Program control flow constructs for alternation. (*a*) **if-then**. (*b*) **if-then-else**. (*c*) **case**.

(parts of the program that decide on the alternative to be executed) and square boxes to denote the possible alternatives.

The simplest case of alternation is when there is one section of a program that we may not want to execute. This is expressed with an **if-then** construct. Here is the format of this construct:

```
if <test> then <program part>
```

The ⟨*test*⟩ corresponds to the circle in figure 2.3a, which decides whether or not the ⟨*program part*⟩ is going to be executed.

A slightly more complicated instance of alternation is when one of two different program parts should be selected. This is shown in figure 2.3b. It is expressed in programming languages by extending **if-then** with **else**. The format of this expression is

```
if <test> then <part 1> else <part 2>
```

The general case for selecting one of several alternatives is handled in a variety of ways in existing programming languages. One way is by cascading **if-then-else** constructs as in the following schematic example:

```
if <first test> then <part 1>
else if <second test> then <part 2>
...
else if <last test> then <part n>
```

An interesting generalization of this is the *guarded command list* proposed by Dijkstra (1976). This is a list of tests and consequents that generalizes the cascaded **if-then-else** by specifying that if more than one of the tests succeeds, any one of the associated consequents can be executed. This is potentially useful because it may allow us to express our thinking about the underlying situation more clearly, without having to specify the order in which the tests are performed. Despite its elegance, the guarded command list has not become widely used.

Another commonly available construct is **case**. It is useful when a set of alternatives depends on the value of a single expression. The format of this construct is something like this:

```
case <expression>
     <first value>⇒<part 1>
     <second value>⇒<part 2>
...
     <last value>⇒<part n>
```

where the symbol ⇒ indicates that control is passed to the corresponding program part. This is essentially equivalent to the following **if-then-else** cascade:

```
if <expression>=<first value> then <part 1>
else if <expression>=<second value> then <part 2>
...
else if <expression>=<last value> then <part n>
```

The **case** construct is better not only because it makes the control structure clearer to the human reader but because the selection of the appropriate program part can often be made much faster by the computer.

Repetition

Another important pattern of control flow is the repetition of a program section over and over again, which is often called *looping*. Here it is necessary to specify how many times the repetition occurs and how the repetition terminates. A typical construct for repetition is the **while** construct, for which the format is

```
while <test> do <program part>
```

The effect of this is to evaluate the ⟨*test*⟩ before each execution of the ⟨*program part*⟩. Repetition terminates when the ⟨*test*⟩ fails. Figure 2.4 shows the flow graph of a **while** construct.

Programming languages have many constructs for repetition. Some constructs put the test after the program part (which guarantees that the program part is executed at least once), and some invert the sense of the test (which causes repetition until the test succeeds).

A more important variation on this theme are constructs that support the use of *auxiliary variables* associated with the repetition. Auxiliary variables are useful for enumerating the elements of collections of data, whether these collections are organized as matrices, lists, sets, or anything else. Here the possibilities for variation are enormous, and programming languages exhibit considerable diversity in the constructs that they provide for this kind of repetition. A canonical example is the **for** statement in Pascal (Jensen and Wirth 1974). The format of the **for** construct is as follows:

```
for <auxiliary variable> :=<initial value> to
  <final value> do <program part>
```

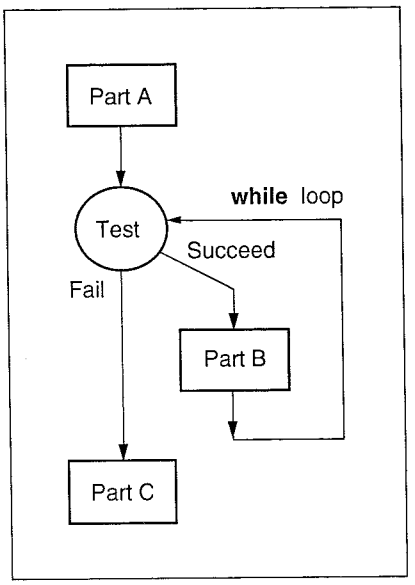

Figure 2.4 Flow graph of a **while** construct. The repetition of *Part B* continues while the tested condition is true. Otherwise the program goes on to *Part C*.

Here ⟨*auxiliary variable*⟩ is an integer that is newly created to go with the **for** statement. Its ⟨*initial value*⟩ is incremented after every repetition. Repetition continues until the variable attains the ⟨*final value*⟩.

The Pascal **for** construct has special restrictions that provide a good example of the constraints facing language designers. The restrictions are as follows:

1. The ⟨*auxiliary variable*⟩ cannot be changed except in the way shown—by incrementing before every repetition.
2. The ⟨*final value*⟩ is only evaluated once, before the repetition starts.

These two rules ensure that when a Pascal **for** loop begins, the number of repetitions is known. The advantage of this is that the **for** loop always terminates and can never run wild. The disadvantage is that, in some situations, it is inconvenient or impossible to determine in advance how many times the loop should be executed. (In this case, one must use the **while** construct in Pascal.)

It is interesting to compare the Pascal **for** construct with the analogous construct in C (Kernighan and Ritchie 1978). The format of the C **for** statement is as follows:

```
for (<initialize>; <test>; <increment>)
  <program part>
```

The use of this construct is similar to that of the **for** statement in Pascal. However, it is easier to understand the C **for** statement as a set of instructions for rewriting the components of the construct, because there are no restrictions as there are in Pascal. In particular, the ⟨*initialize part*⟩ is executed, then if the ⟨*test*⟩ succeeds, the ⟨*program part*⟩ is executed followed by the ⟨*increment part*⟩ and then back to the ⟨*test*⟩ again. This **for** loop can be viewed as a convenience, allowing programmers to write loops in a clear, concise way. By contrast, the Pascal **for** loop, by introducing restrictions and an auxiliary variable, is less general, but at the same time, it is better adapted to the situations it applies to.

These examples hardly exhaust the possibilities for repetitive constructs. Knuth (1974) contains a fascinating and accessible discussion of iterative constructs.

Data Structures

Inside any computer, data are represented in terms of *bits* (binary digits) of memory, which can take the values 1 or 0. These bits are organized into *bytes* and *words*. Bytes are (almost universally) 8-bit quantities that can represent many characters in Western alphabets. Hence, bytes are often used to represent alphanumeric characters.

Words are the "natural units" of operation on data in a computer, so the number of bits in a word varies from computer to computer. Today the most common computers use 32- or 64-bit words, but this was not true in the past and may not be true in the future.

It is important to understand the versatility of computer words. To illustrate, consider figure 2.5, which shows a 16-bit word and several interpretations of it. For our purposes, the specific interpretations are unimportant. Rather, we wish to illustrate that the interpretation of a bit pattern must be known if it is to have any meaning. This leads to our next concept, *data types*.

Data Types

The *type* of a data object (one or more computer words) refers how it is interpreted. Thus we say that the type of a data object x is integer, floating-point number, character, and so on.

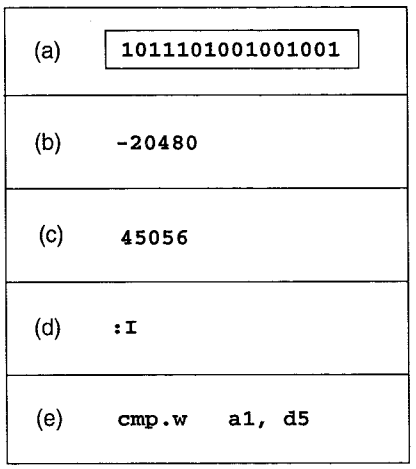

Figure 2.5 A 16-bit computer word and some possible interpretations of it. (*a*) Binary representation. (*b*) Two's complement integer. (*c*) Unsigned integer. (*d*) ASCII characters. (*e*) Machine instruction.

At the hardware level of "raw" computer memory and the machine operations that act on it, only a few types are available. Thus, if we were to stop a program and randomly look at a word in memory, it would be difficult to know for sure what that word represents. It could be an sample value, a small number of characters, a reference to another data object, a machine instruction, or any number of other things. One of the important jobs of a programming language is to present the programmer with an appropriate notion of type for the kind of problems the language is designed to solve. Usually this notion is somewhat different from that found at the machine level and is, as we say, "higher level." Thus, one of the jobs of the programming language *implementation* (that is, the compiler or other program that makes programs in a higher-level language execute on the machine) is to translate between the different notions of type provided by the two levels.

The question of just what notion of type is appropriate has always been controversial, and remains so. We have hinted that it depends on the kinds of problems the programming language is designed to be used for. This is generally acknowledged as a matter of principle, but it muddies the waters enough to make precise, technical discussions difficult. As we shall see, some of the most successful programming languages use a notion of type that is quite close to that provided by the vast majority of computers. On the other hand, much of the academic work that drives advances in programming language design has focused on more abstract notions of type.

Type Declaration

To make an already confusing subject a good deal more confusing still, there is controversy not only about what notion of type should be embodied in programming languages but also about whether these types should be visible in the written form of the program. In languages that make types visible, we say that variables and procedures are *declared*, along with their types. A declaration, in this context, is just a statement that a particular thing (variable, procedure, etc.) has a specific type. For example, we might say

```
integer v1;
```

to declare that the variable *v1* has type **integer**. There are two main arguments for explicit type declarations. One is that the translators can use the information in the declarations to catch programming errors and make the program execute more efficiently. The other argument is that declarations make programs easier for people to understand.

When the types of variables and procedures are not declared in a programming language, it is conventional to say that the language is *untyped.* This is extremely misleading! Any programming language embodies some notion of type, and its translator must deal with computer hardware that embodies a different notion. What is different about a language without required declarations is that the translator takes full responsibility for ensuring that the correct interpretation is given to all data according to its implicit notion of type. A common argument for "untyped" languages is that declaring types is too much work. This argument becomes less persuasive as programs become larger, longer-lived, and harder to understand.

The programming languages that are most commonly used in industry have type declarations. Several of the languages discussed in the last part of this chapter (Lisp, Smalltalk, Prolog) have no declarations, or optional declarations.

To summarize what we have said so far, words of computer memory represent data objects. These objects are characterized by their type, which determines how the computer interprets the data object. A programming language also provides a notion of type, which is sometimes quite different from the notion provided at the machine level. Controversy surrounds the issue of what notion of type is appropriate in programming languages, and to some extent, the answer depends on what applications the language is intended for. There is also some controversy about whether types should be declared, or whether such declarations should be optional.

Let us now consider some common properties of types in more detail. The simplest types are *atomic,* that is, not divisible into simpler types. Examples of atomic types are integers, floating-point numbers, and characters. In general, atomic types are directly supported at the machine level of the computer system, in the sense that the computer has hardware for operations on integers, floating-point numbers, and characters.

Type-building Operations

Programming languages generally provide ways of building up more complex types out of primitive ones. In examining these type-building operations, we will see some good examples of both abstract and concrete approaches to types.

Arrays

As a first example consider the *array*, which is a collection of items that can be selected by looking it up with an *index*. All the items in the array have the same type. If A is an array of type X, then we select an X by indexing A. The indexing operation is usually written $A[i]$, where i is the index. The indexed value can also be assigned to, thus the statement

```
A[i]=1023.99;
```

sets the ith element of the floating-point number array A to 1023.99. Arrays are usually implemented directly as blocks of memory in the underlying computer. The indexes provide a convenient way to access different parts of these memory blocks. As a result of the directness of the implementation, the type of the index value is restricted in most languages. In some languages, indexes are integers from 0 to some maximum, which is fixed when the array is initially declared. In other languages, one or another of these restrictions is removed, with corresponding extra complexity in the implementation.

The form of array declarations depends to some extent on what restrictions are in force in a given language. For example, in a language with integer indexes starting at 0 and a fixed size, only the size and the item type is required, so a statement like the following:

```
array[12] of pitch;
```

might declare an array of twelve items of type *pitch*, indexed by the integers from to 0 to 11. If we can specify both the lower and upper bound, the declaration looks like this:

```
array[19,108] of MIDI_pitch;
```

might declare an array of 88 equal-tempered pitches indexed by the integers from 19 to 108 according to the norms of the MIDI protocol. (See chapter 21 for more on MIDI.)

Arrays can also be *multidimensional*. This means that two or more indexes are required to identify an element of the array. A two-dimensional array can represent a *matrix*, which mathematically is a linear operator on a *vector space*. Since matrices and vector spaces are tremendously important in the scientific use of mathematics, arrays have always been important in scientific computation.

Arrays are common in computer music programs, since a one-dimensional array can represent the amplitude values of an audio waveform or envelope, while a two-dimensional array can represent the probabilities in a Markov chain for composition (see chapter 19) or the pitches of a *magic square* (showing a twelve-tone row in all its transpositions).

Records

Arrays provide one way of organizing a collection of similar kinds of data objects. Another type-building operation handles *records*, which allow programmers to organize heterogenous types of data objects. Each distinct kind of information within a record is called a *component* of the record. Sometimes components are called *slots* or *members* of the record.

Records are useful because they allow programmers to express logical associations, bringing together things that belong together. Properly used, they make programs easier to understand. For example, one might use a record to collect information about several aspects of a note: its pitch, register, duration, and so on. Records are declared as a list of names and types for the components:

```
record
      pitch : character_string;
      register : integer;
      duration : integer;
      ...
end
```

A collection of such records forms a *database*. Indeed, the notion of records was first made available in the Cobol programming language, which was designed for databases and related business applications. But records are much more widely useful, as we will see.

Pointers and Their Discontents

Array and record types are fundamental ways of organizing data, since they bring together collections of data objects into a single, more manageable object. Another fundamental notion is different. A *pointer* type represents a reference to a data object, where the reference is viewed as an object itself. Thus, if X is a type, the type **pointer to** X is the type of data objects that refer to Xs. Pointers have a *dereferencing* operation associated with them. Dereferencing a **pointer to** X gives an X, namely the one the pointer points to. The dereferencing operation is often written with an asterisk, so **ptr* dereferences the pointer *ptr* to produce the value it points to. Given a pointer to something, one can generally change its value with an assignment statement that uses the dereferencing operation. So pointers are like array elements in this way, and also a bit like variables.

When we say that a pointer *refers* to something else, we mean something quite simple—much simpler than reference in natural languages. Determining the reference in an English expression like "the one with the click" can be difficult to decide. In contrast, the notion of reference for pointers is simple. A pointer is implemented as a memory word whose contents are interpreted as the address of another memory word (namely the address where the pointed-to object resides). So the pointer refers to whatever is at the memory address that is the pointer's value.

Although the basic notion is simple, its ramifications are not. The idea of systematically interpreting a computer memory word as the address of another memory word is a fundamental programming technique. It is absolutely essential for doing anything useful or interesting with computers. The justification is a little subtle. Computer memories are addressed in a strictly linear fashion. (That is, there is a memory word whose address is 0, then a word whose address is 1, and so on.) Most problems do not fit into this linear straightjacket. The pointer technique (that is, the technique of interpreting memory words as addresses of other memory words) can be used to make a linearly addressed memory mimic any kind of structure conceivable. Learning in detail how to do this is one of the important aspects of learning how to program.

In order to see how pervasive this notion of interpreting values as addresses can be, let us go back to the array for a moment and think about it. Suppose we have an array A of integers (that is, of memory words), indexed by integers from 0 to 3999. This is like a memory array of 4000 words. Since it both contains and is indexed by integers, we can do the same trick with it as pointers do with the underlying computer memory. An expression like $A[A[12]]$ means to interpret the integer at $A[12]$ as an index (address) and

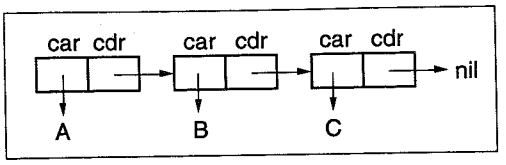

Figure 2.6 A Lisp list **(A B C)** represented as three cons cells. Each cons cell has two pointers. The car of the cons points to the value, while the cdr points to the next cons cell in the list. The final cdr points to **nil**; that is, the cdr contains zero. This indicates the end of the list.

return the value of the array at that index. Indeed, this trick with arrays is widespread in languages that do not have explicit pointer types.

The Lisp programming language does something a good deal more interesting. In so doing it provides a strong piece of evidence for the claim made previously that pointers can be used to map a linearly addressed array into any kind of structure at all. (Note: the following description is not strictly true of modern dialects of Lisp, but accurately reflects their general flavor.) Memory structures in the Lisp world are made up of *conses*—little objects that point to two other objects, which can be other conses or more basic things like numbers (figure 2.6). With conses, you can build lists and a myriad of other strange and wonderful structures. The Lisp world of conses is implemented in the underlying linear array of computer memory hardware.

Pointers are extremely powerful and useful, but with their power and utility come certain dangers. To see why, let us go back to that array again. Earlier we wrote $A[A[12]]$ to interpret the value of $A[12]$ as an index. But suppose the value of $A[12]$ is not a valid index (i.e., an integer between 0 and 3999)? Also, we are now intepreting elements of A in at least two ways, as numbers and as indexes. $A[12]$, at least, is an index, and the item it indexes is a number. There are many ways to make mistakes, many possible confusions.

Because of this danger, the explicit provision of pointers in programming languages has always been a controversial matter. As with other things, the right answer depends on what the programming language is for. The C language is among those in which pointers are provided in an unabashed way, but this is in keeping with its design as a *systems programming language*—one that hews fairly close to the underlying computer. The Lisp language does not have pointers at all, although they are ubiquitous in its implementation, partly because its design goals are so different.

An example of the use of pointers should help make this discussion more concrete. A canonical example is creating *linked lists*. A linked list is a

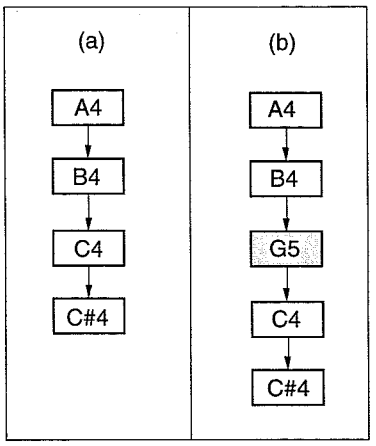

Figure 2.7 A linked list data structure of a melody, containing quoted strings representing pitch names. The last pointer value, to **nil**, is omitted in this figure. (*a*) Original melody. (*b*) New melody made by changing the pointer from "B4" and adding a new element "G5."

collection of records, each of the same type, where each component of each record is a pointer to the succeeding element in the linked list. Figure 2.6 showed a simple linked list as it is represented implicitly in Lisp. The following code is an explicit programming example:

```
linked_list_node record
        pitch : string;
        next_node : pointer to linked_list_node;
end
```

This record has two parts: a part called *pitch* that contains a quoted string, and a pointer called *next_node* that points to another *linked_list_node* record. Linked lists are often drawn as a simple box and arrow diagram, as in figure 2.7.

A more sophisticated example is a linked list record with two pointers. This models a common data structure called a *binary tree*. In a binary tree (figure 2.8), each node can have two branches:

```
linked_list_node record
          starting_time : integer;
          left_node : pointer to linked_list_node;
          right_node: pointer to linked_list_node;
end
```

A binary tree is a good representation to use when editing the performance schedule of a multiple-voice composition. Information organized as

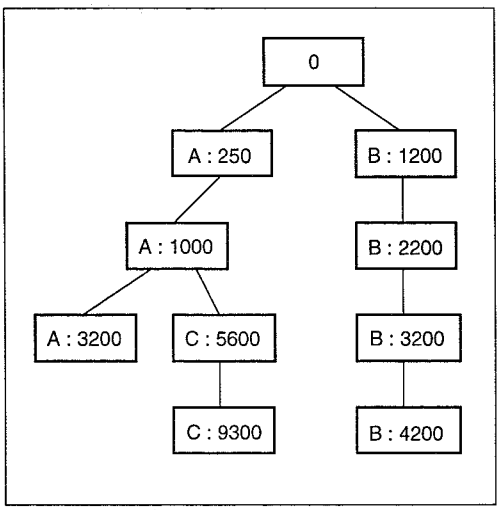

Figure 2.8 A linked list of nodes drawn as a binary tree graph, in which each node can have two subnodes. The characters in each node represent a performance schedule for three musical voices. The numbers in each node indicate the starting time in milliseconds of each event.

a binary tree can also be searched efficiently, so this data structure is a common one in computer science (Knuth 1973b).

These linked list examples bring up a number of issues. One of these is the way linked lists are used. Typically, linked lists are *traversed* by a program that follows the pointer in the subnode components until it reaches a special value that indicates that the end of the list has been reached. One of the things that makes linked lists useful is that it is easy to add or delete elements at any point in the list, by manipulating the value of the pointer components. This stands in contrast to arrays, where inserting a value in the middle of the array means that every value to the right of a new element must be shifted by one index value to the right to make room for the new element. Growing and shrinking arrays is time-consuming, whereas adding and deleting elements to a linked list is efficient.

Many programming environments provide standard procedures such as *insert_node* and *delete_node* for manipulating linked list data types. This idea of associating procedures with data types has been important in programming language research and has been adopted in different ways by two research communities, working respectively on *abstract data types* and *object-oriented programming*. We have more to say about both later.

Another issue raised by our linked list examples is the following. Suppose that, having defined the binary tree linked list, we now need another tree in which *starting_time* is not an integer, but a floating-point number. In most programming languages, this can only be done by defining a new data type

and giving it a different name. This is bad, for at least two reasons. For the human reader, it tends to obfuscate the common pattern of use of all these record types (namely, as linked lists). From a technical point of view, many of the procedures associated with these data types (such as the ones for adding and deleting elements, and counting the number of elements by traversing the list) are exactly the same across all linked list data types.

One experimental approach to clearing up this difficulty is with *polymorphic* data types. Roughly, a polymorphic type takes an argument which is itself a type. A list is a good candidate: we can have lists of integers, of floating-point numbers, and of infinitely many other types of things.

Somewhat Abstract Data Types

Early work in programming language design concentrated on control issues, for example, what sorts of alternation and looping constructs to provide. Since then it has become clear that the issues surrounding data are much more difficult but also more important. This is because data structures in a program model parts of a world (real or imagined). In view of the importance of data structuring, it should not be surprising that there are a number of different research strands concerned with it.

One basic insight is that a data type is defined both by data structure (records, arrays, pointers, and so on) and by the *operations* it permits. Usually it is only these privileged operations that have access to the data—others must obtain access via the operations. We say that the operations *encapsulate* the data, and this is an important theme with many benefits.

At least two distinct sets of researchers have developed languages with data types designed around procedures that encapsulate their data. One group started from otherwise traditional languages and added constructs for these so-called *abstract data types*. Examples of this work include CLU (Liskov et al. 1979) and Alphard (Wulf, London, and Shaw 1976).The most visible success (and perhaps the culmination) of this work is the language Ada, supported by the United States Government (Department of Defense 1980).

Object-oriented Programming

Another, more diverse group proposed to put even more emphasis on the data and developed an approach called *object-oriented programming*. They also developed new ways of talking about their approach to emphasize its radicalness. In particular, data elements are called *objects*. Data types are called *classes* (at least, most of them are, although more traditional data

types can also be present in object-oriented languages). The operations associated with a class are usually called *methods*. Calling a procedure is called *sending a message*. This difference is more than cosmetic. Whereas in traditional programming, all the arguments to a procedure have equal status, in object-oriented programming there is a privileged argument, which is the *receiver* of the message (the object to which it is sent).

Object-oriented programming was pioneered in the language Simula (Dahl and Nygaard 1966; Dahl, Dijkstra, and Hoare 1972), which was designed for programming discrete event simulations, in which time acts on a collection of simulated real-world objects. Although Simula was the first object-oriented programming language, Smalltalk (Goldberg and Robson 1983a,b) has probably been the most influential. Developed over a long period of time at the Xerox Palo Alto Research Center (PARC) and ParcPlace Systems, it has several essential properties that are synonymous with object-oriented programming. Smalltalk is *dynamically typed*, in the sense that variables can hold objects of any class, so that the appropriateness of a message must be checked when it is sent. In other words, Smalltalk is one of those languages sometimes misleadingly referred to as "untyped." Also, Smalltalk-80, the current version of Smalltalk, includes a superb programming environment and provides excellent support for developing graphical user interfaces for music (Krasner 1980; Pope 1991a,b; Scaletti 1989a,b; Scaletti and Hebel 1991).

Inheritance

One of the important contributions of object-oriented programming to data structuring is the idea of *class inheritance*. This means defining new types as variants of existing types (classes), rather than starting from scratch. Inheritance is especially appropriate when one class *specializes* another.

For example, suppose in a performance program we have a class that manages musical keyboards. We call this a **Keyboard** class. It has operations (methods) for translating a key number into a pitch, and for starting a note when a key is struck. Now suppose we need to hook up a *velocity-sensitive keyboard* (i.e., a keyboard that plays louder if you hit the keys harder and therefore faster). Since there are velocity data to be conveyed, we need to rewrite the note-starting operation, but the key-to-pitch translation operation need not change. We can use inheritance to make a **Velocity-Sensitive-Keyboard** class that inherits all the data parts of the **Keyboard** class, possibly adding new ones to deal with velocity information, and that inherits the translation operation while redefining the note-starting operation.

A less commonly available technique in object-oriented programming is *multiple inheritance*. This means that a class can inherit behavior (and data parts) from several other classes. This is an especially useful technique for "mixing in" functionality—adding a new feature to an existing class. For example, we might write a note-recording *mixin* that could be added to any class that has a note instantiation operation, and that would redefine the operation automatically so that its parameters are remembered in an auxiliary array of records each time it is called.

As typical programs grow larger and more complicated, it becomes more important to have facilities for organizing them, incrementally changing them, and so on. Such facilities are a large part of what object-oriented programming is about, and so as the complexity and size of problems increases, object-oriented programming has become increasingly visible and influential in the computing world. Earlier languages (especially Smalltalk) have been somewhat handicapped by slow execution speed, as well as the difficulty of learning them. Consequently, several languages have been developed that add object-oriented facilities to existing mainstream programming languages such as C or Pascal. An example of this trend is the C++ language (Soustrop 1991).

Highly Abstract Data Types

None of the approaches to abstract data types described so far is highly abstract. For example, if you define a class of sets, you must define exactly one implementation (i.e., collection of operations) for it. All sets use that implementation. However, there are a half dozen good ways to implement sets, each appropriate for sets of different sizes and domains. No programming language to date adequately addresses this problem. However, there is another research strand in which the implementation of a data type is given in a very abstract form, which can be viewed as leaving open the possibility for the underlying automatic system to decide how best to implement it. In such approaches, one merely specifies what properties the operations of the data type must satisfy, rather than saying exactly how to execute them. Most commonly, the specification is done with equations. The standard example, which appears hundreds of times in the research literature in this area, describes a *stack*. A stack is a data structure with three operations, **push**, **pop**, and **top.** The **push** operation stores an object on the stack; **pop** removes the most recently stored value, and **top** retrieves the most recently stored value.

These simple rules can be translated into equations as follows. Suppose S is a stack and x is a value. Just two equations define the stack's behavior, given some background assumptions:

pop(**push**(*S*, *x*))=*S*
pop(**push**(*S*, *x*))=*x*

The first equation says that **pop** undoes **push.** The second says that **push** has to remember the pushed value so that **top** can retrieve it.

The key idea behind this abstraction of abstract data types is to interpret the equational definitions as instructions that can effectively carry out the operations so defined. Thus, the equations, which make no reference to computer words and are not instructions in any obvious way, nevertheless become operationally effective. Indeed, we say this is an *operational interpretation* of the equations. From a programming point of view, what is important about this is that we have described the behavior of a stack in a very abstract way, with equations, rather than giving instructions for obtaining that behavior. This is an example of what is called *declarative programming,* discussed later in this chapter.

A good deal of mathematical theory stands behind operational interpretations of equations. The usual first step is to give a *preferred orientation* to the equations, so that they become *rewrite rules*, or in other words, *simplication rules* for expressions involving the given terms (**push**, **pop**, and **top** in our example). Note that in the example, the right-hand sides of both equations are much simpler than the left-hand sides. Thus, the preferred orientation is left to right. Often an orientation can be assigned, sometimes it can even be done automatically. Sometimes, however, it cannot be done at all, much less automatically. For example, a common algebraic law is commutativity. For the addition operator (+), it is written as the following equation:

$$x + y = y + x.$$

The intuitive interpretation is that the order of the arguments to the addition operator does not matter. This equation cannot be given a preferred orientation; the intuitive reason is that neither side is simpler than the other.

The automatic interpretation of very abstract specifications is a desirable goal, but one which is fraught with difficulties. These include many theoretical results showing that various problems are strictly unsolvable, in the sense defined by Gödel, Turing, and other mathematical pioneers. One of these is the problem of determining whether a preferred orientation can be automatically assigned to equations.

Besides this, the equational level of abstraction does not address the practical problem raised at the beginning of this section concerning abstract data types, such as sets, for which different implementations are appropriate according to the circumstances. For this, a more down-to-earth approach is needed. So far, no one has done a convincing job of it.

Programming Language Themes

Up until now we have explored programming by looking at control and data aspects. We conclude with a brief survey of three of the most important themes in programming research: functional programming, logic programming, and constraint programming.

Functional Programming

Functional programming is an approach that says the most important tool we can bring to bear in our problem-solving efforts is the *mathematical function*. A mathematical function defines a certain kind of relationship between a collection of inputs (called *arguments*) and a single output (called the *result*), namely, that for each distinct set of arguments, there is a unique output that is always the same. This is obviously an appropriate point of view when we are concerned with arithmetic operations on numbers. It can be extended in a natural way to many other situations, and it is, indeed, a way of thinking that is often useful. At the beginning of this chapter we stated that procedures are the single most important organizational device for programs, and procedures most closely mimic mathematical functions.

Functional programming is distinguished by its insistence that its "procedures" should depend not at all on context, but should always return the same result given the same inputs. This becomes problematical with operations that receive input from the outside world (as all interactive computer music programs do) or that affect the outside world in some way (and any program that does not is useless). Naturally, functional programming advocates have answers to this objection and others.

Rather than delve any further into controversies about the universality of the functional approach, we wish to point out some of the inarguably valuable work that has been done under its auspices. Functional programming arises historically from mathematical work by Church, who invented the *lambda calculus* (Church 1941) in order to study functions. The lambda calculus inspired the Lisp programming language, which was originally a functional language.

Much important knowledge that is now considered basic to computing came out of research that resulted from the juxtaposition of computers and the lambda calculus. For example, this research clarified the notion of the *recursive procedure* and affected programming practice by showing how in many cases such procedures can be extremely useful. (A recursive procedure is one that can be viewed operationally as calling itself, or more abstractly,

as being defined partly in terms of itself. It is also possible to have *mutual recursion*, in which a collection of procedures are all defined in a mutually referential way.) Another valuable strand of work identified a set of ordering strategies for evaluation of functions and their arguments, demonstrating that the most obvious and cheapest implementation strategy may lead to a nonterminating computation in cases where a more expensive strategy would not.

Historically, functional programming surfaced early, in the form of Lisp. As Lisp developed, it became more pragmatic and lost its functional purity. The most widely used modern form of Lisp is Common Lisp, a large programming environment that brings together several Lisp dialects (Steele 1984). Scheme, a dialect of Lisp that has itself spawned other dialects, attempts to restore the benefits of compactness and purity (Abelson and Sussman 1985).

For a time functional programming was viewed as inefficient and therefore impractical. But functional programming has resurfaced, and an important motive is its potential use on parallel computers. There are a couple of reasons for this. First, if functions really are "pure" (in the sense of always returning the same result for the same arguments), then all the arguments to a function can be evaluated in parallel, and all of their arguments, and so on. Second, functional programming appears to be well suited to problems where large, uniform data sets (such as vectors, matrices, etc.) are acted on by functional operators; this leads to many opportunities for so-called "data parallelism" (Hillis 1987).

Logic Programming

The lambda calculus is one important mathematical development in this century. The *predicate calculus* is an even more important development, although one that began in the last century. A predicate can be thought of as a function that returns either true or false, and predicate calculus is the foundation of mathematical logic.

As functional programming is inspired by the lambda calculus, so *logic programming* is inspired by the predicate calculus. The predicate calculus can be viewed as a language with which to talk about mathematical theorems. An operational interpretation of the predicate calculus would be a theorem prover. Indeed, this is the correct abstract point of view to take with logic programming.

Earlier we described the equational approach to data types as a very abstract one, and pointed out that many problems within that area are

strictly unsolvable. This is also true of the predicate calculus. Thus, the game in logic programming is to find restricted forms of the predicate calculus or restricted theorem-proving strategies that make things computationally tractable while still retaining whatever advantage there might be to stating things in an abstract form similar to mathematical theorems.

By far the most popular logic programming language is Prolog (Clocksin and Mellish 1987), and the source of its popularity is a set of clever ideas that allow Prolog programs to be executed quite efficiently. A Prolog program can be viewed as a set of declarations in a restricted form of the predicate calculus called *Horn clauses*. By stating relations among predicates, these declarations in effect define the predicates. The idea is similar to the equational definitions we saw earlier, except that the relationship between left and right sides is not one of equality but of *implication*. That is, the left-hand side is true if the right-hand side is true. One important difference between this approach and functional programming is that the predicates can represent arbitrary *relations* as easily as functions. Relations are less restrictive than functions, which makes them more flexible representational vehicles in some cases.

A Prolog program attempts to solve a problem involving a predicate defined by the program, some of whose arguments are *variables*. The program's job is to find one or more solutions for the variables. It does this in a predetermined way, searching the declarations (known as *clauses*) one after the other. When the left-hand side of a clause matches the problem, the Prolog interpreter attempts to solve the right-hand side, in just the same way it goes about solving the original problem. If it reaches an impasse (no clause matches), it *backtracks*, going back to the last clause that matched, giving up on it, and trying to find another matching rule.

The backtracking behavior built into Prolog is very powerful, and convenient for problems involving searches, but also somewhat unpredictable and difficult to control. In practice, many predicates do not require backtracking, and various mechanisms have been introduced to restrict it.

An Example in Lisp and Prolog

In order to convey a more concrete image of both Lisp and Prolog, we present definitions of a simple function in the two languages. This is the function that concatenates two lists, known as **append**. In Lisp, **append** takes two lists as arguments and produces the concatenated list as a result. In Prolog, everything is a predicate, and all the interest is in the arguments, so **append** has three arguments, the third being for the "answer." The Lisp definition is in figure 2.9a; the Prolog definition in figure 2.9b.

(a)

```
(defun append (list1 list2)
      (cond ((null list1) list2)
            ((null list2) list1)
            (t (cons (car list1) (append (cdr list1) list2)))))
```

(b)

```
append([], X, X).
append(X, [], X).
append([X1 | X2], Y, [X1 | Z]) :- append(X2, Y, Z).
```

Figure 2.9 Definition of the **append** operation. (*a*) Lisp definition. (*b*) Prolog definition.

Let us first explain these definitions. In both Lisp and Prolog, lists are a fundamental data type. In Lisp, expressions are surrounded by parentheses. The first line introduces the definition of a function (**defun**), giving its name and arguments. The definition consists of a single expression whose first word is **cond**. This is a conditional expression, which tests the first part of each subexpression in turn, executing and returning the rest of the first such expression that evaluates to true. Thus, it says in essence: if *list1* is **null** (empty), return *list2*; if *list2* is **null**, return *list1*; otherwise ("**t**" stands for "true"), return this:

(cons (car *list1*) **(append (cdr** *list1*) *list2*))

This complicated expression glues together the first element of *list1* with the result of appending the rest of *list1* to *list2*. In other words, this is a recursive definition.

In Prolog, predicate arguments are in parentheses after the name, which is a more traditional mathematical notation. Names starting with a capital letter are variables. Usually clauses have many variables, and **append** is no exception. The first rule in figure 2.9b says that if the initial argument is the empty list (written "[]"), then the second and third arguments are equal. (Remember that the third argument is the "result.") The second rule makes a similar statement about first and third arguments. The third rule, which applies only if neither list is empty (because the first two rules are checked

first by a Prolog interpreter), says essentially the same thing as the complicated expression in the Lisp program, in a slightly different way. The right-hand side of the rule is equivalent to Lisp's recursive call to **append**.

No three-line program can tell you very much about a programming language, and this simple example is only intended to give a flavor. Modern Lisp and Prolog compilers execute these programs very quickly.

Constraint Programming

Constraint programming is related to logic programming, although the relationship is obscured by history. Whereas research in logic programming has always been dominated by people well versed in mathematical logic, constraint programming originated in a more intuitive and less rigorous fashion. Most work in this area has been concentrated in certain areas of interest: the solving of geometric constraints (such as keeping lines parallel or perpendicular, etc.), graphics systems (Sutherland 1963; Borning 1979), or constraints imposed by Ohm's law in electrical circuits (Sussman and Steele 1981). More recently, researchers trained in mathematics have explored seriously the relationships between logic programming and constraints (Jaffar and Lassez 1987; Saraswat 1992).

A major attraction of constraint programming is that it allows a programmer to state rules with a higher degree of complexity than Prolog's Horn clauses, or similar logic programming formalisms, and have them solved by more powerful, but also more specialized, interpreters. Ultimately, constraint programming could give programmers very flexible access to a large variety of specialized, mathematical algorithms, which currently require a great deal of expertise to choose and use.

In a slightly different vein, constraints are a potentially useful way of formulating problems that require *incremental* solutions: a given state of the system is represented as a particular solution of a constraint system, and any perturbation causes a new solution to be arrived at, one which typically changes relatively few of the components of the previous solution. Incremental algorithms have become increasingly important in recent years as applications have become more interactive and graphical.

Conclusion

We have tried to convey a sense of what programming is—the issues it involves and the breadth of approaches that can be taken to it. At the

outset, we observed that programming is fundamentally a problem-solving activity and discussed why the basic criterion of correctness may sometimes be difficult to apply. Many people enjoy problem-solving for its own sake. But surely one of the reasons programming is interesting is due to the nature of computer applications rather than in programming itself. Computers work according to simple rules that can be characterized in mathematical terms, and many programming formalisms and languages are derived from, or at least inspired by, mathematics, as has been amply discussed in this chapter. On the other hand, computers are physical objects and can be connected to the real world in interesting ways. When a computer controls a synthesizer, the result is a sound whose properties are much richer than any mathematical formalism (at least to the ear). The programmability of computers allows this abstract, mathematical simplicity to be juxtaposed with real activity and effect. The pleasure of coaxing a prototype music system to work is a pleasure of musical engineering. Programming is unlike earlier varieties of engineering in the ethereal nature of its materials, but it remains an engineering discipline in the way work is rewarded with tangible results.

Another aspect of the excitement is the interdisciplinary nature of computing. This is a point that we need not belabor to those interested in computer music. Computer science itself has borrowed ideas from many areas of mathematics and engineering, and has returned a number of favors as well. As computers become more deeply embedded in our society, the need to marry knowledge of computing with knowledge in other domains becomes more common.

Although computers have only been around for a few years, hundreds of programming languages and variants have been implemented, and thousands more proposed. With all these languages, people unfamiliar with the subject might think the territory has been thoroughly explored. Nothing could be farther from the truth. Programming is evolving continuously and quickly, entrenched languages like Cobol and Fortran notwithstanding. Thus, while many of the ideas and techniques described in this chapter will remain valid in a decade or two, some may not. Almost certainly, the traditional approach to programming, to which the bulk of our attention was devoted, will become relatively less important. As things evolve, some of the techniques described here that remain valid and useful will nevertheless come to be seen in a different light, and used in different ways.

Probably the most important realization to come out of the accumulated experience with programming so far is that it is a problem of organizing and managing complexity—to a far greater extent than was initially realized.

The structured programming movement, which has played an important role for more than 20 years now, can be viewed as a reaction to this. The languages it engendered now dominate programming practice and form the basis of this chapter. Object-oriented programming has the potential to carry these gains a few steps farther since many of its key contributions have to do with managing complexity. Constraint programming, and perhaps some forms of logic programming, may well come to play a more important role as well.