Part I

Introduction

Chapter 1

Programming Paradigms

We begin our exploration of multiparadigm programming by considering that puzzling word "paradigm." If you consult a dictionary with the objective of better understanding the meaning of this term, you are likely to find an entry such as the following, from *The American Heritage Dictionary of the English Language:*

> **par · a · digm** *n.* **1**. A list of all the inflectional forms of a word taken as an illustrative example of the conjugation or declension to which it belongs. **2.** Any example or model. [Late Latin *paradīgma*, from Greek *paradeigma*, model, from *paradeiknunai*, to compare, exhibit.]

At first blush, the conjugation or declension of Latin words would seem to have little to do with computer programming languages. To understand the connection, we must note that the word was brought back into the modern vocabulary through an influential book called *The Structure of Scientific Revolutions*, authored by the historian of science Thomas Kuhn [Kuhn 62]. Kuhn used the term in the second form, to describe a set of theories, standards and methods that together represent a way of organizing knowledge; that is, a way of viewing the world. Kuhn's thesis was that revolutions in science occurred when an older paradigm was reexamined, rejected and replaced by another.

It is in this sense, as a model or example, and as an organizational approach, that Robert Floyd used the term in his 1979 ACM Turing Award lecture, which was titled "The Paradigms of Programming" [Floyd 79]. A programming paradigm is a way of conceptualizing what it means to perform computation, and how tasks that are to be carried out on a computer should be structured and organized.

Linguists have long recognized that the language in which a thought is expressed colors in a fundamental way the nature of an idea. It is not particularly surprising nor, for that matter, particularly noteworthy that different groups of individuals will, over time, develop a specialized vocabulary to discuss topics of interest. While the often repeated story that Eskimo languages have dozens of words for snow may now be known to be apocryphal, similar examples with other natural languages, or within a single language, occur frequently. Arabic languages have many words for "camel," for example. Similarly, a person in the book publication business would describe the page on which this sentence appears using very different terms than would be employed by a casual reader.

This does not mean that my eyes differ physically from those of an arab or those of somebody in the publication business, or that those individuals somehow can view things that I physically cannot, but merely that habits of language will naturally direct thoughts in one direction or another. I can look at a picture and notice only that there are camels in the scene. A native arabic speaker would of habit or necessity notice other features, since to describe the picture in any manner at all they would need to be able to say if the camel was young or old, in good health or bad, and so on. In the same fashion a reader can look at this page and see only the words. A person engaged in publication would, by habit, look at the same page and note features such as fonts, vertical spacing, and interword gaps. In this manner the confines of a language direct (although not necessarily constrain) the nature of a thought. This idea is expressed in the following quotation from the American linguist Edward Sapir:[1]

> "Human beings do not live in the objective world alone, nor alone in the world of social activity as ordinarily understood, but are very much at the mercy of the particular language which has become the medium of expression for their society. It is quite an illusion to imagine that one adjusts to reality essentially without the use of language and that language is merely an incidental means of solving specific problems of communication or reflection. The fact of the matter is that the 'real world' is to a large extent unconsciously built up on the language habits of the group.... We see and hear and otherwise experience very largely as we do because the language habits of our community predispose certain choices of interpretation."

What is true for natural human languages is even more true in the realm of artificial computer languages. That is, the language in which a programmer thinks a problem will be solved will color and alter, in a basic fundamental way, the fashion in which an algorithm is developed.

An example will help illustrate this relationship between computer language and problem solution. Several years ago a student working in genetic research was faced with a task involved in the analysis of DNA sequences. The problem could be reduced to a relatively simple form. The DNA is represented as a vector of N integer values, where N is very large (on the order of tens of thousands). The problem was to discover whether any pattern of length M, where M was a fixed and small constant (say five or ten) is ever repeated in the vector of values.

[1] Quoted in the article "The Relation of Habitual Thought and Behavior to Language" by Benjamin Lee Whorf; reprinted in [Whorf 56].

The programmer dutifully sat down and wrote a simple and straightforward **FORTRAN** program, something like the following:

```
    DO 10 I = 1, N-M
        DO 10 J = 1, N-M
            FOUND = .TRUE.
            DO 20 K = 1, M
20                IF X[I+K-1] .NE. X[J+K-1] THEN FOUND = .FALSE.
            IF FOUND THEN ...
10    CONTINUE
```

The student was somewhat disappointed when trial runs indicated his program would need ten or fifteen hours to complete. He discussed his problem with a second student, who happened to be proficient in the programming language APL. The second student said that she would like to try writing a program for this problem.

The first student was dubious. After all, FORTRAN was known to be one of the most "efficient" programming languages. It was compiled, after all, and APL was only interpreted. Therefore it was with a certain amount of incredulity that he discovered the APL programmer was able to write an algorithm that worked in a matter of minutes, not hours.

What the APL programmer had done was to rearrange the problem. Rather than working with a vector of N elements, she reorganized the data into a matrix with roughly N rows and M columns:

x_1	x_2	$\ldots$	x_m
x_2	x_3	$\ldots$	x_{m+1}
$\ldots$			$\ldots$
x_{n-m}	$\ldots$	$\ldots$	x_{n-1}
$x_{n-(m-1)}$	$\ldots$	x_{n-1}	x_n

The APL programmer then sorted this matrix by rows. If any pattern was repeated, then two adjacent rows in the sorted matrix would have identical values. It was a trivial matter to check for this condition. The reason the APL program was faster had nothing to do with the speed of APL versus FORTRAN, but was simply because the FORTRAN program employed an algorithm that was $O(M \times N^2)$, whereas the sorting solution used by the APL programmer required approximately $O(M \times N \log N)$ operations.

The point of this story is not to indicate that APL is in any way a "better" programming language than FORTRAN, but to ask why it was that the APL programmer was naturally led to discover a better solution. The reason, in this case, is that loops are very difficult to write in APL, whereas sorting is trivial; it is a built-in operator defined as part of the language. Thus, because the operation is so easy to perform, good APL programmers tend to look for novel applications for sorting. It is in this manner that the programming language in which the solution is to be written directs the programmer's mind to view the problem in one fashion or another.

The assertion that the language in which an idea is expressed can influence or direct a line of thought is relatively easy to believe. It should be noted, however, a stronger conjecture, known in linguistics as the Sapir-Whorf hypothesis, goes much further and remains controversal. The Sapir-Whorf hypothesis asserts that it may be possible for an individual working in one language to imagine thoughts or utter ideas which cannot in any way be translated, cannot even be understood, by individuals operating in a different linguistic framework. According to advocates of the hypothesis, this can occur when the language of the second individual has no equivalent words, and lacks even concepts or categories for the ideas involved in the thought. We will return to this topic in a moment when we discuss a similar, but, interestingly enough, almost opposite, conjecture in the field of programming languages; namely, Church's conjecture.

If a paradigm is a way of organizing a world view, and if languages can lead programmers in vastly different directions, then in what sense can a concept or programming language be "multiparadigm"? It would seem as if the term was itself an implicit contradiction, an oxymoron. In fact, this is not the case. Consider, instead of computer programming, the task of a mathematician attempting to prove a conjecture. A mathematician may choose to use techniques that are algebraic, geometric, topological, logical or constructive in nature. Each of these approaches bring with them a history, outlook and approach that differs from the others; thus each can reasonably be described as a separate paradigm. Yet there is no contradiction; rather, the mathematician simply possesses a rich collection of techniques from which to select.

The goal of multiparadigm computing is to provide the programmer with a similar rich set of tools. The well-educated computer programmer should be fluent in a number of different problem solving styles. When faced with a particular task to perform, the programmer should then be able to select a solution technique that best matches the characteristics of the problem to be solved.

To attain this goal it is necessary to create a linguistic framework (that is, a programming language) in which each of the different major paradigms can be easily and naturally expressed, and in which each paradigm draws benefit from the presence of the others. This was our underlying objective in the creation of the programming language Leda.

1.1 Church's Conjecture and Turing Tarpits

Starting in the 1930's and on through the 40's and 50's there was a great deal of interest within the mathematical and nascent computing community in a variety of formalisms that could be used for the calculation of functions. Examples include the notations proposed by Church [Church 36], Post [Post 36], Markov [Markov 51], Turing [Turing 36], Kleene [Kleene 36] and others. Over time a number of arguments were put forth to demonstrate that many of these systems could be used in the simulation of others. Often, for a pair of systems, such arguments could be made in both directions; effectively showing that the systems were identical in computation power. The sheer number of such arguments lead the logician Alonzo Church to pronounce a conjecture that is now associated with his name:[2]

> **Church's Conjecture:** Any computation for which there exists an effective procedure can be realized by a Turing machine.

By nature this conjecture must remain unproven and unprovable, since we have no rigorous definition of the term "effective procedure." Nevertheless, no counterexample has yet been found, and the weight of evidence seems to favor affirmation of this claim.

Acceptance of Church's conjecture has an important and profound implication for the study of programming languages. Turing machines are wonderfully simple mechanisms, and it does not require many features in a language in order to be able to simulate such a device. In the 1960's, for example, it was demonstrated that a Turing machine could be emulated in any language that possessed at least a conditional statement and a looping construct. (This greatly misunderstood result was the major ammunition used to "prove" that the infamous `goto` statement was unnecessary.)

If we accept Church's conjecture, then any language in which it is possible to simulate a Turing machine is sufficiently powerful to perform *any* realizable algorithm. (To solve a problem, find the Turing machine that produces the desired result, which by Church's conjecture must exist, then simulate the execution of the Turing machine in your favorite language). Thus, arguments about the relative "power" of programming languages, if by power we mean "ability to solve problems," are generally vacuous. The late Alan Perlis had a term for such arguments, calling them a "Turing Tarpit," since they are often so difficult to extricate oneself from, and so fundamentally pointless.

Note that Church's conjecture is, in a certain sense, almost the exact opposite of the Sapir-Whorf hypothesis. Church's conjecture states that in a fundamental fashion all programming languages are identical. Any idea that can be expressed in one language can, in theory, be expressed in any language. The Sapir-Whorf hypothesis, you will recall, claimed that it was possible for ideas to be expressed in one language that could not be expressed in another. Many linguists reject the Sapir-Whorf hypothesis and instead adopt a sort of "Turing-equivalence" for natural languages. By this we mean that, with a sufficient amount of work, any idea can be expressed in any language. For example, while the language spoken by a native of a warm climate may not make it instinctive to examine a field a snow and categorize it with respect to different types or uses, with time and training it certainly can be learned. Thus, for both computer and natural languages it is the case that the language will *direct* thoughts, but cannot *proscribe* thoughts.

If we cannot compare the fundamental computing power of different programming languages (since at a basic level they are all the same), then on what basis should one evaluate

[2] My history is slightly imprecise here. Church actually conjectured about partial functions. These were later shown to be equivalent to computations performed using Turing machines. Kleene described the conjecture in the form we have here, also giving it the name by which it has become known.

and select from among the rich babel of languages? The unfortunate answer is that the characteristic most important to an individual programmer is also largely the most subjective and un-quantifiable feature, and this is the question of "ease of use."[3]

Certainly few people would choose to write a computer program as a Turing machine, if other alternatives were available. Nowadays most programmers would similarly decline to program in basic machine assembly language (although this was not always so; in the 1950's many programmers had to be convinced that it would be to their advantage to switch from assembler to the newly developed high-level languages). The near universal disdain of Turing machines and assembly languages as a practical tool for the development of software is not because they can not be used to perform any particular computing task, but because they are difficult to use, difficult to debug, and both take longer to write and produce larger programs.

The same arguments are extended to various families of high-level languages, but with much less agreement as to the outcome. Each of the major programming language communities, such as the object-oriented programming community, the logic programming community, or the functional programming community, (as well as countless smaller schools of thought, such as the constraint programming community or the visual programming languages community), possess partisans who can and do provide convincing arguments that for certain types of problems their particular style of program development produces algorithms that are better, in some sense, than code produced in alternative styles.

Unfortunately, for other problems, the "fit" between language and problem may be clumsy at best, resulting in programs that are far less than elegant. Many examples can be cited, such as trying to code algorithms that manipulate trees in APL (a language designed for dealing with homogeneous rectilinear arrays), or numerical algorithms in Prolog (a logic programming language).[4]

The basic tenet of multiparadigm programming is that the programmer should not be forced to solve all problems in a single style; instead, the programmer should be free to select the programming tools for a problem that best match the nature of the task being performed. While this view may be obvious, there are basic difficulties inherent in the realization of such a goal. The first problem is education; for a programmer to feel comfortable about using multiple paradigms he or she must have knowledge concerning what facilities exist and how they are used. This is tied to the second problem, which is the development of a

[3] This is not to imply that there are no quantifiable characteristics of programs that might be interesting and relevant to measure. But the interpretation of quantifiable measures is itself fraught with difficulty. For example, one might consider measuring the size (number of characters) of programs solving the same task in two or more different languages, assuming that a shorter program would be preferable to a longer one. Using this metric, programs in the language APL would be ranked very highly, since programs in this language are often an order of magnitude smaller than programs solving the same task in other languages. Yet APL is often criticized on this very ground for being "too cryptic" and a "write only language."

[4] A puzzling aspect of computer languages that has been observed repeatedly is that partisans frequently seem incapable of noticing such mismatches. This situation is well summed up by the saying "when all you have is a hammer, everything looks like a nail." The Scandinavian computer scientist Peter Naur has noted this phenomenon [Naur 75], and provides analogies with similar situations in natural languages.

framework in which the different paradigms can not only coexist, but can derive benefits from each other. Finally, the creation of such a language and programming environment must avoid a pair of twin problems. On the one hand, there is the problem of "the kitchen sink." This is where the language provides all tools that anybody might ever need or want, but is so large that few programmers can understand or use all features. The other extreme is "the swiss army knife," where the language is so concise that everything is possible, but nothing is easy.

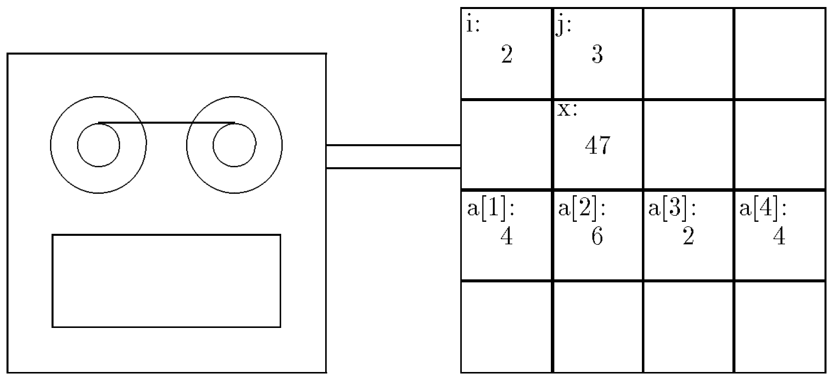
Our fundamental goal in the creation of Leda was to design a language that would be simple and small, easy to master, and yet provide the freedom to permit the elegant solution of problems in a wide variety of styles. It will be up to the reader (or programmer) to judge how well we have succeeded in meeting these objectives.

1.2 Paradigmatic Characterizations

The following sections will briefly describe the basic characteristics of each of the paradigms incorporated into Leda.

1.2.1 Imperative Programming

The imperative paradigm is often viewed as the "traditional" model of computation. The imperative model closely matches the actual execution of the computer on the underlying hardware. Computation is viewed as a task being performed by a "processing unit," which manipulates and modifies "memory." Memory can be envisioned as a sequence of boxes, or pigeon-holes, in which values are stored. Pigeon-holes can be numbered or, in higher-level languages, given symbolic names. The computer itself is a data manager; it follows a pattern of instructions, each instruction causing the computer to extract certain values from memory, transform them in some fashion, then place results back into memory locations. When computation halts the values held in memory reveal the desired solution.



The imperative model is often described using the phrase "incremental changes to state." To illustrate what this term means, we can extend our pigeon-hole metaphor. Language structures such as arrays or records can be thought of as boxes (little memory units with their own form of pigeon-holes) that can be nested within other boxes (the local memory for a procedure, the entire memory for the computer, and so on). As computation proceeds the boxes remain fixed, but the contents continually change. A large structure, such as an array, is manipulated by modifying each individual element one-by-one. Thus, computation proceeds by modifying large structures a little at a time, until finally the intended result is achieved.

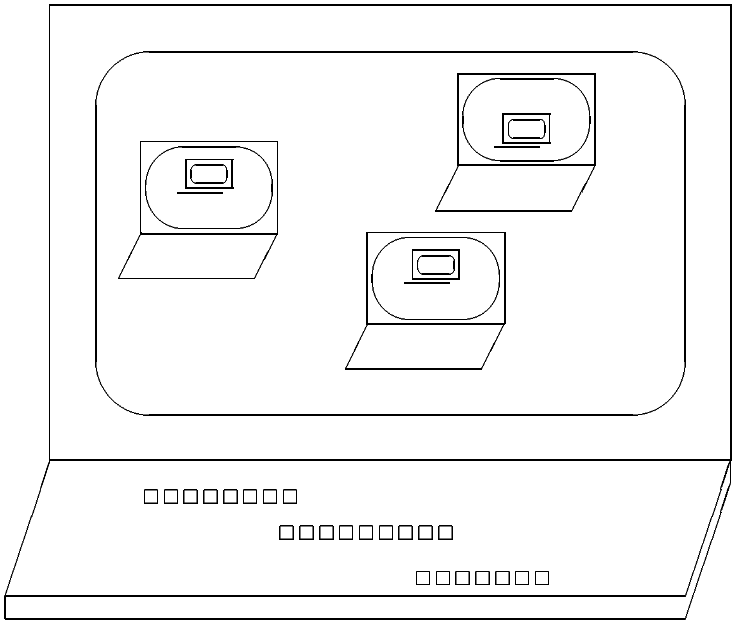
Programming language features usually associated with the imperative style of programming include variables (that is, identifiers that modify their value over time), the assignment statement, arrays, records, and control flow constructs such as `if` and `while` statements.

The major difficulty with the imperative paradigm is not that it presents an inaccurate picture of computation; indeed, as we have already noted it is extremely close to the way in which computation is actually performed at the most fundamental level. Rather, the problem with the imperative paradigm is that it is not particularly close to the way in which human beings (postal workers and pigeon breeders excepted) go about solving problems. Thus, formulating a problem in an imperative style many be a difficult task. Furthermore, the difficulty of this task increases dramatically as the complexity of the problem increases. This, the famous "software crisis," has fueled the interest into new and different ways of envisioning the computation task.

1.2.2 Object-Oriented Programming

The fundamental new insight behind object-oriented computing is the notion of "recursive design." That is, in a complex system, power is achieved by making the structure and abilities of the pieces resemble the structure and abilities of the whole. Many examples of this principle can be found in science as well as in nature. Fractals are a currently popular illustration from the field of mathematics, while objects as mundane as a stalk of broccoli exemplify the principle in nature. (A stalk of broccoli is similar in structure to an individual stem, which is similar to a part of a stem, which is similar to the tip piece, and so on).

When applied to computers, the recursive design approach tells us that rather than viewing the computer as being divided into more primitive components, such as procedures and data structures, we should envision the computer as a universe formed from a network of hundreds or thousands of interacting computing units. Each "computing unit" (called an *object*), like the computer itself, is composed of a processing unit and memory. But in this case memory is again recursively viewed as consisting of yet further objects, and so on ad-infinitum.



Alan Kay, considered by many the father of object-oriented programming, has identified five fundamental characteristics of Smalltalk, the language that first popularized the object-oriented view:

1. Everything is an *object.*

2. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving *messages.* A message is a request for action bundled with whatever arguments may be necessary to complete the task.

3. Each object has its own *memory*, which consists of other objects.

4. Every object is an *instance* of a *class.* A class simply represents a grouping of similar objects, such as integers, or lists.

5. The class is the repository for *behavior* associated with an object. That is, all objects that are instances of the same class can perform the same actions.

Subsequent object-oriented languages, as well as later versions of Smalltalk (Smalltalk–76 and Smalltalk–80), added a sixth principle:[5]

[5] In view of the tremendous emphasis placed on inheritance in much of the subsequent descriptions of

6. Classes are organized into a singly-rooted tree structure, called the *inheritance hierarchy*. Memory and behavior associated with instances of a class are automatically available to any class associated with a descendent in this tree structure.

Thus, the fundamental programming constructs required to support the object-oriented programming approach are classes (possibly organized into an inheritance hierarchy), and instances. Classes describe objects that are encapsulations of behavior (actions) and state (data values). Rather than writing procedures that manipulate data structures, operations and data are viewed as an organic whole, a unit that provides a service (such as list processing) used by other portions of a software system. This view can be succinctly summed by a twist on a well-known quotation: "Ask not what you can do *to* your data structures, but ask what your data structures can do *for* you."

The object-oriented philosophy has received tremendous publicity in both the technical and popular press in large part because the approach to problem solving; namely, organizing a loose collection of autonomous but interacting units, resonates with problem solving skills from other domains. People are used to thinking about the organization of individuals, such as on a committee, or solving problems by piecing together the services provided by many different vendors. Thus, unlike the situation with imperative programming, the metaphor applied in the object-oriented approach is more natural to human problem solvers.

While usually viewed as an improvement over imperative programming, object-oriented programming still suffers from the problem that computation is perceived as a task of producing changes in the state of objects. This view is not shared with the remaining two paradigms in our collection.

1.2.3 Functional Programming

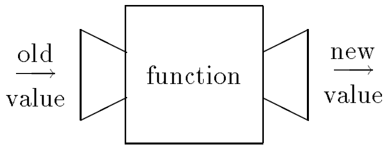
The Functional programming philosophy is often described by three basic characteristics:

- Values are treated as single entities, and not as "boxes with slots." Values can be created, but once created they are never modified. Rather than making small incremental changes to existing structures, values are transformed into new values, which are then independent of the original. For example, the addition of a value to a list results in a completely new list, rather than a modification of the original. (In practice, the two lists may be implemented in such a way that they share a large portion of their structure).

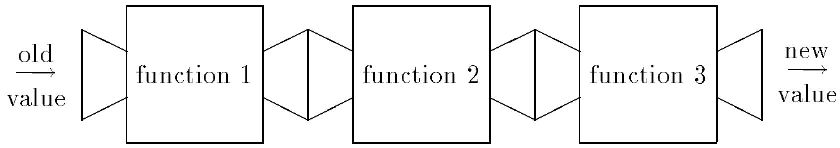
  A corollary of this view is that variables do not correspond to fixed locations in memory with definite size, but are simply linguistic handles through which values are accessed. This is almost the exact opposite of object-oriented programming, since values do not

object-oriented programming (see [Wegner 87] for example), I think it is somewhat surprising that Kay (I think correctly) relegates this issue to a relatively minor role. Inheritance is a useful and powerful implementation technique, but the concept of autonomous objects maintaining their own state and behavior is more fundamental to OOP.

have "state" that is modified over time. Indeed, the notion of "time" is very different in functional programming than it is in object-oriented programming (see below).



- Computation is largely performed by applying functions to values. This may be done repeatedly, applying functions to the result generated by other functions. Assignment, if it is permitted at all, is performed only at the topmost level of a computation, and only to provide a handle by which a value may be named. (In fact, many functional languages do not allow *variables*, that is, named quantities that can change their value over time. Identifiers can denote values, but such values, once created, are fixed and cannot change. Such usage is sometimes referred to as *single assignment form*.)



- Functions are a "first-class" data value. This means that functions can be assigned to identifiers, passed as arguments, or returned as the result of executing other functions. High level tools are created through the use of functions that accept other functions as arguments. Even more importantly, the creation of new functions is facilitated through the use of functions that return functions as results.

An important consequence of the emphasis on transformation rather than on state is the fact that functional programs are *atemporal* ("without time"). That is, within the constraints that a function can only be applied once the arguments are available, and must be applied before the results are used, the execution of functions can be performed in any sequence. If a function produces a certain result when provided with a given set of arguments, it will always produce the exact same result when provided with the same arguments, no matter when or how often the function is executed. This feature of functions is referred to as *referential transparency.* Functions that possess referential transparency can be more easily manipulated as mathematical objects and can therefore more easily be subject to formal validation techniques. It is this similarity to mathematical concepts and the amenability to formal techniqiues that has made functional programming popular with many practioners.

The major programming language feature required to support the functional programming approach is the creation of functions as first-class values. For the implementor of a functional programming language probably the most important consequence of permitting functions as first-class values is the fact that functions must capture and retain the environment in which they are defined. (We will discuss the implementation of functional programming language features in Chapter 18.)

Most adherents to functional programming would not, however, argue about what features needed to be *added* to a programming language to support the style, but rather what features should be taken away. Concepts such as side effects through the access and modification of variables, and assignment to identifiers with values that vary over time are usually cited as devices that fill greatly needed holes. In Leda we have taken the view that those wishing to remain pure in the functional programming paradigm must simply employ discipline to avoid those language features that are at odds with the technique. The benefits of permitting interaction with different styles more than outweighs any loss of paradigmatic purity. Indeed, even functional programming advocates usually permit the prohibition against side effects to be bent at both the lowest level and highest levels of execution. (Input and output, for example, is a process that inherently involves side effects).

1.2.4 Logic Programming

The logic programming approach grew out of work in the artificial intelligence community on the problem of automatic theorem proving, and the technique is still usually described by analogy to theorem proving. The classical propositional proof of a theorem consists of three parts, a set of *axioms* (facts assumed to be self-evidently true, or at least accepted for the purposes of the argument), a set of *rules of inference* (rules by which new information can be derived from already known facts), and a *question* or *query*. The proof itself consists of a path that leads from the assumed facts, using the rules of inference, and ends by showing that the query can be derived from the given information.

Propositional calculus is one well-known form of theorem proving. Propositional calculus has been studied since antiquity, and many classic rules of inference are still known by their Latin names. *Modus Ponens*, for example, is a rule of inference that states that if "`P` implies `Q`" is true, and "`P`" is true, then "`Q`" must be true. The following simple argument illustrates the use of propositional calculus. Assume we accept the following two statements as fact:

1. If the Beavers beat the Ducks or the Beavers beat the Huskies then the Beavers will be conference champions.

2. The Beavers are not conference champions.

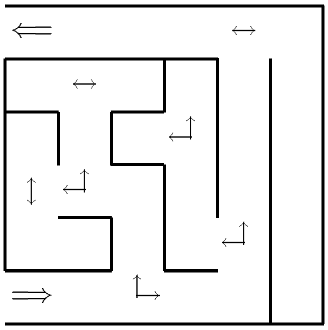
The question we wish to answer is "Did the Beavers beat the Huskies"? The following table shows how this query can be answered using the classical rules of inference from propositional logic, with the steps labeled with their conventional name.

if the Beavers beat the Ducks **or** the Beavers beat the Huskies **then** the Beavers will be conference champions	*assumption*
not(The Beavers are conference champions)	*assumption*
not(Beavers beat the Ducks **or** Beavers beat the Huskies)	*modus tollens* P $\Rightarrow$ Q and not Q then not P
not(Beavers beat the Ducks) **and** **not**(Beavers beat the Huskies)	*DeMorgans Rule* not (A or B) $\equiv$ (not A) and (not B)
not Beavers beat the Huskies	*simplification*

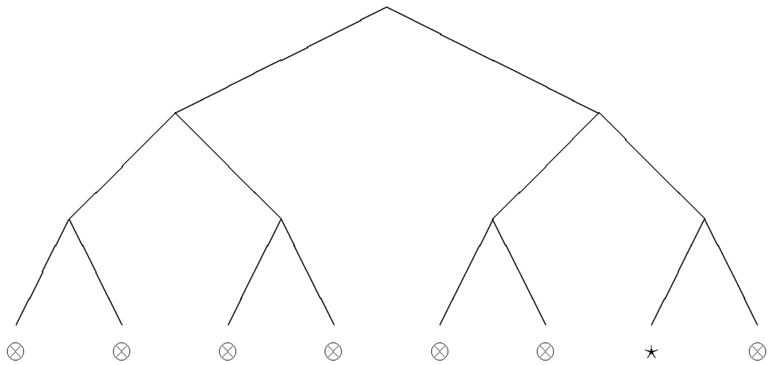
What is most important about the logic programming process is that programming is *declarative*, or *non-procedural*. That is, the programmer simply provides a series of assertions of facts, a collection of rules of inference, and a query. The programmer does not need to specify how the query is to be answered using the information provided. Instead, an underlying searching mechanism works behind the scene to see if the facts can be deduced from the given information.

As anyone who has tried to solve classical syllogisms by hand has observed, it is often the case that a search for a solution will lead in directions that are not ultimately productive, that is, lead to "blind alleys." The searching procedure must therefore be prepared to "back up" in these situations, to return to an earlier point where an alternative selection was possible and try a different approach. This process is known as *backtracking*, and is an important part of the logic programming process. Following the first and most popular logic programming language, Prolog, backtracking in Leda is performed using a depth-first strategy. One path is tried until it can no longer be extended. If the query has not yet been answered, then computation returns to the last point where there was an untried alternative, and computation resumes along the alternative. This proceeds until either the query is answered or no more alternatives can be located.

To illustrate the concept of backtracking, consider attempting to search a maze, such as the following. At several labelled points in the search there is a choice that must be made between traveling left or traveling right. The depth first search technique will always try the first alternative (say, move left), until it can move no further. It then backs up to the last point where there was an untried alternative, and investigates the alternate choice.



We can represent the sequence of choices made by the searching algorithm as a binary tree, where the left child represents a choice to move left, and the right child represents the alternative choice moving right. If we let the symbol $\otimes$ represent the discovery of a dead end, this tree for our maze searching algorithm will look like the following. Notice that many alternatives are tried before the correct solution (represented by the symbol $\star$) is discovered.



Because pure logic programming is declarative it, like functional programming, is as consequence atemporal. The benefits of atemporality (being able to more easily reason about the execution of programs) accrue equally to logic programs as to functional programs.

Unfortunately, pure logic programs are useful for only a very limited domain of problems. Even simple concepts, such as negation, can cause extremely difficult problems for the underlying searching mechanism. For this reason, almost all logic programming languages (such as Prolog) are "multiparadigm" in that they stray from the pure logic framework in one way or another. With Leda we have argued that this deviation is beneficial, and even greater power is possible through the interaction of logic programming features and programming techniques from other programming paradigms.

The major programming language feature required to support the logic programming paradigm is a mechanism for describing facts and rules of inference, and an underlying search mechanism. In Leda both facts and rules of inference are provided through the introduction of a subprogram abstraction called a *relation*, and a special handling of the boolean operators "and" and "or." The implicit search mechanism is provided through the compilation of boolean expressions into a special form. (The implementation of the various language features is described in more detail in Chapter 22).

1.2.5 Other Programming Paradigms

While the imperative, object-oriented, functional and logic programming paradigms are the four targeted by Leda, they certainly do not constitute all the programming paradigms that have been identified. A few other common paradigms are the following:

- The **Access-oriented** programming paradigm. Using this technique, side-effects, or "demons," are attached to variables so that certain functions will be performed as variables are manipulated.

- The **Constraint** programming paradigm. Like logic programming, constraint programming is more declarative than imperative in nature. The programmer specifies a sequence of constraints that must be maintained during program execution (such as a window should always be twice as wide as it is tall, or a certain variable should be the sum of two other variables). An underlying *constraint satisfaction mechanism*, similar to the search mechanism used in logic programming, then assures that the constraints are met as the program executes. (We will investigate a simple form of constraint programming in Chapter 21.)

- The **Parallel** programming paradigm. Computation is defined as taking place with multiple processes, either on a single or on many actual processors. (In fact, it is probably incorrect to, at this time, describe parallel programming as a single paradigm. There are a number of different paradigms, such as the tuple-space model, data parallel programming, data flow techniques, all of which compete for attention in the parallel programming world. It is largely the lack of agreement on a single paradigm which makes, at the present time, programming parallel computers difficult, and parallel programs unportable).

- The **Visual** programming paradigm. Programs are specified visually, rather than textually. In practice, this is also a family of paradigms, as there are functional visual languages, object-oriented visual languages, and so on.

Notes and Bibliography

The collection of papers by the American linguist Benjamin Lee Whorf [Whorf 56] includes several that touch on the issue of the relationship between language and thought. His observations regarding natural languages often have surprising correspondences to similar phenomenon in artificial computer languages. The commonly held observation that Eskimo languages have many words for snow is debunked by Geoffrey Pullum [Pullum 91]. An excellent refutation of the Sapir-Whorf hypothesis is provided by Steven Pinker [Pinker 94].

Another interesting book dealing more with languages than with computer languages is *The Alphabet Effect*, by Robert Logan [Logan 86]. This work attempts to explain, in terms of language, why logic and science developed in the West, while for centuries China in the East had superior technology. (The west having the benefit of a sequential, linear alphabet, while the Chinese did not.) In a more contemporary investigation of the effect of natural language on computer science, J. Marshall Unger [Unger 87] describes the influence of the Japanese language on the much-heralded Fifth Generation project. The Scandinavian computer scientist Peter Naur (partial originator of Backus-Naur notation) has also made some interesting reflections on the relationship between natural and computer languages [Naur 75]. The relationship between paradigms and thought models is invesigated by Rechenberg [Rechenberg 90]. A rather down-beat view of programming languages, which concludes that the best languages will never see wide-spread use, is given by Jürg Gutknecht in [Gutknecht 93].

Church's conjecture was originally posed in [Church 36]. As I noted in an earlier footnote, Church posed his famous thesis in terms of partial recursive functions. Only later was it phrased in the more modern form using Turing machines (which first appeared in [Turing 36]). Rogers [Rogers 67] provides an excellent description of the equivalence of the various formal notations for computation.

The observation that sequential execution, conditionals and loops are sufficient to simulate a Turing machine was proved by Böhm and Jacopini [Bohm 66]. As I mentioned in the text, this result has often been used to "demonstrate" that the `goto` statement is not a necessary component in a programming language. The fact that this result is of only theoretical interest was eventually demonstrated by showing that there exist programs that must by necessity increase exponentially in length when translated from a goto version into a goto-less form. Nevertheless, note that Leda does not have a goto statement.

The "software crisis," mentioned in Section 1.2.1, was pronounced as early as 1968. It is curious to note that we have been in a state of crisis now for more than half the life of Computer Science as a discipline. Despite the end of the cold war, the end of the software crisis seems to be no closer now than it did in 1968.

Alan Kay is usually considered the individual who most importantly crystalized the ideas that were eventually to be known as object-oriented programming. Kay has described much of the early history of OOP in his paper at the Second History of Programming Languages conference [Kay 93]. However, many of the ideas implicit in OOP can be traced (and are traced, by Kay) back to an earlier language Simula, developed in the early 1960's [Dahl 66].

A good description of the origins of Simula was presented by Kristen Nygaard at the first History of Programming languages conference [Nygaard 78].

The ideas of functional programming were popularized by John Backus in his 1987 Turing award presentation talk [Backus 87]. A good, accessible introduction to the basic ideas of functional programming has been written by Bruce MacLennan [MacLennan 90].

Sterling and Shapiro provide a good introduction to logic programming in the language Prolog [Sterling 86]. Details of logic programming in languages other than Prolog can be found in Hogger [Hogger 84]. Some of the difficulties in programming in a purely logical fashion are discussed by MacLennan [MacLennan 90] and by Lloyd [Lloyd 84].

Characterizations of different programming paradigms, including some we have not considered here, can be found in [Hailpern 86, Placer 91b, Rechenberg 90, Ambler 92].

Exercises

1. What are some quantifiable characteristics of computer programs? Describe what difficulties you can encounter in trying to associate this characteristic with some sense of program "quality."

2. If you are familiar with two or more distinct natural languages (or computer languages), give an example of a situation where one language would direct a thought or an utterance in one direction, whereas the second language would suggest a different way of presentation.

3. Give an example of recursive design ("the structure of the part is analogous to the structure of the whole") from another problem domain.

4. Explain how referential transparency can be violated. That is, explain how a function might produce a different value each time it is executed, even when passed the same argument values.