Programming Languages Paradigms

José de Oliveira Guimarães
Computer Science Department
UFSCar
São Carlos, SP
Brazil
e-mail: jose@dc.ufscar.br

16 de março de 2007

Sumário

Preface

This book is about programming languages paradigms. A language paradigm is a way of thinking about a problem, restricting the ways we can build a program to specific patterns that are better enforced by a language supporting that paradigm. Then, the object-oriented paradigm forces one to divide a program into classes and objects while the functional paradigm requires the program to be split into mathematical functions.

Programming languages books usually explain programming language paradigms through several representative languages in addition to the main concepts of the field. There is, in general, a great emphasis on real languages which blurs the main points of the paradigms/concepts with minor languages particularities. We intend to overcome these difficulties by presenting all concepts in a Pascal-like syntax and by explaining only the fundamental concepts. Everything not important was left out. This idea has been proven successful in many programming language courses taught at the Federal University of São Carlos, Brazil.

This book is organized as follows. Chapter 1 covers a miscellany of topics like programming language definition, history of the field, characteristics of good languages, and some discussion on compilers and computer theory. Basic programming language concepts are presented in Chapter 2. The other chapters discuss several paradigms like object oriented, functional, and logic.

Capítulo 1

Introduction

1.1 Basic Questions

A programming language is a set of syntactic and semantic rules that enables one to describe any program. What is a program will be better described at the end of this chapter. For a moment, consider a program any set of steps that can be mechanically carried out.

A language is characterized by its syntax and semantics. The syntax is easily expressed through a grammar, generally a context-free grammar. The semantics specifies the meaning of each statement or construct of the language. The semantics is very difficult to formalize and in general is expressed in a natural language as English. As an example the syntax for the while statement in C++ is

while-stat ::= `while (` *expression* `)` *statement*

and its semantics is "while the *expression* evaluates to a value different from 0, keep executing *statement*". Semantics is a very trick matter and difficult to express in any way. In particular, natural languages are very ambiguous and not adequate to express all the subtleties of programming languages. For example, in the semantics of `while` just described it is not clear what should happen if *expression* evaluates to `0` the first time it is calculated. Should *statement* be executed one time or none ?

There are formal methods to define a language semantics such as Denotational Semantics and Operational Semantics but they are difficult to understand and not intend to be used by the common language programmer.

As a consequence, almost every language has an obscure point that gets different interpretations by different compiler writers. Then a program that works when compiled by one compiler may not work when compiled by other compilers.

1.2 History

The first programming language was designed in 1945 by Konrad Zuse, who build the first general purpose digital computer in 1941 [13]. The language was called plankalkül and only recently it has been implemented [12].

Fortran was the first high level programming language to be implemented. Its design began in 1953 and a compiler was released in 1957 [13]. Fortran, which means FORmula TRANslation, was designed to scientific and engineering computations. The language has gone through a series of modifications through the years and is far from retiring.

Algol (ALGOrithm Language) was also designed in the 1950's. The first version appeared in 1958 and a revision was presented in a 1960 report. This language was extensively used mainly in Europe.

Algol was one of the most (or the most) influential language already designed. Several languages that have been largely used as C, C++, Pascal, Simula, Ada, Java and Modula-2 are its descendents. Algol introduced `begin-end` blocks, recursion, strong typing, call by name and by value, structured iteration commands as `while`, and dynamic arrays whose size is determmined at run time.

COBOL, which stands for COmmon Business Oriented Language, was designed in the late 1950's for business data processing. This language was adequate for this job at that time but today it is obsolete. It has not left any (important) descendent and is being replaced by newer languages.

Lisp (LISt Processing) was designed in 1960 by John MacCarthy. The basic data structure of the language is the list. Everything is a list element or a list, including the program itself. Lisp had greatly influenced programming language design since its releasing. It introduced garbage collection and was the first functional language.

Simula-67 was the first object-oriented language. It was designed in 1967 by a research team in Norway. Simula-67 descends from Simula which is an Algol extension. Simula was largely used to simulation in Europe.

Alan Kay began the design of Smalltalk in the beginning of 1970's. The language was later refined by a group of people in XEROX PARC resulting in Smalltalk-76 and Smalltalk-80, which is the current standard. Smalltalk influenced almost every object-oriented language designed in the last two decades.

The first computers were programmed by given as input to them the bytes representing the instructions (machine code). Then, to add 10 to register R0 one would have to give as input a number representing the machine instruction "`mov R0, 10`". These primitive machine languages are called "first generation languages".

Then the first assembly languages appeared. They allowed to write instructions as

```
mov R0, 10
add R0, R1
```

that were late translated to machine code by a compiler. The assembly languages are second generation languages.

The third generation was born with Plankalkül[1] and encompasses languages as Fortran, Algol, Cobol, PL/I, Pascal, C, and Ada. These languages were the first to be called "high-level languages".

Fourth generation languages have specific purposes as to handle data bases. They are used in narrow domains and are very high level. These languages are not usually discussed in programming language books because they lack interesting and general concepts.

There is no precise definition of language generation and this topic is usually not discussed in research articles about programming languages. The fact a language belongs to the fourth or fifth generation does not make it better than a third or even a second generation language. It may only be a different language adequate to its domain.

1.3 Reasons to Study Programming Languages

Why should one take a programming language course ? Everyone in Computer Science will need to choose a programming language to work since algorithms permeate almost every field of Computer Science and are expressed in languages. Then, it is important to know how to identify the best language for each job. Although more than eight thousand languages have been designed from a dozen different paradigms, only a few have achieved widespread use. That makes it easier to identify the best language for a given programming project. It is even easier to first identify the best paradigm for the job since there are a few of them and then to identify a language belonging to that paradigm. Besides this, there are several motives to study programming languages.

[1] It seems the third generation was born before the second !

- It helps one to program the language she is using. The programmer becomes open to new ways of structuring her program/data and of doing computation. For example, she may simulate object-oriented constructs in a non-object oriented language. That would make the program clearer and easier to maintain. By studying functional languages in which recursion is the main tool for executing several times a piece of code, she may learn how to use better this feature and when to use it. In fact, the several paradigms teach us a lot about alternative ways of seeing the world which includes alternative ways of structuring data, designing algorithms, and maintaining programs.

- It helps to understand some aspects of one's favorite language. Programming language books (and this in particular) concentrates in concepts rather than in particularities of real languages. Then the reader can understand the paradigm/language characteristics better than if she learns how to program a real language. In fact, it is pretty common a programmer ignore important conceptual aspects of a language she has heavily used.

- It helps to learn new languages. The concepts employed in a programming language paradigm are applied to all languages supporting that paradigm. Therefore after learning the concepts of a paradigm it becomes easier to learn a language of that paradigm. Besides that, the basic features of programming languages such as garbage collection, block structure, and exceptions are common to several paradigms and one needs to learn them just one time.

1.4 What Characterizes a Good Programming Language ?

General purpose programming languages are intended to be used in several fields such as commercial data processing, scientific/engineering computations, user interface, and system software. Special purpose languages are designed to a specialized field and are awkward to use everywhere. Smalltalk, Lisp, Java and C++ are general purpose languages whereas Prolog, Fortran, and Cobol are special ones. Of course, a general purpose language does not need to be suitable for all fields. For example, current implementations of Smalltalk makes this language too slow to be used for scientific/engineering computations.

Now we can return to the question "What characterizes a good programming language ?". There are several aspects to consider, explained next.

- The language may have been designed to a particular field and therefore it contains features that make it easy to build programs in that field. For example, AWK is a language to handle data organized in lines of text, each line with a list of fields. A one-line program in AWK may be equivalent to a 1000-line program in C++.

- Clear syntax. Although syntax is considered a minor issue, an obscure syntax makes source code difficult to understand. For example, in C/C++ the statement
  ```
  *f()[++i] = 0["ABC" + 1];
  ```
  is legal although unclear.

- Orthogonality of concepts. Two concepts are orthogonal if the use of one of them does not prevent the use of the other. For example, in C there are the concept of types (`int`, `float`, user defined structs[2]) and parameter passing to functions. In the first versions of this language, all types could be passed to functions by value (copying) except structs. One should always pass a pointer to the struct instead of the structure itself.

[2]Structs are the equivalent of records in other languages as Pascal.

Lack of orthogonality makes the programmer use only part of the language. If she has doubts about the legality of some code, she usually will not even try to use that code [16]. Besides being underused, a non-orthogonal language is more difficult to learn since the programmer has to know if two concepts are valid together or not. In fact, non-orthogonality is an obstacle to learning greater than the language size. A big and orthogonal language may be easier to learn than a median-size and non-orthogonal language.

On the other side, full orthogonal languages may support features that are not frequently used or even not used at all. For example, in some languages values of any type can be passed to procedures by reference and by value. Then it is allowed to pass an array to a procedure by value. This is completly unnecessary[3] and maked the language harder to implement.

- Size of the language. Since a big language has too many constructs, there is a high probability it will not be orthogonal. Since it is difficult for a programmer to master all the peculiarities of a big language, she will get more difficult-to-fix compiler errors and therefore she will tend to use only part of the language. In fact, different people will use different language subsets, a situation that could be avoided by designing a main language with several specialized languages based in it [6].

  It is hard to implement a big language not only because of the sheer number of its constructs but mainly because these constructs interact with each other in many ways. For example, in Algol 68 a procedure can return values of any type. To the programmer, to declare a procedure returning a type is as easy as to declare it returning other type. However the compiler may need to treat each type separately when doing semantic analysis and generating code. Two types may need two completely different code generation schemes. Then the compiler has to worry about the iteration between "return value" and "type". The complexity of this iteration is hidden from the programmer.

  Because of problems as described above, big language compilers frequently are slow, expensive or flawed. Languages are troublesome to specify unambiguously and having a big language make things worse. The result may be different compilers implementing the same language constructs in different ways.

  On the other side, small languages tend to lack important features such as support to separate compilation of modules. This stimulates each compiler writer to introduce this feature by herself, resulting in dozen of language dialects incompatible to each other.

When selecting a language to use, one should also consider factors external to the languages such as the ones described below.

- Availability of good compilers, debuggers, and tools for the language. This may be the determinant factor in choosing a language and it often is. Several good languages are not largely used because they lack good tools. One of the reasons Fortran has been successful is the existence of very good optimized compilers.

- Portability, which is the ability to move the source code of a program from one compiler/machine to another without having to rewrite part of it. Portability involves a series of factors such as the language itself, the language libraries, the machine, and the compiler. We will briefly explain each of these topics.

  Badly specified languages free compiler writers to implement ambiguously defined constructs in different ways. A library furnished with a compiler may not be available with another one, even

[3] The author of this book has never seen a single situation in which this is required.

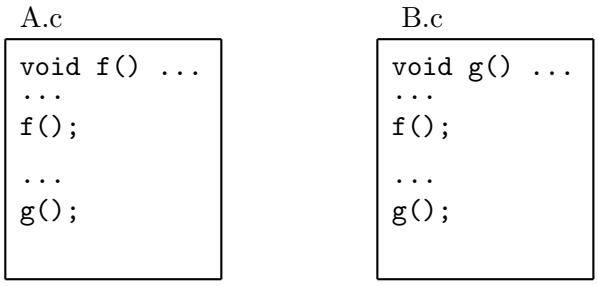

Figura 1.1: Two files of a C program

in the same machine. Differences in machines such as byte ordering of integers or error signaling may introduce errors when porting a program from one machine to another. For example, the code

```
while ( w->value != x && w != NULL )
  w = w->suc;
```

in C would work properly in old micro computers. If `w` is `NULL`, `w->value` would not cause a core dump since old micro computers do not support memory protection. Finally, different compilers may introduce language constructs by themselves. If a program uses these constructs, it will hardly be portable to other compilers.

- Good libraries. The availability of good libraries can be the major factor when choosing a language for a programming project. The use of suitable libraries can drastically reduce the development time and the cost of a system.

1.5 Compilers and Linkers

A compiler is a program that reads a program written in one language L_1 and translates it to another language L_2. Usually, L_1 is a high language language as C++ or Prolog and L_2 is assembly or machine language. However, C as been used as L_2. Using C as the target language makes the compiled code portable to any machine that has a C compiler. If a compiler produces assembler code as output, its use is restricted to a specific architecture.

When a compiler translates a L_1 file to machine language it will produce an output file called "object code" (usually with extension ".o" or ".obj"). In the general case, a executable program is produced by combining several object codes. This task is made by the linker as in the following example.

Suppose files "A.c" and "B.c" were compiled to "A.obj" and "B.obj". File "A.c" defines a procedure `f` and calls procedure `g` defined in "B.c". File "B.c" defines a procedure `g` and calls procedure `f`. There is a call to `f` in "A.c" and a call to `g` in "B.c". This configuration is shown in Figure 1.1. The compiler compiles "A.c" and "B.c" producing "A.obj" and "B.obj", shown in Figure 1.2. Each file is represented by a rectangle with three parts. The upper part contains the machine code corresponding to the C file . In this code, we use

```
call  000
```

for any procedure call since we do not know the exact address of the procedure in the executable file. This address will be determined by the linker and will replace `000`. The middle part contains the names and addresses of the procedures defined in this ".obj" file. Then, the file "A.obj" defines a procedure called `f` whose address is `200`. That is, the address of the first `f` machine instruction is

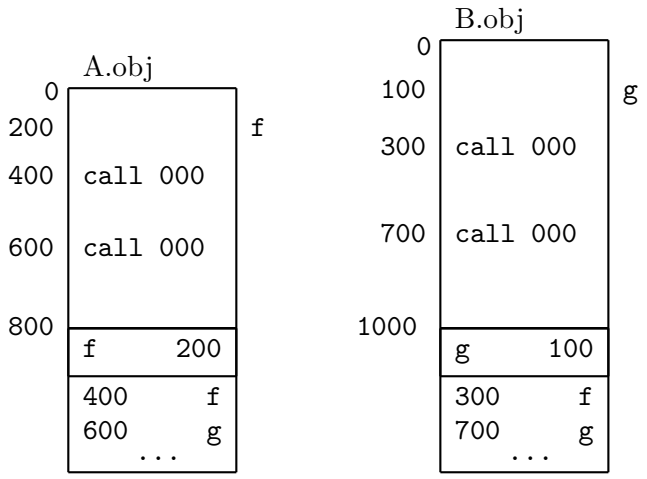

Figura 1.2: Object file configurations

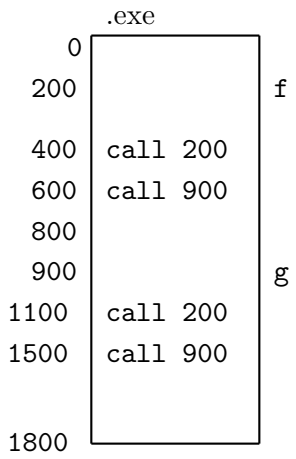

Figura 1.3: Executable file

200 in "A.obj". The lower rectangle part of "A.obj" contains the names of the procedures called in "A.obj" together with the call addresses. Then, procedure f is called at address 400 and g is called in address 600. To calculate the previous numbers (200, 400, 600) we assume the first byte of "A.obj" has address 0.

To build the executable program, the linker groups "A.obj" and "B.obj" in a single file shown in Figure 1.3. As "B.obj" was put after "A.obj", the linker adds to the addresses of "B.obj" the size of "A.obj", which is 800. So, the definition of procedure g was in address 100 and now it is in address 900 (100 + 800).

Since all procedures are in their definitive addresses, the linker adjusted the procedure calls using the addresses of f and g (200 and 900). File "A.c" calls procedures f and g as shown in Figure 1.1. The compiler generated for these calls the code

```
...
call 000   /* call to f. This is a comment */
```

```
...
call 000   /* call to g */
...
```

in which `000` was employed because the compiler did not know the address of `f` and `g`. After calculating the definitive addresses of these procedures, the linker modifies these calls to

```
...
call 200   /* call to f */
...
call 900   /* call to g */
...
```

To execute a program, the operating system loads it to memory and adjusts a register that keeps where the program code begins. Then a call to address `200` means in fact a call to this number plus the value of this register.

1.6 Run-Time System

In any executable program there are machine instructions that were not directly specified in the program. These instructions compose the run-time system of the program and are necessary to support the program execution. The higher level the language, the bigger its run-time system since the language nees to perform a lot of computations that were concealed from the programmers to make programming easier. For example, some languages support garbage collection that reduces errors associated to dynamic memory handling by not allowing explicit memory deallocation by the program (`free` of C, `delete` or `dispose` of other languages). The program becomes bigger because of the garbage collector but the programming is at a higher level when compared with languages with explicit memory deallocation.

Some responsibilities of run-time system are enumerated below.

- When a procedure is called, the RTS allocates memory to its local variables. When a procedure returns, the RTS frees this memory.
- The RTS manages the stack of called procedures. It keeps the return addresses of each procedure in known stack positions.
- When the program begins, the command line with which the program was called is passed to the program.[4] This command line is furnished by the operating system and passed to the program by the run-time system.
- Casting of values from one type to another can be made by the run-time system or the compiler.
- In object-oriented languages, it is the RTS that does the search for a method in message sends.
- When an exception is raised, the RTS looks for a "`when`" or "`catch`" clause in the stack of called procedures to treat the exception.
- In C++, the RTS calls the constructor of a class when an object is created and calls the destructor when the object is destroyed. In Java, the RTS one calls the constructor when the object is created.

[4] In C++ the command line is handled by the arguments `argc` and `argv` of function `main`. In Java, static method `main` of the main class has an array parameter with the arguments.

- The RTS does the garbage collecting.

Part of the run-time system is composed by compiled libraries and part is added by the compiler to the compiled code. In the first case is the algorithm for garbage collection and the code that gets the command line from the operating system. In the second case are all other items cited above.

1.7 Interpreters

A compiler generates machine code linked by the linker and executed directly by the computer. An interpreter generates bytecodes that are interpreted by the interpreter itself. Bytecodes are instructions for a virtual machine. In general, the bytecodes are closely related to the language the interpreter works with.

There are several tradeoffs in using compilers/linkers or interpreters, discussed in the following items.

1. It is easy to control the execution of a program if it is being interpreted. The interpreter can confer the legality of each bytecode before executing it. Then the interpreter can easily prevent illegal memory access, out-of-range array indeces, and dangerous file system manipulations;

2. It is easier to build a debugger for a interpreted program since the interpreter is in control of the execution of each bytecode and the virtual machine is much simpler than a real computer.

3. Interpreters are easier to build than compilers. First they translate the source code to bytecodes and compilers produce machine code. Bytecodes are in general much simpler than machine instructions. Second, interpreters do not need a linker. The linking is made dynamically during the interpretation. Third the run-time system is inside the interpreter itself, written in a high-level language. At least part of the run-time system of a compiled program needs to be in machine language.

4. The sequence Edit-Compile to bytecodes-Interpret is faster than the sequence Edit-Compile-Link-Execute required by compilers. Interpreters do not need linking and changes in one of the files that compose the program do not demand the recompilation of other files. To understand this point consider a program is composed by several files already compiled to bytecodes. Suppose the programmer does a small change in one of the files and asks the interpreter to execute the program. The interpreter will translate the changed file to bytecodes and will interpret the program. Only the modified file should be recompiled.

   If compilation is used, a small change in a file may require recompilation of several files that depend on this. For example, if the programmer changes the value of a constant declared as

   ```
   const Max = 100;
   ```

   in C++ or other languages, the compiler should recompile all files that use this constant.[5]

   After compiling all the files of a program, they are linked to produce the executable file. Compared with the interpretation environments, compilation takes more programmer's time. Interpreters are fast to respond to any programmer's change in the program and are heavily used for rapid prototyping.

5. Compilers produce a much more faster code than interpreters since they produce code that is executed directly by the machine. Usually the compiled code is 10 to 20 times faster than the interpreted equivalent code.

[5] In C++, the compiler should recompile all files that include the header file that defines this constant.

From the previous comparison we conclude that interpreters are best during the program's development phase. In this step it is important to make the program run as fast as possible after changing the source code and it is important to discover as many run-time errors as possible. After the program is ready it should run fast and therefore it should be compiled.

1.8 Equivalence of Programming Languages

There are thousands of programming languages employing hundreds of different concepts such as parallel constructs, objects, exception, logical restrictions, functions as parameters, generic types, and so forth. So one should wonder if there is a problem that can be solved by one language and not by another. The answer is no. All languages are equivalent in their *algorithm* power although it is easier to implement some algorithms in some languages than in others.

There are languages

- supporting parallel constructs. Two or more pieces of the same program can be executed in two or more machine processors;
- without the assignment statement;
- without variables;
- without iterations statements such as `while`'s and `for`'s: the only way to execute a piece of code more than one time is through recursion;
- without iterations statements or recursion. There are pre-defined ways to operate with data structures as arrays that make `while`/`for`/recursion unnecessary.

Albeit these differences, all languages are equivalent. Every parallel program can be transformed in a sequential one. The assignment statement and variables may be discarded if all expressions are used as input to other expressions and there are alternative ways to iterative statements/recursion in handling arrays and other data structures that require the scanning of their elements.

Every program that uses recursion can be transformed in another without it. This generally requires the use of a explicit stack to handle what was once a recursive call.

In general there is no doubt if a given programming language is really a language. If it has a iteration statement as `while`'s (or recursion) and some kind of `if` statement, it is a real language. However, if it is not clear whether a definition `L` is a language, one can build an interpreter of a known language `K` in `L`. Now all algorithms that can be implemented in `K` can also be implemented in `L`. To see that, first implement an algorithm in `K` resulting in a program `P`. Then give `P` to the interpreter written in `L`. We can consider that the interpreter is executing the algorithm. Therefore the algorithm was implemented in `L`. Based in this reasoning we can consider interpreters as programs capable of solving every kind of problem. It is only necessary to give them the appropriate input.

The mother of all computeres is the Turing Machine, devised in 1936. In fact, computability has been defined in terms of it by the Church-Turing thesis:

> Any computable function can be computed by the Turing Machine

or, alternatively,

> Any mechanical computation can be made in a Turing Machine

In fact, this "thesis" is an unproven hypothesis. However, there has never been found any algorithm that cannot be implemented by a Turing machine or any programming language.

There are theoretic limitations in what a language can do. It is impossible, for example, to devise an algorithm `P` that takes the text of another algorithm `Q` as input and guarantees `Q` will finish. To prove that, suppose `P` exists and `P(text of A)` returns `true` if `A` finish its execution, `false` otherwise. If `Q` is defined as

```
proc Q()
  begin
  while P(text of Q) do
    ;
  end
```

what will happen ? The `;` after the `while` is the null statement.

If `P(text of Q)` returns `true` that means `P` says `Q` does stop. But in this case `Q` will not stop because the `while` statement will be equivalent to

```
    while true do
      ;
```

If `P(text of Q)` returns `false` that means `P` says `Q` will never stop. But in this case `Q` will stop.

The above reasoning leads to nonsense conclusions. Then we conclude `P` does not exist since all the other logical reasonings are correct.

If `Q` takes one or more parameters as input, the reasoning above has to be slightly modified. First, all of the parameters should be represented as a single integer number — this can be done although we will no show how. Assume that the language permits integers of any length. Procedures are also represented as integer numbers — to each procedure is associated a single integer.

Now suppose function `P(n, d)` returns true if the procedure associated to number `n` halts[6] when taking the integer `d` as parameter. `P(n, d)` returns false otherwise. If `Q` is defined as

```
proc Q(d)
  begin
  while P(d, d) do
    ;
  end
```

and called as `Q(number of Q)`, there is a paradox. If `Q(number of Q)` halts, `P(d, d)` returns true and the `while` statement above will never stop. Therefore `Q(number of Q)` would not halt. If `Q(number of Q)` does not halt, `P(d, d)` returns false and the `while` statement above will be executed a single time. This means `Q` will soon halts. Contradiction, `P` cannot exists.

In the general case, an algorithm cannot deduce what other one does. So, an algorithm cannot tell another will print "3" as output or produce 52 as result. It is also not possible for an algorithm to decide whether two programs or algorithms are equivalent; that is, whether they always produce the same output when given the same input.

All these restrictions to algorithm power apply to algorithms and not to programs implemented in a computer. The former assumes memory is infinite and in computers memory is always limited. It is therefore possible to devise a program `S` that takes a program `P` as input and discover if `P` will stop when executed in a given machine. Program `S` considers all bits of the machine memory (including registers,

[6]Finish its execution at some time, it does not matter when.

internal flags, I/O ports) as as single number describing the current memory state. Almost every `P` statement modifies this number even if only the program counter or some internal flag.[7] Program `S` can simulate `P` execution and record the numbers resulted after the execution of each statement. This would result in a list

$\mathtt{n}_1$, $\mathtt{n}_2$, $\mathtt{n}_3$, ... $\mathtt{n}_k$

If a number is equal to same previous number (e.g. $\mathtt{n}_k == \mathtt{n}_2$) then program `P` has an infinite loop. The sequence

$\mathtt{n}_2$, $\mathtt{n}_3$, ... $\mathtt{n}_k$

will be repeated forever. This happen because computers are deterministic: each computer state (number) is followed by exactly one other state.

If this sequence reaches a number corresponding to the last program statement, the program stops. Assume all input was put in the computer memory before the execution started.

[7] In fact, the only statement we found that does not modify anything is "L: goto L"

Capítulo 2

Basic Concepts

This chapter explains general concepts found in several programming languages paradigms such as strong and dynamic typing, modules, garbage collection, scope, and block structure. In the following sections and in the remaining chapters of this book we will explain concepts and languages using a Pascal-like syntax as shown in Figure 2.1. To explain each concept or language we will add new syntax to the basic language that will be called S. We will describe only a few constructs of S since most of them can be understood easily by any programmer.

- A procedures in S is declared with the keyword `proc` and it may have return value type as procedure `fat` in the example. In this book, the word procedure is used for subroutine, routine, function, and subprogram.
- Keyword `return` is used to return the value of a procedure with return value type (in fact, a function). After the execution of this statement the procedure returns to the callee.
- `begin` and `end` delimits a group of statements inside a procedure, a command `for`, or `while`. The `if` statement does not need `begin-end` since it finishes with `endif`.
- The assignment statement is "`=`". Language S uses "`==`" for equality comparison.
- The execution of a program begins in a procedure called `main`.
- There are four basic types in S: `integer`, `boolean`, `real`, and `string`.

2.1 Types

A type is a set of values together with the operations that can be applied on them. Then, type `integer` is the set of values -32768, ... 0, 1, ... 32767 plus the operations

```
+  -  *  /  %  and  or
```

in which `%` is the remainder of division and `and` and `or` are applied bit by bit.

2.1.1 Static and Dynamic Type Binding

A variable can be associated to a type at compile or run time. The first case occurs in languages requiring the type in variable declarations as C++, Java and Pascal:

```
var i : integer;
```

These languages are said to support static type binding.

```
  { global variable }
const max = 100;

proc fat( n : integer ) : integer
  { return the factorial of n }
begin
if n <= 1
then
  return 1;
else
  return n*fat(n-1);
endif
end

proc print( n : integer; s : String )
  var i : integer;
begin
for i = 1 to n do
  write(s);
end

proc main()
    { program execution starts here }
  var j, n : integer;
begin
write("Type a number");
read(n);
if n == 0  or n > max
then
  n = fat(1);
endif
j = 1;
while j < n do
  begin
  print(j, "Hello");
  write("*****");
  j = j + 1;
  end
end
```

Figura 2.1: A program in language S

Dynamically typed languages do not allow types to be specified in variable declarations. A variable type will only be know at run time when the variable refers to some number/string/struct or object. That is, the binding variable/type is dynamic. A variable can refer to values of any type since the variable itself is untyped. See the example below in which `b` receives initially a string and then a number or array.

```
  var a, b;
begin
a = ?;
b = "Hello Guy";
if a <= 0
then
  b = 3;
else
      { array of three heterogeneous elements }
  b = #( 1.0,   false,  "here" );
if a == 0
then
  b = b + 1;
else
  write( b[1], " ", b[3] );    { 1 }
end.
```

In this example, if the `?` were replaced by

a) -1, there would be a run-time error;

b) 0, there would be no error;

c) 1, there would not be any error and the program would print

```
1.0  here
```

Since there is no type, the compiler cannot enforce the operations used with a variable are valid. For example, if `?` is replaced by `-1`, the program tries to apply operation `[]` on an integer number — `b` receives `3` and the program tries to evaluate `b[1]` and `b[3]`. The result would be a run-time error or, more specifically, a *run-time type error*. A *type error* occurs whenever one applies an operation on a value whose type does not support that operation.

Static type binding allows the compiler to discovers some or all type errors. For example, in

```
  var s : string;
begin
s = s + 1;
...
end
```

the compiler would sign an error in the first statement since type `string` does not support operation "`+`" with an integer value.

It is easier for a compiler to generate efficient code to a static type binding language than to a dynamically typed one. In the first case, a statement

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

```
proc search(v, n, x)
    { searchs for x in v from v[0] to v[n-1]. Returns the first i such
      that v[i] == x or -1 if x is not found in v. }
  var i;
begin
i = 0;
while i < n do
  if v[i] == x
  then
    return i;
  else
    i = i + 1;
  endif
return -1;
end
```

Figura 2.2: A polymorphic procedure

results in a few machine statements like

```
mov a, b
add a, c
```

In a dynamically typed language the compiler would generate code to:

1. test if `b` and `c` types support operation `+`;
2. select the appropriate operation to execute. There are at least three different operations that can be used in this example: `+` between integers, reals, and strings (to concatenation).
3. execute the operation.

Clearly the first generated code is much faster than the second one. Although dynamically typed languages are unsafe and produce code difficult to optimize (generally very slow), people continue to use them. Why ? First these languages free the programmer from the burden of having to specify types, which makes programming lighter. On *may* think more abstractly since one of the programming concepts (types at compile time) has been eliminated.[1]

Other consequence of this more abstract programming is polymorphism. A polymorphic variable can assume more than one type during its lifetime. So, any untyped variable is polymorphic. Polymorphic variables lead to polymorphic procedures which have at least one polymorphic parameter. A polymorphic value has more than one type, as `NULL` in C++, `null` in Java or `nil` in other languages.

Polymorphic procedures are important because their code can be reused with parameters of more than one type. For example, procedure `search` of Figure 2.2 can be used with arrays of several types as `integer`, `boolean`, `string`, or any other that support the operation "`==`". `search` can even be used with heterogeneous arrays in which the array elements have different types. Then the code of `search` was made just one time and reused by several types. In dynamic typed languages polymorphism comes naturally without any further effort from the programmer to create reusable code since any variable is polymorphic.

[1] Although it is pretty common programmers to introduce types in their variable names as `aPerson`, `aList`, and `anArray`.

2.1.2 Strong and Static Typing

A strongly typed language prevents any run-time type error from happening. Then a type error is caught by the compiler or at run time before it happens. For example in a language with static type binding, the compiler would sign an error in

```
a = p*3;
```

if `p` is a pointer. In a dynamically typed language this error would be pointed at run time before it happens. Languages with dynamic typing are naturally strongly typed since the appropriate operation to be used is chosen at run time and the run-time system may not find this operations resulting in a run-time type error. In a non-strongly typed language the statement above would be executed producing a nonsense result. Some authors consider a language strongly typed only if type errors are caught at compile time. This appear not to be the dominant opinion among programming language people. Therefore we will consider in this book that a language is strongly typed if it requires type errors to be discovered at compile or run time.

A language is statically typed if all type errors can be caught at compile time. Then all statically typed languages are also strongly typed. In general languages in which variables are declared with a type (Pascal, C++, Java) are statically typed. The type error in the code

```
var i : integer;
begin
i = "hello";
...
end
```

would be discovered at compile time.

The definitions of static and strong typing are not employed rigorously. For example, C++ is considered a statically typed language although a C++ program may have type errors that are not discovered even at run time. For example, the code

```
int *p;
char *s = "Hi";

p = (int *) s;
*p =  *p*3;
```

declares a pointer `p` to `int` and a pointer `s` to `char`, assigns `s` to `p` through a type conversion, and multiply by `3` a memory area reserved to a string. In this last statement there is a type error since a memory area is used with an operation that does not belong to its type (string or `char *`).

2.2 Block Structure and Scope

Algol-like languages (like Pascal, Modula, and Ada) support the declaration of procedures inside other procedures as shown in Figure 2.3 in which `Q` is inside `P`. Block structure was devised to restrict the scope of procedures. *Scope* of an identifier is the program region in which it is defined. That is, the region in which it can potentially be used. In general, the scope of a global variable such as `max` of Figure 2.3 extends from the point of declaration to the end of the file. Then `max` can be used in `P` and `Q`. The scope of a local variable of a procedure `X` is the point where it is declared to the end of `X`. This same rule applies to local procedures. Then, the scope of variable `i` of `P` is from `2` to `6` and the scope of `Q` is from `3` to `6`. Variable `k` can be used only inside `Q`.

```
  var max : integer;    { 1 }

proc P( n : integer )
  var i : integer;      { 2 }
  proc Q()              { 3 }
    var k : integer;    { 4 }
  begin { Q }
  ...
  end   { Q }           { 5 }
begin   { P }
...
end     { P }           { 6 }
...
                        { 7 - end of file }
```

Figura 2.3: An example of procedure nesting

A procedure may declare a local variable or procedure with name equal to an outer identifier. For example, one could change the name of variable `k` of `Q` to `max`. Inside `Q`, `max` would mean the local variable `max` instead of the global `max`. Nearest declared identifiers have precedence over outer scope ones.

Visibility of an identifier is the program region where it is visible. In the example above with `k` changed to `max`, the scope of the global `max` is from `1` to `7`. However, `max` is not visible inside `Q`.

Lifetime of a variable is the time interval during the program execution in which memory was allocated to the variable. Then, the lifetime of a global variable is all program execution whereas a variable local to a procedure is created when the procedure is called and destroyed when it returns.

According to the definitions above, scope and visibility are determined at compile time and lifetime at run time. However, in some languages the scope of a variable varies dynamically at run time. When a procedure is called, its local variables are created in a stack. If the procedure calls another ones, these can access its local variables since they continue to be in the stack. The local variables become invisible only when the procedure returns. This strange concept is called dynamic scope.

An example of it is shown in Figure 2.4. The program begins its execution in the `main` procedure where `Q` is called after statement "`max = 5`". At the end of `Q`, after the `if` statement, procedure `P` is called resulting in the call stack of Figure 2.5 (a). Inside `P` the variables `max`, `n`, and `i` are visible. Then it is fine `p` use `n` in the `for` statement. After `P` returns and `Q` returns the execution continues in the `main` procedure and `P` is called at point `{ 1 }`, resulting in the call stack of Figure 2.5 (b). Now `P` tries to use the undefined variable `n` resulting in a run-time error. Note that could have been an error even if `n` existed at that point because `p` might have tried to use `n` as if it had a different type as `string`.

Dynamic scope is intrinsically unsafe as shown in the previous example. It is dangerous to use a variable that is not in the static scope such as `n` in `P`. So this should never be done. But then dynamic scope is unuseful since it degenerates into static scope ! Why then do people use it ? By our knowledge it is used only because it makes it easier to build interpreters to the language: variables are only checked at run time using an already existing stack of local variables.

Now we return to block structures. An example is in Figure 2.6 whose correspondent tree is in Figure 2.7. In this tree, an arrow from `A` to `B` means `B` is inside `A`. The variables visible in a procedure `X` are the global variables plus all local variables of the procedures in the path from `X` to the root of

```
proc P()
  var i : integer;
begin
for i = 1 to n do
  write(i);
end

procedure Q()
  var n : integer;
begin
if max < 10
then
  n = max;
else
  n = 10;
endif
P();
end;

proc main()
  {  main procedure. Program execution starts here }
  var max : integer;
begin
max = 5;
Q();
P();   { 1 }
end.
```

Figura 2.4: Example of dynamic scope

(a) (b)

Figura 2.5: Call stack

```
var max : integer;

proc A()
  var min : integer;
  proc B() : char
    var g : integer;
    proc C()
      var ok : boolean;
    begin { C }
    ...
    end  { C }
  begin { B }
  ...
  end  { B }
  proc D()
    var j : integer;
  begin { D }
  ...
  end  { D }
begin { A }
...
end  { A }
```

Figura 2.6: Nesting of procedures

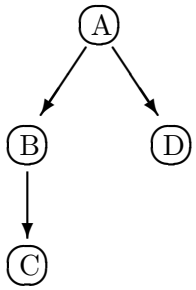

Figura 2.7: Nesting tree

the tree. Then, the variables visible in `C` are the global ones plus those of `C` itself, `B`, and `A`.

The objective of block structure is to create abstraction levels by hinding procedures from the rest of the program. Albeit this noble goal, there are several problems with block structures:

- it requires extra work to find the correct procedure nesting;
- it becomes difficult to read the source code since the procedure header with the local variables and formal parameters is kept far from the procedure body;
- the lowest level procedures are, in general, the deepest nested ones. After all they are the building blocks for the procedures higher in the tree. This is a problem because low level procedures are, in general, the most generic ones. Probably the deepest nested procedures will be needed in other parts of the program. And it may be not easy to move them to outside all nestings because they may use local variables of the procedures higher in the tree. For example, procedure `C` in Figure 2.6 may use variable `g` of `B` and `min` of `A`. When moving `C` outside all nestings, variables `g` and `min` must be introduced as parameters. This requires modifying all calls to `C` in `B`.
- it becomes confusing to discover the visible variables of a procedure. These variables are clearly identified with the help of a tree like that of Figure 2.7 but difficult to identify by reading the source code.

2.3 Modules

A module or package[2] is a set of resources such as procedures, types, constants, classes[3], and variables. Each resource of a module can be public or private. Public resources can be used by other modules that import this one through a special command discussed next. Private resources are only visible inside the module itself.

Modules are declared as shown in Figure 2.8 which presents module `Window` that:

- imports modules `WinData` and `StringHandling`;
- defines in the public section two procedures (`makeWindow` and `closeAll`) and two constants (`maxWidth` and `maxHeight`).

By importing module `StringHandling`, all resources defined in the public section of this module become visible in module `Window`. Then type `String` defined in the public section of `StringHandling` can be used in the header of procedure `makeWindow`.

Only procedure headers (without the body) are put in the public section since to use a procedure one needs only its name and parameter/return value types.

Suppose variable `Max` is defined in module `WinData` *and* in `StringHandling`. Now an ambiguity arises in the use of `Max` in the module `Window` procedures. It could refer to variable `Max` of both modules. The language can consider an error to import two modules that have resources with the same name. Or it may require the use of a module qualification before `Max` such as

```
WinData.Max
```

or

```
StringHandling.Max
```

The language may also require this syntax for all imported resources, not only the ambiguous ones.

[2] Its Ada and Java names.

[3] To be seen in a chapter about object-oriented programming.

```
module Window

import WinData, StringHandling;

public:
    { only procedure headers appear in the public section }
  proc makeWindow( w : WindowData;
                   name : String );
  const maxWidth  = 1024,
        maxHeight = 800;
  proc closeAll();

private:

  const max = 16;
  var AllWindows : array[1..max] of WindowData;

  proc makeWindow( w : WindowData;
                   name : String )
      var i : integer;
    begin
    ...
    end

  proc closeAll()
      var i, k : integer;
    begin
    ...
    end
```

Figura 2.8: A module `Window`

This would make typing difficult but the program clearer since the programmer would know immediately the module each resource comes from. However in general the programmer knows from where each resource comes from, mainly in small programs and when using known libraries. In this case a verbose syntax like "`WinData.Max`" is less legible than a lighter one like "`Max`".

Modules have several important features, described next.

- When a module `B` imports module `A`, `B` can use only resources of the public section of `A`. That means any changes in the private part of `A` will not affect `B` in any way. In some language, `B` not even needs to be recompiled when the private part of `A` in changed. This feature is used to implement data abstraction, also called information hiding. Some languages (e.g. Ada) allow declarations like

  ```
  type Window is private;
  proc drawWindow( w : Window );
  ```

  in the public module section. `Window` is a type *defined* in the private section although *declared* as public. To define a variable is to give an order to the compiler generate code to allocate memory for the variable at run time. So definition requires all information about the variable to be specified. To declare a variable is to give its name and type so that type checking is possible.

  Modules importing module this module `Window` can declare variables of type `Window` but they cannot apply any operation on `Window` variables. For example, if `Window` is a record (C struct or C++ class), these modules cannot select a field of a variable of this type (like "`win.icon`").

- A program can (should) be divided in modules that can be separately compiled and understood. One may only know the public section of a module in order to use it. Then modules work as abstraction mechanisms reducing the program complexity. A well-designed module restricts the dependency from other modules to a minimum. Then it can largely be understood independently from the rest of the program.

2.4 Exceptions

Exceptions are constructs that allow the programmer to separate error signaling from error treatment, thus making programs more readable. Usually there should be a test after each statement that can result in error. This test should verify if the statement was successful. It it was not, code to try to correct the error should be executed. If correction is not possible the program should be terminated.

As an example, the code of Figure 2.9 builds a window from several dynamically allocated parts. After each allocation there is a test to verify if it was successful. "`A.new()`" allocates memory for an instance of type `A`.

The code is populated with a lot of `if`'s to test the success of memory allocation. These `if`'s do not belong to the functional part of the algorithm; that is, that part that fulfills the algorithm specification. The `if`'s are a sort of auxiliary part that should best be kept separate from the main algorithm body. This can be achieved with exceptions as shown by Figure 2.10 which presents another implementation of procedure `buildWindow`. Each call to `new` either returns the required memory or *raises* an exception. An exception is raised by a statement like

```
raises Storage_Error:
```

that transfers the execution to a `when` clause put after keyword `exception` — see the example. In other words, when there is not enough free memory procedure `new`, instead of returning `nil` (or `NULL`), raises exception `Storage_Error`. The control is transferred to the line following

```
proc buildWindow() : Window
  var w : Window;
      op : Option;
      b  : Button;
begin
w = Window.new();
if w == nil then return nil; endif
init( w, x1, y1, x2, y2 );
op = Option.new();
if op == nil then return nil; endif
init( op, "Are you sure ?", xop, yop );
setOption(w, op);
b = Button.new();
if b == nil then return nil; endif
init(b, ...);
setButton(w, b);
b = Button.new();
if b == nil then return nil; endif
init(b, ...);
setButton(w, b);
b = Button.new();
if b == nil then return nil; endif
init(b, ...);
setButton(w, b);
end
```

Figura 2.9: Code to build a window

```
proc buildWindow() : Window
  var w : Window;
      op : Option;
      b  : Button;
begin
w = Window.new();
init( w, x1, y1, x2, y2 );
op = Option.new();
init( op, "Are you sure ?", xop, yop );
setOption(w, op);
b = Button.new();
init(b, ...);
setButton(w, b);
b = Button.new();
init(b, ...);
setButton(w, b);
b = Button.new();
init(b, ...);
setButton(w, b);
exception
  when Storage_Error:
      { failure in memory allocations: not enough free memory }
    return nil;
  when Bad_Data:
      { invalid arguments to procedures }
    return nil;
end
```

Figura 2.10: Use of exceptions to catch memory allocation errors

`when Storage_Error:`

where the execution continues. The procedure then returns `nil`. Then to raise an exception is much like to make an unconditional `goto` to a label given by the `when` clause. The difference from a `goto` is that exceptions can be raised by a procedure and catch (or treated) by any other that is in the procedure call stack.

In the example of Figure 2.10, procedure `buildWindow` could also handle exception `Bad_Data`. This exception may be raised by procedures `init`, `setOption`, `setButton`, or by any procedure called by these ones.

In the general case, a procedure $\texttt{P}_1$ calls procedure $\texttt{P}_2$ that calls $\texttt{P}_3$ and so forth until $\texttt{P}_{n-1}$ calls $\texttt{P}_n$. If $\texttt{P}_n$ raises an exception `Storage_Error`, the run-time system (RTS) begins to search for a `when` clause

`when Storage_Error:`

in the procedures stacked, from $\texttt{P}_n$ to $\texttt{P}_1$. If one of such clauses is found, the statements following it are executed. If no such `when` clause is found, the program is terminated.

Another example of exceptions is in Figure 2.11 in which `main` is the procedure where the program execution begins. The second statement of `main` calls `q` that calls `r`. Depending on the value of `n`, `r` raises exception `Underflow` or `Overflow`. If `Underflow` is raised, the control jumps to the "`when Underflow`" clause of `q`. If `Overflow` is raised, `r` and `q` are skipped and the execution continues in the "`when Overflow`" clause of `main`.

The fourth statement of `p` calls `r`. If `r` raises `Underflow`, a run-time error is signaled and the program is terminated. The stack of called procedures contains only `main` and `r`, and none of these have a "`when Underflow`"clause. Note it does not matter `q` have this clause: it was not in the call stack when the exception was raised.

Then an exception can be raised and treated in some execution paths and not treated in others. This results in unsafe programs that can terminate unexpectedly. There are other problems with exceptions:

- they make the program difficult to understand. An exception is like a `goto` to a label unknown at compile time. The label will only be known at run time and it may vary from execution to execution;
- they require a closer cooperation among the modules of a program weakening their reusability. A module should be aware of internal details of other modules it uses because it needs to know which exceptions they can raise. That means a module should know something about the internal flow of execution of other modules breaking encapsulation;
- they can leave the system in an inconsistent state because some procedures are not terminated. Then some data structures may have non-initialized pointers, files may not have been closed, and user interactions could have been suddenly interrupted.

Although exceptions are criticized because of these problems, they are supported by major languages as Java, Ada, Eiffel, and C++. Exceptions have two main features:

- they separate the functional part of the program (that fulfill its specification) from error handling. This can reduce the program complexity because parts that perform different functions are kept separately;
- they save the programmer's time that would be used to put error codes in return values of procedures and to test for procedure failure after each call.

```
proc r( n : integer )
  var a : A;
begin
if n < 0
then
  raises Underflow;
endif
if n > Max
then
  raises Overflow;
endif
...
end { r }

proc q( n : integer )
begin
r(n);
...
exception
  when Underflow :
      { terminate the program }
    write("Underflow");
    exit(1);
end { q }

proc  main()
  { program starts here }
  var n : integer;
begin
read(n);
q(n);
read(n);
r(n):
exception
  when Overflow :
      { terminate the program }
    write("Overflow");
    exit(1);
end { main }
```

Figura 2.11: An example with exceptions

```
proc r( n : integer ) raises (Underflow, Overflow)
  var a : A;
begin
if n < 0
then
  raises Underflow;
endif
if n > Max
then
  raises Overflow;
endif
   { no exception is raised in following statements of r
     represented by ... }
...
end { r }

proc q( n : integer ) raises (Overflow)
begin
r(n);
...
exception
  when Underflow :
      { terminate the program }
    write("Error: underflow");
    exit(1);
end { q }

proc  main()
  { program starts here }
  var n : integer;
begin
read(n);
q(n);
read(n);
r(n):
exception
  when Overflow :
      { terminate the program }
    write("Error: overflow");
    exit(1);
end { main }
```

Figura 2.12: An example with safe exceptions

Besides that, it *may* be faster to use exception than to test the return values by hand.

Some languages support safe exceptions: the language guarantees at compile time that all exceptions raised at runtime will be caught by `when` clauses. These languages require that a procedure either treats the exceptions it may raise or specifies these exceptions in its interface. For example, Figure 2.12 shows the example of Figure 2.11 in a language with safe exceptions. After the header of each procedure there should appear a declaration

```
raises (E1, E2, ...)
```

with the exceptions `E1`, `E2`, ... that the procedure can raise. For example, `r` can raises exceptions `Underflow` and `Overflow`. The exceptions a procedure can raise are those

- the other procedures it calls can raise;
- it can raise itself;
- that are not treated by any `when` clause at its end.

Procedure `q` calls `r` and therefore could raise `Underflow` and `Overflow`. Since `q` has a clause "`when Underflow`", it can only raise exception `Overflow`. Procedure `main` calls `q` and `r` and therefore it can raise both `Overflow` and `Underflow`. Since `main` has only a `when Overflow` clause, it could raise an `Underflow` exception that would not be catch by any `when` clause. As the language is exception-safe, the compiler would issue an error in procedure `main`. To correct the program a "`when Underflow`" clause should be added to this procedure.

An alternative to using `raise-when` constructs is to use the `try` blocks as shown in the following example.

```
try
  begin
  read(n);
  q(n);
  end
catch ( Underflow )
  begin
  write("Underflow !");
  exit(1);
  end
catch( Overflow )
  begin
  write("Overflow !");
  exit(1);
  end
```

The commands that can raise an exception are put in the `try` block delimited by `begin-end`. The catch clauses are the equivalent of `when` clauses.

When using `when` clauses, there is no way of knowing which procedure command raised the exception. `try` blocks are best fitted in this case since they allow to test for exceptions in several points of the procedure.

2.5 Garbage Collection

Some languages do not allow the explicit deallocation of dynamic allocated memory by the program. That is, there is no command or procedure like `free` of C, `dispose` of Pascal, or `delete` of C++. These

languages are said to support *garbage collection*. This concept applies to languages with explicit or implicit dynamic memory allocation. Explicit allocation occurs in languages as Java, Ada, Smalltalk, C++, Pascal, and Modula-2 that have commands/functions to allocate dynamic memory as `new` or `malloc`. Implicit allocation occurs in Prolog or Lisp-like languages in which dynamic structures as lists shrink and grow automatically when necessary. When a list grows, new memory is automatically allocated to it by the run-time system.

The procedure that follows illustrates the basic point of garbage collection.

```
proc main()
  { do nothing --- just an example }

  var p, t : ^integer;
begin
p = integer.new();   { 1 }
t = p;               { 2 }
p = nil;             { 3 }
t = nil;             { 4 }
end
```

The type "`^integer`"is "pointer to `integer`" and memory for an integer value is allocated by "`integer.new()`". A memory block allocated by `new()` will only be freed by the garbage collector when no pointer points to it. In this example, two pointers will point to the allocated memory after executing statement `2`. After statement `3`, one pointer, `t`, will point to the memory. After statement `4`, there is no reference to the allocated memory and therefore it will never be used by the program again. From this point hereafter, the memory can be freed by the garbage collector.

The garbage collector is called to free unused memory from time to time or when the free available memory drops below a limit. A simple garbage collector (GC) works with the set of all global variables and all local variables/parameters of the procedures of the call stack. We will call this set *Live*. It contains all variables that can be used by the program at the point the GC is called. All memory blocks referenced by the pointers in *Live* can be used by the program and should not be deallocated. This memory may have pointers to other memory blocks and these should not be freed either because they can be referenced indirectly by the variables in *Live*. Extending this reasoning, no memory referenced direct or indirectly by variables in *Live* can be deallocated. This requirement suggests the garbage collector could work as follows.

1. First it finds and marks all memory blocks that can be reached from the set *Live* following pointers.

2. Then it frees all unmarked blocks since these will never be used by the program.

There are very strong reasons to use garbage collection. They become clear by examining the problems, described in the following items, that occur when memory deallocation is explicitly made by the programmer.

- A module may free a memory block still in use by other modules. There could be two live pointers `p` and `t` pointing to the same memory block and the program may execute "`dispose(p)`" or "`free(p)`" to liberate the block pointed to by `p`. When the memory block is accessed using `t`, there may be a serious error. Either the block may have been allocated by another call to `new` or the `dispose`/`free` procedure may have damaged the memory block by using some parts of it as pointers to a linked list of free blocks.

- The program may have memory leaks. That is, there could be memory blocks that are not referenced by the program and that were not freed with `dispose`/`delete`/`free`. These blocks will only be freed at the end of the program by the operating system.

- Complex data structures make it difficult to decide when a memory block can safely be deallocated. The programmer has to foresee all possible behaviors of the program at run time to decide when to deallocate a block. If a block is referenced by two pointers of the same data structure,[4] the program should take care of not deallocating the block twice when deallocating the data structure. This induces the programmer to build her own garbage collector. Programmer's made GC are known to be unsafe and slow when compared with the garbage collectors provided by compilers.

- Different program modules should closely cooperate in order to decide when deallocating memory [4]. This makes the modules tightly coupled thus reducing their reusability. Notice this problem only happens when dynamic memory is passed by procedure parameters from one module to another or when using global variables to refer to dynamic memory.

  This item says explicit memory deallocation breaks encapsulation. One module should know not only the interface of other modules but also *how* their data is structured and *how* their procedures work.

- Different deallocation strategies may be used by the modules of a program or the libraries used by it [4]. For example, the operation `deleteAll` of a Queue data structure[5] could remove all queue elements and free their memory. In another data structures such as `Stack` the operation `clearAll` similar to `deleteAll` of `Queue` could remove all stack elements but *not* free the memory allocated to them.

  The use of different strategies in the same program such as when to deallocate memory reduces the program legibility thus increasing errors caused by dynamic memory.

- Polymorphism makes the execution flow difficult to foresee. In an object-oriented language the compiler or the programmer does not know which procedure `m` (called method) will be called at run time in a statement like

  ```
  a.m(b)
  ```

  There may be several methods called `m` in the program and which method `m` is called is determined only at run time according to the value of `a`. Then the programmer does not know if pointer `b` will be stored by `m` in some non-local variable or if `m` will delete it. In fact, the programmer that wrote this statement may not even know all the methods `m` that may be executed at run time.

  For short, with polymorphism it becomes difficult to understand the execution flow and therefore it becomes harder to decide when it is safe to deallocate a memory block [4].

There are also arguments against garbage collection:

- it is slow;

- it causes long pauses during user interaction;

- it cannot be used in real-time systems in which there should be possible to know at compile time how long it will take to execute a piece of code at run time;

[4] Not necessaryly two pointers of the same struct or record. There may be dozen of structs/records with a lot of pointers linking them in a single composit structure.

[5] `Queue` could be a class of an object-oriented language or an abstract data type implemented with procedures.

- it makes difficult to use two languages in the same program. To understand this point, suppose a program was build using code of two languages: Eiffel that supports garbage collection and C++ that does not. A memory block allocated in the Eiffel code may be passed as parameter to the C++ code that may keep a pointer to it. If at some moment no pointer in the Eiffel code refers to this memory block, it may be freed even if the C++ pointer refer to it. There is no way in which the Eiffel garbage collector can know about the C++ pointer.

All of these problems but the last have been solved or ameliorated. Garbage collectors are much faster today than they were in the past. They usually spend `10` to `30%` of the total program execution time in object-oriented languages and from `20` to `40%` in Lisp programs. When using complex data structures, garbage collection can be even faster than manual deallocation.

Research in garbage collection has produced collectors for a large variety of tastes. For example, incremental GC do not produce long delays in the normal processing and there are even collectors used in real-time systems.

Capítulo 3

Linguagens Orientadas a Objeto

3.1 Introdução

Orientação a objetos utiliza classes como mecanismo básico de estruturação de programas. Uma classe é um tipo composto de variáveis (como *records* e *structs*) e procedimentos. Assim, uma classe é uma extensão de *records*/*structs* com a inclusão de comportamento representado por procedimentos. Um exemplo de declaração de classe está na Figura 3.1 que declara uma classe `Store` com procedimentos `get` e `put` e uma variável `n`. Na terminologia de orientação a objetos, `get` e `put` são métodos e `n` é uma variável de instância.

Uma variável da classe `Store`, declarada como

```
var s : Store;
```

é tratata como se fosse um ponteiro na linguagem S. Assim, deve ser alocada memória para `s` com a instrução

```
s = Store.new();
```

Esta memória é um *objeto* da classe `Store`. Um objeto é o valor correspondente a uma classe assim como `3` é um valor do tipo `int` e "`Alo !`" é um valor do tipo `string`. Objetos só existem em execução e classes só existem em tempo de compilação,[1] pois são tipos. Classes são esqueletos dos quais são criados objetos e variáveis referem-se a objetos. Então o objeto referenciado por `s` possui uma variável `n` e dois métodos.

Os campos de um `record` de Pascal ou `struct` de C são manipulados usando-se "`.`" como em "`pessoa.nome`" ou "`produto.preco`". Objetos são manipulados da mesma forma:

```
s.put(5);
i = s.get();
```

Contudo, fora da classe `Store` apenas os métodos da parte pública são visíveis. É então ilegal fazer

```
s.n = 5;
```

já que `n` pertence à parte privada da classe. A parte pública é composta por todos os métodos e variáveis de instância que se seguem à palavra `public`. A parte pública termina em uma declaração "`private:`" ou no fim da classe (`endclass`). Em S, apenas métodos podem estar na parte pública de uma classe.

Alocando dois objetos, como em

```
  var s, t : Store;
begin
s = Store.new();
t = Store.new();
```

[1] Pelo menos em S.

```
class Store
  public:
    proc get() : integer
      begin
      return n;
      end
    proc put( pn : integer )
      begin
      n = pn;
      end
 private:
   var n : integer;
endclass
```

Figura 3.1: Class `Store`

```
s.put(5);
t.put(12);
write( s.get(), " ", t.get() );
end
```

são alocados espaços para duas variáveis de instância `n`, uma para cada objeto. Em "`s.put(5)`", o método `put` é chamado e o uso de `n` na instrução

```
 n = pn
```

de `put` refere-se a "`s.n`". Um método só é invocado por meio de um objeto. Assim, as referências a variáveis de instância em um método referem-se às variáveis de instância deste objeto. Afinal, os métodos são feitos para manipular os dados do objeto, adicionando comportamento ao que seria uma estrutura composta apenas por dados. Na nomenclatura de orientação a objetos, uma instrução "`s.put(5)`" é o envio da mensagem "`put(5)`" ao objeto referenciado por `s` (ou objeto `s` para simplificar).

3.2 Proteção de Informação

Proteção de informação é um mecanismo que impede o acesso direto à implementação (estruturas de dados) de uma classe. As variáveis de instância só são manipuladas por meio dos métodos da classe. Na linguagem S, todas as variáveis de instância devem ser declaradas como privadas, impedindo a sua manipulação fora dos métodos da classe. Para exemplificar este conceito, usaremos a classe Pilha da Figura 3.2. Uma pilha é uma estrutura de dados onde o último elemento inserido, com `empilhe`, é sempre o primeiro a ser removido, com `desempilhe`. Esta estrutura espelha o que geralmente acontece com uma pilha de objetos quaisquer.

Esta pilha poderia ser utilizada como no programa abaixo.

```
proc main()
  var p, q : Pilha;
begin
p = Pilha.new();
p.crie();      // despreza o valor de retorno
p.empilhe(1);
```

```
const Max = 100;

class Pilha
  private:
    var topo : integer;
          { vetor de inteiros }
        vet  : array(integer)[Max];
  public:
    proc crie() : boolean
      begin
      topo = -1;
      return true;
      end
    proc empilhe( elem : integer ) : boolean
      begin
      if topo >= Max - 1
      then
        return false;
      else
        topo = topo + 1;
        vet[topo] = elem;
        return true;
      endif
      end
    proc desempilhe() : integer
      begin
      if topo < 0
      then
        return -1;
      else
        elem = vet[topo];
        topo = topo - 1;
        return elem;
      endif
      end
    proc vazia() : boolean
      begin
      return topo < 0;
      end
endclass
```

Figura 3.2: Uma pilha em S

```
p.empilhe(2);
p.empilhe(3);
while not p.vazia() do
  write( p.desempilhe() );

q = Pilha.new();
q.crie();
q.empilhe(10);
if not p.empilhe(20)
then
  erro();
endif
end
```

O programador que usa `Pilha` só pode manipulá-la por meio de seus métodos, sendo um erro de compilação o acesso direto às suas variáveis de instância:

```
p.topo = p.topo + 1; { erro de compilacao }
p.vet[p.topo] = 1;   { erro de compilacao }
```

A proteção de informação possui três características principais:

1. torna mais fácil a modificação de representação da classe, isto é, a estrutura de dados usada para a sua implementação. No caso de `Pilha`, a implementação é um vetor (`vet`) e um número inteiro (`topo`).

   Suponha que o projetista de `Pilha` mude a estrutura de dados para uma lista encadeada, retirando o vetor `vet` e a variável `topo` e resultando na classe da Figure 3.3.

   O que foi modificado foi o código dos métodos (veja acima), não o protótipo deles. Assim, todo o código do procedimento `main` visto anteriormente não será afetado. Por outro lado, se o usuário tivesse declarado `vet` e `topo` como públicos e usado

   ```
   p.topo = p.topo + 1;
   p.vet[p.topo] = 1
   ```

   para empilhar `1`, haveria um erro de compilação com a nova representação de `Pilha`, pois esta não possui vetor `vet`. E o campo `topo` é um ponteiro, não mais um inteiro;

2. o acesso aos dados de `Pilha` (`vet` e `topo`) por métodos tornam a programação de mais alto nível, mais abstrata. Lembrando, abstração é o processo de desprezar detalhes irrelevantes para o nosso objetivo, concentrando-se apenas no que nos interessa.

   Nesse caso, a instrução

   ```
   p.empilhe(1)
   ```

   é mais abstrata do que

   ```
   p.topo = p.topo + 1;
   p.vet[p.topo] = 1;
   ```

   porque ela despreza detalhes irrelevantes para quem quer empilhar um número (`1`), como que a `Pilha` é representada como um vetor, que esse vetor é `vet`, que `p.topo` é o topo da pilha, que `p.topo` é inicialmente `-1`, etc;

3. os métodos usados para manipular os dados (`crie`, `empilhe`, `desempilhe`, `vazia`) conferem a utilização adequada dos dados. Por exemplo, "`p.empilhe(1)`" confere se ainda há espaço na pilha, enquanto que em nas duas instruções alternativas mostradas acima o usuário se esqueceu

```
class Pilha
  private:
    var topo : Elem;
  public:
    proc crie() : boolean
      begin  topo = nil;  return true;  end
    proc empilhe( elem : integer ) : boolean
        var w : Elem;
      begin
      w = Elem.new();
      if w == nil  then  return false;
      else
        w.setNum(elem);
        w.setSuc(topo);
        topo = w;
        return true;
      endif
      end
    proc desempilhe() : integer
        var w : Elem;
            elem : integer;
      begin
      if topo == nil then return -1;
      else
        elem = topo.getNum();
        w = topo;
        topo = topo.getSuc();
        delete w;
        return elem;
      endif
      end
    proc vazia() : boolean
      begin
      return topo == nil;
      end
endclass
```

Figura 3.3: Uma classe Pilha implementada com uma lista

```
class A
  public:
    proc put( pn : integer )
      begin
      n = pn;
      end
    proc get() : integer
      begin
      return n;
      end
  private:
    var n : integer;
endclass

class B subclassOf A
  public:
    proc imp()
      begin
      write( get() );
      end
endclass
```

Figura 3.4: A classe `B` herda da classe `A`

disto. Resumindo, é mais seguro usar Proteção de Informação porque os dados são protegidos pelos métodos.

3.3 Herança

Herança é um mecanismo que permite a uma classe `B` herdar os métodos e variáveis de instância de uma classe `A`. Tudo se passa como se em `B` tivessem sido definidos os métodos e variáveis de instância de `A`. A herança de `A` por `B` é feita com a palavra chave `subclassOf` como mostrado na Figura 3.4.

A classe `B` possui todos os métodos definidos em `A` mais aqueles definidos em seu corpo:

```
proc put( pn : integer )
proc get() : integer
proc imp()
```

Assim, podemos utilizar todos estes métodos com objetos de `B`:

```
proc main()
  var b : B;
begin
b = B.new();
b.put(12);          // invoca A::put
b.imp();            // invoca B::imp
write( b.get() );   // invoca A::get
end
```

`A::put` é o método `put` da classe `A`. A classe `B` é chamada de "subclasse de `A`" e `A` é a "superclasse de

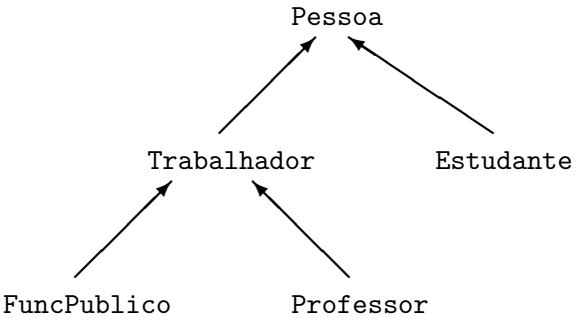

Figura 3.5: Hierarquia da classe `Pessoa`

`B`".[2]

O método `B::imp` possui uma chamada para um método `get`. O método invocado será `A::get`. Esta chamada poderia ser escrita como "`self.get()`" pois `self`, dentro de um método, refere-se ao objeto que recebeu a mensagem que causou a execução do método. Assim, o envio de mensagem "`b.put(5)`" causa a execução do método `A::put` e conseqüentemente da atribuição "`n = pn`". O "`n`" refere-se a "`b.n`" e poderíamos ter escrito esta atribuição como "`self.n = pn`". `self` é o objeto que recebeu a mensagem, `b`.

A classe `B` pode redefinir métodos herdados de `A`:

```
class B subclassOf A
  public:
    proc get() : integer
      begin
      return super.get() + 1;
      end
    proc imp()
      begin
      write( get() );
      end
endclass
```

"`super.get()`" invoca o método `get` da superclasse de `B`, que é `A`. Na chamada a `get` em `imp`, o método `get` a ser usado é o mais próximo possível na hierarquia de classes , que é `B::get`.

Herança é utilizada para expressar relacionamentos do tipo "é um". Por exemplo, um estudante é uma pessoa, um funcionário público é um trabalhador, um professor é um trabalhador, um trabalhador é uma pessoa. Estes relacionamentos são mostrados na hierarquia da Figura 3.5, na qual a herança de `A` por `B` é representada através de uma seta de `B` para `A`. A subclasse sempre aparecerá mais embaixo nas figuras.

Uma subclasse é sempre mais específica do que a sua superclasse. Assim, um trabalhador é mais específico do que uma pessoa porque todo trabalhador *é uma* pessoa, mas o contrário nem sempre é verdadeiro. Se tivéssemos feito `Pessoa` herdar `Trabalhador`, haveria um erro lógico no programa, mesmo se não houvesse nenhum erro de compilação.

Considere agora a hierarquia de classes representando figuras das Figuras 3.6 e 3.7. Se a classe `Circulo` precisar utilizar as variáveis `x` e `y` herdadas de `Figura`, ela deverá chamar os métodos `getX()`

[2]Na terminologia usualmente empregada em C++, `A` é a "classe base" e `B` a "classe derivada".

```
class Figura
  public:
    proc set_xy( px, py : integer )
      begin
      x = px;
      y = py;
      end
    proc imp()
      begin
      write( "Centro(", x, ", ", y, ")" );
      end
    proc getX() : integer
      begin
      return x;
      end
    proc getY() : integer
      begin
      return y;
      end
  private:
    var x, y : integer;
endclass
```

Figura 3.6: Classe `Figura`

```
class Circulo subclassOf Figura
  public:
    proc init( p_raio : real; x, y : integer )
      begin
      super.set_xy(x, y);
      raio = p_raio;
      end
    proc setRaio( p_raio : real )
      begin
      raio = p_raio;
      end
    proc getRaio() : real
      begin
      return raio;
      end
    proc imp()
      begin
      write( "raio = ", raio );
      super.imp();
      end
    proc getArea() : real
      begin
      return PI*raio*raio;
      end
  private:
    raio : real;
endclass
```

Figura 3.7: Classe `Circulo`

e `getY()` desta classe. Uma subclasse *não* pode manipular diretamente a parte privada da superclasse. Se isto fosse permitido, modificações na representação de uma classe poderiam invalidar as subclasses.

Algumas linguagens, como C++ e Eiffel, permitem que uma classe herde de mais de uma superclasse. Esta facilidade causa uma ambigüidade quando dois métodos de mesmo nome são herdados de duas superclasses diferentes. Por exemplo, suponha que uma classe `JanelaTexto` herde de `Texto` e `Janela` e que ambas as superclasses definam um método `getNome`. Que método o envio de mensagem "`jt.getNome()`" deverá invocar se o tipo de `jt` for `JanelaTexto` ? Em C++, há duas formas de se resolver esta ambigüidade:

1. a primeira é especificando-se qual superclasse se quer utilizar:
   ```
   nome = jt.A::getNome()
   ```

2. a segunda é definir um método `getNome` em `JanelaTexto`.

Em Eiffel o nome do método `getNome` herdado de `Texto` ou `Janela` deve ser renomeado, evitando assim a colisão de nomes.

Uma linguagem em que é permitido a uma classe herdar de mais de uma superclasse suporta *herança múltipla*. Este conceito, aparentemente muito útil, não é muito utilizado em sistemas reais e torna os programas mais lentos porque a implementação de envio de mensagens é diferente do que quando só há herança simples. Herança múltipla pode ser simulada, pelo menos parcialmente, declarando-se um objeto da superclasse na subclasse e redirecionando mensagens a este objeto — veja Figura 3.8.

Há ainda outro problema com herança múltipla. Considere que as classes `B` e `C` herdem da classe `A` e a classe `D` herde de `B` e `C`, formando um losango. Um objeto da classe `D` também é um objeto de `A`, `B` e `C`. Este objeto deve ter todas as variáveis de `A`, `B` e `C`. Mas deve o objeto ter um ou dois conjuntos de dados de `A` ? Afinal, a classe `D` herda `A` por dois caminhos diferentes. Em alguns casos, seria melhor `D` ter dois conjuntos de dados de `A`. Em outros, é melhor ter apenas um conjunto. Veja estes exemplos:

- a classe `Pessoa` é herdada por `Estudante` e `Atleta`, que são herdadas por `BolsistaAtleta`.[3] Neste caso, deve-se ter um único nome em objetos da classe `BolsistaAtleta`;
- a classe `Trabalhador` é herdada por `Professor` e `Gerente`, que são herdados por `ProfessorGerente`.[4] Neste caso, deve-se ter os dados do trabalhador, como tempo de serviço e nome do empregador, duplicados em objetos de `ProfessorGerente`. Seria interessante que `Trabalhador` herdasse de `Pessoa`. Assim, um objeto de `ProfessorGerente` teria apenas uma variável para nome e outros dados básicos.

Algumas linguagens optam por uma destas opções enquanto que outras permitem que se escolha uma delas no momento da herança.

3.4 Polimorfismo

Se o tipo de uma variável `w` for uma classe `T`, a atribuição
```
w = nil;
```
estará correta qualquer que seja `T`. Isto é possível porque `nil` é um valor polimórfico: ele pode ser usado onde se espera uma referência para objetos de qualquer classe. Polimorfismo quer dizer faces múltiplas e é o que acontece com `nil`, que possui infinitos tipos.

[3] O atleta ganha uma bolsa de estudos por ser atleta.

[4] O professor trabalha em tempo parcial e também é um gerente.

```
class B
  public:
    proc init()
      begin
      a = B.new();
      end
    proc get() : integer
      begin
      return a.get();
      end
    proc put( pn : integer )
      begin
      a.put(pn);
      end
    proc imp()
      begin
      write(get());
      end
    proc getA()
      begin
      return a;
      end
  private:
    var a : A;
endclass
```

Figura 3.8: Simulação de herança de `A` por `B`

Em S, uma variável cujo tipo é uma classe pode referir-se a objetos de subclasses desta classe. O código

```
proc main()
  var f : Figura;
      c : Circulo;
begin
c = Circulo.new();
c.init( 20.0, 30, 50 );
f = c;
f.imp();
end
```

está correto. A atribuição

```
f = c;
```

atribui uma referência para `Circulo` a uma variável de `Figura`.

S permite atribuições do tipo

```
Classe = Subclasse
```

como a acima, que é

```
Figura = Circulo
```

Uma variável cujo tipo é uma classe `A` sempre será polimórfica, pois ela poderá apontar para objetos de `A` ou qualquer objeto de subclasses de `A`.

Agora, qual método

```
f.imp()
```

irá invocar ? `f` referencia um objeto de `Circulo` e, portanto, seria natural que o método invocado fosse "`Circulo::imp`". Contudo, o tipo de `f` é "`Figura`" e "`f.imp()`" também poderia invocar `Figura::imp`.

O envio de mensagem "`f.imp()`" invocará o método `imp` de `Circulo`. Será feita uma busca em *tempo de execução* por método `imp` na classe do objeto apontado por `f`. Se `imp` não for encontrado nesta classe, a busca continuará na superclasse, superclasse da superclasse e assim por diante. Quando o método for encontrado, ele será chamado. Sendo a busca feita em tempo de execução, será sempre chamado o método mais adequado ao objeto, isto é, se `f` estiver apontando para um círculo, será chamado o método `imp` de `Circulo`, se estiver apontando para um retângulo, será chamado `Retangulo::imp` e assim por diante.

A instrução "`f.imp()`" causará uma busca em *tempo de compilação* por método `imp` na classe *declarada* de `f`, que é `Figura` (`f` é declarado como "`f : Figura`"). Se `imp` não fosse encontrado lá, a busca continuaria na superclasse de `Figura` (se existisse), superclasse da superclasse e assim por diante. Se o compilador não encontrar o método `imp`, ele sinalizará um erro. Isto significa que uma instrução

```
f.setRaio(10);
```

será ilegal mesmo quando tivermos certeza de que `f` apontará em tempo de execução para um objeto de `Circulo` (que possui método `setRaio`). A razão para esta restrição é que o compilador não pode garantir que `f` apontará para um objeto que possui método `setRaio`. A inicialização de `f` pode depender do fluxo de controle:

```
proc m( i : integer )
  var f, aFig : Figura;
      aCir : Circulo;
begin
```

```
aCir = Circulo.new();
aCir.init(20.0, 50, 30);
aFig = Figura.new();
aFig.set_xy( 30, 40 );
if i > 0
then
  f = aFig;
else
  f = aCir;
endif
f.setRaio(10);
...
end
```

Se este procedimento fosse legal, `f` poderia ser inicializado com `aFig`. Em tempo de execução, seria feita uma busca por método `setRaio` na classe `Figura` e este método não seria encontrado, resultando em um erro de tempo de execução com o término do programa.

Como resultado da discussão acima, temos que

```
f.imp()
```

será válido quando `imp` pertencer à classe declarada de `f` ou suas superclasses (se existirem). Já que `f` pode apontar para um objeto de uma subclasse de `Figura`, podemos garantir que a classe deste objeto possuirá um método `imp` em tempo de execução ? A resposta é "sim", pois `f` pode apontar apenas para objetos de `Figura` ou suas subclasses. O compilador garante, ao encontar

```
f.imp()
```

que a classe declarada de `f`, `Figura`, possui método `imp` e, como todas as subclasses herdam os métodos das superclasses, as subclasses de `Figura` possuirão pelo menos o método `imp` herdado desta classe. Assim, `f` apontará para um objeto de `Figura` ou suas subclasses e este objeto certamente possuirá um método `imp`.

Polimorfismo é fundamental para o reaproveitamento de *software*. Quando um método aceitar como parâmetro um objeto de `Figura`, como

```
proc m( f : Figura )
```

podemos passar como parâmetro objetos de qualquer subclasse desta classe. Isto porque uma chamada

```
m(aCir);
```

envolve uma atribuição "`f = aCir`", que será correta se for da forma

```
Classe = Subclasse
```

Então, podemos passar como parâmetro a `m` objetos de `Circulo`, `Retangulo`, etc. Não é necessário construir um método `m` para objetos de cada uma das subclasses de `Figura` — um mesmo método `m` pode ser utilizado com objetos de todas as subclasses. O código de `m` é *reaproveitado* por causa do polimorfismo.

Admitindo que as classes `Retangulo` e `Triangulo` existam e são subclasses de `Figura`, o código a seguir mostra mais um exemplo de polimorfismo.

```
...

proc impVet( v : array(Figura)[]; n : integer )
  { Imprime cada elemento do vetor v de n elementos.
    v é um vetor de figuras. }

  var i : integer;
```

```
begin
for i = 0 to n do
  v[i].imp();
end


proc main()
  var c1     : Circulo;
      r1, r2 : Retangulo;
      t1     : Triangulo;
      vetFig : array(Figura)[];
begin
c1 = Circulo.new();
r1 = Retangulo.new();
r2 = Retangulo.new();
t1 = Triangulo.new();
c1.init( 5.0, 80, 30 );
r1.init( 30, 50, 70, 60 );
r2.init( 20, 100, 80, 150 );
t1.init( 10, 18, 30, 20, 40, 25 );
  { atribui varios elementos a vetFig }
vetFig = ( r1, t1, r2, c1 );
impVet( vetFig, 4 );
end
```

A função `impVet` percorre o vetor `v` enviando a mensagem `imp` a cada um de seus elementos. O método `imp` executado dependerá da classe do objeto apontado por "`v[i]`".

Existe uma outra forma de polimorfismo em C++ que será mostrada acrescentando-se métodos nas classes `Figura` e `Circulo`:

```
class Figura
  public:
    ...
    proc desenhe() begin end  { vazio }
    proc apague()  begin end
    proc mova( nx, ny : integer )
      begin
      apague();
      x = nx;
      y = ny;
      desenhe();
      end
  private:
    var x, y : integer;
endclass

class Circulo subclassOf Figura
  public:
      { Os ... abaixo sao os metodos de Circulo definidos anteriormente }
```

```
    ...
    proc desenhe()
      begin
        { desenha um circulo }
      ...
      end
    proc apague()
      begin
        { apagua o circulo }
      ...
      end
  private:
    var raio : real;
endclass
```

Os métodos `desenhe` e `apague` de `Figura` não fazem nada porque esta classe foi feita para ser herdada e não para se criar objetos dela. O método `mova` apaga o desenho anterior, move a figura e a desenha novamente. Como `desenhe` e `apague` são vazios em `Figura`, `mova` só faz sentido se `desenhe` e `apague` forem redefinidos em subclasses. Em

```
  var c : Circulo;
begin
c = Circulo.new();
c.init( 10.0, 50, 30 );
c.mova( 20, 80 );
...
end
```

o método invocado em "`c.mova(20, 80)`" será "`Figura::mova`". Este método possui um envio de mensagem

```
apague();
```

que é o mesmo que

```
self.apague();
```

que envia a mensagem `apague` ao objeto que recebeu a mensagem `mova`, que é "`c`". Então, a busca por método `apague` será iniciada em `Circulo` (classe do objeto `c`), onde `Circulo::apague` será encontrado e executado. Da mesma forma, a instrução

```
desenhe()
```

em `Figura::mova` invocará `Circulo::desenhe`.

Observando este exemplo, verificamos que não foi necessário redefinir o método `mova` em `Circulo` — o seu código foi reaproveitado. Se tivermos uma classe `Retangulo`, subclasse de `Figura`, precisaremos de definir apenas `desenhe` e `apague`. O método `mova` será herdado de `Figura` e funcionará corretamente com retângulos. Isto é, o código

```
  var r : Retangulo;
begin
r = Retangulo.new();
r.init( 30, 50, 70, 20 );
r.mova( 100, 120 );
...
end
```

invocará os métodos `desenhe` e `apague` de `Retangulo`.

As redefinições de `apague` e `desenhe` em `Circulo` causaram alterações no método `mova` herdado de `Figura`, adaptando-o para trabalhar com círculos. Ou seja, `mova` foi modificado pela redefinição de outros métodos em uma subclasse. Não foi necessário redefinir `mova` em `Circulo` adaptando-o para a nova situação. Dizemos que o código de `mova` foi *reaproveitado* em `Circulo`. O método `mova` se comportará de maneira diferente em cada subclasse, apesar de ser definido uma única vez em `Figura`.

3.5 Modelos de Polimorfismo

Esta seção descreve quatro formas de suporte a polimorfismo empregado pelas linguagens Smalltalk, POOL-I, Java e C++. Naturalmente, os modelos de polimorfismo descritos nas subseções seguintes são abstrações das linguagens reais e podem apresentar diferenças em relação a elas.

3.5.1 Smalltalk

Smalltalk [5] é uma linguagem tipada dinamicamente, o que quer dizer que na declaração de uma variável ou parâmetro não se coloca o tipo. Durante a execução, uma variável irá se referir a um objeto e terá o tipo deste objeto. Conseqüentemente, uma variável pode se referir a objetos de tipos diferentes durante a sua existência.

No exemplo abaixo,

```
   ...
   var a, b;
begin
a = 1;
b = Janela.new();
b.init(a, 5, 20, 30);
a = b;
a.desenha();
...
end
```

se a instrução "`a.desenha()`" for colocada logo após "`a = 1`", haverá o envio da mensagem `desenha` a um número inteiro. Como a classe dos inteiros não possui um método `desenha`, ocorrerá um erro de tipos e o programa será abortado.

Considere agora um método

```
 proc m(y)
   begin
   y.desenha();
   y.move(10, 20);
   end
```

de uma classe `A`. Assuma que exista uma classe `Janela` em Smalltalk, que é aquela da Figura 3.9 sem os tipos das declarações de variáveis. Esta classe possui métodos `desenha` e `move`, sendo que este último não causa erros de tipo se os seus dois parâmetros são números inteiros.

Se um objeto de `Janela` for passado com parâmetro a `m`, como em

```
  a = A.new();
  a.m( Janela.new() );
```

não haverá erros de tipo dentro deste método. Se a `m` for passado um objeto de uma subclasse de `Janela`, também não haverá erros de tipo. A razão é que uma subclasse possui pelo menos todos os

```
class Janela
  private:
    var x, y : integer;
  public:
    proc desenha()  begin ... end
    proc move( novo_x, novo_y : integer )
      begin
      self.x = novo_x;
      self.y = novo_y;
      self.desenha();
      end
    proc init( px, py : integer ); begin x = px; y = py; end
endclass

class JanelaTexto subclassOf Janela
  ...
  public:
    proc desenha() begin ... end
endclass
```

Figura 3.9: Classes `Janela` e `JanelaTexto`

métodos da superclasse. Assim, se um objeto de `Janela` sabe responder a todas as mensagens enviadas a ele dentro de `m`, um objeto de uma subclasse também saberá responder a todas estas mensagens. Estamos admitindo que, se `move` for redefinido em uma subclasse, ele continuará aceitando dois inteiros como parâmetros sem causar erros de tipo.

De fato, o método `m` pode aceitar como parâmetros objetos de *qualquer* classe que possua métodos `move` e `desenha` tal que `move` aceita dois inteiros como parâmetros e `desenha` não possui parâmetros. Não é necessário que esta classe herde de `Janela`. Este sistema de tipos, sem restrição nenhuma que não seja a capacidade dos objetos de responder às mensagens que lhe são enviadas, possui o maior grau possível de polimorfismo.

Se `m` for codificado como

```
proc m(y, b)
  begin
  y.desenha();
  y.move(10, 20);
  if b
  then
    y.icon();
  endif
  end
```

o código

```
    a = A.new();
    a.m( Janela.new(), false );
```

não causará erro de tipos em tempo de execução, pois a mensagem `icon` não será enviada ao objeto de `Janela` em execução. Se fosse enviada, haveria um erro já que a classe `Janela` não possui método

```
class JanelaProcesso
  ...
  public:
    proc desenha()                  begin ... end
    proc move( nx, ny : integer )   begin ... end
    proc init( px, py : integer )   begin ... end
    proc iniciaProcesso()           begin ... end
    proc setProcesso( s : string )  begin ... end
endclass
```

Figura 3.10: Uma classe que é subtipo de `Janela`

`icon`.

Em geral, o fluxo de execução do programa, controlado por `if`'s, `while`'s e outras estruturas, determina quais mensagens são enviadas para cada variável. E este mesmo fluxo determina a capacidade de cada variável de responder a mensagens. Para compreender melhor estes pontos, considere o código

```
if b > 0
then
  a = Janela.new();
else
  a = 1;
endif
if c > 1
then
  a.desenha();
else
  a = a + 1;
endif
```

O primeiro `if` determina quais as mensagens `a` pode responder, que depende da classe do objeto a que `a` se refere. O segundo `if` seleciona uma mensagem a ser enviada à variável `a`. Em Smalltalk, "`+ 1`" é considerado um envio de mensagem.

Então, o fluxo de execução determina a corretude de tipos de um programa em Smalltalk, o que torna os programas muito inseguros. Alguns trechos de código podem revelar um erro de tipos após meses de uso. Note que, como é impossível prever todos os caminhos de execução de um programa em tempo de compilação, é também impossível garantir estaticamente que um programa em Smalltalk é corretamente tipado.

3.5.2 POOL-I

Esta seção descreve o modelo das linguagens POOL-I [1] e Green [8] [7]. Como o sistema de tipos de Green foi parcialmente baseado no de POOL-I, este modelo será chamado de modelo POOL-I. Green e POOL-I são linguagens estaticamente tipada, pois todos os erros de tipo são descobertos em compilação.

Neste modelo, o tipo de uma classe é definido como o conjunto das interfaces (assinaturas ou *signatures*) de seus métodos públicos. A interface de um método é o seu nome, tipo do valor de retorno (se houver) e tipos de seus parâmetros formais (o nome dos parâmetros é desprezado). Por

exemplo, o tipo da classe `Janela` da Figura 3.9 é

```
{ desenha(), move(integer, integer), init(integer, integer)}
```

sendo que { e } são utilizados para delimitar os elementos de um conjunto, como em matemática. Um tipo U será subtipo de um tipo T se U possuir pelo menos as interfaces que T possui. Isto é, $T \subset U$. Como exemplo, o tipo da classe `JanelaProcesso` da Figura 3.10 é um subtipo do tipo da classe `Janela` da Figura 3.9. Como abreviação, dizemos que a classe `JanelaProcesso` é subtipo da classe `Janela`.

Quando uma classe `B` herdar de uma classe `A`, diremos que `B` é *subclasse* de `A`. Neste caso, `B` herdará todos os métodos públicos de `A`, implicando que `B` é *subtipo* de `A`.[5] Observe que toda subclasse é também subtipo, mas é possível existir subtipo que não é subclasse — a classe `JanelaProcesso` da Figura 3.10 é subtipo mas não subclasse de `Janela`.

Neste modelo, uma atribuição

```
t = s
```

estará correta se a classe declarada de `s` for subtipo da classe declarada de `t`. As atribuições do tipo

```
Tipo = SubTipo;
```

são válidas.

Esta restrição permite a detecção de todos os erros de tipo em tempo de compilação, por duas razões:

- Em um envio de mensagem
  `t.m(`b_1`, `b_2`, ... `b_n`)`
  o compilador confere se a classe com que `t` foi declarada possui um método chamado `m` cujos parâmetros formais possuem tipos T_1, T_2, ... T_n tal que o tipo de b_i é subtipo de T_i, $1 \leq i \leq n$. A regra "`Tipo = Subtipo`" é obedecida também em passagem de parâmetros.

- Ao executar este envio de mensagem, é possível que `t` não se refira a um objeto de sua classe, mas de um subtipo do tipo da sua classe, por causa das atribuições do tipo `Tipo = SubTipo`, como `t = s`. De qualquer forma, não haverá erro de execução, pois tanto a sua classe quanto qualquer subtipo dela possuem o método `m` com parâmetros formais cujos tipos são T_1, ... T_n.

Em uma declaração

```
var a : A
```

a variável `a` é associada ao *tipo* da classe `A` e não à classe `A`. Deste modo, `a` pode se referir a objetos de classes que são subtipos sem serem subclasses de `A`. Este é o motivo pelo qual a declaração da variável `a` não aloca memória automaticamente para um objeto da classe `A`.

3.5.3 C++

C++ [14] é uma linguagem estaticamente tipada em que todo subtipo é subclasse. Portanto, as atribuições válidas possuem a forma

```
Classe = Subclasse
```

Assume-se que a variável do lado esquerdo da atribuição seja um ponteiro e que o lado direito seja uma referência para um objeto:

```
Figura *p;
...
p = new Circulo(150, 200, 30);
```

[5] A linguagem POOL-I, ao contrário deste modelo, permite subclasses que não são subtipos. Em Green, todas as subclasses são subtipos.

Não há polimorfismo em C++ quando não se utiliza ponteiros. Se `p` fosse declarado como "`Figura p;`", ele poderia receber apenas objetos de `Figura` em atribuições. Neste modelo assume-se que não há variáveis cujo tipo sejam classes, apenas ponteiros para classes.

O motivo pelo qual o modelo C++ exige que subtipo seja também subclasse é o desempenho. Uma chamada de método é feita através de um vetor de ponteiros para funções e é apenas duas ou três vezes mais lenta do que uma chamada de função normal.

C++ suporta métodos virtuais e não virtuais, sendo que nestes últimos a busca pelo método é feita em compilação — a ligação mensagem/método é estática. Nesta subseção, consideramos que todos os métodos são virtuais.

3.5.4 Java

Java [11] [10] suporta apenas herança simples. Contudo, a linguagem permite a declaração de *interfaces* que podem ser utilizados em muitos casos em que herança múltipla deveria ser utilizada. Uma interface *declara* assinaturas (*signatures* ou interfaces) de métodos:

```
interface Printable {
   void print();
}
```

Uma assinatura de um método é composto pelo tipo de retorno, o nome do método e os parâmetros e seus tipos (sendo os nomes dos parâmetros opcionais).

Uma classe pode *implementar* uma interface:

```
class Worker subclassOf Person implements Printable {
   ...
   real getSalary() { ... }
   void print() { ... }
}
```

Quando uma classe implementa uma interface, ela é obrigada a definir (com o corpo) os métodos que aparecem na interface. Se a classe `Worker` não definisse o método `print`, haveria um erro de compilação. Uma classe pode herdar de uma única classe mas pode implementar várias interfaces diferentes.

Este modelo considera as interfaces como classes de uma linguagem com herança múltipla exceto que as interfaces não podem *definir* métodos. Interfaces são similares a classes abstratas[6] e tudo se passa como se o modelo admitisse herança múltipla onde todas as classes herdadas são completamente abstratas (sem nenhum corpo de método) exceto possivelmente uma delas. Um *tipo* neste modelo é uma classe ou uma interface. Se uma classe `A` implementa uma interface `I`, então `A` é subtipo de `I`. E `A` é subtipo de sua superclasse, se existir. As atribuições válidas são

```
Tipo = subtipo
```

Pode-se declarar uma variável cujo tipo é uma interface. Como exemplo, o código abaixo é válido.

```
Printable p;
Person person;
person = new Worker();
p = person;
```

[6] A diferença é que classes abstratas podem declarar variáveis de instância e o corpo de alguns métodos. E podem possuir métodos privados. Em uma interface, todos os métodos são públicos.

```
proc Q( x : Janela; y : integer )
  begin
  x.desenha();
  if y > 1
  then
    x.move(20, 5);
  endif
  end
```

Figura 3.11: Um procedimento no modelo C++

```
p.print();
p = new NightWorker(); // NightWorker é subclasse de Worker
```

Java é estaticamente tipada. Então, se o tipo de uma variável é uma interface, apenas métodos com assinaturas declaradas na interface podem ser chamadas por meio da variável. Por exemplo, por meio de `p` acima pode-se chamar apenas o método `print`.

Interfaces em Java são uma forma de adicionar os benefícios de herança múltipla à linguagem mas sem alguns dos problemas desta facilidade (como duplicação dos dados de objetos herdados por mais de um caminho — veja página 44).

Neste modelo Java, uma interface não pode herdar de outra interface. Na linguagem Java isto é legal.

3.5.5 Comparação entre os Modelos de Polimorfismo e Sistema de Tipos

Agora podemos comparar o polimorfismo dos modelos de linguagens descritos acima. Considere o método `Q` da Figura 3.11 no modelo C++. Ele pode receber, como primeiro parâmetro (`x`), objetos da classe `Janela` ou qualquer *subclasse* desta classe.

Em Java, se `Janela` é uma interface, o primeiro parâmetro passado a `Q` pode ser objeto de quaisquer classes que implementem esta interface ou que herdem das classes que implementam esta interface. As classes que implementam uma interface geralmente não têm nenhuma relação de herança entre si. Se quisermos passar um objeto de uma classe A para `Q`, basta fazer com que `A` implemente a interface `Janela`. Isto é, `A` deveria implementar os métodos definidos em `Janela` e que possivelmente são utilizados no corpo de `Q`.

Em uma linguagem com herança simples e que não suporte interfaces (como definidas em Java), apenas objetos de `Janela` e suas subclasses poderiam ser passados como primeiro parâmetro (assumindo então que `Janela` é uma classe e não um interface). Para passar objetos de uma classe `A` como parâmetros, deveríamos fazer esta classe herdar de `Janela`, o que não seria possível se `A` já herdasse de uma outra classe.

Se `Janela` for uma classe, poderão ser passados a `Q`, como primeiro parâmetro, objetos da classe `Janela` ou qualquer *subclasse* desta classe, como em C++.

Em POOL-I, os parâmetros passados a `Q` podem ser de qualquer *subtipo* de `Janela`. Todas as classes que herdam de `Janela` (subclasses) são subtipos desta classe e há subtipos que *não* são subclasses. Ou seja, o conjunto dos subtipos de `Janela` é potencialmente maior que o de subclasses de `Janela`. Conseqüentemente, em POOL-I o procedimento `Q` pode ser usado com mais classes do que em C++, pois o conjunto de classes aceito como parâmetro para `Q` nesta última linguagem (subclasses) é potencialmente menor que o conjunto aceito por POOL-I (subtipos).

```
proc Q(x, y)
begin
x.desenha();
if y > 1
then
  x.move(20, 5);
endif
end
```

Figura 3.12: Procedimento `Q` no modelo Smalltalk

Em C++, Java e POOL-I, o compilador confere, na compilação de `Q`, se a classe/interface de `x`, que é `Janela`, possui métodos correspondentes às mensagens enviadas estaticamente a `x`. Isto é, o compilador confere se `Janela` possui métodos `desenha` e `move` e se `move` admite dois inteiros como parâmetros. Estaticamente é garantido que objetos de `Janela` podem ser passados a `Q` (como primeiro parâmetro) sem causar erros de tipo. Em tempo de execução, objetos de subclasses ou subtipos de `Janela` serão passados a `Q`, por causa de atribuições do tipo `Tipo = SubTipo`. Estes objetos saberão responder a todas as mensagens enviadas a eles dentro de `Q`, pois: a) eles possuem pelo menos todos os métodos que objetos da classe `Janela` possuem; b) objetos da classe `Janela` possuem métodos para responder a todas as mensagens enviadas ao parâmetro `x` dentro de `Q`.

Smalltalk dispensa tipos na declaração de variáveis e, portanto, o procedimento `Q` neste modelo seria aquele mostrado na Figura 3.12. Como nem `x` nem `y` possuem tipos, não se exige que o objeto passado como primeiro parâmetro real a `Q` possua métodos `desenha` e `move`. De fato, na instrução

```
Q(a,0)
```

é enviada mensagem `desenha` ao objeto referenciado por `x` (e também por `a`), mas não é enviada a mensagem `move`.

Como conseqüência, esta instrução pode ser executada com parâmetros `a` de qualquer classe que possua um método `desenha` sem parâmetros. Ao contrário de POOL-I, Java e C++, a classe do parâmetro `x` de `Q` não precisa possuir também o método `move`. Logo, o método `Q` pode ser usado com um conjunto de classes (para o parâmetro `x`) potencialmente maior que o conjunto de classes usadas com o procedimento `Q` equivalente de POOL-I. Portanto, Smalltalk possui mais polimorfismo que POOL-I.

Em linguagens convencionais, uma atribuição `a = b` será correta se os tipos de `a` e `b` forem iguais ou `b` puder ser convertido para o tipo de `a` (o que ocorre com perda de informação se o tipo de `b` for mais abrangente do que o de `a`). Em POOL-I, `a = b` será válido se a classe de `b` for subtipo da classe de `a`. Em C++, se a classe de `b` for subclasse da classe de `a`. Em Java, se a classe de `b` for subclasse da classe de `a` (se o tipo de `a` for uma classe) ou implementar (direta ou indiretamente) a interface que é o tipo de `a` (se o tipo de `a` for uma interface).[7] Em Smalltalk, esta operação será sempre correta. Logo, as linguagens orientadas a objeto citadas estendem o significado da atribuição permitindo um número maior de tipos do seu lado direito. Como em passagem de parâmetros existe uma atribuição implícita, procedimentos e métodos podem aceitar parâmetros reais de mais classes do que normalmente aceitariam, o que é o motivo do reaproveitamento de código. Concluindo, podemos afirmar que a mudança do significado da atribuição é o motivo de todo o reaproveitamento de *software* causado pelo polimorfismo descrito neste artigo. Quando mais liberal (Smalltalk — nenhuma restrição ao lado direito de `=`) é a mudança, maior o polimorfismo.

Suponha que estejamos construindo um novo sistema de janelas e seja necessário construir uma

[7] Aqui ignoramos o fato de que na linguagem Java uma interface pode herdar de outra interface.

classe `Window` que possua os mesmos métodos que `Janela`. Contudo, `Window` possui uma aparência visual e uma implementação bem diferentes de `Janela`. Certamente, é interessante poder passar objetos de `Window` onde se espera objetos de `Janela`. Todo o código construído para manipular esta última classe seria reusado pela classe `Window`.

Em C++, `Window` deve herdar de `Janela` para que objetos de `Window` possam ser usados onde se espera objetos de `Janela`. Como `Window` possui uma implementação completamente diferente de `Janela`, as variáveis de instância da superclasse `Janela` não seriam usadas em objetos de `Window`. Então, herança estaria sendo utilizada para expressar *especificação* do problema e não *implementação*. *Especificação* é o que expressa a regra "um objeto de uma subclasse é um objeto de uma superclasse". *Implementação* implica que uma subclasse possui pelos menos as variáveis de instância da sua superclasse e herda algumas ou todas as implementações dos métodos.

Em POOL-I, este problema não existe, pois a *especificação* e *implementação* são desempenhados por mecanismos diferentes. A saber, subtipagem e herança. Em Java, o programador poderia fazer `Janela` e `Window` herdarem de uma interface com todos os métodos originais de `Janela`. Mas isto só será possível se o código fonte de `Janela` estiver disponível. Em POOL-I, isto não é necessário.

Existe um problema ainda maior em C++ por causa da ligação subtipo-subclasse. Considere que a classe `Janela` possua um método

```
proc n( outra : Janela )
  begin
  ...
  w = outra.x;
  ...
  end
```

Dentro deste método é feito um acesso à variável `x` do parâmetro `outra`. Esta variável de instância foi, naturalmente, declarada em `Janela`. Se este parâmetro refere-se a um objeto de `Window`, subclasse de `Janela`, então `outra.x` não foi inicializado. A razão é que a classe `Window` não utiliza as variáveis herdadas de `Janela` e, portanto, não as inicializa. Note que o mesmo problema ocorre com a linguagem Java.

3.6 Classes parametrizadas

A classe `Store` da Figura 3.13, em Smalltalk, permite o armazenamento de objetos de qualquer tipo. Um objeto de `Store` guarda um outro objeto através do método `put` e retorna o objeto armazenado através de `get`.

A classe `Store` em POOL-I é mostrada na Figura 3.14 com o nome de `Store_Int`. Como cada variável possui um tipo nesta linguagem, a classe `Store` se torna restrita — só pode armazenar inteiros. Se quisermos armazenar objetos de outros tipos, teremos que construir outras classes semelhantes a `Store` — uma classe para cada tipo, como `Store_boolean`, `Store_real`, etc.

Em Smalltalk, a classe `Store` é utilizada para todos os tipos, causando reaproveitamento de código. Nesta linguagem, podemos ter uma árvore binária ou lista encadeada genérica, que permite armazenar objetos de qualquer tipo. Os métodos para a manipulação de cada uma destas estruturas de dados é construído uma única vez. Como não há conferência de tipos, é possível inserir objetos de diferentes classes na lista encadeada, criando uma lista heterogênia.

A linguagem POOL-I possui uma construção que oferece um pouco da flexibilidade de Smalltalk, chamada de classes parametrizadas.[8] Uma classe parametrizada possui um ou mais parâmetros, que são tipos ou constantes. Estes parâmetros são substituídos por tipos e valores reais ao se usar a classe.

[8] O modelo apresentado abaixo difere daquele da linguagem real. Ele serve apenas como exemplo.

```
class Store
  private:
    var n;
  public:
    proc put( i )
      begin
      n = i;
      end
    proc get()
      begin
      return n;
      end
endclass
```

Figura 3.13: Classe para armazenar objetos de qualquer tipo, em Smalltalk

```
class Store_Int
  private:
    var n : integer;
  public:
    proc put( i : integer )
      begin
      n = i;
      end
    proc get() : integer
      begin
      return n;
      end
endclass
```

Figura 3.14: Classe que armazena inteiros, em POOL-I

```
class Store[ type T ]
  private:
    var n : T;
  public:
    proc put( i : T )
      begin
      n = i;
      end
    proc get() : T
      begin
      return n;
      end
endclass
```

Figura 3.15: Classe parametrizada `Store` em POOL-I

A classe `Store` da Figura 3.15 é parametrizada e possui um tipo com parâmetro, como indicado pela palavra-chave `type`. Na declaração de uma variável desta classe, deve ser especificado o parâmetro `T`:

```
var
    v_int  : Store[integer];
    v_bool : Store[boolean];
```

Quando o compilador encontrar a declaração de `v_int`, ele executará os seguintes passos:

- copiará o *texto* de `Store` para uma área de memória. Observe que é o texto e não um possível código compilado correspondente aos métodos desta classe;
- o identificador `T` será substituído por `integer` (que é o parâmetro real da classe) em toda a cópia. Isto criará uma nova classe, `Store[integer]`. O texto desta classe será exatamente igual a `Store_Int` da Figura 3.14;
- o texto produzido no item anterior será compilado.

Observe que `Store` não é uma classe, é apenas uma máscara (ou esqueleto — *template* em Inglês) para a criação de classes. Os exemplos (instâncias) construídos a partir de `Store`, como `Store[integer]`, são realmente classes. Eles podem ser usados como tipo de variáveis, em herança, etc. A substituição dos parâmetros de uma classe parametrizada (no caso, `T`) por valores reais (no caso, `integer`) é chamada de instanciação da classe.

Cada instanciação da classe com tipos diferentes causa a criação de um novo texto. Portanto, há duplicação de código — na declaração de `v_int` e `v_bool`, haveria a criação de dois métodos `put` e dois `get`.

Considere agora que um novo método foi acrescentado à classe `Store` da Figura 3.15:

```
proc sum( k : Store[T] )
  begin
  n = n.add( k.get() );
  end
```

Agora, uma mensagem `add` é enviada ao objeto `n`, o que implica que o tipo de `n`, `T`, deve possuir método `add` com parâmetro do tipo `T` (o tipo de `k.get()` é `T`).

Esta restrição torna ilegal as declarações de `v_int` e `v_bool`, pois nem `integer` nem `boolean` possuem método `add`. Estas declarações causariam erro de compilação, que seria sinalizado na análise do método `sum` de `Store[integer]` e `Store[boolean]`. No caso geral, estes errros só são detectados na compilação da classe instanciada. Isto implica em que o código fonte (o texto) dos métodos da classe parametrizada devem ser do conhecimento do compilador na instanciação da classe. Como conseqüência, em geral classes parametrizadas não podem ser compiladas e colocadas em uma biblioteca de arquivos objetos.[9] É necessário que o código delas seja conhecido para que possam ser utilizadas.

Suponha que uma classe `A` possua o método `add` que tome objeto de `A` como parâmetro. Então não haverá erro na declaração

```
var v_A : Store[A];
```

Na manutenção do *software*, é comum reescrever métodos para alcançar diferentes objetivos. Por exemplo, o método `add` poderia ser substituído por método `soma` em `sum` de `Store`, resultando em

[9] Algumas linguagens (ex: Eiffel) não duplicam o código de algumas classes parametrizadas. Então esta afirmação não é verdadeira para estas linguagens.

```
class Heap[ type T, const Max : integer ]
  private:
    var v   : array(T)[Max];
        Num : integer;
  public:
    proc ins( elem : T ) : boolean
      ...
endclass

proc main()
    { procedimento onde se inicia a execucao do programa }
  var h : Heap[ integer, 200 ];
...
```

Figura 3.16: Classe parametrizada com parâmetro que é constante

```
proc sum( k : Store[T] )
  begin
  n = n.soma( k.get() );
  end
```

Esta modificação é interna a `sum`. Ela não deveria alterar os usuários deste método. Afinal, procedimentos e métodos são mecanismos de abstração — para utilizá-los, basta ler a sua documentação e conhecer seus parâmetros. Como ele funciona e que algoritmos utiliza são abstraídos dos seus usuários.

Contudo, na troca de `add` por `soma` causa erro na declaração de `v_A`, admitindo que a classe `A` não possui método `soma`. Ou seja, temos um código compilando corretamente (declaração de `v_A`) e ele torna-se inválido por causa de mudanças internas a um método (`sum`), mudanças que não alteram a semântica (ou a documentação) deste método. Por este motivo, deve-se especificar, com comentários, que métodos os parâmetros que são tipos devem possuir. Exemplo:

```
class Store[ type T ]
  { O tipo T deve possuir o metodo
      add(T) : T
    }
  private:
  ...
endclass
```

Uma vez que a classe parametrizada é liberada para uso, não devem ser acrescentados novos métodos na lista de métodos que cada tipo que é parâmetro deve suportar.

Parâmetros de classes parametrizadas também podem ser constantes, como mostrado no exemplo da Figura 3.16.

3.7 Outros Tópicos sobre Orientação a Objetos

3.7.1 Identidade de Objetos

Identidade é a propriedade de um objeto que o distingue de todos os outros. Identidade não depende do conteúdo ou endereço do objeto. Identidade é necessária em modelagens do mundo real, onde cada

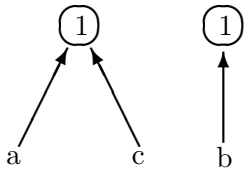

Figura 3.17: Variáveis referindo-se a números diferentes

objeto é único. Por exemplo, as transformações que uma pessoa sofre ao longo de sua existência, da infância à velhice, não modificam a sua identidade. Wegner [15] apresenta uma conversa entre duas pessoas que ilustra a distinção entre valor e identidade:

> – Smith, você mudou. Você era alto e agora está baixo. Você era magro e agora está gordo. Você tinha olhos azuis e agora tem olhos verdes.
> – Mas meu nome não é Smith.
> – Oh, então você mudou seu nome também !

Considere o código

```
proc main()
  var a, b, c : integer;
begin
a = 1;
b = 1;
c = a;
if equal(a, b)  then write('A');  endif
if a == b       then write('B');  endif
if equal(a, c)  then write('C');  endif
if a == c       then write('D');  endif
end.
```

onde `equal(a, b)` testa se `a` e `b` possuem os mesmos valores e `a == b` se `a` e `b` referenciam o mesmo objeto.

`a` e `b` são associados ao número `1`, mas cada associação cria um objeto diferente cujo valor é `1`. Assim, `a` e `b` referenciam objetos diferentes com o mesmo valor, fazendo o código acima imprimir `"ACD"`. A representação gráfica dos objetos e variáveis está na Figura 3.17.

3.7.2 Persistência

Persistência é a habilidade de certos valores persistirem entre ativações de um programa. Os valores persistentes devem ser armazenados em um arquivo e posteriormente recuperados.

A motivação para o suporte à persistência é que o mapeamento entre as estruturas de dados usadas no programa (listas, vetores, tabelas *hash*, etc) e as estruturas dos arquivos ou base de dados não é trivial. Este mapeamento utiliza tipicamente 30% do código de um sistema. E permite que os dados sejam armazenados com um tipo e recuperados com outro (erro de tipos — arquivos não possuem tipo !).

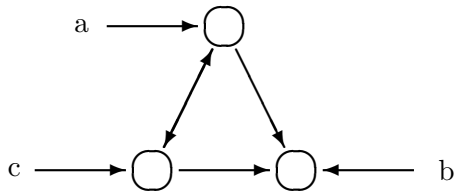

Figura 3.18: Um objeto complexo

Na linguagem S, um objeto `a` é feito persistente pelo comando

```
a.store(NomeArq);
```

onde `store` é o nome de um método que todas as classes possuem automaticamente (não é declarada pelo usuário). `NomeArq` é uma *string* com o nome do arquivo que conterá os dados.

O método `store` grava não só as variáveis de instância de `a` como também os objetos referenciados indiretamente por elas. Os tipos de cada objeto também são armazenados.

A estrutura de dados armazenada em um arquivo é recuperada com método `retrieve`:

```
a.retrieve(NomeArq);
```

A tentativa de recuperar dados com tipos diferentes daqueles armazenados causa um erro de execução. Por exemplo, haveria erro se o tipo de `a` fosse `A` e o armazenado em `NomeArq` fosse do tipo `B` que não é subclasse de `A`.

Os métodos `store` e `retrieve` manipulam adequadamente qualquer grafo de referências que o objeto possua. Por exemplo, o objeto da Figura 3.18 seria recuperado do disco com as mesmas referências que ele possuia quando estava em memória, incluindo as referências simultâneas entre `a` e `c` e as duas referências ao objeto apontado por `b`.

3.7.3 Iteradores

Iteradores são construções que permitem percorrer um conjunto tomando elemento a elemento. Um iterador pode ser uma construção especial da linguagem ou feito com métodos normais, como exemplificado na Figura 3.19.

O método `reset` inicializa o iterador. O método `next` retorna o próximo nó da lista. O procedimento `next` retorna `nil` se o fim da lista foi atingido. Um exemplo de uso de iterador é percorrer um conjunto imprimindo todos os seus elementos como mostrado na Figura 3.20.

Observe a simplicidade do uso de iteradores. O código correspondente sem eles seria muito mais complicado e necessitaria de mais variáveis, em geral. Além disto, utilizaria métodos específicos da classe `List` (estamos admitindo que todos os iteradores são formados por métodos com nomes `reset` e `next`).

Iteradores são mecanismos de abstração — eles eliminam detalhes que não precisamos saber para percorrer um conjunto e obter elemento a elemento. Eles podem ser usados com uma variada gama de tipos abstratos e estruturas de dados, como listas, árvores binárias, tabelas *hash*, *heap*, etc.

3.8 Discussão Sobre Orientação a Objetos

Dentre todos os paradigmas de linguagens, o orientado a objetos é o que possui melhores mecanismos para representação do mundo real em programas. Os elementos da realidade e as estruturas de dados

```
class List
  private:
      { head aponta para o inicio da lista. O tipo de cada no da lista
        é ListNo. }

    var head : ListNo;
        current : ListNo;
  public:
    ...
    proc reset()
      begin
      current = head;
      end
    proc next() : ListNo
        var elem : ListNo;
      begin
      elem = current;
      current = current.suc;
      return elem;
      end
endclass
```

Figura 3.19: Exemplo de Iterador construído com métodos

```
proc main()
  var
    s : List;
    elem : ListNo;
   ...
begin
...
s.reset();
while (elem = s.next()) <> nil do
  elem.write();
end.
```

Figura 3.20: Uso de iterador de uma lista

são representados claramente no programa por meio de classes. Elementos como Pessoa, Governo, Empresa, Balanço de Pagamentos, Texto, Janela, Ícone, Relógio, Carro, Trabalhador, Pilha, Lista, e Fila são representados diretamente por meio de classes. O mapeamento claro entre o mundo real e programas torna mais fácil a compreensão e a manutenção do código. Não só os programas espelham o mundo como é relativamente fácil descobrir o que deve ser modificado no código quando há alguma alteração no mundo real.

Herança permite reaproveitar elegantemente código de superclasses. Uma subclasse define apenas os métodos que devem ser diferentes da superclasse. Hierarquias de herança são criadas incrementalmente com o tempo. As novas subclasses acrescentam funcionalidades ao código existente exigindo poucas ou nenhuma modificação deste.

Polimorfismo é o motivo da alta taxa de reaproveitamento de código encontrada em sistemas orientados a objeto. Código existente pode passar a trabalhar com subclasses sem necessidade de nenhuma alteração. Proteção de informação, estimulada ou mesmo requerida por muitas linguagens orientadas a objeto, impede que modificações nas estruturas de dados de uma classe invalidem outras classes. Este conceito é fundamental para a construção de sistemas, aumentando substancialmente a sua manutenabilidade. De fato, proteção de informação é considerada mais importante do que herança pela comunidade de orientação a objetos.

Capítulo 4

Linguagens Funcionais

4.1 Introdução

Linguagens funcionais consideram o programa como uma função matemática. Todas as computações são feitas por funções que tomam como parâmetros outras funções. Não existe o conceito de variável onde um valor pode ser armazenado por meio da atribuição e utilizado posteriormente. Para compreendermos o paradigma funcional precisamos estudar primeiro o paradigma imperativo, descrito a seguir.

Uma linguagem é chamada imperativa se ela baseia-se no comando de atribuição e, conseqüentemente, em uma memória que pode ser modificada. No paradigma imperativo, variáveis são associadas a posições de memória que podem ser modificadas inúmeras vezes durante a execução do programa através do comando de atribuição. Isto é, dada uma variável `x`, um comando

```
x = expressao
```

pode ser executado várias vezes durante o tempo de vida da variável `x`.

O comando de atribuição desempenha um papel central em linguagens imperativas. Tipicamente 40% dos comandos são atribuições. Todos os outros comandos são apenas auxiliares. Nestas linguagens, o estado da computação é determinado pelo conteúdo das variáveis (que podem ser de tipos básicos, vetores ou dinâmicas) que é, por sua vez, determinado pelo fluxo de execução do programa.

Para compreender este ponto, considere um procedimento `p` que possui, no seu corpo, várias atribuições:

```
proc p( a, b : integer )
   var i, j, k : integer;
begin
i = 1;
j = a*b;
...
while k < j and j < b do
  begin
  if a > i + j
  then
    j = j + 1;
  else
    k = a + b;
  endif
  ...
```

```
  end { while }
a  = k - a;
...
end { p }
```

Para compreendermos o estado da computação após o `while`, temos que imaginar todo o fluxo de execução do algoritmo, que depende das alterações que são feitas nas variáveis visíveis dentro de `p`. Isto é difícil de compreender — os seres humanos não conseguem raciocinar corretamente neste caso porque a execução do programa é dinâmica e depende de muitos fatores (valores das variáveis).

O ponto central deste problema é o comando de atribuição. É ele que permite a alteração do valor das variáveis. O comando

```
x = x + 1
```

é um absurdo se considerada a sua interpretação matemática, mas é válido em linguagens imperativas. É válido porque o `x` do lado esquerdo se refere a uma posição de memória de um tempo futuro em relação ao `x` à direita de "=". Se o `x` da direita existir no tempo `t`, o `x` da esquerda existirá em `t + Δt`. Logo, a atribuição introduz o efeito tempo no programa, o que causa o seu comportamento dinâmico que dificulta a sua compreensão.

As linguagens imperativas foram projetadas tendo em vista as máquinas em que elas seriam usadas. Isto é, elas espelham a arquitetura da maioria dos computadores atuais, que possuem a chamada arquitetura de Von Neumann. Uma das características destas máquinas é a manipulação de uma palavra de memória por vez. Não é possível trabalhar com um vetor inteiro ao mesmo tempo, por exemplo. A restrição "uma palavra de memória por vez"é o gargalo das máquinas Von Neumann. É um dos fatores (o principal) que impede a sua eficiência. Este tipo de máquina realiza computações alterando posições de memória, fazendo desvios e testes. As linguagens imperativas, que espelham computadores Von Neumann, seguem esta filosofia. Como conseqüência, o estado da computação em um certo ponto depende dos valores das variáveis, que dependem do fluxo de execução, que depende dos valores das variáveis e assim por diante.

A atribuição

```
a = b
```

liga o significado de `a` ao de `b`. Após várias atribuições, temos um emaranhado de ligações entre variáveis (ou entre variáveis e expressões) cuja semântica torna-se difícil de entender.

A solução para eliminar características dinâmicas dos algoritmos é eliminar o comando de atribuição. Eliminando-se este comando, devem ser eliminados os comandos de repetição como `for`, `while`, `do-while`, `repeat-until`. Eles dependem da alteração de alguma variável para que possam parar.

Passagem de parâmetros por referência também deixa de ter sentido, pois uma variável deste tipo deve ser alterada dentro da rotina, o que não pode ser conseguido sem atribuição. Variáveis globais não podem existir, uma vez que elas não podem ser alteradas. Mas podem existir constantes globais e locais.

Em passagem de parâmetros, o valor dos parâmetros reais é copiado nos parâmetros formais. Isto é chamado de inicialização e é diferente de atribuição. Inicialização é a criação de uma posição de memória e a colocação de um valor nesta posição imediatamente após a sua criação. Após a inicialização, o valor armazenado nesta memória não pode ser modificado (em linguagens funcionais).

Linguagens que não possuem comando de atribuição são chamadas de linguagens declarativas. Linguagens funcionais são linguagens declarativas em que o programa é considerado uma função matemática. Na maioria das linguagens funcionais o mecanismo de repetição de trechos de código é a recursão.

Uma comparação entre programação funcional (recursão, sem atribuição) e imperativa (repetição, atribuição) é feita abaixo utilizando-se a linguagem S.

```
  { fatorial imperativo }
proc fat( n : integer ) : integer
  var i, p : integer;
begin
p = 1;
for i = 1 to n do
  p = i*p;
return p;
end

  { fatorial funcional }
proc fat( n : integer ) : integer
begin
if n == 0
then
   return 1;
else
  return n*fat(n-1);
endif
end
```

A primeira função `fat` possui duas variáveis locais e três atribuições. Como já foi escrito, variáveis e atribuições dificultam o entendimento do programa. Esta rotina possui também uma iteração (`for`) e precisamos executar mentalmente os passos desta iteração para assegurar a correção do algoritmo. A segunda função `fat` não possui nenhuma variável local nem atribuição. Não há comando de repetição. Como conseqüência, o significado do algoritmo é dado estaticamente. Não precisamos imaginar o programa funcionando para compreendê-lo. Também não é necessário "desenrolar"as chamadas recursivas.

Esta é a diferença entre linguagens imperativas e declarativas. As primeiras possuem significado que depende da dinâmica do programa e as segundas possuem significado estático. As linguagens declarativas aproveitam toda a nossa habilidade matemática já que esta disciplina é baseada principalmente em relações estáticas.

Linguagens funcionais puras (LF) não possuem comandos de atribuição, comandos de repetição, passagem de parâmetros por referência, variáveis globais, seqüência de instruções (colocada entre `begin-end` em S), variáveis locais, ponteiros. LF expressam algoritmos por meio de formas funcionais, que são mecanismos de combinação de funções para a criação de outras funções. Na maioria das LF, a única forma funcional é a composição de funções:

`h(x) = f∘g(x) = f(g(x))`

Por exemplo, podemos construir a função que é combinação de `n` elementos tomados `i` a `i`, chamada de `comb(n,i)`, a partir da função fatorial:

```
proc comb( n, i : integer ) : integer
begin
  return fat(n) div ( fat(n-i)*fat(i) );
end
```

Os operadores aritméticos (`*`, `+`, `/`, `-`, etc) também são funções no sentido funcional do termo.

Em uma expressão formada apenas por variáveis, em uma linguagem imperativa, não há efeitos colaterais e não precisamos nos preocupar onde o computador armazena os resultados intermediários

do cálculo. Por não existir efeitos colaterais, uma função em uma LF retorna sempre o mesmo valor se forem passados os mesmos parâmetros. Assim, é possível avaliar em paralelo as funções presentes em uma expressão. Em

```
... f( g(x), h(x) ) + p(y);
```

pode-se calcular `g(x)`, `h(x)` e `p(y)` ao mesmo tempo (ou `f(...)` e `p(y)`) alocando um processador para calcular cada função. Observe que, se houvesse passagem por referência ou variáveis globais, a ordem de chamada destas funções poderia influenciar o resultado.

Uma função cujo valor de retorno depende apenas dos valores dos parâmetros possui transparência referencial (TR). Isto é, dados os mesmos parâmetros, os valores de retorno são idênticos. Linguagens com transparência referencial são aquelas onde todas as funções apresentam esta característica (ex: linguagens funcionais puras). Uma consequência deste fato é a elevação do nível de abstração — há menos detalhes para serem compreendidos. Por exemplo, é mais fácil entender como uma função funciona porque as suas partes, compostas por expressões, são independentes entre si. O resultado de um trecho não afetará de modo algum outro trecho, a menos que o primeiro trecho seja uma expressão passada como parâmetro ao segundo. Em linguagens com atribuição, o resultado de um pedaço de código altera necessariamente outros segmentos do programa e de uma forma que depende do fluxo de execução.

Em uma LF, tudo são funções, inclusive o comando `if` de seleção, que possui a seguinte forma:

```
  if exp then exp1 else exp2
```

que seria equivalente a uma função de forma explícita

```
  if (exp, exp1, exp2)
```

Não há necessidade de `endif` pois após o `then` ou o `else` existe exatamente *uma* expressão. O ponto e vírgula após a expressão também é desnecessário pois ele separa instruções que não existem aqui. Utilizando este `if`, a função fatorial ficaria

```
proc fat( n : integer ) : integer
  is
    if n == 0
    then
      1
    else
      n*fat(n-1);
```

Utilizaremos esta sintaxe no restante deste capítulo. O corpo da função é colocado após `is` e é formado por uma única expressão.

Uma conseqüência da transparência referencial é a regra da reescrita: cada chamada de função pode ser substituída pelo próprio corpo da função. Assim, para calcular `fat(2)`, podemos fazer:

```
fat(2) = if 2 == 0 then 1 else 2*fat(1) =
         if 2 == 0 then 1 else 2*
            (if 1 == 0 then 1 else 1*fat(0)) =
         if 2 == 0 then 1
         else
           2*(if 1 == 0
              then
                1
              else
                1*(if 0 == 0 then 1 else 0*fat(-1)) )
```

```
(def membro (lambda(x L)
   (cond ( (null L)        nil)
         ( (eq x (car L))  T)
         ( T               (membro x (cdr L)) )
     )
  ))
```

Figura 4.1: Função `membro` em Lisp

```
proc membro( x, L )
  is
    if L == nil
    then
      false
    else
      if x == car(L)
      then
        true
      else
        membro( x, cdr(L) );
```

Figura 4.2: Função `membro` com a sintaxe de S

Avaliando, temos
`fat(2) = 2*1*1 = 2`
O processo acima é chamado de redução e é o meio empregado para executar programas em linguagens funcionais, pelo menos conceitualmente.

4.2 Lisp

Esta seção apresenta algumas das características da linguagem Lisp, a primeira linguagem funcional. Nesta linguagem tudo é representado por listas: o próprio programa, suas funções e os dados que ele utiliza. Listas nesta linguagem são delimitadas por `(` e `)`:

```
(3 carro 2.7)
( (3 azul) -5)
```

A Figura 4.1 mostra uma função `membro` que toma uma lista `L` e um valor `x` como parâmetros que retorna `true` (`T`) se o valor está na lista e `nil` (`false` em Lisp) caso contrário. Em Lisp, `cond` é um `if` estendido para manipular várias expressões. Neste caso, há três expressões, "`(null L)`", "`(eq x (car L))`" e "`T`". Se a primeira expressão for verdadeira, a instrução `cond` returnará `nil`. A função `car` retorna o primeiro elemento da lista e `cdr` retorna a lista retirando o primeiro elemento. Exemplo:
`(car '(1 2 3))` $\longrightarrow$ `1`
`(cdr '(1 2 3))` $\longrightarrow$ `(2 3)`
Após a seta é mostrado o resultado da avaliação da expressão. A comparação de igualdade é feita com `eq`, sendo que "`(eq x (car L))`" compara `x` com `(car L)`. Uma função em S equivalente à função da Figura 4.1 em Lisp está na Figura 4.2.

Tudo o que vem após `(` é considerado uma aplicação de função, a menos que `'` preceda o `(`. Exemplo:

`'(a b c)` $\longrightarrow$ `(a b c)`
`(membro a '(b c a))` $\longrightarrow$ `T`
`(+ 2 (* 3 5))` $\longrightarrow$ `17`
`(comb 5 3)` $\longrightarrow$ `10`

`(comb 5 3)` chama a função `comb` com `5` e `3` como parâmetros.

Não há especificação de tipos na declaração de variáveis — a linguagem é dinamicamente tipada. Logo, todas as funções são polimórficas e podem ocorrer erros de tipo em execução. A função `membro`, por exemplo, pode ser usada em listas de inteiros, reais, símbolos, etc. Exemplo:

`(membro 3 '(12 98 1 3))` $\longrightarrow$ `T`
`(membro azul '(3 verde 3.14 amarelo))` $\longrightarrow$ `nil`

Um erro de execução ocorre na chamada
`(delta azul verde 5)`
da função `delta`:

```
(def delta (lambda (a b c)
   (- (* b b) (* 4 a c))
   )
)
```

Os parâmetros `a` e `b` recebem `azul` e `verde` sobre os quais as operações aritméticas não estão definidas.

Lisp utiliza a mesma representação para programas e dados — listas. Isto permite a um programa construir listas que são executadas em tempo de execução pela função `Eval`:

`(Eval L)`

A função `Eval` tratará `L` como uma função e a executará.

Um grande número de dialetos foi produzido a partir de Lisp, tornando praticamente impossível transportar programas de um compilador para outro. Para contornar este problema, foi criada a linguagem Common Lisp que incorpora facilidades encontradas em vários dialetos de Lisp. A inclusão de orientação a objetos em Common Lips resultou na linguagem Common Lisp Object System, CLOS.

4.3 A Linguagem FP

Outro exemplo de linguagem funcional é FP, projetada por John Backus, o principal projetista de Fortran. FP é puramente funcional, não possui variáveis e oferece muitas possibilidades de combinar funções além da composição.

Uma seqüência de elementos em FP é denotada por $< a_1, a_2, \ldots a_n >$ e a aplicação de uma função `f` ao parâmetro `x` (que pode ser uma seqüência) é denotada por `f:x`. A função `FIRST` extrai o primeiro elemento de uma seqüência e `TAIL` retorna a seqüência exceto pelo primeiro elemento:
`FIRT` : $< 3, 7, 9, 21 > \longrightarrow 3$
`TAIL` : $< 3, 7, 9, 21 > \longrightarrow < 7, 9, 21 >$

A única forma funcional (mecanismo de combinar funções) na maioria das LF é a composição. Em FP, existem outras formas funcionais além desta, sendo algumas delas citadas abaixo.

1. Composição.
   `(f∘g) : x` $\equiv$ `f:(g:x)`
   Exemplo:

DEF quarta $\equiv$ (SQR$\circ$SQR):x

2. Construção.
   [f_1, f_2, ... f_n]:x $\equiv$< f_1:x, ..., f_n:x>
   Exemplo:
   [MIN, MAX]: $< 0, 1, 2 > \equiv$ <MIN:$< 0, 1, 2 >$, MAX:$< 0, 1, 2 >> \equiv < 0, 2 >$

3. Aplique a todos
   α f:x $\equiv$
   ```
   if x == nil then nil
   else if x eh a sequencia <x1, x2, ...xn >
        then
          <f:x1, ..., f:xn >
   ```
   nil é a lista vazia.
   Exemplo:
   α SQR:$< 3, 5, 7 > \equiv$ <SQR:3, SQR:5, SQR:7 $> \equiv < 9, 25, 49 >$

4. Condição
   (IF p f g):x $\equiv$ if p:x == T then f:x else g:x
   T é um átomo que representa true.
   Exemplo
   ```
   (IF PRIMO SOMA1 SUB2):29
   ```

5. Enquanto
   ```
   (WHILE p f):x ≡ if p:x == T then (WHILE p f): (f:x) else x
   ```

   Esta forma funcional aplica f em x enquanto a aplicação de p em x for verdadeira (T).

4.4 SML - Standard ML

SML é uma linguagem funcional fortemente tipada e com um alto grau de polimorfismo. Este polimorfismo é semelhante ao de classes parametrizadas e determinado automaticamente pelo compilador, que analisa cada função e determina a forma mais genérica que ela pode ter. Antes de estudar esta funcionalidade, veremos alguns tópicos básicos desta linguagem.

Além de tipos básicos (integer, boolean, string, etc.), SML suporta listas de forma semelhante a LISP. Uma lista com os três primeiros números é

```
[1, 2, 3]
```

e a lista vazia é []. Sendo SML fortemente tipada, listas heterogêneas (elementos de vários tipos) são ilegais.

Os parâmetros de uma função podem estar declarados sem tipo:

```
proc succ(n)
  is
    n + 1;
```

O compilador descobre que `n` deve ser inteiro, pois a operação `+` (em SML), exige que os seus operandos sejam do mesmo tipo. Como `1` é do tipo `integer`, `n` deve ser `integer` e o resultado também.

O compilador produz o seguinte cabeçalho para `succ`:

```
proc succ(n : integer) : integer
```

O tipo desta função não envolve o nome, sendo representado como

$$\texttt{integer} \longmapsto \texttt{integer}$$

O tipo de uma função

$$\texttt{proc f(} x_1 \texttt{ : } T_1\texttt{; } x_2 \texttt{ : } T_2\texttt{; ...} x_n \texttt{ : } T_n \texttt{) : R}$$

é expresso como

$$T_1 \times T_2 \times \ldots T_n \longmapsto \texttt{R}$$

O tipo da função é o tipo dos parâmetros e do valor de retorno, sendo os primeiros separados por $\times$.

Veja outros dois exemplos dados a seguir.

a)
```
proc calcula(a, b)
  is
    if b > a
    then
      1
    else
      b
```

Para que a função possua tipos corretos, as expressões que se seguem ao `then` e ao `else` devem possuir o mesmo tipo. Assim, `b` (`else`) possui o mesmo tipo que `1` (do `then` — `integer`). As operações de comparação (ex: `>`) só se aplicam a valores do mesmo tipo. Logo, `a` é do mesmo tipo que `b`. O tipo final de `Calcula` é:

$$\texttt{integer} \times \texttt{integer} \longmapsto \texttt{integer}$$

b)
```
proc inutil(a, b)
  is
    if a > 1
    then
      inutil(b-1, a)
    else
      a
```

Por "`a > 1`", `a` é inteiro. Por "`inutil(b-1, a)`", `b` também deve ser inteiro por dois motivos:

- Está em uma subtração com um inteiro ("`b-1`").
- `a` é passado como parâmetro real onde o parâmetro formal é `b`, e `a` é inteiro. O tipo do valor de retorno é igual ao tipo de `a`.

O tipo de `inutil` é:

$$\texttt{integer} \times \texttt{integer} \longmapsto \texttt{integer}$$

Algumas vezes o compilador não consegue deduzir os tipos e há erro, como em

```
proc soma(a, b)
  is
    a + b;
```

Considerando que o operador `+` pode ser aplicado tanto a reais como inteiros, o tipo de `soma` poderia ser qualquer um dos abaixo

`integer` $\times$ `integer` $\longmapsto$ `integer`
`real` $\times$ `real` $\longmapsto$ `real`

e, portanto, há ambigüidade, que é resolvida colocando-se pelo menos um dos tipos (de `a`, `b` ou do valor de retorno). Exemplo:

```
proc soma(a : integer; b) is ...
proc soma(a, b) : integer is ...
```

Se a expressão do `then` e do `else` de um `if` pudessem ser de tipos diferentes, poderia haver erros de tipo em execução, como o abaixo.

```
proc f(a)
  is
    if a > 1
    then
      1
    else
      "Eu sou um erro"

proc g
  is
    f(0) + 1;
```

`f(0)` retorna uma *string* à qual tenta-se somar um inteiro. Por causa das restrições impostas pelo sistema de tipos, erros de execução como este nunca ocorrem em programas SML.

A função

```
proc id(x)
  is
    x;
```

pode ser usada com valores de qualquer tipo, e é válida na linguagem. O seu tipo é
`'a` $\longmapsto$ `'a`
onde `'a` significa um tipo qualquer. Se houvesse mais um parâmetro e este pudesse ser de um outro tipo qualquer, este seria chamado de `'b`.

Um outro exemplo é a função

```
proc nada(x, y)
  is
    x;
```

cujo tipo é
`'a` $\times$ `'b` $\longmapsto$ `'a`
Outra dedução de tipo é apresentada abaixo

```
proc escolhe(i, a, b)
  is
    if i > 0
    then
      a
    else
      b
```

O tipo de `escolhe` é:
`integer` $\times$ `'a` $\times$ `'a` $\longmapsto$ `'a`

A dedução dos tipos corretos para as variáveis é feito por um algoritmo que também determina se há ambigüidade ou não. Este é um fato importante: *a definição de SML utiliza não apenas definições estáticas mas também dinâmicas (algoritmos).* Ist é uma qualidade, pois aumenta o polimorfismo, mas também um problema. Como algoritmos são mais difíceis de entender do que relações estáticas, o programador necessita de um esforço mais para decidir se o código que ele produziu em SML é válido ou não.

Listas são delimitadas por `[` e `]`, como `[1, 2, 3]`, e possuem um tipo que termina sempre com a palavra `list`. Alguns exemplos de tipos de listas estão na tabela seguinte.

Lista	Tipo
`[1, 2, 3]`	`integer list`
`["a", "azul", "b"]`	`string list`
`[ [1, 2], [3], [4] ]`	`integer list list`

O construtor `::` constrói uma lista a partir de um elemento e de outra lista. Exemplos:
`1::[2,3]` resulta em `[1, 2, 3]`.

A aplicação da função

```
proc ins(a : integer; L : integer list) : integer list
  is
    a::L;
```

sobre `1` e `[2, 3]` resulta em `[1, 2, 3]`. Isto é, `ins(1, [2, 3])` $\longrightarrow$ `[1, 2, 3]`.

O tamanho de uma lista pode ser calculado pela função `len`:

```
proc  len([])   is  0
    | len(h::t) is  1 + len(t);
```

A função `len` possui, na verdade, duas definições. Uma para a lista vazia e outra para listas com pelo menos um elemento. As definições são separadas por `|`. Em uma chamada

```
len([1, 2, 3])
```

é feito o emparelhamento do parâmetro `[1, 2, 3]` com a segunda definição de `len`, resultando nas seguintes inicializações:

```
h = 1
t = [2, 3]
```

Então a expressão `1 + len([2, 3])` é calculada e retornada.

De um modo geral, em uma chamada

```
len(L)
```

é utilizada uma das definições de `len` de acordo com o parâmetro `L`. A presença de um `if` em `len`, como

```
if L == [] then 0 else ...
```

torna-se desnecessária. A programação com emparelhamento é ligeiramente mais abstrata (alto nível) do que com `if`.

Admitindo que todos os tipos suportam a operação de igualdade, uma função que testa a presença de `x` em uma lista é:

```
proc   membro(x, [])
         is false
     | membro(x, h::t)
         is
           if x == h
           then
             true
           else
             membro(x,t);
```

E o seu tipo é
`'a` $\times$ `'a list` $\longmapsto$ `boolean`

4.5 Listas Infinitas e Avaliação Preguiçosa

Linguagens funcionais freqüentemente suportam estruturas de dados potencialmente infinitas. Por exemplo,

```
ones = 1 : ones
```

é uma lista infinita de `1`'s em Haskell. A função

```
proc numsFrom( n : integer )
  is
    [n : numsFrom(n + 1)]
```

retorna uma lista infinita de números naturais começando em `n`. Naturalmente, um programa não usa uma lista infinita já que ele termina em um tempo finito. Estas listas são construídas à medida que os seus elementos vão sendo requisitados, em uma maneira preguiçosa (*lazy evaluation*).

Este mecanismo é usado para facilitar a implementação de algoritmos e mesmo para aumentar a eficiência da linguagem. Por exemplo, a função [17]

```
proc cmpTree( tree1, tree2 )
  is
    cmpLists( treeToList(tree1), treeToList(tree2) );
```

compara duas árvores pela comparação dos nós das árvores colocados em forma de lista. A função `treeToList` converte a árvore para lista de maneira preguiçosa. Assim, se o primeiro elemento das duas árvores forem diferentes, `cmpLists` retornará `false`, terminando a função `cmpTree`. Sem construção preguiçosa da lista, seria necessário construir as duas listas totalmente antes de começar a fazer o teste e descobrir que as listas são diferentes logo no primeiro elemento.

4.6 Funções de Ordem Mais Alta

A maioria das linguagens modernas permitem que funções sejam passadas como parâmetros. Isto permite a construção de rotinas genéricas. Por exemplo, pode-se construir uma função `max` que retorna o maior elemento de um vetor qualquer. A operação de comparação entre dois elementos é passada a `max` como uma função. Em uma linguagem sem tipos, `max` seria:

```
proc max( v, n, gt )
  var maior, i;
begin
maior = v[1];
for i = 2 to n do
  if gt(v[i], maior)
  then
    maior = v[i];
  endif
return maior;
end
```

O código de `max` pode ser utilizado com vetores de qualquer tipo `T`, desde que se defina uma função de comparação para o tipo `T`. Exemplo:

```
proc gt_real(a, b)        { para numeros reais }
begin
return a > b;
end
...
m =  max( VetReal,  gt_real );
m1 = max( VetNomes, gt_string );
```

Funções que admitem funções como parâmetros são chamadas funções de mais alta ordem ("*higher order functions*").

Uma função `map` em SML que aplica uma função `f` a todos os elementos de uma lista, produzindo uma lista como resultado, seria:

```
proc   map( proc f('a) : 'b; []   ) is []
     | map( proc f('a) : 'b; h::t )
       is
         f(h)::map(t);
```

seu tipo é:
`('a` $\longmapsto$ `'b)` $\times$ `'a list` $\longmapsto$ `'b list`

Observe que funções como parâmetro são completamente desnecessárias em linguagens orientadas a objeto pois cada objeto é associado a um conjunto de métodos. Quando um objeto for passado como parâmetro, teremos o efeito de passar também todos os seus métodos como parâmetro simulando funções de ordem mais alta.

4.7 Discussão Sobre Linguagens Funcionais

A necessidade de eficiência fez com que na maioria das linguagens funcionais fossem acrescentadas duas construções imperativas, a saber, seqüência e atribuição. Seqüência permite que as instruções de

uma lista sejam executadas sequencialmente, introduzindo a noção de tempo. No exemplo seguinte, esta lista está delimitada por `begin-end`.

```
begin
a = a + 1;
if a > b
then
  return f(a, b)
else
  return f(b, a)
end
```

Obviamente, seqüência só tem sentido na presença de atribuição ou entrada/saída de dados, pois de outro modo o resultado de cada instrução da seqüência seria uma expressão cujo resultado seria perdido após a sua avaliação.

Programadores produzem aproximadamente a mesma quantidade de linhas de código por ano, independente da linguagem. Assim, quanto mais alto nível a linguagem é, mais problemas podem ser resolvidos na mesma unidade de tempo. Uma linguagem é de mais alto nível que outra por possuir menos detalhes, o que implica em ser mais compacta (necessita de menos construções/instruções para fazer a mesma coisa que outra). Como linguagens funcionais são de mais alto nível que a maioria das outras, elas implicam em maior produtividade para o programador.

Vários fatores tornam linguagens funcionais de alto nível, como o uso de recursão ao invés de iteração, ausência de atribuição e alocação e desalocação automática de memória. Este último item é particularmente importante. Não só o programador não precisa desalocar a memória dinâmica (há coleta de lixo) mas ele também não precisa alocá-la explicitamente. As listas utilizadas por linguagens funcionais aumentam e diminuem automaticamente, poupando ao programador o trabalho de gerenciá-las.

É mais fácil definir uma linguagem funcional formalmente do que linguagens de outros paradigmas, assim como programas funcionais são adequados para análise formal. A razão é que as linguagens deste paradigma possuem um parentesco proximo com a matemática, facilitando o mapeamento da linguagem ou programa para modelos matemáticos.

Há dois problemas principais com linguagens funcionais. Primeiro, um sistema real é mapeado em um programa que é uma função matemática composta por outras funções. Logo, não há o conceito de *estado* do programa dado pelas variáveis globais, dificultando a implementação de muitos sistemas que exigem que o programa tenha um estado. Estes sistemas não são facilmente mapeados em funções matemáticas. De fato, o paradigma que representa melhor o mundo real é o orientado a objetos. O conceito de objeto é justamente um bloco de memória (que guarda um estado) modificado por meio de envio de mensagens.

Uma outra face deste problema é entrada e saída de dados em linguagens funcionais. Funções que fazem entrada e saída não suportam transparência referencial. Por exemplo, uma função `getchar()` que retorna o próximo caráter da entrada padrão provavelmente retornará dois valores diferentes se for chamada duas vezes.

O segundo problema com linguagens funcionais é a eficiência. Elas são lentas por não permitirem atribuições. Se, por exemplo, for necessário modificar um único elemento de uma lista, toda a lista deverá ser duplicada. Este tipo de operação pode ser otimizada em alguns casos[1] pelo compilador ou o programador pode encontrar maneiras alternativas de expressar o algoritmo. Neste último caso,

[1] Este tópico não será discutido aqui.

é provável que o modo alternativo de codificação seja difícil de entender por não ser o mais simples possível.

Máquinas paralelas podem aumentar enormemente a eficiência de programas funcionais. Contudo, esta tecnologia ainda não está suficientemente madura para concluirmos que linguagens funcionais são tão eficientes quanto linguagens imperativas.

O uso de atribuição em um programa não elimina todos os benefícios da programação funcional. Um bom programador limita as atribuições ao mínimo necessário à eficiência, fazendo com que grande parte do programa seja realmente funcional. Assim, pelo menos esta parte do código será legivel e fácil de ser paralelizada e otimizada, que são as qualidades associadas à programação funcional.

Capítulo 5

Prolog — Programming in Logic

5.1 Introdução

Prolog é uma linguagem lógica. Ela permite a definicão de fatos e de relacionamentos entre objetos. Nesta linguagem objeto é designa valores de qualquer tipo. Um programa em Prolog consiste de fatos e regras. Um fato é uma afirmação sempre verdadeira. Uma regra é uma afirmação cuja veracidade depende de outras regras ou fatos. Para exemplificar estes conceitos, utilizaremos o seguinte programa em Prolog:

```
homem(jose).
homem(joao).
homem(pedro).
homem(paulo).
mulher(maria).
mulher(ana).
pais(pedro, joao, maria).
pais(paulo, joao, maria).
pais(maria, jose, ana).
```

Neste código só há fatos e cada um deles possui um significado. "`homem(X)`" afirma que `X` é homem e `pais(F, H, M)` significa que `F` é filho de pai `H` e mãe `M`. Este programa representa uma família na qual

- José e Ana são pais de Maria;
- João e Maria são pais de Pedro e Paulo

As informações sobre a família podem ser estendidas por novos fatos ou regras, como pela regra

```
irmao(X, Y) :-
   homem(X),
   pais(X, H, M),
   pais(Y, H, M).
```

A regra acima será verdadeira se as regras que se seguem a `:-` (que funciona como um `if`) forem verdadeiras. Isto é, `X` será irmão de `Y` se `X` for homem e possuir os mesmos pais `H` e `M` de `Y`. A vírgula funciona como um *and* lógico.

Prolog admite que todos os identificadores que se iniciam com letras maiúsculas (como `X`, `Y`, `H` e `M`) são nomes de variáveis. Nomes iniciados em minúscula são *símbolos*. Números (`1`, `52`, `3`) e símbolos são tipos de dados básicos da linguagem e são chamados de *átomos*.

Prolog é uma linguagem interativa que permite a formulação de perguntas através de *goals*. Um *goal* é uma meta que desejamos saber se é verdadeira ou falsa e em que situações. Por exemplo, se quisermos saber se `pedro` é homem, colocamos

```
?- homem(pedro).
```

e o sistema responderá

```
yes
```

sendo que "`homem(pedro)`" é o *goal* do qual queremos saber a veracidade.

Fatos, regras e *goals* são exemplos de *cláusulas*. Um *predicado* é um conjunto de fatos e/ou regras com o mesmo nome e número de argumentos. O programa exemplo definido anteriormente possui os predicados `homem`, `mulher`, `pais` e `irmao`. Veremos adiante que predicado é o equivalente a procedimento em outras linguagens. O conjunto de todos os predicados forma a *base de dados* do programa.

Um *goal* pode envolver variáveis:

```
?- mulher(M).
```

O objetivo desta questão é encontrar os nomes das mulheres armazenados na base de dados. O sistema de tempo de execução de Prolog tenta encontrar os valores de `M` que fazem esta cláusula verdadeira. Ele rastreia todo o programa em busca da primeira cláusula com nome "`mulher`". É encontrado

```
mulher(maria)
```

e é feita a associação

```
M = maria
```

Neste ponto, um valor de `M` que torna `mulher(M)` verdadeiro é encontrado e o sistema escreve a resposta:

```
?- mulher(M).
M = maria
```

Se o usuário digitar `;` (ponto-e-vírgula), Prolog retornará às cláusulas do programa e:

- tornará inválida a associação de `M` com `maria`. Então `M` volta a não estar instanciada — não tem valor. A associação entre uma variável e um valor é chamado de instanciação. Antes de uma instanciação, a variável é chamada de livre e não está associada a nada. Após uma instanciação, uma variável não pode ser instanciada novamente, exceto em *backtracking* (que será visto adiante), quando a instanciação anterior deixa de existir.

- continuará a procurar por cláusula que emparelhe com "`mulher(M)`" tornando esta cláusula verdadeira. Neste processo `M` será instanciada. Esta procura se iniciará na cláusula seguinte à última encontrada. A última foi "`mulher(maria)`" e a seguinte será "`mulher(ana)`".

Então, a busca por `mulher` continuará em `mulher(ana)` e `M` será associado a `ana`:

```
?- mulher(M).
M = maria;
M = ana
```

Digitando ; a busca continuará a partir de

```
pais(pedro, joao, maria)
```

e não será encontrada nenhuma cláusula `mulher`, no que o sistema responderá "`no`":

```
?- mulher(M).
M = maria;
M = ana;
no
```

O algoritmo que executa a busca, por toda a base de dados, por cláusula que emparelha dado *goal* é chamado de algoritmo de unificação. Dizemos que um fato (ou regra) emparelha (*match*) um *goal* se é possível existir uma correspondência entre os dois. Os itens a seguir exemplificam algumas tentativas de emparelhamento. Utilizamos a sintaxe

```
mulher(maria) = mulher(X)
```

para a tentativa de emparelhamento do *goal* "`mulher(X)`" com a cláusula "`mulher(maria)`".

a) `mulher(maria) = mulher(X)`
há emparelhamento e `X` é instanciado com `maria`, que indicaremos como `X = maria`.

b) `mulher(maria) = mulher(ana)`
não há emparelhamento, pois `maria` $\neq$ `ana`

c) `mulher(Y) = mulher(X)`
há emparelhamento e `X = Y`. Observe que nenhuma das duas variáveis, `X` ou `Y`, está instanciada.

d) `pais(pedro, X, maria) = pais(Y, joao, Z)`
há emparelhamento e `Y = pedro`, `X = joao` e `Z = maria`

Uma cláusula composta será verdadeira se o forem todos os seus fatos e regras. Por exemplo, considere a meta

```
?- irmao(pedro, paulo).
```

que produz um emparelhamento com `irmao(X, Y)`, fazendo `X = pedro`, `Y = paulo` e a geração dos *subgoals*

```
homem(pedro),
pais(pedro, H, M),
pais(paulo, H, M).
```

O primeiro *subgoal*, `homem(pedro)`, é verdadeiro (pelos fatos) e pode ser eliminado, restando

```
pais(pedro, H, M),
pais(paulo, H, M).
```

O *subgoal* `pais(pedro, H, M)` emparelha com `pais(pedro, joao, maria)`, produzindo as associações `H = joao` e `M = maria`. Um emparelhamento sempre é verdadeiro e, portanto, o *subgoal*

```
pais(pedro, joao, maria)
```

é eliminado e o *goal* inicial é reduzido a

```
pais(paulo, joao, maria).
```

Que é provado por um dos fatos.

Portanto, a meta

```
irmao(pedro, paulo)
```

é verdadeira.

Podemos perguntar questões como

```
?- irmao(X, Y).
```

que é substituída pelos *subgoals*

```
homem(X),
pais(X, H, M),
pais(Y, H, M).
```

O primeiro *subgoal* emparelha com `homem(jose)`, fazendo `X = jose` e produzindo

```
pais(jose, H, M)
pais(Y, H, M).
```

O primeiro *subgoal* (`pais(jose, H, M)`) não pode ser emparelhado com ninguém e falha. Esta falha causa um retrocesso (*backtracking*) ao *subgoal* anterior, `homem(X)`. A instanciação de `X` com `jose` é destruída, tornando `X` uma variável livre. A busca por cláusula que emparelha com este *goal* continua em `homem(joao)`, que também causa falha em

```
pais(joao, H, M),
pais(Y, H, M).
```

Há retrocesso para `homem(X)` e *matching* com `homem(pedro)`, fazendo `X = pedro`, e resultando em

```
pais(pedro, H, M),
pais(Y, H, M).
```

O primeiro *subgoal* emparelha com

```
pais(pedro, joao, maria)
```

fazendo

```
H = joao, M = maria
```

e resultando em

```
pais(Y, joao, maria).
```

Sempre que um *novo subgoal* dever ser satisfeito, a busca por cláusula para emparelhamento comeca na primeira cláusula do programa, independente de onde parou a busca do *subgoal* anterior (que é `pais(pedro, H, M)`).

O emparelhamento de `pais(Y, joao, maria)` é feito com `pais(pedro, joao, maria)`. O resultado final é

```
X = pedro
Y = pedro
```

Pela nossa definição, `pedro` é irmão dele mesmo. Digitando `;` é feito um retrocesso e a busca por emparelhamento para

```
pais(Y, joao, maria)
```

continua em `pais(paulo, joao, maria)`, que sucede e produz

```
X = pedro
Y = paulo
```

Observe que a ordem das cláusulas no programa é importante porque ela diz a ordem dos retrocessos. Outras regras que seriam úteis para relações familiares são dadas abaixo.

```
pai(P, F) :-              /* P é pai de F */
  pais(F, P, M).
```

```
mae(M, F) :-              /* M é mae de F */
  pais(F, P, M).

avo(A, N) :-              /* A é avo (homem) de N. A = avo, N = neto */
  homem(A),
  pai(A, F),
  pai(F, N).

avo(A, N) :-              /* A é avo (homem) de N. A = avo, N = neto */
  homem(A),
  pai(A, F),
  mae(F, N).

tio(T, S) :-              /* T é tio de S. T = tio, S = sobrinho */
  irmao(T, P),
  pai(P, S).

tio(T, S) :-              /* T é tio de S. T = tio, S = sobrinho */
  irmao(T, P),
  mae(P, S).

filho(F, P) :-            /* F é filho de P */
  homem(F),
  pai(P, F).

filho(F, M) :-            /* F é filho de M */
  homem(F),
  mae(M, F).

paimaeDe(A, D) :-         /* A é pai ou mae de D */
  pai(A, D).

paimaeDe(A, D) :-
  mae(A, D).

ancestral(A, D) :-      /* A é ancestral de D */
  paimaeDe(A, D).

ancestral(A, D) :-
  paimaeDe(A, Y),
  ancestral(Y, D).
```

Uma estrutura cumpre um papel semelhante a um registro (`record` ou `struct`) em linguagens imperativas. Uma estrutura para representar um curso da universidade teria a forma

`curso( nome, professor, numVagas, departamento )`

e poderia ser utilizada em cláusulas da mesma forma que variáveis e números:

```
professor( Nome, curso(_, Nome, _, _) ).
```

```
haVagas( curso(_, _, N, _) ) :-
  N > 0.
```

Uma estrutura pode ser atribuída a uma variável e emparelhada:

```
?- ED = curso( estruturasDeDados, joao, 30, dc ), professor(Nome, ED),
   haVagas(ED).
Nome = joao
yes

?- curso(icc, P, 60, Depart) = curso(C, maria, N, dc).
C = icc
P = maria
N = 60
Depart = dc
yes
```

O sinal = é utilizado tanto para instanciar variáveis como para comparar valores. Assim, se `X` não estiver instanciado,

```
X = 2
```

instanciará `X` com `2`. Se `X` tiver sido instanciado com o valor `3`, "`X = 2`" será avaliado como falso. Estude os exemplos abaixo.

```
cmp(A, B) :-
  A = B.

?- cmp(X, 2).
X = 2
yes

?- cmp(3, 2).
 no

?- X = 2, cmp(X, Y).
X = 2
Y = 2
yes

?- cmp(X, Y).
X = _1
Y = _1
yes
```

`_1` é o nome de uma variável criada pelo Prolog. Obviamente ela não está inicializada.

Prolog não avalia operações aritméticas à direita ou esquerda de =. Observe os exemplos a seguir.

```
?- 6 = 2*3.
no

?- X = 2*3.
```

```
X = 2*3
yes

?- 5 + 1 = 2*3.
no
```

O que seria a expressão "`2*3`" é tratada como a estrutura "`*(2,3)`". Se a avaliação da expressão for necessária, deve-se utilizar o operador `is`. Para resolver "`X is exp`" o sistema avalia `exp` e então:

- compara o resultado com `X` se este estiver instanciado ou;
- instancia `X` com o resultado de `exp` se `X` não estiver instanciado.

Observe que `exp` é avaliado e portanto não pode ter nenhuma variável livre. Pode-se colocar valores ou variáveis do lado esquerdo de `is`. Veja alguns exemplos a seguir.

```
?- X is 2*3.
X = 6
yes

?- 6 is 2*3.
yes

?- 5 + 1 is 2*3.
no

?- X is 2*3, X is 6.
X = 6
yes

?- X is 2*3, X is 3.
no

?- 6 is 2*X.
no
```

Note que o último *goal* falha pois `X` está livre.

Laços do tipo `for` podem ser simulados [17] utilizando-se o operador `is`:

```
for(0).
for(N) :-
  write(N),
  NewN is N - 1,
  for(NewN).
```

Este laço seria equivalente a

```
for i = N downto 1 do
  write(i);
```

em S onde `downto` indica laço decrescente; isto é, `N >= 1`.

Listas são as principais estruturas de dados de Prolog. Uma lista é um conjunto de valores entre colchetes:

```
[1, 2, 3]
[jose, joao, pedro]
[1, joao]
```

Uma lista também é representada por

```
[Head | Tail]
```

onde `Head` é o seu primeiro elemento e `Tail` é a sublista restante. Assim, os exemplos de listas acima poderiam ser escritos como

```
[1 | [2, 3]]
[jose | [joao, pedro]]
[1 | [joao]]
```

O emparelhamento de `[1, 2, 3]` com `[H | T]` produz

```
H = 1
T = [2, 3]
```

Para emparelhar com `[H | T]`, uma lista deve possuir pelo menos um elemento, `H`, pois `T` pode ser instanciado com a lista vazia, `[]`.

Com estas informações, podemos construir um predicado que calcula o tamanho de uma lista:

```
length([], 0).
length([H | T], N) :-
   length(T, M),
   N is M + 1.
```

Este *goal* retornará em `N` o tamanho da lista `L` ou irá comparar `N` com o tamanho da lista. Exemplo:

```
?- length([1, 2, 3], N).
N = 3
yes

?- length([], 0)
yes
```

Pode-se também especificar mais de um elemento cabeça para uma lista:

```
semestre( [1, 2, 3, 4] ).
?- semestre( [X, _, Y | T] ).
X = 1
Y = 3
T = [4]
yes
```

A seguir mostramos alguns outros exemplos de predicados que manipulam listas.

Um predicado `concat(A, B, C)` que concatena as listas `A` e `B` produzindo `C` seria

```
concat([], [], []).
concat([], [H|T], [H|T]).
concat([X|Y], B, [X|D]) :-
  concat(Y, B, D).
```

Um predicado `pertence(X, L)` que sucede se `X` pertencer à lista `L` seria

```
pertence(X, [X|_]).
pertence(X, [_|T]) :-
  pertence(X, T).
```

O predicado `numTotalVagas` utiliza a estrutura `curso` descrita anteriormente e calcula o número total de vagas de uma lista de cursos.

```
  /* numTotalVagas(N, L) significa que N é o numero total de vagas
     nos cursos da lista L */

numTotalVagas( 0, [] ).
numTotalVagas( Total, [ curso(_, _, N, _) | T ] ) :-
  numTotalVagas(TotalParcial, T),
  Total is TotalParcial + N.
```

O predicado `del(X, Big, Small)` elimina o elemento `X` da lista `Big` produzindo a lista `Small`.

```
del(X, [X|L], L).
del(X, [_|Big], Small) :-
  del(X, Big, Small).
```

5.2 Cut e `fail`

Cut é o *fato* `!` que sempre sucede. Em uma meta

```
?- pA(X), pB(X), !, pC(X).
```

o cut (`!`) impede que haja retrocesso de `pC(X)` para `!`.

Considerando a base de dados

```
pA(joao).
pA(pedro).
pB(pedro).
pC(joao).
```

a tentativa de satisfação do *goal* acima resulta no seguinte: é encontrado *matching* para `pA(X)` com `X = joao`. O *goal* `pB(joao)` falha e há retrocesso para `pA(X)`. A busca por *matching* por `pA(X)` continua, resultando em `X = pedro`. O *goal* `pB(pedro)` sucede, como também `!`. O *goal* `pC(pedro)` falha e é tentado retrocesso para `!`, que é proibido, causando a falha de todo o *goal*.

Se o cut estiver dentro de um predicado, como em

```
pD(X) :- pA(X), !, pB(X).
pD(X) :- pA(X), pC(X).
```

a tentativa de retrocesso através do `!` causará a falha de todo o predicado. Por exemplo, a meta

```
?- pD(joao).
```

emparelhará com `pD(X)`, resultando na meta

```
pA(joao), !, pB(joao)
```

`pA(joao)` sucede e `pB(joao)` falha. A tentativa de retrocesso para `!` causará a falha de todo o predicado `pD`, isto é, do *goal* `pD(joao)`. Se apenas o *subgoal* `pA(X)` da primeira cláusula do predicado `pD` falhasse, seria tentado a segunda,

```
pD(X) :- pA(X), pC(X)
```

que seria bem sucedida, já que `pA(joao)` e `pC(joao)` sucedem.

O cut é utilizado para eliminar algumas possibilidades da árvore de busca. Eliminar um retrocesso para um predicado `pA` significa que algumas possibilidades de `pA` não foram utilizadas, poupando tempo.

O operador `fail` sempre falha e é utilizado para forçar o retrocesso para o *goal* anterior. Por exemplo, o *goal*

```
?- homem(X), write(X), write(' '), fail.
jose joao pedro
no
```

força o retrocesso por todas as cláusulas que emparelham "`homem(X)`". Este operador pode ser utilizado [19] para implementar um comando `while` em Prolog:

```
while :-
  pertence( X, [1, 2, 3, 4, 5] ),
  body(X),
  fail.

body(X) :-
  write(X),
  write(' ').

?- while.
1 2 3 4 5
no
```

O operador `fail` com o cut pode ser utilizado para invalidar todo um predicado:

```
fat(N, P) :-         /* fatorial de N é P */
  N < 0, !, fail.
fat(0, 1).
fat(N, P) :-
  N > 0,
  N1 is N - 1,
  fat(N1, P1),
  P is N*P1.
```

Assim, o *goal*

```
?- fat(-5, P).
no
```

falha logo na primeira cláusula. Sem o cut/`fail`, todas as outras regras do predicado seriam testadas.

Com o cut podemos expressar o fato de que algumas regras de um predicado são mutualmente exclusivas. Isto é, se uma sucede, obrigatoriamente as outras falham. Por exemplo, `fat` poderia ser codificado como

```
fat(0, 1) :- !.
fat(N, P) :-           /* fatorial de N é P */
  N > 0,
  N1 is N - 1,
```

```
  fat(N1, P1),
  P is N*P1.
```

Assim, em

```
?- fat(0, P).
P = 1;
no
```

seria feito um emparelhamento apenas com a primeira cláusula, "`fat(0, 1)`". Sem o cut nesta cláusula, o "`;`" que se segue a "`P = 1`" causaria um novo emparelhamento como "`fat(N, P)`", a segunda cláusula do predicado, que falharia.

Observe que o cut foi introduzido apenas por uma questão de eficiência. Ele não altera em nada o significado do predicado. Este tipo de cut é chamado de cut verde.

Um cut é vermelho quando a sua remoção altera o significado do predicado. Como exemplo temos

```
  /* max(A, B, C) significa que C é o maximo entre A e B */

max(X, Y, X) :-
  X >= Y,
  !.
max(X, Y, Y).

pertence(X, [X|_]) :-
  !.
pertence(X, [_|T]) :-
  pertence(X, T).

?- max(5, 2, M).
M = 5;
no

?- pertence(X, [1, 2, 3]).
X = 1;
no
```

Retirando o cut dos predicados, teríamos

```
  /* max(A, B, C) significa que C é o maximo entre A e B */

max(X, Y, X) :-
  X >= Y.
max(X, Y, Y).

pertence(X, [X|_]).
pertence(X, [_|T]) :-
  pertence(X, T).

?- max(5, 2, M).
M = 5;
```

```
M = 2;
no

?- pertence(X, [1, 2, 3]).
X = 1;
X = 2;
X = 3;
no
```

O cut pode tanto melhorar a eficiência (verdes, vermelhos) e o poder expressivo da linguagem (verdes) como tornar o código difícil de entender (vermelhos). Freqüentemente o cut vermelho remove a bidirecionalidade dos argumentos de um predicado, como no caso de `pertence`. Sem o cut, este predicado poderia tanto ser utilizado para recuperar os elementos da lista, um a um, como para testar se um elemento pertence à lista. Com o cut, `pertence` permite recuperarmos apenas o primeiro elemento da lista.

5.3 Erros em Prolog

Prolog é uma linguagem dinamicamente tipada e, conseqüentemente, podem ocorrer erros de tipo em tempo de execução. Por exemplo, em

```
add(X, Y) :-
   Y is X + 1.

?- add ( [a, b], Y).
```

tenta-se somar `1` a uma lista.

Contudo, a maior parte dos que seriam erros de tipo simplesmente fazem as operações de emparelhamento falhar, sem causar erros em execução. Exemplo:

```
dias(jan, 31).
...
dias(dez, 31).

?- dias(N, jan).
no
```

Os compiladores Prolog geralmente não avisam se um predicado não definido é utilizado:

```
while :-
  peretnce(X, [1, 2, 3]),
  write(X),
  write(' '),
  fail.

?- while.
no
```

Neste caso, `pertence` foi digitado incorretamente como `peretnce` que nunca sucederá.

5.4 Reaproveitamento de Código

Prolog é dinamicamente tipada e portanto suporta o polimorfismo causado por esta característica. Os parametros reais passados a um predicado podem ser de qualquer tipo, o que torna todo predicado potencialmente polimorfico. Por exemplo, para o predicado

```
length([], 0).
length([_ | L], N) :-
   length(L, NL), N is NL + 1.
```

podem ser passadas como parâmetro listas de qualquer tipo, reaproveitamento o predicado:

```
?- length([greem, red], NumCores).
NumCores = 2
yes
?- lenght([1, -5, 12], NumElem).
NumElem = 3
yes
```

Em Prolog, não há definição de quais parâmetros são de entrada e quais são de saída de um predicado. De fato, um parâmetro pode ser de entrada em uma chamada e de saída em outra. Utilizando o predicado

```
pertence(X, [X | _]).
pertence(X, [_ | C]) :-
   pertence(X, C).
```

podemos perguntar se um elemento pertence a uma lista:

```
?- pertence(a, [b, e, f, a, g]).
```

e também que elementos pertencem à lista:

```
?- pertence(E, [b, e, f, a, g]).
E = b;
E = e;
E = f;
E = a;
E = g;
no
```

No primeiro caso, o parâmetro formal `X` é de entrada (por valor — `a`) e no segundo (`E`), de saída.

A conseqüência do racíocionio acima é que temos duas funções diferentes utilizando um único predicado. Logo, existe reaproveitamento de código por não ser fixo o tipo de passagem de parâmetros em Prolog. Outras linguagens exigiriam a construção de dois procedimentos, um para cada função de `pertence`.

Podemos comparar a característica acima de Prolog com lógica : dada uma fórmula do cálculo proposicional, como `((a∧b)∨c)`, e valores de algumas da variáveis e/ou resultado da expressão, podemos obter o valor das variáveis restantes. Por exemplo

- se `((a∧b)∨c) = true` e `a = true`, `c = false`, então `b` deverá ser `true`.

- se `((a∧b)∨c) = true` e `a = true`, `c = true`, `b` poderá ser `true` ou `false`.

No predicado `pertence` há uma construção semelhante:

- se `pertence(E, [b, e, f, a, g]) = true`, então `E = b` ou `E = e`, ... ou `E = g`.

`E` pode assumir diversos valores para fazer `pertence(E, [b, e, f, a, g])` `true`, da mesma forma que, na última fórmula, `b` pode assumir dois valores (`true` ou `false`) para fazer a equação verdadeira. Esta forma de reaproveitamento de código é exclusiva das linguagens lógicas e não se relaciona a polimorfismo — são conceitos diferentes.

5.5 Manipulação da Base de Dados

O predicado `assert` sempre sucede e introduz um novo fato ou regra na base de dados:

```
?- otimo( zagalo ).
no

?- assert( otimo(zagalo) ).
yes

?- otimo(zagalo).
yes
```

O predicado `retract` remove um fato ou regra da base de dados:

```
?- retract( otimo(zagalo) ).
yes

?- otimo(zagalo).
no

?- assert( otimo(luxemburgo) ).
yes
```

`assert` pode fazer um programa em Prolog "*aprender*" durante a sua execução. O que ele aprende pode ser gravado em disco e recuperado posteriormente. Por exemplo, considere um predicado `fatorial` que calcula o fatorial de um número e armazena os valores já calculados na base de dados.

```
   /* fat(N, P) significa que o fatorial de N é P */
fat(0, 1).

   /* fatorial(N, P) significa que o fatorial de N é P */
fatorial(N, P) :-
  fat(N, P).
fatorial(N, P) :-
  N1 is N - 1,
  fatorial(N1, P1),
  P is P1*N,
  assert( fat(N, P) ).
```

Inicialmente, há apenas um fato para o predicado `fat`. Quando `fatorial` for invocado, como em

```
?- fatorial(3, P).
P = 6
```

serão introduzidos na base de dados fatos da forma `fat(N, P)`. Neste caso, a base de dados conterá os seguintes fatos de `fat`:

```
fat(0, 1).
fat(1, 1).
fat(2, 2).
fat(3, 6).
```

Agora, quando o fatorial de `3` for novamente requisitado, ele será tomado da base de dados, o que é muito mais rápido do que calculá-lo novamente por sucessivas multiplicações.

`assert` pode também incluir regras na base de dados:

```
?- assert( (otimo(X) :- not (X = zagalo)) ).
```

A regra deve vir dentro de parênteses.

Nos exemplos anteriores, admitimos que os fatos e regras introduzidos na base de dados por `assert` são sempre colocados no fim da base. Se for necessário introduzi-los no início, podemos utilizar `asserta`. Se quisermos explicitar que os fatos ou regras são introduzidos no final da base, podemos utilizar `assertz`.

5.6 Aspectos Não Lógicos de Prolog

Algumas construções de Prolog e o próprio algoritmo de unificação fazem com que esta linguagem não seja completamente lógica. Em lógica, uma expressão "`A and B`" é idêntica a "`B and A`" e "`A or B`" é idêntica a "`B or A`". Em Prolog, isto não é sempre verdadeiro. O "`and`" aparece em regras como

```
R(X) :- A(X), B(X)
```

em que `R(X)` será verdadeiro se `A(X)` *e* `B(X)` forem verdadeiros. Em Prolog, a inversão de `A` e `B` na regra, resultando em

```
R(X) :- B(X), A(X).
```

pode produzir resultados diferentes da regra anterior, violando a lógica.

O "`or`" aparece quando há mais de uma regra para um mesmo predicado ou quando usamos "`;`":

```
R(X) :- A(X) ; B(X).

S(X) :- A(X).
S(X) :- B(X).
```

`R(X)` (ou `S(X)`) será verdadeiro se `A(X)` *ou* `B(X)` o forem. Novamente, os dois predicados acima podem apresentar resultados diferentes se reescritos como

```
R(X) :- B(X) ; A(X).

S(X) :- B(X).
S(X) :- A(X).
```

Em lógica matemática, dada uma expressão qualquer como "`A and B`" e os valores das variáveis, podemos calcular o resultado da expressão. E dado o valor da expressão e de todas as variáveis, exceto uma delas, podemos calcular o valor ou valores desta variável. Assim, se "`A and B`" for falso e `A` for verdadeiro, saberemos que `B` é falso. Em Prolog esta regra nem sempre é verdadeira. Quando não for,

diremos que não há bidirecionalidade entre os argumentos de entrada e saída. Em uma linguagem lógica pura, qualquer argumento pode ser de entrada ou de saída.

Idealmente, um programador de Prolog deveria se preocupar apenas em especificar as relações lógicas entre os parâmetros de cada predicado. O programador não deveria pensar em como o algoritmo de unificação trabalha para satisfazer as relações lógicas especificadas pelo programa. Desta forma, o programador estaria utilizando relações lógicas estáticas, bastante abstratas, ao invés de pensar em relações dinâmicas que são difíceis de entender. Contudo, para tornar Prolog eficiente várias construções não lógicas, citadas a seguir, são suportadas pela linguagem.

- O `is` força uma expressão a ser avaliada quebrando a simetria exigida pela lógica. Isto é, um *goal*

  ```
  6 is 2*X
  ```

  não será válido se `X` não estiver instanciado. Por este motivo, a ordem dos *goals* no corpo de um predicado é importante. O predicado `length` (tamanho de uma lista) não pode ser implementado como

  ```
  length([], 0).
  length([_|L], N) :-
    N is N1 + 1,
    length(L, N1).
  ```

  pois `N` não estaria instanciado no *goal* "`N1 is N + 1`" em uma pergunta

  ```
  ?- length([1, 2], X).
  ```

- O cut também força os *goals* a uma determinada ordem dentro de um predicado. Mudando-se a ordem, muda-se o significado do predicado. A ordem em que as cláusulas são colocadas podem se tornar importantes por causa do cut. Assim, o predicado

  ```
  max(X, Y, X) :-
    X >= Y,
    !.
  max(X, Y, Y).
  ```

  não poderia ser escrito como

  ```
  max(X, Y, Y).
  max(X, Y, X) :-
    X >= Y,
    !.
  ```

  O cut remove a bidirecionalidade da entrada e saída como no caso do predicado `pertence`. Se este for definido como

  ```
  pertence(X, [X | _]) :-
    !.
  pertence(X, [_ | C]) :-
    pertence(X, C).
  ```

o *goal*

```
?- pertence(X, [1, 2, 3]).
```

não serve para obter, por meio do `X`, todos os elementos da lista. Isto é o mesmo que dizer que, dado que `A` é verdadeiro e o resultado de `A and B` é falso, o sistema não consegue dizer o valor de `B`.

- A ordem com que Prolog faz a unificação altera o significado dos predicados. Por exemplo, suponha que o predicado `ancestral` fosse definido como

```
ancestral(A, D) :-       /* A é ancestral de D */
  ancestral(Y, D),
  paimaeDe(A, Y).
ancestral(A, D) :-
  paimaeDe(A, D).
```

Agora o *goal*

```
?- ancestral(jose, pedro).
```

faz o sistema entrar em um laço infinito, apesar do *goal* ser verdadeiro.

- O `not` pode ser definido como

```
not(P) :-
  P, !, fail.
not(P).
```

Para satisfazer um *goal* `not(P)`, Prolog tenta provar que `P` é verdadeiro. Se for, `not(P)` falha. Esta forma de avaliação pode fazer a ordem dos *goals* de um predicado importante. Um exemplo, tomado de [18], é:

```
r(a).
q(b).
p(X) :- not r(X).

?- q(X), p(X).
X = b
```

Mas, invertendo o *goal*,

```
?- p(X), q(X).
no
```

o resultado é diferente.

- As rotinas `assert` e `retract` de manipulação da base de dados podem conduzir aos mesmos problemas que o cut e o `not`. Por exemplo,

```
chuva :- assert(molhado).
?- molhado, chuva.
no
```

```
?- chuva, molhado.
yes

?- molhado, chuva.
yes
```

- relações lógicas não podem representar entrada e saída de dados, que possuem problemas semelhantes a `assert` e `retract`. Por exemplo, um predicado que lê um arquivo pode retornar em um dos seus parâmetros valores diferentes em diferentes chamadas. Então o conceito de estado, estranho à lógica, é introduzido na linguagem.

5.7 Discussão Sobre Prolog

Um programa em Prolog é formado pela base de dados (BD) (cláusulas) e pelo algoritmo de unificação (AU). A BD contém as relações lógicas entre objetos e o AU é o meio de descobrir se dado *goal* é verdadeiro de acordo com a BD. Assim,

programa = BD + AU

O programa em Prolog possui instruções que estão implícitas no algoritmo de unificação e, portanto, não precisam ser colocadas na base de dados. Por exemplo, usando o BD

```
dias(jan, 31).
dias(fev, 28).
...
dias(dez, 31).
```

podemos saber o número de dias do mês de Setembro sem precisar escrever nenhum algoritmo:

```
?- dias(set, N).
```

a busca por `N` é feita pelo AU. No caso geral, parte das repetições (laços, incluindo recursão) e testes estão na BD e parte no AU. No caso acima, a BD não contribui com nenhuma repetição ou teste.

Um outro exemplo é um predicado para verificar se elemento `X` pertence a uma lista:

```
membro(X, [X | L]).
membro(X, [Y | L]) :-
   membro(X, L).
```

A função equivalente utilizando a sintaxe de S é mostrada na Figura 5.1.

A função em S possui muito mais detalhes que a de Prolog. Contudo, o predicado em Prolog possui operações equivalentes àquelas da função em S. Algumas destas operações estão implícitas no algoritmo de unificação e outras explícitas no programa. No predicado `membro`, as operações (`p == nil`) e (`p.elem == x`) estão implícitas no AU, pois estes testes são feitos no emparelhamento com as cláusulas de `membro`. A operação `p == nil` é equivalente a não obter emparelhamento, já que a lista é vazia, e que resulta em falha do predicado. E `p.elem == x` é equivalente a obter emparelhamento com a primeira cláusula, `membro(X, [X|L])`.

A obtenção do elemento da frente da lista é feito com `.` em S e pela convenção de separar uma lista em cabeça e cauda (`[H | T]`) em Prolog. No predicado `membro`, a única operação realmente explícita é a recursão na segunda cláusula que aparece disfarçada de uma definição de cláusula.

A função e o predicado `membro` demonstram que nem todas as operações precisam estar explicitadas nas cláusulas (BD), pois as operações contidas no AU fazem parte do programa final.

A linguagem Prolog é adequada justamente para aquele problemas cujas soluções algoritmicas possuem semelhança com o algoritmo de unificação. Sendo semelhantes, a maior parte da solução

```
proc membro( p, x )
begin
  if p == nil
  then
    return false;
  else
    if p.elem == x
    then
      return true;
    else
      return membro(p.suc, x);
    endif
  endif
end;
```

Figura 5.1: Função para descobrir se `x` é membro da lista `p`

pode ser deixada para o AU, tornando a BD muito mais fácil de se fazer (mais abstrata). A parte não semelhante ao AU deve ser codificada nas regras, como a chamada recursiva a `membro` na segunda cláusula do exemplo anterior.

De um modo geral, uma linguagem é boa para resolver determinados problemas se estes são facilmente mapeados nela. Neste caso, a linguagem possui, implicitamente, os algoritmos e estruturas de dados mais comumente usados para resolver estes problemas.

Capítulo 6

Linguagens Baseadas em Fluxo de Dados

Linguagens baseadas em fluxo de dados (*data flow*) obtém concorrência executando qualquer instrução tão logo os dados que ela utiliza estejam disponíveis. Assim, uma chamada de função `f(x)` (ou uma atribuição "`y = x`") será executada tão logo a variável `x` tenha recebido um valor em alguma outra parte do programa — então variáveis podem estar em um de dois estados: inicializadas ou não. Não interessa onde `f(x)` (ou `y = x`) está no programa ou se as instruções que o precedem textualmente no código fonte já tenham sido executadas: `f(x)` (ou `y = x`) será executada tão logo a variável `x` tenha um valor. Então a execução do programa obedece às dependências entre os dados e não a ordem textual das instruções no código fonte.

A execução de um programa em uma linguagem *data flow* utiliza um grafo de fluxo de dados (GFD) no qual os vértices representam instruções e as arestas as dependências entre elas. Haverá uma aresta `(v, w)` no grafo se `w` depender dos dados produzidos em `v`. Como exemplo, considere as atribuições do procedimento

```
proc m(b, d, e)
    { declara e ja inicializa as variaveis }
  var
    a = b + 1,     { 1 }
    c = d/2 + e,   { 2 }
    i = a + 1,     { 3 }
    f = 2*i + c,   { 4 }
    h = b + c,     { 5 }
    k = f + c;     { 6 }
  is
    return a + c + i + f + h + k;
```

Então há uma aresta (representada por uma seta) de `1` para `3`. O GFD das instruções acima está na Figura 6.

As inicializações do procedimento `m` podem ser executadas em várias ordens possíveis:

```
1 2 3 4 5 6
2 5 1 3 4 6
2 1 3 4 6 5
...
```

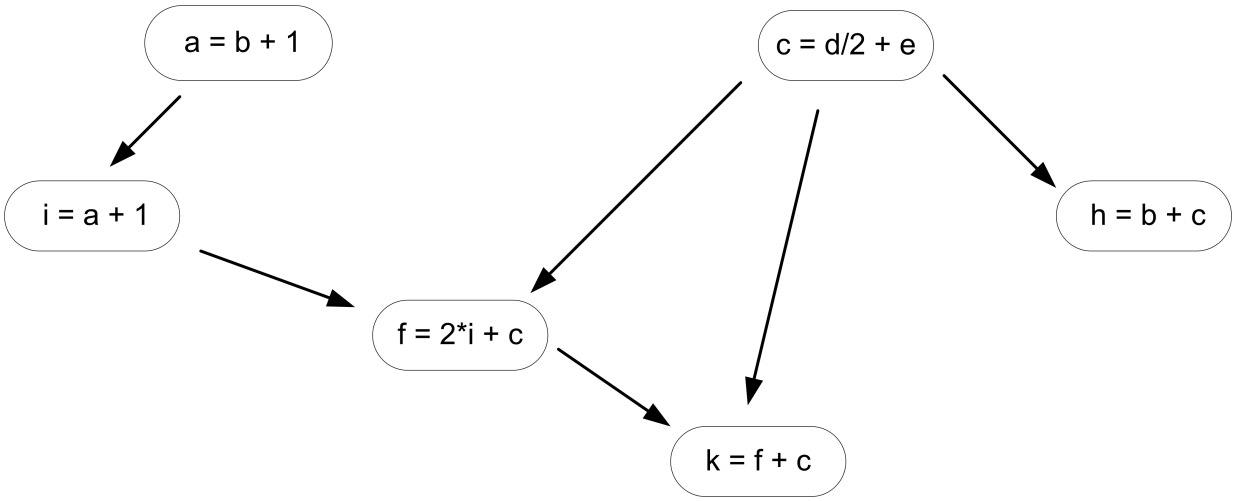

Figura 6.1: Um grafo do fluxo de dados

Se não há caminho ligando a instrução `V` a `U`, então estas instruções são independentes entre si e podem ser executadas concorrentemente. Por exemplo, podem ser executadas em paralelo as instruções

```
1 e 2
3 e 5
4 e 5
5 e 6
...
```

Um programa em uma linguagem *data flow* (LDF) é transladado em um GDF que é executado por uma máquina *data flow*.[1] Esta máquina tenta executar tantas intruções em paralelo quanto é possível, respeitando as dependências entre elas. Conseqüentemente, a ordem de execução não é necessariamente a ordem textual das instruções no programa. Por exemplo, a instrução `5` poderia ser executada antes da `3` ou da `4`.

Podemos imaginar a execução de um programa *data flow* como valores fluindo entre os nós do GDF. A instrução de um dado nó poderá ser executada se houver valores disponíveis nos nós de que ela depende. Um nó representando uma variável possui valor disponível após ela ser usada do lado esquerdo de uma atribuição:

```
a = exp
```

Por exemplo, a instrução `4` só poderá ser executada se há valores em `2` e `3` (valores de `c` e `i`).

Uma conseqüência das regras da dependência é que cada variável deve ser inicializada uma única vez. Caso contrário, haveria não determinismo nos programas. Por exemplo, em

```
proc p() : integer
  var
    a = 1,   { 1 }
    b = 2*a, { 2 }
    a = 5;   { 3 }
  is
    a + b;
```

[1] Obviamente, qualquer computador poderia executar este programa, mas supõe-se que um computador *data flow* seria mais eficiente.

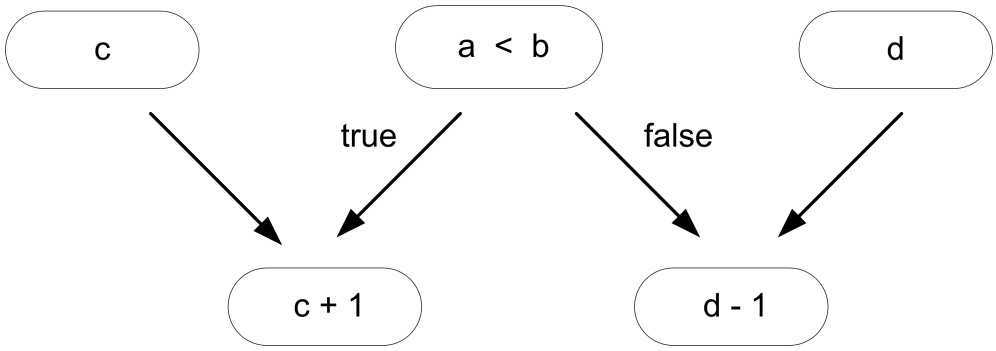

Figura 6.2: Um grafo do fluxo de dados de um comando `if`

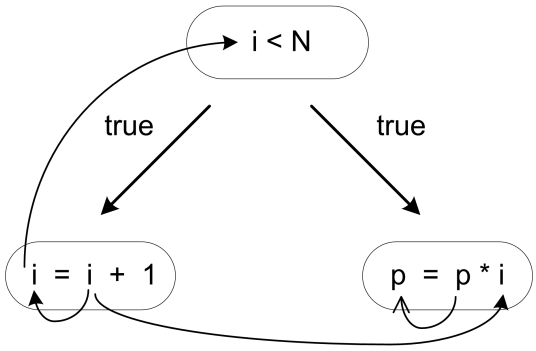

Figura 6.3: Um grafo do fluxo de dados de um comando `while`

o valor final de `b` poderia ser `2` ou `10`, pois a seqüência de execução das atribuições poderia ser

```
1  2  3
```

ou

```
3  2  1
```

entre outras. A exigência de uma única inicialização é chamada de regra da atribuição única.

Um comando `if` em uma linguagem *data flow* típica é da forma

```
if exp then exp1 else exp2
```

como em linguagens funcionais. A expressão `exp1` só será avaliada se a expressão `exp` do `if` for `true`. Como a ordem de execução só depende da disponibilidade de dados, a expressão `exp1` é feita dependente de `exp` da seguinte forma: se `exp` resultar em `true`, `exp1` recebe um *token* que habilita a sua avaliação. Sem este *token*, `exp1` não será avaliada mesmo que todos os outros valores de que ela depende estejam disponíveis. O grafo de fluxo de dados do `if`

```
if a > b
then
  c + 1
else
  d - 1
```

está na Figura 6.

Comandos `while` funcionam de forma semelhante aos `if`'s. Os comandos do corpo do `while` são dependentes de um valor `true` resultante da avaliação da expressão condicional. O GFD do `while` do código

```
i = 1;
```

```
p = 1;
while i <= N do
  begin
  p = p*i;
  i = i + 1;
  end
```

está na Figura 6.

Este código permite a modificação de variáveis de controle dentro do laço, violando a regra de atribuição única. Há duas atribuições para `i` e duas para `p`. Este problema é contornado permitindo a criação de uma nova variável `i` (e `p`) a cada iteração do laço. Assim, temos um *stream* de valores para `i` e outro para `p`. As máquinas *data flow* associam *tags* aos valores de `i` e `p` de tal forma que os valores de um passo do laço não são confundidos com valores de outros passos.

Considere que a multiplicação `p*i` seja muito mais lenta que a atribuição `i = i + 1` e o teste `i <= N`, de tal forma que o laço avança rapidamente na instrução `i = i + 1` e lentamente em `p = p*i`. Isto é, poderíamos ter a situação em que `i = 12` (considerando `N = 15`) mas `p` ainda está sendo multiplicado por `i = 3`. Haveria diversos valores de `i` esperando para serem multiplicados por `p`. Estes valores não se confundem. O valor de `p` da k-ésima iteração é sempre multiplicado pelo valor de `i` da k-ésima iteração para resultar no valor de `p` da (k + 1)-ésima iteração.

Em uma chamada de função, alguns parâmetros podem ser passados antes dos outros e podem causar a execução de instruções dentro da função.

Considere a função

```
proc p(x, y : integer) : integer
  var
    z = x + 3,      { 1 }
    t = x + 1;      { 2 }
  is
    z + t + 2 * y;   { 3 }
```

e a chamada de função `p`

```
proc q()
  var
    a = 1,
    b = f(2),
    k = p(a, b);
  is
    ...
```

onde `a` já recebeu um valor, mas o valor de `b` ainda está sendo calculado. Admita que o cálculo de `f(2)` é demorado. O parâmetro `a` é passado a `p` e causa a execução das instruções `1` e `2`. A instrução `3` é executada tão logo o valor de `b` esteja disponível e seja passado a `p`.

Linguagens *data flow* utilizam granuralidade muito fina de paralelismo gerando uma quantidade enorme de tarefas[2] executadas paralelamente. Os recursos necessários para gerenciar este paralelismo são gigantescos e exigem necessariamente uma máquina *data flow*. Se o código for compilado para uma máquina não *data flow*, mesmo com vários processadores, haverá uma enorme perda de desempenho. Uma linguagem contendo apenas os conceitos expostos neste capítulo deixa de aproveitar as maiores

[2]Tarefas referem-se a trechos de código e não a processos do Sistema Operacional.

oportunidades de paralelismo encontradas em programas reais, que são: a manipulação de vetores inteiros de uma vez pela máquina sobre os quais podem ser aplicadas as operações aritméticas. Uma granuralidade mais alta de paralelismo é também interessante pois a quantidade de comunicação entre os processos em paralelo é minimizada. Contudo a linguagem que definimos não permite a definição de processos com execução interna seqüencial mas executando em paralelo com outros processos.

Referências Bibliográficas

[1] America, Pierre; Linden, Frank van der. A Parallel Object-Oriented Language with Inheritance and Subtyping, *SIGPLAN Notices*, Vol. 25, No. 10, October 1990. ECOOP/OOPSLA 90.

[2] Lippman, Stanley B. *C++ Primer*. Addison-Wesley, 1991.

[3] Deitel, H.M. e Deitel P.J. *C++ How to Program*. Prentice-Hall, 1994.

[4] GC FAQ — draft. Available at http://www.centerline.com/people/chase/GC/GC-faq.html

[5] Goldberg, Adele; Robson, David. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, 1983.

[6] Hoare, Charles A. R. The Emperor's Old Clothes. *CACM*, Vol. 24, No. 2, February 1981.

[7] Guimarães, José de Oliveira. *The Green Language home page*. Available at http://www.dc.ufscar.br/~jose/green.

[8] Guimarães, José de Oliveira. The Green Language. Computer Languages, Systems & Structures, Vol. 32, Issue 4, pages 203-215, December 2006.

[9] Guimarães, José de Oliveira. The Object Oriented Model and Its Advantages. *OOPS Messenger*, Vol. 6, No. 1, January 1995.

[10] Kjell, Bradley. Introduction to Computer Science using Java. Available at http://chortle.ccsu.edu/CS151/cs151java.html.

[11] Niemeyer, P. and Peck, J. (1997) *Exploring Java*. O'Reilly & Associates, Sebastopol.

[12] Rojas, Raúl, et al. (2000). Plankalkül: The First High-Level Programming Language and its Implementation. Institut für Informatik, Freie Universität Berlin, Technical Report B-3/2000. Available at http://www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Plankalkuel-Report/Plankalkuel-Report.htm.

[13] Slater, Robert. Portraits in Silicon. MIT Press, 1989.

[14] Stroustrup, Bjarne. *The C++ Programming Language*. Second Edition, Addison-Wesley, 1991.

[15] Wegner, Peter. *Research Directions in Object-Oriented Programming*, chapter The Object-Oriented Classification Paradigm, pp. 479–559. MIT Press, 1987.

[16] Weinberg, Gerald M. *The Psychology of Computer Programming*, Van Nostrand Reinhold, 1971.

[17] Ben-Ari, M. *Understanding Programming Languages*. John Wiley & Sons, 1996.

[18] Bratko, Ivan. *Prolog Programming for Artificial Intelligence*. International Computer Science Series, 1986.

[19] Finkel, Raphael. *Advanced Programming Language Design*. Addison-Wesley, 1996.