Programming Languages Paradigms

José de Oliveira Guimarães
UFSCar at Sorocaba
Sorocaba, SP
Brazil
e-mail: josedeoliveiraguimaraes@gmail.com, jose@ufscar.br

28 de maio de 2015

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 intended to be used by the regular 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 [16]. The language was called plankalkül and only recently it has been implemented [15].

Fortran was the first high level programming language to be implemented. Its design began in 1953 and a compiler was released in 1957 [16]. 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 1958 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 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, Ada, Java, Ruby, Groovy, Scala, C++, and C#. 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 [19]. 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 makes 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 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, exceptions, and reflection. 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.

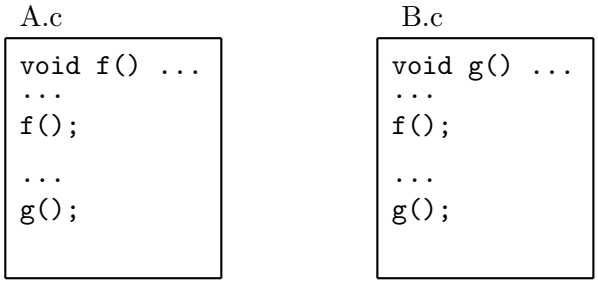


Figura 1.1: Two files of a C program

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 — Optional

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

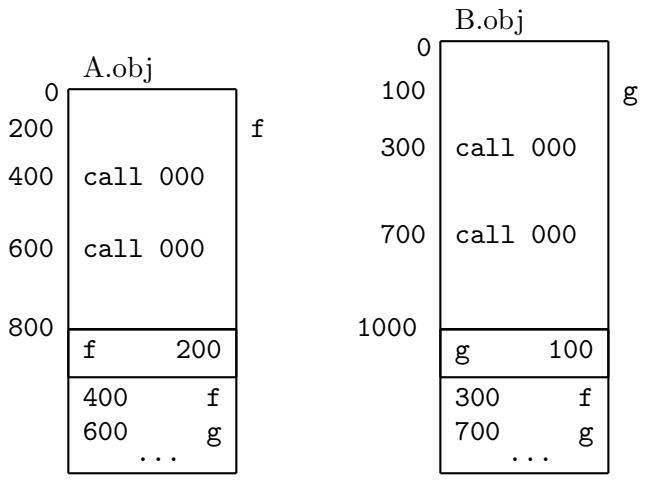


Figura 1.2: Object file configurations

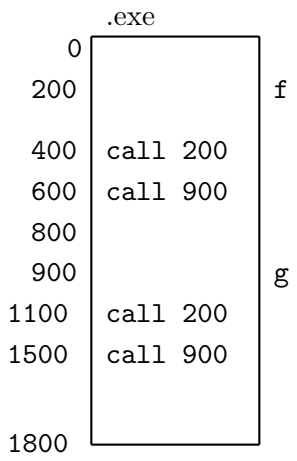


Figura 1.3: Executable file

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 needs 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 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. A compiler can also generate instructions for a virtual machine. These instructions are called "bytecodes" and 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 check 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 may not need a linker. The linking can be made dynamically during the interpretation.[5] 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. Some compiled languages demand that all source files are compiled and then linked before execution. Some files need to be re-compiled because of small changes in others. For example, suppose that a language-C header file "`MyConst.h`" declares

   ```
   const Max = 100;
   ```

   If the value of `Max` is changed, all source files that import `MyConst.h` should be re-compiled. This is usually not necessary in an interpreted language since all linking is usually made at runtime.

   In many dynamically-typed interpreted languages, any changes in any file do not demand the re-compilation of any other files. All the linking between the constructs is made at runtime, including that of classes to their subclasses, types to their variables, and so on. We say that the sequence Edit-Compile to bytecodes-Interpreter is faster than the sequence Edit-Compile-Link-Execute required by compiled languages. Interpreters are usually faster 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.

6. However, some interpreters, such the Java one, are highly sophisticated. They translate the bytecodes to machine instructions at runtime, thus increasing the running speed of the compiled program. On the other side, these interpreters are not easy to build as the regular ones.

[5] Note the words "*may not need*" and "*can be made*". This is not always in this way.

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

> Any effectively calculated 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 a definition of what can be computed by mechanical means. However, there has never been found any algorithm that cannot be implemented by a Turing machine or any programming language. An easy way to prove that a language L is a programming language is to devise an interpreter of a Turing machine in L. If this is possible, L is a 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 halt.

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.

To simplify the definition of a "Programming Language" we can say that a programming language is a language in which an interpreter of a Turing machine can be made. Of course, this definition entails that the code of the language can be executed and the result obtained would be the same as the Turing machine itself.

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 Java-C-like syntax. A method or function called `println` that can take a variable number of parameters will be used for output (basically, printing its parameters in the monitor).

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 -2,147,483,648, ... 0, 1, ... 2,147,483,647 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 language is statically-typed if the compiler knows the type of each expression in the program and checks whether the operations applied to each of them is supported by the corresponding type. Then the compiler knows the type of each variable, parameter, and return value of functions, procedures, or methods. This type may be explicity given by the code or it may be deduced by the compiler. For example, in Java one should declare an `int` variable as

```
int n;
```

In language C, other statically-typed language, a function should also declare the return value type of every function:

```
int f(int n) {
    if ( n <= 0 ) return 1;
    else return n*f(n-1);
}
```

However, there are statically-typed languages in which the compiler can deduce the types of some variables. In Cyan, for example, one can declare `n` as

```
var n = 0;
```

Since the value assigned to `n` is an `Int`, the type of `n` will be `Int`.

One can say that a language supports "static type binding" instead of saying it is "statically-typed".

Dynamically typed languages do not allow types to be specified for variable, parameters, and return value of functions. 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.

In the examples, we will use keyword `var` before variable declarations.

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

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
```

assuming that `write` prints its arguments.

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.

Variables in a dynamically-typed language are in fact language-C-like pointers. When a variable is declared the compiler does not generates code for the memory the variable will point at runtime. It cannot because it does not know the type of the value the variable will refer to.

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

```
var String s;
s = s * 2;
...
```

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

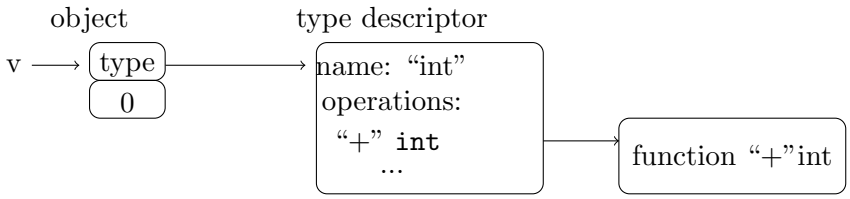
```
a = b + c;
```

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. retrieve the type of `b` and `c`. Each variable is a pointer to a memory area which we will call "object", even if the language is not object-oriented. This object has the expected data: for example, an integer has the integer value, a string has a string. Besides that, the object has a pointer to the object type:



2. test if `b` support operation `+` taking an object of the type of `c` as parameter. That is, if the list of supported operations list "+" with a parameter that has the type of `c`. Int "`b + c`", the parameter to "`+`" is `c`. In the above figure, the type is "`int`" which supports an operation "`+`" with another `int` value;

3. retrieve the function found in the previous item and call it with parameter `c`.

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 functions 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 functions are important because their code can be reused with parameters of more than one type. For example, function `search` given below can be used with arrays of several types as `integer`, `boolean`, `string`, or any other that support the operation "`==`".

```
search(v, n, x) {
    var i;
    i = 0;
    while ( i < n ) {
        if ( v[i] == x )
            return i;
        else
            i = i + 1;
    }
```

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

```
        return -1;
    }
```

`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.

Refactorings are program transformations that change the source code without changing the program behavior. Refactorings are usually supported by tools that automatize the process. As examples of refactorings in object-oriented programming we can cite:

1. add an instance variable to a class;
2. to change the name of an instance variable or a local variable;
3. introduce a new class in the middle of the class hierarchy;
4. rename all methods that have a specific name;
5. move a method down or up in a hierarchy.

Static typing allows more and better refactorings. For example, it is not possible to rename a method of a class in a dynamically-typed language without running the risk of breaking the code. Using the Java syntax, consider the code

```
void print(person) { // no parameter type
    System.out.println( person.getName() );
    ...
}
```

`person` is supposed to refer to an object of an `Person` class at runtime. If a refactoring changes the name of method `getName` to `getId`, the refactoring tool will not change "`person.getName()`" to "`person.getId()`". It cannot know `person` will refer to an objet of `Person` all the time. Either all `getName` methods and calls are changed to `getId` or there will possibly be a runtime error.

So there are many refactorings that cannot be applied to dynamically-typed languages. However, refactorings were created in Smalltype, a typeless language [12] [14].

Statically-typed languages allow for code completion. When the user types "`Person p`" and the types "`p.`" and waits for a second, the IDE can show a list of methods of `Person` and its superclasses. That is difficult to do in dynamically-typed languages.

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

```
int i;
i = "hello";
```

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

Some Algol-like languages (like Pascal, Modula, and Ada) support the declaration of procedures inside other procedures as shown in the example below in which `Q` is inside `P`.

```
int max;            // 1
void P(int n) {
    int i;          // 2
    void Q() {      // 3
        int k;      // 4
        ...
    }               // 5
    ...
}                   // 6
...                 // 7 end of file
```

Block structure was devised to restrict the scope of procedures. The *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` 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`.

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.

```
void P() {
    int i;
    for (i = 0; i < n; ++i)
        println(i);
}

void Q() {
    int n;
    if ( max < 10 )
        n = max;
    else
        n = 10;
    P();
}

void main() {
    // main function. Program execution starts here
    int max;
    max = 5;
    Q();
    P();  // 1
}
```

An example of it is shown in example above. 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 2.1 (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 2.1 (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? One use is to change the program behavior by declaring locally a variable with the same name as a global variable. Suppose all the program output is made using a variable call "`output`" in a software made using an object-oriented language. If a method `change` declares a variable with this same name and makes it refer to an object of a class `MyOutput`, now all output is made thought `MyOutput` till the method

(a) (b)

Figura 2.1: Call stack

returns. All methods called while the method `change` is in the stack (it has not finished its execution) will use variable "`output`" for output.

Besides that, dynamic scope is easy to implement: variables are only checked at run time using an already existing stack of local variables.

```
int max;

void A() {
    int min;

    char B() {
       int g;
       void C() {
           boolean ok;
           // C
           ...
       }
       // B
       ...
    }
    void D() {
        int j;
        // D
        ...
    }
    // A
    ...
}
```

Now we return to block structures. The tree corresponding to the above example is

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 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;

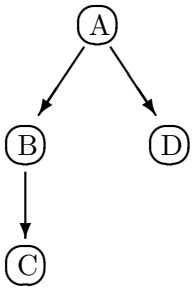


Figura 2.2: Tree representing nested scope

- 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 the Figure 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 2.2 but difficult to identify by reading the source code.

2.3 Packages

A module or package[2] is a set of resources such as procedures, types, constants, classes[3], and variables. Each resource of a package can be public, private, or something else (as protected). Public resources can be used by other packages that import this one through a special command discussed next. Private resources are only visible inside the module itself. Some languages allow *protected* resources that can be seen only in the package in which they are declared. In Java, a class or interface declared without a modifier can only be used inside its own package.

In the example below we will use the Java syntax. In Java a package is a colletion of source files. Each source file can have one or more classes (yet to be seen). But only one of these should be public. The next example shows a source file of package `bankSystem`. This source should have name `Bank.java` because the public class is "`Bank`".

```
package bankSystem;

public class Bank {
   ...
}

private class BankData {
   ...
```

[2]Its Ada and Java names.

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

```
}
```

A package `p` is imported using "`import p;`" as in

```
package company;

import bankSystem;

public class SmallCompany {
   private Bank bank;
   ...
}
```

Every public resource defined in package `bankSystem`, in all source files of it, can be used inside class `SmallCompany`. Then this class can declare a variable whose type is `Bank`. It is also possible to remove the `import` declaration and use the full path of class `Bank`:

```
package company;

public class SmallCompany {
   private bankSystem.Bank bank;
   ...
}
```

Note that class `BankData` can only be used inside source file `Bank.java` since it is private.

There is a conflict if a resource is defined in two or more imported package. For example, another `Bank` class may be defined in package `accounting`:

```
package company;

import bankSystem, accounting;

public class SmallCompany {
   private Bank bank;
   ...
}
```

Now "`Bank`" is ambiguous in this code. This is considered an error. Most languages demand that, in these cases, the resource be qualified by one of the packages:

```
private bankSystem.Bank bank;
```

or

```
private accounting.Bank bank;
```

Some languages demand this syntax for all imported resources, not only the ambiguous ones. This makes typing in the keyboard 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 "`bankSystem.Bank`" is less legible than a lighter one like "`Bank`".

Packages have several important features, described next.

- When a package `B` imports package `A`, `B` can use only public resources 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.
- A program can (should) be divided in packages that can be separately compiled and understood. One may only know the public section of a package in order to use it. Then package work as abstraction mechanisms reducing the program complexity. A well-designed package restricts the dependency from other packages 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. In a language without exception handling, there should be a test after each statement that can result in error (ideally at least). 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. For example, suppose that Java does not have exception handling constructs. Then a method to write an array to a file would be something like the following code.

```
public String writeFile(String filename, char toWrite[])  {
    FileWriter outFile;
    outFile = new FileWriter(filename);
    if ( outFile == null )
        return "Can´t create file " + filename;
    int size = toWrite.length;
    if ( toWrite[size-1] == '\0' )
        size--;
    if ( outFile.write(toWrite, 0, size) == null )
        return "Error writing file to " + filename;
    if ( outFile.close() == null )
        return "Error closing file " + filename;
    return null;
}
```

There should be tests after every part of the code that can fail. That is tedious. Programmers easily forget to do the checks. The code is populated with a lot of `if`'s to test the success of method calls. 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 exception signalling and handling:

```
public String writeFile(String filename, char toWrite[])  {
 FileWriter outFile;

    try {
        outFile = new FileWriter(filename);
        int size = toWrite.length;
        if ( toWrite[size-1] == '\0' )
            size--;
```

```
            outFile.write(toWrite, 0, size);
            outFile.close();
        } catch (IOException e) {
            return "Fail to create or write to file " + filename;
        }
        return null;
    }
```

Inside methods `FileWriter` (the constructor of class `FileWriter`), `write`, and `close` there are statements like

```
thrown new IOException();
```

that throw exception `IOException`. When one of those statements is executed, the runtime system starts looking for a `catch` clause that accepts an `IOException` as parameter. One is found in method `writeFile` above and control is transferred to it. That is, there is a `goto` from the `throw` statement to the `catch` clause. This can be better understood using another example:

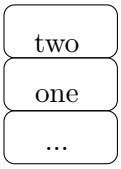
```
void one() {
   int n = readInt();
   try {
       two(n);   // point 1
       System.out.println("After calling two");
   } catch( NegException e ) {
       System.out.println( e.get() + " is negative" );
   }
   catch( ZeroException e ) {
       System.out.println( "n is zero" );
   }
   System.out.println("in the middle");
   try {
       two(n);   // point 2
   }
   catch ( BadLuckNumber e ) {
      System.out.println("What a bad luck!");
    }
}

void two(int n) {
   try {
       if ( n < 0 )
           throw new NegException(n);
       three(n);
   }
   catch( BigNumberException e ) {
       System.out.println( "n is too big" );
    }
}

void three(int n) {
    System.out.println(n);
```

```
        if ( n == 0 )
            throw new ZeroException();
        if ( n > 1000000 )
           throw new BigNumberException(n);
        if ( n == 13 )
           throw new BadLuckNumber();
    }
```

When `one` is called, it read an `int` from the keyboard and calls `two(n)` at point `1`. Supposing that the number is negative, `two` throws exception `NegException`. At this point, the stack of called method has `two` above `one`:



The runtime system starts looking for a `catch` clause in the stack of methods, from top to bottom. First it looks at `two` for a `catch` clause. There is one, but the parameter type, `BigNumberException`, is not supertype of `NegException` — consider that the two types are unrelated.[4] Then the search continues at the next method of the stack, which is `one`. A `catch` clause whose parameter is `NegException` is found. At this point the runtime system finishes the execution of `two` and transfers the control to the `catch` clause. The value given after "`throw`", which is "`new NegException(n)`", is assigned to the `catch` parameter, `e`. After the execution of this clause, execution continues in the next statement. Then "`in the middle`" is printed. But "`After calling two`" is not.

Note that only the `catch` clauses in the stack are used in the search. Then the clause in point `2` is not used in the search because `two` was called at point `1`.

An exception in Java is a value of a class that inherits from class `Throwable` (the details will be omitted). As such, it may have associated data as the number `n` in the above example.

An exception can be thrown and not caught. When this occur the program is terminated. 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 packages of a program weakening their reusability. A package should be aware of internal details of other packages it uses because it needs to know which exceptions they can raise. That means a package should know something about the internal flow of execution of other packages 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, C#, Ada, Eiffel, and C++. Exceptions have two main features:

[4] If a type `A` is supertype of a type `B` then a value of `B` can be used whenever a value of `A` is expected.

- 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.

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

Java support safe exceptions: all exception classes that inherit from class `Exception`, a subclass of `Throwable`, should either be caught by a catch clause or declared in the method header. If all exception classes of the previous example were subclasses of `Exception`, we could write

```
void fouth() throws BadLuckNumber, BigNumberException {

    int n = readInt();
    try {
        three(n);
    }
    catch ( ZeroException e ) {
        System.out.println("zero");
    }
}
```

`three` may throw three exceptions. One of them is caught by `fouth`. The other two are declared. The compiler would sign an error if some of these three exceptions were not caught or not declared.

The exceptions a method can throw are those

- the other methods it calls can throw;
- it can throw itself;
- that are not handled by any `catch` clause in its body.

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++, Groovy, and C# 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.

```
void main() {
    // do nothing --- just an example

    Integer p, t;
    p = new Integer(0);  // 1
    t = p;               // 2
```

```
        p = null;            // 3
        t = null;            // 4
    }
```

Variables `p` and `t` are in fact pointers to objects of type "`Integer`". Memory for `Integer` objects is not allocated by the declaration of these variables. Instead, expression "`new Integer(0)`"creates a new `Integer` object at runtime. 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 package may free a memory block still in use by other packages. 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,[5] 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.

[5]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.

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

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

- Different deallocation strategies may be used by the packages of a program or the libraries used by it [4]. For example, the operation `deleteAll` of a Queue data structure[6] 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;
- 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.

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

2.6 Exercices

1. O que é erro de tipos?

2. Que características possui uma linguagem fortemente tipada? E uma estaticamente tipada?

3. Dê um exemplo de um programa que utilize procedimentos encaixados. O programa deve funcionar, mesmo que não faça nada de útil.

4. Um compilador poderia deduzir o tipo dos parâmetros e do retorno da função abaixo? Explique como, assumindo que não se pode fazer operações com tipos distintos. Isto é, `x + y` só é válido se `x` e `y` têm o mesmo tipo.

```
f(n) {
    if ( n <= 0 )
        return 1;
    else
        return n*f(n-1);
}
```

5. Em uma linguagem estaticamente tipoada o compilador sabe o tipo de todas as variáveis. Mas sabe também o tipo de todas as expressões?

6. Algumas linguagens permite a mistura de código tipado dinamicamente com tipado estaticamente. Suponha que uma variável tipada dinamicamente tenha o "tipo" `dynamic`, que é na verdade uma palavra-chave. Cite as expressões abaixo em que o compilador faria a conferência de tipos e aquelas em que esta conferência seria feita em execução.

```
int x = 1, y = 5;
dynamic z;

x = y*x;      z = y + 1;
y = 3*z;      println( fat(z) );
```

Assuma que a assinatura de `fat` seja

```
int fat(int n)
```

7. Assembler tem tipos? Em assembler pode-se manipular uma região de memória contendo um inteiro como se fosse um número em ponto flutuante?

8. Porque variáveis em linguagens dinamicamente tipadas têm que ser ponteiros? Porque o compilador não pode alocar memória para os objetos que elas apontam na declaração delas?

9. Conceitualmente, quantos "objetos" são criados pela execução do código abaixo em uma linguagem dinamicamente tipada?

```
int i = 0;
```

```
while ( i < 10 ) {
    println(i);
    i = i + 1;
}
```

10. Cite as ações, incluindo as conferências, que são feitas em tempo de execução por causa da seguinte instrução, feita em uma linguagem dinamicamente tipada orientada a objetos.

```
list.add(elem);
```

11. Desenhe a representação de um inteiro em uma linguagem dinamicamente tipada.

12. Faça uma função que encontra o maior elemento de um vetor em uma linguagem dinamicamente tipada.

13. Código feito em linguagens dinamicamente tipadas é provavelmente menos legível do que aquele feito em linguagens com tipo. Explique esta frase, obrigatoriamente com um exemplo.

14. Porque uma IDE pode oferecer mais suporte às linguagens com tipagem estática?

15. Imagine uma função de 30 linhas de código em uma linguagem dinamicamente tipada. Os nomes dos parâmetros, variáveis e da função não são significativos. Não há comentários. Você consegue entendê-lo? Isto é, você conseguirá montar mentalmente a execução desta função com as alterações que ela fará nos dados? E se esta função estivesse em uma linguagem estaticamente tipada?

16. Explique porque refactorings são mais difíceis de fazer em linguagens dinamicamente tipadas.

17. Qual tipo de linguagem exige mais comentários no código: as estaticamente tipadas ou dinamicamente tipadas?

18. Dados os conceitos "escopo", "visibilidade", "tempo de vida", quais são estáticos? Uma variável é sempre visível no seu escopo? A região onde a variável existe é igual ao seu escopo?

19. Cite as desvantagens de estruturas de blocos.

20. Faça um pequeno programa com duas funções com escopo dinâmico de tal forma que :

- não haja erro de compilação nem de execução;
- se a linguagem suportasse apenas escopo estático, haveria um erro em tempo de compilação.

21. Admita que o programa abaixo seja de uma linguagem com escopo dinâmico. O que ele escreve? Naturalmente, a execução do programa começa no procedimento `main`.

```
void P() {
    int i, n;
    n = k + r;
```

```
    for (i = 0; i < n; ++i)
        max = max + i;
}

void Q(int max) {
    int k;
    k = 1;
    P();
    writelne(max);
    k = 3;
    P();
    println(max);
    println(n);
}

void main() {
    int n, r;
    n = 5;
    r = 1;
    Q(n);
}
```

22. Uma linguagem com escopo dinâmico utiliza a variável `output`, declarada na função principal, onde se inicia o programa, para impressão na saída padrão (usualmente o monitor do computador). Como pode uma função redirecionar toda a saída das funções que ela chama para algum outro lugar?

23. O que escreve o programa abaixo? O que acontece se a constante `max` for inicializada com `1`?

```
// variaveis globais
var a, b;

void P( i ) {
    if ( i > 3 ) {
        a = 12;
        b = "abcdefg";
    }
    else {
        a = "abcdefg";
        b = 33;
    }
}

void main() {
    var s, j;
    max = 5;
    s = 1;
    for (j = 0; j < max; ++j)
```

```
            s = s + j;
        P(s);
        a = a + 1;
        println(a);
    }
```

24. Explique como pacotes funcionam. A sua resposta deve explicar o que é parte pública e privada e o que um pacote pode utilizar de outro pacote que ele importa.

25. Cite duas vantagens de se utilizar pacotes.

26. Faça um pequeno exemplo com duas funções de tal forma que uma exceção levantada em um deles seja tratada no outro. Utilizando este exemplo, explique como exceções funcionam.

27. Faça um exemplo onde uma exceção é levantada e não tratada.

28. Cite duas desvantagens de exceções.

29. Cite quatro problemas com a desalocação explícita de memória pelo programador (`delete` de C++, `dispose` de Pascal).

30. Cite dois problemas com coleta de lixo (mesmo se os coletores atuais já resolveram estes problemas, pelo menos parcialmente).

31. Cite todas as disvantagens de linguagens dinamicamente tipadas em relação às estaticamente tipadas.

32. Cite todas as disvantagens de linguagens estaticamente tipadas em relação às dinamicamente tipadas.

33. Dê uma explicação de alto nível de como seria implementada a instrução "`a = b + 1`" em uma linguagem dinamicamente tipada. Baseado nesta resposta, explique porquê código neste tipo de linguagem tem execução mais lenta do que nas linguagens estaticamente tipadas.

Capítulo 3

Linguagens Orientadas a Objeto

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 neste exemplo:

```
class Store {
    public int get() {
        return n;
    }
    public void put(int pn) {
        n = pn;
    }
    private int n;
}
```

É declarada 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. Esta é a terminologia de Smalltalk, embora neste capítulo utilizemos a sintaxe de Java.

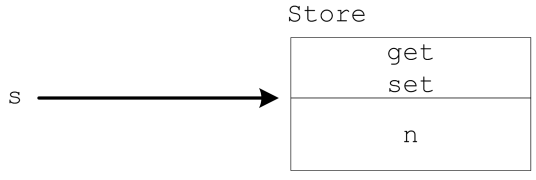
Uma variável da classe `Store`, declarada como

```
Store s;
```

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

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

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:



[1] Pelo menos por enquanto.

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 públicos são visíveis. É então ilegal fazer

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

já que `n` é privado à classe. Métodos e variáveis públicos são prefixados pela palava-chave `public`. Idem para métodos e variáveis privadas.[2]

Alocando dois objetos, como em

```
void main() {
    Store s, t;
    s = new Store();
    t = new Store();
    s.put(5);
    t.put(12);
    System.out.println(s.get() + " " + t.get() );
}
```

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).

3.1 Proteção de Informação

Em algumas linguagens, as variáveis de instância só são manipuladas por meio dos métodos da classe — todas são privadas. Dizemos que estas linguagens suportam *proteção de informação*.

Para exemplificar este conceito, usaremos a classe Pilha:

```
class Pilha {
    private static final Max = 100;
    private int topo;
    private int []vet;

    public void crie() {
        topo = -1;
        vet = new int[Max];
    }
    public boolean empilhe(int elem) {
        if ( topo >= Max - 1) return false;
        else {
```

[2] Java admite variáveis de instância privadas. Contudo, este fato não será utilizado neste livro.

```
            ++topo;
            vet[topo] = elem;
            return true;
        }
    }
    public int desempilhe() {
        if ( topo < 0 ) return -1;
        else {
            elem = vet[topo];
            topo = topo - 1;
            return elem;
        }
    }
    public boolean vazia() {
        return topo < 0;
    }
}
```

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.

```
void main() {
    Pilha p, q;

    p = new Pilha();
    p.crie();   // despreza o valor de retorno
    p.empilhe(1);
    p.empilhe(2);
    p.empilhe(3);
    while ( ! p.vazia() )
        System.out.println( p.desempilhe() );
    q = new Pilha();
    q.crie();
    q.empilhe(10);
    if ( ! p.empilhe(20) )
        erro();
}
```

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 seguinte classe:

```
class Pilha {
    private Elem topo;

    public void crie() {
        topo = null;
    }
    ...
    public boolean vazia() {
        return topo == null;
    }
}
```

O que foi modificado foi o código dos métodos (veja acima), não a interface/assinatura deles.[3] 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 do tipo `Elem`, 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 disto. Resumindo, é mais seguro usar Proteção de Informação porque os dados são protegidos pelos métodos.

3.2 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 `extends` como mostrado no exemplo abaixo.

[3] Considere que a interface ou assinatura de um método é composto pelo seu nome, tipo dos parâmetros e tipo de retorno.

```
class A {
    public void put(int pn) {
        n = pn;
    }
    public int get() {
        return n;
    }
    private int n;
}

class B extends A {
    public void imp() {
        System.out.println( get() );
    }
}
```

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

```
void put(int pn)
int get()
void imp()
```

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

```
void main() {
    B b;
    b = new B();
    b.put(12);         // invoca A::put
    b.imp();           // invoca B::imp
    System.out.println( b.get() );  // invoca A::get

}
```

`A::put` é o método `put` da classe `A`. A classe `B` é chamada de "subclasse de `A`" e `A` é a "superclasse de `B`".[4]

O método `B::imp` possui uma chamada para um método `get`. O método invocado será `A::get`. Esta chamada poderia ser escrita como "`this.get()`" pois `this`, 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 "`this.n = pn`". `this` é o objeto que recebeu a mensagem, `b`. Em outras linguagens como Smalltalk, Self e Cyan, usa-se `self` ao invés de `this`.

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

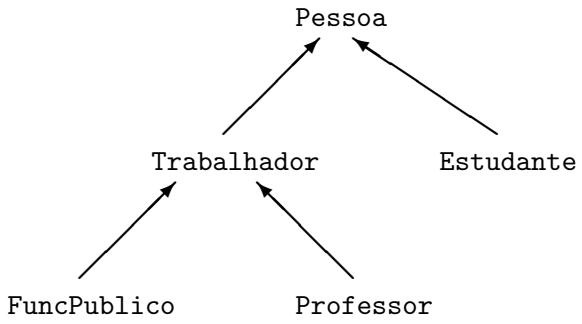
```
class B extends A {
    public int get() {
        return super.get() + 1;
    }
    public void imp() {
        System.out.println( get() );
```

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

```
    }
}
```

"`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 abaixo, 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 dadas abaixo:

```
class Figura {
    public Figura(int px, int py) {
        x = px;
        y = py;
    }
    public void imp() {
        System.out.println( "Centro(" + x + ", " + y +  ")" );
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    private int x, y;
}

class Circulo extends Figura
    public Circulo(float p_raio, int x, int y) {
        super(x, y);
        raio = p_raio;
    }
    public void setRaio( float p_raio ) {
```

```
        raio = p_raio;
    }
    public float getRaio() {
        return raio;
    }
    public void imp() {
        System.out.println("raio = " + raio);
        super.imp();
    }
    public float getArea() {
        return PI*raio*raio;
    }
    private float raio;
}
```

Se a classe `Circulo` precisar utilizar as variáveis `x` e `y` herdadas de `Figura`, ela deverá chamar os métodos `getX()` 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.

Classe `Figura` possui um método com este mesmo nome. Este método é chamado de *construtor* da classe. Quando um objeto é criado, com `new`, os parâmetros para `new` são passados ao construtor. No construtor da classe `Circulo` há uma instrução "`super(x, y)`". Esta instrução invoca o construtor da superclasse `Figura`.

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:

```
class B {
    public B() {
        a = new A();
    }
    public int get() { return a.get(); }
    public void put(int pn) { a.put(pn); }
```

```
    public void imp() { System.out.println(get()); }
    public A getA() { return a; }
}
```

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`.[5] 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`.[6] 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.

Praticamente todas as linguagens orientadas a objeto definem uma classe que é superclasse de todas as outras. Esta classe é chamada de `Object` em Smalltalk, Java e C#. Nesta classe são colocados métodos aplicáveis a todos os objetos, como `equals`, `clone`, `toString` e `hashCode`.

3.3 Polimorfismo

Se o tipo de uma variável `w` for uma classe `T`, a atribuição

```
w = null;
```

estará correta qualquer que seja `T`. Isto é possível porque `null` 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 `null`, que possui infinitos tipos.

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

```
void main() {
    Figura f;
    Circuo c;
    c = new Circulo();
    c.init( 20.0, 30, 50 );
    f = c;
    f.imp();
}
```

está correto. A atribuição

```
f = c;
```

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

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

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

Java 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:

```
void m(int i) {
    Figura f, aFig;
    Circulo aCir;

    aCir = new Circulo(20.0, 50, 30);
    aFig = new Figura(30, 40 );
    if ( i > 0 )
      f = aFig;
    else
      f = aCir;
    f.setRaio(10);
    ...

}
```

Se este método 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

```
void m( Figura f )
```

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.

```
void impVet( Figura []v ) {
    int i;
    for (i = 0; i < v.length; ++i)
        v[i].imp();
}


void main() {
    Circulo c1;
    Retangulo r1, r2;
    Triangulo
    Figura vetFig = {
        new Circulo(5, 80, 30),
        new Retangulo(30, 50, 70, 60),
        new Triangulo(10, 18, 30, 20, 40, 25),
        new Retangulo(20, 100, 80, 150)
    };
    impVet( vetFig );
}
```

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 que será mostrada acrescentando-se métodos nas classes `Figura` e `Circulo`:

```
class Figura {
```

```
    public void desenhe() { }
    public void apague() { }
    public void mova( int nx, int ny ) {
        apague();
        x = nx;
        y = ny;
        desenhe();
    }
    private int x, y;
}


class Circulo extends Figura {
    ... // métodos definidos anteriormente
    public void desenhe() {
        // desenhe um círculo
    }
    public void apague() {
        // apague um círculo
    }
    private float raio;
}
```

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.[7] 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

```
Circulo c;
c = new Circulo(10.0, 50, 30 );
c.mova( 20, 80 );
...
```

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

```
apague();
```

que é o mesmo que

```
this.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

[7] Seriam métodos abstratos.

```
Retangulo r;
r = new Retangulo( 30, 50, 70, 20 );
r.mova( 100, 120 );
...
```

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`.

É possível declarar diversos métodos com mesmo nome mas com parâmetros diferentes em diversas linguagens orientas a objeto. Então podemos ter

```
class Output {
    public void print(int n)   { ... }
    public void print(char ch) { ... }
    public void print(float f) { ... }

}
```

Este tipo de construção é chamado de "sobrecarga de métodos" e considerado um tipo de polimorfismo por alguns autores, embora não haja nenhum reuso de código.

3.4 Redefinição de Métodos

Algumas linguagens orientadas a objeto exigem que, ao redefinir um método na subclasse, se use a palavra-chave `override` (ou semelhante). Em Java, usa-se a anotação `@Override`:

```
class Person {
    public void print() { ... }
    ...
}

class Worker extends Person {
    ...
    @Override public void print() { ... }
}
```

Há bons motivos para se usar `override` ao redeclarar um método:

(a) o compilador irá emitir um erro se o programador está redefinindo, sem saber, um método da superclasse;

(b) o compilador emitirá um erro se o programador cometer algum erro no nome do método ou tipos dos parâmetros e tipo de retorno. Isto é, quando ele pensa que está redefinindo um método quando na verdade ele não está;

(c) se um método é adicionado à superclasse e alguma subclasse define um método com o mesmo nome, haverá um erro de compilação. O programador será avisado de que há uma redefinição do método;

(d) fica claramente documentado que o método em questão é uma redefinição.

Algumas linguagens permitem que, em uma redefinição de um método na subclasse, os tipos dos parâmetros e o tipo de retorno sejam diferentes daqueles do método da superclasse. Para que a linguagem seja estaticamente tipada, o tipo `B` de um parâmetro do método na superclasse pode ser substituído por um supertipo `A` de `B`. E o tipo do valor de retorno `C` da superclasse pode ser substituído por um subtipo `D` de `C`. Então se um método é declarado na superclasse como

`C m( B x )`

ele pode ser redeclarado na subclasse como

`D m( A x )`

Em desenho,

`↓m( ↑ )`

Uma regra de tipos como ↓ é chamada de co-variante. A regra do tipo ↑ é chamada de contra-variante (porque ela segue a direção contrária à herança).

Se esta regra não for obedecida, haverá um erro de tipos em execução. No exemplo abaixo, suponha que `D` herde de `C` que herde de `B` que herde de `A`. A classe `X` define apenas um método `mx` no qual `x` é o nome da classe em minúsculo. Assim a classe `C` define apenas um método `mc`.

```
class T {
    public C m( B x ) {
        x.mb();
        return new C();
    }
}

class R extends T {
    public B m( C x ) {
        x.mc();
        return new B();
    }
}

class Test {
    public void error() {
        T t = new R();
        t.m( new B() );        // 1
        C c = t.m( new C() );  // 2
        c.mc();                // 3
    }
}
```

Não há erros de compilação no código acima, assundo que o compilador permite esta redefinição do método `m` que não obedece à regra acima. Contudo, na execução da instrução `// 1`, um objeto de `B` será passado ao método `R::m` — haverá uma atribuição, nesta passagem de parâmetro, do tipo "`superclasse = classe`". Dentro deste método, mensagem `mc` é enviado ao objeto referenciado por `x` do tipo `B`. Esta classe não possui método `mc`, que está apenas presente em `C` e `D`. Em `// 2`, o tipo que o compilador deduz para o retorno do envio de mensagem é `C`, pois o tipo de `t` é `T` e esta classe define um método `m` cujo tipo de retorno é `C`. Mas o objeto retornado é do tipo `B`, pois o método chamado em execução é `R::m`. Este objeto é associado a `c` que recebe à mensagem `mc` em `// 3`. Há um erro de execução, pois `B` não possui este método.

3.5 Classes e Métodos Abstratos

Um método abstrato é declarado com a palavra-chave `abstract`. O corpo da classe não deve ser fornecido. Uma classe também pode ser declarada como `abstract`. Neste caso, ela pode ter zero ou mais métodos abstratos:

```
abstract class Expr {
    abstract public void genJava();
    public Type getType() {
        return type;
    }
    private Type type;
}
```

Classes abstratas são úteis para representar elementos comuns a várias classes em uma superclasses sendo que esta superclasse não representa um elemento do domínio do programa. Por exemplo, a classe `Figura` utilizada anteriormente deveria ter sido declarada como abstrata. Existem triângulos, retângulos, círculos etc, mas não existe uma figura geomética "figura". A classe `Expr` acima pode ser utilizada como superclasse de classes que representam expressões de uma linguagem qualquer. O compilador de Cyan utiliza uma classe semelhante a esta. Todas as expressões têm que ter um tipo (variável de instância `type`) e todas têm que gerar código em Java (método `genJava`). Há classes como `ExprLiteralInt`, subclasse de `Expr`, para representar um inteiro literal. Esta classe não é abstrata. Portanto ela tem que definir todos os métodos abstratos herdados de `Expr` — neste caso, apenas `genJava`. E não há em Cyan um elemento que seja representado por `Expr`, esta funciona apenas como uma superclasse. Por este motivo, não se pode criar objetos de classes abstratas. Mas pode-se utilizá-las como tipo de variáveis e parâmetros, tipo de retorno de métodos e com o operador `instanceof`.

Em resumo, classes abstratas devem obedecer algumas regras:

(a) métodos abstratos não devem definir um corpo e só podem ser colocados em classes abstratas;

(b) uma classe abstrata pode declarar tudo o que uma classe regular pode mais métodos abstratos (possivelmente nenhum método abstrato);

(c) uma classe que herda de uma classe abstrata deve ser declarada abstrata se ela não implementa todos os métodos abstratos herdados ou define novos métodos abstratos;

(d) não se pode criar objetos de uma classe abstrata, mesmo se ela não define nenhum método abstrato.

3.6 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 apresentam diferenças em relação a elas.

3.6.1 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;
a = 1;
b = new Janela(a, 5, 20, 30);
a = b;
a.desenhe();
...
```

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

Considere agora um método

```
void m(y) {
    y.desenhe();
    y.mova(10, 20);
}
```

de uma classe `A`. Assuma que exista uma classe `Janela` em Smalltalk, que é aquela do exemplo abaixo sem os tipos das declarações de variáveis.

```
class Janela {
    public Janela(int px, int py) { x = px. y = py; }
    private int x, y;
    public void desenhe() { ... }
    public void mova(int novo_x, int novo_y) {
        this.x = novo_x;
        this.y = novo_y;
        this.desenhe();
    }
}

class JanelaTexto extends Janela {
    ...
    public void desenhe() { ... }
}
```

Esta classe possui métodos `desenhe` e `mova`, 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 = new A();
a.m( new Janela() );
```

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 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 `mova` 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 `mova` e `desenhe` tal que `mova` aceita dois inteiros como parâmetros e `desenhe` 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

```
void m(y, b) {
    y.desenhe();
    y.mova(10, 20);
    if ( b )
        y.icon();
}
```

o código

```
a = new A();
a.m( new Janela(), 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 `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 )
    a = new Janela();
else
    a = 1;
if ( c > 1 )
    a.desenhe();
else
    a = a + 1;
```

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.

A linguagem Smalltalk (a linguagem real) emprega *seletores* para a definição de métodos e envios de mensagem. Um método unário, sem parâmetros, consiste de um nome simples como `name` ou `age`. Já um método com parâmetros deve ter um seletor para cada parâmetros, sendo que cada seletor é seguido, sem espaços em branco, por ':'. Então um método para inicializar o nome e a idade de um objeto pessoa poderia ser chamado como

```
pessoa name: 'Isaac Newton' age: 25.
```

Cyan utiliza uma sintaxe semelhante com duas diferenças: (a) um seletor com '`:`' não precisa ser seguido por parâmetro e (b) cada seletor pode ter mais de um parâmetro.

Um método que inicializa um protótipo[8] `Circle` em Cyan pode ser declarado como

```
object Circle
    fun x: (Int nx) y: (Int ny) radius: (Int nr) {
        x = nx;
        y = ny;
        radius = nr;
    }
    ...
end
```

Este método é chamado pela instrução

```
Circle x: 10 y: 40 radius: 5;
```

Uma conseguência desta sintaxe é que os programas se tornam muito legíveis. Neste último envio de mensagem sabe-se claramente qual o `x`, o `y` e o raio do círculo. Contraste esta instrução com a equivalente em outras linguagens:

```
circle.set(10, 50, 5);
```

3.6.2 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 (construtores excluídos). 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` dada anteriormente é

```
{ desenhe(), mova(int, int) }
```

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` é um subtipo do tipo da classe `Janela`.

```
class JanelaProcesso {
    public void desenhe() { ... }
    public void mova( int nx, int ny )   { ... }

    public void iniciaProcesso()         { ... }
    public void setProcesso( String s )  { ... }
}
```

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`.[9] Observe que toda subclasse é também subtipo, mas é possível existir subtipo que não é subclasse — a classe `JanelaProcesso` da é subtipo mas não subclasse de `Janela`.

[8] Como veremos em breve, Cyan declara protótipos e não classes.

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

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`.

3.6.3 C++

C++ [17] 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);
```

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.

3.6.4 Java

Java [13] [11] 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 extends Person implements Printable {
   ...
   public float 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[10] 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. Subtipo é definido indutivamente como:

(a) uma classe `C` é subtipo dela mesma;

(b) uma interface `I` é subtipo dela mesma;

(c) se uma classe `B` herda de uma classe `A`, `B` é subtipo de `A`;

(d) se uma interface `J` herda de uma interface `I`, `J` é subtipo de `I`;

(e) se `R` é subtipo de `S` e `S` é subtipo de `T`, então `R` é subtipo de `T`.

Em resumo, para descobrir as relações de subtipo desenhe um grafo no qual os vértices são as classes e arestas e no qual há aresta de `X` para `Y` se `X` herda de `Y` ou `X` implementa `Y`. Então `X` é subtipo de `Y` se há um caminho de `X` para `Y`.

As atribuições válidas em Java são

```
Tipo = subtipo
```

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

[10] 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.

```
Printable p;
Person person;
person = new Worker();  // cria objeto de Worker
p = person;
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 e métodos da superclass `Object` podem ser chamadas por meio da variável. Por exemplo, por meio de `p` acima pode-se chamar apenas o método `print` e aqueles de `Object`.

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 40).

3.6.5 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` no modelo C++.

```
public void Q( Janela x, int y ) {
    x.desenhe();
    if ( y > 1 )
        x.mova(20, 5);
}
```

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).

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 `desenhe` e `mova` e se `mova` 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

```
public void Q( x, y ) {
    x.desenhe();
    if ( y > 1 )
        x.mova(20, 5);
}
```

Como nem `x` nem `y` possuem tipos, não se exige que o objeto passado como primeiro parâmetro real a `Q` possua métodos `desenhe` e `mova`. De fato, na instrução

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

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

Como conseqüência, esta instrução pode ser executada com parâmetros `a` de qualquer classe que possua um método `desenhe` sem parâmetros. Ao contrário de POOL-I, Java e C++, a classe do parâmetro `x` de `Q` não precisa possuir também o método `mova`. 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) ou se a interface que é o tipo de `b` herdar da interface que é o tipo de `a`. 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 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

```
public wrong( Janela outra ) {
    ...
    w = outra.x;
    ...
}
```

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.

3.7 Herança Mixin

Uma classe mixin é um tipo de classe suportada por algumas linguagens que permite herança múltipla sem alguns dos problemas associados a este tipo de construção. O suporte a mixins varia largamente entre linguagens. Descreveremos estas classes em uma linguagem hipotética.

```
mixin class AgeMix {
    private Int age;
    public String getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}

mixin class NameMix {
    public abstract String getFirstName();
    public abstract String getLastName();
    public String getName() {
        return getFirstName() + " " + getLastName();
    }
}

class Person with NameMix AgeMix {
    private String firstName, lastName;
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }

}
```

A herança `mixin` é feita com a palavra-chave `with` seguida das classes mixin. À classe `Person` são adicionados os métodos e variáveis de instância das classes mixin (elas são *mixed into* `Person`). A diferença com herança múltipla é que classes mixin não podem ser usadas para criar objetos. Então elas podem ser incompletas, podem possuir métodos abstratos que são fornecidos pela classe ao qual elas são acopladas. No exemplo acima, `NameMix` exige que a classe ao qual ela é acoplada tenha métodos `getFirstName` e `getLastName`. Também não há colisão de nomes pois a hierarquia é linearizada. Se ambas as classes `NameMix` e `AgeMix` definissem um método `m`, não haveria colisão, pois `Person` herdaria o método de `NameMix` (pois esta classe está primeiro na lista após a palavra-chave `with`).

A classe `Person` terá métodos `getAge`, `setAge`, `getName`, `getFirstName` e `getLastName`. E variáveis de instância `age`, `firstName` e `lastName`. As classes mixins são classes que não possuem superclasse. Elas não herdam de ninguém.

3.8 Linguagens Baseadas em Protótipos

As linguagens vistas até agora são baseadas em classes, que são utilizadas para estruturar o programa. Classes são esqueletos a partir dos quais objetos são criados. Na maioria das linguagens, elas não existem em tempo de execução — a menos de menção em contrário, assumiremos isto. Objetos existem em execução, classes apenas em compilação.

Nas linguagens baseadas em protótipos não há classes. O papel destas é reservado aos protótipos, que são declarações literais de objetos. Por exemplo, a classe `Store` do início deste capítulo seria transformada, em Cyan, no seguinte protótipo:

```
object Store {
    fun get -> Int {
        return n;
    }
    fun put: Int pn {
        n = pn;
    }
    Int n
end
```

A diferença em relação à classe é que o protótipo é um objeto, ele pode então receber mensagens:

```
Store put: 0;
Out println: (Store get);
```

"`Store put: 0`" é o envio da mensagem "`put: 0`" ao objeto `Store`. "`put:`" é chamado de "seletor" da mensagem. Da mesma forma, "`Store get`" é o envio da mensage `get` ao objeto `Store`. O resultado é passado como parâmetro ao seletor `println:`.

Objetos podem ser criados em execução atráves do método `clone` que todos os objetos possuem:

```
var s = Store clone;
s put: 5;
Out println: (s get);
```

Então `clone` faz o papel de `new` em linguagens com classes. Em Cyan podem existir, para dado protótipo, métodos `new` ou `new:` para a criação de objetos.

A maioria das linguagens baseadas em protótipos é dinamicamente tipada e permite alterações dinâmicas nos protótipos e outros objetos em execução.[11] Então, por exemplo, pode-se adicionar uma variável de instância em execução. Ou um método. Pode-se também adicionar herança, remover herança, remover métodos e variáveis de instância, protótipos etc. Note que modificar um protótipo pode significar modificar todos os objetos criados a partir daquele protótipo. Ou todos os objetos criados a partir dele a partir da modificação. Modificar um objeto que não é protótipo altera apenas o objeto.

Métodos, protótipos, praticamente tudo são objetos. Então pode-se facilmente passar um método de um protótipo `A` como parâmetro para um método de um objeto `B` que irá adicioná-lo a `B`.

Há vários mecanismos para implementar herança neste tipo de linguagem. Tipicamente, um objeto possui uma variável de instância `parent` que referencia um objeto que desempenha o papel de "superclasse". Quando um objeto recebe uma mensagem, ele procura pelo método correspondente nele mesmo. Se não encontra, ele delega a mensagem para o objeto referenciado por `parent`. Mas para `parent` também é passada uma referência ao objeto original. Se há um envio de mensagem para `self` (ou `this`) no objeto referenciado por `parent`, a busca pelo método começa no objeto original. É um mecanismo praticamente igual ao de linguagens baseadas em classes.

Mudar a herança de um objeto é simplemente fazer a variável de instância `parent` referenciar outro objeto.

Há várias vantagens em linguagens baseadas em protótipos (LBP):

(a) elas tornam a programação mais concreta. O que o programa declara são realmente entidades existentes como uma pessoa, um elefante etc. Uma classe é mais abstrata pois é um molde utilizado para criar objetos. O molde e o objeto existem em tempos diferentes: um em compilação e outro em execução. Herança só é feita em compilação, classes não podem ser modificadas, objetos só adquirem vida em execução. As LBP eliminam esta distinção: objetos existem na compilação, herança pode ser feita em execução também;

(b) é fácil criar um novo protótipo em execução. Pode-se clonar um protótipo existente e adicionar um método a ele em seguida. Em programas que exigem uma grande quantidade de classes ou protótipos, isto é realmente útil;

(c) em algumas LBP, a proteção de informação existe até mesmo dentro do próprio objeto. Isto é, o acesso a variáveis de instância é feito atráves de métodos. Então pode-se facilmente modificar a representação do objeto sem alterar o próprio objeto;

(d) objetos únicos, como `Terra`, dos quais devem existir apenas um único exemplar são fáceis de serem criados (este é o padrão de projetos "Singleton");

(e) não há necessidade de meta-classes. Em linguagens baseadas em classes que consideram classes como objetos, as classes têm que ter uma classe, chamada de meta-classe. Mas esta meta-classe também é uma classe e portanto um objeto. E que tem que ter uma classe, que é uma meta-meta-classe. Há uma regressão potencialmente infinita que não é resolvida satisfatoriamente, no nosso ponto de vista, em nenhuma linguagem;

(f) o programa pode alterar a si mesmo em execução. Esta capacidade pode ser utilizada para fazer programas que seriam difíceis de fazer em linguagens sem esta característica.[12]

Há inúmeras críticas a linguagens baseadas em protótipo também:

[11] Aparentemente, em 2014, só existem duas linguagens que podem ser consideradas baseadas em protótipos e estaticamente tipadas: Cyan e Omega. Cuidado: há duas linguagens que se chamam Omega.

[12] Para saber mais, estude "programação adaptativa" ou "adaptive programming".

(a) humanos tendem a abstrair as entidades encontradas no mundo. Então a existência de classes é natural, pois estas são abstrações de entidades encontradas no domínio do sistema que está sendo implementado;

(b) a facilidade de alterar protótipos e objetos em execução torna os programas difíceis de entender. O código que se vê no monitor do computador não é aquele que será executado. O código executado depende do fluxo de execução do próprio código;

(c) a imensa maioria das LBP são dinamicamente tipadas e trazem consigo todas as desvantagens desta tipagem: detecção de erros de tipo somente em execução, desempenho ruim. Além disto, é difícil otimizar o código pois este pode ser alterado em execução.

Como exemplos de linguagens baseadas em protótipos podemos citar Self, Javascript, Cecil, Omega e Cyan.

3.9 Classes parametrizadas

A classe `Store` em Smalltalk permite o armazenamento de objetos de qualquer tipo.

```
class Store {
    private n;
    public put( i ) {
        n = i;
    }
    public get() { return n; }
}
```

Um objeto de `Store` guarda um outro objeto através do método `put` e retorna o objeto armazenado através de `get`.

A classe `Store` em Java é mostrada abaixo com o nome de `StoreInt`.

```
class StoreInt {
    private Int n;
    public void put( int i ) {
        n = i;
    }
    public int get() { return n; }
}
```

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 `StoreBoolean`, `StoreFloat`, 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 Java possui uma construção que oferece um pouco da flexibilidade de Smalltalk, chamada de classes genéricas.

```
public class Store<T> {
    private T n;
    public void put( T i ) {
        n = i;
    }
    public T get() { return n; }
}
```

A classe `Store` acima é genérica e possui um tipo com parâmetro. Na declaração de uma variável desta classe, deve ser especificado o parâmetro `T`:

```
Store<Integer> si = new Store<Integer>();
si.put(0);
System.out.println(si.get());
```

A atribuição de parâmetros reais a uma classe genérica é chamada de instanciação da classe. Em Java, todas as instanciações compartilham o mesmo código. Isto é, há um único código para `Store<Integer>`, `Store<Person>` e assim por diante. Isto limita as operações que podem ser feitas com o tipo dentro da classe genérica. Por exemplo, dentro de `Store` não se pode criar objetos de `T` e não pode existir um cast para `T`. Além disso, exceções não podem ser genéricas, vetores de `Store<Integer>` não podem existir, o operador `instanceof`[13] não pode ser usado com classes genéricas.

Todas estas limitações são causadas porque o código da classe genérica não é duplicada para cada instanciação. Outras linguagens, como Cyan e C++, duplicam o código. Em Cyan não há limitações ao uso dos parâmetros genéricos, embora possa haver erros na compilação quando um parâmetro formal é substituído por um real. Por exemplo, considere o protótipo `Box` nesta linguagem:

```
object Box<T>
    T elem
    fun set: T elem  {
        self.elem = elem;
    }
    fun get -> T { return elem; }
    fun process {
        elem prettyPrint;
    }
end
```

Podemos criar uma instanciação `Box<Int>`. Contudo, haverá um erro de compilação, pois `elem` será do tipo `Int` e este tipo não possui um método `prettyPrint`. Uma mensagem `prettyPrint` é enviada a `elem` no último método.

Observe que, em Cyan, `Box` não é um protótipo, é apenas uma máscara (ou esqueleto — *template* em Inglês) para a criação de protótipos. Os exemplos (instâncias) construídos a partir de `Box`, como `Box<Int>`, são realmente protótipos. Eles podem ser usados como tipo de variáveis, em herança etc. Cada instanciação do protótipo com tipos diferentes causa a criação de um novo código fonte.

Classes ou protótipos genéricos de C++ e Cyan, com duplicação de código, possuem os problemas semelhantes a linguagens dinamicamente tipadas. Alterações na classe ou protótipo genérico pode invalidar instanciações que estavam funcionando adequadamente. Por exemplo, suponha que `Box`

[13] Em java, `x instanceof C` retorna `true` se o objeto referenciado por `x` é uma instância da classe `C`.

tenha sido criado sem o método `process`. Não há nenhum erro na instanciação `Box<Int>`. Contudo, ao acrescentar `process` estaríamos introduzindo um erro nesta instanciação. **O erro não seria detectado na compilação de `Box` e sim na instanciação `Box<Int>`**. Diferente das linguagens dinamicamente tipadas, o erro seria detectado em compilação.

3.10 Closures

Cyan permite a declaração de funções anônimas que podem ter parâmetros e retornar um valor:

```
var b = { (: Int n -> Int :)
    ^ n * n;
};
```

O tipo dos parâmetros e o tipo de retorno (opcional) são declarados entre `(:` e `:)`. Este tipo de função anônima é chamado de `função` ou função anônima em Cyan, que são objetos como todos os outros valores da linguagem.

O valor retornado por uma função é dado após "`^`". Ao executar a instrução acima, a função é atribuído a `b`. Mas a função não é executada, o que deve ser feito enviando-se a mensagem `eval:` à variável `b`:

```
var b = { (: Int n -> Int :)
    ^ n * n;
};
Out println: (b eval: 5);
```

É impresso `25` na saída padrão. Sendo uma função um objeto, `b` é uma instância de um protótipo e portanto tem um tipo. O tipo de `b` é

```
Function<Int, Int>
```

O último `Int` é o valor de retorno. Uma função

```
var concat = { (: Int n, Char ch -> String :)
    ^ (ch + n) asString;
};
```

tem o tipo

```
Function<Int, Char, String>
```

Não se preocupe com a instrução de dentro da função. Ela soma um caráter a um número, resultando em um caráter, e converte para `String`.

Funções podem acessar variáveis locais e de instância:

```
int other = 1;
var b = { (: Int n -> Int :)
    ^ n + other;
};
Out println: (b eval: 0);
other = 2;
Out println: (b eval: 0);
```

O valor da variável `other` utilizado é aquela do momento da avaliação da função, quando o método `eval:` é chamado. Então são impressos os números `1` e `2`. Uma função também pode modificar o valor das variáveis externas que ele acessa:

```
int sum = 0;
var b = { (: Int n :) sum = sum + n; };
b eval: 1;
b eval: 2;
Out println: sum;
```

Será impresso "3".

Vetores em Cyan possuem um método chamado `foreach:` que aceita uma função como parâmetro. Este método chama a função para cada elemento do vetor. Então a função deve aceitar um tipo igual ao do elemento do vetor:

```
int sum = 0;
var Array<Int> v = {# 1, 2, 3, 4, 5 #};
v foreach: { (: Int n :)
    sum = sum + n;
};
Out println: sum;
```

Será impresso "15". Um vetor literal em Cyan é dado entre `{#` e `#}`. Em execução, quando a função acima é criada, é feita uma ligação da variável livre `sum`, que não foi declarada na função, e a variável local `sum` — elas passam a ser a mesma variável. Isto é, a função *close over* suas variáveis livres. É interessante notar que esta notação vem da Lógica e, em particular, do Cálculo Lambda. Em Lógica, as fórmulas $\forall x\,(f(x) = y)$ e $f(x) = y$ têm uma e duas variáveis livres: y no primeiro caso e x e y no segundo. Quando se adicionam os quantificadores, $\exists y\,\forall x\,(f(x) = y)$ e $\forall x\,\forall y\,f(x) = y$, as variáveis passam a ser ligadas (ao quantificador) e as fórmulas são chamadas de fechadas (closed).

Uma *closure* é um bloco de código, possivelmente com variáveis *livres*, no qual estas variáveis foram ligadas a variáveis locais, parâmetros ou variáveis de instância em execução. Então uma *closure* é um objeto que existe em tempo de execução, não é a sequência de instruções que está no código. *Closures* são criadas a partir de funções anônimas (como Cyan) ou não. Pode-se ter funções aninhadas em que a função interior acessa variáveis locais declaradas nas funções mais externas.

Veremos outros exemplos de funções.

```
   // imprime os elementos do vetor
var Array<Int> v = {# 1, 2, 3, 4, 5 #};
v foreach: { (: Int n :)  Out println: n };

   // imprime os número de 1 a 10
1..10 foreach: { (: Int n :)  Out println: n };

   // imprime "cinco vezes" cinco vezes
5 repeat: {
    "cinco vezes"  println;
};
( age < 3 ) ifTrue:  { "baby" println }
            ifFalse: { "non-baby" println };
```

`1..10` é um intervalo em Cyan, que possui um método `foreach:`. O protótipo `Int` possui um método `repeat:` que toma uma função como parâmetro. A última instrução é um `if` implementado com envio de mensagens. Duas funções são passados como parâmetro. Em Smalltalk, não há comando `if` ou `while`. Todas as estruturas de repetição são implementadas com envio de mensagens.

3.11 Meta-programação

Meta-programação é programação sobre programas, o que pode ter inúmeros sentidos: programas que manipulam outros programas, um programa que examina a si mesmo, um program que modifica a si mesmo e criação de código em tempo de compilação.

3.11.1 Reflexão Introspectiva

O mais simples tipo de meta-programação acontece quando um programa examina a si mesmo, o que é chamado de *reflexão introspectiva*. Por exemplo, em Java pode-se listar os métodos e variáveis de instância de uma classe:

```
package main;

import java.lang.reflect.Method;

public class Reflect {

    public void r() {
        Class<?> c = this.getClass();
        Method m[] = c.getDeclaredMethods();
        for (int i = 0; i < m.length; i++)
            System.out.println(m[i].toString());
    }

    public int fat(int n) { return n <= 0 ? 1 : n*fat(n-1); }
    public String asString() { return "Reflect"; }
}
```

A chamada `this.getClass()` retorna um objeto da classe `Class` que descreve a classe de this, que neste exemplo é `Reflect` (pois não há subclasses). Usando este objeto, o envio de mensagem

```
c.getDeclaredMethods()
```

retorna objetos descrevendo os métodos de `Reflect`.

Ao se chamar `r`, como em "`(new Reflect()).r()`", será impresso

```
public void main.Reflect.r()
public int main.Reflect.fat(int)
public java.lang.String main.Reflect.asString()
```

Pode-se invocar um método pelo seu nome, como no código

```
Reflect reflect = new Reflect();
Method m = reflect.getClass().getMethod("asString");
System.out.println( (String) m.invoke(reflect) );
```

O método `getMethod` retorna um objeto que descreve o método cujo nome é o parâmetro. Pode-se fornecer os tipos dos parâmetros. Neste caso, isto não é necessário pois `asString` não possui parâmetros. Neste exemplo, o método `invoke` de `Method` chama o método do objeto `reflect` cujo nome é `asString`.

Em Cyan, pode-se chamar um método cujo nome está em uma string usando-se o operador ‘ (*backquote*, ASCII 96):

```
var String s = "name";
    // cria um objeto Person
var Person p = Person("Newton", 85);
   // envia a p a mensagem 'name'
Out println: ( p `s );
s = "age";
   // envia a p a mensagem 'age'
Out println: ( p `s );
```

3.11.2 Reflexão Comportamental

Linguagens que suportam reflexão comportamental permitem alterar o próprio código do programa durante a execução. Smalltalk, Self, Ruby, Groovy e Cyan são linguagens que suportam este tipo de reflexão, embora os limites do que se possa fazer varie de linguagem a linguagem. Usualmente, pode-se inserir e remover variáveis de instância e métodos em classes, mudar a herança, criar novas classes, eliminar classes etc. As operações de remoção são altamente inseguras e podem causar erros de execução mais facilmente do que as operações de inserção.

Como exemplo de alteração de uma classe, em Groovy pode-se adicionar um método `swapCase` à classe `String` pelo seguinte código, tomado de `http://groovy.codehaus.org/ExpandoMetaClass`.

```
String.metaClass.swapCase = {->
      def sb = new StringBuffer()
      delegate.each {
           sb << (Character.isUpperCase(it as char) ?
                   Character.toLowerCase(it as char) :
                   Character.toUpperCase(it as char))
      }
      sb.toString()
}
```

Depois de executado, temos que `"aBc".swapCase()` retorna `"AbC"`.

Em Cyan, pode-se adicionar ou trocar um método em um protótipo ou objeto através do método `addMethod: ...` definido no super-protótipo `Any` que é herdado por todos os outros. Este é um tipo especial de método chamado de "método de gramática" em que os seletores podem ser descritos utilizando expressões regulares. Para saber mais, consulte o manual da linguagem.

```
public fun (addMethod:
             (selector: String ( param: (Any)+ )?
               )+
              (returnType: Any)?
              body: ContextObject) t
```

Um método `toString` pode ser adicionado a um protótipo `Person` da seguinte forma:

```
Person addMethod:
       selector: #toString returnType: String
       body: { (: Person self :)
           ^ name + " (" + age + ")";
       };
```

A função passada como parâmetro ao seletor `body:` chama-se "função de contexto" e tem `self` como primeiro parâmetro. Dentro desta função as mensagens enviadas a `self`, aquelas sem receptor especificado, devem corresponder a métodos do tipo de `self`, que é `Person`. O código acima está correto se `Person` possui métodos `name` e `age` que retornam objetos do protótipo `String` e `Int`. `+` é a concatenação de uma string com qualquer outro objeto (como em Java).

```
object Person
    fun name -> String { return _name; }
    fun name: (String newName) { _name = newName; }
    fun age -> Int { return _age; }
    fun age: (Int newAge) { _age = newAge; }
    ...
    String _name
    Int _age
end
```

Após a execução do método `addMethod:` pela instrução acima, pode-se enviar a mensagem `toString` para `Person` ou seus objetos:

```
Person name: "Newton";
Person age: 45;
var String s = Person toString;
```

Mas ... o último envio de mensagem, `Person toString` está incorreto se assumirmos que `Person` não tinha um método `toString`. O compilador procurará por um método `toString` em `Person` e não o encontrará, resultando em um erro de compilação. É necessário colocar `?` antes do nome do método. Isto fará com que o compilador não confira se o método existe ou não, exatamente como em linguagens tipadas dinamicamente. O tipo de retorno do método será considerado como `Any`, o super-protótipo de todo mundo.

```
Person name: "Newton";
Person age: 45;
var String s = Person ?toString;
```

O compilador considera que qualquer valor retornado por um método chamado através de `?` é compatível com qualquer tipo. Então o compilador considera que `Person ?toString` retorna uma `String`. Naturalmente, é inserido um teste para, em execução, conferir se este valor pode mesmo ser convertido para `String`. Então esta última linha é equivalente a

```
var String s = String cast: (Person ?toString);
```

Em Cyan pode-se adicionar variáveis de instância a qualquer protótipo que herde o protótipo mixin `AddFieldDynamicallyMixin`. Qualquer protótipo que herde deste mixin pode introduzir variáveis através do envio de mensagens prefixadas por `?`. Por exemplo, considere um protótipo `Pessoa` que não possui nenhum método ou variável de instância, mas que herda de `AddFieldDynamicallyMixin`.

```
object Person mixin AddFieldDynamicallyMixin
end
```

Pode-se adicionar uma variável de instância enviando-se uma mensagem a `Person`:

```
Person ?name: "Newton";
Out println: (Person ?name);
```

Este envio de mensagem cria dois métodos: `name: String` e `name -> String`. O primeiro método inicializa uma variável de instância (que é guardada em uma tabela hash) e o segundo retorna o valor da variável.

O leitor interessado deve procurar uma outra forma de reflexão comportamental chamada de metaobjetos de tempo de execução. Um metaobjeto é um objeto que pode ser acoplado ao outro objeto em execução. Todas as mensagens enviadas ao objeto são redirecionadas ao metaobjeto. Este pode fazer algum processamento e enviar a mensagem original ao objeto. Este tipo de comportamento é implementado por mixins em Groovy e Ruby.

3.11.3 Metaobjetos de Tempo de Compilação

Um metaobjeto de tempo de compilação é um objeto cujos métodos são executados em compilação. O compilador inclui o metaobjeto ao seu próprio código. Métodos do metaobjeto podem modificar como a compilação é feita. Eles podem fazer conferências adicionais, inserir métodos e variáveis de instância em classes e protótipos, modificar métodos existentes etc. Exemplificaremos este conceito usando a linguagem Cyan. Não serão dados detalhes de como os metaobjetos são implementados, apenas detalhes de alto nível em relação à implementação.

Metaobjetos em Cyan são *chamados* em tempo de compilação usando-se `@` como em

```
package people;

object Person
    @init(name, age)
    String name;
    Int age;
end
```

O Metaobjeto `init` é chamado com os parâmetros `name` e `age`. Metaobjetos são declarados em *packages*, mais especificamente, no diretório `meta` de um *package*. Em Cyan, eles são feitos atualmente em Java. O *package* `cyan.lang` é incluído automaticamente por qualquer arquivo fonte em Cyan e, junto com ele, são incluídos os metaobjetos padrões como `init` (usado acima), `prototypeCallOnly`, `annot`, `doc`, `text`, `checkIsA` etc. Para exemplificar, no diretório `cyan\lang\meta` há um arquivo `CyanMetaobjectInit.java` que implementa o metaobjeto `init`. O compilador carrega esta classe e cria um objeto dela, que chamaremos de `metaInit`.

Quando o compilador Cyan encontra `@init`, ele solicita ao metaobjeto `metaInit` que gere o código que deve substituir `@init(name, age)`. Isto é, a mensagem `cyanCode(...)` é enviada a `metaInit`. Obviamente, os parâmetros são passados ao método `cyanCode` como strings. Este método retorna uma string que substitui `@init(name, age)`. O código resultante fica:

```
package people;

object Person
    fun init: String name, Int age {
        self.name = name;
        self.age = age;
    }
    String name;
    Int age;
end
```

Mas como o método `cyanCode` de `CyanMetaobjectInit.java` sabe que `name` é do tipo `String` e `age` é do tipo `Int`? Estas informações são passadas a `cyanCode` através de um objeto `context`:

```
metaInit.cyanCode(parameters, context)
```

Este objeto possui muitas das informações que o compilador possui. Através dele pode-se saber as variáveis de instância do protótipo, os seus métodos, de quem o protótipo herda etc. Como um outro exemplo, suponha a existência de um metaobjeto `beforeAfter` que modifica métodos:

```
package zoo;

object Animal
    @beforeAfter fun print {
        name println;
    }
    ...
end
```

Este metaobjeto poderia introduzir código no método ao qual ele está acoplado. O resultado poderia ser:

```
package zoo;

object Animal
    fun print {
        "before print" println;
        name println;
        "after print" println;
    }
    ...
end
```

Então o metaobjeto `beforeAfter` modifica o próprio método. Isto é feito modificando-se a Árvore de Sintaxe Abstrata (ASA) do método por um método do metaobjeto. A ASA de um método é a representação em forma de objetos do texto do método. Assim, um método é representado por um objeto da classe `MethodDec` (em Java) que possui variáveis de instância para representar os seletores dos métodos, os parâmetros (nome e tipo), tipo de retorno, se é público, privado ou protegido e suas instruções. Um dos métodos do metaobjeto pode modificar ou conferir qualquer aspecto deste objeto. Poderia, por exemplo, mudar o tipo de retorno ou conferir se os seletores estão começando por letra minúscula.

O metaobjeto `prototypeCallOnly` deve ser acoplado a um método. Ele confere se o método foi chamado através de um protótipo. Se não foi, ele sinaliza um erro.

```
package data;

object Date
    @prototypeCallOnly
    fun getCurrentTime -> Long { ... }
    ...
end
```

Pode-se usar

```
ct = Date getCurrentTime;
```

Mas a instrução seguinte causa um erro de compilação

```
ct = (Date new) getCurrentTime;
```

Este metaobjeto declara um método `checkMessageSend` que é chamado pelo compilador em todos os envios de mensagem em que este método possivelmente seria chamado. O compilador passa como parâmetros o receptor da mensagem e os parâmetros. E o método confere se o receptor é um protótipo. Se não for, pede ao compilador para sinalizer um erro.

Poder-se-ia definir um metaobjeto `memoized` (como em Groovy) para memorizar o resultado de um método que realiza alguma computação cara em termos de tempo. Este metaobjeto iria inserir no protótipo uma tabela hash para guardar os resultados já obtidos e modificar o método de tal forma que ele verifique na tabela se a computação com o valor pedido (parâmetro) já não foi realizada antes.

```
package myMath;

object MyMath
    @memoized
    fun fibonacci: (Int n) -> Int { ... }
    ...
end
```

3.11.4 Macros

Macros são funções que são executadas em tempo de compilação e que produzem código que é então compilado.

Para exemplificar este tópico, mostraremos como macros serão definidos em Cyan.[14] Nesta linguagem, um macro pode ter mais de um "seletor", cada um deles com parâmetros. Os seletores não são terminados por ":". Pode-se declarar uma função macro usando-se a palavra-chave `macro` antes de "`fun`":

```
macro fun "unless" (String expr) "do" (String b)  -> String
          where Expr expr, expr getType  == Boolean
          where Function<Nil> b
{
   return "if ! #{expr} #{b}";
}
```

Este macro acrescenta à linguagem Cyan duas novas palavras reservadas: `unless` e `do`. É como se estas duas palavras fossem seletores deste método, cada um tomando uma `String` como parâmetro. O tipo de retorno também é `String`. Assuma que sempre será assim: os tipos dos parâmetros e do retorno serão `Strings`. A cláusula `where` na segunda linha restringe o que pode ser o primeiro parâmetro: `expr` pode ser um objeto do tipo `Expr` e o tipo de `expr` deve ser booleano. A próxima cláusula restring `b` para um objeto de `Function<Nil>`.

`Expr`, `Boolean` e `Function` são classes da Árvore de Sintaxe Abstrata (AST) do compilador Cyan. Estas classes são utilizadas para representar o programa Cyan. `Expr` representa uma expressão. Esta classe possui um método `getType` que retorna o tipo da expressão. Então "`expr getType`" é o envio da mensage `getType` ao objeto referenciado por `expr`.

[14] Este conceito ainda não faz parte da linguagem.

Em Cyan, se existe uma variável `n`, então `"valor = #{n}"` é o mesmo que
`("valor = " + n)`
Isto é utilizado no código acima.

Quando o compilador encontra um uso de `unless`, como em

```
unless n >= 0 do {
    Out println: "Please type an integer >= 0";
};
```

ele procura por um macro que comece por "`unless`". De fato, ele faz outras buscas que não nos interessam antes disto.

O macro declarado anteriormente é encontrado. O compilador confere se que o que está entre `unless` e `do` é uma expressão e se esta expressão é do tipo `Boolean` (cláusula `where` do macro). Após isto é conferido se depois de "`do`" se segue uma função sem parâmetros ou valor de retorno. Nenhuma das conferências resulta em erro. Então o compilador chama a função macro passando como parâmetros as seguintes strings:

```
"n >= 0"
"{\n    Out println: "Please type an integer >= 0";\n}"
```

O macro retorna uma string

```
"if  ! (n >= 0) {\n    Out println: "Please type an integer >= 0";\n}"
```

que o compilador coloca no lugar de

```
unless n >= 0 do {
    Out println: "Please type an integer >= 0";
}
```

Isto é, a "chamada do macro" é apagada do código e o retorno do método macro é inserido. Note que esta mudança no código afeta apenas os dados internos do compilador. O código fonte do usuário não é afetado.

Para entender melhor como macros funcionam, considere o macro

```
macro fun "test"
{
   Out println: "Em compilação";
   return "Out println: \"Em execução\"";
}
```

Em um programa Cyan

```
package main
object Main
    fun run {
        test;
    }
end
```

O compilador iria imprimir `"Em compilação"` na saída padrão (é o que faz o método `println:` de `Out`). Ao executar o método `run`, seria impresso

```
"Em execução"
```

3.12 Linguagens Específicas de Domínio

Uma linguagem específica de domínio[15] (LED) é uma linguagem adequada para um domínio específico. Uma LED pode ser uma linguagem de programação (o mais usual) ou não. No último caso, ela pode apenas descrever dados ou não ter todos os comandos necessários para que seja Turing-completa.

Existe uma enorme quantidade de LED´s disponíveis. Mostraremos algumas delas.

(a) JSON, Javascript Object Notation (json.org). É uma linguagem para descrever dados que torna estes fáceis de visualizar. É uma LED, mas não uma linguagem de programação. O exemplo abaixo foi obtido de `http://json.org/example.html`.

```
{
  "menu": {
      "id": "file",
      "value": "File",
      "popup": {
          "menuitem": [
            {"value": "New", "onclick": "CreateNewDoc()"},
            {"value": "Open", "onclick": "OpenDoc()"},
            {"value": "Close", "onclick": "CloseDoc()"}
          ]
      }
  }
}
```

(b) SQL, para gerenciar dados em banco de dados.

```
select *
    from Estudante
    where nota >= 6
    order by nota;
```

SQL é uma linguagem de programação.

(c) linguagens regulares nas quais um conjunto (linguagem) é descrita suscintamente. Esta linguagem não é Turing-completa.

```
[A-Za-z][A-Za-z0-9_]+
1+(a|b)*cc+
```

(d) HTML, XML, que não são Turing-completas. São apenas descritivas;

(e) a linguagem utilizada no programa `make` do Unix;

(f) TEXe LATEXpara processamento de texto;

(g) linguagens de programas que geram compiladores como YACC, Bison, ANTLR etc;

[15] Em Inglês, DSL, Domain Specific Language.

(h) uma linguagem que controla uma máquina industrial, uma linguagem para controlar um carro de brinquedo, para programar um jogo de computador, para criar uma interface gráfica, para representar os movimentos das peças em um jogo de Xadrez etc.

Antes de continuar, apresentaremos uma LED para movimentar um carro de brinquedo programável. Esta linguagem é implementada em Java.

```
class Car {
    public Car start() { ... return this; }
    public Car forward(int cm) { ... return this; }
    public Car left(int degree) { ... return this; }
    public Car right(int degree) { ... return this; }
    public Car stop() { ... return this; }
    ...
}
```

Usualmente estes métodos retornariam `void`. Aqui eles retornam o objeto que recebeu a mensagem, `this`. Isto permite que envios de mensagem possam ser encadeados:

```
Car c = new Car();
c .start()
  .forward(10)
  .left(50)
  .stop();
```

Este encadeamento torna a aparência do código menos parecido com Java e mais com uma LED. O código fica mais natural.

Existem dois tipos de LEDs, as internas e externas. As internas são aquelas implementadas dentro de uma linguagem de programação, usando a sintaxe usual da linguagem (como a LED acima). As externas exigem que se faça um compilador específico para elas e não precisam de uma linguagem de programação hospedeira. As LEDs externas utilizam uma gramática qualquer, não há restrições quanto à sintaxe ou semântica como a LED interna.

Diversas linguagens dão suporte à construção de LED´s internas, como Cyan, Groovy, Ruby e Scala. Este suporte se dá através de diversas características e construções destas linguagens:

(a) suporte a *closures*. Com o uso destas pode-se criar comandos que tomam blocos de código como parâmetros. Uma pequena LED pode ser utilizada para trabalhar com arquivos em Cyan:

```
object MyFile
    fun open: (String name) eachLine: Block<String> b close: {
    }
    ...
end

MyFile open: "data.txt"  eachLine: { (: String line :)
    if ( line contains: "//" )
        line println; // print only lines with comments
}
close: ;
```

(b) tipagem dinâmica também ajuda na construção de LED´s. Há um item a menos a ser fornecido, o tipo, o que torna as LED´s mais fáceis de programar. Este fato se torna mais importante quando os usuários da LED são leigos em Computação. Por exemplo, pode-se ter uma LED para a prescrição de medicamentos ou para implementar regras de negócios. Em ambos os casos o usuário pode não ser da área de Computação;

(c) listas, tuplas, tabelas hash, vetores e intervalos literais. Estes objetos literais facilitam a criação de estruturas complexas de dados. Veja o exemplo abaixo em Cyan.

```
  // uma tupla literal
var t = [. "Isaac Newton", 45 .];
    // 't f2' retorna  45
Out println: "age is " + (t f2);
    // vetor literal
var Array<Int> v = {# 1, 2, 3, 4, 5 #};
    // uma tabela Hash hipotética
var myHash = [*  "Newton" : 1642, "Leibniz" : 1646, "Neumann" : 1903 *];
    // imprime 1903
Out println: (myHash get: "Neumann");
'a'..'z' foreach: { (: Char ch :) ch println };
```

(d) o uso opcional de parenteses e ';' torna as DSL´s mais legíveis:

```
Car c = new Car
c .start
  .foward 10
  .left   50
  .stop
```

(e) inserção de métodos em tipos básicos como `int` ou simulação desta inserção. Isto permite códigos como

```
int n = howMuchTime();
int numDays = 5.days;
int totalTime = 4.hours + n.hours;
```

Em Ruby, novos métodos podem ser inseridos em tipos básicos. Em Groovy pode-se usar Categorias para simular esta inserção ou realmente inseri-los nos tipos básicos;

(f) métodos com nomes de operadores como `+`, `<<` etc.

```
Matriz m, a, b;
...
m = a * b;
Date d = new Date(01, 04, 2014);
d = d + 5; // d mais cinco dias
```

(g) uso de símbolos ao invés de strings. Símbolos em Smalltalk e Cyan são strings especiais iniciadas por `#`

```
house color: #cyan;
car setColor: #red;
Out println: #first, #second;
```

(h) algumas linguagens criam automaticamente uma tabela hash a partir dos parâmetros. Como exemplo, em Groovy[16]

```
take 1.pill
   of: Chloroquinine,
   after: 6.hours
```

Foi definido um método

```
take(Map m, MedicineQuantity mq)
```

`1.pill` é o valor de `mq`. Já "`of: Chloroquinine`" e "`after: 6.hours`" são agrupados em um `map` (tabela hash) e passados como o primeiro parâmetro.

LED´s possuem inúmeras vantagens sobre linguagens de propósito geral (LPG):

(a) o código é mais legível, mais fácil de manter e mais confiável do que código de uma LPG;

(b) o número de linhas de código é muito menor do que o número de linhas necessárias em uma LPG;

(c) pode ser compreendida por não especialistas (se for projetada com esta finalidade).

E algumas desvantagens também:

(a) há uma curva de aprendizado que não existe se uma LPG for utilizada;

(b) a LED precisa ser projetada e implementada;

(c) a LED pode ser mais ineficiente do que o código implementado em uma LPG. Mas em alguns casos pode o código pode até ter execução mais rápida;

(d) pode ser difícil ou impossível integrar código LED com o código normal de uma aplicação;

(e) frequentemente precisa crescer com o acréscimo de novas funcionalidades;

(f) pode não contar com suporte de uma IDE (mas pode também ter este suporte — veja Eclipse XText).

As LEDs internas possuem também vantagens e desvantagens quando comparadas com as LEDs externas. As vantagens de uma são as desvantagens de outra.

Vantagens de LEDs internas sobre as externas:

(a) não necessita de um compilador, sendo portanto mais fácil de fazer do que as LED´s externas;

(b) todas as construções disponíveis para a linguagem estão também disponíveis para a LED;

(c) pode-se utilizar todas as bibliotecas disponíveis para a linguagem;

[16] Exemplo tomado de GR8Conf 2009: Practical Groovy DSL por Guillaume Laforge.

(d) a curva de aprendizado é menor do que a LED externa, pois ela utiliza os recursos já conhecidos pelo programador;

(e) permite o auxílio do IDE pois este pode ajudar a completar o código enquanto o usuário digita (por exemplo).

Vantagens de LEDs externas sobre as internas:

(a) pode utilizar uma gramática qualquer, não relacionada a qualquer linguagem de programação;

(b) pode-se ter auxílio da IDE para a LED externa. Mas frequentemente isto envolve um trabalho extra do criador da LED (ele/ela deveria implementar um plugin para a IDE). Há ferramentas como o XText que já fornecem o suporte à LED automaticamente. Então uma LED externa feita com o XText (ou outras ferramentas semelhantes) já conta com suporte pelo IDE;

(c) bibliotecas externas podem ser utilizadas, mas a LED deve ter construções que permitam a importação e sintaxe que permita o uso destas bibliotecas.

3.13 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 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.

3.14 Exercícios

34. Defina objeto. É um objeto um valor? Isto é, ele se assemelha mais ao valor 5 que está um `i` após a instrução

```
i = 5;
```

ser executada ou ele se assemelha mais ao:

- tipo `int`;
- variável `i`;

?

35. É um objeto mais semelhante a uma pessoa ou ao seu nome?

36. É um objeto mais semelhante a um projeto de um avião 767 ou a um avião que existe no mundo real?

37. Explique as vantagens de proteção de informação usando um exemplo.

38. Explique porque proteção de informação impede que modificações na representação (variáveis de instância) de uma classe invalidem outras classes do mesmo programa.

39. Proteção de informação faz os programas se tornarem mais rápidos? Explique.

40. Subclasses são mais gerais ou específicas do que as superclasses?

41. Uma subclasse pode redefinir um método herdado da superclasse? Mostre um exemplo.

42. Faça um exemplo de classe onde a palavra chave `super` é utilizada.

43. Porque uma classe que declara variáveis de instância protegidas viola a proteção de informação?

44. O que é uma variável polimórfica? E um parâmetro polimórfico? E um método polimórfico? E um valor polimórfico? Cite exemplos na sua resposta.

45. Explique detalhadamente, através de um exemplo, como polimorfismo causa reuso de código.

46. Uma classe `Desenho` possui um vetor de objetos de `Figura` (classe dada em aula). Um dos métodos desta classe é dado abaixo.

```
class Desenho {
    private Figura []v;
    private int size;

    public void desenhe() {
        int i;
        for (i = 0; i < n; ++i)
            v[i].desenhe();
    }
}
```

Um objeto da classe `Desenho` contém referencias para objetos de `Figura` e suas subclasses `Retangulo`, `Triangulo`, `Circulo`, `Elipse` e `Poligono`. Pergunta-se: esta classe poderia ser feita declarando-se o vetor como

```
Circulo []v;
```

? Responda explicando se `v` poderia se referir aos objetos de `Figura` e suas subclasses.

47. Baseado na hierarquia

```
class A {
    public void s() { }
}

class B extends A {
    public void m() { }
}

class C extends B {
    public void s() { }
    public void m() { }
}
```

responda quais métodos serão executados pelos comandos abaixo. Alguns dos comandos resultam em erros de compilação. Quando isto acontecer, ignore o comando.

```
void main() {
    A a;
    B b;
    C c;

    a = new A();
    a.s();
    a.m();

    b = new B();
    a = b;
    a.s();
    a.m();
    b.s();
    b.m();

    c = new C();
    a = c;
    a.s();
    a.m();
    c.s();
    c.m();

    b = c;
    b.s();
    b.m();
}
```

48. Utilizando a hierarquia do exercício anterior, cite os métodos de cada classe, inclusive os herdados. Cite também a procedência de cada método, como no exemplo:

```
 classe F:
    F::m
```

```
G::p
F::h
```

onde `F` herda o método `p` de `G` e define os métodos `m` e `h`.

49. Descreva em palavras o funcionamento da classe `Politico`.

```
class Pessoa {
    public void facaAlgumaCoisa() { }
}

class Preguicoso extends Pessoa {
    public void facaAlgumaCoisa() {
        println("naooooooo!!!!!!!\n");
    }
}

class Trabalhador extends Pessoa {
    public void facaAlgumaCoisa() {
        println("Ja vou !\n");
    }
}

class Politico extends Pessoa {
    public Politico() {
        // nasce preguicoso
        preguicoso = new Preguicoso();
        corrente = preguicoso;
        trabalhador = new Trabalhador();
    }
    public void facaAlgumaCoisa() {
        corrente.facaAlgumaCoisa();
    }
    public void emEpocaDeEleicao() {
        corrente = trabalhador;
    }
    public void foraDeEpocaDeEleicao() {
        corrente = preguicoso;
    }
    private Pessoa corrente, trabalhador, preguicoso;
}
```

50. Empregando a definição da linguagem POOL-I, qual o tipo da classe `Politico`? E da classe `Trabalhador`?

51. Defina *subtipo* como este conceito é empregado por POOL-I.

52. Por que a definição de subtipo de POOL-I é mais abrangente do que a de subclasse de C++? Cite um exemplo.

53. Faça um programa correto em POOL-I que esteja incorreto em C++.

54. Dada a classe no modelo Smalltalk

```
class Store {
    private n;
    public put( i ) {
        n = i;
    }
    public get() { return n; }
}
```

É necessário transformá-la em classe genérica? Se sim, diga como. Se não, explique.

55. Compare os modelos de linguagens Smalltalk, POOL-I, C++. Qual(is) deles oferece mais reaproveitamento de *software*? Qual(is) oferece mais segurança contra erros de tipos?

56. Faça um programa que resultaria em um erro de compilação (erro de tipo) em POOL-I. O programa equivalente em Smalltalk deve causar erro em execução.

57. Faça um programa que resultaria em um erro de compilação (erro de tipo) em POOL-I. O programa equivalente em Smalltalk **não** deve causar erro em execução.

58. Esta questão utiliza as seguintes classes:

```
class A {
    private int k;
    public void put( int pk ) {
        k = pk;
    }
    public void print() {
        print(k);
    }
}

class B {
    private A a;
    public void put( int k ) {
        a.put(k);
    }
    public void print() {
        print(0);
        a.print();
    }
    public void set( A pa ) {
        a = pa;
    }
}
```

Baseado no código acima, escreva quais os métodos serão executados pelo código em POOL-I abaixo.

```
A a;
B b;

a = new A();
b = new B();
b.set(a);
b.put(12);
b.print();
a.print();
a.put(5);
a = b;
a.print();
```

Este código estaria correto se a linguagem fosse C++? Explique.

59. Utilizando as classes do exercício anterior, o que aconteceria se a linha
`b.set(a)`
fosse substituída por
`b.set(b)`
?

60. Que desvantagens possui o sistema de tipos de Java em relação ao de POOL-I? E que vantagens possui o sistema de tipos de Java em relação a C++?

61. Considerando-se apenas o sistema de tipos, os modelos estudados, pode-se transformar facilmente qualquer programa em Java em C++? E o contrário? Justifique.

62. Considerando-se apenas o sistema de tipos, os modelos estudados, pode-se transformar facilmente qualquer programa em Java em POOL-I? E o contrário? Justifique.

63. Considerando-se apenas o sistema de tipos, os modelos estudados, pode-se transformar facilmente qualquer programa em Smalltalk em POOL-I? E o contrário? Justifique.

64. O que acontece na instrução "`(new C()).r()`"?

```
class A {
    public void p() {
        this.r();
    }
    public void r() { }
}
class B extends A {
    public void m() {
        p();
    }
}
```

```
class C extends B {
    public void r() {
        m();
    }
}
```

65. Defina instanciação de uma classe genérica. Quando ela ocorre? O código dos métodos de uma classe genérica em Java é duplicada em cada instanciação? E o código de um protótipo genérico em Cyan?

66. Faça uma classe parametrizada `Vetor` que tenha pelo menos os métodos
```
T at( int i)
void put( T elem, int i )
```

67. Seria possível, conceitualmente, existir uma linguagem que aceita código semelhante ao dado abaixo? Neste caso, a superclasse `A` poderia acessar métodos da subclasse.

```
class A<T> {
    ...
}

class B extends A<B> {
    ...
}
```

68. Uma vaca é um tipo de animal. Um animal possui um método `void come(Comida c)`. Vacas comem grama, que é um tipo de comida. Este raciocínio está certo?

69. Em Java, `Object []` é supertipo de `String []`. Demonstre que isto pode causar um erro de tipos.

```
Object []objArray;
String []strArray = { "Eu", "sou", "um", "erro" };
objArray = strArray;
objArray[3] = Integer new(0);
System.out.println( ???? )
```

70. Dê um exemplo de uma classe abstrata. Porque é conceitualmente errado criar objetos de uma classe abstrata? Pode-se ter uma classe abstrata genérica? Pode-se ter um construtor em uma classe abstrata? Se sim, quando ele seria chamado?

71. Cite as restrições que se aplicam a classes abstratas.

72. Faça um macro "assert" que pode ser utilizado da seguinte forma:
```
assert i < max;
```

Se o valor da expressão, que deve ser booleana, for falso, então o programa deve ser terminado (isto pode ser feito, em Cyan, por `System exit`).

73. Faça uma DSL para pedidos de pizzas. Implemente-a em sua linguagem favorita. E em Java. E usando métodos de gramática de Cyan.

74. Faça uma DSL para especificar perguntas e respostas, estas em forma de alternativas. Devem existir entre duas e quatro alternativas, sendo uma delas a correta. Implemente esta DSL usando métodos de gramática de Cyan.

75. Quais as vantagens de se usar LEDs? E as desvantagens?

76. Quais as vantagens de LED´s internas quando comparadas com as internas?

77. Quais as vantagens de LED´s externas quando comparadas com as internas?

78. Mostre como implementar uma pequena LED em Java.

Capítulo 4

Linguagens Funcionais

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:

```
void p (int a, b) {
   int i, j, k;

   i = 1;
   j = a*b;
   ...
   while ( k < j && j < b ) {
       if ( a > i + j )
           j = j + 1;
       else
           k = a + b;
       ...
   }
   a = k - a;
   ...
}
```

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` + Δ`t`. 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.[1] 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.

```
    // fatorial imperativo
int fat(int n) {
    int i, p;
```

[1] Assuma isto. Há várias definições do que é "linguagem declarativa".

```
    p = 1;
    for (i = 1; i <= n; ++i)
        p = i*p;
    return p;
}

  // fatorial funcional
int fat(int n) {
    if ( n == 0 )
        return 1;
    else
        return n*fat(n-1);
}
```

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 chaves, { e } em Java/C/C#), 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:

```
int comb(int n, i) {
    return fat(n) / ( fat(n-i)*fat(i) );
}
```

Os operadores aritméticos (`*`, `+`, `/`, `-`, etc) também são funções no sentido funcional do termo. Portanto eles podem ser chamados com a notação usual:

```
int comb(int n, i) {
    return  /( fat(n), *(fat(n-i), fat(i)) );
}
```

Variáveis podem ser inicializadas uma única vez no início de uma função com o comando `let`:

```
String personData( Person p ) {
    let name = p.firstName + p.lastName;
    return name + " " + p.age;
}
```

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 nas quais 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.

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.

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. Em uma linguagem funcional absolutamente pura não há diferença entre a chamada da função e o comando `if` acima. Contudo, em linguagens reais, que não são tão puras, há uma diferença: a chamada da função `if` causa a avaliação de todas as expressões que são parâmetros. Já o comando `if` só avalia a expressão `expr1` se `exp` for `true`. E `expr2` se `exp` for `false`.

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

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

Utilizaremos esta sintaxe no restante deste capítulo. O corpo da função é colocado após `=` 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)) )
```

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.

4.1 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)
```

O código abaixo 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.

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

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` retornará `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 equivalente à função `membro` acima em Lisp é

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

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.

4.2 A Linguagem FP — Opcional

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∘SQR):x`

2. Construção.
   `[f1, f2, ... fn]:x` $\equiv$ `< f1:x, ..., fn: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`).

4.3 Haskell e SML

SML e Haskell são linguagens funcionais puras estaticamente tipadas 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 destas linguagens, mas com uma sintaxe semelhante à que usamos nos capítulos anteriores.[2] No texto a seguir citaremos a linguagem SML mas conceitos semelhantes se aplicam a Haskell também.

Além de tipos básicos (`int`, `boolean`, `String` etc.), SML suporta listas de forma semelhante a LISP. Uma lista com os três primeiros números inteiros positivos 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:

```
succ(n) = 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 `int`, `n` deve ser `int` e o resultado também.

[2] Nas próximas versões pretendemos utilizar a sintaxe real de Haskell.

O compilador produz o seguinte cabeçalho para `succ`:

```
int succ(int n)
```

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

```
int ⟼ int
```

O tipo de uma função

```
R f( T1 x1, T2 x2, ...Tn xn)
```

é expresso como

```
T1×T2× ...Tn ⟼ 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)
```
calcule(a, b) =
    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`. 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 `calcule` é:

```
int × int ⟼ int
```

(b)
```
inutil(a, b) =
    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:

```
int × int ⟼ int
```

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

```
soma(a, b) = a + b
```

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

```
int × int ⟼ int
float × float ⟼ float
```

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

```
soma(int a, b) = a + b
int soma(a, b) = a + b
```

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.

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

g = 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/Haskell.

A função

```
id(x) = 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

```
first(x, y) = x
```

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

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

O tipo de `escolhe` é:
`int` $\times$ `'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/Haskell utiliza não apenas definições estáticas mas também dinâmicas (algoritmos)*. Isto 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/Haskell 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]`	`int list`
`["a", "azul", "b"]`	`String list`
`[ [1, 2], [3], [4] ]`	`int 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

```
int list ins( int a, int list L ) = 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`:

```
int len( [] ) = 0
    len( h::t ) = 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. A Em uma chamada

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

é feito o emparelhamento[3] 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 (*pattern matching*, guarde este nome) 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:

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

[3] *Pattern matching* em Inglês.

E o seu tipo é
'a × 'a list ⟼ boolean

SML emprega o sistema de inferência de tipos Hindley-Milner que também é empregado em Haskell, Objective Caml e Clean.

4.4 Listas Infinitas e Avaliação Preguiçosa

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

```
ones = 1 : ones
```

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

```
numsFrom( int n ) = [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 [20]

```
boolean cmpTree( tree1, tree2 ) = cmpLists( treeToList(tree1), treeToList(tree2) );
```

compara duas árvores pela comparação dos nós das árvores colocados em forma de lista. Assuma que 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.

4.5 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:

```
max( v, n, gt ) {
    var maior, i;

    maior = v[0];
    for (i = 1; i < n; ++i) {
        if ( gt(v[i], maior) )
            maior = v[i];
    }
    return maior;
}
```

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:

```
gt_real(float a, float b)  {      // para numeros reais
    return a > b;
}
...
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:

```
map( func 'b f('a), []   ) = []
map( func 'b f('a), h::t ) = f(h)::map(f, t)
```

seu tipo é:
`('a` $\longmapsto$ `'b)` $\times$ `'a list` $\longmapsto$ `'b list`. Usamos `func` para indicar que dado parâmetro é uma função.

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.

4.6 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 chaves, { e }.

```
{
    a = a + 1;
    if ( a > b )
        return f(a, b);
    else
        return f(b, a);
}
```

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 próximo 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.

Contudo, muitas linguagens funcionais atualmente suportam um conceito chamado de *Monadas* que permite comandos com efeitos colaterais em programas de linguagens puramente funcionais. Este conceito não será estudado neste curso.

O segundo problema com linguagens funcionais é a eficiência. Elas são atualmente mais 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[4] pelo compilador ou o programador pode encontrar maneiras alternativas de expressar o algoritmo. Neste último caso, 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. É possível e provável que linguagens puramente funcionais sejam mais eficientes do que linguagens imperativas quando computadores com centenas de núcleos estiverem disponíveis.

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.

4.7 Exercícios

79. Cite alguns motivos pelos quais linguagens funcionais são de mais alto nível do que as linguagens imperativas (em geral).

80. Explique porque a atribuição implica na introdução do fator "tempo" nos programas tornando o seu entendimento difícil.

81. Por quê eliminando a atribuição devemos eliminar também os comandos `while`, `for` e semelhantes ?

[4] Este tópico não será discutido aqui.

82. Qual a diferença entre inicialização e atribuição ?

83. Por quê uma linguagem funcional pura não pode ter a construção "seqüência de comandos" ? Uma seqüência de comandos é colocada entre `begin` e `end` na linguagem S. Não considere a existência de comandos de entrada e saída na sua resposta.

84. Explique porque em uma linguagem funcional pura não podem existir variáveis globais e passagem por referência.

85. Por quê o comando de atribuição torna as linguagens imperativas de mais baixo nível ?

86. A comparação

```
f() == f()
```

resultará sempre em `true` na linguagem S (C, Pascal, Fortran, etc) ? E em uma linguagem puramente funcional ?

87. O que é transparência referencial ?

88. Como a transparência referencial ajuda na legibilidade dos programas ?

89. Aplique a regra da reescrita sobre `comb(1,0)` usando a função

```
int comb( int n, int i ) = fat(n)/(fat(n-i)*fat(i))
```

em que `fat` é a função fatorial.

90. A regra da reescrita é válida em C, Java, C#, C++ ou alguma outra linguagem imperativa que você conheça ? Por quê ?

91. Explique polimorfismo em LISP, citando um exemplo como auxílio.

92. Dê um exemplo de um erro de execução em Lisp. A linguagem é segura ?

93. Que estrutura de dados são utilizadas para representar programas em Lisp ? E dados ? Pode-se gerar uma função em tempo de execução e executá-la ?

94. Qual é o gargalo das máquinas Von Neumann ?

95. Seria possível colocar tipos em Lisp ? Dê um exemplo de um programa correto (que não produz erros de tipo) em Lisp que seria incorreto em Lisp com tipos. Faça você mesmo a sintaxe da linguagem "Lisp tipada".

96. Seria possível colocar o comando de atribuição em uma linguagem funcional e ainda assim garantir que as funções não produzam efeitos colaterais ? Defina tudo o que for necessário para garantir este objetivo.

97. Compare Lisp com Smalltalk com relação ao polimorfismo.

98. Faça um comando `switch` de C para uma linguagem funcional.

99. Podemos fazer um `if` para uma linguagem funcional onde o `else` é opcional ?

100. Compare polimorfismo de SML/Hashkell com classes parametrizadas.

101. Cite um exemplo onde o compilador de SML/Haskell não conseguiria deduzir os tipos de uma expressão.

102. Cite um exemplo em SML/Haskell de uma função totalmente polimórfica (que aceita parâmetros de infinitos tipos).

103. Que tipos um compilador de SML/Haskell colocaria para as funções abaixo ?

```
f( a, b, c )  =
    if a > 1
    then
        c
    else
        if a == b
        then
            true
        else
            false

g(a, b, c) =
    if a > b
    then
        c
    else
        if b > 5
        then
            a
        else
            c
```

104. Alterando o código de uma função em SML/Haskell podemos alterar o seu tipo e tornar inválido um código que chama esta função ? Cite um exemplo.

105. Que tipo o compilador de SML/Haskell colocaria para a função abaixo ?

```
f( h::t ) = h::h::f(t)
f( [] ) = []
```

O que ela faz ?

106. O que é avaliação preguiçosa ?

107. Cite um exemplo onde uma função é passada como parâmetro a outra função.

108. Por quê programadores são mais produtivos se utilizam linguagens de alto nível ?

109. Cite algumas vantagens de linguagens funcionais sobre linguagens imperativas.

110. Cite as desvantagens de linguagens funcionais.

111. Que característica de linguagens funcionais torna difícil o mapeamento de problemas do mundo real em programas ?

112. Usando-se a palavra-chave `let` pode-se ter atribuições em Haskell?

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 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* ou nomes de fatos ou regras.

Quando estiverem dentro de um fato ou regra, 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 (chamada de **instanciação**)

```
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* (como neste caso), 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. Assuma que não há instruções anteriores a esta.

(d) `Y = maria, mulher(Y) = mulher(X)`
há emparelhamento e `Y = maria` e `X = maria` pois `X` não estava instanciada.

(e) `mulher(Y) = mulher(X), Y = maria`
há emparelhamento e `Y = maria` e `X = maria`. Como nem `X` nem `Y` estavam instanciados, ambos ficam ligados após `mulher(Y) = mulher(X)`. Se uma das variáveis é instanciada, a outra automaticamente o é. Então o resultado seria o mesmo se tivéssemos
`mulher(Y) = mulher(X), mulher(Z) = mulher(Y), X = maria`
Neste caso `Z` também seria instanciada com `maria`.

(f) `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).
```

Para que `irmao(pedro, paulo)` seja considerado verdadeiro pelo sistema de tempo de execução de Prolog, todos os *subgoals*, separados por vírgulas, devem ser verdadeiros. A vírgula funciona como um "e" lógico.

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 é o fato seguinte a `homem(jose)`, 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.
```

Usamos "`_`" se o nome da variável não é importante.

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
```

Quando utilizado com pelo menos uma variável (do lado esquerdo e/ou direito), o símbolo `=` possui três funções:

(a) instanciar uma variável como em

```
?- X = 2, 0 = Y.
```

ambas as variáveis são inicializadas;

(b) associar uma variável a outra se ambas não estão inicializadas:

```
?- X = Y, X = 2.
```

Neste caso, ambas as variáveis estarão associadas ao valor `2` ao final do *goal*;

(c) comparar duas variáveis se as duas estão instanciadas:

```
?- X = 2, Y = 1, X = Y.
```

"`X = Y`" é uma comparação de variáveis. Este é o caso da comparação de uma variável instanciada com um valor:

```
?- X = 2, X = 2.
```

No primeiro `X = 2` a variável `X` é instanciada com `2`. No segundo ela é comparada com `2`.

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
```

Pois `5 + 1` é a estrutura `+(5, 1)`.

```
?- 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 [20] 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, [], []).
del(X, [X|L], L).
del(X, [Z|Big], [Z | 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 [22] 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). Frequentemente 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:

```
?- favorita( cyan ).
no

?- assert( favorita(cyan) ).
yes

?- favorita(cyan).
yes
```

O predicado `retract` remove um fato ou regra da base de dados:

```
?- retract( favorita(cyan) ).
yes

?- favorita(cyan).
no

?- assert( favorita(groovy) ).
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 [21], 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 Java (sem tipos) é:

```
void membro( p, x ) {
    if ( p == null )
        return false;
    else if ( p.elem == x )
        return true;
    else
        return membro( p.suc, x );
}
```

Esta função possui muito mais detalhes que a de Prolog. Contudo, o predicado em Prolog possui operações equivalentes àquelas desta função. 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 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.

5.8 Exercícios

113. Dada a base de dados

```
pA(a).
pA(b).
pB(X):- pA(X), X = b.
```

Quais as respostas (todas) dadas pelo goal abaixo ?

```
?- pB(Y).
```

114. Defina e dê um exemplo de instanciação.

115. Defina e dê um exemplo de retrocesso (*backtracking*).

116. Para as tentativas de emparelhamento abaixo, descubra se há emparelhamento (*matching*) e qual a instanciação final das variáveis.

- `pA([1, 2])` ≡ `pB([1, 2])`
- `map([1, 2, 3])` ≡ `map([H::T])`
- `pE([])` ≡ `pE([H::T])`
- `map([joao, maria])` ≡ `map([X, Y])`
- `map([joao, maria])` ≡ `map([X, Y | T])`
- `pE([1])` ≡ `pE([H::T])`
- `composto(agua, F, oleo, G)` ≡ `composto(A, farinha, oleo, H)`

117. Admita que o predicado `writeln` seja sempre verdadeiro e escreva o seu argumento no vídeo. O que imprime o goal
`?- pC(X).`
considerando a base de dados abaixo ?

```
pA(a):- writeln('A#a').
pA(b):- writeln('A#b').
pA(c):- writeln('A#c').
pB(b):- writeln('B#b').
pB(c):- writeln('B#c').
pC(X):- pA(X), pB(X).
pC(X):- pB(X),
```

118. Faça um programa em Prolog e mostre quais são as suas regras, fatos, cláusulas e predicados. O que é a base de dados ?

119. Dado um conjunto de fatos representados pela estrutura

```
movel( Nome, Fabricante, Peso,
       Altura, Comprimento,
       Tipo )
```

podemos criar operações como:

```
nome_movel( Nome,
   movel(Nome, _, _, _, _, _)
         )
```

O predicado acima retorna, em seu primeiro argumento, o nome do móvel. Um programa que usa apenas estas operações, sem manipular as estruturas `movel` diretamente, é independente da organização desta estrutura. Identifique esta técnica dentre os conceitos já vistos no curso.

120. O que imprime o goal
`?- pE(X).`
se é usada a base de dados abaixo ?

```
pA(a):- writeln('A#a').
pA(b):- writeln('A#b').
pA(c):- writeln('A#c').
pB(b):- writeln('B#b').
pB(c):- writeln('B#c').
pB(d):- writeln('B#d').
pC(c):- writeln('C#c').
pC(d):- writeln('C#d').
pE(X):= pA(X), pB(X), !, pC
```

121. Explique como o *cut* pode aumentar a eficiência de um programa.

122. Faça um pequeno predicado onde a introdução de um *cut* causa a modificação da semântica do predicado. Isto é, pelo menos uma consulta a ela resulta em resposta diferente da anterior (sem *cut*).

123. Por que um programa em Prolog não é formado apenas pela base de dados ?

124. Faça um programa em Prolog em que todo o algoritmo utilizado está implícito na linguagem.

125. O que acontece se um programa em Prolog utiliza algoritmos muitíssimo diferentes do algoritmo de unificação ?

126. Cite um exemplo de erro em execução em Prolog.

127. Cite um exemplo de um erro de tipos em Prolog que não é notado em compilação ou execução. Isto é, o programa está errado, o erro não é identificado pelo sistema mas poderia ser se Prolog fosse estaticamente tipado.

128. Faça um predicado polimórfico em Prolog.

129. Qual a diferença entre *cuts* vermelhos e verdes ?

130. Como funcionam os predicados `assert` e `retract` ?

131. Dada a base de dados

```
fibo(1, 1).
fibo(2, 1).
fibonacci(N, S) :-
    fibo(N, S).
fibonacci(N, S) :-
    N1 is N - 1,
    N2 is N - 2,
    fibonacci(N1, S1),
    fibonacci(N2, S2),
    S is S1 + S2,
    assert( fibo(
```

O que escreve o goal

```
?- fibonacc(5, S), list(fibo)
```

em que `list` é um predicado que sempre sucede e imprime todas as cláusulas do predicado que é seu argumento (parâmetro) ?

132. Cite os motivos que fazem com que Prolog não seja uma linguagem completamente lógica.

133. Faça um predicado em que a ordem em que as cláusulas de um predicado estão textuamente ordenadas é importante.

134. Faça um predicado em que a ordem dos *goals* no corpo de um predicado é importante.

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
    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 6.1.

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
...
```

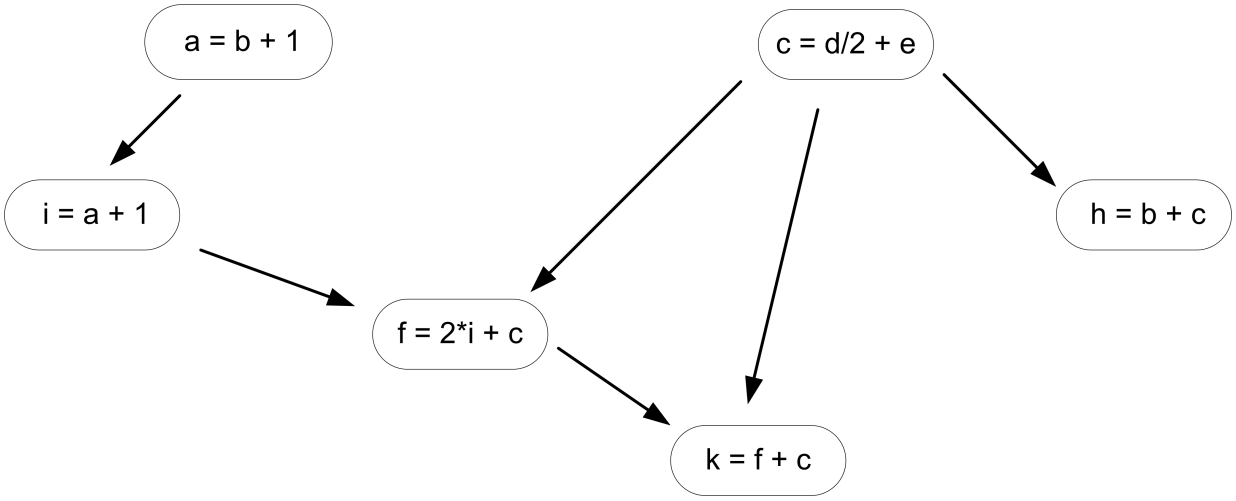


Figura 6.1: Um grafo do fluxo de dados

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.

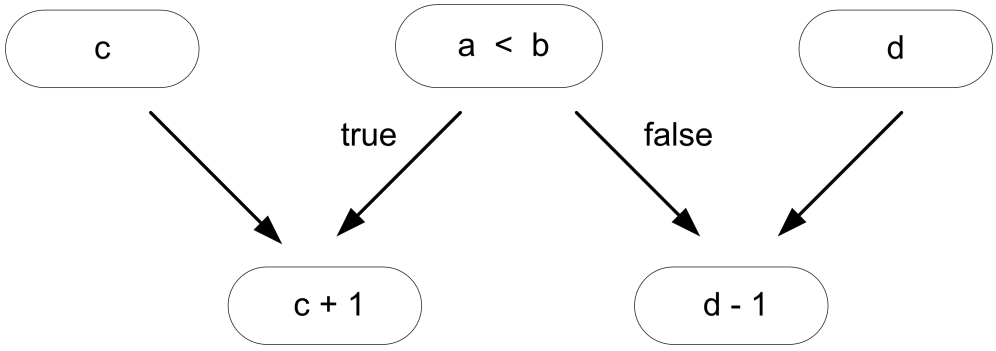


Figura 6.2: Um grafo do fluxo de dados de um comando `if`

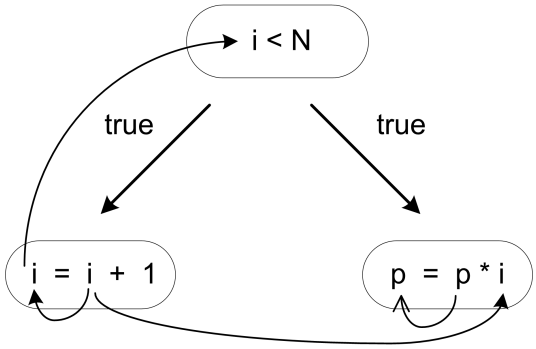


Figura 6.3: Um grafo do fluxo de dados de um comando `while`

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 6.2.

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 6.3.

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.

6.1 Exercícios

135. Monte o gráfico de fluxo de dados para os trechos de código abaixo.

- a)

```
a = b + 1;
if a > 2 then c = b; endif
b = 3;
d = a + b;
```

- b)

```
i = 1;
s = 0;
while i <= N do
    s = s + i;
```

- c)

```
a = b;
b = a;
c = d + 1;
d = a + b;
```

136. Cite duas instruções do exercício anterior que podem e duas que não podem ser executadas em paralelo. Explique.

137. Explique o que é regra de atribuição única. Cite um exemplo que demonstra que ela é necessária.

138. Explique como podemos ter laços `for` e `while` em linguagens data-flow e ainda ter atribuições dentro destes laços. Podemos executar diversas instruções de diversos passos do laço em paralelo ? A regra da atribuição única não é quebrada ? Os valores de uma variável em diversos passos do laço não se confundem ?

139. Qual é o objetivo das linguagens *data flow* ? Como e quando as instruções de um programa são executadas ?

140. Linguagens *Data-Flow* possuem granularidade de paralelismo muito alta — muitas pequenas instruções executando em paralelo. O gerenciamento deste paralelismo é complexo para ser eficiente.

Portanto, uma alternativa para aumentar a eficiência deste tipo de linguagem é diminuir a granularidade. Por exemplo, poderíamos permitir paralelismo apenas entre subrotinas, nunca dentro delas. Desenvolva esta idéia. Cite exemplos.

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] Guimarães, José de Oliveira. The Cyan Language. Disponível em `www.cyan-lang.org`.

[11] Kjell, Bradley. Introduction to Computer Science using Java. Available at http://chortle.ccsu.edu/CS151/cs151java.html.

[12] Opdyke, William. *Refactoring object-oriented frameworks*. PhD Thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.

[13] Niemeyer, P. and Peck, J. (1997) *Exploring Java*. O'Reilly & Associates, Sebastopol.

[14] Roberts, D. Brant, J. Johnson R. A Refactoring Tool for Smalltalk. Theory and Practice of Object Systems - Special issue object-oriented software evolution and re-engineering archive. Volume 3, Issue 4, 1997, Pages 253 - 263.

[15] 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.

[16] Slater, Robert. Portraits in Silicon. MIT Press, 1989.

[17] Stroustrup, Bjarne. *The C++ Programming Language*. Second Edition, Addison-Wesley, 1991.

[18] Wegner, Peter. *Research Directions in Object-Oriented Programming*, chapter The Object-Oriented Classification Paradigm, pp. 479–559. MIT Press, 1987.

[19] Weinberg, Gerald M. *The Psychology of Computer Programming*, Van Nostrand Reinhold, 1971.

[20] Ben-Ari, M. *Understanding Programming Languages*. John Wiley & Sons, 1996.

[21] Bratko, Ivan. *Prolog Programming for Artificial Intelligence*. International Computer Science Series, 1986.

[22] Finkel, Raphael. *Advanced Programming Language Design*. Addison-Wesley, 1996.