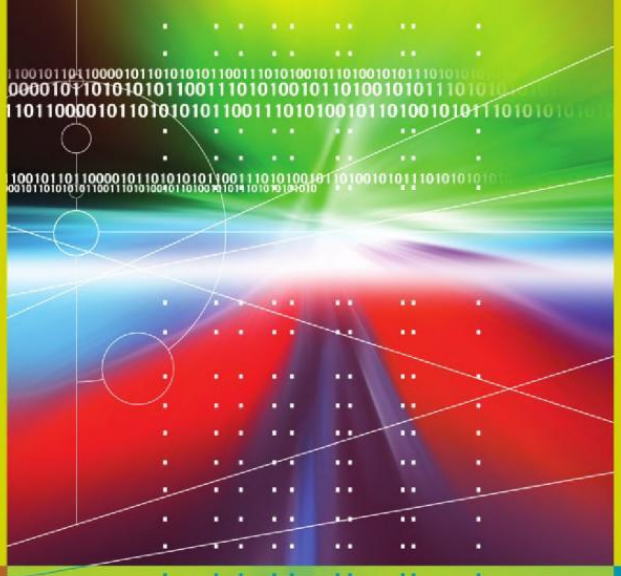


Gilles Dowek

UNDERGRADUATE TOPICS
in COMPUTER SCIENCE

Principles of Programming Languages

Springer

UTiCS

Gilles Dowek

Principles of Programming Languages

Springer

Gilles Dowek
École Polytechnique
France

Undergraduate Topics in Computer Science ISSN 1863-7310
ISBN: 978-1-84882-031-9 e-ISBN: 978-1-84882-032-6
DOI: 10.1007/978-1-84882-032-6

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2008943965

Based on course notes by Gilles Dowek published in 2006 by L'Ecole Polytechnique with the following title: "Les principes des langages de programmation."

Printed on acid-free paper

Springer Science+Business Media
springer.com

1 Imperative Core

1.1 Five Constructs

Most programming languages have, among others, five constructs: assignment, variable declaration, sequence, test, and loop. These constructs form the *imperative core* of the language.

1.1.1 Assignment

The *assignment* construct allows the creation of a statement with a variable `x` and an expression `t`. In Java, this statement is written as `x = t;`. *Variables* are identifiers which are written as one of more letters. *Expressions* are composed of variables and constants with operators, such as `+`, `-`, `*`, `/` — division — and `%` — modulo.

Therefore, the following statements

```
x = y % 3;
```

```
x = y;
```

```
y = 3;
```

```
x = x + 1;
```

G. Dowek, *Principles of Programming Languages*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-1-84882-032-6_1,

are all proper Java statements, while

```
y + 3 = x;
```

```
x + 2 = y + 5;
```

are not.

To understand what happens when you execute the statement `x = t;` suppose that within the recesses of your computer's memory, there is a compartment labelled `x`. Executing the statement `x = t;` consists of filling this compartment with the *value* of the expression `t`. The value previously contained in compartment `x` is erased. If the expression `t` is a constant, for example `3`, its value is the same constant. If it is an expression with no variables, such as `3 + 4`, its value is obtained by carrying out mathematical operations, in this case, addition. If expression `t` contains variables, the values of these variables must be looked up in the computer's memory. The whole of the contents of the computer's memory is called a *state*.

Let us consider, initially, that expressions, such as `x + 3`, and statements, such as `y = x + 3;`, form two disjoint categories. Later, however, we shall be brought to revise this premise.

In these examples, the values of expressions are integers. Computers can only store integers within a finite interval. In Java, integers must be between -2^{31} and $2^{31} - 1$, so there are 2^{32} possible values. When a mathematical operation produces a value outside of this interval, the result is kept within the interval by taking its modulo 2^{32} remainder. Thus, by adding `1` to $2^{31} - 1$, that is to say `2147483647`, we leave the interval and then return to it by removing 2^{32}, which gives -2^{31} or `-2147483648`.

Exercise 1.1

What is the value of the variable `x` after executing the following statement?

```
x = 2 * 1500000000;
```

In Caml, assignment is written `x := t`. *In the expression* `t`, *we designate the value of* `x`, *not with the expression* `x` *itself, but with the expression* `!x`. *Thus, in Caml we write* `y := !x + 1` *while in Java we write* `y = x + 1;`.

In C, assignment is written as it is in Java.

1.1.2 Variable Declaration

Before being able to assign values to a variable `x`, it must be declared, which associates the name `x` to a location in the computer's memory.

Variable declaration is a construct that allows the creation of a statement composed of a variable, an expression, and a statement. In Java, this statement is written `{int x = t; p}` where `p` is a statement, for example `{int x = 4; x = x + 1;}`. The variable `x` can then be used in the statement `p`, which is called the *scope* of variable `x`.

It is also possible to declare a variable without giving it an initial value, for example, `{int x; x = y + 4;}`. We must of course be careful not to use a variable which has been declared without an initial value and that has not been assigned a value. This produces an error.

Apart from the `int` type, Java has three other integer types that have different intervals. These types are defined in Table 1.1. When a mathematical operation produces a value outside of these intervals, the result is returned to the interval by taking its remainder, modulo the size of the interval.

In Java, there are also other *scalar types* for decimal numbers, booleans, and characters. These types are defined in Table 1.1. Operations allowed in the construction of expressions for each of these types are described in Table 1.2.

Variables can also contain objects that are of *composite types*, like arrays and character strings, which we will address later. Because we will need them shortly, character strings are described briefly in Table 1.3.

The integers are of type `byte`, `short`, `int` or `long` corresponding to the intervals $[-2^7, 2^7 - 1]$, $[-2^{15}, 2^{15} - 1]$, $[-2^{31}, 2^{31} - 1]$ and $[-2^{63}, 2^{63} - 1]$, Respectively. Constants are written in base 10, for example, `-666`.

Decimal numbers are of type `float` or `double`. Constants are written in scientific notation, for example `3.14159`, `666` or `6.02E23`.

Booleans are of type `boolean`. Constants are written as `false` and `true`.

Characters are of type `char`. Constants are written between apostrophes, for example `'b'`.

Table 1.1 Scalars types in Java

To declare a variable of type `T`, replace the type `int` with `T`. The general form of a declaration is thus `{T x = t; p}`.

The basic operations that allow for arithmetical expressions are `+`, `-`, `*`, `/` — division — and `%` — modulo.

When one of the numbers `a` or `b` is negative, the number `a / b` is the quotient rounded towards `0`. So the result of `a / b` is the quotient of the absolute values of `a` and `b`, and is positive when `a` and `b` have the same sign, and negative if they have different signs. The number `a % b` is `a - b * (a / b)`. So `(-29) / 4` equals `-7` and `(-29) % 4` equals `-1`.

The operations for decimal numbers are `+`, `-`, `*`, `/`, along with some transcendental functions: `Math.sin`, `Math.cos`, ...

The operations allowed in boolean expressions are `==`, `!=` — different —, `<`, `>`, `<=`, `>=`, `&` — and —, `&&`, `|` — or —, `||` and `!` — not.

For all data types, the expression `(b) ? t : u` evaluates to the value of `t` if the boolean expression `b` has the value `true`, and evaluates to the value of `u` if the boolean expression `b` has the value `false`.

Table 1.2 Expressions in Java

Character strings are of type `String`. Constants are written inside quotation marks, for example `"Principles of Programming Languages"`.

Table 1.3 Character strings in Java

In Caml, variable declaration is written as `let x = ref t in p` *and it isn't necessary to explicitly declare the variable's type. It is not possible in Caml to declare a variable without giving it an initial value.*

In C, like in Java, declaration is written `{T x = t; p}`. *It is possible to declare a variable without giving it an initial value, and in this case, it could have any value.*

In Java and in C, it is impossible to declare the same variable twice, and the following program is not valid.

```
int y = 4;
int x = 5;
int x = 6;
y = x;
```

In contrast, nothing in Caml stops you from writing

```
let y = ref 4
in let x = ref 5
in let x = ref 6
in y := !x
```

and this program assigns the value `6` *to the variable* `y`*, so it is the most recent declaration of* `x` *that is used. We say that the first declaration of* `x` *is* hidden *by the second.*

Java, Caml and C allow the creation of variables with an initial value that can never be changed. This type of variable is called a *constant* variable. A variable that is not constant is called a *mutable* variable. Java assumes that all variables are mutable unless you specify otherwise. To declare a constant variable in Java, you precede the variable type with the keyword `final`, for example

```
final int x = 4;
y = x + 1;
```

The following statement is not valid, because an attempt is made to alter the value of a constant variable

```
final int x = 4;
x = 5;
```

In Caml, to indicate that the variable `x` *is a constant variable, write* `let x = t in p` *instead of writing* `let x = ref t in p`*. When using constant variables, you do not write* `!x` *to express its value, but simply* `x`*. So, you can write* `let x = 4 in y := x + 1`*, while the statement* `let x = 4 in x := 5` *is invalid. In C, you indicate that a variable is a constant variable by preceding its type with the keyword* `const`*.*

1.1.3 Sequence

A *sequence* is a construct that allows a single statement to be created out of two statements p_1 and p_2. In Java, a sequence is written as $\{p_1\ p_2\}$. The statement $\{p_1\ \{p_2\ \{\ \ldots\ p_n\}\ \ldots\}\}$ can also be written as $\{p_1\ p_2\ \ldots\ p_n\}$.

To execute the statement $\{p_1\ p_2\}$ in the state `s`, the statement p_1 is first executed in the state `s`, which produces a new state `s'`. Then the statement p_2 is executed in the state `s'`.

In Caml, a sequence is written as p_1`;` p_2*. In C, it is written the same as it is in Java.*

Exercise 1.3

Write a Java program that reads an integer `n` from the keyboard, and outputs a boolean indicating whether the number is prime or not.

Graphical constructs that allow drawings to be displayed are fairly complex in Java. But, the class `Ppl` contains some simple constructions to produce graphics. The statement `Ppl.initDrawing(s,x,y,w,h);` creates a window with the title `s`, of width `w` and of height `h`, positioned on the screen at coordinates `(x,y)`. The statement `Ppl.drawLine(x1,y1,x2,y2);` draws a line segment with endpoints `(x1,y1)` and `(x2,y2)`. The statement `Ppl.drawCircle(x,y,r);` draws a circle with centre `(x,y)` and with radius `r`. The statement `Ppl.paintCircle(x,y,r);` draws a filled circle and the statement `Ppl.eraseCircle(x,y,r);` allows you to erase it.

1.3 The Semantics of the Imperative Core

We can, as we have below, express in English what happens when a statement is executed. While this is possible for the simple examples in this chapter, such explanations quickly become complicated and imprecise. Therefore, we shall introduce a theoretical framework that might seem a bit too comprehensive at first, but its usefulness will become clear shortly.

1.3.1 The Concept of a State

We define an infinite set `Var` whose elements are called *variables*. We also define the set `Val` of *values* which are integers, booleans, etc. A *state* is a function that associates elements of a finite subset of `Var` to elements of the set `Val`.

For example, the state `[x = 5, y = 6]` associates the value `5` to the variable `x` and the value `6` to the variable `y`. On the set of states, we define an *update* function `+` such that the state `s + (x = v)` is identical to the state `s`, except for the variable `x`, which now becomes associated with the value `v`. This operation is always defined, whether `x` is originally in the domain of `s` or not.

We can then simply define a function called Θ, which for each pair `(t,s)` composed of an expression `t` and a state `s`, produces the value of this expression in this state. For example, Θ`(x + 3,[x = 5, y = 6]) = 8`.

This is a partial function, because a state is a function with a finite domain while the set of variables is infinite. For example, the expression `z + 3` has no

value in the state `[x = 5, y = 6]`. In practice, this means that attempting to compute the value of the expression `z + 3` in the state `[x = 5, y = 6]` produces an error.

Executing a statement within a state produces another state, and we define what happens when a statement is executed using a function called Σ. Σ has a statement `p`, an initial state `s` and produces a new state, Σ`(p,s)`. This is also a partial function. Σ`(p,s)` is undefined when executing the statement `p` in the state `s` produces an error or does not terminate.

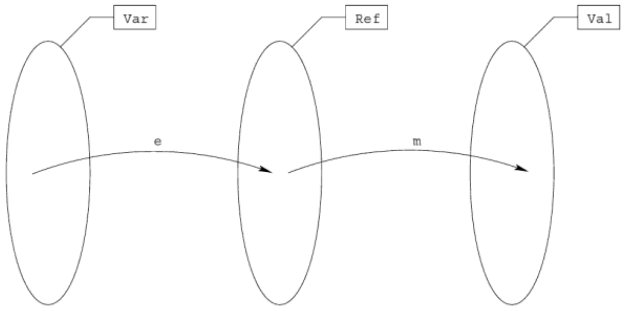
In the case of a statement `p` having the form `x = t;`, the Σ function is defined as follows

$$\Sigma\texttt{(x = t;,s) = s + (x = }\Theta\texttt{(t,s))}.$$

For example, Σ`(x = x + 1;,[x = 5]) = [x = 6]`. This is equivalent to saying 'Executing the statement `x = t;` loads the memory location `x` with the value of expression `t`'.

1.3.2 Decomposition of the State

A state `s` is a function that maps a finite subset of `Var` to the set `Val`. It will be helpful for the next chapter if we decompose this function as the composition of two other functions of finite domains: the first is known as the *environment*, which maps a finite subset of the set `Var` to an intermediate set `Ref`, whose elements are called *references* and the second, is called the *memory state*, which maps a finite subset of the set `Ref` to the set `Val`.

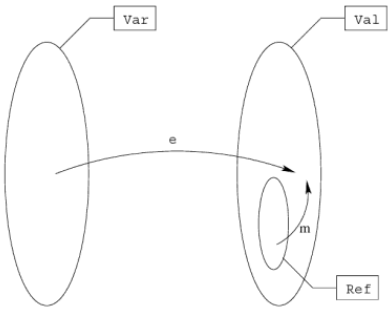


This brings us to propose two infinite sets, `Var` and `Ref`, and a set `Val` of values. The set of *environments* is defined as the set of functions that map a finite subset of the set `Var` to the set `Ref`. The set of *memory states* is defined as the set of functions mapping a finite subset of the set `Ref` to the set `Val`. For the set of environments, we define an update function `+` such that the environment `e + (x = r)` is identical to `e`, except at `x`, which now becomes associated with

the reference `r`. For the set of memory states, we define an update function `+` such that the memory state `m + (r = v)` is identical to `m`, except at `r`, which now becomes associated with the value `v`.

However, constant variables complicate things a little bit. For one, the environment must keep track of which variables are constant and which are mutable. So, we define an environment to be a function mapping a finite subset of the set `Var` to the set `{constant, mutable}` $\times$ `Ref`. We will, however, continue to write `e(x)` to mean the reference associated to `x` in the environment `e`.

Then, at the point of execution of the declaration of a constant variable `x`, we directly associate the variable to a value in the environment, instead of associating it to a reference which is then associated to a value in the memory state. The idea is that the memory state contains information that can be modified by an assignment, while the environment contains information that cannot. To avoid having a target set for the environment function that is overly complicated, we propose that `Ref` is a subset of `Val`, which brings us to propose that the environment is a function that maps a finite subset of `Var` to `{constant, mutable}` $\times$ `Val` and the memory state is a function that maps a finite subset of `Ref` to `Val`.



1.3.3 A Visual Representation of a State

It can be helpful to visualise states with a diagram. Each reference is represented with a box. Two boxes placed in different positions always refer to separate references.

Then, we represent the environment by adding one or more labels to certain references.