

2

Functions

2.1 The Concept of Functions

2.1.1 Avoiding Repetition

```
System.out.print("Flight ");
System.out.print("819");
System.out.print(" to ");
System.out.print("Tokyo");
System.out.print(" takes off at ");
System.out.println("8:50 AM");
System.out.println();
System.out.println();
System.out.println();
```

```
System.out.print("Flight ");
System.out.print("211");
System.out.print(" to ");
System.out.print("New York");
System.out.print(" takes off at ");
System.out.println("8:55 AM");
System.out.println();
System.out.println();
System.out.println();
```

In this program, the block of three statements `System.out.println();`, which skips three lines, is repeated twice. Instead of repeating it there, you can define a *function* `jumpThreeLines`

```
static void jumpThreeLines () {  
    System.out.println();  
    System.out.println();  
    System.out.println();}
```

And use it in the main program

```
System.out.print("Flight ");  
System.out.print("819");  
System.out.print(" to ");  
System.out.print("Tokyo");  
System.out.print(" takes off at ");  
System.out.println("8:50 AM");  
jumpThreeLines();
```

```
System.out.print("Flight ");  
System.out.print("211");  
System.out.print(" to ");  
System.out.print("New York");  
System.out.print(" takes off at ");  
System.out.println("8:55 AM");  
jumpThreeLines();
```

The statement `jumpThreeLines();` that is found in the main program is named the *call* of the function `jumpThreeLines`. The statement that is found in the function and that is executed on each call is named the *body* of the function.

Organising a program into functions allows you to avoid repeated code, or redundancy. As well, it makes programs clearer and easier to read: to understand the program above, it isn't necessary to understand how the function `jumpThreeLines();` is implemented; you only need to understand what it does. This also allows you to organise the structure of your program. You can choose to write the function `jumpThreeLines();` one day, and the main program another day. You can also organise a programming team, where one programmer writes the function `jumpThreeLines();`, and another writes the main program.

This mechanism is similar to that of mathematical definitions that allows you to use the word 'group' instead of always having to say 'A set closed under an associative operation with an identity, and where every element has an inverse'.

2.1.2 Arguments

Some programming languages, like assembly and Basic, have only a simple function mechanism, like the one above. But the example above demonstrates that this mechanism isn't sufficient for eliminating redundancy, as the main program is composed of two nearly identical segments. It would be nice to place these segments into a function. But to deal with the difference between these two copies, we must introduce three parameters: one for the flight number, one for the destination and one for the take off time. We can now define the function `takeOff`

```
static void takeOff
    (final String n, final String d, final String t) {
System.out.print("Flight ");
System.out.print(n);
System.out.print(" to ");
System.out.print(d);
System.out.print(" takes off at ");
System.out.print(t);
System.out.println();
System.out.println();
System.out.println();}
```

and to use it in the main program, we write

```
takeOff("819", "Tokyo", "8:50 AM");
takeOff("211", "New York", "8:55 AM");
```

The variables `n`, `d` and `t` which are listed as arguments in the function's definition, are called *formal arguments* of the function. When we call the function `takeOff("819", "Tokyo", "8:50 AM");` the expressions `"819"`, `"Tokyo"` and `"8:50 AM"` that are given as arguments are called the *real arguments* of the call.

A formal argument, like any variable, can be declared constant or mutable. If it is constant, it cannot be altered inside the body of the function.

To follow up the comparison, mathematical language also uses parameters in definitions: 'The group $\mathbb{Z}/n\mathbb{Z}$ is ...', 'A K -vector space is ...', ...

In Caml, a function declaration is written `let f x y ... = t in p`.

```
let takeOff n d t =
  print_string "Flight ";
  print_string n;
  print_string " to ";
  print_string d;
```

```

    print_string " takes off at ";
    print_string t;
    print_newline ();
    print_newline ();
    print_newline ()
in takeOff "819" "Tokyo" "8:50 AM";
    takeOff "211" "New York" "8:55 AM"

```

Formal arguments are always constant variables. However, if the argument itself is a reference, you can assign to it, just like to any other reference.

In C, a function declaration is written as in Java, but without the keyword `static`.

2.1.3 Return Values

```

a = 3;
b = 4;
c = 5;
d = 12;
u = Math.sqrt(a * a + b * b);
v = Math.sqrt(c * c + d * d);

```

In this program, we want to isolate the computation `Math.sqrt(x * x + y * y)` in a function called `hypotenuse`. But in contrast to the function `takeOff` that performs output, the `hypotenuse` function must compute a value and send it back to the main program. This return value is the inverse of argument passing that sends values from the main program to the body of the function. The type of the returned value is written before the name of the function. The function `hypotenuse`, for example, is declared as follows.

```

static double hypotenuse (final double x, final double y) {
    return Math.sqrt(x * x + y * y);}

```

And the main program is written as follows.

```

a = 3;
b = 4;
c = 5;
d = 12;
u = hypotenuse(a,b);
v = hypotenuse(c,d);

```

In Caml, the function `hypotenuse` is written

```
let hypotenuse x y = sqrt(x *. x +. y *. y)
```

In C, the function `hypotenuse` is written as in Java, but without the keyword `static` and using C's square root function which is written as `sqrt` instead of `Math.sqrt`.

2.1.4 The return Construct

As we have seen, in Caml, the function `hypotenuse` is written

```
let hypotenuse x y = sqrt(x *. x +. y *. y)
```

In Java and in C, in contrast, you must precede the return value with the keyword `return`. So, in Java, instead of writing

```
static double hypotenuse (final double x, final double y) {  
    Math.sqrt(x * x + y * y);};
```

you should write

```
static double hypotenuse (final double x, final double y) {  
    return Math.sqrt(x * x + y * y);};
```

When `return` occurs in the middle of the function instead of the end, it stops the execution of the function. So, instead of writing

```
static int sign (final int x) {  
    if (x < 0) return -1;  
    else if (x == 0) return 0;  
    else return 1;}
```

you can write

```
static int sign (final int x) {  
    if (x < 0) return -1;  
    if (x == 0) return 0;  
    return 1;}
```

Basically, if the value of `x` is negative, the statement `return -1;` interrupts the execution of the function, and the other two statements will not be executed.

Exercise 2.1

In Java, write a function that takes an integer argument called `n` and returns the integer 2^n .

Exercise 2.2

In Java, write a function that takes an integer argument called `n` and returns a boolean that indicates whether `n` is prime or not.

2.1.5 Functions and Procedures

A function can on one hand cause an action to be performed, such as outputting a value or altering memory, and on the other hand can return a value. Functions that do not return a value are called *procedures*.

In some languages, like Pascal, procedures are differentiated from functions using a special keyword. In Caml, a procedure is simply a function that returns a value of type `unit`. Like its name implies, `unit` is a singleton type that contains only one value, written `()`. In Caml, a procedure always returns the value `()`, which communicates no information.

Java and C lie somewhere in the middle, because we declare a procedure in these languages by replacing the return type by the keyword `void`. In contrast to the type `unit` of Caml, there is no actual type `void` in Java and C. For example, you cannot declare a variable of type `void`.

A function call, such as `hypotenuse(a,b)`, is an expression, while a procedure call, such as `takeOff("819", "Tokyo", "8:50 AM");`, is a statement.

There are however certain nuances to consider, because a function call can also be a statement. You can, for example, write the statement `hypotenuse(a,b);`. The value returned by the function is simply discarded. However, even if a language allows it, using functions in this way is considered to be bad form. The Caml compilers, for example, will produce a warning in this case.

In Java and in C, a procedure, that is to say a function with return type of `void` cannot be used as an expression. For example, to write

```
x = takeOff("819", "Tokyo", "8:50 AM");
```

the variable `x` would have to be of the type `void` and we have seen that there is no such variable. In Caml, in contrast, a procedure is nothing but a function with a return type `unit` and you can easily write

```
x := takeOff("819", "Tokyo", "8:50 AM")
```

if the variable `x` is of type `unit ref`. However, if such an assignment is possible, it is not useful.

In general, no matter what the language, it is considered good form to separate functions and procedures. Functions return a value, and do not perform actions such as outputting a value, and are used as expressions. Procedures do not return a value, can perform actions, and are used as statements.

2.1.6 Global Variables

Imagine that we would like to isolate the statement `x = 0;` with a function in the program

```
int x;  
x = 3;  
x = 0;
```

We then would write the function

```
static void reset () {x = 0;}
```

and the main program

```
int x;  
x = 3;  
reset();
```

But this program is not correct, as the statement `x = 0;` is no longer in the scope of variable `x`. For the function `reset` to have access to the variable `x`, you must declare a variable `x` as a *global* variable, and the access to this variable is given to all the functions as well as to the main program

```
static int x;
```

```
static void reset () {x = 0;}
```

and the main program

```
x = 3;  
reset();
```

All functions can use any global variable, whether they are declared before or after the function.

2.1.7 The Main Program

A *program* is composed of three main sections: global variable declarations x_1 , ..., x_n , function declarations f_1 , ..., f_n , and the *main program* p which is a statement.

A program can thus be written as

```
static T1 x1 = t1;  
...
```

```

static Tn xn = tn;

static ... f1 (...) ...
...
static ... fn' (...) ...

```

p

However, in Java, the main program is placed inside a special function called: **main**. The **main** function must not return a value, and must always have an argument of type **String []**. In addition to the keyword **static**, the definition of the **main** function must also be preceded by the keyword **public**.

In addition, the program must be given a name, which is given with the keyword **class**. The general form of a program is:

```

class Prog {

    static T1 x1 = t1;
    ...
    static Tn xn = tn;

    static ... f1 (...) ...
    ...
    static ... fn' (...) ...

    public static void main (String [] args) {p}}

```

For example

```

class Hypotenuse {

    static double hypotenuse (final double x, final double y) {
        return Math.sqrt(x * x + y * y);}

    public static void main (String [] args) {
        System.out.println(hypotenuse(3,4));}}

```

In Caml, there is no main function and the syntax of the language separates functions from the main program

```

let hypotenuse x y = sqrt(x *. x +. y *. y)
in print_float(hypotenuse 3.0 4.0)

```


In C, the main program is also a function called `main`. For historical reasons, the `main` function must always return an integer, and is usually terminated with `return 0`;. You don't give a name to the program itself, so a program is simply a series of global variable and function declarations.

```
double hypotenuse (const double x, const double y) {  
    return sqrt(x * x + y * y);}

int main () {  
    printf("%f\n",hypotenuse(3,4));  
    return 0;}
```

2.1.8 Global Variables Hidden by Local Variables

```
class Prog {

    static int n;

    static int f (final int x) {
        int p = 5;
        return n + p + x;}

    static int g (final int x) {
        int n = 5;
        return n + n + x;}

    public static void main (String [] args) {
        n = 4;
        System.out.println(f(6));
        System.out.println(g(6));}}
```

The value of the expression `f(6)` is 15. The function `f` adds the global variable `n`, which has been initialised to 4 in the main program, the local variable `p`, with a value of 5, and the argument `x`, with a value of 6.

In contrast, the value of the expression `g(6)` is 16, because both occurrences of `n` refer to the local variable `n`, which has a value of 5. In the environment in which the body of function `g` is executed, the global variable `n` is hidden by the local variable `n` and is no longer accessible.

2.1.9 Overloading

In Java, it is impossible to define two functions with the same name, for example

```
static int f (final int x) {
    return x;}

```

```
static int f (final int x) {
    return x + 1;}

```

except when the number or types of their arguments are different. You can, for example, declare three identically named functions

```
static int f (final int x) {
    return x;}

```

```
static int f (final int x, final int y) {
    return x + 1;}

```

```
static int f (final boolean x) {
    return 7;}

```

At the time of evaluation of an expression of the form $f(t_1, \dots, t_n)$, the called function is chosen based on its name as well as the number and types of its arguments. The expressions $f(4)$, $f(4,2)$, and $f(\text{true})$ evaluate to 4, 5, and 7 respectively. In this case, we say that the name f is *overloaded*.

There is no overloading in Caml. The programs

```
let f x = x in let f x = x + 1 in print_int (f 4)

```

and

```
let f x = x in let f x y = x + 1 in print_int (f 4 2)

```

are valid, but the first declaration is simply hidden by the second.

There is also no overloading in C, and the program

```
int f (const int x) {return x;}
int f (const int x, const int y) {return x + 1;}
...

```

is invalid.

2.2 The Semantics of Functions

This brings us to extend the definition of the Σ function. In addition to a statement, an environment, and a memory state, the Σ function now also takes an argument called the *global environment* G . This global environment comprises an environment called e that contains global variables and a function of a finite domain that associates each function name with its definition, that is to say with its formal arguments and the body of the function to be executed at each call.

We must then take into account the fact that, because functions can modify memory, the evaluation of an expression can now modify memory as well. Because of this fact, the result of the evaluation of an expression, when it exists, is no longer simply a value, but an ordered pair composed of a value and a memory state.

Also, we must explain what happens when the statement **return** is executed, in particular the fact that the execution of this statement interrupts the execution of the body of the function.

This brings us to reconsider the definition of the function Σ in the case of the sequence

$$\Sigma(\{p_1 \ p_2\}, e, m, G) = \Sigma(p_2, e, \Sigma(p_1, e, m, G), G)$$

according to which executing the sequence $\{p_1 \ p_2\}$ consists of executing p_1 and then p_2 .

Indeed, if p_1 is of the form **return** t ;, or more generally if the execution of p_1 causes the execution of **return**, then the statement p_2 will not be executed. We will therefore consider that the result $\Sigma(p_1, e, m, G)$ of the execution of p_1 in the state e, m is not simply a memory state, but a more complex object. One part of this object is a boolean value that indicates if the execution of p_1 has occurred normally, or if a **return** statement was encountered. If the execution occurred normally, the second part of this object is the memory state produced by this execution. If the statement **return** was encountered, the second part of this object is composed of the return value and the memory state produced by the execution. From now on, the target set of the Σ function will be $(\{\text{normal}\} \times \text{Mem}) \cup (\{\text{return}\} \times \text{Val} \times \text{Mem})$ where Mem is the set of memory states, that is to say the set of functions that map a finite subset of Ref to the set Val .

Finally, we should also take into account the fact that a function cannot only be called from the main program — the **main** function — but also from inside another function. However, we will discuss this topic later.

2.2.1 The Value of Expressions

The evaluation function of an expression is now defined as

- $\Theta(x, e, m, G) = (m(e(x)), m)$, if x is a mutable variable in e ,
- $\Theta(x, e, m, G) = (e(x), m)$, if x is a constant variable in e ,
- $\Theta(c, e, m, G) = (c, m)$, if c is a constant,
- $\Theta(t \otimes u, e, m, G) = (v \otimes w, m')$ where \otimes is an arithmetical or logical operation, $(v, m') = \Theta(t, e, m, G)$ and $(w, m') = \Theta(u, e, m, G)$,
- if $\Theta(b, e, m, G) = (\text{true}, m')$ then

$$\Theta((b) ? t : u, e, m, G) = \Theta(t, e, m', G),$$

if $\Theta(b, e, m, G) = (\text{false}, m')$ then

$$\Theta((b) ? t : u, e, m, G) = \Theta(u, e, m', G).$$

- $\Theta(f(t_1, \dots, t_n), e, m, G)$ is defined this way.

Let x_1, \dots, x_n be the list of formal arguments and p the body of the function associated with the name f in G . Let e' be the environment of global variables of G . Let $(v_1, m_1) = \Theta(t_1, e, m, G)$, $(v_2, m_2) = \Theta(t_2, e, m_1, G)$, ..., $(v_n, m_n) = \Theta(t_n, e, m_{n-1}, G)$ be the result of the evaluation of real arguments t_1, \dots, t_n of the function.

For the formal mutable arguments x_i , we consider arbitrary distinct references r_i that do not appear either in e' or in m_n . We define the environment $e'' = e' + (x_1 = v_1) + (x_2 = r_2) + \dots + (x_n = r_n)$ in which we associate the formal argument x_i to the value v_i or to the reference r_i according to whether it is constant or mutable, and the memory state $m'' = m_n + (r_2 = v_2) + \dots + (r_n = v_n)$ in which we associate to the values v_i the references r_i associated to formal mutable arguments.

Consider the object $\Sigma(p, e'', m'', G)$ obtained by executing the body of the function in the state formed by the environment e'' and the memory state m'' . If this object is of the form (return, v, m''') then we let

$$\Theta(f(t_1, \dots, t_n), e, m, G) = (v, m''').$$

Otherwise, the function Θ is not defined: the evaluation of the expression produces an error because the evaluation of the body of the function has not encountered a **return** statement.

2.2.2 Execution of Statements

We now define what occurs when a statement is executed.

- When the statement p is a declaration of the form $\{\text{T } x = t; p\}$ or $\{\text{final } T x = t; p\}$, if $\Theta(t, e, m, G) = (v, m')$ then

$$\Sigma(\{\text{T } x = t; p\}, e, m, G) = \Sigma(p, e + (x = r), m' + (r = v), G)$$

where r is an arbitrary reference that does not appear in e and m , and

$$\Sigma(\{\text{final } T x = t; p\}, e, m, G) = \Sigma(p, e + (x = v), m', G).$$

- When the statement p is an assignment of the form $x = t;$, if $\Theta(t, e, m, G) = (v, m')$ then

$$\Sigma(x = t;, e, m, G) = (\text{normal}, m' + (e(x) = v)).$$

- When the statement p is a sequence of the form $\{p_1 \ p_2\}$, if $\Sigma(p_1, e, m, G) = (\text{normal}, m')$ then

$$\Sigma(\{p_1 \ p_2\}, e, m, G) = \Sigma(p_2, e, m', G)$$

and if $\Sigma(p_1, e, m, G) = (\text{return}, v, m')$ then

$$\Sigma(\{p_1 \ p_2\}, e, m, G) = (\text{return}, v, m').$$

- When the statement p is a test of the form $\text{if } (b) \ p_1 \ \text{else } p_2$, if $\Theta(b, e, m, G) = (\text{true}, m')$ then

$$\Sigma(\text{if } (b) \ p_1 \ \text{else } p_2, e, m, G) = \Sigma(p_1, e, m', G)$$

and if $\Theta(b, e, m, G) = (\text{false}, m')$ then

$$\Sigma(\text{if } (b) \ p_1 \ \text{else } p_2, e, m, G) = \Sigma(p_2, e, m', G).$$

- The definition for loops is unchanged

$$\Sigma(\text{while } (b) \ q, e, m, G) = \lim_n \Sigma(p_n, e, m, G)$$

where

$p_0 = \text{if } (b) \ \text{giveup}; \ \text{else skip};$

and $p_{n+1} = \text{if } (b) \ \{q \ p_n\} \ \text{else skip};.$

- When the statement p is of the form `return t;`, if $\Theta(t, e, m, G) = (v, m')$ then

$$\Sigma(\text{return } t; , e, m, G) = (\text{return}, v, m').$$

- Finally, we add the case of functions, which is very similar to the case of functions in the definition of the evaluation of expressions, except that if the object $\Sigma(p, e'', m'', G)$ has the form (normal, m'') , then we let

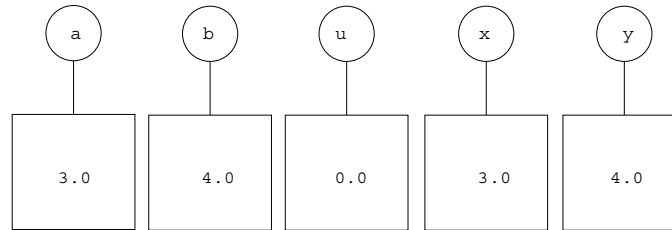
$$\Sigma(f(t_1, \dots, t_n); , e, m, G) = (\text{normal}, m'')$$

and if it has the form (return, v, m'') , we let

$$\Sigma(f(t_1, \dots, t_n); , e, m, G) = (\text{normal}, m'')$$

by ignoring the value v : we have the case where a function is used as a statement.

For example, when we execute the statement `u = hypotenuse(a,b);` in the environment $e = [a = r_1, b = r_2, u = r_3]$, the memory state $m = [r_1 = 3.0, r_2 = 4.0, r_3 = 0.0]$, and the global environment G composed of the environment e and the function declaration `hypotenuse: (x,y), return Math.sqrt(x * x + y * y);`, we start by evaluating the expression `hypotenuse(a,b)`. To do so, we start by evaluating a and b , which produces the values 3.0 and 4.0, without changing the memory state. And we create an environment $e'' = [a = r_1, b = r_2, u = r_3, x = r_4, y = r_5]$ and the memory state $m'' = [r_1 = 3.0, r_2 = 4.0, r_3 = 0.0, r_4 = 3.0, r_5 = 4.0]$



Next, we execute the body of the function, which produces the result $(\text{return}, 5.0, m'')$ and so $\Theta(\text{hypotenuse}(a,b), e, m, G)$ is $(5.0, m'')$. The result of the execution of the statement `u = hypotenuse(a,b);` is then an ordered pair composed of a boolean `normal` and the memory state $m''' = [r_1 = 3.0, r_2 = 4.0, r_3 = 5.0, r_4 = 3.0, r_5 = 4.0]$.

The value of the variable u in the state e, m''' is 5.0.

Exercise 2.3

What happens if the formal arguments `x` and `y` of the function `hypotenuse` are constant variables?

Exercise 2.4

What happens if you execute the statement `u = hypotenuse(a,b);`, with the variables `a`, `b`, and `u` declared in the `main` function?

Finally, we can give the definition of the Σ function for the entire program. Let P be a program formed of global variable declarations `static T1 a1 = t1;`, ..., `static Tn an = tn;` and of function declarations `static U1 f1 (x1) p1;`, ..., `static Un' fn' (xn') pn';`.

Let v_1, \dots, v_n be the initial values given to global variables, that is to say the values of expressions t_i . Let e be the environment $[a_1 = v_1, a_2 = r_2, \dots, a_n = r_n]$ in which we associate the global variable a_i to the value v_i or to the reference r_i whether it is constant or mutable, and m is the memory state $[r_2 = v_2, \dots, r_n = v_n]$, in which we associate the references r_i associated to mutable global variables with the values v_i . Let G be the global environment $(e, [f_1 = (x_1, p_1), \dots, f_{n'} = (x_{n'}, p_{n'})])$.

The memory state $\Sigma(P)$ is defined by

– $\Sigma(P) = \Sigma(\text{main}(\text{null}); e, m, G)$

where `null` is a value of type `String []` which we will discuss later.

Exercise 2.5

The function `f` is defined as follows

```
static int f (final int x) {
    int y = x;
    while (true) {
        y = y + 1;
        if (y == 1000) return y + 1;}}

```

What is returned from the function call `f(500)`?

Exercise 2.6

Imagine that all memory states contain two special references: `in` and `out`. Write the definition of the function Σ for the input and output constructs from Section 1.2.

2.2.3 Order of Evaluation

Since expressions can modify memory, consideration must be given to the fact that in the definition of Σ we have given, arguments of a function are evaluated from left to right. So, we evaluate t_1 in the memory state m , and t_2 in the memory state m_1 produced by the evaluation of t_1 , ... So, the program

```
class Prog {

    static int n;

    static int f (final int x, final int y) {
        return x;}

    static int g (final int z) {
        n = n + z;
        return n;}

    public static void main (String [] args) {
        n = 0;
        System.out.println(f(g(2),g(7)));}}
```

outputs the result 2.

2.2.4 Caml

The definition of the function Σ for Caml is somewhat different from the definition of Σ used for Java. In Caml, all formal arguments are constant variables, so new references are never created at the point of a function call.

Also, in Caml, there is only one name space for functions and variables. In Java, the program

```
class Prog {

    static int f (final int x) {
        return x + 1;}

    static int f = 4;

    public static void main (String [] args) {
        System.out.println(f(f));}}
```


is valid, and in the expression `f(f)`, the first occurrence of `f` is a function name `f` and the second occurrence of `f` is a variable name. In Caml, however, the program

```
let f x = x + 1 in let f = 4 in print_int(f f)
```

is invalid. The function `f` becomes hidden by the variable `f`. There is therefore no global environment: global variables and functions are declared in the environment, like variables. During the call of a function `f`, it is impossible to create the environment in which we must evaluate the body of the function using the global environment. Thus, in the environment, we must associate the name `f`, not only with the list of formal arguments and the body of the function, but also the environment to extend with the arguments for executing the body of the function. This environment is the environment in which the function is defined.

So, the Java program

```
class Prog {

    static int f () {return x;}

    static int x = 4;

    public static void main (String [] args) {
        System.out.println(f ());}}

```

is valid, and outputs 4, while the Caml program

```
let f () = x in let x = 4 in print_int(f())
```

is invalid, because the variable `x` in the body of `f` is not part of the environment of the definition of `f`. It is necessary to declare this variable before `f`

```
let x = 4 in let f () = x in print_int(f())
```

Another difference is that the Caml compilers evaluate the arguments from right to left. For example, the program

```
let n = ref 0
in let f x y = x
in let g z = (n := !n + z; !n)
in print_int (f (g 2) (g 7))
```

results in 9 and not 2.

However, the definition of the Caml language does not specify the order of evaluation of the arguments of a function. Different compilers may evaluate

arguments in a different order. It is up to the programmer to write programs whose result is not dependent on the order of evaluation.

Finally, there is no **return** in Caml, and the result of the execution of a statement, like the evaluation of an expression, is an ordered pair composed of a value, possibly `()`, and a memory state.

Exercise 2.7

Give the definition of the Σ function of Caml, assuming that arguments are always evaluated from right to left.

2.2.5 C

The definition of the Σ function for C is also somewhat different from the definition of Σ for Java.

In C, the references created at the moment of a function call are removed from the memory and the end of the execution of the body of the function.

Like in Caml, there is only one name space for functions and variables, and functions are declared in the same environment as variables. In this environment, we not only associate the name `f` to the list of formal arguments and the body of the function, but also to the environment `e` to extend with the arguments for executing the body of the function. This environment is, like in Caml, the environment of the definition of the function. For example, the program

```
int f () {return x;}
```

```
int x = 4;
```

```
int main () {
  printf("%d\n",f());
  return 0;}
```

is invalid.

C compilers also evaluate a function's arguments from left to right, as in Java. However, the definition of the language, like that of Caml, does not specify the order of evaluation of a function's arguments, and it is up to the programmer to write programs whose result does not depend on the order of evaluation.

Exercise 2.8

Give the definition of the Σ function for C, assuming that arguments are always evaluated from left to right.

2.3 Expressions as Statements

Now that we have defined the result of the evaluation of an expression as an ordered pair composed of a value and a state, we can better understand the link between expressions and statements.

In C, any expression followed by a semicolon is a statement. The value of an expression is simply ignored when it is used as a statement. If $\Theta(t, e, m, G)$ is the ordered pair (v, m') then $\Sigma(t; , e, m, G)$ is the ordered pair composed of the boolean **normal** and the memory state m' . The situation is somewhat similar in Java, except that only certain expressions are eligible to be used as statements. For example, if f is a function, then $f(t_1, \dots, t_n);$ is, as we have seen, a statement, but that is not the case with $1;$. In Caml, there is no difference between statements and expressions, since statements are simply expressions of type **unit**.

Exercise 2.9

In Java and in C, the expression $x = t$ assigns the value of t to x and returns this same value. How would you modify the definition of the Θ function to take into account expressions of this type? What is the output of the following program?

```
class Prog {  
  
    public static void main (String [] args) {  
        int x;  
        int y;  
        x = (y = 4);  
        System.out.println(x);  
        System.out.println(y);}}}
```

2.4 Passing Arguments by Value and Reference

If the initial value of the variable x is 4 and that of the variable y is 7, after executing the statement $\{z = x; x = y; y = z;\}$, variable x has the value 7 and variable y has the value 4. More generally, this statement exchanges the values of these variables, using the *principle of the third glass*



Observe the behaviour of the following program

```
class Prog {

    static int a;

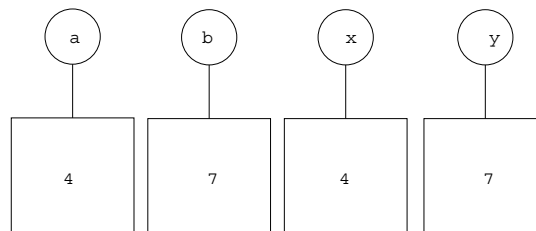
    static int b;

    static void swap (int x, int y) {int z; z = x; x = y; y = z;}

    static public void main (String [] args) {
        a = 4;
        b = 7;
        swap(a,b);
        System.out.println(a);
        System.out.println(b);}}
```

You might expect the values of **a** and **b** have been exchanged and that the numbers 7 and 4 are displayed, but surprisingly, the number 4 is displayed first, followed by the number 7.

In fact, this result is what is expected based on the definition of the Σ function given above. We start with an environment $e = [a = r_1, b = r_2]$ and a memory state $m = [r_1 = 4, r_2 = 7]$. The call of the function `swap(a,b)` computes the values of the expressions **a** and **b** in the environment **e** and the memory state **m**. It obtains 4 and 7 respectively. Then, the environment $[a = r_1, b = r_2, x = r_3, y = r_4]$ and the memory state $[r_1 = 4, r_2 = 7, r_3 = 4, r_4 = 7]$ are created.



The values of the variables `x` and `y` are exchanged, which results in the memory state $[r_1 = 4, r_2 = 7, r_3 = 7, r_4 = 4, r_5 = 4]$ which returns control to the main program. The environment is then $e = [a = r_1, b = r_2]$ with the memory state $[r_1 = 4, r_2 = 7, r_3 = 7, r_4 = 4, r_5 = 4]$. The values of the variables `a` and `b` have not changed.

In other words, the function `swap` ignores the variables `a` and `b`. It can only use their value at the moment of the function call, and cannot modify their value: executing the statement `swap(a,b)`; has the same result as executing the statement `swap(4,7)`;

The mechanism of argument passing that we have described is called *argument passing by value*. It does not allow the creation of a `swap` function that changes the contents of two variables. However, most programming languages have a construct that allows the creation of such a function. But, this construct is somewhat different in each language. Before seeing how this is done in Java, Caml, and C, we will look at the much simpler example of the Pascal language.

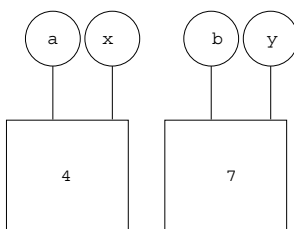
2.4.1 Pascal

The Pascal language has a built in calling mechanism to pass arguments by reference, or by variable. In the definition of the `swap` procedure, we can precede each argument with the keyword `var`.

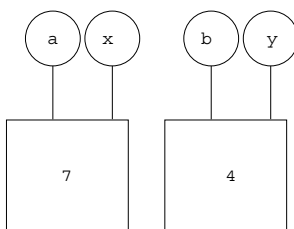
```
procedure swap (var x : integer, var y : integer) ...
```

*When an argument of a procedure or a function is declared using `pass by reference`, we can only apply this procedure or function to a variable. So, we can write `swap(a,b)` but not `swap(4,7)`, nor `swap(2 * a,b)`.*

When we call the procedure `swap(a,b)`, instead of associating the variables `x` and `y` to new references assigning to these references the values of the procedure's arguments, 4 and 7, we associate the variables `x` and `y` to references associated with variables given as arguments to the procedure. So, we call the procedure `swap(a,b)` in an environment $e = [a = r_1, b = r_2]$ and a memory state $m = [r_1 = 4, r_2 = 7]$, instead of creating the environment $[a = r_1, b = r_2, x = r_3, y = r_4]$ and the memory state $[r_1 = 4, r_2 = 7, r_3 = 4, r_4 = 7]$, we create the environment $[a = r_1, b = r_2, x = r_1, y = r_2]$ while keeping the memory state $[r_1 = 4, r_2 = 7]$.



Because of this, the procedure **swap** exchanges the contents of the references r_1 and r_2 and not of the references r_3 and r_4



and after execution of the procedure, the contents of the references associated with the variables **a** and **b** have been exchanged.

Being able to explain the mechanism of passing by reference is the main motivation for decomposing the state into an environment and a memory state by introducing an intermediate set of references, as we have done in the previous chapter.

Exercise 2.10

Give the definition of the Σ function in the case of functions with an argument passed by reference.

2.4.2 Caml

In Caml, passing by reference is not a primitive construct, but it can be accomplished by using the fact that references are also values.

For example, in the environment $[x = r]$ and in the memory state $[r = 4]$, the value of the expression $!x$ is the integer 4, but the value of the expression x is the reference r . This allows the creation of a function **swap** that takes two references as arguments and exchanges the values associated with these references in the memory.

```
let swap x y = let z = ref 0 in (z := !x; x := !y; y := !z)
```

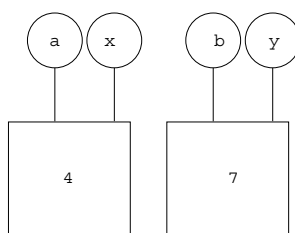
To exchange the values of the variables **a** and **b**, you simply apply the function to the references **a** and **b** and not to the integers **!a** and **!b**.

```

a := 4;
b := 7;
swap a b;
print_int !a;
print_newline();
print_int !b;
print_newline()

```

Indeed, when we call the function `swap a b` in the environment $[a = r_1, b = r_2]$ and the memory state $[r_1 = 4, r_2 = 7]$, we create the environment $[a = r_1, b = r_2, x = r_1, y = r_2]$ in which the constant formal arguments `x` and `y` are linked to the real arguments r_1 and r_2 and we keep the same memory state $[r_1 = 4, r_2 = 7]$



and the function `swap` exchanges the contents of the references r_1 and r_2 and after the execution of the function, the contents of the references associated with the variables `a` and `b` have now also been exchanged.

Exercise 2.11

What does the following program do?

```

let swap x y = let z = !x in (x := !y; y := z)

```

2.4.3 C

In *C* as well, the passing by reference is not a primitive construct, but it can be simulated by using a similar mechanism to that of Caml. The type of references that can be associated with a value of type T in memory, written `T ref` in Caml, is written `T*` in *C*. The dereference construct, written `!` in Caml, is written as `*` in *C*. For example, in the environment $[u = r_1]$ and in the memory state $[r_1 = r_2, r_2 = 4]$, the value of the expression `u` is the reference r_2 and the value of the expression `*u` is the integer 4.

If `x` is a variable, the reference associated with `x` in the environment, written simply as `x` in Caml, is written as `&x` in *C*. For example, in the environment

[$x = r$] and the memory state [$r = 4$] the value of expression x is the integer 4, the value of expression $\&x$ is the reference r and the value of expression $*\&x$ is the integer 4. The $\&$ construct applies to a variable and not to an arbitrary expression.

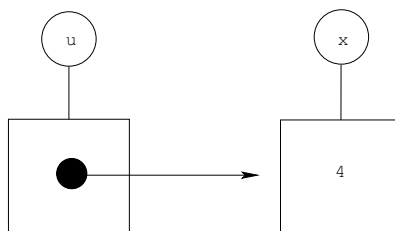
Exercise 2.12

What does the following program output?

```
int main () {
    int x;
    int* u;

    x = 4;
    u = &x;
    printf("%d\n", *u);
    return 0;}
```

Using these constructs, it becomes possible to create states in which some references are associated with other references. It then becomes necessary to update our graphical representation of states. When a memory state has a reference r' associated with a reference r , one solution is to write in the box of r the coordinates of the place where we have drawn the reference r' on the page. A better solution is to draw in the box of r an arrow that points to the reference r' .



If t is an expression of type T^* then the language C has a new assignment construct $*t = u;$, similar to the construct $:=$ of Caml: if the value of t is a reference r and the value of u is v , then the execution of the statement $*t = u;$ associates the value v to the reference r in memory.

Exercise 2.13

Show that the execution of the statement $x = u;$ has the same effect as executing the statement $*\&x = u.$ What can we conclude?

These constructs allow you to write a function `swap` that takes as arguments two references and exchanges the values associated with these references in memory.


```
void swap (int* const x, int* const y) {
    int z;
    z = *x;
    *x = *y;
    *y = z;}

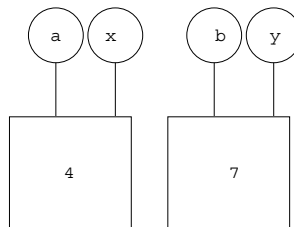
```

To exchange the values of the variables `a` and `b`, you can now apply this function to the references `&a` and `&b`.

```
int main () {
    a = 4;
    b = 7;
    swap(&a,&b);
    printf("%d\n",a);
    printf("%d\n",b);
    return 0;}

```

When we execute the statement `swap(&a,&b);` in the environment $e = [a = r_1, b = r_2]$ and the memory state $m = [r_1 = 4, r_2 = 7]$, we create the environment $e = [a = r_1, b = r_2, x = r_1, y = r_2]$ and the memory state $m = [r_1 = 4, r_2 = 7]$.



And, the function `swap` exchanges the contents of the references r_1 and r_2 and after the execution of the function, the contents of the references associated with the variables `a` and `b` have been exchanged.

In this example, take note of the syntax of the declaration of the argument `x`, `int* const x`, which prevents the assignment `x = t;` but allows the assignment `*x = t;`. The declaration `const int* x`, in contrast allows the assignment `x = t;` but prevents the assignment `*x = t;`. The declaration `const int* const x` prevents both types of assignment.

Exercise 2.14

What is the output of the following program?

```
void swap (int* x, int* y) {
    int z;
    z = *x;

```

```

*x = *y;
*y = z;}

int main () {
  a = 4;
  b = 7;
  swap(&a,&b);
  printf("%d\n",a);
  printf("%d\n",b);
  return 0;}

```

Draw the state in which the body of the function is executed.

Exercise 2.15

What does the following function do?

```

void swap1 (int* x, int* y) {
  int* z;
  z = x;
  x = y;
  y = z;}

```

Exercise 2.16

*Give the definition of the Θ function for expressions of the form $*t$ and $\&x$, and the definition for the Σ function for statements of the form $*t = u$;*

Exercise 2.17

The goal of this exercise is to demonstrate that, in C, you may look for a reference that does not exist.

- In the following Caml program, what is the state in which the statement `print_int !(u)` is executed?*

```

let f p = let n = ref p in let x = ref n in !x
in let u = ref (f 5)
in print_int !(u)

```

Answer the same question for the following program.

```

let f p = let n = ref p in let x = ref n in !x
in let u = ref (f 5)
in let v = ref (f 10)
in print_int !(u)

```

2. Given the following C program

```
int* f (const int p) {  
    int n = p;  
    int* x = &n;  
    return x;}  
  
int main () {  
    int* u = f(5);  
    printf("%d\n", *u);  
    return 0;}
```

In what state is the statement `printf("%d\n", *u);` executed?

Hint: remember that in C, in contrast to Caml, we remove from memory the reference associated with a variable when that variable is removed from the environment.

3. In C, when we use a reference that is not declared in memory, it does not produce an error, and the result will be unpredictable. Try compiling and running the following C program.

```
int* f (const int p) {  
    int n = p;  
    int* x = &n;  
    return x;}  
  
int main () {  
    int* u = f(5);  
    int* v = f(10);  
    printf("%d\n", *u);  
    return 0;}
```

2.4.4 Java

In Java, passing by reference is not a primitive construct, but it can be simulated by using a different mechanism called *wrapper types*. We will explain this later, as it uses language constructs that have yet to be introduced.



<http://www.springer.com/978-1-84882-031-9>

Principles of Programming Languages

Dowek, G.

2009, XII, 159 p., Softcover

ISBN: 978-1-84882-031-9