A Scheme for the GTK-server

Ed Mort

Contents

Chapter 1

Graphic User Interface

Scheme is a computer language that has a very cute feature: Its syntax is so simple that you can write programs that manipulate it. Therefore, you can create tools that help in the design and implementation of computer applications. However, before talking about programs that make programs, let us take a look at the building blocks of Scheme.

Let us assume that you have installed the Bigloo compiler in your machine. After the installation, if you type Bigloo at the command prompt, the Scheme interpreter will wait for an expression, and evaluate it. Here is what the Scheme interpreter looks like:

```
D:\>Bigloo
-------------------------------------------------------------------
Bigloo (2.7a)                                            ,--^,
'a practical Scheme compiler'                          _ ___/ /|/
Sun Nov 27 15:05:30 RST 2005                         ,;'( )__, ) '
Inria -- Sophia Antipolis                           ;;  //   L__.
email: bigloo@sophia.inria.fr                       '  \    /  '
url: http://www.inria.fr/mimosa/fp/Bigloo               ^    ^
-------------------------------------------------------------------

Welcome to the interpreter

1:=>
```

There are three building blocks in Scheme, to wit, applications, lambda expressions, and definitions. Applications is a list containing the name of

an operation, followed by the operands, that are also called arguments. Remember, operation and arguments are enclosed in parentheses. Let us assume that the operation that you want to apply is addition; the operands, i.e., the numbers you need to add are `(34 56 78)`. In this case, the application is

```
(+ 34 56 78)
```

Scheme will perform the desired operation on the arguments, and produce a value, that is the result of the operation.

```
1:=> (+ 34 56 78)
168
1:=>
```

The next step is to add products. There are two operations involved, multiplication and addition. Suppose that the products you want to add are $3 \times 45 \times 63$, 4×5 and 5×7. In Scheme, these three operations are represented by the applications below.

- `(* 3 45 63)`

- `(* 4 5)`

- `(* 5 7)`

In order to add these products together, you need to apply the additon operation to them:

```
1:=> (+ (* 3 45 63) (* 4 5) (* 5 7))
8560
```

Let us invert the situation; let us assume that now you want a product of additions, using the same numbers: $(3 + 45 + 63) \times (4 + 5) \times (5 + 7)$.

```
1:=> (* (+ 3 45 63) (+ 4 5) (+ 5 7))
11988
```

To provide an example with subtraction and division, let us calculate the expression $(3 \times 4 \times 56)/(723 - 212)$.

```
1:=> (/ (* 3 4 56) (- 723 212))
1.3150684931507
```

Now that you know all that there is to know about applications, let us learn about lambda expressions, and definitions, the remaining building blocks of Scheme. A lambda expression introduces variables to generalize applications. Let us assume that you want to create a formula that finds the area of a circle of radius r; the area is given by the expression πr^2. In order to introduce the variable r, we need the lambda expression below.

```
(lambda(r) (* 3.1416 r r))
```

The next step is to give a name to this lambda expression, i.e., to this application with a variable. The definition (that is the third building block of Scheme) takes care of naming things.

```
(define area (lambda(r) (* 3.1416 r r)))
```

After defining `area`, you can use it like any other Scheme operation. Here is how you can calculate circular areas for r= 45, r=8 and r=34:

```
1:=> (define area (lambda(r) (* 3.1416 r r)))
area
1:=> (area 45)
6361.74
1:=> (area 8)
201.0624
1:=> (area 34)
3631.6896
```

There is a syntactical sugar that consists of defining the operation by a pattern that mirrors its use in applications. For instance, here is the definition of `area`:

```
1:=> (define (area r) (* 3.1416 r r))
area
1:=> (area 45)
6361.74
```

The form *define* can be employed also to create constants and global variables. Suppose, for instance, that you want to define π.

```
1:=> (define pi 3.1416)
pi
1:=> pi
3.1416
```

After its definition, one can use *pi* to calculate the area of a circle of radius
r. Below you will see how to do it.

```
1:=> (define pi 3.1415)
pi
1:=> (define (area r) (* pi r r))
area
1:=> (area 3)
28.2735
```

Until now, you have been using an interpreter to calculate applications and
to define lambda expressions. The interpreter is nice, since it is interactive,
and offers an environment where you can build your ideas, and test them.
However, a compiler generates executable code, that is both faster, safer,
and easier to launch. To compile a program, you must provide a module to
indicate from where the computer must start the execution (see figure 1.1).

```
;; File name: c-area.scm
;; Compile:   bigloo c-area.scm -o area

(module circle (main start))

(define (start args)
   (display "Give me the radius: ")
   (flush-output-port (current-output-port))
   (print (area (read)) ) )

(define pi 3.1416)
(define (area r)  (* pi r r))
```

Figure 1.1: A module to calculate the area of a circle

The first two lines of figure 1.1 start with a semicolon. This means that
the foresaid lines are comments. The first line reminds us of the file name.
The second one shows how to compile the program. The module declaration

```
(module circle (main start))
```

introduces the module identifier *circle*, and informs us that execution will
begin in function *start*.

Let us examine the function *start*. Its sole argument is `args`, that contains a list of the command line elements. We can assume that the command line looks like the one given below.

```
area 30
```

In this case, `args` will match `'("area" "30")`. You will see more about that later on. The line

```
(display "Give me the radius: ")
```

will print its argument, that is a string. By the way, a string is a sequence of characters placed between double quotes. However, the computer may pospone printing to the end of the job. Since you want it to print the message `"Give me the radius: "` right away, you must issue the command

```
(flush-output-port (current-output-port))
```

Finally the application

```
(print (area (read)) )
```

reads a radius, calculates the corresponding area, and prints the result, issuing a carriage return at the same time. The reading is executed by the application (`read`), that will obtain the value of the radius from whatever you type on the console. You have noticed that Scheme has other data types, besides numbers; for instance, it has strings:

```
"Give me the radius: "
"main"
"30"
```

It also has lists. For instance, `'("area" "30")` is an example of a list. A few other examples of lists are given below.

- `'(3 45 6 76)`   ;; A list of integers

- `'( )`   ;; An empty list

- `'(rose pencil notebook)`   ;; A list of symbols

- `'("Anna" "Sue" "Margaret")` ;; A list of strings

Integers, real numbers, and fractions are arithmetic data types. There are operations for handling arithmetic data types, and they are amazingly few in number. In fact, you need only four operations $(+, *, /,$ and $-)$ to combine numbers in different ways; you may also need the following boolean operations:

>, =, *not*, **and**, **or**, `real?`, and `integer?`

That is about all that it is necessary for you to do whatever you please with arithmetic data types. There are also operations with lists, and they are amazingly few as well. You will need only four operations to deal with lists.

- `(car xs)` returns the first element of a list `xs`:

    `(car '(a b c)) → a`

- `(cdr xs)` returns the list `xs` with the first element removed:

    `(cdr '(a b c)) → (b c)`

- `(cons x xs)` builds a new list, whose *car* is `x` and whose *cdr* is `xs`:

    `(cons 'a '(b c d)) → (a b c d)`

- `(null? xs)` returns `#t` —true— if `xs` is empty; otherwise, returns `#f`.

    ```
    (define xs '(a b))
    (null? xs) → #f
    (null? (cdr (cdr xs)))) → #t
    ```

Let us go back to the program in figure 1.1. You have learned that the operand `args` contains a list with the elements of the command line. For example, if you type

    `area.exe 30`

the operand `args` will contain the list `'("area.exe"  "30")`. The second element of this list may be the radius of the circle. If this is the case, you can get the radius with `(car (cdr args))`. However, there is a problem here. What you obtain from this operation is the string `"30"` rather than the numeral 30. Happily enough, Scheme has a function to transform a string to a number: `(string->number (car (cdr args)))`

By the way, there is an abbreviation to express `(car(cdr x))`, to wit, `(cadr x)`. Using that abbreviation, you can obtain the radius of the circle

```
(string->number (cadr args))
```

Let us try a first approach to the calculation of the area. From the command line, if you type `area 30`, the program below will produce the area of a circle of radius 30.

```
;; File name: c-area.scm
;; Compile:   bigloo c-area.scm -o area

(module circle (main start))

(define (start args)
   (print (area (string->number (cadr args) ))))

(define pi 3.1416)

(define (area r) (* pi r r))
```

The problem with this program is that there is no room for mistakes. For instance, if you forget to provide a radius, it will try to find the second element of a single element list. Let us make sure that this does not happen by checking that `args` has at least two elements, the name of the program and the radius expressed as a string. A list with two elements has a non *null cdr*. Then, the program becomes:

```
;; File name: c-area.scm
;; Compile:   bigloo c-area.scm -o area

(module circle (main start))

(define (start args)
   (when (not (null? (cdr args)))
       (area (string->number (cadr args) ))))

(define pi 3.1416)
(define (area r) (* pi r r))
```

The application (`when...`) has a condition followed by a sequence of actions, that are realized if and only if the condition is met. There are replacements for (`when...`). For example, (`cond (condition actions...)...`) has a list of condition-actions, and performs the actions in conjunction with the first condition that returns true. We can use `cond` not only to deal with the empty (`cdr args`), but also with a non-numerical argument. In fact, (`string->number x`) returns `#f` (false) if the string `x` has no numerical equivalent. This leads to the following scheme of calculation:

```
;; File name: c-area.scm
;; Compile:   bigloo c-area.scm -o area

(module circle (main start))

(define (start args)
   (cond ( (null? (cdr args)) (print "No argument"))
         ( (string->number (cadr args))
           (print (area (string->number (cadr args)))))
         (else (print "Argument is not numerical"))))

(define pi 3.1416)

(define (area r) (* pi r r))
```

1.1   Graphic User Interface

In order to use the gtk-server, you must install a few tools into your machine. These tools are:

- The package GTK2. **Windows** — You will find a Windows installer in the same place where you got this tutorial. **Linux** has a native GTK installation.

- The GTK-server. From the server, you need two files, that will reside within the same directory as your code. In the case of Windows, these files are:

  ```
  gtk-server.dll
  gtk-server.cfg
  ```

If you are using Linux, you will need

```
gtk-server.so
```

instead of `gtk-server.dll`. Besides this, the file `gtk-server.cfg` is not the same in Linux as in Windows. Let us assume that you have installed everything into your machine. Now, you can try to write a simple program with a GUI.

```
(module test  (main start)
   (include "macro-def.sch")
   (import (gtkserver_gtk "gtk.scm")) )

(define (start argv)
  (gtk-init)
  (let [  (w (gtk-window-new 0))
          (hbx (gtk-hbox-new #t 0))
          (do-button (gtk-button-new-with-label "Hello"))
          (exit-button (gtk-button-new-with-label "Exit")) ]

     (gtk-box-pack-start hbx exit-button 1 1 0)
     (gtk-box-pack-start hbx do-button 1 1 0)
     (gtk-container-add  w hbx)
     (gtk-widget-show-all w)
     (do  ([event (gtk-server-callback) (gtk-server-callback)] )
          ([equal? event exit-button])
       (when (equal? event do-button)
             (print "Hello, world!")
             (flush-output-port (current-output-port)) )
      ); end do
   ); end let
); end define
```

If you examine this program, you will recognize many elements that we have already met. For instance, you know that the declaration

```
(main start)
```

makes `start` the entry point of your program. However, even in the module declaration there are things that you have not seen before. The first one is

```
(include "macro-def.sch")
```

that inserts into your code the contents of `"macro-def.sch"`. You can download the file `"macro-def.sch"` from the `GTK-server4Bigloo` page, and keep it in the same folder as your code. In the end, the Windows folder containing your program will have the following files:

```
your-code.scm
macro-def.sch
gtk-server.dll
gtk-server.cfg
gtk.scm
gtk.o
```

In the case of Linux, I suggest that you read the instructions that go with the bignux distribution. Remember, in Linux the `gtk-server.cfg` is different from the one in Windows. Using the Windows `gtk-server.cfg` file in Linux is a very common mistake made by people who work on both platforms.

In the module declaration, there is another weird line, that I will reproduce below.

```
(import (gtkserver_gtk "gtk.scm"))
```

I will not go into details of what it means. Put it there, and that is all. This leaves us with the *let* form. A *let* form has the following structure:

```
(let [ (x (function-that-calculates-x))      ⎫
       (y (function-that-calculates-y))      ⎬ ;;Local variables
       (z (function-that-calculates-z))      ⎭
       (a value-of-a)
     ]

         ;;Things to be done inside the let
       (gtk ....)
       (gtk ....)
       (gtk ....)
);close let
```

There are two kinds of *let*; in the naked *let* one cannot use a local variable to calculate another one. For instance, in the example above one cannot use

the value of `x` to calculate `y`. However, if you choose to write `let*` (let-star) instead of a naked `let`, you can use any previous local definition to find the value of a given variable. Let us consider the concrete example of *let* that appears in `sta.scm`.

```
(let [  (w (gtk-window-new 0))
        (hbx (gtk-hbox-new #t 0))
        (do-button (gtk-button-new-with-label "Hello"))
        (exit-button (gtk-button-new-with-label "Exit")) ]
                                                            };;Local variables

      (gtk-box-pack-start hbx exit-button 1 1 0)
      (gtk-box-pack-start hbx do-button 1 1 0)
      (gtk-container-add  w hbx)
      (gtk-widget-show-all w)
      (do  ([event (gtk-server-callback) (gtk-server-callback)] )
           ([equal? event exit-button])
        (when (equal? event do-button)
             (print "Hello, world!")
             (flush-output-port (current-output-port)) ) ) ) )
```

In this example, the *let* has four local variables, to wit:

w The local variable **w** will hold the handler of the process main window. It is created by a call to the function **(gtk-window-new 0)**, where 0 indicates a top level window.

exit-button This variable will hold a pointer to the exit button. If you press `"Exit"`, you will leave the program.

do-button In this program, the `do-button` is idle.

hbx You can view **hbx** as a shelf that holds the **exit-button** and the **do-button**. In the body of the *let*, both buttons will be packed into **hbx**. In its turn, hbx will be added to the window.

```
(gtk-box-pack-start hbx exit-button 1 1 0)
(gtk-box-pack-start hbx do-button 1 1 0)
(gtk-container-add  win hbx)
```

Up to this point, we have mounted a window, with a horizontal box that
holds two buttons. Now, we must request that the computer shows this
window. This can be accomplished by the following command:

```
(gtk-widget-show-all w)
```

To close this program, we will put the `gtk-server` in a loop that will last
until someone presses the `exit-button`.

```
(do  ([event (gtk-server-callback) (gtk-server-callback)] )
        ([equal? event exit-button])
     (when (equal? event do-button)
           (print "Hello, world!")
           (flush-output-port (current-output-port)) ) )
```

You can type this program using the Emacs text editor. If you want some-
thing smaller, and easier to install, you can use the Simple IDE, provided with
this documentation. After typing the program and saving it as `sca-bttn.scm`,
you can compile it from the command line:

```
bigloo -c gtk.scm
bigloo -o sca-bttn sca-bttn.scm gtk.o -lgtk-server
```

If you run the program by typing `sca-bttn` from the command line, you will
see the following application pop up:

You may be wondering why the box creation function has two arguments,
that in `sca-bttn` were set to `#t` and 0. The second argument sets the distance
between one component and the next. In this case, it sets the distance
between one button and the other. If the first argument is true, the program
will give both buttons the same width. From `scb-bttn.scm` (vide fig. 1.2),
you can see what happens if one uses the following setting for `hbx`:

```
(hbx (gtk-hbox-new #t 20))
```

Let me make a suggestion here. Study the program `scb-bttn.scm`, compile
it, and run it.

```
(module test (main main)
  (include "macro-def.sch")
  (import (gtkserver_gtk "gtk.scm")) )

(define (main argv)
  (gtk-init)
  (let* ( (win (gtk-window-new 0))
          (hbx (gtk-hbox-new #t 60))
          (exit-button (gtk-button-new-with-label "Exit"))
          (hello-button (gtk-button-new-with-label "Hello"))  )

     (gtk-box-pack-start hbx exit-button 1 1 0)
     (gtk-box-pack-start hbx hello-button 1 1 0)

     (gtk-container-add  win hbx)
     (gtk-widget-show-all win)
     (do  ([event (gtk-server-callback) (gtk-server-callback)] )
          ([equal? event exit-button])
       (when (equal? event hello-button)
             (print "Hello, world!")
             (flush-output-port (current-output-port)) )
     )
   )
)
```

Figure 1.2: File scb-bttn.scm

1.2   The simple IDE

The IDE that goes with this package was designed not to compete with more complete tools, like Bee, but to show you how to write this kind of program. It is simple enough for you to study and understand it, and sufficiently complete to be useful. Figure 1.3 shows what it looks like.

If you type the name of a file on the data entry field, and press the button **load**, the IDE will place the file in the main buffer, where you can modify it. Then, you may write its name back onto the entry field, and save it.

If you need to check parentheses, select the expression, and press the button (((())).
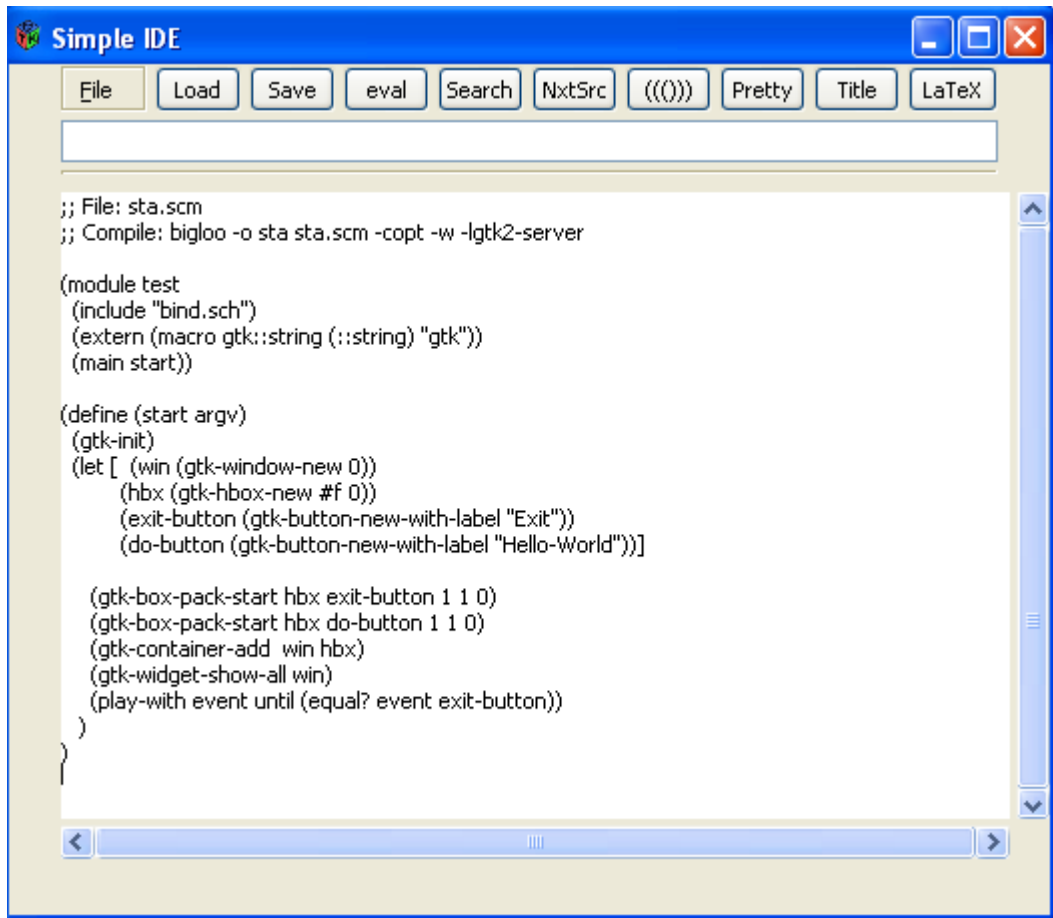
Figure 1.3: A Simple IDE

If you want the pretty format of an expression, select it and press the button `Pretty`. To search for a particular text segment, type it onto the data entry field, and press the button `Search`. To find the next occurence of the same segment, press the button `NxtSrc`. If you want to evaluate a Scheme expression, select it and press `Eval`. Finally, you can browse the file tree. To this end, scroll down the File menu, and choose the option *Browse*. The option *New Buffer* allows you to open a secondary buffer for comparison with the main buffer.

You can copy a text segment to the clipboard by selecting it, and pressing Ctrl-C, as usual in text editors. You can insert the contents of the clipboard by utilizing Ctrl-V. The exit button can be found on the File menu.

```
(module command-line (main main)
  (include "macro-def.sch")
  (import (gtkserver_gtk "gtk.scm")))

(define (main argv)
  (gtk-init)
  (let* ( (win (gtk-window-new 0))
          (hbx (gtk-hbox-new #f 0))
          (button (gtk-button-new-with-label (read-exit argv))))

     (gtk-box-pack-start hbx button 1 1 0)
     (gtk-container-add  win hbx)
     (gtk-widget-show-all win)
     (play-with event until (equal? event button))
   ); end let*
); end define

(define (read-exit xs)
  (with-handler (lambda(e) "Exit")
      (cadr xs))  )
```

Figure 1.4: File scc-argv.scm

1.3   Reading the exit button

Figure 1.4 shows an interesting example, where the label of the exit button
is read from the command line. Let us compile and execute it.

```
$ bigloo -o scc-argv scc-argv.scm gtk.o -lgtk-server
```

See what happens if you pass the argument "halt" to the command line:

```
$ scc-argv halt
```

1.4   Some action

Let us see how we add an action button to a GUI application. The action button is supposed to read a number from a data entry, calculate its factorial, and return the result to the data entry. Therefore, if you want to accomplish these objectives, you must create a data entry. Then you will have three components on your window:

(entr (gtk-entry-new)) ;;The data entry field

(ext (gtk-button-new-with-label "exit")) ;;The exit button

(fact (gtk-button-new-with-label "!")) ;;The factorial button

As for the layout, the idea is to have the factorial button situated on the right hand side of the data entry field, as you can see in the figure below.

To accomplish this kind of layout, you must create a horizontal box (hbx) and a vertical box (vbx). The data entry (entr) and the factorial button (represented by an exclamation mark) will go into hbx. When it comes to its turn, hbx will be stacked inside vbx.

```
(gtk-box-pack-start hbx entr 1 1 0)
(gtk-box-pack-start hbx fact 1 1 0)

(gtk-box-pack-start vbx  ext 1 1 0)
(gtk-box-pack-start vbx hbx 1 1 0)
```

The `play-with` form has a sequence of cond-like conditions that represent activation of the action buttons. In the program of figure 1.5, there is only one action button. Therefore, `play-with` has only one condition.

```
(play-with event until (equal? event ext) thus
   [ fact
      (set! data (string->number (gtk-entry-get-text entr)))
      (when data
         (gtk-entry-set-text entr
         (number->string (factorial data)  ))) ]
```

```scheme
;  Compile: bigloo -o scd scd-factorial.scm gtk.o  -lgtk-server

(module test (main start-here)
  (include "macro-def.sch")
  (import (gtkserver_gtk "gtk.scm")))

(define (start-here argv)
  (gtk-init)
  (let* ( (win (gtk-window-new 0))
          (hbx (gtk-hbox-new #f 0))
          (vbx (gtk-vbox-new #f 0))
          (entr (gtk-entry-new))
          (ext (gtk-button-new-with-label "exit"))
          (fact (gtk-button-new-with-label "!"))
          (data #f) )

      (gtk-box-pack-start hbx entr 1 1 0)
      (gtk-box-pack-start hbx fact 1 1 0)

      (gtk-box-pack-start vbx  ext 1 1 0)
      (gtk-box-pack-start vbx hbx 1 1 0)

      (gtk-container-add  win vbx)
      (gtk-widget-show-all win)
      (play-with event until (equal? event ext) thus
          [ fact
             (set! data (string->number (gtk-entry-get-text entr)))
             (when data
                (gtk-entry-set-text entr
                    (number->string (factorial data)  ))) ]
) )   )

(define (factorial n)
  (if (< n 1) 1
      (* n (factorial (- n 1)))))
```

Figure 1.5: scd-factorial.scm — A program to calculate factorials

1.5   Message dialog

In the next program, there are quite a few things to learn: tables, message dialogs, and the do loop.

```
;; Compile with:
;;;;; bigloo -o sce-msg sce-message-dialog.scm gtk.o -lgtk-server

(module test (main start)
  (include "macro-def.sch")
  (import (gtkserver_gtk "gtk.scm")))

(define (start arqv)
  (gtk-init)
  (let* ( (win (gtk-window-new 0))
          (dlg   (gtk-message-dialog-new win 0 1 5 "Bye!"))
                   ;; 1- warning 5- ok/cancel
          (table  (gtk-table-new 30 30 1))
          (exit-button (gtk-button-new-with-label "Exit"))
          (label (gtk-label-new "Bigloo running GTK") ) )

     (gtk-table-attach-defaults table exit-button 20 29 25 29)
     (gtk-table-attach-defaults table label 5 25 10 20)
     (gtk-container-add  win table)
     (gtk-window-set-title win "Demo with GTK-server lib")
     (gtk-widget-show-all win)

     (do ([event (gtk-server-callback)
                 (gtk-server-callback)] )
         ((equal? event exit-button) ; did you pressed 'exit'?
            ;; If so, print the text and leave:
            (print (gtk-dialog-run dlg))
            ; If you press OK, the program prints -5
            ; If you press CANCEL, it prints -6
            ; YES -> -8, NO -> -9
         );close exit condition
      );closed do
  );close let-star
);close define
```

Tables are containers, i.e., entities that can hold GUI components like buttons and data entries. You have seen two examples of containers before: vbox and hbox. Using tables, it is possible to create a grid in which you can place widgets. The function that creates a table has three arguments:

```
(gtk-table-new  30 ;;;  number of   rows
                30 ;;;  number of columns
                1 );;;  homogeneous grid
```

The first argument is the number of rows contained in the table, while the second is the number of columns. The homogeneous argument shows how the table's boxes are sized. If it is 1, the table's boxes are resized to the size of the largest widget in the table. If homogeneous is 0, the size of a table's boxes is dictated by the tallest widget within the same row, and the widest widget along the column.

The rows and columns are laid out from 0 to n, where n was the number specified in the call to `gtk-table-new`. So, if you specify rows = 3 and columns = 5, the layout would look something like this:
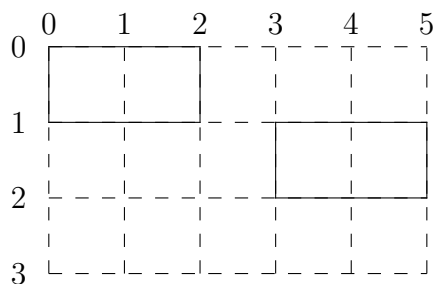
Notice that the coordinate system starts in the upper left hand corner. To place a widget into the box that is at the upper left hand side corner, use the following function:

```
(gtk-table-attach-defaults tbl exit-button 0 2  0  1)
```

To place a widget in the other box, the command is

```
(gtk-table-attach-defaults tbl label-widget  1 2  3 5)
```

You can create a message dialog using the command shown below.

```
(dlg   (gtk-message-dialog-new win 0 1 5 "Bye!"))
```

The first argument (win) is the parent window. The second number code indicates the kind of dialog. In the example, the code is 1, which indicates a *warning* dialog. The possibilities are

```
0   Information
1   Warning
2   Question
3   Error
```

The third number code, that is 5 in the example, represents the kind of buttons and their quantities. The list of codes is given below.

```
0 has no button
1 has an OK button
2 has a CLOSE button
3 has a CANCEL button
4 has YES/NO buttons
5 has OK/CANCELL buttons
```

When the dialog ends, it returns a code that depends on the button pressed. The following table contains the answers given by all the message dialog buttons:

```
;;    OK         --> -5
;;    CANCEL     --> -6
;;    YES        --> -8
;;    NO         --> -9
```

The example opens a message dialog on exit, and prints the code of the dialog button pressed. The example also illustrates the use of the do loop. The do loop has three parts:

```
(do [(i 0 (+ i 1))
     (xs '(a b c) (cdr xs))        } ;;Span of local variables
    ]

     ( (null? xs) )         ← Halting condition

     ;;Body
     (print i " - " (car xs))      } ;;Body
);close do
```

In the span section, you must define the initial value, and how the loop variable changes at the begining of each iteration. In the example, the initial value of i is 0, and it changes by adding 1 to the value of the previous iteration. Therefore, i will receive a sequence of integers: $0, 1, 2 \ldots$ The variable xs will receive the list '(a b c) when the loop starts, and will be updated by taking the `cdr` of the previous iteration. The do loop will stop when xs becomes empty. In this case, the body of the do loop prints the two loop variables. Therefore, it will produce the following output:

```
0 - a
1 - b
2 - c
#t
```

In the program of the example, one has the following do loop:

```
(do ((event (gtk-server-callback)
            (gtk-server-callback)) )
        ((equal? event exit-button)
            (print (gtk-dialog-run dlg))  )  )
```

If the user presses the exit-button, s/he will attribute `exit-button` to event; this will make the clause (`equal? event exit-button`) be true (or `#t`, in Scheme). The result is that the server will interrupt the do loop. The clause

```
((event (gtk-server-callback) (gtk-server-callback)))
```

initializes the variable 'event' to the value returned by (`gtk-server-callback`); the next time that it goes through the do loop, the server updates 'event' to a new value. Events produced by (`gtk-server-callback`) are things like button pressed, or mouse clicked.

1.6 Makefile

At the head of every program, you have found instructions to compile it. This is a good idea, since compilation of complex programs is not trivial. Sources are often split among many files, and it is also neceessary to link the application to more than one library in order to build it. If there is one thing

I hate it is receiving the sources of a program, and not being able to compile it for lack of instructions. However, besides instructions, it is well to ship the sources with a Makefile.

A Makefile is a file that reside in the same directory as the source, and whose name must be `Makefile`. It works like a programming languages, i.e., it has comments, labels, commands, attributions, etc. A comment starts with the hash `#` symbol at the first column. A label, that also starts at the first column, is an identifier followed by colon, tab and a dependence list. For instance, listing 1.6 has five labels, sca-bttn, scb-bttn, scc-argv, scd-factorial, sce-message-dialog.

If you have the Makefile from figure 1.6 in the same directory as the sources, all you need to do to compile a program is to type

```
make scd-factorial
```

where *scd-factorial* is the Makefile label for the source.

Immediately below the dependence list, you will find tab followed by the compilation command. Notice that the command cannot start at the first column; you must insert tab before the command. Finally, one has a command to remove the object files. The entry for `sca-bttn` can be summarized thus:

```
sca-bttn:  sca-bttn.scm  macro-def.sch
           bigloo -o sca-bttn sca-bttn.scm gtk.o -lgtk-server
           rm *.o
```

In a Makefile, a label must be left-justified, and followed by a colon. For instance, at the right hand side of the example, you will find the labels

```
sca-bttn:           ...
scb-bttn:           ...
scc-argv:           ...
scd-factorial:      ...
sce-message-dialog: ...
```

Let us consider the first label, i.e.,

```
# Demonstration of GTK-server
sca-bttn:        sca-bttn.scm  macro-def.sch
                 bigloo -o sca-bttn sca-bttn.scm gtk.o -lgtk-server
                 rm *.o
scb-bttn:        scb-bttn.scm  macro-def.sch
                 bigloo -o scb-bttn scb-bttn.scm gtk.o -lgtk-server
                 rm *.o
scc-argv:        scc-argv.scm  macro-def.sch
                 bigloo -o scc-argv scc-argv.scm gtk.o -lgtk-server
                 rm *.o
scd-factorial:   scd-factorial.scm  macro-def.sch
                 bigloo -o scd-factorial scd-factorial.scm gtk.o
                   -lgtk-server
                 rm *.o
sce-message-dialog: sce-message-dialog.scm  macro-def.sch
                 bigloo -o sce-message-dialog sce-message-dialog.scm
                   gtk.o -lgtk-server
                 rm *.o
```

Figure 1.6: Makefile for sca-bttn, scb-bttn, scc-argv, scd-factorial and sce-message-diaglog.

```
    sca-bttn:        sca-bttn.scm  macro-def.sch
```

The label `sca-bttn` is followed by a colon, tab, and a dependence list. You must be careful to type tab after the colon mark; do not substitute spaces for tab, because this will produce an error. The dependence list indicates that if one modifies the sources of `sca-bttn.scm` or `macro-def.sch`, it is necessary to recompile everything.

If you have a Makefile as in figure 1.6, you can compile `sca-bttn.scm` simply by typing the following request from the command line:

```
    make sca-bttn
```

In the same guise, you can compile `scb-bttn.scm` by typing

```
    make scb-bttn
```

1.7   Text buffer

This example presents the difficult concept of iter. I would like to thank Peter van Eerten for helping my understanding of iters. Basically, an iter is a widget that points to a text buffer. To retrieve the text from a buffer, you need iters.

Here is how to get two brand new iters, one for finding where the buffer starts, and the other for reaching its end.

```
(startit (gtk-frame-new))
(endit   (gtk-frame-new))
```

The GTK command below sets `startit` and `endit` to the bounds of the buffer:

```
(gtk-text-buffer-get-bounds txt startit endit)
```

Now that we know where the text starts and ends, it is possible to retrieve the buffer contents:

```
(set! kw (gtk-text-buffer-get-text txt startit endit #t))
```

However, the example does not use the above function. Instead, it saves the buffer contents directly to a given file; to this end it uses a function that *macro-def.sch* provides for your convenience:

```
(gtk-save-buffer txt fname startit endit)
```

The function `gtk-save-buffer` saves the contents of the buffer `txt` from `startit` to `endit` into a file whose name is `fname`.

   As for the buffer, it is a string that stores every thing that you type on a text-view. Here is how the buffer was created in the program of the example:

```
(txt (gtk-text-buffer-new))
```

An example of how to insert text into the buffer:

```
(gtk-text-buffer-set-text  txt "" -1)
```

To show the buffer, you need to associate it with a text view:

```
(tv (gtk-text-view-new-with-buffer txt))
```

This snippet shows how to write programs with style. You know, if your problem is to get there, any bike will do; but if you want to get there with style, you will need something like a one-wheeled bike, or a very expensive aerodynamic three-wheeled bike. If you take a careful look at the source of stf.scm, you will notice that, in general, only functions that return something are in the definition section of the let*. Functions that do not return a value are like Gorgo's son: they are placed on top of the let* – such as `(gtk-init)` – or behind it. For those who do not know Gorgo's story, she was the beloved wife of Leonidas, king of Sparta. When the Persians invaded Greece, there was a shortage of steel to manufacture swords for everybody who wanted to fight. Therefore one of the swords was shorter than the others. The protocol required that the queen should be the person in charge of the distribution of weapons. Of course her son was in line, waiting for his sword. The Spartan youths were sure that the queen would not give the shorter sword to her son. But she did. The boy protested. "Mom, what can I do with this short sword?" "Step forward", answered the queen. There are computer languages that are short of commands. They lack attribution, or side effect. Programmers that use these languages step forward, and are able to write safer code than the others. Although Scheme does have attribution, you should shun from using this resource.

However, I still did not tell you the story of Gorgo's shield. Defeated soldiers used to throw away their shields. This is reasonable, because the shield is heavy, and it was difficult to flee from the battle field bearing a 80 pound shield. But there were brave soldiers that kept their shield, in order to protect their fellow combatants, and ensure an orderly retreat. If he died in battle, his companions would fetch his body on his shield. This behavior is what Gorgo expected from her son. Thus, she recommended when she handed him the precious shield: Behind it, or on top of it. Like Gorgo's son, functions that do not return values should be put behind the let*, or on top of it.

In this snippet, we did not use if, but cond. Notice that cond gives a more elegant organization to the button actions, spare parenthesis, and makes the logical structure of the program clear.

I think that you have guessed what we are doing! Yes, we are writing a text editor. In this step, we have added a scrolling window, and organized the components in hboxed and vboxed. Let us start with the scrolling window. Instead of inserting tv directly into the component table tbl, we have inserted it into a scrolling window:

```
(gtk-container-add sw tv)
```

The buttons were organized in a hbox:

```
(pack-homogeneously hbx ext sav lod)
```

The entry and this hbox are put inside a vbox:

```
(pack-pack vbx hbx sep en)
```

The vbox and the scroll window are set inside a table:

```
(gtk-table-attach-defaults tbl vbx 0 10 0 4 )
(gtk-table-attach-defaults tbl sw 0 19 5 19)
```

Finally, the table is nailed to a window:

```
(gtk-container-add win tbl )
```

Text buffers store strings in a format that is sui generis. Strings are supposed
to come between double quotes, and new lines must be represented by "\n".
Bigloo does not deal with strings likewise. Therefore, one must perform a lot
of clerical work to retrieve or insert text into buffers. To ease your problems,
`macro-def.sch` provides functions that hides this difficulties from your view.
You have already seen one of these functions:

```
(gtk-save-buffer txt fname startit endit)
```

The above function was used to define the button that loads the contents of
a file. Another function that deals with text is given below:

```
(gtk-load-lines fname) → list-of-strings
```

This function produces a list of strings, each one representing a line of the
file `fname`. What is better, each line is a string ready to insert into a text
buffer. The function for-each is applied to this list, and inserts each element
of it into the buffer `txt`:

```
(for-each
    (lambda(x)
        (gtk-text-buffer-get-end-iter txt endit)
        (gtk-text-buffer-insert txt endit x -1))
        (gtk-load-lines fname) )
```

```
;; Compile with:
;;
;; bigloo -o scf-edit scf-edit.scm gtk.o -lgtk-server
;;

(module test (main main)
  (include "macro-def.sch")
  (import (gtkserver_gtk "gtk.scm")))

(define (main arqv)
  (gtk-init)
  (let* ( (win (gtk-window-new 0))
          (tbl  (gtk-table-new 20 20 1))
          (startit (gtk-frame-new))
          (endit (gtk-frame-new))
          (ext (gtk-button-new-with-label "Exit"))
          (sav (gtk-button-new-with-label "Save"))
          (lod (gtk-button-new-with-label "Load"))
          (en (gtk-entry-new))
          (hbx (gtk-hbox-new #f 0))
          (vbx (gtk-vbox-new #f 2))
          (sep  (gtk-hseparator-new))
          (sw (gtk-scrolled-window-new))
          (txt (gtk-text-buffer-new))
          (tv (gtk-text-view-new-with-buffer txt))
          (fname "")
          (kw "")          )

     (pack-homogeneously hbx ext sav lod)
     (pack-pack vbx hbx sep en)

     (gtk-container-add sw tv)
     (gtk-table-attach-defaults tbl vbx 0 10 0 4 )
     (gtk-table-attach-defaults tbl sw 0 19 5 19)

     (gtk-text-view-set-editable tv 1)
     (gtk-text-view-set-wrap-mode tv 2)
```

```
    (gtk-container-add  win tbl)
    (gtk-window-set-title win "Simple text editor")
    (gtk-widget-show-all win)

    (play-with event until (equal? event ext) thus
        [sav (set! fname (gtk-entry-get-text en))
             (set! fname (choose-file fname))
             (gtk-save-buffer txt fname startit endit)]
        [lod (set! fname (gtk-entry-get-text en))
             (gtk-text-buffer-set-text  txt "" -1)
             (for-each
                 (lambda(x)
                     (gtk-text-buffer-get-end-iter txt endit)
                     (gtk-text-buffer-insert txt endit x -1))
                 (gtk-load-lines fname) ) ]   )))

(define (choose-file x)
    (if (< (string-length x) 2)
        "backup.txt"
        x))
```
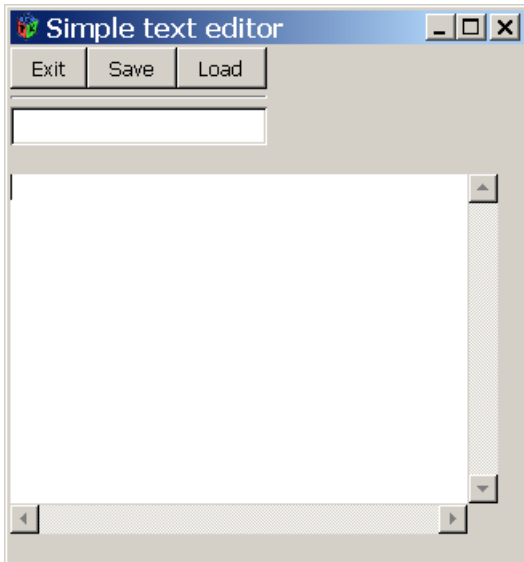
1.8 Graphics

In the next problem, it is necessary to create three graphic entities, that we have not seen yet: Event box, pixmap and graphic context.

```
(EBOX (gtk-event-box-new))
(PIX (gdk-pixmap-new gdkwin xsize ysize -1))
(GC (gdk-gc-new PIX))
```

As its name suggests, the event box will capture events from a graphic canvas; for instance, mouse clicks are events that could occur on a graphic canvas. The graphic context will define a context in which a figure will be drawn. For example, the background and foreground colors are part of the context of any drawing. Here is how to esblish the background and foreground collors of a graphic context:

```
(gdk-color-parse "#ff0000" COLOR)
(gdk-gc-set-rgb-fg-color GC COLOR)
```

The colors are defined as a six digit hexadecimal number; the first two digits specify the amount of red, the central pair of digits specify the amount of green, and finally the last two digits specify the amount of blue of context. In the example, the color is given by `#00ff00`, where the amount of red is 0 (first two digits), the amount of green is ff (= 255), and the amount of blue is also 0. Therefore the figure will be green.

Every time the gtk-server goes through the drawing loop, it must recycle the pixmap. The first step is to give back to the operational system the memory taken by the previous pixmap. Next, one needs to create a new pixmap, where diverse figures will be drawn.

```
(g-object-unref PIX)
(set! PIX (gdk-pixmap-new gdkwin xsize ysize -1))
```

In order to capture mouse clicks, it is necessary to connect the event box to the gtk-server, which is done by the command below.

```
(gtk-server-connect EBOX "button-press-event canvas")
```

When the user left clicks the mouse, the function `(gtk-get-mouse-button)` will return 256. Then, the function `gdk-draw-rectangle` uses the mouse coordinates (`(gtk-mouse-x)`, `(gtk-mouse-y)`) to draw a rectangle.
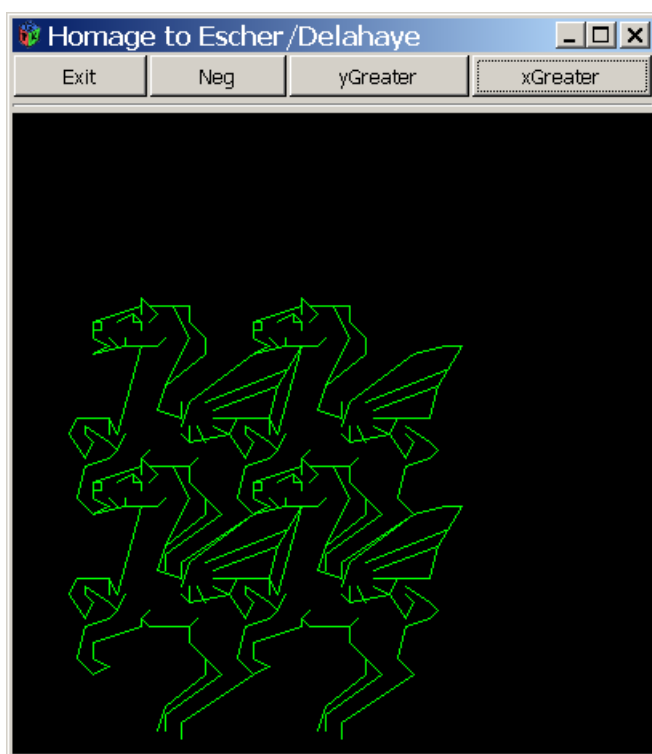
```scheme
;; $ bigloo -o scs-mouse scs-mouse.scm  gtk.o -lgtk-server
(module mus (main main)
   (include "macro-def.sch")
   (import (gtkserver_gtk "gtk.scm")))

(define (main args) (gtk-init)
  (let* ( (win (gtk-window-new 0))
          (image (gtk-image-new))
          (EBOX (gtk-event-box-new))
          (ext  (gtk-button-new-with-label "Exit"))
          (vbx (gtk-vbox-new #f 2))
          (gdkwin #f)
          (PIX #f)
          (GC #f)
          (grt (gtk-widget-create-pango-layout image "Hello"))
          (COLOR (gtk-frame-new)) )
     (gtk-container-add EBOX image)
     (pack-pack vbx ext EBOX)
     (gtk-container-add win vbx )
     (gtk-widget-show-all win)
     (set! gdkwin (gtk-widget-get-parent-window image))
     (set! PIX (gdk-pixmap-new gdkwin 400 300 -1))
     (set! GC (gdk-gc-new PIX))
     (gtk-image-set-from-pixmap image PIX)
     (gtk-window-set-title win "Mouse example!")
     (gtk-server-connect EBOX "button-press-event canvas")
     (play-with evt until (equal? evt ext) thus
         ["canvas"
            (when (equal? (gtk-get-mouse-button) 256)
               (g-object-unref PIX)
               (set! PIX (gdk-pixmap-new gdkwin 400 300 -1))
               (gtk-image-set-from-pixmap image PIX)
               (gdk-color-parse "#ff0000" COLOR)
               (gdk-gc-set-rgb-fg-color GC COLOR)
               (gdk-draw-rectangle PIX GC #t
                       (gtk-mouse-x) (gtk-mouse-y) 80 80))
               (gdk-color-parse "#00ff00" COLOR)
               (gdk-gc-set-rgb-fg-color GC COLOR)
               (gdk-draw-layout PIX GC (+ 20 (gtk-mouse-x))
                                 (+ 20 (gtk-mouse-y)) grt) ] ) ))
```

1.9 Le cheval de Delahaye

This program was designed to exemplify the use of GTK-server's graphic functions. When you enter the program, you will see one of Escher's famous designs (hopefully, his state will not sue me :-). There are four copies of the design, one on the top of the other. If you press the button yGreater, two copies will start to displace vertically in relation to the other two. Keep pressing yGreater until the pieces fit into each other like puzzle pieces made from one of Escher's engravings. If you make a mistake, and the displaced piece goes beyond the desired position, press the button Neg once; then the next time you press the buttons yGreater or xGreater, the pieces will move in the opposite direction. Finally, displace the pieces in the x-direction by pressing the xGreater button. The last step is to study the program, and learn how it works. By the way, the only use for the small green rectangle that is above Escher's horse is to teach me how to draw rectangles using the GTK-server.

```scheme
;;Compile with:
;;  bigloo -o scg-escher scg-escher.scm gtk.o -lgtk-server
(module cheval (main main)
  (include "macro-def.sch")
  (import (gtkserver_gtk "gtk.scm")))

(define (make-horse image gdkwin)
  (let* (  (xsize 400)
           (ysize 400)
           (rx 0) (ry 0)
           (PIX (gdk-pixmap-new gdkwin xsize ysize -1))
           (GC (gdk-gc-new PIX))
           (COLOR (gtk-frame-new)) )
      (lambda(dx dy)
      (g-object-unref PIX)
      (set! PIX (gdk-pixmap-new gdkwin xsize ysize -1))
      (gtk-image-set-from-pixmap image PIX)
      (gdk-color-parse "#00ff00" COLOR)
      (gdk-gc-set-rgb-fg-color GC COLOR)
      (set! rx (+ rx dx))
      (set! ry (+ ry dy))
      (for-each (lambda(x)
          (draw PIX GC x ysize 0)
          (draw PIX GC x (- ysize ry) 0)
          ;; =====================
          (draw PIX GC x ysize rx)
          (draw PIX GC x (- ysize ry) rx)
            ) cheval)  )))


(define  (draw pix gc xs ymax xp)
    (let ((a (car xs)) )
      (for-each
          (lambda (b)
              (gdk-draw-line pix gc
                      (+ xp (* 5 (car a)))
                      (- ymax (* 5 (cadr a)))
                      (+ xp (* 5 (car b)))
                      (- ymax (* 5 (cadr b))))
            (set! a b)
          ) xs) ) )
```

```
(define (main args)
  (gtk-init)
  (let* ( (win (gtk-window-new 0))
          ;; Create widget to display image
          (image (gtk-image-new))

          ;; Create eventbox
          (EBOX (gtk-event-box-new))

          (nnn 2)
          (ext  (gtk-button-new-with-label "Exit"))
          (neg  (gtk-button-new-with-label "Neg"))
          (xGt  (gtk-button-new-with-label "xGreater"))
          (yGt  (gtk-button-new-with-label "yGreater"))

          (hbx (gtk-hbox-new #f 0))
          (vbx (gtk-vbox-new #f 2))
          (horses #f)
          )

      (gtk-container-add EBOX image)
      (pack-homogeneously hbx ext neg yGt xGt)
      (pack-pack vbx hbx  EBOX)
      (gtk-container-add win vbx )
      (gtk-widget-show-all win)
      (set! horses (make-horse image
                     (gtk-widget-get-parent-window image) ))
      (gtk-window-set-resizable win #t)
      (gtk-window-set-default-size win 600 400)

      (gtk-window-set-title win "Homage to Escher/Delahaye")

      (horses 0 0)

      (play-with event until (equal? event ext) thus
          [ neg (set! nnn (- nnn))]
          [xGt (horses nnn 0) ]
          [yGt (horses 0 nnn)]
  )     )   )
```

```
(define cheval '(
    ((10 10)(8 12)(9 16)(12 17)(13 18)(14 20))
    ((13 18)(12 19)(9 21)(9 20)(10 19)(9 17)(7 20)(8 22)(12 22))
    ((12 20)(12 22)(13 20)(16 31)(18 31)(19 32) )
    ((16 31)(14 31 )(14 32) )

    ((14 31)(10 30)(12 31)(10 32)(10 34)(11 34)(11 33)(10 33) )

    ((12 32)(13 31))

    ((10 34)(16 36) )
    ((16 35)(16 37)(18 35)(17 34)) ; ear
    ((17 36)(20 36)(22 32)(19 26) ) ; behind the ear

    ((20 36)(22 36)(22 34)(24 32)(24 30)
                (19 26)  (18 23)(21 22)(21 24) ) ;back
    ((30 30)(34 31)(36 31)(33 26)(32 22)
        (28 22)(27 20)(29 17)(30 19)(29 20) ) ; wing


    ;Legs
    ((29 21)(32 19)(33 18)(32 17)(29 16)(28 12)(30 10)(21 4)(21 2))
    ((16 17)(16 16)(10 14) (10 12)(12 11)(10 10));front leg

    ((18 3)(19 6)(24 10)(24 12) (22 14)(22 16)(23 17) );belly
    ((22 16)(17 16)(16 17)(17 18) )

    ((21 21)(22 24)(30 30) ); upper wing
    ((24 24)(34 28))
    ((25 23)(33 26))
    ((25 21)(27 20));feathers
    ((23 21)(24 19))
    ((27 20)(22 19)(22 21))
    ((22 19)(21 20))

    ((13 34)(15 35)(16 34)(16 33))
    ((15 35)(15 34)(16 34)(15 34)(15 35)); eye
    ((24 12)(26 10)(19 5)(19 3))
    ((28 22)(25 22))
) )
```

A drawing function must receive the pixmap and the context as arguments. For instance, the below function links a sequence of coordinates with straight lines; in this function, `pix` and `gc` hold the pixmap and the graphic context; `xs` receives a list of coordinates; `ymax` and `xp` are used to scale down the coordinates.

```
(define  (draw pix gc xs ymax xp)
    (let ((a (car xs)) )
       (for-each (lambda (b) ( gdk-draw-line pix gc
                                   (+ xp (* 5 (car a)))
                                   (- ymax (* 5 (cadr a)))
                                   (+ xp (* 5 (car b)))
                                   (- ymax (* 5 (cadr b))))
                              (set! a b) ) xs) ) )
```

Suppose, for example, that the argument `xs` receives the list

```
((13 34)(15 35)(16 34)(16 33))
```

The execution starts with `a=(13 34)`, that is the first element of `xs`. The `(lambda(b)...)`-function is applied to each element of the list

```
((13 34)(15 35)(16 34)(16 33))
```

In the first application, `a=(13 34)` and `b=(13 34)` too. In the second application, we have `a=(13 34)` and `b=(15 35)`. Therefore, `gdk-draw-line` will connect the point (13 34) to (15 35). In the next iteration, it will connect `a=(15 35)` to `b=(16 34)`. Finally, it will connect `a=(16 34)` to `b=(16 33)`. The command `(set! a b)` makes the first point of the present iteration equal to the second point of the previous iteration.

Now we will examine the idea of programs that builds another program. Consider the following snippet.

```
(define (make-counter x)
    (lambda() (set! x (+ x 1)) x))
```

This snippet produces as value a function that increments `x`, and returns it. It is a program that produces another program. The daughter program inherits all variables of the mother program. Let us define a daughter program.

```
(define cnt (make-counter 3))
```

The program `cnt` will inherit the variable `x` from `make-counter`. The first time you call `cnt`, it returns 4; the second time, it returns 5; and so on. It is a good idea to test this behavior by yourself. The function `make-horse`, defined on page 36, is a good example of programs that produces programs. In `make-horses`, there are many local variables; besides `image` and `gdkwin`, that are passed as parameters, one defines the following local let-variables:

```
(let* [    (xsize 400)
           (ysize 400)
           (rx 0)
           (ry 0)
           (PIX (gdk-pixmap-new gdkwin xsize ysize -1))
           (GC (gdk-gc-new PIX))
           (COLOR (gtk-frame-new))
       ]
     ... ; body of the let
); end of let
```

When one uses `make-horse` to create `horses` (vide page 36), the daughter function will inherit all these variables, and use them in its body. Each time one calls the function `horses`, it updates its local variables `rx` and `ry`, displacing the four drawings in relation to each other. The pixmap and the graphic context necessary to make the drawings are inherited from `make-horse`.

The `for-each` loop is used twice to draw a horse. Inside the `draw`-function, the `for-each` loop is used to connect the coordinates of each horse segment. Inside the function returned by `make-horse`, the `for-each` loop goes through all horse components.

It is a good idea to examine the workings of the `for-each` loop. Basically, it applies a one argument procedure to each element of a list. Here is an example of its use:

```
1:=> (for-each print '(3 4 5 6))
3
4
5
6
#unspecified
```

One often uses a lambda-expression as the functional argument of the `for-each` loop. For instance, in the definitions of `draw` and `make-horse` (vide page 36), the `for-each` loop was called with lambda-expressions.

1.9.1   Drawing text and images

On page 34, you have seen a program that draws a rectangle, whenever one clicks on the canvas. If you pay close attention to the rectangle, you will see that there is a message written on it. To write a message on the canvas, you must wrap it in a pango-layout.

```
(let* ((win (gtk-window-new 0))
       (image (gtk-image-new))
        ...
       (grt (gtk-widget-create-pango-layout image "Hello"))
        ...
     )
   ;; Body of the let-star
);end of the let-star
```

Now, `grt` contains a string wrapped in a layout. Let us pin it on the canvas, together with a red rectangle (see page 34):

```
(play-with evt until (equal? evt ext) thus
  ["canvas"
    (when (equal? (gtk-get-mouse-button) 256)
       (g-object-unref PIX)
       (set! PIX (gdk-pixmap-new gdkwin 400 300 -1))
       (gtk-image-set-from-pixmap image PIX)
       (gdk-color-parse "#ff0000" COLOR)
       (gdk-gc-set-rgb-fg-color GC COLOR)
       (gdk-draw-rectangle PIX GC #t
           (gtk-mouse-x) (gtk-mouse-y) 80 80))
       (gdk-color-parse "#00ff00" COLOR)
       (gdk-gc-set-rgb-fg-color GC COLOR)
       (gdk-draw-layout PIX GC (+ 20 (gtk-mouse-x))
                               (+ 20 (gtk-mouse-y)) grt) ] )
```

Notice that it is necessary to use different colors to draw the rectangle and the text. In fact, if you had used the same color for both drawables, the text would not be visible.

In the next page you will find an example on how to read a jpg image from a file and show it on the canvas. I hope that the example is simple enough for you to understand it without further explanation.

```scheme
;;  bigloo -o sck-img sck-images.scm gtk.o -lgtk-server

(module images (main main)
  (include "macro-def.sch")
  (import (gtkserver_gtk "gtk.scm")))


(define (main args) (gtk-init)
    (let* ( (win (gtk-window-new 0)) ; 0 means TOP WINDOW

            ;; Now image is defined from a jpg file.
            (image (gtk-image-new-from-file "church.jpg"))

            (EBOX (gtk-event-box-new))
            (_ (gtk-container-add EBOX image))
            (ext  (gtk-button-new-with-label "Exit"))
            (spinoza  (gtk-button-new-with-label "Spinoza's grave"))
            (church  (gtk-button-new-with-label "Church"))
            (hbx (gtk-hbox-new #f 0))
            (vbx (gtk-vbox-new #f 2))
            (sep  (gtk-hseparator-new)) )

       (pack-homogeneously hbx ext spinoza church)
       (pack-pack vbx hbx sep EBOX)
       (gtk-container-add win vbx )
       (gtk-widget-show-all win)
       (gtk-window-set-resizable win #t)
       (gtk-window-set-default-size win 600 400)
       (gtk-window-set-position win 1)
       (gtk-window-set-title win "Spinoza's grave!")
       (do  ( (event (gtk-server-callback) (gtk-server-callback)))
            ( (equal? ext event) (print "Good bye!"))
            (cond ( (equal? spinoza event)
                    (gtk-image-set-from-file image "stone.jpg"))
                  ( (equal? church event)
                    (gtk-image-set-from-file image "church.jpg"))
          );close cond
        );close do
   );close let*
); close define
```

Chapter 2

Introduction to Scheme

Now that you know how to design a GUI, let us learn how to program using the algorithm language Scheme. Since Scheme is a dialect of LISP, we will start with lists. A list is a data structure, i. e., a computational entity that one constructs from elementary components or parts. Data structures have three properties that give hints on how to deal with them:

1. Data structures are complex constructed entities. Therefore, the first thing that we need for dealing with them is a set of tools that build the desired structure. These tools are called **constructors**.

2. Data structures have parts. From this, one can infer that it is also important to have tools to retrieve these parts. Tools that select the parts of a data structure are called **selectors**.

3. Data structures have an external representation. One needs tools to read and write the representation of a given data structure.

The external representation of lists consists of an open left parenthesis, followed by zero or more atoms/lists/vectors/strings, followed by a closed right parenthesis. Atoms can be

- Numbers: 3.1416, 326, 18, etc.

- Symbols: rose, book, Eduardo, etc.

- Chars: #\A, #\B, #\C,...#\1, #\2,...#\tab, #\space, #\newline, etc.

- Boolean values: #t (true) and #\f (false).

When Scheme receives an unquoted list, it thinks that it is a program, and tries to evaluate it. In fact, any program in Scheme is made up of unquoted lists. Besides representing programs, lists can be and are used to represent data. One of the strengths of Scheme is the use of a uniform representation for data and programs.

If your list is not the representation of an operation, you need to quote it. Below you will find a few examples of quoted lists.

`'(* 3 4 5)`	The presence of the quotation mark prevents Scheme from evaluating the operation represented by this list. The list is considered to be nothing more than data. Quoted lists are not evaluated.
`'(Claudia Anna Margarete)`	A list of symbols that represent female names. *Claudia* is a Roman name that means limping woman. Anna is a Phoenician name, and means Holly. Margarete is a Greek name, and means Pearl.
`'( )`	An empty list. The predicate `null?` returns `#t` if a list is empty, and `#f` if it is not empty.

`'( (Sandra Rosa Magdalena)`
`   (Lili Marlene)`
`   (Luiza Miller))`
	To represent full names of females that inspired songs, you need a list of lists.

Since I talked about Roman names, let me elaborate on this theme. Girls did not receive names, only boys had names in Ancient Rome. However, there were only a few names at the disposal of the boy's parents; 20 names, to be exact. Therefore, a lot of Romans had the same name. A function is a map from an element of a domain set to one and only one element of an image set.

If there were a function mapping each Roman to a unique name, life would be a lot easier in Ancient Rome. In real life, Romans needed nick names to prevent confusion. If the boy had an enormous wart on his nose, a wart as big as a bean, he would be called Cicero. If he had a leg shorter than the other, he would be called Claudius (the limping one), and so on.

What about a Roman girl? How was she called, if she did not have a name? She would port the nick name of her father to the feminine gender. For instance, if her father was almost blind, he would be called Caecilius (the blind fellow), and his daughter would be Caecilia (the blind girl), even if she had very sharp vision. If her father had a leg shorter than the other, people would call her Claudia. If her father was a farmer, she would be Agripina. This solved the problem for the older daughter. What would happen if Claudia had sisters? The younger sisters would be numbered; Claudia Secunda (the second Claudia), Claudia Tertia (the third Claudia), and so on. Of course, Claudia Secunda would be called simply Secunda at home. Only to prevent confusion she would need the nick name of her father.

You have learned how to represent lists by an open parenthesis, followed by zero or more elements, followed by a closed parenthesis. Now you will learn about the list constructor, and the list selector. A list has two parts, a **car** that is the first element of the list, and a **cdr**, that points to what remains of the list structure if the first element is removed.

List definition	(car xs)	(cdr xs)

`(define xs '( Claudia`
`          Hanna`
`          Margarete))`
	`Claudia`	`(Hanna Margarete)`

`(define xs '(* 3 4 5))`	`*`	`(3 4 5)`
`(define xs '(1 2 3 4 5))`	`1`	`(2 3 4 5)`

The two list selectors allow one to retrieve any element from a list. For instance, if you want the first element of xs, where xs is a list, you need to take its car; to get the second element, you need the car of the cdr. The third element is given by taking the cdr twice, and the car once, and so on.

xs	(define xs '(1 2 3 4))
(car xs)	1
(car (cdr xs))	2
(car (cdr (cdr xs)))	3
(car (cdr (cdr (cdr xs))))	4

Let us also examine what happens when taking the cdr in succession.

xs	(define xs '(1 2 3 4))
(cdr xs)	(2 3 4)
(cdr (cdr xs))	(3 4)
(cdr (cdr (cdr xs)))	(4)
(cdr (cdr (cdr (cdr xs))))	( )

The best way to learn the primitive functions provided by Scheme is through the use of the interpreter. In figure 2.1, you can see an interaction with the interpreter. There you can see the use of other tools besides the two list selectors. For instance, (set! xs (cdr (cdr xs))) is called destructive attribution, or set-bang for short. It substitutes the (cdr (cdr xs)) for the value of xs. For instance, if

        xs= (1 2 3 4 5)

it will be reduced to (3 4 5) after set-bang: (set! xs (cdr (cdr xs))). As for function (null? xs), it returns #t or #f, informing whether its argument is an empty list or not.

Any funcion, like (null? xs), that returns #t or #f is called a predicate. In Scheme, most predicates end with a question mark. For instance, (equal? xs ys) is a predicate that returns #t if xs is equal to ys.

```
$ bigloo
------------------------------------------------------------
Bigloo (2.8c)                                     ,--^,
'a practical Scheme compiler'                   _ ___/ /|/
Fri Aug 4 17:19:43 CEST 2006                 ,;'( )__, ) '
Inria -- Sophia Antipolis                    ;;  //   L__.
email: bigloo@sophia.inria.fr                '  \    /  '
url: http://www.inria.fr/mimosa/fp/Bigloo         ^   ^
------------------------------------------------------------

1:=> (define xs '(1 2 3 4 5))
xs
1:=> xs
(1 2 3 4 5)
1:=> (car xs)
1
1:=> (cdr xs)
(2 3 4 5)
1:=> (car (cdr xs))
2
1:=> xs
(1 2 3 4 5)
1:=> (car (cdr (cdr xs)))
3
1:=> (cdr (cdr xs))
(3 4 5)
1:=> (set! xs (cdr (cdr xs)))
#unspecified
1:=> xs
(3 4 5)
1:=> (null? (cdr xs))
#f
1:=> (null? (cdr (cdr (cdr xs))))
#t
1:=>
```

Figure 2.1: Testing the list selectors

2.1   Functions

A set is a collection of things. You certainly know that collections do not have repeated items. I mean, if a guy or gall has a collection of stickers, s/he does not want to have two copies of the same sticker in his/her collection. If s/he has a repeated item, s/he will trade it for another item that s/he lacks in his/her collection.

Mathematicians collect other things, besides coins, stamps, and slide rules. They collect numbers, for instance; therefore you are supposed to learn a lot about sets of numbers.

N **is the set of natural integers.** Here is how mathematicians write the elements of N: $\{0, 1, 2, 3, 4 \ldots\}$.

Z **is the set of integers, i.e.,** $Z = \{\ldots - 3, -2, -1, 0, 1, 2, 3, 4 \ldots\}$.

Why is the set of integers represented by the letter Z? I do not know, but I can make an educated guess. The set theory was discovered by Georg Ferdinand Ludwig Philipp Cantor, a Russian whose parents were Danish, but who wrote his *Mengenlehre* in German! In German, integers may have some strange name like *Zahlen*.

You may think that set theory is boring; however, many people think that it is quite interesting. For instance, there is an Argentinean that scholars consider to be the greatest writer that lived after the fall of Greek civilization. In few words, only the Greeks could put forward a better author. You probably heard Chileans saying that Argentineans are somewhat conceited. *You know what is the best possible deal? It is to pay a fair price for Argentineans, and resell them at what they think is their worth.* However, notwithstanding the opinion of the Chileans, Jorge Luiz Borges is the greatest writer who wrote in a language different from Greek. Do you know what was his favorite subject? It was the Set Theory, or *Der Mengenlehre*, as he liked to call it.

When a mathematician wants to say that an element is a member of a set, he writes something like

$$3 \in Z$$

If he wants to say that something is not an element of a set, for instance, if he wants to state that -3 is not an element of N, he writes:

$$-3 \notin N$$

2.2 Irrational Numbers

At Pythagora's time, Ancient Greeks claimed that any pair of line segments is commensurable, i.e., you can always find a meter, such that the lengths of any two segments are given by integers. The following example will help you understand the Greek theory of commensurable lengths at work. Consider the square of figure 2.2.

If the Greeks were right, I would be able to find a meter, possibly a very small one, that produces an integer measure for the diagonal of the square, and another integer measure for the side. Suppose that p is the result of measuring the side of the square, and q is the result of measuring the diagonal. The Pythagorean theorem states that $\overline{AC}^2 + \overline{CB}^2 = \overline{AB}^2$, i.e.,

$$p^2 + p^2 = q^2 \therefore 2p^2 = q^2 \tag{2.1}$$

You can also choose the meter so that p and q have no common factors. For instance, if both p and q were divisible by 2, you could double the length of the meter, getting values no longer divisible by 2. E.g. if $p = 20$ and $q = 18$, and you double the length of the meter, you get $p = 10$, and $q = 9$. Thus let us assume that one has chosen a meter so that p and q are not simultaneously even. But from equation 2.1, one has that q^2 is even. But if q^2 is even, q is even too. You can check that the square of an odd number is always an odd number. Since q is even, you can substitute $2 \times n$ for it in equation 2.1.

$$2 \times p^2 = q^2 = (2 \times n)^2 = 4 \times n^2 \therefore 2 \times p^2 = 4 \times q^2 \therefore p^2 = 2 \times n^2 \tag{2.2}$$

Equation 2.1 shows that q is even; equation 2.2 proves that p is even too. But this is against our assumption that p and q are not both even. Therefore, p and q cannot be integers in equation 2.1, which you can rewrite as
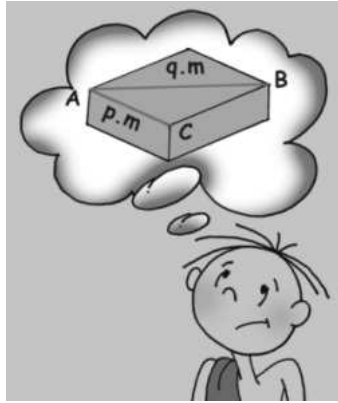
$$\frac{p}{q} = \sqrt{2}$$

The number $\sqrt{2}$, that gives the ratio between the side and the diagonal of any square, is not an element of Q, or else, $\sqrt{2} \notin Q$. It was Hypasus of Metapontum, a Greek philosopher, who proved this. The Greeks were a people of wise men and women. Nevertheless they had the strange habit of consulting with an illiterate peasant girl at Delphi, before doing anything useful. Keeping with this tradition, Hyppasus asked the Delphian priestess— the illiterate girl— what he should do to please Apollo. She told him to measure the side and the diagonal of the god's square altar using the same meter. By proving that the problem was impossible, Hypasus discovered a type of number that can not be written as a fraction. This kind of number is called irrational. By the way, an irrational number is not a crazy, or a stupid number; it is simply a number that you cannot represent as *ratione* (fraction, in Latin).

Figure 2.2: A thinking Greek

2.2.1 Real Numbers

The set of all numbers, integer, irrational, and rational is called R, or the set of real numbers. In the Bigloo number tower, there are quite a few sets that play the role of Z; the most widely used option is **bint**, or Bigloo integer. For real numbers, you can use Bigloo types real ou double. When you define functions, Bigloo allows for type declaration, i.e., you can say to which set your data belongs. Below you will find the definition of a function that takes real arguments, and produces a real value.

```
(module nums (main main))

(define (main args)
   (print(func 3.0 2.0)))

(define func
 ( lambda::real (x::real y::real)
   (sqrt (+ (* x x) (* y y))) ))
```

Scheme compiles this function, even without type declarations, i.e., even when you do not inform that the function takes real arguments, and returns a real value; if there are type errors, Bigloo reports them only at runtime. However, in languages like Clean and OCAML, the compiler requires you to provide enough information for it to type check your program.

If $x \in$ **Integer**, Bigloo programmers say that x has type **bint**. They also say that r has type **real** if $r \in$ **Real**. There are other types besides **bint**, and **real**. Here is a list of primitive types.

bint — Integer numbers between -2147483648 and 2147483647.

real — Reals must be written with a decimal point: 3.4, 3.1416, etc.

string — A quoted of characters: `"3.12"`, `"Hippasus"`, `"pen"`, etc.

char — Characters: `#\A`, `#\b`, `#\3`, `#\space`, `#\newline`, etc.

2.3 Functions definitions

A function is a relationship between an argument and a unique value. Let the argument be $x \in B$, where B is a set; then B is called domain of the function. Let the value be $f(x) \in C$, where C is also a set; then C is the range of the function. Functions can be represented by tables, or lambda expressions. Let us examine each one of these representations in turn.

2.3.1 Tables

Let us consider a function that associates `#t` or `#f` to the letters of the Roman alphabet. If a letter is a vowel, then the value will be `#t`; otherwise, it will be `#f`. The range of such a function is `{#t,#f}`, and the domain is `{#\a,#\b,#\c,#\d,... etc}`.

Domain	Range	Domain	Range	Domain	Range	Domain	Range
a	#t	g	#f	m	#f	t	#f
b	#f	h	#f	n	#f	u	#t
c	#f	i	#t	o	#t	v	#f
d	#f	j	#f	p	#f	w	#f
e	#t	k	#f	q	#f	x	#f
f	#f	l	#f	r	#f	y	#f
				s	#f	z	#f

2.3.2   Lambda expressions

From what you have seen in the last section, you certainly notice that it is pretty tough to represent a function using a table. You must list every case. There are also functions, like sin x, whose domain has an infinite number of values, which makes it impossible to list all entries. Even if you were to try to insert only a finite subset of the domain into the table, it wouldn't be easy. Even so, in the past, people used to build tables. In fact, tables were the only way to calculate many useful functions, like sin x, log x, cos x, etc. In 1814 Barlow published his Tables which give factors, squares, cubes, square roots, reciprocals and hyperbolic logs of all numbers from 1 to 10000. In 1631 Briggs published tables of sin functions to 15 places and tan and sec functions to 10 places. I heard the story of a mathematician who published a table of sinus, and made a mistake. Troubled by the fact that around a hundred sailors lost their way due to his mistake, that mathematician committed suicide. This story shows that the use of tables may be hazardous to your health.

A lambda expression is a kind of rule proposed by the American philosopher Alonzo Church. In order to understand how to represent a function with a lambda expression, let us revisit the vowel table. Using a lambda expression, that table becomes

```
(define vowel?
  (lambda(x)
    (if (member x '(#\a #\e #\i #\o #\u)) #t
        #f)))
```

The if-clause says that a letter is a vowel if it is a member of the set (#\a #\e #\i #\o #\u); otherwise, it is not a vowel.

Functions have a parameter, also called variable, that represents an element of the domain. Thus, the `vowel?` function has a parameter `x`, that represents an element of the set {#\a,#\b,#\c,#\d,... etc}. The variables are introduced by the keyword `lambda`. Next to the keyword `lambda`, and its variables, one finds a sequence of expressions that form the body of the function.

Now, let us consider the Fibonacci function, that has such an important role in the book "The Da Vinci Code". Here is its table for the first 6 entries:

0	1		3	3
1	1		4	5
2	2		5	8

Notice that a given functional value is equal to the sum of the two precedent values, i.e., fib $n = \text{fib}(n-1) + \text{fib}(n-2)$. Assume that $n = 6$. Then,

$$\text{fib } 6 = \text{fib}(6-1) + \text{fib}(6-2) = \text{fib } 5 + \text{fib } 4$$

Of course, this clause is true only for $n > 1$, since there are not two precedent values for $n = 0$, or $n = 1$.

```
:; File: fb.scm
;; Compile:
;;  bigloo fb.scm -o fb

(module fibo (main main))

(define (main args)
  (print(fib
     (string->number
        (cadr args)))))

(define fib
  (lambda::bint (n::bint)
    (if (< n 2) 1
       (+ (fib (- n 1))
          (fib (- n 2)))
) ) )
```

Figure 2.3: Fibonacci function

Listing 2.3 shows how you can state that an expression is valid only under certain conditions. For instance, `fib` returns the value 1 if and only if the condition `(< n 2)` is met; else it returns the result of the expression

```
(+ (fib (- n 1))
   (fib (- n 2))
```

From the discussion on page 11, you can infer that the program of listing 2.3 calculates an element of the Fibonacci's sequence from an integer present in the command line. Compile the program:

```
bigloo fb.scm -o fb
```

Type the following command line:

```
fb 8
```

You will get the eighth element of the Fibonacci sequence.

From page 11, you know that the problem with programs such as the one shown in figure 2.3 is that there is no room for mistakes. If you forget to provide an argument to the program, or if what you provide is not a valid integer, the program will generate a run time error. Therefore, when you write a Scheme program, you must be very careful to trim out all possibilities of usage error. On page 12, you saw that trimming error possibilities can be quite annoying. You must use a `cond`-form, and check for things like forgetting arguments, passing an invalid number, passing a number out of range, etc. In the present case, you must also check whether the number is

```scheme
;; File: fb.scm
;; Compile: bigloo fb.scm -o fb

(module nums (main main))

(define (main args)
  (print
    (with-handler
       (lambda(exc) "Usage: fb <Valid Number>")
       (fib  (string->number (cadr args)))
    ); close with
  );close print
);close define

(define fib
  (lambda::bint (n::bint)
    (if (< n 2) 1
       (+ (fib (- n 1))
          (fib (- n 2)))
      )
  )
)
```

Figure 2.4: Error handler

an integer or not. Happily enough, Scheme has a more practical alternative to analyse[1] exceptions caused by input errors.

In figure 2.4, the form **with-handler** wraps the expression

    (fib (string->number (cadr args))).

This form has two arguments. The first argument must be a lambda expression that produces a value for exceptions, i.e., for error situations; the second argument is the expression you want to calculate. If everything works

---

[1]This word comes from the Greek word ἀνάλυσις; therefore it must be spelled with an S, not with Z.

according to the script, the second argument will produce an element from the Fibonacci's sequence; if there is a mistake, any mistake at all, the **with-handler** form will issue the following help line:

```
Usage: fb <Valid Number>
```

2.3.3   Generic functions

I do not like object oriented programming, but let us talk about it anyway. Suppose that you want to create types of your own. In this case, you need to propose a domain for your functions other than double, bint, etc. In figure 2.5, you can see how to create a set of triangles, and a set of rectangles. What people think it is interesting in this kind of program, is that you can define a function that exhibits a different behavior for a different type of argument. In the case of the program shown in figure 2.5, function area uses different methods for calculating the area of a rectangle, and the area of a triangle. For instance, in the case of a triangle object, it uses the following method:

```
(define-method (area t::triangle)
   (with-access::triangle t (base altitude)
      (* 0.5 base altitude)))
```

The **with-access::triangle** form, as its name indicates, gives access to the values stored in the fields **base** and **altitude**, that provide the dimensions of the triangle **t**. The declarations

```
(static (class triangle
            base::double
            altitude::double)
         (class rectangle
            width::double
            height::double))
```

are enogh for Bigloo to provide the functions **make-rectangle**, and **make-triangle**, that create triangles, and rectangles respectively. By the way, to use this program, you must type a command line that looks like the one shown below.

```
area 20.0 40.0
```

```scheme
;; File: area.scm
;; Compile: bigloo area.scm -o area

(module area (main main)
  (static (class triangle
                base::double
                altitude::double)
          (class rectangle
              width::double
              height::double)))

(define (main xs)
  (with-handler
     (lambda(e) (print "Usage: area 3.0 4.0"))
     (print "Rectangle: "
           (area (make-rectangle (fst xs) (snd xs)) ) "; "
           "Triangle: " (area (make-triangle  (fst xs) (snd xs)))
     );close print
  );close with
);close define

(define (fst xs) (string->number (cadr xs)))

(define (snd xs) (string->number (caddr xs)))

(define-generic (area geo::object))

(define-method (area t::triangle)
    (with-access::triangle t (base altitude)
       (* 0.5 base altitude)))

(define-method (area r::rectangle)
    (with-access::rectangle r (width height)
       (* width height)))
```

Figure 2.5: Defining classes

Chapter 3

Recursion

The mathematician Peano invented a very interesting axiomatic theory for natural numbers. I cannot remember the details, but the idea was the following:

1. Zero is a natural number.

2. Every natural number has a successor: The successor of 0 is 1, the successor of 1 is 2, the successor of 2 is 3, and so on.

3. If a property is true for zero and, after assuming that it is true for n, you prove that it is true for n+1, then it is true for any natural number.

Did you get the idea? For this very idea can be applied in many other situations. When they asked Myamoto Musashi, the famous Japanese Zen assassin, how he managed to kill a twelve year old boy protected by his mother's 37 samurais[1], he answered:

> *I defeated one of them, then I defeated the remaining 36. To defeat 36, I defeated one of them, then I defeated the remaining 35. To defeat 35, I defeated one of them, then I defeated the remaining 34...*
> *...   ...   ...*
> *To defeat 2, I defeated one of them, then I defeated the other.*

---

[1]The boy's father had been killed by Musashi. His uncle met the the same fate. His mother hired her late husband's students to protect the child against Musashi.

3.1   Recursive functions

A close look will show that the function **app** of Listing 3.1 acts like Musashi.
The first clause of the *cond* form says: If xs is empty, then the result of
appending xs with ys is ys itself. The second clause deals with the general
case, i.e., the case when ys is not empty. In this case, one must add (car xs)
to the result of appending (cdr xs) with ys.

```
;; File: rec1.scm
(define (app xs ys)
   (cond ( (null? xs) ys)
         (else (cons (car xs)
                     (app (cdr xs) ys)) )
   );close cond
); close define
```

Figure 3.1: A recursive function

To test the program of figure 3.1, you may want to enter the interpreter.
From the command prompt, type `Bigloo`. Then, load the file `"rec1.scm"`.
Below, you can see the result of the test.

```
1:=> (load "rec1.scm")
app
rec1.scm
1:=> (app '(1 2 3) '(a b c d))
(1 2 3 a b c d)
1:=>
```

Let us pick a concrete instance of the problem. Assume that you want to ap-
pend '(1 2 3) to '(a b c d). Since '(1 2 3) is not empty, (null? xs) will
return #f, and the first condition will fail. Therefore, the program will select
the second condition. Since xs=(1 2 3), (car xs)= 1 and (cdr xs)=(2 3),
one has

```
(cons (car xs) (app (cdr xs) ys))=
        (cons 1 (app (cdr '(2 3))  '(a b c d)))
```

In the second call to *app*, one has `xs=2`, and `ys=(3)`, which lead us to

```
(cons 1 (cons 2 (app (cdr '(3))  '(a b c d))))
```

At this point, one has `(car xs)=3` and `xs= '( )`, which produces

```
(cons 1 (cons 2 (cons 3 (app '()  '(a b c d)))))
```

In the last call to *app*, one has `xs='( )`, and the cond-form will select its first clause, that will make `(app '( ) ys)= ys= '(a b c d)`. Therefore the program arrives at the following conclusion:

```
 (cons 1 (cons 2 (cons 3 '(a b c d))))= '(1 2 3 a b c d)
```

The scheme of listing 3.1 provides a solution to any problem, where one must apply a two place operation between the elements of a list. In the case of appending lists, the two place operation is `cons`, and the result of the last call is `ys`. If one must add a list, the two place operation is `+` and the last call must produce 0.

```
;; File: rec2.scm
(define (sum xs)
   (cond ( (null? xs) 0)
         (else (+ (car xs) (sum (cdr xs))) )
   );close cond
); close define
```

One can capture this pattern in a function that, traditionally, is called `fold`. Here is how to define it:

```
;; rec4.scm

(define (fold f2 xs a)
   (cond ( (null? xs) a)
         (else (f2  (car xs)
             (fold f2 (cdr xs) a)) )
   );close cond
); close define

(define (sum xs) (fold + xs 0))
(define (app xs ys) (fold cons xs ys))
(define (app-strings xs) (fold string-append  xs  ""))
```

If you have `fold`, it is a cinch to define a function that appends lists. You can just as easily define a function to add the elements of a list, or append a list of strings, as you can see for yourself in the above listing. You will find below a test for the fold-based definitions of sum, app, and app-strings.

```
1:=> (load "rec4.scm")
rec4.scm
1:=> (sum '(2 3 4))
9
1:=> (app-strings '("one " "two " "three"))
one two three
1:=>
```

3.2   Classifying rewrite rules

Typically a recursive definition has two kinds of conditions:

1. Trivial cases, which can be resolved using primitive operations.

2. General cases, which can be broken down into simpler cases.

Let us classify the two equations of **app**:

`( (null? xs) ys)`	The first condition is certainly trivial
`(else (cons (car xs)` `        (app (cdr xs) ys)))`	The second condition can be broken down into simpler operations: Appending two lists with one element removed from the first one, and inserting the element left out into the result.

Chapter 4

Input/Output

Scheme has input/output devices that are both convenient and productive. Input/ouput can be performed into and from a *port*. Ports are devices that can be associated with the console, files or strings.

4.1 Reading from a port

Let us consider the three possibilities. Enter the interpreter.

```
Welcome to the interpreter              ;;Greetings!
1:=> (define st "327 (Lili Marlene)")   ;; Store a string in st
st
1:=> (define pin (open-input-string st)) ;;Open an input port
pin
1:=> (read pin)                         ;;Read from pin
327                                     ;; A number read from
                                        a string

1:=> (read pin)
(Lili Marlene)
1:=> (read pin)
#eof-object                             ;;Nothing else to read.
```

4.2   Writing to a port

In the above example, I opened a string to read from. I can also open an output string. Let us see how to do it.

```
1:=> (define ws (open-output-string))
ws
1:=> (display "Think!" ws)
#<output_string_port>
1:=> (newline ws)
#<output_string_port>
1:=> (write "Le Penseur" ws)
#<output_string_port>
1:=> (close-output-port ws)
Think!
"Le Penseur"
1:=> ws
#<output_string_port>
1:=>
```

As you can see from the example, the difference between **display** and **write** is in the handling of strings; **write** prints the string with double quotes.

4.3   Standard input/output

In general, if you read something from the standard input port, the reading function does not take an argument; the same happens if you write things into the standard output port, i. e., the output function does not take an argument. Let us see an example.

```
1:=> (let ( (x (read)))
        (print "Square: " (* x x))
        (display "sin(x)= ")
        (display (sin x))
        (newline))
3.1416
Square: 9.86965056
sin(x)= -7.3464102066436e-6
#<output_port:stdout>
```

4.4 Reading and writing to files

In Scheme, reading things from a file, and writing things into a file is as easy as doing it with strings.

```
1:=> (define pout (open-output-file "scratch.txt"))
pout
1:=> (write "Hello, world" pout)
#<output_port:scratch.txt>
1:=> (newline pout)
#<output_port:scratch.txt>
1:=> (display (/ 3.1416 2) pout)
#<output_port:scratch.txt>
1:=> (close-output-port pout)
#<output_port:scratch.txt>
1:=> (define ip (open-input-file "scratch.txt"))
ip
1:=> (read ip)
Hello, world
1:=> (read ip)
1.5708
1:=> (read ip)
#eof-object
1:=> (close-input-port ip)
#<input_port:scratch.txt.22>
1:=>
```

The **print** command is the only output tool that works only with the console. It prints its arguments, and also a new line as bonus. The **read** command inputs any Scheme data structure from the console or from files.

4.5 Automatic port closing

The procedures call-with-input-file and call-with-output-file are very handy, because they take care of opening and closing a port after you're done with it. The procedure call-with-input-file takes as parameters a filename, and a single argument lambda expression. The lambda expression is applied to an input port opened on the file.

```
----------------------------------------------------------------
Bigloo (2.7a)                                            ,--^,
'a practical Scheme compiler'                         _ ___/ /|/
Sun Nov 27 15:05:30 RST 2005                      ,;'( )__, ) '
Inria -- Sophia Antipolis                         ;;  //   L__.
email: bigloo@sophia.inria.fr                     '   \   /  '
url: http://www.inria.fr/mimosa/fp/Bigloo             ^   ^
----------------------------------------------------------------


Welcome to the interpreter

1:=> (call-with-output-file "scratch.txt"
        (lambda(out)
           (write "Hello, world" out) (newline out)
           (display 327 out)
           (newline out)))
#<output_port:scratch.txt>
1:=> (call-with-input-file "scratch.txt"
        (lambda(in)
           (print (read in))
           (print (read in))
           (print (read in)) ))
Hello, world
327
#eof-object
#eof-object
```

4.6   Thunks

A thunk is a zero-argument lambda expression. Thunks are mostly used to
delay the evaluation of an expression. For instance, the thunk

```
(lambda() (print "Hello, world!"))
```

freezes the **print** function, that will produce output only after the evaluation
of the thunk.

   Scheme has forms that take a file name and a thunk as parameters, and
perform all input/output operations into or from the file. Of course, you can
substitute a string for the file.

```
1:=> (with-output-to-file "scratch.txt"
        (lambda() (print "Hello")
          (print (* 3 4 5)) ))
60
1:=> (with-input-from-file "scratch.txt"
        (lambda() (print (read))
          (print (read)) ))
Hello
60
60
1:=> (define str (with-output-to-string
        (lambda() (write "Hello!") (display "  ")
          (write "Hi, World!"))))
str
1:=> str
"Hello!"  "Hi, World!"
1:=> (with-input-from-string str (lambda() (read)))
Hello!
```

4.7 Reading C strings

There are occasions that one needs to read strings according to the convention used in the C programming language. For instance, this is the case when you need to retrieve a string from a GTK text buffer. To deal with such a situation, Bigloo offers the function **open-input-c-string**, whose usage is similar to the usage of **open-input-c-string**. We will test this procedure with an input/output function that we have not tried yet, to wit, **read-line**, which does what it means, i.e., reads a line from the input port.

```
1:=> (define pin (open-input-c-string "Hello,\n World!"))
pin
1:=> (read-line pin)
Hello,
1:=> (read-line pin)
 World!
```

In the programming language C, the code "\n" means #\newline; for this reason, (read-line pin) found two lines in the string "Hello,\n World!".

4.8   More input/output functions

Besides **read**, **write**, **display**, **print**, and **newline**, Scheme has other input/output functions. You have already seen one of them, to wit, **read-line**. A more complete list is given below.

- **(read-lines *port*)** accumulates all the lines of an input port into a list.

```
1:=> (with-output-to-file "scratch.txt"
         (lambda() (print "Hello, world!")
            (print "Hi, folks!")))
Hi, folks!
1:=> (with-input-from-file "scratch.txt"
         (lambda() (read-lines)))
(Hello, world! Hi, folks!)
```

- **(read-char *port*)** reads a char from a port.

- **(eof-object? *obj*)** recognizes whether the result of an input operation is the *end of file* object.

```
1:=> (with-input-from-file "scratch.txt"
        (lambda() (do ( (c (read-char) (read-char)))
              ( (eof-object? c) #t)
              (display c) )) )
Hello, world!
Hi, folks!
#t
```

Of all input procedures that we have discussed, the most powerful by far is **(read)**, since it can parse any Scheme object. However, there are occasions when you need to deal with objects whose syntax does not conform to the conventions of the Programming Language Scheme. For instance, you may need to read a period written in a natural language, like Botanical Latin, or a mathematical equation, or even a LaTeX document. For these occasions, Bigloo has to offer a very powerful piece of heavy artillery, which we will study in the next section.

4.9 Reading with grammars

Let us assume that one needs to read a line of Botanical Latin. Of course, we could cheat a little, and convince the botanist to write his sentence in the form of a Scheme list. However, we won't do that. Instead, we will write a grammar that will build a list, and recognize the punctuation marks as they are used in English and Latin. Let me elaborate on this point. Scheme syntax requires that every token be separated by spaces from the previous and next token. Latin and English do not allow spaces between punctuation marks and the previous token. Therefore, we need a tokenizer that acts accordingly to the rules of Latin.

```
(define *g*
   (regular-grammar ()
      ( (+ (or #\tab #\space)) (ignore) )
      ( (+ (out #\newline #\space #\tab ",.;?!"))
        (cons 'word (the-string)))
      (#\newline 'nl)
      (#\, 'VG)
      (#\. 'PT)
      ( (in ";?!")
        (cons 'pmark (the-string)))  ))
```

The tokenizer is a regular grammar that has a finite set of rules. Let us consider the first rule of the Botanical Latin tokenizer.

```
( (+ (or #\tab #\space)) (ignore) )
```

It says that if one finds a `#\tab` or a `#\space`, you can ignore it. The plus sign means at least one, possibly more than one `#\space` (or `#\space`). If the plus sign means *at least one*, the asterisk means zero or more instances of a char. Let us go to the next rule.

```
( (+ (out #\newline #\space #\tab ",.;?!"))
        (cons 'word (the-string)))
```

It says that the parser should build a token (the-string), if it finds one or more chars that is not one of the following: `#\newline`, `#\space`, `#\tab`, or `",.;?!"`. The token produced by this rule will be classified as a word by the expression: `(cons 'word (the-string))`

Let us suppose that the token is `"fulgida"`. In this case, the token will
be (`word . "fulgida"`). You have noticed that there is a dot separating
the **car** from the **cdr** of this list. Whenever a list ends in something different
from the empty list `'( )`, you must use a dot to separate the final element
from the others. The rules

```
(#\newline 'nl)
(#\, 'VG)
(#\. 'PT)
```

say that if the tokenizer finds newline, comma or period, it must produce the
tokens `nl`, `VG`, or `PT` respectively. The last rule

```
( (in ";?!")
        (cons 'pmark (the-string)))
```

says that if the tokenizer finds a punctuation mark, it should produce some-
thing like (`pmark . "?"`).

The tokenizer separates one token from the other, and classifies the strings
that it recognizes. Usually it follows a very simple grammar, that Chomsky
called regular. What you learned about the tokenizer is enough for most
applications. The next step is to do the parsing of a list of tokens.

As I told you before, and repeat here, If you want only to read a list of
words, you do not need the heavy artillery that grammars offer you. The
read function is enough to handle Scheme lists. What we are dealing with is
the case of lists that follow rules that are alien to Scheme.

In order to parse a list of tokens, we need an `Lalr` grammar. The rules
of `Lalr` grammars have the following shape:

```
( rule-label
        ((pattern1) (rewriting-expression1))
        ((pattern2) (rewriting-expression2))...)
```

The grammar that builds lists has a single rule, whose label is **words**,
as you can see in figure 4.1. The first clause of this rule says that, if the
parser finds the pattern (), it must rewrite it as the empty list `'( )`. The
second clause says that the parser rewrites the pattern (`word words`) as
(`cons word words`). One must pass both the tokenizer and the parser to the
read/lalrp function in order to read a text that follows the given grammar.

```
;; Compile: bigloo parser.scm -o pars

(module countword (main start-here))

(define *g*
   (regular-grammar ()
      ( (+ (or #\tab #\space)) (ignore) )
      ( (+ (out #\newline #\space #\tab ",.;?!"))
        (cons 'word (the-string)))
      (#\newline 'nl)
      (#\, 'VG)
      (#\. 'PT)
      ( (in ";?!")
        (cons 'pmark (the-string))) ))

(define *gram*
   (lalr-grammar
      (word pmark VG PT nl)
      (words (() '())
            ((word words) (cons word words))
            ((pmark words) (cons pmark words))
            ((PT words) (cons "." words))
            ((VG words) (cons "," words))   )))

(define (start-here argsv)
  (let* ( (s (read-line))
          (pin (open-input-string s)))
     (write (read/lalrp *gram* *g* pin))))
```

Figure 4.1: Lalr grammar

In order to close this chapter, you may want to take a look at one of the examples provided in the Bigloo manual. The example implements a calculator that accepts input in the normal, algebraic notation of highschool math.

```scheme
;; Compile: bigloo expr.scm -o expr

(module expr (main start-here))

(define *g* (regular-grammar ()
      ( (+ (or #\tab #\space)) (ignore) )
      (#\newline 'nl)
      ( (+ digit) (cons 'const (string->number (the-string))))
      (#\+ 'plus)
      (#\- 'minus)
      (#\* 'mult)
      (#\/ 'div)
      (#\( 'lpar)
      (#\) 'rpar) ))

(define *gram* (lalr-grammar
      (nl plus mult minus div const lpar rpar)
      (lines
         (())
         ((lines expression nl)  (print "--> " expression) )
         ( (lines nl) ) )
       (expression
           ( (expression plus term)  (+ expression term))
           ( (expression minus term) (- expression term))
           ( (term)                  term))
       (term
          ( (term mult factor)       (* term factor))
          ( (term div factor)        (/ term factor))
          ( (factor)                 factor))
       (factor
          ( (lpar expression rpar)    expression)
          ( (const)                   const))) )

(define (start-here argsv)
   (let* ((s (read-line) )
           (pin (open-input-string (string-append s "\n"))))
     (write (read/lalrp *gram* *g* pin) )
     (reset-eof pin) ))
```

Chapter 5

Loop Schemes

A loop is a repetion of a sequence of operations. Each repetion is called an iteration, a word that comes from the Latin root for path. Therefore, iteration means *walking in circles*. However, in a well designed loop, an iteration is not exactly equal to the previous one, but a step towards a goal; although a step of a stairway looks alike the previous one, it represents further progress to the top. A loop has the following components:

Accumulators are variables used to build the answer, or to gauge progress. Consider the following do-loop that calculates the factorial of 5:

```
1:=> (do ( (i 1 (+ i 1)) (acc 1 (* acc i)) )
         ( (> i 5) (newline) acc)
       (display (list i acc)) )
(1 1)(2 1)(3 2)(4 6)(5 24)
120
```

There are two accumulators, `i` and `acc`. The accumulator `i` gauges progress towards the goal, that is the factorial of 5; in fact, we know that we have reached the goal when the predicate `(> i 5)` becomes `#t`.

Halting conditions are predicates used to stop the iterations, after reaching the goal. In the case of the do-loop that calculates the factorial of 5, the only halting condition is `(> i 5)`.

Body is the part of the iterative process that contains the instructions that the program must repeat. In the case of the example, the body is `(display (list i acc))`.

When designing loops, one must be sure that at least the accumulator that will be tested for halting progress towards the goal, and that the halting condition recognizes that one has reached the goal. It is also necessary that the accumulators build the solution and the halting condition incrementally; the variable that builds the halting condition is called the control variable; the stepwise construction of the solution is called *induction*. Late Professor Doris of Aragon used to say that there are two very important classes of induction, to wit

- Induction over natural numbers. This kind of induction is very important historically because it was treated for the first time in a book that is the landmark of Formal Logic; I am talking about the small volume that Giuseppe Peano wrote in Latin: *Arithmetices principia nova methodo exposita*. In this kind of induction, the control variable is a natural number, and is usually modified by adding an integer to it.

- Induction over the length of a list. The control variable is a list, and it is usually modified by taking its `cdr`.

The halting condition for induction is also called the trivial case. This denomination comes from Mathematics, where a trivial case has a straightforward demonstration. As a curiosity, let us show how Peano would prove that the sum of the first n natural numbers can be written as

$$\sum_{i=0}^{n} i = \frac{n \times (n+1)}{2} \tag{5.1}$$

- The trivial case is n=0. It is easy to show both that the sum is zero, and that the formula holds.

- Let us assume that the formula holds for n, and prove that it holds for (n+1).

$$\sum_{i=0}^{n+1} i = (n+1) + \sum_{i=0}^{n} i = \frac{2n + 2 + n \times (n+1)}{2} = \frac{(n+1)(n+2)}{2}$$

where we have used the equality $(n+1) \times (n+2) = n^2 + 3n + 2$, and equation 5.1, in order to perform the two final steps of the demonstration. It was possible to use equation 5.1, because we assumed that the formula holds for n.

There are people that find it difficult to accept Peano's rule of inference. Therefore, let us linger a little on it. Equation 5.1 is true for n=0, since this is the trivial case of our demonstration. It is also true for n=1, since it is true for n=0, and we have demonstrated that it is true for (n+1) whenever it is true for n. It is also true for n=2, since it is true for n=1, and we demonstrated that it is true for (n+1), if it is true for n. Got the idea?

In the Algorithm Language Scheme there are three ways to perform interations, and inductions, to wit:

do-loop Below you will find how to define factorial using a do-loop.

```
1:=> (define (fact n)
        (do ( (i 1 (+ i 1)) (acc 1 (* i acc)) )
          ( (> i n) acc) ) )
fact
1:=> (fact 5)
120
```

Named let

```
1:=> (define (factorial n)
        (let iter ( (i 1) (acc 1) )
          (if (> i n) acc
            (iter (+ i 1) (* i acc)) )) )
factorial
1:=> (factorial 5)
120
```

Tail recursive function

```
1:=> (define (tail-fact i n acc)
        (cond ( (> i n) acc)
              (else (tail-fact (+ i 1) n (* i acc)) )) )
tail-fact
1:=> (define (fact n) (tail-fact 1 n 1))
fact
1:=> (fact 5)
120
```

5.1   Tail recursive functions

Let us consider the following definition for the factorial function.

```
(define (rec-fac n)
  (cond ((< n 2) 1)
        (else (* n (rec-fac (- n 1)) )))) )
```

Let us use this snippet to build the factorial of 3, step by step.

1. (`rec-fac 3`); in this case, n=3, and does not meet the halting condition. Therefore, the program must use the expression

   ```
   (* n (rec-fac (- n 1)) )
   ```

   in order to calculate the factorial. Since n=3, this expression becomes
   (`* 3 (rec-fac 2)`). Before performing the multiplication, Scheme
   must find the value of (`rec-fac 2`), which can be done by calling
   rec-fac again. However, to be able to return to the expression

   ```
   (* n (rec-fac 2)), with n=3
   ```

   after finding the value of (`rec-fac 2`), Scheme must store both the
   address of the above expression, and the value of n, which happens to
   be 3. Experts say that Scheme must store the state of the computation
   of $(* n_{=3} (\text{rec-fac} (- n_{=3} 1)))$ in a stack of pending calculations before
   proceeding to the calculation of (`rec-fac 2`).

2. In this step, Scheme tries to calculate the expression (`rec-fac 2`), and
   faces the same difficulties as in the previous iteration. The expression
   (`rec-fac 2`) chooses the else-branch of the **cond** and is reduced to

   ```
   (* n (rec-fac 1), with n=2
   ```

   Again, it is necessary to store the state of the computation in the stack
   of pending calculations before proceeding to (`rec-fac 1`).

3. The expression (`rec-fac 1`) does meet the halting condition, since
   n=1, which is less than 2. Therefore (`rec-fac 1`)=1.

Now that Scheme knows that `(rec-fac 1)=1`, it must use this result to find the value of `(rec-fac 2)`, whose expression was pushed to the stack of pending calculations in step 2. Poping the expression `(* 2 (rec-fac 1))` from the stack, Scheme finds that `(rec-fac 2)=2`. Finally, Scheme pops the last expression from the stack of pending calculations,

    (* 3 (rec-fac 2))

and finds the factorial of 3: `(rec-fac 3)= (* 3 (rec-fac 2))= 6`. The table below summarizes what has been said.

Call	Expression	Stack of pending operations
(rec-fac 3)	(* 3 (rec-fac 2))	
(rec-fac 2)	(* 2 (rec-fac 1))	(* 3 (rec-fac 2))
(rec-fac 1)	1	(* 2 (rec-fac 1)) (* 3 (rec-fac 2))
(* 2 (rec-fac 1))	= (* 2 1)	(* 3 (rec-fac 2))
(rec-fac 2)	=2	
(* 3 (rec-fac 2))		
(* 3 (* 2 1))	=6	

Now, consider the program below, where the function `fact-aux` has an extra parameter that accumulates the product necessary to calculate the factorial.

```
(define (fact-aux i n acc)
   (cond ( (> i n) acc)
         (else (fact (+ i 1) n (* i acc)) )) )

(define (fact n) (fact-aux 1 n 1))
```

1. (`fact 3`) — This call is expanded to (`fact-aux 1 3 1`)

2. (`fact-aux i 3 acc`), with i=1, acc=1 — Since the value of `i` does not satisfy the halting condition, the call is expanded to the else-expression: (fact 2 3 (* 1 1)). No pending operation is left behind.

3. (`fact i 3 acc`), with i=2, acc=1 — Once again, the call is expanded to the else-expression: (fact 3 n (* 1 1 2)). No pending operation is left in the stack.

4. (`fact i 3 acc`), with i=3, acc=2. For the third time, the call is expanded to (`fact 4 3 (* 1 1 2 3)`)

5. (`fact i 3 acc`), with i=4, acc=(* 1 1 2 3)=6. Now the halting condition is satisfied and the program returns the accumulated solution.

5.2   Named let

Programs like the last one, that do not leave pending calculations behind, are said to be tail recursive. They can be used to implement very efficient loops. However, they have the inconvenience of requiring a separate definition whenever one needs a loop. A named-let allow us to create a loop definition in place, and initialize the control variables at the same time.

```
1:=> (define (fact n)
        (let iter ( (i 1) (acc 1))
           (if (> i n) acc
              (iter (+ i 1) (* acc i)) )))
fact
1:=> (fact 5)
120
```

Let us examine the anatomy of the named-let.

```
(define (fact n)

   (let iter
     [ (i 1)                              ⎫
       (acc 1)                            ⎬ ;;control vars are initialized
      ]                                   ⎭

       (if (> i n) acc ;; halting condition
           (iter (+ i 1) (* i acc)) ) ;; control updating
) )
```

5.3 Loops prête à porter

Besides tail recursive definitions, and the named-let, Scheme has a loop prête
à porter. It is called do-loop, and you already have had many opportunities
to use it. In this very chapter you have seen an example of do-loop, that I
repeat below for ready reference.

```
1:=> (define (fact n)
        (do ( (i 1 (+ i 1)) (acc 1 (* i acc)) )
           ( (> i n) acc)
         )
      )
fact
1:=> (fact 5)
120
```

As you can see, the do-loop is too general, and many people consider it
intimidating. It is for situations like this one that Scheme has the possibilities
of defining new syntax structures. In figure 5.1, you can see the definition of
a while-loop. The definition is achieved by a construction called macro. It
is composed of the keyword **define-syntax**, the name of the desired form (in
this case, the name is *while*), and a set of syntax rules. Each syntax rule has
a pattern and an expansion.

```scheme
;; Compile: bigloo -o tes whtest.scm

(module whiletest (main main))

(define-syntax while
      (syntax-rules()
         ( (while test body1 ...)
              (let loop ()
                  (cond
                    (test body1 ... (loop)))))))


(define (main argv)
  (let ( (i '(1 2 3)))
   (while (not (null? i))
      (print (car i))
      (set! i (cdr i)))   ))
```

<div align="center">

Figure 5.1: while-loop

</div>

For the case of the while-loop, the pattern is `(while test body1 ...)`, and the expansion is given by

```scheme
(let loop ()
        (cond
            (test body1 ... (loop))))
```

The sequence of three dots is called ellipsis, and indicates a repetition of the pattern element `body1`. In the main function, you can see an example of the newly defined while-loop in action.

Chapter 6

Pattern match

Most modern functional languages have pattern match. Scheme is by no means modern, but it is flexible enough to adapt itself to the changing times. For instance, consider the definition below.

```
(define (app xs ys)
  (match-case xs
     ( (?x . ?s) (cons x (app s ys)))
     ( () ys)))
```

As you can see, Bigloo offers special patterns to select the head and the tail of a list. The pattern (?x . ?s) matches any list with more than one element. When this happens, x matches the head, and y matches the tail. Let's consider a concrete case. If (?x . ?s) matches the list (5 2 6 3), x receives the value of 5, and s receives (2 6 3), which is the tail of the original list. Now, let us try the definition of app. Enter the interpreter.

```
Welcome to the interpreter

1:=> (define (app xs ys)
        (match-case xs
           ((?x . ?s) (cons x (app s ys)))
           ( () ys)))
app
1:=> (app '(1 2 3) '(a b))
(1 2 3 a b)
```

Be careful to insert a space before and after the dot in the pattern (?x . ?s). In general, all Scheme tokens are surrounded by spaces.

6.1 Quasiquote

A match rule has a pattern and a rewrite expression. For instance, in the
definition of `app` the pattern of the first rule is `(?x . ?y)`, and the rewrite
expression is `(cons x (app s ys))`. A quasiquote is a handier notation for
the rewriting expression. It works like a quote, but one can use a comma
to unquote an sub-expression. In this case, the unquoted sub-expression is
evaluated, and its value becomes a list element. Example:

```
Welcome to the interpreter

1:=> `(a b ,(* 3 4 5) 87)
(a b 60 87)
1:=>
```

There is also a tool to splice a sub-list into the quasiquoted list, as you can
see below.

```
1:=> `(a b ,@(list 4 (* 5 8) 6) 89)
(a b 4 40 6 89)
1:=>
```

In the above example, the expression `(list 4 (* 5 8) 6) 89)` is evaluated
to `(4 40 6)` and spliced into `(a b ... 89)`. What remains to be said is
that the quasiquote and the quotation symbols look alike, except for having
opposite inclinations. Compare carefully the two symbols:

Quotation symbol: `'(a b c)`

Quasiquotation symbol: `` `(a b c) ``

Let us now examine two ways of defining append using quasiquote. The first
way uses splice.

```
(define (app2 xs ys)
   (match-case xs
      ( (?x . ?s) `(,x  ,@(app2 s ys)) )
      ( () ys)))
```

The second way uses improper list, i.e., a list that does not end in '( ), and is written in dot-notation for this reason.

```
(define (app3 xs ys)
   (match-case xs
       ( ( ?x . ?s) `(,x . ,(app3 s ys)))
       ( () ys)))
```

6.2   Pattern combinations

The so called modern functional languages, like Clean, Haskell, and OCAML, adopt a style of programming that relies strongly on pattern match. Although Scheme does not encourage pattern match, its pattern language is more flexible, and powerful than the one found in Haskell or Clean. For one thing, Scheme has two ways of combining patterns:

(and pat1 pat2...) The `and`-combination succeeds if all patterns match.

(or pat1 pat2... The `or`-combination succeeds if one of its patterns match.

Other functional languages have patterns that are compared for equality. In Scheme, one can use any predicate to compare a pattern to an expression. For instance, if you want to check whether the first element of a list is an interger, and unify it with a variable being successful, you can use the following composite pattern:

```
((and (? integer?) ?x) . ?xs)
```

Let us define and test a function that uses this very same pattern to filter out the non-integer elements of a list.

```
1:=> (define (ints s)
       (match-case s
         ( ((and (? integer?) ?x) . ?xs) `(,x . ,(ints xs)))
         ( (?x . ?xs) (ints xs))
         ( ( ) '( )) ))
ints
1:=> (ints '(3 4 5.6 p q 9 g))
(3 4 9)
1:=>
```

6.3   Patterns that do not match

A very useful pattern modifier is **(not pat)**, that succeeds if its pattern does
not match. Let us test it with a completely useless function that removes all
instances of `a` from a list.

```
1:=>
(define (rem-a s)
   (match-case s
      ( ( (and (not a) ?x) . ?xs)
        (cons x (rem-a xs)))
      ( (?x . ?xs) (rem-a xs))
      ( ( ) '( )) ))
rem-a
1:=> (rem-a '(b c d a a g a h))
(b c d g h)
1:=>
```

6.4   Ellipsis

The pattern language has ellipse, like **define-syntax**. The example below
checks whether a list contains only integers.

```
1:=> (define (ints? xs)
        (match-case xs
           ( ( (? integer?) ...) xs)
           (?any #f)))
int?
1:=> (ints? '(3 4 5 6 7))
(3 4 5 6 7)
1:=> (ints? '(3 4.8 7))
#f
```

6.5   Anonymous pattern variable

The pattern `?-` matches anything. You can substitute it for pattern vari-
ables when you do not need to bind an identifier to the matched expression.

Chapter 7

String processing

Bigloo has very powerful string processing capabilities, although I miss a few
functionalities in this domain. In any case, let us test the interpreter for
what we got.

```
Welcome to the interpreter

1:=> (string? "Hello?")
#t
1:=> (define s1 (make-string 4))
s1
1:=> (begin (write s1) (newline) "")
"    "

1:=> (string-length s1)
4
1:=> (string-length "Hello!")
6
1:=> (make-string 3 #\a)
aaa
1:=> (string #\a)
a
1:=> (write (string #\a))
"a"#<output_port:stdout>
1:=> (begin (write (string #\a)) (newline))
"a"
#<output_port:stdout>
```

From the examples, one can infer the behavior of the string processing functions that we have tested. This behavior is summarized below.

(`string?  "Hello?"`) recognizes whether an object is a string or not.

(`make-string 4`) builds a space filled string of a given length.

(`write s1`) prints a string preserving the double quotes.

(`string-length s1`) provides the length of a string.

(`make-string 3 #\a`) builds a string of given length and char.

(`string  #\a `) builds a string from a char.

7.1   Strings: Indexed access

Strings are arrays of char. Being such, one can retrieve a string element of a given index. One can also replace an element for another at a given position.

```
Welcome to the interpreter

1:=> (define greetings "Hello!")
greetings
1:=> (string-ref greetings 0)
H
1:=> (string-ref greetings 1)
e
1:=> (string-ref greetings 5)
!
1:=> (string-set! greetings 0 #\h)
#unspecified
1:=> (begin (write greetings) #\newline)
"hello!"

1:=> (string-set! greetings 5 #\.)
#unspecified
1:=> greetings
hello.
1:=>
```

7.2 Strings: Type conversion

String can be used to visually represent many data types. Therefore, Bigloo provides tools that produce the string representation of objects like numbers and chars. There are also tools to obtain the object, given its string representation. The most general of these tools are `with-output-to-string` and `with-input-from-string`, in the sense that these functions can convert any Bigloo object to a string and from a string respectively.

```
Welcome to the interpreter

1:=> (define str (with-output-to-string
                    (lambda() (write '(a b c))) ))
str
1:=> (begin (write str) #\newline)
"(a b c)"
1:=> (define str "(3.4 2.8 7.5)")
str
1:=> (begin (write str) #\newline)
"(3.4 2.8 7.5)"

1:=> (car str)
*** ERROR:_car:
Type 'pair' expected, 'bstring' provided -- (3.4 2.8 7.5)
    0.interp
    1.engine
    2.main
1:=> (define lst (with-input-from-string str
                    (lambda() (read)) ))
lst
1:=> (car lst)
3.4
1:=> (car (cdr lst))
2.8
1:=> (begin (write lst) #\newline)
(3.4 2.8 7.5)

1:=>
```

From this example, you can see that (`car str`) raised an error condition, because `str` points to a string, that has no **car**. As for `lst`, it is the result of converting `str` to a list; therefore, one can apply the list selectors to `lst`.

Bigloo provides string conversion functions for its primitive types, like integers, numbers, and chars. A few examples will show you how to use them.

(**string–>number** *string*) converts a string representation to a number. If the conversion fails, this function returns `#f` (false).

```
1:=> (define str "34.56")
str
1:=> (+ str 4)
*** ERROR:#<procedure:1001e760.-1>:
not a number -- 34.56
    0.interp
    1.engine
    2.main
1:=> (set! str (string->number str))
#unspecified
1:=> (+ str 4)
38.56
1:=> (string->number "3u5")
#f
```

(**string–>integer** *string*) converts a string representation to an integer.

(**string–>real** *string*) converts a string representation to a real number.

(**number–>string** *nn*) converts a number to its string representation.

(**integer–>string** *nn*) converts an integer to its string representation.

(**real–>string** *nn*) converts a real number to its string representation.

(**string–>list** *string*) converts a string to a list of chars.

```
1:=> (define str "abc")
str
1:=> (begin (write (string->list str)) #\newline)
(#\a #\b #\c)
```

(**list—>string** *lst*) converts a list of chars to a string.

```
Welcome to the interpreter

1:=> (begin (write (list->string '(#\a #\b #\newline)))
            #\newline)
"ab\n"

1:=>
```

(**string-for-read** *str*) returns a copy of the string with each special char-
acter replaced by a escape sequence. This is a very useful function,
that I use a lot in the Bigloo bindings for the GTK-server in order to
read Scheme programs and put them into a text buffer. Since programs
are represented as strings, it is impossible to handle source code that
contains string constants without escape sequences.

```
Welcome to the interpreter

1:=> (define lst '(#\" #\a #\b #\" #\newline))
lst
1:=> lst
(" a b "
)
1:=> (begin (write lst) #\newline)
(#\" #\a #\b #\" #\newline)

1:=> (define str (list->string lst))
str
1:=> str
"ab"

1:=> (string-for-read str)
\"ab\"\n
1:=>
```

7.3   Strings: appending, splinting, comparing

There are many methods of appending and spliting strings. For one thing, Bigloo provides quite a few functions for handling these functionalities.

(**string-append** *str1 str2 str3...*) appends its arguments.

```
Welcome to the interpreter

1:=> (define voc "Oh")
voc
1:=> (define excl "!")
excl
1:=> (define comma ",")
comma
1:=> (define spc " ")
spc
1:=> (define catilina (string-append voc spc
                                "tempora" excl))
catilina
1:=> (begin (write catilina) #\newline)
"Oh tempora!"

1:=>
```

(**format**   "write style: ~s; display style:~a\n" *str1 str2*) The first argument of format specifies how the other arguments will be inserted into the formatted string. If the escape code is "~s", then the argument is inserted as it were printed by **write**; if the escape code is "~a", the argument is inserted as if printed by **display**. To obtain a newline, one can use the escape code "~%". Finally, if a logician needs a tilde to write comments on a medieval book written in Latin, s/he can use a sequence of two tildes in the format string.

```
1:=> (format "write style: ~s; display style: ~a\n"
        "ROSA, ROSAE" (* 3 4 5))
write style: "ROSA, ROSAE"; display style: 60

1:=>
```

Strictly speaking, `format` is unecessary, since one can use

```
1:=> (with-output-to-string
        (lambda() (display "write style: ")
                  (write "ROSA, ROSAE")
                  (display "; display style: ")
                  (display (* 3 4 5)) ))
write style: "ROSA, ROSAE"; display style: 60
```

in order to produce the same effect. However, there are people who prefer `format`, since it is similar to a functionality found in the C programming language.

Splitting can be obtained using a tool that we have already learned about in a previous chapter, i.e., `read/lalrp`. However, there are occasions when simpler tools are more practical than powerful ones. For these occasions, Bigloo offers the following functionalities:

(string-split *str*) splits a string into substrings delimited by spaces.

(string-split *str* delimiters) splits a string into substrings whose delimiters are given in the second argument. The second argument is a string, whose characters will serve as delimiters.

```
1:=> (string-split "rosa rosae rosae rosa rosam")
(rosa rosae rosae rosa rosam)
1:=> (string-split "doceo/docere/docui/doctum" "/")
(doceo docere docui doctum)
1:=>
```

(substring *str start end*) where *start* is greater than zero, and smaller or equal to *end*; *end* is greater or equal to *start*, and smaller than the length of the string. The result of this function is a newly allocated substring formed from the characters of *str* beginning at the index *start*, and stopping short of reaching the index *end*.

```
Welcome to the interpreter

1:=> (substring "ROSA ROSAE" 5 10)
ROSAE
1:=>
```

There are a few useful functions that one can use to obtain information
about a string, modify it, or make updated copies.

(string-contains *str s-str*) returns the index in *str* where *s-str* occurs first
as a substring. If *s-str* is not a substring of *str*, the result is `#f`.

```
Welcome to the interpreter

1:=> (string-contains "ROSAE" "ROSA ROSAE")
#f
1:=> (string-contains "ROSA ROSAE" "ROSAE")
5
```

(string-contains-ci *str s-str*) works like `string-contains`, but it is case
insensitive.

(blit-string! *src i str j L*) fills string *str* starting at index j with L char-
acters taken out from `src` from index `i`.

```
1:=> (define blck (make-string 10 #\-))
blck
1:=> blck
----------
1:=> (blit-string! "ROSA" 0 blck
              (- (string-length blck) 4) 4)
#unspecified
1:=> blck
------ROSA
```

(string-downcase *str*) returns a newly allocated version of *str*, where each
uppercase letter is replaced by a lowercase letter.

(string-downcase! *str*) modifies *str* with in loco substitution of lowercase
letters for uppercase ones.

(string-upcase *str*) returns a newly allocated version of *str*, where each
lowercase letter is replaced by an uppercase letter.

(string-upcase! *str*) modifies *str* with in loco substitution of uppercase
letters for lowercase ones.

The reader should pay attention to functions that destructively update strings. In Scheme, if a function performs destructive updates, its name contains an exclamation mark, that one reads *bang!* For instance, `set!` is called *set-bang*, and `string-upcase!` is denominated *string-upcase-bang*. If you think that you may need the original string, you must make a copy of it before utilizing a bang function.

```
1:=> (define fst-declension "rosa rosae rosae rosa rosam")
fst-declension
1:=> (define fst-copy (string-copy fst-declension))
fst-copy
1:=> fst-copy
rosa rosae rosae rosa rosam
1:=> (string-upcase! fst-copy)
ROSA ROSAE ROSAE ROSA ROSAM
1:=> fst-copy
ROSA ROSAE ROSAE ROSA ROSAM
1:=> fst-declension
rosa rosae rosae rosa rosam
```

Predicates are functions whose range is `{#t,#f}`. In general, Scheme predicates end with a question mark. Below, you will find a few string predicates.

(string=? *str1 str2*) returns #t if `str1` is equal to `str2`.
    (string-ci=? str1 str2) ignores case.

(string<? *str1 str2*) returns #t if `str1` comes before `str2`.
    (string-ci<? str1 str2) ignores case.

(string>? *str1 str2*) returns #t if `str1` comes after `str2`.
    (string-ci>? {\em str1 str2}) ignores case.

(string<=? *str1 str2*) returns #t if `str1` comes before or is equal to `str2`.
    (string<=? {\em str1 str2}) ignores case.

(string>=? *str1 str2*) returns #t if `str1` comes after or is equal to `str2`.
    (string>=? {\em str1 str2}) ignores case.

(string-null? *str*) returns #t if *str* is the null string.

7.4   My string wish list

Bigloo has a lot of string processing functions. Even so, I miss a couple
of string tools badly. However, instead of complaining about it, let us im-
plement my wish list; `(rep-first str s z)` replaces the first occurrence
of `s` with `z`; `(rep-all str s z)` replaces all occurences of `s` with `z`; and
`(string-insert str p z)` inserts `z` before the index `p`. You will find defi-
nitions of my favorite string functions below.

```
;; File: string-wishlist.scm

(define (string-insert str p z)
   (let ( (sz (string-length str)))
      (when (and (<= p sz) (>= p 0))
         (format "~a~a~a" (substring str 0 p) z
                          (substring str p sz)))))

(define (rep-first str s z)
   (pregexp-replace s str z))

(define (rep-all str s z)
   (pregexp-replace* s str z))

(define (replace-first str s z)
   (let ( (p (string-contains str s))
          (sz-str (string-length str))
          (sz-s (string-length s)) )
      (when p (format "~a~a~a" (substring str 0 p) z
                  (substring str (+ p sz-s) sz-str))
      )
   )
)
```

Function **(string-insert str p z)** checks whether the index **p** is greater than
0 and smaller than the string length. If the answer is yes, it splits the string
in two segments, one going from 0 to **p** (exclusive), and the other from **p**
(inclusive) to the string length (exclusive). Then **z** is sandwiched between
these two segments.

The function `(rep-first str s z)` calls `(pregexp-replace s str z)` to replace the first occurrence of `s` with `z`, while `(rep-all str s z)` makes use of `(pregexp-replace* s str z)` to replace all instances of `s` with `z`.

I have also defined `(replace-first str s z)`, that replaces the first occurrence of `s` with `z`, but does not use `(pregexp-replace s str z)`. It works exactly like `(rep-first str s z)`.

```
1:=> (load "string-wishlist.scm")
replace-first
string-insert
string-wishlist.scm
1:=> (string-insert "012345" 5 "ABC")
01234ABC5
1:=> (string-insert "012345" 6 "ABC")
012345ABC
1:=> (string-insert "012345" 0 "ABC")
ABC012345
1:=> (string-insert "012345" -1 "ABC")
#f
1:=> (replace-first "012345" "23" "-II-III-")
01-II-III-45
```

Both `pregexp-replace` and `pregexp-replace*` takes a regular expression pattern as first argument. Most of the characters of a regular expression pattern match instances of themselves in a string. If you type

```
    1:=> (pregexp-replace "a pattern"
                          "here is a pattern"
                          "an expression")
    here is an expression
    1:=> (pregexp-replace* "pat" "one pat, two pats" "exp")
    one exp, two exps
    1:=>
```

There are other regular expression functions besides `pregexp-replace` and `pregexp-replace*`. A very useful one is `pregexp-split`:

```
    1:=> (pregexp-split "and" "cats and dogs and horses")
    (cats   dogs   horses)
    1:=>
```

Another one is `pregexp-match-positions`, that returns the starting index (inclusive) and the ending index (exclusive) of the matching substring.

```
1:=> (pregexp-match-positions "ab" "12ab34")
((2 . 4))
1:=>
```

7.5   Posix regular expressions

Regular expressions is a codified method of searching proposed by Stephen Kleene, an American mathematician. We will use the function

(`pregexp-match-positions` *pattern string*)

in order to exemplify the main features of regular expressions. A regular expression is composed from literals, metacharacters, and escape sequences.

literal A literal is a character that stands for what it means, i.e., a character that matches itself in the search string.

```
1:=> (pregexp-match-positions "ab" "12ab34")
((2 . 4))
```

metacharacter A metacharacter is a special characters that has a unique meaning and is not used as literal in the search expression. For instance, `^` and `$` identify the begining and the end of the search string respectively. For example, if one wants to match `Linux` at the begining of a string, one must use the pattern `"^Linux"`, otherwise the search would succeed even if `"Linux"` were in the middle of the search string.

```
1:=> (pregexp-match-positions "^Linux" "Linux or Windows")
((0 . 5))
1:=> (pregexp-match-positions "^Linux" "Windows or Linux")
#f
```

escape sequence An escape sequence indicates that one wants to use a metacharacter as literal.

```
1:=> (pregexp-match-positions "\\^" "a*x^2+b*x +c=0")
((3 . 4))
```

Function `pregexp-match-positions` can take two extra-arguments that indicate the range within which the matching takes place.

```
1:=> (pregexp-match-positions "fl" "flip-flop" 2 9)
((5 . 7))
```

After this somewhat long introduction, let us see examples of the most useful metacharacters.

Square brackets match anything inside the brackets. A dash inside square brackets allows one to define a range. In the example, the range spans from 3 to 9.

```
1:=> (pregexp-match-positions "[0123-9]" "Bigloo (2.7a)")
((8 . 9))
1:=>
```

A caret inside square brackets negates an expression. Therefore, `"[^0-9]"` means anything, except digits.

```
1:=> (pregexp-match-positions "[^0-9]" "152=CLII")
((3 . 4))
```

A question mark matches the preceeding character 0 or 1 times only. As you can see below, `"colou?r"` will find both the American and the English spelling of *color*.

```
1:=> (pregexp-match-positions "colou?r" "many colours")
((5 . 11))
1:=> (pregexp-match-positions "colou?r" "many colors")
((5 . 10))
```

An asterisk matches the preceding character 0 or more times. For instance, `"tre*"` will match both `"tread"` (one `"e"`), and `"trade"` (zero occurrences of `"e"`).

```
1:=> (pregexp-match-positions "tre*" "tree")
((0 . 4))
1:=> (pregexp-match-positions "tre*" "trade")
((0 . 2))
```

A plus sign matches the previous character 1 or more times.

```
1:=> (pregexp-match-positions "tre+" "trade")
#f
1:=> (pregexp-match-positions "tre*" "tree")
((0 . 4))
```

A number in braces matches the preceding pattern a fixed number of
times. For instance, if one wants to pick the first three digits of a
telephone number, s/he could use the pattern below.

```
1:=> (pregexp-match-positions "\\d{3}-" "741-8657")
((0 . 4))
```

Parentheses groups parts of the pattern together. A vertical bar indicates
alternation. By the way, *alternation* is a Latin word that means *find
either the left hand or right hand side values.* The pattern below finds
both the American and the English spelling of *gray*.

```
 1:=> (pregexp-match-positions "gr(e|a)y" "grey is a color")
((0 . 4) (2 . 3))
```

Notice that `pregexp-match-positions` returned a list with both the
position of the main match (the word *grey*) and the range of the sub-
match (the vowel *e* that matches `"(a|e)"`). Submatches can be used
in the insert string argument of the procedure `pregexp-replace`, as
you can see below.

```
1:=> (pregexp-replace "(live)s to (eat)"
        "Socrates says that he lives to eat"
         "\\2s to \\1")
Socrates says that he eats to live
```

Chapter 8

Vector

A vector is a data structure that contains a fixed number of elements and provides indexed access to them; this means that if you have a vector and an index, you can recover or replace the element stored at the address indicated by the index, and this operation takes the same amount of time, regardless of the position of the element. Below you can see how to create a vector, store an element at a given index, and retrieve an element from a given position.

```
1:=> (define v5 (make-vector 5 0.0))
v5
1:=> v5
#(0.0 0.0 0.0 0.0 0.0)
1:=> (vector-set! v5 0 11)
#unspecified
1:=> v5
#(11 0.0 0.0 0.0 0.0)
1:=> (vector-set! v5 1 21)
#unspecified
1:=> v5
#(11 21 0.0 0.0 0.0)
1:=> (vector-set! v5 3 31)
#unspecified
1:=> (vector-ref v5 3)
31
1:=> (vector-ref v5 1)
21
```

From the example, you can see that a vector is a mutable data structure, i.e., it is possible to replace an element of a vector with a different value. In this respect, a vector is like a container, whose contents can be changed.

The main vector manipulation functions are **make-vector**, **vector-set!**, and **vector-ref**; if you have these three functions, you can do anything that is possible to do with a vector. However, there are a few other functions that may be useful to speed things up, and make your life easier.

```
1:=> (define v6 '#(1 2 3 4 5 6))
v6
1:=> (vector-ref v6 0)
1
1:=> (vector-ref v6 1)
2
1:=> (vector-length v6)
6
1:=> (vector->list v6)
(1 2 3 4 5 6)
1:=> (vector? v6)
#t
1:=> (define u6 (vector-copy v6))
u6
1:=> u6
#(1 2 3 4 5 6)
1:=> v6
#(1 2 3 4 5 6)
1:=> (vector-set! u6 4 71)
#unspecified
1:=> u6
#(1 2 3 4 71 6)
1:=> v6
#(1 2 3 4 5 6)
1:=> (vector-fill! v6 0)
#unspecified
1:=> u6
#(1 2 3 4 71 6)
1:=> v6
#(0 0 0 0 0 0)
```

8.1 Arrays

A two-dimensional array is a data type in which expressions are stored in places identified by two integer indexes. Bigloo and most other Scheme implementation don't support arrays, but this can be easily fixed. In fact, if the need arises for a two dimensional array, you can map it to a vector. Let us assume that you need the following array:

```
#( 0.0 0.1 0.2
   1.0 1.1 1.2
   2.0 2.1 2.2 )
```

The indexes of the element `0.1` are $(i = 0, j = 1)$; the indexes of `1.2` are $(i = 1, j = 2)$, and so on. You can map this array to the following vector:

```
#( 0.0 0.1 0.2   1.0 1.1 1.2   2.0 2.1 2.2 )
```

The pair of array indexes $(i = 1, j = 2)$ becomes the sole vector index $k = 5$. In general, the vector index k is given by the following expression:

$$k = i \times \text{line-length} + j$$

The function `make-array` that creates a vector to represent an array returns four values: the vector itself, a lambda expression to calculate a position in the vector, the greatest value for the line index, and the greatest value for the column index. The form

$$(\text{values } v_1 \quad v_2 \quad v_3 \ldots)$$

allows a function to produce more than one value. This form would be useless without means of receiving multiple values. Therefore, Bigloo offers a let-like tool that binds functional results to variables.

```
1:=> (multiple-value-bind (x) (list 4 5) (print x))
(4 5)
(4 5)
1:=> (define (polar x y)
        (values (sqrt (+ (* x x) (* y y))) (atan y x)))
polar
1:=> (multiple-value-bind (r theta)
        (polar 3.0 4.0)
        (vector r theta))
#(5.0 0.92729521800161)
```

The function **(polar x y)** returns two values corresponding to the polar coordinates of a point given by its Cartesian coordinates. Now that we know how to use (values v_1 v_2 v_3 v_4), let us define and test `make-array`.

```
;; File: myarray.scm

(define (make-array n m)
   (values (make-vector (* n m) 0.0)
           (lambda(i j) (+ (* m i) j))
           (- n 1) ;; max-line
           (- m 1) )  ) ;; max-column

(define (mat-test n m)
  (multiple-value-bind (mat idx maxLin maxCol)
      (make-array n m)

      (do ( (i 0 (+ i 1))) ((> i maxLin))
         (do ( (j 0 (+ j 1))) ((> j maxCol))
             (vector-set! mat (idx i j) (+ i (/ j 10)))))

      (do ( (i 0 (+ i 1))) ((> i maxLin))
                (newline)
                (do ( (j 0 (+ j 1))) ((> j maxCol))
                    (display (vector-ref mat (idx i j) ) )
                    (display "  ")))
      (newline)))
```

I will perform the tests from the interpreter as usual. If I load the file "myarray.scm" and type (mat-test 2 3), I will get the following output:

```
1:=> (load "myarray.scm")
make-array
mat-test
myarray.scm
1:=> (mat-test 2 3)

0  0.1  0.2
1  1.1  1.2
#<output_port:stdout>
```

8.2 Homogeneous arrays

What we have done this far works fine, but it is not efficient, since arrays are much in demand for number crunching, and Scheme vectors are designed for symbolic computation, and flexibility. In fact, you can store anything in a vector slot; this is a powerful feature, but slows down your code. Therefore, Will Farr devised a method of creating vectors of doubles, that speeds up the code generated by the Bigloo compiler, and shortens execution time.

```scheme
;; Compile: bigloo srffour.scm -o srffour

(module f64vector (main start-here)
   (type (tvector f64vector (double)))
   (include "loops.sch")
   (eval (export-all)))

(define (make-array n m)
   (values (make-f64vector (* n m) 0.0)
           (lambda(i j) (+ (* m i) j))
           (- n 1) ;; max-line
           (- m 1) )) ;; max-column

(define (mat-test n m)
   (multiple-value-bind (mat idx maxLin maxCol)
      (make-array n m)
      (for i from 0 to maxLin do
        (for j from 0 to maxCol do
            (f64vector-set! mat (idx i j) (+ i (/ j 10.0)))))
      (for i from 0 to maxLin do
        (for j from 0 to maxCol do
                (display (f64vector-ref mat (idx i j)))
                (display " "))  (newline))  ))

(define (start-here argv)
  (let ( (n (string->number (cadr argv)))
         (m (string->number (caddr argv))))
   (mat-test n m)))
```

8.3   for-loop

In the program above, I have used a Pascal-like `for-loop`, instead of a standard `do-loop`. Let us test the `for-loop`.

```
1:=> (load "loops.sch")
#unspecified
loops.sch
1:=> (for i from 0 to 5 do
        (print i))
0
1
2
3
4
5
#t
```

The problem with the `for-loop` is that, like arrays, it does not come with the standard distribution of Bigloo. Happily enough, it is as easy to implement as arrays.

```
;; File: loops.sch

(define-syntax for
   (syntax-rules (from to do by)
      [ (for var from low to high do body ...)
        (let ( (high-value high))
           (let loop ( (var low))
              (cond ( (> var high-value))
                    (else body ... (loop (+ var 1))))) )]
      [ (for var from low to high by del do body ...)
          (let ( (high-value high))
            (let loop ( (var low))
              (cond ( (> var high-value))
                    (else body ... (loop (+ var del)) ))))]
   )
)
```

Index