Scheme in 3 days

Wolfgang De Meuter and Theo D'Hondt
{ wdmeuter, tjdhondt }@vub.ac.be

Laboratorium voor Programmeerkunde
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel
Belgium

This 3-day crash course in intended for students who are already acquainted with one or more programming languages. The goal of the course is to give students a minimal hands-on experience in Scheme in order to reach a level at which they can write their own simple Scheme programs and at which they can confidently read sophisticated Scheme programs written by third parties.

The course has been conceived as a manual to hands-on sessions in a computer class. Hence, take a machine, open that Scheme environment and start following the instructions in this "manual".

1. Introduction

Scheme is a pretty established language that exists for almost 30 years now. As such, there are many good introductions to Scheme. An extremely well-written book is the following. It not only provides a decent overview of the language, but also a refreshing view on computation, computers, computer science and program construction.

> **Structure and Interpretation of Computer Programs**
> H. Abelson; G.J. Sussman; J. Sussman
> MIT Press
> ISBN: 0-262-01153-0
> http://mitpress.mit.edu/sicp/sicp.html
> This book is freely available online:
> (http://mitpress.mit.edu/sicp/full-text/book/book.html)

Why Scheme ?
There are two answers to this question: a simplistic answer and a full answer. The simplistic answer is that Scheme is a prerequisite for the course "OO Languages and their Implementation" you will get during the masters program. The full answer to the question why Scheme is taught at this level is that:

- Scheme is a pearl in the realm of programming languages. Together with languages like Smalltalk and Lisp, it is an example (and in contrast to other languages like Java and C++) of how good programming language design ought to be done.
- Scheme's concepts are really fundamental to computer science. This is not a surprise, as Scheme is a dialect of Lisp. Lisp was directly based on the famous λ-calculus, one of the mose important outcomes of fundamental research in computer science.
- Scheme is really old (Lisp was designed in 1958! It is – after fortran – the oldest programming language available.) but still extremely relevant and modern. Its concepts are to some extent incorporated in about every computer language available.

• Scheme is an extremely small language with very few concepts. But in order to be able to write good programs in Scheme, you really really really have to understand those concepts deeply.

Important Remark Before Taking Off:
Do not judge Scheme on the basis of the current programming environments. A programming language and a programming environment are different things!There are really horrible languages (Basic, Cobol, C++) with very professional environments and vice versa (many academic languages).

OK, let's go !
We'll be using **Dr. Scheme**, a freely downloadable implementation. Check out:

http://download.plt-scheme.org/drscheme/

Normally this package should be installed on the machines in the classroom.

2. Expressions and The REPL

Most programming languages you know are so called imperative programming languages. As the name of the paradigm suggest, these languages involve giving instructions to some (virtual machine) "to make the computer do something". In these languages, all constructs are somewhat arbitrarily divided into "statements" and "expressions". Expressions are the things that have a value, while statements are "pure" instructions that don't have a value.
Examples of expressions are `3+4`, `test ? ifTrue : ifFalse`, `new Name()`. Examples of statements are if-statements, for-statements and so on. So, in most languages, expressions yield a value, while statements do not yield a value.
In Scheme, this distinction is not made. Every program and every part of a program is considered to be an expression. Some expressions do not have a meaningful value. In that case, the value is `'undefined'`. Conversely, sometimes the value of an expression is not needed as the expression was simply employed to instruct the computer to do something (i.e. the expression was used as a statement in a more conventional language). In that case, the value of the Scheme expression is simply ignored (and garbage collected). So: **Expressions Have A Value!**
In Scheme, **a program** is an expression (which usually consists of several sub-expressions) Expressions always have a value. Sometimes, the value is undefined. This makes a language considerably simpler: e.g. Scheme only has 1 'if'-construct whereas C has 2 'if'-constructs (C has the `if-else` statement and it has the `?:` expression).

Evaluating Expressions: The REPL:
A Scheme programming system is "a world" in which you are constantly in the REPL, the Read-Eval-Print-Loop:

- Read: the phase that checks the syntax of your expression.
- Eval: the phase that 'runs' the expression.
- Print: the phase that shows the result of that run.

Here is an example of a user interaction with the REPL in DrScheme:

```
Welcome to DrScheme, version 103p1.
Language: Graphical Full Scheme (MrEd).
> 12
12
> "hello"
```

```
"hello"
```

The Environment:
A Scheme REPL always runs with respect to an environment. The environment can be thought of as a dictionary associating names to values. A name is looked up in the dictionary each time you refer to it. A name is added to the dictionary with the `define` construction of Scheme:

```
Welcome to DrScheme, version 103p1.
Language: Graphical Full Scheme (MrEd).
> (define x 10)
> x
10
```

The general form of that construction is `(define new-name initial-value)`.

Simple Arithmetic Expressions:
Scheme uses Prefix notation: an operator and its arguments are put between parenthesis, with the operator up front. Examples are shown in the following transcript. Study it carefully!

```
> 54
54
> (+ 23 55)
78
> (+ 23 44 99)
166
> (+ 23 (- 55 44 33) (* 2 (/ 8 4)))
5
> (/ 66 43)
66/43
> (/ 10 x)
1
```

Simple Boolean Expressions:
Here is a transcipt that describes the boolean system of Scheme. Study it carefully.

```
> (= 2 3)
#f
> (= 3 3)
#t
> (if (= 4 4) 5 6)
5
> (if (and (= 4 4) (> 6 5) (< 3 4)) "yes" (and #f #t))
"yes"
> (not (or (= 3 4) (= 5 6)))
#t
```

Conditional expressions:
There are two types of conditional expressions. Make sure you understand their general form (not that the "else"-branch is not compulsory, nor in the `if`-case, nor in the `cond`-case).

```
> (if (= 1 1) "waaw" "brrr")
"waaw"
```

```
> (cond ((= 1 1) "waaw 1")
        ((= 2 2) "waaw 2")
        ((= 3 3) "waaw once more")
        (else "waaw final"))
"waaw 1"
```

Exercises:
1) Predict the outcome of the following expressions assuming that they are evaluated in the listed order. Check whether you were right or not.

```
(+ 5 4 3)
(define a 3)
(define b (+ a 1))
(+ a b (* a b))
(= a b)
(if (> a b) a b)
(if (and (> b a) (< b (* a b))) b a)
(+ 2 (if (> a b) a b))
(* (cond ((> a b) a)
         ((< a b) b)
         (else –1))
   (+ a 1))
((if (< a b) + -) a b)
```

2) Write a Scheme expression that implements the following arithmetical expression:

(12/19 + (5+9) / 2) / ((10+11)*20 / 3)

3. Lambda Expressions & Special Forms

Function Applications
Since a Scheme environment is full of native functions, we can discuss how functions are applied even before we introduce how functions are created. The syntax for function application is as follows:

```
(f a1 a2 a3 ... an)
```

Hence, the "operator" expressions of the previous section are nothing but function applications. Indeed, in Scheme, there are hardly any restrictions on the names of functions. Hence, function names can be +, - and * e.g. So `(+ 3 4)` is an application of the `+` function.

Everytime the REPL encounters a function application that follows this syntax, it applies the following evaluation rules for function applications:

1) Evaluate `f`
2) Evaluate `ai` (order undetermined)
3) Check the number of arguments
4) Apply `f` to the evaluated arguments

Evaluation of "f" usually implies that it is searched for in the environment. But this doesn't always have to be so simple as is exemplified by the following expression:

```
((if (= 1 1) + *) 4 5)
```

of which the outcome is 9. Make sure you understand what is happening here !

Lambda Expresssions
Now that we know how to apply functions, let us make some of our own. This is done with the `lambda` construction (for those of you who know λ-calculus... it is the same thing as λ).

```
(lambda (param1 param2 param3 ... param-n) exp)
```

This should be read as "make me a function with `param1`, `param2`, ... `param-n` as parameters and `exp` as body". Whenever the REPL encounters such an expression, it will internally construct the function and return it as a result. Here is an example expression that forces the REPL to "make" the successor-function:

```
(lambda (x) (+ x 1))
```

This should be read as "make me a function of one parameter that adds one to that parameter". Of course, here's how to apply it to 5 ("count" the parentheses!!!!):

```
( (lambda (x) (+ x 1)) 5)
```

Of course, you can use `define` to add functions to the environment:

```
(define plus1 (lambda (x) (+ x 1)))
(plus1 5)
```

Exercises
1) Although `lambda`, `define`, `if` and `cond` follow the regular `(f a1 a2 ... an)` syntax, they are not functions. Prove this using what you know so far!

2) Write two procedures `celcius-to-fahrenheit` and `fahrenheit-to-celcius`. The conversion formula is F = (C+ 40)*1,8-40

3) Write recursive procedures `fac`, `fib` and `gcd`. Their algorithms are as follows:

a) fac(0) = 1
   fac(n) = n*fac(n-1) if n≠0
b) fib(0) = 1
   fib(1) = 1
   fib(n) = fib(n-1)+fib(n-2) if n>1

4 a) Write a recursive procedure `display-n` that accepts two parameters, a character and a number. It will (using the regular display) display the given character as many times as indicated by the number provided.

4 b) Write a recursive procedure `squares` that accepts a number and that displays four squares in such a way that everything together forms a square. Use local procedures if necessary! Example:

```
(squares 3) =>        (squares 5) =>
      ******                **********
      * ** *                *    **    *
      ******                *    **    *
      * ** *                *    **    *
      ******                **********
```

```
*   **   *
*   **   *
*   **   *
**********
```

Special Forms
As we saw in the previous sections, some expressions of the form (f a1 a2 ... an) are not "real" function applications since their arguments are not evaluated immediately. This is the case whenever "the f" refers to a special name such as `lambda`, `if`, and so on. At that time, the REPL recognizes this and will apply a dedicated application procedure to them. Such "f"'s are called special forms and Scheme has quite a lot of them. Do the following experiments in the evaluator:

```
> (and #f (/ 1 0))
#f
> (and #t (/ 1 0))
/: division by zero
> (if #t 2 (/1 0))
2
> (if #f 2 (/ 1 0))
/: division by zero
> (and #t #t #f (/ 1 0))
#f
> (and #t #t #t (/ 1 0))
/: division by zero
```

4. Higher Order Functions

In Scheme, functions are "things" of the same "level" as numbers, booleans, strings and characters. This means that functions can be used as arguments of functions and functions are allowed to return functions as a result. Functions that manipulate other functions are known as higher order functions.

Make sure you understand the following interaction with the REPL:

```
> ((lambda (x y z) (+ x y z)) 1 2 3)
6
> (((lambda (x) (lambda (y) (+ x y))) 5) 6)
11
> (define f (lambda (x) (+ x 10)))
> (f 5)
15
> (f (f 5))
25
> ((f 5) (f 5))
procedure application: expected procedure, given: 15; arguments were: 15
> (f f)
+: expects type <number> as 1st argument, given: #<procedure:f>; other
arguments were: 10
```

Now consider the following. Which mathematical concept is being implemented by the `mysterious-`function ?

```
Welcome to DrScheme, version 103p1.
Language: Graphical Full Scheme (MrEd).
```

```
> (define f (lambda (x) (* x 10)))
> (define g (lambda (y) (+ y 5)))
> (define mysterious (lambda (f g) (lambda (x) (f (g x)))))
> (define fg (mysterious f g))
> (f 10)
100
> (g 10)
15
> (fg 10)
150
> (define gf (mysterious g f))
> (gf 10)
105
```

Now do you understand when we say that “functions are treated on the same level as any other value”? We say that, in Scheme, functions are first class-citizens, whilst in many other programming languages, functions are second class or even third class citizens.

Finally, there is one more thing that needs to be said about functions: since the combination of `lambda` and `define` occurs so frequently and since the combination engenders a lot of parentheses, Scheme introduces so-called “syntactic sugar” for this combination. I.e. the expression:

```
(define (name arg1 arg2 arg3 ... arg-n)
    body)
```

is exactly the same as

```
(define name (lambda (arg1 arg2 arg3 ... arg-n)
    body))
```

Hence,

```
(define (fac n)
    (if (= n 0) 1 (* n (fac (- n 1)))))
```

is the same as

```
(define fac (lambda (n)
    (if (= n 0) 1 (* n (fac (- n 1))))))
```

Exercises

1 a) Write a procedure `(sum term a next b)` that accepts two numbers `a` and `b` and two functions `term` and `next`. The idea is to add together all `(term i)` where the i's lie between `a` and `b`. The ‘next’ i is obtained from applying `next` to the previous i.
1 b) Implement a procedure `(product factor a next b)` in the same way.
1 c) Write `fac` using `product`.

2 a) Write a procedure `(accumulate combiner null-value term a next b)` that “abstracts away” from `sum` and `product` as in the previous exercise.
2 b) Implement `sum` and `product` using `accumulate`.

5. Local Variables

Sometimes you need an expression twice inside a big expression. In that case, it is safer, faster and more readable to "abstract that expression out" into a variable. But in order not to polute the global environment, we can abstract it out into a variable that is only visible to the big expression. This is accomplished with the `let`-special form. Its general appearance is as follows:

```
(let((name1 expression1)
     (name2 expression2)
     (name3 expression3)
     ...
     (name-n expression-n))
  body-using-name1-name2-...-namen)
```

(Notice that Scheme does not define the order in which the variables are bound! Hence, evaluation happens in random order. If you want to enforce a top-town order, use `let*`) Here is an example.

```
(define (roots a b c)
    (let ((delta (- (* b b) (* 4 a c))))
        (if (> 0 delta)
            (display "No real root")
            (if (= 0 delta)
                (display (list "1 root:" (/ (- b) (* 2 a))))
                (display     (list
                          "2 roots:"
                          (/ (+ (- b) (sqrt delta))
                            (* 2 a))
                          "and:"
                          (/ (- (- b) (sqrt delta))
                            (* 2 a)))))))))
```

Exercise
1) The `let`-special form is not essential to Scheme. It is syntactic sugar for a more complicated expression. Find that expression using what you know so far.

2) Why does the following program generate an error (and which error do you think it is)? Fix the program in two different ways.

```
(let     ((x 1)
      (y (* 2 x)))
   (+ x y))
```

6. Dotted Pairs & Lists

As you already know, Scheme is a dialect of Lisp. Lisp is an acronym for "List Processor" as it introduced lists as the most important data structure. But in fact, lists are not really essential or native to Scheme. Instead, they are constructed using so-called dotted pairs, and these are essential. Dotted pairs are manipulated using three primitive procedures called `cons`, `car` and `cdr`.

• `cons` takes two arguments and returns a "dotted pair" whose first component corresponds to the first argument of `cons` and whose second component corresponds to the second argument of cons.
• `car` is applied to a dotted pair. It returns the first component of the dotted pair.
• `cdr` selects the second component of a dotted pair.

Furthermore () is the empty dotted pair that has no `car` and no `cdr`.

The name "dotted pair" comes from the fact that `(cons 1 2)` is printed as `(1 . 2)`. Dotted pairs are also known as cons-cells or simply cells. `cons`, `car` and `cdr` are governed by the following axioms:

```
(car (cons x y)) => x
(cdr (cons x y)) => y
```

A box-and-pointer diagram is a drawing where each dotted pair is represented by a box with two parts that point to their content. All other things (like #t, 3 and ()) are "simply" drawn without boxes and/or pointers.

Exercises
1) Evaluate the following expressions in the REPL. Draw box-and-point-diagrams of the resulting structures.

a) `()`
b) `(cons 1 2)`
c) `(car (cons (cons 1 2) (cons 3 4)))`
d) `(cons (cons (cons (cons 1 2) 3) 4) 5)`
e) `(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 ()))))))`
f) `(list 1 2 3 4 5)`
g) `(car (list 1 2 3 4 5))`
h) `(cdr (list 1 2 3 4 5))`
i) `(cadr (list 1 2 3 4 5))`
j) `(caddr (list 1 2 3 4  5))`

2) Let's do some Lis.P!
   a) Implement `member?` that checks whether a given value is a member of a list.
   b) Implement `length`, a procedure that calculates the length of a list.
   c) Implement `append` to concatenate 2 given lists.
   d) Implement `reverse`.
3) Write a procedure that maps a function over every element of a provided list.

7. Destructive Operations and Imperative Programming

When beginning these exercises, we assumed you already know about imperative programming. So we will not practice this in these sessions. We just want to mention here that this is also possible in Scheme. In imperative programming, it is custom to write programs as lists of instructions that manipulate memory locations (i.e. variables as introduced by `define`). In Scheme, these lists of instructions are grouped in using the `begin` special form. The value of the entire `begin` construction is the value of the last expression. Hence, the value of

```
(begin (display "haha")
       (newline)
       (display 4)
       6)
```

is 6. Notice that this kind of programming adds two dimensions of complexity to your programs:

1) Because the instructions are executed one after the other, you have to start thinking about the order of expressions, hence, you have to take time into account.

2) Because the imperative instructions typically manipulate memory, you have to think about memory locations that change over time.

Therefore imperative programs are harder to think about than the functional programs we have written so far: you don't have to reason about expressions that yield a value, but you have to reason about a memory that changes over time. Difficult and error-prone!

As we mentioned, imperative programs manipulate variables. In Scheme variables are manipulated through 3 special forms:

`(set! var value)`
updates a variable so that it contains a new value.
This corresponds to := in Pascal or to = in C, C++ and Java.

`(set-car! cell value)`
updates the car-part of a cell. Notice that the same cell is reused.

`(set-cdr! cell value)`
updates the cdr-part of a cell.

Exercises

1) Construct a ring with elements (1,2,3,4).

2) Suppose I am writing a sorting routine. Why do you think the following local procedure will NOT work?

```
(define (swap a b)
    (let ((temp a))
      (begin (set! a b)
             (set! b temp))))
```

3) Consider

```
(define a-list (list 1 2 3 4 5))
```

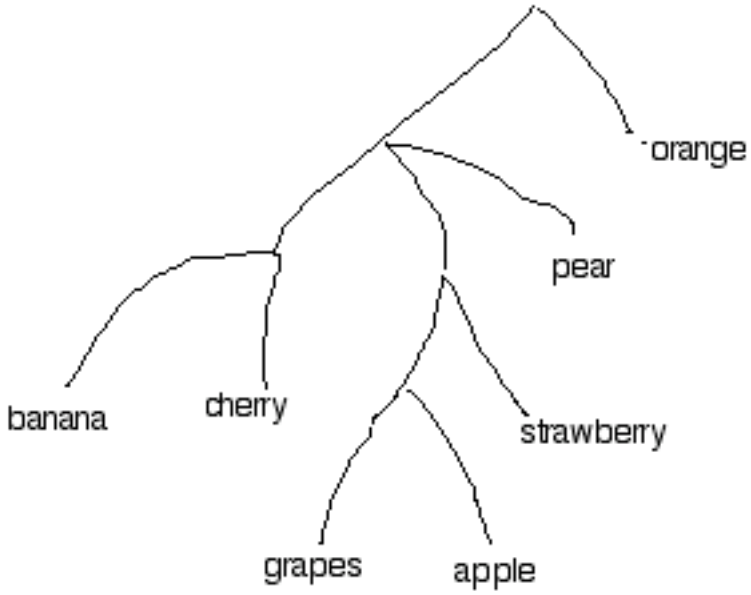
Now write a 1-line expression that changes the "3" into "haha".

8. Quoting

A Scheme primitive `quote` takes one argument. It is said to quote the argument, i.e. "just take the argument and return it unevaluated". When the argument is a list of expressions, a list is created with the quoted expressions as elements. Furthermore, `(quote exp)` can be abbreviated to `'exp`. When a name (i.e. an identifier) is quoted, we talk about symbols. It is important to understand that symbols are really a new kind of data value in Scheme. Symbols are "treated on the same level" as strings, numbers, functions, charaters and so on. Note that symbols are fundamentally different from strings. Whereas strings are rows of characters, symbols are indivisible things: they are internally represented by a unique pointer.

Exercises

1) Construct the following tree in two different ways using quoting. One way that uses "quote" once. Another way that uses "quote" many times.



2) Have a look at the following procedures that calculate the derivative of an expression .

```
(define (is-operator? op exp)
  (and (pair? exp)
       (eq? (car exp) op)))

(define (is-symbol? exp)
  (eq? exp 'x))

(define (is-constant? exp)
  (number? exp))

(define (derive exp)
  (cond ((is-constant? exp) 0)
        ((is-symbol? exp) 1)
        ((is-operator? '+ exp) (list '+
                                     (derive (cadr exp))
                                     (derive (caddr exp))))
        ((is-operator? '- exp) (list '-
                                     (derive (cadr exp))
                                     (derive (caddr exp))))
        ((is-operator? '* exp) (list '+
                                   (list '*
                                         (derive (cadr exp))
                                         (caddr exp))
                                   (list '*
                                         (cadr exp)
                                         (derive (caddr exp)))))))
```

a) Adjust the procedure “derive” such that you can also use it to calculate derivatives of fractions.

b) Adjust the procedure such that it can also be used to calculate $\partial \frac{f}{x}$ where x is an arbitrary variable.

9. Vectors and Objects

Vectors
Vectors in Scheme are known as tables or arrays in other languages. There are four fundamental special forms to manipulate vectors:

```
(make-vector size)
(make-vector size initial)
(vector-ref vector index)
(vector-set! vector index value)
```

Closures
Objects are not really a language feature in Scheme, but the result of a programming technique that uses closures. In order to understand closures, you first have to understand the scoping rules of Scheme. Consider the following program

```
(define (class x)
    (lambda (message)
        (cond
            ((eq? message 1) (display x))
            ((eq? message 2) (set! x (+ x 1))))))
```

then each time, `class` is called, we get a function `(lambda (message..))` that can still refer to the x variable, even though the class function "doesn't live anymore". This function we get back can still refer to the variables of the function that created it. It is called a closure. Closures are used in Scheme to simulate objects. Here is an example of how they are used:

```
(define o1 (class 0))
(define o2 (class 10))
(o1 1) -> displays 0
(o1 2)
(o1 1) -> displays 1
(o2 1) -> displays 10
```

Now just image what happens when we turn the message into a symbol rather than a number:

```
(define (class x)
    (lambda (message)
        (cond
            ((eq? message 'disp) (display x))
            ((eq? message 'tick) (set! x (+ x 1))))))
```

We use it as follows:

```
(define o1 (class 0))
(define o2 (class 10))
(o1 'disp) -> displays 0
(o1 'tick)
(o1 'disp) -> displays 1
```

```
(o2 'disp) -> displays 10
```

Variable Length Arguments
A final feature that we want to discuss in this section are lambda's with a variable number of arguments. In the following expression

```
(lambda (x y z) (+ x y z))
```

the parameters are actually "conceived as a list". I.e., when calling the function, only calls that provide exactly three arguments will match the list (x y z). In the following example.

```
(lambda somelist
    (+ (car somelist) (cadr somelist) (caddr somelist)))
```

calling the procedure will always succeed, no matter how many arguments are provided. Of course, the body of the procedure will only work with lists that contain at least three elements. But when calling the procedure with 10 arguments, it will still work and the last 7 arguments will simply be discarded. Hence, a lambda of the form:

```
(lambda par body)
```

can be applied to any number of arguments. Those arguments will be collected into a list. The list will be bound to `par`.

So, until know we have procedures with a fixed number of parameters and procedures with a variable number of parameters. A mixed form is also possible:

```
(lambda (x y z . rest) ...)
```

This procedure can be called with any number of arguments as long as at least three are provided. When this happens, the first argument is bound to x, the second is bound to y, the third is bound to z and all the others are collected into a list (see the dot!) which is bound to "rest". The list can be decomposed further with car, cdr and the like.

Exercises
1) Let us implement an ADT "Stack" using vectors. You will have to implement a constructor to create an empty stack and manipulation functions push ,pop and top.

a) Implement a function `make-stack` using a vector. Remember that you will need an index as the top of the stack

b) Implement `push`, `pop` and `top`.

2) Make a '`class`' stack that implements a stack. Make two versions: the first one uses an array implementation and the second one uses a linked list implementation.

10. Macros, Quasi-Quoting, Unquote (Splice)

You can extend Scheme's special form set using macros. Macros are defined like procedures. However, when calling a macro, the arguments will not be evaluated. Try out the following:

```
(define-macro (show-all . all)
  (display all)
  (newline)
```

```
  "That's all Folks!")
```

and then

```
(show-all (sin 3 +) x (define do if))
```

Let's explain what happens. Macros don't evaluate anything, the merely expand their body at the call-site of the macro. The arguments are passed "as is", i.e. without evaluating.

The above example merely returns the given argument expression(s). Sometimes you want to return a new "constructed" expression in which the arguments of the macro have been incorporated. Then you'll have to use quasiquoting. Here is an example:

```
(define-macro (while bool first . body)
  `(letrec ((*loop* (lambda (res)
                      (if res
                        (begin ,first
                               ,@body
                               (*loop* ,bool))))))
     (*loop* ,bool)))
```

Exercises

1) Create a macro "for" that takes a variable name (check it!), an initial value expression, a stop condition, an "update expression" and a sequence of "body expressions". It acts like a for in Java, C++ or Pascal. The variable is defined and bound to the initial value. Then the body expressions are evaluated, the update condition executed and the stop condition evaluated. If it yields #f the iteration stops. Otherwise it loops again.

2) A drawback of anonymous lambdas is that they cannot be recursive. Create philosophers with macros. Philosophers are nameless lambdas that can call themselves using the name "self". Example:

```
    (philosopher (n) (if (= n 0) 1 (* n (self (- n 1)))))
```

Philosophers follow the same syntax rules as lambdas: first there is a list of parameters and then follows a list of expressions