Functional Programming in Scheme

Functional programming languages - or applicative languages - follow closely the Lambda calculus model. Their strengths and weaknesses can be found in the theoretical model seen through the lens of practical programming. Here we are using Scheme as a discussion basis.

Side effects

Ideally, in a functional program there are no side effects. There are no assignments and a variable is just a name of a value that is non-modifiable once the variable is bound to it. The benefit of banning completely the side effects can be judged from the C example below.

```
int F(int* X) {return (*X=0);}
X = F(&X) + ++X;
```

The value assigned to **X** depends on the evaluation order of the expression **F(&X)+ ++X** for any initial value **a** of **X** different than **0**. The result of the expression **F(&X) + ++X** is **1**, for the left-to-right evaluation, and **a+1** if the evaluation is right-to-left.

Banning side effects have important consequences on the way problem solving is seen and on the programming style itself. There are no assignments, and programs have no state. Basically, a pure functional program can be seen as a functional composition that encodes the flow of transformations ***output = program(input)***.

However, when banning the side effects, we have to distinguish between the pure functional programming and the practical approach, where the pure functional style is sacrificed for the sake of imperative programming commodities such as assignment. In some of the existing functional programming languages, the pure functional programming style can be met rather by a self-accepted programming discipline. It is somewhat similar to structured programming in a language that encourages it but is equipped with the infamous **goto**.

Here we adhere mostly to the pure functional programming paradigm and pretend the language we use does not allow the use of side effects (although it does). More specifically we ignore assignments, even if they are possible in the language.

The freedom from side effects allows for *referential transparency*. An expression must stand for the same value regardless the way it is written as far as it is correct. A counter example is taken again from C:

```
int f(int x) {return x * x--;}
int g(int x) {return x-- * x;}
```

The applications **f(1)** and **g(1)** return always different results. For instance, if the evaluation of multiplication is left-to-right the result of **f(1)** is **1**, whereas the result of **g(1)** is **0**.

Functions as first class values

The essential characteristic of the functional programming paradigm is that functions are treated as first class values. Functions can be applied on functions and, moreover, can return functions. It is important to watch more closely what it is meant by *first class* values.

Consider the exercise of writing a function, called **compose**, which takes as parameters two **1**-ary functions (an **n**-ary function has **n** parameters) **f: B → C** and **g: A → B**, and returns a function which, computationally, behaves as **f o g**, i.e. **(f o g)(x) = f(g(x))**.

Can we have a definition of **`compose`** in C? Let us try the definition below, where **`A`**, **`B`** and **`C`** are identifiers of assumed legal C types:

```
C compose(C (*f)(B), B (*g)(A), A x) {return (*f)((*g)(x));}
```

Indeed, **`compose(f,g,x)`** works fine, except for the following apparently insignificant "details".

- The types **`A, B`** and **`C`** are fixed. Therefore, our functional composition is not general.
- The application **`compose(f,g,x)`** is the result returned by the **`(f o g)(x)`** and not the function that corresponds to the composition **`(f o g)`**, as required.

What we want is to write something like **`f_o_g = compose(f,g)`**, so that, later on, to be able to write **`(f_o_g)(x)`**. Therefore, we expect that the function **`compose`** returns a newly fabricated function. This is impossible in C although it is ordinary computation in a functional programming language. For instance, in Scheme we could write, as we wanted:

```
(define compose (lambda (f g) (lambda (x) (f (g x)))))

(define double_succ
    (compose (lambda (x) (* x 2))  (lambda (x) (+ x 1))))
(double_succ 3)
8
```

Here, the function **`compose`** returns an **`1`**-ary function, which is the composition of its functional parameters. Since functions are first class values, it follows that there are types corresponding to functions and there are values of these types. For example, the expression **`(lambda (x) (+ x 1))`** is the Scheme representation of the value that corresponds to the successor function. The conventional representation of a particular value of a type is a constant of that type. The expression **`(lambda (x) (+ x 1))`** can be seen as a constant belonging to the type **`number`** $\rightarrow$ **`number`**. In general, the Scheme expression

(*lambda* (*p*$_1$...*p*$_n$) *expr*$_1$...*expr*$_m$)

stands for an **`n`**-ary uncurried function[1] the body of which is ***expr*$_1$...*expr*$_m$**. As far as the function application is concerned, the expression **(*expr*$_f$ *ap*$_1$...*ap*$_n$)** designates a function application (the prefix format is used throughout the language constructs, and so there is no problem of precedence or associativity). The functions use the applicative-order evaluation when applied. In addition, the body of a function is not evaluated prior to application, i.e. the functional normal form is observed.

1. All the expressions of the application **(*expr*$_f$ *ap*$_1$...*ap*$_n$)**are evaluated in an unspecified order. If side effects are banned the order is irrelevant. Note ***f*** the result of ***expr*$_f$** and ***ef*$_i$** the value resulted from evaluating ***ap*$_i$**; ***f*** must be a function.

2. The function ***f*** is applied on ***ef*$_1$...*ef*$_n$**.

For example, **`((compose (lambda (x) (* x 2)) (lambda (x) (+ x 1))) 3)`** applies the result of **`(compose (lambda (x) (* x 2)) (lambda (x) (+ x 1))`** on the value designated by the decimal constant **`3`**.

[1] We prefer the term *function* instead of *procedure*, as used in Scheme. The reason is that most of the Scheme procedures we will write do not have side effects and will always return a meaningful result.

No explicit type declarations are required when defining a Scheme function. Scheme is *a dynamically typed* language or, equivalently, *latent typed*. In a latent typed language, types are associated to values rather than to the program identifiers and the type checking is performed at run time. Also Scheme is sometimes considered as a *weak-typed* language since values of any type can be mixed and assembled without restrictions to form data structures. The only must is to make sure that the actual parameters when applying a function have the types required by the function. This responsibility is delegated to the programmer, in opposition to the static-typed languages where the compiler does the full bulk of type checking.

For some authors the terms *statically typed* and *dynamically typed* have different meaning from *weak-typed* and *strong-typed*. A language is considered strong-typed if the type checking is performed in the language, at compile time, run time or both. A language is considered statically typed if the correctness of the program operations (type checking) is performed at compile time. The language is dynamically typed if the type checking is done at run time, when effectively performing the computations.

We know from Lambda calculus that anonymous functions can be used as computational devices, eventually recursive. From this point of view, the definition

```
(define succ (lambda (x) (+ x 1)))
```

is not merely the definition of the function **succ**. It stands for defining a variable, called **succ**, the value of which is a function. The function results from evaluating the expression **(lambda (x) (+ x 1))**. If the value of the variable **succ** cannot be modified (it does in Scheme), the variable can be seen as the name of a function. Therefore, notice that the expression **(define *variable value*)** is not an assignment. It creates a ***variable*** bound to a ***value***.

For the sake of a smooth start in Scheme we will avoid using fixed-point combinators, at least for a while. A recursive function will be defined in the classical way, by using the "name" of the function as a free variable within the body of the function itself as in **factorial** definition below.

```
(define factorial
    (lambda (n) (if (= n 1) 1 (* n (factorial (- n 1))))))
```

Observe that the decisional expression **(if *predicate* $expr_1$ $expr_2$)** returns a value: the value of $expr_1$ - if the ***predicate*** evaluates to true (represented by the constant **#t**) - or the value of $expr_2$ - if the ***predicate*** evaluates to false (represented by the constant **#f**). Scheme is an *expressional* language. Most Scheme constructs are expressions that return values. The expressions themselves are function applications in the prefix *format* **(*function* $param_1$ ... $param_n$)** .

Since functions are first class values, a function can be written in such a way so it can be applied partially on to its parameters, i.e. in the *curry* format. Let's define the function **sum** almost as in the Lambda calculus, where we are writing **sum** $\equiv_{def}$ **λx. λy.((+ x) y)**.

```
(define sum  (lambda (x) (lambda (y) (+ x y))))
```

It can be applied on both parameters or only on a single (first) parameter. We can write:

```
((sum 3) 2)
5
(define succ (sum 1))
(succ 3)
4
```

If a function is not curried then, if necessary, it can be transformed into an equivalent curried function by using the **curry** function below (defined for binary functions). The reciprocal is also possible. As an exercise, define a function called **uncurry** that transforms a binary function from the curried format to the uncurried format.

```
(define curry (lambda (f) (lambda (x) (lambda (y) (f x y)))))

(define plus (lambda (x y) (+ x y))) ;uncurried binary function
(define succ ( (curry plus) 1))
                     ↑
                  curried binary function
```

Scoping and Closures

An attentive reader could notice that the mechanisms sketched above would not work correctly without an additional feature needed to compensate for the textual substitution of the Lambda calculus. Reconsider the **sum** example. When defining the function **succ** the result must be, according to the **β**-reduction rule from Lambda calculus:

```
((lambda(x) (lambda(y) (+ x y))) 1) →β
            (lambda(y) (+ x y))[1/x] = (lambda (y) (+ 1 y)).
```

Nevertheless, textual substitution is performed neither in Scheme, nor in other functional languages. In this case how does **succ** "know" that in its body, i.e. in the function **(lambda (y) (+ x y))**, the free variable **x** is bound to **1**?

When a function is generated as a value it "closes" its computational environment (which depends on whether the language is a statically scoped language, such as Scheme[2], or a dynamically scoped language, such as early versions of Lisp). The word “closes” means in fact that the function saves its computational environment together with a pointer to its own code.

A language is *statically scoped* if the program objects accessible (via identifiers) from a point of the program are controlled statically by ways of structuring the text of the program. This is what the program blocks do in most conventional programming languages. Therefore, the variables accessible from a specific program point can be determined from the textual structure of the program.

A language is *dynamically scoped* if the program objects accessible (via identifiers) from a point of the program are controlled dynamically, based on the moment of time when they are created. Usually, if several objects share the same name then the newest object is accessible when using its name.

In a lexically scoped (or statically scoped) language the *computational environment* of a point **P** of a program at a given moment **t** in time is the set of all variables that contain the point **P** in their lexical scope of at the moment **t**.

In a dynamically scoped language the computational environment of point **P** of a program at a given time **t** is the set of all the most recent variables created before **t** and whose identifiers are referenced from **P**. Therefore, if there are several variables with the same identifier then the freshest is used.

[2] Top level variables are dynamically scoped in Scheme. Only variables that appear inside expressions, such as **lambda, let, do**, are statically scoped.

To understand the essential differences between static scoping and dynamic scoping consider the following C program.

```
int x = 1;

int f(void) {return x;}
int g(void) { int x = 2; return f();}

g();
```

Since C is statically scoped, the result of **`g()`** is certainly **`1`**. The computational environment of the function **`f`** contains the variable **`x=1`**. Assume that C is dynamically scoped. In this case the result of **`g()`** is **`2`**. The reason is that the computational environment of the function **`f`** contains the *newest* created variable **`x=2`**, since **`f`** uses the identifier **`x`**.

Scheme is basically lexically scoped, except for the top-level variables that have dynamic scope. For example, the function **`f`** below returns different results according to the time it is called.

```
(define f (lambda() (g 1)))
(define g (lambda(x) (+ x 1)))
(f)
2

(define g (lambda(x) (- x 1)))
(f)
0
```

It is pretty clear that referential transparency is ruined due to the dynamic scoping even if there are no side effects! As a small compensation for the potential dangers of dynamic scoping observe that the top-level functions of a Scheme program can be defined in any order.

In Scheme, the scope of the formal parameters of a function is the body of the function, as in Lambda calculus. In addition, there are special expressions for controlling the scope of identifiers (practically, the scope of variables).

(let ((var_1 val_1) (var_2 val_2) … (var_n val_n)) $expr_1$ $expr_2$ … $expr_m$)

The scope of var_i **i=1,n** is the $expr_1$ $expr_2$ … $expr_m$. The expressions val_i are evaluated in an unspecified order. The variable var_i is bound to the result of evaluating val_i and then $expr_1$ $expr_2$ … $expr_m$ are evaluated in sequence. The result of $expr_m$ is the result of the **`let`** expression.

(let* ((var_1 val_1) (var_2 val_2) … (var_n val_n)) $expr_1$ $expr_2$ … $expr_m$)

The scope of var_i is the part of **`let`** that follows the pair **(**var_i val_i**)** . The expressions val_i are evaluated sequentially. The variable var_i is bound to the result of evaluating val_i and then $expr_1$ $expr_2$ … $expr_m$ are evaluated in sequence. The result of $expr_m$ is result of the **`let*`** expression.

(letrec ((var_1 val_1) (var_2 val_2) … (var_n val_n)) $expr_1$ $expr_2$ … $expr_m$)

The scope of var_i is the entire **`letrec`**. The expressions val_i are evaluated in an unspecified order. The variable var_i is bound to the result of evaluating val_i and *then* $expr_1$ … $expr_m$ are evaluated in sequence. The result of $expr_m$ is result of the **`letrec`** expression.

A function in Scheme, as in many other functional programming languages as well, is a *functional closure*. Take the example:

```
(define sum (lambda (x) (lambda (y) (+ x y))))

(define succ (sum 1))
```

The function **`(lambda (y) (+ x y))`** closes its computational environment at the moment it is produced. In this environment **`x`** is bound to **`1`**. Therefore, the function above is equivalent to **`(lambda(y) (+ 1 y))`**. Clearly, the closure is a practical substitute for the textual substitution present in the theoretical model.

Similarly, the **`compose`** function works properly since it is a functional closure. Indeed, the result produced by the application

```
(compose (lambda (x) (* x 2)) (lambda (x) (+ x 1)))
```

is the closure corresponding to **`(lambda (x) (f (g x)))`** where **`f`** is bound to the function resulting from **`(lambda(x)(* x 2))`**, and **`g`** is bound to the function resulting from the evaluation of **`(lambda(x)(+ x 1))`**.

Apart from its scope, a variable has another important property: its lifetime called the variable *extent*. The variables of a closure have unlimited extent. They live as long as the closure is not garbage collected. In conventional programming languages, as in C for example, the extent of a dynamic variable is limited. The variable exists as long as the function it belongs to is executed. The variable is destroyed (popped off the stack) on exit from the function.

By convention, since the closure mechanism is similar to textual substitution, we represent a closure by textually substituting the variables closed by their values. Nevertheless, observe that this convention is not entirely correct in Scheme, where top level variables are dynamically scoped.

Data types in Scheme

Scheme is a practical language and, therefore, the user is not forced to build data on a pure functional basis as in the experimental λ_0 language. Predefined data types exist, including: number (a wealth of numeric types, including exact and inexact numbers), symbol, boolean, char, string, pair, list (as a variant of pair), procedure, continuation etc. The type of a program value is determined at run time. The only explicit use of types is through the constants and through specific operators that, eventually, can be used to find the type of a value.

Built-in predicates such as **`boolean?, symbol?, pair?, list?, number?, procedure?`** can be used to test the type of a value. For example, the following function is implementing the overloaded binary function **`plus`** which performs the addition for numbers and the concatenation for strings.

```
(define plus
    (lambda (A B)
        (cond ((and (list? A) (list? B)) (append A B))
              ((and number? A) (number? B)) (+ A B))
              (else (throw “*** Illegal types in plus”))))
```

Here, **`throw`** suggests a sort of exception handling mechanism. Exceptions are not part of Scheme but they can be implemented using the more powerful facility of computation *continuation*, discussed later.

Lists and symbols are widely used in Scheme programs. The empty list is represented by the constant **()**, whereas a list with the elements e_1 e_2 … e_n is written (e_1 e_2 … e_n). The predefined pair and list functions are those from λ_0: **car, cdr, cons, null?, length, reverse** etc. As a simple exercise, the following is a program to compute the list of all prime numbers less than or equal to a given limit.

```
(define mod (lambda (x y) (- x (* y (quotient x y)))))

(define succ (lambda (x) (+ 1 x)))

(define sift
  (lambda (x candidates)
     (cond ((null? candidates) '())
           ((= 0 (mod (car candidates) x)) (sift x (cdr candidates)))
           (else (cons (car candidates) (sift x (cdr candidates)))))))

(define primes
  (lambda (Limit)
     (letrec ((primes
                 (lambda (candidates)
                   (if (null? candidates) '()
                       (cons (car candidates)
                             (primes (sift (car candidates)
                                           (cdr candidates))))))))
         (primes (make_sequence 2 Limit)))))

(define make_sequence
      (lambda (n Limit)
         (if (= n Limit) (list n)
             (cons n (make_sequence (succ n) Limit)))))

(primes 50)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47)
```

Note the conditional **cond**, a more general alternative of the **if** selection expression.

```
         (cond (predicate_1 expr_11 ... expr_1n1)
               (predicare_2 expr_21 ... expr_2n2)
                ...
               (predicate_m expre_m1 ... expr_m nm)
               (else expr_1 ... expr_q))
```

It is equivalent to

```
           (if predicate_1
               ;then
                    (begin expr_11 ... expr_1n1)
               ;else
                    (if predicate_2
                        ;then
                              (begin expr_12 ... expr_1n2)
                        ;else
                              ...
                                  (if predicate_m
                                      ;then
                                           (begin expr_m1 ... expr_m nm)
                                      ;else
                                           (begin expr_1 ... expr_q)...)
```

Higher-order functions and normal-order evaluation

Functions that take functions as parameters and/or return functions as results are called *higher-order* functions or, simply, *functionals*. Functionals help writing compact programs. What is normally specified in several lines of code in a conventional (imperative) programming language may take a single line in a functional language. For instance,

```
(lambda (L) (foldleft + 0 L))
```

is a Scheme function for computing the sum of a list of numbers, where **foldleft**[3] is a functional that comes with most Scheme libraries.

```
(define foldleft
   (lambda (fun init list)
      (if (null? list) init
          (foldleft fun                  ; fun
                   (fun init (car list))  ; init
                   (cdr list) ))))        ; list
```

The theoretical λ_0 language works with normal-order evaluation. Scheme and some other functional languages use the applicative-order evaluation, i.e. functions evaluate their parameters before application. Exceptions are some predefined functions such as **and**, **or**, **cond** and **if**, which are using normal-order evaluation. For instance, **or** does not evaluate its second argument if the first argument is **#t** (true), and **and** does not evaluate its second argument if the first is **#f** (false). How could we write non-strict functions when necessary?

The solution is remarkably simple. Since functions are first-class values then we can pack the effective parameters of a non-strict function as anonymous **0**-nary functions (in effect, functional closures with no arguments). These functional parameters can be called when needed during the evaluation of the non-strict function. Consider defining the Boolean **nand** function **nand (x,y) = ¬ (x ∧ y)**. Observe that

```
nand(false,y) = true
nand(true,y) = ¬ y
```

Therefore, we have to evaluate **y** only when **x** is **true**. Defined as a non-strict function **nand** works with functions as arguments:

```
(define nand (lambda (fx fy) (if (fx) (not (fy)) #t)))

(nand (lambda() #f) (lambda() #t))
#t
(nand (lambda() #t)
      (lambda() (nand (lambda() #f) (lambda() #t))))
#f
```

The inconvenience of explicitly packing values as functions can be overcome by adding to the language some macro-definition sugar.

The normal-order evaluation is also compensated in Scheme by the possibility to explicitly control the evaluation time of an expression. There are two behaviorally opposite functions: **quote** and **eval**[4].

[3] In DrScheme from PLT, the functional if called **foldl**.

[4] **eval** is not part of the standard Scheme. However, it is present in most Scheme implementations as a worthwhile heritage from Lisp.

- **`quote`** returns its own parameter unevaluated. Therefore, **`quote`** acts like a shield against evaluation. For example, the expression **`(quote symbol)`** or, equivalently, **`'symbol`** produces the literal **`symbol`** as the result.

- Opposite to **`quote`**, **`eval`** evaluates the value of its parameter. For instance,

  ```
  (eval (list 'lambda '(x) '(+ 1 x)))
  ```

  produces the function constructed by evaluating the expression **`(lambda (x) (+ 1 x))`**.

These last examples show that data can be turned into functions dynamically. They also show that Scheme programs can work with symbolic data representations such as the list of symbols below that describes an instance of the blocks-world problem.

```
'((on a table) (on b a) (on_model b table) (on_model a b))
```

Streams

In similar ways as for defining non-strict functions, functions that process infinite objects can be written. First, it is important to observe that instead of representing an object as a cliché, we can describe the object by its making. The value *three* can be represented as the decimal constant **`3`** or by **`(succ (succ ( succ 0)))`**. The second representation shows that *three* is the value obtained by applying three times the **`succ`** operator starting from the constant **`0`**.

An infinite object, say the stream ⟨**`0 1 2 ... n ...`** ⟩ of natural numbers called **`naturals_stream`**, can be described as a pair made of the current number and a function with no parameters. When called, this function is able to compute the next pair of the stream (i.e. the remaining part of the stream). Similarly, this pair contains the next number of the stream and a function able to compute the next pair of the stream, and so on.

`The first value of the naturals_stream` $\equiv_{\text{def}}$
      `(0 . Function able to generate the next number of the naturals_stream)`

`The k_th value of the naturals_stream` $\equiv_{\text{def}}$
      `(k . Function able to generate the next number of the naturals_stream)`

If we call **`naturals`** the type of streams of natural numbers, then values of **`naturals_stream`** are pairs from the Cartesian product **`natural×(unit→naturals)`**, where the type **`unit→naturals`** corresponds to **`0`**-ary functions that compute streams of natural numbers.

```
(define succ (lambda (n) (list 'succ n)))

(define make_naturals
  (lambda (k)
     (cons k (lambda() (make_naturals (succ k))))))

(define naturals_stream  (make_naturals 0))

(define take
  (lambda (n stream)
    (if (= n 0) '()
        (cons (car stream) (take (- n 1) ((cdr stream)))))))

(take 4 naturals_stream)
(0 (succ 0) (succ (succ 0)) (succ (succ (succ 0))))
```

Observe the structure of the result obtained by applying **make_naturals** on **0**. It is the pair

```
(0 . the_closure)
    (lambda() (make_naturals (succ k)))
     k←0, make_naturals←a_function, succ←a_function
```

Since **(lambda() (make_naturals (succ k)))** is a closure, it saves the variables: **k** – bound to **0**, **make_naturals** – bound to the function that produces pairs of the form above, and **succ** – bound to a function that produces a symbolic representation of the successor of a number. The body of the closure is not evaluated unless the closure is explicitly applied on zero parameters. The expression **((cdr stream))** does exactly this job and, therefore, computes the next pair of the stream, i.e. the remaining part of the stream.

```
(1 . the_closure)
    (lambda() (make_naturals (succ k)))
     k←(succ 0), make_naturals←a_function, succ←a_function
```

It is in this way the stream unfolds in a controlled way, little by little. The function **take** is a good example in this respect. It computes the list of the first **n** natural numbers. Let's comment it.

```
(define take
  (lambda (n stream)
    (if (= n 0)
        then return the empty list '()
        else return the list made by cons applied on
               -  (car stream): the natural number at the top of stream
               -  (take (- n 1) ((cdr stream))): the list of the next n-1
                  numbers from the tail of the stream. The tail of the
                  stream is the result of ((cdr stream))
```

Certainly, the way of representing numbers, by the history of their making, can be dropped in favor of the more efficient cliché representation, where decimal constants are used. In order to modify the program above, we can benefit from the dynamic scoping of the top-level variables. We change simply the definition of **succ**.

```
(define succ (lambda (x) (+ x 1))
(take 4 naturals_stream)
(0 1 2 3)
```

In ways similar to the stream of natural numbers, other infinite objects or non-terminating computations can be represented in a finite way. As an additional interesting example, take the stream of Fibonacci numbers

$$\mathbf{Fibo} \equiv_{def} t_0\ t_1\ t_2\ t_3\ \ldots \text{, where } t_0=0,\ t_1=1,\ t_k=t_{k-2}+t_{k-1},\ k \geq 2.$$

The additive relationship between the terms of the stream can be used to cleverly compute **Fibo** as the term-wise addition of two streams: the first stream is **Fibo** itself, the second stream is the tail of **Fibo**, i.e. **(tail Fibo)** $= t_1\ t_2\ t_3\ \ldots$ To get the complete stream of Fibonacci numbers we have to insert in front of the stream computed by the addition the terms t_0 and t_1.

$$\begin{array}{rll}
\mathbf{Fibo} = & & t_0\ t_1\ t_2\ t_3\ \ldots\ t_{k-2}\ \ldots\quad + \\
\mathbf{(tail\ Fibo)} = & & t_1\ t_2\ t_3\ t4\ \ldots\ t_{k-1}\ \ldots \\
\hline
\mathbf{Fibo} = & t_0\ t_1 & t_2\ t_3\ t_4\ t_5\ \ldots\ t_k\ \ \ldots
\end{array}$$

If we call **add** the operation of adding two streams of numbers, then the construction of **Fibo** can be represented in a finite way as **Fibo =** t_0 t_1 **(add Fibo (tail Fibo))** and can be seen as illustrated in figure 1.

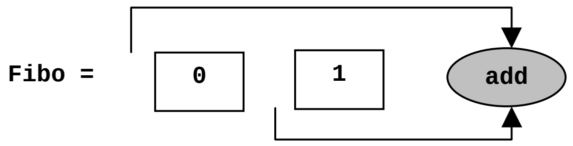

Figure 1. A finite device for computing the stream of Fibonacci numbers

First we have to define the operation **add**. If the streams added term by term are not finite, then **add** does not terminate. Our goal is to define **add** in such a way that it unfolds one step at a time and continues one step more only when necessary.

```
(define add
  (lambda (s1 s2)
    (cond ((null? s1) s2)
          ((null? s2) s1)
          (else (cons (+ (car s1) (car s2))
                      (lambda () (add ((cdr s1)) ((cdr s2)))) )))))
                                          ↑          ↑
                                       tail s1    tail s2
                       delayed computation
```

Observe that

(add $s1_0$ $s1_1$ $s1_2$ ... $s2_0$ $s2_1$ $s2_2$ ...) =

(cons (+ $s1_0$ $s2_0$) (lambda () (add $s1_1$ $s1_2$... $s2_1$ $s2_2$...)))

function that will perform the next addition step only when called

Therefore, the next term of the addition is computed on demand. In this way, we manage to specify in a finite way an eventually non-terminating computation. For instance, the **Fibo** stream can be easily constructed as shown below.

```
(define Fibo
  (cons 0
        (lambda() (cons 1
                        (lambda() (add Fibo ((cdr Fibo))))))))

(take 20 Fibo)
(0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181)
```

The mechanism of specifying streams and, more generally, the mechanism of specifying non-terminating computations, relies on mimicking normal-order evaluation in a language that works with applicative-order evaluation. We have to admit that the way we implemented the normal-order evaluation mechanism has two major pitfalls.

- The mechanism is clumsy. Writing **0**-ary closures that stand for the unevaluated expressions is cumbersome and non-natural for the common programmer.

- The implementation is inefficient. Trying the computations **(take 30 Fibo) (take 30 Fibo)** takes a long time. The reason is that some operations are performed repeatedly although they return always the same result. Consider that a non-strict function **F** is using

n times its unevaluated actual parameter **P** (the **0**-ary closure that stands for **P**). Then **n** times **F** has to evaluate the closure to obtain the same result. Indeed, it is exactly this way that things go in the Lambda calculus. However, Scheme is a programming language and we are justified when seeking for efficiency.

What do we want?

- We want a nice way to delay the evaluation of expressions and to force their evaluation whenever necessary. Note that **quote** and **eval** cannot be used, simply because they do not save the computational context of the expression whose evaluation we want to delay.

- We want to cache the value of a delayed expression when first evaluated and then to return the cached value for any subsequent evaluation of the expression.

Scheme comes with two predefined functions that are satisfying our goals: **delay** and **force**. They are important not only as control mechanisms in what is coined as *lazy evaluation* but, in addition, as a way of implementing the call-by-need parameter transfer mode.

- We also would like to be able to write at a level of abstraction close to our λ_0 language.

This last goal is fully possible. As proof, just have a look at and play with the **Lambda0.scm** file given below. It contains a Scheme transcription of λ_0. Moreover, if the first two goals mentioned above are solved, the Scheme transcription of λ_0 can be rewritten to be efficient. Rewriting the transcription, using **delay** and **force** Scheme primitives, is left as an exercise.

```
; The Scheme transcription of the Lambda_0 language.

; In Scheme there are predefined functions that have
; names identical to the names of data operators in Lambda_0.
; In order to avoid ambiguities due to name clashes, the
; names of Lambda_0 operators get the suffix "_0".

; Since Scheme uses applicative-order evaluation, all
; functions apart from if_0 are strict. if_0 is defined
; as a macro to act as a non-strict function.
; _________________________________________________________

(define true_0 (lambda (x y) x))
(define false_0 (lambda (x y) y))
(define not_0 (lambda (x) (x false_0 true_0)))
(define and_0 (lambda (x y) (x y false_0)))
(define or_0 (lambda (x y) (x true_0 y)))

(define-macro if_0
  (lambda (p x y) `((,p (lambda () ,x) (lambda () ,y)))))

(define cons_0 (lambda (x y) (lambda (z) (z x y))))
(define nil_0 (lambda (z) true_0))

(define car_0 (lambda (l) (l true_0)))
(define cdr_0 (lambda (l) (l false_0)))
(define null_0 (lambda (l) (l (lambda (x y) false_0))))

(define zero_0 nil_0)
(define succ_0 (lambda (n) (cons_0 'succ_0 n)))
(define pred_0 cdr_0)
(define zerop_0 null_0)
```

```
; ________________________________________________________
; Fixed-point combinators for 1-ary and 2-ary functions

(define c1
  (lambda (f)
    ((lambda (g) (lambda (x) (f (g g) x)))
     (lambda (g) (lambda (x) (f (g g) x))))))

(define c2
  (lambda (f)
    ((lambda (g) (lambda (x y) (f (g g) x y)))
     (lambda (g) (lambda (x y) (f (g g) x y))))))

; ________________________________________________________
; Recursive operators for lists and numbers

(define append_0
  (lambda (a b)
    ((c1 (lambda (cont l)
           (if_0 (null_0 l) b
                 (cons_0 (car_0 l) (cont (cdr_0 l))))))
     a)))

(define length_0
  (c1 (lambda (cont L)
        (if_0 (null_0 L) zero_0 (succ_0 (cont (cdr_0 L)))))))

(define reverse (lambda (L) ((c2 rv) nil_0 L)))

(define rv
  (lambda (cont R L)
    (if_0 (null_0 L) R (cont (cons_0 (car_0 L) R) (cdr_0 L)))))

(define lt_0
  (c2 (lambda (cont n m)
        (if_0 (zerop_0 m) false_0
              (if_0 (zerop_0 n) true_0
                    (cont (pred_0 n) (pred_0 m)))))))

(define eq_0
  (c2 (lambda (cont n m)
        (if_0 (zerop_0 n) (zerop_0 m)
              (if_0 (zerop_0 m) false_0
                    (cont (pred_0 n) (pred_0 m)))))))

(define +_0 append_0)

(define -_0
  (c2 (lambda (cont x y)
        (if_0 (zerop_0 y) x (cont (pred_0 x) (pred_0 y))))))

(define *_0
  (lambda (x y)
    ((c1 (lambda (cont x)
           (if_0 (zerop_0 x) zero_0 (+_0 y (cont (pred_0 x))))))
     x)))

(define div_0
  (lambda (x y)
    ((c1 (lambda (cont x)
           (if_0 (lt_0 x y) zero_0 (succ_0 (cont (-_0 x y))))))
     x)))
```

```
(define mod_0
  (lambda (x y)
    ((c1 (lambda (cont x)
           (if_0 (lt_0 x y) x  (cont (-_0 x  y)))))
     x)))

; ________________________________________________________
; Streams. The stream of prime numbers

(define one (succ_0 zero_0))
(define two (succ_0 one))

(define is_prime
  (lambda (n)
    ((c1 (lambda (cont divisor)
           (if_0 (eq_0 divisor one) true_0
                 (and_0 (not_0 (zerop_0 (mod_0 n divisor)))
                        (cont (pred_0 divisor))))))
     (div_0 n two))))

(define primes
  ((c1 (lambda (cont n)
         (if_0 (is_prime n) (cons_0 n (lambda () (cont (succ_0 n))))
               (cont (succ_0 n)))))
   two))

; ________________________________________________________
; (make_stream f t0) builds the stream t0 f(t0) f(f(t0)) ...
; Since f does not vary make_stream c1 fixed-point combinator

(define make_stream
  (lambda (f t0)
    ((c1 (lambda (cont t) (cons_0 t (lambda () (cont (f t))))))
     t0)))

; hd_0 returns the first element of a stream
; tl_0 returns the tail of a stream, which is a stream itself

(define hd_0 car_0)
(define tl_0 (lambda (stream) ((cdr_0 stream))))

; nth returns the n-th element of a stream.
; The first stream element has the index 0.

(define nth
  (lambda (n stream)
    ((c2 (lambda (cont n stream)
           (if_0 (zerop_0 n) (hd_0 stream)
                 (cont (pred_0 n) (tl_0 stream)))))
     n stream)))

;________________________________________________________
; The stream of natural numbers and the stream of Fibonacci
; numbers

(define natural (make_stream succ_0 zero_0))

(define fibo
  (make_stream
   (lambda (pair) (cons_0 (cdr_0 pair)
                          (+_0 (car_0 pair) (cdr_0 pair))))
   (cons_0 zero_0 (succ_0 zero_0))))
```

```
; The n-th Fibonacci number

(define nth-fibo
  (lambda (n) (car_0 (nth n fibo))))

;_________________________________________________________
; Conversion functions Scheme <-> Lambda_0 (textual recursive)

; number(n) converts a Scheme number n to the equivalent
; number in Lambda_0

(define number
  (lambda (n)
    (if (= n 0) zero_0 (succ_0 (number (- n 1))))))

; value(n) converts a number n from Lambda_0 to the
; equivalent number in Scheme

(define value
  (lambda (n)
    (if_0 (eq_0 n zero_0) 0 (+ 1 (value (pred_0 n))))))
;_________________________________________________________
; Testing some operations from Lambda_0

(value (nth (number 8) primes))
(value (nth-fibo (number 10)))
(value (div_0 (number 20) (*_0 two two)))


;_________________________________________________________
; The stream of prime numbers

(define sift
  (c2 (lambda (cont n candidates)
        (if_0 (zerop_0 (mod_0 (hd_0 candidates) n))
              (cont n (tl_0 candidates))
              (cons_0 (hd_0 candidates)
                      (lambda () (cont n (tl_0 candidates))))))))

(define eratostene
  (c1 (lambda (cont candidates)
        (cons_0 (hd_0 candidates)
                (lambda () (cont (sift (hd_0 candidates)
                                       candidates)))))))

(define all_primes
  (eratostene (tl_0 (tl_0 natural))))

(define nth_primes
  (lambda (n) (value (nth (number n) all_primes))))

(nth_primes 3)
(nth_primes 7)
```