Continuations, Exceptions, Coroutines and Generators

In conventional programming the application of a function is considered as a monolith computational process. All the intermediate transformation steps performed by the function upon the arguments are hidden to the function caller. Another possible point of view is to consider the function as a pipe fed at one end with its arguments and producing at the other end the expected result. Data flowing through the pipe are transformed according to the logic of the function. With this view, we can ask whether the pipe can be cut at some point thus obtaining two fragments:

- A fragment that performs a part of the computation process carried out by a particular function application.
- A fragment that corresponds to the continuation of the computation process performed by the function.

If this would be possible then we could consider the continuation of the computation process as a first class value that later could be fed with data to complete the computation of the function application. This scenario opens ways to interesting programming techniques since the solving process of a problem could then be divided into a set of cooperating fragments that resume and interrupt one another according to a distributed control schema.

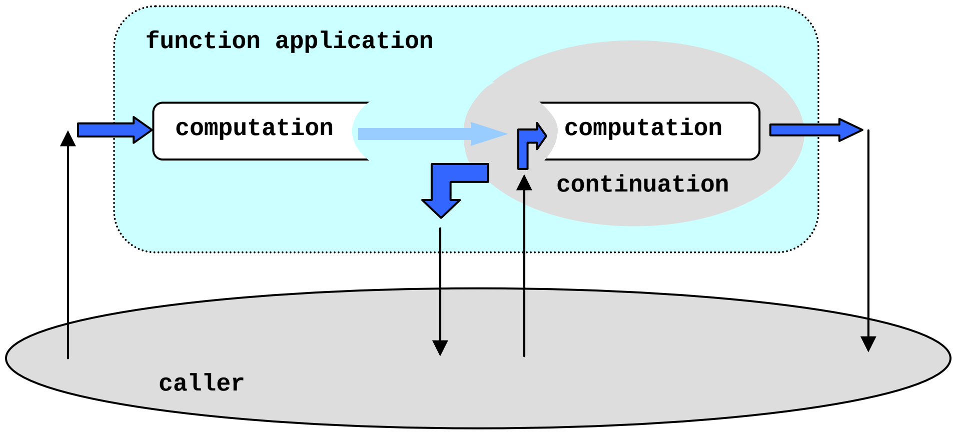


Figure 1. The continuation of a computation process

Scheme offers the *continuation* mechanism as a way of controlling the processing flow of a program. In Scheme a continuation is an **1**-ary function (a closure) that saves the computation environment of an interrupted computation process. The body of the function is the continuation of the interrupted process. Since functions are first class values, a continuation can be bound to a variable, transferred as parameter to a function or returned as the result of a function. When called with an actual parameter **V**, the continuation abandons the current computation process and resumes the computation process closed within the continuation. The resumed computation process works on **V**. The primitive Scheme function **call_with_current_continuation** (or **call/cc** for short) acts as the constructor of continuations.

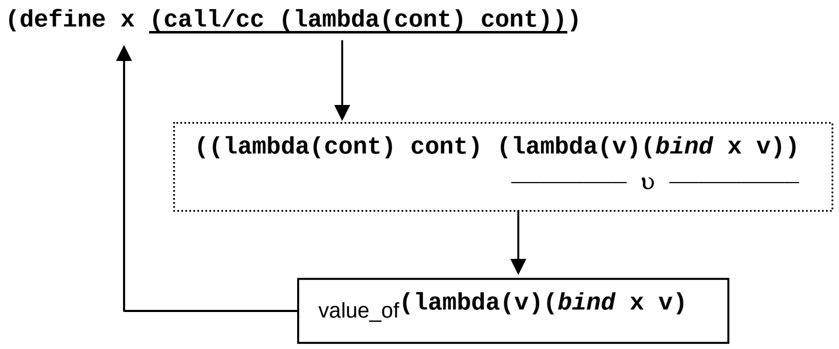
As a first example, consider the following three expressions. The first expression binds the top-level variable **x** to a continuation created by using **call/cc**. The second expression applies the continuation on a generic expression ***expr***. The third expression evaluates **x**. The result is $_{\texttt{value\_of}}$***expr***.

$$\underbrace{\texttt{(define x (call/cc (lambda (cont) cont)))}}_{\mu}$$

```
(x expr)
x
```

$_{\texttt{value\_of}}$***expr***

The evaluation of the first expression proceeds as illustrated in figure 2 and explains why, finally, the value of **x** is $_{\texttt{value\_of}}$***expr*** instead of the continuation that was initially bound to **x**.



(define x (call/cc (lambda(cont) cont)))
((lambda(cont) cont) (lambda(v)(bind x v))
υ
value_of(lambda(v)(bind x v)

Figure 2. Binding **x** to the process of its own binding

The evaluation of **define** starts. For **x** to be bound, the value of the expression

$$\underbrace{\texttt{(call/cc (lambda (cont) cont))}}_{\mu}$$

must be available. It is the value that will be bound to **x**. The evaluation of μ is as follows:

1. A continuation is build by **call/cc**. The continuation is in fact the continuation of the binding process of **x**, process which expects a value **v** to be bound to **x**. Since the value **v** is not known at this time (the **call/cc** application is not yet terminated) the continuation is equivalent to the **1**-ary closure produced by the expression:

$$\underbrace{\texttt{(lambda (v) (}\textit{bind}\texttt{ x v))}}_{\upsilon}$$

2. The function application **((lambda (cont) cont)** υ**)** is evaluated. The result is the closure $_{\texttt{value\_of}}\upsilon$. Nevertheless, $_{\texttt{value\_of}}\upsilon$ has a special effect when applied: it abandons the current computation process replacing it by the process of the $_{\texttt{value\_of}}\upsilon$ evaluation. For this reason continuations have a treatment different from that of ordinary closures.

Finally, the top-level variable **x** is bound to the value returned by **call/cc**, i.e. $_{\texttt{value\_of}}\upsilon$

$$\texttt{x} \leftarrow {}_{\texttt{value\_of}}\upsilon \;=\; {}_{\texttt{value\_of}}\texttt{(lambda (v) (}\textit{bind}\texttt{ x v))}$$

Now consider the application **(x *expr*)**, where ***expr*** is any Scheme expression. The application proceeds as follows:

1. The parameters of the application are evaluated in an unspecified order. The value of **`x`** is `value_of`**`(lambda (v) (`*`bind`*` x v))`**; the value of ***`expr`*** is `value_of`***`expr`***.

2. The current computation process is abandoned and replaced by the evaluation of the application of the closure `value_of`**`(lambda (v) (`*`bind`*` x v))`** on `value_of`***`expr`***. The variable **`x`** is bound to `value_of`***`expr`***.

The example above shows the way **`call/cc`** and continuations are behaving: (a) **`call/cc`** expects an **`1`**-ary function as argument

$$\texttt{(call/cc } \underbrace{\texttt{(lambda (cont) }\textit{body}\texttt{)}}_{\mathbf{F}}\texttt{)}$$

(b) constructs a continuation that is equivalent to the value of the expression

$$\underbrace{\texttt{(lambda(x) continuation of current computation that expects a value x to proceed)}}_{\mathbf{C}}$$

and (c) returns the value of the application **`(F C)`**. Therefore, the expression **`(call/cc id)`** where **`id`** is the identity function **`(lambda (x) x)`** returns the continuation of the current computation.

Applying a continuation **`C`** on to an actual parameter, call it ***`expr`***, is similar to the application of a function. The effect of **`(C`** ***`expr`*****`)`** is:

- The current computation process is abandoned.
- The computation process closed within the continuation **`C`** restarts with the value `value_of`**`expr`**.

As a simple application of continuations consider the simulation of a long jump used to exit from the depth of a computation process and to transfer the computed value to a distant calling function. This case occurs when a function is terminal recursive, i.e. it does not modify the computed value along the return chain. An example is the function **`length`** for computing the length of a list. It is predefined in Scheme. Below is a possible redefinition.

```
(define length
  (lambda (L)
    (call/cc (lambda (throw)
               (letrec ((aux (lambda(L lg) (if (null? L) (throw lg)
                                               (aux (cdr L) (+ 1 lg))))))
                 (length L 0))))))
```

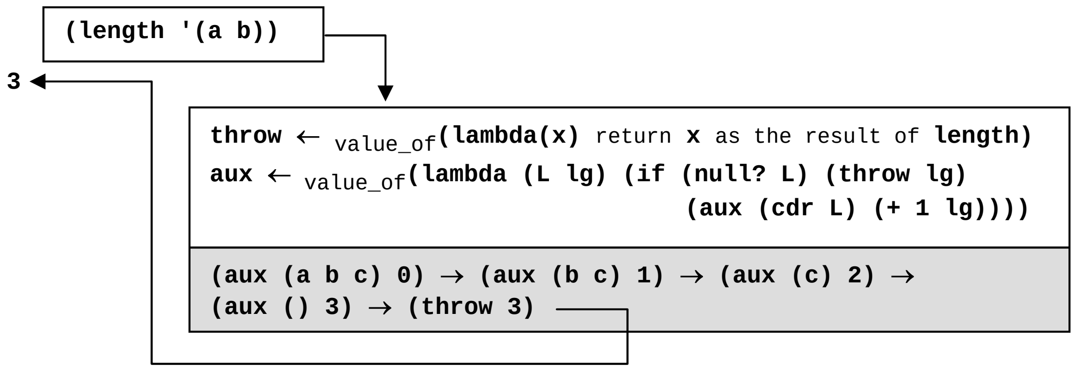

(length '(a b))
3
throw ← value_of(lambda(x) return x as the result of length)
aux ← value_of(lambda (L lg) (if (null? L) (throw lg)
(aux (cdr L) (+ 1 lg))))
(aux (a b c) 0) → (aux (b c) 1) → (aux (c) 2) →
(aux () 3) → (throw 3)

Figure 3. Throwing the result of a terminal recursive function to the top level call

Exceptions

Exceptions make a useful control mechanism that is present in modern programming languages such as C++ and Java, and is common for long in Lisp and other functional programming languages. Basically, the exception mechanism allows for the direct transfer of a value from a point of a computation environment to a point within an enclosing computation environment. The mechanism has two phases:

1. Setting an exception trap, which is a computation **`T`** expecting a special value - called exception - to proceed.

2. Executing a computation **`P`** from within the computation environment of the trap.
   - If **`P`** is terminating normally, i.e. without throwing (or raising) an exception) the trap **`T`** returns the value as its own value.
   - If **`P`** throws an exception, the computation **`P`** is abandoned and the exception is caught by the trap **`T`**. The trap starts to work and processes the exception. The result is a specific value and/or a specific side effect (such as aborting the program or reporting an error).

Normally, exceptions are tagged so the traps of a program can selectively catch them. Since the exception mechanism is dynamic and has no relationship with the textual structure of the program, the tag (label) of a trap behaves as a dynamic scoped variable. The tag is bound on the dynamic chain of the program so that the LIFO strategy applies for the exception traps.

Scheme is not equipped with a predefined exception handling mechanism. The reason is clear: continuations can well substitute the exception mechanism. In the **`length`** example above, the trap set by the **`call/cc`** catches the length of a list. The result is thrown from somewhere deep in the computation process of the length of a given list.

In this section we extend Scheme by implementing a more versatile and explicit exception handling mechanism of the kind found in Lisp. The exception mechanism uses two functions: **`catch`** and **`throw`**.

`(catch` ***`expression label`*****`)`** sets the trap for the exceptions tagged by ***`label`*** and starts to evaluate the ***`expression`***. If the ***`expression`*** throws an exception tagged ***`label`***, and the trap catches this exception[1], the result of **`catch`** is the value packed within the exception. If no exception is thrown, the value of **`catch`** is the value of the ***`expression`***.

`(throw` ***`expression label`*****`)`** throws an exception whose packed value is the value of the ***`expression`*** and whose tag is ***`label`*** (unevaluated).

The essential operation here is that **`throw`** must be able to resume the computation environment of the matched trap. Then **`(catch`** ***`expression label`*****`)`** must create such an environment before starting the evaluation of the ***`expression`***. The environment is a continuation of the **`catch`** operation. It is stored as a pair **`(`*****`label . catch_continuation`*****`)`** in a global stack used by the catch-throw mechanism.

An exception itself is a pair **`(label . thrown_value)`** fabricated by **`throw`**. The tag of the exception is searched in the global stack. If not found, there is an error. If found, the top part of the stack down to and including the found entry is popped off the stack and the ***`catch_continuation`*** from the found entry is applied on the **`thrown_value`**.

[1] There might be several traps with the same tag that are set during the evaluation of the ***`expression`***.

```
(define $catch-stack$ '())

; __________ catch ________________________________________________
; (catch expr label) catches the value thrown while evaluating expr
; _________________________________________________________________
(define-macro catch
  (lambda (expr label)
    `($catch-core$ (lambda () ,expr) ',label)))

(define $catch-core$
  (lambda (expr_closure label)
     (let* ((stack-top $catch-stack$)
            (result (call/cc
                       (lambda (cont)
                          (set! $catch-stack$
                                (cons (cons label cont) $catch-stack$))
                          (expr_closure)))))
        (set! $catch-stack$ stack-top)
        result)))

; __________ throw ________________________________________________
; (throw value label) throws the value to the corresponding
; catch (as indicated by the label). If there are several identical
; labeled catches the LIFO strategy is used.
; _________________________________________________________________
(define-macro throw
  (lambda (expr label)
    `($throw-core$ ,expr ',label)))

(define $throw-core$
  (lambda (value label)
     (let ((entry (assq label $catch-stack$)))
       (if entry
           ((cdr entry) value)
           (begin
              (set! $catch-stack$ '())
              (error "undefined catch for" label))))))
```

The macro call **(catch *expression label*)** generates and evaluates the application

($catch-core$ (lambda () *expression*) '*label*))

Indeed, the ***expression*** must not by evaluated before the body of **$catch-core$** is evaluated since it may throw the exception before the trap is set. Moreover, the ***label*** must not be evaluated at all and therefore it is protected by **quote**. These points show an interesting feature of Scheme and other similar languages (e.g. Lisp): programming is in two dimensions. One is the logic of the program; the other is the evaluation time of the expressions that encode the logic of the program. Later on, when discussing statically typed functional languages such as ML, a third dimension will be added: types.

The function **$catch-core$** saves the current top of **$catch-stack$** and then binds the local variable **result** to the result of

```
(call/cc
  (lambda (cont)
     (set! $catch-stack$
           (cons (cons label cont) $catch-stack$))
     (expr_closure)))
```

This expression is the core of the full catch-throw operation. **call/cc** creates the ***catch_continuation*** and passes it to an anonymous function which:

- Pushes the pair **(*label . catch_continuation*)** into the **$catch-stack$**.
- Calls the **expr_closure** in order to evaluate the closed expression, call it **E**.

If the expression **E** does not throw an exception, then its value is the value of the anonymous closure and implicitly of **call/cc**. The call of **catch** returns with this value.

If the expression **E** throws an exception then the ***catch_continuation*** will be called with the value packed within the exception. The current computation process is abandoned and the binding process of the **result** variable is resumed. Finally, the variable **result** is bound to the value packed within the exception and the **catch** call returns with this value.

The macro call **(throw *expression label*)** generates and evaluates the application

($throw-core$ *expression 'label*)

This time the ***expression*** must be evaluated prior to the effective application of **$throw-core$**, whereas ***label*** must not be evaluated and therefore is quoted. The first action of **$throw-core$** is to bind the local variable **entry** to the topmost entry in the **$catch-stack$** that matches the ***label***. Then **(cdr entry)** is the ***catch_continuation*** that must be called with the value of the **expression**. If no entry in the **$catch-stack$** matches the ***label***, an error is signaled and the whole program is aborted.

To exercise the catch-throw exception mechanism consider defining a predicate **is-list?** that tests whether its argument is a list. The predicate is using **catch** and **throw** as a shortcut escape from a possible deep terminal recursion.

```
; An example of using catch/throw. is-list? tests if the argument is
; a list (i.e. <list> ::= () | (<value> . <list>)
; ________________________________________________________________
(define is-list?
  (lambda (l)
     (letrec ((f (lambda (l)
                   (cond ((eq? l '()) (throw #t is-list-catch))
                         ((not (pair? l)) (throw #f is-list-catch))
                         (else (f (cdr l)))))))
        (catch (f l) is-list-catch))))
```

The definition of the catch-throw exception handling mechanism in Scheme speaks itself for the power of the languages in the Scheme class. The language can be extended with sophisticated control and processing facilities written directly in Scheme. This is often the case with many add-on-top facilities that go beyond the standard and that are offered by different implementations of the language.

Coroutines

Coroutines are similar to continuations in the respect that they are control abstractions represented by a closure (a code address and a computation environment). However, there is an essential difference between continuations and coroutines. Each time a continuation is called, it executes the same computation. In other words, a continuation is constant from the point of view of how it behaves. It has no internal state provided we are in a pure functional programming realm.

A coroutine changes its own state each time when called:
- The coroutine does not destroy the current computation process.
- The coroutine saves its exit (return) point. When called again the coroutine resumes work from its last exit point.

A coroutine has an internal state. Not only the entry point varies in time but also the saved computation environment may change over time. Coroutines are computation environments that change their state, exist simultaneously and execute serially, one at a time. The transfer of control from a coroutine to another coroutine is explicit, using coroutine references. A schematic view of coroutine behavior is given in figure 4. Two coroutines, **`A`** and **`B`**, call one another. Each coroutine restarts work from the exit point that is set using a **`resume`** command addressed to the other coroutine. On re-entry, the **`resume`** returns the value transferred by the calling coroutine.

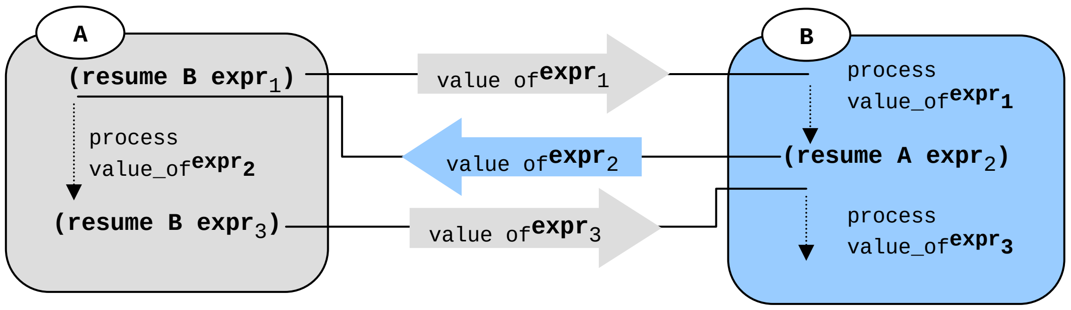


Figure 4. Two coroutines

The coroutining mechanism can be implemented quite easily in Scheme using continuations. A coroutine is an object characterized by:

- An **`1`**-ary skeleton function that encodes the computation performed by the coroutine.
- A state corresponding to the computation environment of the skeleton function (saved when the skeleton function is interrupted). This state is a continuation that closes the computation environment and the re-entry point of the skeleton function.

From the representation point of view, a coroutine is a tagged pair

`(coroutine` ***`skeleton`*** **`.`** ***`skeleton_continuation`*** **`)`**

where ***`skeleton`*** is the skeleton function and the ***`skeleton_continuation`*** is the continuation of the skeleton function. A global variable, called **`this`**, stores the tagged pair corresponding to the active coroutine.

The operations for coroutine handling are enumerated below. The parameter transfer is by value in all cases.

- **`(coroutine`** ***`function`*** **`)`** returns a new coroutine the skeleton of which is the `value_of`***`function`***. Note that there can be several coroutines with the same skeleton that exist simultaneously in a program.

```
(define coroutine
  (lambda (skeleton)
    `(coroutine ,skeleton . ,skeleton)))
```

- **`(coroutine?`** ***`object`*** **`)`** tests whether the `value_of`***`object`*** is a coroutine.

```
(define coroutine?
  (lambda (cor)
    (and (pair? cor) (eq? (car cor) ‘coroutine))))
```

- **(resume *coroutine expression*)** resumes the value_of***coroutine*** transferring the value_of***expression***. In order to be able to keep trace of the last activated coroutine we use a global variable called **this**. It is guaranteed that while a coroutine is active the value of **this** is the coroutine itself. Initially, the value of **this** is a **$top_level_coroutine$** whose skeleton is the identity function. This coroutine has a major role in interfacing the native Scheme environment with the implemented coroutine system.

```
(define $top_level_coroutine$ (coroutine (lambda (x) x)))
(define this $top_level_coroutine$)

(define resume
  (lambda (cor arg)
    (if (coroutine? cor)
        (call/cc (lambda (cont)
                   (set-cdr! (cdr this) cont)
                   (set! this cor)
                   ((cddr cor) arg)))
        (error "Can't resume a non coroutine"))))
```

Notice that the variable **cont** is bound by **call/cc** to the continuation of the current active coroutine (bound to **this**). This value is saved in the coroutine itself and then the variable **this** is changed to the resumed coroutine **cor**. Finally **cor** is called with the value from the **resume** operation, i.e. the value passed by the former active coroutine.

Resuming a coroutine has side effects on the coroutine that issues the command. The content of the data structure that represents the coroutine is modified. Here the **set-cdr!** Scheme primitive is used. The application **(set-cdr! *pair expression*)** expects that ***pair*** evaluates to a pair **(*x . y*)**, and physically replaces the value ***y*** within the pair with the value of the ***expression***. Therefore, the modification of the pair will be felt by all the expressions that use the pair. In our case, modifying a coroutine will be implicitly felt by all the users of the coroutine (other coroutines).

- **(return *expression*)** interrupts the active coroutine and performs a normal return. Usually, a coroutine does not return. It simply resumes another coroutine. In the particular case when a coroutine must perform a normal return, in order to exit from the coroutine system, it has to use the **(return *expression*)** operation instead of **resume**. The **return** operation resumes the default coroutine called **$top_level_coroutine$**. The default coroutine stores the continuation of the first **resume** operation performed from outside the coroutine system. This continuation restarts working with the value of the ***expression***.

```
(define return
  (lambda (arg) (resume $top_level_coroutine$ arg)))
```

- **(reset *coroutine*)** resets the entry point of the skeleton function of the value_of***coroutine***. The subsequent entry will be at the top of the function.

```
(define reset
  (lambda (cor)
    (if (coroutine? cor)
        (begin (set-cdr! (cdr cor) (cadr cor)) cor)
        (error " Can't reset a non coroutine "))))
```

The coroutine operators **resume** and **reset** as implemented above are subject to some restrictions. For example it would be strange for a coroutine to **resume** or **reset** itself. In addition, there is a slight non-uniformity of interfacing the Scheme functional environment with the coroutine system.

Generators

Consider a problem **P** the solution of which results through the combination of a set of solutions of some sub-problems P_i, **i=1,n** of **P**. Let G_i be an object capable of producing, each time it is activated, a new solution of the sub-problem P_i. Obviously, G_i must work independently and must save its state in such a way that, when re-activated, be able to continue the interrupted computation process for generating the next solution. We say that G_i is a *generator* of partial solutions for the problem **P**. The solving process of **P** can be seen as a process of generating partial solutions for the sub-problems of **P** and then combining these solutions into complete solutions of **P**. The activation of the generators can be controlled by a central core function or it can be distributed over the generators themselves. In the later case the generators behave as a group of independent active objects that cooperate to solve the problem. We say that the problem solving is distributed over the set of generators. This technique has been used extensively in the AI domain. More recently the technique has been borrowed as support for distributed applications in computer networks. A distributed application splits the logical solving process of a problem into smaller fragments executed on several physical machines that are interconnected according to a given topology and controlled by a specific communication protocol.

Here the distributed problem solving - as different from a distributed application - is seen as the conceptual fragmentation of the solving process of a problem into a set cooperating active components. An orthogonal viewpoint of distributed problem solving addresses the way the activations of the solving components are interleaved. If the components are executed in parallel we say that the problem solving is performed concurrently.

Generators are useful when the given problem is solved by navigating in a large solution space (eventually infinite) in search for an acceptable solution. The CAD problems are in this category. Partial solution generators may prove simpler than monolith algorithms. Moreover, the distributed solving is an ideal base for applying interesting programming techniques that simplify the overall solving program.

As an application, let us rewrite the breadth-first search function using coroutines. The resulting coroutine can be used to produce, in a controlled way, different solutions for a given problem. Notice that the set of solutions may not be finite, which justifies the use of such an incremental generator. In particular, what we want is to define a skeleton function, called **search**, which when packed into a coroutine is able to compute, each time when called, a new solution of the problem solved. The skeleton function gets a single parameter: the calling coroutine. Each time it computes a solution, **search** resumes its **caller** thus suspending itself. Each time it is called (resumed), **search** is re-entered from its last interruption point.

The problem solved by **search** is described given:

- An initial state, called **ini_state**. The search process starts from this state.
- A predicate, called **solution?**, able to test whether a given state is a solution.
- A function, called **expand**, which generates the set (represented as a list) of the valid successor states of a given problem state.

```
(define search
  (lambda (ini_state solution? expand)
    (lambda (caller)
      (let search ((states (list ini_state)))
        (cond ((null? states) (resume caller #f))
              ((solution? (car states))
               (resume caller (car states))))
        (search (append (cdr states) (expand (car states))))))))
```

As an application, we define a palindrome coroutine builder and then a function able to construct several independent generators for a given palindrome problem.

```
; (gen_gen C) returns a solution generator for the solutions
; that can be generated by the coroutine C. The generator is
; a function which when called with an integer N returns the
; next N solutions generated by C
; _________________________________________________________
(define gen_gen
  (lambda (root)
    (lambda (n)
      (let cycle ((nr n) (solutions '()))
        (if (<= nr 0) (reverse solutions)
            (let ((solution (resume root this)))
              (if solution
                  (cycle (- nr 1) (cons solution solutions))
                  (begin (reset root) (reverse solutions))))))))))


; (palindrome min_length symbols) returns the coroutine able
; to step-wise generate palindromes made of symbols and that
; are at least min_length long
; _________________________________________________________
 (define palindrome
   (lambda (min_length symbols)
     (coroutine
       (search '()
             (lambda (state)
               (and (>= (length state) min_length)
                    (equal? state (reverse state))))
             (lambda (state)
               (map (lambda (symb) (cons symb state)) symbols))))))


; (gen1 N) and (gen2 N) squeeze the next N palindromes
; out of different coroutines that have the same skeleton
; _________________________________________________________
(define gen1 (gen_gen (palindrome 4 '(a b c))))
(define gen2 (gen_gen (palindrome 4 '(a b c))))

; Try them and notice they work independently
; ___________________________________________

(define g1
  (lambda(n) (printf "gen1 is squeezing ~a palindomes~n~a" n (gen1 n))))

(define g2
  (lambda(n) (printf "gen2 is squeezing ~a palindomes~n~a" n (gen2 n))))

(begin (g1 4) (g2 5) (g1 3) (g2 2) (g1 3))
gen1 is squeezing 4 palindomes
((a a a a) (a b b a) (a c c a) (b a a b))
gen2 is squeezing 5 palindomes
((a a a a) (a b b a) (a c c a) (b a a b) (b b b b))
gen1 is squeezing 3 palindomes
((b b b b) (b c c b) (c a a c))
gen2 is squeezing 2 palindomes
((b c c b) (c a a c))
```

Notice that **gen_gen** and **search** are general: **search** can be used for any problem solved by breadth-first search; **gen_gen** can be used to build solution generators for any solving coroutine provided that the coroutine is signaling the termination of solutions by the Boolean constant **#f**.

Iterators

Iterators are particular generators. They select elements from data containers according to specific selection rules. For example, an iterator for a binary search tree may incrementally select the keys of the tree following the left-root-right traversal of the tree. An iterator of a list may select the list elements according the left-to-right list traversal etc. In all cases the essential characteristic of an iterator is the incremental nature of the selection process. Each time the iterator is called a group of the next elements of the container is selected and returned.

Here we adopt the following conventions:

- An iterator of a given container is a function that takes an integer parameter **N** and returns the list of the next **N** selected elements from the container.
- If the elements of the container are exhausted, the length of the list returned is shorter than **N**. Eventually, the list returned can be empty.
- If called again, after the elements of the container are exhausted, the iterator restarts to work afresh.

We have at hand powerful control abstractions that make it possible to implement in a simple and elegant way a general iterator building mechanism. The function below builds an iterator provided it gets a container and a selection rule for the elements of the container. The selection rule is a function that when applied onto a given container returns a coroutine. When resumed, the coroutine returns the next element from the container. If the elements are exhausted, the coroutine returns **#f**. The control mechanism of the iterator is provided by the **gen_gen** function.

```
; (make_iterator container select_rule) builds an iterator
; for the container. The coroutine produced by the application
; (select_rule container) is used for selecting the container
; elements one at a time.
; ____________________________________________________________
(define make_iterator
  (lambda (container select_rule)
    (gen_gen (select_rule container))))
```

The above simple function is general. As an example, let us define two iterators:

- An iterator for a list that selects the elements from left to right.
- An iterator for a binary search tree that selects the elements according to the left-root-right traversal order of the tree.

<u>Defining an iterator for a list</u>

```
; list_select_rule is the selection coroutine for a list
; _______________________________________________________
 (define list_select_rule
  (lambda (list)
    (coroutine
     (lambda (caller)
       (let traverse ((L list))
         (if (null? L) (resume caller #f)
             (begin (resume caller (car L)) (traverse (cdr L)))))))))

(define list_iter
  (make_iterator '(a b c d e f g h i k) list_select_rule))
```

Defining an iterator for a binary search tree

```
(define empty? null?)
(define node (lambda (ls k rs) `(,k ,ls . ,rs)))
(define ls cadr)
(define rs cddr)
(define key car)
(define empty_tree '())

(define insert_key
  (lambda (k tree)
    (if (empty? tree) (node empty_tree k empty_tree)
        (let ((l (ls tree)) (x (key tree)) (r (rs tree)))
          (cond ((> k x) (node l x (insert_key k r)))
                ((< k x) (node (insert_key k l) x r))
                (else tree))))))


; (build_tree key_list) builds a binary search tree
; from the keys in the key_list. The keys are inserted
; in the order they occur in the key_list.
; ____________________________________________________
(define build_tree
  (lambda (key_list)
    (foldl insert_key empty_tree key_list)))


; tree_select_rule is the selection coroutine for a key
; container that is structured as a binary search tree.
; Each time the coroutine is resumed it returns the next
; key from the tree according to the left-root-right
; tree traversal.
; ______________________________________________________
(define tree_select_rule
  (lambda (tree)
    (coroutine
     (lambda (caller)
       (let search ((tree tree))
         (if (not (empty? tree))
             (begin (search (ls tree))
                    (resume caller (key tree))
                    (search (rs tree)))))
       (resume caller #f)))))

(define tree_iter
  (make_iterator (build_tree '(4 1 6 8 2 3 9 3 7 0 2 5 10))
                 tree_select_rule))

; Try the iterators
(list_iter 2)
(a b)
(tree_iter 3)
(0 1 2)
(list_iter 5)
(c d e f g)
(tree_iter 10)
(3 4 5 6 7 8 9 10)
(tree_iter 1)
(0)
(tree_iter 20)
(1 2 3 4 5 6 7 8 9 10)
(list_iter 30)
(h i k)
```