

Principles of Programming Languages

Continuation Passing Style

We return to Scheme and investigate functional techniques to model and understand advanced control structures in addition to conditionals, function invocation and iteration.

The traditional control structures in programming languages are:

- Conditionals: are provided in Scheme with the special forms **if** and **cond** - they must be supported as part of the operational semantics of the language.

- Sequence: are supported in Scheme in the body of procedures and let expressions; they are useful when we execute procedures with side effects.

- Recursion and Iteration: There is no special form in Scheme for loops and iterations. Instead, loops are implemented using a specific form of recursion.

Consider the comparison between these two functions computing the factorial of a number n :

Conditionals, Sequence, Recursion and Iteration in Scheme

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

```
(define fact-iter
  (lambda (n acc)
    (if (= n 0)
        acc
        (fact-iter (- n 1) (* n acc)))))
```

The evaluation of **fact** generates a **recursive process** - the steps of the evaluation look as follows:

```
(fact 6)
(* 6 (fact 5))
...
(* 6 (* 5 (* 4 (* 3 (* 2 (* 1 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1))))))
(* 6 (* 5 (* 4 (* 3 2))))
...
(* 6 120)
720
```

In contrast, the **fact-iter** function generates an **iterative** process - the steps of the evaluation look as follows:

```
(fact-iter 6 1)
(fact-iter 5 6)
(fact-iter 4 30)
(fact-iter 3 120)
(fact-iter 2 360)
(fact-iter 1 720)
720
```

Observe that each call of **fact** in the trace is executed in a control context - which indicates what computation is to be done after the call completes. We see that this context grows in each successive call - until the base case of the recursion is reached.

At this point, the accumulated context is executed, step by step, and this control context is consumed, until the result of the recursion is obtained. This type of recursive process execution consumes memory in this case proportional to the input parameter n . The memory is consumed on a stack of frames.

In contrast, in the process generated by **fact-iter** there is no control context generated: each time the call to **fact-iter** is executed, there is no next computation that needs to be done with the return value of **fact-iter**. We say that **fact-iter** occurs in **tail position** in the function. Such procedures can be executed without consuming stack frames.

The operational semantics of Scheme requires that the execution of **tail recursive** procedures do not consume control memory. Hence, tail recursion is the construct used in Scheme to support **iteration**.

Observe that the iterative procedure **fact-iter** has two parameters - one represents the current number over which is to be computed, the second one is an accumulator, in which we incrementally compute the result of the computation.

While Scheme requires that tail recursion be implemented in an iterative manner, JavaScript does not impose this behavior. How can we find out whether Node.js implementation is tail-recursive?

The easiest way is to try it: recursive calls consume memory on a stack. This stack is usually limited in size to avoid an infinite recursive call from consuming all the RAM of the process.

If we run a tail-recursive procedure in JavaScript in Node.js, we find out it fails with an error of type “Stack Overflow” - which indicates the execution consumes stack memory - or in other words, Node.js does not implement tail recursion in an iterative manner.

Tail Recursive Implementation

```
const factIter = (n: number, acc: number): number =>
  n === 0 ? acc :
  factIter(n - 1, n * acc);

console.log(factIter(1000000, 1));
// RangeError: Maximum call stack size exceeded
```

What can we infer from this fact about the following question:

- Our interpreter for L5 is written in JavaScript, which does not implement tail recursion as iteration.
- Does our interpreter implement tail recursion as iteration?

Tail Recursive Implementation

We can test empirically this question using the same method as we did for testing whether Node.js implements tail recursion as iteration: run the following L5 program in our interpreter:

```
bind(parseL5(`
  (L5 (define factIter
    (lambda (n acc)
      (if (= n 0)
          acc
          (factIter (- n 1) (* n acc)))))
    (factIter 10000 1))
  `), evalProgram); // RangeError: Maximum call
                    // stack size exceeded
```

In other words, *L5* does not implement the Scheme requirement that the interpreter execute tail-recursive programs in an iterative manner.

In this chapter, we design a strategy that enables us to meet this requirement: implement an interpreter that executes tail-recursive programs in an iterative manner **even though** the meta-language (JavaScript in our case) does **not** provide this behavior.

We do not complete the code for this interpreter, but develop the strategy through the analysis of a general code transformation strategy called **CPS - Continuation Passing Style**.

An important usage of procedure expressions is delaying computation: when a procedure returns a closure, the computation embedded in the returned closure is not applied; instead it is delayed until the closure is applied.

For example:

```
(define make-delayed  
  (lambda (n)  
    (lambda ()  
      (fact n)))))
```

When invoked, this function constructs a delayed computation:

```
(let ((f4 (make-delayed 4)))  
  (do-something)  
  (f4))
```

`f4` is a procedure of no argument which encapsulates the computation `(fact 4)`. When it is created, a closure is constructed - but it is not invoked. We can then do other activities, and later, when the programmer decides, in the future, this delayed computation can be invoked in the form `(f4)`.

This closure plays a role similar to the tasks we reviewed in the previous lecture when discussing the Node event loop and asynchronous functions. The difference is that, in the Scheme case, we have not yet defined the framework in which we post tasks and which decides when to invoke these tasks.

This simple mechanism of using closures for delayed computations allows the Scheme programmer to design advanced control structures.

Let us consider two examples where this technique is applied:

- Write procedures which compute explicitly delayed computations with one or two continuations
- Model streams which are similar to the generator pattern reviewed in JavaScript.

Consider these two variants of the factorial procedure:

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

```
(define fact$
  (lambda (n cont)
    (if (= n 1)
        (cont 1)
        (fact$ (- n 1)
                (lambda (res) (cont (* n res)))))))
```

Observe the function `fact$` (we use the convention of naming procedures which follow this pattern with a dollar sign at the end). This procedure receives a parameter which we conventionally name `cont` which represents a delayed computation - that is, a procedure which expects a single parameter, and applies a delayed computation to this value.

The type of this parameter is simply a closure which receives as argument the type that the overall function returns - in the case of `fact$` it has type `[Number -> T]`.

The procedure `fact$` models the following computation:

- If the base case of the recursion is reached, apply the continuation **cont** to the base value (**cont 1**).

- Else, build a new continuation
`(lambda (res) (cont (* n res)))`
which represents the context of “what should be done after we compute the recursive case `(fact (- n 1))`” - then recursively invoke `(fact$ (- n 1))` with this new continuation.

Continuation Passing

Let us trace this function to understand how it models the control context explicitly:

```
(fact$ 3 (lambda (x) x))  
;; k3 = (lambda (x) x)  
  
(fact$ 2 (lambda (res) (k3 (* 3 res))))  
;; k2 = (lambda (res) (k3 (* 3 res)))  
  
(fact$ 1 (lambda (res) (k2 (* 2 res))))  
;; k1 = (lambda (res) (k2 (* 2 res)))  
  
(k1 1)  
(k2 (* 2 1))  
(k3 (* 3 2))  
((lambda (x) x) 6) ;; => 6
```

The execution of the function does not consume stack space - **fact\$** is a tail recursive iterative function. But observe that the parameter **cont** **grows** in each successive call:

```
k3 = (lambda (x) x)
k2 = (lambda (res) ((lambda (x) x) (* 3 res)))
k1 = (lambda (res)
      ((lambda (res)
         ((lambda (x) x) (* 3 res)))
        (* 2 res)))
```

In a schematic manner, we see that this version explicitly represents the control context in the form of a closure. We can think of this procedure as an iterative procedure which constructs its own control stack in its local parameter **cont** instead of relying on the interpreter's control stack.

How can we tell a program will yield an iterative process when it is evaluated in Scheme as opposed to a recursive one?

In other words, let us determine through **static analysis** of the syntactic structure of a program whether its evaluation will consume unbounded control space - stack space which is proportional to the arguments and thus can grow to a non-constant size.

This analysis can be formalized and automated, so that an expression can be proven to create iterative processes.

This analysis is based on the distinction of two syntactic positions within the AST of the language:

- Head position: sub-expressions of an expression which must be evaluated before the value of the overall expression is computed through primitive combination.

- Tail position: sub-expressions of an expression which are evaluated and whose value is not further combined with any other computation to return the value of the overall expression.

For example, consider the expression:

```
(if (> x 2) (* x 3) x)
```

```
(if (> x 2) (* x 3) x)
```

To compute this expression, according to the computation rule of **IfExp** AST nodes in the operational semantics of the language, we must first compute the test sub-expression (> x 2).

This sub-expression is in Head position.

Characterizing Iteration: Head and Tail Positions

```
(if (> x 2) (* x 3) x)
```

Then, based on the value which is obtained, we compute either `(* x 3)` or `x` - without combining their resulting value with any other computation. These sub-expressions are in Tail position of the `if` expression.

Tail positions are positions whose evaluations
is the last to occur.

Let us review the whole abstract syntax of our language and identify Head positions marked as **H** and tail positions marked as **T**. We only need to review compound expressions, since the definition of Head and Tail positions is only relevant for positions within a compound expression:

Characterizing Iteration: Head and Tail Positions

1. (**define** var H)
2. (**if** H T T)
3. (**lambda** (v1 ... vn) E ... E)
4. (**let** ((v1 H) ...) H ... T)
5. (H H ... H)

In a procedure expression (`lambda`), the sub-expressions are neither Head nor Tail position - since they are not computed when we compute the value of the `lambda` expression - and we note this with **E** position.

However, if we want to analyze a complete program to determine whether it has the potential to create a non-iterative process, we must also analyze the body of all the procedures as if they could be applied.

When we perform this analysis, we must also analyze the body of the procedures and attribute positions **H** for all the sub-expressions in the sequence except the last which is Tail - as in the body of a **let** expression.

Characterizing Iteration: Head and Tail Positions

Definition: Tail Form

An expression is in **tail form** if its head positions do not include applications with non-primitive operators, and its sub-expressions are all in tail form. Atomic expressions and combinations of atomic applications are in tail form.

Examples:

- `(+ 1 x)` is in tail form.
- `(* (* x x) (+ x x))` is in tail form
(combination of primitive applications).
- `(if p x (+ 1 (+ 1 x)))` is in tail form.
- `(f (+ x y))` is in tail form.

Characterizing Iteration: Head and Tail Positions

Examples:

- `(+ 1 (f x))` is not in tail form (but `(f x)` is in tail form) because after `(f x)` is computed, the result must be passed to further computation.
- `(if p x (f (- x 1)))` is in tail form.
- `(if (f x) x (f (- x 1)))` is not in tail form - because the head call `(f x)` must be followed by other calls.
- `(lambda (x) (f x))` is in tail form.

Examples:

- `(lambda (x) (+ 1 (f x)))` is not in tail form because the sub-expression `(+ 1 (f x))` is not in tail form.
- `(lambda (x) (g (f 5)))` is not in tail form.

Expressions in tail form create iterative processes when they are evaluated.

We will present an argument supporting this conclusion later.

First, let us present a systematic transformation which generates a tail-form expression for any given expression which is equivalent to the original expression in a specific sense.

Continuation Passing Style (CPS) is a programming technique which assumes that every user defined procedure f has a specific argument called its continuation, which is used to carry the control context of the computation.

Consider a regular function f - used in a specific context:

```
(+ (f a) (* a a))
```

The context in which $(f\ a)$ is invoked is:

$$(+\ \textcolor{green}{<r>\ } (*\ a\ a))$$

where $\textcolor{green}{<r>}$ represents the result of computing $(f\ a)$.

Continuation Passing Style (CPS)

In CPS, we encapsulate this context into a continuation procedure - which receives the result of computing `(f a)` and pass it to the CPS function `f$`:

```
(f$ a (lambda (res) (+ res (* a a))))
```

In general, the continuation parameter passed to CPS functions represents the future computation that needs to be applied once the computation of f ends.

We say that a procedure

$(f\$ x1 \dots xn \text{ cont})$

is CPS-equivalent to a function

$(f x1 \dots xn)$

if - for every parameters $(x1 \dots xn)$, and
every continuation procedure k :

$(f\$ x1 \dots xn k) = (k (f x1 \dots xn)).$

Note the type relations between a procedure f and its CPS-equivalent procedure $f\$$:

$$f: [T_1 * \dots * T_n \rightarrow T]$$
$$f\$: [T_1 * \dots * T_n * [T \rightarrow K] \rightarrow K]$$

In particular, using the identity procedure $\text{id} = (\text{lambda } (x) x)$, the equivalence $(f\$ x1 \dots xn \text{id}) = (f x1 \dots xn)$ holds when f and $f\$$ are CPS-equivalent for all possible arguments $(x1 \dots xn)$.

It is surprisingly possible to transform systematically any expression of our language *L5* to a CPS-equivalent expression which is in tail-form.

The method consists of identifying any head position in the expression which is not a primitive combination, to extract it as the first call of the CPS version, and to encapsulate the resulting context as the continuation of the CPS version.

The process is repeated until all head positions in the expression are left as primitive compositions and all the user-procedure calls are encapsulated into nested continuations.

Let us review examples of this transformation:

In the following cases, the original procedures are already in tail-form, the CPS transformation consists of simply adding a continuation parameter:

```
(define square (lambda (x) (* x x)))  
(define add1 (lambda (x) (+ x 1)))
```

```
(define square$ (lambda (x cont) (cont (* x x))))  
(define add1$ (lambda (x cont) (cont (+ x 1))))
```

Let us consider the next increment of complexity, with an expression which is still in tail-form - but with a user-defined procedure. In this case, we transform the user procedure with its CPS-equivalent and simply pass the continuation argument to the CPS version:

```
(define h  
  (lambda (x) (add1 (+ x 1))))
```

```
(define h$  
  (lambda (x cont) (add1$ (+ x 1) cont)))
```

The next case is a nested user procedure call:
the function is not in tail-form.

```
(define h1  
  (lambda (x) (square (add1 (+ x 1)))))
```

```
(define h1$  
  (lambda (x cont)  
    (add1$ (+ x 1) (lambda (add1-res)  
                     (square$ add1-res cont))))
```

In this case, we identify the first head position in the body of the procedure: this is the call `(add1 (+ x 1))`. This sub-expression is the first expression which is evaluated when the body of the procedure is evaluated.

It is evaluated in a non-tail position – because the result of this evaluation must be passed to the context: `(square <res>)`. The CPS transformation abstracts away this context into the continuation and moves the inner-most user-function application as the body of the procedure.

The context is:

```
cont1 = (lambda (add1-res) (square add1-res))
```

We read the resulting CPS `h1$` functions as:

- First compute the user procedure
`(add1$ (+ x 1) ...)`

- And pass the result of this procedure to the continuation

(**lambda** (**add1-res**) ...). Note that by convention we name the parameter of the continuation with the name **<name-of-user-procedure>-res** - in this case **add1-res** - to indicate that this parameter will be bound to the result of the **add1** application when the continuation will be invoked.

- Transform the context procedure into a CPS form - `(square add1-res)` becomes `(square$ add1-res cont)` - as was discussed in the previous example.

CPS Determines the Order of Evaluation of Arguments

When we specified the operational semantics of application expressions in applicative order, we indicated that the Scheme semantics specifically indicate that the order of evaluation of arguments within an application (**rator rand1 ... randn**) is not specified and can be determined by the implementation. That is, the interpreter is free to evaluate **rand2** before **rand1** or vice-versa.

When we transform an expression of this type to CPS, we *must* select a sub-expression (an operand) that will become the first head to be evaluated.

Consider the following example:

```
(define mult  
  (lambda (x y) (* x y)))
```

```
(define h2  
  (lambda (x y) (mult (square x) (add1 y))))
```

CPS Determines the Order of Evaluation of Arguments

;; In CPS:

```
(define mult$  
  (lambda (x y cont) (cont (* x y))))
```

```
(define h2-1$  
  (lambda (x y cont)  
    (square$ x  
      (lambda (square-res)  
        (add1$ y  
          (lambda (add1-res)  
            (mult$ square-res add1-res cont))))))))
```

CPS Determines the Order of Evaluation of Arguments

In this transformation, we identified two user-procedure applications - **square** and **add1** - in non-tail position (within the context of `(mult s a)`). We decide to move the first operand (**square x**) to the first evaluated expression in CPS - and move the rest of the calls to the continuation.

The order of evaluations in the **h2-1\$** version is:

1. First compute **(square x)**
2. Then pass the result **square-res** to the continuation which computes **(add1 y)**
3. Then pass the result **add1-res** to the continuation which computes **(mult s a)**

We can also generate the alternative version
where `(add1 y)` is first computed, and
`(square x)` second:

```
(define h2-2$  
  (lambda (x y cont)  
    (add1$ y  
      (lambda (add1-res)  
        (square$ x  
          (lambda (square-res)  
            (mult$ square-res add1-res cont)))))))
```

By convention, in the CPS transformation we will select arguments in left-to-right order - even if this is not the only possible transformation.

Let us consider the following example of a recursion which generates a tree-recursive process when evaluated, and transform it to a CPS-equivalent version:

```
(define sum-odd-squares
  (lambda (tree)
    (cond ((empty? tree) 0)
          ((not (list? tree))
           (if (odd? tree) (square tree) 0))
          (else (+ (sum-odd-squares (car tree))
                    (sum-odd-squares (cdr tree)))))))
```

We consider the branches of the **cond** expression one by one. In branches 1 and 2, both tests are primitive combinations, and the consequents are in tail-form. The **else** branch has no user-procedure call in head position. Hence the CPS-transformation does not change the structure of the **cond** expression. The first stage of the transformation is thus:

CPS of Tree Recursion Procedures

```
(define sum-odd-squares$  
  (lambda (tree cont)  
    (cond ((empty? tree) (cont 0))  
          ((not (list? tree))  
           (if (odd? tree) (square$ tree cont) (cont 0)))  
          (else <<CPS-transform (+ (sum-odd-squares (car tree))  
                                     (sum-odd-squares (cdr tree)))>>)))  
  ))
```

Let us now consider the consequent of the last branch. There are two user-procedure calls which are in non-tail position. We pick the first one, and turn it as the first call of the CPS.

CPS of Tree Recursion Procedures

```
(define sum-odd-squares$  
  (lambda (tree cont)  
    (cond ((empty? tree) (cont 0))  
          ((not (list? tree))  
           (if (odd? tree) (square$ tree cont) (cont 0)))  
          (else (sum-odd-squares$ (car tree)  
                                   (lambda (sum-odd-squares-car-res)  
                                     <<CPS-transform (+ sum-odd-squares-car-res  
                                                           (sum-odd-squares (cdr tree)))>>  
                                     ))))))
```

We have pushed the CPS transformation inside
- let us complete it:

CPS of Tree Recursion Procedures

```
(define sum-odd-squares$  
  (lambda (tree cont)  
    (cond ((empty? tree) (cont 0))  
          ((not (list? tree))  
           (if (odd? tree) (square$ tree cont) (cont 0)))  
          (else (sum-odd-squares$ (car tree)  
                                   (lambda (sum-odd-squares-car-res)  
                                     (sum-odd-squares$ (cdr tree)  
                                                         (lambda (sum-odd-square-cdr-res)  
                                                           (cont (+ sum-odd-square-car-res  
                                                                sum-odd-square-cdr-res))))))))))
```

Let us consider the transformation of function which receives a function as a parameter - for example the `map` operator:

```
(define map
  (lambda (f lst)
    (if (empty? lst)
        lst
        (cons (f (car lst))
                (map f (cdr lst))))))
```

This procedure includes two user-procedure calls, nested within a **cons** application - hence not in tail position. Therefore, the process is not iterative. The two nested user procedure calls appear in the arguments of the **cons** application.

Note that the test of the **if** expression and the then-part are all primitive combinations, hence, no CPS transformation is required besides wrapping the tail value in the **cont** call.

We select the first argument of cons as the first nested call and move it as the first step of the CPS-transformed version:

CPS Transformation of Higher-Order Procedures

```
(define map$  
  (lambda (f$ lst cont)  
    (if (empty? lst)  
        (cont lst)  
        (f$ (car lst)  
             (lambda (f-res)  
               <<CPS-transform (cons f-res  
                                     (map f$ (cdr lst)))>>))))))
```

Note that in this transformation, we consider that the **f** parameter we receive is a user procedure in CPS format as well. This is necessary, because CPS as a discipline is an all-or-nothing proposition: all user-defined procedures must be transformed in CPS to be able to obtain a coherent CPS program.

We next observe the remaining segment to be transformed in CPS:

`(cons f-res (map f (cdr lst)))`. The first sub-expression to be evaluated in this context is `(map f (cdr lst))` which we push outside as the first step of the CPS transformation - and we eventually obtain:

CPS Transformation of Higher-Order Procedures

```
(define map$  
  (lambda (f$ lst cont)  
    (if (empty? lst)  
        (cont lst)  
        (f$ (car lst)  
             (lambda (f-car-res)  
               (map$ f$ (cdr lst)  
                    (lambda (map-f-cdr-res)  
                      (cont (cons f-car-res map-f-cdr-res))))))))))
```

Consider the CPS transformation of the `filter` procedure:

```
(define filter
  (lambda (pred? lst)
    (cond ((empty? lst) lst)
          ((pred? (car lst))
           (cons (car lst) (filter pred? (cdr lst))))
          (else (filter pred? (cdr lst)))))
```

The new configuration we meet is that the test part of the second branch is a user-procedure application (`pred? (car lst)`). In this case, we must change the structure of the `cond` so that we first compute the test value outside the `cond`, then route the conditional inside the continuation:

CPS Transformation of Conditionals with Non-primitive Tests

```
(define filter$  
  (lambda (pred?$ lst cont)  
    (cond ((empty? lst) (cont lst))  
          (else (pred?$ (car lst)  
                        (lambda (pred-res)  
                          <<CPS-transformation  
                            (cond (pred-res (cons (car lst)  
                                                  (filter pred?$ (cdr lst))))  
                                (else (filter pred?$ (cdr lst))))>>))))))
```

The transformation here is that the **cond** structure is split - in branches which can be evaluated all in tail-form and those which follow the test evaluation.

The next step of the CPS-transform is as observed in previous examples:

CPS Transformation of Conditionals with Non-primitive Tests

```
(define filter$  
  (lambda (pred?$ lst cont)  
    (cond ((empty? lst) (cont lst))  
          (else (pred?$ (car lst)  
                        (lambda (pred-res)  
                          (cond (pred-res (filter$ pred?$ (cdr lst)  
                                                    (lambda (filter-cdr-res)  
                                                      (cont (cons (car lst)  
                                                                    filter-cdr-res))))  
                                (else (filter$ pred?$ (cdr lst) cont))))))))))
```

Summary: CPS Transformation

We have learned how to transform any expression in the language into a tail-form CPS-equivalent expression. When evaluating the CPS-equivalent version of the expression, our interpreter yields a tail-recursive iterative process which does not consume control space (the stack does not grow in an unbounded way dependent on the parameters fed to the program).

Instead, the explicit continuation parameter which is added to all procedure calls records the control context of the procedure applications. This parameter does grow according to the size of the parameters.

Note how the CPS transformation is similar to the transformation from synchronous function calls to asynchronous function calls which we discussed in the previous lecture:

Application expressions appear in the CPS version in the same order in which they are evaluated; composed functions appear nested inside the body of the continuation in the same way as they appeared nested in the body of the callbacks of the asynchronous functions.

Summary: CPS Transformation

The difference between the event-driven model we discussed in the Node.js environment and the CPS model discussed in Scheme is that in the asynchronous model, the continuations (the tasks generated by asynchronous functions) are posted on the task-queue and executed by the interpreter according to the event-model while in the CPS model in Scheme, continuations are immediately invoked when the preceding call completes.

A variant of the CPS model just introduced allows us to handle error conditions in a more flexible manner than was possible in the style we have discussed so far.

Success-fail continuations remind us of the structure of callbacks in Node.js and of promises - which expect one continuation for successful calls - passed in the `.then(success)` method, and one for the failure cases, passed in the `.catch(fail)` method.

Consider a procedure which receives a heterogeneous list and computes the sum of the numeric items in the list. If any item in the list is not a number - we want to trigger an error and stop the computation (that is, we stop the traversal of the list as soon as a non-number item is detected).

Let us first write a traditional implementation of this procedure, using the **error** primitive, which is the equivalent of throwing an exception:

Success-Fail Continuations

```
;; Signature: sumlist(li)
;; Purpose: Sum the elements of a number list.
;;          If the list includes a non number
;;          element -- produce an error.
;; Type: [List -> Number union ???]
(define sumlist
  (lambda (li)
    (cond ((empty? li) 0)
          ((not (number? (car li)))
           (error "non numeric value!"))
          (else (+ (car li) (sumlist (cdr li)))))))
```

Consider an iterative CPS version of this procedure, which uses success/fail continuations:

Success-Fail Continuations

```
(define sumlist2
  (lambda (li)
    (letrec ((sumlist$
              (lambda (li succ-cont fail-cont)
                (cond ((empty? li) (succ-cont 0))
                      ((number? (car li))
                       (sumlist$ (cdr li)
                                  (lambda (sum-cdr) ;; success cont.
                                    (succ-cont (+ (car li) sum-cdr)))
                                  fail-cont))
                      ;; error condition: invoke error handler
                      (else (fail-cont))))))
      (sumlist$ li
                (lambda (x) x)
                (lambda () (display "non numeric value!"))))))
```

Observe first that the presence of errors is problematic for our type system. We do not know how to indicate, in the type of the procedure, the fact that it potentially throws an exception. For comparison, think of how this condition is handled in Java.

In the CPS style, we handle errors by using continuations - instead of using one continuation as we did in the transformation discussed above, we carry 2 continuations: one for the successful computation path, and one for error conditions.

Observe the type of the `sumlist$` procedure above:

```
[List * [Number -> Number]  ;; success  
  * [Empty -> Void]         ;; fail  
  -> (Number | Void)]
```

Let us consider a procedure which traverses a data structure (for example an AST), searching for a node that meets a given criterion. If such a node is met, we perform a success computation; else we perform a fail computation.

In this case, we will use the **success** continuation to indicate that the search has completed, and that we continue the computation by processing the element found.

The **fail** continuation will be used to drive the search for other elements within the data structure, until, possibly, the data structure has been exhausted, in which case, the original **fail** continuation is applied, indicating a failure to find an element meeting the criterion.

The tree in this example is an unlabeled tree, that is, a tree whose branches are unlabeled, and whose leaves are labeled by atomic values. This is the standard view of an heterogeneous list. For example, $((3\ 4)\ 5\ 1)$ is a list representation of such a tree.

Tree interface:

- Constructors: `make-tree(t1,t2, ...)`,
`add-subtree(first,rest)`,
`make-leaf(data)`, `empty-tree`

Tree interface:

- Accessors: `first-subtree(tree)`,
`rest-subtree(tree)`,
`leaf-data(leaf)`

Tree interface:

- Predicates: `composite-tree?(t)`,
`leaf?(t)`, `empty-tree?(t)`

Using Success-Fail Continuations for Search

In our simple implementation, we use:

```
(define make-tree list)
(define add-subtree cons)
(define make-leaf (lambda (x) x))
(define empty-tree? empty?)
(define first-subtree car)
(define rest-subtree cdr)
(define leaf-data (lambda (x) x))
(define composite-tree? list?)
(define leaf? (lambda (x) (not (composite-tree? x))))
```

The CPS procedure uses a success and a fail continuations.

The important point in this procedure, is that when an interior node is processed, the procedure makes a non-deterministic decision: it starts the traversal of the first sub-tree, which may or may not contain the searched item.

If this first traversal succeeds, the search procedure completes; else we need to backtrack and search the other sub-trees. We need to keep track of this decision point explicitly - so that this further search can be performed in the future if needed.

This is accomplished by constructing a fail-continuation which remembers what are the next sub-trees to be traversed. The fail continuation is applied when the search reaches a leaf and fails.

Using Success-Fail Continuations for Search

```
;; Signature: leftmost-even(tree)
;; Purpose: Find the leftmost even leaf of an
;;          unlabeled tree whose leaves are
;;          labeled by numbers.
;;          If no leaf is even, return #f.
;; Type: [List -> Number union Boolean]
;; Examples: (leftmost-even '((1 2) (3 4 5))) ==> 2
;;          (leftmost-even '((1 1) (3 3) 5)) ==> #f
(define leftmost-even
  (lambda (tree)
    (leftmost-even$ tree
      (lambda (x) x)
      (lambda () #f))))
```

Using Success-Fail Continuations for Search

```
(define leftmost-even$  
  (lambda (tree succ-cont fail-cont)  
    (cond ((empty-tree? tree) (fail-cont))  
          ((leaf? tree)  
           (if (even? (leaf-data tree))  
               (succ-cont tree)  
               (fail-cont))))  
    (else ; Composite tree  
      (leftmost-even$ (first-subtree tree)  
                       succ-cont  
                       (lambda ()  
                         (leftmost-even$ (rest-subtrees tree) ;*  
                                           succ-cont  
                                           fail-cont)))))))
```

The `leftmost-even` procedure performs an exhaustive search on the tree, until an even leaf is found. Whenever the search in the first sub-tree fails, it invokes a recursive search on the rest of the sub-trees.

This kind of search is a backtracking search policy: If the decision to search in the first sub-tree appears wrong, a retreat to the decision point occurs, and an alternative route is selected.

Note that the fail continuation that is passed to the fail continuation that is constructed in the decision point (marked by `*`) is the fail continuation that is passed to `leftmost-even$` as an argument.

Using Success-Fail Continuations for Search

To understand that think about the decision points:

- If the search in (`first-subtree tree`) succeeds, then the future is `succ-cont`.
- If it fails, then the future is the search in (`rest-subtrees tree`).
- If the search in (`rest-subtrees tree`) succeeds, the future is `succ-cont`.
- If it fails, then the future is `fail-cont`.

Finally, a non-CPS version:

Using Success-Fail Continuations for Search

```
(define leftmost-even
  (lambda (tree)
    (letrec ((iter
              (lambda (tree)
                (cond ((empty-tree? tree) #f)
                      ((leaf? tree)
                       (if (even? (leaf-data tree))
                           (leaf-data tree)
                           #f))
                      (else
                       (let ((res-first (iter (first-subtree tree))))
                         (if res-first
                             res-first
                             (iter (rest-subtrees tree)))))))
              (iter tree))))
```