Programming Languages
Chapter 4 of SICP
Metalinguistic Abstraction
Supplemental Material

Atsushi Igarashi
Dept. of Communications and Computer Engineering
Graduate School of Informatics, Kyoto University
e-mail: igarashi@kuis.kyoto-u.ac.jp

In these notes, we discuss the notion of *continuations*, which will be important (or even necessary) to understand Section 4.3 of the textbook, through implementing an exception handling mechanism on top of the meta-circular interpreter.

4.2$\frac{1}{2}$.1 Exceptions (or run-time errors) and Exception Handling

An *exception* stands for an abnormal situation, in which computation has to abort for some reason, such as referencing to an undefined variable, application of a non-functional value, or division by zero. The following expressions include such exceptions. We often say "an exception is raised (or thrown)" when such an abnormal situation occurs.

```
foo
(5 + 3)
(/ 2 0)
```

In many Scheme implementations (including `jakld`), there is built-in procedure `error`, which just aborts program execution and come back to the prompt (without executing the rest of the program).

```
(+ 3
  (begin (error "My error!") (display "This will be ignored.") 4))
```

It may not be, however, desirable, to abort the whole program execution once an exception is raised. *Exception handling* is a mechanism to detect exceptions, run some error-recovery code, and resume execution of the rest of the program. Some Scheme implementations provide exception handling but we introduce so-called "`catch`-and-`throw`" mechanism, provided by many implementations of Lisp, an ancestor of Scheme, and implement it on top of our meta-circular interpreter.

An exception handling mechanism consists of:

- an mechanism to detect raised exceptions, and
- an mechanism to resume the program execution

and the former is realized by special form `throw` and the latter by `catch`. Let's take a look at how `throw` works. It takes two arguments as follows.

```
(throw 'a (+ 2 3))
```

The first argument is a tag to represent the kind of an exception raised. It can be anything but we mostly use symbols. The second argument is a datum, which often represents detailed information on the exception.

If `throw` is called (without `catch`), its behavior is almost the same as `error`—that is, the program execution aborts and goes back to the prompt without doing the rest of the execution.

```
(define (my-list-ref l n)
  (cond ((< n 0) (throw 'negative-index n))
        ((zero? n) (car l))
        (else (my-list-ref (cdr l) (- n 1)))))

(my-list-ref '(1 2 3) 1)
⟹ 2
(my-list-ref '(1 2 3) -5)
⟹ ;; uncaught exception
```

`catch` can be used to resume the execution, aborted by `throw`. The next expression shows a basic usage of `catch`.

```
(catch 'negative-index
  (my-list-ref '(1 2 3) 1))
⟹ 2
```

`Catch` first starts evaluation of `(my-list-ref ...)`. If it results in a value (with no exceptions thrown), then the value becomes the value of the whole `catch` expression. In this example, the value of `(my-list-ref ...)` is `2`, so the value of the whole expression is also `2`. Otherwise, if an exception is thrown and the tag of the thrown exception is equal to the first argument to `catch` (in this case the symbol `negative-index`), then the value attached to the exception will be the value of the whole `catch` expression.

```
(+ (catch 'negative-index
     (my-list-ref '(1 2 3) -5))
   1)
⟹ -4
```

`(my-list-ref ...)` in the expression above doesn't return a value but rather throw an exception with tag `'negative-index` and value `-5`. Since the tag in the thrown exception and the tag in the `catch` expression is equal, the value of `catch` is `-5`; and the value of the whole expression is `-4`.

If the tags are not equal, the thrown exception is not caught here and an outer `catch` may catch it. If there is no outer `catch`, the execution aborts and goes back to the prompt with the message "uncaught exception".

```
(catch 'another-tag
  (my-list-ref '(1 2 3) -5))
⟹ ;; uncaught exception
```

Here is a summary of the special form `catch` and its behavior.

`(catch` *tag* exp_1 `...` exp_n`)`

1. Evaluate *tag* and obtain a tag. Exceptions thrown during the evaluation of this expression will *not* be caught by this `catch` (it's an outer `catch`'s job to catch them).

2. Then, evaluate the body exp_i sequentially. If the evaluation terminates normally, the value of the last expression exp_n is the value of the whole expression. If `throw` is executed, the tag of the thrown exception and the tag obtained in the previous step are compared by `eq?`. If they are equal, then the value in the thrown exception is that of the whole expression; otherwise, the same exception is thrown again (and an outer `catch` may catch it).

Exercise 1 Define procedure `prod-list`, which takes an integer list and returns the product of the integers in it.

If a list contains `0`, then the return value of `prod-list` will be `0`. Use `catch` and `throw` so that the procedure returns *immediately*—without performing further multiplication—when `0` is found in the list.

Exercise 2 Define procedure `pos-in-list`, which takes a list and an element in it and returns where the element occurs in the list. (If the element occurs as n-th element, then it should return $n - 1$.) Moreover, if the given element does *not* occurs in the list, the procedure should return `-1`. (Use `eq?` for comparison.)

Exercise 3 The following procedure `change` breaks given money into small money.

```
(define (change coins amount)
  (if (zero? amount) '()
      (let ((c (car coins)))
        (if (> c amount) (change (cdr coins) amount)
          (cons c (change coins (- amount c)))))))
```

It takes a list of available coins (sorted in the descending order) and the amount of money and returns a list of coins, whose total is the same as the input.

```
(define us-coins '(25 10 5 1))      ;; American coins
(define gb-coins '(50 20 10 5 2 1)) ;; British coins
(change gb_coins 43)
(change us_coins 43)
```

However, since this definition tries to use the first coin as much as possible, it may fail even when there is a possible combination of coins:

```
(change '(5 2) 16)
```

Let's use exception handling to solve this problem. Complete the following definition by filling "..." so that the procedure returns a combination of coins whenever there is one.

```
(define (change coins amount)
  (cond ((zero? amount) '())
        ((null? coins) ...)
```

```
(else
  (let ((c (car coins)))
    (if (> c amount) (change (cdr coins) amount)
        (or (catch 'fail
              (cons c (change coins (- amount c))))
            ...)))))
```

$4.2\frac{1}{2}.2$ Continuations

Continuation stands for the rest of computation at a given point in time in a (long) computation process. One may call it "To-Do" list. This "To-Do" list shrinks when some work is finished and grows when a big task is divided into smaller ones.

For example, consider the process of evaluating `(+ (* ` e_1 e_2`)` e_3`)` . The continuation after finding out the expression is a function application is:

1. Evaluate `(*` e_1 e_2`)` (and let's call its value v_1);
2. Evaluate e_3 (and let's call its value v_2); and
3. Compute the sum of v_1 and v_2 (by `apply`).

Now we start evaluating `(*` e_1 e_2`)` and will soon find out the expression is again a function application. At this point, the continuation will be

1. Evaluate e_1 (and let's call its value v_{11});
2. Evaluate e_2 (and let's call its value v_{12});
3. Compute the product of v_{11} and v_{12} by `apply` (and let's call its value v_1);
4. Evaluate e_3 (and let's call its value v_2);
5. Compute the sum of v_1 and v_2 by `apply`.

by breaking the first step into smaller steps. Then, suppose we finish the (new) first step, then the first item is removed and the continuation becomes a slightly shorter list:

1. Evaluate e_2 (and let's call its value v_{12});
2. Compute the product of v_{11} and v_{12} by `apply` (and let's call its value v_1);
3. Evaluate e_3 (and let's call its value v_2);
4. Compute the sum of v_1 and v_2 by `apply`.

In this way, continuation changes over time.

Note that an outer operation—the application of `+` in this exapmle—appears as a later entry in the "To-Do" list.

Exceptions and Continuations. Now consider the behavior of `error`, which is to abort program execution and go back to the prompt, in terms of continuations. The behavior of `error` can be rephrased as "discarding all the continuation (all the entries in the "To-Do" list)".

For example, consider `(+ (error` e`) (* 3 4))` . The continuation after the evaluation of e is

1. Apply `error` to the value of e (and let's call its value v_1);
2. Evaluate `(* 3 4)` (and let's call its value v_2); and
3. Apply `+` to v_1 and v_2 (to compute their sum).

However, `error` immediately discards the entries to terminate the whole computation (abnormally).

The behavior of `catch` and `throw` can be understood as manipulation of a continuation. First, `catch` leaves a "mark" to denote *the end of the body of `catch`* in the continuation before entering the evaluation of its body. The mark includes the tag value specified in `catch`. If the evaluator encounters this mark after a normal evaluation process, it will just remove the mark from the continuation and continue to the next item—so, for normal evaluation, the mark means nothing. This mark plays a significant role when `throw` is called during the evaluation of the body of `catch`. Namely, `throw` discards a continuation just as `error`, but it discards only a part of the continuation: all the items *until the first* mark in the "To-Do" list. Then, the tag recorded in the mark and the tag in the thrown exception are compared. If they are equal, resume the program execution from there; otherwise, the items until the next mark will be discarded, the tags are compared, and so on.

In order to implement exception handling, the evaluator has to manipulate continuations as data. Such an evaluator, in which continuations are data and can be manipulated by the evaluator, is called *continuation-passing evaluator*.

In what follows, we describe a metacircular continuation-passing interpreter for Scheme without exception handling, and then describe an implementation of `catch` and `throw`.

4.2$\frac{1}{2}$.3 Continuation-Passing Evaluator

4.2$\frac{1}{2}$.3.1 Structure of Continuation-Passing Evaluator

The main procedures in the evaluator in Section 4.1 (of the textbook) are

- `eval`, which conducts case analysis on the form of a given expression and dispatch an appropriate procedure; and
- `apply`, which implements function application.

They call each other to evaluate a given expression.

In a continuation-passing evaluator, `eval` takes as arguments not only an expression and an environment but also a continuation, which represents the rest of computation after the evaluation of the given expression. Also, there is another procedure `apply-cont`, which takes a continuation and a value as arguments and performs "the rest of computation" that the continuation represents. `Eval` and `apply-cont` call each other in such a way that:

- `eval` (or auxiliary procedures named `eval-`*`XXX`*) call `apply-cont` to perform the rest of computation when they obtain a value of the given expression.

```
(define (eval exp env cont)
  (cond
   ((self-evaluating? exp) (apply-cont cont exp))
   ((variable? exp) (apply-cont cont (lookup-variable-value exp env)))
   ((quoted? exp) (apply-cont cont (text-of-quotation exp)))
   ((assignment? exp) (eval-assignment exp env cont))
   ((definition? exp) (eval-definition exp env cont))
   ((if? exp) (eval-if exp env cont))
   ((lambda? exp)
    (apply-cont cont
     (make-procedure (lambda-parameters exp)
                     (lambda-body exp)
                     env)))
   ((begin? exp)
    (eval-sequence (begin-actions exp) env cont))
   ((cond? exp) (eval (cond->if exp) env cont))
   ((application? exp)
    (eval (operator exp) env
          (make-operandsc (operands exp) env cont)))
   (else
    (error "Unknown expression type -- EVAL" exp))))
```

Figure 1: Continuation-Passing Evaluator (1)

- `apply-cont` inspects items in the "To-Do" list (namely, continuation) and may call `eval` if necessary.

(As we will see later, the *three* procedures `eval`, `apply`, and `apply-cont` call each other.)

Definition of `eval` Figure 1 shows the definition of `eval`. As discussed above, the third argument `cont` to represent a continuation is added. When a value can be obtained without calling `eval` recursively—namely, in the case of constants, variables, quotations, and lambdas—the evaluator calls `apply-cont` with the obtained value to perform the rest of computation.

Next, let's take a look at how auxiliary procedures, which involved recursive calls to `eval`, work. Figure 2 shows auxiliary procedures such as `eval-if` (the original definition in Section 4.1 is also shown as a comment). What `eval-if` does is

1. to evaluate the condition part (`if-predicate`);

2. to test whether the obtained value satisfies `true?`, evaluate the `if-consequent` or `if-alternative` part (according to the result of the test), and returns its value as the value of the whole `if`.

A main trick of a continuation-passing evaluator is

to add all the tasks except the first to the continuation and recursively call `eval` with the new continuation.

```
;; (define (eval-if exp env)
;;  (if (true? (eval (if-predicate exp) env))
;;      (eval (if-consequent exp) env)
;;      (eval (if-alternative exp) env)))

(define (eval-if exp env cont)
  (eval (if-predicate exp) env
        (make-testc (if-consequent exp) (if-alternative exp) env cont)))

(define (eval-sequence exps env cont)
  (cond ((last-exp? exps) (eval (first-exp exps) env cont))
        (else (eval (first-exp exps) env
                    (make-beginc (rest-exps exps) env cont)))))

(define (eval-assignment exp env cont)
  (eval (assignment-value exp) env
        (make-assignc (assignment-variable exp) env cont)))

(define (eval-definition exp env cont)
  (eval (definition-value exp) env
        (make-definec (definition-variable exp) env cont)))
```

Figure 2: Continuation-Passing Evaluator (2)

In the new definition of `eval-if`, `make-testc` creates the new continuation and `eval` is called recursively to evaluate the condition part with the new continuation. Once the value of the condition part is obtained, the rest of computation (test and evaluation of one of the two branches) will be performed (by `apply-cont`).

In what follows, `make-`*`XXX`*`c` (`c` stands for "c" in "continuation") is used for names of procedures to various kinds of continuations. A function to create a continuation takes information necessary to perform the rest of computation—in the case of `make-testc`, expressions from `if-consequent` and `if-alternative` parts, and an environment under which one of them is evaluated. Of course, we can't forget the continuation of `if`, so such a continuation will be part of an input to `make-testc`. (Since a continuation is a "To-Do" list, `make-`*`XXX`*`c` works as `cons`, which adds a new element to the head of a list.)

Similarly to `eval-if`, `eval-sequence` just calls `eval` recursively after adding the rest of computation to the given continuation. Also, in the `application?` case in `eval`, the evaluator adds an item that represents "evaluation of arguments" to the continuation by `make-operandc` and calls itself recursively with the new continuation.

Here is a summary of continuation-creating procedures and what they add to the continuation:

`make-testc`	to test a condition and evaluate an appropriate expression
`make-beginc`	to evaluate the following expressions in `begin`
`make-assignc`	to assign a value to the given variable and return `'ok`
`make-definec`	to define a variable and returns `'ok`
`make-operandsc`	to evaluate arguments and call `apply` to perform function application

Definition of `apply-cont` Now, let's take a look at the definition of `apply-cont` (Figure 3). It takes a continuation and a value, which was obtained by the preceding computation; it conducts case analysis on the form of (the head item in) the continuation, and performs computation corresponding to the continuation.

The first five cases are for the kinds of continuations we have seen so far. (Procedure *`XXX`*`c?` returns true if the continuation is created by procedure `make-`*`XXX`*`c`.) For example, in the case of `testc?`, the procedure tests whether the given value is true or not and evaluates one of the branches accordingly. Notice that the second half of the original `eval-if` is performed here!

`testc-true`, `testc-false`, `testc-env`, and `testc-cont` are selector procedures to extract the expression to be evaluated when the test is true, the expression to be evaluated when the test is false, the environment under which one of these expressions is evaluated, and the continuation of `if`, respectively.

Similarly, the second halves of the original `eval-`*`XXX`* are moved into `apply-cont`. Note that when an evaluator returns `'ok` as in `define` and `set!`, `apply-cont` is called recursively just as the constant case in (new) `eval`.

In the case for `operandsc?`, the task represented by the continuation consists of the following two steps:

1. to evaluate the argument list sequentially and create a list of values; and
2. call `apply` with the argument list and `val`, which is (supposed to be) a function.

So, `apply-cont` splits the continuation, creates a new continuation that includes only the second step by `make-applyc`, and pass it to `list-of-values`, which performs the first step above (Figure 4). The case for the continuation created by `make-applyc` is vely simple: it just calls `apply`.

The task that `list-of-values` has to perform is split into the following three steps:

1. evaluation of the first argument;
2. evaluation of the rest of the arguments; and
3. `cons`ing these results.

`make-restopsc` is used to create a continuation that represents the second and third steps. This continutation is handled in

```
(define (apply-cont cont val)
  (cond ...
        ((restopsc? cont)
         (list-of-values (restopsc-rest cont) (restopsc-env cont)
                         (make-consc val (restopsc-cont cont))))
        ...
))
```

Here, the evaluation of the rest of the arguments is performed by `list-of-values` and a continuation for the third step is created by `make-consc`. The continuation created here has to record the value (`val`) obtained by the first step in the continuation in order to `cons` it later.

The process for this kind of continuation is to `cons val`, which stands for the list of the argument values obtained by the second step, and `(consc-val cont)`, which stands for the value of the first argument stored in the continuation, and then perform the rest of computation (namely `(consc-cont cont)`).

```
(define (apply-cont cont val)
  (cond ...
        ((consc? cont)
         (apply-cont (consc-cont cont)
                     (cons (consc-val cont) val)))
        ...
))
```

Definition of `apply`: Actually, the definition of `apply` is mostly the same as before (Figure 5). Main changes are:

- it takes a continuation of the application as an additional argument;
- it passes the continuation to `apply-primitive-procedure`, which applies a primitve, or `eval-sequence`, which evaluates the function body.

The result of a primitive procedure application is passed to `apply-cont`.

Implementation of REPL REPL[1] creates a continuation to denote the end of evaluation by `make-haltc` to pass it to `eval`.

This continuation just returns the given value as it is. The following code snippet shows the corresponding part of `apply-cont`.

```
(define (apply-cont cont val)
  (cond ...
        ((haltc? cont) val)
))
```

Representing Continuations: Figures 6 and 7 show procedures to manipulate continuations. Here is a summary of continuations created by `make-`*XXX*`c`:

XXX	functionality
`testc`	to test a condition and evaluate an appropriate exrpession
`assignc`	to assign a value to the given variable and return `'ok`
`definec`	to define a variable and returns `'ok`
`beginc`	to evaluate the following expressions in `begin`
`operandsc`	to evaluate arguments and call `apply` to perform function application
`applyc`	to apply a function (the last part of the continuation represented by `operandsc`)
`restopsc`	to evaluate the rest of the arguments and `cons` with the value of the first argument
`consc`	`cons` the first argument and the list of the values of the rest of the arguments
`haltc`	the end of evaluation; the given value is to be displayed

4.2$\frac{1}{2}$.4 Implementing `catch` and `throw`

Actually, it is no more difficult to implement `catch` and `throw` than a continuation-passing evaluator. Firstly, the evaluation of `catch` consists of the two steps:

1. evaluation of the tag part; and

[1] read-eval-print-loop: See the handout for Chapter 4.

```
(define (apply-cont cont val)
  (cond ((testc? cont)
         (if (true? val)
             (eval (testc-true cont) (testc-env cont) (testc-cont cont))
             (eval (testc-false cont) (testc-env cont) (testc-cont cont))))
        ((assignc? cont)
         (set-variable-value! (assignc-var cont) val (assignc-env cont))
         (apply-cont (assignc-cont cont) 'ok))
        ((definec? cont)
         (define-variable! (definec-var cont) val (definec-env cont))
         (apply-cont (definec-cont cont) 'ok))
        ((beginc? cont)
         (eval-sequence
          (beginc-rest-exps cont)
          (beginc-env cont)
          (beginc-cont cont)))
        ((operandsc? cont)
         (list-of-values (operandsc-exps cont)
                         (operandsc-env cont)
                         (make-applyc val (operandsc-cont cont))))
        ((applyc? cont)
         (my-apply (applyc-proc cont) val (applyc-cont cont)))
        ((restopsc? cont)
         (list-of-values (restopsc-rest cont) (restopsc-env cont)
                         (make-consc val (restopsc-cont cont))))
        ((consc? cont)
         (apply-cont (consc-cont cont)
                     (cons (consc-val cont) val)))
        ((haltc? cont) val)
        (else (error "Unknown continuation type -- APPLY-CONT" cont))
))
```

Figure 3: Continuation-Passing Evaluator (3)

```
(define (list-of-values exps env cont)
  (if (no-operands? exps)
      (apply-cont cont '())
      (eval (first-operand exps) env
            (make-restopsc (rest-operands exps) env cont))))
```

Figure 4: Continuation-Passing Evaluator (4)

```
(define (my-apply procedure arguments cont)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments cont))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))
           cont))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))

(define (apply-primitive-procedure proc args cont)
  (apply-cont cont (apply (primitive-implementation proc) args)))
```

Figure 5: Continuation-Passing Evaluator (5)

2. marking the continuation and evaluation of the body.

So, we need a continuation to represent the second half. Similarly, the evaluation of `throw` consists of the three steps:

1. evaluation of the tag part; and

2. evaluation of the expression to be thrown; and

3. throwing an exception (or, search for the mark by `catch`).

So, we need a continuation to represent the second and third steps. The procedures to create these continuations are called `make-cabodyc` and `make-thbodyc`. The new definition of `eval` appears in Figure 8. (The figure also shows procedures to manipulate `catch` and `throw` expressions.)

Then, Figure 9 shows `apply-cont`.

Since the body of `catch` is a sequence of expressions, we use `eval-sequence` to evaluate the body. The continuation is marked here by `make-catchc`. The mark comes with the value of the lastly evaluated expressions (that is, the tag value form `catch`). The clause with `catchc?` is executed only when the evaluation of the body of `catch` terminates normally. In this case, the value is passed directly to the outer continuation (`(catchc-cont cont)`).

The clause with `thbodyc?` evaluates the second argument to `throw`. The new continuation created by `make-throwc` represents throwing an exception and the corresponding behavior is described in the last clause in `apply-cont`. There, an auxiliary procedure `first-matching-catch` is used to find the mark of `catch` that can catch the thrown exception and then the marked continuation is applied to the thrown value. If there is no such mark in the continuation, `first-matching-catch` will return `false`; in this case, `apply-cont` returns the symbol `uncaught` together with the thrown

```
(define (make-testc true-exp false-exp env cont)
  (list 'testc true-exp false-exp env cont))
(define (testc? cont) (tagged-list? cont 'testc))
(define (testc-true cont) (cadr cont))
(define (testc-false cont) (caddr cont))
(define (testc-env cont) (cadddr cont))
(define (testc-cont cont) (car (cddddr cont)))

(define (make-beginc exps env cont)
  (list 'beginc exps env cont))
(define (beginc? cont) (tagged-list? cont 'beginc))
(define (beginc-rest-exps cont) (cadr cont))
(define (beginc-env cont) (caddr cont))
(define (beginc-cont cont) (cadddr cont))

(define (make-assignc var env cont) (list 'assignc var env cont))
(define (assignc? cont) (tagged-list? cont 'assignc))
(define (assignc-var cont) (cadr cont))
(define (assignc-env cont) (caddr cont))
(define (assignc-cont cont) (cadddr cont))

(define (make-definec var env cont) (list 'definec var env cont))
(define (definec? cont) (tagged-list? cont 'definec))
(define (definec-var cont) (cadr cont))
(define (definec-env cont) (caddr cont))
(define (definec-cont cont) (cadddr cont))

(define (make-operandsc exps env cont)
  (list 'operandsc exps env cont))
(define (operandsc? cont) (tagged-list? cont 'operandsc))
(define (operandsc-exps cont) (cadr cont))
(define (operandsc-env cont) (caddr cont))
(define (operandsc-cont cont) (cadddr cont))
```

Figure 6: Continuation-Passing Evaluator (6)—Representation of Continuations (1)

```
(define (make-applyc proc cont)
  (list 'applyc proc cont))
(define (applyc? cont) (tagged-list? cont 'applyc))
(define (applyc-proc cont) (cadr cont))
(define (applyc-cont cont) (caddr cont))

(define (make-restopsc exps env cont)
  (list 'restopsc exps env cont))
(define (restopsc? cont) (tagged-list? cont 'restopsc))
(define (restopsc-rest cont) (cadr cont))
(define (restopsc-env cont) (caddr cont))
(define (restopsc-cont cont) (cadddr cont))

(define (make-consc val cont)
  (list 'consc val cont))
(define (consc? cont) (tagged-list? cont 'consc))
(define (consc-val cont) (cadr cont))
(define (consc-cont cont) (caddr cont))

(define (make-haltc) 'haltc)
(define (haltc? cont) (eq? cont 'haltc))
```

Figure 7: Continuation-Passing Evaluator (6)—Representation of Continuations(2)

```
(define (eval exp env cont)
  (cond
   ...
   ((catch? exp)
    (eval (catch-tag exp) env (make-cabodyc (catch-body exp) env cont)))
   ((throw? exp)
    (eval (throw-tag exp) env (make-thbodyc (throw-body exp) env cont)))
   ...
))

(define (catch? exp) (tagged-list? exp 'catch))
(define (catch-tag exp) (cadr exp))
(define (catch-body exp) (cddr exp))

(define (throw? exp) (tagged-list? exp 'throw))
(define (throw-tag exp) (cadr exp))
(define (throw-body exp) (caddr exp))
```

Figure 8: Implementation of `catch` and `throw` (1): `eval` and representing expressions.

```
(define (apply-cont cont val)
  (cond ...
    ((cabodyc? cont)
     (eval-sequence (cabodyc-body cont) (cabodyc-env cont)
                    (make-catchc val (cabodyc-cont cont))))
    ((catchc? cont)
     (apply-cont (catchc-cont cont) val))
    ((thbodyc? cont)
     (eval (thbodyc-body cont) (thbodyc-env cont)
           (make-throwc val (thbodyc-cont cont))))
    ((throwc? cont)
     (let ((stripped-cont
             (first-matching-catch (throwc-tag cont) (throwc-cont cont))))
       (if stripped-cont
           (apply-cont stripped-cont val)
           (list 'uncaught (throwc-tag cont) val))))
    ((haltc? cont) (list 'normal val))
    ...
))
```

Figure 9: Implementation of `catch` and `throw` (2): `apply-cont`

exception to signal that the exception is not caught.[2] Figure 10 shows `first-matching-catch`. The basic idea is to traverse the continuation until a mark is found. When such a continuation (that satisfies `catchc?`) is found, the tag in the thrown exception nad that of `catch`, obtained by `catchc-tag`, are compared; if they agree, then it returns the continuation of the mark.

Figure 11 shows procedures to minipulate continuations related to `catch` and `throw`; Figure 12 shows the new definition of `driver-loop`.

Exercise 4 The special form `catch` introduced here returns the thrown value *as it is*. It can be inconvenient if one wants to distinguish whether the evaluation of the body terminateed normally or abnormally.

Modify the behavior of `catch` so that an *exception handler* is called when an exception is thrown. More concretely, extend the syntax of `catch` to the following form:

`(catch` *tag handler* exp_1 ... exp_n`)`

Then, modify the evaluator so that, if an exception is thrown during the evaluation of the body (exp_1 ... exp_n) and the tag in the thrown exception is equal to *tag*, then *handler* is applied to the thrown value and the result of the application becomes the final value of `catch`.

Finally, rewrite the definition of `change` in Excercise 3 by using new `catch` as follows.

```
(define (change coins amount)
  (cond ((zero? amount) '())
```

[2] If evaluation terminates normally (the clause with `haltc?` in `apply-cont`, `apply-cont` returns the symbol `normal` togetehr with value of the expression. `Driver-loop` has to be modified accordingly. See Figure 12.

```
(define (first-matching-catch thrown-tag cont)
  (define (loop cont)
    (cond ((haltc? cont) false)
          ((testc? cont) (loop (testc-cont cont)))
          ((assignc? cont) (loop (assignc-cont cont)))
          ((definec? cont) (loop (definec-cont cont)))
          ((beginc? cont) (loop (beginc-cont cont)))
          ((operandsc? cont) (loop (operandsc-cont cont)))
          ((applyc? cont) (loop (applyc-cont cont)))
          ((restopsc? cont) (loop (restopsc-cont cont)))
          ((consc? cont) (loop (consc-cont cont)))
          ((cabodyc? cont) (loop (cabodyc-cont cont)))
          ((catchc? cont)
           (if (eq? thrown-tag (catchc-tag cont))
               (catchc-cont cont)
               (loop (catchc-cont cont))))
          ((thbodyc? cont) (loop (thbodyc-cont cont)))
          ((throwc? cont) (loop (throwc-cont cont)))
          (else (error "Unknown continuation type -- FIRST-MATCHING-CATCH" cont))))
  (loop cont))
```

Figure 10: Implementation of `catch` and `throw` (3): `first-matching-catch`

```
      ((null? coins) ...)
      (else
        (let ((c (car coins)))
          (if (> c amount) (change (cdr coins) amount)
              (catch 'fail
                 (lambda (x) ...)
                 (cons c (change coins (- amount c))))))))))
```

```
(define (make-cabodyc body env cont) (list 'cabodyc body env cont))
(define (cabodyc? cont) (tagged-list? cont 'cabodyc))
(define (cabodyc-body cont) (cadr cont))
(define (cabodyc-env cont) (caddr cont))
(define (cabodyc-cont cont) (cadddr cont))

(define (make-catchc tag cont) (list 'catchc tag cont))
(define (catchc? cont) (tagged-list? cont 'catchc))
(define (catchc-tag cont) (cadr cont))
(define (catchc-cont cont) (caddr cont))

(define (make-thbodyc body env cont) (list 'thbodyc body env cont))
(define (thbodyc? cont) (tagged-list? cont 'thbodyc))
(define (thbodyc-body cont) (cadr cont))
(define (thbodyc-env cont) (caddr cont))
(define (thbodyc-cont cont) (cadddr cont))

(define (make-throwc tag cont) (list 'throwc tag cont))
(define (throwc? cont) (tagged-list? cont 'throwc))
(define (throwc-tag cont) (cadr cont))
(define (throwc-cont cont) (caddr cont))
```

Figure 11: Implementation of `catch` and `throw` (4): Representing continuations

```
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let* ((input (read))
         (output (eval input the-global-environment (make-haltc))))
    (cond ((tagged-list? output 'normal)
           (announce-output output-prompt)
           (user-print (cadr output)))
          ((tagged-list? output 'uncaught)
           (display ";;; Uncaught exception: ")
           (display (cadr output))
           (display " ")
           (user-print (caddr output))))
    (driver-loop)))
```

Figure 12: Implementation of `catch` and `throw` (5): `driver-loop`