Applications of Continuations

Daniel P. Friedman
Indiana University

Organization:

The lecture is in three parts. In the first part we define, primarily by example, the key terms. These are conventional procedures, escape procedures and continuations. The primitive notions of `lambda^` and `call/cc` are presented. In the second part we develop examples of simple applications, specifically a simple LISP-like `BREAK` and a `CYCLE` procedure which loops indefinitely, but makes available the ability to break out of the loop. Also in this section we point out that `call/cc` is not strictly necessary if all procedures are written in continuation-passing-style. We complete this section by demonstrating that even the unpopular GO TO style programming of LISP's `PROG` looks reasonable in the presence of the proper uses of `call/cc`. In the last section we develop a `DISPATCHER` for synchronous processes. At the end of these notes are two special features. The first is a set of ten exercises that we hope you will enjoy. Some of them are puzzles designed to challenge and others are more realistic like adding asynchronous processes and exception handling to the `DISPATCHER`. Finally we have included two appendices. In Appendix A we have presented a meta-circular interpreter for a fully curried version of a subset of Scheme which includes `call/cc` and `lambda^`. In Appendix B we have included the remaining code necessary to test `DISPATCHER`.

Outline:

I. Key Terms and Definitions

I-1. **Properties of Scheme**

The programming language Scheme is a call-by-value lexically scoped dialect of LISP with first-class procedures. A condition imposed on any implementation is that it be properly tail-recursive. In simple terms this means that there is no control stack growth when unnecessary and that, in particular, the cost of simple loops written using recursion is minimal. A subset of the language is functional, but the entire language supports imperative concepts such as input/output, assignment to lexical variables and first-class continuations.

I-2. **Description of escape procedures**

Suppose we consider the following simple expression:

```
(* (/ 24 (f 0)) 3)
```

Possible outcomes:

1. `(f 0)` is 4, so the result is 18.
2. `f` is undefined at 0, leading to an error message and causing the computation to abort.
3. `f` is undefined at 0, leading to an infinite loop, *e.g.*,
   ```
   (define f
     (lambda (n)
       (if (zero? n) (f n) n)))
   ```
4. `(f 0)` is 4, but `f` is an escape procedure so the result is 4.

I-3. **Notation for characterizing escape procedures**

We introduce a notation for characterizing escape procedures. If `f` is a procedure, its corresponding escape procedure is `f^`. `f^` does exactly what `f` does, but `(f^ ...)` escapes as the answer and it is subsequently printed. Sometimes this is referred to as *escaping to the top level or read-eval-print loop.*

The simplest escape procedure corresponds to the identity: `I`. If `f^` = `I^`, then the result is 0 and no division occurs. A more powerful escape procedure is `+^`.

Consider: `(* 3 (+^ 4 5))` $\Longrightarrow$ `9`.

But note that this is similar to `(* 3 (I^ (+ 4 5)))` $\Longrightarrow$ `9`.

I-4. **Invoking escape procedures is replacing the control stack**

In the expression above the contents of the control stack at the time of invoking `(+^ 4 5)` is <3, *>. Invoking the escape procedure has the effect of forgetting the current contents of the control stack and processing only the `+^` and its arguments. Although it appears that `(I^ (+ 4 5))` is the same as `(+^ 4 5)` they are *not* operationally the same. There is more growth of the control stack with `(I^ (+ 4 5))`. This understanding of the growth

of the control stack will play a rôle later when we look at sophisticated uses of escape procedures.

I-5. `lambda^` : the creator of general escape procedures

Given any lambda expression we can form its corresponding escape counterpart.

So that `(lambda (x) ...)` becomes `(lambda^ (x) ...)`.

We assume the existence of such a capability. Later we clarify how such objects are built.

I-6. `call/cc` : a creator of escape procedures

Scheme has a feature for creating escape procedures internally and making them available to the user. In Scheme these procedures also interact with fluids. However, for purposes of this lecture we will not discuss the vagaries of fluids. See [9] for a discussion of continuations in the presence of fluids.

These escape procedures correspond to the control stack in the run-time architecture. The control stack corresponds to the continuation in a continuation semantics. That is why these particular escape procedures are often referred to as continuations or continuation objects.

In Scheme one writes

```
(call-with-current-continuation e)
```

This has the effect of invoking `(e k^)` where `k^` is the escape procedure corresponding to the expression instance `(call-with-current-continuation e)`. From here on "`call-with-current-continuation`" is shortened to "`call/cc`".

I-7. Describing escape procedures created by `call/cc`

For example, suppose we have the expression:

```
(+ 3 (call/cc (lambda (k^) ...)))
```

then `k^` can be formed by replacing the `call/cc` expression by a fresh variable, say `v`, and abstracting with `v` over the result:

```
k^ = (lambda^ (v) (+ 3 v))
```

I-8. A very simple example of `call/cc`

Let's practice:

```
(+ (call/cc
     (lambda (k^)
       (/ (k^ 5) 4)))
   8)
```

$\Longrightarrow$ First form `k^ = (lambda^ (v) (+ v 8))`
$\Longrightarrow$ Evaluate `(/ (k^ 5) 4)` knowing the value of `k^`
$\Longrightarrow$ Evaluate `(k^ 5)`
$\Longrightarrow$ 13 is the result, the waiting division is forgotten since `k^` is an escape procedure.

I-9. **Continuations describe the rest of the computation**

Another simple example:

```
(* (+ (call/cc
       (lambda (k^)
         (/ (k^ 5) 4)))
     8)
   3)
```

$\Longrightarrow$ First form `k^ = (lambda^ (v) (* (+ v 8) 3))`
$\Longrightarrow$ Evaluate `(/ (k^ 5) 4)`
$\Longrightarrow$ Evaluate `(k^ 5)`
$\Longrightarrow$ Evaluate `((lambda^ (v) (* (+ v 8) 3)) 5)`
$\Longrightarrow$ Result is 39.

The use of "`*`" is arbitrary here. What we mean is whatever is left to do. In this case the only thing left to do is multiplication. Of course, it could have been any f and in that case we would have all the work of f left to do.

This next example shows that it is not always completely obvious what is meant by *the rest of the computation.*

```
(* (+ (let ([u (+ 3 2)])
       (call/cc
         (lambda (j^)
           (/ (j^ u) 4))))
    8)
 3)
```

Here the work of `u` "becoming `5`" occurs prior to determining the value for `j^`. The continuation `j^` is the same as `k^`. Since `(+ 3 2)` is `5`, both expressions yield the same result.

I-10. **Continuations are first-class objects**

With `call/cc` we might choose to save away the escape procedure, but not invoke it until later in the computation.

```
(+ (call/cc
     (lambda (k^)
       (begin
         (set! +8^ k^)
         (display "inside body")
         5)))
   8)
```

$\Longrightarrow$ First form `k^ = (lambda^ (v) (+ v 8))`
$\Longrightarrow$ Evaluate the body
```
(begin
   (set! +8^ k^)
   (display "Inside body")
   5)
```
$\Longrightarrow$ The lexical variable `+8^` is set to `(lambda^ (v) (+ v 8))`,
"Inside body" is printed,
and 5 is returned as the result of the body.
$\Longrightarrow$ 13 is the answer of the entire expression.

Later we may invoke `(* (/ (+8^ 35) 0) 100)` $\Longrightarrow$ 43

II. Meta-Programming

II-1. **Default Law of** `call/cc`

The reason that 5 is sent to the waiting `+` in the previous example is that we claim:

`(call/cc (lambda (k^) e))` = `(call/cc (lambda (k^) (k^ e)))`

That is, the continuation waiting for the result is the same as the escape procedure created by `call/cc`. From this equation we know that `e` defaults to `(k^ e)`.

Thus in our example,

```
(+ (call/cc
     (lambda (k^)
       (k^ (begin
             (set! +8^ k^)
             (display "Inside body")
             5))))
   8)
```

The result is `(k^ 5)` or 13.

II-2. **With `lambda^` the Default Law of `call/cc` is not true**

Escaping by applying `call/cc` to an escape procedure (*i.e.*, a procedure `(lambda^...))` imposes on the user even more control responsibility. The user of the language is then required to make every decision about exiting from the body of the applied escape procedure. Look at the following two examples. They appear equivalent using the Default Law from above. But that equation used `(lambda ...)` not `(lambda^...)`. With `(lambda^ ...)` being applied to the current continuation we get different behaviors.

```
(+ 3 (call/cc (lambda^ (k^) (k^ 8))))  Here we would get: 11
(+ 3 (call/cc (lambda^ (k^) 8)))       But here we would just get: 8
```

We conclude this portion of the lecture with the reminder that `call/cc` may be passed either a conventional or an escape procedure, and the continuation is *always* an escape procedure like `k^`.

II-3. **A simple LISP-like `BREAK`**

By saving continuations we can write a `BREAK` procedure that allows for the computation to `RESUME` at the discretion of the user. Upon invocation of `BREAK` a message will be returned to top level. The system listening loop will then be in control until the user executes `(RESUME ...)`, at which time the argument to `RESUME` will be the value of the original `(BREAK ...)` invocation.

```
(define BREAK
  (lambda (message)
    (call/cc
      (lambda (k^)
        (set! RESUME k^)
        ((lambda^ (x) x) message)))))
```

`BREAK` is a fascinating example where the use of first-class continuations leads to an elegant solution. Brian Smith chose a more sophisticated version of `BREAK` as an example of the power of reflection [25].

II-4. **How to construct `lambda^`**

Next we consider how to form `(lambda^ (i ...) e ...)` and note why we cannot think about `(lambda^...)` as part of Scheme, but must always construct it at run time. We define a procedure `INVOKE/NO-CONT` which takes as an argument a procedure `f` of no arguments and invokes `f` as if it were an escape procedure. Given `INVOKE/NO-CONT` we can syntactically express `lambda^`.

```
(lambda^ (id ...) e ...)
  ≡
(lambda (id ...) (INVOKE/NO-CONT (lambda () e ...)))
```

Below is the procedure `make-INVOKE/NO-CONT` followed by its invocation.

```
(define make-INVOKE/NO-CONT
  (lambda ()
    ((call/cc
       (lambda (k^)
         (set! INVOKE/NO-CONT (lambda (th) (k^ th)))
         (lambda () 'INVOKE/NO-CONT))))))

(make-INVOKE/NO-CONT)
```

The `(make-INVOKE/NO-CONT)` must be invoked from the top-level. When we form the `k^` for building the definition of `INVOKE/NO-CONT,` we see that it is `(lambda^ (v) (v))`, with no continuation. Hence, we are invoking the procedure of zero arguments in the "empty" or "top-level" continuation when we pass it to `INVOKE/NO-CONT.` The clue that this is happening is the *double left* parentheses around the `call/cc`. By binding `INVOKE/NO-CONT` to a procedure whose sole purpose is to invoke its argument, we can guarantee that there is no continuation waiting for its result. Think what would happen if instead we invoked `(+ 5 (make-INVOKE/NO-CONT))` at top-level.

II-5. **Programming in continuation-passing-style**

The following program yields the sum of the nodes in a binary search tree. If a 0 is found in the tree, the result is 0.

```
(define sum-bst
  (lambda (t)
    (call/cc
      (lambda (exit^)
        (letrec
          ([sum-bst
             (lambda (t)
               (cond
                 [(null? t) 0]
                 [(zero? (info t)) (exit^ 0)]
                 [else (+ (info t)
                          (sum-bst (left t))
                          (sum-bst (right t)))]))])
          (sum-bst t))))))
```

Below is the equivalent definition in continuation-passing-style [26].

```
(define sum-bst
  (lambda (t)
    (letrec
      ([sum-bst
         (lambda (t k)
           (cond
             [(null? t) (k 0)]
             [(zero? (info t)) 0]
             [else (sum-bst (left t)
                     (lambda (result1)
                       (sum-bst (right t)
                         (lambda (result2)
                           (k (+ (info t) result1 result2))))))]))])
      (sum-bst t (lambda (x) x)))))
```

Such programs can be read and written with practice. Consider the `else` line. It says: "Imagine that you have the result of the left tree and call it `result1`, then imagine you have the result of the right tree and call it `result2`, then take the sum of the information at the root of the tree with the sum of `result1` and `result2` and pass this value to the waiting continuation `k`". The zero test line says to abandon the waiting continuation and return the value 0 to the continuation of the original invoker of `bst`. Some find this a bit awkward, but the real difficulty is that writing in this style often imposes its will on every program. For instance, if one uses a mapping procedure then the procedure that is being mapped might also be required to be in this style.

To quote James S. Miller [19] "Unfortunately, the procedures resulting from the conversion process are often difficult to understand. The argument that continuations need not be added to the Scheme language is factually correct. It has as much validity as the statement that *the names of the formal parameters can be chosen arbitrarily.* And both of these arguments have the same basic flaw: the form in which a statement is written can have a major impact on how easily a person can understand the statement. While understanding that the language does not inherently need any extensions to support programming using continuations, the Scheme community nevertheless chose to add one operation to the language to ease the chore."

II-6. **Meta-Programming with `call/cc`: `CYCLE`**

We can meta-program:

`CYCLE` works for infinite loops with an `EXIT-CYCLE-WITH`

```
(define CYCLE
  (lambda (f)
    (call/cc
      (lambda (k^)
        (letrec ([loop (lambda ()
                         (f k^)
                         (loop))])
          (loop))))))
```

The protocol for `CYCLE` follows:

```
(CYCLE (lambda (EXIT-CYCLE-WITH) e ...))
```

The expectation is that somewhere within the `e ...` there is at least one invocation of `(EXIT-CYCLE-WITH ...)`. This way, the infinite loop will terminate with the argument to the first `(EXIT-CYCLE-WITH ...)` that it encounters.

II-7. **LISP's `PROG` with `GO` and `RETURN`: "GO TO" programming revisited**

Let us consider LISP's `PROG`:

```
(PROG (id ...)
  label_1 e_1 ...
  ...
  label_{n-1} e_{n-1} ...
  label_n e_n ...)
```

≡

```
 (let ([id '()] ...)
  ((call/cc
    (lambda (GO)
      (let ([RETURN (lambda (v) (GO (lambda () v)))])
        (letrec
          ([label_1 (lambda () e_1 ... (label_2))]
           ...
           [label_{n-1} (lambda () e_{n-1} ... (label_n))]
           [label_n (lambda () e_n ... (RETURN '()))])
          (label_1))))))
```

With this definition we avoid any risk of losing tail recursion. Even if the program contains an instance of bizarre code like

```
(begin
  (GO x)
  (print 3))
```

it will still be run as if the instance were just `(GO x)`. The reason that we need invoke only `(label`$_i$`)` instead of `(GO label`$_i$`)` is that these particular invocations are always in tail-recursive position.

These definitions are strictly more general than necessary. For example, it is possible to have instances of `(f label`$_i$`)`, `(f RETURN)`, and `(f GO)` within the `e ...`. Furthermore, `label`$_i$, `RETURN` and `GO` may be stored away for later use. This is how some of the mechanisms for non-blind backtracking such as `a-bien-tot` and `au-revoir` of Conniver[27] were implemented.

III. Synchronous Processes

III-1. Continuations as synchronous processes

We will next focus on the process-oriented aspects of continuations. A continuation represents a locus of control and we can use this to implement processes [30]. A process scheduler can be designed based on a ready queue of continuations.

If we think about a sequence of expressions `(begin S`$_1$ `S`$_2$ `...)` then whenever `S`$_i$ binds a continuation, its continuation looks like `(lambda^ (waste) S`$_{i+1}$ `S`$_{i+2}$ `...)`. These are called command continuations because we are not interested in the value of the continuation invocation. We show this by choosing the argument name "waste" which should not occur free within `S`$_i$'s.

For example:

```
(define foo
  (lambda ()
    (display 2)
    (call/cc (lambda (ak^) ...))
    (display 3)
    (call/cc (lambda (bk^) ...))
    (display 5)
    (call/cc (lambda (ck^) ...))
    (display 7)))
```

If we invoke `foo` at the top level (with no context) then:

```
ak^ = (lambda^ (waste)
         (display 3)
         (call/cc (lambda (bk^) ...))
         (display 5)
         (call/cc (lambda (ck^) ...))
         (display 7))

bk^ = (lambda^ (waste)
         (display 5)
         (call/cc (lambda (ck^) ...))
         (display 7))

ck^ = (lambda^ (waste)
         (display 7))
```

III-2. **Extending the language of processes**

We extend the language of expressions to include `(PAUSE-HANDLER)`.

If in our example we replace `(call/cc (lambda (ik^) ...))` by a procedure invocation of `PAUSE-HANDLER` we will get something that looks like this:

```
(define foo
  (lambda ()
    (display 2)
    (PAUSE-HANDLER)
    (display 3)
    (PAUSE-HANDLER)
    (display 5)
    (PAUSE-HANDLER)
    (display 7)))
```

Then the procedure for `PAUSE-HANDLER` would be responsible for the `call/cc`.

```
(define PAUSE-HANDLER
  (lambda ()
    (call/cc
      (lambda (k^)
        ....))))
```

What is left is to decide what to do with these continuations that we have bound with `call/cc`. In this simple example we will

1. Insert the continuation into the process queue.
2. Delete `k^` (*i.e.,* a process object) from the front of the queue.

3. Invoke (*i.e.,* `RUN`) the continuation `k^`.

Hence we define `PAUSE-HANDLER` as

```
(define PAUSE-HANDLER
  (lambda ()
    (call/cc
      (lambda (k^)
        (swap-run the-process-q k^)))))
```

III-3. **Swapping and running the current process**

`swap-run` does the three steps alluded to above.

```
(define swap-run
  (lambda (q k^)
    (q 'en-q! k^)
    (RUN (q 'de-q!))))

(define RUN (lambda (k^) (k^ 'waste)))
```

`RUN` invokes the continuation that is removed from the process queue.

III-4. **Creating processes**

The only issue remaining is the development of a protocol for processes: "code which uses the `PAUSE-HANDLER` feature". What should we do when there is nothing left to process? A simple solution is to require an insertion of a `HALT` command. We clarify this protocol by introducing `CREATE-PROCESS`. Processes, like continuations that are carved out of processes, must be procedures of one argument.

```
(define CREATE-PROCESS
  (lambda (th)
    (lambda (v)
      (th)
      (HALT))))
```

III-5. `HALT` **and** `DISPATCHER`

We next define the `DISPATCHER` procedure as a simple synchronous process scheduler. Each time `DISPATCHER` is invoked a new queue is created. The queue created by `(create-q exit^)` is built with the `exit^` continuation. The queue invokes the `exit^` continuation whenever there is an attempt to delete a process from an empty queue. This will exit `DISPATCHER`. If the queue is not empty we sucessfully remove an element from the queue

and RUN it. We include PAUSE-HANDLER in the definition of DISPATCHER so that it will close over the free variable the-process-q.

```
(define DISPATCHER
  (lambda (initialize-q)
    (call/cc
      (lambda (exit^)
        (let ([the-process-q (create-q exit^)])
          (set! HALT
            (lambda ()
              (RUN (the-process-q 'de-q!))))
          (set! PAUSE-HANDLER
            (lambda ()
              (call/cc
                (lambda (k^)
                  (swap-run the-process-q k^)))))
          (initialize-q the-process-q)
          (HALT))))))
```

The invoker of DISPATCHER passes a procedure which takes a queue as an argument. This procedure is invoked just prior to RUNning (surprisingly with HALT) the first process. We use it to initialize the queue. For a demonstration of how we used the initialize-q procedure and for all the code necessary to run DISPATCHER, see Appendix B.

III-6. **Exercises**

1. The `BREAK` program is not very general. Suppose that another invocation of `(BREAK ...)` occurs. Then the `RESUME` would not be correct. Redesign the `BREAK` procedure so that the `RESUME` interface is unchanged, but that many instances of `BREAK` can co-exist.
2. A slightly more general variant of `CYCLE` is to allow the user's program the ability to loop back to the top. Here the protocol would be as follows:

```
(CYCLE (lambda (EXIT-CYCLE-WITH AGAIN) e ...))
```

   An example:

```
((lambda (n m)
   (CYCLE
     (lambda (EXIT-CYCLE-WITH AGAIN)
       (if (= n m) (EXIT-CYCLE-WITH n)
           (if (= n (+ m 5))
               (begin (display "explicit looping")
                      (newline)
                      (set! n (sub1 n))
                      (AGAIN))
               (set! n (sub1 n))))
       (display "implicit looping")
       (newline))))
  24 3)
```

   If the variable `AGAIN` is free in `e ...`, then any invocation of `(AGAIN)` within `e ...` will take it to the top of the loop. Be careful to guarantee that these invocations of `(AGAIN)` are tail recursive.
3. Reynolds [22] has a different approach to `PROG`. He builds as many escape procedures as there are labels but each procedure is the concatenation of all the statements below the label. This has the disadvantage of generating larger programs and the advantage of avoiding the generated uses of $(\texttt{label}_i)$. Implement `PROG` using his approach.
4. Find out how exception handlers are bound in your Scheme system and wire `PAUSE-HANDLER` to it. Run test programs which have no instances of `PAUSE-HANDLER` physically within the program but which invoke `PAUSE-HANDLER` as a result of an exception.
5. Find out how pre-emption works in your Scheme system and wire `PAUSE-HANDLER` to it. Run test programs which have no instances of `PAUSE-HANDLER` physically within the program but which invoke `PAUSE-HANDLER` as a result of something external to the program. This yields asynchronous processes. Some Scheme systems use engines for modeling timed preemption [12].
6. We're just a step away from a coroutine system. In a coroutine system the decision as to which process is run when a pause occurs becomes the responsibility of the writer of

the code. We can use the continuations to represent coroutines just as we used them to represent processes. We introduce a new command, `(RESUME c)`, which relinquishes control and runs the coroutine `c`. Add `(RESUME ...)` as a command to the `DISPATCHER` by including another operation on the queue. For a completely different approach to coroutines see [13] or [29].

7. What changes should be made to insure that the language supported by DISPATCHER would allow invocations of `(DISPATCHER ...)`. Hint: The solution to this exercise might surprise you.
8. The programming language ICON [9] has an interesting control facility. Each expression in ICON has 0, 1, or many values. Create an interface which extends the notion of expressions to those having multiple values as is done in ICON.
9. Are continuations too general? Consider the following puzzle. Suppose that we only have a primitive `STATE` which we define as follows:

```
(define STATE
  (lambda ()
    (call/cc
      (lambda (k^)
        (k^ k^)))))
```

   Is it possible to define `call/cc` in terms of `STATE`?
10. Consider a program that includes, in addition to the standard expressions, three types of special purpose expressions: `(MILESTONE ...)`, `(DEVIL ...)`, and `(ANGEL ...)`. The computation described by the program has the goal of finishing despite the existence of devils. A devil sends control back to the last milestone. The value given to the devil is passed to the continuation commencing at the milestone, as if that value were the result returned by the milestone expression. Presumably this allows the computation to take a different path, possibly avoiding the devil that is lurking somewhere ahead on the previously used path. If another devil, or maybe even the same devil, is subsequently encountered, then control passes back to the penultimate milestone, not to the one just used. In other words, each milestone can be returned to exactly once; a succession of devils pushes the computation back to earlier and earlier states. An angel sends the computation forward to where it was when it most recently encountered a devil. The value passed as a parameter to the angel is given to the devil's continuation as if it were the value of the devil. A succession of angels pushes the computation forward to more advanced states. A milestone is a procedure of one argument that acts as the identity, as well as recording the current context for later use by devils. If a devil (or angel) is encountered with no milestone (or devil) remaining, the devil (or angel) has no effect. To recharge any milestone the milestone must be re-evaluated. Implement the procedures `MILESTONE`, `DEVIL`, and `ANGEL`. There is no non-determinism; this is, however, an example of non-blind backtracking [8]. For an example of blind backtracking with continuations see [11].

1. Burge, W. *Recursive Programming Techniques.* Addison-Wesley, Reading, Mass., 1975.

2. Clinger, W.D., D.P. Friedman and M. Wand. A scheme for a higher-level semantic algebra. In *Algebraic Methods in Semantics*, edited by J. Reynolds and M. Nivat. Cambridge University Press, London, 1985, 237–250.

3. Dybvig, R. Kent. *Three Implementation Models for Scheme.* Ph.D. dissertation, University of North Carolina at Chapel Hill, 1987.

4. Dybvig, R. Kent. *The Scheme Programming Language*, Prentice Hall, Englewood Cliffs, New Jersey, 1987.

5. Felleisen, M. *The Calculi of Lambda-v-CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* Ph.D. dissertation, Indiana University, 1987.

6. Felleisen, M. The theory and practice of first-class prompts. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, 1988, to appear.

7. Felleisen, M., D.P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control, *Theor. Comput. Sci.***52**(3), 1987, 205–237.

8. Friedman, D.P., C.T. Haynes, and E. Kohlbecker. Programming with continuations. In *Program Transformations and Programming Environments*, edited by P. Pepper. Springer-Verlag, Heidelberg, 1985, 263–274.

9. Griswold, Ralph E.. The Evaluation of Expressions in ICON. *ACM Trans. Program. Lang. Syst.* **4**(4), 1982, 563–584.

10. Haynes, C.T. Logic continuations. *J. Logic Program.* **4**, 1987, 157–176.

11. Haynes, C.T. and D.P. Friedman. Embedding continuations in procedural objects. *ACM Trans. Program. Lang. Syst.* **9**(4), 1987, 582–598.

12. Haynes, C.T. and D.P. Friedman. Abstracting timed preemption with engines. *Journal of Computer Languages* (Pergamon Press) **12**(2), 1987, 109-121.

13. Haynes, C.T., D.P. Friedman, and M. Wand. Obtaining coroutines from continuations. *Journal of Computer Languages* (Pergamon Press) **11**, 1986, 143–153.

14. Johnson, G.F and D. Duggan. Stores and partial continuations as first-class objects in a language and its environment. In *Proc. 15th ACM Symposium on Principles*

of Programming Languages, 1988, to appear.

15. Johnson, G.F. GL—A language and environment for interactively experimenting with denotational definitions. In *Proc. SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques SIGPLAN Notices* **22**(7), 1987, 165–176.

16. Landin, P.J. A correspondence between ALGOL 60 and Church's lambda notation. *Commun. ACM* **8**(2), 1965, 89–101; 158–165.

17. Landin, P.J. The next 700 programming languages. *Commun. ACM* **9**(3), 1966, 157–166.

18. Landin, P.J. An abstract machine for designers of computing languages. In *Proc. IFIP Congress*, 1965, 438–439.

19. Miller, James S. *Multischeme: A Parallel Processing System Based on MIT Scheme.* Ph.D. dissertation, Massachusetts Institute of Technology, 1987.

20. Rees J. and W. Clinger (Eds.). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices* **21**(12), 1986, 37–79.

21. Reynolds, J.C. Definitional interpreters for higher-order programming languages. In *Proc. ACM Annual Conference*, 1972, 717–740.

22. Reynolds, J.C. GEDANKEN—A simple typeless language based on the principle of completeness and the reference concept. *Commun. ACM* **13**(5), 1970, 308–319.

23. Rojas, Guillermo J. *A Computational Model for Observation in Quantum Mechanics.* Master's Thesis, Massachusetts Institute of Technology, 1986.

24. Smith, Brian Cantwell. *Reflection and Semantics in a Procedural Language.* Ph.D. dissertation, Massachusetts Institute of Technology, 1982.

25. Smith, Brian Cantwell. Reflection and semantics in Lisp. *Proc. 11th ACM Symposium on Principles of Programming Languages*, 1984, 23–35.

26. Steele, Guy L. *Rabbit: A Compiler for Scheme.* Master's Thesis, Massachusetts Institute of Technology, 1978.

27. Sussman, G.J. and D.V. McDermott. From Planner to Conniver—A genetic approach. In *Proceedings of the Fall Joint Computer Conference* **41**. AFIPS Press, Reston, Va., 1972, 1171–1179.

28. Sussman, G.J. and G. Steele. Scheme: An interpreter for extended lambda calculus. Memo 349, MIT AI Lab, 1975.

29. Talcott, C. *The Essence of Rum—A Theory of the Intensional and Extensional Aspects of Lisp-type Computation.* Ph.D. dissertation, Stanford University, 1985.

30. Wand, M. Continuation-based multiprocessing. In *Proc. 1980 ACM Conference on Lisp and Functional Programming*, 1980, 19–28.

31. Wand, M. and D.P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, 1986, 298–307.

Appendices

Appendix-A: A meta-circular interpreter which includes `lambda^` and `call/cc`

In order to help clarify the meanings of the forms used in this lecture we include an interpreter which will run in Scheme. This interpreter is derived from one in [3]. Issue's of side-effects are germane, but are ignored as the goals of this lecture are primarily about the applications of continuations. See Johnson and Duggan [14] for a further study of stores. As usual we assume the expression is curried. A good exercise is to rewrite this program in continuation-passing-style. `(record-case e ...)` is described in detail in Dybvig [4].

It matches against the `(car e)` and lexically binds while pairing against the `(cdr e)`.

```
(define M
  (let ([extend (lambda (r var val)
                  (lambda (i) (if (eq? i var) val (r i))))])
    (lambda (e r)
      ((call/cc
        (lambda (I^)
          (letrec
            ([U
              (lambda (e r)
                (cond
                  [(number? e) e]
                  [(symbol? e) (r e)]
                  [else (record-case e
                          [quote (lit) lit]
                          [lambda (fs b)
                           (lambda (v)
                             (U b (extend r (car fs) v)))]
                          [lambda^ (fs e)
                           (lambda (v)
                             (I^ (lambda () (U e (extend r (car fs) v)))))]
                          [if (test t-pt f-pt)
                           (if (U test r) (U t-pr r) (U f-pt r))]
                          [call/cc (f) (call/cc (U f r))]
                          [else  ((U (car e) r) (U (cadr e) r))])]))])
            (lambda () (U e r))))))))))
```

The special form `execute` below builds an environment with just two identifiers: `+` and `cons`. Feel free to add additional symbols to the environment, but remember that everything is curried.

```
(extend-syntax (execute)
  [(execute e) (M 'e (let ([+ (lambda (x) (lambda (y) (+ x y)))]
                           [cons (lambda (x) (lambda (y) (cons x y)))])
                       (lambda (i)
                         (cond
                           [(eq? i '+) +]
                           [(eq? i 'cons) cons]
                           [else (error i "undefined-id")])))))])
```

Appendix-B: Supporting code for `DISPATCHER`

Queues created by `create-q` respond to `en-q!` and `de-q!` messages. If an attempt is made to delete something from an empty queue then the continuation which was passed as an argument to `create-q` is invoked. The test for an empty queue is not needed if an enqueue has just preceded it. A good exercise is to add a new message that combines the actions of `en-q!` and `de-q!`. Then `swap-run` becomes `(RUN (q 'en-q-de-q! k^))`.

```
(define create-q
  (lambda (where-to-exit-when-empty^)
    (let ([head '()] [tail t])
      (lambda msg
        (case (car msg)
          [en-q! (if (null? head)
                     (begin (set! head (cons (cadr msg) head))
                            (set! tail head))
                     (begin (set-cdr! tail (cons (cadr msg) '()))
                            (set! tail (cdr tail))))]
          [de-q! (if (null? head)
                     (where-to-exit-when-empty^ 'done)
                     (begin0 (car head)
                             (if (eq? head tail)
                                 (set! head '())
                                 (set! head (cdr head)))))])))))
```

The procedures `process-maker`, `build-q` and `test` create a test program.

```
(define process-maker
  (lambda (n)
    (CREATE-PROCESS
      (lambda ()
        (display n)
        (PAUSE-HANDLER)
        (newline)
        (display "about to halt")
        (newline)
        (PAUSE-HANDLER)
        (display n)))))


(define build-q
  (lambda (maker n)
    (lambda (q)
      (letrec
        ([loop (lambda (n)
                 (cond
                  [(zero? n) 'queue-built]
                  [else (q 'en-q! (maker n))
                        (loop (sub1 n))]))])
        (loop n)))))

(define test
  (lambda ()
    (DISPATCHER (build-q process-maker 8))))
```