CSSE 304 Days 27 - 29

Receivers

Escape procedures

Intro to `call/cc`

`call/cc` examples

Receivers

Escape procedures

`call/cc` involves both receivers and escape procedures, so we look at both of those first.

WARM-UP FOR CALL/CC

What we'll do today and next time is loosely based the book *Scheme and the Art of Programming* by George Springer and Daniel Friedman.

Review of Continuations

- Consider the evaluation of the expression:

```
(let ([x (+ y 2)])
   (if (< x 4) 5 (- x 6)))
```

What is the continuation of

```
(+ y 2) ?                  6 ?

(- x 6) ?                  (< x 4)
```

Receivers

- A **receiver** is an argument (which also happens to also be a procedure) passed to a procedure, with the intention that the procedure will eventually pass values to that receiver.
- **Example:** The continuations that we pass to CPS procedures (with Scheme procedure continuations) are receivers.
- Sometimes receivers are called "callbacks"

Old Receiver Example: call-with-values

- ```
  > (call-with-values
       (lambda () (values 3 4))
       list)
  (3 4)
  ```
- **`list`** is a receiver

  (we previously called it the *consumer*)
  ```

new receiver example

From TSPL: The following shows the use of **`call-with-output-file`** to write a list of objects (the value of `list-to-be-printed`), separated by newlines, to the file named by "myfile.ss."

```
(call-with-output-file "myfile.ss"
   (lambda (p) ; this is the "receiver"
    (let f ([ls list-to-be-printed])
       (if (not (null? ls))
           (begin
             (write (car ls) p)
             (newline p)
             (f (cdr ls)))))))
```

The receiver expects to receive an output port as its argument

```
(define call-with-output-file
  (lambda (filename proc)
    (let ((p (open-output-file filename)))
      (let ((v (proc p)))
        (close-output-port p)
        v))))
```

An escape procedure

- **Pretend that we have a procedure `escape-+` that adds its arguments and returns this sum as the final answer, no matter what the context.**

```
(* (escape-+ 5 6) 3)   →
(escape-+ (escape-+ 2 4) 5) →
```

An escape procedure

- **Pretend that we have a procedure** `escape-+` **that adds its arguments and returns this sum as the final answer, no matter what the context.**

```
(* (escape-+ 5 6) 3)   ➔ 11
(escape-+ (escape-+ 2 4) 5) ➔ 6
```

Escaper (a mostly fictitious procedure)

- **More generally, suppose that we have a procedure** `escaper` **that takes a procedure as an argument and returns an equivalent escape procedure.**
- `(escaper +)` **creates a procedure that is equivalent to** `escape-+`
- `(+ 3 ((escaper +) 4 5))` ➔
-
```
(+ ((escaper (lambda (x)
               (- (* x 3) 7)))
     5)
   4)                                  ➔
```

Escaper (a mostly fictitious procedure)

- More generally, suppose that we have a procedure `escaper` that takes a procedure as an argument and returns an equivalent escape procedure.
- `(escaper +)` creates a procedure that is equivalent to `escape-+`
- `(+ 3 ((escaper +) 4 5))` → 9
-
```
(+ ((escaper (lambda (x)
                (- (* x 3) 7)))
     5)
   4)
```
→ 8

You can experiment with **escaper**

- **You can define `escaper` by loading escaper.ss in the following way:**

`escaper.ss` is linked from the schedule page

```
sliderule 1:12pm > petite escaper.ss
Petite Chez Scheme Version 6.7
Copyright (c) 1985-2001 Cadence Research Systems
> ((call/cc receiver-4))
"escaper is defined"
> (cdr ((escaper cdr) '(4 5 6)))
(5 6)
```

Escape Procedures

- **Let *p* be a procedure. If an application of *p* abandons the current continuation and does something else instead, we call *p* an *escape procedure*.**
- **An example of a Scheme escape procedure that we have already used:**
- **Is `escaper` an escape procedure?**

"call-with" procedures

- **(call-with-values producer consumer)**
  - The receiver is the **`consumer`**.
  - It receives the values returned by a call to the `producer`.
- **(call-with-input-file** filename proc)
  - The receiver is **`proc`**.
  - It receives the input port obtained by opening the input file whose name is `filename`.
- **(call-with-current-continuation receiver)**
  - The `receiver` receives the current continuation.

dining out example

from Springer and Friedman, Part 5 intro

```
(define dine-out
  (lambda (restaurant)
    (enter restaurant)
    (read-menu)
    (let ([food-I-ordered
           (order-some-food)])
      (eat food-I-ordered)
      (pay-for food-I-ordered restaurant)
      (exit restaurant))))
```

Read excerpt from the book

CALL/CC DEFINITION AND EXAMPLES

call/cc

- **call/cc** is an abbreviation for call-with-current-continuation .
- call/cc is a procedure that takes one argument; the argument is a *receiver*.
- this receiver is a procedure that takes one argument; that argument (in this case) is a *continuation*.
- A continuation is a procedure (that takes one argument); that continuation embodies the context of the application of **call/cc**. The continuation is an escape procedure.
- The application (call/cc receiver) has the same effect as (receiver continuation), where the **continuation** is
  - an escape procedure that embodies the execution context of the entire **`call/cc`** expression.

call/cc definition summary

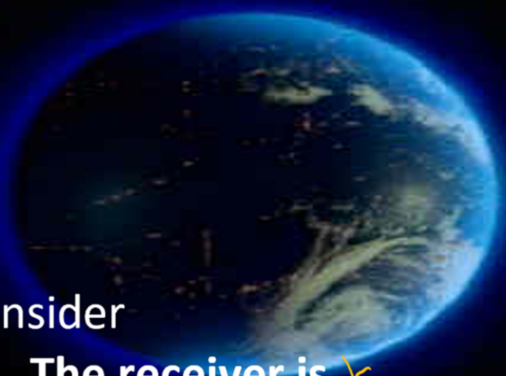
- **(call/cc receiver)** ➔ **(receiver continuation)**,
- Hence the name:
  call-with-current-continuation.
- **Rephrasing it:** What is that continuation?

  If `c` is a procedure that represents the execution context of this application of `call/cc`, then the continuation is equivalent to `(escaper c)`.

call/cc example

- `(call/cc receiver) ➔ (receiver continuation)`
- Consider `(+ 3 (call/cc (lambda (k) (* 2 (k 5)))))`
  - **The receiver is**
  - **The context is**
  - **The continuation is**
  - **Thus** `(+ 3 (call/cc (lambda (k) (* 2 (k 5)))))` **is equivalent to**

call/cc example



```
(call/cc receiver) → (receiver continuation)
```

- Consider `(+ 3 (call/cc (lambda (k) (* 2 (k 5)))))`
  - The receiver is r_1: the procedure that is created when we evaluate (λ(k)(* 2 (k 5)))
  - The context is c_1: the procedure (λ(v)(+ 3 v))
  - The continuation is k_1 (escaper c_1)
  - Thus `(+ 3 (call/cc (lambda (k) (* 2 (k 5)))))` is equivalent to

⟹ (+ 3 (r_1 k_1)) def of call/cc

⟹ (+ 3 (* 2 (k_1 5))) def of procedure application

⟹ (k_1 5) k_1 is an escape procedure.

⟹ ((λ(v) (+ 3 v)) 5) ⟹ 8

More call/cc examples

```
(call/cc receiver) → (receiver continuation)

 (+ 3 (call/cc (lambda (k) (* 2 (k 5)))))
```

a) `(+ 3 (call/cc (lambda (k) (* 2 5))))`

b) `(+ 3 (call/cc (lambda (k) (k (* 2 5)))))`

More call/cc examples

`(call/cc receiver) ➔ (receiver continuation)`

c)
```
(define xxx #f)
(+ 5 (call/cc (lambda (k)
                (set! xxx k)
                2))) ; xxx is equivalent to?
(* 7 (xxx 4))
```

More call/cc examples

`(call/cc receiver) → (receiver continuation)`

c)
```
(define xxx #f)
(+ 5 (call/cc (lambda (k)          ; take the photograph
                (set! xxx k)       ; save the photograph
                2))) ; xxx is equivalent to?
(* 7 (xxx 4))                      ; rub the photograph
```

take the photograph

save the photograph

rub the photograph

A simple call/cc example

(call/cc receiver) ➔ (receiver continuation)

d) **(call/cc procedure?)**

List-index

- Standard approach:

```
(define (list-index item L)
  (cond
    [(null? L) -1]
    [(eq? (car L) item) 0]
    [else (+ 1 (list-index item
                           (cdr L)))]))
```

But "standard recursion" seems so much more natural!

Can we use call/cc to escape with the -1 answer?

What is the problem with this?

One solution: accumulator approach

e)
```
(define list-index
  (lambda (sym L)
   (call/cc
    (lambda (answer)
     (let loop ([L L])
      (cond [(null? L) (answer -1)]
            [(eqv? sym (car L)) 0]
            [else (+ 1
                     (loop (cdr L)))]))))))
 > (list-index 'a '(b a c))
  1
 > (list-index 'a '(b d c))
  -1
```

f)
```
((car (call/cc list)) (list cdr 1 2 3))
```

e)
```
(define list-index
  (lambda (sym L)
   (call/cc
    (lambda (answer)
     (let loop ([L L])
      (cond [(null? L) (answer -1)]
            [(eqv? sym (car L)) 0]
            [else (+ 1
                     (loop (cdr L)))]))))))
> (list-index 'a '(b a c))
 1
> (list-index 'a '(b d c))
 -1
```

```
(call/cc receiver) → (receiver continuation)
```

f)
```
((car (call/cc list))
  (list cdr 1 2 3))
```

Interlude: quotes

- Premature optimization is the root of all evil in programming. - *C.A.R. Hoare*
  - Do you know what he is famous for?

- There is no code so big, twisted, or complex that maintenance can't make it worse. - *Gerald Weinberg*

- Computer Science is the only discipline in which we view adding a new wing to a building as being maintenance. – *Jim Horning*

All this from that short code?

(call/cc receiver) → (receiver continuation)

g)
```
(let ([f 0] [i 0])
  (call/cc (lambda (k) (set! f k)))
  (printf "~a~n" i)
  (set! i (+ i 1))
  (if (< i 10) (f "ignore")))
```

Strange indeed!

h)
```
(define strange1
  (lambda (x)
    (display 1)
    (call/cc x)
    (display 2)))

(strange1
  (call/cc
    (lambda (k) k)))
```

“mondo bizarro” example

i)

```
(define strange2
  (lambda (x)
    (display 1)
    (call/cc x)
    (display 2)
    (call/cc x)
    (display 3)))

(strange2 (call/cc (lambda (k) k)))
```

We probably will not do this one in class; good practice for you.