Other Control Flow ideas: Throw, Catch, Continuations and Call/CC

Lecture 24

Lecture Outline

- Conventional control flow is not everything
- Throw/catch are dynamic returns in Lisp
- Call/CC goes even further (in Scheme)
- Continuation passing is really neat

What control is missing in conventional languages?

Consider these variations:

Type declaration… var x: (if z then int else boolean)

x= (if (z) then foo else bar) (u) /*possible in functional MJ */

Define a method…
Void f() {goto(quit);} /*somewhere (where!?) is a label quit: */

What is NOT dynamic in MJ?

Return can only be to the saved location prior to the call

Exceptions (e.g. Java's try) don't exist, nor is there a way to handle them.

Asynchronous operations (e.g. spawned processes) missing

All these (and more) are in some languages.

Non-local Exits

Rationale: something extremely rare and (generally) bad happens in your program, and you want to pop up to some level, rejecting partial computations.

You cannot wind your way out of a deeply nested chain of calls in any reasonable fashion: you would have to check the return value of every function.

You would like to "goto" some currently active program with some information.

(Note that without some information you would not know how you got to this spot, or that anything went wrong!)

Basis for error handling, but also other applications… maybe you don't want to exit? E.g. Replace all divide-by-zero results by "INF", but CONTINUE program sequence.

Catch and Throw in Lisp / similar to Java try

```
(catch   tag body)

(throw  tag  value)


(catch 'tag (print 1) (throw 'tag 2) (print 3))
```

Prints 1 and returns 2, without printing 3

Catch and Throw / another example

```
(defun print-table(L)
(catch 'not-a-number (mapcar #'print-sqrt L)))

(defun print-sqrt(x)(print (sqrt (must-be-number x))))

(defun must-be-number(x)
   (if (numberp x) x (throw 'not-a-number 'Huh?)))

(print-table '(1 4 x))   ➔

1
2
Huh?
```

Note that the catch is NOT IN THE LEXICAL SCOPE of the throw and bypasses the return from must-be-number, print-sqrt, mapcar.

Other errors..

Built-in errors do a kind of throw. CL has an elaborate error-handler for catching these errors according to classifications, and ways of substituting computations But here is the most primitive…

```
(ignore-errors (/ 0 0))
nil
#<division-by-zero @ #x20e9b6da>
```

Returns 2 values: nil, if an error, and the type of error

Specifically, ignore-errors executes forms in a dynamic environment where a handler for conditions of type error has been established; if invoked, it handles such conditions by returning two values, nil and the condition that was signaled, from the ignore-errors form.

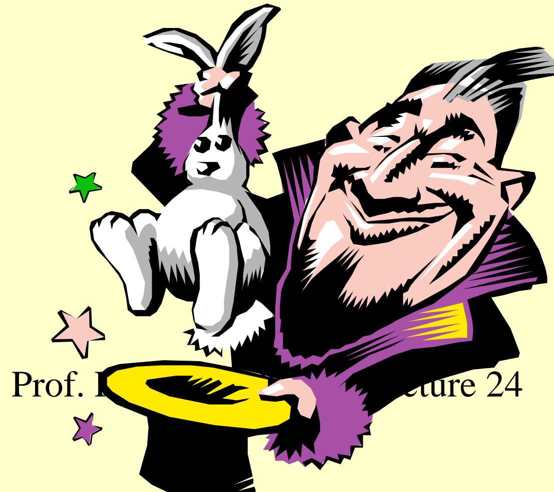
(more refined handling of errors is done by handler-case)

Call with Current Continuation, call/cc is part of Scheme: More powerful

Call/cc is a normal procedure that takes a single argument, say comp. The function comp is itself a procedure of one argument.

(call/cc comp) ;; calls comp and returns what comp returns

The trick is, what is the argument to comp? It is a procedure. Call it cc, which is the current continuation point. If cc is applied to some value Z, that value Z is returned as the value of the call to call/cc



Examples of call/cc in Scheme

Call/cc is a normal procedure that takes a single argument, say comp. The function comp is itself a procedure of one argument.

```
(+ 1 (call/cc (lambda(cc)(+ 20 300))));; doesn't use the cc
  → 321

(+ 1 (call/cc (lambda(cc)(+ 20 (cc 300)))));;  uses cc
  → 301
;; effectively this throws 300 out of the call/cc. Equivalent
;; to
((lambda(val)(+ 1 val))
(catch 'cc ((lambda(v)(+ 20 v))(throw 'cc 300))))

Or  ((lambda(val) (+ 1 val)) 300)
```

Print-table using call/cc in Scheme

```
(define (print-table L)
(call/cc
  (lambda(escape)
   (set! not-a-number escape)  ;;set some global variable
   (map print-sqrt L))))

(define (print-sqrt x)(write (sqrt (must-be-number x))))

(define (must-be-number x)
        (if (numberp x) x (not-a-number 'Huh?))
```

An amazing additional feature of call/cc in Scheme

```
(+ 1 (call/cc (lambda(cc)
   (set! old-cc cc)
    (+ 20 (cc 300)))));; doesn't use the cc, but saves it
  → 301

 (old-cc 500);;  uses cc
  → 501
;; effectively  old-cc returns FOR THE SECOND TIME to the point
;; in the computation where 1 is added, this time returning 501.

;; Can't do this in common lisp. Throw/catch do not have "indefinite
    extent"...
(+ 1 (catch 'tag (+ 20 (throw 'tag 300)))) → 301

(throw 'tag 500) → error
```

An application: “automatic backtracking” or “nondeterministic programming”

You may have seen this in CS61a.

Let amb be the “ambiguous” operator that returns one of its two arguments, chosen at random. Amb must be a special form or macro since it must not evaluate both arguments… (amb x y) =
(if (random-choice) (lambda() x)(lambda() y))

```
(define (integer) (amb 1 (+1 (integer))))  ;; returns a random integer

(define (prime)  ;;return some prime number
  (let ((n (integer)))
    (if (prime? n) (fail))))
```

An application: “automatic backtracking” or “nondeterministic programming”

```
(define backtrack-points nil)

(define (fail)
  (let ((last-choice (car backtrack-points)))
     (set! backtrack-points (cdr backtrack-points))
     (last-choice)))

(define (random-choice f g)
  (if  ( = 1 (random 2) )  ;; (random 2) returns 0 or 1 randomly
             (choose-first f g)
             (choose-first g f)))

(define (choose-first f g)
 (call/cc
  (lambda(k)
   (set! backtrack-points
        (cons (lambda() (k (g))) backtrack-points))
   (f))))
```

To implement Call/CC we first need to talk about continuations

We can write continuation-based programs from first principles, at least in languages like Lisp. (First class functions)

Basic idea: You never have to return a value. You call another function with your answer. This may seem odd but it is both coherent as a theoretical notion, and also useful in compilation!

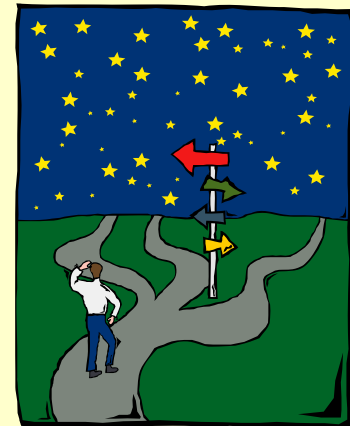
Huh? I'm confused.

Basic idea: You Never Return. You call another function with your answer. We contrast conventional and continuation passing

Consider

(print (+ (* 3 4) 5)) ;; conventional version. Trace + and *

```
0: (* 3 4)              ;; call to *
0: returned 12
0: (+ 12 5)             ;;call to +
0: returned 17
```



17 ;; result of printing. We used the return value (12) along the way

Continuation version of +, *

*Basic idea: You Never Return. You call another function with your answer. Here's a continuation version of + and **

```
(defun cc+ (a b cc)(funcall cc (+ a b)));; compute a+b, call cc on result
(defun cc* (a b cc)(funcall cc (* a b)))

;;Usage
(cc* 3 4 #'print)
12 ;; print 12

(cc* 3 4 #'(lambda(r)(cc+ r 5 #'print)))
17   ;; print 17
```

Continuation version of +, *, traced

```
(cc* 3 4 #'(lambda(r)(cc+ r 5 #'print)))

 0: (cc* 3 4 #<some function>)
   1: (cc+ 12 5 #'print)

17   ;;; THE ANSWER IS PRINTED HERE
   1: returned 17  ;; who cares?  We knew that!!
 0: returned 17   ;; who cares?  No One!
```

Note that we have encapsulated “the place where you added 5 to something.” Seems implausible in (+ (* 3 4) 5), but that’s what we have done in #’(lambda(r)(cc+ …))

A slightly more elaborate example

```
(print (+ (* 3 (+ 2 2)) 5))   ;; 3*(2+2)+5

(cc+ 2 2 #'(lambda(s)(cc* s 3 #'(lambda(r)(cc+ r 5 #'print)))))

 0: (cc+ 2 2 #<function>)  ;; run this in lisp and trace it
   1: (cc* 3 4 #<function>)
     2: (cc+ 12 5 #'print)

17
     2: returned 17
   1: returned 17
 0: returned 17  ;; BORING AND UNNECESSARY TO RETURN
```

Compiling from continuations is obvious

```
(print (+ (* 3 (+ 2 2)) 5))   ;; 3*(2+2)+5

(cc+ 2 2 #'(lambda(s)(cc* s 3 #'(lambda(r)(cc+ r 5 #'print)))))
(pushi 2)(pushi 2)(+)    (pushi 3) (*)            (pushi 5)(+) (print)
```

In fact, this can be a very refreshing way of organizing a compiler or interpreter….

How hard is it to implement an interpreter supporting call/cc?

Two steps:

1. Rewrite the interpreter to use continuations
2. Add a few new functions to deal with call/cc specifically.

What is the flavor of the interpreter with continuations?
Details in Norvig's *Paradigms of AI Programming*

Interpreter with continuations

```
(defun interp(x env cc)
 "Evaluate the expression x in the environment env,
  and pass the result to the continuation cc."
(cond  ((symbolp x)(funcall cc (get-value x env)))
       ((atom x)(funcall cc x))
…
(case (first x)
  (if   (interp  (second x) env
           #'(lambda(pred)(interp (if pred (third x)(fourth x)) env
      cc))))
  (begin  (interp-begin (cdr x) cc)
…
```

Interp-begin with continuation

```
(defun interp-begin (body env cc)
 "Evaluate the body in the environment env,
  and pass the result of the last subexpression to the continuation
      cc."
(interp (first body) env
   #'(lambda(val)
       (if (null (rest body)) (funcall cc val)
                               (interp-begin (cdr body) env cc)))))
```

……….

The main call to the interpreter is then

```
(interp (read) env #'print)
```

Tracing the interpreter supporting call/cc

```
(interp '(begin 3 4) nil #'print) →
(interp-begin  (3 4) nil #'print) →
(interp   3 nil  #'(lambda(val) (if (null '(4))(funcall #'print val)
                                          (interp-begin '(4) env #'print))))
→
  the continuation will be called on 3, but '(4) is not null…
(interp-begin (4) nil #'print)  →
(interp  4 nil #'(lambda(val) (if (null nil) (funcall #'print val) …..)

→
The continuation will be called on 4… and (null nil) ≡ true so
                    (funcall #'print 4)   happens.
```

How about adding call/cc?

Once we have a separate notion of "This is the continuation of the calculation from this point on" .. It is, after all, just the function cc, and we can give it an argument and then continue the computation…. we can preserve it and re-execute it at our leisure!

We can't use a regular stack if we plan to use call/cc

Few people need the generality; though if you have this you can implement virtually any control structure in any other language and quite a few that exist nowhere else.

Even some Schemes do not support it.

Common Lisp does not preserve the control necessary.