

Control

Handout #38

CS251 Lectures 34-35

May 4, 2001

What is Control?

- In program execution, *control* refers to “where” the computation currently is.
- Control is characterized by two components:
 - (1) the expression (or statement) currently being evaluated.
 - CS111: the red control dot.
 - CS240: the program counter.
 - CS251: the argument to `subst-eval` in the substitution model
 - (2) The *continuation* = all the pending operations that need to be performed when the value of the expression currently being evaluated is returned.
 - CS111: the pending execution frames in the Java Execution Mode.
 - CS240: the stack of procedure call activation frames.
 - CS251: the surrounding expressions in the Scheme substitution model

We will call the pair of (1) and (2) a *control point*.

- All computation is an iteration through control points.

Control Point Example 1

`(/ (+ (* 6 5) (- 7 3)) 2)`

<i>Expression</i>	<i>Continuation</i>
<code>(/ (+ (* 6 5) (- 7 3)) 2)</code>	<code>top</code>
<code>(+ (* 6 5) (- 7 3))</code>	<code>((v1) (top (/ v1 2)))</code>
<code>(* 6 5)</code>	<code>((v2) (top (/ (+ v2 (- 7 3)) 2)))</code>
<code>(- 7 3)</code>	<code>((v3) (top (/ (+ 30 v3) 2)))</code>
<code>(+ 30 4)</code>	<code>((v4) (top (/ v4 2)))</code>
<code>(/ 34 2)</code>	<code>top</code>

17

Notes:

- **top** designates the top-level continuation
- The above assumes left-to-right evaluation of arguments (MIT Scheme evaluates them right-to-left.)

Control Point Example 2

(fact 3)

Expression

Continuation

(fact 3)	top
(fact 2)	((v1) (top (* 3 v1)))
(fact 1)	((v2) (top (* 3 (* 2 v2))))
(fact 0)	((v3) (top (* 3 (* 2 (* 1 v3)))))
(* 1 1)	((v4) (top (* 3 (* 2 v4))))
(* 2 1)	((v5) (top (* 3 v5)))
(* 3 2)	top

Control Aspects of Familiar Constructs

- Evaluating nested subexpressions requires choosing an order and remembering what to do next.
 - Argument evaluation order is left-to-right in most language.
 - Evaluation order unspecified in Scheme (right-to-left in MIT Scheme).
- Sequencing of statements in imperative language.
- Conditionals allow branches in control flow.
- Loops/tail recursion specify iterations.
- Function/procedure call and return
 - In many execution models (e.g., C, Pascal, Java), calling a procedure pushes an activation frame on the call stack and returning from a procedure pops the activation from from the call stack.
 - In properly tail-recursive languages (e.g. Scheme, most ML implementations) stack is pushed by subexpression evaluation and procedure calls act like “gotos that pass arguments” (see Guy Steele’s, “*Debunking the Expensive Procedure Call Myth or Lambda: The Ultimate Goto.*”)

Altering the Normal Flow of Control

Sometimes want to “break out” from the normal flow of control in a program:

- Want to immediately stop execution of the program, at user request (typing Control-C) or due to finding an error. E.g. Scheme’s `error`; `halt` opcode in assembly language.
- Discover an answer “early” and want to return it immediately without processing all pending computations. E.g. encountering a zero when finding the product of a list or array.
- Encounter an unusual situation that may need to be handled differently in different contexts. E.g., division by zero, out-of-bounds array access, unbound variables in environment lookup.
- Altering the normal flow of control can be very convenient and efficient, but can also lead to “spaghetti code”. Dijkstra’s “*Goto Considered Harmful*” and the structured programming movement of the 1970s advocated control constructs with one control input and one control output.

Non-local Exits: Return

In C, C++, and Java, return can force “early” exit of a function/method.

Example (Java): calculating array product. Want to return early if encounter a zero. Also suppose that encountering any negative number should cause the result to be -1.

```
public static int arrayProd (int[] a) {  
    int prod = 1;  
    for (int i = 0; i < a.length; i++) {  
        if (a[i] < 0)  
            return 0; // Non-local exit from loop  
        else if (a[i] == 0) then  
            return -1; // Non-local exit from loop  
        else  
            prod = a[i] * prod;  
        }  
    return prod;  
}
```

Non-local Exits: Break

Java has labeled **break** statements for breaking out of a loop.

```
public static int sumArrayProds (int[][] a) {
    int sum = 0;
    outer:for (int i = 0; i < a.length; i++) {
        int prod = 1;
        inner:for (int j = 0; i < a[i].length; j++) {
            if (a[i][j] < 0)
                break outer; // Return current sum on negative num
            else if (a[i][j] == 0) {
                prod = 0; break inner;
                // Alternatively: continue outer;
            }
            else
                prod = a[i][j] * prod;}
        sum = sum + prod;}
    return sum;}
```

- Java's labeled **continue** statement jumps to end of specified loop.
- C's unlabeled **break** and **continue** that work on innermost enclosing loop.

Non-Local Exits: Goto

In Pascal, can only express non-local exits via **goto**:

```
function listProd (outer_lst: intlist): integer;  
  label 17; {labels are denoted by numbers 0 to 9999}  
  function inner (lst: intlist): integer;  
    begin  
      if lst = nil then  
        inner := 1  
      else if lst^.data = 0 then  
        begin  
          product := 0; {Sets return value of function}  
          goto 17; {Control jumps to label 17}  
        end;  
      else  
        inner := lst^.data * inner(lst^.next)  
      end;  
    begin  
      product := inner (outer_lst);  
    17:  
  end;
```

Non-Local Exits: Label and Jump

We will study non-local exits in Scheme by extending it with the following label and jump constructs:

```
(label I E)
```

Evaluates E in a lexical environment in which the name I is bound to a first-class *control point* that represents the continuation of the entire label expression.

```
(jump E1 E2)
```

Returns the value of E2 to the control point that is the value of E1. (jump signals an error if E1 is not a control point.)

Label and Jump: Simple Examples

```
(+ 1 (label exit (* 2 (- 3 (/ 4 1))))))
```

```
(+ 1 (label exit (* 2 (- 3 (/ 4 (jump exit 5))))))
```

```
(+ 1 (label exit  
      (* 2 (- 3 (/ 4 (jump exit (+ 5 (jump exit 6))))))))
```

```
(+ 1 (label exit1  
      (* 2 (label exit2  
            (- 3 (/ 4 (+ (jump exit2 5)  
                          (jump exit1 6))))))))
```

Label and Jump: List Product

```
(define product
  (lambda (outer-list)
    (label return
      (letrec ((inner (lambda (lst)
                        (if (null? lst)
                            1
                            (if (= (car lst) 0)
                                (jump return 0)
                                (* (car lst)
                                   (inner (cdr lst))))))))))
      (inner outer-list))))))
```

Label and Jump: List Product Alternative

```
(define product
  (lambda (outer-list)
    (label return
      (foldr (lambda (x ans)
              (if (= x 0)
                  (jump return 0)
                  (* x ans)))
            1
            outer-list))))
```

Control Points Introduced by `label` are First-Class

```
(define fact
  (lambda (n)
    (let ((loop `later)
          (ans 1))
      (begin
        (label top (set! loop (lambda () (jump top `ignore))))
        (if (= n 0)
            ans
            (begin
              (set! ans (* n ans))
              (set! n (- n 1))
              (loop))))))))
```

First-class Control Points can do Strange and Wondrous Things!

```
(let ((g (lambda (x) x)))
  (letrec ((fact (lambda (n)
                 (if (= n 0)
                     (label base
                        (begin
                          (set! g (lambda (y)
                                    (begin
                                      (set! g (lambda (z) z))
                                      (jump base y))))))
                     1))
            (* n (fact (- n 1)))))))
  (+ (g 10)
     (+ (fact 3) ;Cont. = (lambda (v) (+ 10 (+ v (+ ...))))
        (+ (g 10)
           (+ (fact 4) ;Cont. = (abs (v) (+ 10 (+ 60 (+ 10 (+ v ...))))))
              (g 10)))))))))
```

Scheme's `call-with-current-continuation`

Off-the-shelf Scheme does not support `label` and `jump`. But it does support `call-with-current-continuation`, which can be used to implement most advanced control constructs.

`(call-with-current-continuation Eproc)` behaves like:

```
(let ((Iproc Eproc)) ;; Assume Iproc fresh
  (label here
    (Iproc (lambda (val) (jump here val))))))
```


Exception Handling

Want to be able to “signal” exceptional situations and handle them differently in different contexts.

Many languages provide exception systems:

- Java’s throw and try
- ML’s raise and handle
- Common Lisp’s throw and catch

Raise, trap, and handle

We will study exception handling in a version of Scheme extended with the following constructs:

- `(raise T E)`
Evaluate `E` to value `V` and raise exception with tag `T` and value `V`.
- `(trap T E_handler E_body)`
First evaluate `E_handler` to a one-argument handler function `V_handler`. Then evaluate `E_body` to value `V_body`. If no exception is encountered, return `V_body`. If an exception is raised with tag `T` and value `V_val`, the call to *raise* returns with the value of `(V_handler V_val)` evaluated at the point of the raise.
- `(handle T E_handler E_body)`
First evaluate `E_handler` to a one-argument handler function `V_handler`. Then evaluate `E_body` to value `V_body`. If no exception is encountered, return `V_body`. If an exception is raised with tag `T` and value `V_val`, the call to *handle* returns with the value of `(V_handler V_val)` evaluated at the point of the handle.

Exception Handling Examples

```
(define test
  (lambda ()
    (let ((raiser (lambda (x)
                    (if (< x 0)
                        (raise negative x)
                        (if (even? x)
                            (raise even x)
                            x))))))
      (+ (raiser 1) (+ (raiser -3) (raiser 4))))))
```

What is the value of the following, where `handler_1` and `handler_2` range over `{trap, handle}`? First assume left-to-right evaluation of arguments, then right-to-left.

```
(handler_1 negative (lambda (v) (- v))
  (handler_2 even (lambda (v) (* v v))
    (test)))
```

```
(handler_1 even (lambda (v) (* v v))
  (handler_2 negative (lambda (v) (- v))
    (test)))
```