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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(/ (+ (* 6 5) (- 7 3)) 2)</td>
<td>top</td>
</tr>
<tr>
<td>(+ (* 6 5) (- 7 3))</td>
<td>(λ (v1) (top (/ v1 2)))</td>
</tr>
<tr>
<td>(* 6 5)</td>
<td>(λ (v2) (top (/ (+ v2 (- 7 3)) 2)))</td>
</tr>
<tr>
<td>(- 7 3)</td>
<td>(λ (v3) (top (/ (+ 30 v3) 2)))</td>
</tr>
<tr>
<td>(+ 30 4)</td>
<td>(λ (v1) (top (/ v1 2)))</td>
</tr>
<tr>
<td>(/ 34 2)</td>
<td>top</td>
</tr>
</tbody>
</table>

→ 17

Notes:
- Continuations are modeled as single-argument functions.
- 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: Recursive Factorial

(define (fact-rec n)
  (if (= n 0)
      1
      (* n (fact-rec (- n 1)))))

<table>
<thead>
<tr>
<th>Expression</th>
<th>Continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(fact-rec 3)</td>
<td>top</td>
</tr>
<tr>
<td>(fact-rec 2)</td>
<td>(λ (v1) (top (* 3 v1)))</td>
</tr>
<tr>
<td>(fact-rec 1)</td>
<td>(λ (v2) (top (* 3 (* 2 v2))))</td>
</tr>
<tr>
<td>(fact-rec 0)</td>
<td>(λ (v3) (top (* 3 (* 2 (* 1 v3))))))</td>
</tr>
<tr>
<td>(* 1 1)</td>
<td>(λ (v2) (top (* 3 (* 2 v2))))</td>
</tr>
<tr>
<td>(* 2 1)</td>
<td>(λ (v1) (top (* 3 v1)))</td>
</tr>
<tr>
<td>(* 3 2)</td>
<td>top</td>
</tr>
</tbody>
</table>

→ 6

Note the stack-like nature of continuations.
Control Point Example 3: Iterative Factorial

```
(define (fact-iter n) (fact-tail n 1))

(define (fact-tail num ans)
  (if (= num 0)
    ans
    (fact-tail (- num 1) (* num ans)))))
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(fact-iter 3)</td>
<td>top</td>
</tr>
<tr>
<td>(fact-tail 3 1)</td>
<td>top</td>
</tr>
<tr>
<td>(fact-tail 2 3)</td>
<td>top</td>
</tr>
<tr>
<td>(fact-tail 1 6)</td>
<td>top</td>
</tr>
<tr>
<td>(fact-tail 0 6)</td>
<td>top</td>
</tr>
</tbody>
</table>

→ 6

*Note:* A function call is tail recursive if it does not alter continuation

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, due to request from user (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.

```java
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.

```java
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`:

```pascal
function product (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^.head = 0 then
        begin
            product := 0; {Sets return value of function}
            goto 17; {Control jumps to label 17}
        end;
    else
        inner := lst^.head * inner(lst^.tail)
    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:

\[(\text{label } I \ E)\]

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.

\[(\text{jump } E1 \ E2)\]

Returns the value of \(E2\) to the control point that is the value of \(E1\).

\(\text{jump}\) signals an error if \(E1\) is not a control point.

Label and Jump: Simple Examples

\[(+ 1 (\text{label } \text{exit} (* 2 (- 3 (/ 4 1))))))\]

\[(+ 1 (\text{label } \text{exit} (* 2 (- 3 (/ 4 (\text{jump } \text{exit} 5)))))))\]

\[(+ 1 (\text{label } \text{exit})
\quad (* 2 (- 3 (/ 4 (\text{jump } \text{exit} (+ 5 (\text{jump } \text{exit} 6))))))))\]

\[(+ 1 (\text{label } \text{exit1})
\quad (* 2 (\text{label } \text{exit2})
\quad \quad (* 2 (- 3 (/ 4 (+ (\text{jump } \text{exit2} 5)
\quad \quad (\text{jump } \text{exit1} 6))))))))))\]
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) ; don’t care about initial value
           (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))))))

Example of call-with-current-continuation

(define product
  (lambda (outer-list)
    (call-with-current-continuation
      (lambda (return)
        (letrec
          ((inner (lambda (lst)
                     (cond ((null? lst) 1)
                           ((= 0 (car lst)) (return 0))
                           (else (* (car lst)
                                    (inner (cdr lst))))
                       ))))
          (inner outer-list))))))
Continuation Passing Style (CPS)

The constructs we have seen so far rely on *implicit* continuations. It is possible to model non-local control flow by passing *explicit* continuations in a style known as *continuation-passing style*.

For example, here is a CPS version of recursive factorial:

```
(define fact-rec-cps
  (lambda (n k) ; k is the explicit continuation
    (if (= n 0)
      (k 1)
      (fact-rec-cps (- n 1)
        (lambda (v) (k (* n v)))))))

(fact-rec-cps 3 (lambda (v) v))
(fact-rec-cps 4 (lambda (v) (+ 1 (* 2 v))))
```

CPS version of *product*

```
(define product
  (lambda (outer-list)
    (letrec ((inner
      (lambda (lst k) ; k is the explicit cont.
        (if (null? lst)
          (k 1)
          (if (= (car lst) 0)
            0 ; return 0 directly,
            ; thus punting continuation
            (inner (cdr lst)
              (lambda (v)
                (k (* (car lst) v)))))))))
      (inner outer-list (lambda (v) v))))
```
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/catch`
• 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 argument evaluation, 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)))