What is Control?

In program execution, control refers to the computation currently in progress. Control is characterized by two components:

1. the expression/statement currently being evaluated:
   - CS111: the red control dot.
   - CS240: the program counter.
   - CS251: the argument to `eval` in the substitution model.

2. The **continuation** = all pending operations to be performed when the value of current expression is returned:
   - CS111: the pending frames in the Java Execution Model.
   - CS240: the stack of procedure call activation frames.
   - CS251: the context surrounding the current expression in the 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

**Expression**

\[
\left( \frac{(+ (* 6 5) (- 7 3))}{2} \right)
\]

\[
\Rightarrow (+ (* 6 5) (- 7 3))
\]

\[
\Rightarrow (* 6 5)
\]

\[
\Rightarrow (- 7 3)
\]

\[
\Rightarrow (+ 30 4)
\]

\[
\Rightarrow (/ 34 2)
\]

\[
\Rightarrow 17
\]

**Continuation**

\[
k_{top}
\]

\[
k_1 = (\lambda (v_1) (k_{top} (/ v_1 2)))
\]

\[
k_2 = (\lambda (v_2) (k_1 (+ v_2 (- 7 3))))
\]

\[
k_3 = (\lambda (v_3) (k_1 (+ 30 v_3)))
\]

\[
k_1
\]

\[
k_{top}
\]

**Notes:**

- Continuations are modeled as single-argument functions.
- \(k_{top}\) 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

\[
\text{(define (fact-rec n)}
\begin{align*}
& \text{(if (= n 0)} \\
& \quad 1 \\
& \quad (* n (fact-rec (- n 1))))
\end{align*}
\]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Rightarrow (\text{fact-rec 3}) )</td>
<td>( k_{top} )</td>
</tr>
<tr>
<td>( \Rightarrow (\text{fact-rec 2}) )</td>
<td>( k_1 = (\lambda (v_1) (k_{top} (* 3 v_1))) )</td>
</tr>
<tr>
<td>( \Rightarrow (\text{fact-rec 1}) )</td>
<td>( k_2 = (\lambda (v_2) (k_1 (* 2 v_2))) )</td>
</tr>
<tr>
<td>( \Rightarrow (\text{fact-rec 0}) )</td>
<td>( k_3 = (\lambda (v_3) (k_2 (* 1 v_3))) )</td>
</tr>
<tr>
<td>( \Rightarrow (* 1 1) )</td>
<td>( k_2 )</td>
</tr>
<tr>
<td>( \Rightarrow (* 2 1) )</td>
<td>( k_1 )</td>
</tr>
<tr>
<td>( \Rightarrow (* 3 2) )</td>
<td>( k_{top} )</td>
</tr>
<tr>
<td>( \Rightarrow 6 )</td>
<td></td>
</tr>
</tbody>
</table>
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>( k_{top} )</td>
</tr>
<tr>
<td>⇒ (fact-tail 3 1)</td>
<td>( k_{top} )</td>
</tr>
<tr>
<td>⇒ (fact-tail 2 3)</td>
<td>( k_{top} )</td>
</tr>
<tr>
<td>⇒ (fact-tail 1 6)</td>
<td>( k_{top} )</td>
</tr>
<tr>
<td>⇒ (fact-tail 0 6)</td>
<td>( k_{top} )</td>
</tr>
<tr>
<td>⇒ 6</td>
<td></td>
</tr>
</tbody>
</table>

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 *The Expensive Procedure Call Myth or Lambda: The Ultimate Goto*).
Altering the Normal Flow of Control

Sometimes want to "break out" out from the normal flow of control:

• Want to immediately stop execution of the program, due to request from user (typing Control-C) or encountering an error. E.g. halt opcode in assembly language; error in HOFL, Scheme;

• Discover an answer and want to return it immediately without processing all pending computations. E.g. encountering a zero when finding the product of elements in a list, array, or tree.

• 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 normal flow of control can be very convenient and efficient, but can 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:** \texttt{return}

In C, C++, and Java, \texttt{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)
            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` 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 of program}
end;
```
Non-Local Exits: \texttt{label} and \texttt{jump}

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

\begin{itemize}
  \item \texttt{(label \hspace{1em} I_{cp} \hspace{1em} E_{body})} evaluates \( E_{body} \) in a lexical environment in which the name \( I_{cp} \) is bound to a first-class control point that represents the continuation of the entire label expression. \texttt{label} returns the value of \( E_{body} \) unless \texttt{jump} is called on \( I_{cp} \), in which case the value supplied to \texttt{jump} is returned.
  \item \texttt{(jump \hspace{1em} E_{cp} \hspace{1em} E_{val})} returns the value of \( E_{val} \) to the control point that is the value of \( E_{cp} \). \texttt{jump} signals an error if \( E_{cp} \) is not a control point.
\end{itemize}
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
       (* 2 (- 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)
                                      (* (car lst)
                                         (inner (cdr lst))))))))
            (inner outer-list)))))))
label and jump: List Product Alternative

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

Unlike the previous version, a jump is performed here on the way out of the recursion rather than on the way in.
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 are Strange and Powerful

(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 1)
    (+ (fact 3) ; Cont. = (\(v\) (+ 1 (+ v ...)))
      (+ (g 10)
        (+ (fact 4) ; Cont. = (\(v\) (+ 1 (+ 60 (+ 10 (+ v ...)))))
          (g 100)))))))
Scheme’s call-with-current-continuation

Off-the-shelf Scheme does not support label and jump. But it does support call-with-current-continuation (sometimes abbreviated cwcc) which encapsulates both label and jump and can be used to implement many advanced control constructs.

(call-with-current-continuation \(E_{\text{proc}}\)) behaves like:

\[
\begin{align*}
\text{let ((body-proc } & \! E_{\text{proc}}) \\
\text{label return} & \\
\text{body-proc} & \! (\lambda (val) \\
\text{jump return} & \! \text{val})
\end{align*}
\]
Example of \texttt{call-with-current-continuation}

\begin{verbatim}
(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))))))
\end{verbatim}
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 (CPS).

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`
- OCaml’s `raise` and `try/with`
- Common Lisp’s `throw` and `catch`
raise, handle, and trap

We study exception handling in Scheme extended with:

- \((\text{raise } I_{tag} \; E)\) Evaluate \(E\) to value \(V\) and raise exception with tag \(I_{tag}\) and value \(V\).

- \((\text{handle } I_{tag} \; E_{\text{handler}} \; E_{\text{body}})\) First evaluate \(E_{\text{handler}}\) to a one-argument handler function \(V_{\text{handler}}\). Then evaluate \(E_{\text{body}}\) to value \(V_{\text{body}}\). If no exception is encountered, return \(V_{\text{body}}\). If an exception is raised with tag \(I_{tag}\) and value \(V_{\text{body}}\), the call to handle returns with the value of the application \((V_{\text{handler}} \; V_{\text{body}})\) evaluated at the point of the handle (termination semantics).

- \((\text{trap } I_{tag} \; E_{\text{handler}} \; E_{\text{body}})\) is evaluated like \((\text{handle } I_{tag} \; E_{\text{handler}} \; E_{\text{body}})\) except that if an exception is raised with tag \(I_{tag}\) and value \(V_{\text{body}}\), the call to raise returns with the value of the application \((V_{\text{handler}} \; V_{\text{body}})\) evaluated at the point of the raise (resumption semantics).

\(\text{handle/trap}\) effectively bind \(V_{\text{handler}}\) in a dynamically scoped exception handler namespace, and \((\text{raise } I_{tag} \; E)\) looks up \(I_{tag}\) in this namespace.
Exception Handling Examples 1

(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 \{\text{handle, trap}\}? 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)))
Exception Handling Examples 2

What are the value of the following expressions, where handler ranges over \{\text{handle, trap}\}?

; Expression 1
(handler a (lambda (x) (+ 4000 x))
  (handler b (lambda (x) (+ 300 (raise a (+ x 4))))
    (handler a (lambda (x) (+ 20 x))
      (+ 1 (raise b 2))))))

; Expression 2
(handler c (lambda (x) (* x 10))
  (+ 1 (raise c (+ 2 (raise c 4)))))
Exception Handling In OCaml

OCaml’s `raise` and `try/with` uses termination semantics. In `raise E`, `E` must evaluate to an exception packet created by an exception constructor (where exceptions are effectively an extensible datatype).

try \( E_{\text{body}} \) with \( \text{clauses} \) evaluates \( E_{\text{body}} \) and returns its value unless an exception is raised, in which case the matching clause in \( \text{clauses} \) is evaluated and its value is returned as the value of `try`. 
OCaml Exception Example

exception Neg of int
exception Even of int

let raiser x =
  if x < 0 then
    raise (Neg x)
  else if (x mod 2) = 0 then
    raise (Even x)
  else
    x

let test () = (raiser 1) + (raiser -3) + (raiser 4)

let innerTest () = try test() with
  Neg y -> raiser(7 + -y)
  | Even z -> 3 * z

let outerTest () = try innerTest() with
  Neg y -> -y
  | Even z -> z * z

Can translate this example into Java using throw and try/catch.
Implementing \textit{raise}

\[(\text{raise } I_{\text{tag}} \ E) \sim (\text{raise-tag } 'I_{\text{tag}} \ E)\]

\[
\text{(define raise-tag}
\begin{array}{l}
\quad \text{(lambda (tag value)}
\quad \begin{array}{l}
\quad \text{(let ((handler}
\quad \begin{array}{l}
\quad \begin{array}{l}
\quad \text{;; Look up handler in current handler env.}
\quad \text{;; Handlers are dynamically scoped!}
\quad \text{(env-lookup tag (get-handler-env)))}
\quad \text{(if (unbound? handler)}
\quad \begin{array}{l}
\quad \text{(error (string-append "Unhandled exception "}
\quad \begin{array}{l}
\quad \text{(symbol->string tag)}
\quad \text{"\: ")})}
\quad \text{(handler value)))})
\quad \text{(lambda (tag value)}
\quad \begin{array}{l}
\quad \text{(let ((handler}
\quad \begin{array}{l}
\quad \text{;; Look up handler in current handler env.}
\quad \text{;; Handlers are dynamically scoped!}
\quad \text{(env-lookup tag (get-handler-env)))}
\quad \text{(if (unbound? handler)}
\quad \begin{array}{l}
\quad \text{(error (string-append "Unhandled exception "}
\quad \begin{array}{l}
\quad \text{(symbol->string tag)}
\quad \text{"\: ")})}
\quad \text{(handler value)))})
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
}
(define with-handler
  (lambda (tag make-handler try-thunk)
    (begin
      (let ((old-env (get-handler-env)))
        (begin
          ;; Remember handler in dynamic environment
          (set-handler-env! (env-bind tag
                              (make-handler old-env)
                              (get-handler-env)))
          ;; Evaluate try-thunk
          (let ((try-value (try-thunk)))
            ;; In normal case, pop handler
            (begin
              (set-handler-env! old-env) ; reinstate old handler env.
              (try-value))))))))) ;; Return value
Implementing `handle` and `trap` 2

(trap tag handler body) desugars to

(let ((*handler* handler) ; only evaluate once
    (*thunk* (lambda () body))) ; avoid capturing *handler*
  (with-handler 'tag
    (lambda (old-env)
      (lambda (value) (*handler* value))) ; ignores old-env
    *thunk*))

(handle tag handler body) desugars to

(let ((*handler* handler) ; only evaluate once
    (*thunk* (lambda () body))) ; avoid capturing *handler*
  (call-with-current-continuation
   (lambda (handle-cont)
     (with-handler 'tag
       (lambda (old-env)
         (lambda (value)
           ;; Invoking HANDLE-CONT returns directly to
           ;; appropriate handle, ignoring current continuation.
           (begin
             (set-handler-env! old-env) ; reinstall old-env
             (handle-cont (*handler* value))))))
         *thunk*))))