Control
Our Languages So Far
Our Languages So Far
What We Sometimes Need
What We Sometimes Need

• Escaping because of an error (exceptions)
• Escaping because we found the answer (early return)
• Revisiting an earlier decision we made (backtracking)
• Alternating between different computations (coroutines)

• These are all forms of control operations
  o i.e., of deviating from the normal control flow of our program
Control

- Control is all about deciding what to execute next
- May not be what directly follows in the program!

- **Our strategy:** make "what to execute next" explicit in our interpreter
  - Then implementing control operators is just a matter of messing with that
Continuation-passing style

**Key idea:** convert the interpreter into a style where all remaining work is explicit as an argument to the interpreter

Kind of like what we did with `interp2` when we implemented state using a store

Then we can swap in and out different pieces of work as we decide what we want to run!
Continuation-passing style

We will transform our interpreter from:

\[
\text{interp} : (\rightarrow \text{FAE DefSub TRFAE-Value})
\]

into a function with this type:

\[
(\rightarrow \text{FAE DefSub (FAE-Value} \rightarrow \alpha) \alpha)
\]

If we also have a store as a result, where does it go?

\[
\text{interp} : (\rightarrow \text{BFAE DefSub DefSub Store (BFAE-Value*Store} \rightarrow \alpha) \alpha)
\]

(But we won’t worry about stores for now.)
What follows in the FAE interpreter, transformed in continuation-passing style. Each future step of computation is explicitly packaged up into a more complex $k$ argument to be supplied to the next call to interp.
(define-type FAE
  [num (n number?)])
  [add (lhs FAE?)
    (rhs FAE?)]
  [sub (lhs FAE?)
    (rhs FAE?)]
  [id (name symbol?)]
  [fun (param-name symbol?)
    (body FAE?)]
  [app (fun-expr FAE?)
    (arg-expr FAE?)])
(define-type FAE-Value
  [numV (n number?)])
[closureV (param-name symbol?)
  (body FAE?)
  (ds DefSub?)])

(define-type DefSub
  [mtSub]
  [aSub (name symbol?)
    (value FAE-Value?)
    (rest DefSub?)])
(define (interp-expr a-fae)
  (interp a-fae (mtSub)
    (λ (x) x)))
; FAE? DefSub? (FAE-Value? -> any) -> any
(define (interp a-fae ds k)
  (type-case FAE a-fae
    [num (n) (k (numV n))]
    [add (l r) (numop + l r ds k)]
    [sub (l r) (numop - l r ds k)]
    [id (name) (k (lookup name ds))]
    [fun (param-name body)
      (k (closureV param-name body ds))]
    [app (fun-expr arg-expr)
      the next slide contains this case])))
... 

[app (fun-expr arg-expr)
  (interp fun-expr ds
   (λ (fun-val)
     (interp arg-expr ds
      (λ (arg-val)
       (interp
        (closureV-body fun-val)
        (aSub (closureV-param-name fun-val) arg-val
         (closureV-ds fun-val))
       k)))))]
(define (numop f l r ds k)
  (interp l ds
    (λ (l-v)
      (interp r ds
        (λ (r-v)
          (k (numV
            (f (numV-n l-v)
              (numV-n r-v))))))))))
(define (lookup name ds)
  (type-case DefSub ds
    [mtSub () (error 'lookup "free variable")]
    [aSub (n num rest)
      (if (symbol=? n name)
        num
        (lookup name rest))])))
Let’s add early return to our language!

(define-type KFAE
  [num (n number?)]
  [add (lhs KFAE?)
       (rhs KFAE?)]
  [sub (lhs KFAE?)
       (rhs KFAE?)]
  [id (name symbol?)]
  [fun (param-name symbol?)
       (body KFAE?)]
  [app (fun-expr KFAE?)
       (arg-expr KFAE?)]
  [ret (ret-expr KFAE?)]
)
Return

\[
\begin{align*}
\{\{\text{fun} \ \{x\} \ \{+ \ x \ \{\text{ret} \ 2\}\}\}\} \\
5 & \Rightarrow 2
\end{align*}
\]

\[
\begin{align*}
\{\{\text{fun} \ \{x\} \ \{+ \ x \ \{\text{ret} \ \{+ \ 4 \ \{\text{ret} \ 2\}\}\}\}\}\} \\
5 & \\
& \Rightarrow 2
\end{align*}
\]

\[
\begin{align*}
\{+ \ \{\{\text{fun} \ \{x\} \ \{+ \ x \ \{\text{ret} \ 2\}\}\}\} \\
5 & \\
\} \{3\} \\
& \Rightarrow 5
\end{align*}
\]

\[
\{\text{ret} \ 2\} \Rightarrow \text{error: not inside a function}
\]
Return

\[
\ldots
[\text{ret}\ (\text{ret-expr})
\quad (\text{interp}\ \text{ret-expr}\ \text{ds}
\quad (\lambda\ (x)\ x))]
\]

• We don’t *have* to call our continuation.

• If we ignore it, we skip its work!
Return

\[
{\{+ \{\{\text{fun} \ {x} \ {+ \ x \ {\text{ret} \ 2}\}}\}\}}
\]
\[
5\}
\]
\[
3\}
\]
\[\Rightarrow 2\]

• Oops, we return too far!

• All the way to the beginning, in fact!

• Solution: two continuations! One for normal execution, one for returning!
(define (interp-expr a-kfae)
  (interp a-kfae (mtSub)
    (λ (x) x)
    (λ (x)
      (error 'interp
        "not inside a function"))))
; KFAE? DefSub?
; (KFAE-Value? -> any) (KFAE-Value? -> any)
; -> any
(define (interp a-kfae ds k ret-k)
  (type-case KFAE a-kfae
    [num (n) (k (numV n))]
    [add (l r) (numop + l r ds k ret-k)]
    [sub (l r) (numop - l r ds k ret-k)]
    [id (name) (k (lookup name ds))]
    [fun (param-name body)
      (k (closureV param-name body ds))]
  ...
))
\[
[\text{app} \ (\text{fun-expr} \ \text{arg-expr}) \\
(\text{interp} \ \text{fun-expr} \ \text{ds} \\
\ (\lambda \ (\text{fun-val}) \\
\ (\text{interp} \ \text{arg-expr} \ \text{ds} \\
\ (\lambda \ (\text{arg-val}) \\
\ (\lambda \ (\text{arg-val}) \\
(\text{interp} \ \\
(\text{closureV-body} \ \text{fun-val}) \\
(\text{aSub} \ (\text{closureV-param-name} \ \text{fun-val}) \\
\text{arg-val} \\
(\text{closureV-ds} \ \text{fun-val})) \\
\text{k} \\
; \text{early return is the same!} \\
\text{k})) \\
\text{ret-k})) \\
\text{ret-k}]]
\]
...[ret (ret-expr)
  (interp ret-expr ds
    ; when you're done, return!
  ret-k
  ; in case a subexpression returns
  ret-k)]