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 computations (coroutines)

- These are all forms of ***control*** operations
  - I.e., of deviating from the normal control flow of our program
  - Control is all about messing with ***what happens next***

Control Example

```
int main () {
    int z = do_stuff(...);
    process(z);
}

int do_stuff (int x) {
    if (is_prime(x)) {
        x = x + 1;
    }

    y = x * 15;
    printf("y is %d", y);
    return y;
}
```

You are here.
What happens next?

Control Example

```
int main () {
    int z = do_stuff(...);
    process(z);
}

int do_stuff (int x) {
    if (is_prime(x)) {
        x = x + 1;
    }

    y = x * 15;
    printf("y is %d", y);
    return y;
}
```

You are here.
What happens next?

This. Comes after the `if`.

Control Example

```
int main () {
    int z = do_stuff(...);
    process(z);
}

int do_stuff (int x) {
    if (is_prime(x)) {
        x = x + 1;
        return x;
    }
    y = x * 15;
    printf("y is %d", y);
    return y;
}
```

You are here.
What happens next?

Control Example

```
int main () {
    int z = do_stuff(...);
    process(z);
}

int do_stuff (int x) {
    if (is_prime(x)) {
        x = x + 1;
        return x;
    }
    y = x * 15;
    printf("y is %d", y);
    return y;
}
```

⬅ This. We returned early.

⬅ You are here.
What happens next?

⬅ And we just ignored this part.

KFAE

Let's add (two flavors of) early return to FAE!

```
<KFAE>  ::=  <num>
          |  {+ <KFAE> <KFAE>}
          |  {- <KFAE> <KFAE>}
          |  <id>
          |  {fun {<id>} <KFAE>}
          |  {<KFAE> <KFAE>}
          |  {ret-0}
          |  {ret <KFAE>}
```

```
{+ {{fun {x} {+ x {ret 10}}}
    5}
   3} ⇒ 13
```

KFAE

```
{{fun {x} {+ x {ret-0}}} 5} ⇒ 0

{+ {{fun {x} {+ x {ret-0}}} 5}
   3}
⇒ 3

{{fun {x} {+ x {ret 2}}} 5} ⇒ 2

{+ {{fun {x} {+ x {ret 10}}} 5}
   3}
⇒ 13

{ret 2} ⇒ error: not inside a function
```

KFAE

```
(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-0] ;                    just return 0
  [ret (ret-expr KFAE?)]) ; return any value
```

KFAE, Parlor Trick Version

Just like we did with state, let's start by implementing object-language control with meta-language control.

Just to get used to programming with control a bit.

Specifically, we'll use PLAI exceptions to implement KFAE early return.

- ***Key idea:*** when we return (early or not), we go back to the call site
    - I.e., we are done interpreting the body of the function

KFAE, Parlor Trick Version

Raise an exception to escape out of where we are.

Use the return value as the value of the exception.

PLAI **`raise`** = **`throw`** in some other languages.

```
; interp : KFAE? DefSub? -> KFAE-Value?
(define (interp an-fae ds)
  (type-case KFAE an-fae
    ...
    [ret-0 ()
           (raise (numV 0))]
    ...))
```

KFAE, Parlor Trick Version

Raise an exception to escape out of where we are.

Use the return value as the value of the exception.

PLAI **`raise`** = **`throw`** in some other languages.

```
; interp : KFAE? DefSub? -> KFAE-Value?
(define (interp an-fae ds)
  (type-case KFAE an-fae
    ...
    [ret-0 ()
           (raise (numV 0))]
    [ret (ret-expr)
         (raise (interp ret-expr ds))]))
```

KFAE, Parlor Trick Version

`with-handlers` = functional variant of **`try-catch`**.

Run the body. If an exception is raised, run the appropiate handler.

The result of **`with-handlers`** is the result of the body (if no exception) or the result of the handler (if exception).

```
[app (fun-expr arg-expr)
     (type-case KFAE-Value (interp fun-expr ds)
       [closureV (param-name body closed-ds)
                 ; outside the with-handlers!
                 ; we're not inside the function body yet
                 (define arg-val (interp arg-expr ds))
                 (with-handlers ([; if you catch a KFAE-Value...
                                  KFAE-Value?
                                  ; ...use this handler
                                  (lambda (v) v)])
                   (interp body (aSub param-name
                                      arg-val
                                      closed-ds)))])]
```

KFAE, Parlor Trick Version

To return, we need to be inside a function body.

So until we enter a function body, returning is an error.

```
(define (interp-expr a-kfae)
  (with-handlers
    ([KFAE-Value?
      (lambda (v)
        (error 'interp "not inside a function"))])
    (interp a-kfae (mtSub))))
```

KFAE, For Real

Ok, cool. But how do we *really* implement control?

- We'll use ***continuation-passing style*** (CPS).

- ***Key idea:*** split up the work we do *right now* and the work we do *next*.
  - And represent the latter as an explicit value: the ***continuation***
  - Pass it around as an argument, like we do with deferred substitutions and stores
    - That's what the `finish` argument to `interp2` was!
  - To change what we do next, change the continuation!

- ***Analogy:*** Store : the heap :: continuation : the stack!
  - Generalized stack: don't just push when we call a function; push any work we keep for later

Continuations

- Typically represented as functions: calling a continuation = doing the work we saved for later (c.f., callbacks in JavaScript)
- We'll use explicit data structures instead (c.f. promises in JS).

```
; one kind of "stack frame" per kind of work we leave for later
(define-type Cont
  [done]
  [numop-do-right   (rhs KFAE?)
                    (ds  DefSub?)
                    (op (-> number? number? number?))
                    ; link each "next step" to the next
                    (rest-k Cont?)]
  [numop-do-op      (l-val KFAE-Value?)
                    (op (-> number? number? number?))
                    (rest-k Cont?)]
  [app-do-arg       (arg-expr KFAE?)
                    (ds  DefSub?)
                    (rest-k Cont?)]
  [app-do-body      (fun-val KFAE-Value?)
                    (rest-k Cont?)]
  [app-do-return    (rest-k Cont?)])
```

Dramatis Personae

- **`interp`** takes an additional continuation argument

```
; KFAE? DefSub? Cont? -> KFAE-Value?
(define (interp a-kfae ds k)
  ...)
```

- **`interp-expr`** gets the ball rolling
  - After interpreting the whole program, nothing to do next

```
; KFAE? -> KFAE-Value?
(define (interp-expr a-kfae)
  (interp a-kfae (mtSub) (done)))
```

- **`interp-cont`** "pops the stack", does the next computation
  - Takes the result of the previous computation as argument

```
; KFAE-Value? Cont? -> KFAE-Value?
(define (interp-cont v k)
  ...)
```

Interp

Base cases. The only work we do is producing a value.

Our part is done, so we move on to the continuation.

```
; KFAE? DefSub? Cont? -> KFAE-Value?
(define (interp a-kfae ds k)
  (type-case KFAE a-kfae
    [num (n) (interp-cont (numV n) k)]
    [id  (name) (interp-cont (lookup name ds) k)]
    [fun (param-name body)
         (interp-cont (closureV param-name
                                body
                                ds)
                      k)]
    ...))
```

Interp

Numeric operations are really three steps:

- Interpreting the first operand
- Interpreting the second operand
- Doing the numeric operation

In CPS, we do first step right now, leave rest for later

- I.e., create a continuation to do step 2

```
[add (l r)
     (interp l ds
             ; new continuation!
             (numop-do-right r ds + k))]
[sub (l r)
     (interp l ds
             (numop-do-right r ds - k))]
```

Interp-Cont

When we get around to doing that next step...

```
; KFAE-Value? Cont? -> KFAE-Value?
(define (interp-cont v k)
  (type-case Cont k
    [numop-do-right (rhs ds op rest-k)
      ; Step 2: interpret the right hand side...
      (interp rhs ds
              ; ...then do step 3 later
              (numop-do-op v op rest-k))]
    ...))
```

Interp-Cont

Then finally step 3

```
; KFAE-Value? Cont? -> KFAE-Value?
(define (interp-cont v k)
  (type-case Cont k
    ...
    [numop-do-op (l-val op rest-k)
      (unless (numV? l-val)
        (error 'interp "expected number"))
      (unless (numV? v)
        (error 'interp "expected number"))
      (interp-cont (numV (op (numV-n l-val)
                             (numV-n v)))
                   ; we're done with our work
                   ; continue with the original continuation
                   rest-k)]
    ...))
```

Interp

Function application is four steps:

- Interpret the function position
- Interpret the argument position
- Interpret the function body
- Return (was implicit before)

```
(define (interp a-kfae ds k)
  (type-case KFAE a-kfae
    ...
    [app (fun-expr arg-expr)
         (interp fun-expr ds ; step 1
                 (app-do-arg arg-expr ds k))]))
```

Interp-Cont

```
(define (interp-cont v k)
  (type-case Cont k
    ...
    [app-do-arg (arg-expr ds rest-k)
       (interp arg-expr ds ; step 2: argument
               (app-do-body v rest-k))]
    [app-do-body (fun-val rest-k)
       ; step 3: function body
       (type-case KFAE-Value fun-val
         [closureV (param-name body ds)
                   (interp body (aSub param-name v ds)
                           (app-do-return rest-k))]
         [else
          (error 'interp "expected function")])]
    ; step 4: return (nothing more to do)
    [app-do-return (rest-k) (interp-cont v rest-k)]
    ...))
```

Interp-Cont

And when we're done, just return the result

```
(define (interp-cont v k)
  (type-case Cont k
    ...
    [done () v]))
```

Whither early return?

Returning early = skipping all work until the end of the app

Skipping work = dropping it from the continuation!

```
(define (interp a-kfae ds k)
  (type-case KFAE a-kfae
    ...
    [ret-0 () (return (numV 0) k)]))

(define (return v k)
  (type-case Cont k
    ; not the end of an app; skip
    [numop-do-right (rhs ds op rest-k) (return v rest-k)]
    [numop-do-op (l-val op rest-k)     (return v rest-k)]
    [app-do-arg (arg-expr ds rest-k)   (return v rest-k)]
    [app-do-body (fun-val rest-k)      (return v rest-k)]
    ; *this* is the end of an app; back to work!
    [app-do-return (rest-k)            (interp-cont v rest-k)]
    ; we tried to return, but did not find an app
    [done () (error 'interp "not inside a function")]))
```

Returning arbitrary values

Need to interpret return value, ***then*** return

- then = build up the continuation!

```
(define-type Cont
  ...
  [do-early-return (rest-k Cont?)])

(define (interp a-kfae ds k)
  (type-case KFAE a-kfae
    [ret (ret-expr)
         (interp ret-expr ds
                 (do-early-return k))]))

(define (interp-cont v k)
  (type-case Cont k
    ...
    [do-early-return (rest-k) (return v rest-k)]))
```

And just skip these **`do-early-return`** frames when returning

Ret within Ret

`ret` is an expression

So can have `ret` inside `ret`!

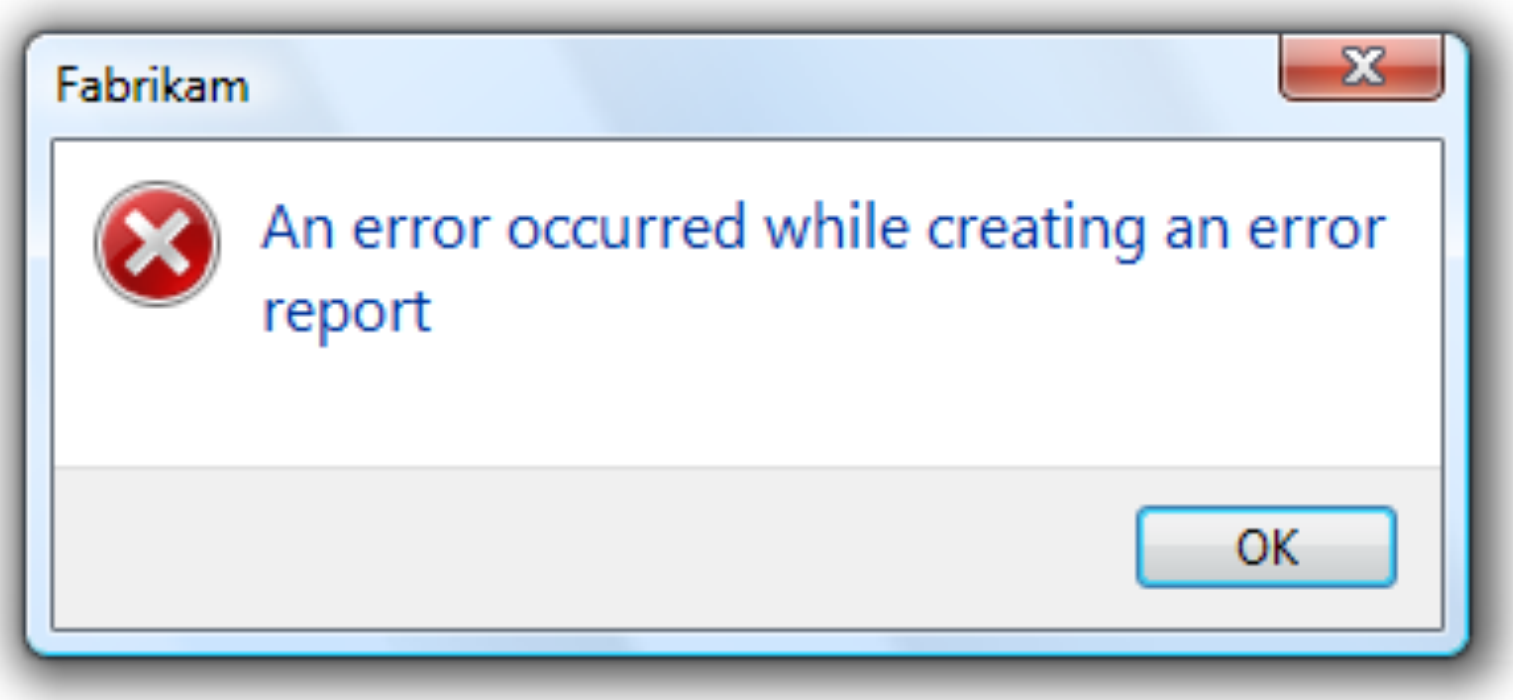
```
{{fun {x} {ret {ret 2}}}
 5} ⇒ 2

{{fun {x} {+ x {ret {+ 4 {ret 2}}}}}
 5}
⇒ 2
```

That's a bit weird, but it follows naturally from our rules.

This kind of behavior makes sense for, e.g., exceptions.

Exception within Exception



Source: https://docs.microsoft.com/en-us/windows/desktop/uxguide/mess-error

Continuation-Passing Style, with Functions

- Control is probably the trickiest concept we'll see
- So let's see a ***third*** way to implement it!

"Classic" CPS uses functions as continuations:

```
interp : KFAE? DefSub?
         (KFAE-Value? -> KFAE-Value?)
         -> KFAE-Value?
```

That's closer to what we did with **`interp2`**

The function represents "the work that is left to do"

To do that work: we call the function

To change that work: we use a different function

interp-expr

After we're done interpreting the whole program, nothing more to do!

```
; KFAE? -> KFAE-Value?
(define (interp-expr a-kfae)
  (interp a-fae (mtSub)
          (λ (x) x)))
```

Base cases. Our work is done, move on to continuation (by calling it!).

```
; interp : KFAE? DefSub?
;          (KFAE-Value? -> KFAE-Value?)
;          -> KFAE-Value?
(define (interp a-fae ds k)
  (type-case FAE a-fae
    [num (n) (k (numV n))]
    [id (name) (k (lookup name ds))]
    [fun (param-name body)
         (k (closureV param-name body ds))]
    ...))
```

Numeric operations are 3 steps: interp left, interp right, then do the operation.

```
(define (numop op l r ds k)
  ; step 1
  (interp l ds
          (λ (l-v)
            ; step 2
            (interp r ds
                    (λ (r-v)
                      ; step 3
                      (k (numV
                          (op (numV-n l-v)
                              (numV-n r-v)))))))))
```

Application is 4 steps: interp fun, interp arg, interp body, return

```
[app (fun-expr arg-expr)
     ; step 1
     (interp fun-expr ds
        (λ (fun-val)
          ; step 2
          (interp arg-expr ds
             (λ (arg-val)
               ; step 3
               (interp
                (closureV-body fun-val)
                (aSub (closureV-param-name fun-val)
                      arg-val
                      (closureV-ds fun-val))
                ; step 4; nothing to do
                (lambda (result)
                  (k result)))))))]
```

ret-0, take 1

Can't call **k** since that would not skip any work

```
[ret-0 () (numV 0)]
```

so let's not do that!

```
{+ {{fun {x} {+ x {ret-0}}}
    5}
   3}
⇒ 0
```

Oops, we return too far!

All the way to the beginning, in fact.

Makes sense: we drop **all** our continuation

Two Continuations

Solution: when we interpret, we can either return, or continue as normal.

Two possible things to run = two continuations!

```
; interp : KFAE? DefSub?
;          (KFAE-Value? -> KFAE-Value?)
;          (KFAE-Value? -> KFAE-Value?)
;          -> KFAE-Value?
```

Two Continuations

Returning is only valid inside a function.

If we try to call the return continuation outside of a function, error.

```
(define (interp-expr a-kfae)
  (interp a-kfae (mtSub)
          (λ (x) x)
          (λ (x)
            (error 'interp
                   "not inside a function"))))

; interp : KFAE? DefSub?
;          (KFAE-Value? -> KFAE-Value?)
;          (KFAE-Value? -> KFAE-Value?)
;          -> KFAE-Value?
```

Two Continuations

For the base cases, we don't return; use the same continuation as before.

```
; interp : KFAE? DefSub?
;          (KFAE-Value? -> KFAE-Value?)
;          (KFAE-Value? -> KFAE-Value?)
;          -> KFAE-Value?
(define (interp a-fae ds k ret-k)
  (type-case FAE a-fae
    [num (n) (k (numV n))]
    [id (name) (k (lookup name ds))]
    [fun (param-name body)
         (k (closureV param-name body ds))]
    ...))
```

Two Continuations

Enter a function body ⇒ change what return means!

Now it means: return from ***this*** function body

```
[app (fun-expr arg-expr)
     (interp fun-expr ds ; step 1
      (λ (fun-val)
        (interp arg-expr ds ; step 2
         (λ (arg-val)
           (interp (closureV-body fun-val) ; step 3
                   (aSub (closureV-param-name fun-val)
                         arg-val
                         (closureV-ds fun-val))
            ; step 4
            (lambda (result) (k result))
            ; if you return, continue with this k!
            (lambda (result) (k result))))
         ; if we return when interpreting fun or arg
         ; we're still in the old function body
         ret-k))
      ret-k)]
```

ret-0, take 2

When we return, use the new continuation!

```
(define (interp a-fae ds k ret-k)
  (type-case FAE a-fae
    ...
    [ret-0 () (ret-k (numV 0))]
    ...))
```

We continue execution after the current function body!

```
{+ {{fun {x} {+ x {ret-0}}}
    5}
   3}
⇒ 3
```

ret

After we get the return value, go to the return continuation!

If we have to return along the way, we return!

```
(define (interp a-fae ds k ret-k)
  (type-case FAE a-fae
    ...
    [ret (ret-expr)
         (interp ret-expr ds
                 (lambda (ret-val)
                   (ret-k ret-val))
                 (lambda (ret-val)
                   (ret-k ret-val)))]
    ...))
```

For completeness

```
(define (numop op l r ds k ret-k)
  (interp l ds
          (lambda (l-v)
            (interp r ds
                    (lambda (r-v)
                      (k (numV
                          (op (numV-n l-v)
                              (numV-n r-v)))))
                    ret-k))
          ret-k))
```

Pass `ret-k` along in case either operand returns.

Otherwise continue execution as normal

What if I want to add stores?

Anything that would normally be an output to the interpreter becomes an input to the continuation!

```
interp : BFAE? DefSub? Store?
         (Value*Store? -> Value*Store?)
         -> Value*Store?
```

CPS is Awesome

- Used in compilers for higher-order languages
  - See Guy Steele's MS thesis (1978)
  - One of the two MS theses in CS that people actually read
- Used in linguistics
- Connections to embeddings of double-negation in intuitionistic logic
- Cheney on the MTA (see last lecture of the quarter)

- Reynolds's "The Discoveries of Continuations" (1993)
  - Independently discovered in a variety of settings

Beyond Early Return

- Homework 7: Exceptions

- More general: ***first-class*** continuations
  - Continuations as values in the object language!
  - Can pass them around, put them in boxes, invoke them later!
  - Keyword: call-with-current-continuation (call/cc)