How functions finish

Direct: `return answer;`

True CPS: `throw k answer;`

uScheme: `(k answer)`

Design Problem: Missing Value

Provide a witness to existence:

```
(witness p? xs) == x, where (member x xs),
                            provided (exists? p? xs)
```

Problem: What if there exists no such `x`?

Solution: A New Interface

Success and failure continuations!

Laws:

```
(witness-cps p? xs succ fail) = (succ x)
     ; where x is in xs and (p? x)

(witness-cps p? xs succ fail) = (fail)
     ; where (not (exists? p? xs))
```

Refine the laws

```
(witness-cps p? xs succ fail) = (succ x)
     ; where x is in xs and (p? x)
(witness-cps p? xs succ fail) = (fail)
     ; where (not (exists? p? xs))

(witness-cps p? '() succ fail) = ?

(witness-cps p? (cons z zs) succ fail) = ?
    ; when (p? z)

(witness-cps p? (cons z zs) succ fail) = ?
    ; when (not (p? z))
```

Coding `witness` with continuations

```
(define witness-cps (p? xs succ fail)
   (if (null? xs)
       (fail)
       (let ([x (car xs)])
         (if (p? x)
             (succ x)
             (witness-cps p? (cdr xs) succ fail)))))
```

“Continuation-Passing Style”

All tail positions are continuations or recursive calls

```
(define witness-cps (p? xs succ fail)
   (if (null? xs)
       (fail)
       (let ([x (car xs)])
         (if (p? x)
             (succ x)
             (witness-cps p? (cdr xs) succ fail)))))
```

Compiles to tight code

Example Use: Instructor Lookup

```
-> (val 2016f '((Fisher 105)(Hescott 170)(Chow 116)))
-> (instructor-info 'Fisher 2016f)
(Fisher teaches 105)
-> (instructor-info 'Chow 2016f)
(Chow teaches 116)
-> (instructor-info 'Souvaine 2016f)
(Souvaine is-not-on-the-list)
```

Instructor Lookup: The Code

```
 ; info has form: '(Fisher 105)
 ; classes has form: '(info_1 ··· info_n)
(define instructor-info (instructor classes)
  (let (
     [s                    ; success continuation
                                                    ]
     [f                    ; failure continuation
                                                   ])
    (witness-cps  pred
                  classes s f))
```

Instructor Lookup: The Code

```
 ; info has form: '(Fisher 105)
 ; classes has form: '(info_1 ··· info_n)
(define instructor-info (instructor classes)
  (let (
     [s                    ; success continuation
                                                    ]
     [f                    ; failure continuation
                                                   ])
    (witness-cps (o ((curry =) instructor) car)
                 classes s f))
```

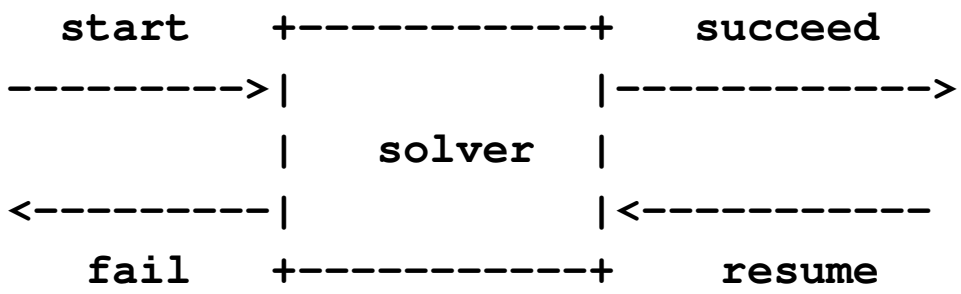
Instructor Lookup: The Code

```
 ; info has form: '(Fisher 105)
 ; classes has form: '(info_1 ··· info_n)
(define instructor-info (instructor classes)
  (let (
     [s (lambda (info) ; success continuation
          (list3 instructor 'teaches (cadr info)))]
     [f                  ; failure continuation
                                                  ])
    (witness-cps (o ((curry =) instructor) car)
                 classes s f))
```

Instructor Lookup: The Code

```
 ; info has form: '(Fisher 105)
 ; classes has form: '(info_1 ··· info_n)
(define instructor-info (instructor classes)
  (let (
     [s (lambda (info) ; success continuation
          (list3 instructor 'teaches (cadr info)))]
     [f (lambda ()     ; failure continuation
          (list2 instructor 'is-not-on-the-list))])
    (witness-cps (o ((curry =) instructor) car)
                 classes s f)))
```

Continuations for Search



start Gets partial solution, `fail`, `succeed`
(On homework, "solution" is assignment)

`fail` Partial solution won't work (no params)

`succeed` Gets improved solution + `resume`

`resume` If improved solution won't work,
try another (no params)

A composable unit!

Continuations for the solver

A big box contains two smaller boxes A and B

There are two ways to wire them up (board)

Imagine A and B as formulas

Imagine A as a formula, B as a *list* of formulas!

Solving a literal

```
(define satisfy-literal-true (x current succ fail)
  (if (bound? x current)
      (if (find x current)
          (succ current fail)
          (fail))
      (succ (bind x #t current) fail)))
```