Naive list reversal

```
(define reverse (xs)
   (if (null? xs)
       '()
       (append (reverse (cdr xs))
               (list1 (car xs)))))
```

Tail calls

Intuition: In a function, a call is in tail position if it is the last thing the function does.

```
(define reverse (xs)
   (if (null? xs)
       '()
       (append (reverse (cdr xs))
               (list1 (car xs)))))
```

Tail calls

Intuition: In a function, a call is in tail position if it is the last thing the function does.

```
(define reverse (xs)
   (if (null? xs)
       '()
       (append (reverse (cdr xs))
               (list1 (car xs)))))
```

A tail call is a call in tail position.

What is tail position?

Tail position is defined inductively:

- **The body of a function is in tail position**
- **When `(if e1 e2 e3)` is in tail position, so are `e2` and `e3`**
- **When `(let (...) e)` is in tail position, so is `e`, and similary for `letrec` and `let*`.**
- **When `(begin e1 ... en)` is in tail position, so is `en`.**

Idea: The last thing that happens

Tail-call optimization & accumulating parameters

Tail-call optimization

- Anything in tail position is the last thing executed, so its stack frame can be recycled
- Before executing a call in tail position, abandon current stack frame
- Results in asymptotic space savings

Accumulating Parameter

- Add a parameter to *accumulate* answer
- Moves recursive call to tail position

Example: Accumulating reverse function

Key ideas:

- **Recursive call is in a tail position**
- **Accumulating parameter collects answer**

Contract:

```
(revapp xs ys) = (append (reverse xs) ys)
```

Laws:

```
(revapp '()          ys) == ys
(revapp (cons z zs) ys) ==
            (revapp zs (cons z ys))
```

Reversal by accumulating parameters

```
; laws: (revapp '()          ys) = ys
;        (revapp (cons z zs) ys) =
;                    (revapp zs (cons z ys))

(define revapp (xs ys)
   ; return (append (reverse xs) ys)
   (if (null? xs)
       ys
       (revapp (cdr xs)
               (cons (car xs) ys))))


(define reverse (xs) (revapp xs '()))
```

Tail position in `revapp`

```
(define revapp (xs zs)
   (if (null? xs)
       zs
       (revapp (cdr xs) (cons (car xs) zs))))
```

Tail position in `revapp`

```
(define revapp (xs zs)
   (if (null? xs)
       zs
       (revapp (cdr xs) (cons (car xs) zs))))
```

Values `xs` and `zs` go in machine registers.

Code compiles to a loop.

Tail calls as gotos with arguments

`Gotos`

- Transfer control to another point in the code
- Use no stack space
- Can only go to code marked with labels

Tail calls

- Transfer control to another point in the code
- Use no stack space
- Can transfer control to any function
- Can take parameters!

Continuations: First-class tail calls

A continuation is code that represents "the rest of the computation."

- Not a normal function call because continuations never return
- Think "goto with arguments"

Context: an advanced technique, heavily used in event-driven programming:

- GUIs, games, web, embedded devices

Different coding styles

Direct style

- **Last action of a function is to return a value.**
  (This style is what you are used to.)

Continuation-passing style (CPS)

- **Last action of a function is to "throw" answer to a *continuation.***
  (For us, tail call to a parameter.)

Uses of continuations

- Compiler representation:
  - Compilers for functional languages often convert direct-style user code to CPS because CPS matches control-flow of assembly.
- First-class continuations.
  - Some languages provide a construct for *capturing* the *current continuation* and giving it a name `k`.
  - Control can be resumed at captured continuation by *throwing* to `k`.
- A style of coding that can mimic *exceptions*
- Callbacks in GUI frameworks and web services
- Event loops in game programming and other concurrent settings

Implementation

- We're going to simulate continuations with function calls in tail position.
- First-class continuations require compiler support.
  - primitive function that materializes "current continuation" into a variable.

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`?

Bad ideas:

- `nil`
- special symbol `fail`
- run-time error

Good idea: exception (but not available in uScheme)

Solution: A New Interface

Success and failure continuations!

Contract written using properties (not algorithmic)

```
(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))
```

From contract to 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))
```

Where do we have forms of data?

```
(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 ([z (car xs)])
         (if (p? z)
             (succ z)
             (witness-cps p? (cdr xs) succ fail)))))
```

“Continuation-Passing Style” by laws

The right-hand side of every law calls a parameter

```
(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))
```

“Continuation-Passing Style” by code

All tail positions are continuations or recursive calls

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

Compiles to tight code

Example Use: Instructor Lookup

```
-> (val 2020f '((Fisher 105)(Monroe 40)(Chow 116)))
-> (instructor-info 'Fisher 2020f)
(Fisher teaches 105)

-> (instructor-info 'Chow 2020f)
(Chow teaches 116)

-> (instructor-info 'Votipka 2020f)
(Votipka 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)))
```