# Programming Abstractions

## Lecture 33: Continuation Passing Style

Stephen Checkoway

# Continuations

Suppose expression E contains a subexpression S

The **continuation** of S in E consists of all of the steps needed to complete E after the completion of S

Example: `(- 4 (+ 1 1))`

- The subexpression S, (+ 1 1), is called the redex ("reducible expression")
- The continuation is `(- 4 □)` where □ takes the place of S

Example: `(displayln (foo (bar (* 2 3))))`

- The continuation of `(bar (* 2 3))` is `(displayln (foo □))`

What is the continuation of `(fact (sub1 n))` in the expression `(* n (fact (sub1 n)))`

A. `(* n (fact (sub1 n)))`

B. `(* n (fact (sub1 □)))`

C. `(* n (fact □))`

D. `(* n □)`

E. `□`

# A continuation is really a dynamic construct

A continuation is determined by the expression's evaluation context at run time

```
(define (fact n)
  (cond [(zero? n) 1]
        [else (* n (fact (sub1 n)))]))
```

At the point 1 is evaluated in the call `(fact 0)`, the continuation is `□`

At the point 1 is evaluated in the call `(fact 1)`, the continuation is `(* 1 □)`

At the point 1 is evaluated in the call `(fact 2)`, the continuation is
`(* 2 (* 1 □))`

Key: The continuation is **all** the rest of computation

# Continuations can be quite complicated!

Starting with a positive integer $n$, construct a sequence where each successive term is obtained by the current term $n$

- If the current term $n$ is 1, then stop.
- If the current term $n$ is even, the next term is $n$ / 2
- If the current term $n$ is odd, the next term is 3$n$ + 1

(The Collatz conjecture says that the sequence produced starting with any positive integer eventually stops.)

# Continuations of the Collatz computation

# Continuations of the Collatz computation

```
(define (collatz n)
  (cond [(= 1 n) '(1)]
        [(even? n) (cons n (collatz (/ n 2)))]
        [else (cons n (collatz (add1 (* 3 n))))]))
```

Continuations of `'(1)` in the call `(collatz n)` for several values of `n`

# Continuations of the Collatz computation

```
(define (collatz n)
  (cond [(= 1 n) '(1)]
        [(even? n) (cons n (collatz (/ n 2)))]
        [else (cons n (collatz (add1 (* 3 n))))]))
```

Continuations of `'(1)` in the call `(collatz n)` for several values of `n`

- `n = 1: □`

# Continuations of the Collatz computation

```
(define (collatz n)
  (cond [(= 1 n) '(1)]
        [(even? n) (cons n (collatz (/ n 2)))]
        [else (cons n (collatz (add1 (* 3 n))))]))
```

Continuations of `'(1)` in the call `(collatz n)` for several values of `n`

- `n = 1: □`
- `n = 2: (cons 2 □)`

# Continuations of the Collatz computation

```
(define (collatz n)
  (cond [(= 1 n) '(1)]
        [(even? n) (cons n (collatz (/ n 2)))]
        [else (cons n (collatz (add1 (* 3 n))))]))
```

Continuations of `'(1)` in the call `(collatz n)` for several values of `n`

- `n = 1: □`
- `n = 2: (cons 2 □)`
- `n = 3:`
  `(cons 3 (cons 10 (cons 5 (cons 16 (cons 8 (cons 4 (cons 2 □)))))))`

# Continuations of the Collatz computation

```
(define (collatz n)
  (cond [(= 1 n) '(1)]
        [(even? n) (cons n (collatz (/ n 2)))]
        [else (cons n (collatz (add1 (* 3 n))))]))
```

Continuations of `'(1)` in the call `(collatz n)` for several values of `n`

- `n = 1: □`
- `n = 2: (cons 2 □)`
- `n = 3:`
  `(cons 3 (cons 10 (cons 5 (cons 16 (cons 8 (cons 4 (cons 2 □)))))))`
- `n = 4: (cons 4 (cons 2 □))`

# Continuations of the Collatz computation

```
(define (collatz n)
  (cond [(= 1 n) '(1)]
        [(even? n) (cons n (collatz (/ n 2)))]
        [else (cons n (collatz (add1 (* 3 n))))]))
```

Continuations of `'(1)` in the call `(collatz n)` for several values of `n`

- `n = 1: □`
- `n = 2: (cons 2 □)`
- `n = 3:`
  `(cons 3 (cons 10 (cons 5 (cons 16 (cons 8 (cons 4 (cons 2 □)))))))`
- `n = 4: (cons 4 (cons 2 □))`
- `n = 5: (cons 5 (cons 16 (cons 8 (cons 4 (cons 2 □)))))`

```
(define (length lst)
  (cond [(empty? lst) 0]
        [else (add1 (length (rest lst)))]))
```

What is the continuation at the point `0` is evaluated in the call
`(length '(a b c))`

A. `3`

B. `(length lst)`

C. `(add1 (length □))`

D. `(add1 (add1 (add1 0)))`

E. `(add1 (add1 (add1 □)))`

# Viewing continuations as procedures

We can view a continuation as a procedure of one argument

Example: `(- 4 (+ 1 1))`

- The continuation is `(- 4 □)` where □ takes the place of S
- `(λ (x) (- 4 x))`

Example: `(displayln (foo (bar (* 2 3))))`

- The continuation of `(bar (* 2 3))` is `(displayln (foo □))`
- `(λ (x) (displayln (foo x)))`

# Continuation-passing style

A new way to implement recursive procedures

- Each procedure has an extra **continuation** parameter typically called `k`
- The continuation `k` says what to do with the result

# Continuation-passing style example

## Summing numbers in a list

```
(define (sum-k lst k)
  (cond [(empty? lst) (k 0)]
        [else (sum-k (rest lst)
                     (λ (x) (k (+ x (first lst)))))]))
```

Two things to notice:

- In the base case, we call the continuation with our base value `(k 0)`
- In the recursive case, we pass a new continuation procedure that calls `k` with the result of adding `x` to the head of `lst`

# Calling our function

What should we use as the top-level continuation when we call sum-k?

```
(define (sum-k lst k)
  (cond [(empty? lst) (k 0)]
        [else (sum-k (rest lst)
                     (λ (x) (k (+ x (first lst)))))]))
```

It depends what we want to do with it, typically, we'd want to return the value

- We can use `(λ (x) x)` which Racket predefines as `identity`

```
(sum-k '(1 2 3 4) identity) => 10
```

# Compare with accumulator-passing style

```
(define (sum-k lst k)
  (cond [(empty? lst) (k 0)]
        [else (sum-k (rest lst)
                     (λ (x) (k (+ x (first lst)))))]))

(define (sum-a lst acc)
  (cond [(empty? lst) acc]
        [else (sum-a (rest lst) (+ acc (first lst)))]))
```

In CPS, the extra parameter is a procedure that says what to do with the result of the computation

In APS, the extra parameter is the intermediate value in the computation

# CPS guidelines for recursive procedures

Continuations are procedures with 1 argument

The recursive procedure has a continuation parameter, `k`

The continuation argument is called once for each branch of computation (think base case and recursive case)

- Not calling the continuation on one of the cases is a common mistake

At the top-level, the continuation is usually identity

Recursive calls must be tail-recursive