C311/B521/A596 Programming Languages

Home | Policies | Instructors | Resources | Schedule | Assignments | B621

Login

Assignment 6: Continuation-Passing Style

> Say you're in the kitchen in front of the refrigerator, thinking about a sandwich. You take a continuation right there and stick it in your pocket. Then you get some turkey and bread out of the refrigerator and make yourself a sandwich, which is now sitting on the counter. You invoke the continuation in your pocket, and you find yourself standing in front of the refrigerator again, thinking about a sandwich. But fortunately, there's a sandwich on the counter, and all the materials used to make it are gone. So you eat it.
>
> *—Luke Palmer*

Note

In addition to your notes from class, you may find the following resources helpful as you work through this assignment

- A CPS Refresher on converting procedures to continuation-passing style.
- Notes from the Feb 16, 2010 lecture on continuation-passing style.
- Notes from a previous AI on converting procedures to CPS.

Assignment

For this assignment, you will convert several short programs to continuation-passing style. Please observe the following guidelines:

- When CPSing, you may treat built-in procedures such as `null?`, `add1`, `assq`, `car`, `<`, and the like as “simple”.
- Test your CPSed procedures using the initial continuation returned from `empty-k`.
- We provide a small test suite a6-student-tests for Part 1; to test Part 2 you should modify the calls provided in this assignment to be invocations of your CPSed implementations.

```
(define empty-k
  (lambda ()
    (let ((once-only #f))
      (lambda (v)
        (if once-only
     (error 'empty-k "You can only invoke the empty continuation once")
     (begin (set! once-only #t) v))))))
```

You may have seen `empty-k` defined as the following.

```
(define empty-k
  (lambda ()
    (lambda (v) v)))
```

However, the one above is much better, in that it will help you better detect if you have made a mistake in cps-ing.

Part I : call/cc

For part 1, you should *not* CPS these procedures or use `empty-k`.

1. Complete the following definition of `last-non-zero`, a function which takes a list of numbers and returns the last cdr whose car is 0. In other words, when starting from the right of the list, it should be all numbers before the first 0 is reached. See the test cases below and student test file for examples. Your solution should be naturally recursive, and should not contain any calls to member-like operations, nor should you be reversing the list.

```
(define last-non-zero
  (lambda (ls)
    (call/cc
     (lambda (k)
       (letrec
         ((lnz
           (lambda (ls)




      )))
         (lnz ls))))))
```

```
> (last-non-zero '(0))
```

```
()
> (last-non-zero '(1 2 3 0 4 5))
(4 5)
> (last-non-zero '(1 0 2 3 0 4 5))
(4 5)
> (last-non-zero '(1 2 3 4 5))
(1 2 3 4 5)
>
```

2. Direct vs. accumulator-passing vs. call/cc.

Consider the following definitions of `mult` and `mult/acc`, a function which takes a list of numbers and returns the product, respectively written in direct and accumulator-passing styles.

```
> (define my-*
    (lambda (m n)
      (* m n)))
> (define mult
    (lambda (n*)
      (letrec
        ((m
           (lambda (n*)
             (cond
               ((null? n*) 1)
               ((zero? (car n*)) 0)
               (else (my-* (car n*) (mult (cdr n*))))))))
         (m n*))))
> (define mult/acc
    (lambda (n*)
      (letrec
        ((m/acc
           (lambda (n* acc)
             (cond
               ((null? n*) acc)
               ((zero? (car n*)) 0)
               (else (m/acc (cdr n*) (my-* (car n*) acc)))))))
         (m/acc n* 1))))
```

Complete the definition below to build a third version, `mult/cc`, which instead uses a system continuation `k` to return with 0 if the list contains a 0. Your implementation should be naturally-recursive.

```
> (define mult/cc
    (lambda (n*)
      (call/cc
       (lambda (k)
         (letrec
           ((m/cc
              (lambda (n*)



                )))
            (m/cc n*))))))
```

If you instead require the `C311/trace` library, and define my-* to be traced, we see an interesting property.

```
> (require C311/trace)
> (trace-define my-*
    (lambda (m n)
      (* m n)))
> (mult '(1 2 3 4 0 6 7 8 9))
>(my-* 4 0)
<0
>(my-* 3 0)
<0
>(my-* 2 0)
<0
>(my-* 1 0)
<0
0
> (mult/acc '(1 2 3 4 0 6 7 8 9))
>(my-* 1 1)
<1
>(my-* 2 1)
<2
>(my-* 3 2)
<6
>(my-* 4 6)
<24
0
```

```
> (mult/cc '(1 2 3 4 0 6 7 8 9))
0
```

When you have a `0` in the list, both the direct and accumulator passing versions perform multiplications until they reach a `0`, yet `mult/cc` performs *no* multiplications in the presence of a `0`. Which is neat.

Part II : CPS

3. Define and test a procedure `times-cps` that is a CPSed version of the following `times` procedure.

```
(define times
  (lambda (ls)
    (cond
      [(null? ls) 1]
      [(zero? (car ls)) 0]
      [else (* (car ls) (times (cdr ls)))])))
```

Here are some examples of calls to `times`:

```
> (times '(1 2 3 4 5))
120
> (times '(1 2 3 0 3))
0
```

4. Define a modified version of your `times-cps` above, called `times-cps-shortcut` that doesn't apply `k` in the zero case. Instead, maintain the behavior of the `zero?` case in `times` - simply returning the 0 and not performing further computation. While this certainly violates the standard rules of CPSing the program, it provides an interesting look at optimizations CPSing allows us.

5. Define and test a procedure `plus-cps` that is a CPSed version of the following `plus` procedure:

```
(define plus
  (lambda (m)
    (lambda (n)
      (+ m n))))
```

Here are some examples of calls to `plus`:

```
> ((plus 2) 3)
5
> ((plus ((plus 2) 3)) 5)
10
```

6. Define and test a procedure `count-syms*-cps` that is a CPSed version of the following `count-syms*` procedure:

```
(define count-syms*
  (lambda (ls)
    (cond
      [(null? ls) 0]
      [(pair? (car ls)) (+ (count-syms* (car ls)) (count-syms* (cdr ls)))]
      [(symbol? (car ls)) (add1 (count-syms* (cdr ls)))]
      [else (count-syms* (cdr ls))])))
```

Here are some example calls to `count-syms*`

```
> (count-syms* '(a 1 b 2 c 3))
3
> (count-syms* '((a 1) (b 2) (c 3)))
3
> (count-syms* '(1 (b (3 (d (5 e) 7) (g)) 9) ((h))))
5
```

7. Define and test a procedure `cons-cell-count-cps` that is a CPSed version of the following `cons-cell-count` procedure:

```
(define cons-cell-count
  (lambda (ls)
    (cond
      [(pair? ls)
       (add1 (+ (cons-cell-count (car ls)) (cons-cell-count (cdr ls))))]
      [else 0])))
```

8. Define and test a procedure `walk-cps` that is a CPSed version of the following `walk` procedure:

```
(define walk
  (lambda (v ls)
    (cond
      [(symbol? v)
```

```
        (let ((p (assq v ls)))
          (cond
            [p (walk (cdr p) ls)]
            [else v]))]
      [else v])))
```

Here are some sample calls to `walk`:

```
> (walk 'a '((a . 5) (b . 6) (c . 7)))
5
> (walk 'a '((a . b) (b . c) (c . 7)))
7
> (walk 'a '((a . q) (r . s) (q . r)))
s
> (walk 'a '((a . q) (r . s) (q . r) (s . 10)))
10
```

9. Define and test a procedure `ack-cps` that is a CPSed version of the following `ack` procedure:

```
;; ack: computes the Ackermann function
;; (http://en.wikipedia.org/wiki/Ackermann_function).  Warning: if you
;; run this program with m >= 4 and n >= 2, you'll be in for a long
;; wait.
(define ack
  (lambda (m n)
    (cond
      [(zero? m) (add1 n)]
      [(zero? n) (ack (sub1 m) 1)]
      [else (ack (sub1 m)
                 (ack m (sub1 n)))])))
```

10. Define and test a procedure `fib-cps` that is a CPSed version of the following `fib` procedure:

```
(define fib
  (lambda (n)
    ((lambda (fib)
       (fib fib n))
     (lambda (fib n)
       (cond
  [(zero? n) 0]
  [(= 1 n) 1]
  [else (+ (fib fib (sub1 n)) (fib fib (sub1 (sub1 n))))])))))
```

11. Define and test a procedure `unfold-cps` that is a CPSed version of the following `unfold` procedure:

```
(define unfold
  (lambda (p f g seed)
    ((lambda (h)
       ((h h) seed '()))
     (lambda (h)
       (lambda (seed ans)
  (if (p seed)
      ans
      ((h h) (g seed) (cons (f seed) ans))))))))
```

An example of its use is demonstrated below:

```
> (unfold null? car cdr '(a b c d e))
(e d c b a)
```

When testing your `unfold-cps`, you should consider its arguments to be serious, so include the following helper definitions when testing your code.

```
> (define null?-cps
    (lambda (ls k)
      (k (null? ls))))
> (define car-cps
    (lambda (pr k)
      (k (car pr))))
> (define cdr-cps
    (lambda (pr k)
      (k (cdr pr))))
> (unfold-cps null?-cps car-cps cdr-cps '(a b c d e) (empty-k))
(e d c b a)
```

12. Here is the definition of `unify` with its helpers. The current version uses the version of `walk` given in question 6. Define and test a procedure `unify-cps` that uses your `walk-cps` from question 5. Treat `extend-s` as simple.

```
(define empty-s
```

```
  (lambda ()
    '()))

(define extend-s
  (lambda (x v s)
    (cons `(,x . ,v) s)))

(define unify
  (lambda (v w s)
    (let ([v (walk v s)])
      (let ([w (walk w s)])
        (cond
          [(eqv? v w) s]
          [(symbol? v) (extend-s v w s)]
          [(symbol? w) (extend-s w v s)]
          [(and (pair? v) (pair? w))
           (let ((s (unify (car v) (car w) s)))
             (cond
               [s (unify (cdr v) (cdr w) s)]
               [else #f]))]
          [(equal? v w) s]
          [else #f])))))
```

Here are some example calls to `unify`:

```
> (unify 'x 5 (empty-s))
((x . 5))
> (unify 'x 5 (unify 'y 6 (empty-s)))
((x . 5) (y . 6))
> (unify '(x y) '(5 6) (empty-s))
((y . 6) (x . 5))
> (unify 'x 5 (unify 'x 6 (empty-s)))
#f
> (unify '(x x) '(5 6) (empty-s))
#f
> (unify '(x y z) '(5 x y) (empty-s))
((z . 5) (y . 5) (x . 5))
```

13. Define and test a procedure `M-cps` that is a CPSed version of `M`, which is a curried version of `map`. Assume for the CPSed version that any `f` passed in will also be CPSed.

```
(define M
  (lambda (f)
    (lambda (ls)
      (cond
        ((null? ls) '())
        (else (cons (f (car ls)) ((M f) (cdr ls))))))))
```

14. Consider the corresponding call to `M`, called `use-of-M`. Using your CPSed `M-cps`, re-write `use-of-M` to call `M-cps`, and make all the appropriate changes (including CPSing the argument). Name it `use-of-M-cps`

```
(define use-of-M
  ((M (lambda (n) (add1 n))) '(1 2 3 4 5)))
```

Brainteaser

15. CPS the following program, and call it `strange-cps`:

```
(define strange
  (lambda (x)
    ((lambda (g) (lambda (x) (g g)))
     (lambda (g) (lambda (x) (g g))))))
```

16. Consider the following use of `strange`, called `use-of-strange`. Using your CPSed `strange`, re-write `use-of-strange` to call `strange-cps`, and make all the appropriate changes. Name it `use-of-strange-cps`.

```
(define use-of-strange
  (let ([strange^ (((strange 5) 6) 7)])
    (((strange^ 8) 9) 10)))
```

17. CPS the following program, and call it `why-cps`:

```
(define why
  (lambda (f)
    ((lambda (g)
       (f (lambda (x) ((g g) x))))
     (lambda (g)
       (f (lambda (x) ((g g) x)))))))
```

To get you started, you may find it useful to see the following-call to `why`.

```
> (define almost-length
    (lambda (f)
      (lambda (ls)
        (if (null? ls)
            0
            (add1 (f (cdr ls)))))))
> ((why almost-length) '(a b c d e))
5
```

Just Dessert

18. CPS `why-cps`, and call it `why-cps-cps`.