Login

# Assignment 8: Registerization

> A trampoline is strong enough to catch you before you hit the ground, but not so cushy that you can live on it forever

## Note

As you proceed with this assignment, you may find it useful the following resources helpful.

- Notes on registerization and trampolinizing.

Do also consult your class notes and any files distributed to you from lecture or lab.

As usual, you can use the a8-student-tests.rkt tests file to test your submission, although these tests are not exhaustive.

## Assignment

Consider the following four procedures:

```
(define ack
  (lambda (m n k)
    (cond
      [(zero? m) (k (add1 n))]
      [(zero? n) (ack (sub1 m) 1 k)]
      [else (ack m (sub1 n) (lambda (v) (ack (sub1 m) v k)))])))

(define depth
  (lambda (ls k)
    (cond
      [(null? ls) (k 1)]
      [(pair? (car ls))
       (depth (car ls)
              (lambda (l)
                (depth (cdr ls)
                       (lambda (r)
                         (let ((l (add1 l)))
                           (if (< l r) (k r) (k l)))))))]
      [else (depth (cdr ls) k)])))

(define fact
  (lambda (n k)
    ((lambda (fact k)
       (fact fact n k))
     (lambda (fact n k)
       (cond
```

```
         [(zero? n) (k 1)]
         [else (fact fact (sub1 n) (lambda (v) (k (* n v))))]))
     k)))

(define pascal
  (lambda (n k)
    (let ((pascal
           (lambda (pascal k)
             (k (lambda (m a k)
                  (cond
                    [(> m n) (k '())]
                    [else (let ((a (+ a m)))
                            (pascal pascal (lambda (f) (f (add1 m) a (lambda (v) (k (cons a v)))))))]))))))
      (pascal pascal (lambda (f) (f 1 0 k))))))
```

Here are examples of how to call these procedures:

```
> (ack 2 2 (empty-k))
7
> (depth '(1 (2 (3 (4)))) (empty-k))
4
> (fact 5 (empty-k))
120
> (pascal 10 (empty-k))
(1 3 6 10 15 21 28 36 45 55)
```

For each of the four initial programs, you should write a registerized version of the original. You **should not** trampoline these procedures. You should also construct driver functions, to be invoked as follows. Each of these driver programs will take the arguments taken by the original, pre-CPSed program. The job of the driver will be to populate the registers with initial values and call the registerized program. Once again, you don't need to trampoline your registerized programs.

```
> (ack-reg-driver 2 2)
7
> (depth-reg-driver '(1 (2 (3 (4)))))
4
> (fact-reg-driver 5)
120
> (pascal-reg-driver 10)
(1 3 6 10 15 21 28 36 45 55)
```

As you do the assignment, keep the following points in mind:

- All definitions must be *representation-independent with respect to continuations*. You must use a *data-structural* representation of continuations.
- **Make sure to name your drivers exactly as above, or we will not be able to grade your submission!**

When you are finished, place all of your code in a file named `a8.rkt` and submit it to Canvas.

## Brainteaser

Trampolines can be used to execute multiple trampolinized programs simultaneously. Write a trampoline-like procedure `rampoline` that takes three thunks containing trampolinized procedure calls, and executes them in random order, returning the value of the first to complete. Note that the procedures may go into an infinite loop. To show that `rampoline` works correctly, CPS and trampolinize the following definition of `fib`:

```
(define fib
  (lambda (n)
    (cond
      [(and (not (negative? n)) (< n 2)) n]
      [else (+ (fib (sub1 n)) (fib (sub1 (sub1 n))))])))
```

Use the following as a driver to your trampolinized fib:

```
(define fib-ramp-driver
  (lambda (n1 n2 n3)
    (let/cc jumpout
      (rampoline
        (lambda ()
          (fib n1 (ramp-empty-k jumpout)))
        (lambda ()
          (fib n2 (ramp-empty-k jumpout)))
        (lambda ()
          (fib n3 (ramp-empty-k jumpout)))))))
```

## Just Dessert

A trampoline can be used to interleave executions of two trampolinized programs, we've seen (or can imagine) how executing two such programs in a trampoline can return the first answer that completes. But what if we want /both/ answers?

Devise a way to return both answers, wherein:

- We still interleave executions of both programs with a trampoline.
- We get back a list with both answers.
- The answers are listed in the order they complete.
- We don't redo any work after finding the first answer.

You will need to define a `bi-trampoline`, a `bi-tramp-driver`, and a trampolinized version of `trib` (that generates elements of the tribonacci sequence [https://en.wikipedia.org/wiki/Generalizations_of_Fibonacci_numbers#Tribonacci_numbers]). Automated tests are set to run `bi-tramp-driver` with two numbers. Note: here, you do not need to registerize in order to trampolinize.

```
(define trib
  (lambda (n)
    (cond
      ((< n 3) 1)
      (else
       (+ (trib (- n 3))
```

```
          (trib (- n 2))
          (trib (- n 1)))))))
```

```
> (bi-tramp-driver 3 4)
(3 5)
> (bi-tramp-driver 4 3)
(3 5)
> (bi-tramp-driver 6 6)
(17 17)
```

assignment-8.txt · Last modified: 2018/10/17 10:37 by mvc