CS 421 --- CPS Activity

Manager	Keeps team on track	
Recorder	Records decisions / QC	
Reporter	Reports to class	
Reflector	Assesses team performance	

Please write your name/netid legibly in dark ink. Hand in one copy per team. Do not staple or mangle the corners.

Learning Objectives

1. Explain the characteristics of continuations and functions written in continuation passing style (CPS).
2. Critique code that represents a failed attempt to convert a function to CPS.
3. Convert some functions to CPS using the supplied transformation rules.
4. Change the result of a computation by reordering continuations.

Characteristics

Direct Style

```
0 fact 0 = 1
1 fact n = n * fact (n-1)
2
3 sumList []     = 0
4 sumList (x:xs) = x + sumList xs
5
6 prodList []     = 1
7 prodList (0:xs) = 0
8 prodList (x:xs) = x * prodList xs
```

CPS

```
0 fact 0 k = k 1
1 fact n k = fact (n-1) (\v -> k (n * v))
2
3 sumList []     k = k 0
4 sumList (x:xs) k = sumList xs (\v -> k (x + v))
5
6 prodList xx k = aux xx k k
7  where aux []     ks ka = ks 1
8        aux (0:xs) ks ka = ka 0
9        aux (x:xs) ks ka = aux xs (\v -> ks (x * v)) ka
```

Problem 1) What is the relationship between the return value of a direct-style function and the value passed into `k` in the CPS equivalent functions?

Problem 2) There are several anonymous functions in the CPS code. Their parameters are all named `v`. What kinds of values are being passed into these parameters?

Problem 3) What percentage of the recursive calls are in tail form in the CPS functions?

Problem 4) We like to pretend that functions written in CPS ``never return''. What justifies that?

Problem 5) Suppose we call direct style `prodList` with argument `[2,3,0,4,9,8,7]`. How many times will the multiplication operator be invoked? Suppose we call CPS `prodList` with the same list and continuation `print`. (The `print` function prints its argument to the screen and does not return any value[1].)

Now It's Ruined

Consider the following three programs. They are not in CPS.

```
0 decList [] k = []
1 decList (x:xs) k = (x-1 : decList xs k)
```

```
0 mink a b k = if a < b then k a else k b
1 minList [x] k = k x
2 minList (x:xs) k = minList xs (\v -> mink v x id)
```

```
0 maxk a b k = if a > b then k a else k b
1 maxList [x] k = k x
2 maxList (x:xs) k = maxk x (maxList xs k) k
```

Problem 6) For `decList`, `maxList`, and `minList`, explain why they are not in continuation passing style.

[1] Well, it returns `unit` in the IO monad.

Conversion

Here are some of the conversion rules.

$$C[\![f\ arg\ =\ e]\!] \Rightarrow f\ arg\ k\ =\ C[\![e]\!]_k$$

	$C[\![a]\!]_k \Rightarrow\ k\ a$
arg is simple	$C[\![f\ arg]\!]_k \Rightarrow\ f\ arg\ k$
	$C[\![f\ arg]\!]_k \Rightarrow\ C[\![arg]\!]_{(\lambda v.f\ v\ k)}$, where v is fresh.
e_1, e_2 are simple	$C[\![e_1 + e_2]\!]_k \Rightarrow\ k(e_1 + e_2)$
e_2 is simple	$C[\![e_1 + e_2]\!]_k \Rightarrow\ C[\![e_1]\!]_{(\lambda v -> k(v+e2))}$where v is fresh.
	$C[\![e_1 + e_2]\!]_k \Rightarrow\ C[\![e_1]\!]_{(\lambda v_1 -> C[\![e_2]\!]_{\lambda v_2 -> k(v_1+v_2)})}$where v_1 and v_2 are fresh.

Problem 7) The phrase ``where v is fresh'' appears a lot here. Why do we need to be concerned about that?

Problem 8) Suppose we want to convert these functions into CPS. There are helper functions `f`, `g`, and `h`. During the conversion process, we are also going to convert `f` and `g` to CPS, but leave `h` in direct style.

```
0 foo a b = f a + g b
1
2 bar x y = h x + y
3
4 baz c = 3 + c
5
6 quux d = h (g d)
```

Which subexpressions in the code above are simple? Can you think of way to describe what being simple means in this context?

Convert To

Problem 9) Convert `map` to CPS. Assume `f` is written in direct style.

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Problem 10) Do it again, but this time assume `f` is written in CPS and takes one continuation.

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Problem 11) Convert the following code to CPS, preserving the order of operations that would be used if Haskell were an eager language. Note: you will need a *nested continuation* to make this work.

```
min a b = if a < b then a else b
min4 a b c d = min (min a b) (min c d)
```

Reordering Computations

On the off chance we have extra time, here's something to try.

Suppose you have a calculator which has an accumulator and a list of instructions. `Add i` adds `i` to the accumulator, and `Sub i` subtracts `i` from the accumulator.

```
data Calc = Add Integer
          | Sub Integer
    deriving (Eq,Show)
```

The only problem is that our accumulator cannot ever be negative! Use continuations to fix this.

Here's the original calculator:

```
calc xx = aux 0 xx
  where aux a [] = a
        aux a ((Add i):xs) = aux (a+i) xs
        aux a ((Sub i):xs) = aux (a-i) xs
```

Hint: you will need *two* continuations to make this work.

CPS Activity--- Reflector's Report

Manager	Keeps team on track	
Recorder	Records decisions	
Reporter	Reports to Class	
Reflector	Assesses team performance	

1. What was a strength of your team's performance for this activity?

2. What could you do next time to increase your team's performance?

3. What insights did you have about the activity or your team's interaction today?

CPS Activity --- Team's Assessment (SII)

Manager or Reflector: Consider the objectives of this activity and your team's experience with it, and then answer the following questions after consulting with your team.

1. What was a **strength** of this activity? List one aspect that helped it achieve its purpose.

2. What is one things we could do to **improve** this activity to make it more effective?

3. What **insights** did you have about the activity, either the content or at the meta level?