CS 421 --- CPS Activity

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

Observations

Consider the following three programs:

```
fact 0 k = k 1
fact n k = fact (n-1) (\v -> k (n * v))
```

```
sumList [] k = k 0
sumList (x:xs) = sumList xs (\v -> k (x + v))
```

```
prodList [] ks ka = ks 1
prodList (0:xs) ks ka = ka 0
prodList (x:xs) ks ka = prodList xs (\v -> ks (x * v)) ka
```

Problem 1) Which of the above calls to `k`, `ka`, or `ks` are in tail position?

Problem 2) Which of the above recursive calls are in tail position?

Problem 3) What is the role of `v` in the above codes?

Problem 4) What is the role of `ka` and `ks` in the `prodList` code?

Now It's Ruined

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

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

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

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

Problem 5) For each of the programs above, describe why each fails to be in continuation passing style.

Conversion

Here are the conversion rules.

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

$C[\![a]\!]_k \Rightarrow\ k\ a$
$C[\![f\ arg]\!]_k \Rightarrow\ f\ arg\ k$
$C[\![f\ arg]\!]_k \Rightarrow\ C[\![arg]\!]_{(\lambda v.f\ v\ k)}$, where v is fresh.
$C[\![e_1 + e_2]\!]_k \Rightarrow\ k(e_1 + e_2)$
$C[\![e_1 + e_2]\!]_k \Rightarrow\ C[\![e_1]\!]_{(\lambda v -> k(v+e2))}$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 6) The phrase ``where v is fresh'' appears a lot here. Why do we need to be concerned about that?

Problem 7) A lot of these rules seem to have more than one form. How do you know which version to use?

Convert To

Problem 8) Convert `map` to CPS. Assume `f` is written in direct style.

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

Problem 9) 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 10) Convert the following code to CPS. 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.

```haskell
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:

```haskell
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?