Computer Science 131, Fall 2000
Assignment 4: Continutations

Out: Tuesday, October 10

Due: Friday, October 20, 5:00pm

Despite the length of this writeup, this assignment only requires you to write around a dozen lines of code. To complete this assignment, retrieve the file `assign4.sml` from the course web pages. Run SML/NJ on turing using `/cs/cs131/bin/sml-cs131-a4`. This is again a version of the compiler with important functions predefined. Complete the `assign4.sml` file and turn it in using `cs131submit`.

1 Continuation Passing Style

In a previous class on exceptions you saw code for a function `coins`.

```
exception Impossible
fun coins (_,0)     = []
  | coins ([],_)    = raise Impossible
  | coins (c::cs,n) =
       if (c <= n) then
          (c :: (coins(c::cs,n-c)))
              handle Impossible => coins(cs,n)
       else
          coins(cs,n)
```

This function takes a list of values for available coins and an amount of money, and attempts to find a collection of coins that add up to that amount. For example, if we have 2-cent and 5-cent coins available and want a total of 8 cents, the call `coins([5,2],8)` returns the list `[2,2,2,2]` meaning that one can make 8 cents with four 2-cent coins. If there is no way to satisfy the inputs, for example in the case of `coins([5,2],3)`, the function raises the exception `Impossible`.

The code works by extending the obvious greedy algorithm (use as many as you can of the first coin without going over, then use as many as you can of the second coin, etc., until you either reach your goal or run out of possible coin values) with backtracking. If it turns out to be a bad idea to have used a coin, the recursive call will fail (raise the `Impossible` exception); this exception is caught by the handler, which then investigates the case where we don't use that coin.

To convince yourself that the code is correct, it may be useful to consider how one would prove this fact. To prove that a recursive function is correct, the obvious approach is induction. The first two clauses in the function correspond to base cases and are obviously correct. For the third and last clause of the function definition, inductively assume that recursive calls give the correct answer (returning a list of coins if one exists and raising `Impossible` otherwise) and convince yourself that the code for this last case checks all possibilities. (It should be intuitively obvious that the recursive calls are applied to "smaller" arguments so that we can apply the inductive hypothesis; this can be easily formalized.)

In a compiler like SML/NJ, all control flow is handled by modifying the user's code to use continuation functions. The purpose of this problem is to examine how such a compiler might compile the function `coins` above.

1. First, we convert all recursive calls in this function to use an extra parameter, a continuation function. Define the function

   ```
   coins2 : int list * int * (int list -> 'a) -> 'a
   ```

   which takes a list of coin values, the sum to be achieved, and a continuation. The call `coins2(cs,n,k)` should either return the result of calling `k` with a correct list of coins, or raise `Impossible` if no such list exists. Your code should take the following skeleton and filling in the correct arguments to the recursive calls.

   ```
   fun coins2(_,0,k)  = k[]
     | coins2([],_,_) = raise Impossible
     | coins2(c::cs,n,k) =
         if (c <= n) then
            coins2( ...arguments...)
              handle Impossible => coins2( ...arguments...)
         else
            coins2( ...arguments...)
   ```

   The `assign4.sml` file also provides an initial continuation `k2` that can be used for testing purposes; it merely prints the answer. For example, the call `coins2([5,2],13,k2)` will print "`5 2 2 2 2`" and the call `coins2([5,2],3,k2)` will raise `Impossible`.

2. The first recursive call in `coins2` is not actually a tailcall so a stack would still be required. (Values must be saved in case an exception is raised and we have to run the code for the exception handler.) We can turn all function calls into tailcalls and eliminate the uses of `raise` and `handle` by having the function take *two* continuation functions as parameters. One is the "success" continuation (what should be done next if a normal answer is computed) and one is the "failure" continuation (what should be done next otherwise). A `raise` then corresponds to invoking the failure continuation (and throwing away the normal continuation), while `handle` corresponds to modifying the failure continuation for the code wrapped by the exception handler. Since there is only one exception we care about here, the failure continuation can just be a function expecting a unit argument. (In the general case, the failure continuation would take an exception of type `exn` as its argument.)

   Define the function

   ```
   coins3 : int list * int * (int list -> 'a) * (unit -> 'a) -> 'a
   ```

   which takes the list of coin values, the sum to be achieved, a success continuation, and a failure continuation. The call `coins3(cs,n,k,h)` will either return the result of applying `k` to a correct list of coins, or return the result of `h()` if no such solution exists. Your code should fill in the arguments in the following code skeleton:

```
fun coins3(_,0,k,h)  = k[]
  | coins3([],_,_,h) = h()
  | coins3(c::cs,n,k,h) =
      if (c <= n) then
         coins3( ...arguments...)
      else
         coins3( ...arguments...)
```

The `assign4.sml` file contains initial success and failure continuations `k3` (which prints the list of coins) and `h3` (which prints an error message) that can be used for testing purposes. So the call `coins3([5,2],13,k3,h3)` prints "`5 2 2 2 2`" while the call `coins3([5,2],3,k3,h3)` prints "`No solution found`"

2 Coroutines and `callcc`

For this problem, your task is to implement a version of Unix pipes that can serve as a better buffer for connecting producers and consumers. As a slight simplification, we will assume that pipes can store arbitrary amounts of data.

The provided SML/NJ binary contains definitions for the thread operations given in class:

```
val fork  : (unit -> unit) -> unit   (* runs child first *)
val fork' : (unit -> unit) -> unit   (* queues child to run later *)
val exit  : unit -> 'a
val yield : unit -> unit
val reset : unit -> unit   (* clears the ready queue of all threads *)
```

and definitions for the queue operations:

```
type 'a queue
val mkQueue  : unit -> 'a queue
val enqueue  : 'a queue * 'a -> unit
val dequeue  : 'a queue -> 'a option
```

The representation of a pipe will be a record containing two queues: a queue of values waiting to be read by consumers, and a queue of continuations, representing consumers waiting for values. At any point, at least one of these two queues should be empty: there should never simultaneously be values in the pipe and consumers waiting for a value.

```
type 'a pipe = {values : 'a queue,
                consumers : ('a cont) queue}
```

Your job is to implement the following three functions:

```
val newPipe   : unit -> 'a pipe
val readPipe  : 'a pipe -> 'a
val writePipe : 'a pipe * 'a -> unit
```

The function `newPipe` should create a new empty pipe (fairly trivial). The function `readPipe` will return the first value in the pipe's queue of values if one is available. Otherwise the thread calling `readPipe` should store its continuation in the pipe and stop running. The function `writePipe` either stores a new value in the pipe (if no consumers are waiting for a value) and otherwise re-starts the first waiting consumer by providing it with a value; `writePipe` then returns `()` to the calling thread, which should continue on.

A few hints:

- Your code will require `callcc` and `throw` as well as `fork` and `exit`.
- Be careful about using `throw`, and remember that a thread that does the `throw` has its continuation thrown away. However the thread calling `writePipe` should not disappear.
- Depending on your code, calls to `readPipe` and `writePipe` may or may not have the effect of yielding to let other ready threads run for a while.
- Remember that if you run a test that forks some threads and you stop it with Control-C, the ready queue does not automatically clear; old threads are still there waiting for their chance to run. This can cause bizarre behavior if you run another test. Use the function `reset` to clear the ready queue.
- It may be difficult to write `readPipe` without thinking about `writePipe`, and vice-versa.
- The `assign4.sml` file contains a simple test function `run` which uses a pipe to connect 2 consumers with 1 producer.