CS2104 – Programming Languages
School of Computing
National University of Singapore

October 16, 2012

Problem Set 6

Semester 1, 2012/13

Due: ***October 28, 23:59***

Marks: 8

Submission: In IVLE, in the cs2104 workbin, you will find a folder called "`Homework submissions`". In that folder, there are currently 2 ***subfolders:*** **`PS601,`** and **`PS6P02`**. The last two digits of the folder name indicate the solution that is to be submitted into that folder: the solution to *Question 1* into **`PS6P01`**, and so on (that is, you need to submit 2 separate solutions to 2 problems). A solution should consist of a ***single text file*** that can be compiled, or loaded into the interpreter of interest and executed. You should provide as much supplementary information about your solution as you can, ***in the form of program comments***. Moreover, if you work in a team, state the members of the team at the beginning of the file, in a comment. You do not need to submit the same file twice, one submission per team is sufficient.

Problem 1 [4 marks, submit to `PS6P01`]

The following Haskell program simulates digital circuits. Read carefully the code and its comments to understand how it works. For instructions on how to run this program, watch the video recording available in the media workbin. The code is available in `sim.hs`.

```
-- Haskell program to simulate digital circuits
--
--  a signal is an infinite list of Bool; each element
--     represents the value of the signal for a specific
--     clock cycle
--
--  logic gates are higher order functions that take
--     signals as arguments and output a signal ;
--     each gate also has a certain delay
--
--  complex circuits, like the /SR NAND latch given below
--     are obtained by connecting inputs and outputs of gates
--     through signals, which are transmitted by wires ;
--     thus, there will be a signal for every wire in the circuit

module Sim where

-- the signal that is always on "High"
high::[Bool]
high = True:high

-- the signal that is always on "Low"
```

```
low::[Bool]
low = False:low

-- create a limited list, of length "n", filled with logical value "fill"
-- useful to create signals by appending finite sequences
set :: Integer -> Bool -> [Bool]
set 0 _ = []
set n fill | n>0 = fill:(set (n-1) fill)

-- delay a signal by "n" clock cycles
--   prepends "n" instances of "fill" to the signal
delay :: Integer -> Bool -> [Bool] -> [Bool]
delay n fill s = (set n fill) ++ s

-- the inverter gate inverts every level for the entire signal,
-- and also delays the signal by one clock cycle
not_gate :: [Bool] -> [Bool]
not_gate s = delay 1 True (map not s)

-- a clock can be obtained by connecting an inverter's output to its input
--         |\       Clock
--      +--| >o----+--
--      |  |/      |
--      |           |
--      +-----------+
clock :: [Bool]
clock = not_gate clock

-- and gate delays its output by 2 clock cycles
and_gate :: [Bool] -> [Bool] -> [Bool]
and_gate i1 i2 = delay 2 True (zipWith (&&) i1 i2)
                               -- "zipWith" is similar to map
                                                -- but it takes a binary operator
                                                -- and two lists, and produces
                                                -- a list of results of the binary
                                                -- operator being applied to corresponding
                                                -- elements in the argument list
                                                -- (&&) is the conjunction operator

--  A /SR latch (http://en.wikipedia.org/wiki/Latch_%28electronics%29)
--     is a device made up of 4 gates and 4 wires. Notice how the
--     signals on the wires are defined in terms of outputs of the
--     corresponding gates.
--              ____
--     /S ----|    \   w1 |\       Q
--            |     |-----| >o----+--
--         +--|____/      |/      |
--         |                      |
--         +--------------------+ |
--                              | |
--         +--------------------  -+
--         |   ____               |
--         +--|    \   w2 |\    |  /Q
--            |     |-----| >o--+----
--     /R ----|____/      |/
--
```

```
srneg_latch :: [Bool]->[Bool]->[Bool]
srneg_latch s r =
  let (q,qbar,w1,w2) = (not_gate w1,not_gate w2,and_gate s qbar,and_gate r q) in q


-- Consider now the following signals: (each dash represents 1 clock cycle)
--
--                +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
--  /S      6     |
--     -+-+-+-+-+-+
--
--     -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+           +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
--  /R             15                |     6     |
--                                   +-+-+-+-+-+-+
--
--  The following expression, when evaluated at the top level (you will have to
--  cut and paste), will return the the Q signal of the latch for the first 40 clock cycles
--
--  take 40 (srneg_latch ((set 6 False)++high) ((set 15 True)++(set 6 False)++high))
```

The code given above implements the components that allow the simulation of digital circuits: logic gates, clocks, and latches. The essential element of the simulation is the *wire*, which is implemented as an infinite Haskell stream. The hardware primitives mentioned above define simultaneous relationships between the wires (with delay) via tuple pattern matching.

These components can be regarded as the primitives of a hardware simulation language.

Using the given components, and possibly implementing some new components of your own (one suggestion would be JK flip-flops, implement a simulation of a synchronous four-bit counter circuit (http://en.wikipedia.org/wiki/Counter). Write Haskell expressions that demonstrate that your implementation works (i.e. connect your counter to a clock, and showcase the 4 outputs of the counter).

Problem 2 [4 marks, submit to PS6P02]

Python Generators and Lazy Infinite Streams

In this question, we shall explore a very interesting feature of Python, called ***generators***, which are implemented via the statement **`yield`**. The original purpose of this statement was to allow the implementation of *iterators*, which are a central feature of Python loops. The **`yield`** statement can also be used to implement *co-routines* (simulated concurrency), which will be explored in a subsequent problem set. However, our concern for this exercise is to ***find an equivalent to Haskell's lazy streams***.

A ***Python generator*** is a procedure which, instead of **`return`** statements, only has **`yield`** statements. Similar to the **`return`** statement, the **`yield`** statement stops the execution of the procedure and returns a value to the outer environment. Nevertheless, in the case of the **`yield`** statement, the *state of the procedure is preserved*, and subsequent invocations will continue the execution of the procedure with the statement appearing right after the last executed **`yield`**.

In fact, the above description is a bit over-simplified. When the generator (i.e the **`yield`**-containing procedure) is called, the procedure returns immediately, and the return value is ***a handle***, that is, an object that can be passed to the procedure **`next()`**, which returns, for every call, the values placed in **`yield`** statements. The following example illustrates this concept.

```
>>> def ints():  # returns a generator of natural numbers
     n = 1 ;
     while True :
          yield n # returns n but keeps procedure's state active
          n += 1  # continues from here on subsequent calls

>>> intgen = ints() # intgen is the generator's handle
>>> next(intgen) # execute generator up to the the first yield
1
>>> next(intgen) # execute from n += 1 up to next execution of yield
2
>>> next(intgen) # can be called an unbounded # of times
3
>>> next(intgen)
4
>>> next(intgen)
5
>>>
```

Let us now consider an alternative definition that generates the same sequence of values:

```
def ints2(k):
     yield k
     g = ints2(k+1)
     while True :
          yield next(g)
>>> i = ints(1)
>>> next(i)
1
>>> next(i)
2
```

The definition above resembles the following Haskell stream:

```
ints k = k:(ints (k+1))
```

This hints at the fact that there might be a systematic translation scheme of Haskell stream definitions into Python generators. The Python generators would be expected to generate all the elements of the Haskell stream in the same order upon successive calls to **`next()`**. More hints of this translation scheme would be provided by the following example:

```
def tail(g): # no haskell equivalent since no pattern matching
        def p():
                h = g()
                next(h)
                for x in h:
                    yield x
        return p

def head(g): # no haskell equivalent since no pattern matching
        return next(g())

# haskell: map f l = (f (head l)):(map f (tail l))
def mapgen(f,g):
        def rets():
                yield f(head(g))
                for x in mapgen(f,tail(g))() : yield x
        return rets

# haskell: ints = 1:map (\x -> x+1) ints
def ints():
        yield 1
        h=mapgen(lambda x:x+1,ints)()
        for x in mapgen(lambda x:x+1,ints)(): yield x
```

If you look carefully at the functions **`mapgen`** and **`ints`**, they can be obtained by direct syntactic translation from their Haskell correspondents. (do not worry about the **`head`** and **`tail`** functions, since we regard them as primitives of the stream language).

Your job for this exercise is to find the systematic translation scheme that converts Haskell stream definitions into Python generators. Then, using the scheme you discovered, implement the equivalent of the **`zipWith`** Haskell function, and use **`zipWith`** to define the Fibonacci stream.

Your submission must be a Python file that contains your **`zipWith`** implementation, along with the definition of the Fibonacci stream. Your file must also provide a description of the systematic translation scheme, in the form of a comment at the beginning of the file.

Further Practice Problems

These problems are for your own individual practice. Solutions are not to be submitted, and will not be marked. You are, however, allowed to post your solutions in the forum for comparison and discussion. Good posts will earn marks.

Further Practice Problem 1

Devise a systematic translation scheme of Haskell streams into C.