CPSC 201: Introduction to Computer Science
Carsten Schürmann
Date: February 27, 2002

Homework 2

Due: Wednesday, February 13, 2002.

Guidelines

While we acknowledge that beauty is in the eye of the beholder, you should nonetheless strive for elegance in your code. Not every program which runs deserves full credit. Make sure to state invariants in comments which are sometimes implicit in the informal presentation of an exercise. If auxiliary functions are required, describe concisely what they implement. Do not reinvent wheels, and try to make your functions small and easy to understand. Use tasteful layout and avoid long winded and contorted code. None of the problems requires more than a few lines of SML code.

Problem 1: Circuit Simulator (50 points)

For this problem, consider yourself working for a testing company who is making a lot of money with testing circuit designs. Your companies major strength is that it has a very efficient format to define circuits. However, in order to stay competitive, your company is looking for a simulator, that can test such a circuit. This is where you come into the game.

Circuits are represented by the following datatype declaration.

```
datatype Circuit
  = NOTGate  of Circuit
  | NANDGate of Circuit * Circuit
  | NORGate  of Circuit * Circuit
  | BIND     of Circuit * (Circuit -> Circuit)
  | SIGNAL   of BOOL
```

The first three constants defined here are self explanatory, the `BIND` and `SIGNAL` however require some explanation. Let's address `BIND` first.

The `BIND` operator introduces a name for a point where a split in a connection occurs. Consider for example, the following circuit.

```
a-------|
        |NAND|----c----|
b-------|         |    |NAND|--- d
                  |----|
```

which can be represented using this representation as

```
fun circuit a b = BIND (NANDGate (a, b), fn c => NANDGate (c, c))
```

Note, how `fn c => NANDGate (c, c)` is a function with expects as an input a `Circuit`, named `c`. Don't be scared about the fact that `BIND` takes a function as argument. It is a very elegant way of doing things.

The `SIGNAL` construct will allow us to propagate a signal through a circuit. It can either be high or low, that is `I` or `O`, using the notation from class. To inialize this process, apply the circuit to the test signals, let's choose `I` and `O`, respectively, for this example. A circuit is closed when it is of type `Circuit`. In our example this would be `circuit (SIGNAL I) (SIGNAL O)`.

Write a function that simulates a closed circuit, and returns a truth value as result:

```
simulate : Circuit -> Bool
```

Hint: The answer to this problem is very short. Do not write any other function besides `simulate`. You won't need them.

```
(* simulate : Circuit -> Bool
   simulate G = B

   Invariant:
   If   C *)

fun simulate (BIND (C, C')) =
      simulate (C' (SIGNAL (simulate C)))
  | simulate (NOTGate C) =
      NOT (simulate C)
  | simulate (NANDGate (C1, C2)) =
      NAND (simulate C1) (simulate C2)
  | simulate (NORGate (C1, C2)) =
      NOR (simulate C1) (simulate C2)
  | simulate (SIGNAL B) =
      B
```

Problem 2: Memory (50 points)

Memory can be thought off as a long list of bits. 1KB memory would correspond to a list of 1024 bits. Bits can be either `0` or `1`, and therefore we can simply reuse the type `BOOL` from class. The state of memory is difficult to interpret, maybe even impossible to understand, when we are just seeing `O`'s and `I`'s. Nevertheless, we tend to attach meaning to the bits by saying, that these 8 bits form a byte, or these four bytes from a floating point number. In this assignment we write an interpretation function, that converts memory to a list of data objects, which are represented in memory, and convert them into human readable form. For now, we include only strings, characters, integers, real numbers, and — for simplicity — tags that contain information of what is what. The memory itself is represented as a list of `BOOL` values.

```
type memory = BOOL list
```

Question 1: (15 points)

Memory can therefore be interpreted as a byte list. We call this interpretation `layer1`. It can be expressed in ML the following way.

```
datatype BYTE
  = Byte of BOOL list
type layer1 = BYTE list
```

Write a function `layer1` that converts memory into a list of bytes.

```
layer1 : memory -> layer1
```

```
(* layer1 : memory -> layer1 *)


fun layer1 nil = nil
  | layer1 M =
    let
      val (B', M') = getbyte 8 M
    in
      Byte B' :: (layer1 M')
    end

and getbyte 0 M = ([], M)
  | getbyte n (b :: M) =
    let
      val (B, M') = getbyte (n-1) M
    in
      ((b :: B), M')
    end
```

Question 2: (35 points)

In order to attribute more meaning to the list of bytes, let us define four tags: for strings, for characters, for integers, and for reals.

```
datatype Tag
  = String                                (* byte 0 *)
  | Char                                  (* byte 1 *)
  | Integer                               (* byte 2 *)
  | Real                                  (* byte 3 *)
```

These tags determine the meaning of a bit list found in memory. We will use a tag that corresponds to the number 0 as `String`, with number 1 as `Char`, ... For memory layout, we follow the following conventions: A tag is always one byte, a string of length n contains $n+1$ bytes, where the first byte encodes the length of a string. Therefore, strings can be at most 255 bytes long. A character is one byte, an integer two, and a real number 2. How to convert integers and reals will be discussed in class.

Represented in SML/NJ, the datatype that corresponds to the content is defined as follows.

```
datatype Content
  = Ta of Tag
  | St of string
  | Ch of char
  | In of int
  | Re of real

type layer2 = Content list
```

Write a function `layer2` that converts `layer1` into content.

```
layer2 : layer1 -> layer2

fun byteToTag (Byte [0, 0, 0, 0, 0, 0, 0, 0]) = String
  | byteToTag (Byte [0, 0, 0, 0, 0, 0, 0, I]) = Char
  | byteToTag (Byte [0, 0, 0, 0, 0, 0, I, 0]) = Integer
  | byteToTag (Byte [0, 0, 0, 0, 0, 0, I, I]) = Real
  | byteToTag (Byte [0, 0, 0, 0, 0, I, 0, 0]) = Double


fun bitsToInteger nil = (0, 1)
  | bitsToInteger (b :: B) =
    let
      val (i, e) = bitsToInteger B
    in
      (case b
         of 0 => (i, 2*e)
          | I => (i+e, 2*e))
    end

fun getbyte 0 M = ([],M)
  | getbyte n (b :: M) =
    let
      val (B,M') = getbyte (n-1) M
    in
      ((b::B),M')
    end

fun layer2 nil = nil
  | layer2 (B :: Bs) =
    let
      val tag = byteToTag B
    in
      (case tag
         of String =>  layer2string Bs
          | Char =>    layer2char Bs
          | Integer => layer2integer Bs
          | Real =>    layer2real Bs)
    end
```

```
and layer2integer (Byte bs :: Byte bs' :: Bs) =
    let
      val (i, _) = bitsToInteger (bs@bs')
    in
      In i :: layer2 Bs
    end

and layer2char (Byte bs :: Bs) =
    let
      val (i, _) = bitsToInteger bs
    in
      Ch (Char.chr i) :: layer2 Bs
    end

and layer2string (Byte bs :: Bs) =
    let
      val (i, _) = bitsToInteger bs
      val (s, Bs') = layer2string' i Bs
    in
      St s :: layer2 Bs'
    end

and layer2string' 0 Bs = ("", Bs)
  | layer2string' n (Byte bs :: Bs) =
    let
      val (i, _)   = bitsToInteger bs
      val (s, Bs') = layer2string' (n-1) Bs
    in
      (String.str (Char.chr i) ^ s, Bs')
    end

and layer2real (Byte bs :: Byte bs' :: Bs) =
    let
      val (mantissa,exponent) = getbyte 10 (bs@bs')
      val mantissaSign = if (hd mantissa = I) then ~1.0 else 1.0
      val exponentSign = if (hd exponent = I) then ~1.0 else 1.0
      val (mantissaValue,_) = bitsToInteger (tl mantissa)
      val (exponentValue,_) = bitsToInteger (tl exponent)
    in
       Re (mantissaSign * (real)mantissaValue *
                         Math.pow(10, exponentSign * (real)exponentValue) :: layer2 Bs
    end
```

Hand-in Instructions

In the course directory /c/cs201/bin, there are five programs that support you in submitting your solution to the homework.

```
submit    assignment-number file(s)
unsubmit  assignment-number file(s)
check     assignment-number
protect   assignment-number file(s)
unprotect assignment-number file(s)
```

`submit` allows you to submit one of several files. For example

```
/c/cs201/bin/submit 1 hw1.sml hw1-examples.sml
```

submits your file `hw1.sml` and `hw1-examples.sml`. `check` can give you the peace of mind that your solution has really been submitted, and if you would like to make last minute changes to your solution, use `unsubmit` before submitting the updated version. `protect` and `unprotect` give you the power to protect/unprotect submitted solutions from deletion.