Solutions to Worksheet 4: Basic Haskell Programming

Due: Midnight, Monday, 21 March 2010.
Total marks: 53

Question 1: Some Task (8 marks)

The purpose of this question is to let you explore Haskell syntax without having to worry about being overly clever with algorithms. The computational task is simple: given two integers $x < y$, compute the function :

$$f(x,y) = \sum_{i=x}^{y} i$$

1. (2 marks) Using simple recursion, and if-then-else.

   Model Solution

```haskell
sum1 :: Integer -> Integer -> Integer
sum1 lo hi = if (lo == hi) then hi else lo + (sum1 (lo + 1) hi)
```

   The definition does not have to be on a single line, and can be laid out in a more traditional manner. It would be a waste of space, but space is cheap in a file.

2. (2 marks) Using simple recursion, and guarded clauses.

   Model Solution

```haskell
sum2 :: Integer -> Integer -> Integer
sum2 x y
  | x == y = y
  | otherwise = x + (sum2 (x+1) y)
```

   It's a very common typographical error to put a `=` on the top line of a definition using guarded clauses. Here's an alternative layout that helps to avoid this error:

```haskell
sum2 :: Integer -> Integer -> Integer
sum2 x y | x == y = y
         | otherwise = x + (sum2 (x+1) y)
```

   The guarded clauses are similar to a case or switch statement in other languages, and is a tidied up version of LISP's `cond` function.

3. (2 marks) Using tail recursion. Make your helper function local to the interface function. The helper function should use guarded clauses (not if-then-else), so that you can learn the use of white-space for scoping. Hint: keep the helper and interface separate until you are confident they both work, then edit the code so that the helper is local.

   Model Solution

```
sum3 :: Integer -> Integer -> Integer
sum3 x y = sum3h x y y
  where sum3h x y s
          | x == y = s
          | otherwise = sum3h (x+1) y (s + x)
```

The tail recursion is not too important, except to know that it can be done. Because of lazy evaluation, some of the savings you might expect from a tail recursive program are not obtained because Haskell has to represent the expression to be evaluated, and some of the need for them is also avoided.

4. (2 marks) Using tail recursion, and using patterns for your helper function. Hint: For functions on integers, your patterns will need to use at least one literal value (e.g., 0).

   Model Solution

```
sum4 :: Integer -> Integer -> Integer
sum4 x y = sum4h (y-x) x
  where sum4h 0 s = s
        sum4h n s = sum4h (n-1) (s+x+n)
```

Grading

Give full marks if the functions use the required syntax, and are accompanied by a correct type signature. If the required syntax is not used by any of the subparts, give zero marks for that subpart. I expect type signatures for every function, so a maximum of 4/8 if no type signatures are given.

Question 2: More Peano Practice (11 marks)

For more practice, we will implement the Peano numbers in Haskell. The algorithms are the same as for Prolog, but the syntax is of course a little different.

Define them as follows:

```
data Peano = Zero | S Peano
  deriving (Show)
```

The `deriving (Show)` clause tells Haskell to create automatic code to make `Peano` numbers part of the `Show` class, with a default operator which allows them to be displayed by the interactive Haskell system. Without the clause, your functions may in fact work, but the results won't be displayable.

1. (2 marks) Implement `add :: Peano -> Peano -> Peano`.

   Model Solution:

```
add :: Peano -> Peano -> Peano
add Zero x = x
add (S x) y = S (add x y)
```

Note: Because we are using a new datatype, patterns must be used. Guards and if-then-else cannot be used, because they presuppose operators like `(==)`. It is not valid to use automatic instance declarations for `Eq` or `Ord` either.

2. (2 marks) Implement `sub :: Peano -> Peano -> Peano`.

   Model Solution:

```
sub :: Peano -> Peano -> Peano

sub x Zero = x
sub (S x) (S y) = sub x y
sub _ _ = error "obscure error message about negative Peano numbers"
```

   Note: it is not necessary to test for the case that the first argument is greater than the second, since this case is handled implicitly by the third pattern.

3. (2 marks) Implement `multiply :: Peano -> Peano -> Peano`.

   Model Solution:

```
multiply :: Peano -> Peano -> Peano

multiply Zero _ = Zero
multiply (S x) y = add y (multiply x y)
```

4. (3 marks) Implement `equals :: Peano -> Peano -> Bool`.

   Model Solution:

```
equals :: Peano -> Peano -> Peano

equals Zero Zero = True
equals (S x) (S y) = equals x y
equals _ _ = False
```

   Note: Unlike Prolog, the function needs a catch-all for the case where `False` is returned. That's because in Prolog, the programs don't return "yes" or "no" – that's the response of the Prolog engine, not the program.

5. (2 marks) Make `Peano` a member of the `Eq` class using an `instance` declaration (you don't need to define the `Eq` class, as it is already defined).

   Model Solution:

```
instance Eq Peano where
  (==) = equals
```

Note: It was possible to define an equals-like function here, but it's simpler to use the one from the previous part.

Evaluation

For each part, give full marks if the functions are correct, and are accompanied by a correct type signature. Each missing type signature will result in a deduction of 1 mark. All definitions should use patterns, as shown, and it is not valid to derive automatic instance declarations for type classes like `Eq` or `Ord`. Deduct 2 marks total from the question if the data definition was changed to include `Eq` or `Ord`.

Question 3: The Zeno Zone (24 marks)

Note: Some significant revisions were made to this question. Please read carefully. Also, I tried to keep the ordering of the sub-parts consistent with the original ordering, but if you havne't started, it makes sense to do parts 3-6 first, then 1,2 and the bonus question using the implementations of parts 3-6.

The Zeno numbers are a variation of the Peano numbers, defined as follows:

```haskell
data ZenoBit = One | Z ZenoBit
  deriving (Show)

type Zeno = [ZenoBit]
```

In words, a Zeno number is a list of ZenoBits. Each ZenoBit represents a power of 2: `One` represents the integer 1, `Z One` represents the integer 2, `Z (Z One)` represents the integer 4, and in general, `Z x` represents the integer $2x$ (where `x` must be a ZenoBit).

The best way to understand Zeno numbers is as a variation of the binary number system. Each ZenoBit is a power of 2. The list of ZenoBits in this question corresponds roughly to a sequence of 0s and 1s in a hardware register inside a real machine. One notable difference is that there is no pre-determined limit to a Zeno number: it can be as large as you have space to store. Another difference is that the hardware representation has to store 0s. The Zeno number only stores the 1s.

The number 5 is represented as a Zeno number using a list of ZenoBits: `[Z (Z One), One]`. The definition above does not prevent you from using `[One, One, One, One, One]` to represent the number 5, but you should take care not to allow duplicate ZenoBits in your Zeno numbers. The number 0 is represented as the empty list `[]`, and we will not bother to represent negative numbers in this exercise.

Note: there is nothing especially practical or useful about this representation. The exercise is useful for what you can learn about Haskell by working with these numbers.

1. (4 marks) Implement `add :: Zeno -> Zeno -> Zeno`.

   Model Solution:

```haskell
-- Note that this implementation assumes Eq ZenoBit and Ord ZenoBit
-- (parts 3-6 below)
```

```
add :: Zeno -> Zeno -> Zeno

add [] x = x
add x [] = x
add (x:xs) (y:ys)
  | x == y     = add_helper (Z x) (add xs ys)
  | x < y      = add_helper y (add (x:xs) ys)
  | otherwise = add_helper x (add xs (y:ys))
    where add_helper b []   = [b]
          add_helper b (c:zs)
            | b == c     = (Z b) : zs
            | otherwise = b:c:zs
```

Comments This was the trickiest part of the homework. My solution used `Eq ZenoBit` and `Ord ZenoBit`, which I show in handout-order, but which I did before doing this part.

The tricky part here is handling the carry. For example, if you are adding the number `[Z (Z (Z One)),Z (Z One),Z  One, One]` to the number `One`, then the carry bit is only known for sure when you reach the end of the longer number, and the carry bit affects every bit in the first number.

The way I handle it is by not committing to any bit until I see whether there will be a carry bit. This is done by passing what might be the current bit to a helper function `add_helper`, which also receives the result (the second argument to `add_helper)` of the recursive addition. The helper function can then decide what to do based on what the first bit (c) is in the remaining sum. If the first bit in the list is the same as the bit (b) I am deciding about, then I have to do the carry. Otherwise, I just put the bit b on the front of the whole list.

I have already promised to grant full marks to solutions that are significantly different. In particular, a simple approach is to append the two numbers together, sort them, and then process the result, handling duplications. The first 2 steps of this process are easy, but the processing of duplicates is no simpler than the case analysis I gave in my solution. So I don't really consider this a vast simplification.

You can simplify things a little more if you reverse the lists first. Then processing the duplicates can be done using a function that passes bits into the recursive call. This does seem to be a bit easier.

To grade this part of the question, there has to be a type signature. There has to be an implementation that does actually computes addition, one way or another. Deduct one mark if the function doesn't remove duplicates, and deduct 2 marks if the function doesn't compute the sum. Note that `(++)` actually computes a list whose value is equal to the desired sum, which is worth 2 out of 3 marks (excluding the type signature).

The main purpose of this question was to work with data types and making instances of type classes. The algorithm here is not particularly important, so we won't penalize inefficient solutions.

2. (4 marks) Implement `multiply :: Zeno -> Zeno -> Zeno`.

   Model Solution:

```haskell
-- a helper function for multiplying ZenoBits
multZBits :: ZenoBit -> ZenoBit -> ZenoBit

multZBits One x = x
multZBits (Z x) y = Z (multZBits x y)

-- the main function
multiply :: Zeno -> Zeno -> Zeno

multiply [] y = []
multiply (x:xs) y = add (map (multZBits x) y) (multiply xs y)
```

Comments: Note the use of `map` here, which multiplies every ZenoBit in the second list by the first ZenoBit of the other number. Of course, `map` was not required, and it's fine to have a second recursive function to do that job.

With a well-behaved `add` function on Zeno numbers, this is pretty simple.

3. (6 marks) Implement `equals :: ZenoBit -> ZenoBit -> Bool`.

   Note: this is a major revision to the question. You may have already implemented `equals` for previous parts. If so, just rename it. There's no need to implement it a second time.

   Model Solution:

   ```haskell
   equals :: ZenoBit -> ZenoBit -> Bool

   equals One One = True
   equals (Z x) (Z y) = equals x y
   equals _ _ = False
   ```

   As noted above, I coded this part before part 1, so that I could use the next step in part 1 as well.

4. (2 marks) Make `ZenoBit` a member of the `Eq` class using an `instance` declaration (you don't need to define the `Eq` class, as it is already defined).

   You can use the function `equals` as the implementation for `(==)`.

   Model Solution:

   ```haskell
   instance Eq ZenoBit where
     (==) = equals
   ```

5. (6 marks) Implement `zbleq :: ZenoBit -> ZenoBit -> Bool`, which tests if a ZenoBit number is less then or equal to another ZenoBit number.

   Note: The notes and the text have an error, which states that default operators for `Ord` are defined in terms of `(<)`. A correct implementation of the default operators is specified by giving an implementation for `(<=)`. Since the next part of the question is to make ZenoBit an instance of the Ord class, you'll need `zbleq`. You may have already implemented some kind of comparison function (e.g., greater than, or less than) for ZenoBit numbers in previous parts. If so, you may want to revise the functions add and multiply to use the operators defined by Eq and Ord.

Model Solution:

```haskell
zbleq :: ZenoBit -> ZenoBit -> Bool

zbleq One One = True
zbleq One (Z _) = True
zbleq (Z _) One = False
zbleq (Z x) (Z y) = zbleq x y
```

6. (2 marks) Make `ZenoBit` a member of the `Ord` class using an `instance` declaration (you don't need to define the `Ord` class, as it is already defined).

   Note: By now you're tired of notes. I just wanted to refer to the previous part, and the error mentioned there. If you are reading this out of order, the pointer might be useful. To answer this part, use the `zbleq` function as the implementation for `(<=)`.

   Model Solution:

```haskell
instance Ord ZenoBit where
  (<=) = zbleq
```

7. (Bonus: 3 marks) Implement `greaterThan :: Zeno -> Zeno -> Bool`, which tests if a Zeno number is greater than or equal to another ZenoBit number.

   Model Solution:

```haskell
greaterThan :: Zeno -> Zeno -> Bool
greaterThan [] [] = False
greaterThan [] (_:_) = False
greaterThan (_:_) [] = True
greaterThan (x:xs) (y:ys)
  | x > y  = True
  | x == y = greaterThan xs ys
  | otherwise = False
```

   Comment: My solution shows the list processing solution. However, once `Eq ZenoBit` and `Ord ZenoBit` have been defined, lists of ZenoBits are already comparable using `(>)`, since the standard Haskell library (Prelude) defines `Ord a => Ord [a]`; this is another way of saying that the list class is already an instance of `Ord` provided that the contained type is `Ord`. So it was sufficient to write:

```haskell
greaterThan :: Zeno -> Zeno -> Bool
greaterThan = (>)
```

Evaluation

For each part, you will receive full marks if your function is correct, and is accompanied by a correct type signature. Each missing type signature will result in a deduction of 1 mark. Marking for part 1 should be generous, and the rest of the question is fairly easy.

Question 4: Exercising the Lambda Nature (10 marks)

Consider the following function[1] definitions (in Haskell syntax):

```
frotz x y = 2*x + y
xyzzy f a b = f b a
```

The first function is not very interesting, and was intended to be asymmetric. The second function is more interesting: it "swaps" the arguments for a given function f.

Note: I used an exercise like this last year, and the year before, always with similar functions, but different names. So submissions that borrow too much for solutions to previous years' assignments are likely to have weird inconsistencies.

1. (1 mark) `frotz`

   Solution:
   $$frotz = \lambda x.\lambda y.(2 \times x + y)$$

2. (1 mark) `xyzzy`

   Solution:
   $$xyzzy = \lambda f.\lambda a.\lambda b.(f\ b\ a)$$

3. (2 marks) `frotz 4`

   Solution:
   $$\begin{aligned} & frotz\ 4 \\ &= (\lambda x.\lambda y.(2 \times x + y))\ 4 \\ &= \lambda y.(8 + y) \end{aligned}$$

   Marking: Two marks. One for substituting for `frotz`, and one for using the application rule on the argument 3. The result must be a function.

4. (2 marks) `xyzzy frotz`

[1]The words frotz and xyzzy are nonsense words, associated with very early text-based computer games. The word xyzzy is the name of a spell in the game *Colossal Cave Adventure*. One of the in-jokes of the game was that when you learned a spell, you weren't told what it did, so you had to exeriment. The word frotz is associated with games from a company called InfoCom, and I remember it being the name of a spell as well. However, I can't seem to find any evidence for that memory, so maybe I am remembering incorrectly. You can find both words in the Hacker's Dictionary.

Solution:

$$\begin{array}{rl} & xyzzy\ frotz \\ = & (\lambda f.\lambda a.\lambda b.(f\ b\ a))\ frotz \\ = & \lambda a.\lambda b.(frotz\ b\ a) \end{array}$$

Marking: Two marks. One for substituting for `xyzzy`, and one for using the application rule on the argument `frotz`. The result must be a function.

Note: This is where Haskell would stop evaluating this expression, (unless there were some reason to carry on). It is possible (and acceptable in the lambda calculus) to carry on by using the application rule for for $frotz$ inside the body of the result here. We do that below.

5. (2 marks) `xyzzy frotz 4`

   Solution:

$$\begin{array}{rl} & xyzzy\ frotz\ 4 \\ = & (\lambda f.\lambda a.\lambda b.(f\ b\ a))\ frotz\ 4 \\ = & \lambda a.\lambda b.(frotz\ b\ a)\ 4 \\ = & \lambda b.(frotz\ b\ 4) \end{array}$$

   Marking: Two marks. One for substituting for `xyzzy`, and applying it to the argument `frotz`, and the second mark for applying the result to the argument 4. The result must be a function. The answer may use the result of the previous part, in which case the first mark is "free".

   Again, this is where Haskell would stop. However, as in the previous example, it is entirely reasonable to replace $frotz$ with its lambda expression, and use the application rule to simplify.

6. (2 marks) `xyzzy (xyzzy frotz)`

   Solution: Some of this is repeated from the above. We could start with the result of a previous part, but I will show a different route:

$$\begin{array}{rl} & xyzzy\ (xyzzy\ frotz) \\ = & (\lambda f.\lambda a.\lambda b.(f\ b\ a))\ (xyzzy\ frotz) \\ = & \lambda a.\lambda b.((xyzzy\ frotz)\ b\ a)) \end{array}$$

   Note: This is where Haskell would stop, using lazy evaluation. However, the question implied that you should carry on to show that

$$xyzzy\ (xyzzy\ frotz) = frotz$$

   At this point, we could use the value of $xyzzy\ frotz$ evaluated above:

$$(xyzzy\ frotz) = \lambda a.\lambda b.(2 \times b + a)$$

The parameter names are irrelevant, and we can rewrite this expression using any consistent rewriting (i.e., an α-conversion), e.g.

$$\lambda a.\lambda b.(2 \times b + a) = \lambda u.\lambda v.(2 \times v + u)$$

So, we can carry on from where we left off above::

$$\begin{array}{ll} \multicolumn{2}{l}{xyzzy\ (xyzzy\ frotz)} \\ \quad = & \lambda a.\lambda b.((xyzzy\ frotz)\ b\ a)) \\ \quad = & \lambda a.\lambda b.((\lambda u.\lambda v.(2 \times v + u))\ b\ a)) \\ \quad = & \lambda a.\lambda b.((\lambda v.(2 \times v + b))\ a)) \\ \quad = & \lambda a.\lambda b.(2 \times a + b) \end{array}$$

Again, the parameter names are irrelevant; the final result is a function that is identical to $frotz$.

We could generalize this proof to show that $xyzzy\ (xyzzy\ g) = g$. This is not actually an astounding result of itself. Probably more significant is the idea that Haskell programs can be manipulated using simple (but tedious) mathematical operations.

Marking: Three marks. Give full marks, unless there is some mistake in the use of the application rule, or if the final result is not equivalent to the function `frotz`. Deduct a mark for any errors, up to 3 times.