COMP90048 Declarative Programming
Semester 1, 2018
Peter J. Stuckey

Declarative Programming

Answers to workshop exercises set 3.

QUESTION 1
If you were working on a program that functioned as a web server, and thus
its output was in the form of web pages, you could:

(a) have the program write out each part of the page as soon as it has decided
    what it should be;
(b) have the program generate the output in the form of a string, and then
    print the string;
(c) have the program generate the output in the form of a representation
    such as the HTML type of the previous questions, and then convert that
    to a string and then print the string.

Which of these approaches would you choose, and why?

ANSWER
The best choice for nearly all non-trivial applications will be (c).

There are several reasons for structuring data. One you are probably familiar with is so more efficient algorithms can be used (e.g. using trees to allow O(log N) search and update). Another is to incorporate more meaning (and to eliminate things which have no meaning). For example, the HTML data type can only represent correct HTML. Strings, on the other hand, can contain arbitrary character sequences, most of which are not
valid HTML. A function which returns the HTML data type is not GUARANTEED to be correct, but there are a whole class of bugs it cannot exhibit that functions that generate strings or do I/O directly CAN exhibit. The data type also leads to a natural way of structuring the code, which helps with correctness.

QUESTION 2

Implement a function ftoc :: Double -> Double, which converts a temperature
in Fahrenheit to Celsius. Recall that C = (5/9) * (F - 32).
What is the inferred type of the function if you comment out the type declaration? What does this tell you?

ANSWER

```
>ftoc :: Double -> Double
>ftoc f = (5/9) * (f - 32)
```

With the type declaration removed, the inferred type is:

```
ftoc :: Fractional a => a -> a
```

This means that the ftoc function would work for any "Fractional" type, such as Double or Float or Rational.

QUESTION 3
Implement a function quadRoots :: Double -> Double -> Double -> [Double], which computes the roots of the quadratic equation defined by
0 = a*x^2 + b*x + c, given a, b, and c. See
http://en.wikipedia.org/wiki/Quadratic_formula for the formula.
What is the inferred type of the function if you comment out the type declaration? What does this tell you?

ANSWER

```
>quadRoots :: Double -> Double -> Double -> [Double]
>quadRoots 0 0 _ = error "Either a or b must be non-zero"
>quadRoots 0 b c = [-c / b]
>quadRoots a b c
>     | disc < 0  = error "No real solutions"
>     | disc == 0 = [tp]
>     | disc > 0  = [tp + temp, tp - temp]
>     where   disc = b*b - 4*a*c
>             temp = sqrt(disc) / (2*a)
>             tp   = -b / (2*a)
```

With the type declaration removed, the inferred type is:

```
quadRoots :: (Floating a, Ord a) => a -> a -> a -> [a]
```

Floating types are more general than Fractional types, as the latter includes irrational types, while the former are purely rational.

QUESTION 4

Write a Haskell function to merge two sorted lists into a single sorted list

ANSWER

```
>merge :: Ord a => [a] -> [a] -> [a]
>merge [] ys = ys
>merge (x:xs) [] = x:xs
>merge (x:xs) (y:ys)
>    | x <= y = x : merge xs (y:ys)
>    | x >  y = y : merge (x:xs) ys
```

We could code it somewhat better, e.g. by using @ patterns to avoid repeating some expressions (see later lecture). Here we keep it simple. Note that since we compare list elements, we must constrain the list element type to be a member of the Ord type class.

QUESTION 5

Write a Haskell version of the classic quicksort algorithm for lists. (Note that while quicksort is a good algorithm for sorting arrays, it is not
actually that good an algorithm for sorting lists; variations of merge sort
generally perform better. However, that fact has no bearing on this exercise.)

ANSWER

Here is one version of this function:

```
>qsort1 :: (Ord a) => [a] -> [a]
>qsort1 [] = []
>qsort1 (pivot:xs) = qsort1 lesser ++ [pivot] ++ qsort1 greater
>    where
>    lesser  = filter (< pivot)  xs
>    greater = filter (>= pivot) xs
```

QUESTION 6

Given the following type definition for binary search trees from lectures,

```
>data Tree k v = Leaf | Node k v (Tree k v) (Tree k v)
```

```
>        deriving (Eq, Show)
```

define a function

```
>same_shape :: Tree a b -> Tree c d -> Bool
```

which returns True if the two trees have the same shape: same arrangement of nodes and leaves, but possibly different keys and values in the nodes.

ANSWER

```
>same_shape Leaf Leaf = True
>same_shape Leaf (Node _ _ _ _) = False
>same_shape (Node _ _ _ _) Leaf = False
>same_shape (Node _ _ l1 r1) (Node _ _ l2 r2)
>   = same_shape l1 l2 && same_shape r1 r2
```

QUESTION 7
Consider the following type definitions, which allow us to represent expressions containing integers, variables "a" and "b", and operators for addition, subtraction, multiplication and division.

```
>data Expression
>       = Var Variable
>       | Num Integer
>       | Plus Expression Expression
>       | Minus Expression Expression
>       | Times Expression Expression
>       | Div Expression Expression

>data Variable = A | B
```

For example, we can define exp1 to be a representation of 2*a + b as follows:

```
>exp1 = Plus (Times (Num 2) (Var A)) (Var B)
```

Write a function eval :: Integer -> Integer -> Expression -> Integer which takes the values of a and b and an expression, and returns the value of the expression. For example eval 3 4 exp1 = 10.

ANSWER

```
>eval a b (Var A) = a
```

```
>eval a b (Var B) = b
>eval a b (Num n) = n
>eval a b (Plus  e1 e2) = (eval a b e1) + (eval a b e2)
>eval a b (Minus e1 e2) = (eval a b e1) - (eval a b e2)
>eval a b (Times e1 e2) = (eval a b e1) * (eval a b e2)
>eval a b (Div   e1 e2) = (eval a b e1) `div` (eval a b e2)
```