Programming Languages: Functional Programming Practicals 0: Functions and Definitions

Shin-Cheng Mu

Sep. 17, 2020

You should have installed GHC, with its commandline interface GHCi. Open your favourite text editor, create a new plain text file. The filename extension must end in `.hs`. This will be your working file for this practical. Type `ghci <filename>.hs` in the command line to load the working file into GHCi.

1. Define a function $myeven :: \mathsf{Int} \to \mathsf{Bool}$ that determines whether the input is an even number. You may use the following functions:

   $$\begin{aligned} &mod :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} \ , \\ &(==) \ :: \mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool} \ . \end{aligned}$$

   (Types of the functions written above are not in their most general form.)

   Solution:

   $$\begin{aligned} &myeven :: \mathsf{Int} \to \mathsf{Bool} \\ &myeven\ x = x \ `mod`\ 2 == 0 \ . \end{aligned}$$

2. Define a function that computes the area of a circle with given radius r (using $22 \,/\, 7$ as an approximation to π). The return type of the function might be Float.

   Solution:

   $$\begin{aligned} &area \ :: \mathsf{Float} \to \mathsf{Float} \\ &area\ r = (22 \,/\, 7) \times r \times r \ . \end{aligned}$$

3. Part-time students in Institute of Information Science are paid NTD 130 per hour. Define a function $payment :: \mathsf{Int} \to \mathsf{Int}$ that, when applied to the numbers of weeks a student work, compute the amount of money the Institute has to pay the student.

(a) Assume that there are five working days in a week, eight working hours per day. Define *payment*. For clarity, use **let** to define local variables recording number of days worked, number of hours worked, etc.

Solution:

```
payment :: Int → Int
payment weeks = let days  = 5 × weeks
                    hours = 8 × days
                in 130 × hours .
```

(b) Define *payment* again, but declare the local variables using **where**. Which style do you prefer?

Solution:

```
payment :: Int → Int
payment weeks = 130 × hours
  where hours = 8 × days
        days  = 5 × weeks .
```

(c) The regulation states that students are considered workers, and if a worker works for more than 19 weeks, the Institute has to pay, in addition to the salary, health insurance and pension reserves for the worker. The amount is 6% of the worker's salary.

Update definition of *payment* in the form:

```
payment :: Int → Int
payment weeks | weeks > 19 = ...
              | otherwise  = ...
```

You may need a function *fromIntegral* to convert Int to Float, and a function *round* that rounds a floating point number to the nearest integer.

In this case, should you use **let** or **where**?

Solution:

```
payment :: Int → Int
payment weeks | weeks > 19 = round (fromIntegral baseSalary × 1.06)
              | otherwise  = baseSalary
  where baseSalary = 130 × hours
        hours = 8 × days
        days  = 5 × weeks .
```

For this situation, **where** works better than **let**, since we want the scope of *baseSalary* to extend to both guarded branches.

4. More on **let**.

   (a) Guess what the value of *nested* would be. Type it into your working file and evaluated in in GHCi to see whether you guessed right. Note that indentation matters.

```
nested :: Int
nested = let x = 3
         in (let x = 5
             in x + x) + x  .
```

Solution: *nested* evaluates to 13, since the x in $x + x$ refers to 5 and the x in .. + x refers to 3.

   (b) Guess what the value of *recursive* would be. Try it in GHCi.

```
recursive :: Int
recursive = let x = 3
            in let x = x + 1
               in x  .
```

Solution: The computation does not terminate, since the x in $x + 1$ refers to itself.

5. Type in the definition of *smaller* into your working file.

```
smaller :: Int → Int → Int
smaller x y = if x ⩽ y then x else y  .
```

   Then try the following:

   (a) In GHCi, type `:t smaller` to see the type of *smaller*.

   (b) Try applying it to some arguments, e.g. *smaller* 3 4, *smaller* 3 1.

   (c) Use `:t` to see the type of *smaller* 3 4.

   (d) Use `:t` to see the type of *smaller* 3.

   (e) In your working file, define a new function *st3* = *smaller* 3.

   (f) Find out the type of *st3* in GHCi. Try *st3* 4, *st3* 1. Explain the results you see.

6. More practice on curried functions.

(a) Define a function *poly* such that *poly a b c x* = $a \times x^2 + b \times x + c$. All the inputs and the result are of type *Float*.

(b) Reuse *poly* to define a function *poly1* such that *poly1 x* = $x^2 + 2 \times x + 1$.

(c) Reuse *poly* to define a function *poly2* such that *poly2 a b c* = $a \times 2^2 + b \times 2 + c$.

Solution:

```
poly :: Float → Float → Float → Float → Float
poly a b c x = a × x × x + b × x + c

poly1 :: Float → Float
poly1 = poly 1 2 1

poly :: Float → Float → Float → Float
poly2 a b c = poly a b c 2
```

7. Type in the definition of *square* in your working file.

(a) Define a function *quad* :: Int → Int such that *quad x* computes x^4.

Solution:

```
quad :: Int → Int
quad x = square (square x) .
```

(b) Type in this definition into your working file. Describe, in words, what this function does.

```
twice      :: (a → a) → (a → a)
twice f x = f (f x) .
```

(c) Define *quad* using *twice*.

Solution:

```
quad :: Int → Int
quad = twice square .
```

8. Replace the previous *twice* with this definition:

```
twice   :: (a → a) → (a → a)
twice f = f · f .
```

(a) Does *quad* still behave the same?

(b) Explain in words what this operator $(\cdot)$ does.

9. Functions as arguments, and a quick practice on sectioning.

   (a) Type in the following definition to your working file, without giving the type.

   $$forktimes\ f\ g\ x = f\ x \times g\ x\ .$$

   Use : *t* in GHCi to find out the type of *forktimes*. You will end up getting a complex type which, for now, can be seen as equivalent to

   $$(t \to Int) \to (t \to Int) \to t \to Int\ .$$

   Can you explain this type?

   (b) Define a function that, given input x, use *forktimes* to compute $x^2 + 3 \times x + 2$. **Hint**: $x^2 + 3 \times x + 2 = (x + 1) \times (x + 2)$.

   Solution:

   $$compute :: \text{Int} \to \text{Int}$$
   $$compute = forktimes\ (+1)\ (+2)\ .$$

   (c) Type in the following definition into your working file: $lift2\ h\ f\ g\ x = h\ (f\ x)\ (g\ x)$. Find out the type of *lift2*. Can you explain its type?

   Solution:

   $$lift2 :: (a \to b \to c) \to (d \to a) \to (d \to b) \to d \to c\ .$$

   (d) Use *lift2* to compute $x^2 + 3 \times x + 2$.

   Solution:

   $$compute :: \text{Int} \to \text{Int}$$
   $$compute = lift2\ (\times)\ (+1)\ (+2)\ .$$

10. Let the following identifiers have type:

$$f :: Int \to Char$$
$$g :: Int \to Char \to Int$$
$$h :: (Char \to Int) \to Int \to Int$$
$$x :: Int$$
$$y :: Int$$
$$c :: Char$$

Which of the following expressions are type correct?

1. $(g \cdot f)\ x\ c$
2. $(g\ x \cdot f)\ y$
3. $(h \cdot g)\ x\ y$
4. $(h \cdot g\ x)\ c$
5. $h \cdot g\ x\ c$

You may type the expressions into Haskell and see whether they type check. To define f, for example, include the following in your working file:

$$\begin{aligned} &f :: \mathit{Int} \to \mathit{Char} \\ &f\ =\ \mathit{undefined} \end{aligned}$$

However, it is better if you can explain why the answers are as they are.

Solution:

1. $(g \cdot f)\ x\ c$. We calculate:

$$\begin{aligned} & (g \cdot f)\ x\ c \\ = \quad & \{\ \text{function application binds to the left}\ \} \\ & ((g \cdot f)\ x)\ c \\ = \quad & \{\ \text{definition of } (\cdot)\ \} \\ & (g\ (f\ x))\ c\ \ . \end{aligned}$$

   One can then see that there is a type error: $f\ x$ is a Char, while g expects Int as its first argument.

2. $(g\ x \cdot f)\ y$. We calculate:

$$\begin{aligned} & (g\ x \cdot f)\ y \\ = \quad & \{\ \text{definition of } (\cdot)\ \} \\ & g\ x\ (f\ y)\ \ , \end{aligned}$$

   which type-checks because

   - we have $g :: \text{Int} \to \text{Char} \to \text{Int}$, and
   - x is an Int, thus $g\ x :: \text{Char} \to \text{Int}$.
   - Furthermore, $f\ y$ is a Char. Thus $g\ x\ (f\ y) :: \text{Int}$.

   The result type is Int.

3. $(h \cdot g)\ x\ y$. We calculate:

$$\begin{aligned} & (h \cdot g)\ x\ y \\ = \quad & \{ \text{ function application binds to the left } \} \\ & ((h \cdot g)\ x)\ y \\ = \quad & \{ \text{ definition of } (\cdot) \} \\ & (h\ (g\ x))\ y \quad . \end{aligned}$$

   Now we reason:

   - Recall $h :: (\text{Char} \to \text{Int}) \to \text{Int} \to \text{Int}$.
   - Since $g :: \text{Int} \to \text{Char} \to \text{Int}$ and $x :: \text{Int}$, we have $g\ x :: \text{Char} \to \text{Int}$
   - Thus $h\ (g\ x) :: \text{Int} \to \text{Int}$.
   - Since $y :: \text{Int}$, we have $(h\ (g\ x))\ y :: \text{Int}$.

   Thus the expression type-checks, with type Int.

4. $(h \cdot g\ x)\ c$. We calculate:

$$\begin{aligned} & (h \cdot g\ x)\ c \\ = \quad & \{ \text{ definition of } (\cdot) \} \\ & h\ (g\ x\ c) \quad . \end{aligned}$$

   We reason:

   - The part $g\ x\ c$ type-checks, since $g :: \text{Int} \to \text{Char} \to \text{Int}$, $x :: \text{Int}$, and $c :: \text{Char}$. We have that $g\ x\ c :: \text{Int}$.
   - However, h::(Char $\to$ Int) $\to$ Int $\to$ Int expects a function of type Char $\to$ Int as an argument, not Int. Thus the expression fails to type-check.

5. $h \cdot g\ x\ c$. Similar to the reasoning above, $g\ x\ c :: \text{Int}$. However, function composition $(\cdot)$ expect to compose functions together, and $g\ x\ c$ is not a function. Thus the expression fails to type-check.