Programming Languages: Functional Programming Practicals 1: Functions and Definitions

Shin-Cheng Mu

Sep. 04, 2025

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

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

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

   Solution:

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

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:

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

3. Part-time students in Institute of Information Science are paid NTD 130 per hour. Define a function $payment :: \mathrm{Int} \to \mathrm{Int}$ 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:

$$
\begin{array}{l}
\mathit{payment} :: \mathrm{Int} \to \mathrm{Int} \\
\mathit{payment}\ \mathit{weeks} = \mathbf{let}\ \mathit{days}\ \ = 5 \times \mathit{weeks} \\
\qquad\qquad\qquad\qquad\quad\ \ \mathit{hours} = 8 \times \mathit{days} \\
\qquad\qquad\qquad\quad\ \ \mathbf{in}\ 130 \times \mathit{hours}\ .
\end{array}
$$

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

Solution:

$$
\begin{array}{l}
\mathit{payment} :: \mathrm{Int} \to \mathrm{Int} \\
\mathit{payment}\ \mathit{weeks} = 130 \times \mathit{hours} \\
\quad \mathbf{where}\ \mathit{hours} = 8 \times \mathit{days} \\
\qquad\qquad\ \ \mathit{days}\ \ = 5 \times \mathit{weeks}\ .
\end{array}
$$

(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:

$$
\begin{array}{l}
\mathit{payment} :: \mathrm{Int} \to \mathrm{Int} \\
\mathit{payment}\ \mathit{weeks} \mid \mathit{weeks} > 19 = \ldots \\
\qquad\qquad\qquad\ \ \mid \mathit{otherwise}\ \ = \ldots
\end{array}
$$

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:

$$
\begin{array}{l}
\mathit{payment} :: \mathrm{Int} \to \mathrm{Int} \\
\mathit{payment}\ \mathit{weeks} \mid \mathit{weeks} > 19 = \mathit{round}\ (\mathit{fromIntegral}\ \mathit{baseSalary} \times 1.06) \\
\qquad\qquad\qquad\ \ \mid \mathit{otherwise}\ \ = \mathit{baseSalary} \\
\quad \mathbf{where}\ \mathit{baseSalary} = 130 \times \mathit{hours} \\
\qquad\qquad\ \ \mathit{hours} = 8 \times \mathit{days} \\
\qquad\qquad\ \ \mathit{days}\ \ = 5 \times \mathit{weeks}\ .
\end{array}
$$

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.

   $$\begin{aligned}&\mathit{nested} :: \mathrm{Int}\\&\mathit{nested} = \mathbf{let}\ x = 3\\&\qquad\quad \mathbf{in}\ (\mathbf{let}\ x = 5\\&\qquad\qquad\ \ \mathbf{in}\ x + x) + x\ \ .\end{aligned}$$

   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.

   $$\begin{aligned}&\mathit{recursive} :: \mathrm{Int}\\&\mathit{recursive} = \mathbf{let}\ x = 3\\&\qquad\qquad\ \ \mathbf{in}\ \mathbf{let}\ x = x + 1\\&\qquad\qquad\qquad \mathbf{in}\ x\ \ .\end{aligned}$$

   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.

   $$\begin{aligned}&\mathit{smaller} :: \mathrm{Int} \to \mathrm{Int} \to \mathrm{Int}\\&\mathit{smaller}\ x\ y = \mathbf{if}\ x \leqslant y\ \mathbf{then}\ x\ \mathbf{else}\ y\ \ .\end{aligned}$$

   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:

$$
\begin{aligned}
&poly :: \mathrm{Float} \to \mathrm{Float} \to \mathrm{Float} \to \mathrm{Float} \to \mathrm{Float} \\
&poly\ a\ b\ c\ x = a \times x \times x + b \times x + c \\
&poly1 :: \mathrm{Float} \to \mathrm{Float} \\
&poly1 = poly\ 1\ 2\ 1 \\
&poly :: \mathrm{Float} \to \mathrm{Float} \to \mathrm{Float} \to \mathrm{Float} \\
&poly2\ a\ b\ c = poly\ a\ b\ c\ 2
\end{aligned}
$$

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

(a) Define a function $quad :: \mathrm{Int} \to \mathrm{Int}$ such that $quad\ x$ computes x^4.

Solution:

$$
\begin{aligned}
&quad :: \mathrm{Int} \to \mathrm{Int} \\
&quad\ x = square\ (square\ x)\ \ .
\end{aligned}
$$

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

$$
\begin{aligned}
&twice \quad :: (a \to a) \to (a \to a) \\
&twice\ f\ x = f\ (f\ x)\ \ .
\end{aligned}
$$

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

Solution:

$$
\begin{aligned}
&quad :: \mathrm{Int} \to \mathrm{Int} \\
&quad = twice\ square\ \ .
\end{aligned}
$$

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

$$
\begin{aligned}
&twice \quad :: (a \to a) \to (a \to a) \\
&twice\ f = f \cdot f\ \ .
\end{aligned}
$$

(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.

   $$\mathit{forktimes}\ f\ g\ x = f\ x \times g\ x\ \ .$$

   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

   $$(t \to \mathit{Int}) \to (t \to \mathit{Int}) \to t \to \mathit{Int}\ \ .$$

   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:

   $$\begin{aligned}&\mathit{compute} :: \mathrm{Int} \to \mathrm{Int}\\&\mathit{compute} = \mathit{forktimes}\ (+1)\ (+2)\ \ .\end{aligned}$$

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

   Solution:

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

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

   Solution:

   $$\begin{aligned}&\mathit{compute} :: \mathrm{Int} \to \mathrm{Int}\\&\mathit{compute} = \mathit{lift2}\ (\times)\ (+1)\ (+2)\ \ .\end{aligned}$$

10. Let the following identifiers have type:

$$\begin{aligned}f &:: \mathit{Int} \to \mathit{Char}\\ g &:: \mathit{Int} \to \mathit{Char} \to \mathit{Int}\\ h &:: (\mathit{Char} \to \mathit{Int}) \to \mathit{Int} \to \mathit{Int}\\ x &:: \mathit{Int}\\ y &:: \mathit{Int}\\ c &:: \mathit{Char}\end{aligned}$$

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

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

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