Introduction to Functional Programming

Carlos Encarnación
encarnacion.carlos@gmail.com

November 5, 2014

Lecture 1: A taste of functional programming

The syllabus

Read the syllabus.

What is functional programming?

Functional programming is a *method* of program construction.

Characteristics of the method

1. Emphasises functions and their application.
2. Allows program to be defined clearly and concisely.
3. Supports equational reasoning.

Programming language

Haskell

Function

In Haskell, the type of a function f is denoted by

```
f :: a -> b
```

but, in these lectures we are going to use a unicode symbol for the arrow symbol

$$f :: a \to b$$

Function definitions: *succ*

There is a function *succ* called the *succesor function* such that

$$n \leadsto n + 1$$

In Haskell we write this function as follows

$$\begin{aligned} &succ \quad :: Integer \rightarrow Integer \\ &succ\ n = n + 1 \end{aligned}$$

Function definitions: *double*

Similarly, there is a function *double* called the *doubling function* such that

$$x \rightsquigarrow 2 * x$$

In Haskell we write this function as follows

$$\begin{aligned} double &:: Double \to Double \\ double\ x &= 2 * x \end{aligned}$$

Function application

In mathematics sometimes we write

$$f(x)$$

to denote function application. However, in Haskell we can *always* write

$$f\ x$$

to denote function application.

Function applications: applying *succ*

$$\mathit{succ}\ 0 = 1$$

Function applications: applying *double*

$$\textit{double}\ (\textit{double}\ 1) = 4$$

Functional composition

Given functions $f :: a \to b$ and $g :: b \to c$ we define the function $(\circ)$ as follows

$$\begin{aligned}(\circ) \quad &:: (b \to c) \to (a \to b) \to (a \to c)\\ (g \circ f)\ x &= g\ (f\ x)\end{aligned}$$

Here we use the usual mathematical symbol. But the equivalent ASCII character is '.'. Thus, in that case we write

```
(g . f) x = g (f x)
```

Problem

Design a function

$$commonWords :: Int \rightarrow [\,Char\,] \rightarrow [\,Char\,]$$

that, given a text returns a list of the n most common words.

Some instances

Thus, it is required that

$$\textit{commonWords}\; 1\; \texttt{"Lorem ipSUM Lorem ipsum lorem"}$$
$$= \texttt{"lorem: 3\textbackslash n"}$$

and that

$$\textit{commonWords}\; 2\; \texttt{"Lorem ipSUM Lorem ipsum lorem"}$$
$$= \texttt{"lorem: 3\textbackslash nipsum: 2\textbackslash n"}$$

Text as words

After a few minutes of thought we realize that we need a function that breaks text into words.

Maybe something along the line of

$$words :: [\,Char\,] \rightarrow [[\,Char\,]]$$

such that

$$\begin{aligned} &words\ \texttt{"Lorem ipSUM Lorem ipsum"} \\ &\quad = [\texttt{"Lorem"}, \texttt{"ipSUM"}, \texttt{"Lorem"}, \texttt{"ipsum"}] \end{aligned}$$

Decluttering types with type synonyms

The type of the function *words* looks a little bit too complex.

$$words :: [\,Char\,] \rightarrow [\,[\,Char\,]\,]$$

What can we do about it?

Use type synonyms.

$$\begin{array}{l}\textbf{type}\ Text = [\,Char\,] \\ \textbf{type}\ Word = [\,Char\,]\end{array}$$

Text as words, revisited

Now, we can write the function *words* as follows

$$words :: Text \to [Word]$$

Hence, type synonyms aid understanding.

Words disambiguation: part 1

But, "TeX" and "tex" should be treated as the same word. How do we solve this problem?

We can start by assuming that there is a function that converts a character to lowercase.

For instance, we might assume that

$$toLower :: Char \rightarrow Char$$

What should we do next?

Words disambiguation: part 2

Well, we can assume that there is a function that given a function f and a list xs, applies f to each element of the list xs.

The type of such function would be as follows

$$map :: (a \to b) \to [a] \to [b]$$

Words disambiguation: part 3

Finally, we find that

$$map\ toLower :: Text \to Text$$

In particular, we have

$$map\ toLower\ \texttt{"Lorem ipSUM Lorem ipsum"} \\ = \texttt{"lorem ipsum lorem ipsum"}$$

So far

We have managed to convert a text to lowercase and the break it into words.

$$words \circ map\ toLower$$

Just for illustrative purpose only, let define an intermediate function f such that

$$f = words \circ map\ toLower$$

therefore applying f we find that

$$\begin{aligned} &f\ \texttt{"Lorem ipSUM Lorem ipsum"} \\ &\quad = [\texttt{"lorem"}, \texttt{"ipsum"}, \texttt{"lorem"}, \texttt{"ipsum"}] \end{aligned}$$

Sorting to obtain information

In algorithm design it is a good idea to sort to obtain information. Therefore, we need a function that sorts words.

The type should be something like

$$sortWords :: [Word] \to [Word]$$

So far

Where are we? Let's see, now we can convert a text to lowercase, break it into words, and then sort the list of words.

$$\mathit{sortWords} \circ \mathit{words} \circ \mathit{map}\ \mathit{toLower}$$

Following our previous scheme we define a intermediate function f to evaluate our progress.

$$f = \mathit{sortWords} \circ \mathit{words} \circ \mathit{map}\ \mathit{toLower}$$

Then we find that

$$\begin{aligned} & f\ \texttt{"Lorem ipSUM Lorem ipsum"} \\ & \quad = [\texttt{"ipsum"}, \texttt{"ipsum"}, \texttt{"lorem"}, \texttt{"lorem"}] \end{aligned}$$

Counting runs

Think for a moment on what we have done. Take the function we have defined in the previous slide. What should we do next?

$$f \; \texttt{"Lorem ipSUM Lorem ipsum"} \\ = [\texttt{"ipsum"}, \texttt{"ipsum"}, \texttt{"lorem"}, \texttt{"lorem"}]$$

That's right! We need a function that count runs.

$$\mathit{countRuns} :: [\mathit{Word}] \rightarrow [(\mathit{Int}, \mathit{Word})]$$

So far

Think again. What do we have?

$$\mathit{countRuns} \circ \mathit{sortWords} \circ \mathit{words} \circ \mathit{map}\ \mathit{toLower}$$

Again, define the intermediate function f

$$f = \mathit{countRuns} \circ \mathit{sortWords} \circ \mathit{words} \circ \mathit{map}\ \mathit{toLower}$$

In particular we have

$$f\ \texttt{"Lorem ipSUM Lorem ipsum lorem"} = [(2, \texttt{"ipsum"}), (3, \texttt{"lorem"})]$$

Sorting runs

Have you noticed a problem? Observe again.

$$f\ \texttt{"Lorem ipSUM Lorem ipsum lorem"}$$
$$= [(2, \texttt{"ipsum"}), (3, \texttt{"lorem"})]$$

Yes! We need a function that sort runs.

$$\mathit{sortRuns} :: [(\mathit{Int}, \mathit{Word})] \to [(\mathit{Int}, \mathit{Word})]$$

Then we have

$$\mathit{sortRuns}\ [(2, \texttt{"ipsum"}), (3, \texttt{"lorem"})]$$
$$= [(3, \texttt{"lorem"}), (2, \texttt{"ipsum"})]$$

So far

This one is too long to write it down. By now you should know how it goes.

$$\mathit{sortRuns} \circ \mathit{countRuns} \circ \mathit{sortWords} \circ \mathit{words} \circ \mathit{map\ toLower}$$

Just take what we need

Remember that we do not need all runs. Hence, we need an initial segment of the list. That's exactly what the function *take* does.

$$\mathit{take} :: \mathit{Int} \to [a] \to [a]$$

For instance, we have

$$\begin{aligned} &\mathit{take}\ 1\ [(3, \texttt{"lorem"}), (2, \texttt{"ipsum"})] \\ &\quad = [(3, \texttt{"lorem"})] \end{aligned}$$

Show run

At the end of the day, we need to display something.

$$showRun :: (Int, String) \rightarrow String$$

So, we define a function such that

$$showRun\ (3, \texttt{"lorem"}) = \texttt{"lorem: 3\textbackslash n"}$$

Show runs

Since we have a list of runs, we use *map* again.

$$\mathit{showRuns} :: [(\mathit{Int}, \mathit{String})] \rightarrow [\mathit{String}]$$
$$\mathit{showRuns} = \mathit{map}\ \mathit{showRun}$$

hence

$$\mathit{showRuns}\ [(3, \texttt{"lorem"}), (2, \texttt{"ipsum"})]$$
$$= [\texttt{"lorem: 3\textbackslash n"}, \texttt{"ipsum: 2"}]$$

Joining runs

Hey! But the function *map* created another problem: now we have a list of strings, but what was required is a string.

Calm down, we can define another function for that purpose.

$$concat :: [[a]] \to [a]$$

Such that

$$\begin{aligned} &concat\,[\texttt{"lorem: 3\textbackslash n"}, \texttt{"ipsum: 2\textbackslash n"}] \\ &\quad = \texttt{"lorem: 3\textbackslash nipsum: 2\textbackslash n"} \end{aligned}$$

Putting it all together

Finally, we have what was required.

$$
\begin{aligned}
&\mathit{commonWords} \quad :: \mathit{Int} \to \mathit{Text} \to \mathit{String} \\
&\mathit{commonWords}\ n = \mathit{concat} \circ \mathit{showRuns} \circ \mathit{take}\ n \circ \\
&\quad \mathit{sortRuns} \circ \mathit{countRuns} \circ \mathit{sortWords} \circ \mathit{words} \circ \mathit{map}\ \mathit{toLower}
\end{aligned}
$$

Problem

That was fun! Let's play the game again. Design a function

$$convert :: Int \to String$$

that, given an nonnegative number less than a million returns a string that represents the number in words.

Preliminaries

First we define some auxiliary lists.

$$\mathit{units} = [\texttt{"zero"}, \texttt{"one"}, \texttt{"two"}, \texttt{"three"}, \texttt{"four"}, \texttt{"five"}, \texttt{"six"}, \texttt{"seven"}, \texttt{"eight"}, \texttt{"nine"}]$$

$$\mathit{teens} = [\texttt{"ten"}, \texttt{"eleven"}, \texttt{"twelve"}, \texttt{"thirteen"}, \texttt{"fourteen"}, \texttt{"fifteen"}, \texttt{"sixteen"}, \texttt{"seventeen"}, \texttt{"eighteen"}, \texttt{"nineteen"}]$$

$$\mathit{tens} = [\texttt{"twenty"}, \texttt{"thirty"}, \texttt{"forty"}, \texttt{"fifty"}, \texttt{"sixty"}, \texttt{"seventy"}, \texttt{"eighty"}, \texttt{"ninety"}]$$

Divide and conquer

Think small.

$$\begin{aligned} & \mathit{convert1} && :: \mathit{Int} \rightarrow \mathit{String} \\ & \mathit{convert1}\ n && = \mathit{units}\ !!\ n \end{aligned}$$

The next order of magnitude: $0 \leqslant n < 100$

Take one step further.

$$convert2 :: Int \to String$$

First: divide

Split the digits.

$$\begin{aligned} &\mathit{digits2} \quad :: \mathit{Int} \to (\mathit{Int}, \mathit{Int}) \\ &\mathit{digits2}\ n = (\mathit{div}\ n\ 10, \mathit{mod}\ n\ 10) \end{aligned}$$

We could also write it in another way

$$\begin{aligned} &\mathit{digits2} \quad :: \mathit{Int} \to (\mathit{Int}, \mathit{Int}) \\ &\mathit{digits2}\ n = (n\ \text{`}\mathit{div}\text{`}\ 10, n\ \text{`}\mathit{mod}\text{`}\ 10) \end{aligned}$$

Next: combine

Given two digits convert it into words.

$$
\begin{aligned}
&\mathit{combine2} && :: (\mathit{Int}, \mathit{Int}) \rightarrow \mathit{String} \\
&\mathit{combine2}\ (t, u) \mid t \equiv 0 && = \mathit{units}\ !!\ u \\
&\qquad\qquad\qquad\ \ \mid t \equiv 1 && = \mathit{teens}\ !!\ u \\
&\qquad\qquad\qquad\ \ \mid t > 1 \wedge u \equiv 0 && = \mathit{tens}\ !!\ (t - 2) \\
&\qquad\qquad\qquad\ \ \mid \mathit{otherwise} && = \mathit{tens}\ !!\ (t - 2) +\!\!+ \texttt{"-"} +\!\!+ \\
& && \quad\ \mathit{units}\ !!\ u
\end{aligned}
$$

convert2: solution

Compose *combine2* and *digits2* to find that

$$
\begin{aligned}
&convert2 :: Int \to String \\
&convert2 = combine2 \circ digits2
\end{aligned}
$$

The next order of magnitude: $100 \leqslant n < 1000$

Move forward.

$$
\begin{array}{lll}
\mathit{convert3} & & :: \mathit{Int} \rightarrow \mathit{String} \\
\mathit{convert3}\ n & \mid h \equiv 0 & = \mathit{convert2}\ t \\
& \mid t \equiv 0 & = \mathit{units}\ !!\ h +\!\!+ \texttt{" hundred"} \\
& \mid \mathit{otherwise} & = \mathit{units}\ !!\ h +\!\!+ \texttt{" hundred and "} +\!\!+ \\
& & \quad \mathit{convert2}\ t \\
\multicolumn{3}{l}{\quad \textbf{where}\ (h, t) = (n\ \text{`}\mathit{div}\text{`}\ 100, n\ \text{`}\mathit{mod}\text{`}\ 100)}
\end{array}
$$

$1000 \leqslant n < 1000000$

Take it all.

$$
\begin{array}{lll}
\mathit{convert6} & & :: \mathit{Int} \to \mathit{String} \\
\mathit{convert6}\ n & \mid m \equiv 0 & = \mathit{convert3}\ h \\
& \mid h \equiv 0 & = \mathit{convert3}\ m +\!\!+ \texttt{" thousand"} \\
& \mid \mathit{otherwise} & = \mathit{convert3}\ m +\!\!+ \texttt{" thousand"} +\!\!+ \\
& & \quad \mathit{link}\ h +\!\!+ \mathit{convert3}\ h \\
\multicolumn{3}{l}{\quad \textbf{where}\ (m, h) = (n\ `\mathit{div}`\ 1000, n\ `\mathit{mod}`\ 1000)}
\end{array}
$$

$$
\begin{array}{l}
\mathit{link} \quad :: \mathit{Int} \to \mathit{String} \\
\mathit{link}\ h = \textbf{if}\ h < 100\ \textbf{then}\ \texttt{" and "}\ \textbf{else}\ \texttt{" "}
\end{array}
$$

Solution

Rest. We are done.

$$convert :: Int \to String$$
$$convert = convert6$$

The Haskell Platform

Notion

By definition

1. Every well-formed expression has a well-formed type.
2. Each well-formed expression has a value.

GHCi

Evaluation steps:

1. Syntax analysis.
2. Type inference analysis.
3. Reduction (to normal form).
4. Try to print the value.

Definitions

1. Names for functions and values begin with lowercase letter, except for data constructors.
2. Types, type classes and modules begin with uppercase letter.

Operators

Operators are functions that can appear in infix position. Since $(*)$ is an operator, hence we can write

$$6 * 7$$

Furthermore, functions can be converted to operators by enclosing it in backticks. For instance,

$$\mathit{mod}\ 6\ 2$$

can be written as

$$6\ `\mathit{mod}`\ 2$$

Precedence

When two or more operators appear in the same expression, given that parentheses are not present, we need a rule in order to know which operator is applied first. For example, what is the value of the following expression?

$$1 + 2\,/\,3$$

There are two possibilities, either

$$1 + 2\,/\,2 = 2$$

or

$$1 + 2\,/\,2 = 1.5$$

Associativity order

Similarly, when an operator appears more than once in an expression we need a rule to know in which order the operands are grouped. For example, consider the following type

$$map :: (a \to b) \to [a] \to [b]$$

Can we write this type without parentheses?

No! Because, the operator $\to$ associates to the right. Hence

$$map :: a \to b \to [a] \to [b]$$

is equivalent to

$$map :: a \to (b \to ([a] \to [b]))$$

Rules for arithmetic operators

$\uparrow$	-- right
$*\ /$	-- left
$+\ -$	-- left

Sections

Operators can be partialy applied.

$$succ = (+1)$$

$$double = (2*)$$

$$reciprocal = (1/)$$

Lambda abstractions

Yet another way to write *succ*

$$succ = \lambda x \to x + 1$$

```
succ = \n -> n + 1
```

For instance, we can increment each element of a list by

$$map\ (\lambda x \to x + 1)\ [1, 2, 3] = [2, 3, 4]$$

```
map (\x -> x + 1) [1,2,3] = [2,3,4]
```

Square

Consider the following function.

$$
\begin{aligned}
&sqr \;\; :: Integer \to Integer \\
&sqr \;\; x \;\; = x * x
\end{aligned}
$$

Evaluation strategies: Example 1

Eager evaluation

$$\begin{aligned} & \mathit{sqr}\ (3+4) \\ &= \mathit{sqr}\ 7 \\ &= \textbf{let}\ x = 7\ \textbf{in}\ x * x \\ &= 49 \end{aligned}$$

Lazy evaluation

$$\begin{aligned} & \mathit{sqr}\ (3+4) \\ &= \textbf{let}\ x = 3+4\ \textbf{in}\ x * x \\ &= \textbf{let}\ x = 7\ \textbf{in}\ x * x \\ &= 49 \end{aligned}$$

Evaluation strategies: Example 2

Eager evaluation

$$\begin{aligned} & \mathit{fst}\ (\mathit{sqr}\ 3, \mathit{sqr}\ 4) \\ &= \mathit{fst}\ (3 * 3, \mathit{sqr}\ 4) \\ &= \mathit{fst}\ (9, \mathit{sqr}\ 4) \\ &= \mathit{fst}\ (9, 4 * 4) \\ &= \mathit{fst}\ (9, 16) \\ &= 9 \end{aligned}$$

Lazy evaluation

$$\begin{aligned} & \mathit{fst}\ (\mathit{sqr}\ 3, \mathit{sqr}\ 4) \\ &= \mathbf{let}\ p = (\mathit{sqr}\ 3, \mathit{sqr}\ 4)\ \mathbf{in}\ \mathit{fst}\ p \\ &= \mathit{sqr}\ 3 \\ &= 3 * 3 \\ &= 9 \end{aligned}$$

To infinity and beyond

Now suppose we have

$$\begin{aligned} &\mathit{infinity} :: \mathit{Integer} \to \mathit{Integer} \\ &\mathit{infinity} = 1 + \mathit{infinity} \end{aligned}$$

$$\begin{aligned} &\mathit{three} \quad :: a \to \mathit{Integer} \\ &\mathit{three}\ x = 3 \end{aligned}$$

Evaluation strategies: Example 3

Eager evaluation

$$\begin{aligned} & \mathit{three}\ \mathit{infinity} \\ & = \mathit{three}\ (1 + \mathit{infinity}) \\ & = \mathit{three}\ (1 + (1 + \mathit{infinity})) \\ & = \ldots \end{aligned}$$

Lazy evaluation

$$\begin{aligned} & \mathit{three}\ \mathit{infinity} \\ & = \textbf{let}\ x = \mathit{infinity}\ \textbf{in}\ 3 \\ & = 3 \end{aligned}$$

Factorial

Consider one more equation.

$$\begin{aligned} \mathit{factorial} &:: \mathit{Integer} \to \mathit{Integer} \\ \mathit{factorial}\ n &= \mathit{fact}\ (n, 1) \end{aligned}$$

$$\begin{aligned} \mathit{fact} &:: (\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer} \\ \mathit{fact}\ (x, y) &= \mathbf{if}\ x \equiv 0\ \mathbf{then}\ y\ \mathbf{else}\ \mathit{fact}\ (x - 1, x * y) \end{aligned}$$

Evaluation strategies: Example 4

Eager evaluation

$$\begin{aligned} & \mathit{factorial}\ 3 \\ & = \mathit{fact}\ (3, 1) \\ & = \mathit{fact}\ (3-1, 3*1) \\ & = \mathit{fact}\ (2, 3) \\ & = \mathit{fact}\ (2-1, 3*2) \\ & = \mathit{fact}\ (1, 6) \\ & = \mathit{fact}\ (1-1, 6*1) \\ & = \mathit{fact}\ (0, 6) \\ & = 6 \end{aligned}$$

Lazy evaluation

$$\begin{aligned} & \mathit{factorial}\ 3 \\ & = \mathit{fact}\ (3, 1) \\ & = \mathit{fact}\ (3-1, 3*1) \\ & = \mathit{fact}\ (2-1, 2*(3*1)) \\ & = \mathit{fact}\ (1-1, 1*(2*(3*1))) \\ & = 1*(2*(3*1)) \\ & = 1*(2*3) \\ & = 1*6 \\ & = 6 \end{aligned}$$

Pros and cons of lazy evaluation

+ Terminates whenever any reduction order terminates.
+ It never takes more steps than eager evaluation and sometimes infinitely fewer.
- It can take a lot more space than eager evaluation.
- It is more difficult to understand the precise order in which things happen.

Strict and non-strict functions

A Haskell function f is said to be strict if $f \perp = \perp$ and non-strict otherwise.

Built-in types

An example of datatype declaration.

$$\textbf{data}\ \mathit{Bool} = \mathit{False} \mid \mathit{True}$$

How many values does the type *Bool* have?

Dilemma

Consider the following ~~dilemma~~ function.

$$\begin{aligned} &to \quad :: Bool \to Bool \\ &to\ b = \neg\,(to\ b) \end{aligned}$$

What is the value of *to True*?

$$to\ True = \bot$$

Compound built-in types

Consider the following types.

$$[Int]$$
$$(Int, String)$$
$$(Int, Float, Double)$$
$$[Int \to Int]$$
$$()$$

How many values does the type () have?

Polymorphic functions

Now, consider the following types.

$$
\begin{aligned}
&take :: Int \to [a] \to [a] \\
&(+\!\!+) :: [a] \to [a] \to [a] \\
&map :: (a \to b) \to [a] \to [b] \\
&(\circ) :: (b \to c) \to (a \to b) \to (a \to c)
\end{aligned}
$$

1. The functions *take*, *map* and $(\circ)$ are said to be *polymorphic* functions.
2. The types a, b and c are *type variables*.

What is the type of $(+)$?

Suggestions

What do you think about the following types? Do they fit $(+)$?

$$(+) :: Int \to Int \to Int$$
$$(+) :: Float \to Float \to Float$$
$$(+) :: a \to a \to a$$

Answer: type classes

Then the type of $(+)$ is the type $a \to a \to a$ whenever the type a is an instance of the type class *Num*.

$$(+) :: Num\ a \Rightarrow a \to a \to a$$

The type class *Eq*

Declaration of the type class *Eq*.

$$
\begin{array}{l}
\textbf{class } Eq\ a \textbf{ where} \\
\quad (\equiv), (\not\equiv) :: a \rightarrow a \rightarrow Bool \\
\quad x \not\equiv y \quad\ = \neg\,(x \equiv y)
\end{array}
$$

A type class, such as *Eq*, has a collection of named methods. In particular, we have the named methods $(\equiv)$ and $(\not\equiv)$. Therefore type classes provide for *overloaded* functions.

An instance of the class *Eq*

Below we present the declaration that Bool is an instance of the type class *Eq*.

$$\textbf{instance}\ \mathit{Eq}\ \mathit{Bool}\ \textbf{where}$$
$$\quad x \equiv y = \textbf{if}\ x\ \textbf{then}\ y\ \textbf{else}\ \neg\, y$$

The type class *Ord*

A type class can be a subclass of another class.

$$
\begin{array}{l}
\textbf{class}\ (Eq\ a) \Rightarrow Ord\ a\ \textbf{where} \\
\quad \begin{array}{ll}
(<), (\leqslant), (>), (\geqslant) & ::\ a \rightarrow a \rightarrow Bool \\
x < y & = \neg\ (x \geqslant y) \\
x \leqslant y & = x \equiv y \lor (x < y) \\
x > y & = \neg\ (x \leqslant y) \\
x \geqslant y & = x \equiv y \lor (x > y)
\end{array}
\end{array}
$$

The instances of the type class *Ord* are those types which support the notion of ordering.

The type class *Show*

Last but not least, we have the type class *Show*.

> **class** *Show a* **where**
> $\quad show :: a \rightarrow String$

The instances of the type class *Show* are those types which can be converted to a string.

Mystery

In a GHCi session, what happen if I type the following?

```
"lorem: 3\nipsum: 2\n"
```

The function *putStrLn*

To get the answer we expect we must use the *command putStrLn*.

putStrLn `"lorem: 3\nipsum: 2\n"`
lorem : 3
ipsum : 2

The type of *putStrLn* is quite remarkable.

$$putStrLn :: String \rightarrow IO\ ()$$

Indeed, in Haskell all actions have the type *IO a* where *a* represents the result type of the computation.

do-notation

Actions have to be sequenced in other to work properly.

$$
\begin{array}{ll}
\mathit{cwords} & :: \mathit{Int} \to \mathit{FilePath} \to \mathit{FilePath} \to \mathit{IO}\ () \\
\mathit{cwords}\ n\ \mathit{inf}\ \mathit{outf} = \mathbf{do} & \{ \mathit{text} \leftarrow \mathit{readFile}\ \mathit{inf}; \\
& \ \ \mathit{writeFile}\ \mathit{outf}\ (\mathit{commonWords}\ n\ \mathit{text}); \\
& \}
\end{array}
$$

module *CommonWords* (*commonWords*) **where**
  import ...
  type *Word* = ...
  commonWords *n* = ...

Explicit layout

We can use braces and semicolons.

$$
\begin{aligned}
&\mathit{roots} && :: (\mathit{Double}, \mathit{Double}, \mathit{Double}) \\
& && \rightarrow (\mathit{Double}, \mathit{Double}) \\
&\mathit{roots}\ (a, b, c) \mid a \equiv 0 && = \mathit{error}\ \texttt{"not quadratic"} \\
&\qquad\qquad\quad\ \mid d < 0 && = \mathit{error}\ \texttt{"complex roots"} \\
&\qquad\qquad\quad\ \mid \mathit{otherwise} && = ((-b - r)\ /\ e, (-b + r)\ /\ e) \\
&\quad \textbf{where}\ \{d = b * b - 4 * a * c; r = \mathit{sqrt}\ d; e = 2 * a\}
\end{aligned}
$$

Implicit layout

Also, we can choose not to use braces and semicolons.

$$
\begin{array}{lll}
\mathit{roots}' & :: & (\mathit{Double}, \mathit{Double}, \mathit{Double}) \\
 & \rightarrow & (\mathit{Double}, \mathit{Double}) \\
\mathit{roots}'\ (a, b, c) \mid a \equiv 0 & = & \mathit{error}\ \texttt{"not quadratic"} \\
\qquad\qquad\qquad\ \mid d < 0 & = & \mathit{error}\ \texttt{"complex roots"} \\
\qquad\qquad\qquad\ \mid \mathit{otherwise} & = & ((-b - r)\ /\ e, (-b + r)\ /\ e)
\end{array}
$$

$$
\begin{array}{l}
\quad \textbf{where} \\
\qquad d = b * b - 4 * a * c \\
\qquad r = \mathit{sqrt}\ d \\
\qquad e = 2 * a
\end{array}
$$

Implicit layout

As a final example, consider the following.

$$
\begin{aligned}
&\mathit{cwords} && :: \mathit{Int} \rightarrow \mathit{FilePath} \rightarrow \mathit{FilePath} \rightarrow \mathit{IO}\ ()\\
&\mathit{cwords}\ n\ \mathit{inf}\ \mathit{outf} && = \mathbf{do}\\
&\quad \mathit{text} \leftarrow \mathit{readFile}\ \mathit{inf}\\
&\quad \mathit{writeFile}\ \mathit{outf}\ (\mathit{commonWords}\ n\ \mathit{text})
\end{aligned}
$$