Programming in Haskell
Solutions to Exercises

Graham Hutton
University of Nottingham

Contents
Chapter 1 - Introduction 1
Chapter 2 - First steps 3
Chapter 3 - Types and classes 4
Chapter 4 - Defining functions 5
Chapter 5 - List comprehensions 7
Chapter 6 - Recursive functions 9
Chapter 7 - Higher-order functions 13
Chapter 8 - Functional parsers 15
Chapter 9 - Interactive programs 19
Chapter 10 - Declaring types and classes 21
Chapter 11 - The countdown problem 23
Chapter 12 - Lazy evaluation 24
Chapter 13 - Reasoning about programs 26
Chapter 1 - Introduction

Exercise 1

double (double 2)
double (2 + 2)
= (2 + 2)
= (2 + 2) + (2 + 2)
= (2 + 2) + (2 + 2)
= (2 + 2) + 4
= (2 + 2) + 4
= 4 + 4
= 8

or

double (double 2)
= double (double 2)
= double (double 2) + (double 2)
= double (2 + 2) + (double 2)
= double (2 + 2) + 4
= double (2 + 2) + 4
= 4 + 4
= 8

There are a number of other answers too.

Exercise 2

sum [x]
= { applying sum }
x + sum []
= { applying sum }
x + 0
= { applying + }
x

Exercise 3

(1)

product [] = 1
product (x : xs) = x * product xs
\begin{align*}
\text{product} \ [2, 3, 4] = & \text{ applying product } \\
2 \times (\text{product} \ [3, 4]) = & \text{ applying product } \\
2 \times (3 \times \text{product} \ [4]) = & \text{ applying product } \\
2 \times (3 \times (4 \times \text{product} \ [\ ])) = & \text{ applying product } \\
2 \times (3 \times (4 \times 1)) = & \text{ applying } \ast \\
24
\end{align*}

Exercise 4

Replace the second equation by
\[ qsort \ (x : xs) = qsort \ larger \ ++ \ [x] \ ++ \ qsort \ smaller \]
That is, just swap the occurrences of \textit{smaller} and \textit{larger}.

Exercise 5

Duplicate elements are removed from the sorted list. For example:
\begin{align*}
\text{qsort} \ [2, 2, 3, 1, 1] = & \text{ applying qsort } \\
\text{qsort} \ [1, 1] ++ [2] ++ \text{qsort} \ [3] = & \text{ applying qsort } \\
(qsort \ [\ ] ++ [1] ++ qsort \ [\ ]) ++ [2] ++ (qsort \ [\ ] ++ [3] ++ qsort \ [\ ]) = & \text{ applying qsort } \\
([\ ] ++ [1] ++ [\ ]) ++ [2] ++ ([\ ] ++ [3] ++ [\ ]) = & \text{ applying ++ } \\
[1] ++ [2] ++ [3] = & \text{ applying ++ } \\
[1, 2, 3]
\end{align*}
Chapter 2 - First steps

Exercise 1

\[(2 \uparrow 3) \times 4\]
\[(2 \times 3) + (4 \times 5)\]
\[2 + (3 \times (4 \uparrow 5))\]

Exercise 2

No solution required.

Exercise 3

\[n = a \text{ `div` } \text{length } xs\]
\[\text{where}\]
\[a = 10\]
\[xs = [1, 2, 3, 4, 5]\]

Exercise 4

\[\text{last } xs = \text{head } (\text{reverse } xs)\]
or
\[\text{last } xs = xs!!(\text{length } xs - 1)\]

Exercise 5

\[\text{init } xs = \text{take } (\text{length } xs - 1) xs\]
or
\[\text{init } xs = \text{reverse } (\text{tail } (\text{reverse } xs))\]
Chapter 3 - Types and classes

Exercise 1

\[
\begin{align*}
[Char] \\
(Char, Char, Char) \\
[(Bool, Char)] \\
([Bool], [Char]) \\
[[a] \rightarrow [a]]
\end{align*}
\]

Exercise 2

\[
\begin{align*}
[a] \rightarrow a \\
(a, b) \rightarrow (b, a) \\
a \rightarrow b \rightarrow (a, b) \\
Num a \Rightarrow a \rightarrow a \\
Eq a \Rightarrow [a] \rightarrow Bool \\
(a \rightarrow a) \rightarrow a \rightarrow a
\end{align*}
\]

Exercise 3

No solution required.

Exercise 4

In general, checking if two functions are equal requires enumerating all possible argument values, and checking if the functions give the same result for each of these values. For functions with a very large (or infinite) number of argument values, such as values of type \textit{Int} or \textit{Integer}, this is not feasible. However, for small numbers of argument values, such as values of type of type \textit{Bool}, it is feasible.
Chapter 4 - Defining functions

Exercise 1

\[ \text{halve xs} = \text{splitAt} \left( \text{length xs} \div 2 \right) \text{xs} \]

or

\[ \text{halve xs} = (\text{take n xs}, \text{drop n xs}) \]

where

\[ n = \text{length xs} \div 2 \]

Exercise 2

(a)

\[ \text{safetail xs} = \begin{cases} \text{[]} & \text{if null xs} \\ \text{tail xs} & \text{otherwise} \end{cases} \]

(b)

\[ \text{safetail xs} \mid \text{null xs} = [\text{}] \]

\[ \mid \text{otherwise} = \text{tail xs} \]

(c)

\[ \text{safetail \[\text{}\] = \[\text{}\]} \]

\[ \text{safetail xs} = \text{tail xs} \]

or

\[ \text{safetail \[\text{}\] = \[\text{}\]} \]

\[ \text{safetail (\_; xs) = xs} \]

Exercise 3

(1)

\[ \text{False} \lor \text{False} = \text{False} \]
\[ \text{False} \lor \text{True} = \text{True} \]
\[ \text{True} \lor \text{False} = \text{True} \]
\[ \text{True} \lor \text{True} = \text{True} \]

(2)

\[ \text{False} \lor \text{False} = \text{False} \]
\[ \text{\_} \lor \text{\_} = \text{True} \]

(3)

\[ \text{False} \lor b = b \]
\[ \text{True} \lor \_ = \text{True} \]

(4)

\[ b \lor c \mid b \equiv c = b \]
\[ \mid \text{otherwise} = \text{True} \]
Exercise 4

\[ a \land b = \text{if } a \text{ then } \text{if } b \text{ then True else False else False} \]

Exercise 5

\[ a \land b = \text{if } a \text{ then } b \text{ else False} \]

Exercise 6

\[ \text{mult} = \lambda x \to (\lambda y \to (\lambda z \to x \times y \times z)) \]
Chapter 5 - List comprehensions

Exercise 1

\[ \text{sum} \left[ x \uparrow 2 \mid x \leftarrow [1..100] \right] \]

Exercise 2

\[ \text{replicate} \ n \ x = \left[ x \mid _- \leftarrow [1..n] \right] \]

Exercise 3

\[ \text{pyths} \ n = \left[ (x, y, z) \mid x \leftarrow [1..n], \\
\quad y \leftarrow [1..n], \\
\quad z \leftarrow [1..n], \\
\quad x \uparrow 2 + y \uparrow 2 == z \uparrow 2 \right] \]

Exercise 4

\[ \text{perfects} \ n = \left[ x \mid x \leftarrow [1..n], \text{sum} \left( \text{init} \ (\text{factors} \ x) \right) == x \right] \]

Exercise 5

\[ \text{concat} \left[ \left[ (x, y) \mid y \leftarrow [4, 5, 6] \right] \mid x \leftarrow [1, 2, 3] \right] \]

Exercise 6

\[ \text{positions} \ x \ xs = \text{find} \ x \ (\text{zip} \ xs \ [0..n]) \]
\[ \text{where} \ n = \text{length} \ xs - 1 \]

Exercise 7

\[ \text{scalarproduct} \ xs \ ys = \text{sum} \left[ x \ast y \mid (x, y) \leftarrow \text{zip} \ xs \ ys \right] \]

Exercise 8

\[ \text{shift} :: \ \text{Int} \rightarrow \text{Char} \rightarrow \text{Char} \]
\[ \text{shift} \ n \ c \mid \text{isLower} \ c = \text{int2low} \ ((\text{low2int} \ c + n) \ \text{`mod`} \ 26) \]
\[ \mid \text{isUpper} \ c = \text{int2upp} \ ((\text{upp2int} \ c + n) \ \text{`mod`} \ 26) \]
\[ \mid \text{otherwise} = c \]

\[ \text{freqs} :: \ \text{String} \rightarrow \left[ \text{Float} \right] \]
\[ \text{freqs} \ xs = \left[ \text{percent} \ (\text{count} \ x \ xs') \ n \mid x \leftarrow ['a'..'z'] \right] \]
\[ \text{where} \]
\[ xs' = \text{map} \ \text{toLowerCase} \ xs \]
\[ n = \text{letters} \ xs \]

\[ \text{low2int} :: \ \text{Char} \rightarrow \text{Int} \]
\[ \text{low2int} \ c = \text{ord} \ c \ - \ \text{ord} \ 'a' \]
\[
\text{int2low} \quad :: \quad \text{Int} \to \text{Char} \\
\text{int2low} \ n \quad = \quad \text{chr} \ (\text{ord} \ 'a' + n) \\
\text{upp2int} \quad :: \quad \text{Char} \to \text{Int} \\
\text{upp2int} \ c \quad = \quad \text{ord} \ c - \text{ord} \ 'A' \\
\text{int2upp} \quad :: \quad \text{Int} \to \text{Char} \\
\text{int2upp} \ n \quad = \quad \text{chr} \ (\text{ord} \ 'A' + n) \\
\text{letters} \quad :: \quad \text{String} \to \text{Int} \\
\text{letters} \ \text{xs} \quad = \quad \text{length} \ [x \mid x \leftarrow \text{xs}, \text{isAlpha} \ x]
\]
Chapter 6 - Recursive functions

Exercise 1

(1)
\[ m \uparrow 0 = 1 \]
\[ m \uparrow (n + 1) = m \times m \uparrow n \]

(2)
\[ 2 \uparrow 3 = \{ \text{applying } \uparrow \} \]
\[ 2 \times (2 \uparrow 2) \]
\[ = \{ \text{applying } \uparrow \} \]
\[ 2 \times (2 \times (2 \uparrow 1)) \]
\[ = \{ \text{applying } \uparrow \} \]
\[ 2 \times (2 \times (2 \uparrow 0)) \]
\[ = \{ \text{applying } \uparrow \} \]
\[ 2 \times (2 \times 1) \]
\[ = \{ \text{applying } \ast \} \]
\[ 8 \]

Exercise 2

(1)
\[ \text{length} \left[ 1, 2, 3 \right] = \{ \text{applying } \text{length} \} \]
\[ 1 + \text{length} \left[ 2, 3 \right] \]
\[ = \{ \text{applying } \text{length} \} \]
\[ 1 + (1 + \text{length} \left[ 3 \right]) \]
\[ = \{ \text{applying } \text{length} \} \]
\[ 1 + (1 + (1 + 0)) \]
\[ = \{ \text{applying } + \} \]
\[ 3 \]

(2)
\[ \text{drop} 3 \left[ 1, 2, 3, 4, 5 \right] = \{ \text{applying } \text{drop} \} \]
\[ \text{drop} 2 \left[ 2, 3, 4, 5 \right] = \{ \text{applying } \text{drop} \} \]
\[ \text{drop} 1 \left[ 3, 4, 5 \right] = \{ \text{applying } \text{drop} \} \]
\[ \text{drop} 0 \left[ 4, 5 \right] = \{ \text{applying } \text{drop} \} \]
\[ [4, 5] \]
\text{(3)}

\begin{align*}
\text{init} [1, 2, 3] &= \{ \text{applying init} \} \\
1 : \text{init} [2, 3] &= \{ \text{applying init} \} \\
1 : 2 : \text{init} [3] &= \{ \text{applying init} \} \\
1 : 2 : [] &= \{ \text{list notation} \} \\
[1, 2] &=
\end{align*}

\textbf{Exercise 3}

\begin{align*}
\text{and} [\ ] &= \text{True} \\
\text{and} (b : bs) &= b \land \text{and} \ bs \\
\text{concat} [\ ] &= [] \\
\text{concat} (xs : xss) &= xs ++ \text{concat} \ xss \\
\text{replicate} 0 \_ &= [] \\
\text{replicate} (n + 1) \ x &= x : \text{replicate} \ n \ x \\
(x : \_)(!!) 0 &= x \\
(\_ : xs)(!!)(n + 1) &= xs!!n \\
\text{elem} \ x [\ ] &= \text{False} \\
\text{elem} \ x (y : ys) | x == y &= \text{True} \\
| \text{otherwise} &= \text{elem} \ x \ ys
\end{align*}

\textbf{Exercise 4}

\begin{align*}
\text{merge} [\ ] ys &= ys \\
\text{merge} \ xs [\ ] &= xs \\
\text{merge} (x : xs)(y : ys) &= \begin{cases}
\text{if} \ x \leq y \ \text{then} & x : \text{merge} \ xs \ (y : ys) \\
\text{else} & y : \text{merge} (x : xs) \ ys
\end{cases}
\end{align*}

\textbf{Exercise 5}

\begin{align*}
\text{halve} \ xs &= \text{splitAt} (\text{length} \ xs \ ' \div \ 2) \ xs \\
\text{msort} [\ ] &= [] \\
\text{msort} [x] &= [x] \\
\text{msort} \ xs &= \text{merge} (\text{msort} \ ys) (\text{msort} \ zs) \\
\text{where} (ys, zs) &= \text{halve} \ xs
\end{align*}
Exercise 6.1

Step 1: define the type

\[ \text{sum} :: [\text{Int}] \to \text{Int} \]

Step 2: enumerate the cases

\[ \text{sum} \ [] = \]

\[ \text{sum} \ (x : xs) = \]

Step 3: define the simple cases

\[ \text{sum} \ [] = 0 \]

\[ \text{sum} \ (x : xs) = \]

Step 4: define the other cases

\[ \text{sum} \ [] = 0 \]

\[ \text{sum} \ (x : xs) = x + \text{sum} \ xs \]

Step 5: generalise and simplify

\[ \text{sum} :: \text{Num} \ a \Rightarrow [a] \to a \]

\[ \text{sum} = \text{foldr} (+) 0 \]

Exercise 6.2

Step 1: define the type

\[ \text{take} :: \text{Int} \to [a] \to [a] \]

Step 2: enumerate the cases

\[ \text{take} \ 0 \ [] = \]

\[ \text{take} \ 0 \ (x : xs) = \]

\[ \text{take} \ (n + 1) \ [] = \]

\[ \text{take} \ (n + 1) \ (x : xs) = \]

Step 3: define the simple cases

\[ \text{take} \ 0 \ [] = [] \]

\[ \text{take} \ 0 \ (x : xs) = [] \]

\[ \text{take} \ (n + 1) \ [] = [] \]

\[ \text{take} \ (n + 1) \ (x : xs) = \]

Step 4: define the other cases

\[ \text{take} \ 0 \ [] = [] \]

\[ \text{take} \ 0 \ (x : xs) = [] \]

\[ \text{take} \ (n + 1) \ [] = [] \]

\[ \text{take} \ (n + 1) \ (x : xs) = x : \text{take} \ n \ xs \]

Step 5: generalise and simplify

\[ \text{take} :: \text{Int} \to [a] \to [a] \]

\[ \text{take} \ 0 = [] \]

\[ \text{take} \ (n + 1) \ [] = [] \]

\[ \text{take} \ (n + 1) \ (x : xs) = x : \text{take} \ n \ xs \]
Exercise 6.3

Step 1: define the type

\[ \text{last} :: [a] \rightarrow [a] \]

Step 2: enumerate the cases

\[ \text{last} (x : xs) = \]

Step 3: define the simple cases

\[ \text{last} (x : xs) | \text{null} xs = x \]
\[ | \text{otherwise} = \]

Step 4: define the other cases

\[ \text{last} (x : xs) | \text{null} xs = x \]
\[ | \text{otherwise} = \text{last} xs \]

Step 5: generalise and simplify

\[ \text{last} :: [a] \rightarrow [a] \]
\[ \text{last} [x] = x \]
\[ \text{last} (\_ : xs) = \text{last} xs \]
Chapter 7 - Higher-order functions

Exercise 1

\[ \text{map } f (\text{filter } p \text{ xs}) \]

Exercise 2

\[ \text{all } p = \text{and } \circ \text{map } p \]

\[ \text{any } p = \text{or } \circ \text{map } p \]

\[ \text{takeWhile } [] = [] \]

\[ \text{takeWhile } p (x : \text{xs}) \]
\[ \text{takeWhile } p (x : \text{xs}) \]
\[ \text{dropWhile } [] = [] \]

\[ \text{dropWhile } p (x : \text{xs}) \]
\[ \text{dropWhile } p (x : \text{xs}) \]

Exercise 3

\[ \text{map } f = \text{foldr } (\lambda x \text{ xs } \rightarrow \text{f } x : \text{xs}) [] \]

\[ \text{filter } p = \text{foldr } (\lambda x \text{ xs } \rightarrow \text{if } p x \text{ then } x : \text{xs } \text{ else } \text{xs}) [] \]

Exercise 4

\[ \text{dec2nat} = \text{foldl } (\lambda x \text{ y } \rightarrow 10 * x + y) 0 \]

Exercise 5

The functions being composed do not all have the same types. For example:

\[ \text{sum} :: \text{[Int]} \rightarrow \text{Int} \]

\[ \text{map} (\text{[2]}) :: \text{[Int]} \rightarrow \text{[Int]} \]

\[ \text{filter even} :: \text{[Int]} \rightarrow \text{[Int]} \]

Exercise 6

\[ \text{curry} :: ((a, b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c) \]

\[ \text{curry } f = \lambda x \text{ y } \rightarrow \text{f } (x, y) \]

\[ \text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow ((a, b) \rightarrow c) \]

\[ \text{uncurry } f = \lambda(x, y) \rightarrow \text{f } x y \]
Exercise 7

\[
\begin{align*}
\text{chop8} & = \text{unfold null (take 8) (drop 8)} \\
\text{map } f & = \text{unfold null (f \circ \text{head}) tail} \\
\text{iterate } f & = \text{unfold (const False) id } f
\end{align*}
\]

Exercise 8

\[
\begin{align*}
\text{encode} & \quad :: \quad \text{String} \rightarrow [\text{Bit}] \\
\text{encode} & = \text{concat} \circ \text{map (addparity} \circ \text{make8} \circ \text{int2bin} \circ \text{ord)} \\
\text{decode} & \quad :: \quad [\text{Bit}] \rightarrow \text{String} \\
\text{decode} & = \text{map (chr} \circ \text{bin2int} \circ \text{checkparity)} \circ \text{chop9} \\
\text{addparity} & \quad :: \quad [\text{Bit}] \rightarrow [\text{Bit}] \\
\text{addparity } bs & = (\text{parity } bs):bs \\
\text{parity} & \quad :: \quad [\text{Bit}] \rightarrow \text{Bit} \\
\text{parity } bs \mid \text{odd (sum } bs) & = 1 \\
\text{otherwise} & = 0 \\
\text{chop9} & \quad :: \quad [\text{Bit}] \rightarrow [[[\text{Bit}]]) \\
\text{chop9 } [] & = [] \\
\text{chop9 } \text{bits} & = \text{take 9 bits : chop9} (\text{drop 9 bits}) \\
\text{checkparity} & \quad :: \quad [\text{Bit}] \rightarrow [\text{Bit}] \\
\text{checkparity } (b : bs) & \mid b == \text{parity } bs = bs \\
\text{otherwise} & = \text{error "parity mismatch"}
\end{align*}
\]

Exercise 9

No solution required.
Chapter 8 - Functional parsers

Exercise 1

\[
\text{int} = \text{do char } \neg
\]
\[
\quad n \leftarrow \text{nat}
\]
\[
\quad \text{return } (-n)
\]
\[
\quad +++\text{nat}
\]

Exercise 2

\[
\text{comment} = \text{do string "--" }
\]
\[
\quad \text{many } (\text{sat } (\neq 'n'))
\]
\[
\quad \text{return }()
\]

Exercise 3

(1)

(2)
Exercise 4

(1)

```
expr
  └── term      +      expr
      └── factor
        └── nat
            2
```

(2)

```
expr
  └── term
      └── factor
          *      term
              └── nat
                  2
                  └── nat
                      3
```

(3)

```
expr
  └── term      +      expr
      └── factor
          └── ( expr )
              └── term
                  +      expr
                      └── nat
                          4
                          └── factor
                              └── nat
                                  3
```

16
Exercise 5

Without left-factorising the grammar, the resulting parser would backtrack excessively and have exponential time complexity in the size of the expression. For example, a number would be parsed four times before being recognised as an expression.

Exercise 6

```plaintext
expr = do t ← term
    do symbol "+
        e ← expr
        return (t + e)
    +++ do symbol "-
        e ← expr
        return (t - e)
    +++ return t

term = do f ← factor
    do symbol "*
        t ← term
        return (f * t)
    +++ do symbol "/
        t ← term
        return (f \div t)
    +++ return f
```

Exercise 7

(1)
```plaintext
factor ::= atom (↑ factor | epsilon)
atom ::= (expr) | nat
```

(2)
```plaintext
factor ::= Parser Int
factor = do a ← atom
    do symbol "^"
        f ← factor
        return (a ↑ f)
    +++ return a

atom ::= Parser Int
atom = do symbol "("
    e ← expr
    symbol ")"
    return e
    +++ natural
```
Exercise 8

(a)

\[
\begin{align*}
\text{expr} & ::= \text{expr} \ - \ \text{nat} \mid \text{nat} \\
\text{nat} & ::= 0 \mid 1 \mid 2 \mid \cdots 
\end{align*}
\]

(b)

\[
\begin{align*}
\text{expr} = \ & \text{do } e \leftarrow \text{expr} \\
& \text{symbol } "-" \\
& n \leftarrow \text{natural} \\
& \text{return } (e - n) \\
& \text{+++ natural}
\end{align*}
\]

(c)

The parser loops forever without producing a result, because the first operation it performs is to call itself recursively.

(d)

\[
\begin{align*}
\text{expr} = \ & \text{do } n \leftarrow \text{natural} \\
& ns \leftarrow \text{many } (\text{do } \text{symbol } "-" \\
& \text{natural}) \\
& \text{return } (\text{foldl } (-) n ns)
\end{align*}
\]
Chapter 9 - Interactive programs

Exercise 1

\[
\begin{align*}
\text{readLine} & \ = \ get "" \\
\text{get xs} & \ = \ \text{do } x \leftarrow \text{getChar} \\
& \quad \text{case } x \text{ of} \\
& \quad \quad \backslash n' \rightarrow \text{return xs} \\
& \quad \quad \backslash \text{DEL}' \rightarrow \text{if } \text{null xs} \text{ then } \\
& \quad \quad \quad \text{get xs} \\
& \quad \quad \quad \text{else} \\
& \quad \quad \quad \quad \text{do } \text{putStr } \backslash \text{ESC}[1D} \ ackslash \text{ESC}[1D} \\
& \quad \quad \quad \quad \text{get } \text{(init xs)} \\
& \quad \quad \quad \quad \rightarrow \ 	ext{get } (xs ++ [x])
\end{align*}
\]

Exercise 2

No solution available.

Exercise 3

No solution available.

Exercise 4

No solution available.

Exercise 5

No solution available.

Exercise 6

\[
\begin{align*}
\text{type } \text{Board} & \ = \ [\text{Int}] \\
\text{initial} & \ :: \ \text{Board} \\
\text{initial} & \ = \ [5,4,3,2,1] \\
\text{finished} & \ :: \ \text{Board} \rightarrow \text{Bool} \\
\text{finished } b & \ = \ \text{all } (== 0) \ b \\
\text{valid} & \ :: \ \text{Board} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \\
\text{valid } b \ \text{row} \ \text{num} & \ = \ b \ !\! (\text{row} - 1) \geq \text{num} \\
\text{move} & \ :: \ \text{Board} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Board} \\
\text{move } b \ \text{row} \ \text{num} & \ = \ \text{[if } \text{r} == \text{row} \ \text{then } \text{n} - \text{num} \ \text{else } \text{n}} \\
& \quad \quad \quad \mid \ (r, n) \leftarrow \text{zip } [1..5] \ b \\
\text{newline} & \ :: \ \text{IO } () \\
\text{newline} & \ = \ \text{putChar } \backslash \text{n'}
\end{align*}
\]
\begin{verbatim}
putBoard :: Board \to IO ()
putBoard \[a, b, c, d, e]\ = do putRow 1 a
                      putRow 2 b
                      putRow 3 c
                      putRow 4 d
                      putRow 5 e

putRow :: Int \to Int \to IO ()
putRow row num = do putStr (show row)
                   putStr "": "
                   putStrLn (stars num)

stars :: Int \to String
stars n = concat (replicate n "* ")

getDigit :: String \to IO Int
getDigit prom = do putStr prom
                 x ← getChar
                 newline
                 if isDigit x then
                   return (ord x - ord '0')
                 else
                   do putStrLn "ERROR: Invalid digit"
                      getDigit prom

nim :: IO ()
nim = play initial 1

play :: Board \to Int \to IO ()
play board player = do newline
                     putBoard board
                     if finished board then
                       do newline
                           putStr "Player 
                           putStr (show (next player))
                           putStrLn " wins!!"
                     else
                       do newline
                           putStr "Player 
                           putStrLn (show player)
                           r ← getDigit "Enter a row number: 
                           n ← getDigit "Stars to remove : 
                           if valid board r n then
                             play (move board r n) (next player)
                           else
                             do newline
                               putStrLn "ERROR: Invalid move"
                               putStrLn "Player 
                           play board player

next :: Int \to Int
next 1 = 2
next 2 = 1
\end{verbatim}
Chapter 10 - Declaring types and classes

Exercise 1

\[
\begin{align*}
mult \text{ } m \text{ } \text{Zero} & = \text{Zero} \\
mult \text{ } m \text{ } (\text{Succ} \text{ } n) & = \text{add} \text{ } m \text{ } \text{(mult} \text{ } m \text{ } n) \\
\end{align*}
\]

Exercise 2

\[
\begin{align*}
\text{occurs} \text{ } m \text{ } (\text{Leaf} \text{ } n) & = m \text{ } == \text{ } n \\
\text{occurs} \text{ } m \text{ } (\text{Node} \text{ } l \text{ } n \text{ } r) & = \text{case} \text{ } \text{compare} \text{ } m \text{ } n \text{ } \text{of} \\
& \text{LT} \rightarrow \text{occurs} \text{ } m \text{ } l \\
& \text{EQ} \rightarrow \text{True} \\
& \text{GT} \rightarrow \text{occurs} \text{ } m \text{ } r \\
\end{align*}
\]

This version is more efficient because it only requires one comparison for each node, whereas the previous version may require two comparisons.

Exercise 3

\[
\begin{align*}
\text{leaves} \text{ } (\text{Leaf} \text{ } \underline{\text{.}}) & = 1 \\
\text{leaves} \text{ } (\text{Node} \text{ } l \text{ } r) & = \text{leaves} \text{ } l + \text{leaves} \text{ } r \\
\text{balanced} \text{ } (\text{Leaf} \text{ } \underline{\text{.}}) & = \text{True} \\
\text{balanced} \text{ } (\text{Node} \text{ } l \text{ } r) & = \text{abs} \text{ } (\text{leaves} \text{ } l - \text{leaves} \text{ } r) \leq 1 \\
& \land \text{balanced} \text{ } l \land \text{balanced} \text{ } r \\
\end{align*}
\]

Exercise 4

\[
\begin{align*}
\text{halve} \text{ } xs & = \text{splitAt} \text{ } (\text{length} \text{ } xs \text{ } \text{div} \text{ } 2) \text{ } xs \\
\text{balance} \text{ } [x] & = \text{Leaf} \text{ } x \\
\text{balance} \text{ } xs & = \text{Node} \text{ } (\text{balance} \text{ } ys) \text{ } (\text{balance} \text{ } zs) \\
\text{where} \text{ } (ys, zs) & = \text{halve} \text{ } xs \\
\end{align*}
\]

Exercise 5

\[
\begin{align*}
\text{data} \text{ } \text{Prop} & = \cdots | \text{Or} \text{ } \text{Prop} | \text{Equiv} \text{ } \text{Prop} \text{ } \text{Prop} \\
\text{eval} \text{ } s \text{ } (\text{Or} \text{ } p \text{ } q) & = \text{eval} \text{ } s \text{ } p \lor \text{eval} \text{ } s \text{ } q \\
\text{eval} \text{ } s \text{ } (\text{Equiv} \text{ } p \text{ } q) & = \text{eval} \text{ } s \text{ } p \text{ } == \text{eval} \text{ } s \text{ } q \\
\text{vars} \text{ } (\text{Or} \text{ } p \text{ } q) & = \text{vars} \text{ } p \text{ } ++ \text{vars} \text{ } q \\
\text{vars} \text{ } (\text{Equiv} \text{ } p \text{ } q) & = \text{vars} \text{ } p \text{ } ++ \text{vars} \text{ } q \\
\end{align*}
\]

Exercise 6

No solution available.
Exercise 7

\textbf{data} \texttt{Expr} & =& \text{Val} \text{ Int} \mid \text{Add} \text{ Expr} \text{ Expr} \mid \text{Mult} \text{ Expr} \text{ Expr} \\
\textbf{type} \texttt{Cont} & =& [\text{Op}] \\
\textbf{data} \texttt{Op} & =& \text{EVALA} \text{ Expr} \mid \text{ADD} \text{ Int} \mid \text{EVALM} \text{ Expr} \mid \text{MUL} \text{ Int} \\
\textit{eval} &::& \text{Expr} \rightarrow \text{Cont} \rightarrow \text{Int} \\
\textit{eval} (\text{Val} \ n) \text{ ops} &=& \text{exec} \text{ ops} \ n \\
\textit{eval} (\text{Add} \ x \ y) \text{ ops} &=& \text{eval} \ x \ (\text{EVALA} \ y \ : \text{ ops}) \\
\textit{eval} (\text{Mult} \ x \ y) \text{ ops} &=& \text{eval} \ x \ (\text{EVALM} \ y \ : \text{ ops}) \\
\textit{exec} &::& \text{Cont} \rightarrow \text{Int} \rightarrow \text{Int} \\
\textit{exec} [\ ] \ n &=& n \\
\textit{exec} (\text{EVALA} \ y \ : \text{ ops}) \ n &=& \text{eval} \ y \ (\text{ADD} \ n \ : \text{ ops}) \\
\textit{exec} (\text{ADD} \ n \ : \text{ ops}) \ m &=& \text{exec} \text{ ops} \ (n + m) \\
\textit{exec} (\text{EVALM} \ y \ : \text{ ops}) \ n &=& \text{eval} \ y \ (\text{MUL} \ n \ : \text{ ops}) \\
\textit{exec} (\text{MUL} \ n \ : \text{ ops}) \ m &=& \text{exec} \text{ ops} \ (n * m) \\
\textit{value} &::& \text{Expr} \rightarrow \text{Int} \\
\textit{value} \ e &=& \text{eval} \ e \ [\ ] \\

Exercise 8

\textbf{instance} \textit{Monad} \texttt{Maybe} \textbf{where} \\
\textit{return} &::& a \rightarrow \text{Maybe} \ a \\
\textit{return} \ x &=& \text{Just} \ x \\
(\gg\gg) &::& \text{Maybe} \ a \rightarrow (a \rightarrow \text{Maybe} \ b) \rightarrow \text{Maybe} \ b \\
\textit{Nothing} \gg\gg &=& \text{Nothing} \\
(\text{Just} \ x) \gg\gg \ f &=& \text{f} \ x \\

\textbf{instance} \textit{Monad} \texttt{[\ ]} \textbf{where} \\
\textit{return} &::& a \rightarrow \text{[} a \text{]} \\
\textit{return} \ x &=& \text{[} x \text{]} \\
(\gg\gg) &::& \text{[} a \text{]} \rightarrow (a \rightarrow \text{[} b \text{]} ) \rightarrow \text{[} b \text{]} \\
\textit{xs} \gg\gg \ f &=& \text{concat} \ (\text{map} \ f \ \textit{xs})
Chapter 11 - The countdown problem

Exercise 1

\[ choices \, xs = [zs \mid ys \leftarrow \text{subs} \, xs, zs \leftarrow \text{perms} \, ys] \]

Exercise 2

\[
\begin{align*}
\text{removeone} \, x \, [] & = [] \\
\text{removeone} \, x \, (y : ys) & = \begin{cases} x == y & \Rightarrow y \\
\text{otherwise} & \Rightarrow y : \text{removeone} \, x \, ys \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
isChoice \, [] & = \text{True} \\
isChoice \, (x : xs) \, [] & = \text{False} \\
isChoice \, (x : xs) \, ys & = \text{elem} \, x \, ys \land \text{isChoice} \, xs \, (\text{removeone} \, x \, ys)
\end{align*}
\]

Exercise 3

It would lead to non-termination, because recursive calls to \text{exprs} would no longer be guaranteed to reduce the length of the list.

Exercise 4

\[
\begin{align*}
> \, \text{length} \, [e \mid ns' \leftarrow choices \, [1, 3, 7, 10, 25, 50], e \leftarrow \text{exprs} \, ns]
& = 33665406 \\
> \, \text{length} \, [e \mid ns' \leftarrow choices \, [1, 3, 7, 10, 25, 50], e \leftarrow \text{exprs} \, ns, \text{eval} \, e \neq []]
& = 4672540
\end{align*}
\]

Exercise 5

Modifying the definition of \text{valid} by

\[
\begin{align*}
\text{valid} \, \text{Sub} \, x \, y & = \text{True} \\
\text{valid} \, \text{Div} \, x \, y & = y \neq 0 \land x \mod y == 0
\end{align*}
\]

gives

\[
\begin{align*}
> \, \text{length} \, [e \mid ns' \leftarrow choices \, [1, 3, 7, 10, 25, 50], e \leftarrow \text{exprs} \, ns', \text{eval} \, e \neq []]
& = 10839369
\end{align*}
\]

Exercise 6

No solution available.
Chapter 12 - Lazy evaluation

Exercise 1

(1)  
2 * 3 is the only redex, and is both innermost and outermost.

(2)  
1 + 2 and 2 + 3 are redexes, with 1 + 2 being innermost.

(3)  
1 + 2, 2 + 3 and \( \text{fst} (1 + 2, 2 + 3) \) are redexes, with the first of these being innermost and the last being outermost.

(4)  
2 * 3 and \( (\lambda x \rightarrow 1 + x) \) (2 * 3) are redexes, with the first being innermost and the second being outermost.

Exercise 2

Outermost:

\[
\text{fst} (1 + 2, 2 + 3) = \{ \text{applying fst } \} 1 + 2 = \{ \text{applying + } \} 3
\]

Innermost:

\[
\text{fst} (1 + 2, 2 + 3) = \{ \text{applying the first + } \} \text{fst} (3, 2 + 3) = \{ \text{applying + } \} \text{fst} (3, 5) = \{ \text{applying fst } \} 3
\]

Outermost evaluation is preferable because it avoids evaluation of the second argument, and hence takes one less reduction step.

Exercise 3

\[
\text{mult} 3 4 = \{ \text{applying mult } \} (\lambda x \rightarrow (\lambda y \rightarrow x * y)) 3 4 = \{ \text{applying } \lambda x \rightarrow (\lambda y \rightarrow x * y) \} (\lambda y \rightarrow 3 * y) 4 = \{ \text{applying } \lambda y \rightarrow 3 * y \} 3 * 4 = \{ \text{applying * } \} 12
\]
Exercise 4

\[ \text{fibs} = 0 : 1 : [x + y \mid (x, y) \leftarrow \text{zip} \ \text{fibs} \ (\text{tail} \ \text{fibs})] \]

Exercise 5

(1) \[ \text{fib} \ n = \ \text{fibs} !! n \]

(2) \[ \text{head} \ (\text{dropWhile} \ (\leq 1000) \ \text{fibs}) \]

Exercise 6

\begin{align*}
\text{repeatTree} & :: \ a \rightarrow \text{Tree} \ a \\
\text{repeatTree} \ x & = \ \text{Node} \ t \ x \ t \\
& \quad \text{where} \ t = \text{repeatTree} \ x \\
\text{takeTree} & :: \ \text{Int} \rightarrow \text{Tree} \ a \rightarrow \text{Tree} \ a \\
\text{takeTree} \ 0 & = \ \text{Leaf} \\
\text{takeTree} \ (n + 1) \ \text{Leaf} & = \ \text{Leaf} \\
\text{takeTree} \ (n + 1) \ (\text{Node} \ l \ x \ r) & = \ \text{Node} \ (\text{takeTree} \ n \ l) \ x \ (\text{takeTree} \ n \ r) \\
\text{replicateTree} & :: \ \text{Int} \rightarrow a \rightarrow \text{Tree} \ a \\
\text{replicateTree} \ n & = \ \text{takeTree} \ n \circ \text{repeatTree}
\end{align*}
Chapter 13 - Reasoning about programs

Exercise 1

\[
\begin{align*}
\text{last} & :: [a] \to a \\
\text{last} [x] & = x \\
\text{last} (\text{nil} : xs) & = \text{last} xs
\end{align*}
\]

or

\[
\begin{align*}
\text{init} & :: [a] \to [a] \\
\text{init} [\text{nil}] & = [] \\
\text{init} (x : xs) & = x : \text{init} xs
\end{align*}
\]

or

\[
\begin{align*}
\text{foldr1} & :: (a \to a \to a) \to [a] \to a \\
\text{foldr1} [x] & = x \\
\text{foldr1} f (x : xs) & = f x (\text{foldr1} f xs)
\end{align*}
\]

There are a number of other answers too.

Exercise 2

Base case:

\[
\begin{align*}
\text{add} \text{ Zero} (\text{Succ} m) & = \{ \text{applying add} \} \\
\text{Succ} m & = \{ \text{unapplying add} \} \\
\text{Succ} (\text{add} \text{ Zero} m)
\end{align*}
\]

Inductive case:

\[
\begin{align*}
\text{add} (\text{Succ} n) (\text{Succ} m) & = \{ \text{applying add} \} \\
\text{Succ} (\text{add} n (\text{Succ} m)) & = \{ \text{induction hypothesis} \} \\
\text{Succ} (\text{add} (\text{Succ} n) m) & = \{ \text{unapplying add} \} \\
\text{Succ} (\text{add} (\text{Succ} n) m)
\end{align*}
\]

Exercise 3

Base case:

\[
\begin{align*}
\text{add} \text{ Zero} m & = \{ \text{applying add} \} \\
\text{m} & = \{ \text{property of add} \} \\
\text{add} \text{ m Zero}
\end{align*}
\]
Inductive case:

\[
\text{add} (\text{Succ } n) m = \text{Succ} (\text{add } n m) = \text{induction hypothesis} = \text{property of add} = \text{add } m (\text{Succ } n)
\]

Exercise 4

Base case:

\[
\text{all } (== x) (\text{replicate } 0 x) = \text{applying } \text{replicate} = \text{all } (== x) [] = \text{applying } \text{all} = \text{True}
\]

Inductive case:

\[
\text{all } (== x) (\text{replicate } (n + 1) x) = \text{applying } \text{replicate} = \text{all } (== x) (x : \text{replicate } n x) = \text{applying } \text{all} = x == x \land \text{all } (== x) (\text{replicate } n x) = \text{applying } == = \text{True} \land \text{all } (== x) (\text{replicate } n x) = \text{applying } \land = \text{all } (== x) (\text{replicate } n x) = \text{induction hypothesis} = \text{True}
\]

Exercise 5.1

Base case:

\[
[] ++ [] = \text{applying } ++ = []
\]

Inductive case:

\[
(x : xs) ++ [] = \text{applying } ++ = x : (xs ++ []) = \text{induction hypothesis} = x : xs
\]
Exercise 5.2

Base case:

\[
[] ++ (ys ++ zs) = \begin{cases} \text{applying } ++ \end{cases} ys ++ zs = \begin{cases} \text{unapplying } ++ \end{cases} ([]) ++ ys ++ zs
\]

Inductive case:

\[
(x : xs) ++ (ys ++ zs) = \begin{cases} \text{applying } ++ \end{cases} x : (xs ++ (ys ++ zs)) = \begin{cases} \text{induction hypothesis} \end{cases} x : ((xs ++ ys) ++ zs) = \begin{cases} \text{unapplying } ++ \end{cases} (x : (xs ++ ys)) ++ zs = \begin{cases} \text{unapplying } ++ \end{cases} ((x : xs) ++ ys) ++ zs
\]

Exercise 6

The three auxiliary results are all general properties that may be useful in other contexts, whereas the single auxiliary result is specific to this application.

Exercise 7

Base case:

\[
\begin{align*}
\text{map } f (\text{map } g []) &= \begin{cases} \text{applying the inner map} \end{cases} \\
\text{map } f [] &= \begin{cases} \text{applying } \text{map} \end{cases} \\
[] &= \begin{cases} \text{unapplying } \text{map} \end{cases} \\
\text{map } (f \circ g) []
\end{align*}
\]

Inductive case:

\[
\begin{align*}
\text{map } f (\text{map } g (x : xs)) &= \begin{cases} \text{applying the inner map} \end{cases} \\
\text{map } f (g x : \text{map } g xs) &= \begin{cases} \text{applying the outer map} \end{cases} \\
f (g x) : \text{map } f (\text{map } g xs) &= \begin{cases} \text{induction hypothesis} \end{cases} \\
f (g x) : \text{map } (f \circ g) xs &= \begin{cases} \text{unapplying } \circ \end{cases} \\
(f \circ g) x : \text{map } (f \circ g) xs &= \begin{cases} \text{unapplying } \text{map} \end{cases} \\
\text{map } (f \circ g) (x : xs)
\end{align*}
\]
Exercise 8

Base case:
\[
\begin{align*}
take 0 xs \oplus \ drop 0 xs &= \{ \text{applying take, drop} \} \\
\emptyset \oplus xs &= \{ \text{applying \oplus} \} \\
xs &= \{ \text{applying +} \}
\end{align*}
\]

Base case:
\[
\begin{align*}
take (n + 1) \emptyset \oplus \ drop (n + 1) \emptyset &= \{ \text{applying take, drop} \} \\
\emptyset \oplus \emptyset &= \{ \text{applying +} \} \\
\emptyset &= \{ \text{applying leaves} \}
\end{align*}
\]

Inductive case:
\[
\begin{align*}
take (n + 1) (x : xs) \oplus \ drop (n + 1) (x : xs) &= \{ \text{applying take, drop} \} \\
(x : take n xs) \oplus (drop n xs) &= \{ \text{applying +} \} \\
x : (take n xs \oplus drop n xs) &= \{ \text{induction hypothesis} \} \\
x : xs &= \{ \text{unapplying leaves} \}
\end{align*}
\]

Exercise 9

Definitions:
\[
\begin{align*}
leaves (\text{Leaf } n) &= 1 \\
leaves (\text{Node } l r) &= \text{leaves } l + \text{leaves } r \\
nodes (\text{Leaf } n) &= 0 \\
nodes (\text{Node } l r) &= 1 + \text{nodes } l + \text{nodes } r
\end{align*}
\]

Property:
\[
\begin{align*}
\text{leaves } t &= \text{nodes } t + 1
\end{align*}
\]

Base case:
\[
\begin{align*}
nodes (\text{Leaf } n) + 1 &= \{ \text{applying nodes} \} \\
0 + 1 &= \{ \text{applying +} \} \\
1 &= \{ \text{unapplying leaves} \} \\
\text{leaves } (\text{Leaf } n) &= \{ \text{applying leaves} \}
\end{align*}
\]
Inductive case:

\[
\text{nodes} (\text{Node } l \; r) + 1 = \\
\{ \text{applying } \text{nodes} \} \\
1 + \text{nodes } l + \text{nodes } r + 1 = \\
\{ \text{arithmetic} \} \\
(\text{nodes } l + 1) + (\text{nodes } r + 1) = \\
\{ \text{induction hypotheses} \} \\
\text{leaves } l + \text{leaves } r = \\
\{ \text{unapplying leaves} \} \\
\text{leaves} (\text{Node } l \; r)
\]

Exercise 10

Base case:

\[
\text{comp}' (\text{Val } n) \; c = \\
\{ \text{applying } \text{comp}' \} \\
\text{comp} (\text{Val } n) \; c = \\
\{ \text{applying } \text{comp} \} \\
[\text{PUSH } n] \; c = \\
\{ \text{applying } \text{++} \} \\
\text{PUSH } n : c
\]

Inductive case:

\[
\text{comp}' (\text{Add } x \; y) \; c = \\
\{ \text{applying } \text{comp}' \} \\
\text{comp} (\text{Add } x \; y) \; c = \\
\{ \text{applying } \text{comp} \} \\
(\text{comp } x \; \text{comp } y \; [\text{ADD}]) \; c = \\
\{ \text{associativity of } \text{++} \} \\
\text{comp } x \; (\text{comp } y \; ([\text{ADD}] \; c)) = \\
\{ \text{applying } \text{++} \} \\
\text{comp } x \; (\text{comp } y \; (\text{ADD} : c)) = \\
\{ \text{induction hypothesis for } y \} \\
\text{comp } x \; (\text{comp}' y \; (\text{ADD} : c)) = \\
\{ \text{induction hypothesis for } x \} \\
\text{comp}' x (\text{comp}' y (\text{ADD} : c))
\]

In conclusion, we obtain:

\[
\text{comp}' (\text{Val } n) \; c = \text{PUSH } n : c \\
\text{comp}' (\text{Add } x \; y) \; c = \text{comp}' x (\text{comp}' y (\text{ADD} : c))
\]