

# INDEX

## Symbols & Numbers

- `&&` (double ampersand)
  - as Boolean operator conjunction, 2
  - using with folds and lists, 78–79
- ' (apostrophe)
  - using with functions, 7
  - using with types, 149–150
- \* (asterisk)
  - as multiplication function, 3
  - using with kinds, 150
- \*\* (exponentiation), using with RPN functions, 207–208
- \ (backslash), declaring lambdas with, 71
- ` (backticks) using with functions, 4–5
- : (colon)
  - as cons operator
    - bytestring version of, 200
    - using with applicatives, 238–239
    - using with lists, 8–9
  - using with infix constructors, 134
- :: (double colon)
  - using in record syntax, 116
  - using with type annotations, 30, 118
  - using with types, 24
- :k command, identifying kinds with, 150–151
- \$ (function application operator), 80–81, 83
- / (division), using with RPN functions, 207–208
- /= (not-equal-to) operator, 3, 28
- = (equal) sign
  - using with data keyword, 109
  - using with data types, 122
  - using with functions, 5
- == (double equal sign), 3
  - using with Eq type class, 28
  - using with type instances, 139–140
- !! (double exclamation point)
  - in Data.List module, 182
  - using with lists, 9
- > (greater-than) operator, using with lists, 9–10
- >> function, replacing, 279
- >>= (bind) function
  - in A Knight's Quest, 292
  - nested use of, 280
  - using with functions as monads, 311
  - using with monads, 269–270, 272, 274–280, 283–284, 286
  - using with Reader monad, 312
  - using with State monad, 316–317
  - using with Writer type, 302
- > (arrow)
  - in type signature, 60–61
  - using with functions, 25
  - using with lambdas, 71
- > r as functor and monad, 311
- < (less-than) operator, using with lists, 9–10
- <\*> function
  - calling with applicative values, 236
  - left-associative, 233
  - specializing for IO, 234
  - using with applicative style, 232
  - using with liftM function, 325
  - using with zip lists, 237
- <= operator, using with lists, 9–10
- <\$>, using with applicative style, 231–232
- <->, using with I/O actions and functors, 219
- (minus) operator, using with sections, 62
- () (parentheses)
  - minimizing use of, 81, 83
  - placement with functions, 7
  - using with operations, 2, 5
  - using with sections, 62
- (,,) function, using with zip lists, 238

- . (period), using with functions, 89
- .. (dots), using with value constructors, 113–114
- + (plus) operator, 3, 5
- ++ (concatenation) operator
  - excluding from pattern matching, 40
  - using with lists, 8
- ; (semicolon), using with let expressions, 46
- [] (square brackets), using with lists, 7, 24
- [Char] and String types, 30, 127–128
- \_ (underscore)
  - in pattern matching, 38
  - using with lists, 18
- | (vertical pipe)
  - using with data keyword, 109
  - using with data types, 122
  - using with guards, 41
- || as Boolean operator disjunction, 2, 256
- o flag, using in Heathrow to London example, 216
- 3D vector type, implementing, 121–122

## A

- accumulators
  - using with folds, 73
  - using with right folds, 75
  - using with scanl and scanr, 79–80
- addDrink function, 301–302
- algebraic data structures, 137
- algebraic data types, 126–127, 133. *See also* data types
- algebraic expressions, writing, 203–208
- All type, using with monoids, 257
- ampersands (&&)
  - as Boolean operator conjunction, 2
  - using with folds and lists, 78–79
- and function
  - using with applicative functors, 241
  - using with lists, 78
- any function, 92
- Any newtype constructor, using with monoids, 256–257
- apostrophe (')
  - using with functions, 7
  - using with types, 149–150
- appendFile function
  - in to-do list example, 180
  - using in I/O, 180
- applicative functors, 227–228, 237–238, 323. *See also* functors
  - Applicative type class, 228–229, 323
  - functions as, 235–236

- liftA2 function, 238–239
- lists as, 232–234, 243–244, 285–287
- Maybe types as, 269–270
- sequenceA function, 239–242
- upgrading, 267–269
- zip lists, 237
- applicative laws, 238
- applicative operators, vs. monads, 278
- applicative style, using on lists, 233–234
- Applicative type class, 228–229, 323
  - Maybe implementation, 229–230
  - style of pure, 230–232
- applyLog function
  - using with monoids, 300
  - using with Writer monad, 299–300
- arithmetic expressions, 2
- arrow (->)
  - in type signature, 60–61
  - using with functions, 25
  - using with lambdas, 71
- askForNumber function, 197
- as-pattern, 40
- association lists, 98–100. *See also* lists
- associativity
  - defined, 251
  - using with monads, 294–296
- asterisk (\*)
  - as multiplication function, 3
  - using with kinds, 150

## B

- baby.hs* file
  - appending code to, 6
  - saving, 5
- backslash (\), declaring lambdas with, 71
- backticks (`) using with functions, 4–5
- Banana on a Wire example, 278–280
- base case, reaching, 51
- binary functions
  - using on values, 251
  - using with folds, 73
- binary search tree, implementing, 135–137
- bind (>>=) function
  - in A Knight's Quest, 292
  - nested use of, 280
  - using with functions as monads, 311
  - using with monads, 269–270, 272, 274–280, 283–284, 286
  - using with Reader monad, 312
  - using with State monad, 316–317
  - using with Writer type, 302
- binding to variables, 39

- birds
  - ignoring in Pierre example, 278–280
  - representing in Pierre example, 275–278
- BMI (body mass index)
  - calculation of, 41–42
  - listing of, 45
  - repeating calculations of, 43
- Boolean algebra, 2
- Boolean expressions, using with guards, 41
- Boolean values
  - generating randomly, 191
  - for tossing coin, 193
- Bool type, 26, 143–144, 256–257
- Bounded type class, 31–32, 126–127
- bracket function, using in I/O, 178–179
- bracketOnError function, 183–184
- breadcrumbs
  - in filesystem, 355
  - representing in trees, 346–348
  - using with lists and zippers, 352–353
- bytestrings, 198–202. *See also* lists
  - changing types of, 300
  - copying files with, 201–202
  - module functions, 201
  - as monoids, 300
  - strict and lazy, 199–201

## C

- Caesar cipher, 92–94
- calculations, performing once, 42–45
- capital letters, restriction of, 7
- capslocker.hs* program
  - exiting, 171
  - getContent I/O action, 171
  - saving and compiling, 170
- Car data type, 119–120
- case expressions, 48–49
  - vs. if else statements, 48
  - vs. let expressions, 48
  - syntax, 48
- cat program, 180–181
- characters
  - converting into numbers, 96
  - shifting, 93
- [Char] and String types, 30, 127–128
- CharList value constructor, 245–246, 250
- Char type, 26
- chessboard example, 290–292
- circles, representing, 110–112
- class constraints, 140, 142
- class declarations, 140

- code blocks, excluding, 48–49
- coin-toss function, 193–195
- Collatz sequence, 69–70
- colon (:)

  - as cons operator
    - bytestring version of, 200
    - using with applicatives, 238–239
    - using with lists, 8–9
  - using with infix constructors, 134

- command-line arguments, 184–185
- compare function
  - using Ordering type with, 29
  - using with guards, 42
  - using with monoids, 259
- computations
  - deferred, 199
  - performing, 52
- concatenation (++) operator
  - excluding from pattern matching, 40
  - using with lists, 8
- concrete types, 150–151. *See also* data types
- conditions, adding to list comprehensions, 16
- conjunction (&&) Boolean operator, 2
- cons (: ) operator, using with lists, 8–9
- Cons constructor, 133
- context of failure, adding to values, 321.
- Control.Exception bracketOnError, 183–184
- copyFile function, 201
- copying files with bytestrings, 201–202
- Cube.hs* file in Geometry module, 107
- Cuboid.hs* file in Geometry module, 106
- curried functions, 59–62, 222
  - max, 60
  - printing functions, 63
  - sections, 62–63
- cycle function, using with lists, 14

## D

- Data.ByteString.Lazy module, 199
- Data.Char module, 93, 96
- data keyword, 109–110
  - vs. newtype, 244–245, 248–249
  - using, 250
- Data.List module, 88–89. *See also* lists
  - !! function, 182
  - any function, 92
  - delete function, 182
  - group function, 90
  - tails function, 91–92
  - words function, 90

- Data.Map module, 114
    - fromListWith function, 103
    - lookup function, 100
    - Map k v parameterized type, 120
  - Data.Monoid module
    - Product type, 255–256
    - Sum type, 255–256
  - data structures. *See also* zippers
    - reducing to values, 73
    - using zippers with, 352
  - data types. *See also* algebraic data types;
    - concrete types; recursive data structures; type constructors; type parameters; types
    - 3D vector, 121–122
    - applying to type constructors, 150–152
    - defining, 109–110, 122
    - for describing people, 114–117, 123–124
    - identifying, 150–151
    - making, 250
    - record syntax, 116–117
    - wrapping with newtype keywords, 244–245
  - Day type, 127
  - deferred computation, 199
  - definitions, functions as, 7
  - deletetodo.hs* program, saving and compiling, 182
  - derived instances, 122–127. *See also* type classes
    - equating people, 123–124
    - Read type class, 124–125
    - Show type class, 124–125
  - deriving keyword, using with newtype, 245
  - dictionaries, 98
  - difference lists, using, 307–309
  - digitToInt function, 96
  - disjunction (||) Boolean operator, 2
  - div function, 4–5
  - division (/), using with RPN functions, 207–208
  - do expressions. *See also* monads
    - actions of, 219
    - failure of pattern matching in, 284
    - let lines in, 282
    - monadic expressions in, 282
    - monadic values in, 282
    - results of, 318
    - writing, 283
  - do notation, 280–285, 290
    - and <-, 156
    - and list comprehensions, 288
    - pattern matching and failure, 284–285
    - using with Writer monad, 303–304
  - dots (..), using with value constructors, 113–114
  - double colon (::)
    - using in record syntax, 116
    - using with type annotations, 30, 118
    - using with types, 24
  - double equal sign (==), 3
    - using with Eq type class, 28
    - using with type instances, 139–140
  - Double type, 26
  - drop function, using with lists, 12
- ## E
- Either, kind of, 151
  - Either a b type, 130–132, 149–150
  - Either e a type, 321–322
  - elem function
    - using recursively, 55–56
    - using with lists, 12
  - end-of-file character, issuing, 170
  - Enum type class, 31, 126–127
  - equal (=) sign
    - using with data keyword, 109
    - using with data types, 122
    - using with functions, 5
  - equality (== and /=) operators, 3
  - equality testing, 28
  - Eq type class, 28, 122–124, 138–139, 141, 250
  - erroneous computation, representing, 247
  - error function
    - calling, 178
    - using in pattern matching, 39
  - Error instance, 322
  - error messages, 3
  - Euclid’s algorithm, 304–305
  - exceptions, raising, 178, 247–248
  - exponentiation (\*\*), using with RPN functions, 207–208
  - exporting
    - functions, 104
    - shapes in modules, 113–114
  - expressions
    - determining types of, 24
    - equivalent examples of, 71–72
    - lambdas as, 71
    - using operations in, 2
- ## F
- factorial function, 25, 36
  - failure, adding context of, 321
  - False Boolean value, 2–3

- Fibonacci sequence, specifying recursively, 51–52
- file contents vs. handles, 177
- files
  - copying with bytestrings, 201–202
  - processing as strings, 199
  - reading and writing, 175–180
- filesystem
  - manipulating, 357–358
  - moving up and down in, 356–357
  - representing via zippers, 353–358
- filter function, 67–70
  - vs. takeWhile, 80
  - using fold with, 77
- filtering over lists, 198–199
- FilterM monadic function, 328–331
- fixity declaration, 134
- flip function, 65–66, 78
- floating-point numbers, precision of, 337
- Floating type class, 32
- Float type, 25–26
- fmap function
  - concept of, 223
  - as function composition, 222
  - as infix function, 221
  - vs. liftM, 324–325
  - using over functions, 221
  - using with newtype, 246
- folding function
  - using with monoids, 262–265
  - using with RPN, 206–207
- foldl function, 74, 76
  - vs. scanl, 79
  - stack overflow errors, 94–95
- FoldM monadic function, 331–332
- fold pattern, example of, 99
- foldr function, 75–76, 78–79. *See also* right fold function
  - vs. scanr, 79
  - using binary search tree with, 137
- folds
  - accumulators, 73
  - binary functions, 73
  - concept of, 77–78
  - examples, 76–77
  - left vs. right, 75
- forever I/O function, 165–166
- for loops, 198
- forM I/O function, 166–167
- fromListWith function, 103
- fst function
  - type of, 27
  - using with pairs, 20
- function application operator (\$), 80–81, 83
- function composition, 82–84
  - fmap as, 222
  - module functions, 91
  - with multiple parameters, 83–84
  - performing, 89
  - point-free style, 84–85
  - right-associative, 82
- function f, mapping over function g, 310–311
- function parameters, pattern matching on, 48–49
- functional programming, pattern in, 22
- functions
  - . (period) symbol used with, 89
  - accessing, 88
  - as applicatives, 235–236
  - applying for monads, 275–276
  - applying to lists, 66–67
  - applying with - (minus) operator, 347
  - behavior of, 153–154
  - calling, 3–6
  - combining, 6
  - concept of, 61
  - creating, 5–7, 310–311
  - defining, 35–36
    - as definitions, 7
  - exporting from modules, 104
  - filter, 67–70
    - as functors, 220–223, 311
  - importing from modules, 89
  - infix, 3–4
  - lifting, 222
  - loading, 6
  - in local scope, 46
  - map, 66–70
    - mapping with multiple parameters, 70–71
  - as monads, 311
  - optimal path, 212–215
  - partially applied, 60, 64, 71
  - polymorphic, 27
  - prefix, 3–4
  - printing, 63
  - referencing from modules, 89
  - relating to people, 115
  - searching for, 88
  - for shapes, 112–113
  - with side effects, 153–154
  - syntax, 5
  - type declarations, 205
  - types of, 24–25

- functions (*continued*)
  - using, 6–7
  - using once, 71–73
  - value constructors as, 110, 112, 114
  - values returned by, 6–7
  - for vectors, 121–122
  - in where blocks, 45
- functor laws
  - 1 and 2, 223–225
  - breaking, 225–227
- functors, 218, 323. *See also* applicative functors
  - converting maps into, 149–150
  - functions as, 220–223
  - I/O actions as, 218–220
- Functor type class, 146–150, 227
  - definition of, 152
  - Either a type constructor, 149–150
  - Maybe type constructor, 147–148
  - Tree type constructor, 148–149
- functor values, functions in, 227

## G

- gcd function, 304–306
- gcdReverse function, efficiency of, 309
- generics vs. type variables, 27
- gen generator example, 313
- Geometry module, 104–107
- getContentContents I/O action, 171–173
- get function, using with state, 318–319
- getStdGen I/O action, 195–196
- GHC compiler, invoking, 155
- GHCi, let expressions in, 47
- ghci, typing, 1
- ghci> prompt, 1
- girlfriend.txt* file
  - caps-locked version of, 180
  - opening, 175
- global generator, implementing, 195
- greater-than (>) operator, using with lists, 9–10
- greatest common divisor, calculating, 304–305
- group function, using with words function, 90–91
- guard function, using with monads, 289
- guards. *See also* functions
  - vs. if/else trees, 41
  - vs. if expressions, 40–41
  - otherwise, 41
  - vs. patterns, 40–41
  - using, 41–42

## H

- haiku.txt* input, 170
- handles vs. file contents, 177
- Haskell
  - laziness of, 247
  - as pure language, 313
- haystack and needle lists, 91–92
- head function, using with lists, 10–11
- Heathrow to London example
  - optimal path function, 212–215
  - quickest path, 209–211
  - road system, 211–212
  - road system from input, 215–216
  - stack overflow errors, 216
- Hello, world! program
  - compiling, 154–155
  - defining main, 154
  - function types, 155
  - printed output, 155
  - running, 155
- hierarchical modules, 104–106
- higher-order functions. *See also* functions
  - curried functions, 59–64
  - flip, 65–66
  - map, 66–70
  - type declaration, 63
  - zipWith, 64–65
- Hoogle search engine, 88

## I

- id function, 144, 223–224
- if else statements vs. case expressions, 48
- if/else trees vs. guards, 41
- if expressions, 40–41, 143, 145
- if statement, 6–7
- I'll Fly Away example, 276–278
- importing modules, 88–89
- infinite lists, using, 14
- infix functions, 3–5, 12, 27. *See also* functions
  - applying, 62–63
  - defining automatically, 133–134
- init function, using with lists, 10–11
- input, transforming, 173–175
- input redirection, 170
- input streams, getting strings from, 171–173
- instance declarations, 142
- instance keyword, 139
- Integer type, 25
- Integral type class, 33
- interactive mode, starting, 1
- Int type, 25

- I/O (input and output)
  - appendFile function, 180
  - bracket function, 178–179
  - files and streams, 169–175
  - and randomness, 195–198
  - readFile function, 179
  - withFile function, 177–178
  - writeFile function, 179–180
- I/O actions
  - <- vs. let bindings, 159
  - binding names, 158–159
  - do blocks, 219
  - do notation, 156–161
  - as functors, 218–220
  - getArgs, 184–185
  - getContents, 171–173
  - getLine type, 156
  - getProgName, 184–185
  - gluing together, 156–161
  - let syntax, 158–159
  - making from pure value, 160
  - vs. normal values, 157
  - performing, 155, 157
  - results yielded by, 153, 157
  - return function, 160–161
  - reverseWords function, 159–161
  - review, 167
  - in System.Environment module, 184–185
  - tellFortune function, 157
  - using sequenceA function with, 242
  - using with monads, 293
- I/O functions
  - forever, 165–166
  - forM, 166–167
  - mapM, 165
  - print, 162–163
  - putChar, 162
  - putStr, 161–162
  - sequence, 164–165
  - when, 163–164
- IO instance of Applicative, 234–235
- isPrefixOf function, using with strings, 92

## J

- join monadic function, 326–328

## K

- :k command, identifying kinds with, 150–151
- key/value mappings, achieving, 98–104
- Knight's Quest, A (example), 290–292

## L

- lambdas, 71–73. *See also* functions
  - declaring, 71
  - in function composition, 82
  - in Heathrow to London example, 216
  - using with folds, 74
- landLeft and landRight functions, 276–277
- last function, using with lists, 10–11
- less-than (<) operator, using with lists, 9–10
- left fold function, 74. *See also* foldl function
  - in Heathrow to London example, 213–215
  - using with RPN function, 205
- Left value, feeding to functions, 322
- length function, using with lists, 11, 17–18
- let expressions
  - vs. case expressions, 48
  - in GHCi, 47
  - in list comprehensions, 46–47
  - pattern matching with, 46
  - using, 45–46
  - vs. where bindings, 45–46
- let keyword
  - using with lists, 16
  - using with names, 8
- liftA2 function, using with applicative functors, 238–239
- liftM monadic function, 323–326
- list comprehensions, 15–18
  - and do notation, 288
  - pattern matching with, 38–40
  - using with tuples, 21–22
- list monad, 285–287. *See also* monads
- list operations
  - cycle function, 14
  - drop function, 12
  - elem function, 12
  - head function, 10–11
  - init function, 10–11
  - last function, 10–11
  - length function, 11
  - maximum function, 12
  - null function, 11
  - odd function, 16
  - repeat function, 14
  - replicate function, 15
  - reverse function, 11
  - sum function, 12
  - tail function, 10–11
  - take function, 12
- list ranges, using Enum type in, 29. *See also* ranges

- lists. *See also* association lists; bytestrings;
    - Data.List module; task list program;
    - zip lists
    - accessing elements of, 9
    - adding to, 8
    - and function, 78
    - as applicative functors, 232–234, 237–238, 243–244, 285–287
    - applying functions to, 66–67
    - binding elements from, 15
    - checking empty status of, 11
    - combining, 15–18
    - comparing, 9–10
    - concatenation, 8–9
    - construction of, 306–307
    - converting trees to, 265
    - drawing elements from, 15
    - efficiency of, 306–307
    - filtering, 15–18, 198–199
    - folding, 73–74
    - getting last elements of, 77
    - including predicates in, 16–17
    - infinite, 14
    - inside lists, 9
    - managing via module functions, 91–92
    - mapping over, 198–199
    - as monoids, 253–254, 300
    - as nondeterministic computations, 233
    - number ranges in, 13–15
    - pattern matching with, 38–40
    - promise of, 199
    - recursive functions on, 99
    - replacing odd numbers in, 16
    - sorting, 56–58
    - square brackets ([]) used with, 7
    - transforming, 15–18
    - vs. tuples, 18, 20, 24
    - using applicative style on, 233
    - using with filter function, 67
    - using with RPN functions, 205–206
    - using zippers with, 352–353
  - locker codes, looking up, 132
  - logging, adding to programs, 304–306
  - logical or (||), using with monoids, 256
  - log type, changing type of, 300
  - log values. *See also* values
    - applyLog function, 299–300
    - implementing, 305–306
    - using Writer monad for, 298
- M**
- main
    - defining for Hello, world!, 154–155
    - defining for task list, 186
  - map function, 66–70, 73, 75
  - mapM I/O function, 165
  - mappend function
    - using with folds and monoids, 263
    - using with Maybe and Monoid, 260
    - using with Monoid type class, 252, 254
    - using with Ordering values, 258–259
    - using with Writer monad, 300
    - using with Writer type, 303
  - mapping over lists, 198–199
  - maps. *See also* Data.Map module
    - vs. association lists, 100
    - converting association lists to, 100
    - converting into functors, 149–150
    - type of keys in, 120
  - maxBound function, using with Bounded type, 31
  - max function, curried, 60
  - maximum function
    - in recursion example, 52–53
    - using with lists, 12
  - max prefix function, calling, 4
  - Maybe instance, using with Monad type class, 273–280
  - Maybe monad
    - using with trees, 358
    - vs. Writer monad, 299
  - Maybe type, 118–119
    - Applicative implementation, 229–230
    - for folds and monoids, 262
    - as functor, 147–148
    - identifying, 151
    - implementation of >>=, 280
    - as instance of Monoid, 260–261
    - as monad, 269–271
    - wrapping with newtype, 261
  - mconcat function, using with Monoid type class, 252–254, 261
  - mempty function
    - using with Monoid type class, 252, 254–255
    - using with Writer type, 303
    - vs. mzero, 288–289
  - messages
    - decoding, 94
    - encoding, 93
  - minBound function, using with Bounded type, 31
  - min prefix function, calling, 4
  - minus (-) operator, using with sections, 62
  - module functions
    - Caesar cipher, 93–94
    - counting words, 90–91
    - finding numbers, 95–98
    - list management, 91–92
    - on strict left folds, 94–95

- modules. *See also* functions
    - accessing from GHCi, 88
    - advantages of, 87
    - exporting functions, 104
    - exporting shapes in, 113–114
    - geometry, 104–106
    - hierarchical, 106–107
    - importing, 88–89
    - loosely coupled, 87
    - qualified imports of, 89
    - reading source code for, 89
    - referencing functions from, 89
  - monadic functions
    - composing, 335–336
    - FilterM, 328–331
    - FoldM, 331–332
    - join, 326–328
    - liftM, 323–326
  - Monad instance, 311
  - monad laws, 292–293, 339–340
  - MonadPlus type class, 288
  - monads, 323. *See also* do expressions; list
    - monad; monoids; Reader monad; State monad; Writer monad
    - applying functions, 275–276
    - associativity, 294–296
    - do notation, 280–285
    - functions as, 311
    - guard function, 289
    - left identity, 293
    - making, 336–341
    - Maybe types as, 269–271
    - as monoids, 288
    - in mt1 package, 297
    - nested use of >>=, 280
    - nondeterministic values, 285–287
    - purpose of, 268–269
    - right identity, 294
    - using with trees, 358–359
  - MonadState type class, 318–319
  - Monad type class
    - >> function, 273, 279
    - >>= (bind) function, 272–273
    - fail function, 273, 278, 284
    - Maybe instance, 273
    - return function, 272
  - monoids. *See also* monads
    - All type, 257
    - Any newtype constructor, 256–257
    - attaching to values, 302
    - Bool type, 256–257
    - bytestrings as, 300
    - comparing strings, 258–259
    - composition of, 252
    - Data.Monoid module, 255
      - defined, 252
      - folding with, 262–265
      - laws, 253, 255
      - lists as, 253–254, 300
      - monads as, 288
      - newtype keyword, 243–244
      - numbers as, 254–255
      - Ordering type, 257–259
      - type class, 252
      - using with Writer monad, 306–307
  - Monoid type class
    - defining, 252
    - mappend function, 252, 254, 263
    - mconcat function, 252–254, 261
    - mempty function, 252, 254–255
    - newtype keyword, 243–244
  - monoid values, including, 304
  - mt1 package, monads in, 297
  - multiplication (\*) function, 3
  - mzero vs. mempty, 288–289
- ## N
- "\n" (newline) character, adding, 180
  - names
    - defining, 8
    - functions as, 7
  - needle and haystack lists, 91–92
  - negative number constants, 2
  - newline ("\n") character, adding, 180
  - newStdGen action, 196
  - newtype declarations, using record syntax
    - in, 250
  - newtype keyword, 249–250
    - vs. data keyword, 244–245, 248–249
    - using, 247–249
    - using with monoids, 243–244
    - using with Product and Sum types, 255–256
    - using with type class instances, 246–247
    - using with Writer type, 302
    - wrapping Maybe with, 261
  - newtype wrapper, using with State
    - monad, 317
  - NO! alert, 143, 145
  - nondeterministic values
    - representing, 336
    - using with monads, 285–287
  - not Boolean operator, 2
  - not-equal-to (/=) operator, 3, 28
  - Nothing value
    - in do notation, 281
    - in pattern matching, 284–285
    - producing in Banana on a Wire, 278–279
  - null function, using with lists, 11
  - number constants, negative, 2

- number ranges, listing, 13–15
- numbers. *See also* random generators; RPN
  - expressions
    - converting characters into, 96
    - filtering, 288
    - finding via modules, 95–98
    - getting chain of, 69–70
    - guessing, 196–197
    - inserting in `phoneBook`, 101–102
    - as monoids, 254–255
- `Num` type class, 32, 140

## O

- `odd` function, using with lists, 16
- operations
  - precedence of, 4
  - using in expressions, 2
- `or` (`||`) Boolean operator, 2
- `Ordering` type, using with monoids, 257–260
- order of operations, specifying, 2
- `Ord` type class, 28–29, 125–126, 250
- otherwise guards, 41
- output, filtering via list
  - comprehensions, 288

## P

- package, defined, 297
- pairs, storing data in, 20
- parameterized types, 120–122
- parameters, using `=` operator with, 5
- parentheses, ()
  - minimizing use of, 81, 83
  - placement with functions, 7
  - using with operations, 2, 5
  - using with sections, 62
- pattern matching, 35–37
  - as-pattern, 40
  - error function, 39
  - failure in `do` notation, 284–285
  - failure of, 37
  - on function parameters, 48–49
  - with `let` expressions, 46
  - with list comprehensions, 38–40
  - with lists, 38–40
  - tell function, 39
  - with tuples, 37–38
  - using with constructors, 111
  - using with monads, 338
  - using with `newtype` keywords, 247
  - using with type class instances, 140
  - with `where` keyword, 44–45
  - `x:xs` pattern, 38

- patterns
  - vs. guards, 40–41
  - using with RPN functions, 206
- people, describing via data types, 123–124
- performance
  - comparing via `Writer` monad, 309–310
  - enhancing via `bytestrings`, 202
- period (`.`), using with functions, 89
- `phoneBook`
  - association list, 99, 101–104
  - using type synonyms with, 128–129
- Pierre example
  - of `do` notation, 282–284
  - of monads, 274–280
- plus (+) operator, 3, 5
- `Point` data type, using with shapes, 112–113
- point-free style
  - converting function to, 206
  - defining functions in, 84–85
- pole, representing in Pierre example, 274–277
- polymorphic functions, 27
- `pop` function, using with stacks, 314–315, 317
- `powerset`, getting, 330
- predicates
  - adding to list comprehensions, 16–17
  - using with `filter` function, 67
- prefix functions, calling, 3–4
- `Prelude`> prompt, 1
- printing
  - functions, 63
  - text files to terminal, 180–181
- `print` I/O function, 162–163
- probabilities, expressing, 337–339
- problems, implementing solutions to, 205
- `Product` type, using with monoids, 255–256
- programs, 87
  - adding logging to, 304–306
  - exiting, 174
- prompt, changing, 1
- pure method
  - using with applicative functors, 228–230, 232
  - using with zip lists, 237
- `push` function, using with stacks, 314–315
- `putChar` I/O function, 162
- `put` function, using with state, 318–319
- `putStr` I/O function, 161–162
- `putStrLn` function, type of, 155

## Q

- quicksort algorithm, 56–58

## R

- > r, as functor and monad, 311
  - random data, getting, 190–198
  - random function, 320. *See also* functions
    - RandomGen type class, 191
    - Random type class, 191
    - StdGen type, 192
    - type signature, 191
    - using, 192
  - random generators, 313. *See also* numbers
    - making, 192
    - regenerating, 196
  - randomness and I/O, 195–198
  - randoms function, 194–195
  - random string, generating, 195–196
  - ranges. *See also* list ranges
    - using with floating-point numbers, 15
    - using with lists, 13–15
  - Rational data type, 337
  - readability, improving via where keyword, 43
  - Reader monad, 312. *See also* monads
  - readFile function, 179
  - reading files, 175–180
  - Read type class, 29–31
  - record syntax
    - using in newtype declarations, 250
    - using to create data types, 116–117
  - rectangles, representing, 110–112
  - recursion, 51
    - approaching, 58
    - base case, 51
    - in Heathrow to London example, 215
    - in mathematics, 51–52
    - using with applicative functors, 239
    - using with Functor type class, 148–149
  - recursive data structures, 132–137. *See also* data types
    - algebraic data types, 132–133
    - binary search tree, 135–137
    - infix functions, 133–135
  - recursive definition, 194
  - recursive functions, 36, 38. *See also* functions
    - defining, 51
    - elem, 55–56
    - maximum, 52–53
    - operating on lists, 99
    - repeat, 55
    - replicate, 53–54
    - reverse, 55
    - take, 54–55
    - writing, 52–53
    - zip, 55–56
  - repeat function
    - using recursively, 55
    - using with lists, 14
  - replicate function
    - using recursively, 53–54
    - using with lists, 15
  - return function
    - in Monad type class, 272
    - using with Writer type, 303
  - reverse function
    - using fold with, 76–77
    - using recursively, 55
    - using with lists, 11
  - reverse polish notation (RPN), 203–208
  - right fold function, 75–76. *See also* foldr function
  - right triangle, finding, 21–22
  - Right value, feeding to functions, 322
  - road system
    - getting from input, 215–216
    - representing, 211–212
  - RPN (reverse polish notation), 203–208
  - RPN calculator
    - failures, 334
    - folding function, 333–334
    - making safe, 332–334
    - reads function, 333
  - RPN expressions, calculating, 204. *See also* expressions; numbers
  - RPN functions. *See also* functions
    - sketching, 205–206
    - writing, 205–207
  - RPN operators, 207–208
- ## S
- scanl function, 79–80
  - scanr function, 79–80
  - sections, using with infix functions, 62–63
  - semicolon (;), using with let expressions, 46
  - sequenceA function, using with applicative functors, 239–242
  - sequence I/O function, 164–165
  - set comprehensions, 15
  - shapes
    - exporting in modules, 113–114
    - improving with Point data type, 112–113
    - representing, 110–112
  - shortlinesonly.hs program, compiling, 173
  - shortlines.txt file
    - redirecting contents of, 173
    - saving, 172
  - Show type class, 29
  - side effects, 153–154

- snd function, using with pairs, 20
- sorting lists, 56–58
- source code, reading for modules, 89
- Sphere.hs* file, in *Geometry* module, 106
- square brackets ([ ]), using with lists, 7, 24
- square roots, getting for natural numbers, 80
- stack overflow errors, 94, 216
- stacks
  - keeping for RPN functions, 205–206
  - modeling for stateful computations, 314–315
  - popping elements from, 314
  - pushing elements to, 314
- state, getting and setting, 318–319
- stateful computations, 313–314
  - assigning types to, 314
  - stack modeling, 314–315
- State monad. *See also* monads
  - and randomness, 320
  - using, 315–318
- steps, using with ranges in lists, 13–14
- String and [Char] type, 30, 127–128
- strings, 8
  - comparing via monoids, 258–259
  - converting to uppercase, 128
  - encoding, 93
  - getting, 196
  - getting from input streams, 171–173
  - isPrefixOf function, 92
  - processing files as, 199
  - representing values as, 29
- String type, using with type synonyms, 129, 131–132
- subclassing type classes, 140
- subtrees, focusing on, 346–347
- succ: function, calling, 4
- sum function
  - using with fold, 74
  - using with lists, 12, 17–18
- Sum type, using with monoids, 255–256
- System.Environment module
  - getArgs I/O action, 184–185
  - getProgName I/O action, 184–185
- System.IO, openTempFile function, 182
- System.Random module
  - getStdGen I/O action, 195
  - mkStdGen function, 192
  - random function, 191–192

**T**

- :t (type) command, 24, 26, 65
- tail function, using with lists, 10–11
- tails function, 91–92
- take function
  - using recursively, 54–55
  - using with lists, 12
- takeWhile function, 69, 80
- task list program, 188–189. *See also* lists
  - add function, 186–187, 190
  - bad input, 190
  - calling, 186–187
  - dispatch function, 189–190
  - implementing functions, 186–187
  - list-viewing functionality, 187
  - remove function, 187–188
  - running, 189
  - view function, 187
- tasks. *See* to-do list
- tell function, using with log values, 305–306
- terminal
  - printing text files to, 180–181
  - reading from, 175
  - writing to, 175
- text files, printing to terminal, 180–181
- threeCoins stateful computation, 320
- thunk, defined, 199
- to-do list
  - adding tasks to, 185
  - appendFile function, 180
  - bracketOnError function, 183
  - cleaning up, 183–184
  - deleting items from, 181–183
  - functionality, 185
  - removing tasks from, 186
  - viewing tasks, 186
- traffic light, defining states of, 139–140, 144–145
- trees. *See also* zippers
  - balancing, 135
  - converting to lists, 265
  - going to tops of, 351
  - manipulating under focus, 350–351
  - mapping, 148
  - moving up in, 348–350
  - nodes for monoids, 265
  - nonempty node for monoids, 264
  - providing safety nets for, 358–360
  - representing breadcrumbs, 346–348
  - subtrees of, 346–347
  - using monads with, 358–359
  - using with folds and monoids, 263
  - in zippers example, 344–346
- Tree type constructor, as instance of Functor, 148–149
- triangle, right, 21–22

- triples
  - pattern matching, 38
  - using with road system, 212
- True Boolean value, 2–3
- tuples
  - changing vectors to, 19
  - fixed size of, 19–20
  - vs. lists, 18, 20, 24
  - pairs, 19–20
  - pattern matching with, 37–38
  - triples, 19, 21–22
  - as types, 26
  - using, 19–20
  - using commas with, 19
  - using parentheses with, 19
  - using with list comprehensions, 21–22
  - using with road system, 212
  - using with shapes, 110
- two-dimensional vector, representing, 19–20
- type annotations, 29
- type class constraints, 120–121
- type classes, 27, 33, 122–123. *See also*
  - derived instances
    - Bounded, 31–32, 126–127
    - displaying instances of, 142–143
    - Enum, 31, 126–127
    - Eq, 28, 123–124, 138–139, 141
    - Floating, 32
    - Functor, 146–150
    - instances of, 141–143
    - Integral, 33
    - minimum complete definition of, 139
    - Monad, 272–273
    - Num, 32
    - open quality of, 217
    - Ord, 28–29, 125–126
    - Read, 29–31, 124–125
    - reviewing, 138
    - Show, 29, 124–125
    - subclassing, 140
    - using, 250
    - YesNo, 143–146
- type class instances, using newtype with, 246–247
- type constructors, 117. *See also* data types
  - applying types to, 150–152
  - as instances of Functor type class, 218, 225–226
  - parameters, 150
  - type parameters for, 141
  - vs. value constructors, 122, 130
- type declarations, 24–25, 205
  - in higher-order functions, 63
  - for zipWith function, 64
- type inference, 23
- type instances, making, 139–140
- type keyword, 128, 249
- type names, capitalization of, 24, 26
- type parameters, 117–119. *See also*
  - data types
    - passing types as, 118
    - using, 119–121
- types. *See also* data types
  - Bool, 26
  - Char, 26
  - Double, 26
  - Float, 25–26
  - of functions, 24
  - Int, 25
  - Integer, 25
  - tuples as, 26
- type signatures, 110
- type synonyms, 127–132, 249–250
  - Either a b type, 130–132
  - for knight’s position, 290
  - parameterizing, 129–130
  - for zipper in filesystem, 355
- type system, 23
- type variables, 26–27, 231

## U

- undefined value, 247–248
- underscore (`_`)
  - in pattern matching, 38
  - using with lists, 18

## V

- value constructors
  - for Either a b type, 130–131
  - exporting, 113–114
  - as functions, 110, 112, 114
  - parameters, 117
  - vs. type constructors, 122, 130
  - using `..` (dots) with, 113–114
  - using with shapes, 110
- values. *See also* log values
  - adding context of failure to, 321
  - applying functions to, 347
  - attaching monoids to, 302
  - concept of, 343
  - expressing as strings, 29
  - mapping keys to, 98–104
  - reducing data structures to, 73
  - returning in functions, 6–7
  - testing for equality, 3
  - using Ord type class with, 28–29

- values with contexts, using monads with, 268–269
- variables
  - binding to, 39
  - binding via `let` expressions, 45
- vectors
  - changing to tuples, 19
  - implementing types for, 121–122
- vertical pipe (`|`)
  - using with `data` keyword, 109
  - using with data types, 122
  - using with guards, 41

## W

- when I/O function, 163–164
- where bindings vs. `let` expressions, 45–46
- where blocks, functions in, 45
- where keyword, 42–43
  - pattern matching with, 44–45
  - scope of, 44
- while loops, 198
- `withFile` function, using in I/O, 177–178
- words, counting, 90–91
- `words.txt` file, creating and saving, 175
- `writeFile` function, 179–180
- Writer monad, 298–300. *See also* monads
  - adding logging to programs, 304–306
  - `applyLog` function, 299
  - changing log type, 300
  - comparing performance, 309–310
  - difference lists, 307–309
  - inefficient list construction, 306–307
  - vs. `State` monad, 316
  - using `do` notation with, 303–304
  - using monoids with, 300–302, 306–307
- Writer type, 302–303
- writing files, 175–180

## X

- `x:xs` pattern, using, 38

## Y

- YEAH! alert, 143, 145
- `YesNo` type class, 143–146

## Z

- `zip` function
  - using recursively, 55–56
  - using with pairs, 20
- zip lists, 237, 244. *See also* lists
- zippers. *See also* data structures; trees
  - defined, 350
  - filesystem example, 353–358
  - focus of, 350–351
  - for lists, 352–353
  - using with data structures, 352
- `zipWith` function, 64–65, 73