

第十三章：数据结构

关联列表

我们常常会跟一些以键为索引的无序数据打交道。

举个例子，UNIX 管理猿可能需要这么一个列表，它包含系统中所有用户的 UID，以及和这个 UID 相对应的用户名。这个列表根据 UID 而不是数据的位置来查找相应的用户名。换句话说，UID 就是这个数据集的键。

Haskell 里有几种不同的方法来处理这种结构的数据，最常用的两个是关联列表（association list）和 `Data.Map` 模块提供的 `Map` 类型。

关联列表非常简单，易于使用。由于关联列表由 Haskell 列表构成，因此所有列表操作函数都可以用于处理关联列表。

另一方面，`Map` 类型在处理大数据集时，性能比关联列表要好。

本章将同时介绍这两种数据结构。

关联列表就是包含一个或多个 `(key, value)` 元组的列表，`key` 和 `value` 可以是任意类型。一个处理 UID 和用户名映射的关联列表的类型可能是 `[(Integer, String)]`。

[注：关联列表的 `key` 必须是 `Eq` 类型的成员。]

关联列表的构建方式和普通列表一样。Haskell 提供了一个 `Data.List.lookup` 函数，用于在关联列表中查找数据。这个函数的类型签名为 `Eq a => a -> [(a, b)] -> Maybe b`。它的使用方式如下：

```
Prelude> let al = [(1, "one"), (2, "two"), (3, "three"), (4, "four")]

Prelude> lookup 1 al
Just "one"

Prelude> lookup 5 al
Nothing
```

`lookup` 函数的定义如下：

```
-- file: ch13/lookup.hs
myLookup :: Eq a => a -> [(a, b)] -> Maybe b
myLookup _ [] = Nothing
myLookup key ((thiskey, thisval):rest) =
  if key == thiskey
  then Just thisval
  else myLookup key rest
```

`lookup` 在输入列表为空时返回 `Nothing`。如果输入列表不为空，那么它检查当前列表元素的 `key` 是否就是我们要找的 `key`，如果是的话就返回和这个 `key` 对应的 `value`，否则就继续递归处理剩余的列表元素。

再来看一个稍微复杂点的例子。在 Unix/Linux 系统中，有一个 `/etc/passwd` 文件，这个文件保存了用户名称，UID，用户的 HOME 目录位置，以及其他一些数据。文件以行分割每个用户的资料，每个数据域用冒号隔开：

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
jgoerzen:x:1000:1000:John Goerzen,,,:/home/jgoerzen:/bin/bash
```

 v: latest ▾

以下程序读入并处理 `/etc/passwd` 文件，它创建一个关联列表，使得我们可以根据给定 UID，获取相应的用户名：

```

-- file: ch13/passwd-al.hs
import Data.List
import System.IO
import Control.Monad(when)
import System.Exit
import System.Environment(getArgs)

main = do
    -- Load the command-line arguments
    args <- getArgs

    -- If we don't have the right amount of args, give an error and abort
    when (length args /= 2) $ do
        putStrLn "Syntax: passwd-al filename uid"
        exitFailure

    -- Read the file lazily
    content <- readFile (args !! 0)

    -- Compute the username in pure code
    let username = findByUID content (read (args !! 1))

    -- Display the result
    case username of
        Just x -> putStrLn x
        Nothing -> putStrLn "Could not find that UID"

    -- Given the entire input and a UID, see if we can find a username.
    findByUID :: String -> Integer -> Maybe String
    findByUID content uid =
        let al = map parseline . lines $ content
            in lookup uid al

    -- Convert a colon-separated line into fields
    parseline :: String -> (Integer, String)
    parseline input =
        let fields = split ':' input
            in (read (fields !! 2), fields !! 0)

    -- Takes a delimiter and a list.
    -- Break up the list based on the delimiter.
    split :: Eq a => a -> [a] -> [[a]]

    -- If the input is empty, the result is a list of empty lists.
    split _ [] = [[]]
    split delimiter str =
        let
            -- Find the part of the list before delimiter and put it in "before".
            -- The result of the list, including the leading delimiter, goes in "remainder".
            (before, remainder) = span (/= delimiter) str
            in before : case remainder of
                [] -> []
                x -> -- If there is more data to process,
                    -- call split recursively to process it
                    split delimiter (tail x)

```

`findByUID` 是整个程序的核心，它逐行读入并处理输入，使用 `lookup` 从处理结果中查找给定 UID：

```

*Main> findByUID "root:x:0:0:root:/root:/bin/bash" 0
Just "root"

```

`parseline` 读入并处理一个字符串，返回一个包含 UID 和用户名的元组：

```

*Main> parseline "root:x:0:0:root:/root:/bin/bash"
(0,"root")

```

`split` 函数根据给定分隔符 `delimiter` 将一个文本行分割为列表：

 v: latest ▾

```
*Main> split ':' 'root:x:0:0:root:/root:/bin/bash'
["root","x","0","0","root","/root","/bin/bash"]
```

以下是在本机执行 `passwd-al.hs` 处理 `/etc/passwd` 的结果：

```
$ runghc passwd-al.hs /etc/passwd 0
root

$ runghc passwd-al.hs /etc/passwd 10086
Could not find that UID
```

Map 类型

`Data.Map` 模块提供的 `Map` 类型的行为和关联列表类似，但 `Map` 类型的性能更好。

`Map` 和其他语言提供的哈希表类似。不同的是，`Map` 的内部由平衡二叉树实现，在 Haskell 这种使用不可变数据的语言中，它是一个比哈希表更高效的表示。这是一个非常明显的例子，说明纯函数式语言是如何深入地影响我们编写程序的方式：对于一个给定的任务，我们总是选择合适的算法和数据结构，使得解决方案尽可能地简单和有效，但这些（纯函数式的）选择通常不同于命令式语言处理同样问题时的选择。

因为 `Data.Map` 模块的一些函数和 `Prelude` 模块的函数重名，我们通过 `import qualified Data.Map as Map` 的方式引入模块，并使用 `Map.name` 的方式引用模块中的名字。

先来看看如何用几种不同的方式构建 `Map`：

```
-- file: ch13/buildmap.hs
import qualified Data.Map as Map

-- Functions to generate a Map that represents an association list
-- as a map

al = [(1, "one"), (2, "two"), (3, "three"), (4, "four")]

-- Create a map representation of 'al' by converting the association
-- list using Map.fromList
mapFromAL =
    Map.fromList al

-- Create a map representation of 'al' by doing a fold
mapFold =
    foldl (\map (k, v) -> Map.insert k v map) Map.empty al

-- Manually create a map with the elements of 'al' in it
mapManual =
    Map.insert 2 "two" .
    Map.insert 4 "four" .
    Map.insert 1 "one" .
    Map.insert 3 "three" $ Map.empty
```

`Map.insert` 函数处理数据的方式非常『Haskell 化』：它返回经过函数应用的输入数据的副本。这种处理数据的方式在操作多个 `Map` 时非常有用，它意味着你可以像前面代码中 `mapFold` 那样使用 `fold` 来构建一个 `Map`，又或者像 `mapManual` 那样，串连起多个 `Map.insert` 调用。

[译注：这里说『Haskell 化』实际上就是『函数式化』，对于函数式语言来说，最常见的函数处理方式是接受一个输入，然后返回一个输出，输出是另一个独立的值，且原输入不会被修改。]

现在，到 `ghci` 中验证一下是否所有定义都如我们所预期的那样工作：

```
Prelude> :l buildmap.hs
[1 of 1] Compiling Main           ( buildmap.hs, interpreted )
Ok, modules loaded: Main.

*Main> al
Loading package array-0.4.0.0 ... linking ... done.
Loading package deepseq-1.3.0.0 ... linking ... done.
Loading package containers-0.4.2.1 ... linking ... done.
```

 v: latest ▾

```
[(1, "one"), (2, "two"), (3, "three"), (4, "four")]

*Main> mapFromAL
fromList [(1, "one"), (2, "two"), (3, "three"), (4, "four")]

*Main> mapFold
fromList [(1, "one"), (2, "two"), (3, "three"), (4, "four")]

*Main> mapManual
fromList [(1, "one"), (2, "two"), (3, "three"), (4, "four")]
```

注意，`Map` 并不保证它的输出排列和原本的输入排列一致，对比 `mapManual` 的输入和输出可以看出这一点。

`Map` 的操作方式和关联列表类似。`Data.Map` 模块提供了一组函数，用于增删 `Map` 元素，对 `Map` 进行过滤、修改和 `fold`，以及在 `Map` 和关联列表之间进行转换。`Data.Map` 模块本身的文档非常优秀，因此我们在这里不会详细讲解每个函数，而是在本章的后续内容中，通过例子来介绍这些概念。

函数也是数据

Haskell 语言的威力部分在于它可以让我们方便地创建并操作函数。

以下示例展示了怎样将函数保存到记录的域中：

```
-- file: ch13/funcsecs.hs

-- Our usual CustomColor type to play with
data CustomColor =
  CustomColor {red :: Int,
               green :: Int,
               blue :: Int}
  deriving (Eq, Show, Read)

-- A new type that stores a name and a function.
-- The function takes an Int, applies some computation to it,
-- and returns an Int along with a CustomColor
data FuncRec =
  FuncRec {name :: String,
          colorCalc :: Int -> (CustomColor, Int)}

plus5func color x = (color, x + 5)

purple = CustomColor 255 0 255

plus5 = FuncRec {name = "plus5", colorCalc = plus5func purple}
always0 = FuncRec {name = "always0", colorCalc = \_ -> (purple, 0)}
```

注意 `colorCalc` 域的类型：它是一个函数，接受一个 `Int` 类型值作为参数，并返回一个 `(CustomColor, Int)` 元组。

我们创建了两个 `FuncRec` 记录：`plus5` 和 `always0`，这两个记录的 `colorCalc` 域都总是返回紫色（purple）。`FuncRec` 自身并没有域去保存所使用的颜色，颜色的值被保存在函数当中——我们称这种用法为**闭包**。

以下是示例代码：

```
*Main> :l funcsecs.hs
[1 of 1] Compiling Main           ( funcsecs.hs, interpreted )
Ok, modules loaded: Main.

*Main> :t plus5
plus5 :: FuncRec

*Main> name plus5
"plus5"

*Main> :t colorCalc plus5
colorCalc plus5 :: Int -> (CustomColor, Int)

*Main> (colorCalc plus5) 7
```

 v: latest ▾

```
(CustomColor {red = 255, green = 0, blue = 255}, 12)

*Main> :t colorCalc always0
colorCalc always0 :: Int -> (CustomColor, Int)

*Main> (colorCalc always0) 7
(CustomColor {red = 255, green = 0, blue = 255}, 0)
```

上面的程序工作得很好，但我们还想做一些更有趣的事，比如说，在多个域中使用同一段数据。可以使用一个类型构造函数来做到这一点：

```
-- file: ch13/funcrcs2.hs
data FuncRec =
  FuncRec {name :: String,
           calc :: Int -> Int,
           namedCalc :: Int -> (String, Int)}

mkFuncRec :: String -> (Int -> Int) -> FuncRec
mkFuncRec name calcfunc =
  FuncRec {name = name,
           calc = calcfunc,
           namedCalc = \x -> (name, calcfunc x)}

plus5 = mkFuncRec "plus5" (+ 5)
always0 = mkFuncRec "always0" (\_ -> 0)
```

`mkFuncRecs` 函数接受一个字符串和一个函数作为参数，返回一个新的 `FuncRec` 记录。以下是对 `mkFuncRecs` 函数的测试：

```
*Main> :l funcrcs2.hs
[1 of 1] Compiling Main           ( funcrcs2.hs, interpreted )
Ok, modules loaded: Main.

*Main> :t plus5
plus5 :: FuncRec

*Main> name plus5
"plus5"

*Main> (calc plus5) 5
10

*Main> (namedCalc plus5) 5
("plus5", 10)

*Main> let plus5a = plus5 {name = "PLUS5A"}

*Main> name plus5a
"PLUS5A"

*Main> (namedCalc plus5a) 5
("plus5", 10)
```

注意 `plus5a` 的创建过程：我们改变了 `plus5` 的 `name` 域，但没有修改它的 `namedCalc` 域。这就是为什么调用 `name` 会返回新名字，而 `namedCalc` 依然返回原本使用 `mkFuncRecs` 创建时设置的名字 —— 除非我们显式地修改域，否则它们不会被改变。

扩展示例：/etc/passwd

以下是一个扩展示例，它展示了几种不同的数据结构的用法，根据 `/etc/passwd` 文件的格式，程序处理并保存它的实体（entry）：

```
-- file: ch13/passwdmap.hs
import Data.List
import qualified Data.Map as Map
import System.IO
import Text.Printf (printf)
import System.Environment (getArgs)
import System.Exit
import Control.Monad (when)
```

 v: latest ▾

```

{- | The primary piece of data this program will store.
   It represents the fields in a POSIX /etc/passwd file -}
data PasswdEntry = PasswdEntry {
  userName :: String,
  password :: String,
  uid :: Integer,
  gid :: Integer,
  gecos :: String,
  homeDir :: String,
  shell :: String}
  deriving (Eq, Ord)

{- | Define how we get data to a 'PasswdEntry'. -}
instance Show PasswdEntry where
  show pe = printf "%s:%s:%d:%d:%s:%s:%s"
    (userName pe) (password pe) (uid pe) (gid pe)
    (gecos pe) (homeDir pe) (shell pe)

{- | Converting data back out of a 'PasswdEntry'. -}
instance Read PasswdEntry where
  readsPrec _ value =
    case split ':' value of
      [f1, f2, f3, f4, f5, f6, f7] ->
        -- Generate a 'PasswdEntry' the shorthand way:
        -- using the positional fields. We use 'read' to convert
        -- the numeric fields to Integers.
        [(PasswdEntry f1 f2 (read f3) (read f4) f5 f6 f7, [])]
      x -> error $ "Invalid number of fields in input: " ++ show x
  where
    {- | Takes a delimiter and a list. Break up the list based on the
       - delimiter. -}
    split :: Eq a => a -> [a] -> [[a]]

    -- If the input is empty, the result is a list of empty lists.
    split _ [] = [[]]
    split delim str =
      let -- Find the part of the list before delim and put it in
          -- "before". The rest of the list, including the leading
          -- delim, goes in "remainder".
          (before, remainder) = span (/= delim) str
      in
        before : case remainder of
          [] -> []
          x -> -- If there is more data to process,
                -- call split recursively to process it
                split delim (tail x)

-- Convenience aliases; we'll have two maps: one from UID to entries
-- and the other from username to entries
type UIDMap = Map.Map Integer PasswdEntry
type UserMap = Map.Map String PasswdEntry

{- | Converts input data to maps. Returns UID and User maps. -}
inputToMaps :: String -> (UIDMap, UserMap)
inputToMaps inp =
  (uidmap, usermap)
  where
    -- fromList converts a [(key, value)] list into a Map
    uidmap = Map.fromList . map (\pe -> (uid pe, pe)) $ entries
    usermap = Map.fromList .
      map (\pe -> (userName pe, pe)) $ entries
    -- Convert the input String to [PasswdEntry]
    entries = map read (lines inp)

main = do
  -- Load the command-line arguments
  args <- getArgs

  -- If we don't have the right number of args,
  -- give an error and abort

```

```

when (length args /= 1) $ do
  putStrLn "Syntax: passwdmap filename"
  exitFailure

-- Read the file lazily
content <- readFile (head args)
let maps = inputToMaps content
mainMenu maps

mainMenu maps@(uidmap, usermap) = do
  putStr optionText
  hFlush stdout
  sel <- getLine
  -- See what they want to do. For every option except 4,
  -- return them to the main menu afterwards by calling
  -- mainMenu recursively
  case sel of
    "1" -> lookupUserName >> mainMenu maps
    "2" -> lookupUID >> mainMenu maps
    "3" -> displayFile >> mainMenu maps
    "4" -> return ()
    _ -> putStrLn "Invalid selection" >> mainMenu maps

where
lookupUserName = do
  putStrLn "Username: "
  username <- getLine
  case Map.lookup username usermap of
    Nothing -> putStrLn "Not found."
    Just x -> print x
lookupUID = do
  putStrLn "UID: "
  uidstring <- getLine
  case Map.lookup (read uidstring) uidmap of
    Nothing -> putStrLn "Not found."
    Just x -> print x
displayFile =
  putStr . unlines . map (show . snd) . Map.toList $ uidmap
optionText =
  "\npasswdmap options:\n\
  \\n\
  \1 Look up a user name\n\
  \2 Look up a UID\n\
  \3 Display entire file\n\
  \4 Quit\n\
  \Your selection: "

```

示例程序维持两个 Map：一个从用户名映射到 PasswdEntry，另一个从 UID 映射到 PasswdEntry。有数据库使用经验的人可以将它们看作是两个不同数据域的索引。

根据 /etc/passwd 文件的格式，PasswdEntry 的 Show 和 Read 实例分别用于显示 (display) 和处理 (parse) 工作。

扩展示例：数字类型 (Numeric Types)

我们已经讲过 Haskell 的类型系统有多强大，表达能力有多强。我们已经讲过很多利用这种能力的方法。现在我们来举一个实际的例子看看。

在 **数字类型** 一节中，我们展示了 Haskell 的数字类型类。现在，我们来定义一些类，然后用数字类型类把它们和 Haskell 的基本数学结合起来，看看能得到什么。

我们先来想想我们想用这些新类型在 ghci 里干什么。首先，一个不错的选择是把数学表达式转成字符串，并确保它显示了正确的优先级。我们可以写一个 prettyShow 函数来实现。稍后我们就告诉你怎么写，先来看看怎么用它。

```

ghci> :l num.hs
[1 of 1] Compiling Main           ( num.hs, interpreted )
Ok, modules loaded: Main.
ghci> 5 + 1 * 3
8

```

 v: latest ▾

```
ghci> prettyShow $ 5 + 1 * 3
"5+(1*3)"
ghci> prettyShow $ 5 * 1 + 3
"(5*1)+3"
```

看起来不错，但还不够聪明。我们可以很容易地把 $1 * 3$ 从表达式里拿掉。写个函数来简化怎么样？

```
ghci> prettyShow $ simplify $ 5 + 1 * 3
"5+3"
```

把数学表达式转成逆波兰表达式（RPN）怎么样？RPN 是一种后缀表示法，它不要求括号，常见于 HP 计算器。RPN 是一种基于栈的表达式。我们把数字放进栈里，当碰到操作符时，栈顶的数字出栈，结果再被放回栈里。

```
ghci> rpnShow $ 5 + 1 * 3
"5 1 3 * +"
ghci> rpnShow $ simplify $ 5 + 1 * 3
"5 3 +"
```

能表示含有未知符号的简单表达式也很不错。

```
ghci> prettyShow $ 5 + (Symbol "x") * 3
"5+(x*3)"
```

跟数字打交道时，单位常常很重要。例如，当你看见数字5时，它是5米，5英尺，还是5字节？当然，当你用5米除以2秒时，系统应该推出来正确的单位。而且，它应该阻止你用2秒加上5米。

```
ghci> 5 / 2
2.5
ghci> (units 5 "m") / (units 2 "s")
2.5_m/s
ghci> (units 5 "m") + (units 2 "s")
*** Exception: Mis-matched units in add
ghci> (units 5 "m") + (units 2 "m")
7_m
ghci> (units 5 "m") / 2
2.5_m
ghci> 10 * (units 5 "m") / (units 2 "s")
25.0_m/s
```

如果我们定义的表达式或函数对所有数字都合法，那我们就应该能计算出结果，或者把表达式转成字符串。例如，如果我们定义 `test` 的类型为 `Num a => a`，并令 `test = 2 * 5 + 3`，那我们应该可以：

```
ghci> test
13
ghci> rpnShow test
"2 5 * 3 +"
ghci> prettyShow test
"(2*5)+3"
ghci> test + 5
18
ghci> prettyShow (test + 5)
"((2*5)+3)+5"
ghci> rpnShow (test + 5)
"2 5 * 3 + 5 +"
```

既然我们能处理单位，那我们也应该能处理一些基本的三角函数，其中很多操作都是关于角的。让我们确保角度和弧度都能被处理。

```
ghci> sin (pi / 2)
1.0
ghci> sin (units (pi / 2) "rad")
1.0_1.0
ghci> sin (units 90 "deg")
1.0_1.0
```

 v: latest ▾


```
ghci> (units 50 "m") * sin (units 90 "deg")
50.0_m
```

最后，我们应该能把这些都放在一起，把不同类型的表达式混合使用。

```
ghci> ((units 50 "m") * sin (units 90 "deg")) :: Units (SymbolicManip Double)
50.0*sin(((2.0*pi)*90.0)/360.0)_m
ghci> prettyShow $ dropUnits $ (units 50 "m") * sin (units 90 "deg")
"50.0*sin(((2.0*pi)*90.0)/360.0)"
ghci> rpnShow $ dropUnits $ (units 50 "m") * sin (units 90 "deg")
"50.0 2.0 pi * 90.0 * 360.0 / sin *"
ghci> (units (Symbol "x") "m") * sin (units 90 "deg")
x*sin(((2.0*pi)*90.0)/360.0)_m
```

你刚才看到的一切都可以用 Haskell 的类型和类型类实现。实际上，你看到的正是我们马上要实现的 `num.hs`。

第一步

我们想想怎么实现上面提到的功能。首先，用 `ghci` 查看一下可知，`(+)` 的类型是 `Num a => a -> a -> a`。如果我们想给加号实现一些自定义行为，我们就必须定义一个新类型并声明它为 `Num` 的实例。这个类型得用符号的形式来存储表达式。我们可以从加法操作开始。我们需要存储操作符本身、左侧以及右侧内容。左侧和右侧内容本身又可以是表达式。

我们可以把表达式想象成一棵树。让我们从一些简单类型开始。

```
-- file: ch13/numsimple.hs
-- 我们支持的操作符
data Op = Plus | Minus | Mul | Div | Pow
        deriving (Eq, Show)

{- 核心符号操作类型 (core symbolic manipulation type) -}
data SymbolicManip a =
    Number a                -- Simple number, such as 5
  | Arith Op (SymbolicManip a) (SymbolicManip a)
    deriving (Eq, Show)

{- SymbolicManip 是 Num 的实例。定义 SymbolicManip 实现 Num 的函数。如(+)等。 -}
instance Num a => Num (SymbolicManip a) where
    a + b = Arith Plus a b
    a - b = Arith Minus a b
    a * b = Arith Mul a b
    negate a = Arith Mul (Number (-1)) a
    abs a = error "abs is unimplemented"
    signum _ = error "signum is unimplemented"
    fromInteger i = Number (fromInteger i)
```

首先我们定义了 `Op` 类型。这个类型表示我们要支持的操作。接着，我们定义了 `SymbolicManip a`，由于 `Num a` 约束的存在，`a` 可替换为任何 `Num` 实例。我们可以有 `SymbolicManip Int` 这样的具体类型。

`SymbolicManip` 类型可以是数字，也可以是数学运算。`Arith` 构造器是递归的，这在 Haskell 里完全合法。`Arith` 用一个 `Op` 和两个 `SymbolicManip` 创建了一个 `SymbolicManip`。我们来看一个例子：

```
Prelude> :l numsimple.hs
[1 of 1] Compiling Main                ( numsimple.hs, interpreted )
Ok, modules loaded: Main.
*Main> Number 5
Number 5
*Main> :t Number 5
Number 5 :: Num a => SymbolicManip a
*Main> :t Number (5::Int)
Number (5::Int) :: SymbolicManip Int
*Main> Number 5 * Number 10
Arith Mul (Number 5) (Number 10)
*Main> (5 * 10)::SymbolicManip Int
Arith Mul (Number 5) (Number 10)
*Main> (5 * 10 + 2)::SymbolicManip Int
Arith Plus (Arith Mul (Number 5) (Number 10)) (Number 2)
```

 v: latest ▾

可以看到，我们已经可以表示一些简单的表达式了。注意观察 Haskell 是如何把 $5 * 10 + 2$ “转换”成 `SymbolicManip` 值的，它甚至还正确处理了求值顺序。事实上，这并不是真正意义上的转换，因为 `SymbolicManip` 已经是一等数字（first-class number）了。就算 `Integer` 类型的数字字面量（numeric literals）在内部也是被包装在 `fromInteger` 里的，所以 `5` 作为一个 `SymbolicManip Int` 和作为一个 `Int` 同样有效。

从这儿开始，我们的任务就简单了：扩展 `SymbolicManip`，使它能表示所有我们想要的操作；把它声明为其它数字类型类的实例；为 `SymbolicManip` 实现我们自己的 `Show` 实例，使这棵树在显示时更友好。

完整代码

这里是完整的 `num.hs`，我们在本节开始的 `ghci` 例子中用到了它。我们来一点一点分析这段代码。

```
-- file: ch13/num.hs
import Data.List

-----

-- Symbolic/units manipulation
-----

-- The "operators" that we're going to support
data Op = Plus | Minus | Mul | Div | Pow
    deriving (Eq, Show)

{- The core symbolic manipulation type. It can be a simple number,
a symbol, a binary arithmetic operation (such as +), or a unary
arithmetic operation (such as cos)

Notice the types of BinaryArith and UnaryArith: it's a recursive
type. So, we could represent a (+) over two SymbolicManips. -}
data SymbolicManip a =
    Number a           -- Simple number, such as 5
  | Symbol String       -- A symbol, such as x
  | BinaryArith Op (SymbolicManip a) (SymbolicManip a)
  | UnaryArith String (SymbolicManip a)
    deriving (Eq)
```

我们在这段代码中定义了 `Op`，和之前我们用到的一样。我们也定义了 `SymbolicManip`，它和我们之前用到的类似。在这个版本中，我们开始支持一元数学操作（unary arithmetic operations）（也就是接受一个参数的操作），例如 `abs` 和 `cos`。接下来我们来定义自己的 `Num` 实例。

```
-- file: ch13/num.hs
{- SymbolicManip will be an instance of Num. Define how the Num
operations are handled over a SymbolicManip. This will implement things
like (+) for SymbolicManip. -}
instance Num a => Num (SymbolicManip a) where
    a + b = BinaryArith Plus a b
    a - b = BinaryArith Minus a b
    a * b = BinaryArith Mul a b
    negate a = BinaryArith Mul (Number (-1)) a
    abs a = UnaryArith "abs" a
    signum _ = error "signum is unimplemented"
    fromInteger i = Number (fromInteger i)
```

非常直观，和之前的代码很像。注意之前我们不支持 `abs`，但现在可以了，因为有了 `UnaryArith`。接下来，我们再定义几个实例。

```
-- file: ch13/num.hs
{- 定义 SymbolicManip 为 Fractional 实例 -}
instance (Fractional a) => Fractional (SymbolicManip a) where
    a / b = BinaryArith Div a b
    recip a = BinaryArith Div (Number 1) a
    fromRational r = Number (fromRational r)

{- 定义 SymbolicManip 为 Floating 实例 -}
instance (Floating a) => Floating (SymbolicManip a) where
    pi = Symbol "pi"
    exp a = UnaryArith "exp" a
```

 v: latest ▾

```

log a = UnaryArith "log" a
sqrt a = UnaryArith "sqrt" a
a ** b = BinaryArith Pow a b
sin a = UnaryArith "sin" a
cos a = UnaryArith "cos" a
tan a = UnaryArith "tan" a
asin a = UnaryArith "asin" a
acos a = UnaryArith "acos" a
atan a = UnaryArith "atan" a
sinh a = UnaryArith "sinh" a
cosh a = UnaryArith "cosh" a
tanh a = UnaryArith "tanh" a
asinh a = UnaryArith "asinh" a
acosh a = UnaryArith "acosh" a
atanh a = UnaryArith "atanh" a

```

这段代码直观地定义了 `Fractional` 和 `Floating` 实例。接下来，我们把表达式转换字符串。

```

-- file: ch13/num.hs
{- 使用常规代数表示法，把 SymbolicManip 转换为字符串 -}
prettyShow :: (Show a, Num a) => SymbolicManip a -> String

-- 显示字符或符号
prettyShow (Number x) = show x
prettyShow (Symbol x) = x

prettyShow (BinaryArith op a b) =
  let pa = simpleParen a
      pb = simpleParen b
      pop = op2str op
  in pa ++ pop ++ pb
prettyShow (UnaryArith opstr a) =
  opstr ++ "(" ++ show a ++ ")"

op2str :: Op -> String
op2str Plus = "+"
op2str Minus = "-"
op2str Mul = "*"
op2str Div = "/"
op2str Pow = "**"

{- 在需要的地方添加括号。这个函数比较保守，有时候不需要也会加。
Haskell 在构建 SymbolicManip 的时候已经处理好优先级了。 -}
simpleParen :: (Show a, Num a) => SymbolicManip a -> String
simpleParen (Number x) = prettyShow (Number x)
simpleParen (Symbol x) = prettyShow (Symbol x)
simpleParen x@(BinaryArith _ _ _) = "(" ++ prettyShow x ++ ")"
simpleParen x@(UnaryArith _ _) = prettyShow x

{- 调用 prettyShow 函数显示 SymbolicManip 值 -}
instance (Show a, Num a) => Show (SymbolicManip a) where
  show a = prettyShow a

```

首先我们定义了 `prettyShow` 函数。它把一个表达式转换成常规表达形式。算法相当简单：数字和符号不做处理；二元操作是转换后两侧的内容加上中间的操作符；当然我们也处理了一元操作。`op2str` 把 `Op` 转为 `String`。在 `simpleParen` 里，我们加括号的算法非常保守，以确保优先级在结果里清楚显示。最后，我们声明 `SymbolicManip` 为 `Show` 的实例然后用 `prettyShow` 来实现。现在，我们来设计一个算法把表达式转为 RPN 形式的字符串。

```

-- file: ch13/num.hs
{- Show a SymbolicManip using RPN. HP calculator users may
find this familiar. -}
rpnShow :: (Show a, Num a) => SymbolicManip a -> String
rpnShow i =
  let toList (Number x) = [show x]
      toList (Symbol x) = [x]
      toList (BinaryArith op a b) = toList a ++ toList b ++
        [op2str op]
      toList (UnaryArith op a) = toList a ++ [op]
  join :: [a] -> [[a]] -> [a]

```

 v: latest ▾

```
join delim l = concat (intersperse delim l)
in join " " (toList i)
```

RPN 爱好者会发现，跟上面的算法相比，这个算法是多么简洁。尤其是，我们根本不用关心要从哪里加括号，因为 RPN 天生只能沿着一个方向求值。接下来，我们写个函数来实现一些基本的表达式化简。

```
-- file: ch13/num.hs
{- Perform some basic algebraic simplifications on a SymbolicManip. -}
simplify :: (Eq a, Num a) => SymbolicManip a -> SymbolicManip a
simplify (BinaryArith op ia ib) =
    let sa = simplify ia
        sb = simplify ib
    in
        case (op, sa, sb) of
            (Mul, Number 1, b) -> b
            (Mul, a, Number 1) -> a
            (Mul, Number 0, b) -> Number 0
            (Mul, a, Number 0) -> Number 0
            (Div, a, Number 1) -> a
            (Plus, a, Number 0) -> a
            (Plus, Number 0, b) -> b
            (Minus, a, Number 0) -> a
            _ -> BinaryArith op sa sb
simplify (UnaryArith op a) = UnaryArith op (simplify a)
simplify x = x
```

这个函数相当简单。我们很轻易地就能化简某些二元数学运算——例如，用1乘以任何值。我们首先得到操作符两侧操作数被化简之后的版本（在这儿用到了递归）然后再化简结果。对于一元操作符我们能做的不多，所以我们仅仅简化它们作用于的表达式。

从现在开始，我们会增加对计量单位的支持。增加之后我们就能表示“5米”这种数量了。跟之前一样，我们先来定义一个类型：

```
-- file: ch13/num.hs
{- 新数据类型: Units. Units 类型包含一个数字和一个 SymbolicManip, 也就是计量单位。
   计量单位符号可以是 (Symbol "m") 这个样子。 -}
data Units a = Units a (SymbolicManip a)
    deriving (Eq)
```

一个 Units 值包含一个数字和一个符号。符号本身是 SymbolicManip 类型。接下来，将 Units 声明为 Num 实例。

```
-- file: ch13/num.hs
{- 为 Units 实现 Num 实例。我们不知道如何转换任意单位，因此当不同单位的数字相加时，我们报告错误。
   对于乘法，我们生成对应的新单位。 -}
instance (Eq a, Num a) => Num (Units a) where
    (Units xa ua) + (Units xb ub)
        | ua == ub = Units (xa + xb) ua
        | otherwise = error "Mis-matched units in add or subtract"
    (Units xa ua) - (Units xb ub) = (Units xa ua) + (Units (xb * (-1)) ub)
    (Units xa ua) * (Units xb ub) = Units (xa * xb) (ua * ub)
    negate (Units xa ua) = Units (negate xa) ua
    abs (Units xa ua) = Units (abs xa) ua
    signum (Units xa _) = Units (signum xa) (Number 1)
    fromInteger i = Units (fromInteger i) (Number 1)
```

现在，我们应该清楚为什么要用 SymbolicManip 而不是 String 来存储计量单位了。做乘法时，计量单位也会发生改变。例如，5米乘以2米会得到10平方米。我们要求加法运算的单位必须匹配，并用加法实现了减法。我们再来看几个 Units 的类型类实例。

```
-- file: ch13/num.hs
{- Make Units an instance of Fractional -}
instance (Eq a, Fractional a) => Fractional (Units a) where
    (Units xa ua) / (Units xb ub) = Units (xa / xb) (ua / ub)
    recip a = 1 / a
    fromRational r = Units (fromRational r) (Number 1)

{- Floating implementation for Units.

Use some intelligence for angle calculations: support deg and rad
```

 v: latest ▾

```

-}

instance (Eq a, Floating a) => Floating (Units a) where
  pi = (Units pi (Number 1))
  exp _ = error "exp not yet implemented in Units"
  log _ = error "log not yet implemented in Units"
  (Units xa ua) ** (Units xb ub)
    | ub == Number 1 = Units (xa ** xb) (ua ** Number 1)
    | otherwise = error "units for RHS of ** not supported"
  sqrt (Units xa ua) = Units (sqrt xa) (sqrt ua)
  sin (Units xa ua)
    | ua == Symbol "rad" = Units (sin xa) (Number 1)
    | ua == Symbol "deg" = Units (sin (deg2rad xa)) (Number 1)
    | otherwise = error "Units for sin must be deg or rad"
  cos (Units xa ua)
    | ua == Symbol "rad" = Units (cos xa) (Number 1)
    | ua == Symbol "deg" = Units (cos (deg2rad xa)) (Number 1)
    | otherwise = error "Units for cos must be deg or rad"
  tan (Units xa ua)
    | ua == Symbol "rad" = Units (tan xa) (Number 1)
    | ua == Symbol "deg" = Units (tan (deg2rad xa)) (Number 1)
    | otherwise = error "Units for tan must be deg or rad"
  asin (Units xa ua)
    | ua == Number 1 = Units (rad2deg $ asin xa) (Symbol "deg")
    | otherwise = error "Units for asin must be empty"
  acos (Units xa ua)
    | ua == Number 1 = Units (rad2deg $ acos xa) (Symbol "deg")
    | otherwise = error "Units for acos must be empty"
  atan (Units xa ua)
    | ua == Number 1 = Units (rad2deg $ atan xa) (Symbol "deg")
    | otherwise = error "Units for atan must be empty"
  sinh = error "sinh not yet implemented in Units"
  cosh = error "cosh not yet implemented in Units"
  tanh = error "tanh not yet implemented in Units"
  asinh = error "asinh not yet implemented in Units"
  acosh = error "acosh not yet implemented in Units"
  atanh = error "atanh not yet implemented in Units"

```

虽然没有实现所有函数，但大部分都定义了。现在我们来定义几个跟单位打交道的工具函数。

```

-- file: ch13/num.hs
{- A simple function that takes a number and a String and returns an
appropriate Units type to represent the number and its unit of measure -}
units :: (Num z) => z -> String -> Units z
units a b = Units a (Symbol b)

{- Extract the number only out of a Units type -}
dropUnits :: (Num z) => Units z -> z
dropUnits (Units x _) = x

{- Utilities for the Unit implementation -}
deg2rad x = 2 * pi * x / 360
rad2deg x = 360 * x / (2 * pi)

```

首先我们定义了 `units`，使表达式更简洁。`units 5 "m"` 肯定要比 `Units 5 (Symbol "m")` 省事。我们还定义了 `dropUnits`，它把单位去掉只返回 `Num`。最后，我们定义了两个函数，用来在角度和弧度之间转换。接下来，我们给 `Units` 定义 `Show` 实例。

```

-- file: ch13/num.hs
{- Showing units: we show the numeric component, an underscore,
then the prettyShow version of the simplified units -}
instance (Eq a, Show a, Num a) => Show (Units a) where
  show (Units xa ua) = show xa ++ "_" ++ prettyShow (simplify ua)

```

很简单。最后我们定义 `test` 变量用来测试。

```

-- file: ch13/num.hs
test :: (Num a) => a
test = 2 * 5 + 3

```

 v: latest ▾

回头看看这些代码，我们已经完成了既定目标：给 `SymbolicManip` 实现更多实例；我们引入了新类型 `Units`，它包含一个数字和一个单位；我们实现了几个 `show` 函数，以便使用不同的方式来转换 `SymbolicManip` 和 `Units`。

这个例子还给了我们另外一点启发。所有语言——即使那些包含对象和重载的——都有从某种角度看很独特的地方。在 Haskell 里，这个“特殊”的部分很小。我们刚刚开发了一种新的表示法用来表示像数字一样基本的东西，而且很容易就实现了。我们的新类型是一等类型，编译器在编译时就知道使用它哪个函数。Haskell 把代码复用和互换 (interchangeability) 发挥到了极致。写通用代码很容易，而且很方便就能把它们用于多种不同类型的东西上。同样容易的是创建新类型并使它们自动成为系统的一等功能 (first-class features)。

还记得本节开头的 `ghci` 例子吗？我们已经实现了它的全部功能。你可以自己试试，看看它们是怎么工作的。

练习

1. 扩展 `prettyShow` 函数，去掉不必要的括号。

把函数当成数据来用

在命令式语言当中，拼接两个列表很容易。下面的 C 语言结构维护了指向列表头尾的指针：

```
struct list {
    struct node *head, *tail;
};
```

当我们想把列表 B 拼接列表 A 的尾部时，我们将 A 的最后一个节点指向 B 的 `head` 节点，再把 A 的 `tail` 指针指向 B 的 `tail` 节点。

很显然，在 Haskell 里，如果我们想保持“纯”的话，这种方法是有限性的。由于纯数据是不可变的，我们无法原地修改列表。Haskell 的 `(++)` 操作符通过生成一个新列表来拼接列表。

```
-- file: ch13/Append.hs
(++ :: [a] -> [a] -> [a])
(x:xs) ++ ys = x : xs ++ ys
_      ++ ys = ys
```

从代码里可以看出，创建新列表的开销取决于第一个列表的长度。

我们经常需要通过重复拼接列表来创建一个大列表。例如，在生成网页内容时我们可能想生成一个 `String`。每当有新内容添加到网页中时，我们会很自然地想到把它拼接已有 `String` 的末尾。

如果每一次拼接的开销都正比与初始列表的长度，每一次拼接都把初始列表加的更长，那么我们将陷入一个很糟糕的情况：所有拼接的总开销将会正比于最终列表长度的平方。

为了更好地理解，我们来研究一下。`(++)` 操作符是右结合的。

```
ghci> :info (++)
(++ :: [a] -> [a] -> [a]) -- Defined in GHC.Base
infixr 5 ++
```

这意味着 Haskell 在求值表达式 `"a" ++ "b" ++ "c"` 时会从右向左进行，就像加了括号一样：`"a" ++ ("b" ++ "c")`。这对于提高性能非常好处，因为它会让左侧操作数始终保持最短。

当我们重复向列表末尾拼接时，我们破坏了这种结合性。假设我们有一个列表 `"a"` 然后想把 `"b"` 拼接上去，我们把结果存储在一个新列表里。稍后如果我们想把 `"c"` 拼接上去时，这时的左操作数已经变成了 `"ab"`。在这种情况下，每次拼接都让左操作数变得更长。

与此同时，命令式语言的程序员却在偷笑，因为他们重复拼接的开销只取决于操作的次数。他们的性能是线性的，我们的是平方的。

当像重复拼接列表这种常见任务都暴露出如此严重的性能问题时，我们有必要从另一个角度来看问题了。

表达式 `("a"++)` 是一个 **节** (section)，一个部分应用的函数。它的类型是什么呢？

```
ghci> :type ("a"++)
("a"++) :: [Char] -> [Char]
```

 v: latest ▾

由于这是一个函数，我们可以用 `(.)` 操作符把它和另一个节组合起来，例如 `("b"++)`。

```
ghci> :type ("a"++) . ("b"++)
("a"++) . ("b"++) :: [Char] -> [Char]
```

新函数的类型和之前相同。当我们停止组合函数，并向我们创造的函数提供一个 `String` 会发生什么呢？

```
ghci> let f = ("a"++) . ("b"++)
ghci> f []
"ab"
```

我们实现了字符串拼接！我们利用这些部分应用的函数来存储数据，并且只要提供一个空列表就可以把数据提取出来。每一个 `(++)` 和 `(.)` 部分应用都代表了一次拼接，但它们并没有真正完成拼接。

这个方法有两点非常有趣。第一点是部分应用的开销是固定的，这样多次部分应用的开销就是线性的。第二点是当我们提供一个 `[]` 值来从部分应用链中提取最终列表时，求值会从右至左进行。这使得 `(++)` 的左操作数尽可能小，使得所有拼接的总开销是线性而不是平方。

通过使用这种并不太熟悉的数据表示方式，我们避免了一个性能泥潭，并且对“把函数当成数据来用”有了新的认识。顺便说一下，这个技巧并不新鲜，它通常被称为 *差异列表* (difference list)。

还有一点没讲。尽管从理论上差异列表非常吸引人，但如果在实际中把 `(++)`、`(.)` 和部分应用都暴露在外面的话，它并不会非常好用。我们需要把它转成一种更好用的形式。

把差异列表转成库

第一步是用 `newtype` 声明把底层的类型隐藏起来。我们会创建一个 `DList` 类型。类似于普通列表，它是一个参数化类型。

```
-- file: ch13/DList.hs
newtype DList a = DL {
  unDL :: [a] -> [a]
}
```

`unDL` 是我们的析构函数，它把 `DL` 构造器删掉。我们最后导出模块函数时会忽略掉构造函数和析构函数，这样我们的用户就没必要知道 `DList` 类型的实现细节。他们只用我们导出的函数即可。

```
-- file: ch13/DList.hs
append :: DList a -> DList a -> DList a
append xs ys = DL (unDL xs . unDL ys)
```

我们的 `append` 函数看起来可能有点复杂，但其实它仅仅是围绕着 `(.)` 操作符做了一些簿记工作，`(.)` 的用法和我们之前展示的完全一样。生成函数的时候，我们必须首先用 `unDL` 函数把它们从 `DL` 构造器中取出来。然后我们在把得到的结果重新用 `DL` 包装起来，确保它的类型正确。

下面是相同函数的另一种写法，这种方法通过模式识别取出 `xs` 和 `ys`。

```
-- file: ch13/DList.hs
append' :: DList a -> DList a -> DList a
append' (DL xs) (DL ys) = DL (xs . ys)
```

我们需要在 `DList` 类型和普通列表之间来回转换。

```
-- file: ch13/DList.hs
fromList :: [a] -> DList a
fromList xs = DL (xs++)

toList :: DList a -> [a]
toList (DL xs) = xs []
```

 v: latest ▾

再次声明，跟这些函数最原始的版本相比，我们在这里做的只是一些簿记工作。

如果我们想把 `DList` 作为普通列表的替代品，我们还需要提供一些常用的列表操作。

```
-- file: ch13/DList.hs
empty :: DList a
empty = DL id

-- equivalent of the list type's (:) operator
cons :: a -> DList a -> DList a
cons x (DL xs) = DL ((x:) . xs)
infixr `cons`

dfoldr :: (a -> b -> b) -> b -> DList a -> b
dfoldr f z xs = foldr f z (toList xs)
```

尽管 `DList` 使得拼接很廉价，但并不是所有的列表操作都容易实现。列表的 `head` 函数具有常数开销，而对应的 `DList` 实现却需要将整个 `DList` 转为普通列表，因此它比普通列表的实现昂贵得多：它的开销正比于构造 `DList` 所需的拼接次数。

```
-- file: ch13/DList.hs
safeHead :: DList a -> Maybe a
safeHead xs = case toList xs of
    (y:_) -> Just y
    _ -> Nothing
```

为了实现对应的 `map` 函数，我们可以把 `DList` 类型声明为一个 `Functor`（函子）。

```
-- file: ch13/DList.hs
dmap :: (a -> b) -> DList a -> DList b
dmap f = dfoldr go empty
    where go x xs = cons (f x) xs

instance Functor DList where
    fmap = dmap
```

当我们实现了足够多的列表操作时，我们回到源文件顶部增加一个模块头。

```
-- file: ch13/DList.hs
module DList
(
    DList
  , fromList
  , toList
  , empty
  , append
  , cons
  , dfoldr
) where
```

列表、差异列表和幺半群（monoids）

在抽象代数中，有一类简单的抽象结构被称为**幺半群**。许多数学结构都是幺半群，因为成为幺半群的要求非常低。一个结构只要满足两个性质便可称为幺半群：

- 一个满足结合律的二元操作符。我们称之为 $(*)$ ：表达式 $a * (b * c)$ 和 $(a * b) * c$ 结果必须相同。
- 一个单位元素。我们称之为 e ，它必须遵守两条法则： $a * e == a$ 和 $e * a == a$ 。

幺半群并不要求这个二元操作符做什么，它只要求这个二元操作符必须存在。因此很多数学结构都是幺半群。如果我们把加号作为二元操作符，0作为单位元素，那么整数就是一个幺半群。把乘号作为二元操作符，1作为单位元素，整数就形成了另一个幺半群。

Haskell 中幺半群无所不在。`Data.Monoid` 模块定义了 `Monoid` 类型类。

```
-- file: ch13/Monoid.hs
class Monoid a where
    mempty :: a           -- the identity
    mappend :: a -> a -> a -- associative binary operator
```

 v: latest ▾

如果我们把 `(++)` 当做二元操作符，`[]` 当做单位元素，列表就形成了一个幺半群。

```
-- file: ch13/Monoid.hs
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

由于列表和 `DLists` 关系如此紧密，`DList` 类型也必须是一个幺半群。

```
-- file: ch13/DList.hs
instance Monoid (DList a) where
    mempty = empty
    mappend = append
```

在 `ghci` 里试试 `Monoid` 类型类的函数。

```
ghci> "foo" `mappend` "bar"
"foobar"
ghci> toList (fromList [1,2] `mappend` fromList [3,4])
[1,2,3,4]
ghci> mempty `mappend` [1]
[1]
```

Note

尽管从数学的角度看，整数可以以两种不同的方式作为幺半群，但在 `Haskell` 里，我们却不能给 `Int` 写两个不同的 `Monoid` 实例：编译器会报告重复实例错误。

如果我们真的需要在同一个类型上实现多个 `Monoid` 实例，我们可以用 `newtype` 创建不同的类型来达到目的。

```
-- file: ch13/Monoid.hs
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype AInt = A { unA :: Int }
    deriving (Show, Eq, Num)

-- monoid under addition
instance Monoid AInt where
    mempty = 0
    mappend = (+)

newtype MInt = M { unM :: Int }
    deriving (Show, Eq, Num)


-- monoid under multiplication
instance Monoid MInt where
    mempty = 1
    mappend = (*)
```

这样，根据使用类型的不同，我们就能得到不同的行为。

```
ghci> 2 `mappend` 5 :: MInt
M {unM = 10}
ghci> 2 `mappend` 5 :: AInt
A {unA = 7}
```

在这一节（The writer monad and lists）中，我们还会继续讨论差异列表和它的幺半群性质。

Note

跟 `functor` 规则一样，`Haskell` 没法替我们检查幺半群的规则。如果我们定义了一个 `Monoid` 实例，我们可以  `v: latest` 一些 `QuickCheck` 性质来得到一个较高的统计推断，确保代码遵守了幺半群规则。

通用序列

不论是 Haskell 内置的列表，还是我们前面定义的 `DList`，这些数据结构在不同的地方都有自己的性能短板。为此，`Data.Sequence` 模块定义了 `Seq` 容器类型，对于大多数操作，这种类型能都提供良好的效率保证。

为了避免命名冲突，`Data.Sequence` 模块通常以 `qualified` 的方式引入：

```
Prelude> import qualified Data.Sequence as Seq
Prelude Seq>
```

`empty` 函数用于创建一个空 `Seq`，`singleton` 用于创建只包含单个元素的 `Seq`：

```
Prelude Seq> Seq.empty
fromList []

Prelude Seq> Seq.singleton 1
fromList [1]
```

还可以使用 `fromList` 函数，通过列表创建出相应的 `Seq`：

```
Prelude Seq> let a = Seq.fromList [1, 2, 3]

Prelude Seq> a
fromList [1,2,3]
```

`Data.Sequence` 模块还提供了几种操作符形式的构造函数。但是，在使用 `qualified` 形式载入模块的情况下调用它们会非常难看：

[译注：操作符形式指的是那种放在两个操作对象之间的函数，比如 $2 * 2$ 中的 $*$ 函数。]

```
Prelude Seq> 1 Seq.<| Seq.singleton 2
fromList [1,2]
```

可以通过直接载入这几个函数来改善可读性：

```
Prelude Seq> import Data.Sequence((><), (<|), (|>))
```

现在好多了：

```
Prelude Seq Data.Sequence> Seq.singleton 1 |> 2
fromList [1,2]
```

一个帮助记忆 `(<|)` 和 `(|>)` 函数的方法是，函数的『箭头』总是指向被添加的元素：`(<|)` 函数要添加的元素在左边，而 `(|>)` 函数要添加的元素在右边：

```
Prelude Seq Data.Sequence> 1 <| Seq.singleton 2
fromList [1,2]

Prelude Seq Data.Sequence> Seq.singleton 1 |> 2
fromList [1,2]
```

不管是从左边添加元素，还是从右边添加元素，添加操作都可以在常数时间内完成。对两个 `Seq` 进行追加（append）操作同样非常廉价，复杂度等同于两个 `Seq` 中较短的那个 `Seq` 的长度的对数。

追加操作由 `><` 函数完成：

```
Prelude Seq Data.Sequence> let left = Seq.fromList [1, 3, 3]

Prelude Seq Data.Sequence> let right = Seq.fromList [7, 1]

Prelude Seq Data.Sequence> left >< right
fromList [1,3,3,7,1]
```

 v: latest ▾

反过来，如果我们想将 `Seq` 转换回列表，那么就需要 `Data.Foldable` 模块的帮助：

```
Prelude Seq Data.Sequence> import qualified Data.Foldable as Foldable
Prelude Seq Data.Sequence Foldable>
```

这个模块定义了一个类型，`Foldable`，而 `Seq` 实现了这个类型：

```
Prelude Seq Data.Sequence Foldable> Foldable.toList (Seq.fromList [1, 2, 3])
[1, 2, 3]
```

`Data.Foldable` 中的 `fold` 函数可以用于对 `Seq` 进行 `fold` 操作：

```
Prelude Seq Data.Sequence Foldable> Foldable.foldl' (+) 0 (Seq.fromList [1, 2, 3])
6
```

`Data.Sequence` 模块还提供了大量有用的函数，这些函数都和 Haskell 列表的函数类似。模块的文档也非常齐全，还提供了函数的时间复杂度信息。

最后的疑问是，既然 `Seq` 的效率这么好，那为什么它不是 Haskell 默认的序列类型呢？答案是，列表类型更简单，消耗更低，对于大多数应用程序来说，列表已经足够满足需求了。除此之外，列表可以很好地处理惰性环境，而 `Seq` 在这方面做得还不够好。

讨论

0条评论

Real World Haskell 中文版


1 登录

♥ 推荐

🐦 推文

f 分享

最新发布



开始讨论...

通过以下方式登录

或注册一个 DISQUS 帐号 ?

姓名

来做第一个留言的人吧！

在 REAL WORLD HASKLL 中文版 上还有


Real World Haskell 中文版 — Real World Haskell 中文版

4条评论 • 6年前

 Jojo — 更新好慢呀

Real World Haskell 中文版

2条评论 • 6年前

 yadsun — 校正：ghc是生成快速本底代码的优化编译器。中 本底 - ->本地


第三章：Defining Types, Streamlining Functions¶

4条评论 • 4年前

 Wengel An —

第十章：代码案例学习：解析二进制数据格式 — Real World Haskell 中文版

1条评论 • 4年前

 Y — 此处少个"+": instance Show Greymap where show (Greymap w h m _) = "Greymap " ++ show w ++ "x" ++ show h ++ " " ++ show m

📧 订阅

🔒 在您的网站上使用 Disqus添加 Disqus添加

🔒 Disqus 隐私政策隐私政策隐私