Following the Grammar with Haskell

Gary T. Leavens

CS-TR-13-01
January 2013

Keywords: Recursion, programming recursive functions, recursion pattern, inductive definition, BNF grammar, follow the grammar, functional programming, list recursion, programming languages, concrete syntax, abstract syntax, helping functions, Haskell.

2001 CR Categories: D.1.1 [*Programming Techniques*] Applicative (Functional) Programming — design, theory; D.2.4 [*Software Engineering*] Coding Tools and Techniques — design, theory; D.3.1 [*Programming Languages*] Formal Definitions and Theory — syntax; D.3.3 [*Programming Languages*] Language Constructs and Features — recursion;

Computer Science
University of Central Florida
4000 Central Florida Blvd.
Orlando, FL 32816-2362 USA

Following the Grammar with Haskell

Gary T. Leavens
437D Harris Center (Bldg. 116)
Computer Science, University of Central Florida
4000 Central Florida Blvd., Orlando, FL 32816-2362 USA
leavens@eecs.ucf.edu

January 23, 2013

Abstract

This document[1] explains what it means to "follow the grammar" when writing recursive programs, for several kinds of different grammars. It is intended to be used in classes that teach functional programming using Haskell, especially those used for teaching principles of programming languages. In such courses traversal over an abstract syntax tree defined by a grammar are fundamental, since they are the technique used to write compilers and interpreters.

1 Introduction

An important skill in functional programming is being able to write a program whose structure mimics the structure of a context-free grammar. This is important for working with programming languages [4, 9], as they are described by such grammars. Therefore, the proficient programmer's motto is "follow the grammar." This document attempts to explain what "following the grammar" means, by explaining a series of graduated examples.

The "follow the grammar" idea was the key insight that allowed computer scientists to build compilers for complex languages, such as Algol 60. It is fundamental to modern syntax-directed compilation [1]. Indeed the idea of syntax-directed compilation is another expression of the idea "follow the grammar." It finds clear expression in the structure of interpreters used in the Friedman, Wand and Haynes book *Essentials of Programming Languages* [4].

The idea of following the grammar is not new and not restricted to programming language work. An early expression of the idea is Michael Jackson's method [6], which advocated designing a computation around the structure of the data in a program. Object-oriented design [2, 10] essentially embodies this idea, as in object-oriented design the program is organized around the data.

Thus the "follow the grammar" idea has both a long history and wide applicability.

1.1 Grammar Background

To explain the concept of following a grammar precisely, we need a bit of background on grammars.

A context-free grammar consists of several nonterminals (names for sets), each of which is defined by one or more alternative productions. In a context-free grammar, each production may contain recursive uses of nonterminals. For example, the context-free grammar below has four non-terminals, ⟨Stmt⟩, ⟨Var⟩, `Exp`, and ⟨StmtList⟩.

⟨Stmt⟩ ::= `Skip` | `Assign` ⟨Var⟩ ⟨Exp⟩ | `Compound` ⟨StmtList⟩
⟨StmtList⟩ ::= `[]` | ⟨Stmt⟩ `:` ⟨StmtList⟩
⟨Var⟩ ::= ⟨String⟩
⟨Exp⟩ ::= ⟨Integer⟩

[1]This paper is adapted from an Oz verion, [8].

The nonterminal ⟨Stmt⟩ has three alternatives (separated by the vertical bars). The production for ⟨Stmt⟩ has a recursive call to ⟨StmtList⟩, which, in turn, recursively calls ⟨Stmt⟩.

Context-free grammars correspond very closely to Haskell data declarations. For example, the data declaration below corresponds directly to the grammar for ⟨Stmt⟩ above.

```
data Stmt = Skip | Assign Var Exp | Compound StmtList
type StmtList = [Stmt]
type Var = String
type Exp = Integer
```

Notice how the built-in Haskell types for lists, Strings, and Integers are used. More importantly, however, notice that `Skip`, `Assign`, and `Compound` are constructors of the datatype `Stmt`.

1.2 Definition of Following the Grammar

Following the grammar means making a set of functions whose structure mimics that of the grammar in a certain way. This is explained in the following definition.

Definition 1.1 *Consider a context-free grammar, G and a set of functions Prog. We say that Prog* follows the grammar *G if:*

1. *For every nonterminal, ⟨X⟩, in G, there is a function, fX, in Prog that takes an argument from the set described by the nonterminal ⟨X⟩.*
2. *If a nonterminal ⟨X⟩ has alternatives, then the corresponding function fX decides between the alternatives offered in ⟨X⟩'s grammatical productions, and there is (at least) one case in the definition of fX for each alternative production for ⟨X⟩.*
3. *If the nonterminal ⟨X⟩ has an alternative whose production contains a nonterminal ⟨Y⟩ that is defined in the grammar G, then:*
   (a) *the corresponding case in fX has a call to fY, where fY is the function in Prog that handles data described by the nonterminal ⟨Y⟩, and*
   (b) *each call from fX to fY passes to fY a part of the data fX received as an argument, including at least that part described by ⟨Y⟩.*

Since this definition does not prescribe the details of the set of functions, we often say that *Prog has an outline that follows a grammar G* if *Prog* follows *G*.

We give many examples in the sections that follow.

Following the grammar thus organizes the program around the structure of the data, in much the same way as one would organize the methods of an object-oriented program in classes, so that each method deals with data of the class in which it resides. In a functional program, each kind of data (i.e., each nonterminal) has its own function that only handles that kind of data. This division of responsibility makes it easy to understand and modify the program.

A closer object-oriented analogy is to the structure of the methods in the Visitor pattern [5], for a particular visitor. The story there is the same: each kind of data has a method (the visitor) that handles that particular kind of data.

1.3 Overview

In the following we will consider several different examples of grammars and functions that follow them. Our exploration is gradual, and based on increasing complexity in the structures of the grammars we treat. In Section 2, we show how to follow a grammar that only has alternatives. In Section 3, we do the same for grammars that only have recursion. In Section 4 we add the complication of multiple nonterminals. Finally, in Section 5 we treat a series of grammars that combine these features.

2 Only Alternatives, No Recursion

The simplest kind of grammar has no recursion, but just has alternatives.

2.1 Temperature Grammar

For example, consider the following grammar for temperatures. In this grammar, all of the alternatives are base cases.

```
⟨Temperature⟩ ::=
      Cold
    | Warm
    | Hot
```

In Haskell, this corresponds to the following data definition.

```
data Temperature = Cold | Warm | Hot
```

Notice that the nonterminal ⟨Temperature⟩ is translated into the type named `Temperature` in Haskell, and each of the alternatives in this grammar is translated into a constructor (`Cold`, `Warm`, and `Hot`) of the Haskell data type.

2.1.1 Example

A function that takes a Temperature as an argument will have the outline typified by the following example.

```
selectOuterwear :: Temperature -> String
selectOuterwear Cold = "down jacket"
selectOuterwear Warm = "wind breaker"
selectOuterwear Hot = "none"
```

Notice that there are three alternatives in the grammar, and so there are three cases in the function, each of which corresponds to a condition tested in the body of the function. There is no recursion in the Temperature grammar, so there is no recursion in the function.

2.1.2 isFreezing Exercise

Which of the following has a correct outline for a function

```
isFreezing :: Temperature -> Bool
```

that follows the grammar for Temperature?

1.
   ```
   isFreezing (Cold:_) = True
   isFreezing [] = False
   isFreezing (_:ts) = isFreezing ts
   ```

2.
   ```
   isFreezing [] = False
   isFreezing (t:ts) = (t == Cold) || (isFreezing ts)
   ```

3.
   ```
   isFreezing Cold = True
   isFreezing Hot = False
   isFreezing _ = False
   ```

4.
   ```
   isFreezing temp =
       case temp of
         Cold -> isFreezing temp
         Hot -> not (isFreezing temp)
         _ -> False
   ```

Answer: 3. Note that 4 has recursion, which does not follow the grammar in this case, since the grammar is not recursive.

2.2 Color Grammar Exercises

Consider another example with simple alternatives and no recursion

⟨Color⟩ ::= `Red` | `Yellow` | `Green` | `Blue`

and which corresponds to the following Haskell data definition.

```
data Color = Red | Yellow | Green | Blue  deriving (Show)
```

Write the function:

```
equalColor :: Color -> Color -> Bool
```

that takes two Colors and returns **`True`** just when they are the same. For example `equalColor Red Red` would be **`True`**. (Note that you cannot use == in your code, since there is no instance of `Eq` for this type available.)

3 Only Recursion, No Alternatives

Another kind of grammar is one that just has recursion, but no alternatives.

3.1 Infinite Sequence Grammar

The following is an example of a grammar with no alternatives, which is a grammar of infinite Integer sequences.

⟨ISeq⟩ ::= ⟨Integer⟩ `:#` ⟨ISeq⟩

The above corresponds to the following Haskell data definition, where `:#` is used as an infix constructor

```
data ISeq = Integer :# ISeq
```

In Haskell one can easily create such infinite sequences using lazy evaluation, as in the following two examples

```
ones :: ISeq
ones = 1 :# ones
nats :: ISeq
nats = let from n = n :# (from (n+1))
       in (from 0)
```

3.1.1 Example

A function

```
iSeqMap :: (Integer -> Integer) -> ISeq -> ISeq
```

that follows the above grammar is the following.

```
iSeqMap f (n:#ns) = (f n) :# (iSeqMap f ns)
```

Following the grammar in this example means that the function does something with the number `n` in the head of the pattern and recurses on the tail of the pattern `ns`, which is an ISeq. In this example, there is no base case or stopping condition, because the grammar has no alternatives to allow one to stop, which is why it is so helpful that functions in Haskell are lazy by default.

Although this example does not have a stopping condition, other functions that work on this grammar might allow stopping when some condition holds, as in the next example.

3.1.2 AnyNegative Exercise

Which of the following has a correct outline for a function

```
anyNegative :: ISeq -> Bool
```

that follows the grammar for ISeq?

1.
   ```
   anyNegative [] = False
   anyNegative (n:ns) = (n<0) || (anyNegative ns)
   ```

2.
   ```
   anyNegative iseq =
     case iseq of
       (n :# ns) -> (n<0) || (anyNegative ns)
   ```

3.
   ```
   anyNegative iseq = (n<0) || (anyNegative ns)
   ```

4.
   ```
   anyNegative Cold = True
   anyNegative _ = False
   ```

5.
   ```
   anyNegative (n:#ns) = (n<0) || (anyNegative ns)
   ```

Answer: 2 and 5. Note that 3 would be okay if n and ns were defined in its definition, but as it is, it does not type check.

3.1.3 Filter Infinite Sequence Exercise

Write a lazy function

```
filterISeq :: (Integer -> Bool) -> ISeq -> ISeq
```

that takes a predicate, `pred`, an infinite sequence, `iseq`, and returns an infinite sequence of all elements in `iseq` for which `pred` returns `True` when applied to the element. Elements retained are left in their original order. For example:

```
filterISeq (\n -> n `mod` 2 == 0) nats
```

would return an `ISeq` that is `0:#2:#4:#6:#8:#`..., since that ISeq is only the even elements of `nats`.

4 Multiple Nonterminals

When the grammar has multiple nonterminals, there should be a function for each nonterminal in the grammar, and the recursive calls between these functions should correspond to the recursive uses of nonterminals in the grammar. That is, when a production for a nonterminal, ⟨X⟩, uses another nonterminal, ⟨Y⟩, there should be a call from the function for ⟨X⟩ to the function for ⟨Y⟩ that passes an instance of ⟨Y⟩ as an argument.

4.1 Rectangle Grammar

Consider the following Haskell data definitions for Rectangles and Points. (We dispense with the standard form of the grammar from now on, since the Haskell data definitions correspond to them very closely.)

```
data Rectangle = Rect {ul :: Point, lr :: Point} deriving (Show)
data Point = Integer :@ Integer  deriving (Show)
```

4.1.1 Example

To follow this grammar when writing a program like

```
moveUp :: Rectangle -> Integer -> Rectangle
```

one would structure the code into two functions, one for each of the two types (i.e., nonterminals), as shown in Figure 1. Since the type (i.e., production) for `Rectangle` uses the type `Point` twice, the function `moveUp` calls the `moveUpPoint` function twice, once on each of the points in the Rectangle. Note that the arguments to these functions are parts of the Integer described by the corresponding nonterminals, and are extracted from the number by the pattern match in the body of `MoveUp`

```
moveUp (Rect {ul = pul, lr = plr}) i =
       (Rect {ul = moveUpPoint pul i, lr = moveUpPoint plr i})
moveUpPoint (x :@ y) i = x :@ (y+i)
```

Figure 1: The two functions that move Rectangles up.

4.1.2 DoubleRect Exercise

Which of the following is a correct outline of a function

```
doubleRect :: Rectangle -> Rectangle
```

that follows the grammar for Rectangles?

1.
   ```
   doubleRect (Rect {ul = (xul:@yul), lr = (xlr:@ylr)}) =
                Rect {ul = 2*xul:@2*yul, lr = 2*xlr:@2*ylr}
   ```

2.
   ```
   doubleRect shape =
        case shape of
          (Rect {ul = pul, lr = plr}) -> Rect {ul = doubleRect pul,
                                               lr = doubleRect plr}
          (x:@y) -> (2*x) :@ (2*y)
   ```

3.
   ```
   doubleRect (ul:lr:[]) = doublePoint ul : doublePoint lr : []
   doublePoint (x :@ y) = (2*x) :@ (2*y)
   ```

4.
   ```
   doubleRect (Rect {ul = pul, lr = plr}) =
             (Rect {ul = doublePoint pul, lr = doublePoint plr})
   doublePoint (x :@ y) = (2*x) :@ (2*y)
   ```

Answer: 4. Note that 2 does not type check in Haskell! Also 1 while it type checks, does not follow the rule about having one function per type, since there is no function for Points given.

4.1.3 ShrinkRect Exercise

Write a function,

```
shrinkRect :: Rectangle -> Integer -> Rectangle
```

that takes a Rectangle, `rect`, and a Number `factor` (which is greater than or equal to 1) and shrinks the rectangle's sides by `factor`, leaving its upper left point in place.

4.2 Multiple Nonterminal Exercise

Suppose you have a grammar with 10 nonterminals, how many functions would be contained in an outline that followed that grammar?

5 Combination of Different Grammar Types

Most interesting examples of grammars involve a combination of the three types discussed above. That is, there are alternatives, and recursions, and multiple nonterminals.

The discussion in this section starts with some examples that only involve two of these features. The grammar for flat lists and the grammar for "window layouts" both involve only one nonterminal. The grammar for boolean expressions that follows has multiple nonterminals, but does not have mutual recursion.

Following these simpler combination examples is a discussion of an example of a "statement and expression" grammar that involves all three features at once.

5.1 Flat Lists

The grammar for flat lists is simpler than other combinations, as it has alternatives and recursion, but only one nonterminal. A grammar for flat lists of elements of type t is as follows.

$\langle$[t]$\rangle$ ::= `[]` | $\langle$t$\rangle$ **:** $\langle$[t]$\rangle$

where $\langle$t$\rangle$ is the nonterminal that generates elements of type t. This corresponds to the following Haskell data definition for its built-in flat lists.

```
data [] t = [] | t : [t]
```

The above grammar means we are not interested in the structure of the $\langle$t$\rangle$ elements of the lists; that is the sense in which the list is "flat."

(For flat lists, "following the grammar" is essentially equivalent to following the first and fourth commandments in Friedman and Felleisen's book *The Little Schemer* [3].)

5.1.1 Example of Recursion over Flat Lists

The **`map'`** function in Figure 2, whose type is shown below,

```
map' ::  (t -> s) -> [t] -> [s]
```

is a paradigmatic example of following the grammar for flat lists.

```
map' f [] = []
map' f (x:xs) = (f x) : (map' f xs)
```

Figure 2: The `map'` function for flat lists.

5.1.2 ExtractNames Exercise

Which, if any, of the following is a correct outline for a function

```
extractNames :: [Person] -> [String]
```

where

```
data Person = Human {name :: String, address :: String}
                deriving (Eq, Show)
```

that follows the grammar for flat lists? List all that have a correct outline for recursion over flat lists. (Note: we are mainly asking whether these have the right outline, but having the wrong outline will cause them not to work as they should.) Note that the function

```
name :: Person -> String
```

is defined due to the declaration of the field `name` in the definition of the type `Person`.

1.
   ```
   extractNames (p:ps) = (name p) : (extractNames ps)
   ```

2.
   ```
   extractNames [] = []
   extractNames (p:ps) = (name p) : (extractNames ps)
   ```

3.
   ```
   extractNames lst = if lst == []
                      then []
                      else (name (head lst)) : extractNames (tail lst)
   ```

4.
   ```
   extractNames lst = case lst of
                        (p:ps) -> (name p) : ps
                        [] -> []
   ```

5.
   ```
   extractNames [] = []
   extractNames (Human {name = n, address = _}) -> [n]
   ```

6.
   ```
   extractNames = map name
   ```

Answer: 2, 3, 6. In 6, the `map` function is following the grammar on behalf of `extractNames`.

5.1.3 DeleteListing Exercise

Which, if any, of the following is a correct outline for a function

```
deleteListing :: String -> [Person] -> [Person]
```

that follows the grammar for flat lists? List all that have a correct outline for recursion over flat lists.

(Recall that the function `name` is defined in the definition of the `Person` type.)

1.
   ```
   deleteListing _ [] = []
   deleteListing n (p:ps) = if name p == n
                            then (deleteListing n ps)
                            else p:(deleteListing n ps)
   ```

2.
   ```
   deleteListing _ [] = []
   deleteListing n (p:ps)
        | name p == n = (deleteListing n ps)
        | otherwise   = p:(deleteListing n ps)
   ```

3.
```
deleteListing n ps = filter ((/= n) . name) ps
```

4.
```
deleteListing n ps =
  case ps of
     [] -> []
     (per:pers) -> append if n = name per then [] else [per]
                          (deleteListing n pers)
```

5.
```
deleteListing n ps =
  case ps of
    (per:pers) | n == (name per) -> (deleteListing n pers)
    _ -> per:(deleteListing n pers)
```

6.
```
deleteListing n (per:pers) = if name per == n
                             then (deleteListing n pers)
                             else per:(deleteListing n pers)
```

7.
```
deleteListing n ps = [p | p <- ps, name p == n]
```

Answer: 1, 2, 3, 4, 7. Note that in 1 the function `filter` is following the grammar, and in 7 the language is following the grammar in the semantics of the list comprehension.

5.2 Window Layouts

For purposes of this paper, a "window layout" is an instance of the data type below. This type is designed to describe a data structure for placing windows on a computer screen, similar in some ways to layout managers in various graphical user interface libraries.

The window layout grammar has only one nonterminal, but more alternatives than the grammar for flat lists. Thus functions that follow its grammar have more recursive calls than functions that follow the grammar for flat lists.

```
module WindowLayout where
data WindowLayout = Window {wname :: String, width :: Int, height :: Int}
                  | Horizontal [WindowLayout]
                  | Vertical [WindowLayout]
                  deriving (Show, Eq)
```

5.2.1 Example

An example function that follows the above grammar is shown in Figure 3. This example is coded using Haskell's built-in `map` function (a version of which was shown in Figure 2 on page 7). Doing that avoids having to write out a separate function for recursing over a list of WindowLayouts. Writing out a separate function (or two different functions, in general) to recurse over the two lists in the grammar would still follow the grammar. These helping function(s) would, of course, follow the grammar for flat lists.

Note, however, that it would *not* be following the grammar to write something like `badDoubleSize` shown in Figure 4 on the next page. The problem with `badDoubleSize` is that the function works on both lists and window layouts. Writing code like this leads to confusion, especially as grammars get more complicated. Sometimes the function also plays a different role depending on what nonterminal it serves, and so the same argument, if it occurs as part of two different nonterminals can make it very difficult to write the function correctly. While these are not problems in this simple example, they will cause difficulties in larger and more complex examples.

In the calls to `map` in Figure 3, the function argument is `doubleSize` itself. This is equivalent to (but slightly more efficient than) passing the function `(\ wl -> doubleSize wl)`. However, in other problems using such an anonymous function (or a named helping function) may be needed, because the

```
module DoubleSize where
import WindowLayout

doubleSize :: WindowLayout -> WindowLayout
doubleSize (Window {wname = n, width = w, height = h}) =
            Window {wname = n, width = 2*w, height = 2*h}
doubleSize (Horizontal wls) = Horizontal (map doubleSize wls)
doubleSize (Vertical wls) = Vertical (map doubleSize wls)
```

Figure 3: The function `doubleSize`, which follows the grammar for window layouts.

```
badDoubleSize (Window {wname = n, width = w, height = h}) =
            Window {wname = n, width = 2*w, height = 2*h}
badDoubleSize (Horizontal wls) = Horizontal (badDoubleSize wls)
badDoubleSize (Vertical wls) = Vertical (badDoubleSize wls)
-- the following cases make no sense in this function!
badDoubleSize [] = []
badDoubleSize (w:ws) then (badDoubleSize w):(badDoubleSize ws)
```

Figure 4: The function `badDoubleSize`, which does not follow the grammar for window layouts, because it incorrectly combines two recursions in one function. Indeed, this does not even type check in Haskell!

```
module AddToSize where
import WindowLayout

addToSize n wl =
   let addWithFixedN wl = addToSize n wl
   in case wl of
     (Window {wname = nm, width = w, height = h}) ->
        (Window {wname = nm, width = n+w, height = n+h})
     (Horizontal wls) -> Horizontal (map addWithFixedN wls)
     (Vertical wls) -> Vertical (map addWithFixedN wls)
```

Figure 5: The function `AddToSize` that illustrates the use of a helper function when calling `map`.

function argument to `map` must be a function that only needs one argument to do its work. For an example, consider writing a function `addToSize`, such that `{addToSize n wl}` returns a window layout that is exactly like `wl`, except that each window has `n` added to both its width and height. In solving such a problem it is convenient to use a function that takes a window layout, `wl` and returns the value of `{addToSize n wl}` for a fixed `n`. This is shown in Figure 5.

In Haskell, one can partially apply `addToSize`, so that one can simplify the second line in Figure 5 to:

```
let addWithFixedN wl = addToSize n wl
```

An even shorter version would be to let that line be just the following (by cancelling the `wl` from both sides of the above):

```
let addWithFixedN = addToSize n
```

Also in Haskell, one can avoid giving a name to this function and just use the expression `(addToSize n)` wherever `addWithFixedN` is used in Figure 5.

5.2.2 MultSize Exercise

Which, if any, of the following is a correct outline for a function

```
multSize :: Int -> WindowLayout -> WindowLayout
```

that follows the grammar for window layouts? List all that have a correct outline for recursion over window layouts.

1.
```
multSize m (Window {wname = nm, width = w, height = h}) =
            Window {wname = nm, width = m*w, height = m*h}
multSize m (Horizontal wls) = Horizontal (map (multSize m) wls)
multSize m (Vertical wls) = Vertical (map (multSize m) wls)
```

2.
```
multSize m wl =
  case wl of
    (Window {wname = nm, width = w, height = h}) ->
     Window {wname = nm, width = m*w, height = m*h}
    (Horizontal wls) -> Horizontal (multSize m wls)
    (Vertical wls) -> Vertical (multSize m wls)
    [] -> []
    (w:ws) -> (multSize m w):(multSize m ws)
```

3.
```
multSize m (Window {wname = nm, width = w, height = h}) =
      Window {wname = nm, width = m*w, height = m*h}
multSize m (Horizontal wls) = Horizontal (multSizeList m wls)
multSize m (Vertical wls) = Vertical (multSizeList m wls)

multSizeList m [] = []
multSizeList m (w:ws) = (multSize m w):(multSizeList m ws)
```

4.
```
multSize m wl =
   case wl of
     (Window {wname = nm, width = w, height = h}) ->
      Window {wname = nm, width = m*w, height = m*h}
     (Horizontal wls) -> Horizontal (multSizeList m wls)
     (Vertical wls) -> Vertical (multSizeList m wls)
    where
      multSizeList m [] = []
      multSizeList m (w:ws) = (multSize m w):(multSizeList m ws)
```

5.
```
multSize m wl =
   case wl of
     (Window {wname = nm, width = w, height = h}) ->
      Window {wname = nm, width = m*w, height = m*h}
     (Horizontal wls) -> Horizontal wls'
     (Vertical wls) -> Vertical wls'
    where
      wls' = (map (multSize m) wls)
```

6.
```
multSize m wl =
   case wl of
     (Window {wname = nm, width = w, height = h}) ->
      Window {wname = nm, width = m*w, height = m*h}
     (Horizontal wls) -> Horizontal (multSize m wls)
     (Vertical wls) -> Vertical (multSize m wls)
     [] -> []
     (w:[]) -> (multSize m w):[]
     (w:ws) -> (multSize m w):(multSize m ws)
```

Answer: 1, 3, 4, 5. It's important to note that the others do *not* follow the grammar. The problem is that they combine two recursions in one function, which leads to confusion.

5.2.3 TotalWidth Exercise

Write a function,

```
totalWidth :: WindowLayout -> Int
```

that takes a WindowLayout, `wl`, and returns the total width of the layout. The width is defined by cases. The width of a WindowLayout of the form

`Window { wname =` nm`, width =` w`, height =` h`}`

is w. The width of a ⟨WindowLayout⟩ of the form

`Horizontal [`wl_1`, ...,` wl_m`]`

is the sum of the widths of wl_1 through wl_m (inclusive). The width of a ⟨WindowLayout⟩ of the form

`Vertical [`wl_1`, ...,` wl_m`]`

is the maximum of the widths of wl_1 through wl_m (inclusive). If the list is empty, the width should be taken as 0.

Several example tests that show how `totalWidth` works are shown in Figure 6. (These use the `eqTest` function from the course library's `Testing` module. Note that the testing module assumes that `totalWidth` is implemented in a module called `TotalWidth`, which is imported.)

```
-- $Id: TotalWidthTests.hs,v 1.2 2013/01/24 19:29:19 leavens Exp leavens $
module TotalWidthTests where
import Testing
import WindowLayout
import TotalWidth

main = dotests "TotalWidthTests $Revision: 1.2 $" tests

tests :: [TestCase Int]
tests =
 [(eqTest (totalWidth (Window {wname = "olympics", width = 50, height = 33}))
          "==" 50)
 ,(eqTest (totalWidth (Horizontal [])) "==" 0)
 ,(eqTest (totalWidth (Vertical [])) "==" 0)
 ,(eqTest (totalWidth
            (Horizontal [(Window {wname = "olympics", width = 80, height = 33})
                        ,(Window {wname = "news", width = 20, height = 10})]))
   "==" 100)
 ,(eqTest (totalWidth
           (Vertical [(Window {wname = "olympics", width = 80, height = 33})
                     ,(Window {wname = "news", width = 20, height = 10})]))
   "==" 80)
 ,(eqTest (totalWidth
           (Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                    ,(Window {wname = "olympics", width = 80, height = 33})
                    ,(Window {wname = "news", width = 20, height = 10})]))
   "==" 80)
 ,(eqTest (totalWidth
         (Horizontal
            [(Vertical [(Window {wname = "Tempest", width = 200, height = 100})
                      ,(Window {wname = "Othello", width = 200, height = 77})
                      ,(Window {wname = "Hamlet", width = 1000, height = 600})])
            ,(Horizontal [(Window {wname = "baseball", width = 50, height = 40})
                        ,(Window {wname = "track", width = 100, height = 60})
                        ,(Window {wname = "golf", width = 70, height = 30})])
            ,(Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                      ,(Window {wname = "olympics", width = 80, height = 33})
                      ,(Window {wname = "news", width = 20, height = 10})])
            ]))
        "==" 1300)
 ]
```

Figure 6: Tests for the `totalWidth` exercise.

Feel free to use Haskell's `map` and `max` functions (as well as `foldl` or `foldr` if you have seen those).

```
module SalesData where

data SalesData =
    Store { address :: String, amounts :: [Int] }
  | Group { gname :: String, members :: [SalesData] }
    deriving (Show, Eq)
```

Figure 7: Data declaration for the type `SalesData`.

5.2.4 Design Your own WindowLayout Problem Exercise

Design another problem for the type `WindowLayout`. Give an English explanation, the function's type, and some examples. Then solve your problem, first on paper, then on the computer.

For example, you might do something like computing the total area of the window layout, or a list of all the names of the windows.

5.3 Sales Data

The grammar for `SalesData` is shown in Figure 7. It has a single nonterminal and like the window layouts grammar, also uses lists. The grammar for ⟨String⟩ is the same as for Haskell, which is essentially `[Char]`.

5.3.1 NormalizeSalesData Example

The grammar for sales data is interesting in that its fields contain several different kinds of lists. Since the different lists play different roles in the sales data grammar, it is thus especially important to follow the grammar in the sense that only data of type `SalesData` should be passed to functions that work on that type, and no lists should be passed to such functions.

The reason this is important is illustrated by the following example. Suppose we want to write a function

```
normalizeSalesData :: SalesData -> SalesData
```

that takes a sales data argument, `sd`, and returns a result that is just like `sd`, except that in each store record, each address string is put into ALL CAPITAL LETTERS and the amounts list is trimmed to be just the first 5 elements of the argument's list, and in each group record, the name field is put into all capital letters, and each of the members is also normalized.

Figure 8 on the following page gives some examples of how this program is supposed to work. These use the `Test` procedure from the course library.

We suggest that you try to write out a solution for this problem before looking at our solution.

Correct code for `NormalizeSalesData` is shown in Figure 9 on page 16. This follows the grammar in that it only passes sales data to the function `NormalizeSalesData`. Note that lists of characters and numbers are handled by helper functions `capitalize`, `map`, and the built-in function `take`. Can you see why this code works correctly?

Now consider the `NormalizeSalesDataBad`, shown in Figure 10 on page 16, which attempts to solve the same problem. This does not follow the grammar, but resembles programs that some students try to write because they don't want to use helper functions. Although it still correctly uses `take`, all other lists are handled by making all recursive calls to itself. These recursive calls do not follow the grammar, because they pass lists to a function that is supposed to receive Sales Data records. Does the code in Figure 10 on page 16 even type check? No, it does not, and there are several type errors that Haskell will point out to try to steer you back to following the grammar.

Indeed the code in Figure 10 on page 16 would not be correct, even if Haskell would execute it! Can you see why? First, remember that in Haskell, Strings are lists of Chars. The last two cases in the function definition are attempting to deal with lists of characters separately from lists of sales data records, but even if

```
module NormalizeSalesDataTests where
import Testing
import SalesData
import NormalizeSalesData
main = dotests "$Revision: 1.1 $" tests
tests =
 [(eqTest (normalizeSalesData
           (Store {address = "1005 Alafaya Trail Rd."
                  ,amounts = [101, 102, 103, 104, 105, 106]}))
   "==" (Store {address = "1005 ALAFAYA TRAIL RD."
               ,amounts = [101, 102, 103, 104, 105]}))
 ,(eqTest (normalizeSalesData
           (Store {address = "227 International Dr."
                  ,amounts = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}))
   "==" (Store {address = "227 INTERNATIONAL DR."
               ,amounts = [1, 2, 3, 4, 5]}))
 ,(eqTest (normalizeSalesData
           (Group {gname = "Target"
                  ,members = [(Store {address = "227 International Dr."
                                     ,amounts=[1,2,3,4,5,6,7,8,9,10]})
                             ,(Store {address = "1005 Alafaya Trail Rd."
                                     ,amounts=[101,102,103,104,105,106]})]}))
   "==" (Group {gname = "TARGET"
               ,members = [(Store {address = "227 INTERNATIONAL DR."
                                  ,amounts=[1,2,3,4,5]})
                          ,(Store {address = "1005 ALAFAYA TRAIL RD."
                                  ,amounts=[101,102,103,104,105]})]}))
 ,(eqTest (normalizeSalesData
           (Group {gname = "Target"
                  ,members =
                      [(Group {gname = "Target Florida"
                              ,members = [(Store {address="227 International Dr."
                                                 ,amounts=[1,2,3,4,5,6,7,8,9,10]})
                                         ,(Store {address="1005 Alafaya Trail Rd."
                                                 ,amounts=[101,102,103,104,105,106]})]})
                      ,(Group {gname = "Target Iowa"
                              ,members = [(Store {address="1024 Binary Ave."
                                                 ,amounts=[1,2,4,8]})
                                         ,(Store {address="256 Logical Blvd."
                                                 ,amounts=[7,25,15,44,36,8,3,7]})]})
                      ,(Group {gname = "Target New York"
                              ,members =
                                  [(Group {gname = "Target NYC"
                                          ,members = [(Store {address = "Fifth Ave."
                                                             ,amounts= [8,2,1,3,3,5]})
                                                     ,(Store {address= "Albany"
                                                             ,amounts = []})]})]})]}))
  "==" (Group {gname = "TARGET"
              ,members=
               [(Group {gname = "TARGET FLORIDA"
                       ,members = [(Store {address = "227 INTERNATIONAL DR."
                                          ,amounts=[1,2,3,4,5]})
                                  ,(Store {address = "1005 ALAFAYA TRAIL RD."
                                          ,amounts=[101,102,103,104,105]})]})
               ,(Group {gname = "TARGET IOWA"
                       ,members = [(Store {address = "1024 BINARY AVE."
                                          ,amounts=[1,2,4,8]})
                                  ,(Store {address = "256 LOGICAL BLVD."
                                          ,amounts=[7,25,15,44,36]})]})
               ,(Group {gname = "TARGET NEW YORK"
                       ,members= [(Group {gname = "TARGET NYC"
                                         ,members = [(Store {address= "FIFTH AVE."
                                                            ,amounts=[8,2,1,3,3]})
                                                    ,(Store {address="ALBANY"
                                                            ,amounts=[]})]})]})]})]
```

```
module NormalizeSalesData (normalizeSalesData) where
import SalesData
import Data.Char (toUpper)

normalizeSalesData :: SalesData -> SalesData
normalizeSalesData (Store {address = addr, amounts = amts}) =
    (Store {address = (capitalize addr), amounts = take 5 amts})
normalizeSalesData (Group {gname = name, members = membs}) =
    (Group {gname = (capitalize name), members = map normalizeSalesData membs})

capitalize str = map toUpper str
```

Figure 9: The function `normalizeSalesData`, which follows the Sales Data grammar.

```
module NormalizeSalesData (normalizeSalesDataBad) where
import SalesData
import Data.Char (toUpper)

normalizeSalesDataBad :: SalesData -> SalesData
normalizeSalesDataBad (Store {address = addr, amounts = amts}) =
    (Store {address = (normalizeSalesDataBad addr), amounts = take 5 amts})
normalizeSalesDataBad (Group {gname = name, members = membs}) =
    (Group {gname = (normalizeSalesDataBad name)
           , members = normalizeSalesDataBad membs})
-- The following code should not be in this function!
normalizeSalesDataBad [] = []
normalizeSalesDataBad (sd:sds) = (normalizeSalesDataBad sd)
                                 : (normalizeSalesDataBad sds)
normalizeSalesDataBad "" = ""
normalizeSalesDataBad (c:cs) = (toUpper c) : (normalizeSalesDataBad cs)
```

Figure 10: The function `NormalizeSalesDataBad`, which does not follow the Sales Data grammar.

this type checked, the pattern matching would never get to these cases. Can you see how the pattern matching is confused? Is the problem solved by reordering the last 4 cases? Why not?

5.4 Boolean Expressions

The grammar for Boolean expressions is shown in Figure 11 on the next page. It has multiple nonterminals, but does not have mutual recursion among the nonterminals. The grammar type `String` is Haskell's built-in String type. These Strings represent variable identifiers in Boolean expressions.

5.4.1 Example

An example using this grammar is shown in Figure 12.

5.4.2 Beval Exercise

Write a function

```
bEval :: (Eq a) => Bexp -> (String -> a) -> Bool
```

```
module Bexp where

data Bexp = And Bexp Bexp | Or Bexp Bexp | Not Bexp | Comp Comparison
            deriving (Eq, Show)
data Comparison = Equals String String | NotEquals String String
                  deriving (Eq, Show)
```

Figure 11: Grammar for Boolean expressions.

```
module NegateBexp where
import Bexp

negateBexp :: Bexp -> Bexp
negateBexp (And l r) = (Or (negateBexp l) (negateBexp r))
negateBexp (Or l r) = (And (negateBexp l) (negateBexp r))
negateBexp (Not e) = e
negateBexp (Comp c) = (Comp (negateComparison c))

negateComparison :: Comparison -> Comparison
negateComparison (Equals v1 v2) = (NotEquals v1 v2)
negateComparison (NotEquals v1 v2) = (Equals v1 v2)
```

Figure 12: The function `negateBexp`, which follows the Boolean expression grammar.

that takes 2 arguments: a `Bexp`, e, and a function f from atoms to values of some Eq type. Assume that f is defined on each String literal that occurs in e. The function `bEval` should evaluate the expression e, using f to determine the values of all Strings that occur within it. The `bEval` function should: conjoin the values that result from the evaluation of the subexpressions in an `And` expression, disjoin the values of subexpressions in an `Or` negate the value of the subexpression in a `Not`, and use the value of the comparison contained in a `Comp` expression. For the `Comparison` expressions, the value of `Equals` v w should be true just when $fv == fw$, and the value of `NotEquals` v w should be true just when fv /= fw. Examples are shown in Figure 13 on the following page.

5.5 Statements and Expressions

The grammar for statements and expressions below involves mutual recursion. Statements can contain expressions and expressions can contain statements. This mutual recursion allows for arbitrary nesting. The type definition is shown in Figure 14 on page 19.

5.5.1 Examples

An example that follows this grammar is shown in Figure 15. It is instructive to draw arrows from each use of `StmtAdd1` and `ExpAdd1`, in the figure, to where these names are defined, and to draw arrows from the uses of the corresponding nonterminals in the grammar to where they are defined. That helps show how the recursion patterns match.

Note that, in Figure 15 the recursive call from `expAdd1` to `StmtAdd1` occurs in the case for `beginExp`, following the grammar. However, this recursive call is actually made by `map`, since the recursion in the grammar is inside a list. It would be fine to use a separate helping function for this list recursion, and that will be necessary in cases that `map` cannot handle.

```
-- $Id: BEvalTests.hs,v 1.1 2013/01/25 01:24:45 leavens Exp leavens $
module NegateBexpTests where
import Testing
import Bexp
import BEval

main = dotests "BEvalTests $Revision: 1.1 $" tests

-- This function is just for ease of testing
stdEnv v =
   case v of
     "p" -> 1
     "q" -> 2
     "r" -> 4020
     "x" -> 76
     "y" -> 0
     _ -> undefined


tests :: [TestCase Bool]
tests =
 [(assertTrue (bEval (Comp (Equals "q" "q")) stdEnv))
 ,(assertFalse (bEval (Comp (NotEquals "q" "q")) stdEnv))
 ,(assertFalse (bEval (Comp (Equals "q" "r")) stdEnv))
 ,(assertTrue (bEval (Comp (NotEquals "p" "q")) stdEnv))
 ,(assertTrue (bEval (Or (Comp (NotEquals "p" "q"))
                         (Comp (Equals "x" "x"))) stdEnv))
 ,(assertFalse (bEval (And (Comp (NotEquals "p" "q"))
                           (Comp (NotEquals "x" "x"))) stdEnv))
 ,(assertFalse (bEval (And (Not (Comp (Equals "p" "p")))
                           (Comp (Equals "x" "x"))) stdEnv))
 ,(assertTrue (bEval (Not (And (Not (Comp (Equals "p" "p")))
                               (Comp (Equals "x" "x")))) stdEnv))
 ,(assertTrue (bEval (Or (Not (And (Not (Comp (Equals "p" "p")))
                                   (Comp (Equals "x" "x"))))
                         (Or (Comp (Equals "p" "q"))
                             (Comp (Equals "x" "x")))) stdEnv))
 ,(assertFalse (bEval (Or
                       (And (Not (Comp (Equals "p" "p")))
                            (Comp (Equals "x" "x")))
                       (Or (Comp (Equals "p" "q"))
                           (Comp (Equals "x" "y")))) stdEnv))
 ]
```

Figure 13: Testing for `BEval`.

```haskell
-- $Id: StatementsExpressions.hs,v 1.1 2013/01/25 13:44:12 leavens Exp leavens $
module StatementsExpressions where

data Statement = ExpStmt Expression
               | AssignStmt String Expression
               | IfStmt Expression Statement
                 deriving (Eq, Show)

data Expression = VarExp String
     | NumExp Integer
     | EqualsExp Expression Expression
     | BeginExp [Statement] Expression
       deriving (Eq, Show)
```

Figure 14: The Statements and Expressions grammar.

```haskell
-- $Id$
module StmtAdd1 where
import StatementsExpressions

stmtAdd1 :: Statement -> Statement
stmtAdd1 (ExpStmt e) = (ExpStmt (expAdd1 e))
stmtAdd1 (AssignStmt v e) = (AssignStmt v (expAdd1 e))
stmtAdd1 (IfStmt e s) = (IfStmt (expAdd1 e) (stmtAdd1 s))

expAdd1 :: Expression -> Expression
expAdd1 (VarExp v) = (VarExp v)
expAdd1 (NumExp i) = (NumExp (i+1))
expAdd1 (EqualsExp e1 e2) = (EqualsExp (expAdd1 e1) (expAdd1 e2))
expAdd1 (BeginExp stmts e) = (BeginExp (map stmtAdd1 stmts) (expAdd1 e))
```

Figure 15: The functions `StmtAdd1` and `ExpAdd1`. These mutually-recursive functions work on the statement and expression grammar.

5.5.2 SubstIdentifier Exercise

Write a function

```
substIdentifier :: Statement -> String -> String -> Statement
```

that takes a statement `stmt` and two Strings, `new` and `old`, and returns a statement that is just like `stmt`, except that all occurrences of `old` in `stmt` are replaced by `new`. Examples are shown in Figure 16 on the following page.

5.5.3 More Statement and Expression Exercises

Invent one or more other problem on the statement and expression grammar above. For example, try reversing the list of statements in every `BeginExp` whose expression contains a ⟨NumExp⟩ with a number less than 3. For a more challenging exercise, write an evaluator for this grammar.

Acknowledgments

Thanks to Faraz Hussain for corrections to the Oz version of this paper [8], and to Brian Patterson and Daniel Patanroi for comments on drafts of a Scheme version of this paper [7]. Earlier versions were supported in part by NSF grants CCF-0428078, CCF-0429567, CNS 07-09217, and CCF-08-16350. This version was supported in part by NSF grants CCF-0916715, 0916350, 1017262, 1017334, and 1228695.

References

[1] R. Sethi A. V. Aho and J. D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *SIMULA Begin*. Auerbach Publishers, Philadelphia, Penn., 1973.

[3] Daniel P. Friedman and Matthias Felleisen. *The Little Schemer*. MIT Press, fourth edition, 1996.

[4] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press, New York, NY, second edition, 2001.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[6] Michael A. Jackson. *Principles of Program Design*. Academic Press, London, 1975.

[7] Gary T. Leavens. Following the grammar. Technical Report 05-02a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, January 2006. Available by anonymous ftp from ftp.cs.iastate.edu.

[8] Gary T. Leavens. Following the grammar. Technical Report CS-TR-07-10b, School of EECS, University of Central Florida, Orlando, FL, 32816-2362, November 2007.

[9] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass., 2004.

[10] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, NJ 07632, 1990.

```
module SubstIdentifierTests where
import StatementsExpressions
import SubstIdentifier
import Testing

main = dotests "BEvalTests $Revision$" tests

tests :: [TestCase Statement]
tests =
 [(eqTest (substIdentifier (ExpStmt (NumExp 7)) "x" "y")
   "==" (ExpStmt (NumExp 7)))
 ,(eqTest (substIdentifier (ExpStmt (VarExp "y")) "x" "y")
   "==" (ExpStmt (VarExp "x")))
 ,(eqTest (substIdentifier
           (ExpStmt (BeginExp [(AssignStmt "x" (NumExp 7))]
                              (EqualsExp (VarExp "x") (VarExp "q"))))
           "z" "x")
   "==" (ExpStmt (BeginExp [(AssignStmt "z" (NumExp 7))]
                           (EqualsExp (VarExp "z") (VarExp "q")))))
 ,(eqTest (substIdentifier
           (ExpStmt (BeginExp [] (EqualsExp (VarExp "z") (VarExp "z"))))
           "a" "z")
   "==" (ExpStmt (BeginExp [] (EqualsExp (VarExp "a") (VarExp "a")))))
 ,(eqTest (substIdentifier (ExpStmt (VarExp "a")) "a" "z")
   "==" (ExpStmt (VarExp "a")))
 ,(eqTest (substIdentifier (AssignStmt "y" (VarExp "m")) "q" "y")
   "==" (AssignStmt "q" (VarExp "m")))
 ,(eqTest (substIdentifier (IfStmt (EqualsExp (VarExp "y") (NumExp 10))
                                   (AssignStmt "y" (NumExp 0)))
                           "w" "y")
   "==" (IfStmt (EqualsExp (VarExp "w") (NumExp 10))
                (AssignStmt "w" (NumExp 0))))
 ,(eqTest (substIdentifier (IfStmt
                     (BeginExp [(AssignStmt "x" (NumExp 3))
                               ,(AssignStmt "y" (VarExp "q"))
                               ,(AssignStmt "b"
                                             (EqualsExp (VarExp "x") (NumExp 2)))]
                               (EqualsExp (VarExp "b") (NumExp 0)))
                     (AssignStmt "q" (NumExp 17)))
           "z" "x")
   "==" (IfStmt
         (BeginExp [(AssignStmt "z" (NumExp 3))
                   ,(AssignStmt "y" (VarExp "q"))
                   ,(AssignStmt "b" (EqualsExp (VarExp "z") (NumExp 2)))]
                   (EqualsExp (VarExp "b") (NumExp 0)))
         (AssignStmt "q" (NumExp 17))))
 ]
```

Figure 16: Tests for the function `SubstIdentifier`.