Following the Grammar

Gary T. Leavens

CS-TR-07-10d
September 2007, revised October 2007, November 2007, October 2009, February 2011

Keywords: Recursion, programming recursive procedures, recursion pattern, inductive definition, BNF grammar, Kleene star, follow the grammar, functional programming, list recursion, programming languages, concrete syntax, abstract syntax, helping procedures, parsing procedures, Oz.

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;

(cc) (BY:) This document is distributed under the terms of the Creative Commons Attribution License, version 2.0, see `http://creativecommons.org/licenses/by/2.0/`.

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

Following the Grammar

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

October 21, 2007

Abstract

This document 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 Oz, 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, 8], 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, 9] 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 ⟨s-list⟩.

⟨stmt⟩ ::= `skip` | `assign(`⟨var⟩ ⟨exp⟩`)` | `compound(`⟨s-list⟩`)`
⟨s-list⟩ ::= `nil` | ⟨stmt⟩ '`|`' ⟨s-list⟩

The nonterminal ⟨stmt⟩ has three alternatives (separated by the vertical bars). The production for ⟨stmt⟩ has a recursive call to ⟨s-list⟩, which, in turn, recursively calls ⟨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, $\langle X \rangle$, in G, there is a function, FX, in Prog that takes an argument from the set described by the nonterminal $\langle X \rangle$.*
2. *If a nonterminal $\langle X \rangle$ has alternatives, then the corresponding function FX decides between the alternatives offered in $\langle X \rangle$'s grammatical productions, and there is (at least) one case in the body of FX for each alternative production for $\langle X \rangle$.*
3. *If the nonterminal $\langle X \rangle$ has an alternative whose production contains a nonterminal $\langle Y \rangle$ that is defined in the grammar G, then:*
   - *(a) the corresponding case in the function FX has a call to function FY, where FY is the function in Prog that handles data described by the nonterminal $\langle Y \rangle$, and*
   - *(b) each call from FX to FY passes to FY a part of the data FX received as an argument, namely that part described by $\langle Y \rangle$.*

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'll 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⟩ ::=
      hot
    | warm
    | cold
```

2.1.1 Example

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

```
declare
fun {SelectOuterwear Temp}
   case Temp of
      hot then none
   [] warm then windBreaker
   else downJacket
   end
end
```

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: <fun {$ <Temperature>}: <Bool>>
```

that follows the grammar for Temperature?

1. ```
   declare
   fun {IsFreezing Temp}
      case Temp of
         nil then false
      [] cold|_ then true
      [] _|T then {IsFreezing T}
   end
   ```

2. ```
   declare
   fun {IsFreezing Temp}
      case Temp of
           nil then false
      else H|T then H == cold orelse {IsFreezing T}
   end
   ```

3. ```
   declare
   fun {IsFreezing Temp}
      case Temp of
         cold then true
      [] hot then false
      else false
   end
   ```

4. ```
   declare
   fun {IsFreezing Temp}
      case Temp of
         cold then {IsFreezing Temp}
      [] hot then {Not {IsFreezing Temp}}
      else false
   end
   ```

Answer: 3.

2.2 Color Grammar Exercises

Consider another example with simple alternatives and no recursion:

$\langle$Color$\rangle$ ::= `red` | `yellow` | `green` | `blue`

Write the function:

```
EqualColor : <fun {$ <Color> <Color>} : <Bool>>
```

that takes two colors and returns **`true`** if they are the same, and **`false`** otherwise.

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 such a grammar.

$\langle$ISeq$\rangle$ ::=
    `iseq(`$\langle$Number$\rangle$ $\langle$ISeq$\rangle$`)`

In Oz one can create such infinite sequences using unbound variables, as in the following example.

```
declare Ones
Ones = iseq(1 Ones)
```

3.1.1 Example

A lazy function

```
ISeqMap : <fun {$ <ISeq> <fun {$ <Number>} : <Number>>} : <ISeq>>
```

that follows the above grammar is the following example.

```
declare
fun lazy {ISeqMap Seq F}
   case Seq of
      iseq(N Tail) then iseq({F N} {ISeqMap Tail F})
   end
end

% some tests
Ones = iseq(1 Ones)
fun {Add2 X} X+2 end
Threes = {ISeqMap Ones Add2}
{Browse Threes.2.1}
```

Following the grammar in this example means that the function does something with the number `N` of the ISeq and recurses on its `Tail`, just like the grammar describes instances as having a number and a tail, which is a ISeq where it recurses. 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 the function is lazy. In the tests, the `Browse` shows the numeral 3.

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 : <fun {$ <ISeq>}: Bool>
```

that follows the grammar for ISeq?

1. ```
   declare
   fun {AnyNegative Seq}
      case Seq of
         nil then false
      [] N|Tail then N < 0 orelse {AnyNegative Tail}
      end
   end
   ```

2. ```
   declare
   fun {AnyNegative Seq}
      case Seq of
         iseq(N Tail) then N < 0 orelse {AnyNegative Tail}
      end
   end
   ```

3. ```
   declare
   fun {AnyNegative Seq}
      N < 0 orelse {AnyNegative Tail}
   end
   ```

4. ```
   declare
   fun {AnyNegative Seq}
      case Seq of
         cold then true
      else false
      end
   end
   ```

Answer: 2.

3.1.3 Filter Infinite Sequence Exercise

Write a lazy function

```
FilterISeq : <fun {$ <fun {$ <Number>}: <Bool>> ISeq}: ISeq>
```

that takes a predicate, `Pred`, an infinite sequence, `Seq`, and returns an infinite sequence of all elements in `Seq` for which `Pred` returns `true` when applied to the element, in their original order. For example,

```
{FilterISeq fun {$ N} N mod 2 == 0 end Integers}
```

would return an `ISeq` that is just like `Integers` but without its odd elements.

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, $\langle X \rangle$, uses another nonterminal, $\langle Y \rangle$, there should be a call from the function for $\langle X \rangle$ to the function for $\langle Y \rangle$ that passes an instance of $\langle Y \rangle$ as an argument.

4.1 Rectangle Grammar

Consider the following example grammar, for rectangles.

⟨Rectangle⟩ ::= `rectangle(ul: ⟨Point⟩ lr: ⟨Point⟩)`
⟨Point⟩ ::= `point(x: ⟨Number⟩ y: ⟨Number⟩)`

4.1.1 Example

To follow this grammar when writing a program like

```
MoveUp : <fun {$ <Rectangle> <Number>} : <Rectangle>>
```

one would structure the code into two functions, one for each of the two nonterminals, as shown in Figure 1. Since the production for ⟨Rectangle⟩ uses the nonterminal ⟨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 number described by the corresponding nonterminals, and are extracted from the number by the pattern match in the body of `MoveUp`

```
declare
fun {MoveUp Rect Delta}
   case Rect of
      rectangle(ul: UL lr: LR) then rectangle(ul: {MoveUpPoint UL Delta}
                                              lr: {MoveUpPoint LR Delta})
   end
end

fun {MoveUpPoint Pt Delta}
   case Pt of
      point(x: X y: Y) then point(x: X y: Y+Delta)
   end
end
```

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 : <fun {$ <Rectangle>}: <Rectangle>>
```

that follows the grammar for Rectangles.

1.
```
declare
fun {DoubleRect Rect}
   case Rect of
      rectangle(ul: point(x:ULX y: ULY)
                lr: point(x:LRX y: LRY)
      then rectangle(ul: point(x: ULX+ULX y: ULY+ULY)
                     lr: point(x: LRX+LRX y: LRY+LRY)
   end
end
```

2.
```
declare
fun {DoubleRect Rect}
   case Rect of
      rectangle(ul: UL lr: LR) then rectangle(ul: {DoubleRect UL}
                                              lr: {DoubleRect LR})
   [] point(x: X y: Y) then point(x: X+X y: Y+Y)
   end
end
```

3.
```
declare
fun {DoubleRect Rect}
   case Rect of
      UL|LR|nil then {DoublePoint UL}|{DoublePoint LR}|nil
   end
end
fun {DoublePoint Pt}
   case Pt of
      X|Y|nil then (X+X)|(Y+Y)|nil
   end
end
```

4.
```
declare
fun {DoubleRect Rect}
   case Rect of
      rectangle(ul: UL lr: LR) then rectangle(ul: {DoublePoint UL}
                                              lr: {DoublePoint LR})
   end
end
fun {DoublePoint Pt}
   case Pt of
      point(x: X y: Y) then point(x: X+X y: Y+Y)
   end
end
```

Answer: 4.

4.1.3 ShrinkRect Exercise

Write a function,

```
ShrinkRect : <fun {$ <Rectangle> <Number>} : <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.

$\langle$List T$\rangle$ ::= $\langle$T$\rangle$ ‘|’ $\langle$List T$\rangle$
  | `nil`

where $\langle$T$\rangle$ is the nonterminal that generates elements of type T.

This 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 : <fun {$ <List T> <fun {$ <T>} : <S>>}: <List S>>
```

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

```
declare
fun {Map Lst F}
   case Lst of
      T|Ts then {F T}|{Map Ts F}
   else nil
   end
end
```

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 : <fun {$ <List <Person>>} : <List String>>
```

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.) Assume that the function `GetName` is defined elsewhere.

1.
```
declare
fun {ExtractNames Lst}
   case Lst of
      Person|Persons then {GetName Person}|Persons
   else nil
   end
end
```

2. ```
   declare
   fun {ExtractNames Lst}
      case Lst of
         Person|Persons then {GetName Person}|{ExtractNames Persons}
      else nil
      end
   end
   ```

3. ```
   declare
   fun {ExtractNames Lst}
      if Lst == nil
      then nil
      else {GetName Lst.1}|{ExtractNames Lst.2}
      end
   end
   ```

4. ```
   declare
   fun {ExtractNames Lst}
      if Lst == nil
      then nil
      else {GetName Lst.1}|Lst.2
      end
   end
   ```

5. ```
   declare
   fun {ExtractNames Lst}
      case Lst of
         Person|Persons then {GetName Person}|{ExtractNames Persons}
      end
   end
   ```

6. ```
   declare
   fun {ExtractNames Lst}
      {Map Lst GetName}
   end
   ```

Answer: 2, 3, 6.

5.1.3 DeleteListing Exercise

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

```
DeleteListing : <fun {$ <String> <List <Person>>} : <List Person>>
```

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

(As previously, assume that the function `GetName` is defined elsewhere.)
```

1.
```
declare
fun {DeleteListing Name Lst}
   case Lst of
      Person|Persons then if Name == {GetName Person}
                          then {DeleteListing Name Persons}
                          else Person|{DeleteListing Name Persons}
                          end
   else nil
   end
end
```

2.
```
declare
fun {DeleteListing Name Lst}
   case Lst of
      Person|Persons andthen Name == {GetName Person}
                     then {DeleteListing Name Persons}
   [] Person|Persons then Person|{DeleteListing Name Persons}
   else nil
   end
end
```

3.
```
declare
fun {DeleteListing Name Lst}
   if Lst == nil
   then nil
   elseif Name == {GetName Lst.1}
   then {DeleteListing Name Lst.2}
   else Lst.1|{DeleteListing Name Lst.2}
   end
end
```

4.
```
declare
fun {DeleteListing Name Lst}
   case Lst of
      Person|Persons then {Append
                            if Name == {GetName Person}
                            then nil else [Person]
                            end
                            {DeleteListing Name Persons}
                          }
   else nil
   end
end
```

5.
```
declare
fun {DeleteListing Name Lst}
   case Lst of
      Person|Persons andthen Name == {GetName Person}
                     then {DeleteListing Name Persons}
   else Person|{DeleteListing Name Persons}
   end
end
```

6.
```
declare
fun {DeleteListing Name Lst}
   case Lst of
      Person|Persons if Name == {GetName Person}
                     then {DeleteListing Name Persons}
                     else Person|{DeleteListing Name Persons}
                     end
   end
end
```

Answer: 1, 2, 3, 4.

5.2 Window Layouts

For purposes of this paper, a "window layout" is an instance of the grammar below. The grammar is designed to describe an (imagined) 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.

⟨WindowLayout⟩ ::=
    `window(name:` ⟨Atom⟩ `width:` ⟨Number⟩ `height:` ⟨Number⟩`)`
  | `horizontal(`⟨List WindowLayout⟩`)`
  | `vertical(`⟨List WindowLayout⟩`)`

In the above grammar, the nonterminals ⟨Number⟩ and ⟨Atom⟩ have the same syntax as in Oz.

5.2.1 Example

An example function that follows the above grammar is shown in Figure 3. This example is coded using Oz's built-in `Map` function (which was also shown in Figure 2 on page 8). 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 be perfectly fine, however. These helping function(s) would, of course, follow the grammar for flat lists.

```
declare
fun {DoubleSize WL}
   case WL of
      window(name: N width: W height: H)
      then window(name: N width: 2*W height: 2*H)
   [] horizontal(WLs) then horizontal({Map WLs DoubleSize})
   [] vertical(WLs) then vertical({Map WLs DoubleSize})
   end
end
```

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

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.

```
declare
fun {BadDoubleSize WL}
   case WL of
      window(name: N width: W height: H)
      then window(name: N width: 2*W height: 2*H)
   [] horizontal(WLs) then horizontal({BadDoubleSize WLs})
   [] vertical(WLs) then vertical({BadDoubleSize WLs})
   %% The following case clauses should not be in this function!
   [] nil then nil
   [] Head|Tail then {BadDoubleSize Head}|{BadDoubleSize Tail}
   end
end
```

Figure 4: The function `BadDoubleSize`, which does not follow the grammar for window layouts, because it incorrectly combines two recursions in one function.

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 **`fun`** `{$ WL} {DoubleSize WL}` **`end`**. However, in other problems using such an anonymous function (or a named helping function) may be needed, because the function argument to `Map` must be a function that takes exactly one argument. 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 the fixed `N`. This is shown in Figure 5.

```
declare
fun {AddToSize N WL}
   fun {AddWithFixedN WL} {AddToSize N WL} end
in
   case WL of
      window(name: Name width: W height: H)
      then window(name: Name width: W+N height: H+N)
   [] horizontal(WLs) then horizontal({Map WLs AddWithFixedN})
   [] vertical(WLs) then vertical({Map WLs AddWithFixedN})
   end
end
```

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

5.2.2 MultSize Exercise

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

```
MultSize : <fun {$ <WindowLayout> <Number>} : <WindowLayout>>
```

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

1. ```
   declare
   fun {MultSize WL N}
      case WL of
         window(name: N width: W height: H) then
         window(name: N width: N*W height: N*H)
      [] horizontal(WLs) then
         horizontal({Map WLs fun {$ WL} {MultSize WL N} end})
      [] vertical(WLs) then
         vertical({Map WLs fun {$ WL} {MultSize WL N} end})
      end
   end
   ```

2. ```
   declare
   fun {MultSize WL N}
      case WL of
         window(name: N width: W height: H) then
         window(name: N width: N*W height: N*H)
      [] horizontal(WLs) then
         horizontal({MultSize WLs N})
      [] vertical(WLs) then
         vertical({MultSize WLs N})
      [] nil then nil
      [] Head|Tail then {MultSize Head N}|{MultSize Tail N}
      end
   end
   ```

3. ```
   declare
   fun {MultSize WL N}
      case WL of
         window(name: N width: W height: H) then
         window(name: N width: N*W height: N*H)
      [] horizontal(WLs) then
         horizontal({MultSizeList WLs N})
      [] vertical(WLs) then
         vertical({MultSizeList WLs N})
      end
   end
   fun {MultSizeList WLs N}
      case WL of
         nil then nil
      [] Head|Tail then {MultSize Head N}|{MultSizeList Tail N}
      end
   end
   ```
   ```

4.
```
declare
fun {MultSize WL N}
   fun {MultSizeList WLs}
      case WL of
         nil then nil
      [] Head|Tail then {MultSize Head N}|{MultSizeList Tail}
      end
   end
in
   case WL of
      window(name: N width: W height: H) then
      window(name: N width: N*W height: N*H)
   [] horizontal(WLs) then
      horizontal({MultSizeList WLs})
   [] vertical(WLs) then
      vertical({MultSizeList WLs})
   end
end
```

5.
```
declare
fun {MultSize WL N}
   fun {MS WL}
      case WL of
         window(name: N width: W height: H) then
         window(name: N width: N*W height: N*H)
      [] horizontal(WLs) then
         horizontal({Map WLs MS})
      [] vertical(WLs) then
         vertical({Map WLs MS})
      end
  end
in
   {MS WL}
end
```

6.
```
declare
fun {MultSize WL N}
   case WL of
      window(name: N width: W height: H) then
      window(name: N width: N*W height: N*H)
   [] horizontal(WLs) then
      horizontal({MultSize WLs N})
   [] vertical(WLs) then
      vertical({MultSize WLs N})
   [] nil then nil
   [] Head|nil then {MultSize Head N}|nil
   [] Head|Tail then {MultSize Head N}|{MultSize Tail N}
   end
end
```

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 : <fun {$ <WindowLayout>}: <Number>>
```

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(name:` M `width:` N_1 `height:` N_2`)`

is N_1. The width of a ⟨WindowLayout⟩ of the form

`horizontal([`W_1 `...` W_m`])`

is the sum of the widths of W_1 through W_m (inclusive). The width of a ⟨WindowLayout⟩ of the form

`vertical([`W_1 `...` W_m`])`

is the maximum of the widths of W_1 through W_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 `Assert` procedure from the course library.)

```
\insert 'Assert.oz'
{Assert {TotalWidth window(name: olympics width: 50 height: 33)} == 50}
{Assert {TotalWidth horizontal(nil)} == 0}
{Assert {TotalWidth vertical(nil)} == 0}
{Assert {TotalWidth
         horizontal([window(name: olympics width: 80 height: 33)
                     window(name: localNews width: 20 height: 10)])}
        == 100}
{Assert {TotalWidth
         vertical([window(name: olympics width: 80 height: 33)
                   window(name: localNews width: 20 height: 10)])}
        == 80}
{Assert {TotalWidth
         vertical([window(name: starTrek width: 40 height: 100)
                   window(name: olympics width: 80 height: 33)
                   window(name: localNews width: 20 height: 10)])}
        == 80}
{Assert {TotalWidth
         horizontal(
            [vertical([window(name: tempest width: 200 height: 100)
                       window(name: othello width: 200 height: 77)
                       window(name: hamlet width: 1000 height: 600)])
             horizontal([window(name:baseball width: 50 height: 40)
                         window(name: track width: 100 height: 60)
                         window(name: equestrian width: 70 height: 30)])
             vertical([window(name: starTrek width: 40 height: 100)
                       window(name: olympics width: 80 height: 33)
                       window(name: localNews width: 20 height: 10)])
            ])}
        == 1300}
```

Figure 6: Tests for the `TotalWidth` exercise.

Feel free to use Oz's `Map` and `Max` functions (as well as `FoldL` or `FoldR` if you have seen those).

5.2.4 Design Your own WindowLayout Problem Exercise

Design another problem over WindowLayouts. 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 Sales Data 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 Oz, which is essentially ⟨List ⟨Char⟩⟩.

⟨SalesData⟩ ::=
      `store(address:` ⟨String⟩ `amounts:` ⟨List ⟨Int⟩⟩`)`
    | `group(name:` ⟨String⟩ `members:` ⟨List ⟨SalesData⟩⟩`)`

Figure 7: Grammar for the type ⟨SalesData⟩.

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 : <fun {$ <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 next 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.

```
{Test {NormalizeSalesData
       store(address: "1005 Alafaya Trail Rd."
             amounts: [101 102 103 104 105 106 107])}
 '==' store(address: "1005 ALAFAYA TRAIL RD."
            amounts: [101 102 103 104 105])}
{Test {NormalizeSalesData
       store(address: "227 International Dr."
             amounts: [1 2 3 4 5 6 7 8 9 10 11 1])}
 '==' store(address: "227 INTERNATIONAL DR." amounts: [1 2 3 4 5])}
{Test {NormalizeSalesData
       group(name: "Target"
             members: [store(address: "227 International Dr."
                             amounts: [1 2 3 4 5 6 7 8 9 10 11 1])
                       store(address: "1005 Alafaya Trail Rd."
                             amounts: [101 102 103 104 105 106 107])])}
 '==' group(name: "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])])}
{Test {NormalizeSalesData
       group(name: "Target"
             members:
                [group(name: "Target Florida"
                       members: [store(address: "227 International Dr."
                                       amounts: [1 2 3 4 5 6 7 8 9 10 11 1])
                                 store(address: "1005 Alafaya Trail Rd."
                                       amounts: [101 102 103 104 105 106 107])])
                 group(name: "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(name: "Target New York"
                       members: [group(name: "Target NYC"
                                       members: [store(address: "Fifth Ave."
                                                       amounts: [8 2 1 3 3 5])])
                                 store(address: "Albany" amounts: nil)])])}
 '==' group(name: "TARGET"
            members:
               [group(name: "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(name: "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(name: "TARGET NEW YORK"
                      members: [group(name: "TARGET NYC"
                                      members: [store(address: "FIFTH AVE."
                                                      amounts: [8 2 1 3 3])])
                                store(address: "ALBANY" amounts: nil)])])}
```

Figure 8: Testing for the function `NormalizeSalesData`.

Our code for `NormalizeSalesData` is shown in Figure 9. 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 `Capitals`, `Map`, and the built-in `List.take`. Can you see why this code works correctly?

```
declare
fun {NormalizeSalesData SD}
   case SD of
      store(address: Addr amounts: Amts) then
      store(address: {Capitals Addr} amounts: {List.take Amts 5})
   [] group(name: Name members: Membs) then
      group(name: {Capitals Name} members: {Map Membs NormalizeSalesData})
   end
end

% Return the argument in all capital letters
fun {Capitals Str} {Map Str Char.toUpper} end
```

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

Now consider the `NormalizeSalesDataBad`, shown in Figure 10, 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 `List.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 work correctly?

```
declare
fun {NormalizeSalesDataBad SD}
   case SD of
      store(address: Addr amounts: Amts) then
      store(address: {NormalizeSalesDataBad Addr} amounts: {List.take Amts 5})
   [] group(name: Name members: Membs) then
      group(name: {NormalizeSalesDataBad Name}
            members: {NormalizeSalesDataBad Membs})
   %% The following cases shouldn't be in this function!
   [] nil then nil
   [] Store1|OtherStores then
      {NormalizeSalesDataBad Store1}|{NormalizeSalesDataBad OtherStores}
   [] Char1|Rest then {Char.toUpper Char1}|{NormalizeSalesDataBad Rest}
   end
end
```

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

Indeed the code in Figure 10 is not correct! Can you see why? First, remember that in Oz Strings are lists of characters. The last two case clauses are attempting to deal with lists of characters separately from lists of sales data records. Can you see how the pattern matching is confused in the last two case clauses? Which one will execute? What will happen when it does? Is the problem solved by reordering the last 2 cases? Why not?

5.4 Boolean Expressions

The grammar for Boolean expressions is shown in Figure 11. It has multiple nonterminals, but does not have mutual recursion among the nonterminals. The grammar for $\langle$Atom$\rangle$ is the same as for Oz. These Atoms represent variable identifiers in boolean expressions.

$\langle$Bexp$\rangle$ ::=
    `andExp(`$\langle$Bexp$\rangle$ $\langle$Bexp$\rangle$`)`
  | `orExp(`$\langle$Bexp$\rangle$ $\langle$Bexp$\rangle$`)`
  | `notExp(`$\langle$Bexp$\rangle$`)`
  | `comp(`$\langle$Comp$\rangle$`)`
$\langle$Comp$\rangle$ ::= `equals(`$\langle$Atom$\rangle$ $\langle$Atom$\rangle$`)`
  | `notequals(`$\langle$Atom$\rangle$ $\langle$Atom$\rangle$`)`

Figure 11: Grammar for Boolean expressions.

5.4.1 Example

An example using this grammar is shown in Figure 12.

```
declare
fun {NegateBexp BE}
   case BE of
      andExp(L R) then orExp({NegateBexp L} {NegateBexp R})
   [] orExp(L R) then andExp({NegateBexp L} {NegateBexp R})
   [] notExp(E) then E
   [] comp(C) then comp({NegateComp C})
   end
end

fun {NegateComp C}
   case C of
      equals(A B) then notequals(A B)
   [] notequals(A B) then equals(A B)
   end
end
```

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

5.4.2 Beval Exercise

Write a function

```
BEval : <fun {$ <Bexp> <fun {$ <Atom>}: <Value>>}: <Bool>>
```

that takes 2 arguments: a `Bexp`, E, and a function from atoms to values, F. Assume that F is defined on each $\langle$Atom$\rangle$ that occurs in a $\langle$Bexp$\rangle$. The function `BEval` should evaluate the expression E, using F to determine the values of all $\langle$Atom$\rangle$s that occur within it. It should conjoin the values that result from the evaluation of the subexpressions in an `andExp` record, it should disjoin the values of subexpressions in an `orExp` record, negate the value of the subexpression in a `notExp` record, and use the value of the comparison contained in a `comp` record. For the $\langle$Comp$\rangle$ records, the value of `equals(A B)` should be true just when `{F A} == {F B}`, and the value of `notequals(A B)` should be true just when `{F A} \= {F B}`. Examples are shown in Figure 13.

```
\insert 'Assert.oz'
declare
fun {StdEnv A} % This function is just for ease of testing
   case A of
      p then 1
   [] q then 2
   [] r then 4020
   [] x then 76
   [] y then 0
   else raise stdEnvIsUndefinedOn(A) end
   end
end

{Assert {BEval comp(equals(q q)) StdEnv} == true}
{Assert {BEval comp(notequals(q q)) StdEnv} == false}
{Assert {BEval comp(equals(q r)) StdEnv} == false}
{Assert {BEval comp(notequals(p q)) StdEnv} == true}
{Assert {BEval andExp(comp(notequals(p q))
                      comp(equals(x x))) StdEnv} == true}
{Assert {BEval andExp(comp(notequals(p q))
                      comp(notequals(x x))) StdEnv} == false}
{Assert {BEval andExp(notExp(comp(equals(p p)))
                      comp(equals(x x))) StdEnv} == false}
{Assert {BEval notExp(andExp(notExp(comp(equals(p p)))
                             comp(equals(x x)))) StdEnv} == true}
{Assert {BEval orExp(notExp(andExp(notExp(comp(equals(p p)))
                                   comp(equals(x x))))
                     orExp(comp(equals(p q))
                           comp(equals(x x)))) StdEnv} == true}
{Assert {BEval orExp(andExp(notExp(comp(equals(p p)))
                            comp(equals(x x)))
                     orExp(comp(equals(p q))
                           comp(equals(x y)))) StdEnv} == false}
```

Figure 13: Testing for `BEval`.

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. In the following grammar the nonterminal ⟨Atom⟩, which stands for variable identifiers, has the same syntax as in Oz.

⟨Statement⟩ ::=
    `expStmt(`⟨Expression⟩`)`
  | `assignStmt(`⟨Atom⟩ ⟨Expression⟩`)`
  | `ifStmt(`⟨Expression⟩ ⟨Statement⟩`)`
⟨expression⟩ ::=
    `varExp(`⟨Atom⟩`)`
  | `numExp(`⟨Number⟩`)`
  | `equalsExp(`⟨Expression⟩ ⟨Expression⟩`)`
  | `beginExp(`⟨List Statement⟩ ⟨Expression⟩`)`

5.5.1 Examples

An example that follows this grammar is shown in Figure 14. It's 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.

```
declare
fun {StmtAdd1 Stmt}
   case Stmt of
      expStmt(E) then expStmt({ExpAdd1 E})
   [] assignStmt(I E) then assignStmt(I {ExpAdd1 E})
   [] ifStmt(E S) then ifStmt({ExpAdd1 E} {StmtAdd1 S})
   end
end

fun {ExpAdd1 Exp}
   case Exp of
      varExp(I) then varExp(I)
   [] numExp(N) then numExp(N+1)
   [] equalsExp(E1 E2) then equalsExp({ExpAdd1 E1} {ExpAdd1 E2})
   [] beginExp(Stmts E) then beginExp({Map Stmts StmtAdd1} {ExpAdd1 E})
   end
end
```

Figure 14: The functions `StmtAdd1` and `ExpAdd1`. These mutually-recursive functions work on the statement and expression grammar.

Note that, in Figure 14 the recursive call from `ExpAdd1` to `StmtAdd1` occurs in the case for `beginExp`, following the grammar. However, this recursive call is actually made inside `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. Another thing to note about this example is that the varExp case of `ExpAdd1` could be simplified to return `Exp` instead of returning `varExp(I)`. The form used in Figure 14 is more complex and slower, but shows the general pattern more clearly.

5.5.2 Subst-Identifier Exercise

Write a function

```
SubstIdentifier : <fun {$ <Statement> <Atom> <Atom>}: <Statement>>
```

that takes a statement `Stmt` and two atoms, `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 15.

```
\insert 'Assert.oz'
{Assert {SubstIdentifier expStmt(varExp(q)) p q} == expStmt(varExp(p))}
{Assert {SubstIdentifier expStmt(varExp(r)) p q} == expStmt(varExp(r))}
{Assert {SubstIdentifier assignStmt(a varExp(a)) n a}
                      == assignStmt(n varExp(n))}
{Assert {SubstIdentifier
         ifStmt(equalsExp(varExp(id) numExp(0)) assignStmt(id varExp(b)))
         var id}
      == ifStmt(equalsExp(varExp(var) numExp(0)) assignStmt(var varExp(b)))}
{Assert {SubstIdentifier expStmt(beginExp(nil varExp(a))) n a}
                      == expStmt(beginExp(nil varExp(n)))}
{Assert {SubstIdentifier
         expStmt(beginExp([ifStmt(equalsExp(varExp(a) numExp(0))
                                  assignStmt(a varExp(b)))
                           assignStmt(a varExp(a))
                          ]
                          beginExp(nil varExp(a))))
         n a}
      == expStmt(beginExp([ifStmt(equalsExp(varExp(n) numExp(0))
                                  assignStmt(n varExp(b)))
                           assignStmt(n varExp(n))
                          ]
                          beginExp(nil varExp(n))))}
```

Figure 15: Tests for the function `SubstIdentifier`.

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, 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, and CCF-0429567. This version was supported in part by NSF grant CNS 07-09217.

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

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