Following the Grammar with Python

Gary T. Leavens

CS-TR-17-01
February 2017

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, Python

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;

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

Following the Grammar with Python

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

February 5, 2017

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 Python, especially those used for teaching introductory programming.

1 Introduction

An important skill in 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, 10], as they are described by such grammars, but also for many other programming tasks, because recursively-structured data is fairly common and important. 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, 11] 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 in Figure 1 on the following page has five non-terminals, ⟨Stmt⟩, ⟨StmtList⟩, ⟨Stmts⟩, ⟨Var⟩, and ⟨Exp⟩. The nonterminal ⟨Stmt⟩ has three alternatives (separated by the vertical bars). The production for ⟨Stmt⟩ has a call to ⟨StmtList⟩, which calls `Stmts`, which, in turn, recursively calls both ⟨Stmts⟩ and ⟨Stmt⟩.

Context-free grammars correspond very closely to Python class declarations. For example, the class declarations in Figure 2 on the next page correspond directly to the grammar for ⟨Stmt⟩ above. In the figure, the declared superclass `abc.ABC` is used to make sure that `Stmt` is an abstract base case (hence ABC), and

[1] This paper is adapted from a Haskell version [9], which itself is adapted from an Oz version, [8].

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

Figure 1: Example of a context-free grammar.

thus cannot be directly instantiated. New objects can only be made by creating objects of its subtypes `Skip`, `Assign`, or `Compound`.[2]

```
# $Id: Stmt.py,v 1.1 2017/01/11 19:57:48 leavens Exp $
import abc   # abc means "abstract base class"
class Stmt(abc.ABC):
    pass

class Skip(Stmt):
    pass

class Assign(Stmt):
    def __init__(self, var, exp):
        """var should be a string, exp should be an int"""
        self.var = var
        self.exp = exp

class Compound(Stmt):
    def __init__(self, stmtList):
        """stmtList should be a list of Stmts"""
        self.stmtList = stmtList
```

Figure 2: Classes that implement the grammar for ⟨Stmt⟩. Note that the `__init__` method defines how a newly created object is initialized. For example, to create an object of type `Assign`, one executes `Assign(v,e)`, which creates the object and calls the `__init__` method with **`self`** as the newly created object, `var` as the value of `v`, and `exp` as the value of `e`.

Examples of Python objects that are instances of this grammar include expressions that generate objects of one of the subtypes of `Stmt`, including: `Skip()`, `Assign("spam", 3)`, `Compound([])`, and the following.

```
Compound([Assign("spam", 3), Assign("spam", 4)])
```

1.2 Definition of Following the Grammar

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

[2] One could also use case classes as defined in macropy to do this kind of example, but macropy doesn't seem to work with Python 3 yet.

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 definition 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 fX has a call to 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, including at least 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 well-organized program, each kind of data (i.e., each nonterminal) has its own function that only handles that kind of data. This organization makes it easy to understand and modify the program, because if the grammar changes, then the impact on functions is well-defined and minimized.

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 Python, this corresponds to the class definitions in Figure 3 on the following page.

Notice that the nonterminal ⟨Temperature⟩ is translated into the class named `Temperature` in Python, and each of the alternatives in this grammar is translated into a very simple class (`Cold`, `Warm`, and `Hot`) of the so that each of these can be instances.

```
# $Id: Temperature.py,v 1.1 2017/01/08 18:27:14 Gary Exp $
import abc
class Temperature(abc.ABC):
    pass
class Cold(Temperature):
    pass
class Warm(Temperature):
    pass
class Hot(Temperature):
    pass
```

Figure 3: Classes that implement the grammar for ⟨Temperature⟩.

2.1.1 Example

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

```
# $Id: selectOuterWear.py,v 1.1 2017/01/08 18:24:23 Gary Exp Gary $
from Temperature import *
def selectOuterWear(temp):
    """return description of appropriate outerwear for temp"""
    if (isinstance(temp,Hot)): return "none"
    elif (isinstance(temp,Warm)): return "wind breaker"
    else: #(isinstance(temp,Cold))
        return "down jacket"
```

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.

The following file tests the above `selectOuterWear` function.

```
# $Id: test_selectOuterWear.py,v 1.1 2017/01/08 18:24:06 Gary Exp $
from selectOuterWear import selectOuterWear
from Temperature import *

def test_selectOuterWear():
    assert selectOuterWear(Hot()) == "none"
    assert selectOuterWear(Warm()) == "wind breaker"
    assert selectOuterWear(Cold()) == "down jacket"
```

2.1.2 isFreezing Exercise

Which of the following has a correct outline for a function `isFreezing` that takes a `Temperature` argument, returns a **`bool`** result, and follows the grammar for Temperature?

1. ```
   from Temperature import *
   def isFreezing(temp):
       if (isinstance(temp,Hot)):
            return False
       else:
            return isFreezing(temp-1)
   ```

2. ```
   from Temperature import *
   def isFreezing(temp):
       if (temp == []):
           return False
       else:
           return isinstance(temp,Cold) or isFreezing(temp[1:])
   ```

3. ```
   from Temperature import *
   def isFreezing(temp):
       if (isinstance(temp,Hot)): return False
       elif (isinstance(temp,Warm)): return False
       else:
           return True
   ```

4. ```
   from Temperature import *
   def isFreezing(temp):
       if (isinstance(temp,Cold)):
           return isFreezing(temp)
       elif (isinstance(temp,Hot)):
           return not isFreezing(temp)
       else:
           return True
   ```

Answer: 3. Note that 1 and 4 have recursion, which do 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 Python class hierarchy.

```
# $Id: Color.py,v 1.1 2017/01/09 13:22:16 leavens Exp $
import abc
class Color(abc.ABC): pass
class Red(Color): pass
class Yellow(Color): pass
class Green(Color): pass
class Blue(Color): pass
```

Write a function `equalColor` that takes two Colors and returns `True` just when they are the same. For example `equalColor(Red(), Red())` would be `True`. (Note that because == in Python compares object identities, an expression such as `Red() == Red()` returns `False`, which is the wrong answer.)

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⟩ ::= `ISeq(`⟨int⟩`,` ⟨ISeq⟩`)`

The above corresponds to the Python class in Figure 4, with the `tail()` method added for convenience.

```
# $Id: ISeq.py,v 1.2 2017/01/10 15:05:17 leavens Exp $
class ISeq:
    def __init__(self, val, rest):
        """val should be an int, and
           rest should be a zero-argument function that returns an ISeq."""
        self.val = val
        self.rest = rest
    def tail(self):
        return self.rest()
```

Figure 4: The class `ISeq` of infinite sequences.

In Python one can create such infinite sequences using zero-argument lambdas (anonymous functions) as the second argument to an `ISeq`, as shown in Figure 5 on the following page.

3.1.1 Example

A function `iSeqMap`, shown in Figure 6 on page 8 takes a function `f` (which itself takes a single argument and returns it) and an `ISeq` `seq`, that applies `f` to each element of `seq` and returns an `ISeq` of the results in the same order as the arguments in `seq`. Note how `iSeqMap` follows the above grammar.

Following the grammar in this example means that the function does something with the argument `n` and makes a recursive call on the tail of `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 Python 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.

Figure 7 on page 9 contains some tests of `iSeqMap`.

```
# $Id: test_ISeq.py,v 1.4 2017/02/02 03:21:14 leavens Exp $
from ISeq import *
from math import isclose
# an infinite sequence of ones
ones = ISeq(1, (lambda : ones))

def nats_from(n):
    """The sequence of natural numbers starting at n"""
    return ISeq(n, (lambda : nats_from(n+1)))

# the natural numbers
nats = nats_from(0)

def halves_from(n):
    """The infinite sequence of fractions 1/2**0, 1/2**1, 1/2**2, 1/2**3, etc."""
    return ISeq(1/2**n, (lambda : halves_from(n+1)))

# the sequence 1, 0.5, 0.25, 0.125, ...
halves = halves_from(0)

def test_ISeq():
    """tests for ISeq basics."""
    assert ones.val == 1
    assert ones.tail().tail().val == 1
    assert nth_elem(ones, 5) == 1
    assert nth_elem(ones, 99) == 1
    assert nats.val == 0
    assert nats.tail().val == 1
    assert nats.tail().tail().val == 2
    assert isclose(halves.val, 1.0)
    assert isclose(halves.tail().val, 0.5)

def nth_elem(seq, n):
    """Return the nth (counted from 0) element of the infinite sequence seq"""
    if (n == 0):
        return seq.val
    else:
        return nth_elem(seq.tail(), n-1)

def test_nth_elem():
    """Tests that use nth_elem to look at other elements in the sequence."""
    assert nth_elem(nats, 0) == 0
    assert nth_elem(nats, 1) == 1
    assert nth_elem(nats, 2) == 2
    assert nth_elem(nats, 99) == 99
    assert isclose(nth_elem(halves, 2), 0.25)
    assert isclose(nth_elem(halves, 3), 0.125)
    assert isclose(nth_elem(halves, 10), 1/2**10)
```

Figure 5: Testing for the `ISeq` type.

```
# $Id: iSeqMap.py,v 1.1 2017/01/10 15:05:17 leavens Exp $
from ISeq import *

def iSeqMap(f, seq):
    """Apply the function f to each element of the infinite sequence seq,
       returning a new infinite sequence.
       The function f operates on elements of the sequence individually."""
    return ISeq(f(seq.val), (lambda : iSeqMap(f, seq.tail())))
```

Figure 6: The function `iSeqMap` that maps a function over `ISeq`s.

```
# $Id: test_iSeqMap.py,v 1.3 2017/02/02 03:21:14 leavens Exp $
from ISeq import *
from iSeqMap import *
from test_ISeq import ones, nats, halves, nth_elem
from math import isclose
def ident(n):
    """Return the argument unchanged. This is the identity function."""
    return n

def inc(n):
    """Return the successor of the argument."""
    return n+1

def test_iSeqMap_ident():
    """Tests of iSeqMap usng the ident (identity) function."""
    onesmapped = iSeqMap(ident, ones)
    assert onesmapped.val == 1
    assert nth_elem(onesmapped, 101) == 1

def test_iSeqMap_inc():
    """Tests of iSeqMap using the inc function."""
    twos = iSeqMap(inc, ones)
    assert twos.val == 2
    assert nth_elem(twos, 500) == 2
    ints = iSeqMap(inc, nats)
    assert ints.val == 1
    assert nth_elem(ints,5) == 6
    assert nth_elem(ints,70) == 71
    fours = iSeqMap(inc, iSeqMap(inc, iSeqMap(inc, ones)))
    assert fours.val == 4
    assert nth_elem(fours, 321) == 4
    oneplus = iSeqMap(inc, halves)
    assert oneplus.val == 2
    assert isclose(nth_elem(oneplus,2), 1.25)
    assert isclose(nth_elem(oneplus,3), 1.125)

def test_iSeqMap_lambdas():
    """Tests of iSeqMap using lambdas."""
    # Note that (lambda n: n+1) is the same function as inc
    threes = iSeqMap((lambda n: n+1), iSeqMap(inc, ones))
    assert threes.val == 3
    assert nth_elem(threes, 333) == 3
    squares = iSeqMap((lambda n: n*n), nats)
    assert nth_elem(nats, 3) == 3
    assert nth_elem(squares, 3) == 9
    assert nth_elem(squares, 27) == 27*27
    negs = iSeqMap((lambda n: -n), nats)
    assert nth_elem(negs, 3) == -3
    assert nth_elem(negs, 28) == -28
```

Figure 7: Testing for `iSeqMap`, which shows how it can be used.

3.1.2 AnyNegative Exercise

Which of the following has a correct outline for a function `anyNegative` that takes as an argument an `ISeq` whose elements are numbers, and returns a **`bool`** that is `True` just when some element of the given sequence is a negative number. (Note that it is not required that the function terminate if no element is negative.)

1.
```
def anyNegative(seq):
    if (seq == []):
        return False
    else:
        return anyNegative(seq.tail())
```

2.
```
def anyNegative2(seq):
    return (seq.val < 0) or anyNegative(seq.tail())
```

3.
```
def anyNegative2(seq):
    return anyNegative(seq.tail())
```

4.
```
def anyNegative(seq):
    if (isinstance(seq, Cold))
        return True
    else:
        return False
```

5.
```
def anyNegative(seq):
    if (seq.val < 0):
        return True
    else:
        return anyNegative(seq.tail())
```

Answer: Both 2 and 5 are correct. Note that 3 ignores the first and hence all elements.

3.1.3 Filter Infinite Sequence Exercise

Write a function `filterISeq` that takes a predicate, `pred` (which itself takes an integer and returns a **`bool`**) and an `ISeq` (which contains integers), and which returns an `ISeq` of all elements in `seq` for which `pred` returns `True` when applied to the element. Elements retained are left in their original order. For example:

```
filterISeq((lambda n: n \% 2 == 0), nats)
```

would return an `ISeq` that contains the even integers (0, 2, 4, 6, 8, etc.).

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 grammar for Rectangles and Points.

⟨Rectangle⟩ ::= `Rectangle(`⟨Point⟩`,` ⟨Point⟩`)`
⟨Point⟩ ::= `Point(`⟨int⟩`,` ⟨int⟩`)`

The Python class definitions that correspond to the above grammar are shown in Figure 8.

```
# $Id: Rectangle.py,v 1.2 2017/02/05 02:54:57 leavens Exp $
class Rectangle():
    def __init__(self, ul, lr):
        """Initialize self so the the upper left point is ul, and
        the lower right point is lr. Both args should be Points."""
        self.ul = ul
        self.lr = lr

    def __eq__(self, r2):
        """Return True just when self is equivalent to r2 structurally"""
        return self.ul == r2.ul and self.lr == r2.lr

    def __repr__(self):
        """Return a string representing this rectangle."""
        return "Rectangle(" + repr(self.ul) + ", " + repr(self.lr) + ")"

class Point():
    def __init__(self, x, y):
        """Initialize self with the x and y coordinates of the point"""
        self.x = x
        self.y = y

    def __eq__(self, p2):
        """Return True just when self is equivalent to p2 structurally"""
        return self.x == p2.x and self.y == p2.y

    def __repr__(self):
        """Return a string representing this point."""
        return "Point(" + repr(self.x) + ", " + repr(self.y) + ")"
```

Figure 8: The classes `Rectangle` and `Point`. The methods named `__eq__()` define what == means for objects of that class. Thus the uses of == from `Rectangle`'s `__eq__()` method call `Point`'s `__eq__()` method. The methods named `__repr__()` define what a printed representation of objects of that type looks like. This representation is printed by the Python interpreter and can also be obtained by the expression **`repr`**`(o)` for an object `o`; thus the calls to **`repr`**() in the definition of `Rectangle`'s `__repr__()` method call the `__repr__()` method in class `Point`.

4.1.1 Example

To follow this grammar when writing a function like `moveUp` that takes a `Rectangle` and an **`int`** and returns a new `Rectangle` one would structure the code into two functions, one for each of the two types (i.e., nonterminals), as shown in Figure 9 on the following page. 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.

```
# $Id: moveUp.py,v 1.1 2017/01/11 19:57:48 leavens Exp $
from Rectangle import *
def moveUp(rect, delta):
    """rect is a Rectangle and delta is an integer amount to move upwards"""
    return Rectangle(moveUpPoint(rect.ul, delta),
                     moveUpPoint(rect.lr, delta))

def moveUpPoint(pt, delta):
    """pt is a Point and delta is an integer amount to move upwards"""
    return Point(pt.x, pt.y + delta)
```

Figure 9: The two functions that move Rectangles and Points up.

4.1.2 DoubleRect Exercise

Which of the following is a correct outline of a function `doubleRect(rect)` that takes a `Rectangle` as an argument and returns a Rectangle, and which that follows the grammar for Rectangles?

1. ```
   def doubleRect(rect):
       return Rectangle(doubleRect(rect.ul), doubleRect(rect.lr))
   ```

2. ```
   def doubleRect(rect):
       return Rectangle(Point(rect.ul.x, rect.ul.y * 2), rect.lr)
   ```

3. ```
   def doubleRect(rect):
       return Rectangle(doubleRect(rect.ul), rect.lr)

   def doubleRect(p):
       return Point(p.x, p.y * 2)
   ```

4. ```
   def doubleRect(rect):
       return Rectangle(doublePointY(rect.ul), rect.lr)

   def doublePointY(pnt):
       return Point(pnt.x, pnt.y * 2)
   ```

Answer: 4. Note that 1 goes into an infinite loop. Also 2 while it works, does not follow the rule about having one function per type, since there is no function for Points given. The correct answer is 4, which follows the grammar.

4.1.3 ShrinkRect Exercise

Write a function, `shrinkRect(rect, factor)` that takes a Rectangle, `rect`, and a Number `factor` (which is greater than or equal to 1) and returns a new rectangle whose sides are smaller than those of `rect` by `factor`, but with the same upper left point as `rect`. Tests for `shrinkRect` appear in Figure 10 on the next page.

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?

```
# $Id: test_shrinkRect.py,v 1.1 2017/02/05 02:54:57 leavens Exp $
from Rectangle import *
from shrinkRect import *
from math import isclose
def closeRect(r1, r2):
    return closePoint(r1.ul, r2.ul) and closePoint(r1.lr, r2.lr)

def closePoint(p1, p2):
    return isclose(p1.x, p2.x) and isclose(p1.y, p2.y)

def test_shrinkRect():
    r = Rectangle(Point(0,0), Point(-5,-6))
    rs = shrinkRect(r,3)
    assert closeRect(rs, Rectangle(Point(0,0), Point(-5/3, -6/3)))
    rs10 = shrinkRect(r,10)
    assert closeRect(rs10, Rectangle(Point(0,0), Point(-5/10, -6/10)))
```

Figure 10: Tests for the `shrinkRect` problem.

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.

To start, we consider only examples that involve two of these features. The grammar for flat (Lisp) lists and the grammar for "binary trees" both involve only one nonterminal, but have alternatives and recursion. The grammar for "sales data" has all three features: alternatives, recursions, and multiple nonterminals.

5.1 Flat Lists as in Lisp

The grammar for flat lists, as in Lisp, 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.

⟨list(t)⟩ ::= `Nil()` | `Cons(`⟨t⟩`,` ⟨list(t)⟩`)`

where ⟨t⟩ is the nonterminal that generates elements of type t.

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

Lisp lists are similar to, but not the same as, the built-in lists in Python. They correspond more closely to the classes defined in Figure 11 on the following page. (In the figure the method `__str__()` is similar to `__repr__()`, but is supposed to produce a string suitable for printing. The expression **`str`**`(o)` calls `o`'s `__str__()` method, as does the built-in `print` function in Python. Note that the implementation of the `__str__()` method in class `Cons` uses a helping method, `elements_str`, to produce a comma separated string showing all the elements of the list; this method follows the grammar for non-empty Lisp lists, as every `Cons` instance is a non-empty list. Another thing to note about the figure is that the implementation of the `__eq__()` method in class `Cons` is recursive, since the last call to == (on the tails of the lists) calls the `__eq__()` method. You should convince yourself that the implementation of this `__eq__()` method follows the grammar for Lisp lists; indeed it is an example of simultaneous recursion.)

Note that since Lisp lists are objects, the methods that operate on them, such as `isEmpty()` are invoked in an object-oriented fashion, with a receiver expression (e.g., the object being tested, which is passed as the `self` argument), followed by a dot (`.`), followed by the method name, and the rest of the arguments in parentheses. See the tests for Lisp lists are found in Figure 12 on page 15, as well as the way the code is written in Figure 11 on the following page itself.

```
# $Id: LispList.py,v 1.4 2017/02/05 22:21:24 leavens Exp $
import abc
class LispList(abc.ABC):
    pass
class Nil(LispList):
    def __init__(self):
        """Initialize this empty list"""
        pass
    def __eq__(self, lst):
        """Return True just when lst is also an instance of Nil."""
        return isinstance(lst, Nil)
    def __repr__(self):
        """Return a string representing this Nil instance."""
        return "Nil()"
    def __str__(self):
        """Return a string showing the elements of self."""
        return "[]"
    def isEmpty(self):
        """Return whether this list is empty."""
        return True

class Cons(LispList):
    def __init__(self, hd, tl):
        """Initialize this Cons with head hd and tail tl."""
        self.car = hd
        self.cdr = tl
    def __eq__(self, lst):
        """Return True just when self is structurally equivalent to lst."""
        return isinstance(lst, Cons) and lst.first() == self.first() \
            and lst.tail() == self.tail()
    def __repr__(self):
        """Return a string representing this list."""
        return "Cons(" + repr(self.first()) + ", " + repr(self.tail()) + ")"
    def elements_str(self):
        """Return a string of the elements of self, separated by commas."""
        if self.tail().isEmpty():
            return str(self.first())
        else:
            return str(self.first()) + ", " + self.tail().elements_str()
    def __str__(self):
        """Return a string showing the elements of self."""
        return "[" + self.elements_str() + "]"
    def isEmpty(self):
        """Return whether this list is empty."""
        return False
    def first(self):
        """Return the first element of this list."""
        return self.car
    def tail(self):
        """Return the rest of this list."""
        return self.cdr
```

Figure 11: Classes that implement lists as in Lisp, in the file `LispList.py`. See the text for more explanation.

```
# $Id: test_LispList.py,v 1.1 2017/02/05 02:54:57 leavens Exp $
from LispList import *
def test_nil():
    """Tests only using Nil."""
    nil = Nil()
    assert nil == nil
    assert repr(nil) == 'Nil()'
    assert str(nil) == '[]'
    assert nil.isEmpty()

def test_cons():
    """Tests that use Cons."""
    nil = Nil()
    lst = Cons(1, Cons(2, Cons(3, nil)))
    assert lst == lst
    assert Cons(4, lst) == Cons(4, lst)
    assert lst != nil
    assert lst != Cons(5, lst)
    assert lst != lst.tail()
    assert repr(lst) == "Cons(1, Cons(2, Cons(3, Nil())))"
    assert str(lst) == "[1, 2, 3]"
    assert repr(lst.tail()) == "Cons(2, Cons(3, Nil()))"
    assert lst.tail().first() == 2
    assert lst.first() == 1
    assert lst.tail().tail().first() == 3
    assert lst.tail().tail().tail() == nil
    assert not lst.isEmpty()
    assert not lst.tail().isEmpty()
```

Figure 12: Tests for the `LispList` module.

5.1.1 Examples of Recursion over Flat (Lisp) Lists

The `lcopy(lst)` function, shown in Figure 13, demonstrates the form of functions that follow the grammar for flat (Lisp) lists. They test to see if the list argument is empty, and if so return some constant, otherwise they use the `.first()` and `.tail()` methods to access the first element of the list and the tail of the list, and combine some operation (in this case doing nothing) on the first element with a recursive call to the function to process the rest of the list.

```
from LispList import *
def lcopy(lst):
    """Return a copy, in new storage, of lst."""
    if lst.isEmpty():
        return Nil()
    else:
        return Cons(lst.first(), lcopy(lst.tail()))
```

Figure 13: The function `lcopy` that copies a Lisp list.

Tests for `lcopy` are shown in Figure 14.

```
from LispList import *
from lcopy import *
def test_lcopy():
    """Tests for lcopy."""
    nil = Nil()
    assert lcopy(nil) == Nil()
    lst = Cons("every", Cons("day", nil))
    assert lcopy(lst) == lst
    assert lcopy(lcopy(lst)) == lst
```

Figure 14: Tests for `lcopy`.

A more useful example that follows the grammar for flat (Lisp) Lists is the `incAll(lst)` function, which takes a list `lst` all of whose elements are numbers, and returns a new list that is like `lst`, but with each number incremented. See Figure 15.

```
from LispList import *
def incAll(lst):
    """Requires lst is a list of numbers.
    Ensures result is a new list that is just like lst,
    but with each number incremented."""
    if lst.isEmpty():
        return Nil()
    else:
        return Cons(lst.first()+1, incAll(lst.tail()))
```

Figure 15: The function `incAll`.

Some simple tests for `incAll` are shown in Figure 16 on the following page.

```
# $Id: test_incAll.py,v 1.1 2017/02/05 05:02:43 leavens Exp $
from LispList import *
from incAll import *
def test_incAll():
    nil = Nil()
    lst = Cons(3, Cons(2, Cons(2, Cons(3, nil))))
    assert incAll(nil) == nil
    assert incAll(lst) == Cons(4, Cons(3, Cons(3, Cons(4, nil))))
```

Figure 16: Tests for `incAll`.

A generalization of this pattern is given in the function `lmap(f,lst)`, shown in Figure 17. This function takes a function `f` (which itself takes an argument of some type t and returns a value of some type s), and a Lisp list of elements of type t, `lst`, and returns a Lisp list of elements of type s, which is the result of applying `f` to each element of `lst`, is a paradigmatic example of following the grammar for flat lists.

```
from LispList import *
def lmap(f, lst):
    if lst.isEmpty():
        return Nil()  # or return lst
    else:
        return Cons(f(lst.first()), lmap(f, lst.tail()))
```

Figure 17: The `lmap` function for flat (Lisp) lists.

The way in which functions are passed to `lmap` is shown in Figure 18. Note that the assignment `inc = (lambda i: i+1)` is a shorthand way to define a function that could be written

```
def inc(i):
    return i+1
```

The **lambda** expression creates a function (without a name) that behaves like `inc` above.

```
# $Id: test_lmap.py,v 1.2 2017/02/05 05:02:43 leavens Exp $
from LispList import *
from lmap import *
def test_lmap():
    nil = Nil()
    to10 = Cons(0, Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, \
                Cons(6, Cons(7, Cons(8, Cons(9, nil))))))))))
    from1to10 = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, \
                     Cons(6, Cons(7, Cons(8, Cons(9, Cons(10, nil))))))))))
    odds = Cons(1, Cons(3, Cons(5, Cons(7, Cons(9, nil)))))
    evens = Cons(0, Cons(2, Cons(4, Cons(6, Cons(8, nil)))))
    inc = (lambda i: i+1)
    assert lmap(inc, to10) == from1to10
    assert lmap(inc, evens) == odds
    double = (lambda n: 2*n)
    assert lmap(double, evens) == Cons(0, Cons(4, Cons(8, Cons(12, Cons(16, nil)))))
    assert lmap(double, lmap(double, odds)) == Cons(4, Cons(12, Cons(20, Cons(28, Cons(36, nil)))))
    assert lmap(inc, lmap(double, evens)) == Cons(1, Cons(5, Cons(9, Cons(13, Cons(17, nil)))))
```

Figure 18: Tests for `lmap`, showing how it is used.

5.2 SumList Example

Students sometimes overgeneralize the pattern of recursion over flat lists and think that always returning `Nil()` in the base case is the right thing to do. But this example shows that this is not always right. The example is a function `sumList(lst)` that takes a list of numbers, `lst`, and returns the total of all the numbers in that list, added together. Note how the code in Figure 19 follows the grammar for flat (Lisp) lists.

```
# $Id: sumList.py,v 1.1 2017/02/05 22:21:24 leavens Exp $
from LispList import *
def sumList(lst):
    """Return the sum of the Lisp list of numbers, lst."""
    if lst.isEmpty():
        return 0
    else:
        return lst.first() + sumList(lst.tail())
```

Figure 19: The function `sumList` that follows the grammar for flat (Lisp) lists.

Tests for `sumList` are found in Figure 20 on the following page.

```
# $Id: test_sumList.py,v 1.1 2017/02/05 22:21:24 leavens Exp $
from LispList import *
from sumList import *
def test_sumList():
    """Tests for sumList."""
    nil = Nil()
    assert sumList(nil) == 0
    assert sumList(Cons(1, Cons(10, nil))) == 11
    assert sumList(Cons(32, Cons(23, Cons(100, nil)))) == 155
    assert sumList(Cons(1, Cons(1, Cons(1, Cons(1, nil))))) == 4
```

Figure 20: Tests for `sumList`.

5.3 Take Example

Sometimes integers and lists jointly control a recursion. This is demonstrated by the function `take(lst, n)` in Figure 21. This function returns the first `n` elements of `lst`. In this example, in the recursive calls the list argument gets smaller and the argument `n` also gets smaller. This is an example of simultaneous recursion, as the function is recursing on both the list and the number arguments. Note that the base case test checks to see if either the list has run out or if the number has reached 0, and either of these stops the recursion.

```
# $Id: take.py,v 1.1 2017/02/05 22:21:24 leavens Exp $
from LispList import *
def take(lst, n):
    """Return the first n elements of Lisp list lst as a Lisp list."""
    if lst.isEmpty() or n <= 0:
        return Nil()
    else:
        return Cons(lst.first(), take(lst.tail(), n-1))
```

Figure 21: The function `take` that simultaneously recurses on a list and an integer.

Tests for `take` are found in Figure 22 on the following page. Note that these tests use a helping function, `fromTo`, that recurses on its (first) integer argument and produces a list. The figure also contains tests for `fromTo`.

```
# $Id: test_take.py,v 1.1 2017/02/05 22:21:24 leavens Exp $
from LispList import *
from take import *
def test_take():
    """Testing for the take function."""
    to10 = fromTo(1,10)
    to100 = fromTo(1, 100)
    assert take(to10, 5) == fromTo(1,5)
    assert take(to100, 23) == fromTo(1, 23)

def fromTo(start, end):
    """Return a list of the form Cons(start, Cons(start+1, ..., Cons(end, Nil()))."""
    if start > end:
        return Nil()
    else:
        return Cons(start, fromTo(start+1, end))

def test_fromTo():
    """Testing for fromTo()."""
    assert fromTo(1, 3) == Cons(1, Cons(2, Cons(3, Nil())))
    assert fromTo(4,4) == Cons(4, Nil())
    assert fromTo(0,5) == Cons(0, Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil()))))))
```

Figure 22: Tests for `take`, and `fromTo`, with code for a helping function `fromTo`.

5.3.1 ExtractNames Exercise

Which, if any, of the following is a correct outline for a function `extractNames(records)`, that takes a Lisp list of pairs of names and addresses, written `("Jane Doe", "1 Elm St.")` in Python, with the name first in the pair, 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(rec)` returns the name from a pair of a name and an address.

1. ```
   def extractNames(records):
       return Cons(name(records.first()), extractNames(records.tail()))
   ```

2. ```
   def extractNames(records):
       if records.isEmpty():
           return Nil()
       else:
           return Cons(name(records.first()), extractNames(records.tail()))
   ```

3. ```
   def extractNames(records):
       if records.isEmpty():
           return records
       else:
           return Cons(name(records.first()), extractNames(records.tail()))
   ```

4. ```
   def extractNames(records):
       if records.isEmpty():
           return records
       else:
           return Cons(name(records.first()), records.tail())
   ```

5. ```
   def extractNames(records):
       if records == []:
           return []
       else:
           return [name(records[0])] + extractNames(records[1:])
   ```

6. ```
   from lmap import *
   def extractNames(records):
       return lmap(name, records)
   ```

Answer: 2, 3, 6. In 6, the `lmap` function is following the grammar on behalf of `extractNames`. Note that 5 follows the grammar of Python lists, not the Lisp lists that are called for in the problem.

5.3.2 DeleteListing Exercise

Which, if any, of the following is a correct outline for a function `deleteListing(name,records)` that takes a string, `name`, and a Lisp list, `records` of pairs of names and addresses, and that follows the grammar for flat lists? List all that have a correct outline for recursion over flat lists.

(Again we assume that there is a function `name` the returns a name from a person record.)

1. ```
   def deleteListing(nm, records):
       if records.isEmpty():
           return Nil()
       else:
   ```
   ```

```
        fst = records.first()
        if name(fst) == nm:
            return deleteListing(nm, records.tail())
        else:
            return Cons(fst, deleteListing(nm, records.tail()))
```

2.
```
def deleteListing(nm, records):
    if records.isEmpty():
        return Nil()
    else:
        fst = records.first()
        return deleteListing(nm, records.tail()) \
                if name(fst) == nm \
                   else Cons(fst, deleteListing(nm, records.tail()))
```

3.
```
def deleteListing(nm, records):
    fst = records.first()
    if name(fst) == nm:
        return deleteListing(nm, records.tail())
    else:
        return Cons(fst, deleteListing(nm, records.tail()))
```

4.
```
def deleteListing(nm, records):
    fst = records.first()
    return Cons(fst, deleteListing(nm, records.tail()))
```

5.
```
def deleteListing(nm, records):
    fst = records.first()
    return deleteListing(nm, Cons(fst, records.tail()))
```

6.
```
def deleteListing(nm, records):
    ret = []
    for (n,a) in records:
        if nm != a:
            ret = ret.append([(n,a)]
    return ret
```

Answer: 1 and 2 (which uses an if expression) follow the grammar. Note that 3 and 4 have no base case. 6 doesn't follow the grammar for Lisp lists.

5.4 Binary Trees

For purposes of this paper, a "binary tree" is an instance of the class `BinTree` shown in Figure 23 on the next page. This set of classes corresponds to the following grammar, where t is some type.

⟨BinTree(t)⟩ ::= `EmptyTree()` | `Branch(`⟨t⟩`,` ⟨BinTree(t)⟩`,` ⟨BinTree(t)⟩`)`

The binary tree grammar has only one nonterminal, but has more recursion 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.

5.4.1 Example

An example function that follows the above grammar is `doubleTree(tr)`, which takes a binary tree of numbers and returns a binary tree of numbers; it is shown in Figure 24 on page 24. Notice that in the else case the function makes two recursive calls, one on the left subtree and one on the right subtree.

Tests for `doubleTree` are shown in Figure 25 on page 24.

```
# $Id: BinTree.py,v 1.3 2017/02/05 22:21:24 leavens Exp $
import abc
class BinTree(abc.ABC):
    pass
class EmptyTree(BinTree):
    def __init__(self):
        """Initialize this empty tree."""
        pass
    def __eq__(self, tr):
        """Return True just when tr is also an empty tree."""
        return isinstance(tr, EmptyTree)
    def __repr__(self):
        """Return a string representing this EmptyTree instance."""
        return "EmptyTree()"
    def __str__(self):
        """Return a string showing the elements of self."""
        return "{}"
    def isEmpty(self):
        """Return whether this tree is empty."""
        return True

class Branch(BinTree):
    def __init__(self, value, left, right):
        """Initialize this Branch with the given value and with the given
        left and right subtrees."""
        self.val = value
        self.lft = left
        self.rght = right
    def __eq__(self, tr):
        """Return True just when self is structurally equivalent to tr."""
        return isinstance(tr, Branch) and tr.value() == self.value() \
                and tr.left() == self.left() and tr.right() == self.right()
    def __repr__(self):
        """Return a string representing this tree."""
        return "Branch(" + repr(self.value()) + ", " + repr(self.left()) \
            + ", " + repr(self.right()) + ")"
    def __str__(self):
        """Return a string showing the elements of self."""
        return "{value: " + str(self.value()) + ", left: " + str(self.left()) \
            + ", right: " + str(self.right()) + "}"
    def isEmpty(self):
        """Return whether this tree is empty."""
        return False
    def value(self):
        """Return the value in the root of self."""
        return self.val
    def left(self):
        """Return the left subtree of self."""
        return self.lft
    def right(self):
        """Return the right subtree of self."""
        return self.rght
```

Figure 23: Binary tree classes. `BinTree` is an abstract class, with subtypes `EmtpyTree` and `Branch`.

```
# $Id: doubleTree.py,v 1.1 2017/02/05 17:32:30 leavens Exp $
from BinTree import *
def doubleTree(tr):
    if tr.isEmpty():
        return EmptyTree()
    else:
        return Branch(2 * tr.value(), doubleTree(tr.left()), doubleTree(tr.right()))
```

Figure 24: The function `doubleTree`, which follows the grammar for binary trees.

```
# $Id: test_doubleTree.py,v 1.1 2017/02/05 17:33:23 leavens Exp $
from BinTree import *
from doubleTree import *
def test_doubleTree():
    et = EmptyTree()
    one = Branch(1, et, et)
    two = Branch(2, et, et)
    three = Branch(3, one, two)
    four = Branch(4, three, et)
    assert doubleTree(et) == et
    assert doubleTree(one) == two
    six = Branch(6, two, Branch(4, et, et))
    assert doubleTree(three) == six
    assert doubleTree(four) == Branch(8, six, et)
```

Figure 25: Testing for the function `doubleTree`.

5.4.2 SumTree Exercise

Which, if any, of the following is a correct outline, that follows the grammar for binary trees, for a function `sumTree, (tr)` that takes a binary tree of numbers, `tr`, as an argument, and returns the sum of all of the numbers in `tr`?

1.
```
def sumTree(tr):
    if tr.isEmpty():
        return 0
    else:
        return tr.value() +  sumTree(tr.left()) + sumTree(tr.right())
```

2.
```
def sumTree(tr):
    return tr.value() + sumTree(tr.left()) + sumTree(tr.right())
```

3.
```
def sumTree(tr):
    ret = 0
    for i in tr:
        ret += i
    return ret
```

4.
```
def sumTree(tr):
    if tr.isEmpty():
        return EmptyTree()
    else:
        return Branch(tr.value(), sumTree(tr.right()))
```

5.
```
def sumTree(tr):
    if tr.isEmpty():
        return Nil()
    else:
        return tr.first() + sumTree(tr.tail())
```

6.
```
def sumTree(tr):
    return sumTree(tr.left()) + sumTree(tr.right())
```

Answer: 1 is the only one that follows the grammar.

5.4.3 ReduceTree Exercise

Write a function, `reduceTree(tr, f, z)`, which for some types t and s, takes a binary tree whose elements have type t, `tr`, a function `f` (which takes 3 arguments, a value of type t, and two values of type s, and which returns a value of type s), and a value `z` of type s, and which returns a value of type s that satisfies the following equations, for all functions (of the type described above) `f`, values `z` of type s, values `v` of type t, and BinTree(t) values `tl` and `tr`:

```
    reduceTree(EmptyTree(),f,z)  =  z
reduceTree(Branch(v,tl,tr),f,z)  =  f(v, reduceTree(tl,f,z), reduceTree(tr,f,z))
```

Tests that demonstrate how `reduceTree` could be used are shown in the Figure 26 on the following page.

```
# $Id: test_reduceTree.py,v 1.1 2017/02/05 18:17:22 leavens Exp $
from BinTree import *
from reduceTree import *
def test_reduceTree():
    et = EmptyTree()
    one = Branch(1, et, et)
    two = Branch(2, et, et)
    three = Branch(3, one, two)
    four = Branch(4, three, et)
    six = Branch(6, two, Branch(4, et, et))
    # add3 adds its 3 arguments
    add3 = (lambda v, tlv, trv: v + tlv + trv)
    # mkBr is the same as the Branch() function
    mkBr = (lambda v, tlv, trv: Branch(v, tlv, trv))
    # sumTree adds all the numbers in a given tree of numbers
    sumTree = (lambda tr: reduceTree(tr, add3, 0))
    # copyTree makes a new copy of the given tree
    copyTree = (lambda tr: reduceTree(tr, mkBr, EmptyTree()))
    assert sumTree(et) == 0
    assert sumTree(one) == 1
    assert sumTree(three) == 6
    assert sumTree(four) == 10
    assert sumTree(six) == 12
    assert copyTree(six) == six
    assert copyTree(four) == four
    # the lambda below acts like the built-in max() function for 3 arguments
    assert reduceTree(six, (lambda v, tlv, trv: max(v,tlv,trv)), 0) == 6
```

Figure 26: Tests for `reduceTree`.

5.4.4 Design Your own BinTree Problem Exercise

Design another problem for the type `BinTree`. Give an English explanation, and some tests. Then solve your problem, first on paper, then on the computer.

5.5 Sales Data

The grammar for `SalesData` is shown in Figure 27. It has a single nonterminal, but two alternatives, each of which contains a Lisp list.

⟨SalesData⟩ ::= `Store(`⟨String⟩`,` ⟨LispList(int)⟩`)`
| `Group(`⟨String⟩`,` ⟨LispList(SalesData)⟩`)`

Figure 27: The grammar for `SalesData`.

The grammar for ⟨String⟩ is the same as for Python; the strings in this grammar represent names of stores or groups. The grammar corresponds to the class declarations given in Figure 28 on the following page. (Note that `SalesData` itself is an abstract base class.)

```
# $Id: SalesData.py,v 1.1 2017/02/05 22:21:24 leavens Exp $
import abc
class SalesData(abc.ABC):
    pass
class Store(SalesData):
    def __init__(self, address, amounts):
        """Requires address is a string, amounts is a Lisp list of ints.
        Effect: initialize self with address and amounts."""
        self.addr = address
        self.amts = amounts

    def __eq__(self, sd):
        """Return True just when sd is structurally equal to self."""
        return isinstance(sd, Store) and sd.address() == self.address() \
            and sd.amounts() == self.amounts()

    def __repr__(self):
        """Return a string representation of self."""
        return "Store(" + repr(self.address()) + ", " + repr(self.amounts()) + ")"

    def address(self):
        """Return the address of this store."""
        return self.addr

    def amounts(self):
        """Return a Lisp list of the amounts of sales from this store."""
        return self.amts

class Group(SalesData):
    def __init__(self, gname, members):
        """Requires gname is a string and members is a Lisp list of SalesData.
        Effect: initialize this group with the given name and members."""
        self.gnm = gname
        self.mbrs = members

    def __eq__(self, sd):
        """Return True just when sd is structurally equal to self."""
        return isinstance(sd, Group) and sd.gname() == self.gname() \
            and sd.members() == self.members()

    def __repr__(self):
        """Return a string representation of self."""
        return "Group(" + repr(self.gname()) + ", " + repr(self.members()) + ")"

    def gname(self):
        """Return the name of this group."""
        return self.gnm

    def members(self):
        """Return the Lisp list of sales data for this group."""
        return self.mbrs
```

Figure 28: The type `SalesData` and its subtypes `Store` and `Group`.

5.5.1 NormalizeSalesData Example

The grammar for sales data is interesting in that both of its alternatives contain a different kind of list. 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(sd)`, that takes a sales Data value `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 29 (which uses the code in Figure 30 on the following page) gives some examples of how this program is supposed to work.

```
# $Id: test_normalizeSalesData.py,v 1.1 2017/02/05 22:21:24 leavens Exp $
from normalizeSalesData import *
from normalizeSalesDataTesting import *
def test_normalizeSalesData():
    """Testing for normalizeSalesData."""
    normalizeSalesDataTesting(normalizeSalesData)
```

Figure 29: Testing for the function `normalizeSalesData`. These tests import from `normalizeSalesData` and then pass the function to the code in Figure 30 on the following page.

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 31 on page 30. 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? (Note, the code uses `take` from Figure 21 on page 19 as a helping function.)

Another way to write `normalizeSalesData()` correctly is to use the `lmap` function (from Figure 17 on page 17) to map `normalizeSalesData()` over the list of members in a group. This is shown in Figure 32 on page 30. This still follows the grammar, because `lmap` is following the grammar for lists of sales data values. Thus the code has the same execution pattern as in Figure 31 on page 30; the only difference is that the call to `lmap` is used instead of `normalizeList`.

```
# $Id: normalizeSalesDataTesting.py,v 1.1 2017/02/05 22:21:24 leavens Exp $
from SalesData import *
from normalizeSalesData import *
from LispList import *
def normalizeSalesDataTesting(fun):
    """Testing for fun, which is supposed to implement
    the specification for normalizeSalesData."""
    # some data...
    hundreds = Cons(101, Cons(102, Cons(103, Cons(104, Cons(105, Cons(106, Nil()))))))
    hundreds5 = Cons(101, Cons(102, Cons(103, Cons(104, Cons(105, Nil())))))
    ones = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Cons(7, Cons(8, Cons(9, Nil()))))))))))
    ones5 = Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil())))))
    ats = Store("3223 Alafaya Trail Rd.", hundreds)

    idr = Store("3502 International Dr.", ones)
    targetfl = Group("Target Florida", Cons(idr, Cons(ats, Nil())))
    powof2 = Cons(1, Cons(2, Cons(4, Cons(8, Nil()))))
    binary = Store("1024 Binary Ave.", powof2)
    primes = Cons(2, Cons(3, Cons(5, Cons(7, Cons(9, Cons(11, Nil()))))))
    primes5 = Cons(2, Cons(3, Cons(5, Cons(7, Cons(9, Nil())))))
    logical = Store("256 Logical Blvd.", primes)
    targetia = Group("Target Iowa", Cons(binary, Cons(logical, Nil())))
    target = Group("Target usa", Cons(targetfl, Cons(targetia, Nil())))
    # now for the tests...
    assert fun(ats) == Store("3223 ALAFAYA TRAIL RD.", hundreds5)
    assert fun(idr) == Store("3502 INTERNATIONAL DR.", ones5)
    nfl = fun(targetfl)
    assert nfl == Group("TARGET FLORIDA", Cons(fun(idr), \
                        Cons(fun(ats), Nil())))
    nia = fun(targetia)
    assert nia == Group("TARGET IOWA", Cons(fun(binary), \
                        Cons(fun(logical), Nil())))
    assert fun(target) == Group("TARGET USA", Cons(nfl, Cons(nia, Nil())))
```

Figure 30: Testing code that does the testing for `normalizeSalesData` passed in as the function argument `fun`. This allows these tests to be used for several different implementations of `normalizeSalesData` without repeating this code.

```
# $Id: normalizeSalesData.py,v 1.1 2017/02/05 22:21:24 leavens Exp $
from SalesData import *
from LispList import *
from take import take
def normalizeSalesData(sd):
    """Return a similar SalesData object as sd,
    but with each address or group name capitalized."""
    if isinstance(sd, Store):
        return Store(capitalize(sd.address()), take(sd.amounts(), 5))
    else: # it's a Group
        return Group(capitalize(sd.gname()), normalizeList(sd.members()))

def normalizeList(lst):
    """Requries: lst is a Lisp list of SalesData.
    Ensures: result is the same as lst but with each SalesData item normalized."""
    if lst.isEmpty():
        return Nil()
    else:
        return Cons(normalizeSalesData(lst.first()), \
                    normalizeList(lst.tail()))

def capitalize(s):
    """Return the string s with all lowercase characters made into uppercase."""
    return str.upper(s)
```

Figure 31: The function `normalizeSalesData`, which follows the Sales Data grammar.

```
# $Id: normalizeSalesData_lmap.py,v 1.1 2017/02/05 22:21:24 leavens Exp $
from SalesData import *
from lmap import lmap # mapping for Lisp lists
from take import take
def normalizeSalesData(sd):
    if isinstance(sd, Store):
        # str.upper converts a string to uppercase
        return Store(str.upper(sd.address()), take(sd.amounts(), 5))
    else: # it's a Group
        return Group(str.upper(sd.gname()), lmap(normalizeSalesData, sd.members()))
```

Figure 32: The function `normalizeSalesData`, which follows the Sales Data grammar. This implementation uses `lmap` from Figure 17 on page 17.

Now consider the implementation of `normalizeSalesData` shown in Figure 33, 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. Ask yourself this: how does the code make sure that it only calls methods on `sd` that are defined for by the class of `sd`? The code only works because it does dynamic type tests, using **`isinstance`**`()`, and because Python does not type check code before running it. Unlike the good versions of `normalizeSalesData` shown above, this bad version cannot be used as a pattern to follow in other (statically typed) programming languages.

```
# $Id: normalizeSalesDataBad.py,v 1.1 2017/02/05 22:21:24 leavens Exp $
from SalesData import *
from LispList import *
from take import take
def normalizeSalesData(sd):
    """Return a similar SalesData object as sd,
    but with each address or group name capitalized."""
    if isinstance(sd, Store):
        return Store(str.upper(sd.address()), take(sd.amounts(), 5))
    elif isinstance(sd, Group):
        return Group(capitalize(sd.gname()), normalizeSalesData(sd.members()))
    elif sd.isEmpty():
        return Nil()
    else:
        normalizeSalesData(sd.first(), normalizeSalesData(sd.tail()))
```

Figure 33: The function `NormalizeSalesDataBad`, which does not follow the Sales Data 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, CCF-0916715, 0916350, 1017262, 1017334, and 1228695. This version was supported in part by NSF grants CCF0916715 and CCF1017262.

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] Gary T. Leavens. Following the grammar with Haskell. Technical Report CS-TR-13-01, Dept. of EECS, University of Central Florida, Orlando, FL, 32816-2362, January 2013.

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

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