Following the Grammar

Gary T. Leavens

TR #05-02a
February 2005, Revised January 2006

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, Scheme.

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;

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

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Following the Grammar

Gary T. Leavens
Department of Computer Science, 229 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1041 USA
leavens@cs.iastate.edu

January 17, 2006

Abstract

This document explains what it means to "follow the grammar" for several different grammars. It is intended to be used in class that teach functional programming using Scheme, especially those used for teaching principles of programming languages. In such courses traversal is over abstract syntax tree is 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 grammar. This is important for working with programming languages [4], as they are described by 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, 8] 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 two non-terminals, ⟨sym-exp⟩ and ⟨s-list⟩.

```
⟨sym-exp⟩ ::= ⟨symbol⟩ | ⟨s-list⟩
⟨s-list⟩ ::= ( {⟨sym-exp⟩}* )
```

The nonterminal ⟨sym-exp⟩ has two alternatives (separated by the vertical bar). The production for ⟨s-list⟩ has a recursive call to ⟨sym-exp⟩. Due to the use of the Kleene star (*) in the production for ⟨s-list⟩, there really are two alternatives for ⟨s-list⟩, as can be seen by writing its production without the Kleene star, using the dotted-pair notation for lists, as follows.

```
⟨s-list⟩ ::= () | ( ⟨sym-exp⟩ .  ⟨s-list⟩ )
```

1.2 Definition of Following the Grammar

Following the grammar means making a set of procedures 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 procedures P. We say that P* follows the grammar G *if:*

1. *For every nonterminal, $\langle X \rangle$, in G, there is a procedure, pX, in P 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 procedure pX decides between the alternatives offered in $\langle X \rangle$'s grammatical productions, and there is (at least) one case in the body of pX 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 procedure pX has a call to procedure pY, where pY is the procedure in P that handles data described by the nonterminal $\langle Y \rangle$, and*
   (b) *each call from pX to pY passes to pY a part of the data pX received as an argument, namely that part described by $\langle Y \rangle$.*

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

We give many examples in the sections that follow.

One way to look at this is that following the grammar 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 procedure 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 procedures 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, with comments on the right side enclosed in quotations (“ and ”). In this grammar, all of the alternatives are base cases.

```
⟨temperature⟩ ::=
      hot                                   "hot ()"
    | warm                                  "warm ()"
    | cold                                  "cold ()"
```

2.1.1 Helping Procedures

The comments in the grammar are designed to aid the reader in remembering the names of the various helping procedures. The comment in each alternative gives a *production name*, found to the left of the parentheses in the comment. These are `hot`, `warm`, and `cold` in the grammar above. These production names are used to name the helping procedures.

For each alternative, there is a *constructor* procedure that creates an instance of the production. Each constructor has a name that is the production name for that alternative. Since there is no data stored in the alternatives in this grammar (nothing in the parentheses in the comment for each alternative production), these helping procedures take no arguments. The types of these constructors are shown below; these types use the name of the nonterminal, `temperature` as a type. (The type notation used is defined in a report by Leavens, Clifton, and Dorn [7].)

```
hot : (-> () temperature)
warm : (-> () temperature)
cold : (-> () temperature)
```

For example, evaluating the expression `(hot)` returns the internal representation for that temperature, which might be, for example, the symbol `hot`. For each alternative, there is also a test that determines if the data is an instance of that alternative's production. These are given names based on the production name as well.

```
hot? : (-> (temperature) boolean)
warm? : (-> (temperature) boolean)
cold? : (-> (temperature) boolean)
```

Helping procedures for this grammar are found in the course library file `temperature-mod.scm`, which can be used in code by writing a `require` special form, as in the following example.

2.1.2 Example

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

```
(require (lib "temperature-mod.scm" "lib342"))

(deftype select-outerwear (-> (temperature) symbol))
(define select-outerwear
  (lambda (temp)
     (cond
       ((hot? temp) 'none)
       ((warm? temp) 'wind-breaker)
       (else 'down-jacket))))
```

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

2.1.3 Possibly-Freezing? Exercise

Which of the following is a correct outline for a procedure

```
possibly-freezing? : (-> (temperature) boolean)
```

that follows the grammar for temperature?

1.
```
(define possibly-freezing?
  (lambda (temp)
```

```
        (cond
          ((null? temp) #f)
          ((cold? (car temp)) #t)
          (else (possibly-freezing? (cdr temp))))))
```

2.
```
(define possibly-freezing?
  (lambda (temp)
     (cond
       ((sym-exp-symbol? temp) (eq? 'cold (sym-exp->symbol temp)))
       (else (possibly-freezing-s-list? (sym-exp->s-list temp))))))
(define possibly-freezing-s-list?
  (lambda (slist)
    (if (null? slist)
        #f
        (or (possibly-freezing? (car slist))
            (possibly-freezing-s-list? (cdr slist))))))
```

3.
```
(define possibly-freezing?
  (lambda (temp)
     (cond
       ((cold? temp) #t)
       ((hot? temp) #f)
       (else #f))))
```

4.
```
(define possibly-freezing?
  (lambda (temp)
     (cond
       ((cold? temp) (possibly-freezing? temp)))))
```

Answer: 3.

2.2 Color Grammar Exercises

Consider another example with simple alternatives and no recursion:

⟨`color`⟩ ::= `red` | `yellow` | `green` | `blue`

Assuming you have helping procedures `red?`, `yellow?`, `green?`, and `blue?`, write the procedure:

```
equal-color? : (-> (color color) boolean)
```

that takes two colors and returns `#t` if they are the same, and `#f` otherwise.

3 Only Recursion, No Alternatives

Another kind of grammar is one that just has recursion, but no alternatives. (Instances of such grammars are tricky to create, since they appear to be infinitely long. However, it's useful to see such grammars for purposes of explaining the "follow the grammar" idea.)

3.1 Infinite Sequence Grammar

The following is an example of such a grammar, with comments on the right side enclosed in quotations (" and "). The comments are an aid to remembering the helping procedures found in the course library file `iseq-mod.scm`.

⟨`iseq`⟩ ::=
  ⟨`number`⟩ ⟨`iseq`⟩ "`iseq (head tail)`"

3.1.1 Helping Procedures

The helping procedures for this grammar are the following.

```
iseq : (-> (number (-> () iseq)) iseq)
iseq->head : (-> (iseq) number)
iseq->tail : (-> (iseq) iseq)
```

Again, it's also worthwhile to look at how the names of these helping procedures, and their types, relate to the comments in the grammar above. The constructor, `iseq` takes the head and a procedure that returns the tail, and returns a value of type `iseq`. Its name is thus the production's name, `iseq`. The `iseq` constructor is a bit unusual, because it takes a procedure as its second argument. This allows the creation of instances of this grammar. For example, the following creates an infinite sequence in which elements are the number 1.

```
(define ones (iseq 1 (lambda () ones)))
```

The second and third helpers are *observers*, which take instances of the production and extract information. These observers have names that are formed from the name of the production, `iseq`, a right arrow `->`, and one of the field names for the production. The *field names* describe the information in the production, and are found within the parentheses of the production's comment. In this case, there are two field names, `head` and `tail`, and hence two observer procedures, `iseq->head` and `iseq->tail`. For example, given the definition above, `(iseq->head ones)` returns 1. Somewhat unusually, the return type of the `iseq->tail` procedure is not the same as the type of the second argument to the constructor; for example, `(iseq->tail ones)` has type `iseq`. This difference in types arises because of the need to delay evaluation of the second argument to the `iseq` constructor, which allows the creation of infinite sequences, like `ones` above.

3.1.2 Example

A procedure outline that follows the above grammar is the following example.

```
(require (lib "iseq-mod.scm" "lib342"))

(deftype iseq-map
  (-> ((-> (number) number) iseq) iseq))
(define iseq-map
  (lambda (f seq)
    (iseq (f (iseq->head seq))
          (lambda () (iseq-map f (iseq->tail seq))))))
```

Following the grammar in the `iseq-map` example means that the procedure does something with the head of the iseq and recurses on the tail of the iseq, just like the grammar has a head, which is 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. But other procedures might allow stopping for other reasons.

3.1.3 Any-Negative? Exercise

Which of the following is a correct outline of a procedure

```
any-negative? : (-> (iseq) boolean)
```

that follows the grammar for iseq?

1.
```
(define any-negative?
  (lambda (seq)
     (cond
```

```
        ((null? seq) #f)
        ((negative? (car seq)) #t)
        (else (any-negative? (cdr seq))))))
```

2.
```
(define any-negative?
  (lambda (seq)
     (cond
       ((negative? (iseq->head seq)) #t)
       (else (any-negative? (iseq->tail seq))))))
```

3.
```
(define any-negative?
  (lambda (seq)
     (cond
       ((sym-exp-symbol? seq) (eq? 'negative (sym-exp->symbol seq)))
       (else (any-negative-s-list? (sym-exp->s-list seq))))))
(define any-negative-s-list?
  (lambda (slist)
    (and (not (null? slist))
         (or (any-negative? (car slist))
             (any-negative-s-list? (cdr slist))))))
```

4.
```
(define any-negative?
  (lambda (seq)
     (cond
       ((cold? seq) #f)
       ((warm? seq) #f)
       (else #t))))
```

Answer: 2.

3.1.4 Nth-Element Exercise

Write a procedure,

```
filter-iseq : (-> ((-> (number) boolean) 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 `#t`, in their original order.

4 Multiple Nonterminals

When the grammar has multiple nonterminals, there should be a procedure for each nonterminal in the grammar, and the recursive calls between these procedures 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 procedure for ⟨X⟩ to the procedure for ⟨Y⟩ that passes an instance of ⟨Y⟩ as an argument.

4.1 Phone Number Grammar

Consider the following example grammar, for phone numbers, with comments on the right side enclosed in quotations (“ and ”). The comments are an aid to remembering the helping procedures found in the course library file `phone-number-mod.scm`.

⟨phone-number⟩ ::=	
(⟨exchange⟩ . ⟨subscriber⟩)	“phone-number (exchange subscriber)”
⟨exchange⟩ ::= ⟨number⟩	“exchange (number)”
⟨subscriber⟩ ::= ⟨number⟩	“subscriber (number)”

```
(require (lib "phone-number-mod.scm" "lib342"))

(deftype valid-number? (-> (phone-number) boolean))
(define valid-number?
  (lambda (num)
    (let ((ex-num (phone-number->exchange num)))
      (and (valid-exchange? ex-num)
           (valid-subscriber? (lookup-exchange ex-num)
                              (phone-number->subscriber num))))))

(deftype valid-exchange? (-> (exchange) boolean))
(define valid-exchange?
  (lambda (ex-num)
     (member ex-num *known-exchanges*)))

(deftype valid-subscriber? (-> ((list-of subscriber) subscriber) boolean))
(define valid-subscriber?
  (lambda (exchanges sub-num)
     (member sub-num exchanges)))
```

Figure 1: The three procedures that check validity of phone numbers.

4.1.1 Helping Procedures

The helping procedures for the above grammar are found in the file `phone-number-mod.scm`. They have the following types.

```
phone-number : (-> (exchange subscriber) phone-number)
exchange : (-> (number) exchange)
subscriber : (-> (number) subscriber)

phone-number->exchange : (-> (phone-number) exchange)
phone-number->subscriber : (-> (phone-number) subscriber)
exchange->number : (-> (exchange) number)
subscriber->number : (-> (subscriber) number)
```

Which of these constructors and which are observers?

4.1.2 Example

To follow this grammar when writing a program like

```
valid-number? : (-> (phone-number) boolean)
```

one would structure the code into three procedures, one for each of the three nonterminals, as shown in Figure 1. Since the production for ⟨phone-number⟩ uses the nonterminals ⟨exchange⟩ and ⟨subscriber⟩, the procedure `valid-number?` calls the corresponding procedures for the other nonterminals, namely `valid-exchange?` and `valid-subscriber?`. Note that the arguments to these procedures are parts of the number described by the corresponding nonterminals, and are extracted from the number by helping procedures described above.

Figure 1 uses some names specific to its problem. The types of these names are as follows.

```
*known-exchanges* : (list-of exchange)
lookup-exchange : (-> (exchange) (list-of subscriber))
```

These data structures consider an "exchange" to be a group of subscribers; this may not correspond to reality for present-day phone systems.

4.1.3 Lookup-Name Exercise

Which of the following is a correct outline of a procedure

```
lookup-name : (-> (phone-number) string)
```

that follows the grammar for phone-number?

1.
```
(define lookup-name
  (lambda (num)
     (cond
       ((null? num) "none")
       (else (lookup-name (lookup-exchange (car num))
                          (cdr num))))))
```

2.
```
(define lookup-name
  (lambda (num)
     (if (zero? num)
         num
         (lookup-name (- num 1)))))
```

3.
```
(define lookup-name
  (lambda (num)
     (cond
       ((sym-exp-symbol? num) (eq? 'negative (sym-exp->symbol num)))
       (else (lookup-name-s-list? (sym-exp->s-list num))))))
(define lookup-name-s-list?
  (lambda (slist)
    (and (not (null? slist))
         (or (lookup-name (car slist))
             (lookup-name-s-list? (cdr slist))))))
```

4.
```
(define lookup-name
  (lambda (num)
    (lookup-subscriber (lookup-exchange (phone-number->exchange num))
                       (phone-number->subscriber num))))
(define lookup-exchange
  (lambda (ex-num)
     (get-subscribers-for ex-num)))
(define lookup-subscriber
  (lambda (exchange sub-num)
     (table-lookup exchange sub-num)))
```

Answer: 4.

4.1.4 To String Exercise

Write a procedure,

```
phone-number->string : (-> (phone-number) string)
```

that takes a phone number, `num`, and returns a string with the conventional formatting for the phone number (e.g., "555-1212").

4.2 Multiple Nonterminal Exercise

Suppose you have a grammar with 10 nonterminals, how many procedures would be contained in an outline that followed the 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 examples that involve all three features at once. The grammar for statements and expressions is a nice combination of these ideas.

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, which shows these alternatives clearly, uses dot notation. The following grammar, has comments on the right side enclosed in quotations (" and "). The comments are an aid to remembering the Scheme built-in procedures that work with (flat) lists. (However, the procedure names for lists are not derived in the usual way from the names in the comments; there is no zero-argument `null` procedure nor are there procedures named `cons->car` or `cons->cdr`.)

```
⟨list-of-T⟩ ::= ()                                  "null ()"
    | ( ⟨T⟩ .  ⟨list-of-T⟩ )                        "cons (car cdr)"
```

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

This 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.

5.1.1 Specialized Definition for Recursion over Flat Lists

Note that the grammar for flat lists does not have multiple nonterminals. Thus only one procedure is needed to follow the grammar when solving a problem involving recursion over flat lists.

Hence a procedure that takes a flat list as an argument and recurses over its structure "follows the grammar" for flat lists if:

1. If it has at least two cases,
   (a) the first of which asks if the list argument is null, and
   (b) and the other(s) handle the case where the list is not null.
2. There is no recursive call in the case that handles the null list, but there may be a recursive call in the case(s) that handle non-null lists.
3. Since the grammar recurses on the cdr of the list, in the recursive cases, any recursive calls pass the cdr of the list as an argument to the recursive call.

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

5.1.2 Example of Recursion over Flat Lists

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

```
(deftype map (forall (S T) (-> ((-> (S) (T)) (list-of S)) (list-of T))))
(define map
   (lambda (f ls)
      (cond
        ((null? ls) '())                    ; base case, no recursion
        (else (cons (f (car ls))            ; recursive case
                    (map f (cdr ls)))))))   ; note (cdr ls) as argument
```

5.1.3 Extract-Names Exercise

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

```
extract-names : (-> ((list-of person)) (list-of 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 procedure `get-name` is defined elsewhere.

1.
```
(define extract-names
  (lambda (listings)
     (cond
       ((null? listings) '())
       (else (extract-names (get-name (car listings))
                            (cdr listings))))))
```

2.
```
(define extract-names
  (lambda (listings)
     (cond
       ((null? listings) '())
       (else (cons (get-name (car listings))
                   (extract-names (cdr listings)))))))
```

3.
```
(define extract-names
  (lambda (listings)
     (if (null? listings)
         '()
         (cons (get-name (car listings))
               (extract-names (cdr listings))))))
```

4.
```
(define extract-names
  (lambda (listings)
     (if (null? listings)
         '()
         (begin (cdr listings)
                (cons (get-name (car listings))
                      (extract-names listings))))))
```

5.
```
(define extract-names
  (lambda (listings)
     (cons (get-name (car listings))
           (extract-names (cdr listings)))))
```

6.
```
(define extract-names
  (lambda (listings)
     (if (null? listings)
         listings
         (cons (get-name (car listings))
               (extract-names (cdr listings))))))
```

Answer: 2, 3, 6.

5.1.4 Delete-Listing Exercise

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

```
delete-listing : (-> ((list-of person)) (list-of person))
```

that follows the grammar for flat lists? List all that have a correct outline for recursion over flat lists. Assume that the procedure `get-name` is defined elsewhere.

1.
```
(define delete-listing
  (lambda (name listings)
     (cond
       ((null? listings) '())
       (else (cond
               ((equal? name (get-name (car listings)))
                (delete-listing name (cdr listings)))
               (else (cons (car listings)
                           (delete-listing name (cdr listings)))))))))
```

2.
```
(define delete-listing
  (lambda (name listings)
     (cond
       ((null? listings) '())
       ((equal? name (get-name (car listings)))
        (delete-listing name (cdr listings)))
       (else (cons (car listings)
                   (delete-listing name (cdr listings)))))))
```

3.
```
(define delete-listing
  (lambda (name listings)
     (if (null? listings)
         '()
       (if (equal? name (get-name (car listings)))
           (delete-listing name (cdr listings))
           (cons (car listings)
                 (delete-listing name (cdr listings)))))))
```

4.
```
(define delete-listing
  (lambda (name listings)
     (if (null? listings)
         '()
       (append
        (if (equal? name (get-name (car listings)))
            '()
            (list (car listings)))
        (delete-listing name (cdr listings))))))
```

5.
```
(define delete-listing
  (lambda (name listings)
     (cond
       ((equal? name (get-name (car listings)))
        (delete-listing name (cdr listings)))
       (else (cons (car listings)
                   (delete-listing name (cdr listings)))))))
```

6.
```
(define delete-listing
  (lambda (name listings)
     (cond
       ((null? listings) listings)
       (else (cond
               ((equal? name (get-name (car listings)))
                (delete-listing name (cdr listings)))
               (else (cons (car listings)
                           (delete-listing name (cdr listings))))))))
```

7.
```
(define delete-listing
  (lambda (name listings)
     (cond
       ((null? listings) '())
       (else (cond
               ((equal? name (get-name (car listings)))
                (delete-listing name (cdr listings)))
               (else (cons (car listings)
                           (delete-listing name listings))))))))
```

Answer: 1, 2, 3, 4, 6.

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 procedures that follow its grammar have more recursive calls than procedures that follow the grammar for flat lists. In the following grammar, comments on the right side enclosed in quotations (" and ") are an aid to remembering the helping procedures found in the course library file `window-layout-mod.scm`.

```
⟨window-layout⟩ ::=
      (window ⟨symbol⟩ ⟨number⟩ ⟨number⟩)          "window (name width height)"
    | (horizontal {⟨window-layout⟩}*)             "horizontal (subwindows)"
    | (vertical {⟨window-layout⟩}*)               "vertical (subwindows)"
```

In the above grammar, the nonterminals ⟨number⟩ and ⟨symbol⟩ have the same syntax as in Scheme.

5.2.1 Helping Procedures

The helping procedures for this grammar are found in the library file `window-layout-mod.scm`. They have the types shown in Figure 2.

In addition to the helping procedures listed in Figure 2, the file `window-layout-mod.scm` also has a parsing procedure

```
window? : (-> (window-layout) boolean)
horizontal? : (-> (window-layout) boolean)
vertical? : (-> (window-layout) boolean)

window : (-> (symbol number number) window-layout)
horizontal : (-> ((list-of window-layout)) window-layout)
vertical : (-> ((list-of window-layout)) window-layout)

window->name : (-> (window-layout) symbol)
window->width : (-> (window-layout) number)
window->height : (-> (window-layout) number)
horizontal->subwindows : (-> (window-layout) (list-of window-layout))
vertical->subwindows : (-> (window-layout) (list-of window-layout))
```

Figure 2: Helping procedures for the window-layout grammar, from the course library file `window-layout-mod.scm`.

```
parse-window-layout : (-> (datum) window-layout)
```

This parsing procedure takes a Scheme datum, and checks that it conforms to the grammar for ⟨window-layout⟩. We always assume that inputs to programs over grammars have been checked by such a parser. This allows you, as a programmer to assume that a procedure argument of type `window-layout` conforms to the grammar, and avoids the need to write error checking code.

Furthermore, a parser can convert the input data from an external format into some internal data structure. In compiling, the external syntax is often called the *concrete syntax* for the language. The internal representations are called *abstract syntax trees*. Such an internal representation is often more efficient to access than the external syntax, and it captures the information in an unambiguous way that, nevertheless, avoids extraneous information (such as punctuation). Viewed in this way, the output of a parser is always equivalent to some combination of calls to the constructor operations of the grammar, as shown in Figure 3,

Such parsing procedures are often used to state problems, because they allow the input to be abbreviated, and because they avoid errors in problem inputs. However, parsing is assumed to be an expensive operation, and so is normally avoided in writing code to manipulate or directly create abstract syntax trees.

5.2.2 Example

An example procedure that follows the above grammar is shown in Figure 4. This example is coded using Scheme's `map` procedure. Doing that avoids having to write out a separate procedure for recursing over a list of window-layouts. Writing out a separate procedure (or two different procedures, in general) to recurse over the two lists implicit in the use of the Kleene star (*) in the grammar would be perfectly fine, however. These helping procedure(s) would, of course, follow the grammar for flat lists.

In the calls to `map` in Figure 4, the procedure argument is `double-size` itself. This is equivalent to (but slightly faster than) passing the procedure `(lambda (wl) (double-size wl)`. However, in other problems using a `lambda` (or a named helping procedure) may be needed, because the procedure argument to `map` must be a procedure that takes exactly one argument. For an example, consider writing a procedure `add-n-to-size`, such that `(add-n-to-size 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 such a problem it would be convenient to use `(lambda (wl) (add-n-to-size n wl))`, as the procedure argument to `map`.

5.2.3 Total-Width Exercise

Write a procedure,

```
total-width : (-> (window-layout) number)
```

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

$$\texttt{(window}\ S\ N_1\ N_2\texttt{)}$$

is N_1. The width of a ⟨window-layout⟩ of the form

$$\texttt{(horizontal}\ W_1\ W_2\ \ldots\ W_m\texttt{)}$$

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

$$\texttt{(vertical}\ W_1\ W_2\ \ldots\ W_m\texttt{)}$$

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.

The following are examples. For brevity these use the parsing procedure described above.

```
(parse-window-layout '(window olympics 50 33))
  = (window 'olympics 50 33)

(parse-window-layout '(horizontal (window olympics 80 33)
                                  (window local-news 20 10)))
  = (horizontal (list (window 'olympics 80 33)
                      (window 'local-news 20 10)))

(parse-window-layout '(vertical (window olympics 80 33)
                                (window local-news 20 10)))
  = (vertical (list (window 'olympics 80 33)
                    (window 'local-news 20 10)))
```

Figure 3: The effect of the `parse-window-layout` procedure. These are equations between Scheme expressions.

```
(require (lib "window-layout-mod.scm" "lib342"))

(deftype double-size (-> (window-layout) window-layout))
(define double-size
  (lambda (layout)
    ;; ENSURES: result is the same as layout, but with all sizes doubled
    (cond
       ((window? layout) (window (window->name layout)
                                 (* 2 (window->width layout))
                                 (* 2 (window->height layout))))
       ((horizontal? layout)
        (horizontal (map double-size
                         (horizontal->subwindows layout))))
       (else ;; vertical case
         (vertical (map double-size
                        (vertical->subwindows layout)))))))
```

Figure 4: The procedure `double-size`, which follows the grammar for window layouts.

```
(total-width (parse-window-layout '(window olympics 50 33))) ==> 50
(total-width
   (parse-window-layout '(horizontal ))) ==> 0
(total-width
   (parse-window-layout '(vertical ))) ==> 0
(total-width
   (parse-window-layout '(horizontal (window olympics 80 33)
                                     (window local-news 20 10)))) ==> 100
(total-width
   (parse-window-layout '(vertical (window olympics 80 33)
                                   (window local-news 20 10)))) ==> 80
(total-width
   (parse-window-layout '(vertical (window star-trek 40 100)
                                   (window olympics 80 33)
                                   (window local-news 20 10)))) ==> 80
(total-width
   (parse-window-layout
      '(horizontal
          (vertical (window tempest 200 100)
                    (window othello 200 77)
                    (window hamlet 1000 600))
          (horizontal (window baseball 50 40)
                      (window track 100 60)
                      (window equestrian 70 30))
          (vertical (window star-trek 40 100)
                    (window olympics 80 33)
                    (window local-news 20 10))))) ==> 1300
```

Helpers for this grammar can be loaded by using the following `require` form `(require (lib "window-layout-mod.scm" "lib342"))`.

Feel free to use Scheme's `map` and `max` procedures. But beware that `max` requires at least one argument.

You can test your code using `(test-ex "total-width")`.

5.2.4 Design Your own Window-Layout Problem Exercise

Design another problem over window-layouts. Give an English explanation, the deftype, 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 Boolean expressions

The grammar below, for boolean expressions has multiple nonterminals, but does not have mutual recursion among the nonterminals.

⟨bexp⟩ ::=	
⟨varref⟩	"var-exp (varref)"
| (and ⟨bexp⟩ ⟨bexp⟩)	"and-exp (left right)"
| (or ⟨bexp⟩ ⟨bexp⟩)	"or-exp (left right)"
| (not ⟨bexp⟩)	"not-exp (arg)"
⟨varref⟩ ::=	
P	"P ()"
| Q	"Q ()"

5.3.1 Helping procedures

Helping procedures for the above grammar are found in the course library file `bexp-mod.scm`. They have the types shown in Figure 5.

For this grammar there is also a parsing procedures in `bexp-mod.scm`. As explained in Section 5.2.1, the parsing procedure checks for errors in inputs, which allows you to assume that the input conforms to the grammar. It may also convert the concrete syntax of a ⟨bexp⟩ into abstract syntax trees. The effect of `parse-bexp` is illustrated below.

```
(parse-bexp 'P) = (var-exp (P))
(parse-bexp 'Q) = (var-exp (Q))
(parse-bexp '(and P Q)) = (and-exp (var-exp (P)) (var-exp (Q)))
(parse-bexp '(or P Q)) = (or-exp (var-exp (P)) (var-exp (Q)))
(parse-bexp '(not (or P Q)) = (not-exp (or-exp (var-exp (P)) (var-exp (Q))))
```

5.3.2 Example

An example using this grammar is shown in Figure 6.

The example in Figure 6 shows the general form of programs over the bexp grammar. However, in this particular example, the `negate-varref` procedure isn't really needed, as it just returns its argument unchanged. Hence it could be left out entirely for this particular problem. If that were done, then the `var-exp` case of `negate-bexp` would be written as follows instead of the (unoptimized) coding shown in Figure 6.

```
((var-exp? be)
 (not-exp be))
```

5.3.3 Beval Exercise

Write a procedure

```
beval : (-> (bexp boolean boolean) boolean)
```

that takes 3 arguments: `bexp`, which is a ⟨bexp⟩, `p-val`, which is the value for `P`, and `q-val`, which is the value for `Q`. It evaluates the expression ⟨bexp⟩ for those values. (Do this without using Scheme's `eval` function.)

The following are examples.

```
(beval (parse-bexp 'P) #t #f) ==> #t
(beval (parse-bexp 'Q) #t #f) ==> #f
(beval (parse-bexp '(not P)) #t #f) ==> #f
(beval (parse-bexp '(or (not P) Q)) #t #f) ==> #f
(beval (parse-bexp '(and P (or Q (not Q)))) #t #f) ==> #t
(beval (parse-bexp '(and (or P P) (or (or P P) (or Q Q)))) #f #t)
   ==> #f
(beval (parse-bexp '(and (or P P) (or (or P P) (or Q Q)))) #t #f)
   ==> #t
(beval (parse-bexp '(or (not (and P P))
                        (and (not (not (and P P)))
                             (not (and P Q))))) #t #f)
   ==> #t
```

Helpers for this grammar can be loaded using `(require (lib "bexp-mod.scm" "lib342"))`. You can test your code using `(test-ex "beval")`.

```
var-exp? : (-> (bexp) boolean)
and-exp? : (-> (bexp) boolean)
or-exp? : (-> (bexp) boolean)
not-exp? : (-> (bexp) boolean)
P? : (-> (varref) boolean)
Q? : (-> (varref) boolean)

var-exp : (-> (varref) bexp)
and-exp : (-> (bexp bexp) bexp)
or-exp : (-> (bexp bexp) bexp)
not-exp : (-> (bexp) bexp)
P : (-> () varref)
Q : (-> () varref)

var-exp->varref : (-> (bexp) varref)
and-exp->left : (-> (bexp) bexp)
and-exp->right : (-> (bexp) bexp)
or-exp->left : (-> (bexp) bexp)
or-exp->right : (-> (bexp) bexp)
not-exp->arg : (-> (bexp) bexp)
```

Figure 5: Types of the helping procedures in `bexp-mod.scm`.

```
(require (lib "bexp-mod.scm" "lib342"))

(deftype negate-bexp (-> (bexp) bexp))
(define negate-bexp
  (lambda (be)
    (cond
      ((var-exp? be)
       (not-exp (var-exp (negate-varref (var-exp->varref be)))))
      ((and-exp? be)
        (or-exp (negate-bexp (and-exp->left be))
                (negate-bexp (and-exp->right be))))
      ((or-exp? be)
       (and-exp (negate-bexp (or-exp->left be))
                (negate-bexp (or-exp->right be))))
      (else ;; for the not-exp we take advantage of double negation
        (not-exp->arg be)))))

(deftype negate-varref (-> (varref) varref))
(define negate-varref
  (lambda (vr)
     vr))
```

Figure 6: The procedure `negate-bexp`, which follows the bexp grammar.

5.3.4 Bswap Exercise

Using the bexp grammar and helpers as in the previous problem, write a procedure

```
bswap : (-> (bexp) bexp)
```

that takes a ⟨bexp⟩ and returns a ⟨bexp⟩ with the same shape, but with every `P` changed to `Q` and every `Q` changed to `P`. For example, the following equations hold (note these are not evaluations, but are equations between Scheme terms):

```
(bswap (var-exp (P))) = (var-exp (Q))
(bswap (var-exp (Q))) = (var-exp (P))
(bswap (not-exp (var-exp (P)))) = (not-exp (var-exp (Q)))
(bswap (or-exp (not-exp (var-exp (P))) (var-exp (Q))))
     = (or-exp (not-exp (var-exp (Q))) (var-exp (P)))
(bswap (and-exp (var-exp (P))
                (or-exp (var-exp (Q)) (not-exp (var-exp (Q))))))
     = (and-exp (var-exp (Q))
                 (or-exp (var-exp (P)) (not-exp (var-exp (P)))))
```

Helpers for this grammar can be loaded using `(require (lib "bexp-mod.scm" "lib342"))`. You can test your code using `(test-ex "bswap")`.

5.4 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 ⟨identifier⟩ has the same syntax as a Scheme ⟨symbol⟩.

```
⟨statement⟩ ::=
      ⟨expression⟩                                  "exp-stmt (exp)"
    | (set!  ⟨identifier⟩ ⟨expression⟩)             "set-stmt (id exp)"
⟨expression⟩ ::=
      ⟨identifier⟩                                  "var-exp (id)"
    | ⟨number⟩                                      "num-exp (num)"
    | (begin {⟨statement⟩}* ⟨expression⟩)           "begin-exp (stmts exp)"
```

5.4.1 Helping Procedures

The file `statement-expression.scm` contains helping procedures for the above grammar. The types of these helpers are shown in Figure 7.

The file `statement-expression.scm` also contains a parsing procedure, `parse-statement`. See Section 5.2.1 for an explanation of what such a parsing procedure does in general.

5.4.2 Examples

An example that follows this grammar is shown in Figure 8. It's instructive to draw arrows from each use of `stmt-add1` and `exp-add1`, in the figure, to where these names are define, and to draw arrows form the uses of the corresponding nonterminals in the grammar to where they are defined. In what sense do the recursion patterns match?

Note that, in Figure 8 the recursive call from `exp-add1` to `stmt-add1` occurs in the case for `begin-exp`, following the grammar; this recursive call is inside a use of `map`, since the recursion in the grammar is inside a Kleene star. It would be fine to use a separate helping procedure 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 var-exp case of `exp-add1` could be simplified to the following.

```
((var-exp? exp) exp)
```

The form used in Figure 8 is more complex and slower, but shows the general pattern more clearly.

```
exp-stmt? : (-> (statement) boolean)
set-stmt? : (-> (statement) boolean)

var-exp? : (-> (expression) boolean)
num-exp? : (-> (expression) boolean)
begin-exp? : (-> (expression) boolean)

exp-stmt : (-> (expression) statement)
set-stmt : (-> (symbol expression) statement)

var-exp : (-> (symbol) expression)
num-exp : (-> (number) expression)
begin-exp : (-> ((list-of statement) expression) expression)

exp-stmt->exp : (-> (statement) expression)
set-stmt->id : (-> (statement) symbol)
set-stmt->exp : (-> (statement) expression)

var-exp->id : (-> (expression) symbol)
num-exp->num : (-> (expression) number)
begin-exp->stmts : (-> (expression) (list-of statement))
begin-exp->exp : (-> (expression) expression)
```

Figure 7: Types of the helping procedures in the course library for the statement and expression grammar.

```
(require (lib "statement-expression.scm" "lib342"))

(deftype stmt-add1 (-> (statement) statement))
(define stmt-add1
  (lambda (stmt)
    (cond
       ((exp-stmt? stmt) (exp-stmt (exp-add1 (exp-stmt->exp stmt))))
       (else ;; set-stmt
         (set-stmt (set-stmt->id stmt)
                   (exp-add1 (set-stmt->exp stmt)))))))

(deftype exp-add1 (-> (expression) expression))
(define exp-add1
  (lambda (exp)
    (cond
       ((var-exp? exp) (var-exp (var-exp->id exp)))
       ((num-exp? exp) (num-exp (+ 1 (num-exp->num exp))))
       (else ;; begin-exp
         (begin-exp (map stmt-add1 (begin-exp->stmts exp))
                    (exp-add1 (begin-exp->exp exp)))))))
```

Figure 8: The procedures `stmt-add1` and `exp-add1`. These are an example on the statement and expression grammar.

5.4.3 Subst-Identifier Exercise

Write a procedure

```
subst-identifier : (-> (symbol symbol statement) statement)
```

that takes two symbols, `new` and `old`, and a statement, `stmt`, and returns a statement that is just like `stmt`, except that all occurrences of `old` in `stmt` are replaced by `new`. The following are examples.

```
(subst-identifier 'new 'old (exp-stmt (var-exp 'old)))
                          = (exp-stmt (var-exp 'new))
(subst-identifier 'new 'old (exp-stmt (var-exp 'x)))
                          = (exp-stmt (var-exp 'x))
(subst-identifier 'x 'a (exp-stmt (var-exp 'a)))
                      = (exp-stmt (var-exp 'x))
(subst-identifier 'x 'a (set-stmt 'a (num-exp 3)))
                      = (set-stmt 'x (num-exp 3))
(subst-identifier 'x 'a
    (set-stmt 'a (begin (list (set-stmt 'a (num-exp 3))) (var-exp 'b))))
  = (set-stmt 'x (begin (list (set-stmt 'x (num-exp 3))) (var-exp 'b)))
(subst-identifier 'x 'a
    (parse-statement '(set! a (begin (set! a 3) (set! b a) b))))
  = (parse-statement '(set! x (begin (set! x 3) (set! b x) b)))
(subst-identifier 'x 'a
    (parse-statement '(begin a)))
  = (parse-statement '(begin x))
(subst-identifier 'x 'a
    (parse-statement '(set! a (begin (begin a)))))
  = (parse-statement '(set! x (begin (begin x))))
(subst-identifier 'x 'a
    (parse-statement
     '(begin (set! a (begin 3))
             (set! a (begin (set! a (begin 4)) a))
             5 6 a (begin a) 5 (begin (begin (begin b) a))
             (begin (set! b (begin a)) b))))
  = (parse-statement
     '(begin (set! x (begin 3))
             (set! x (begin (set! x (begin 4)) x))
             5 6 x (begin x) 5 (begin (begin (begin b) x))
             (begin (set! b (begin x)) b)))
```

Helpers for this grammar can be loaded using `(require (lib "statement-expression.scm" "lib342"))`.

You can test your code using `(test-ex "subst-identifier")`.

5.4.4 Subst-Identifier-Exp Exercise

In solving the previous problem, did you write a procedure

```
subst-identifier-exp : (-> (symbol symbol expression) expression)
```

as a helper? If not, follow the grammar by doing that. If so, test that also.

You can test your code using `(test-ex "subst-identifier-exp")`.

5.4.5 More Statement and Expression Exercises

Invent one or more other problem on the statement and expression grammar above. For example, try incrementing all the numbers found in a statement, or reversing the list of statements in every `begin` expression whose expression contains a ⟨num-exp⟩ with a number less than 3.

5.5 Other Grammars

Work some problems like those on the homework for the lambda+if-exp grammar. For example, substitute for all bound occurrences of an identifier in an expression.

Or invent your own grammar, and some problems on it.

Acknowledgments

Thanks to Brian Patterson and Daniel Patanroi for comments on drafts of this paper. This work was supported in part by NSF grants CCF-0428078, and CCF-0429567.

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, Curtis Clifton, and Brian Dorn. A type notation for Scheme. Technical Report 05-18a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, January 2006. Available by anonymous ftp from ftp.cs.iastate.edu.

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