Essentials of Programming Languages, Third Edition

by Daniel P. Friedman and Mitchell Wand

Chapter 3: Expressions

In this chapter, we study the binding and scoping of variables. We do this by presenting a sequence of small languages that illustrate these concepts. We write specifications for these languages, and implement them using interpreters, following the interpreter recipe from chapter 1. Our specifications and interpreters take a context argument, called the *environment*, which keeps track of the meaning of each variable in the expression being evaluated.

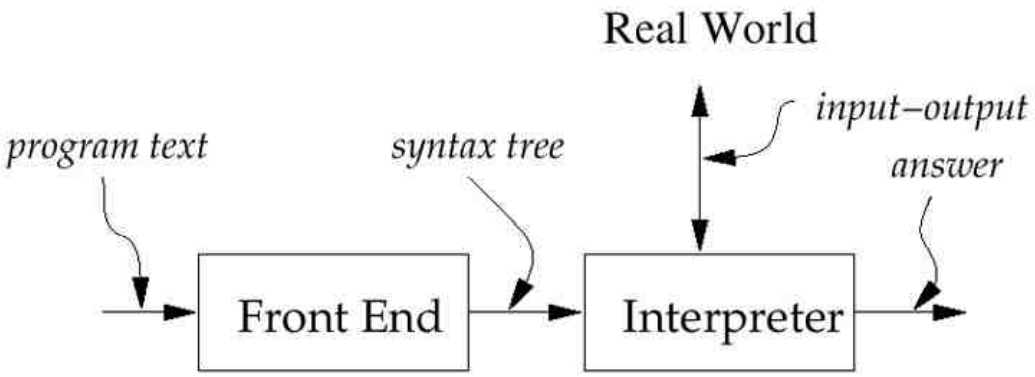
3.1 Specification and Implementation Strategy

Our specification will consist of assertions of the form

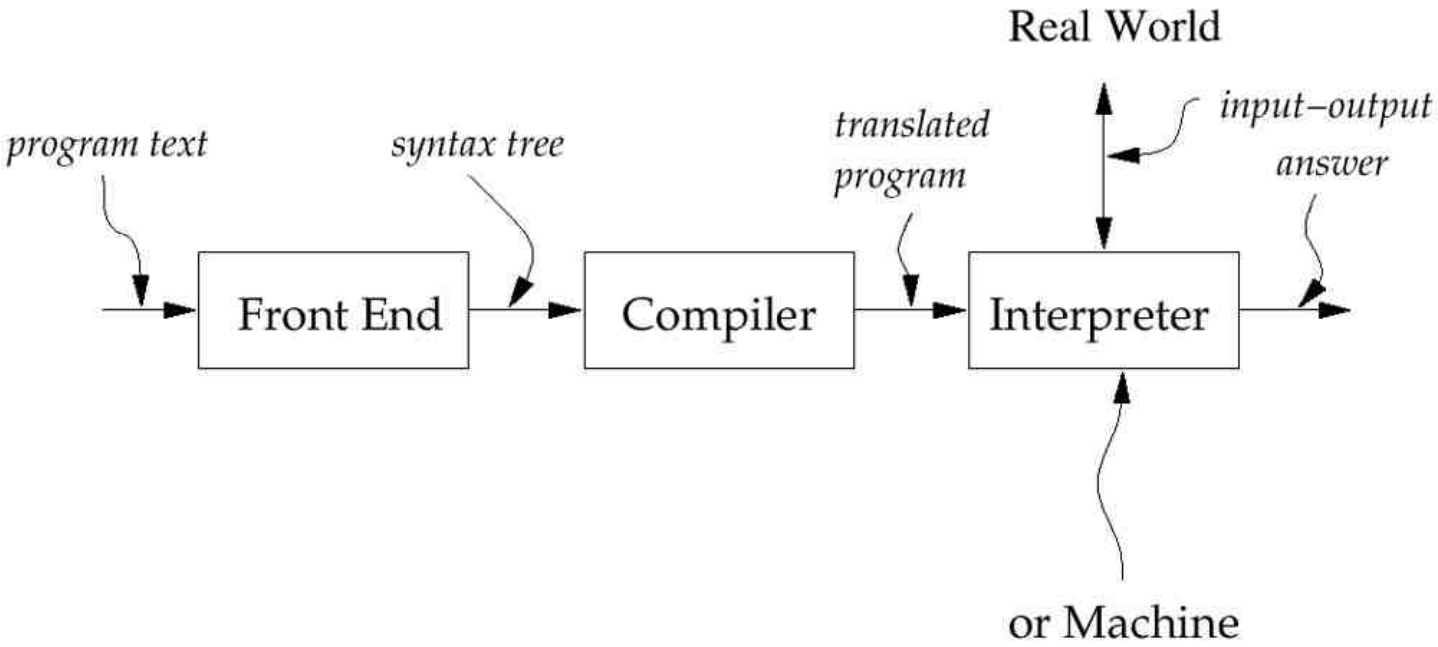
```
(value-of exp ρ) = val
```

meaning that the value of expression *exp* in environment *ρ* should be *val*. We write down rules of inference and equations, like those in chapter 1, that will enable us to derive such assertions. We use the rules and equations by hand to find the intended value of some expressions.

But our goal is to write a program that implements our language. The overall picture is shown in figure 3.1(a). We start with the text of the program written in the language we are implementing. This is called the *source language* or the *defined language*. Program text (a program in the source language) is passed through a front end that converts it to an abstract syntax tree. The syntax tree is then passed to the interpreter, which is a program that looks at a data structure and performs some actions that depend on its structure. Of course the interpreter is itself written in some language. We call that language the *implementation language* or the *defining language*. Most of our implementations will follow this pattern.



(a) Execution via interpreter



(b) Execution via Compiler

Figure 3.1: Block diagrams for a language-processing system

Another common organization is shown in figure 3.1(b). There the interpreter is replaced by a compiler, which translates the abstract syntax tree into a program in some other language (the *target language*), and that program is executed. That target language may be executed by an interpreter, as in figure 3.1(b), or it may be translated into some even lower-level language for execution.

Most often, the target language is a machine language, which is interpreted by a hardware machine. Yet another possibility is that the target machine is a special-purpose language that is simpler than the original and for which it is relatively simple to write an interpreter. This allows the program to be compiled once and then executed on many different hardware platforms. For historical reasons, such a target language is often called a *byte code*, and its interpreter is called a *virtual machine*.

A compiler is typically divided into two parts: an *analyzer* that attempts to deduce useful information about the program, and a *translator* that does the translation, possibly using information from the analyzer. Each of these phases may be specified either by rules of inference or a special-purpose specification language, and then implemented. We study some simple analyzers and translators in chapters 6 and 7.

No matter what implementation strategy we use, we need a *front end* that converts programs into abstract syntax trees. Because programs are just strings of characters, our front end needs to group these characters into meaningful units. This grouping is usually divided into two stages: *scanning* and *parsing*.

Scanning is the process of dividing the sequence of characters into words, numbers, punctuation, comments, and the like. These units are called *lexical items*, *lexemes*, or most often *tokens*. We refer to the way in which a program should be divided up into tokens as the *lexical specification* of the language. The scanner takes a sequence of characters and produces a sequence of tokens.

Parsing is the process of organizing the sequence of tokens into hierarchical syntactic structures such as expressions, statements, and blocks. This is like organizing (diagramming) a sentence into clauses. We refer to this as the *syntactic* or *grammatical* structure of the language. The parser takes a sequence of tokens from the scanner and produces an abstract syntax tree.

The standard approach to building a front end is to use a *parser generator*. A parser generator is a program that takes as input a lexical specification and a grammar, and produces as output a scanner and parser for them.

Parser generator systems are available for most major languages. If no parser generator is available, or none is suitable for the application, one can choose to build a scanner and parser by hand. This process is described in compiler textbooks. The parsing technology and associated grammars we use are designed for simplicity in the context of our very specialized needs.

Another approach is to ignore the details of the concrete syntax and to write our expressions as list structures, as we did for lambda-calculus expressions with the procedure parse-expression in section 2.5 and exercise 2.31.

3.2 LET: A Simple Language

We begin by specifying a very simple language, which we call LET, after its most interesting feature.

3.2.1 Specifying the Syntax

Figure 3.2 shows the syntax of our simple language. In this language, a program is just an expression. An expression is either an integer constant, a difference expression, a zero-test expression, a conditional expression, a variable, or a let expression.

Program ::= *Expression*

```
a-program (exp1)
```

Expression ::= *Number*

```
const-exp (num)
```

Expression ::= - (*Expression* , *Expression*)

```
diff-exp (exp1 exp2)
```

Expression ::= zero? (*Expression*)

```
zero?-exp (exp1)
```

Expression ::= if *Expression* then *Expression* else *Expression*

```
if-exp (exp1 exp2 exp3)
```

Expression ::= *Identifier*

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

Expression ::= let *Identifier* = *Expression* in *Expression*

```
let-exp (var exp1 body)
```

Figure 3.2: Syntax for the LET language

Here is a simple expression in this language and its representation as abstract syntax.

```
(scan&parse "-(55, -(x,11))")
#(struct:a-program
   #(struct:diff-exp
      #(struct:const-exp 55)
      #(struct:diff-exp
         #(struct:var-exp x)
         #(struct:const-exp 11))))
```

3.2.2 Specification of Values

An important part of the specification of any programming language is the set of values that the language manipulates. Each language has at least two such sets: the *expressed values* and the *denoted values*. The expressed values are the possible values of expressions, and the denoted values are the values bound to variables.

In the languages of this chapter, the expressed and denoted values will always be the same. They will start out as

$$ExpVal = Int + Bool$$
$$DenVal = Int + Bool$$

Chapter 4 presents languages in which expressed and denoted values are different.

In order to make use of this definition, we will need an interface for the data type of expressed values. Our interface will have the entries

```
num-val          : Int → ExpVal
bool-val          : Bool → ExpVal
expval->num  : ExpVal→ Int
expval->bool : ExpVal→ Bool
```

We assume that expval->num and expval->bool are undefined when given an argument that is not a number or a boolean, respectively.

3.2.3 Environments

If we are going to evaluate expressions containing variables, we will need to know the value associated with each variable. We do this by keeping those values in an *environment*, as defined in section 2.2.

An environment is a function whose domain is a finite set of variables and whose range is the denoted values. We use some abbreviations when writing about environments.

- ρ ranges over environments.
- [] denotes the empty environment.
- [*var* = *val*] ρ denotes (extend-env *var val* ρ).
- [var_1 = val_1, var_2 = val_2] ρ abbreviates [var_1 = val_1]([var_2 = val_2] ρ), etc.
- [var_1 = val_1, var_2 = val_2, …] denotes the environment in which the value of var_1 is val_1, etc.

We will occasionally write down complicated environments using indentation to improve readability. For example, we might write

```
[x=3]
 [y=7]
  [u=5] ρ
```

to abbreviate

```
(extend-env 'x 3
  (extend-env 'y 7
    (extend-env 'u 5 ρ)))
```

3.2.4 Specifying the Behavior of Expressions

There are six kinds of expressions in our language: one for each production with *Expression* as its left-hand side. Our interface for expressions will contain seven procedures: six constructors and one observer. We use *ExpVal* to denote the set of expressed values.

constructors:

```
const-exp   : Int →Exp
zero?-exp   : Exp → Exp
if-exp          : Exp × Exp × Exp → Exp
diff-exp       : Exp × Exp → Exp
var-exp        : Var → Exp
let-exp        : Var × Exp × Exp → Exp
```

observer:

```
value-of : Exp × Env → ExpVal
```

Before starting on an implementation, we write down a specification for the behavior of these procedures. Following the interpreter recipe, we expect that value-of will look at the expression, determine what kind of expression it is, and return the appropriate value.

```
(value-of (const-exp n) ρ) = (num-val n)
```

```
(value-of (var-exp var) ρ) = (apply-env ρ var)

(value-of (diff-exp exp1 exp2) ρ)
= (num-val
    (-
      (expval->num (value-of exp1 ρ))
      (expval->num (value-of exp2 ρ))))
```

The value of a constant expression in any environment is the constant value. The value of a variable reference in an environment is determined by looking up the variable in the environment. The value of a difference expression in some environment is the difference between the value of the first operand in that environment and the value of the second operand in that environment. Of course, to be precise we have to make sure that the values of the operands are numbers, and we have to make sure that value of the result is a number represented as an expressed value.

Figure 3.3 shows how these rules work together to specify the value of an expression built by these constructors. In this and our other examples, we write «*exp*» to denote the AST for expression *exp*. We also write ⌈*n*⌉ in place of (num-val *n*), and ⌊*val*⌋ in place of (expval->num *val*). We will also use the fact that ⌈⌊*n*⌋⌉ = *n*.

```
Let ρ = [i=1, v=5, x=10].

(value-of
  <<-(-(x,3), -(v,i))>>
  ρ)

= ┌(-
    └(value-of <<-(x, 3)>> ρ)┘
    └(value-of <<-(v, i)>> ρ)┘)┐

= ┌(-
    (-
      └(value-of <<x>> ρ)┘
      └(value-of <<3>> ρ)┘)
    └(value-of <<-(v,i)>> ρ)┘)┐

= ┌(-
    (-
      10
      └(value-of <<3>> ρ)┘)
    (value-of <<-(v,i) >> ρ))┐

= ┌(-
    (-
      10
      3)
    └(value-of <<-(v,i)>> ρ)┘)┐

= ┌(
     7
     └(value-of <<-(v, i)>> ρ)┘)┐

= ┌(-
     7
     (-
       └(value-of <<v>> ρ)┘
       └(value-of <<i>> ρ)┘))┐

= ┌(-
     7
     (-
        5
       └(value-of <<i>> ρ)┘))┐

= ┌(-
     7
     (-
        5
       1))┐

= ┌(-
     7
     4)┐

= ┌3┐
```

Figure 3.3: A simple calculation using the specification

Exercise 3.1 [*] In figure 3.3, list all the places where we used the fact that ⌈⌊*n*⌋⌉ = *n*.

Exercise 3.2 [**] Give an expressed value *val* ∈ *ExpVal* for which ⌈⌊*val*⌋⌉ ≠ *val*.

3.2.5 Specifying the Behavior of Programs

In our language, a whole program is just an expression. In order to find the value of such an expression, we need to specify the values of the free variables in the program. So the value of a program is just the value of that expression in a suitable initial environment. We choose our initial environment to be [i=1, v=5, x=10].

```
(value-of-program exp)
= (value-of exp [i=⌈1⌉, v=⌈5⌉, x=⌈10⌉])
```

3.2.6 Specifying Conditionals

The next portion of the language introduces an interface for booleans in our language. The language has one constructor of booleans, zero?, and one observer of booleans, the if expression.

The value of a zero? expression is a true value if and only if the value of its operand is zero. We can write this as a rule of inference like those in definition 1.1.5. We use bool-val as a constructor to turn a boolean into an expressed value, and expval->num as an extractor to check whether an expressed value is an integer, and if so, to return the integer.

$$\frac{\texttt{(value-of}\ exp_1\ \rho) = val_1}{\texttt{(value-of (zero?-exp}\ exp_1)\ \rho) = \begin{cases}\texttt{(bool-val \#t)} & \text{if } \texttt{(expval->num}\ val_1) = 0\\ \texttt{(bool-val \#f)} & \text{if } \texttt{(expval->num}\ val_1) \neq 0\end{cases}}$$

An if expression is an observer of boolean values. To determine the value of an if expression (if-exp exp_1 exp_2 exp_3), we must first determine the value of the subexpression exp_1. If this value is a true value, the value of the entire if-exp should be the value of the subexpression exp_2; otherwise it should be the value of the subexpression exp_3. This is also easy to write as a rule of inference. We use expval->bool to extract the boolean part of an expressed value, just as we used expval->num in the preceding example.

$$\frac{\texttt{(value-of}\ exp_1\ \rho) = val_1}{\texttt{(value-of (if-exp}\ exp_1\ exp_2\ exp_3)\ \rho) = \begin{cases}\texttt{(value-of}\ exp_2\ \rho) & \text{if } \texttt{(expval->bool}\ val_1) = \texttt{\#t}\\ \texttt{(value-of}\ exp_3\ \rho) & \text{if } \texttt{(expval->bool}\ val_1) = \texttt{\#f}\end{cases}}$$

Rules of inference like this make the intended behavior of any individual expression easy to specify, but they are not very good for displaying a deduction. An antecedent like (value-of exp_1 ρ) = val_1 denotes a sub-computation, so a calculation should be a tree, much like the one on page 5. Unfortunately, such trees can be difficult to read. We therefore often recast our rules as equations. We can then use substitution of equals for equals to display a calculation.

For an if-exp, the equational specification is

```
(value-of (if-exp exp1 exp2 exp3) ρ)
= (if (expval->bool (value-of exp1 ρ))
    (value-of exp2 ρ)
    (value-of exp3 ρ))
```

Figure 3.4 shows a simple calculation using these rules.

```
Let ρ = [x=33, y=22].

(value-of
  <<if zero?(-(x,11)) then -(y,2) else -(y,4)>>
  ρ)

= (if (expval->bool (value-of <<zero?(-(x,11))>> ρ))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if (expval->bool (bool-val #f))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if #f
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (value-of <<-(y,4)>> ρ)
```

= ⌈18⌉

Figure 3.4: A simple calculation for a conditional expression

3.2.7 Specifying let

Next we address the problem of creating new variable bindings with a let expression. We add to the interpreted language a syntax in which the keyword let is followed by a declaration, the keyword in, and the body. For example,

```
let x = 5
in -(x,3)
```

The let variable is bound in the body, much as a lambda variable is bound (see section 1.2.4).

The entire let form is an expression, as is its body, so let expressions may be nested, as in

```
let z = 5
in let x = 3
   in let y = -(x,1)     % here x = 3
      in let x = 4
         in -(z, -(x,y)) % here x = 4
```

In this example, the reference to x in the first difference expression refers to the outer declaration, whereas the reference to x in the other difference expression refers to the inner declaration, and thus the entire expression's value is 3.

The right-hand side of the let is also an expression, so it can be arbitrarily complex. For example,

```
let x = 7
in let y = 2
   in let y = let x = -(x,1)
              in -(x,y)
      in -(-(x,8), y)
```

Here the x declared on the third line is bound to 6, so the value of y is 4, and the value of the entire expression is ((−1) − 4) = −5.

We can write down the specification as a rule.

$$\frac{(\texttt{value-of}\ \ exp_1\ \ \rho) = val_1}{\begin{array}{c}(\texttt{value-of}\ \ (\texttt{let-exp}\ \ var\ \ exp_1\ \ body)\ \ \rho) \\ = \ \ (\texttt{value-of}\ \ body\ \ [var = val_1]\rho)\end{array}}$$

As before, it is often more convenient to recast this as the equation

```
(value-of (let-exp var exp1 body) ρ)
= (value-of body [var=(value-of exp1 ρ)]ρ)
```

Figure 3.5 shows an example. There ρ_0 denotes an arbitrary environment.

```
(value-of
  <<let x = 7
    in let y = 2
       in let y = let x = -(x,1) in -(x,y)
          in -(-(x,8),y)>>
  ρ0)

= (value-of
    <<let y = 2
      in let y = let x = -(x,1) in -(x,y)
         in -(-(x,8),y)>>
    [x=⌈7⌉]ρ0)
```

```
= (value-of
    <<let y = let x = -(x,1) in -(x,y)
      in -(-(x,8),y)>>
    [y=⌈2⌉][x=⌈7⌉]ρ0)

Let ρ1 = [y=⌈2⌉][x=⌈7⌉]ρ0.

= (value-of
    <<-(-(x,8),y)>>
    [y=(value-of <<let x = -(x,1) in -(x,y)>> ρ1)]
     ρ1)

= (value-of
    <<-(-(x,8),y)>>
    [y=(value-of <<-(x,2)>> [x=(value-of <<-(x,1)>> ρ1)]ρ1)]
     ρ1)

= (value-of
    <<-(-(x,8),y)>>
    [y=(value-of <<-(x,2)>> [x=⌈6⌉]ρ1)]
     ρ1)

= (value-of
    <<-(-(x,8),y)>>
    [y=⌈4⌉]ρ1)

= ⌈(- (- 7 8) 4)⌉

= ⌈-5⌉
```

Figure 3.5: An example of `let`

3.2.8 Implementing the Specification of LET

Our next task is to implement this specification as a set of Scheme procedures. Our implementation uses SLLGEN as a front end, which means that expressions will be represented by a data type like the one in figure 3.6. The representation of expressed values in our implementation is shown in figure 3.7. The data type declares the constructors num-val and bool-val for converting integers and booleans to expressed values. We also define extractors for converting from an expressed value back to either an integer or a boolean. The extractors report an error if an expressed value is not of the expected kind.

```
(define-datatype program program?
  (a-program
    (exp1 expression?)))

(define-datatype expression expression?
  (const-exp
    (num number?))
  (diff-exp
    (exp1 expression?)
    (exp2 expression?))
  (zero?-exp
    (exp1 expression?))
    (if-exp
  (exp1 expression?)
    (exp2 expression?)
    (exp3 expression?))
  (var-exp
    (var identifier?))
  (let-exp
    (var identifier?)
    (exp1 expression?)
    (body expression?)))
```

Figure 3.6: Syntax data types for the LET language

```
(define-datatype expval expval?
  (num-val
    (num number?))
```

```
  (bool-val
    (bool boolean?)))

expval->num : ExpVal → Int
(define expval->num
  (lambda (val)
    (cases expval val
      (num-val (num) num)
      (else (report-expval-extractor-error 'num val)))))

expval->bool : ExpVal → Bool
(define expval->bool
  (lambda (val)
    (cases expval val
      (bool-val (bool) bool)
      (else (report-expval-extractor-error 'bool val)))))
```

Figure 3.7: Expressed values for the LET language

We can use any implementation of environments, provided that it meets the specification in section 2.2. The procedure init-env constructs the specified initial environment used by value-of-program.

```
init-env : () → Env
usage: (init-env) = [i=1,v=5,x=10]
(define init-env
  (lambda ()
    (extend-env
     'i (num-val 1)
     (extend-env
      'v (num-val 5)
      (extend-env
       'x (num-val 10)
       (empty-env))))))
```

Now we can write down the interpreter, shown in figures 3.8 and 3.9. The main procedure is run, which takes a string, parses it, and hands the result to value-of-program. The most interesting procedure is value-of, which takes an expression and an environment and uses the interpreter recipe to calculate the answer required by the specification. In the listing below we have inserted the relevant specification rules to show how the code for value-of comes from the specification.

run : *String* → *ExpVal*

```
(define run
  (lambda (string)
    (value-of-program (scan&parse string))))
```

value-of-program : *Program* → *ExpVal*

```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))
```

value-of : *Exp* × *Env* → *ExpVal*

```
(define value-of
  (lambda (exp env)
    (cases expression exp
```

```
      (value-of (const-exp n) ρ) = n
      (const-exp (num) (num-val num))

      (value-of (var-exp var) ρ) = (apply-env ρ var)
      (var-exp (var) (apply-env env var))

      (value-of (diff-exp exp1 exp2) ρ) =
        ⌈(- ⌊(value-of exp1 ρ)⌋ ⌊(value-of exp2 ρ)⌋)⌉
      (diff-exp (exp1 exp2)
        (let ((val1 (value-of exp1 env))
              (val2 (value-of exp2 env)))
          (let ((num1 (expval->num val1))
                (num2 (expval->num val2)))
            (num-val
              (- num1 num2))))))
```

Figure 3.8: Interpreter for the LET language

$$\frac{\text{(value-of } exp_1 \ \rho) = val_1}{\text{(value-of (zero?-exp } exp_1) \ \rho) = \begin{cases}\text{(bool-val \#t)} & \text{if (expval->num } val_1) = 0 \\ \text{(bool-val \#f)} & \text{if (expval->num } val_1) \neq 0\end{cases}}$$

```
(zero?-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((num1 (expval->num val1)))
      (if (zero? num1)
        (bool-val #t)
        (bool-val #f)))))
```

$$\frac{\text{(value-of } exp_1 \ \rho) = val_1}{\text{(value-of (if-exp } exp_1 \ exp_2 \ exp_3) \ \rho) = \begin{cases}\text{(value-of } exp_2 \ \rho) & \text{if (expval->bool } val_1) = \text{\#t} \\ \text{(value-of } exp_3 \ \rho) & \text{if (expval->bool } val_1) = \text{\#f}\end{cases}}$$

```
(if-exp (exp1 exp2 exp3)
  (let ((val1 (value-of exp1 env)))
    (if (expval->bool val1)
      (value-of exp2 env)
      (value-of exp3 env))))
```

$$\frac{\text{(value-of } exp_1 \ \rho) = val_1}{\text{(value-of (let-exp } var \ exp_1 \ body) \ \rho) = \text{(value-of } body \ [var = val_1]\rho)}$$

```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var val1 env))))))
```

Figure 3.9: Interpreter for the LET language, continued

In the following exercises, and throughout the book, the phrase “extend the language by adding …” means to write down additional rules or equations to the language specification, and to implement the feature by adding or modifying the associated interpreter.

Exercise 3.3 [*] Why is subtraction a better choice than addition for our single arithmetic operation?

Exercise 3.4 [*] Write out the derivation of figure 3.4 as a derivation tree in the style of the one on page 5.

Exercise 3.5 [*] Write out the derivation of figure 3.5 as a derivation tree in the style of the one on page 5.

Exercise 3.6 [*] Extend the language by adding a new operator minus that takes one argument, *n*, and returns –*n*. For example, the value of minus(-(minus(5),9)) should be 14.

Exercise 3.7 [*] Extend the language by adding operators for addition, multiplication, and integer quotient.

Exercise 3.8 [*] Add a numeric equality predicate equal? and numeric order predicates greater? and less? to the set of operations in the defined language.

Exercise 3.9 [**] Add list processing operations to the language, including cons, car, cdr, null? and emptylist. A list should be able to contain any expressed value, including another list. Give the definitions of the expressed and denoted values of the language, as in section 3.2.2. For example,

```
let x = 4
in cons(x,
        cons(cons(-(x,1),
                  emptylist),
             emptylist))
```

should return an expressed value that represents the list (4 (3)).

Exercise 3.10 [**] Add an operation list to the language. This operation should take any number of arguments, and return an expressed value containing the list of their values. For example, should return an expressed value that represents the list (4 3 1).

```
let x = 4
in list(x, -(x,1), -(x,3))
```

Exercise 3.11 [*] In a real language, one might have many operators such as those in the preceding exercises. Rearrange the code in the interpreter so that it is easy to add new operators.

Exercise 3.12 [*] Add to the defined language a facility that adds a cond expression. Use the grammar

Expression ::= cond {*Expression* ==> *Expression*}* end

In this expression, the expressions on the left-hand sides of the ==>'s are evaluated in order until one of them returns a true value. Then the value of the entire expression is the value of the corresponding right-hand expression. If none of the tests succeeds, the expression should report an error.

Exercise 3.13 [*] Change the values of the language so that integers are the only expressed values. Modify if so that the value 0 is treated as false and all other values are treated as true. Modify the predicates accordingly.

Exercise 3.14 [**] As an alternative to the preceding exercise, add a new nonterminal *Bool-exp* of boolean expressions to the language. Change the production for conditional expressions to say

Expression ::= if *Bool-exp* then *Expression* else *Expression*

Write suitable productions for *Bool-exp* and implement value-of-bool-exp. Where do the predicates of exercise 3.8 wind up in this organization?

Exercise 3.15 [*] Extend the language by adding a new operation print that takes one argument, prints it, and returns the integer 1. Why is this operation not expressible in our specification framework?

Exercise 3.16 [**] Extend the language so that a let declaration can declare an arbitrary number of variables, using the grammar

Expression ::= let {*Identifier* = *Expression*}* in *Expression*

As in Scheme's let, each of the right-hand sides is evaluated in the current environment, and the body is evaluated with each new variable bound to the value of its associated right-hand side. For example,

```
let x = 30
in let x = -(x,1)
       y = -(x,2)
   in -(x,y)
```

should evaluate to 1.

Exercise 3.17 [**] Extend the language with a let* expression that works like Scheme's let*, so that

```
let x = 30
in let* x = -(x,1) y = -(x,2)
```

```
   in -(x,y)
```

should evaluate to 2.

Exercise 3.18 [**] Add an expression to the defined language:

Expression ::= unpack {*Identifier*}* = *Expression* in *Expression*

so that unpack x y z = lst in ... binds x, y, and z to the elements of lst if lst is a list of exactly three elements, and reports an error otherwise. For example, the value of

```
let u = 7
in unpack x y = cons(u,cons(3,emptylist))
   in -(x,y)
```

should be 4.

3.3 PROC: A Language with Procedures

So far our language has only the operations that were included in the original language. For our interpreted language to be at all useful, we must allow new procedures to be created. We call the new language PROC.

We will follow the design of Scheme, and let procedures be expressed values in our language, so that

$$ExpVal = Int + Bool + Proc$$
$$DenVal = Int + Bool + Proc$$

where *Proc* is a set of values representing procedures. We will think of *Proc* as an abstract data type. We consider its interface and specification below.

We will also need syntax for procedure creation and calling. This is given by the productions

$$Expression ::= \texttt{proc}\ (Identifier)\ Expression$$
```
proc-exp (var body)
```

$$Expression ::= (Expression\ \ Expression)$$
```
call-exp (rator rand)
```

In (proc-exp *var body*), the variable *var* is the *bound variable* or *formal parameter*. In a procedure call (call-exp exp_1 exp_2), the expression exp_1 is the *operator* and exp_2 is the *operand* or *actual parameter*. We use the word *argument* to refer to the value of an actual parameter.

Here are two simple programs in this language.

```
let f = proc (x) -(x,11)
in (f (f 77))

(proc (f) (f (f 77))
 proc (x) -(x,11))
```

The first program creates a procedure that subtracts 11 from its argument. It calls the resulting procedure f, and then applies f twice to 77, yielding the answer 55. The second program creates a procedure that takes its argument and applies it twice to 77. The program then applies this procedure to the subtract-11 procedure. The result is again 55.

We now turn to the data type *Proc*. Its interface consists of the constructor procedure, which tells how to build a procedure value, and the observer apply-procedure, which tells how to apply a procedure value.

Our next task is to determine what information must be included in a value representing a procedure. To do this, we consider

what happens when we write a proc expression in an arbitrary position in our program.

The lexical scope rule tells us that when a procedure is applied, its body is evaluated in an environment that binds the formal parameter of the procedure to the argument of the call. Variables occurring free in the procedure should also obey the lexical binding rule. Consider the expression

```
let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```

Here we evaluate the expression proc (z) -(z,x) twice. The first time we do it, x is bound to 200, so by the lexical scope rule, the result is a procedure that subtracts 200 from its argument. We name this procedure f. The second time we do it, x is bound to 100, so the resulting procedure should subtract 100 from its argument. We name this procedure g.

These two procedures, created from identical expressions, must be have differently. We conclude that the value of a proc expression must depend in some way on the environment in which it is evaluated. Therefore the constructor procedure must take three arguments: the bound variable, the body, and the environment. The specification for a proc expression is

```
(value-of (proc-exp var body) ρ)
= (proc-val (procedure var body ρ))
```

where proc-val is a constructor, like bool-val or num-val, that builds an expressed value from a *Proc*.

At a procedure call, we want to find the value of the operator and the operand. If the value of the operator is a proc-val, then we want to apply it to the value of the operand.

```
(value-of (call-exp rator rand) ρ)
= (let ((proc (expval->proc (value-of rator ρ)))
        (arg (value-of rand ρ)))
    (apply-procedure proc arg))
```

Here we rely on a tester expval->proc, like expval->num, to test whether the value of (value-of *rator* ρ), an expressed value, was constructed by proc-val, and if so to extract the underlying procedure.

Last, we consider what happens when apply-procedure is invoked. As we have seen, the lexical scope rule tells us that when a procedure is applied, its body is evaluated in an environment that binds the formal parameter of the procedure to the argument of the call. Furthermore any other variables must have the same values they had at procedure-creation time. Therefore these procedures should satisfy the condition

```
(apply-procedure (procedure var body ρ) val)
= (value-of body [var=val]ρ)
```

3.3.1 An Example

Let's do an example to show how the pieces of the specification fit together. This is a calculation using the *specification*, not the implementation, since we have not yet written down the implementation of procedures. Let ρ be any environment.

```
(value-of
  <<let x = 200
    in let f = proc (z) -(z,x)
       in let x = 100
          in let g = proc (z) -(z,x)
             in -((f 1), (g 1))>>
  ρ)

= (value-of
    <<let f = proc (z) -(z,x)
      in let x = 100
         in let g = proc (z) -(z,x)
            in -((f 1), (g 1))>>
    [x=⌈200⌉]ρ)


= (value-of
    <<let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))>>
```

```
    [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
     [x=⌈200⌉]ρ)

= (value-of
    <<let g = proc (z) -(z,x)
      in -((f 1), (g 1))>>
    [x=⌈100⌉]
     [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
      [x=⌈200⌉]ρ)

= (value-of
    <<-((f 1), (g 1))>>
    [g=(proc-val (procedure z <<-(z,x)>>
                   [x=⌈100⌉][f=...][x=⌈200⌉]ρ))]
     [x=⌈100⌉]
      [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
       [x=⌈200⌉]ρ)

= ⌈(-
      (value-of <<(f 1)>>
        [g=(proc-val (procedure z <<-(z,x)>>
                       [x=⌈100⌉][f=...][x=⌈200⌉]ρ))]
         [x=⌈100⌉]
          [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
           [x=⌈200⌉]ρ)
      (value-of <<(g 1)>>
        [g=(proc-val (procedure z <<-(z,x)>>
                       [x=⌈100⌉][f=...][x=⌈200⌉]ρ))]
         [x=⌈100⌉]
          [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
           [x=⌈200⌉]ρ))⌉

= ⌈(-
      (apply-procedure
        (procedure z <<-(z,x)>> [x=⌈200⌉]ρ)
        ⌈1⌉)
      (apply-procedure
        (procedure z <<-(z,x)>> [x=⌈100⌉][f=...][x=⌈200⌉]ρ)
        ⌈1⌉))⌉

= ⌈(-
      (value-of <<-(z,x)>> [z=⌈1⌉][x=⌈200⌉]ρ)
      (value-of <<-(z,x)>> [z=⌈1⌉][x=⌈100⌉][f=...][x=⌈200⌉]ρ))⌉

= ⌈(- -199 -99)⌉

= ⌈-100⌉
```

Here f is bound to a procedure that subtracts 200 from its argument, and g is bound to a procedure that subtracts 100 from its argument, so the value of (f 1) is –199 and the value of (g 1) is –99.

3.3.2 Representing Procedures

According to the recipe described in section 2.2.3, we can employ a procedural representation for procedures by their action under apply-procedure. To do this we define procedure to have a value that is an implementation-language procedure that expects an argument, and returns the value required by the specification

```
(apply-procedure (procedure var body ρ) val )
= (value-of body (extend-env var val ρ))
```

Therefore the entire implementation is

```
proc? : SchemeVal → Bool
(define proc?
  (lambda (val)
    (procedure? val)))

procedure : Var × Exp × Env → Proc
```

```
(define procedure
  (lambda (var body env)
    (lambda (val)
      (value-of body (extend-env var val env)))))

apply-procedure : Proc × ExpVal → ExpVal
(define apply-procedure
  (lambda (proc1 val)
    (proc1 val)))
```

The function proc?, as defined here, is somewhat inaccurate, since not every Scheme procedure is a possible procedure in our language. We need it only for defining the data type expval.

Alternatively, we could use a data structure representation like that of section 2.2.2.

```
proc? : SchemeVal → Bool
procedure : Var × Exp × Env → Proc
(define-datatype proc proc?
  (procedure
    (var identifier?)
    (body expression?)
    (saved-env environment?)))

apply-procedure : Proc × ExpVal → ExpVal
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body (extend-env var val saved-env))))))
```

These data structures are often called *closures*, because they are self-contained: they contain everything the procedure needs in order to be applied. We sometimes say the procedure is *closed over* or *closed in* its creation environment.

Each of these implementations evidently satisfies the specification for the procedure interface.

In either implementation, we add an alternative to the data type expval

```
(define-datatype expval expval?
  (num-val
    (num number?))
  (bool-val
    (bool boolean?))
  (proc-val
    (proc proc?)))
```

and we need to add two new clauses to value-of

```
(proc-exp (var body)
  (proc-val (procedure var body env)))
(call-exp (rator rand)
  (let ((proc (expval->proc (value-of rator env)))
        (arg (value-of rand env)))
    (apply-procedure proc arg)))
```

Reminder: be sure to write down specifications for each language extension. See the note on page 70.

Exercise 3.19 [*] In many languages, procedures must be created and named at the same time. Modify the language of this section to have this property by replacing the proc expression with a letproc expression.

Exercise 3.20 [*] In PROC, procedures have only one argument, but one can get the effect of multiple argument procedures by using procedures that return other procedures. For example, one might write code like

```
let f = proc (x) proc (y) ...
in ((f 3) 4)
```

This trick is called *Currying*, and the procedure is said to be *Curried*. Write a Curried procedure that takes two arguments and returns their sum. You can write $x + y$ in our language by writing $-(x, -(0, y))$.

Exercise 3.21 [**] Extend the language of this section to include procedures with multiple arguments and calls with multiple operands, as suggested by the grammar

```
Expression ::= proc ({Identifier}*(,)) Expression
Expression ::= (Expression {Expression}*)
```

Exercise 3.22 [***] The concrete syntax of this section uses different syntax for a built-in operation, such as difference, from a procedure call. Modify the concrete syntax so that the user of this language need not know which operations are built-in and which are defined procedures. This exercise may range from very easy to hard, depending on the parsing technology being used.

Exercise 3.23 [**] What is the value of the following PROC program?

```
let makemult = proc (maker)
                proc (x)
                 if zero?(x)
                 then 0
                 else -(((maker maker) -(x,1)), -4)
in let times4 = proc (x) ((makemult makemult) x)
   in (times4 3)
```

Use the tricks of this program to write a procedure for factorial in PROC. As a hint, remember that you can use Currying (exercise 3.20) to define a two-argument procedure times.

Exercise 3.24 [**] Use the tricks of the program above to write the pair of mutually recursive procedures, odd and even, as in exercise 3.32.

Exercise 3.25 [*] The tricks of the previous exercises can be generalized to show that we can define any recursive procedure in PROC. Consider the following bit of code:

```
let makerec = proc (f)
               let d = proc (x)
                        proc (z) ((f (x x)) z)
               in proc (n) ((f (d d)) n)
in let maketimes4 = proc (f)
                     proc (x)
                      if zero?(x)
                      then 0
                      else -((f -(x,1)), -4)
   in let times4 = (makerec maketimes4)
      in (times4 3)
```

Show that it returns 12.

Exercise 3.26 [**] In our data-structure representation of procedures, we have kept the entire environment in the closure. But of course all we need are the bindings for the free variables. Modify the representation of procedures to retain only the free variables.

Exercise 3.27 [*] Add a new kind of procedure called a traceproc to the language. A traceproc works exactly like a proc, except that it prints a trace message on entry and on exit.

Exercise 3.28 [**] *Dynamic binding* (or *dynamic scoping*) is an alternative design for procedures, in which the procedure body is evaluated in an environment obtained by extending the environment at the point of call. For example in

```
let a = 3
in let p = proc (x) -(x,a)
       a=5
   in -(a,(p 2))
```

the a in the procedure body would be bound to 5, not 3. Modify the language to use dynamic binding. Do this twice, once using a procedural representation for procedures, and once using a data-structure representation.

Exercise 3.29 [**] Unfortunately, programs that use dynamic binding may be exceptionally difficult to understand. For example, under lexical binding, consistently renaming the bound variables of a procedure can never change the behavior of a program: we can even remove all variables and replace them by their lexical addresses, as in section 3.6. But under dynamic binding, this transformation is unsafe.

For example, under dynamic binding, the procedure proc (z) a returns the value of the variable a in its caller's environment. Thus, the program

```
let a = 3
in let p = proc (z) a
```

```
   in let f = proc (x) (p 0)
      in let a = 5
         in (f 2)
```

returns 5, since a's value at the call site is 5. What if f's formal parameter were a?

3.4 LETREC: A Language with Recursive Procedures

We now define a new language LETREC, which adds recursion to our language. Since our language has only one-argument procedures, we make our life simpler by having our letrec expressions declare only a single one-argument procedure, for example

```
letrec double(x)
        = if zero?(x) then 0 else -((double -(x,1)), -2)
in (double 6)
```

The left-hand side of a recursive declaration is the name of the recursive procedure and its bound variable. To the right of the = is the procedure body. The production for this is

Expression ::= `letrec` *Identifier* `(` *Identifier* `)` `=` *Expression* `in` *Expression*

```
letrec-exp (p-name b-var p-body letrec-body)
```

The value of a letrec expression is the value of the body in an environment that has the desired behavior:

```
(value-of
  (letrec-exp proc-name bound-var proc-body letrec-body)
  ρ)
= (value-of
    letrec-body
    (extend-env-rec proc-name bound-var proc-body ρ))
```

Here we have added a new procedure extend-env-rec to the environment interface. But we still need to answer the question: What is the desired behavior of (extend-env-rec *proc-name bound-var proc-body ρ*)?

We specify the behavior of this environment as follows: Let ρ_1 be the environment produced by (extend-env-rec *proc-name bound-var proc-body ρ*). Then what should (apply-env ρ_1 *var*) return?

1. If the variable *var* is the same as *proc-name*, then (apply-env ρ_1 *var*) should produce a closure whose bound variable is *bound-var*, whose body is *proc-body*, and with an environment in which *proc-name* is bound to this procedure. But we already have such an environment, namely ρ_1 itself! So

   ```
   (apply-env ρ1 proc-name)
   = (proc-val (procedure bound-var proc-body ρ1))
   ```

2. If *var* is not the same as *proc-name*, then

   ```
   (apply-env ρ1 var) = (apply-env ρ var)
   ```

Figures 3.10 and 3.11 show an example. There in the last line of figure 3.11, the recursive call to double finds the original double procedure, as desired.

```
(value-of <<letrec double(x) = if zero?(x)
                               then 0
                               else -((double -(x,1)), -2)
           in (double 6)>> ρ0)

= (value-of <<(double 6)>>
    (extend-env-rec double x <<if zero?(x) ...>> ρ0))

= (apply-procedure
    (value-of <<double>> (extend-env-rec double x
                           <<if zero?(x) ...>> ρ0))
    (value-of <<6>> (extend-env-rec double x
                      <<if zero?(x) ...>> ρ0)))
```

```
= (apply-procedure
    (procedure x <<if zero?(x) ...>>
      (extend-env-rec double x <<if zero?(x) ...>> ρ0))
  ⌈6⌉)

= (value-of
    <<if zero?(x) ...>>
   [x=⌈6⌉](extend-env-rec
                        double x <<if zero?(x) ...>> ρ0))

...

=(-
    (value-of
      <<(double -(x,1))>>
      [x=⌈6⌉](extend-env-rec
                          double x <<if zero?(x) ...>> ρ0))
    -2)
```

Figure 3.10: A calculation with `extend-env-rec`

```
=(-
    (apply-procedure
      (value-of
        <<double>>
        [x=⌈6⌉](extend-env-rec
                            double x <<if zero?(x) ...>> ρ0))
      (value-of
        <<-(x,1)>>
        [x=⌈6⌉](extend-env-rec
                            double x <<if zero?(x) ...>> ρ0)))
      -2)

=(-
    (apply-procedure
      (procedure x <<if zero?(x) ...>>
        (extend-env-rec double x <<if zero?(x) ...>> ρ0))
    ⌈5⌉)
    -2)

= ...
```

Figure 3.11: A calculation with `extend-env-rec`, cont'd.

We can implement extend-env-rec in any way that satisfies these requirements. We'll do it here for the abstract-syntax representation. Some other implementation strategies are discussed in the exercises.

In an abstract-syntax representation, we add a new variant for an extend-env-rec in figure 3.12. The env on the next-to-last line of apply-env corresponds to ρ_1 in the discussion above.

```
(define-datatype environment environment?
  (empty-env)
  (extend-env
    (var identifier?)
    (val expval?)
    (env environment?))
  (extend-env-rec
    (p-name identifier?)
    (b-var identifier?)
    (body expression?)
    (env environment?)))

(define apply-env
  (lambda (env search-var)
    (cases environment env
      (empty-env ()
        (report-no-binding-found search-var))
      (extend-env (saved-var saved-val saved-env)
        (if (eqv? saved-var search-var)
```

```
          saved-val
          (apply-env saved-env search-var)))
      (extend-env-rec (p-name b-var p-body saved-env)
        (if (eqv? search-var p-name)
          (proc-val (procedure b-var p-body env))
          (apply-env saved-env search-var))))))
```

Figure 3.12: `extend-env-rec` added to environments.

Exercise 3.30 [*] What is the purpose of the call to proc-val on the next-to-last line of apply-env?

Exercise 3.31 [*] Extend the language above to allow the declaration of a recursive procedure of possibly many arguments, as in exercise 3.21.

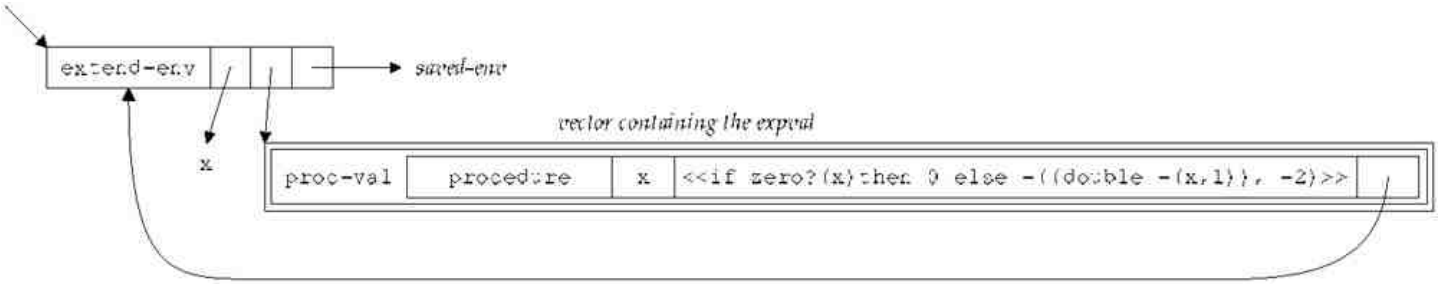
Exercise 3.32 [**] Extend the language above to allow the declaration of any number of mututally recursive unary procedures, for example:

```
letrec
  even(x) = if zero?(x) then 1 else (odd -(x,1))
  odd(x) = if zero?(x) then 0 else (even -(x,1))
in (odd 13)
```

Exercise 3.33 [**] Extend the language above to allow the declaration of any number of mutually recursive procedures, each of possibly many arguments, as in exercise 3.21.

Exercise 3.34 [***] Implement extend-env-rec in the procedural representation of environments from section 2.2.3.

Exercise 3.35 [*] The representations we have seen so far are inefficient, because they build a new closure every time the procedure is retrieved. But the closure is the same every time. We can build the closures only once, by putting the value in a vector of length 1 and building an explicit circular structure, like



Here's the code to build this data structure.

```
(define extend-env-rec
  (lambda (p-name b-var body saved-env)
    (let ((vec (make-vector 1)))
      (let ((new-env (extend-env p-name vec saved-env)))
        (vector-set! vec 0
          (proc-val (procedure b-var body new-env)))
        new-env))))
```

Complete the implementation of this representation by modifying the definitions of the environment data type and apply-env accordingly. Be sure that apply-env always returns an expressed value.

Exercise 3.36 [**] Extend this implementation to handle the language from exercise 3.32.

Exercise 3.37 [*] With dynamic binding (exercise 3.28), recursive procedures may be bound by let; no special mechanism is necessary for recursion. This is of historical interest; in the early years of programming language design other approaches to recursion, such as those discussed in section 3.4, were not widely understood. To demonstrate recursion via dynamic binding, test the program

```
let fact = proc (n) add1(n)
in let fact = proc (n)
                if zero?(n)
                then 1
                else *(n,(fact -(n,1)))
   in (fact 5)
```

using both lexical and dynamic binding. Write the mutually recursive procedures even and odd as in section 3.4 in the defined language with dynamic binding.

3.5 Scoping and Binding of Variables

We have now seen a variety of situations in which variables are declared and used. We now discuss these ideas in a more systematic way.

In most programming languages, variables may appear in two different ways: as *references* or as *declarations*. A variable reference is a use of the variable. For example, in the Scheme expression

```
(f x y)
```

all the variables, f, x, and y, appear as references. However, in

```
(lambda (x) (+ x 3))
```

or

```
(let ((x (+ y 7))) (+ x 3))
```

the first occurrence of x is a declaration: it introduces the variable as a name for some value. In the lambda expression, the value of the variable will be supplied when the procedure is called. In the let expression, the value of the variable is obtained from the value of the expression (+ y 7).

We say that a variable reference is *bound by* the declaration with which it is associated, and that it is *bound to* its value. We have already seen examples of a variable being bound by a declaration, in section 1.2.4.

Declarations in most programming languages have a limited scope, so that the same variable name may be used for different purposes in different parts of a program. For example, we have repeatedly used lst as a bound variable, and in each case its scope was limited to the body of the corresponding lambda expression.

Every programming language must have some rules to determine the declaration to which each variable reference refers. These rules are typically called *scoping* rules. The portion of the program in which a declaration is valid is called the *scope* of the declaration.

We can determine which declaration is associated with each variable use without executing the program. Properties like this, which can be computed without executing the program, are called *static* properties.

To find which declaration corresponds to a given use of a variable, we search *outward* from the use until we find a declaration of the variable. Here is a simple example in Scheme.

```
(let ((x 3)            Call this x1
      (y 4))
  (+ (let ((x          Call this x2
            (+ y 5)))
       (* x y))        Here x refers to x2
     x))               Here x refers to x1
```

In this example, the inner x is bound to 9, so the value of the expression is

```
(let ((x 3)
      (y 4))
  (+ (let ((x
            (+ y 5)))
       (* x y))
     x))

= (+ (let ((x
             (+ 4 5)))
       (* x 4))
    3)

= (+ (let ((x 9))
       (* x 4))
    3)

= (+ 36
    3)
```

```
= 39
```

Scoping rules like this are called *lexical scoping* rules, and the variables declared in this way are called *lexical variables*.

Under lexical scoping, we can create a hole in a scope by redeclaring a variable. Such an inner declaration *shadows* the outer one. For instance, in the example above, the inner x shadows the outer one in the multiplication (* x y).

Lexical scopes are nested: each scope lies entirely within another scope. We can illustrate this with a *contour diagram*. Figure 3.13 shows the contour diagram for the example above. A box surrounds each scope, and a vertical line connects each declaration to its scope.

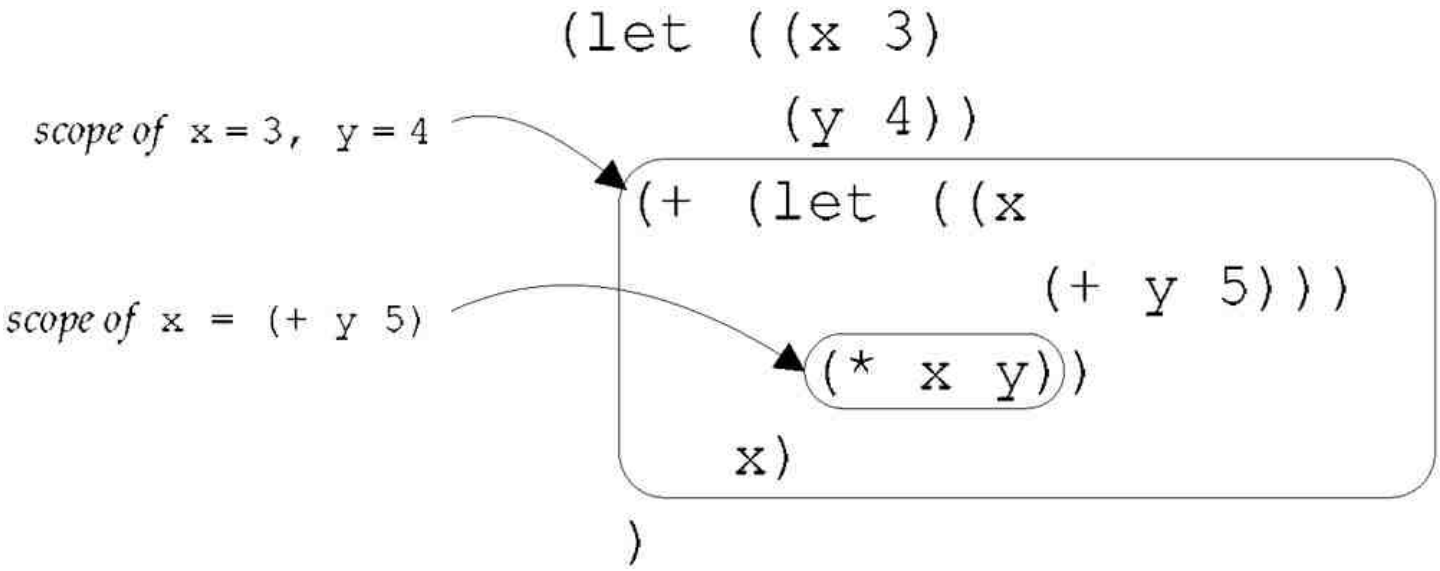


Figure 3.13: A simple contour diagram

Figure 3.14 shows a more complicated program with the contours drawn in. Here there are three occurrences of the expression (+ x y z), on lines 5, 7, and 8. Line 5 is within the scope of x2 and z2, which is within the scope of z1, which is within the scope of x1 and y1. So at line 5, x refers to x2, y refers to y1, and z refers to z2. Line 7 is within the scope of x4 and y2, which is within the scope of x2 and z2, which is within the scope of z1, which is within the scope of x1 and y1. So at line 7, x refers to x4, y refers to y2, and z refers to z2. Last, line 8 is within the scope of x3, which is within the scope of x2 and z2, which is within the scope of z1, which is within the scope of x1 and y1. So at line 8, x refers to x3, y refers to y1, and z refers to z2.

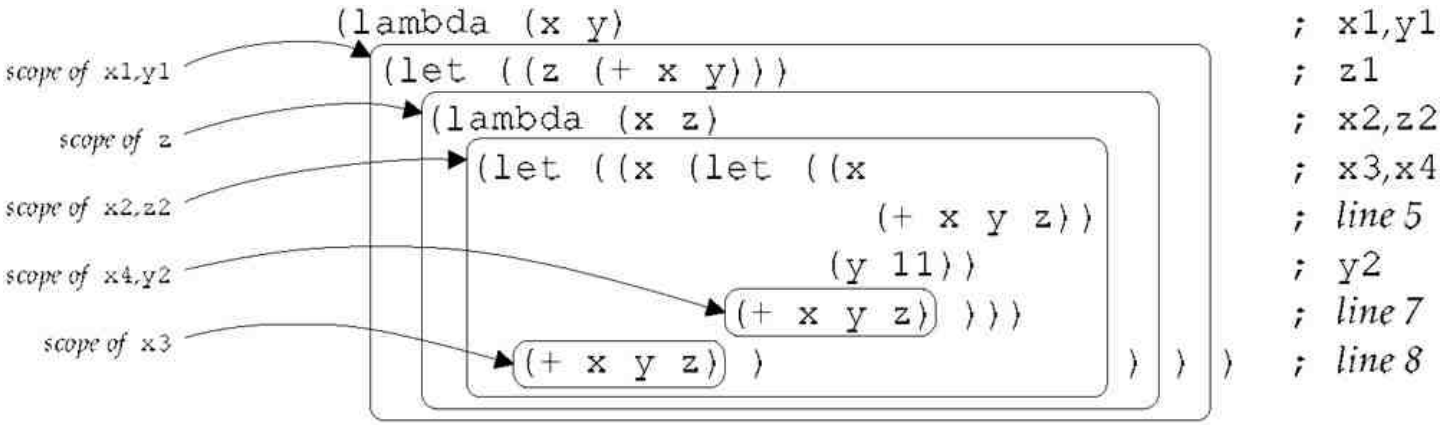


Figure 3.14: A more complicated contour diagram

The association between a variable and its value is called a *binding*. For our language, we can look at the specification to see how the binding is created.

A variable declared by a proc is bound when the procedure is applied.

```
(apply-procedure (procedure var body ρ) val)
= (value-of body (extend-env var val ρ))
```

A let-variable is bound by the value of its right-hand side.

```
(value-of (let-exp var val body) ρ)
= (value-of body (extend-env var val ρ))
```

A variable declared by a letrec is bound using its right-hand side as well.

```
(value-of
  (letrec-exp proc-name bound-var proc-body letrec-body)
  ρ)
= (value-of
    letrec-body
    (extend-env-rec proc-name bound-var proc-body ρ))
```

The *extent* of a binding is the time interval during which the binding is maintained. In our little language, as in Scheme, all bindings have *semi-infinite* extent, meaning that once a variable gets bound, that binding must be maintained indefinitely (at least potentially). This is because the binding might be hidden inside a closure that is returned. In languages with semi-infinite extent, the garbage collector collects bindings when they are no longer reachable. This is only determinable at run-time, so we say that this is a *dynamic* property.

Regrettably, “dynamic” is sometimes used to mean “during the evaluation of an expression” but other times is used to mean “not calculable in advance.” If we did not allow a procedure to be used as the value of a let, then the let-bindings would expire at the end of the evaluation of the let body. This is called *dynamic* extent, and it is a *static* property. Because the extent is a static property, we can predict exactly when a binding can be discarded. Dynamic binding, as in exercise 3.28 *et seq.*, behaves similarly.

3.6 Eliminating Variable Names

Execution of the scoping algorithm may then be viewed as a journey outward from a variable reference. In this journey a number of contours may be crossed before arriving at the associated declaration. The number of contours crossed is called the *lexical* (or *static*) *depth* of the variable reference. It is customary to use “zero-based indexing,” thereby not counting the last contour crossed. For example, in the Scheme expression

```
(lambda (x)
  ((lambda (a)
     (x a))
   x))
```

the reference to x on the last line and the reference to a have lexical depth zero, while the reference to x in the third line has lexical depth one.

We could, therefore, get rid of variable names entirely, and write something like

```
(nameless-lambda
  ((nameless-lambda
     (#1 #0))
   #0))
```

Here each nameless-lambda declares a new anonymous variable, and each variable reference is replaced by its lexical depth; this number uniquely identifies the declaration to which it refers. These numbers are called *lexical addresses* or *de Bruijn indices*. Compilers routinely calculate the lexical address of each variable reference. Once this has been done, the variable names may be discarded unless they are required to provide debugging information.

This way of recording the information is useful because the lexical address *predicts* just where in the environment any particular variable will be found.

Consider the expression

```
let x = exp1
in let y = exp2
   in -(x,y)
```

in our language. In the difference expression, the lexical depths of y and x are 0 and 1, respectively.

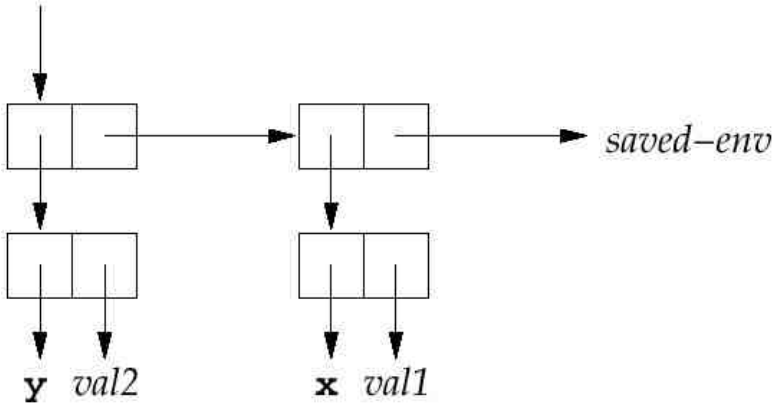
Now assume that the values of exp_1 and exp_2, in the appropriate environments, are val_1 and val_2. Then the value of this expression is

```
(value-of
```

```
  <<let x = exp1
    in let y = exp2
       in -(x,y)>>
  ρ)
=
(value-of
  <<let y = exp2
    in -(x,y)>>
  [x=val1]ρ)
=
(value-of
  <<-(x,y)>>
  [y=val2][x=val1]ρ)
```

so that when the difference expression is evaluated, y is at depth 0 and x is at depth 1, just as predicted by their lexical depths.

If we are using an association-list representation of environments (see exercise 2.5), then the environment will look like



so that the values of x and y will be found by taking either 1 cdr or 0 cdrs in the environment, regardless of the values val_1 and val_2.

The same thing works for procedure bodies. Consider

```
let a = 5
in proc (x) -(x,a)
```

In the body of the procedure, x is at lexical depth 0 and a is at depth 1.

The value of this expression is

```
(value-of
  <<let a = 5 in proc (x) -(x,a)>>
  ρ)
= (value-of <<proc (x) -(x,a)>>
    (extend-env a ⌈5⌉ ρ))
= (proc-val (procedure x <<-(x,a)>> [a=⌈5⌉]ρ))
```

The body of this procedure can only be evaluated by apply-procedure:

```
(apply-procedure
  (procedure x <<-(x,a)>> [a=⌈5⌉]ρ)
  ⌈7⌉)
= (value-of <<-(x,a)>>
    [x=⌈7⌉][a=⌈5⌉]ρ)
```

So again every variable is found in the environment at the place predicted by its lexical depth.

3.7 Implementing Lexical Addressing

We now implement the lexical-address analysis we sketched above. We write a procedure translation-of-program that takes a program and removes all the variables from the declarations, and replaces every variable reference by its lexical depth.

For example, the program

```
let x = 37
in proc (y)
    let z = -(y,x)
    in -(x,y)
```

is translated to

```
#(struct:a-program
   #(struct:nameless-let-exp
      #(struct:const-exp 37)
      #(struct:nameless-proc-exp
         #(struct:nameless-let-exp
            #(struct:diff-exp
               #(struct:nameless-var-exp 0)
               #(struct:nameless-var-exp 1))
            #(struct:diff-exp
               #(struct:nameless-var-exp 2)
               #(struct:nameless-var-exp 1))))))
```

We then write a new version of value-of-program that will find the value of such a nameless program, without putting variables in the environment.

3.7.1 The Translator

We are writing a translator, so we need to know the source language and the target language. The target language will have things like nameless-var-exp and nameless-let-exp that were not in the source language, and it will lose the things in the source language that these constructs replace, like var-exp and let-exp.

We can either write out define-datatype's for each language, or we can set up a single define-datatype that includes both. Since we are using SLLGEN as our front end, it is easier to do the latter. We add to the SLLGEN grammar the productions

Expression ::= `%lexref` *number*

```
nameless-var-exp (num)
```

Expression ::= `%let` *Expression* `in` *Expression*

```
nameless-let-exp (exp1 body)
```

Expression ::= `%lexproc` *Expression*

```
nameless-proc-exp (body)
```

We use names starting with % for these new constructs because % is normally the comment character in our language.

Our translator will reject any program that has one of these new nameless constructs (nameless-var-exp, nameless-let-exp, or nameless-proc-exp), and our interpreter will reject any program that has one of the old nameful constructs (var-exp, let-exp, or proc-exp) that are supposed to be replaced.

To calculate the lexical address of any variable reference, we need to know the scopes in which it is enclosed. This is *context* information, so it should be like the inherited attributes in section 1.3.

So translation-of will take two arguments: an expression and a *static environment*. The static environment will be a list of variables, representing the scopes within which the current expression lies. The variable declared in the innermost scope will be the first element of the list.

For example, when we translate the last line of the example above, the static environment should be

```
(z y x)
```

So looking up a variable in the static environment means finding its position in the static environment, which gives a lexical address: looking up x will give 2, looking up y will give 1, and looking up z will give 0.

Entering a new scope will mean adding a new element to the static environment. We introduce a procedure extend-senv to do this.

Since the static environment is just a list of variables, these procedures are easy to implement and are shown in figure 3.15.

```
Senv = Listof(Sym)
Lexaddr = N

empty-senv : () → Senv
(define empty-senv
  (lambda ()
    '()))

extend-senv : Var × Senv → Senv
(define extend-senv
  (lambda (var senv)
    (cons var senv)))

apply-senv : Senv × Var → Lexaddr
(define apply-senv
  (lambda (senv var)
    (cond
      ((null? senv)
       (report-unbound-var var))
      ((eqv? var (car senv))
       0)
      (else
        (+ 1 (apply-senv (cdr senv) var))))))
```

Figure 3.15: Implementation of static environments

For the translator, we have two procedures, translation-of, which handles expressions, and translation-of-program, which handles programs.

We are trying to translate an expression e which is sitting inside the declarations represented by senv. To do this, we recursively copy the tree, as we did in exercises 1.33 or 2.26, except that

1. Every var-exp is replaced by a nameless-var-exp with the right lexical address, which we compute by calling apply-senv.
2. Every let-exp is replaced by a nameless-let-exp. The right-hand side of the new expression will be the translation of the right-hand side of the old expression. This is in the same scope as the original, so we translate it in the same static environment senv. The body of the new expression will be the translation of the body of the old expression. But the body now lies in a new scope, with the additional bound variable *var*. So we translate the body in the static environment (extend-senv *var* senv).
3. Every proc-exp is replaced by a nameless-proc-exp, with the body translated with respect to the new scope, represented by the static environment (extend-senv *var* senv).

The code for translation-of is shown in figure 3.16.

```
translation-of : Exp × Senv → Nameless-exp
(define translation-of
  (lambda (exp senv)
    (cases expression exp
      (const-exp (num) (const-exp num))
      (diff-exp (exp1 exp2)
        (diff-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)))
      (zero?-exp (exp1)
        (zero?-exp
          (translation-of exp1 senv)))
      (if-exp (exp1 exp2 exp3)
        (if-exp
          (translation-of exp1 senv)
```

```
          (translation-of exp2 senv)
          (translation-of exp3 senv)))
      (var-exp (var)
        (nameless-var-exp
          (apply-senv senv var)))
      (let-exp (var exp1 body)
        (nameless-let-exp
          (translation-of exp1 senv)
          (translation-of body
            (extend-senv var senv))))
      (proc-exp (var body)
        (nameless-proc-exp
          (translation-of body
            (extend-senv var senv))))
      (call-exp (rator rand)
        (call-exp
          (translation-of rator senv)
          (translation-of rand senv)))
      (else
        (report-invalid-source-expression exp)))))
```

Figure 3.16: The lexical-address translator

The procedure translation-of-program runs translation-of in a suitable initial static environment.

```
translation-of-program : Program → Nameless-program
(define translation-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (a-program
          (translation-of exp1 (init-senv)))))))

init-senv : () → Senv
(define init-senv
  (lambda ()
    (extend-senv 'i
      (extend-senv 'v
        (extend-senv 'x
          (empty-senv))))))
```

3.7.2 The Nameless Interpreter

Our interpreter takes advantage of the predictions of the lexical-address analyzer to avoid explicitly searching for variables at run time.

Since there are no more variables in our programs, we won't be able to put variables in our environments, but since we know exactly where to look in each environment, we don't need them!

Our top-level procedure will be run:

```
run : String → ExpVal
(define run
  (lambda (string)
    (value-of-program
      (translation-of-program
        (scan&parse string)))))
```

Instead of having full-fledged environments, we will have nameless environments, with the following interface:

```
nameless-environment? : SchemeVal → Bool
empty-nameless-env      :()→ Nameless-env
extend-nameless-env     : Expval × Nameless-env → Nameless-env
apply-nameless-env       : Nameless-env × Lexaddr → DenVal
```

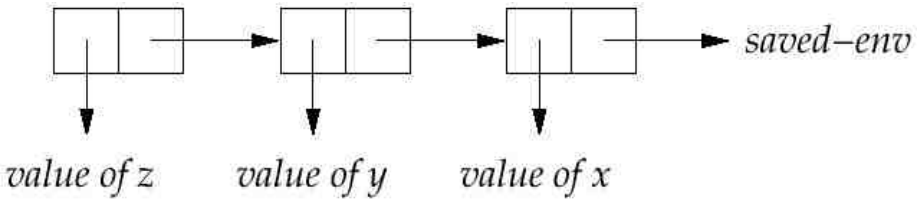
We can implement a nameless environment as a list of denoted values, so that apply-nameless-env is simply a call to list-ref. The implementation is shown in figure 3.17.

```
nameless-environment? : SchemeVal → Bool
(define nameless-environment?
  (lambda (x)
```

```
    ((list-of expval?) x)))

empty-nameless-env : () → Nameless-env
(define empty-nameless-env
  (lambda ()
    '()))

extend-nameless-env : ExpVal × Nameless-env → Nameless-env
(define extend-nameless-env
  (lambda (val nameless-env)
    (cons val nameless-env)))

apply-nameless-env : Nameless-env × Lexaddr → ExpVal
(define apply-nameless-env
  (lambda (nameless-env n)
    (list-ref nameless-env n)))
```

Figure 3.17: Nameless environments

At the last line of the example on page 93, the nameless environment will look like



Having changed the environment interface, we need to look at all the code that depends on that interface. There are only two things in our interpreter that use environments: procedures and value-of.

The revised specification for procedures is just the old one with the variable name removed.

```
(apply-procedure (procedure body ρ) val)
= (value-of body (extend-nameless-env val ρ))
```

We can implement this by defining

```
procedure : Nameless-exp × Nameless-env → Proc
(define-datatype proc proc?
  (procedure
    (body expression?)
    (saved-nameless-env nameless-environment?)))

apply-procedure : Proc × ExpVal → ExpVal
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (body saved-nameless-env)
        (value-of body
          (extend-nameless-env val saved-nameless-env))))))
```

Now we can write value-of. Most cases are the same as in the earlier interpreters except that where we used env we now use nameless-env. We do have new cases, however, that correspond to var-exp, let-exp, and proc-exp, which we replace by cases for nameless-var-exp, nameless-let-exp, and nameless-proc-exp, respectively. The implementation is shown in figure 3.18. A nameless-var-exp gets looked up in the environment. A nameless-let-exp evaluates its right-hand side exp_1, and then evalutes its body in an environment extended by the value of the right-hand side. This is just what an ordinary let does, but without the variables. A nameless-proc produces a proc, which is then applied by apply-procedure.

```
value-of : Nameless-exp × Nameless-env → ExpVal
(define value-of
  (lambda (exp nameless-env)
    (cases expression exp

      (const-exp (num) …as before…)
      (diff-exp (exp1 exp2) …as before…)
      (zero?-exp (exp1) …as before…)
      (if-exp (exp1 exp2 exp3) …as before…)
      (call-exp (rator rand) …as before…)
```

```
      (nameless-var-exp (n)
        (apply-nameless-env nameless-env n))

      (nameless-let-exp (exp1 body)
        (let ((val (value-of exp1 nameless-env)))
          (value-of body
            (extend-nameless-env val nameless-env))))

      (nameless-proc-exp (body)
        (proc-val
          (procedure body nameless-env)))

      (else
        (report-invalid-translated-expression exp)))))
```

Figure 3.18: `value-of` for the nameless interpreter

Last, here's the new value-of-program:

```
value-of-program : Nameless-program → ExpVal
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-nameless-env))))))
```

Exercise 3.38 [*] Extend the lexical address translator and interpreter to handle cond from exercise 3.12.

Exercise 3.39 [*] Extend the lexical address translator and interpreter to handle pack and unpack from exercise 3.18.

Exercise 3.40 [**] Extend the lexical address translator and interpreter to handle letrec. Do this by modifying the context argument to translation-of so that it keeps track of not only the name of each bound variable, but also whether it was bound by letrec or not. For a reference to a variable that was bound by a letrec, generate a new kind of reference, called a nameless-letrec-var-exp. You can then continue to use the nameless environment representation above, and the interpreter can do the right thing with a nameless-letrec-var-exp.

Exercise 3.41 [**] Modify the lexical address translator and interpreter to handle let expressions, procedures, and procedure calls with multiple arguments, as in exercise 3.21. Do this using a nameless version of the ribcage representation of environments (exercise 2.11). For this representation, the lexical address will consist of two nonnegative integers: the lexical depth, to indicate the number of contours crossed, as before; and a position, to indicate the position of the variable in the declaration.

Exercise 3.42 [***] Modify the lexical address translator and interpreter to use the trimmed representation of procedures from exercise 3.26. For this, you will need to translate the body of the procedure not (extend-senv *var senv*), but in a new static environment that tells exactly where each variable will be kept in the trimmed representation.

Exercise 3.43 [***] The translator can do more than just keep track of the names of variables. For example, consider the program

```
let x = 3
in let f = proc (y) -(y,x)
   in (f 13)
```

Here we can tell statically that at the procedure call, f will be bound to a procedure whose body is -(y,x), where x has the same value that it had at the procedure-creation site. Therefore we could avoid looking up f in the environment entirely. Extend the translator to keep track of "known procedures" and generate code that avoids an environment lookup at the call of such a procedure.

Exercise 3.44 [***] In the preceding example, the only use of f is as a known procedure. Therefore the procedure built by the expression proc (y) -(y,x) is never used. Modify the translator so that such a procedure is never constructed.