if \( n \) is a multiple of 10 then \( \text{decade}(n / 10) \)
else if \( n < 20 \) then \( \text{spell}(n) \)
else \( \text{decade}(n / 10) \) \( \| \) \( \text{spell}(n \mod 10) \)

Functions \( \text{spell} \) and \( \text{decade} \) satisfy the equalities:

\[
\text{spell}(1) = \text{one}, \ \text{spell}(2) = \text{two}, \ldots, \ \text{spell}(19) = \text{nineteen} \\
\text{decade}(0) = \text{zero}, \ \text{decade}(1) = \text{ten}, \ldots, \ \text{decade}(9) = \text{ninety}
\]

The rule associated with the production \( P := D \) is as follows:

\[
P \cdot \text{trans} := \text{if } D \cdot \text{val} = 0 \text{ then } \text{decade}(P \cdot \text{in}) \\
\ \ \ \ \ \text{else if } P \cdot \text{in} \leq 1 \text{ then } \text{spell}(10 \cdot P \cdot \text{in} + D \cdot \text{val}) \\
\ \ \ \ \ \ \ \ \ \ \ \ \ \text{else } \text{decade}(P \cdot \text{in}) \ \| \ \text{spell}(D \cdot \text{val})
\]

The synthesized attribute \( P \cdot \text{trans} \) is defined in terms of the inherited attribute \( P \cdot \text{in} \) and the synthesized attribute \( D \cdot \text{val} \).

Inherited attributes make the flow of information explicit. They are not necessary, however, since anything that can be defined using inherited and synthesized attributes can be defined using synthesized attributes alone. The attribute grammar in Fig. 13.6 can be modified into an equivalent grammar using synthesized attributes alone. The reason for considering it is that it illustrates how inherited attributes can be used.

### 13.3 Natural Semantics

**Natural semantics** associates logical rules with the syntax of a language. The logical rules can be used to deduce the meaning of a construct. The rules translate directly into Prolog, so they can be run. Scheme implementations based on such rules appear in later sections. Natural semantics sidesteps issues of parsing and evaluation order, so it allows the semantics of a language to be captured in a small set of rules.

**A Calculator**

We begin with a preliminary version of natural semantics that can be thought of as associating synthesized attributes with constructs in a language. In order to do this self-contained, we use the abstract syntax \( \text{num}(\text{val}) \) for numbers, where \( \text{num} \) is a token, and \( \text{val} \) is its associated value. Expressions therefore have the form


\[
E ::= \text{num}(\text{w}l) \\
| \text{plus } E_1 E_2 \\
| \text{times } E_1 E_2
\]

The preliminary version of natural semantics can handle these expressions, but it needs to be extended to handle variables, whose values depend on the context.

The value of \text{plus } E_1 E_2 is the sum of the values of \(E_1\) and \(E_2\). This rule can be written as

\[
\begin{align*}
E_1 : v_1 & \quad E_2 : v_2 \\
\text{plus } E_1 E_2 & : v_1 + v_2
\end{align*}
\]

(sum)

In words, if \(E_1\) has value \(v_1\) and \(E_2\) has value \(v_2\), then \text{plus } \(E_1 E_2\) has value \(v_1 + v_2\). To the left of the colon in the formula

\[
E : v
\]

is an expression \(E\) whose value \(v\) appears to the right of the colon. Thus, in the formula

\[
\text{plus } E_1 E_2 : v_1 + v_2
\]

the expression \text{plus } \(E_1 E_2\) belongs to the defined language, and \(v_1 + v_2\) belongs to the defining language.

The rules for numbers, sums, and products are collected in Fig. 13.7. Each rule has a name, in parentheses to its right. As usual in logical rules, conditions appear above a line and the conclusion appears below the line. If there are no conditions, as in the rule for numbers, then the conclusion appears by itself, without a line. Rules without conditions are called axioms. The axiom for numbers says that the value of a number \text{num}(\text{w}l) is simply its associated attribute, \text{w}l.

Environments Bind Names to Values

The value of an expression \text{plus } a b depends on the values of the variables \(a\) and \(b\). To handle variables, we introduce environments, which bind a variable to a value.

Environments will be treated as objects with two operations:

1. \text{bind}(x, v, \text{env}) is a new environment that binds variable \(x\) to value \(v\); the bindings of all other variables are as in the environment \text{env}.
2. \text{lookup}(x, \text{env}) is the value bound to variable \(x\) in environment \text{env}.

The empty environment \(\text{nil}\) binds no variables.

Environments are shown explicitly in natural semantics by writing formulas called sequents, of the form

\[
\begin{align*}
\text{env} \vdash E_1 : v_1 & \quad \text{env} \vdash E_2 : v_2 \\
\text{env} \vdash \text{plus } E_1 E_2 : v_1 + v_2
\end{align*}
\]

Let Bindings

The defined language in Fig. 13.8 contains numbers, variables, sum, products, and let expressions. The rules for numbers, sums, and products are obtained by adding environments to the rules in Fig. 13.7.

Read the following axiom as, "In environment \text{env}, variable \(x\) has value \text{lookup}(x, \text{env})";

\[
\begin{align*}
\text{env} \vdash x : \text{lookup}(x, \text{env})
\end{align*}
\]

(variable)

Read the following rule for let expressions backwards, from the conclusion to the conditions:

\[
\begin{align*}
\text{env} \vdash E_1 : v_1 & \quad \text{bind}(x, v_1, \text{env}) \vdash E_2 : v_2 \\
\text{env} \vdash \text{let } x \equiv E_1 \text{ in } E_2 : v_2
\end{align*}
\]

(let)

The value of the expression \text{let } x \equiv E_1 \text{ in } E_2 is \(v_2\) in environment \text{env}, if

1. \(E_1\) evaluates to \(v_1\) in environment \text{env}.
2. \(E_2\) evaluates to \(v_2\) in the environment with \(x\) bound to \(v_1\).

\[
\begin{align*}
\text{num}(\text{w}l) & : \text{w}l \\
E_1 : v_1 & \quad E_2 : v_2 \\
\text{plus } E_1 E_2 & : v_1 + v_2 \\
E_1 : v_1 & \quad E_2 : v_2 \\
\text{times } E_1 E_2 & : v_1 \times v_2
\end{align*}
\]


\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{natural-semantics}
\caption{A preliminary example of natural semantics.}
\end{figure}
natural semantics can be implemented readily in Prolog

A Prolog Implementation

The rules in the natural semantics of a language can be encoded directly in Prolog. Expressions in the defined language are encoded as terms, axioms are encoded as facts, and semantic rules are encoded as Prolog rules. The semantics in Fig. 13.8 leads to the Prolog rules in Fig. 13.9.

The defined language in Fig. 13.9 consists of terms of the form

\[ E ::= \text{num}(\text{nat}) \mid \text{var}(\text{atom}) \mid \text{plus}(E_1, E_2) \mid \text{times}(E_1, E_2) \mid \text{let}(\text{var}(\text{atom}), E_1, E_2) \]

A little "syntactic sugar" — a minor change in syntax — converts these terms into expressions in the defined language of Fig. 13.8.

A sequent \( env \vdash E : v \) is encoded as \( \text{seq}(\text{Env}, E, v) \). Variables in Prolog begin with uppercase letters, so \( env \) is encoded as \( \text{Env} \) and \( v \) is encoded as \( V \).

The axiom

\[ env \vdash \text{num}(\text{nat}) : \text{nat} \] (number)

in Fig. 13.8 leads to the following fact in Fig. 13.9:

\[ \text{seq}(\text{Env}, \text{num}(\text{Val})), \text{Val}) \] (number)
\[ \text{seq}(\text{Env}, \text{plus}(E_1, E_2), V) \vdash \text{seq}(\text{Env}, E_1, V_1), \text{seq}(\text{Env}, E_2, V_2), V \text{ is } V_1 + V_2 \] (sum)
\[ \text{seq}(\text{Env}, \text{times}(E_1, E_2), V) \vdash \text{seq}(\text{Env}, E_1, V_1), \text{seq}(\text{Env}, E_2, V_2), V \text{ is } V_1 \times V_2 \] (product)
\[ \text{seq}(\text{Env}, \text{var}(X), V) \vdash \text{lookup}(X, \text{Env}, V) \] (variable)
\[ \text{seq}(\text{Env}, \text{let}(\text{var}(X), E_1, E_2), V_3) \vdash \text{seq}(\text{Env}, E_1, V_1), \text{seq}(\text{bind}(X, V_1, \text{Env}), E_2, V_2) \] (let)
\[ \text{lookup}(X, \text{bind}(X, V_1, \text{Env}), V) \vdash \text{lookup}(X, \text{Env}, V) \]

Figure 13.9 A Prolog version of the natural semantics in Fig. 13.8.
The following rule in the natural semantics

\[
\begin{align*}
\text{env} &\vdash E_1 : v_1 \quad \text{env} \vdash E_2 : v_2 \\
\text{env} &\vdash (\text{plus } E_1, E_2) : v_1 + v_2
\end{align*}
\]

leads to the Prolog rule

\[
\text{seq}(	ext{Env}, \text{plus}(E_1, E_2), V) :- \\
\text{seq}(	ext{Env}, E_1, V_1), \text{seq}(	ext{Env}, E_2, V_2), V \leftarrow V_1 + V_2.
\]

The conclusion appears before the conditions in Prolog, with \( \leftarrow \) appearing between them. The goal

\[
V \leftarrow V_1 + V_2
\]

evaluates the expression \( V_1 + V_2 \) and associates its value with \( V \).

**Example 13.4** Environments are represented simply as terms in Fig. 13.9.

The environment

\[
\text{bind}(y, 2, \text{nil})
\]

binds variable \( \text{var}(y) \) to the value \( 2 \).

The following query uses the rules in Fig. 13.9:

\[
? - E = \text{let}([\text{var}(y), \text{num}(2), \text{var}(y)]), \text{seq}(	ext{nil}, E, V) \\
\quad E = \text{let}([\text{var}(y), \text{num}(2), \text{var}(y)]) \\
\quad V = 2
\]

For convenience, this query uses variable \( E \) to refer to the expression to be evaluated; the actual evaluation is done in response to the query

\[
\text{seq}(	ext{nil}, E, V).
\]

The result, \( 2 \), is the value computed by Prolog for \( V \).

Another example is

\[
? - Y = \text{var}(y), S = \text{let}(Y, \text{num}(2), \text{plus}(Y, Y)), \text{seq}(	ext{nil}, E, V) \\
\quad Y = \text{var}(y) \\
\quad S = \text{let}([\text{var}(y), \text{num}(2), \text{plus}(\text{var}(y), \text{var}(y))]) \\
\quad V = 4
\]

### 13.4 Denotational Semantics

The term "denotational" comes from the verb "denote." In the denotational approach, constructs in a language denote or represent meanings, which are usually functions. The meaning of an expression \( a + b \) will be a function from environments to values.

A denotational semantics is in two parts:

1. **Domains** are like types; they identify the syntactic and semantic objects relevant to a language. For the language of prefix expressions, the semantic objects are values and environments; the syntactic objects are expressions, variables, constants, and operators.

2. **Semantic rules** synthesize the meaning of a construct in terms of that of its components.

A fundamental contribution of the work on denotational semantics is a firm mathematical foundation for domains and semantic rules. These mathematical aspects are beyond the scope of this book.

Since let-expressions and environments have already been introduced in Section 13.3, we are ready to write the rules for let-expressions.

The meaning of an expression \( E \), written as \( [\mathcal{E}] \), is a function from environments to values. A function is applied by writing it next to its argument, as in Chaps. 8 and 9, and \( fab \) is equivalent to \( (f(a))b \), the application of \( f \) to \( a \) to \( b \).

The value of \( E \) with environment \( \text{env} \) is therefore given by

\[
[\mathcal{E}] \text{env}
\]

With this notation, the value of a variable \( x \) with environment \( \text{env} \) is

\[
[x] \text{env} = \text{lookup}(x, \text{env})
\]

Read this rule as, "when the meaning of \( x \) as an expression is applied to environment \( \text{env} \), we get the value \( \text{lookup}(x, \text{env}) \)."

The rules for sums and products are

\[
\begin{align*}
[\text{plus } E_1, E_2] \text{env} & = [E_1] \text{env} + [E_2] \text{env} \\
[\text{times } E_1, E_2] \text{env} & = [E_1] \text{env} \times [E_2] \text{env}
\end{align*}
\]

In either case, the same environment \( \text{env} \) is used for an expression and its subexpressions.
In the rule for let-expressions, note that environment \( env \) is used for \( E_1 \) and that a modified environment is used for \( E_2 \):

\[
\text{\( \langle \text{let } x = E_1 \ \text{in } E_2 \rangle \ env = [E_2] \ \text{bind}(x, [E_1] \ env, env) \)}
\]

The value of the left-expression is the value of the subexpression \( E_2 \) in an environment with \( x \) bound to \( [E_1] \ env \), the value of \( E_1 \).

### 13.5 A CALCULATOR IN SCHEME

Sections 13.5–13.8 lead up to an interpreter for a small subset of Scheme. This section introduces notation, by redoing expression evaluation. Section 13.6 explores lexical scope. The interpreter itself appears in Sections 13.7 and 13.8.

Expressions consisting of numbers, sums, and products are simple enough that their interpreter can be shown in its entirety in Fig. 13.10. Its main routine calc uses predicates constant?, sum?, and product? to analyze the syntax of expression \( E \). Once the syntax of \( E \) is recognized, a corresponding function is called to evaluate \( E \).

```scheme
(define (calc E) ; the main routine
  (cond ((constant? E) (calc-constant E))
        ((sum? E) (calc-sum E))
        ((product? E) (calc-product E))
        (else (error "calc: cannot parse" E)) )

(define constant? number?) ; constants are numbers
(define (calc-constant E) E) ; evaluating to themselves

(define (sum? E) ; a sum is a list with head plus
  (and (pair? E) (equal? 'plus (car E)) )
(define (calc-sum E) ; evaluate subexpressions and apply +
  (apply + (map calc (cdr E))))

(define (product? E) ; a product is a list with head times
  (and (pair? E) (equal? 'times (car E)) )
(define (calc-product E) ; evaluate subexpressions and apply *
  (apply * (map calc (cdr E))))

Figure 13.10 An interpreter for numbers, sums, and products.
```

Constants in the defined language are numbers. Predicate constant? is therefore implemented by number?:

```scheme
(define constant? number?)
```

Scheme interprets a number by returning its value, so calc-constant simply returns its argument:

```scheme
(define (calc-constant E) E)
```

Scheme has a built-in lexical analyzer that converts the character representation of a number into an internal form, so we can think of numbers as numbers, not as tokens with associated values.

A sum has the form

```scheme
(plus E_1 E_2 \ldots E_k)
```

Predicate sum? returns true if it is applied to a list, and the head of the list is the symbol plus:

```scheme
(define (sum? E)
  (and (pair? E) (equal? 'plus (car E)) )
)
```

The interpreter evaluates a sum by evaluating each of the subexpressions and then using the Scheme function + to compute their sum. The subexpressions of \( E \) are given by (cdr E). A list consisting of their values is computed by

```scheme
(map calc (cdr E))
```

which applies calc to each element of the list of subexpressions (cdr E). The function apply in Scheme satisfies the equality

```scheme
(apply + (list E_1 E_2 \ldots E_k)) = (+ E_1 E_2 \ldots E_k)
```

Function apply is used by calc-sum to add the values of the subexpressions of \( E \):

```scheme
(define (calc-sum E)
  (apply + (map calc (cdr E))))
```

Since calc-sum is called only on sums, it does not check whether its argument \( E \) is indeed a sum.

Products are handled similarly.
13.6 LEXICALLY SCOPED LAMBDA EXPRESSIONS

With all functions, including those denoted by lambda expressions, there is a distinction between the environment in which the function is defined and the environment in which the function is applied. Call these the definition and activation environments, respectively.

For the lambda expression \( \text{lambda}(y) \, (\ast \, y \, z) \) in

\[
\text{(let ([(f (\text{lambda}(y) \, (\ast \, y \, z)))) ; definition environment} \\
\text{...} \\ 
\text{(f 2))}) ; activation environment
\]

the definition environment is the one in which the lambda expression is bound to \( f \), and the activation environment is the one in which function \( f \) is applied during the evaluation of \((f 2)\).

A difficulty arises if a lambda expression contains free variables, as in

\[
\text{(lambda} \, y) \, (\ast \, y \, z)\)
\]

where \( z \) is free. The value of the formal parameter \( y \) will become available when this lambda expression is applied, but what is the value of the free variable \( z \)?

Under the lexical scope rules of Scheme, the value of a free variable is taken from the definition environment. Under the dynamic scope rules of other Lisp dialects, the value of a free variable is taken from the activation environment. The following example explores the distinction between these scope rules.

Example 13.5 Most expressions have the same values under both the lexical and dynamic scope rules. This example develops an expression that has different values under the two scope rules.

The essential building block is a lambda expression containing a free variable \( z \):

\[
\text{(lambda} \, y) \, (\ast \, y \, z)\)
\]

The following fragment binds \( f \) to this function and later applies it to the actual parameter \( 2 \):

\[
\text{(let ([(f (lambda} \, y) \, (\ast \, y \, z)))] ; definition environment} \\
\text{...} \\ 
\text{(f 2))}) ; activation environment
\]

The two pieces missing from this fragment are bindings for \( z \). We will choose them to ensure that the value of \( z \) in the definition environment is different from its value in the activation environment.

The definition environment binds \( z \) to \( 0 \):

\[
\text{(let ([(z 0)])} \\
\text{(let ([(f (lambda} \, y) \, (\ast \, y \, z)))]} \\
\text{...} \\
\text{(f 2))})
\]

Already, we can say that Scheme will evaluate \((f 2)\) to \( 0 \), independent of the missing piece. The complete expression is

\[
\text{(let ([(z 0)])} \\
\text{(let ([(f (lambda} \, y) \, (\ast \, y \, z)))]} \\
\text{(let ([(z 1)])} \\
\text{(f 2))})
\]

Lisp dialects that use dynamic scope evaluate this expression to \( 2 \) instead of \( 0 \).

Natural Semantics of Lambda Expressions

In the natural semantics in Fig. 13.11, the definition and activation environments are denoted by the variables def-env and act-env.

Before studying the rules, let us reexamine the example

\[
\text{(let ([(f (lambda} \, y) \, (\ast \, y \, z)))] ; definition environment} \\
\text{...} \\
\text{(f 2))}) ; activation environment
\]

The treatment of the lambda expression in this example can be explained as follows:

1. When the lambda expression is bound to \( f \), the value of the free variable \( z \) is frozen.
2. The body \((\ast \, y \, z)\) is evaluated when the value of the formal parameter \( y \) becomes known. That is, a multiplication occurs during the evaluation of \((f 2)\).
The two rules in Fig. 13.11 correspond to these two steps. Rule (lambda) simply saves a lambda expression and its definition environment into a data structure called a closure. A closure formalizes the notion of freezing the values of the free variables in the lambda expression; whenever the value of a free variable is needed, it will be taken from the saved environment.

An explanation of rule (apply-lambda) is as follows. (Evaluation takes place in the activation environment, unless specified otherwise.) The expression \( (F \ A) \), the application of \( F \) to \( A \), has value \( v \) if three conditions hold:

1. \( F \) evaluates to a closure. The lambda expression in the closure has formal parameter \( x \) and body \( B \). The environment saved in the closure is \( def\text{-}env \).
2. \( A \) evaluates to \( a \).
3. The body \( B \) of the lambda expression evaluates to \( v \) in an environment \( bind(x, a, def\text{-}env) \). In this environment, the formal parameter \( x \) is bound to the value \( a \) of the actual parameter. The definition environment \( def\text{-}env \), saved in the closure, is used for the free variables in \( B \).

\[ bind(x, a, def\text{-}env) \]

Example 13.6 An application of the rule (apply-lambda) yields the value 0 for the subexpression \((\text{let (x z) \( (\text{let (y (\text{lambda (y) (* y z)))})\)) \text{ (let (z 1)) \text{ (f 2))}}\)\) in the expression from Example 13.5:

\[
\begin{align*}
\text{Let (x z) } & \text{ (let (y (\text{lambda (y) (* y z)))}) \\
\text{ (let (z 1)) } & \text{ (f 2))}
\end{align*}
\]

Rule (lambda)

\[
\text{def-env } \vdash (\text{lambda (x) E}) : \text{closure((lambda (x) E), def-env)}
\]

Rule (apply-lambda)

\[
\begin{align*}
\text{act-env } \vdash F : \text{closure((lambda (x) B), def-env)} \\
\text{act-env } \vdash A : a \\
\text{bind(x, a, def-env) } \vdash B : v \\
\text{act-env } \vdash (F \ A) : v
\end{align*}
\]

Figure 13.11 Call-by-value evaluation of lexically scoped lambda expressions.

Suppose that evaluation starts in the empty environment \( nil \). The definition environment for the lambda expression is therefore \( bind(z, 0, nil) \). The details of the activation environment \( act\text{-}env \) are not shown. All we need to know is that \( act\text{-}env \) binds \( z \) to the closure shown in the following:\(^{2}\)

\[
\begin{align*}
\text{act-env } \vdash z : \text{closure((lambda (y) (* y z)), bind(z, 0, nil))} \\
\text{act-env } \vdash 2 : 2 \\
\text{bind(y, 2, bind(z, 0, nil)) } \vdash (* y z) : 0 \\
\text{act-env } \vdash (\text{let (x (\text{lambda (y) (* y z))) \text{ (let (z 1)) \text{ (f 2))}))} \\
\end{align*}
\]

An Implementation of Environments

Environments can be implemented in Scheme using key-value pairs called association lists or a-lists. The association list

\[
((a 1) (b 2) (c 3) \cdots)
\]

implements an environment that binds \( a \) to 1, binds \( b \) to 2, and so on.

Association lists and the following operations on them were introduced in Section 10.3:

1. \( \text{bind} \) returns an association list with a new binding for a key.
2. \( \text{bind-all} \) binds keys in a list \( \text{keys} \) to values in a corresponding list \( \text{values} \).
3. \( \text{assoc} \) returns the most recent binding for a key.

The code for \( \text{bind} \) and \( \text{bind-all} \) is repeated here; see Section 10.3 for details:

\[
\begin{align*}
\text{(define (bind key value env)} \\
\text{(cons (list key value) env))} \\
\text{(define (bind-all keys values env)} \\
\text{(append (map list keys values) env))}
\end{align*}
\]

13.7 AN INTERPRETER

This section develops an interpreter \( \text{val} \) that takes two parameters: an expression \( E \) to be evaluated and an environment \( env \) holding values for the variables. Function \( \text{val} \) has a case for each of the following constructs: numbers, quoted items, variables, conditionals, let expressions, lambda expressions, and function applications. Function applications are kept to the end because any

\(^{2}\) Technically, we should use the abstract syntax \( \text{num(2)} \) instead of 2 in the rules.
list that does not match one of the earlier forms is by default treated as a function application.

\[
\begin{align*}
\text{(define (val E env)} & \text{(val-quote E env))} \\
\text{(cond \{(constant? E)} & \text{(val-constant E env)} \\
\text{(quote? E)} & \text{(val-quote E env)} \\
\text{(variable? E)} & \text{(val-variable E env)} \\
\text{((if? E)} & \text{(val-if E env)} \\
\text{((let? E)} & \text{(val-let E env)} \\
\text{((lambda? E)} & \text{(val-lambda E env)} \\
\text{((application? E)} & \text{(val-application E env)} \\
\text{(else (error \quote{val: cannot parse} E))})}
\end{align*}
\]

Now we examine the constructs one by one. The representation of a construct is examined only by its predicate, its evaluation function, and any supporting functions. For example, the representation of a let expression is examined only by the predicate let? and the evaluation function val-let. Supporting functions will be defined only for lambda expressions.

**Constants**

Constants are numbers.

\[
\begin{align*}
\text{(define constant? number?)}
\end{align*}
\]

Constants evaluate to themselves:

\[
\begin{align*}
\text{(define (val-constant E env) E)}
\end{align*}
\]

**Quoted Items**

The syntax of a quoted item is

\[
\begin{align*}
\text{(quote item)}
\end{align*}
\]

Hence, quote? checks whether its argument is a pair with quote as its first element:

\[
\begin{align*}
\text{(define (quote? E) \{(starts-with? E \quote{quote})\}})}
\end{align*}
\]

\[
\begin{align*}
\text{(define \{(starts-with? E symbol)} & \text{(pair? E) (equal? symbol \quote{car E})}})
\end{align*}
\]

A quoted item is evaluated by stripping the quote and returning the item:

\[
\begin{align*}
(\text{define (val-quote E env) (cadadr E)}))
\end{align*}
\]

**Variables**

A variable is represented as a symbol:

\[
\begin{align*}
\text{(define variable? symbol?)}
\end{align*}
\]

A variable is looked up in the environment. If operation assq finds a binding, the value from it is returned; otherwise, an error is reported.

\[
\begin{align*}
\text{(define (val-variable E env)} & \text{(let \{(found (assq E env))\}}) \\
\text{((if found (cadadr found) (error \quote{value unbound} E))}})
\end{align*}
\]

**Conditionals**

A conditional has the form

\[
\begin{align*}
\text{(if E}_1 \text{ E}_2 \text{ E}_3)
\end{align*}
\]

For simplicity, predicate if? simply tests whether its argument is a list with if as its head. Further checking and error messages are left as an exercise.

\[
\begin{align*}
\text{(define (if? E) \{(starts-with? E \quote{if})\}})
\end{align*}
\]

Function val-if uses caddr to pick out the subexpressions \(E_1\), \(E_2\), and \(E_3\) in a conditional (see Fig. 13.12):

\[
\begin{align*}
\text{(define (val-if E env)} & \text{(if (val (cadadr E) env) \\
\text{(val (cadadr E) env)) \text{ (val (cadadr E) env))}})
\end{align*}
\]

\[
\begin{align*}
\text{Figure 13.12 A conditional expression (**if** \(E_1\) **if** \(E_2\) **if** \(E_3\)).}
\end{align*}
\]
**Let Expressions**

A let expression has the following form (see Fig. 13.13):

\[
\text{let } ((x_1 \ E_1) \ (x_2 \ E_2) \ \ldots \ (x_n \ E_n)) \ F
\]

Predicate `let?` returns true if its argument starts with keyword `let`:

\[
\text{(define (let? } E) \ (\text{starts-with? } E \ 'let))
\]

Function `val-let` extracts a list `vars` of variables \((x_1, x_2, \ldots, x_n)\). A corresponding list `exprs` consists of expressions \((E_1, E_2, \ldots, E_n)\). The values of these expressions are collected in list `values`, and `bind-all` is called to bind the variables to the values. Expression `F` is evaluated in the new environment after the variables are bound.

\[
\text{(define (val-let } E \ env) \ (let* \ ((vars (map car (cadr E)))) \ (exprs (map cadr (cadr E)))) \ (values (map (lambda (x) (val x env)) exprs)) \ (new-env (bind-all vars values env))) \ (val (caddr E) new-env))
\]

**Lambda Expressions**

A lambda expression has the form:

\[
(\text{lambda } \text{Formals } \ E)
\]

Figure 13.13  The structure of \((\text{let } ((x_1 \ E_1) \ (x_2 \ E_2) \ \ldots \ (x_n \ E_n)) \ F)\).
Within this environment, the symbol + evaluates to the Scheme procedure +. To distinguish between these two uses of +, initial-env includes a symbol please-add, also bound to the Scheme procedure +.

Using the Interpreter

Finally, here are some examples of the use of the interpreter:

- A number evaluates to itself in any environment, even the empty environment (λ):
  \[ \text{val } 3.14 \quad 3.14 \]

- The expression in
  \[ \text{val } (\text{please-add } 2 \ 3) \ \text{initial-env} \]
  \[ 5 \]

  is recognized as a function application. Symbol please-add evaluates to the Scheme addition procedure and 2 and 3 evaluate to themselves. The response 5 is the result of applying the addition procedure to the list of values 2 and 3.

- A lambda expression by itself evaluates to a closure:
  \[ \text{val } (\lambda (x) \ x) \ '() \]
  \[ \text{closure } (\lambda (x) \ x) \ () \]

  The closure is a list consisting of the symbol closure, the original lambda expression, and its empty definition-time environment.

- A similar example involving nested lambda expressions is
  \[ \text{val } (\lambda (x) \ (\lambda (y) \ x)) \ '() \]
  \[ \text{closure } (\lambda (x) \ (\lambda (y) \ x)) \ () \]

  The response remains the same if we give name \( K \) to this lambda expression and ask for \( K \):
  \[ \text{val } (\text{let } (\text{K } (\lambda (x) \ (\lambda (y) \ x)))) \ '() \]
  \[ \text{closure } (\lambda (x) \ (\lambda (y) \ x)) \ () \]

- Now, apply \( K \) to \( 5 \):
  \[ \text{val } (\text{let } (\text{K } (\lambda (x) \ (\lambda (y) \ x))) \ (\text{K } 5)) \ '() \]
  \[ \text{closure } (\lambda (y) \ x) \ ((\text{K } 5)) \]
13.8 AN EXTENSION: RECURSIVE FUNCTIONS

Interpreters like the one in Section 13.7 can be readily extended, making them useful for experiments with language design. This section extends the interpreter to support recursive functions.

Recursive functions will be implemented by modifying the implementation of lambda expressions. The notation for recursive functions is therefore a variant of lambda notation; it is not supported by Scheme.

A function that multiplies its parameter \( x \) by itself can be written using lambda notation:

\[
\text{lambda (x) (* x x)}
\]

This expression has the form

\[
\text{lambda (formals body)}
\]

A variant of this notation includes a name \( f \) that can appear recursively within the body:

\[
\text{rec f (lambda (formals body))}
\]

Using this syntax, the factorial function is defined by

\[
\text{rec fact (lambda (x)) (if (eq? x 0) 1 (* x (fact (- x 1))) )}
\]

(13.5)

Recursive Functions as Values

From Section 13.6, lambda expressions can be implemented as follows:

1. A lambda expression evaluates to a closure consisting of the lambda expression and its definition environment. The definition environment is saved in the closure because it provides values for the free variables in the body of the lambda expression.

2. A closure is applied as a function by evaluating the body of the lambda expression with formal parameters bound to actuals (see Section 13.6 for details).

The key difference with recursive functions is the treatment of the function name:

1. An expression \((\text{rec f lambda-expression})\) will evaluate to a data structure called a label closure, consisting of the function name in addition to the lambda expression and the definition environment. Occurrences of the name \( f \) in the body of the lambda expression correspond to recursive calls of \( f \).

2. A label closure is applied like a closure, with one key difference, a special binding for the function name. The body of the lambda expression is evaluated with \( f \) bound to the label closure itself.

Example 13.7 The expression (13.5) for the recursive factorial function has the form

\[
\text{rec fact lam}
\]

where \( lam \) represents the lambda expression in (13.5). This expression evaluates to the label closure

\[
\text{(label fact lam def-env)}
\]

With argument \( 4 \), the application

\[
((\text{rec fact lam}) 4)
\]

is implemented by evaluating

\[
(lam 4)
\]

using an environment with a binding for \( fact \):

\[
(\text{bind fact (label fact lam def-env) def-env})
\]

Since \( lam \) represents the lambda expression in (13.5) and \( 4 \) is not equal to 0, the evaluation of \( (lam 4) \) yields
A recursive function is applied by building a suitable lambda expression and an environment:

\[
\begin{align*}
\text{(define (apply-rec lab actuals)} & \text{)} \\
\text{(let+ ((fan (cadr lab)} & \text{)} \\
\text{(lam (cadadr lab)} & \text{)} \\
\text{(def-env (cadddr lab)} & \text{)} \\
\text{(new-env (bind fn lab def-env)} & \text{)} \\
\text{(val (cons lam (map (lambda (x) (list 'quote x) actuals)} & \text{)} \\
\text{new-env)} & \text{)}
\end{align*}
\]

Exercises

13.1 The natural semantics in Fig. 13.8, page 526, allows an expression in the defined language to be a number, a variable, a sum, a product, or a let expression.
   a. Extend the defined language to allow subtraction and division operations within expressions.
   b. Extend the defined language to allow a list of expressions to appear wherever a single expression is now allowed. The value of an expression list is the value of the last expression. A special variable previous refers to the value of the previous expression in a list.

13.2 Implement the semantic rules from Exercise 13.1 in Prolog.

13.3 Write an evaluator in Scheme corresponding to the Prolog rules in Fig. 13.9. Use syntactic sugar, as needed, to make the defined language easier to manipulate in Scheme.

13.4 Under dynamic scope rules, the value of a free variable is taken from the activation environment. Give semantic rules similar to those of Fig. 13.1 for call-by-value evaluation of dynamically scoped lambda expressions.

13.5 Suppose that a list built-ins consists alternately of symbols and their associated procedures, as follows:

\[
(+ + 'please-add + + 'car car ...)
\]

Using the Scheme predicates procedure? and symbol?, define a function to build the initial environment initial-env of Section 13.5, from built-ins.

13.6 Add the following constructs to the defined language of the interpreter val in Section 13.7:
   a. The cond construct of Scheme.
   b. The construct