

## 4 Predicates

A function whose output is always a boolean (i.e., *true* or *false*) is called a *predicate*. (This is just convenient terminology; there is no *predicate* type in Scheme.) This section describes some of the commonly used, built-in Scheme predicates and illustrates their use.

### 4.1 Type-Checker Predicates

Scheme includes a bunch of primitive data types, including: *number*, *boolean*, *symbol*, *null* and *function*. Scheme also includes a compound data type called *list*. For each one of these data types, Scheme includes a primitive function called a *type-checker predicate*. When a type-checker predicate is applied to some Scheme entity, it outputs *true* if that entity belongs to the indicated data type; otherwise, it outputs *false*. Thus, the type-checker predicate associated with the *number* data type outputs *true* whenever the input belongs to the *number* data type. Similarly, the type-checker predicate associated with the *list* data type outputs *true* whenever the input datum belongs to the *list* data type. And so on.

For convenience, each of these type-checker predicates has an easy-to-remember *name*. In other words, for each type-checker predicate there is an entry in the global environment that links a particular symbol with that predicate. Thus, those symbols can be used to refer to the type-checker predicates. For example, the symbol `number?` evaluates to the type-checker predicate for the *number* data type; the symbol `boolean?` evaluates to the type-checker predicate for the *boolean* data type; and so on, as illustrated by the following Interactions Window session.

```
> number?
#<procedure:number?>
> symbol?
#<procedure:symbol?>
> boolean?
#<procedure:boolean?>
> list?
#<procedure:list?>
> null?
#<procedure:null?>
> procedure?
#<procedure:procedure?>
```

Notice that the symbols mirror the names of the corresponding data types, except that the symbol associated with the type-checker predicate for functions is `procedure?`, not `function?`.<sup>8</sup>

Each type-checker predicate is a function that can be applied to a single input. That input can be any type of Scheme entity. A type-checker predicate returns *true* if that input entity is of the appropriate data type, as illustrated below.

```
(number? 3)      ~> #t
(number? #t)    ~> #f
(boolean? #f)   ~> #t
(boolean? 'x)   ~> #f
(symbol? +)     ~> #f
(symbol? '+)    ~> #t
(null? ())      ~> #t
(null? '(+ 1 2)) ~> #f
(procedure? +)  ~> #t
(procedure? '+) ~> #f
(list? '(+ 1 2)) ~> #t
(list? ())      ~> #t
(list? +)       ~> #f
```

Each of these expressions represents a list that is evaluated according to the default rule for evaluating non-empty lists. In each case, the first element of the list is a symbol that evaluates to a function, which is then applied to whatever the second element evaluates to. Notice that the `+` symbol in `(procedure? +)` evaluates to the addition

<sup>8</sup>This text uses *function* and *procedure* interchangeably; however, the term *function* seems better suited given that Scheme is typically referred to as a *functional* programming language.

function, whereas the '+ expression in (`procedure? '+`) evaluates to the + symbol. Notice too that the `list?` type-checker predicate returns *true* for any list, whether empty or non-empty.

The following is an Interactions Window session that illustrates the evaluation of some of the same expressions seen above:

```
> (number? 3)
#t
> (number? #t)
#f
> (boolean? #f)
#t
> (boolean? 'x)
#f
> (symbol? +)
#f
> (symbol? '+)
#t
```

## 4.2 Arithmetic Predicates

In addition to the primitive arithmetic functions for addition, subtraction, multiplication and division, Scheme includes several arithmetic predicates, such as *greater-than*, *less-than* and *equal*.<sup>9</sup> To enable us to refer to such predicates, each is associated with a particular symbol in the global environment.

```
> greater than
>= greater than or equal to
= equal to
< less than
<= less than or equal to
```

Each of these predicates, when applied to two numeric inputs, generates the expected boolean output, as illustrated below.<sup>10</sup>

```
(> 3 4)  ~> #f
(> 4 3)  ~> #t
(>= 4 3) ~> #t
(= 3 4)  ~> #f
(= 3 3)  ~> #t
```

The following Interactions Window session demonstrates the same evaluations:

```
> (> 3 4)
#f
> (> 4 3)
#t
> (>= 4 3)
#t
> (= 3 4)
#f
> (= 3 3)
#t
```

## 5 Defining Functions

So far, what we know about Scheme is enough to enable us to use the Interactions Window like we would a glorified calculator. There are a lot of built-in functions that we can apply to various kinds of input. Each built-in function has a more-or-less convenient *name* (i.e., for each built-in function there is an entry in the

<sup>9</sup>In other contexts, these predicates are commonly called *relational operators*.

<sup>10</sup>These predicates can also be applied to more than two inputs; however, we shall postpone discussion of such things until the section on recursion.

Global Environment that links a particular symbol to that function). However, the fun won't really begin until we can design our own functions to do whatever we want them to do. This section describes how to do this in the Scheme programming language.

## 5.1 Defining Functions vs. Applying Them to Inputs

**Example.** In a math class, you might see a function defined using an equation such as

$$f(x) = x * x$$

In this case, the name of the function is  $f$ , and we might casually describe it as the *squaring* function—because for each possible input value,  $x$ , the corresponding output value is the square of  $x$  (i.e.,  $x^2$ ).

Notice that the definition of the function,  $f$ , gives a prescription for generating appropriate output values should  $f$  ever happen to be applied to any input values. In particular, the definition of  $f$  includes an *input parameter*,  $x$ , which is used to refer to potential input values. In addition, the expression,  $x * x$ , on the righthand side of the equation indicates how to compute the corresponding output value for any given value of  $x$ . (The expression on the righthand side is sometimes referred to as the *body* of the function.) For example, if we want to know the value of  $f(3)$  (i.e., the output value corresponding to the input 3), then we first *substitute* the value 3 for  $x$  in the expression,  $x * x$ , yielding  $3 * 3$ . *Evaluating* this expression yields the output value 9. Similarly, if we want to know the value of  $f(4)$ , we first substitute the value 4 for  $x$  in the expression,  $x * x$ , yielding  $4 * 4$ , which evaluates to 16.

**Another Example.** In the preceding example, the function  $f$  took a single input value. However, we can similarly define functions that take multiple inputs. For example, the function,  $g$ , defined below takes two inputs, represented by the input parameters  $w$  and  $h$ :

$$g(w, h) = w * h$$

This function can be used to compute the area of a rectangle whose width is  $w$  and height is  $h$ . To apply this function to the input values, 3 and 7, we first substitute 3 for  $w$ , and 7 for  $h$  in the expression,  $w * h$ , yielding  $3 * 7$ . Evaluating this expression results in the desired output value, 21.

**Summary.** The definition of a function specifies how to generate appropriate output values should it ever be applied to any input values. A function definition includes a list of *input parameters* and a *body*. Once a function has been defined, it can be applied to appropriate input values as follows. First, the desired input values are substituted for the appropriate input parameters in the body of the function. Next, the resulting expression is evaluated, thereby yielding the desired output value.

**Yet Another Example.** The following defines a function,  $v$ , that can be used to compute the volume of a cone:

$$v(r, h) = \frac{1}{3}\pi r^2 h$$

It has two input parameters,  $r$  and  $h$ , that respectively represent the radius and height of the cone. To compute the volume of a cone of radius 3 and height 2, we apply the function  $v$  to the input values 3 and 2 (i.e., we compute the value  $v(3, 2)$ ), as follows. First, we substitute the values 3 and 2 for  $r$  and  $h$ , respectively, in the body,  $\frac{1}{3}\pi r^2 h$ , yielding the expression,  $\frac{1}{3}\pi * 3^2 * 2$ . Evaluating this expression yields the desired output value,  $6\pi$ .

## 5.2 The lambda Special Form

The Scheme programming language provides the `lambda` special form to enable us to define functions of our own design.

⇒ The use of the `lambda` symbol in a `lambda` special form comes from the fact that the underlying mathematical theory, originally developed in the 1930s, is called the *Lambda Calculus*.

Like any special form in Scheme, the `lambda` special form is a list whose first element is a keyword symbol—in this case, the symbol `lambda`. The second element in a `lambda` special form is used to specify the input parameter(s) for the function being defined. The rest of the elements in the `lambda` special form constitute the *body* of the function being defined. If you're wondering where the *name* of the function is specified; recall that the `define` special form is used to assign names to things in Scheme. Thus, the `lambda` special form defines everything about a function *except its name*.

⇒ The character sequence that represents a `lambda` special form is called a *lambda expression*. Thus, a *lambda expression* is a piece of syntax, whereas a `lambda` special form is a Scheme list entity.

**Example: The Squaring Function in Scheme.** Recall the mathematical definition of the squaring function:

$$f(x) = x * x$$

This mathematical definition does three things:

- It specifies a single input parameter,  $x$ , for the function being defined;
- It specifies a body,  $x * x$ , for the function being defined; and
- It specifies a name,  $f$ , for the function being defined.

In Scheme, the first two jobs are handled by the `lambda` special form. Once a function is defined, we can then use the `define` special form to give it a name. In particular, the following `lambda` expression can be used to define a squaring function in Scheme:

```
(lambda (x) (* x x))
```

This *lambda expression* represents a `lambda` special form (i.e., a Scheme list entity whose first element happens to be the `lambda` symbol). Like any special form, a `lambda` special form has its own, special rule for being evaluated. For now, suffice it to say that:

⇒ The evaluation of a `lambda` special form always results in a function entity.

Thus, if the expression, `(lambda (x) (* x x))`, is typed into the Interactions Window, DrScheme will report that a function entity has been created, as illustrated below:

```
> (lambda (x) (* x x))
#<procedure>
```

Admittedly, the character sequence generated by DrScheme is not very descriptive. It simply says that the evaluation of the corresponding `lambda` special form has resulted in a function.

⇒ At this point, it is important to stress that the function entity has been created; however, it has not yet been applied to any inputs!

We can demonstrate that the function created above behaves like a squaring function by first giving it a name and then applying it to a variety of input values. The following Interactions Window session demonstrates how to name our function:

```
> (define square (lambda (x) (* x x)))
>
```

The `define` special form is used to create an entry in the Global Environment that associates the `square` symbol with the function specified in the *lambda expression*. Recall that when a `define` special form is evaluated, the given symbol—in this case, `square`—is not evaluated; however, the given expression—in this case, `(lambda (x) (* x x))`—is evaluated. Thus, the value associated with the `square` symbol is the function that results from evaluating the given `lambda` special form, as demonstrated below.

```
> square
#<procedure:square>
```

Once we have given a name to our function, we can then use it like any of the built-in functions, as demonstrated below:

```
> (square 3)
9
> (square 4)
16
> (square -8)
64
```

Each of the above expressions is evaluated using the default rule for evaluating non-empty lists. In each case, the `square` symbol evaluates to the function that we defined earlier, which is then applied to the desired input value.

Incidentally, it is possible to define and apply a function without ever having given it a name, as the following Interactions Window session demonstrates:

```
> ((lambda (x) (* x x)) 4)
16
```

The default rule for evaluating non-empty lists is used to evaluate the above expression. In the process, each element of the list is evaluated. The first element of the list is the `lambda` special form, which evaluates to the squaring function. The second element of the list evaluates to the number *four*. The result of applying that function to that input yields the desired output, *sixteen*. Later on, we shall encounter situations where it is more convenient to use functions without bothering to name them.

**Additional Examples.** The following Interactions Window session demonstrates how to define, name, and apply functions analogous to the functions,  $g(w, h) = w * h$  and  $v(r, h) = \frac{1}{3}\pi r^2 h$ , seen earlier:

```
> (define rect-area (lambda (w h) (* w h)))
> (rect-area 2 3)
6
> (rect-area 3 8)
24
> (define cone-volume (lambda (r h) (* 1/3 3.14159 r r h)))
> (cone-volume 3 2)
18.849539999999998
> (cone-volume 10 1)
104.71966666666665
```

In the cone function, 3.14159 is used as an approximation of  $\pi$ . In addition, the expression, `(* 1/3 3.14159 r r h)`, takes advantage of the fact that the built-in multiplication function can be applied to any number of input values.

### 5.3 The Syntax and Semantics of Lambda Expressions

This section presents the syntax and semantics of lambda expressions. Initially, it restricts attention to those in which the *body* consists of a single expression; later, it addresses those in which the body consists of multiple expressions.

**The Syntax of a Lambda Expression.** A lambda expression has the following syntax:

$$(\text{lambda } (\mathcal{C}_1 \mathcal{C}_2 \dots \mathcal{C}_n) \mathcal{B})$$

where:

- each  $\mathcal{C}_i$  is a character sequence representing some Scheme symbol,  $s_i$ ; and
- $\mathcal{B}$  is a character sequence representing a Scheme entity,  $S_B$ , of any kind.

Thus,  $\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_n$  specify the *input parameters* for the lambda expression, and  $\mathcal{B}$  specifies the *body* of the lambda expression.

The following are examples of well-formed lambda expressions:

- `(lambda (x) (* x x))`
- `(lambda (w h) (* w h))`
- `(lambda (r h) (* 1/3 3.14159 r r h))`
- `(lambda (x y z) (* x (- y z)))`

For the last expression, `(x y z)` specifies the parameter list and `(* x (- y z))` specifies the body.

**The Semantics of a Lambda Expression.** The semantics of a lambda expression stipulates what Scheme entity the lambda expression represents, as well as how that Scheme entity is evaluated. As suggested by the preceding examples, a lambda expression invariably represents a list—called a `lambda` special form—and the evaluation of that list invariably results in a Scheme function entity. The semantics of the lambda expression also includes a description of the subsequent behavior of that function should it ever be applied to any input(s).

A lambda expression of the form

$$(\text{lambda } (C_1 C_2 \dots C_n) B)$$

represents a Scheme *list* whose elements are as follows:

- the `lambda` symbol;
- a list containing the  $n$  symbols,  $s_1, s_2, \dots, s_n$ ; and
- the Scheme entity,  $S_B$

where:

- for each  $i$ , the expression  $C_i$  represents the symbol,  $s_i$ ; and
- the expression  $B$  represents the Scheme entity  $S_B$ .

This list is referred to as a `lambda` special form.

By now, you should be getting used to the fact that a piece of syntax, such as `(lambda (x) (* x x))`, represents a Scheme entity—in this case, a Scheme *list* containing the `lambda` symbol and two subsidiary lists. Although it is important to be able to correctly distinguish expressions from the Scheme entities they represent, doing so can get quite tedious when writing a handout such as this. Therefore, for the sake of expository convenience, the rest of this handout shall frequently blur this distinction. Thus, we may talk of the list `(1 2 3)`, even though we really mean the list *represented by the expression* `(1 2 3)`. Similarly, we may say that the expression `(lambda (x) (* x x))` evaluates to a function, when we really mean that the *list* represented by the expression `(lambda (x) (* x x))` evaluates to a function.

### The Evaluation of a lambda Special Form.

⇒ The most important thing to know about the evaluation of a `lambda` special form is that the result is invariably a *function* entity; however, the evaluation of a `lambda` special form only *creates* the function entity; it does not *apply* it to any input(s).

For convenience, we shall refer to such function entities as `lambda` functions. Thus, a `lambda` function is a function that resulted from having evaluated a `lambda` special form.

⇒ Although evaluating a `lambda` special form only creates the corresponding function entity, it is necessary to describe what that function would do *if* it ever were applied to input values.

#### 5.3.1 Applying a lambda Function to Input Values

**Example: Applying the Squaring Function.** Consider the lambda expression, `(lambda (x) (* x x))`. As noted above, it evaluates to a Scheme function entity. When this `lambda` function is applied to some input value, say 4, the following things happen:

- A *local environment* is set up containing a single entry which associates the value 4 with the symbol `x`.
- The expression, `(* x x)`, is evaluated *with respect to the newly created local environment*. This means that any occurrence of the symbol `x` is evaluated according to the local environment, not the global environment. The evaluation of `(* x x)` therefore yields the result 16.
- That value, 16, is taken to be the output value that results from applying the `lambda` function to the input value 4.

This process is illustrated in Fig. 1.

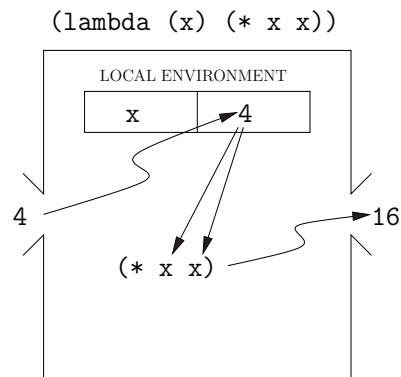


Figure 1: Applying a lambda function to the value 4

**Another Example: Computing the Volume of a Sphere.** Recall that the volume of a sphere of radius,  $r$ , is given by the function  $f(r) = \frac{4}{3}\pi r^3$ . Thus, for example, the volume of a sphere of radius 1 is  $f(1) = \frac{4}{3}\pi$ ; and the volume of a sphere of radius 2 is  $f(2) = \frac{32}{3}\pi$ .

The following Interactions Window session first creates a global variable, `pi`, to hold the value 3.14159. It then defines a function, named `sphere-volume`. Finally, it applies this function to some sample input values.

```
> (define pi 3.14159)
> (define sphere-volume (lambda (r) (* 4/3 pi r r r)))
> (sphere-volume 1)
4.188786666666666
> (sphere-volume 2)
33.51029333333333
```

Consider the evaluation of the expression, `(sphere-volume 2)`. It involves the following steps:

- First, a local environment is created containing a single entry that associates the symbol `r` with the input value 2.
- Next, the expression `(* 4/3 pi r r r)` is evaluated with respect to that local environment. In the process, the symbol `r` evaluates to 2, and the symbol `pi` evaluates to 3.14159. The resulting value is: 33.51029333333333.
- Finally, the value 33.51029333333333 is reported as the output value generated by applying the `sphere-volume` function to the input value 2.

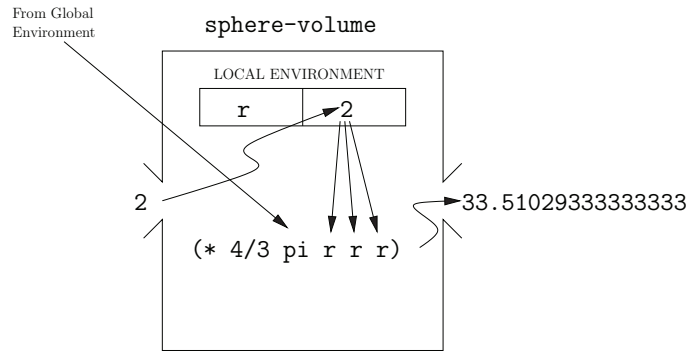
Notice that in the second step, `r`'s value came from the local environment, whereas `pi`'s value came from the global environment.

⇒ When evaluating a symbol such as `r` or `pi` with respect to a local environment, if the symbol has an entry in the local environment, that entry is used; otherwise, the symbol's value is derived from the global environment.

The evaluation of `(sphere-volume 2)` is illustrated in Fig. 2.

The following Interactions Window session (continuing from the one given above) illustrates that the existence of a global variable named `r` has no effect on the local variable that also happens to be named `r`. In contrast, changing the value of the global variable, `pi`, has disastrous effects! (That is one of many reasons why the use of global variables should be very carefully restricted!)

```
> (define r 55)
> (sphere-volume 1)
4.188786666666666
> (sphere-volume 2)
33.51029333333333
```

Figure 2: Applying the `sphere-volume` function to the value 2

```
> (define pi 100)    ;; YIKES!!
> (sphere-volume 1) ;; YIKES!!
400/3
```

Incidentally, any character sequence beginning with a semi-colon is ignored by drScheme. (Such character sequences are called *comments*.) Thus, for example, the sequence, `;; YIKES!!`, has no effect on the evaluation of the expression, `(sphere-volume 1)`, above.

**Example: More Complex Input Expressions.** So far, the examples have involved simple input expressions such as 1 or 2. This example demonstrates that complex input expressions can be handled without requiring any new evaluation tools. Consider the following Interactions Window session:

```
> (define square (lambda (x) (* x x)))
> (square (+ 2 3))
25
> (square (- 8 5))
9
> (square (square 10))
10000
```

The evaluation of the first expression simply defines a squaring function, as seen in previous examples. The evaluation of the expression, `(square (+ 2 3))`, is done according to the default rule for evaluating non-empty lists. In particular:

- The `square` symbol evaluates to the squaring function;
- The expression, `(+ 2 3)`, evaluates to 5;
- The squaring function is applied to the input value 5, generating the output value 25.

Similar remarks apply to the evaluation of the expressions, `(square (- 8 5))` and `(square (square 10))`. In each case, the input expressions, no matter how complex, are evaluated first to generate the corresponding input values. For example, the evaluation of `(square (square 10))` involves the following steps:

- The `square` symbol evaluates to the squaring function;
- The expression, `(square 10)`, evaluates to 100;
- The squaring function is applied to 100, yielding the output value, 10000.

Notice that the evaluation of the input expression, `(square 10)`, itself required using the default rule for evaluating non-empty lists. In particular:

- The `square` symbol evaluates to the squaring function;
- The expression, `10`, evaluates to 10; and
- The squaring function is applied to 10, yielding the output value 100.

Here's an example of a function that takes more than one input argument.

```
> (define discriminant
  (lambda (a b c)
    (- (* b b) (* 4 a c))))
> (discriminant 1 2 -4)
20
> (discriminant 1 0 -3)
12
```

Notice that the syntax of Scheme allows expressions to occupy multiple lines. This is quite useful when writing longer expressions. DrScheme automatically indents sub-expressions to make longer expressions easier to read. Hitting the *tab* key will automatically cause the current line to snap to the appropriate amount of indentation.

**A Lambda Expression with a Bigger Body!** The following example illustrates that a `lambda` expression can have more than one expression in its body.

```
> (define useless-function
  (lambda (input)
    input
    (* input input)
    (* input input input)
    input
    ()))
> (useless-function 35)
()
> (useless-function 888)
()
```

In this case, the body of the function includes five expressions (i.e., everything after the parameter list).

⇒ The semantics of Scheme stipulates that when a lambda function having multiple expressions in its body is subsequently applied to input(s), the expressions in the body are evaluated sequentially, one after the other.

⇒ Furthermore, the value of the last expression in the body is taken to be the output value for the function.

Thus, in the above example, each of the expressions in the body is evaluated in turn; furthermore, the value of the *last* expression serves as the output value.

⇒ This function is kind of silly since the values of the first four expressions in its body are simply thrown away.

⇒ The only way that intermediate expressions in the body of a function could have any impact is if they caused *side effects*.

In the following function, the built-in `printf` function is used to print several lines of text prior to evaluating the last expression in the function's body.

```
> (define verbose-func
  (lambda (a b)
    (printf "Hi. This is verbose-func!~\%")
    (printf "The value of the first input is: ~A~\%" a)
    (printf "The value of the second input is: ~A~\%" b)
    (printf "Their product is:~\%")
    (* a b)))
> (verbose-func 3 4)
Hi. This is verbose-func!
The value of the first input is: 3
The value of the second input is: 4
Their product is:
12
>
```

The `printf` function, and the *strings* that it operates on, are described in more detail later on. For now, take it on faith that the `printf` function generates no output (i.e., no Scheme entity that serves as the output of the function); however, it does cause the convenient side effect of printing information out to the screen.

⇒ **In this class, we will be exploring how much can be accomplished *without* using side effects. Therefore, most of the functions we write will include only a single expression in the body. However, we will allow the use of the `printf` function, which has a harmless, but very useful side effect—namely, displaying information in the Interactions Window.**