

Boolean Values and Predicate Procedures

Summary: Many of Scheme’s control structures, such as conditionals (which you’ll learn about subsequently), need mechanisms for constructing tests that return true or false. These tests are also useful for gathering information about values. In this reading, we consider Scheme’s structures that support such tests.

Contents:

- Boolean Values
- Predicates
 - Some Basic Predicates
- Boolean Procedures
- And and Or
- And and Or as Control Structures
- Keywords vs. Procedures
- Separating the World into Not False and False

Boolean Values

A *Boolean value* is a datum that reflects the outcome of a single yes-or-no test. For instance, if one were to ask Scheme to compute whether the empty list has five elements, it would be able to determine that it does not, and it would signal this result by displaying the Boolean value for “no” or “false”, which is `#f`. There is only one other Boolean value, the one meaning “yes” or “true”, which is `#t`. These are called “Boolean values” in honor of the logician George Boole, who was the first to develop a satisfactory formal theory of them. (Some folks now talk about “fuzzy logic” that includes values other than “true” and “false”, but that’s beyond the scope of this course.)

Predicates

A *predicate* is a procedure that always returns a Boolean value. A procedure call in which the procedure is a predicate performs some yes-or-no test on its arguments. For instance, the predicate `number?` (the question mark is part of the name of the procedure) takes one argument and returns `#t` if that argument is a number, `#f` if it does not. Similarly, the predicate `even?` takes one argument, which must be an integer, and returns `#t` if the integer is even and `#f` if it is odd. The names of most Scheme predicates end with question marks, and Grinnell’s computer scientists recommend this useful convention, even though it is not required by the rules of the programming language. (If you ever notice that I’ve failed to include a question mark in a predicate and you’re the first to tell me, I’ll give you some extra credit.)

Some Basic Predicates

Scheme provides a few predicates that let you test the “type” of value you’re working with.

- `number?` tests whether its argument is a number.
- `symbol?` tests whether its argument is a symbol.
- `string?` tests whether its argument is a string.
- `procedure?` tests whether its argument is a procedure.
- `boolean?` tests whether its argument is a Boolean value.
- `list?` tests whether its argument is a list. (*Warning! It can be quite expensive to determine whether or not something is a list.*)

Scheme provides one other basic predicate for working with lists.

- `null?` tests whether its argument is the “empty list” value.

Scheme provides a variety of predicates for testing equality.

- `eq?` tests whether its two arguments are identical, in the very narrow sense of occupying the same storage location in the computer’s memory. In practice, this is useful information only if at least one argument is known to be a symbol, a Boolean value, or an integer.
- `eqv?` tests whether its two arguments “should normally be regarded as the same object” (as the language standard declares). Note, however, that two lists can have the same elements without being “regarded as the same object”. Also note that in Scheme’s view the number 5, which is “exact”, is not necessarily the same object as the number 5.0, which might be an approximation.
- `equal?` tests whether its two arguments are the same or, in the case of lists, whether they have the same contents.
- `=` tests whether its arguments, which must all be numbers, are numerically equal; 5 and 5.0 are numerically equal for this purpose.

Scheme also provides many numeric predicates, some of which you may have already explored.

- `<` tests whether its arguments, which must all be numbers, are in strictly ascending numerical order. (The `<` operation is one of the few built-in predicates that does not have an accompanying question mark.)
- `>` tests whether its arguments, which must all be numbers, are in strictly descending numerical order.
- `<=` tests whether its arguments, which must all be numbers, are in ascending numerical order, allowing equality.
- `>=` tests whether its arguments, which must all be numbers, are in descending numerical order, allowing equality.
- `even?` tests whether its argument, which must be an integer, is even.
- `odd?` tests whether its argument, which must be an integer, is odd.
- `zero?` tests whether its argument, which must be a number, is equal to zero.
- `positive?` tests whether its argument, which must be a real number, is positive.
- `negative?` tests whether its argument, which must be a real number, is negative.

Boolean Procedures

Another useful Boolean procedure is `not`, which takes one argument and returns `#t` if the argument is `#f` and `#f` if the argument is anything else. For example, one can test whether the square root of 100 is unequal to the absolute value of negative twelve by giving the command

```
(not (= (sqrt 100) (abs -12)))
```

If Scheme says that the value of this expression is `#t`, then the two numbers are indeed unequal.

Two other useful Boolean operations are `and` and `or`. Can you guess what they do?

And and Or

The `and` and `or` keywords have simple logical meanings. In particular, the *and* of a collection of Boolean values is true if all are true and false if any value is false, the *or* of a collection of Boolean values is true if any of the values is true and false if all the values are false. For example,

```
> (and #t #t #t)
#t
> (and (< 1 2) (< 2 3))
#t
> (and (odd? 1) (odd? 3) (odd? 5) (odd? 6))
#f
> (and)
#t
> (or (odd? 1) (odd? 3) (odd? 5) (odd? 6))
#t
> (or (even? 1) (even? 3) (even? 4) (even? 5))
#t
> (or)
#f
```

And and Or as Control Structures

But `and` and `or` can be used for so much more. In fact, they can be used as control structures.

In an `and`-expression, the expressions that follow the keyword `and` are evaluated in succession until one is found to have the value `#f` (in which case the rest of the expressions are skipped and the `#f` becomes the value of the entire `and`-expression). If, after evaluating all of the expressions, none is found to be `#f` then the value of the last expression becomes the value of the entire `and` expression. This evaluation strategy gives the programmer a way to combine several tests into one that will succeed only if all of its parts succeed.

This strategy also gives the programmer a way to avoid meaningless tests. For example, we should not make the comparison `(< a b)` unless we are sure that both `a` and `b` are numbers.

In an `OR` expression, the expressions that follow the keyword `OR` are evaluated in succession until one is found to have a value other than `#f`, in which case the rest of the expressions are skipped and this value becomes the value of the entire `OR`-expression. If all of the expressions have been evaluated and all have the value `#f`, then the value of the `OR`-expression is `#f`. This gives the programmer a way to combine several tests into one that will succeed if *any* of its parts succeeds.

In these cases, `and` returns the last parameter it encounters (or false, if it encounters a false value) while `or` returns the first non-false value it encounters. For example,

```
> (and 1 2 3)
3
> (define x 'two)
> (define y 3)
> (+ x y)
+: expects type <number> as 1st argument, given: two; other arguments were: 3
> (and (number? x) (number? y) (+ x y))
#f
> (define x 2)
> (and (number? x) (number? y) (+ x y))
5
> (or 1 2 3)
1
> (or 1 #f 3)
1
> (or #f 2 3)
2
> (or #f #f 3)
3
```

We can use the ideas above to make an addition procedure that returns `#f` if either parameter is not a number. We might say that such a procedure is a bit safer than the normal addition procedure.

```
;;; Procedure:
;;;   safe-add
;;; Parameters:
;;;   x, a number [verified]
;;;   y, a number [verified]
;;; Purpose:
;;;   Add x and y.
;;; Produces:
;;;   sum, a number.
;;; Preconditions:
;;;   (No additional preconditions)
;;; Postconditions:
;;;   sum = x + y
;;; Problems:
;;;   If either x or y is not a number, sum is #f.
(define safe-add
  (lambda (x y)
    (and (number? x) (number? y) (+ x y))))
```

Let's compare this version to the standard addition procedure, `+`.

```

> (+ 2 3)
5
> (safe-add 2 3)
5
> (+ 2 'three)
+: expects type <number> as 2nd argument, given: three; other arguments were: 2
> (safe-add 2 'three)
#f

```

If we'd prefer to return 0, rather than #f, we could add an or clause.

```

(define safer-add
  (lambda (x y)
    (or (and (number? x) (number? y) (+ x y))
        0)))

```

In most cases, `safer-add` acts much like `safe-add`. However, when we use the result of the two procedures as an argument to another procedure, we get a little bit further through the calculation.

```

> (* 4 (+ 2 3))
20
> (* 4 (safer-add 2 3))
20
> (* 4 (+ 2 'three))
+: expects type <number> as 2nd argument, given: three; other arguments were: 2
> (* 4 (safe-add 2 'three))
*: expects type <number> as 2nd argument, given: #f; other arguments were: 4
> (* 4 (safer-add 2 'three))
0

```

Different situations will call for different choices between those strategies.

Keywords vs. Procedures

You may note that we were careful to describe `and` and `or` as “keywords” rather than as “procedures”. The distinction is an important one. Although keywords look remarkably like procedures, Scheme distinguishes keywords from procedures by the order of evaluation of the parameters. For procedures, all the parameters are evaluated and then the procedure is applied. For keywords, not all parameters need be evaluated, and particular orders of evaluation are possible.

If `and` and `or` were procedures, we could not guarantee their control behavior. We'd also get some ugly errors. For example, consider the revised definition of `even?` below:

```

(define new-even?
  (lambda (val)
    (and (integer? val) (even? val))))

```

Suppose `new-even?` is called with `2/3` as a parameter. In the keyword implementation of `and`, the first test, `(integer? 2/3)`, fails, and `new-even?` returns false. If `and` were a procedure, we would still evaluate the `(even? val)`, and that test would generate an error.

Separating the World into Not False and False

Although most computer scientists, philosophers, and mathematicians prefer the purity of dividing the world into “true” and “false”, Scheme supports a somewhat more general separation. In Scheme, anything that is not false is considered true. Hence, you can use expressions that return values other than truth values wherever a truth value is expected. For example,

```
> (and #t 1)
1
> (or 3 #t #t)
3
> (not 1)
#f
> (not (not 1))
#t
```

Copyright © 2006 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.