

Boolean Values and Predicate Procedures

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.)

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 I 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 (you’ll learn about strings in a few days).
- `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.

Scheme provides one basic predicate for working with lists (other than the `list?` predicate).

- `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.

Scheme also provides many numeric predicates.

- `=` tests whether its arguments, which must all be numbers, are numerically equal; 5 and 5.0 are numerically equal for this purpose.
- `<` 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 procedures are `and` and `or`. Can you guess what they do?

And and Or

The `and` and `or` procedures 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
```

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) or all of the expressions have been evaluated (in which case the value of the last expression becomes the value of the `and`-expression). This gives the programmer a way to combine several tests into one that will succeed only if all of its parts succeed.

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) or all of the expressions have been evaluated (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.