Boolean Values and Predicate Procedures

Summary: Many of Scheme's control structures, such as conditionals (which you'll learn about in a subsequent reading), need mechanisms for constructing tests that return true or false. These tests can also be useful for gathering information about values. In this reading, we consider the types, basic procedures, and mechanisms for combining results, that support such tests.

Contents:

Introduction

When writing complex programs, we often need to ask questions about the values with which we are computing. Is this pixel a shade of red? Is this image at least 100x100? Are these two colors close enough to be indistinguishable? Is this a light or dark color? Most frequently, these questions (which we often phrase as *tests*) are used in control structures. For example, we might decide to do one thing for large images and another for small images or we might replace light colors by white and dark colors by black.

To express these kinds of questions, we need a variety of tools. First, we need a type in which to express the valid answers to questions. Second, we need a collection of procedures that can answer simple questions. Third, we need ways to combine questions. Finally, we need control structures that use these questions. In the subsequent sections of this reading, we consider each of these issues. We return to more complex control structures in a subsequent reading.

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 pure red has a high blue component, 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.)

Scheme provides a wide variety of basic predicates and DrFu adds a few more. We will consider a few right now, but learn more as the course progresses.

Type 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.
- `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.

DrFu adds a few special predicates that are tailored to working with colors and images.

- `image?` tests whether its argument can be interpreted as an image.
- `rgb?` tests whether its argument can be interpreted as an RGB color.
- `cname?` tests whether its argument can be interpreted as a color name.

Equality Predicates

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.

For this class, you are not required to understand the difference between the `eq?` and `eqv?` procedures. In particular, you need not plan to use the `eqv?` procedure. At least for the first half of the semester, you also need not understand the difference between the `eq?` and `equal?` procedures. Feel free to use `equal?` almost exclusively, except when dealing with numbers, in which case you should use `=`.

Numeric Predicates

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.

Negating Boolean Values with `not`

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.

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
```

Detour: 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 custom 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, since `even?` can only be called on integers.

Another Detour: Separating the World into Not False and False

Although many 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 "truish". 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
```

Some Color Predicates

Can we write predicates that work with colors? Certainly. One simple question is whether we might consider two colors near to each other. What are criteria for making that decision? One possibility is that we will consider two colors similar if all of their components are within 8 of each other. We can define that as follows:

```
;;; Procedure:
;;;   colors.similar?
;;; Parameters:
;;;   color1, an RGB color
;;;   color2, an RGB color
;;; Purpose:
;;;   Determines if color1 and color2 are similar.
;;; Produces:
;;;   similar?, a Boolean value
;;; Preconditions:
;;;   [none]
;;; Postconditions:
;;;   If color1 and color2 are close enough to be considered similar,
;;;     then similar? is #t.
;;;   Otherwise, similar? is #f.
;;;   We use a proprietary technique to decide what "close enough" means.
(define colors.similar?
  (lambda (color1 color2)
    (and (>= 8 (abs (- (rgb.red color1) (rgb.red color2))))
         (>= 8 (abs (- (rgb.green color1) (rgb.green color2))))
         (>= 8 (abs (- (rgb.blue color1) (rgb.green color2)))))))
```

Here's a pair of useful predicates: One computes whether a color might reasonably be considered light; another computes whether a color might reasonably consider dark.

```
;;; Procedure:
;;;   rgb.light?
;;; Parameters:
;;;   color, an RGB color
;;; Purpose:
;;;   Determine if the color seems light.
;;; Produces:
;;;   light?, a Boolean value
;;; Preconditions:
;;;   [None]
;;; Postconditions:
;;;   light? is true (#t) if color's intensity is relatively high.
;;;   light? is false (#f) otherwise.
(define rgb.light?
  (lambda (color)
    (<= 192 (+ (* .30 (rgb.red color)) (* .59 (rgb.green color)) (* .11 (rgb.blue color))))))

;;; Procedure:
;;;   rgb.dark?
;;; Parameters:
;;;   color, an RGB color
;;; Purpose:
;;;   Determine if the color seems dark.
;;; Produces:
```

```
;;;   dark?, a Boolean value
;;; Preconditions:
;;;   [None]
;;; Postconditions:
;;;   dark? is true (#t) if color's intensity is relatively low.
;;;   dark? is false (#f) otherwise.
(define rgb.dark?
  (lambda (color)
    (>= 64 (+ (* .30 (rgb.red color)) (* .59 (rgb.green color)) (* .11 (rgb.blue color))))))
```

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.

An Application: Black, Grey, and White

Here's a simple application of the preceding: We can write a procedure that, given a color, returns black if the color is dark, white if the color is light, and grey if the color is neither dark nor light.

How? Well, we can use `and` to compute either black, if the color is dark, or `#f`, if the color is not dark.

```
(and (rgb.dark? color) black)
```

Similarly, we can use `and` to compute either white, if the color is light, or `#f` if the color is not light.

```
(and (rgb.light? color) white)
```

Finally, we can use `or` to put it all together.

```
;;; Procedure:
;;;   rgb.bgw
;;; Parameters:
;;;   color, an RGB color
;;; Purpose:
;;;   Convert an RGB color to black, grey, or white, depending on
;;;   the intensity of the color.
;;; Produces:
;;;   bgw, an RGB color
;;; Preconditions:
;;;   rgb.light? and rgb.dark? are defined.
;;; Postconditions:
;;;   If (rgb.light? color) and not (rgb.dark? color), then bgw is white.
;;;   If (rgb.dark? color) and not (rgb.light? color), then bgw is black.
;;;   If neither (rgb.light? color) nor (rgb.dark? color), then bgw is
;;;     grey.
;;;   Otherwise, bgw is one of black, white, or grey.
(define black (rgb.new 0 0 0))
(define white (rgb.new 255 255 255))
(define grey (rgb.new 128 128 128))
(define rgb.bgw
  (lambda (color)
    (or (and (rgb.light? color) white)
        (and (rgb.dark? color) black)
        grey)))
```
