Numeric Values

Summary: We examine a variety of issues pertaining to numeric values in Scheme, including the types of numbers that Scheme supports and some common numeric functions.

Contents:

Introduction

Computer scientists write algorithms for a variety of problems. Some types of computation, such as representation of knowledge, use symbols and lists. Others, such as the construction of Web pages, may involve the manipulation of strings (sequences of alphabetic characters). However, as you've seen with your initial experiments with images, a significant amount of computation involves numbers.

One advantage of doing numeric computation with a programming language, like Scheme, is that you can write your own algorithms to make the computer automate repetitive tasks. As you do numeric computation in any language, you must first discover what *types of numbers* the language supports (some languages support only integers, some only real numbers, some combinations) and what *numeric operations* the language supports. Fortunately, Scheme supports many types of numbers (as you may have discovered in the first few labs) and a wide variety of operations on those numbers.

Categories of Numbers

As you probably learned in secondary school, there are a variety of kinds of numbers. The most common types are the *integers* (numbers with no fractional component), *rational numbers* (numbers that can be expressed as the ratio of two integers), and *real numbers* (numbers that can be plotted on a number line). DrFu supports only integers and real numbers.

In some Scheme implementations, other numeric types are available, such as the *complex numbers* (numbers with a possible imaginary component). While you will not use those other types in this course, we alert you to their availability so that you can think of applications outside of the primary focus of this course. Why does DrFu leave out some kinds of numbers? Because the implementers did not see a need for them in the kinds of applications DrFu was intended for. The standard language definition for Scheme says that an implementation of the language does not have to support all categories of numbers.

Scheme provides two *predicates* that let us check whether or not a value has a particular type: `integer?` and `real?`.

```
> (integer? 2)
#t
> (real? 2)
#t
> (integer? 2.5)
#f
> (real? 2)
#t
> (integer? "two")
#f
```

DrFu uses integers to represent colors and image identifiers, so `integer?` will return true for them, too.

```
> (integer? (rgb.new 0 0 0))
#t
> (integer? (image.load "/home/rebelsky/glimmer/samples/rebelsky-stalkernet.jpg"))
#t
> (integer? "/home/rebelsky/glimmer/samples/rebelsky-stalkernet.jpg")
#f
```

We will return to predicates when we consider conditionals.

Within each of these categories of numbers, Scheme distinguishes between *exact numbers*, which are guaranteed to be calculated and stored internally with complete accuracy (no rounding off), and *approximations*, also called *inexact numbers*, which are stored internally in a form that conserves the computer's memory and permits faster computations, but allows small inaccuracies (and occasionally ones that are not so small) to creep in. Since there's no great advantage in obtaining an answer quickly if it may be incorrect, we shall avoid using approximations in this course, except when the data for our problems are themselves obtained by inexact processes of measurement

To determine whether Scheme is representing a particular number exactly or inexactly, use one of the predicates `exact?` and `inexact?`. Real numbers are never represented exactly, and integers can be represented exactly or inexactly.

```
> (exact? 2)
#t
> (exact? 2.0)
#f
```

The Scheme standard does not directly support the familiar category of *natural numbers*, but we can think of them as being just the same things as Scheme's exact non-negative integers.

Numeric procedures

Section 6.2.5 of the *Revised^5 report on the algorithmic language Scheme* lists Scheme's primitive procedures for numbers. Read through the list at this point to get a feel for what Scheme supports. The following notes explain some of the subtler features of commonly used numerical procedures. As you read about procedures, think about how you might use them in writing color filters or in other graphical algorithms.

Warning! The output from DrFu is not always consistent with our expectations. DrFu is also evolving. In a few cases, you may see slightly different resopnses than appear in this reading.

The addition and multiplication procedures, `+` and `*`, accept any number of arguments. You can, for instance, ask Scheme to imitate a cash register with a command like this one:

```
> (+ 1.19
      .43
      .43
     2.59
      .89
     1.39
     5.19
      .34
  )
12.45
```

You can call the `-` procedure or the `/` procedure to operate on a single argument. The `-` procedure returns the additive inverse of a single argument (its negative), the result of subtracting it from 0.

The `max` procedure returns the largest of its parameters and the `min` procedure returns the smallest of its parameters. As we've already seen, `max` can be useful when you want to ensure that a computation returns a value no smaller than a certain value and `min` can be useful when you want to ensure that a computation returns a value no larger than a desired maximum value.

There are four procedures relating to division (`/`, `quotient`, `remainder`, and `modulo`). The `/` procedure returns the multiplicative inverse of a single argument (its reciprocal), the result of dividing 1 by it. The `quotient` and `remainder` procedures apply only to integers and perform the kind of division you learned in elementary school, in which the quotient and the remainder are separated: "Four goes into thirteen three times with a remainder of one":

```
> (quotient 13 4)
3
> (remainder 13 4)
1
> (quotient 1 2.5)
quotient: expects type <integer> as 2nd argument, given: 2.5; other arguments were: 1
```

As the final example suggests, `quotient` can only be applied to integers. The `/` procedure, on the other hand, can be applied to numbers of any kind (except that you can't use zero as a divisor) and yields a single result.

The `modulo` procedure is like `remainder`, except that it always yields a result that has the same sign as the divisor. In particular, this means that when the divisor is positive and the dividend is negative, `modulo` yields a positive (or zero) result. (When can a remainder be negative? Consider -5 divided by 2. Do we think of -5 as -2*2-1 or -3*2+1? Scheme makes the former decision for remainder and the latter decision for modulo.)

```
> (remainder -13 4)
-1
> (modulo -13 4)
3
```

The modulo procedure can be particularly useful for color computations. As you have noted, sometimes we end up computing a component value greater than 255 or less than 0. In those cases, we can ensure that they fall within the appropriate range with `max` and `min`. We can get somewhat different effects by using `(modulo` *`computed-value`* `256)`. This expression ensures that the value is between 0 and 255, but causes larger numbers to *wrap-around* to become smaller numbers.

```
> (define blue-component 250)
> (min 255 (+ 32 blue-component))
255
> (mod (+ 32 blue-component) 256)
26
```

At times, we will have a real number and will want to convert it to a nearby integer. For example, if you are working with images, the components of an RGB should be integers; weird things can happen if you try to use real numbers (not always, but sometimes). Scheme provides four basic procedures for this conversion: `round`, `truncate`, `floor`, and `ceiling`. You will explore the differences between these procedures in the corresponding lab.

Scheme provides five basic predicates for comparing numeric values, `<` (less than), `<=` (less than or equal to), `=` (equal to), `>=` (greater than or equal to), and `>` (greater than). When given two arguments, they return `#t` if the indicated relation holds between the two arguments.

```
> (< 5 10)
#t
> (> 5 10)
#f
```

These predicates and also take more than two arguments. The predicate returns `#t` only if the relation holds between each pair of adjacent arguments.

```
> (< 2 3 4)
#t
> (< 2 3 1)
#f
```

The `log` procedure, despite its name, computes natural (base e) logarithms rather than common (base ten) logarithms. You can convert a natural logarithm into a common logarithm by dividing it by the natural logarithm of 10. In case you've forgotten, the common logarithm of *n* is "the power to which you raise 10 in order to get *n*".

```
> (log 100)
4.605170185988092
> (/ (log 100) (log 10))
2.0
```

Scheme provides the standard host of trigonometric functions, which include `sin`, `cosine`, and `tan`. When using these functions, bear in mind that all angles are measured in radians, not degrees.

```
> (sin 90)
0.8939966636005579
> (cos 90)
-0.4480736161291701
```

```
> pi
3.141592654
> (exact? pi)
#f
> (sin (/ pi 2))
1.0
> (cos (/ pi 2))
6.123031769e-17
```

You may wonder why the cosine of pi-over-2 (a right angle) is not 0. It's because `pi` is not exactly the value of pi. However, as scientific notation indicates, the value is pretty close to 0. (There are sixteen leading 0's.)

We can use the trigonometric functions when we start doing more involved drawings (e.g., they can help us draw polygons). The trigonometric functions also provide the opportunity to do some interesting color transformations.
