

## Defining Your Own Scheme Procedures

**Summary:** We explore why and how you might define your own procedures in Scheme.

### Contents:

- The Problem: Naming Expressions
- Defining Procedures
  - A `fracpart` Procedure
  - A Square Procedure
- Documenting Your Procedures
- Procedures with More Than One Parameter
- Some `fracpart` Oddities

## The Problem: Naming Expressions

In previous labs, you've seen that it's possible to define complex expressions from simpler expressions. For example, we might write the following to determine the fractional part of a real number explicitly as a rational number.

```
(inexact->exact (- val (truncate val)))
```

What happens when we want to try this expression using different values? One possibility is to redefine `val` and then re-execute the expression.

```
> (define val (sqrt 2))
> (inexact->exact (- val (truncate val)))
1865452045155277/4503599627370496
> (define val 5)
> (inexact->exact (- val (truncate val)))
0
> (define val 22/7)
> (inexact->exact (- val (truncate val)))
1/7
> (define val 0.75)
> (inexact->exact (- val (truncate val)))
3/4
```

But this seems cumbersome. It would be nice to simply give a name to this expression and use that name. Unfortunately, our current way of naming doesn't quite work.

```
> (define val 1.75)
> (define fracpart (inexact->exact (- val (truncate val))))
> fracpart
3/4
> (define val 5)
> fracpart
3/4
```

What's going on? Scheme evaluated the expression that accompanies `fracpart` *once*, when it was first defined. Hence, since the expression had a particular value once, it retains that value forever.

What we'd really like to do is to say that “`frac` is a *procedure* that takes a value as an input and returns an appropriate fraction”. You know that Scheme has procedures, since you've used lots of built-in procedures, including `sqrt`, `*`, `cons`, and `list`. But can you define your own procedures? **Certainly.**

## Defining Procedures

You use `define` to give names to procedures, just as you use it to give names for values. The values just look different. The general form of a procedure is

```
(lambda (formal-parameters)  
  expression)
```

### A `fracpart` Procedure

For example, we might write our `fracpart` procedure as

```
(define fracpart  
  (lambda (val)  
    (inexact->exact (- val (truncate val)))))
```

Our `fracpart` procedure can now be called as if it were a built-in procedure.

```
> (fracpart (sqrt 2))  
1865452045155277/4503599627370496  
> (fracpart 5)  
0  
> (fracpart 22/7)  
1/7  
> (fracpart 0.75)  
3/4  
> (fracpart 1.5)  
1/2  
> (fracpart 1.2)  
900719925474099/4503599627370496
```

See the notes at the end of this document for explanation of the stranger results.

### A Square Procedure

We can define procedures for anything we already know how to do in Scheme. For example, here is a simple `square` procedure.

```
(define square  
  (lambda (n)  
    (* n n)))
```

We can test it.

```
> (square 2)
4
> (square -4)
16
> (square square)
*: expects type <number> as 1st argument, given: #<procedure:square>; other arguments were: #<procedure:square>
> (square 'a)
*: expects type <number> as 1st argument, given: a; other arguments were: a
```

## Documenting Your Procedures

Convention in Scheme (and all programming languages) is that we carefully document what our procedures do, including input values, output values, and assumptions. We use comments provide information to the reader of our program (that is, to people instead of the computer). In Scheme, comments begin with a semicolon and end with the end of the line.

```
;;; Samuel A. Rebelsky
;;; Department of Mathematics and Computer Science
;;; Grinnell College
;;; rebelsky@cs.grinnell.edu

;;; Procedure:
;;;   square
;;; Parameters:
;;;   val, a number
;;; Purpose:
;;;   Compute val*val
;;; Produces:
;;;   result, a number
;;; Preconditions:
;;;   val must be a number
;;; Postconditions:
;;;   result is the same "type" of number as val (e.g., if
;;;   val is an integer, so is result; if val is exact,
;;;   so is result).
;;; Citations:
;;;   Based on code created by John David Stone dated March 17, 2000
;;;   and contained in the Web page
;;;   http://www.math.grin.edu/~stone/courses/scheme/procedure-definitions.xhtml
;;; Changes to
;;;   Parameter names
;;;   Formatting
;;;   Comments
(define square
  (lambda (value)
    (* value value)))
```

Yes, that's a lot of documentation for very little code. However, it is better to err on the side of too much documentation than too little documentation. More importantly, as you start writing more procedures, their purpose and details will be much less obvious. Finally, when you carefully document procedures, you begin to think more carefully about what they really need to do and how you ensure that they do so for all cases.

Here's another set of documentation, this time for the `fracpart` procedure that we wrote earlier. When documenting `fracpart`, we are forced to think about (1) what kinds of numbers it works on (in this case, it does not work on complex numbers); (2) what, precisely, the relationship of the result to the input is; and (3) what type the result has.

```

;;; Procedure:
;;;   fracpart
;;; Parameters:
;;;   val, a real number
;;; Purpose:
;;;   Express the fractional part of val as a fraction.
;;; Produces:
;;;   rat, a rational number.
;;; Preconditions:
;;;   val cannot be complex.
;;; Postconditions:
;;;   0 <= rat < 1.
;;;   (- val rat) is a whole number (or a a close approximation).
;;;   rat is exact.
;;;   rat is approximately equal to the fractional part of val
;;;     (within some unknown level of accuracy).
;;;   rat is the ratio of two integers.
(define fracpart
  (lambda (val)
    (inexact->exact (- val (truncate val)))))

```

## Procedures with More Than One Parameter

At times, we will want to write procedures that take more than one parameter. Such procedures look just like procedures with one parameter, except that you can list more parameters between the parentheses.

```

(lambda (param1 param2 ... paramn)
  expression
)

```

For example, here is a simple procedure that finds the average of two numbers.

```

;;; Samuel A. Rebelsky
;;; Department of Mathematics and Computer Science
;;; Grinnell College
;;; rebelsky@cs.grinnell.edu

;;; Procedure:
;;;   pairave
;;; Parameters:
;;;   val1, an exact number
;;;   val2, an exact number
;;; Purpose:
;;;   Compute the average of two numbers.
;;; Produces:
;;;   ave, The average of those two numbers.
;;; Preconditions:
;;;   Both val1 and val2 are exact numbers.
;;; Postconditions:

```

```
;;; ave is an exact number.
;;; ave is equidistant from val1 and val2. That is
;;; (abs (- val1 ave)) equals (abs (- val2 ave))
(define pairave
  (lambda (val1 val2)
    (/ (+ val1 val2) 2)))
```

As this example may suggest, in your documentation it is particularly important to think about what you can guarantee about the results of your procedure. In this case, what does it mean to be the *average* of two values.

## Some `fracpart` Oddities

You may have observed some odd behavior with some of the calls to `fracpart`, such as

```
> (fracpart (sqrt 2))
1865452045155277/4503599627370496
> (fracpart 1.2)
900719925474099/4503599627370496
```

What's going on with these examples? It turns out that our implementation of DrScheme currently represents the decimal part of many inexact numbers as a ratio of some number and 4503599627370496, which happens to be  $2^{52}$ . (Most computers like powers of 2.) That's why it's inexact. There's no number,  $n$ , such that  $n/4503599627370496$  is 0.2. However, 0.5 is  $2251799813685248/4503599627370496$ , which can be simplified to  $1/2$ .

In the case of the fractional part of the square root of 2, that number is approximated to a lot of digits, so you expect it to have a relatively large denominator.

---

Copyright © 2007 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.