Tutorial 6: Recursion, Lists, and Tally Sets

CS 135 Fall 2007

October 17-19, 2007

In this week's tutorial, we'll be using recursion to write functions that consume and/or produce numbers in two different representations. First we will use the built-in numbers in Scheme and write our own versions of some of the basic arithmetic functions. Then we will use a list-of-lists representation which corresponds to writing tally sets. The material covered here is from lecture module 6 in the course notes (the last module which will be covered on your second midterm).

1 Basic Arithmetic Functions

In the starter code for this tutorial (t6-starter.scm), one constant, `zero`, and three function definitions are given: `my-zero?`, `my-sub1`, and `my-add1`. Initially, these just call the corresponding built-in Scheme functions on numbers.

Using **only the functions above and the ones you write** (no other built-in Scheme functions), give definitions for the following functions, whose behaviour should correspond to the behaviour of the standard Scheme functions. You'll probably want to implement them in the order they're given below.

You may also assume that all numbers consumed and produced are non-negative integers. So for example the second argument to `my-` will never be greater than the first. All of these functions should be binary (i.e. they take two arguments).

- `my+`
- `my-`

- `my*`
- `my<`
- `my-quotient`
- `my-remainder`

2 Tally sets

A more primitive way of writing numbers than the Arabic numerals we are used to is tally sets. These are just collections of lines, usually structured in groups of 5, with at most one incomplete group. The number represented by a tally set is just the number of lines in the set.

To represent a tally set in Scheme, we will use a list of non-empty lists of `1`'s. Each `1` corresponds to a line, and each list of `1`'s corresponds to a single (possibly incomplete) group with at least one line. See the data definition in the starter code for more details.

2.1 Conversion

Write the functions `tally->num` and `num->tally`, which convert between the tally set standard number representation of a non-negative integer.

2.2 Arithmetic

Change the definitions of the constant `zero` and the three functions originally given in the starter code (`my-zero?`, `my-sub1`, and `my-add1`) to consume and produce tally sets instead of regular numbers. Now all the arithmetic you wrote should work on tally sets (you may have to change some test cases though).

3 Slightly more complicated recursion

3.1 Setting up the machinery

Using only the functions `my-zero?`, `my-sub1`, and `my-add1`, write four new functions which consume a single tally set: `my-even?`, `my-odd?`, `my-double`,

and `my-half`. `my-even?` returns true iff the given tally set represents an even number, `my-double`returns a tally set representing twice the given one, and `my-half` consumes a tally set representing an even number and produces a tally set representing one half of that number.

3.2 Faster multiplication

Now, using the definition of a natural number as either 0, 2 times a natural number, or 2 times a natural number plus 1, rewrite your definition of `my*` above. Why do you think this implementation might be faster than the original one?