Lab: Mutual Recursion with S-Expressions

COMP 50

Fall 2013

The theme of this lab is *restriction*:

- Not every list of symbols is a set of symbols.
- Not every S-expression is an arithmetic expression.

There are three groups of problems:

- Problems 1–3 ask you to compute the set of symbols in an S-expression. You will get practice working with S-expressions and you will write another function that consumes two lists.
- Problems 4 and 5 ask you to define a form of arithmetic expressions and an evaluation function; problems 7 and 8 ask you to find sets of symbols within these expressions.
- Problems 9 and 10 are bonus enrichment problems on evaluation of expressions with multiple variables.

Expect to finish either the first or the second group of problems.

Follow the design recipe for mutually referential data definitions. Unless you develop multiple functions in parallel, you will go horribly wrong. If you do follow the design recipe, you will walk out of this lab with substantial experience writing templates and functions that work with mutual self-reference.

S-Expressions

An S-expression models the way BSL and Racket (and other languages in the same family) represent source code.

An *S-expr* (S-expression) is one of:

- Atom
- SL

An *SL* (S-list) is one of:

- `empty`
- `(cons S-expr SL)`

An *Atom* is one of:

- Number
- String
- Symbol

Exploring how DrRacket evaluates expressions

1. **Data definition**: A *set-of-symbols* is one of:

   - `empty`
   - `(cons symbol set-of-symbols)`, where the given `symbol` does not occur anywhere in the given `set-of-symbols`

   Define a function `has-symbol?` which tells whether a given set of symbols contains a given symbol.

2. *Define a function* that computes the union of two sets of symbols, returning a set of symbols. You will need to use one of the three templates for consuming two lists.

3. *Define a function* that computes a set containing all the symbols in an S-expression.

4. *Develop a data definition* for arithmetic *expressions without variables*.
   Here are some examples:

   - `7`
   - `'(+ 3 4)`
   - `'(- (* 2 4) 1)`

   Hint: You can use the definition of S-expressions as a guide.

5. An expression without variables can be evaluated. For example, all three of the examples above evaluate to 7.

   Design a function that evaluates a given arithmetic expression without variables. If the only symbols in the expression are those in the set `'(+ - *)`, then evaluation must succeed.

6. Here are some arithmetic expressions that may contain variables:

   - `7`
   - `'y`
   - `'(+ (* 3 x) 1)`

   Develop a data definition for such expressions. Your data definition should **distinguish between *operators* and *variables***.

7. *Define a function* that returns the set of *operator* symbols in an arithmetic expression with variables.

8. *Define a function* that returns the set of *variable* symbols in an arithmetic expression with variables.

9. If `y` is 7 and `x` is 2, the expressions in Problem 6 all evaluate to 7. We can simulate the evaluation the same way the stepper does, by "plugging in" the *value* of the variable in the expression. The computer-science word for "plugging in" is *substitution*.

   Design a function `substitute` that consumes a given arithmetic expression (which may contain variables), a , and a number, and that produces a arithmetic expression that is *identical* to the given expression, except that all occurrences of the given variable are replaced with the given number.

   For example, substituting 2 for `x` in `'(+ (* 3 x) 1)` should produce the expression `'(+ (* 3 2) 1)`. Substituting 7 for `z` in the expression `'(+ 1 (* z z))` should produce the expression `'(+ 1 (* 7 7))`.

10. We can represent "the values of variables" in a *symbol-number-association-list*, which is defined as follows:

    A *symbol-number-association-list* is one of:

    - empty
    - `(cons (cons symbol (cons number empty)) symbol-number-association-list)`

Here are some data examples:

- `'()`
- `'((x 7))`
- `'((x 7) (y 15) (z 12))`

The symbol-number-association-list is used to assign values to variables. Given such a list, we can write an evaluator for expressions with variables.

Define a function that takes a given arithmetic expression with variables, and a given symbol-number-association-list, and returns the result of evaluating the expression with the given numbers substituted for the given variables.

Examples:

- Evaluating `'(* (+ x y) (- x y))` with `'((x 5) (y 2))` produces 21.
- Evaluating `'(+ z 1)` with `'((x 5) (z 16) (y 2))` produces 17.
- Evaluating `'(+ 7 1)` with `'((x 5) (z 16) (y 2))` produces 8.
- Evaluating `'(+ z w)` with `'((x 5) (z 16) (y 2))` calls `error`.

Submitting the lab

Five minutes before the end of the lab, put the following text at the beginning of a DrRacket file. You may use an empty file or the source code you have been working on:

```
#|
What I did during this lab:
   (you fill in this part)

What I learned during this lab:
   (you fill in this part)

|#
```

The lab staff will help you articulate what you learned.

Finally, *submit this file through the handin server* as `lab-sexp`. You will need to submit it using *two* usernames connected by a + sign, as in

```
Jane.Doe+Richard.Roe
```

You submit using Jane Doe's password.