MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1995

Problem Set 4

Rewriting Symbolic Expressions

Issued: Tuesday, 7 March

For **odd**-numbered sections due in recitation: **Wednesday**, 15 March
For **even**-numbered sections due in recitation: **Friday**, 17 March

Reading assignment::

- Supplementary notes: A Term Rewriting Model for Scheme
- code files attached: `smeval.scm`, `smsyntax.scm`, `tests.scm`

This problem set is about handling symbolic expressions as abstract data. The main application will be to the Term Rewriting Model (TRM) for a language similar to Scheme, as described in the Supplementary Notes handout (which you should read before reading further in this Problem Set). In this problem set, the abstract data represents the subset of Scheme expressions specified in the handout.

Loading the problem set will allow you to run an implementation of the TRM. There are only two parts of the implementation you need to learn about for this problem set; these are in the attached files `smeval.scm` and `smsyntax.scm`.

The TRM can be used to observe selected steps in the execution of a *body*. (A body is a sequence of zero or more definitions followed by an expression.) For example, we can use the procedure `smeval` to observe the execution of the factorial program as follows:

```
(smeval
 '(
   (define (fact n)
     (if (= n 0)
         1
         (* n (fact (- n 1)))))
   (fact 3)
   ))
```

The resulting printout shows selected intermediate steps in the execution, and the final value

```
;==(1)==>
((define fact
   (lambda (n)
     (if (= n 0)
         1
         (* n (fact (- n 1))))))
 ((lambda (n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))) 3))

;==(2)==>
((define fact
   (lambda (n)
     (if (= n 0)
         1
         (* n (fact (- n 1))))))
 (define n#8 3)
 (if (= n#8 0)
     1
     (* n#8 (fact (- n#8 1)))))

... Lots of steps ...

;==(30)==>
((define fact
   (lambda (n)
     (if (= n 0)
         1
         (* n (fact (- n 1))))))
 (define n#8 3)
 (define n#9 2)
 (define n#10 1)
 (define n#11 0)
 (* 3 (* 2 1)))

;==(33)==>
6
Syntactic Value was returned
;Value: 6
```

The procedure `smeval` constructs a list of selected "steps" where each step consists of a step number n and the body which results after n reduction steps, then prints the steps, and returns the final body. The main procedure of the TRM implementation is `one-step-body`. When applied to a body and a list of definitions, it does a single rewrite step, if possible. For example:

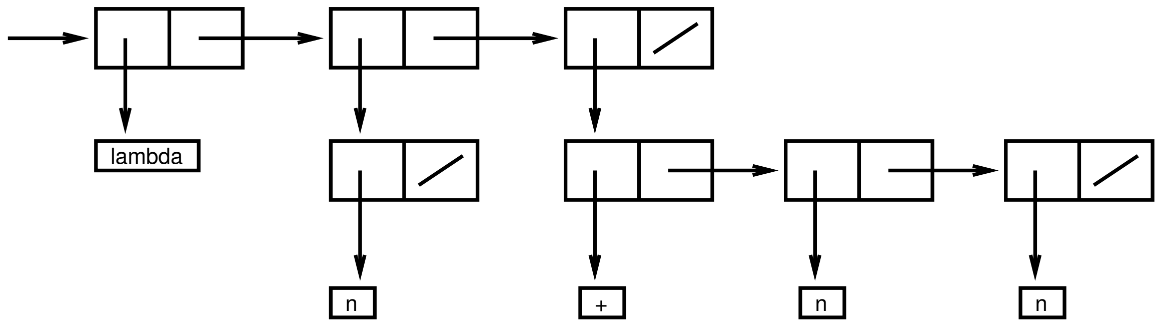
```
(pp
 (one-step-body
  '(
    (define (fact n)
      (if (= n 0)
          1
          (* n (fact (- n 1)))))
    ((lambda (n)
       (if (= n 0)
           1
           (* n (fact (- n 1)))))
     3)
    )
  '()))
(stepped
 ((define (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1)))))
  (define n#13 3)
  (if (= n#13 0)
      1
      (* n#13 (fact (- n#13 1))))))
;Value: #[useless-value]
```

The value returned by `one-step-body` is a "tagged body", i.e., a list of a "tag" and either a body (as defined above) or a data structure describing an error condition. The tag is one of the three symbols `stepped`, `val`, or `stuck`. If the tag is either `stepped` or `val` the second part of the pair is a body. If the tag is `stuck` then the second part is an error description. If the tag is `val` then the second part of the pair is a "fully simplified" body, i.e., no further computational steps can be done.

Abstract Syntax

The Scheme printer prints out lists using the kind of matched-parenthesis syntax that the Scheme reader expects as input for evaluation. So when we want to process expressions themselves as syntactic data—as opposed to evaluating them—it is natural to represent them as list-structures which "look the same." For example, we represent the lambda expression `(lambda (n) (+ n n))` by the list-structure whose box-and-pointer diagram is:

It is sometimes useful to allow for additional representations of expressions. One can treat expressions as abstract data objects which are constructed and accessed using abstract constructors and selectors. Some examples of abstract constructors and selectors are given below. First, we define the procedure `tagged-pair?`:




```
(define (tagged-pair? tag)
  (lambda (exp)
    (and (pair? exp)
         (eq? (car exp) tag))))
```

We can use this to easily make a test for lambda expressions.

```
(define lambda-expression? (tagged-pair? 'lambda))
```

or we can define a selector for the formals of lambda expression:

```
(define formals-of-lambda cadr)
```

Similarly, we can test whether an expression is a conditional:

```
(define if? (tagged-pair? 'if))
```

and we can define a constructor and selectors for conditionals:

```
(define (make-if test consequent alternative)
  (list 'if test consequent alternative))

(define test-of-if cadr)
(define consequent-of caddr)
(define alternative-of cadddr)
```

Scheme code written using standard test/selector/constructor names like those defined above leads to more understandable and *debuggable* code than if we tried to save characters by typing, say, `cadddr` instead of `alternative-of`. All of the code for the TRM implementation (with one exception which you will be asked to rectify) abides by the discipline that expressions be treated as *abstract data*, to be examined and manipulated only by test/selector/constructor procedures which are given as part of the abstract data specification.

The Assignment

For each of the exercises below in which you write Scheme programs, turn in a listing of your programs along with test results demonstrating that they work.

Lab exercise 1: From the 6.001 Edwin editor, use `M-x load-problem-set` to load the code for Problem Set 4. You should now have a file `tests.scm` in which you will find a recursive and an iterative definition of factorial. The procedure `smeval` constructs a list of selected steps. Whether a given step is selected for inclusion is determined by the procedure `save-this-step?` which takes two arguments, a step number and a body. Modify the procedure `save-this-step?` so that every step is saved and then try `smeval`ing factorial of 4 using each of the two definitions given in `tests.scm`.

Lab exercise 2: You might note that the examples on pages 32 and 33 of the text only show a few interesting steps in a computation of factorial. However, the steps selected for printing by `smeval` are too many and show too much detail to convey the insights. In this problem we will develop a version of `save-this-step?` and a version of the printer that that selects only interesting steps and prints them in a way that develops figures similar to Fig. 1.3 and Fig. 1.4. of the text.

2.A Here we write a predicate procedure `simple?` which tells us whether a given expression is simple. By "simple" we will mean that the expression is composed entirely of numbers, the truth values `#t` and `#f`, variables, and combinations. The file `smsyntax.scm` contains abstract constructors, selectors, and predicates such `variable?`, `combination?`, `operator`, and `operands` that we may use in the definition of `simple?`. (Note that `number?` and `boolean?` are primitives in Scheme.)

We will use the `simple?` predicate in a new definition of `save-this-step?` as follows:

```
(define (save-this-step? step-number body)
  (simple? (expression-of-body body)))
```

Give an appropriate definition of `simple?`. Install it and the definition of `save-this-step?` that uses it (above).

After you make this change show an execution of

```
(smeval
  '((define fact
      (lambda (n)
        (if (= n 0)
            1
            (* n (fact (- n 1))))))
    (fact 4)))
```

2.B The bodies being printed contain the definitions, but we only want to see the expressions to illustrate our point. Pick up the definition of `print-stepped-message` from the file `smeval.scm` and modify it to only print the final expression part of the body. Demonstrate your change on the recursive factorial example you used to demonstrate modifications for part 2.A. Also show an iterative factorial computation, a recursive fibonacci computation (be careful—do not try to do more than `(fib 4)`—we don't want to kill many trees!), and an iterative fibonacci computation.

2.C Unfortunately, the specification of `simple?` given in part 2.A admits expressions that contain ugly generated variables, such as

```
(* 4 (* 3 (* 2 (* n#34 (fact (- n#34 1)))))).
```

Write a version of `simple?` that excludes these cases, by excluding any expression with a variable in an operand position. Demonstrate your change on the same examples you used to demonstrate modifications for part 2.B. Notice that the recursive fibonacci computation does not show the expected complexity (although it is really there!). Explain why the recursive computation looks simpler than it actually is with this version of `simple?`.

Pre-Lab exercise 3.A: The `desugar` procedure in `smsyntax.scm` desugars `let`'s into lambda applications, and `cond`'s into `if`'s. The Revised^4 Report on the Algorithmic Language Scheme explains how several further Scheme constructs can be desugared into the subset of Scheme handled by the TRM (which now includes `let`'s and `cond`'s, since we already know how to desugar them). Pick one of these further constructs, and add a case to `desugar` to handle it. Note that `desugar` treats bodies as abstract data which it manipulates only by standard constructors and selectors; your revision of it should also conform to this discipline.

A straightforward form to desugar is `let*`. Some of the other desugarings, such as the one for `and`, are a little tricky because they are designed to work in the general case with side-effects. For our TRM without side-effects, a satisfactory desugaring of `(and` *A* *B* `...)` is `(let ((x` *A*`)) (if x (and` *B* `...) x)`, where the inner `and` must itself be recursively desugared.

Lab exercise 3.B: Test your definition of `desugar`. (Some tests for `let*` are included in `tests.scm`.)