MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1995-96

Problem Set 5

Issued: Tuesday, March 5, 1996

Due: Solutions should be sent by email to your tutor by Wednesday, March 13 (odd numbered recitation sections) or Friday March 15 (even numbered recitation sections).

Exercises

Work the following exercises before going to lab. You may check your answers in lab using the Scheme interpreter. Write up your solutions and include them as part of your homework.

Exercise 5.1 Work Exercise 2.28 in the notes.

Exercise 5.2 Work Exercise 2.29 in the notes.

Exercise 5.3 Work Exercise 2.30 in the notes.

Pattern Transformations

As discussed in Sec. 2.3.2 of the notes, symbolic differentiation can be accomplished by the `deriv` Scheme procedure that deals with algebraic expressions represented as trees (lists of lists). The differentiation rules, e.g. $D_x[u+v] \to D_x[u] + D_x[v]$, are implemented by a case analysis. Unfortunately the results produced by the `deriv` procedure, although mathematically correct, can be difficult to use because they are unsimplified. Thus `(deriv '(+ 1 x) x)` returns the expression `(+ 0 1)` rather than the simpler equivalent expression `1`.

Although the problem of simplifying algebraic expressions is complex, some fundamental simplifications can be implemented by modifying the constructor procedures `make-sum` and `make-product`. For example, the modified `make-sum` procedure explicitly tests for whether the addend is the number 0 and if so returns the augend, `1` in this case.

The strategy of modifying the constructor procedures `make-sum` and `make-product` is an attractive one, there are certain disadvantages with the approach advocated in Sec. 2.3.2 of *the notes*.

- The embedding of the rules of symbolic differentiation in Scheme programs implies that only someone familiar with Scheme can extend the rules, e.g. Ex. 2.31 in the notes.
- It is unclear how this strategy would handle expressions that involve both addition and multiplication, e.g. `(+ (* 2 x) (* 3 x))` is equivalent to the simpler `(* 5 x)`.
- There is a strong parallelism between the form of rules for symbolic differentiation, e.g.
  1. $D_x[u + v] \rightarrow D_x[u] + D_x[v]$
  2. $D_x[u * v] \rightarrow v * D_x[u] + u * D_x[v]$

  and those of algebraic simplification

  1. $0 + u \rightarrow u$
  2. $0 * v \rightarrow 0$
  3. $1 * u \rightarrow u$

  However this parallelism is not reflected in a parallel structure of the programs `deriv`, `make-sum` and `make-product`.

The balance of this problem set develops a set of procedures that can be used to accomplish both symbolic differentiation and algebraic simplification. The programs implement a "pattern transformer" whose inputs are a set of rules and a symbolic expression that produces a new expression that reflects the application of the rules.

To implement the transformer it is necessary to develop procedures that can determine when two symbolic expressions have the same structure, so that a given rule can be applied.

Problem 1. Two trees `t1` and `t2` are said to *match* one another with respect to the predicate `p?` if they have the same structure and if for each pair of corresponding leaves `l1` and `l2` `(p? l1 l2)` is true.

To test whether two trees match, a recursive strategy similar to that used in procedure `countatoms` can be used

- if both `t1` and `t2` are empty lists, a match occurs
- if both `t1` and `t2` are not pairs, a match occurs if `(p? t1 t2)` is true
- otherwise a match occurs if `(cdr t1)` matches `(cdr t2)` only if `(car t1)` matches `(car t2)`.

Below are three incomplete procedure definitions that implement this strategy. Complete these definitions.

```
(define (match1? t1 t2 pred)
  (cond ((null? t1) (null? t2))
        ((null? t2) false)
        <clause3>
        <clause4>
        (else (and (match1? <exp1> <exp2> pred)
                   (match1? (car t1) (car t2) pred)))))
```

```
(define (match2? t1 t2 pred)
  (cond ((null? t1) (null? t2))
        ((null? t2) false)
        <clause3>
        <clause4>
        ((match2? <exp1> <exp2> pred)
         (match2? (car t1) (car t2) pred))
        (else false)))
(define (match3? tree1 tree2 equal-test)
  (define (helper t1 t2 result)
    (cond ((not result) false)
          ((null? t1) (null? t1))
          ((null? t2) false)
          <clause4>
          <clause5>
          (else (helper <exp1>
                        <exp2>
                        (helper (car t1) (car t2) <exp3>)))))
  (helper <exp4> <exp5> true))
```

To verify your understanding of `match1?`, `match2?`, and `match3?`, specify the arguments to the successive calls to `pred` when they are applied to the trees `(1 (2 3))` and `(1 (2 3))` using `=` as the predicate. Check your analysis by defining

```
(define (p x y) (= x y))

(trace p)
```

and performing the comparison using a Scheme interpreter.

Are these procedures equivalent at a black-box level, i.e., Given two trees, will they always return the same value? Are they guaranteed to apply `pred` to the leaves of the trees in the same order?

In the next part of the assignment you are to extend the notion of matching by allowing one tree to contain "pattern variables" that can be required to satisfy certain restrictions. A restriction is a unary predicate Scheme procedure, such as `odd?`, `(lambda (x) (pair? (cdr x)))`, etc. The syntax for a variable is `(? <var><restriction>)`, e.g. `(? c constant?)`. For example the tree `(1 (2 3) 2)` should

- match `(1 (2 3) 2)`
- match `(1 ((? VAR EVEN?) 3) 2)`
- fail to match `(1 ((? VAR ODD?) 3) 2)`
- match `(1 ((? VAR EVEN?) 3) (? VAR))`
- fail to match `(1 ((? VAR EVEN?) (? VAR)) 2)`

A "dictionary" is used to implement this enhanced matcher by keeping track of associations between pattern variables and parts of trees. A dictionary is a list of associations, which are also implemented as lists. If the dictionary is "empty", there are no associations. Otherwise, the `car` of each association is a symbol (the key) and the `cdr` is a list of the value associated with the key. When matching trees it is necessary to distinguish between a match that succeeds but produces no associations and a match that fails because structures or values associated with pattern variables are inconsistent. To allow for the latter situation, the notion of dictionary is extended to include the value `failed`.

The association that contains a specified key can be determined by the `assq` procedure:

```
(define (assq key dictionary)
  (cond ((null? dictionary) false)
        ((eqv? key (caar dictionary)) (car dictionary))
        (else (assq key (cdr dictionary)))))
```

When a dictionary is to be extended by adding a new association, there are several possibilities. If the key is not already one of the keys in the dictionary, a new association is added at the head of the list. If the key is included among the set of keys in the dictionary, then if the value associated with the key is the same as the new value, no modification is required. Othewise the attempt to add the new association fails.

Procedure `extend-dictionary` takes `key` (a symbol), `dat` (an arbitrary symbolic expression), and `dictionary` as arguments and returns either a dictionary that includes the new association, the existing dictionary, or the symbol `failed` if the new association for `key` is inconsistent with an association in `dictionary`.

Problem 2. Complete the following definition of `extend-dictionary`

```
(define (extend-dictionary key data dictionary)
  (let ((v (assq key dictionary)))
    (cond (...))))
```

When the process of matching two trees begins the dictionary is the empty association list.

The generalized matcher can be implemented using a structure similar to that in `match3?` as follows:

```
(define (match pattern expression dictionary)
  (cond ((eq? dictionary 'failed) 'failed)
        ((literal? pattern)
         (if (eqv? pattern expression) dictionary 'failed))
        ((pattern-variable? pattern)
         (let ((result (apply-restriction pattern expression)))
           (if result
               (extend-dictionary
                (pattern-variable-name pattern)
                expression
                dictionary)
               'failed)))
        ((not (pair? expression)) 'failed)
        (else (match (cdr pattern)
                     (cdr expression)
                     (match (car pattern) (car expression) dictionary)))))
```

where the selectors are defined as

```
(define (literal? pattern)
  (not (pair? pattern)))

(define (pattern-variable? pattern)
  (and (pair? pattern)
       (eq? (car pattern) '?)))

(define (pattern-variable-name pattern)
  (cadr pattern))
```

Problem 3. In order to use predicates to determine whether restrictions are satisfied we employ Scheme's ability to evaluate symbolic expressions. The following procedure can be used for this purpose

```
(define (evaluate exp)
  (eval exp t-g-e))

(define t-g-e the-global-environment)
```

Thus `(evaluate '(+ 2 3))` returns 5, whereas `(evaluate '+)` returns the primitive addition procedure.

Recall that restrictions are pattern-variable expressions that contain a third element that is the name of a Scheme unary predicate, such as `(? x odd?)`. Making use of the `evaluate` procedure complete the definition of procedure `apply-restriction`

```
(define (apply-restriction expression data)
  (let ((rest (cddr expression)))
    (if <a>
        <b>
        ((evaluate (car rest)) data))))
```

Sample restrictions are defined below

```
;;; Restrictions
(define (constant? exp) (number? exp))
(define (variable? exp) (symbol? exp))
```

At this point you have developed the code for a fairly general tree matcher and are in a position to develop it into a powerful program that can transform patterns represented by trees. The importance of such transformations can be seen in their ability to accomplish the tasks of both symbolic differentiation and algebraic simplification in an orderly fashion.

As discussed in Sec. 2.3.2 of the notes, symbolic differentiation can be accomplished by the `deriv` Scheme procedure that deals with algebraic expressions represented as trees (lists of lists). The differentiation rules, e.g. $D_x[u+v] \to D_x[u] + D_x[v]$, are implemented by a case analysis. Unfortunately the results produced by the `deriv` procedure, although mathematically correct, can be difficult to use because they are unsimplified. Thus `(deriv '(+ 1 x) x)` returns the expression `(+ 0 1)` rather than the simpler equivalent expression `1`.

An alternate approach is to implement the rules of both symbolic differentiation and algebraic simplification explicitly, then to apply these rules to mathematical expressions as appropriate.

For example, a mathematician might express the rules of algebraic simplification that pertain to addition as

1. $C_1 + C_2 \to [C_1 + C_2]$
2. $< \exp > +C_1 \to C_1 + < \exp >$
3. $0+ < \exp > \to < \exp >$
4. $C_1 + (C_2 + < \exp3 >) \to [C_1 + C_2] + < \exp3 >$
5. $< \exp1 > +(C_2 + < \exp3 >) \to C_2 + (< \exp2 > + < \exp3 >)$
6. $(< \exp1 > + < \exp2 >) + < \exp3 > \to < \exp1 > +(< \exp2 > + < \exp3 >)$

where

- $\to$ indicates that the expression on the left is to be replaced by that on the right.
- C_1 and C_2 are constants (numbers).
- $< \exp >$, $< \exp1 >$, $< \exp2 >$, and $< \exp3 >$ are arbitrary expressions.
- $[\cdots]$ represents the value of the expression contained in the brackets.

Note that whereas each rule is an algebraic identity, equally true if the expressions to the left and right sides of the "→" were interchanged, the rules produce "simplified" expressions only when left-side expressions are replaced by right-side expressions. Moreover the order in which the rules are applied is important. For example, unless the second rule is applied before the third, the expression `(+ x 0)` will not be replaced by the simpler expression `x`.

For simplicity the syntax of compound expressions is the same as that for Scheme compound expressions restricted to binary operators, as in the text.

```
(define addition-rules
  '(
    ( (+ (? C1 constant?) (? C2 constant?))                ;;; Rule 1
      (*value-of* (+ (: C1) (: C2)))    )
    ( (+ (? E1) (? C2 constant?))                          ;;; Rule 2
      (+ (: C2) (: E1))        )
    ( (+ 0 (? E))                                          ;;; Rule 3
      (: E)                          )
    ( (+ (? C1 constant?) (+ (? C2 constant?) (? E3)))     ;;; Rule 4
      (+ (*value-of* (+ (: C1) (: C2))) (: E3))      )
    ( (+ (? E1) (+ (? C2 constant?) (? E3)))               ;;; Rule 5
      (+ (: C2) (+ (: E1) (: E3)))  )
    ( (+ (+ (? E1) (? E2)) (? E3))                         ;;; Rule 6
      (+ (: E1) (+ (: E2) (: E3)))  )
    ))
```

Each rule is a two element list, the first element is the pattern, the second the skeleton.

```
(define (pattern  rule) (car  rule))
(define (skeleton rule) (cadr rule))
```

The pattern describes the syntax of the expression to which the rule applies. The skeleton describes the form of the result when the rule is applied. For example, the pattern of Rule 1 is a list of three elements. The first element is the literal `+`, the second and third elements (`C1` and `C2`) must each satisfy the restriction of being constants (numbers) for the rule to be applicable. When the rule is applicable, e.g. for an expression such as `(+ 2 3)` in the case of Rule 1, the skeleton is instantiated by replacing each pattern variable with its associated element in the pattern, e.g. `2` for `C1` and `3` for `C2`.

A list of the form (:< var >) in the skeleton is instantiated (i.e., replaced by the element in the pattern associated with < var >). The skeleton for Rule 1 consists of two elements. The first element `*value-of*` indicates that after the pattern variables in the second element are instantiated, the resulting expression is to be evaluated (in the global environment) as a Scheme expression.

Problem 4. Complete the definition of procedure `instantiate` below. You should make use of procedure `lookup` and the abstract selector procedures provided below.

```
(define (instantiate skeleton dictionary)
  (cond ((not (pair? skeleton)) <exp1>)
        ((skeleton-variable? skeleton) <exp2>)
        ((skeleton-evaluation? skeleton)
         (evaluate <exp3>))
        (else (cons (instantiate (car skeleton) dictionary)
                    (instantiate (cdr skeleton) dictionary)))))
(define (skeleton-variable? skeleton)
  (and (pair? skeleton) (eq? (car skeleton) ':)))
(define (substitution-variable skeleton)
  (cadr skeleton))
(define (skeleton-evaluation? skeleton)
  (and (pair? skeleton) (eq? (car skeleton) '*value-of*)))
(define (evaluation-expression skeleton)
  (cadr skeleton))
```

We are now in a position to consider how to transform symbolic expressions. For simplicity we consider first the case where there is only one rule, such as Rule 1 of the addition rules. We need to be able to handle cases such as

- `17`. The rule doesn't apply. There is no transformation. The value 17 is returned.
- `(+ 4 5)` The rule applies. The value 9 is returned.
- `(+ 4 (+ 2 3))` The rule does not apply to the expression, but it does apply to the subexpression `(+ 2 3)`. After applying the rule to this subexpression, the rule also applies to the resulting expression, `(+ 4 5)`.

```
(define (one-rule-transform rule)
  (define (transform-exp exp)
    (try-rule (if (pair? exp)
                  (map transform-exp exp)
                  exp)))
  (define (try-rule exp)
    (let ((dictionary (match (pattern rule)
                             exp
                             (make-empty-dictionary))))
      (if (eq? dictionary 'failed)
          exp
          (transform-exp
           (instantiate (skeleton rule) dictionary)))))
  transform-exp)
```

We now consider the more general case of multiple rules. One way to do this is to try each rule in sequence. If a rule applies instantiate the skeleton of that rule, then try each rule on the result. If there are no more rules to try, the expression cannot be transformed.

Problem 5. Complete the definition of procedure `make-transformer` below.

```
(define (make-transformer the-rules)
  (define (transform-exp exp)
    (try-rules (if (pair? exp)
                   (map transform-exp exp)
                   exp)))
  (define (try-rules exp)
    (define (scan rules)
      (if (null? rules)
          <exp1>
          (let ((dictionary (match (pattern <exp2>)
                                   exp
                                   (make-empty-dictionary))))
            (if (eq? dictionary 'failed)
                (scan <exp3>)
                (transform-exp (instantiate (skeleton <exp4>)
                                            dictionary))))))
    (scan <exp5>))
  <exp6>)
```

We are now in a position to consider one fairly complete set of rules of algebraic simplification.

1. $C_1 \mathrm{op} C_2 \rightarrow [C_1 \mathrm{op} C_2]$
2. $<\mathrm{exp}>\ \mathrm{op} C_1 \rightarrow C_1 \mathrm{op} <\mathrm{exp}>$
3. $0+<\mathrm{exp}>\rightarrow<\mathrm{exp}>$
4. $1\times<\mathrm{exp}>\rightarrow<\mathrm{exp}>$
5. $0\times<\mathrm{exp}>\rightarrow 0$
6. $C_1 \times (C_2 \times <\mathrm{exp}>) \rightarrow (C_1 \times C_2) \times <\mathrm{exp}>)$
7. $<\mathrm{exp1}> \times (C_2 \times <\mathrm{exp3}>) \rightarrow C_2 \times (<\mathrm{exp1}> \times <\mathrm{exp3}>)$
8. $(<\mathrm{exp1}> \times <\mathrm{exp2}>) \times <\mathrm{exp3}> \rightarrow <\mathrm{exp1}> \times (<\mathrm{exp2} \times <\mathrm{exp3}>)$
9. $C_1 + (C_2 + <\mathrm{exp3}>) \rightarrow [C_1 + C_2] + <\mathrm{exp3}>$
10. $<\mathrm{exp1}> +(C_2 + <\mathrm{exp3}> \rightarrow C_2 + (<\mathrm{exp2}> + <\mathrm{exp3}>$
11. $(<\mathrm{exp1}> + <\mathrm{exp2}>) + <\mathrm{exp3}> \rightarrow <\mathrm{exp1}> +(<\mathrm{exp2}> + <\mathrm{exp3}>)$
12. $C_1 \times <\mathrm{exp}> + C_2 \times <\mathrm{exp}> \rightarrow [C_1 + C_2] \times <\mathrm{exp}>$
13. $<\mathrm{exp1}> \times (<\mathrm{exp2}> + <\mathrm{exp3}>) \rightarrow (<\mathrm{exp1}> \times <\mathrm{exp2}>) + (<\mathrm{exp1}> + <\mathrm{exp3}>)$

where `op` is either $+$ or $\times$.

Using our conventional notation:

```
(define algebra-rules
  '(
    ( ((? OP) (? C1 constant?) (? C2 constant?))                   ;;; Rule 1
      (*value-of* ((: OP) (: C1) (: C2)))    )
    ( ((? OP) (? E1) (? C2 constant?))                             ;;; Rule 2
      ((: OP) (: C2) (: E1))       )
    ( (+ 0 (? E))                                                  ;;; Rule 3
      (: E)                        )
    ( (* 1 (? E))                                                  ;;; Rule 4
      (: E)                        )
    ( (* 0 (? E))                                                  ;;; Rule 5
      0                            )
    ( (* (? C1 constant?) (* (? C2 constant?) (? E3)))             ;;; Rule 6
      (* (*value-of* (* (: C1) (: C2))) (: E3))      )
    ( (* (? E1) (* (? C2 constant?) (? E3)))                       ;;; Rule 7
      (* (: C2) (* (: E1) (: E3)))  )
    ( (* (* (? E1) (? E2)) (? E3))                                 ;;; Rule 8
      (* (: E1) (* (: E2) (: E3)))  )
    ( (+ (? C1 constant?) (+ (? C2 constant?) (? E3)))             ;;; Rule 9
      (+ (*value-of* (+ (: C1) (: C2))) (: E3))      )
    ( (+ (? E1) (+ (? C2 constant?) (? E3)))                       ;;; Rule 10
      (+ (: C2) (+ (: E1) (: E3)))  )
    ( (+ (+ (? E1) (? E2)) (? E3))                                 ;;; Rule 11
      (+ (: E1) (+ (: E2) (: E3)))  )
    ( (+ (* (? C1 constant?) (? E)) (* (? C2 constant?) (? E)))   ;;; Rule 12
      (* (*value-of* (+ (: C1) (: C2))) (: E))        )
    ( (* (? E1) (+ (? E2) (? E3)))                                 ;;; Rule 13
      (+ (* (: E1) (: E2))
    (* (: E1) (: E3)))          )
    ))
```

We can then define an algebraic simplifier as:

```
(define alg-simp (make-transformer algebra-rules))
```

Problem 6. Test this simplifier on expressions such as:

- `17`
- `(+ 2 3)`
- `(+ 2 (* 3 1)))`
- `(+ (+ a 0) (* a 1))`

Are the results what you expected? Can you suggest how the rules might be changed to improve the results.

Problem 7. We are now in a position to use these procedures to accomplish symbolic differentiation.

```
(define deriv (make-transformer deriv-rules))
```

Where

```
(define deriv-rules
  '(
    ((DD (? C constant?) (? V variable?))
     0                                 )
    <more rules>
    ))
```

Complete the definition of `deriv-rules` to include all the cases of Sec. 2.3.2 of the notes. Add a rule that handles the rule for differenting u^n as in E2.57. Test your answer by differentiating $x \times (3 + y \times x) + y \times x^2$ with respect to x.

Problem 8. We are now in a position to simplify the result of differentiating a symbolic expression. Show how this can be done by completing the following procedure definition.

```
(define (simp-deriv expression)
   < exp >)
```

Test your `simp-deriv` procedure definition by differentiating (with respect to x) the algebraic expressions such as $x + 3$, $x \times y$, and $x \times y \times (x + 3)$.

Turning in your answers

The section on "Feedback" in the *Don't Panic* manual describes how to send Email from the 6.001 lab. Send your answer buffer to your tutor. Do this by starting a mail message and using the Edwin `M-x insert-buffer` command. Don't forget to include a reply email address (or else put your name in the message) so your tutor will know that the message is coming from you.

How much time did you spend on this homework assignment? Report separately time spent before going to lab and time spent in the 6.001 lab.

If you cooperated with other students working this problem set please indicate their names in the space below. As you should know, as long as the guidelines described in the *6.001 Policy on Cooperation* handout are followed, such cooperation is allowed and encouraged.