

CIS4/681 – Assignment 3 – Symbolic Differentiation/Matching

Due: April 24, 2008 – 3 weeks

1 Introduction

This assignment gives some additional experience with Lisp programming and with two general techniques commonly used in AI systems: data driven programming and rule based systems.

Data driven programming is a technique in which lisp programs, or executable lisp expressions, are associated with certain data items or classes of data items. A general program is written to process a stream of incoming data and operates, in part, by invoking the code associated with each piece of incoming data.

Rule based systems are a general class of systems in which much of the processing is encoded in the form of rules. A rule typically has two parts - a pattern and an action. Data is processed by finding a set of rules whose patterns “match” the data and then executing the corresponding actions.

2 The Problem

This assignment asks you to complete a symbolic differentiation system that uses both of these techniques. The partial system is listed in the figures of this document and will be available off of the web-site and in the file `~mccoys/cis4-681/progs/differentiator.lisp` (on either the composers or the cis machines). The goal of the system is to take an arithmetic expression containing variables and to produce another representing its first derivative with respect to the variable X. Thus given the expression:

$X^2 + 3X + 2$ -or- `(plus (power x 2) (times 3 x) 2)`

We want to produce the new expression:

$2X + 3$ -or- `(plus (times 2 x) 3)`

The standard rules for differentiation are given in Figure 1. In this figure “x” stands for the variable “x”, “*<constant>*” for any constant symbol, “u” and “v” for any arbitrary arithmetic expressions, and “du” and “dv” for their derivatives. Note that the rules are recursive. The fourth rule, for example, states that the derivative of the sum of two expressions is the sum of their derivatives.

form	derivative
x	1
<constant>	0
-u	-du
u + v	du + dv
u - v	du - dv
u * v	(u * dv) + (v * du)
u / v	(v*du - u*dv) / v*v
u**n	n * u**(n-1) * du
sin(u)	cos(u) * du
cos(u)	-sin(u) * du

Figure 1: Differentiation Rules

3 The Differentiator

Figure 2 shows the incomplete differentiation program. The function DDX takes a single lisp arithmetic expression and returns its first derivative with respect to the variable X. If the expression is X itself, then

```

(defun ddx (E)
  "this function takes as input an arithmetic
  expression and returns its derivative with
  respect to x"
  (declare (special E))
  (cond ((equal E 'x) 1)
        ((atom E) 0)
        ((and (atom (car E)) (get (car E) 'differentiator))
         (eval (get (car E) 'differentiator)))
        (t (list 'ddx E))))

(defun diff (function-name code)
  "this function is used to put differentiator
  code on the property lists of the function name"
  (setf (get function-name 'differentiator) code))

; u + v -> du + dv
(diff 'plus '(cons (car E) (mapcar (function ddx) (cdr E))))

```

Figure 2: The Differentiator

“1” is returned. Any other atomic symbol is taken as a constant and a “0” is returned. If the expression has an atomic CAR which has a DIFFERENTIATOR property, then the value of that property is evaluated to yield the derivative. Note that the variable “E” is bound to the expression to be differentiated. If none of the above conditions are true, then DDX represents the derivative of E by (DDX E).

As a part of this assignment you will provide DIFFERENTIATOR properties for the appropriate atoms to allow this program to handle all of the rules in Figure 1 (e.g., PLUS, DIFFERENCE, TIMES, POWER, etc...). If you do this in a straight forward way you may be surprised to discover that DDX sometimes returns derivative expressions which, although correct, are in a form that we would not like to accept. For example:

```

--> (ddx '(plus (power x 2) (times x 3) 2))

(plus (times (times 2 (power x (difference 2 1))) 1)
      (plus (times x 0) (times 3 1))
      0)

```

One solution to this problem is to make the differentiation rules more specific, checking for cases which would produce reducible expressions. The rule for PLUS, for example, (in *very bad lisp style*) might be:

```

(diff 'plus '(prog (args)
  ;; differentiate the arguments
  (setf args (mapcar 'ddx (cdr E)))
  ;; x + 0 -> x
  (setf args (delete 0 args))
  ;; check for cases of 0 or 1 argument left
  (return (cond ((null args) 0)
                ((null (cdr args)) (car args))
                (t (cons 'plus args))))))

```

Although this approach will certainly work, a more general, and preferable, one is to write a general package that knows how to “reduce” or “simplify” arithmetic expressions.

4 The Simplifier

A second part of your assignment is to complete a partial implementation of a rule driven simplification package for reducing Lisp expressions. The partial package is shown in Figure 3. The simplifier consists of two main functions: SIMPLIFY and TRANSFORM and a global list of simplification rules stored in the variable “*simplification-rules*”. The function SIMPLIFY takes an arbitrary Lisp arithmetic expression and returns a simplified version. If its argument E is an atom, it returns it unchanged. Otherwise it recursively applies itself to the elements of E and then feeds the resulting list to the function TRANSFORM which is responsible for the application of appropriate simplification rules.

TRANSFORM iterates down the list of known simplification rules (the value of “*simplification-rules*”) until one is found that applies and then applies it, replacing the s-expression with the resulting simplified one. It continues by looking anew for rules to apply to this s-expression. Transform stops and returns the s-expression when no rule in its list is found to apply.

4.1 Simplification Rules and Pattern Matching

A simplification rule is represented by a tuple and has the form: (*pattern*)*result*). The *pattern* specifies the class of s-expressions that the rule applies to and the *result* gives the transformed s-expression. In a rule, atoms that begin with the character “&” or “*” are treated as variables in the pattern matching process. The variable “&FOO”, for example, will match any single s-expression and the matching process will return a binding for “&FOO that corresponds to the single element that was matched. The variable “*BAR” can match any sequence of zero or more s-expressions in a list and its returned binding will be the list containing the sequence of s-expressions that it matched. The following rule, for example, eliminates single argument PLUS calls:

```
( (PLUS &x) &x)
```

where as this one merges embedded calls to PLUS:

```
( (PLUS *A (PLUS *B) *C) (PLUS *A *B *C) )
```

If the pattern part of a rule is found to NOT match an s-expression it will return FAIL. If it does match then MATCHER will return a binding list and then the result part of this rule invocation is formed via a call to the function INSTANTIATE. This function simply takes an s-expression and a bindings list and returns a new s-expression formed by replacing all pattern match variables with their values from the binding list. Note that the value of a * type variable must be “spliced” into the list it occurs in.

For the second part of your assignment you will be asked to provide simplification rules appropriate for the differentiation problem. Figure 4 gives some examples of appropriate rules you should include. In order to actually run the simplifier, however, you will have to complete yet another incomplete package, the MATCHER.

5 The Matcher

The third partial system you must complete involves a general pattern matching function and an associated pattern instantiation function. MATCHER takes two s-expressions as arguments. The first is some datum to be matched and the second is the “pattern” which may contain variables. If the datum matches the pattern, then MATCHER returns a list of variable binding pairs; it returns FAIL otherwise. The variable binding list is of the form: ((var1 val1) (var2 val2) ... (varn valn)). E.g.,

```
--> (matcher '(foo (bar 1 2 3 4 5)) '(foo (bar &a *b)))  
((&a 1) (*b (2 3 4 5)))
```

The final function for this part of the assignment is INSTANTIATE. INSTANTIATE takes some s-expression possibly containing variables, and a list of variable-bindings (where each variable in the s-expression must occur in the variable-bindings list). INSTANTIATE should return a new expression formed by “replacing” any variables in the original expression with their values as indicated by the variable-binding

```

(defun simplify (E)
  "this function is used to simplify a Lisp expression"
  (cond ((atom E) E)
        (t (transform (mapcar 'simplify E)))))

(setf *simplification-rules* nil)

(defun transform (E)
  "transform an expression E by applying all rules
   (from *simplification-rules*) matching E and returning
   the new expression"
  (prog (rules pattern match-bindings result)

    top
    ;start with the initial list containing all rules
    (setf rules *simplification-rules*)

    next-rule
    ; if there are no more rules, then we are done
    (cond ((null rules) (return E)))

    ; pull off the next rule
    (setf rule (car rules))
    (setf rules (cdr rules))
    ; get the rule's pattern and result part
    (setf pattern (car rule))
    (setf result (cadr rule))

    ; if the pattern matches, then replace E with the result
    ; after instantiating it
    (setf match-bindings (matcher E pattern))

    (cond ((listp match-bindings)
           (setf E (instantiate result
                                match-bindings))
           (go top))
          (t (go next-rule)))))

(defun srule (pattern result)
  "function for defining *simplification rules*"
  (setf *simplification-rules* (cons
                                (list pattern result)
                                *simplification-rules*)))

;simplification rules for (PLUS ...)
(srule '(plus *a 0 *b) '(plus *a *b))
(srule '(plus *a (plus *b) *c) '(plus *a *b *c))
(srule '(plus &a (minus &b)) '(difference &a &b))
(srule '(plus (minus &b) &a) '(difference &a &b))

```

Figure 3: The Simplifier

pattern	result
x+0, 0+x	x
x-0	x
0-x	-x
x*0, 0*x	0
x*1, 1*x	x
x/1	x
0/x	0
--x	x
x**0	1
x**1	x
x**-n	1/x**n
-x+y	y-x
x-(-y)	x+y

Figure 4: Some Simplification Rules (in Infix)

```
(defun matcher (data pattern)
  "takes an s-expression and a pattern -- which is an s-expression
  which may contain variables of both the & and the * type. Recall that
  a &-variable must match exactly one element of the input while a
  *-variable matches 0 to n elements of the data. If the data does not
  match the pattern, the function returns FAIL. Otherwise matcher
  returns a list of variable-value pairs such that if the variables in
  the pattern were replaced by their values (in the appropriate way) the
  data and the pattern would be equal. Example:
  (matcher '(1 2 foo 3) '(*a foo &b *c)) returns
  ((*a (1 2)) (&b 3) (*c ()))"
  ; <<put your code here!>>
)

(defun instantiate (E bindings)
  "this function replaces &x and *x variables with their
  values in the expression E, bindings is a list containing
  variable-value pairs where each variable in E is contained in the
  bindings list. E.g. if &a = '(foo bar) and *b = '(1 2 3) then
  (instantiate '(foo (&a) *b) '(&a (foo bar)) (*b (1 2 3))))
  returns: (foo ((foo bar)) 1 2 3)"
  ; <<put your code here!>>
)

;;; the function explode and char1 may be useful and are provided
;;; in the file "differentiator.lisp"
```

Figure 5: The Matcher

list. Note, however, that “&” variables are simply replaced by their values but the values of “*” variables must be spliced into place. As an example:

```
---> (instantiate '(mumble *b &a) '((&a 1) (*b (2 3 4 5))))  
(mumble 2 3 4 5 1)
```

5.1 For 681 Students

We might want to have a rule that implements the following transformation:

```
x+x --> 2 * x
```

Try adding the simplification rule:

```
(srule '(plus &a &a) '(times 2 &a))
```

Make sure that your MATCHER function is written in such a way so as to allow this rule to work.

6 The Assignment

Your assignment is as follows:

1. (20 points) Complete the basic differentiation package by supplying differentiation forms for the functions PLUS, DIFFERENCE, TIMES, QUOTIENT, ADD1, SUB1, MINUS, POWER, SIN, and COS. Test DDX on the following expressions:

```
(plus (power x 2) (times x 3) 2)
```

```
(quotient (times 3 (power x 3)) (plus x y))
```

```
(difference (cos x) (sin x))
```

```
(cos (quotient (plus x 1) (plus x (minus 1))))
```

2. (20 points) Complete the simplification package by adding the appropriate simplification rules for arithmetic expressions (e.g., $-(-x) = x$, $x*0 = 0$, etc.). This can not be tested until you complete the next part of the assignment.
3. (60 points) Write the heart of the matcher by writing the MATCHER and INSTANTIATE functions. Now define a new function, DIFFERENTIATE, that returns a simplified form of the first derivative of an arithmetic expression (using, of course, both SIMPLIFY and DDX). Show the result of this function on the four test cases above.