CS5820 AI - Homework 1

Cody Hanson
University of Colorado Colorado Springs
1420 Austin Bluffs Pkwy
Colorado Springs, Colorado USA 80918
chanson@uccs.edu

Abstract

This assignment's goal was to build a 'rules engine' which could apply patterns to text. As a practical application of our rules engine, we were also tasked with putting it to work using it to build a symbolic differentiation system, which can take derivatives of arbitrary mathematical expressions.

Introduction

One of the key things to consider when approaching this problem, is which tool to use during implementation. The examples were mostly in Lisp syntax which hinted that perhaps Lisp would be an ideal match for this kind of problem. I initially started out using Python to code my rules engine, but I quickly found it to be kind of unwieldy, when manipulating patterns in the way that we needed to for this assignment.

I then switched to Lisp, and after a bit of time getting up to speed on the language, I found that it was an ideal choice. The ability to easily parse and manipulate arbitrary tree structures proved invaluable for building the rules engine and the differentiation program. Lisp was also able to succinctly express algorithms using recursion, which help to minimize the amount of code that I had to write.

Manipulating Rules

The core of my rules engine was implemented by 3 main functions: 'sub' for taking in a given association list and making substitutions, 'match' which determined if the left hand side of a rule matched an input string, and if it matched, generate association list to be input into 'sub', and 'applyOneRule', which applied the first matching rule from a list of rules, or none if the rules didn't match.

The way that I determined if the left hand side of a rule matched a phrase was to implement a 'zip-uneven' function, which allowed me to match up the elements of a phrase with the elements of a rule, to see if they were compatible. 'zip-uneven' worked by walking each list and creating pairs of items, and padding with NIL in case one list was longer than the other. Then with the 'match' function, I would walk the pairs and determine if they were equal to each other (in the case of two atoms) or if one atom lined up with a 'substitution pattern' such as '(? x)', then we would know that we needed to substitute those two.

I chose to implement 'applyOneRule' with foresight to the differentiation application. With differentiation, you woudn't want to apply more than one rule at a time, otherwise you wouldn't get the 1st derivative, but the 2nd or 3rd.

File input techniques

I devised a system to be able to read rules from a file, as well as test inputs and expected outputs. This enabled the program to be more flexible, and easier to add new rules and test inputs, rather than hard code each test case in the main program.

In a rules file, there was one rule per line, with the left hand side and right hand side being separated by a space. Due to the way Lisp parsed this file, you are allowed to add a comment to the end of a line after a valid rule, and it would be ignored. This was useful to be able to leave notes on rules and test cases.

In a test input file, there would be one test input and expected output pair per line. I was able to compute the expected outputs by hand, because I had knowledge of the kinds of rules that would be present. By having arbitrarily many test inputs in a file, and a main loop in the 'Rules Engine Demo' program, I was able to easily add new test cases, without changing the code, as well as automatically evaluate if the program output matched what the expected output was.

Symbolic Differentiation

My approach to implementing a differentiation engine was to leverage the rules engine to perform a single differentiation step, based on rules stored in a file. The key next step was to write the rules in such a way such that, if a rule required another differentiation step, that the output of the first rule application would have an embedded '(ddx ¡some expression¿)' in it. After the top level differentiation step, I input the output of that first step into a function which would recursively look for embedded 'ddx' symbols, which would require the application of another rule. This would continue until all embedded 'ddx' symbols were evaluated.

Testing and results

To ensure adequate coverage of all cases, I specified a test input that exercised every differentiation rule. I also included differentiable internal functions, to test that recursive differentiation was tested, such as in the rules for the cosine.

I was able to obtain successful answers for all expressions that I attempted. The answers were programmatically checked against a human derived lisp format string using the lisp function ‘tree-equal’.

Future Work

One way that I could improve my system is to find a way to mitigate stack overflow issues. Due to the use of recursion, given a sufficiently complex input, it could cause the Common Lisp interpreter to suffer a stack overflow error, due to too many recursive calls.

Partial differentiation, with respect to variables besides ‘x’ could also be a good improvement, to allow for use in more complex mathematical problems.

Allowing the user to have an interactive input on the command line would enable the testing of expressions without adding them to the test input file. If possible being able to specify the expected output as well could allow the program to automatically check the success of the differentiation would reduce human error, similarly to how I implemented automatic checking in my file input method.

Another thing to consider, is that of the rules database files. Currently they are applied in the order they are in the file, such that the first to match will win. Care should be taken to ensure that rules for differentiation are placed in the correct order.

Finally, given more time, it would be a big improvement to implement some simplification logic. For instance, if there was a term such as ‘(difference 2 1)’ this could easily be evaluated to ‘1’. This would make some of the output expressions less lengthy, and easier for a human to process.

Appendix A - Rules Engine Code and Output

Rules Engine Code

```
;Cody Hanson
;CS5820 Artificial Intelligence
;Spring 2014 — Professor Kalita

;returns the right hand side of a rule
(defun rhs (rule)
    (nth 1 rule))

;returns the left hand side of a rule
(defun lhs (rule)
    (nth 0 rule))

;returns true if the argument is a valid rule. false otherwise
(defun ruleP (rule)
  (and (listp (nth 0 rule)) ;first item is a list
       (listp (nth 1 rule)) ;second item is a list
       (null  (nth 2 rule))); and there are only 2 items
  )

;performs a substitution, with the rhs of a rule,
;and an association list of substitutions
(defun sub (input subs)
   (cond
      ;if no input, nothing to do.
      ((null input) NIL)
      ;if no subs, just return the input
      ((null subs) input)
      ;if it is a simple word, just keep going.
      ((atom (car input)) (cons (car input) (sub (cdr input) subs)))
      ;if it is a nested list, and also a replacement marker (? something), do replacement
      ((and (listp (car input)) (equal (car (car input)) '?))
          (loop for pair in subs do
              (if (equal (car pair) (nth 1 (car input)))
                  (return (cons (nth 1 pair)  (sub (cdr input) subs)))
              )
          ))
      ;if it is just a nested list, recurse on both the first element, and rest of the list.
      ((listp (car input)) (cons (sub (car input) subs) (sub (cdr input) subs)))
    ))

;Applys a single rule to an arbitrary tree structure.
(defun applyRule (tree rule)
  ;(format t "tree: ~S~%" tree)
  ;(format t "rule: ~S~%" rule)
    ;matches return the applied rule
  (setf matches (match tree (lhs rule)))
  (cond
    ((not (endp matches)) (sub (rhs rule) matches)) ;there was a match
    (T tree) ;else just return the tree.
    )
  )


;Iterates through a list of rules, applying each of them in order.
;If a rule doesn't match, the tree is passed over unaffected.
(defun applyAllRules (tree rules)
  ;(print 'applyingRules)
  (loop for rule in rules do (setf tree (applyRule tree rule)))
  ;(format t "tree after all rules: ~A~&" tree)
  tree
)
```

```
;Iterates through a list of rules, applying the first one that matches
;If a rule doesn't match, the tree is passed over unaffected.
(defun applyOneRule (tree rules)
  ;(print "applying One Rule")
  (loop for rule in rules do
     (setf newtree (applyRule tree rule))
     ;if the newtree is different, we know a rule was applied. so stop looping
     (if (not (tree-equal tree newtree))
       ;if the newTree is a single 1 element list
       ;we want to simplify it and convert it into an atom.
     (progn
       (if (and (eq 1 (list-length newtree)) (atom (car newtree)))
         (setf newtree (car newtree))
       )
       ;now return the tree with the one rule applied
       ;(format t "applying One Rule newtree: ~A~%" newtree)
       (return-from applyOneRule newtree)
     )
     )
  )
  ;otherwise, just return tree, no rules were applied.
  tree
)



;Determines whether a rule matches a given piece of text.
;If so, returns the association list, for the substitution. otherwise, NIL
(defun match (tree lhsrule)
  (let (associations null)
    (setf pairs (zip-uneven tree lhsrule))

    ;first we address the special case where we have a single atom, and a single replacement.
    ;this enables doing a direct match and replace, without a larger pattern for context.
    (if (and  (listp tree) ;its a list
          (eq 1 (list-length tree)) ;of one length
          (atom (car tree)) ;and the one thing is a simple atom.
          (listp lhsrule) ;and the rule is a list
          (eq 1 (list-length lhsrule)) ;of one length
          (atom (car lhsrule)) ;and the one item is an atom
          (eq (car lhsrule) (car tree))) ;and they match each other
       (return-from match (cons '(? x) (car lhsrule)))
     )

    ;(format t "Match Pairs ~S~%" pairs)
    (loop for pair in pairs do
     (cond
      ;if the pairs of atoms are equal, then keep going.
      ((and (atom (nth 0 pair)) (atom (nth 1 pair)) (eq (nth 0 pair) (nth 1 pair))) T)
      ;if the first is an atom, and the next is a substitution,
      ;like (? x),add to the subs
      ((and (atom (nth 0 pair))
          (listp (nth 1 pair))
          (eq (nth 0 (nth 1 pair)) '?)
          (atom (nth 1 (nth 1 pair)))) (setf associations (cons (list (nth 1 (nth 1 pair)) (
              nth 0 pair)) associations)))
      ;if the first is a nested list, and the next is a substitution,
      ;like (? x),add to the subs
      ((and (listp (nth 0 pair))
          (listp (nth 1 pair))
          (eq (nth 0 (nth 1 pair)) '?)
          (atom (nth 1 (nth 1 pair)))) (setf associations (cons (list (nth 1 (nth 1 pair)) (
              nth 0 pair)) associations)))
      ;if not, abort, they don't match. return an empty subs list
      (T (return-from match null)) ;TODO should this be NIL instead of null?
      ))
    ;(format t "Match Associations: ~S~%" associations)
    (return-from match associations)
  )
)

;Zips together two lists into pairs of items, even if they are not the same length
;padds with NIL if one list is longer.
(defun zip-uneven (list1 list2)
  (cond
    ((and (endp list1) (endp list2)) '()) ;base case. return empty list.
    ((and (endp list1) (not (endp list2)))  ;list2 is longer than list1
      (cons (list NIL (car list2)) (zip-uneven NIL (cdr list2))))
    ((and (not (endp list1)) (endp list2)) ;list1 is longer than list2
      (cons (list (car list1) NIL) (zip-uneven (cdr list1) NIL)))
    (T  ;both the same length
      (cons (list (car list1) (car list2)) (zip-uneven (cdr list1) (cdr list2))))
    )
  )

;Opens a filename and reads lines into a list
(defun get-file (filename)
    (with-open-file (stream filename)
      (loop for line = (read-line stream nil)
          while line
          collect line)))


;found this at http://faculty.hampshire.edu/lspector/courses/string-to-list.lisp
;converts a string of items into a list
(defun string-to-list (string)
  "Returns␣a␣list␣of␣the␣data␣items␣represented␣in␣the␣given␣list."
  (let ((the-list nil) ;; we'll build the list of data items here
        (end-marker (gensym))) ;; a unique value to designate "done"
    (loop (multiple-value-bind (returned-value end-position)
                               (read-from-string string nil end-marker)
            (when (eq returned-value end-marker)
              (return the-list))
            ;; if not done, add the read thing to the list
            (setq the-list
                  (append the-list (list returned-value)))
            ;; and chop the read characters off of the string
            (setq string (subseq string end-position))))))

;takes a list of strings, and turns each of them into a list.
;returns the list of lists
(defun process-list-of-strings (stringList)
    (cond
      ((null stringList) (list))
      (T (cons (string-to-list (car stringList)) (process-list-of-strings (cdr stringList))))
    )
  )
```

Rules Engine Demo Code

```
;Demo program for the rules engine
;examples not specific to calculus.

;import the rules engine functions
(load "rules_engine.lisp")

;load the rules.
```

```
(setf ruleStrings (get-file "./testrules"))
(setf rulesList (process-list-of-strings ruleStrings))

;load the test cases.
(setf testPhrases (get-file "./testphrases"))
(setf testPhrasesList (process-list-of-strings testPhrases))

;;;;;;;;;;;;;;;;;;;;;;;;;
;Main Program
;;;;;;;;;;;;;;;;;;;;;;;;;


;Run the tests on each case in the input list
(loop for testCase in testPhrasesList do
  (print "------------------------------------------------")
  (setf input (nth 0 testCase))
  (setf expectedResult (nth 1 testCase))
  ;compute the derivative
  (setf result (applyAllRules input rulesList))
  ;determine if it matches what we expected.
  (format t "~&input- ~A~&" input)
  (format t "result- ~A~&" result)
  (format t "expectedResult- ~A~&" expectedResult)
)
```

Rules Engine Demo Rules

```
(Man (? x)) (Mortal (? x))
(Woman (? x)) (Mortal (? x))
(God (? x)) (Immortal (? x))
(Good things are (? e)) (Is your product very (? e))
(x) (1) ;testing simple substitution
```

Rules Engine Demo Input

```
(Man Socrates) (Mortal Socrates)
(Woman Madonna) (Mortal Madonna)
(God Athena) (Immortal Athena)
(Good things are expensive) (Is your product very expensive)
(x) (1)
```

Rules Engine Demo Output

```

"------------------------------------------------"
input- (MAN SOCRATES)
result- (MORTAL SOCRATES)
expectedResult- (MORTAL SOCRATES)

"------------------------------------------------"
input- (WOMAN MADONNA)
result- (MORTAL MADONNA)
expectedResult- (MORTAL MADONNA)

"------------------------------------------------"
input- (GOD ATHENA)
result- (IMMORTAL ATHENA)
expectedResult- (IMMORTAL ATHENA)

"------------------------------------------------"
input- (GOOD THINGS ARE EXPENSIVE)
result- (IS YOUR PRODUCT VERY EXPENSIVE)
```

```
expectedResult- (IS YOUR PRODUCT VERY EXPENSIVE)

"______________________________________________________"
input- (X)
result- (1)
expectedResult- (1)
```

Appendix B - Differentiation Code and Output

Differentiation Code

```
; Symbolic Differentiation, using rules_engine.lisp
; Cody Hanson
; Spring 2014 - CS5860-AI UCCS

;import the rules engine functions
(load "rules_engine.lisp")
(print "Rules Engine Loaded")

;load the rules.
(setf ruleStrings (get-file "./derivativerules"))
(setf rulesList (process-list-of-strings ruleStrings))

(print "Rules List Loaded")

;load the test cases.
(setf testDerivatives (get-file "./derivativeTestInput"))
(setf testDerivativesList (process-list-of-strings testDerivatives))

(print "Test case List Loaded")

;returns true if a list is a single level deep.
(defun flat-list (aList)
  (if (atom aList)
    (return-from flat-list NIL))
  (loop for item in aList do
    (cond
      ((not (atom item)) (return-from flat-list NIL))
    )
  )
  T
)

;Define our derivative function.
(defun ddx (expression rulesList)
  ;(format t "expression: ~S~%" expression)
  (cond
    ;the order of rules is important.
    ((null expression) NIL)
    ((atom (nth 0 expression))
        (embeddedDdx (applyOneRule expression rulesList) rulesList))
    (T (print 'GotDownHere))
    )
  )

;given an expression which has been differentiated once already,
;search for embedded (DDX ) froms, that require further derivations, and apply the ddx
    function.
(defun embeddedDdx (expression rulesList)
  ;(print "embeddedDdx")
  ;(print expression)
    (cond
      ((null expression) NIL)
      ((atom expression) expression) ;nothing to do!
      ((and (atom (nth 0 expression)) (eq 'DDX (nth 0 expression))) ;found an embedded ddx
       (ddx (car (cdr expression)) rulesList))
      ((atom (nth 0 expression))
        (cons (car expression) (embeddedDdx (cdr expression) rulesList))
      )
      ((listp  (nth 0 expression))
        (cons (embeddedDdx (car expression) rulesList) (embeddedDdx (cdr expression)
            rulesList))
        )
      )
)
;;;;;;;;;;;;;;;;;;;;;;;;
;Main Program
;;;;;;;;;;;;;;;;;;;;;;;;

(setf resultsList (list))

;Run the tests on each case in the input list
(loop for testCase in testDerivativesList do
  (print "—————————————————————————————————")
  (setf input (nth 0 testCase))
  (setf expectedResult (nth 1 testCase))
  ;compute the derivative
  (setf result (ddx input rulesList))
  ;determine if it matches what we expected.
  (setf matched (tree-equal  expectedResult result))
  (format t "~&input-␣~A~&" input)
  (format t "result-␣~A~&" result)
  (format t "expectedResult-␣~A~&" expectedResult)
  (format t "Match?:␣~A~&" matched)
  (setf resultsList (append resultsList (list matched)))
)

(format t "~&Finished.␣Results␣Array:␣~A~&" resultsList)
```

Differentiation Rules

```
(power x (? n) ) (times (? n) (power x (difference (? n) 1))) ; exponentiation by constant
(plus (? u) (? v)) (plus (ddx (? u)) (ddx (? v)))
(difference (? u) (? v)) (difference (ddx (? u)) (ddx (? v)))
(times (? n) x) ((? n)) ; multiplication by constant
(times (? u) (? v)) (plus (times (? u) (ddx (? v))) (times (? v) (ddx (? u)))) ;
    multiplication by two functions
(divide (? u) (? v)) (divide (difference (times (? u) (ddx (? v))) (times (? v) (ddx (? u))))
     ( power (? v) 2)) ; division of two functions
(negative (times 2 x)) (negative 2) ;ddx of a negative is the negative of the derivative of a
     positive
(cos (? u)) (times (negative (sin (? u))) (ddx (? u))) ; derivative of cosine
(sin (? u)) (times (cos (? u)) (ddx (? u))) ; derivative of sin
(tan (? u)) (times (plus 1 (power (tan (? u)) 2)) (ddx (? u))) ; derivative of tan
(exp (? u)) (times (exp (? u)) (ddx (? u ))) ;derivative of the exponential
(sqrt (? u)) (times (divide 1 2) (divide (ddx (? u)) (sqrt (? u))))
(log (? u)) (divide (ddx (? u)) (? u)) ; logarithms.
(x) (1) ; ddx x = 1 needs to be above ddx c = 0
((? n)) (0) ; ddx c = 0
```

Differentiation Test Input

```
(times 100 x) 100 ;multiplication by constant
(times (power x 2) (power x 5)) (plus (times (power x 2) (times 5 (power x (difference 5 1)))
    ) (times (power x 5) (times 2 (power x (difference 2 1)))))
(divide (power x 2) (power x 5)) (divide (difference (times (power x 2) (times 5 (power x (
    difference 5 1)))) (times (power x 5) (times 2 (power x (difference 2 1))))) (power (
    power x 5) 2))
(power x 2) (times 2 (power x (difference 2 1))) ;x raised to a power
(plus (power x 2) (times 3 x))  (plus (times 2 (power x (difference 2 1))) 3) ;derivitive of
    sum is sum of derivatives
(difference (power x 2) (times 3 x))  (difference (times 2 (power x (difference 2 1))) 3)
(cos (power x 2)) (times (negative (sin (power x 2))) (times 2 (power x (difference 2 1))))
(sin (power x 2)) (times (cos (power x 2)) (times 2 (power x (difference 2 1))))
```

```
(tan (power x 2)) (times (plus 1 (power (tan (power x 2)) 2)) (times 2 (power x (difference 2
    1)))) ; derivative of tan
(exp (power x 2)) (times (exp (power x 2)) (times 2 (power x (difference 2 1))))
(log (power x 100)) (divide (times 100 (power x (difference 100 1))) (power x 100))
(sqrt (power x 2)) (times (divide 1 2) (divide (times 2 (power x (difference 2 1))) (sqrt (
    power x 2))))
(x) 1 ;derivation of x
(10) 0 ;derivation of a constant
```

Differentiation Test Output

```
"Rules Engine Loaded"
"Rules List Loaded"
"Test case List Loaded"
"_______________________________________________________"
input- (TIMES 100 X)
result- 100
expectedResult- 100
Match?: T

"_______________________________________________________"
input- (TIMES (POWER X 2) (POWER X 5))
result-
(PLUS (TIMES (POWER X 2) (TIMES 5 (POWER X (DIFFERENCE 5 1))))
 (TIMES (POWER X 5) (TIMES 2 (POWER X (DIFFERENCE 2 1)))))
expectedResult-
(PLUS (TIMES (POWER X 2) (TIMES 5 (POWER X (DIFFERENCE 5 1))))
 (TIMES (POWER X 5) (TIMES 2 (POWER X (DIFFERENCE 2 1)))))
Match?: T

"_______________________________________________________"
input- (DIVIDE (POWER X 2) (POWER X 5))
result-
(DIVIDE
 (DIFFERENCE (TIMES (POWER X 2) (TIMES 5 (POWER X (DIFFERENCE 5 1))))
  (TIMES (POWER X 5) (TIMES 2 (POWER X (DIFFERENCE 2 1)))))
 (POWER (POWER X 5) 2))
expectedResult-
(DIVIDE
 (DIFFERENCE (TIMES (POWER X 2) (TIMES 5 (POWER X (DIFFERENCE 5 1))))
  (TIMES (POWER X 5) (TIMES 2 (POWER X (DIFFERENCE 2 1)))))
 (POWER (POWER X 5) 2))
Match?: T

"_______________________________________________________"
input- (POWER X 2)
result- (TIMES 2 (POWER X (DIFFERENCE 2 1)))
expectedResult- (TIMES 2 (POWER X (DIFFERENCE 2 1)))
Match?: T

"_______________________________________________________"
input- (PLUS (POWER X 2) (TIMES 3 X))
result- (PLUS (TIMES 2 (POWER X (DIFFERENCE 2 1))) 3)
expectedResult- (PLUS (TIMES 2 (POWER X (DIFFERENCE 2 1))) 3)
Match?: T

"_______________________________________________________"
input- (DIFFERENCE (POWER X 2) (TIMES 3 X))
result- (DIFFERENCE (TIMES 2 (POWER X (DIFFERENCE 2 1))) 3)
expectedResult- (DIFFERENCE (TIMES 2 (POWER X (DIFFERENCE 2 1))) 3)
Match?: T

"_______________________________________________________"
```

```
input- (COS (POWER X 2))
result- (TIMES (NEGATIVE (SIN (POWER X 2))) (TIMES 2 (POWER X (DIFFERENCE 2 1))))
expectedResult- (TIMES (NEGATIVE (SIN (POWER X 2))) (TIMES 2 (POWER X (DIFFERENCE 2 1))))
Match?: T

"_______________________________________________"
input- (SIN (POWER X 2))
result- (TIMES (COS (POWER X 2)) (TIMES 2 (POWER X (DIFFERENCE 2 1))))
expectedResult- (TIMES (COS (POWER X 2)) (TIMES 2 (POWER X (DIFFERENCE 2 1))))
Match?: T

"_______________________________________________"
input- (TAN (POWER X 2))
result-
(TIMES (PLUS 1 (POWER (TAN (POWER X 2)) 2))
 (TIMES 2 (POWER X (DIFFERENCE 2 1))))
expectedResult-
(TIMES (PLUS 1 (POWER (TAN (POWER X 2)) 2))
 (TIMES 2 (POWER X (DIFFERENCE 2 1))))
Match?: T

"_______________________________________________"
input- (EXP (POWER X 2))
result- (TIMES (EXP (POWER X 2)) (TIMES 2 (POWER X (DIFFERENCE 2 1))))
expectedResult- (TIMES (EXP (POWER X 2)) (TIMES 2 (POWER X (DIFFERENCE 2 1))))
Match?: T

"_______________________________________________"
input- (LOG (POWER X 100))
result- (DIVIDE (TIMES 100 (POWER X (DIFFERENCE 100 1))) (POWER X 100))
expectedResult- (DIVIDE (TIMES 100 (POWER X (DIFFERENCE 100 1))) (POWER X 100))
Match?: T

"_______________________________________________"
input- (SQRT (POWER X 2))
result-
(TIMES (DIVIDE 1 2)
 (DIVIDE (TIMES 2 (POWER X (DIFFERENCE 2 1))) (SQRT (POWER X 2))))
expectedResult-
(TIMES (DIVIDE 1 2)
 (DIVIDE (TIMES 2 (POWER X (DIFFERENCE 2 1))) (SQRT (POWER X 2))))
Match?: T

"_______________________________________________"
input- (X)
result- 1
expectedResult- 1
Match?: T

"_______________________________________________"
input- (10)
result- 0
expectedResult- 0
Match?: T
Finished. Results Array: (T T T T T T T T T T T T T T)
```