# University of Colorado, Colorado Springs

## Home Work Assignment 1

Due 03-03-2014

---

I would like for you to provide me with a formal write-up with your programs. Use the AAAI author style for this write-up. Please make a section for each of the questions, and a sub-section for each of the sub-questions. Please explain briefly in each section or sub-section how you approached the problem, what data you used for testing, what results you obtained, what problems you faced, how you overcame your problems, how you can improve the current program, etc. The write-up for each section doesn't have to be long, but try to address relevant issues as mentioned above. Attach your code and test runs as appendices to this document.

---

1. **Manipulating Rules:** In this home work assignment, we are going to work with rules. You will develop your own simple rule engine in this part of the home work.

   A *rule* is simply a list with two patterns.

   (a) Let a *rule* be a list of two patterns called the Left Hand Side and the Right Hand Side. Define a function ruleP which checks to see if a list given to it is a rule. Next define functions lhs and rhs so that when given a rule, they return the appropriate patterns.

   (b) Define a function substitute which takes two arguments. The first argument is a list with pattern variables. The second argument is an association list in which each element is a pair. The first element of the pair is the name of a variable and the second element is its replacement value. We are writing this in LISP syntax below; we can write the same using a variety of syntaxes in other languages. We may be able to use regular expressions as well.

```
>(substitute '(color (? x) (? y)) '((x apple) (y red)))
(color apple red)
>(substitute '(red (* x) banana (?y))
             '((x (apples and plums.)) (y yellow)))
(red apples and plums. banana yellow)
```

   (c) Define a function applyRule which takes two arguments tree and rule. tree is an arbitrary hierarchical or tree structure or expression (in a language like LISP) with pre-order syntax, and rule is a rule defined earlier. If (lhs rule) matches tree, applyRule should return something like

```
(substitute (rhs rule) (match (lhs rule) tree)).
```

   Otherwise, applyRule should return tree itself. Test on examples such as

```
>(applyRule '(I am depressed)
            '((I am (? x)) (Why are you (? x) ?)))
(Why are you  depressed ?)
>(applyRule '(Man Socrates)
            '((Man (? x)) (Mortal (? x))))
(Mortal Socrates)
```

You may think of rewriting the applyRule function so that you can specify a rule's name from a database (e.g., an association list) of rules instead of specifying the whole rule in the function call. For example a call may look like

```
>(applyRule '(I am depressed) 'rule1)
```

where rule1 indexes a rule in a database of rules.

Once again, we have used a LISP-like syntax everywhere.

To write a simple rule engine, given an input your program will cycle through all the rules (at least once) in some order (maybe, the order in which the rules are stored) see if any rule can be applied, and if so apply it. A more sophisticated version of the rule engine may allow all matching rules in some order to apply or even re-cycle through the rule list till no rule applies.

2. **Symbolic differentiation:**

   This problem asks you to complete a symbolic differentiation function. The goal of the function is to take an arithmatic expression containing variables and to produce another representing its first derivative with respect to the variable `x`. Thus given one of the expressions: `(plus (power x 2) (times 3 x))` we want want to produce the new expression: `(plus (times 2 x) 3)`

   The standard rules for differentiation are given below. In this list, $x$ stands for a variable `x`, $c$ for any constant symbol, $u$ and $v$ for any arbitrary arithmatic 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.

   (a) $\frac{d}{dx}[c] = 0$, where $c$ is a numeric constant
   (b) $\frac{d}{dx}[x] = 1$
   (c) $\frac{d}{dx}[v] = 0$, where $v$ is a variable other than $x$
   (d) $\frac{d}{dx}[u + v] = \frac{d}{dx}[u] + \frac{d}{dx}[v]$
   (e) $\frac{d}{dx}[u - v] = \frac{d}{dx}[u] - \frac{d}{dx}[v]$
   (f) $\frac{d}{dx}[-v] = -\frac{d}{dx}[v]$
   (g) $\frac{d}{dx}[u * v] = u * \frac{d}{dx}[v] + v * \frac{d}{dx}[u]$
   (h) $\frac{d}{dx}[u/v] = (v * \frac{d}{dx}[u] - u * \frac{d}{dx}[v])/v^2$

(i) $\frac{d}{dx}[u^c] = c * u^{c-1} * \frac{d}{dx}[u]$ , where $c$ is constant.[1]
(j) $\frac{d}{dx}[sqrt(u)] = (1/2) * \frac{d}{dx}[u]/sqrt(u)$
(k) $\frac{d}{dx}[log(u)] = (\frac{d}{dx}[u])/u$
(l) $\frac{d}{dx}[exp(u)] = exp(u) * \frac{d}{dx}[u]$
(m) $\frac{d}{dx}[sin(u)] = cos(u) * \frac{d}{dx}[u]$
(n) $\frac{d}{dx}[cos(u)] = -sin(u) * \frac{d}{dx}[u]$
(o) $\frac{d}{dx}[tan(u)] = (1 + (tan(u))^2) * \frac{d}{dx}[u]$

Write a function `ddx` that takes a single arithmatic expression in LISP or pre-order format and returns its first derivative with respect to the variable $x$. We need to store the rules of differentiation somewhere. One possibility is to store all of them in a file. This will allow you to add, remove or change differentiation rules without having to change the main part of the code. (Please note that I do not want your program to contain a a big `if` statement checking for the operator (e.g., $+, -, *$, etc.) and taking appropriate actions. )

If the expression is $x$ itself, then 1 is returned. Any other atomic symbol is taken as a constant and a 0 is returned. If the expression has an atomic first element which happens to be a mathematical operator or function, then the proper formula has to be found among the rules and applied to yield the derivative.

An example run in LISP syntax may look like

```
> (ddx '(plus (power x 2) (times x 3)) 'x)
   (plus (times 2 (power x (difference 2 1)))
         (plus (times x 0) (times 3 1))))
```

Note that in LISP, the single quote is used to create an expression that is roughly like a string, something that is what it looks like and cannot be evaluated.

**Extra Credit:** Note that `ddx` in many cases (including the example run above) returns derivative expressions which, although correct, are in a form that we may have difficulty accepting. For example, it is possible to simplify the above to something like `(times 2 x)`. How can we go about doing such simplifications?

You can take the following approach. We will store simplifying patterns such as the following.

| pattern | $x+0$ | $0+x$ | $x*0$ | $0*x$ | $x*1$ | $1*x$ | $x/1$ | $0/x$ |
|---|---|---|---|---|---|---|---|---|
| result | $x$ | $x$ | 0 | 0 | $x$ | $x$ | $x$ | 0 |

In the returned expression, you look for any of the above and other additional patterns that you will write, and if you find any replace it by the corresponding result. And, keep on doing this till no more simplifications can be done. Note that $x$ above need not be a symbol; it can be any complicated expression.

[1] We will only consider the case where c is constant.