# CSE 341 — Assignment 7
# December 7, 1998 — Solution

1. Write and test a Miranda function to do symbolic differentiation of arbitrary expressions involving variables, constants, and the operators + - and *. Your function should take two strings as input – the expression and a variable – and return a new string that is an expression representing the derivative of the expression with respect to the variable. I suggest using Scheme syntax for the input and output expressions. (You can use C or Java Syntax if you prefer but it will be much harder to parse the input.)

   Your result should be at least minimally simplified, using the following rules:

```
0 + expr --> expr
expr + 0 --> expr
num + num --> their sum

0 * expr --> 0
expr * 0 --> 0
1 * expr --> expr
expr * 1 --> expr
num * num --> their product
```

   Examples:

```
deriv "(+ x 3)"  "x"   =>  "1"
deriv "(* x y)"  "x"   =>  "y"
deriv "3"  "x"         =>  "0"
deriv "(+ (* x 3) (* x 2))"  "x"  =>  "5"
deriv "(* (* x y) (+ x 3))"  "x"  =>  "(+ (* x y) (* y (+ x 3)))"
```

   **Solution 1:** (following the hints)

```
|| Miranda Project - Symbolic Differentiation
||          (Solution as suggested by hints)

|| type definitions

|| define string as a synomym for list of character
string == [char]

arith ::= Add | Sub | Mult

token ::= VarToken [char] | ConstToken num | ArithToken arith
                          | RightParen | LeftParen
```

```
expr ::= Binop arith expr expr | Const num | Var [char]

to_digit :: char -> num
to_digit ch = (code ch) - (code '0'), digit ch

tokens :: string -> [token]
tokens [] = []
tokens (' ':rest) = tokens rest
tokens ('*':rest) = ArithToken Mult : tokens rest
tokens ('-':rest) = ArithToken Sub: tokens rest
tokens ('+':rest) = ArithToken Add: tokens rest
tokens ('(':rest) = LeftParen : tokens rest
tokens (')':rest) = RightParen : tokens rest
tokens (ch:rest)  = tokens_var [ch] rest, letter ch
                  = tokens_const (to_digit ch) rest, digit ch


tokens_var var []
        = [(VarToken var)]
tokens_var var (ch:rest)
        = tokens_var (var ++ [ch]) rest, if letter ch
        = (VarToken var) : (tokens (ch:rest)), otherwise

tokens_const c []
        = [(ConstToken c)]
tokens_const c (ch:rest)
        = tokens_const ((c * 10) + (to_digit ch)) rest, if digit ch
        = (ConstToken c) : (tokens (ch:rest)), otherwise


parse :: [token] -> (expr,[token])

parse (VarToken str : ts) = (Var str , ts)

parse (ConstToken str : ts) = (Const str , ts)

parse (LeftParen : ArithToken arith : ts)
        = (Binop arith arg1 arg2 , rest)
          where (arg1,t1) = parse ts
                (arg2, RightParen : rest) = parse t1

parse ts = error ("syntax error -- tokens = " ++ show ts)

binop_to_string Add  = "+"
```

```
binop_to_string Mult = "*"
binop_to_string Sub  = "-"

const_to_string c = shownum c
var_to_string var = var

expr_to_string :: expr -> string
expr_to_string (Binop a e1 e2) = "(" ++ (binop_to_string a) ++
                                 " " ++ (expr_to_string e1) ++
                                 " " ++ (expr_to_string e2) ++ ")"
expr_to_string (Const c)       = (const_to_string c)
expr_to_string (Var var)       = (var_to_string var)

simplify :: expr -> expr
simplify (Binop a e1 e2)
        = simplify_help (Binop a (simplify e1) (simplify e2))
          where
          simplify_help (Binop a (Const c1) (Const c2))
                = Const (do_binop a c1 c2)
                  where
                  do_binop Add c1 c2 = c1+c2
                  do_binop Sub c1 c2 = c1-c2
                  do_binop Mult c1 c2 = c1*c2
          simplify_help (Binop Add (Const 0) e)
                = e
          simplify_help (Binop Add e (Const 0))
                = e
          simplify_help (Binop Mult (Const 1) e)
                = e
          simplify_help (Binop Mult e (Const 1))
                = e
          simplify_help (Binop Mult (Const 0) e)
                = Const 0
          simplify_help (Binop Mult e (Const 0))
                = Const 0
          simplify_help default
                = default
simplify default
        = default

expr_deriv (Binop a e1 e2) v
        = (Binop a (expr_deriv e1 v) (expr_deriv e2 v)),
                              a = Add \/ a = Sub
        = (Binop Add
```

```
                (Binop Mult e1 (expr_deriv e2 v))
                (Binop Mult (expr_deriv e1 v) e2)
          ), otherwise
expr_deriv (Var var) v
        = Const 0, var ~= v
        = Const 1, otherwise
expr_deriv (Const c) v
        = Const 0

deriv :: string -> string -> string
deriv str v = deriv_checker result remnant
              where
      (expr,remnant) = parse (tokens str)
              result = expr_to_string
                           (simplify (expr_deriv expr v))

|| make sure there isn't anything left over in parse
deriv_checker :: string -> [token] -> string
deriv_checker result remnant
        = result, if remnant = []
        = error ("bad parse -- remaining tokens: "
                ++ show remnant), otherwise


showderiv str v = "derivative of: " ++ str ++ " wrt: "
                    ++ v ++ " = " ++ deriv str v

t1 = showderiv "(+ x 3)" "x"
t2 = showderiv "x" "x"
t3 = showderiv "3" "x"
t4 = showderiv "(* x y)"  "x"
t5 = showderiv "(+ (* x 3) (* x 2))"  "x"
t6 = showderiv "(* (* x y) (+ x 3))"  "x"

t7 = showderiv "(- x 3)" "x"
t8 = showderiv "(- x y)" "y"

||
|| the end
||
```

**Solution 2:** (shift/reduce parser)

```
|| Miranda Project - Symbolic Differentiation
```

```
||                   (shift/reduce parser)

|| type definitions

arith ::= Add | Sub | Mult

token ::= VarToken [char] | ConstToken num | ArithToken arith
                          | RightParen | LeftParen

expr ::= Binop arith expr expr | Const num | Var [char]

tokenexpr ::= Token token | Expr expr

|| convert a digit character to a number.
to_digit ch = (code ch) - (code '0'), digit ch

|| convert a string to a list of tokens.
tokens [] = []
tokens (' ':rest) = tokens rest
tokens ('*':rest) = ArithToken Mult : tokens rest
tokens ('-':rest) = ArithToken Sub: tokens rest
tokens ('+':rest) = ArithToken Add: tokens rest
tokens ('(':rest) = LeftParen : tokens rest
tokens (')':rest) = RightParen : tokens rest
tokens (ch:rest)  = tokens_var [ch] rest, letter ch
                  = tokens_const (to_digit ch) rest, digit ch

|| handle reading in a VarToken
tokens_var var []
        = [(VarToken var)]
tokens_var var (ch:rest)
        = tokens_var (var ++ [ch]) rest, letter ch
        = (VarToken var) : (tokens (ch:rest)), otherwise

|| Handle reading in a ConstToken
tokens_const c []
        = [(ConstToken c)]
tokens_const c (ch:rest)
        = tokens_const ((c * 10) + (to_digit ch)) rest, digit ch
        = (ConstToken c) : (tokens (ch:rest)), otherwise


|| This parses and simplifies a stream of tokens.
parse toks = map (simplify.expr_only) (parse_help [] toks)
```

```
            where expr_only (Expr e)
                      = e
                  expr_only (Token t)
                      = error "Parse finished with token on stack"

|| This helper function does shift/reduce parsing
|| When it runs out of tokens, it returns its stack.
|| The a correct parse would yield a stack with only
|| a single (Expr exp) on it.  This 'exp' is the AST
|| we are looking for.
parse_help exps []
        = exps
parse_help exps (tok:toks)
        = parse_help (reduce ((Token tok):exps)) toks
          where
          reduce ((Token RightParen):(Expr e):(Token LeftParen):rest)
                = reduce ((Expr e) : rest)
          reduce ((Expr e2):(Expr e1):(Token (ArithToken op)):rest)
                = reduce ((Expr (Binop op e1 e2)) : rest)
          reduce ((Token (ConstToken c)):rest)
                = reduce ((Expr (Const c)) : rest)
          reduce ((Token (VarToken var)):rest)
                = reduce ((Expr (Var var)) : rest)
          reduce default
                = default

|| compute the AST from a string.
ast = parse.tokens


|| for converting an expression to a string.

|| convert a binop to a string.
binop_to_string Add  = "+"
binop_to_string Mult = "*"
binop_to_string Sub  = "-"

|| convert const expression.
const_to_string c = shownum c

|| convert variable expression.
var_to_string var = var

|| convert any compound expression to a string.
```

```
expr_to_string (Binop a e1 e2) = "(" ++ (binop_to_string a) ++
                                 " " ++ (expr_to_string e1) ++
                                 " " ++ (expr_to_string e2) ++ ")"
expr_to_string (Const c)       = (const_to_string c)
expr_to_string (Var var)       = (var_to_string var)

|| simplify an expression.
simplify (Binop a e1 e2)
        = simplify_help (Binop a (simplify e1) (simplify e2))
          where
          simplify_help (Binop a (Const c1) (Const c2))
                = Const (do_binop a c1 c2)
                  where
                  do_binop Add c1 c2 = c1+c2
                  do_binop Sub c1 c2 = c1-c2
                  do_binop Mult c1 c2 = c1*c2
          simplify_help (Binop Add (Const 0) e)
                = e
          simplify_help (Binop Add e (Const 0))
                = e
          simplify_help (Binop Mult (Const 1) e)
                = e
          simplify_help (Binop Mult e (Const 1))
                = e
          simplify_help (Binop Mult (Const 0) e)
                = Const 0
          simplify_help (Binop Mult e (Const 0))
                = Const 0
          simplify_help (Binop Sub e (Const 0))
                = e
          simplify_help default
                = default
simplify default
        = default

|| take the derivative of an expression.
expr_deriv (Binop a e1 e2) v
        = (Binop a (expr_deriv e1 v) (expr_deriv e2 v)),
                                   a = Add \/ a = Sub
        = (Binop Add
                (Binop Mult e1 (expr_deriv e2 v))
                (Binop Mult (expr_deriv e1 v) e2)
          ), otherwise
expr_deriv (Var var) v
```

```
        = Const 0, var ~= v
        = Const 1, otherwise
expr_deriv (Const c) v
        = Const 0

|| make sure there is only one element of a list
|| and return that element.
first [a] = a

deriv str v = (expr_to_string.simplify)
                 (expr_deriv ((first.parse.tokens) str) v)

||
|| the end
||
```

**Output:**

```
Miranda deriv "(+ x 3)"  "x"
1
Miranda deriv "(* x y)"  "x"
y
Miranda deriv "3"  "x"
0
Miranda deriv "(+ (* x 3) (* x 2))"  "x"
5
Miranda deriv "(* (* x y) (+ x 3))"  "x"
(+ (* x y) (* y (+ x 3)))
Miranda deriv "(+ 10 hello)" "x"
0
Miranda deriv "(+ 10 hello)" "hello"
1
Miranda deriv "(* catch 22)" "catch"
22
Miranda deriv "(* (+ 5 1) x)" "x"
6
```