

Homework 8: Derivation of arithmetic expressions

Stephen Wagstaff
CS 431
November 1, 2018

You will use the following algebraic data types defined for arithmetic expressions.

```
SML  datatype exp = Const of int                | Var of string                | Plus of exp * exp
| Times of exp * exp                | Pow of exp * int;
```

For example, the variable `e` as defined `val e = Times (Times (Var "x", Var "y"), Plus (Var "x", Const 3))`; represents the expression $(x \times y) \times (x + 3)$. The variable `e1` as defined `val e1 = Pow (Var "x", 4)`; represents the expression x^4 . The following are some rules for derivations.

```
dc/dx = 0 -> where c is a constant
dx/dx = 1
dy/dx = 0 -> where y != x
d(u+v)/dx = du/dx + dv/dx
d(u*v)/dx = (du/dx) * v + u * (dv/dx)
d(un)/dx = n * un-1 * (du/dx)
```

-
1. Implement a function `eval : exp -> (string * int) list -> int` to evaluate an arithmetic expression with a context for the variables in the expression. A context is a list of string and integer tuples. For example `eval e [("x", 2), ("y", 3)]` evaluates to 30 because $(x \times y) \times (x + 3)$ is $(2 * 3) \times (2 + 3) = 6 * 5 = 30$. Also, `eval e1 [("x", 2)]` evaluates to 16 because x^4 is $2^4 = 16$. For this `eval` function, you also need helper function `lookup` to look up the value of a variable in a context and `pow` function to calculate the power expression. For example `pow(2,4)` should return 16. The variable look-up is allowed to fail.

```
fun eval (Var variable) context =
  let
    fun lookup nil _ = raise Fail "Variable lookup failed: Empty lookup context passed."
      | lookup [(key, value)] searchKey =
          if key = searchKey
          then value
          else raise Fail "Variable lookup failed: Search key not found in passed context."
      | lookup ((key, value)::rest) searchKey =
          if key = searchKey
          then value
          else lookup rest searchKey
  in
    lookup context variable
  end
| eval (Const(constant)) context = constant
| eval (Plus(exp1, exp2)) context =
  (eval exp1 context) + (eval exp2 context)
| eval (Times(exp1, exp2)) context =
  (eval exp1 context) * (eval exp2 context)
| eval (Pow(exp1, int)) context =
  let
    fun pow(int1, 0) = 1
      | pow(int1, int2) =
          if int2 > 0
          then int1 * pow(int1, (int2-1))
          else raise Fail "Unsupported operation, can not raise to negative value."
  in
    pow((eval exp1 context), int)
  end;
```

2. Implement a function `print: exp -> string` to convert an arithmetic expression to its string representation. For example, `print e` should return the string `"((x * y) * (x + 3))"` and `print e1` should return the string `"(x^4)"`.

```
fun print (Const constant) = Int.toString constant
  | print (Var variable) = variable
  | print (Plus (exp1, exp2)) = "(" ^ print exp1 ^ " + " ^ print exp2 ^ ")"
  | print (Times (exp1, exp2)) = "(" ^ print exp1 ^ " * " ^ print exp2 ^ ")"
  | print (Pow (exp1, int)) = "(" ^ print exp1 ^ "^" ^ Int.toString int ^ " )";
```

3. Implement a function `deriv: exp -> string -> exp` that takes an arithmetic expression `u` and a string `x` and return the derivative du/dx . Note that the second parameter of the function `deriv` is a variable as string. For example, `print (deriv e "x")` should return `"(((1 * y) + (x * 0)) * (x + 3)) + ((x * y) * (1 + 0))"` while `print (deriv e1 "x")` should return `"((4 * (x^3)) * 1)"`.

```
fun deriv (Const _) _ = Const 0
  | deriv (Var variable) dx = if variable = dx then Const 1 else Const 0
  | deriv (Plus (exp1, exp2)) dx = Plus((deriv exp1 dx), (deriv exp2 dx))
  | deriv (Times (exp1, exp2)) dx = Plus(Times((deriv exp1 dx), exp2), Times(exp1, (deriv exp2 dx)))
  | deriv (Pow (exp1, int)) dx = Times(Times(Const int, Pow(exp1, int-1)), (deriv exp1 dx));
```

4. Implement a function `simplify: exp -> exp` to simplify an arithmetic expression as much as possible. For example, `print (simplify (deriv e "x"))` should return `"((y * (x + 3)) + (x * y))"` while `print (simplify (deriv e1 "x"))` should return `"(4 * (x^3))"`. Also, if `val e2 = Pow (Plus (Var "x", Const 0), 2)`, then `print e2` should return `"((x + 0)^2)"` while `print (simplify e2)` should return `"x^2"`.

Hint: for this question, you may want to define a helper function `simp` to simplify obvious expressions. `simp(e*0) = 0`, `simp(e*1) = e`, `simp(e+0) = e`, etc. The function `simplify` should call `simp` after recursively calls itself on components of plus, times, and pow expressions.

```
fun simplify expression =
let
  fun attemptRootSimplification (Times(Const 1, x)) = x
    | attemptRootSimplification (Times(x, Const 1)) = x
    | attemptRootSimplification (Times(Const 0, _)) = Const 0
    | attemptRootSimplification (Times(_, Const 0)) = Const 0
    | attemptRootSimplification (Plus(Const 0, x)) = x
    | attemptRootSimplification (Plus(x, Const 0)) = x
    | attemptRootSimplification (Pow(x, 1)) = x
    | attemptRootSimplification (Pow(_, 0)) = Const 1
    | attemptRootSimplification exp = exp;

  fun doSimplification (Const x) = Const x
    | doSimplification (Var x) = Var x
    | doSimplification (Times(exp1, exp2)) =
      attemptRootSimplification(Times((doSimplification (attemptRootSimplification exp1)),
      (doSimplification(attemptRootSimplification exp2))))
    | doSimplification (Plus(exp1, exp2)) =
      attemptRootSimplification(Plus((doSimplification (attemptRootSimplification exp1)),
      (doSimplification(attemptRootSimplification exp2))))
    | doSimplification (Pow(exp, int)) =
      attemptRootSimplification(Pow(doSimplification(attemptRootSimplification exp),int));
in
  doSimplification expression
end;
```