Imperial College of Science, Technology and Medicine	University of London
Computer Science (CS) / Software Engineering (SE)	BEng and MEng Examinations Part I
Department of Computing	Integrated Laboratory Course
Laboratory work is a continuously assessed part of the examinations and is a required part of the degree assessment. Laboratory work must be handed in for marking by the due date. Late submissions may not be marked.	

Exercise: 5	Working: Individual
Title: Calculus in Haskell	
Issue date: 3rd Nov 2003	Due date: 10th Nov 2003
System: Linux	Language: Haskell

Aim

- To gain experience in user-defined data types and higher-order functions in Haskell.
- To write a program which performs partial differentiation on arbitrary expressions.
- To introduce the idea of abstract syntax trees, symbolic computation and language interpreters.

The Problem

- Write a Haskell script **Calculus.hs** which provides the following types and operations. **Note:** You should use the **exact** type declarations specified below for the types **Expr**, **Environment** and the function headers, so that we can autotest your submission.
  - A type **Expr** representing expressions in an arbitrary number of variables.
  - A type **Environment** representing lookup tables mapping identifiers to floating point values.
  - A function **eval** which evaluates an expression. The function has a second argument which is an environment containing a list of 2-tuples. Each tuple associates an identifier with a numeric value.
  - A function **differentiate** which applies the rules for *symbolically* partially differentiating expressions in a variable using the rules given below.
  - A function **putExpr** which converts objects of the data type **Expr** into lists of characters which show the expressions they represent.
  - A function **maclaurin** which computes successive approximations to a function $f(x)$ using the Maclaurin series expansion of f.

Expressions

- An *expression* can be either
  - a number,
  - an identifier which stands for a value and which is made up of a string of alphabetic characters,
  - or the sum, difference, product or division of two *expressions*.

  This is a recursive definition: an expression may contain other expressions. Some examples of expressions that conform to this definition are:

1	a number
x	a variable
$y + ((2 \times z) - 5)$	the sum of two expressions
$(x + 2) \times (y - 6)$	the product of two expressions
$(h + 3) \div (w + z)$	the division of an expression by another

  Parentheses are used above to show the order in which the expressions are evaluated.

  In Haskell new types can be defined using **data constructors**. Given below is the definition of the types **Environment** and **Expr** in Haskell using data constructors. As you can see below the definition of this type is recursive in that an **Expr** can contain another **Expr**.

```
type Environment = [(String, Float)]
data Expr = Num Float | Id String
          | Add Expr Expr | Sub Expr Expr | Mul Expr Expr | Div Expr Expr
          deriving (Eq, Ord, Show)
```

  Note that objects of type **Expr** can be thought of as *abstract syntax trees* in the sense that they are data structures (trees) that represent real (concrete) expressions like those above.

Differentiation rules

- The rules for differentiation of simple expressions with respect to a variable are:

$$\frac{dn}{dx} = 0 \qquad \text{where } n \text{ is a number}$$

$$\frac{dx}{dx} = 1$$

$$\frac{dy}{dx} = 0 \qquad \text{where } y \text{ is} \neq x$$

$$\frac{d(E_1+E_2)}{dx} = \frac{dE_1}{dx} + \frac{dE_2}{dx} \qquad \text{where } E_1,\ E_2 \text{ are any expressions}$$

$$\frac{d(E_1-E_2)}{dx} = \frac{dE_1}{dx} - \frac{dE_2}{dx}$$

$$\frac{d(E_1\times E_2)}{dx} = \frac{dE_1}{dx} \times E_2 + E_1 \times \frac{dE_2}{dx}$$

$$\frac{d(E_1\div E_2)}{dx} = \frac{\frac{dE_1}{dx}\times E_2 - \frac{dE_2}{dx}\times E_1}{E_2\times E_2}$$

The Maclaurin Series

- The Maclaurin series expansion of a function f defines f as as infinite sum, thus:

$$f(x) = f(0) + xf^1(0) + \frac{x^2}{2!}f^2(0) + \frac{x^3}{3!}f^3(0) + ...$$

  where $f^n(x)$ is the n^{th} derivative of f, i.e. $\frac{d^n}{dx^n}f(x)$.

  Given f and its higher derivatives we can compute the value of $f(x)$, for a given value of x, to an arbitrary degree of accuracy[1] by truncating the Maclaurin series expansion at an arbitrary point. For example, the *zeroth-order* approximation is given by $f(0)$, the *first-order* approximation by $f(0) + xf^1(0)$ and so on.

Submit by Monday 10th November 2003

What To Do

- Create a Haskell script **Calculus.hs** containing the definitions of the types **Expr** and **Environment** given above. Before going on you should check that these definitions compile OK.

- Write a recursive Haskell function **eval** which given an expression and an environment evaluates the expression and returns a real number. The type of **eval** should be declared as

```
eval :: Expr -> Environment -> Float
```

[1] At least if f is "well-behaved", in a sense defined in analytic function theory. You do not need to worry about this!

We say that **eval** implements an *interpreter* for the expression "language" defined above.

The environment used by the function provides a lookup table and you will need to write an auxiliary function that when given a string and an environment returns the appropriate **Float** value.

To save typing the environment expression explicitly each time you call **eval** you could define a constant function, say **testEnv**, and use it for your own testing purposes.

- Write a recursive Haskell function **differentiate** that applies the rules for symbolic differentiation in a variable to the type **Expr**. The type of **differentiate** should be declared as:

```
differentiate :: Expr -> String -> Expr
-- Symbolically differentiates the given Expr with respect
-- to the given variable
```

Note: We say that **differentiate** performs *symbolic* differentiation since it takes (the representation of) a source expression and symbolically generates (the representation of) its derivative.

- Write a function **putExpr** that converts objects of your data type into lists of characters showing the expressions they represent. For example:

putExpr (Mul (Id "x") (Sub (Num 5) (Id "y")))
"(x*(5.0-y))"

The type of **putExpr** should be declared as

```
putExpr :: Expr -> String
```

To convert a number in an expression to a list of characters you should use the predefined function **show** that converts a **Float** to a String.

When converting an expression to text, insert parentheses as necessary to distinguish between different expressions. For example, in the example above it would have been misleading if the output had been given as "x*5.0-y". (You do not need to follow the exact parentheses convention we have illustrated above; for example "x*(5.0-y)" is also OK. However it is simplest to put parentheses around every expression with sub-expressions, as we have done in the example.)

- The following expressions are given in the mathematical notation that we normally use. You should express them using the data type **Expr**. Then use them to test your functions.

$2 \times x$
$y + z$
$v^5 - u$
$t^2 + (-1 \times x)$

$(a + 1) \times (b - 1)$
$(d^4) \div (d^3)$
$x^2 - 1$

Note you will have to represent v^5 as $v \times v \times v \times v \times v$ and '(-1)' as a number.

- Write a function **maclaurin** which, given a function f, generates successive approximations to $f(x)$, for a given value of x, using the Maclaurin series. You can solve this problem entirely without explicit recursion, through the use of higher-order functions and credit will be given throughout for using them correctly. You should refer to the notes, and to the list of Haskell built-in functions given at the back of the list of unassessed problems which details the higher-order functions required. You should break the problem down into the following sub-problems:

  - Write a function `derivatives :: Expr -> [Expr]`, in terms of your earlier function `differentiate` and the built-in function `iterate`, which will produce an infinite list containing the higher derivatives of a given function,
    i.e. $[\ f(x),\ f^1(x),\ f^2(x), ...,\ ]$. The argument should be an object of type `Expr` representing the right-hand side of the function definition and should contain references to the variable "x" only. (**Note:** you may find this easier to write if you use the built-in `flip` function to enable the arguments to `differentiate` to be supplied in the reverse order; `flip` has the property `flip f x y = f y x`.)
  - Using `map`, write a function `coeffs :: Float -> [ Float ]` which, given a value x, will compute the infinite list $[\ 1,\ x,\ x^2/2!,\ x^3/3!, ...\ ]$. Note that you can define $n!$ in Haskell without using recursion, as follows:

    ```
    fact n = product [ 1..n ]
    ```

    and you can use the operator (`^`) to compute x^n.
  - Write a function `zerovals :: Expr -> [Float]` which, given an expression, uses `derivatives` to compute the list of higher derivatives, and returns the list of values obtained by evaluating the derivatives at $x = 0$. To do this use `map` and your `eval` function associating the value `0.0` for "x". Again, you may wish to use the `flip` function to reverse the arguments to `eval`.
  - Write a function `series :: Expr -> Float -> [Float]` which uses `zerovals` and `coeffs`, and the higher-order function `zipWith`, to compute the list of Maclaurin series terms
    $$[\ f(0),\ xf^1(0),\ \frac{x^2}{2!}f^2(0),\ ...\ ]$$
    for a given function f (an object of type `Expr`) and value x (a `Float`).
  - Write a function
    `maclaurin :: Expr -> Float -> [Float]`
    using `series` and the built-in higher-order function `scanl` (or `scanl1`) to build the list of successive approximations to $f(x)$ for a given f and x, i.e. $[\ a_0,\ a_1,\ a_2,\ ...\ ]$ where a_k is the k^{th}-order approximation to $f(x)$ given by:
    $$a_k = \sum_{i=0}^{k} \frac{x^i}{i!} f^i(0)$$

Note that the i^{th} term is obtained by summing the first i terms in the list generated by `series`.

Unassessed

- Try to write a function **simplify** which simplifies expressions, *e.g.* it could simplify **Mul (Num 0) (Id "x")** to **Num 0** etc.
- Define an environment using a function of type `(String -> Float)` rather than a list of tuples. To test this you will need to define a new environment type, say **Environment'** and a new evaluation function, say **eval'**.

  You will need to define two functions, say **empty** and **add**, to implement the environment. The (constant) function **empty** should return an empty environment. An attempt to look up an identifier in the empty environment should print an error message.

  ```
  empty :: Environment'
  ```

  The function **add** say, which should add a new (identifier, value) binding into an existing environment:

  ```
  add :: (String,Float) -> Environment' -> Environment'
  ```

  The value returned by add will need to be a *function* of type `Environment'`. This function must represent the original environment augmented with the new binding *(x,v)*. So given an identifier y this new function should return v if $x == y$, otherwise it should return the binding of y in the original environment.

  Any particular environment can then be built as a single lookup table, i.e. using a single nested conditional, or as a sequence of calls to **add** using the empty environment as a base in a way analagous to building a list using `[]` and successive calls to `(:)`.

  To simplify testing you may wish to define a constant function of type environment. For example:

  ```
  testEnv' :: Environment'
  ```

Submission

- Submit your **Calculus.hs** script by typing **submit 5** at your Unix prompt.

Assessment

`eval`	`1.0`
`differentiate`	`1.5`
`putExpr`	`0.5`

maclaurin	2.0
Design, Style and Readability	5.0
Total	10.0