Abstract syntax trees, symbolic derivation, and numerical equation solving

Håkan Jonsson

April 10, 2017

1 Introduction

This is an exercise to prepare for Lab Assignment 2. It is *optional* and will *not be graded* but, if you do the exercise and like, you could get comments on your solution.

1.1 Overview

In this exercise your task is first to add code to the type `EXPR` and the functions `parse`, `unparse`, `eval`, `diff`, and `simplify` so they work as described below. Skeleton code of all this is available for download from within Canvas.

As a second task, based on the type and the functions, you should write a "function generator" and use it in a third task to solve equations of the kind $f(x) = 0$.

2 Parsing

Download the skeleton file and load it into your Haskell environment. Test `parse` on some arithmetic expressions like `"10"`, `"x"`, `"10+x"`, `"1+2*(3-4/5)"` and then on expressions with functions like `"sqrt(1+sin(x))"`. After this, take a look at the contents of the file:

- `EXPR` is a recursively defined algebraic data type for expressions that can contain integer constants, names of variables, binary operators with two operands of type `EXPR`, and function applications where a function is applied on a value of type `EXPR`. The two last kinds of values are tree nodes.
- `parse` interprets an expression in a string as a value of type `EXPR`. We say that it *parses* the string.
- `unparse` turns an `EXPR` into a string.

- `eval` evaluates the value of an `EXPR`. Note that we can only write integers in our strings but `eval` still returns a floating-point number.
- `diff` differentiates an `EXPR` symbolically with respect to a variable. The derivative is also an `EXPR`.
- `simplify` makes an `EXPR` simpler by applying a number of simplification rules.

You first task is now to add code so the functions also work for *function application*. The type `EXPR` and `parse` are already prepared for this. It is enough if `eval` and `diff` can handle the functions sine (`sin`), cosine (`cos`), natural logarithm (`log`), and exponential function with base e (`exp`). It is OK to write `simplify` so it just simplifies the argument of functions.

When you have added all the code, you should be able to have the Haskell system evaluate

```
unparse (simplify (diff (Var "x") (parse "exp(sin(2*x))")))
```

and get the derivative printed.

3 Creating functions

Our functions can be used to analyze functions of one real variable. We will now use them to solve equations with Newton-Raphson's method. Since this method is based on derivatives, we will define functions with `EXPR` and get the derivatives with `diff`.

Your second task is to write a function

```
mkfun ::  (EXPR, EXPR) -> (Float -> Float)
```

that, given a pair `(body,var)`, returns a function of the variable `var`, defined by the expression `body`, and with type `Float -> Float`. The expression `var` must always be an `EXPR` constructed with `Var` and a string (the name of the variable). For instance, the line

```
mkfun (parse "x*x+2", Var "x")
```

should return the function

```
\x -> x*x + 2.0 :: Float -> Float
```

If we bind what is returned from `mkfun` above to `f`, the expression `f 3.0` should evaluate to `11.0`.

Note that `mkfun (body,var)` is a Haskell function in one variable `var`. For each value of `var`, the function can be evaluated by applying `eval` to `body` in a context where `var` is bound to its value. Study `eval` to understand how this is done.

4 Newton-Raphson

We will now write a function that solves equations of the kind $f(x) = 0$ numerically with Newton-Raphson's method. The input to this function is the name of the unknown variable, the function f, and the start value for the iterations. To get the derivative f' we will use the function `diff` to differentiate f symbolically. Because of this, we need to give f in terms of an `EXPR` instead of a string.

So, **your third task** is to write a function

```
findzero :: String -> String -> Float -> Float
```

that takes first a string `s1` with the name of an unknown variable, then a string `s2` with the body of a function, and finally a start value `x0` that will be used in the first iteration of the Newton-Raphson method. As the name of `findzero` suggests, it computes the zero of the function defined by `s2`, with `s1` as the unknown variable, when `x0` is the first guess. The iterations should go on until the absolute difference is at most 0.0001. Make sure you compute f and f' once only, and not in every iteration (because that would be a total waste of time). Examples:

- `findzero "x" "x*x*x+x-1" 1.0` should evaluate to 0.68232775.
- `findzero "y" "cos(y)*sin(y)" 2.0` should evaluate to 1.5707964.

Also, try with some other examples that you come up with yourself or take from your book on Calculus. Convince yourself that it works and how it works.

This exercise has hopefully showed you the power of higher order functions and how the inner workings of a simple calculator with an expression parser can be implemented.

Acknowledgement

This exercise builds upon a exercise developed by Leif Kusoffsky and Björn von Sydow.