2024 Haskell January Test
Symbolic Integration

THREE HOURS TOTAL

The maximum mark is 30.

- Please make your swipe card visible on your desk.
- Work in the file named `Int.hs` inside the `integration/src` subfolder of your Home folder. **Do not move any files**.
- You are **not permitted** to edit the folder structure, or edit the `hft.cabal`, `cabal.project`, `src/Utilities.hs`, or `src/Types.hs` files without having been directed by the Examiners; any changes will be reverted.
- You may add additional tests, which will be reflected in the final script (*like in the PPTs*), but these will not be assessed. However, any changes made to the code that cause the **original given** tests to not compile will incur a compilation penalty.

It is important that you comment out any code that does not compile before you complete the test. There will be a TWO MARK PENALTY applied to any program that does not compile.

Credit will be awarded throughout for style, clarity and the appropriate use of Haskell's various language features including list comprehensions, higher-order functions etc.

1 Introduction

Symbolic integration is the process of determining the integral of a function algebraically, i.e. by transforming the *representation* of a function to that of its integral. Symbolic integration (and also differentiation) tools are key features of mathematical software packages such as Mathematica[1] and MATLAB[2] and have numerous applications in mathematics, science and engineering. In this exercise we'll focus on *indefinite* integrals, although it is relatively simple to extend this to definite integrals via substitution[3].

Implementing symbolic differentiation is straightforward, but symbolic integration is *much* harder, as there is no equivalent of the chain rule, which says that if $h(x) = f(g(x))$, then

$$h'(x) = f'(g(x))\, g'(x)$$

However, there are a number of 'tricks' that one can play that collectively address most functions of practical significance. Implementing these in full goes far beyond the scope of this exercise, so we will focus on some of the basic rules of integration, together with the *inverse chain rule* (or 'reverse' chain rule), which is explained below. The differentiation and integration rules that are relevant to this exercise are given in Figure 1.

1.1 The inverse chain rule

The inverse chain rule applies to a product of the form $g'(x)\, f(g(x))$, or $f(g(x))\, g'(x)$. The trick is to observe that by setting $u = g(x)$ we get $du/dx = g'(x)$, which means that $g'(x)\, dx = du$. Hence:

$$\int g'(x)\, f(g(x))\, dx = \int f(u)\, du \quad \text{where } u = g(x)$$

So, for example,

$$\int \frac{1}{x}\sqrt{\log(x)}\, dx = \int \frac{1}{x}\left(\log(x)\right)^{\frac{1}{2}}\, dx = \int u^{\frac{1}{2}}\, du = \frac{2}{3}u^{\frac{3}{2}} + c = \frac{2}{3}\left(\log(x)\right)^{\frac{3}{2}}$$

Here, $u = g(x) = \log(x)$, $f(u) = u^{\frac{1}{2}}$ and $g'(x) = \frac{1}{x}$. c is the constant of integration.

2 Representation

In this exercise 'base' expressions, such as 4, x, $-\frac{2}{3}x$, $2x^3 - 2x + 2$ etc. will be polynomials. All polynomials will be expressed in *standard form*, which means that the individual *terms* will be written in decreasing powers of a single variable, x, as in the examples above. For the purposes of the exercise the exponent (power) of each polynomial term will be a non-negative `Integer` and its coefficient a `Rational`.

All expressions in this exercise are assumed to be relative to the variable x, so when representing polynomials we only need to encode the coefficient and exponent of each term. The representation used is a list of (coefficient, exponent) pairs:

```
type Coefficient = Rational
type Exponent = Integer
type Term = (Coefficient, Exponent)
type Polynomial = [Term]
```

[1]https://www.wolfram.com/mathematica/

[2]https://uk.mathworks.com/products/matlab.html

[3]E.g. $\int_b^a x^2\, dx = \left[\int x^2\, dx\right]_b^a = \left[\frac{x^3}{3}\right]_b^a = \frac{a^3-b^3}{3}$.

$$\begin{array}{rclrcl}
\frac{d}{dx}x & = & 1 & & & \\
\frac{d}{dx}a & = & 0 & \int a\, f(x)\, dx & = & a \int f(x)\, dx \\
\frac{d}{dx}f(x) + g(x) & = & f'(x) + g'(x) & \int f(x) + g(x)\, dx & = & \int f(x)\, dx + \int g(x)\, dx \\
\frac{d}{dx}f(x)\, g(x) & = & f(x)\, g'(x) + f'(x)\, g(x) & \int x^n\, dx & = & \frac{x^{n+1}}{n+1} + c, \quad n \neq -1 \\
\frac{d}{dx}x^n & = & n\, x^{n-1} & \int x^{-1} & = & \int \frac{1}{x}\, dx = \log(x) + c \\
\frac{d}{dx}log(x) & = & \frac{1}{x} & \int \log(x)\, dx & = & x\,(\log(x) - 1) + c \quad \text{(see footnote}^4\text{)} \\
\frac{d}{dx}f(g(x)) & = & f'(g(x))\, g'(x) \ \text{(chain rule)} & & &
\end{array}$$

Figure 1: Some differentiation and integration rules. a is an arbitrary constant, c is the constant of integration. In this exercise the exponent n will either be an integer (polynomials) or a rational (expressions).

Using this, the polynomials above will be represented in standard form by `[(4,0)]`, `[(1,1)]`, `[(-2 % 3,1)]`, `[(2,3),(-2,1),(2,0)]` respectively. Recall that a `Rational` can be built using the constructor `:%`, the operator `%`, or the `Fractional` operator `/`, e.g. `-2 :% 3`, `-2 % 3`, `-2 / 3`.

Important: the constant 0 is represented by `[(0,0)]` and `[]` is therefore not a valid polynomial. Throughout the exercise you may assume that a function expecting a polynomial argument will never be applied to `[]`. Furthermore, all constants will be polynomials of the form `[(c, 0)]`; thus, for example, logarithms and powers of constant expressions will not occur.

2.1 Functions

A function $f(x) = e$ will be encoded by the representation of the defining expression e. For simplicity there are just five types of expression: (base) polynomials, sums, products, powers and logarithms. These will be represented by the `Expr` type defined thus:

```
data Expr = P Polynomial
          | Add Expr Expr
          | Mul Expr Expr
          | Pow Expr Rational
          | Log Expr
          deriving (Eq, Ord, Show)
```

There is no subtraction function so, for example, the expression $e-1$, will be represented by `Add e (P [(-1, 0)])`. Similarly, $-e$ will be represented by multiplying `e` by `-1`. Note that for powering, the exponent is a `Rational`, whereas the exponent in a polynomial term is a non-negative integer. The type definitions can be found in the module `Types`.

[4] You can derive this using integration by parts:

$$\int \log(x)\, dx = \int \log(x) \times 1\, dx = \log(x) \int 1\, dx - \int \left(\frac{d}{dx} \log(x) \int 1\, dx \right) = x \log(x) - x + c = x(\log(x) - 1) + c$$

.

A `Utilities` module is provided, which includes the functions `pretty` and `prettyPrint` for pretty-printing objects, respectively in the form of a `String`, and using `IO`. The functions are overloaded and are defined to operate on both polynomials and expressions.

To make clear the distinction between '`+`' in a polynomial and '`+`' in an expression, polynomials are displayed in square brackets with multiplication expressed via juxtaposition, as in `[2x]`, for example. The multiplication of two `Expr`s will be written using '`.`' (i.e. '`.`' does *not* denote composition!). Rationals are rendered by the pretty-printing functions using parentheses, e.g. `(2/3)` for `2 :% 3`. Various polynomials (`p1`, ..., `p5`) and expressions (`e1`, ..., `e16`, with `ei` = `P pi` for `i`=1..5) are defined in the template for testing purposes. The variable `x` is also defined in the template to be the same as `e2`=`P [(1,1)]`). So, for example:

```
*Main> p5
[(2 % 1,3),((-2) % 1,1),(2 % 1,0)]
*Int> pretty p5
"[2x^3 - 2x + 2]"
*Main> prettyPrint p5
[2x^3 - 2x + 2]
*Main> prettyPrint e2
[x]
*Main> prettyPrint x
[x]
*Main> prettyPrint [(2 / 3, 1)]
[(2/3)x]
*Main> prettyPrint e12
[4x - 3] . log[2x^2 - 3x + 1]
```

and so on.

2.2 Normal forms

There is no notion of a "normal form" for a mathematical expression, i.e. a single canonical way of writing it, because in general it is not possible two determine whether two functions, f and g satisfy the property that $\forall x, f(x) = g(x)$. However, humans typically apply a number of simplification rules when performing algebraic manipulation "by hand", in order to keep expressions short and maintain a consistent structure. These rules include:

$$e + 0 = 0 + e = e; \;\; e \times 1 = 1 \times e = e; \;\; e^1 = e; \;\; a\,e + b\,e = (a + b)\,e$$

$$ee^n = e^{n+1}; \;\; e^m\,e^n = e^{m+n}$$

$$\log(a) + \log(b) = \log(a\,b); \;\; \log(e^n) = n\,\log(e)$$

Encoding these rules is straightforward but long-winded and (very!) messy, so a function `simplify` is also provided in the `Utilities` module which simplifies a given expression to something akin to a 'syntactic normal form'. `simplify` encodes the above rules and applies various refactorings of polynomials, including the extraction of common constant factors. It also orders the terms in an expression to reflect the ordering (`Ord`) derived for `Expr` above; for example, the simplified version of $\log(x) + 5x^2$ will place the polynomial $5x^2$ before the $\log(x)$, viz. `[5x^2] + log[x]`. Here are some more examples:

```
*Main> prettyPrint e5
[2x^3 - 2x + 2]
```

```
*Main> prettyPrint (simplify e5)
[2] . [x^3 - x + 1]
*Main> prettyPrint (simplify (Pow e5 2))
[4] . [x^6 - 2x^4 + 2x^3 + x^2 - 2x + 1]
*Main> prettyPrint (Add (Pow e5 (1/2)) e2)
([2x^3 - 2x + 2])^(1/2) + [x]
*Main> prettyPrint (simplify (Add (Pow e5 (1/2)) e2))
[x] + ([2] . [x^3 + x + 1])^(1/2)
```

3 What to do

There are three parts to the exercise. Part III is considerably harder than Parts I and II, as there is much less hand-holding, so you might wish to complete Parts I and II before attempting it.

When performing integration the output should always include a constant of integration. In this exercise the constant of integration will be ignored, on the understanding that a complete implementation will mostly likely add it at the end. For example, if a symbolic integration rule yields $x^2 + 3$ then we should strictly write it as $x^2 + c$, as the '3' will be absorbed into the constant of integration. However, for the purposes of the exercise we will leave the constant intact: when you see one or more constants in the integration result, consider them collectively to be equivalent to 'c'.

About the tests: In many cases, two expressions that denote the same function will be identical after applying the `simplify` function. However, *if you happen to construct a result that is evidently correct but which fails a test because of the limitations of* `simplify`, or otherwise, *don't worry; just add a short comment in your code in order to alert the marker.*

3.1 Part I

1. Define functions `addP :: Polynomial -> Polynomial -> Polynomial` and `mulP :: Polynomial -> Polynomial -> Polynomial` that will respectively add and multiply two polynomials in standard form, delivering the result in standard form. For example:

   ```
   *Main> prettyPrint p2
   [x]
   *Main> prettyPrint (addP p2 p2)
   [2x]
   *Main> prettyPrint (addP p3 p4)
   [2x^2 + x - 2]
   *Main> prettyPrint (addP p5 [(0,0)])
   [2x^3 - 2x + 2]
   *Main> prettyPrint (mulP p3 p4)
   [8x^3 - 18x^2 + 13x - 3]
   ```

   Note that each term in `[2x^3 - 2x + 2]` above has a common factor, 2. It is not necessary to factor this out here; where necessary any such factorisation will be done via `simplify` later on.

   [8 Marks]

2. Define functions `sumP :: [Polynomial] -> Polynomial` and `prodP :: [Polynomial] -> Polynomial` that will respectively compute the sum and product of a list of polynomials. For example,

```
*Main> prettyPrint (sumP [[(0,0)]])
[0]
*Main> prettyPrint (sumP [p1,p2,p3,p4,p5])
[2x^3 + 2x^2 + 5]
*Main> prettyPrint (prodP [p3,p5])
[4x^5 - 6x^4 - 2x^3 + 10x^2 - 8x + 2]
```

[1 Mark]

3. Define functions `diffT :: Term -> Term` and `intT :: Term -> Term` that will respectively deliver the derivative and integral of a polynomial term, with respect to `x`. For example,

```
*Main> diffT (1,0) -- 1
(0 % 1,0)
*Main> diffT (2,3) -- 2x^3
(6 % 1,2)
*Main> intT (1,0)
(1 % 1,1)
*Main> intT (2,3)
(1 % 2,4)
```

   Note that the constant of integration is omitted, as explained above.

[3 Marks]

4. Using `diffT` and `intT`, or otherwise, define functions `diffP :: Polynomial -> Polynomial` and `intP :: Polynomial -> Polynomial` that will respectively deliver the differential and integral of a given polynomial, with respect to `x`. For example,

```
*Main> prettyPrint (diffP p3)
[4x - 3]
*Main> prettyPrint (intP p3)
[(2/3)x^3 + (-3/2)x^2 + x]
```

[1 Mark]

3.2 Part II

This question relates to expressions (`Expr`). A function `toExpr` is defined in the template that will convert a `Rational` to an `Expr` by wrapping it up as a polynomial:

```
toExpr :: Rational -> Expr
toExpr n = P [(n, 0)]
```

A function `isConstant` is also defined that returns `True` iff the argument is a constant (recall that all constants in the exercise are assumed to be polynomials):

```
isConstant (P [(_, 0)]) = True
isConstant _ = False
```

Feel free to use these as you see fit.

Using `diffP`, define a function `diffE :: Expr -> Expr` that will differentiate a given expression with respect to the variable `x`. To help with testing, a function `simplifiedDiff :: Expr -> Expr` is defined in the template that simplifies the result generated by `diffE`:

```
simplifiedDiff :: Expr -> Expr
simplifiedDiff = simplify . diffE
```

During testing you may find it useful to inspect the result of both `diffE` and `simplifiedDiff`, but recall that the `simplify` function is not guaranteed to produce a unique normal form. Also, your version of `diffE` may produce output that is syntactically different to that shown in the examples, e.g. including terms like `+ 0`, `. [1]` etc.; however it should be mathematically identical to that shown. The tests are based on the results *after* simplification: any output that is mathematically correct after simplification will be sufficient for the purposes of awarding credit (see the above note "About the tests").

If you want to pretty-print a simplified derivative, a function `printDiff = prettyPrint . simplifiedDiff` is also provided for this purpose. Hence, for example:

```
*Main> prettyPrint e3
[2x^2 - 3x + 1]
*Main> prettyPrint (diffE e3)  -- Your output may look slightly different
[4x - 3 + 0]
*Main> prettyPrint (simplifiedDiff e3)
[4x - 3]
*Main> printDiff e3   -- Shorthand for the above
[4x - 3]
*Main> prettyPrint e10
[4x - 3] . [2x^2 - 3x + 1]
*Main> prettyPrint (diffE e10)
[4x - 3] . [4x - 3 + 0] + [4 + 0] . [2x^2 - 3x + 1]
*Main> printDiff e10
[24x^2 - 36x + 13]
*Main> prettyPrint e11
[x]^-1 . log[x]
*Main> prettyPrint (diffE e11)
[x]^-1 . [1] . [x]^-1 + [1] . [-1] . [x]^-2 . log[x]
*Main> printDiff e11
[-1] . [x]^-2 . log[x] + [x]^-2
```

[7 Marks]

3.3 Part III

Because we're only implementing a small set of rules for symbolic integration it may be the case that an attempt to integrate a given expression will fail. To allow for this, the integration function for `Expr`essions is defined to return a `Maybe Expr` rather than an `Expr`. If the integration succeeds with result `i` the result will be `Just i`; if it fails it will return `Nothing`.

Your job is to define the function `intE :: Expr -> Maybe Expr`. The function should implement the basic rules of integration given in Figure 1, together with the inverse chain rule, described in Section 1.1. You can proceed in any way you see fit, but the suggestion is that you break the problem down by starting with the first three rules for `intE` which will go something like this:

1. If the argument is a polynomial, use `intP`.
2. If the argument is the sum of two expressions, use the integration rule for addition (Figure 1).

3. If the argument is the product of a constant (`isConstant`) and an arbitrary expression (or vice versa), use the corresponding integration rule (Figure 1).

At this point you'll need an additional, temporary, 'catch-all' rule which delivers `Nothing` in all other cases; you can replace this later on.

To help with testing, a function `simplifiedInt :: Expr -> Maybe Expr` applies `simplify` to the result of `intE`, if there is one (`Just ...`) and returns `Nothing` otherwise. Mirroring `printDiff` above, a function `printInt :: Expr -> IO ()` shows the integral of the given function, if one is found (`Just ...`), and "Fail" otherwise. The function `fromJust` from `Data.Maybe` might also be useful for testing. For example,

```
*Main> intE e3
Just (P [(2 % 3,3),((-3) % 2,2),(1 % 1,1)])
*Main> simplifiedInt e3
Just (Mul (P [(1 % 6,0)]) (P [(4 % 1,3),((-9) % 1,2),(6 % 1,1)]))
*Main> printInt e3
[(1/6)] . [4x^3 - 9x^2 + 6x]
*Main> prettyPrint e6
[2x^2 - 3x + 1] + [2x^3 - 2x + 2]
*Main> prettyPrint (fromJust (intE e6))
[(2/3)x^3 + (-3/2)x^2 + x] + [(1/2)x^4 - x^2 + 2x]
*Main> prettyPrint (fromJust (simplifiedInt e6))
[(1/6)] . [3x^4 + 4x^3 - 15x^2 + 18x]
*Main> printInt e6
[(1/6)] . [3x^4 + 4x^3 - 15x^2 + 18x]
*Main> printInt e16
Fail
```

3.3.1 Completing the exercise

Recall that the inverse chain rule states that:

$$\int g'(x)\, f(g(x))\, dx = \int f(g(x))\, g'(x)\, dx = \int f(u)\, du$$

where $u = g(x)$. In the first instance the function f refers here to powering (`Pow`) and logarithm (`Log`). However, it can also refer to the *identity* function $id(x) = x$. In this case,

$$\int g'(x)\, g(x)\, dx = \int g'(x)\, id(g(x))\, dx = \int u\, du = \frac{u^2}{2}, \quad \text{where } u = g(x)$$

In other words, the product of a function and its derivative is also an instance of the rule.

What about simple functions of x, like $\sqrt{x} = x^{\frac{1}{2}}$ or $\log(x)$? You could code these separately, but they can also be seen as instances of the inverse chain rule because the expression e is the same as $1 \times e$; thus, when $g(x) = x$ we have $g'(x) = 1$ so that $g'(x) \times f(g(x)) = 1 \times f(x)$. Note that this 'trick' introduces an additional multiplication that wasn't there before.

You're now in a position to complete the definition of `intE` by replacing the 'catch-all' case above as follows:

4. If the argument is the product of two expressions, try the inverse chain rule, including the case for the identity function (see above).

5. In all other cases, try the 'multiplication by 1' trick above.

If you're following this approach you'll likely end up needing a helper function `applyICR :: Expr -> Expr -> Maybe Expr` to implement the inverse chain rule, where the two expressions are $g'(x)$ and $f(g(x))$, or vice-versa – you choose! The type signature is provided, should you wish to use it. For example (for brevity these examples only show the results of applying `printInt`):

```
*Main> printInt e8
[x] . ([-1] + log[x])
*Main> printInt e9
log[x]
*Main> printInt e10
[(1/2)] . [4x^4 - 12x^3 + 13x^2 - 6x + 1]
*Main> printInt e11
[(1/2)] . (log[x])^2
*Main> printInt e12
[2x^2 - 3x + 1] . ([-1] + log[2x^2 - 3x + 1])
*Main> printInt e13
[(2/5)] . ([2x^2 - 3x + 1])^(5/2)
*Main> printInt e14
[(2/25)] . ([5] . [2x^2 - 3x + 1])^(5/2)
*Main> printInt e15
[(1/3)] . [x^3 - 3x] . ([-1] + log[x^3 - 3x])
*Main> printInt e16
Fail
```

Note that `e16` cannot be integrated using the rules covered here (but maybe you can do better?!)

[10 Marks]

Hints

When implementing the inverse chain rule on the product $e1 \times f(e2)$, say, you might like to start by using `==` to determine whether the result of differentiating $e2$ is identical to $e1$. You may also need to try the terms the other way round, viz. $f(e2) \times e1$[5]. This will get you a long way, but there's a twist!

If you encounter the 'twist' then the following may help: if one polynomial is a constant multiple of another then their simplified forms will differ only in their respective factors, e.g.

```
*Main> prettyPrint (simplify (Mul e1 e5))
[10] . [x^3 + x + 1]
*Main> prettyPrint (simplify (Mul e5 e1))
[10] . [x^3 + x + 1]
```

Good luck!

[5] Even if you simplify each expression before attempting to integrate it, it may not be the case that the differential of $e2$ is 'smaller than' $e1$ in the `Expr` ordering.