Programming Paradigms Assignment 1

Author: Martin Raunkjær Anderse
AAU study number: 20136030
AAU email adress: marand13@student.aau.dk

The status of the program (relative to the assignment)

1. Complete: Define suitable datatype for univariate expressions
2. Complete: Write functions that support the manipulation of univariate expressions
3. Incomplete: subject to all identities. Some of the ones that are not relevant for reduction has been left out, such as (6) and (9)
4. Incomplete: Pretty print reduction sequences in Latex. The program prettyprints the output in my language, not Latex.
5. Complete: The user can interactively specify expressions in the command line, which are then reduced

Descriptions of possible external program parts and/or inspirations

I used the happy parser and alex Tokenizer to parse strings to my types. I was heavily inspired by a github repo[1], where a basic arithmetic parser lives.

An enumeration of possible non-completed tasks

PolScale Const can only be on left side of an expression, probably because of grammar ambiguity Const is Integer, not Rational.

Unfortunately I did not have the time to implement all the reductions well enough. the Big one (number 13) cannot be reduced at this time. Here are some testcases that can be:

- $x\hat{2} * x\hat{3}$
- $x\hat{2} / x\hat{3}$
- $\sin(x)\hat{2} + \cos(x)\hat{2}$

[1] ghulette/haskell-parser-examples

- e * e
- 1 * e1
- dx(x$\hat{4}$ + dx(cos (x)))

in the Program folder, there is a test.sh file, that runs the tests for each identity

Other remarks of relevance (optional)

I have foregone a LOT of equality checking of variables, because the expressions are supposed to be univariate, so whenever a variable is met in an expression is assumed that it is the same variable. But no actual logic prohibits the inclusion of more variables.

I have added a constant for the Polynomial Type, so reduction cases like (sin x) $\hat{2}$ + (cos x) $\hat{2}$ = 1 can be done without jumping through too many hoops. The parser does not map any grammar rule to it, so the user cannot reach the constant on their own (ie. the user cannot input "5" to the program"). It could also be done by using the PolScale and PolPow types like so: 1 * x ^ 0, but that doesn't look as nice.

The add polynomial rule was foregone because it was ambiguous with the add expression rule. (expressions can also be polynomials)