Haskell Interpreter - OneLiner

Introduction

OneLiner is an interpreter written in Haskell for working on **streams** - potentially unbounded sequence of data. This has had an impact on several design choices we have made during development. The interpreter was developed using:

- **Haskell**
- **Alex** - lexer, which reads the input and breaks it into tokens
- **Happy** - parser, which performs a syntax analysis to check whether the input conforms the grammar rules

Features of the interpreter:

- Functional programming style
    - performs actions on input as the evaluation of mathematical functions
    - avoids mutable data
- Lazy Evaluation of functions
    - function evaluation is delayed until the time it is needed
- Strong/Dynamic typing
    - does not allow access to private data and/or memory in order to prevent the program from crashing the machine.
    - the type checking is done at run time.

Syntax

OneLiner was inspired by the hard drive sectors alignment and the fact that the input would be streams of data **in columns**. It operates on the input row by row. A sample program written in OneLiner looks as follows:

Input	Program	Output
-5	[| $1+2 | $1*3]	-3 -15
0		2 0
3		5 9

Programs are contained in `[...|...]` and consist of blocks. The very first block, which in the example above is empty, is reserved (remember the inspiration from Hard Drive sectors) for the accumulators (if any). Blocks are separated using a pipe `|` and the number of blocks, excluding the first one correspond to the number of columns in the output (as in our case - 2). Recall that **OneLiner** operates on streams, which consist of columns of data, with `$` one can access the column number (e.g. `$1` - first column from the input).

As a Functional Language, inside the block one writes the functions to be applied to the corresponding column in the output. For the sake of simplicity, the first column of the output of the program above is the **first column of the input + 2**, the second column of the output - **first column of the input * 3**. Therefore, the functions we apply to each row of the input are **addition** & **multiplication**, accordingly. The language supports **addition, multiplication, division, power**, and **modulo**. Where is subtraction? The language supports **negation** but not **subtraction**. The reason for that is siplicity of the grammar and to encourage creativity. One can perform subtraction using addition and negation.

The grammar consists of:

- `App` which is all the programs in an application.
- `Prefix/Suffix` can be used before and/or after an `App`.
- `Prog` is the core of the language. This is where all the computational logic happens.