Project 1

Due: Friday, March 14th 11:59pm

Description

In this project you will create a prefix-notation expression calculator. The calculator will prompt the user for an expression in prefix notation and calculate the result of the inputted expression. A history will be kept so that previous results can be used in expressions. You are expected to handle possible errors in the user's input. Do not forget to follow the "Git/Project Instructions".

Details

The program can run in two modes: interactive and batch. If the program is ran with the "-b" or "--batch" flag, then it should run in batch mode. Otherwise, it runs in interactive mode. When in batch mode, the only output should be the results (including errors). All other behavior is identical to the interactive mode, which is described below. I have provided code you can copy and paste into your program to detect the mode. It defines an identifier `interactive?` which is true in interactive mode and false in batch mode.

When ran, the program should immediately prompt the user for an expression. It will evaluate that expression, printing an error message if an error is encountered. A simple generic message "Invalid Expression" is alright. An error message should be prefixed with "Error:".

There should be a history represented as a simple list of values. Every time the program succeeds in evaluating the expression, the result will be added to the history and printed to the screen. The printed result should be prefixed by the history id (with no leading zeros) followed by ": ". The number should be converted to a float using the `real->double-flonum` function, and then printed with `display`. No preprocessing such as rounding should be done.

Remember, you are programming in a functional language. The history should be a parameter to your eval loop function. So, adding a value to history should involve cons-ing it with the existing history. When the value is printed to screen, you should include its history id (or index). The id will always be the order it was added to history.

So, the first value added will be id 1, the second id 2, and so on. Using the cons operator to construct the history will result the values being in reverse order of their id. That is, the most recent value will be first in the list. This can be remedied by reversing the list before getting the value.

Expressions

An expression is a value, binary operation, or a unary operation.
`+` – A binary operator that adds together the result of two expressions.
`*` – A binary operator that multiplies together the result of two expressions.

`/` – A binary operator that divides (integer division) the result of the first expression by the result of the second. Don't forget that the user might try to divide by zero. This is an error.
`-` – A unary operator that negates the value of an expression. There is no subtraction. To subtract we can add a negative number.
$n – n should be an integer. A value specifying to use the history value corresponding to id n.
any number – a value.
Expressions are in prefix notation and read from left to right. So, if I wanted to evaluate $2 * \$1 + \$2 + 1$, I would write `+*2$1+$2 1`. Whitespace can be used to divide tokens (like the `$2 1` in the previous example), but otherwise insignificant. The previous example could also be written as `+ *2 \$1 + \$2 1`. It is an error, if the expression is evaluated, but there is still text remaining. For instance, `+1 2 2` is an error. The expression is $1 + 2$. The second `2` is not part of the expression.

Some Tips

- Don't try to do everything in one function.
- Create functions that evaluates a particular type of expression and yields a two element list. The first element would be the result of the evaluation, and the second the remaining characters in the expression.
- Solve the problem in stages. First think about how you would write this program in a language you are familiar with. Maybe write some psudocode. Translate the algorithm to use the functional way of computation. Then, write the Haskell code.
- Test your functions individually.

Some useful functions

- `string->list`
- `list->string`
- `char=?`
- `char-numeric?`
- `char-whitespace?`
- `read-line`
- `display`
- `displayln`
- `string->number`
- `map`