B5 - Advanced Functional Programming

B-FUN-500

Functional EvalExpr

A better way

1.5.0

Functional EvalExpr

binary name:	funEvalExpr
group size:	2
repository name:	fun_evalexpr
repository rights:	ramassage-tek
language:	Haskell, OCaml, Scala
compilation:	stack, opam & Makefile, sbt

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.

Your program must be built using a Makefile. You can use stack/opam/sbt but it must be wrapper in the Makefile.
You must also initialise your build tool, if needed, (stack setup/opam init) from Makefile.

There should be no big surprises in this subject, you know what an EvalExpr is.
But this time, you'll have to implement your parser with a functional language.

You program MUST be able to parse a string from the command line argument, and output the resulting value, followed by a new line:
Your parser has to be implemented as a Packrat parser, following a Parsing Expression Grammar (PEG) and using the primitives you wrote in the previous projects; you are building a library that you are going to use in the whole unit (and probably in B-GCC-500 too).

Error messages have to be written on the error output, and, if necessary, the program should exit with a non-zero value.

You program MUST handle these operators:

- Sum: `+`
- Difference: `-`
- Unary plus and minus: `+ -`
- Product: `*`
- Division: `/`
- Power: `^`
- Squareroot: `v`
- Grouping: `( )`

The list above is sorted by precedence, from lower to higher.

You are FORBIDEN to use the imperative constructs of OCaml.
Everything MUST be functional.

You are obviously NOT allowed to use any parsing library you could find.

The final result must be rounded half away from zero to two digits after the comma.
So, for instance, 1.66666666 becomes 1.67

Examples

```
Terminal
~/B-FUN-500> ./funEvalExpr "3 + 5"
8.00
~/B-FUN-500> ./funEvalExpr "v(2*(3+1))"
2.83
```

Bonus

You could implement pretty much anything you want (related to the unit of course), but here are some examples:

- Assignment: `=`
- Reference: `variableName`
  It represents a way to assign and use a variable in your evalexpr.