Assignment 0 (main part): Arithmetic Expressions

Version 1.0

Due: Wednesday, September 9 at 20:00

The objective of this assignment is to gain some initial hands-on programming experience with Haskell. Please read through the *entire* assignment text before you start working on it.

Note This assignment also includes a collection of simple warm-up exercises, as described on Absalon. For those, you are *only* asked to submit your working code, but not a separate design/implementation document, assessment, or evidence of testing. If you want to communicate anything extra about your solutions to the warm-up exercises, place your remarks as comments in the source code.

1 Simple arithmetic expressions

Consider the following algebraic data type, representing simple *arithmetic expressions*:

```
data Exp =
    Cst Integer
  | Add Exp Exp
  | Sub Exp Exp
  | Mul Exp Exp
  | Div Exp Exp
  | Pow Exp Exp
--    ... (additional constructors; see next section)
```

That is, an arithmetic expression is either an (unbounded) integer constant, or one of the following five operations on two subexpressions: addition, subtraction, multiplication, division, or power (aka. exponentiation). Note that there is no provision for including explicit parentheses in the representation, as the tree structure of an `Exp`-typed value already encodes the intended grouping: the arithmetic expression conventionally written as $2 \times (3 + 4)$ is represented as `Mul (Cst 2) (Add (Cst 3) (Cst 4))`, whereas $(2 \times 3) + 4$ corresponds to `Add (Mul (Cst 2) (Cst 3)) (Cst 4)`.

1.1 Printing expressions

Define a function

```
showExp :: Exp -> String
```

that renders an arithmetic expression as a string, using normal mathematical/Haskell infix notation, as in the examples above. Use "+", "-", "*", "/", and "^" to represent the five arithmetic operators. Include enough parentheses in the output to ensure that the output string can be correctly read and evaluated as a Haskell expression, and that any two different `Exp`-trees have distinct renderings (even if they would evaluate to the same result, such as `"(2+3)+4"` and `"2+(3+4)"`). You are explicitly *allowed* to insert nominally redundant (according to the usual mathematical conventions) parentheses in the output, e.g., `"((2*3)+4)"`.

If the expression to be printed is not one of the above-mentioned six forms (i.e., if it belongs to the commented-out part of the `data Exp` declaration), your code should explicitly report the problem with an appropriate message (using the standard function `error`), rather than crash out with a non-exhaustive pattern-match error.

1.2 Evaluating expressions

Arithmetic expressions can be evaluated to numeric results. In this assignment, we only consider integer arithmetic, with n/m (for $m \neq 0$) defined as $\lfloor \frac{n}{m} \rfloor$, where $\lfloor r \rfloor$ (the *floor* of r) is the greatest integer less than or equal to r. (So $\lfloor \frac{7}{2} \rfloor = 3$, while $\lfloor \frac{-7}{2} \rfloor = -4$.) Also, we require that the exponent (second subexpression) in a `Pow`-operation is non-negative, and we specify that $n^0 = 1$ for all integers n, including 0.

Define a Haskell function

```
evalSimple :: Exp -> Integer
```

such that `evalSimple` e computes the value of e, under the interpretation of the arithmetic operators specified above. If an error occurs inside a builtin operation (e.g., a division by zero), it is fine to just abort with the relevant Haskell runtime error. And, like for printing, expressions not in the "simple" fragment of `Exp` should be explicitly reported as errors. If there are *multiple* errors in an expression, it doesn't matter which one gets reported.

2 Extended arithmetic expressions

We now consider a richer class of expressions, given by the full datatype:

```
data Exp =
-- ... (6 constructors from above)
  | If {test, yes, no :: Exp}
  | Var VName
  | Let {var :: VName, aux, body :: Exp}
  | Sum {var :: VName, from, to, body :: Exp}

type VName = String
```

Here, the expression form `If` e_1 e_2 e_3 (or, more verbosely, `If {test =` e_1`, yes =` e_2`, no =` e_3`}`) represents a *conditional expression* (analogous to e_1`?`e_2`:`e_3 in C). That is, its value is either the value of e_2, if e_1 evaluates to a non-zero number; or the value of e_3, if e_1 evaluates to zero. Only the selected branch, e_2 or e_3, is evaluated; for example, evaluating the expression

```
  If {test = Sub (Cst 2) (Cst 2),
      yes = Div (Cst 3) (Cst 0),
      no = Cst 5}
```

should return 5, and *not* abort with a division by zero.

The expression `Var` v, where v is a *variable name* (represented as a Haskell string), returns the current value (as specified below) of the variable v. If the variable has no current value, an error is signaled.

Conversely, the expression `Let` v e_1 e_2 is used to *bind* the variable v to the value of e_1 for (only) the duration of evaluating e_2; afterwards, the previous binding (if any) of v is reinstantiated. Thus, for example, the expression

```
Let {var = "x", aux = Cst 5,
     body = Add (Let {var = "x", aux = Add (Cst 3) (Cst 4),
                      body = Mul (Var "x") (Var "x")})
                (Var "x")}
```

should evaluate to $(3+4)^2 + 5 = 54$.

It is deliberately left *unspecified* (i.e., you as the implementer may decide) whether an error occurring in the auxiliary expression e_1 should be signaled if the bound variable v is not actually used in the body e_2. For instance,

```
Let "x"  (Div (Cst 4) (Cst 0))  (Cst 5)
```

is allowed to evaluate to 5, or to abort with a division-by-zero error (but nothing else).

Finally, the expression form `Sum` v e_1 e_2 e_3 corresponds to the mathematical construct $\sum_{v=e_1}^{e_2} e_3$. That is, it first evaluates e_1 and e_2 to numbers n_1 and n_2, and then computes the sum of the results of evaluating e_3, where v is bound to each of the values $n_1, n_1+1, ..., n_2$ in turn. For example, the expression

```
Sum "x"  (Cst 1)  (Add (Cst 2) (Cst 2))
    (Mul (Var "x") (Var "x"))
```

evaluates to $1^2 + 2^2 + 3^2 + 4^2 = 30$. If $n_1 > n_2$, the sum is defined to be 0 (and e_3 should not be evaluated at all).

We keep track of variable bindings in an *environment*, which maps variable names to their values (if any). We represent environments as *functional values*:

```
type Env = VName -> Maybe Integer
```

That is, for an environment r `::` `Env` and variable v `::` `VName`, the application r v returns `Nothing` if v has no binding in r, and `Just` n if v is bound to the integer n. The environment in which all variables are unbound can be written as simply `initEnv = \v -> Nothing`, while the one in which variable `"ans"` is bound to 42 (and all others are unbound) would be represented as the following function:

```
\v -> if v == "ans" then Just 42 else Nothing
```

Define a function

```
extendEnv :: VName -> Integer -> Env -> Env
```

such that `extendEnv` v n r returns a new environment r', in which v is bound to n, and all other variables have the same bindings as they did in r.

Then define a function

```
evalFull :: Exp -> Env -> Integer
```

that evaluates an expression in a given environment. As before, errors should be signaled with `error`, and may now – in addition to erroneous arithmetic operations – include attempts to access unbound variables. On the other hand, all expression forms in `Exp` should now be covered.

Be sure to explain in the report how you chose to deal with errors in unneeded parts of `Let`-expressions, and why. (Simplicity of implementation is a perfectly acceptable justification.)

3 Returning explicit errors

Promptly aborting evaluation with a Haskell `error` is a fairly drastic step, and in particular makes it impossible to recover gracefully from a conceptually non-fatal problem. A more flexible approach makes the evaluator function return an explicit indication of what went wrong, and lets the caller decide what to do next. Accordingly, we first enumerate some possibilities:

```
data ArithError =
    EBadVar VName   -- unbound variable
  | EDivZero        -- attempted division by zero
  | ENegPower       -- attempted raising to negative power
  | EOther String   -- any other errors, if relevant
```

Define a Haskell function,

```
evalErr :: Exp -> Env -> Either ArithError Integer
```

such that `evalErr` e r attempts to evaluate e in environment r, as in `evalFull`, but now returns either an error value (e.g. `Left ENegPower`) or a proper result (e.g. `Right 42`). `evalErr` should *never* cause a Haskell runtime error.

For `evalErr`, we also specify explicitly that all subexpressions are to be evaluated left-to-right, so that, e.g., if we are evaluating `Add` e_1 e_2, and e_1 returns an error, e_2 should not be evaluated. However, it is still left unspecified (meaning: you should choose) whether errors in unused `Let`-bindings should be reported or ignored. Again, justify your choice in the report.

Hint: The code of `evalErr` may become somewhat verbose and repetitive; try to abstract common code snippets into (higher-order) auxiliary functions, so that you only have to write them once. Do not attempt to use Haskell's builtin imprecise-exception facility: it is quite finicky, and will most likely not do what you need.

4 Optional extensions

The problems in this section are a bit more challenging, and hence not mandatory, but still recommended for extra practice. They are independent, so you may do any subset of them.

Note that your performance on the optional problems will *not* affect whether you pass the assignment: an incomplete or buggy solution here will not drag down an otherwise acceptable solution of the mandatory part; and conversely, you cannot save an otherwise failing solution of the main problems by successfully solving one or more of the optional ones.

4.1 Proper error propagation in exponentiation

The semantics of exponentiation is specified above as $n^0 = 1$ for all integers n, just like $n \times 0 = 0$ for all n. However, in a general expression `Pow` e_1 e_2, if evaluation of the subexpression e_1 results in an *error* (division by zero, unbound variable, etc.), we still want evaluation of the whole expression to report that same error, even if e_2 evaluates to 0 (so that e_1's actual value is not really needed). Make your implementations of especially `evalSimple` and `evalFull` ensure this, so that, e.g., `(Pow (Div (Cst 0) (Cst 0)) (Cst 0))` results in an error, and *not* a successful result of 1. (If your evaluator already behaves like this, you don't need to do anything here!)

4.2 Printing with minimal parentheses

Define a variant of the `showExp` function,

```
showCompact :: Exp -> String
```

that prints a simple arithmetic expression using the minimal number of parentheses (and no extra spaces). The function must still be one-to-one, i.e., no two different values of type `Exp` should be rendered into the same string. You should assume that the arithmetic operators have the conventional precedences and associativities (e.g. `Add (Cst 2) (Mul (Cst 3) (Cst 4))` should print as `"2+3*4"`, and `Add (Cst 2) (Add (Cst 3) (Cst 4))` as `"2+(3+4)"`). Note in particular that `"2^3^4"` corresponds to 2^{3^4}, which conventionally means $2^{(3^4)}$, not $(2^3)^4$.

Hint: You will probably want to define `showCompact` in terms of an auxiliary recursive function that takes as parameters both the expression to be printed, and some additional data to determine whether explicit parentheses are needed around it, given the context it appears in.

4.3 Explicitly eager/lazy semantics

In `evalErr`, the exact semantics of `Let`-expressions was left partially unspecified. There are in fact two natural interpretations of an expression `Let` v e_1 e_2: the *eager* one, where e_1 is always evaluated proactively, regardless of whether v is used in e_2; and the *lazy* one, where e_1 is only evaluated as and when its value is needed for computing e_2.

Accordingly, define two functions

```
evalEager :: Exp -> Env -> Either ArithError Integer
evalLazy :: Exp -> Env -> Either ArithError Integer
```

implementing the two variants (for all `Let`s occurring in the first argument). For example, the call `evalLazy (Let "x" (Var "y") (Cst 0)) initEnv` should now return `Right 0`, whereas `evalEager (Let "x" (Var "y") (Cst 0)) initEnv` should return `Left (BadVar "y")`

Hint: If your `evalErr` from the mandatory part already implements precisely one of the two behaviors, you can simply use it directly as the relevant definition. And again, you may want to express one or both of the main evaluation functions in terms of another function with additional and/or differently typed parameters; in particular, your internal function might use a slightly different type for environments.

5 What to hand in

5.1 Code

Form To facilitate both human and automated feedback, it is very important that you closely follow the code-packaging instructions in this section. We provide skeleton/stub files for all the requested functionality in both the warm-up and the main part. These stub files are packaged in the handed-out `code.zip`. It contains a single directory `code/`, with a couple of subdirectories organized as Stack projects. You should edit the provided stub files as directed, and leave everything else unchanged.

It is crucial that you not change the provided types of any exported functions, as this will make your code incompatible with our testing framework. Also, *do not* remove the bindings for any functions you do not implement; just leave them as `undefined`.

When submitting the assignment, package your code up again as a single `code.zip` (*not* `.rar`, `.tar.gz`, or similar), with exactly the same structure as the original one. When rebuilding `code.zip`, please take care to include only files that constitute your actual submission: your source code and supporting files (build configuration, tests, etc.), but *not* obsolete/experimental versions, backups, editor autosave files, revision-control metadata, `.stack-work` directories, and the like. If your final `code.zip` is *substantially* larger than the handed-out version, you probably included something that you shouldn't have.

For the warm-up part, just put your function definitions in `code/part1/src/Warmup.hs`, where indicated.

For the main part, your code must be placed in the file `code/part2/src/Arithmetic.hs`. It should only export the requested functionality. Any tests or examples should be put in a separate module under `code/part2/tests/`. For inspiration, we have provided a very minimalistic (and far from adequate) test suite in `code/part2/tests/Test.hs`. If you are using Stack (and why wouldn't you be?), you can build and run the suite by `stack test` from the directory `code/part2/`.

The definitions for this assignment (e.g. type `Exp`) are available in file `.../src/Definitions.hs`. You should only import from this module, and not directly copy its contents into `Arithmetic`. And again, do not modify anything in `Definitions`.

Content As always, your code should be appropriately commented. In particular, try to give brief informal specifications for any auxiliary "helper" functions you define, whether locally or globally. On the other hand, avoid trivial comments that just rephrase in English what the code is already saying in Haskell. Try to use a consistent indentation style, and avoid lines of over 80 characters.

You may (but shouldn't need to, for this assignment) import additional functionality from the core GHC libraries only: your solution code should compile with a `stack build` issued from the directory `code/part`*n*`/`, using the provided `package.yaml`. (For *testing*, you may optionally use additional packages from the course-mandated version of the Stack LTS distribution, e.g., test frameworks. Later in the course, we will use `Tasty` and `QuickCheck`.)

Your code should ideally give no warnings when compiled with `ghc(i) -W`; otherwise, add a comment explaining why any such warning is harmless or irrelevant in each particular instance. If some problem in your code prevents the whole file from compiling at all, be sure to comment out the offending part before submitting, or all the automated tests will fail.

5.2 Report

In addition to the code, you must submit a short (normally 2–3 pages) report, covering the following two points, for the main (not warm-up) part only:

- Document any (relevant) *design* and *implementation* choices you made. This includes, but is not limited to, answering any questions explicitly asked in the assignment text. Focus on high-level aspects and ideas, and explain *why* you did something non-obvious, not only *what* you did. It is rarely appropriate to do a detailed function-by-function code walk-through in the report; technical remarks about how the functions work belong in the code as comments.

- Give a honest, justified *assessment* of the quality of your submitted code, and the degree to which it fulfills the requirements of the assignment (to the best of your understanding and knowledge). Be sure to clearly explain any known or suspected deficiencies.

  It is very important that you document on what your assessment is based (e.g., wishful thinking, scattered examples, systematic tests, correctness proofs?). Include any automated tests you did with your source submission, make it clear how to run them, and summarize the results in the report. If there were some aspects or properties of your code that you couldn't easily test in an automated way, explain why.

Your report submission should be a single PDF file named `report.pdf`, uploaded along with (not inside!) `code.zip`. The report should include a listing of your code and tests (but not the already provided auxiliary files) as an appendix.

5.3 General

Detailed upload instructions, in particular regarding the logistics of group submissions, can be found on the Absalon submission page.

We also *expect* to provide an automated system to give you preliminary feedback on your planned code submission, including matters of form, style, correctness, etc. The details will be announced on Absalon. You are **strongly advised** to take advantage of this opportunity to validate your submission, and – if necessary – fix or otherwise address (e.g., by documenting as known flaws) any legitimate problems it uncovers.

Note, however, that passing the automated tests is *not* a substitute for doing *and documenting* your own testing. Your assessment must be able to stand alone, without leaning on the output from our tool.