The University of Hong Kong

COMP3259: Principles of Programming Languages

Assignment 1

Tutor
Wan Qianyong `qywan@cs.hku.hk`

Instructor
Bruno Oliveira `bruno@cs.hku.hk`

Deadline: 23:59, 9 March 2025

Table of Contents

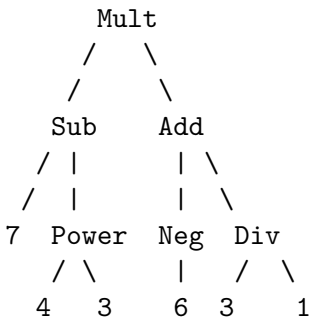
Note

Your solution must be submitted in a single zip file, named as *A1_XXX.zip*, with *XXX* replaced by your UID. Please submit your solution on Moodle before the deadline. Your code should be well-written (e.g. proper indentation, readable names, and type signatures). You can test your code using `stack test`. Please at least make sure your code compiles!

Please do this assignment on your own; if, for a small part of an exercise, you use something from the Internet or were advised by your classmate, please mark and attribute the source in a comment. Do not use publicly accessible code sharing websites for your assignment to avoid being suspected of plagiarism.

1 Extending the Calculator

In the second tutorial, we have learned how to deal with a simple language for arithmetic expressions, including addition, subtraction, multiplication, division, exponentiation, and negation. As a recap, please first answer the following questions:



Question 1. (4 pts.) Please write down the corresponding Haskell definition for the expression represented by the abstract syntax tree above in `Declare.hs`.

Question 2. (1 pts.) What's the result of evaluating this expression? (Please answer it as comments in `Declare.hs`)

Now let's make our calculator more powerful!

Your first task is to extend the datatype `Exp` with the following operations:

- `Fact` (`x!`): factorial of an integer;
- `Mod` (`x % y`): modulo operation (remainder after division)

We have already added the new operations in `Declare.hs`, and provided a parser that supports factorial and modulo operators for you. The operator `%` has the same precedence as `*` and `/`; and `!` has the same precedence as `^`.

Question 3. (5 pts.) Complete the definition of `showExp`.

During the tutorials, we have learned how to use the `Either` type to handle errors in Haskell. Instead of raising errors by the `error` function and crashing the program as we

did in the previous exercises, we can use `Either` to return error messages and propagate them through the computation. In this exercise, we will use `Either` to handle errors in the calculator. We have provided a function `evaluate` in `Interp.hs` that evaluates an expression with basic arithmetic operations and returns an `Either` value. The function should return `Right` with the result of the evaluation if the expression is valid, and `Left` with an error message if the expression is invalid.

Question 4. (10 pts.) Complete the definition of `evaluate` with factorial and modulo operations. The function should report error messages for invalid expressions:

- Evaluating `n!` when `n` is a negative number will report: `Factorial of a negative number: [expression of n]`
- Taking the modulo of zero will report: `Divided by zero: [the expression that produces zero]`

After finishing the tasks above, you can also test your evaluator with the `parseExpr` and `evaluate` function with a concrete syntax expression as input.

2 Variables and Substitutions

It is also useful to have variables in our language. We can define a variable as a string, and an environment that maps variables to values. We have defined the `Env` datatype to represent the environment in `Declare.hs` as `type Env = [(String, Int)]`. In this exercise, we assume that there are no name clashes in the environment, i.e. all the variables in the environment are distinct from each other.

Question 5. (10 pts.) Please complete the `subst` function in `Declare.hs` to substitute the value of a variable in an expression. The function should take a string as the variable name to be replaced, an integer and an expression as input, and return a new expression with all the occurrences of the variable in the original expression replaced by the interger value. For example `subst "x" 4 (Add (Var "x") (Num 3))` should return `Add (Num 4) (Num 3)`.

Question 6. (5 pts.) We can define multiple substitutions based on the single substitution function `subst`, by repeatedly applying `subst` on each pair of bindings in the environment. Please complete the `substs` function in `Declare.hs` to substitute multiple variables in an expression. The function should take an environment and an expression as input, and return a new expression with all the variables replaced by their values in the environment. Note that you are required to use the `subst` function in the implementation of `substs`. Failure to do so will result in a deduction of points.

With the `substs` function, we can now evaluate an expression with variables in the environment. For example, given an environment `env = [("x", 4), ("y", 3)]`, the expression `Add (Var "x") (Var "y")` should be first substituted to `Add (Num 4) (Num 3)`, and then evaluated to `7`. We have provided a function `calc` in `Interp.hs` that combines these two steps, together with the parser, to evaluate an expression with variables in the environment. Feel free to test your code with the `calc` function.

However, it's invalid for a substituted expression to still contain variables. We should also extend our `evaluate` function to handle such case:

Question 7. (5 pts.) Complete the definition of `evaluate` with variables. The function should simply report an error when encountering variables: `Variable not found: [variable name]`.

3 Taming Eithers

If you are such a programmer who has a good taste of code, you won't be much happy with your solution for `evaluate` so far. Though it works, the code is ugly, and the nested creeping indentation may drive you crazy.

Notice that there is a pattern recurring in the code. Every time we evaluate some expressions, we get an `Either` value, we do pattern matching on it and fork the computation: if the result is `Right`, we make the contained value available to the rest of computation; if the result is `Left`, we skip the rest of the computation and propagate the error message.

A good programmer is able to recognize the pattern, and try to abstract it to improve the code. Let's do it!

Let's start with a function called `andThen`, which accepts an `Either` value and a function that encapsulates the rest of the computation, and returns another `Either` value. More specifically, `andThen` has the following type signature:

```
andThen :: Either a b -> (b -> Either a c) -> Either a c
```

Don't get intimidated by the type signature above! The definition is just like what we had in the `evaluate` function: We do pattern matching on the first argument, if it is `Left`, we just return it; If it is `Right`, we apply the second argument (which is a function) to the value contained in `Right`, so that we get another `Either` as the final return value.

Question 8. (5 pts.) Complete the definition of `andThen`.

Having `andThen` functions at our service, we can start to refactor `evaluate` to make it neater. The general idea is, when you have a sequence of tasks with each having a result of `Either`, you can finish the first one, and encapsulate the remaining work into a function ("continuation"), then use `andThen` to compose them.

For example, suppose we have a function `(!?) :: [a] -> Int -> Maybe a` that returns `Nothing` if the indexed element does not exist in list, otherwise `Just` the element, and an `andThen` function defined similarly for `Maybe`, we can use it to simplify the following two-step conditional index:

```
case [[1,2],[3,4]] !? 1 of
  Nothing -> Nothing
  Just a -> case a !? 0 of
    Nothing -> Nothing
    r@(Just _) -> r
```

```
[[1,2],[3,4]] !? 1 `andThen` \a ->
  a !? 0 `andThen` \b ->
    Just b
```

Question 9. (15 pts.) Create a new function `evaluate2` that is similar to `evaluate`, but uses `andThen`. Note that now you don't have to explicitly deal with `Either` values (e.g. pattern matching), except when you have to create them. The resulting code should be significantly simpler than `evaluate`.

After answering the questions above, you may appreciate the power of abstraction. Actually this kind of pattern has a bizarre name in the functional programming literature: we call `Either` a *monad*. Don't worry if you don't understand it at the moment. We will teach you more about it in the future!

4 Eliminating Zeros

Imagine that we want to evaluate the following expression:

$$(1 + 2 \times 3 + 4 \div 5 - 6 + 7^8 - 9) \times 0$$

We may work very hard in evaluating the left sub-expression inside the parentheses. Once we finish it, we will see that it is immediately multiplied by 0, which gives 0 eventually! In this case, it's endurable; after all, we only did evaluation involving 8 operators and 9 numbers. But how about thousands of operators and numbers? I wish I could know in advance that the result is going to be 0 anyway, so I can save my efforts!

Let's write a function that can help us detect it.

Question 10. (20 pts.) Define a new function `elim0` that does the following:

- Simplify $n + 0$ and $0 + n$ to n;
- Simplify $n - 0$ to n;
- Simplify $n \times 0$ and $0 \times n$ to 0;
- Simplify $0 \div n$ and $0 \ \% \ n$ to 0;
- Simplify n^0 to 1 and 0^n to 0;
- Simplify -0 to 0;
- Simplify $0!$ to 1;
- Otherwise leave the expression unchanged.

For example:

```
*Interp> elim0 (Mult (Num 3) (Num 0))
0
```

In this exercise, we don't bother about the validity of the expression, meaning that it's legal to simplify expressions like $0 \div n$ no matter what n is.

Notice that you need to do it recursively:

```
*Interp> elim0 (Add (Num 4) (Neg (Mult (Num 3) (Num 0))))
4
```

5 Inference Rules

In the lecture, we have shown the inference rule for multiplication, namely $E\times$:

$$\frac{e_1 \to n_1 \qquad e_2 \to n_2}{e_1 \times e_2 \to n_1 \times n_2}\ \mathrm{E}\times$$

Similar to what we did for eliminating zeros, we want to optimize the evaluation a bit. Thus, besides the existing one, we add two more rules:

$$\frac{e_1 \to n_1 \qquad n_1 = 0}{e_1 \times e_2 \to 0}\ \mathrm{E}\times\mathrm{L0} \qquad\qquad \frac{e_2 \to n_2 \qquad n_2 = 0}{e_1 \times e_2 \to 0}\ \mathrm{E}\times\mathrm{R0}$$

Question 11. (10 pts.) Write a derivation tree for the expression e below with the new set of inference rules.

$$e = (4 + 8) \times (1 - 1) + 16 \times 3$$

Question 12. (10 pts.) Based on our new inference rules, can there be more than one derivation? If so, show an example and try to change the rules to make sure only one derivation is possible. If not, try to explain why.

The answers to the two questions above can be written in any form (e.g. hand-written, typeset, or ASCII art). Please remember to pack a soft copy of your answers into the zip file you submit.