Tutorial 2

The goal of this tutorial is to build a simple interpreter that supports basic arithmetic calculations with error handling. We will use the concepts learned so far in the lectures and put them in good use.

Tutorial Bundle

For this tutorial there is a bundle of files that can be downloaded from Moodle. Look for Bundle for Tutorial 2 under section *Tutorials and Assignments*.

A Language of Arithmetic

Recall that in Monday's lecture, we were presented with the following definition for the abstract syntax of the arithmetic language we will be using (and later extending):

```haskell
data Exp = Num Int
         | Add Exp Exp
         | Sub Exp Exp
         | Mult Exp Exp
         | Div Exp Exp
```

Question 1: Add a unary operator "negation" (e.g., `Neg 3` stands for -3) and a binary operator "exponentiation" (e.g., `Power 2 3` stands for 2^3) to the above datatype `Exp`. Complete your definition in the file *Declare.hs*. Please use `Neg` and `Power` as the constructor names.

Concrete Syntax, Abstract Syntax and Abstract Syntax Tree

For any abstract syntax expression of type `Exp`, we can draw a corresponding abstract syntax tree. For example, the expression:

```haskell
e1 :: Exp
e1 = Add (Num 3) (Num 4)
```

has the abstract syntax tree as shown by Figure 1.

For non-recursive constructors (such as `Num`) we omit the constructor name. For recursive constructors, we use the name of the constructor to create a tree node and, for each argument of the constructor we create a sub-tree.

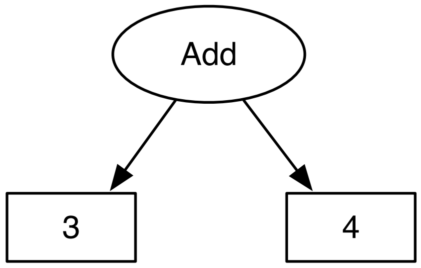
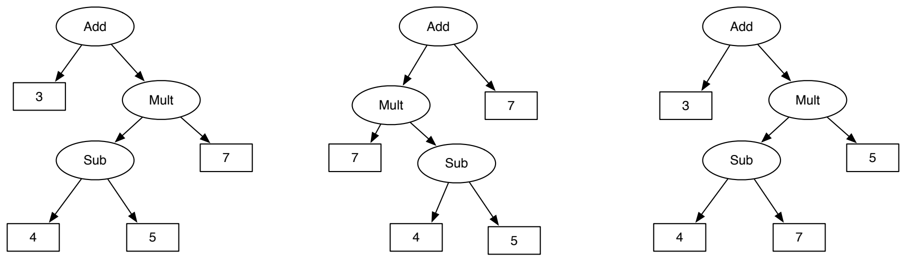


Figure 1: Abstract Syntax Tree of e1

Question 2: What is the correct abstract syntax tree for the following expression?

```
e2 :: Exp
e2 = Add (Num 3) (Mult (Sub (Num 4) (Num 5)) (Num 7))
```



Option A Option B Option C

Lexer, Parser and Interpreter

In Monday's lecture, we have been shown the definition of a interpreter for arithmetic expressions as follows:

```
evaluate :: Exp -> Int
evaluate (Num n)    = n
evaluate (Add a b)  = evaluate a + evaluate b
evaluate (Sub a b)  = evaluate a - evaluate b
evaluate (Mult a b) = evaluate a * evaluate b
evaluate (Div a b)  = evaluate a `div` evaluate b
```

Question 3: Following the logic of the above `evaluate` function, extend it to support negation and exponentiation operations in the file *Interp.hs*. (You can use ˆ operator like `2 ^ 3` to calculate “2 to the power of 3”, and `negate` function like `negate 3` in Haskell, have a try in `ghci`.)

It is cumbersome to have to write long `Exps` like `Add (Num 3) (Num 4)` and then feed it to `evaluate`. We humans like to write the string `"3 + 4"`, and then let the interpreter tell us the answer. To achieve this, we need to find a way to tell the computer what those symbols `(, *, +, )` and so on mean.

This is where parsing comes onto stage. Parsing is a process that converts between *concrete syntax* and *abstract syntax*. For example, the expression $1 + 8 \times 2$ would be parsed to

```
Add (Num 1) (Mult (Num8) (Num 2))
```

Writing parsers can sometimes be unbearably tedious. Good news is there are lots of *parser generator* tools that can help automate the work. In this tutorial, we will be using Happy, a parser generator for Haskell.

A parser needs a set of rules called *grammar* that specify valid combinations of tokens to form expressions of a language. Fortunately for our tiny language, the grammar is as simple as the following:

```
Term     ::= Term '+' Factor
           | Term '-' Factor
           | Factor

Factor   ::= Factor '*' Primary
           | Factor '/' Primary
           | Primary

Primary  ::= digits
           | '(' Term ')'
```

In the bundle, we have included a Happy grammar file named *Parser.y*. Happy will take this grammar file and generate a Haskell module that provides a parser called `parseExpr`. This can be done as follows:

```
> happy Parser.y
```

Now you should see a Haskell file named *Parser.hs* in the directory. To use it, just load the file on `ghci`:

```
> ghci Parser.hs
```

You can type in any arithmetic expression and feed it to `parseExpr`. For example:

```
*Parser> parseExpr "(1 + 2) * 3 / 4"
Div (Mult (Add (Num 1) (Num 2)) (Num 3)) (Num 4)
```

It is recommended that you should try some expressions involving negation and/or exponentiation operations to see if your definition of `Exp` is working correctly.

Putting all together, now you should have a working interpreter that could do simple arithmetic calculations.

Question 4: In the file *Interp.hs*, write a function called `calc`, which, given an arithmetic expression, evaluates it and returns the result. The type signature of `calc` is as follows:

```
calc :: String -> Int
```

(Hint: Think about what helper functions are at your service and how to compose them.)

Pretty printing and Type Classes

Congratulations! At this point, you have created a fully working calculator all by yourself. If you like, you can extend it to have more operations like finding the square root of a given number, or calculating factorials.

One nice feature is to have the interpreter printing the textual representation of the abstract syntax. The existing interpreter we have now does support a function called `show` that can be used to print the abstract syntax of an expression. This function is automatically generated by Haskell using `deriving Show` in the declaration of the datatype `Exp`. For example, trying:

```
*Declare> e2
Add (Num 3) (Mult (Add (Num 4) (Num 5)) (Num 7))
```

on the console will automatically invoke that `show` function and print the textual representation of `e2`.

However, as you also notice, the textual representation generated is not very pretty because it is based on the abstract syntax.

Now you are asked to define a variant of the `show` function that is prettier and prints output based on the concrete syntax instead. For example, the expression `e2` would be printed as follows:

```
*Declare> e2
3 + (4 + 5) * 7
```

Question 5: Implement the variant of the `show` function.

To define this function, do the following:

1. Go to the file *Declare.hs* and comment or remove `deriving Show` from the datatype definition of `Exp`.
2. Create an instance of the type class `Show` for `Exp` by using the following code:

   ```
   instance Show Exp where
     show = showExpr
   ```

   This code allows you to define your own version of the `show` function to print values of type `Exp`.
3. In the same file, look for the following definition

   ```
   showExpr :: Exp -> String
   showExpr = error "TODO: Question 5"
   ```

   You need to replace this definition of `showExpr` by a definition that creates a string representation of the input expression. Once this definition is done you should be able to get the same result (not exactly) as shown before.

For simplicity, put brackets around every expression using `Add`, `Sub`, `Mult`, `Div` and other operators, and also add spaces around each operator. The result will not be as pretty as it could be, but we will avoid having to figure out when to omit brackets based on the precedence and associativity of the operators.

Another example for you to try on:

```
e3 :: Exp
e3 = Sub (Div (Add (Num 1) (Num 2)) (Num 3)) (Mult (Sub (Num 5) (Num 6)) (Num 8))
```

Here is the expected output in `ghci`:

```
*Declare> e3
(((1 + 2) / 3) - ((5 - 6) * 8))
```

Error Handling

Though our tiny interpreter is good at doing arithmetic calculations, it falls to provide us useful information when it's malfunctioning. For example, when typing in:

```
*Interp> calc "(1+2)/(3-3)"
*** Exception: divide by zero
```

Oops, a *divide-by-zero* exception! This kind of error message is fine when the expression is small enough. We can immediately spot that, in the divisor, $3 - 3$ evaluates to 0. But when the expression itself becomes larger and larger, we would like to know more than just that.

If you remember, we've seen one way of dealing with errors: calling `error` function, which terminates the program immediately. In functional programming, if a function is not defined for all values of arguments (e.g., raises an exception, loops forever), we say that this function is *partial*. You can guess the opposite ones are called *total* functions – functions defined for all values of their arguments. In Haskell, we would like to have all functions being *totoal*. How to do that? The trick is to have functions returning a different type, a type that encodes both failure and success of a computation!

Let us define a datatype that represents values of two possibilities:

```
data Either a b = Left a | Right b
```

That is to say, a value of type `Either a b` is either `Left a` or `Right b`. By convention, the `Left` constructor is used to hold an error value, and the `Right` constructor is used to hold a correct ("right" also means "correct") value.

Here is how we can use `Either` to make a safe version of `head` function (compared with the one implemented by `Maybe` datatype). Remember that when `head` is applied to an empty list, it throws an exception.

```
safeHead :: [a] -> Either String a
safeHead []     = Left "can't access the head of an empty list"
safeHead (x:_)  = Right x
```

Now let's try to apply `safeHead` to an empty list and see what happens:

```
*Interp> safeHead []
Left "can't access the head of an empty list"
```

To incorporate `Either` into our interpreter, we need to change the return type of `evaluate` as follows:

```
evaluate2 :: Exp -> Either String Int
```

Question 6: In the file *Interp.hs*, complete the definition of `evaluate2` with error handling. Also create a similar function `calc2` that uses `evaluate2`.

Here is an example of evaluating `Add a b` (you should follow the example and complete others):

```
evaluate2 (Add a b) = case evaluate2 a of
                        Left msg -> Left msg
                        Right a' -> case evaluate2 b of
                                     Left msg -> Left msg
                                     Right b' -> Right (a' + b')
```

Notice that the burden now is on the caller to pattern match the result of any such call so that, it either continues with the successful result, or handles the failure (in our case, just propagate the error massage).

Specifically, there are two possible errors when doing integer calculations:

1. When divide by zero (e.g., $2 \div 0$).
2. When raised to a negative number (e.g., 2^{-3}).

You can issue whatever error messages you like, just make sure the above two situations have different error messages, and provide some information about where the error arises. For simplicity, just print out the problematic expression using the pretty printer you just made.

Here are examples of possible error messages:

```
*Interp> calc "(2 * 6) / (3-3)"
Left "Divide by zero: (3 - 3)"

*Interp> calc "(2 - 3) ^ (2 - 4)"
Left "Raised to a negative number: (2 - 4)"
```

If you are such a programmer who has a good taste of code, you won't be much happy of your solution in the last question. Though it works, the code is so ugly, and the nested creeping indentation makes you want to stratch your head.

Notice that there is a pattern recurring in our code. Every time we evaluate some expression, we get an `Either` value, we pattern match 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!

For starters, let's define a function called `bindE`, which given an `Either` value and a function that encapsulates the rest of the computation, returns another `Either` value.

More specifically, `bindE` has the following type signature:

```
bindE :: Either a b -> (b -> Either a b) -> Either a b
```

Don't get intimidated by the above type signature! The definition is just like what we had in the `evaluate2` function: We pattern match the first argument, if it is `Left`, we just return it; If it is `Right`, we apply the value contained in `Right` to the second argument (which is a function) and get another `Either` value.

Question 7: Complete the `bindE` function.

Having `bindE` functions at our service, we can start to refactor `evaluate2` to make it much cleaner.

Question 8: Create a new function `evaluate3` that is similar to `evaluate2`, but uses `bindE`. 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 `evaluate2`.

Epilogue

As you finish your last question, you should be able to appreciate the power of abstraction. Actually this kind of pattern has a bizarre name in the functional programming literature, we call it an `Either` **Monad**. Don't worry if you don't understand, we will be dealing with it a lot in the future tutorials!

Assignment 2

Your second assignment is to try to complete as many questions in the tutorial as you can. All questions are **graded**. You should submit your solutions to Moodle. Look for Submission of Tutorial 2 under section *Submissions and Score Reporting*.

The deadline is **October 1st**.

Please submit your answers using the modified Haskell files. **Try to ensure that the files you submit compile. If you submit files that do not compile, points will be removed from your grade. Try to comment the code explaining what you intended to achieve. If you cannot create code that compiles, it is especially important to write some comments explaining what you intended to achieve.**