The University of Hong Kong

COMP3259: Principles of Programming Languages

Tutorial 2

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

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

18 February 2025

Table of Contents

1 Introduction

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.

2 A Language of Arithmetic

First, please open `Declare.hs` under the `src` folder.

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

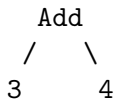
```
data Exp = Num Int
         | Add Exp Exp
         | Sub Exp Exp
         | Mult Exp Exp
         | Div Exp Exp
         | Power Exp Exp
         | Neg Exp
         deriving Show
```

2.1 Abstract Syntax Tree

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

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

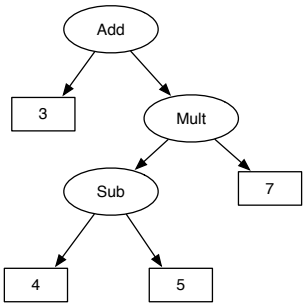
has the abstract syntax tree as follows:



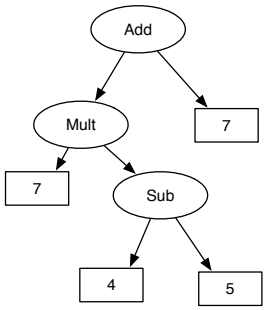
For non-recursive constructors (such as `Num`), we omit the constructor name. For recursive constructors, we use the name of the constructor to create a tree node and represent each argument of the constructor as a sub-tree.

Question 1. Which one is the correct abstract syntax tree for the following expression?

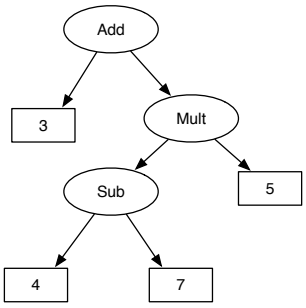
```
e2 :: Exp
e2 = Add (Num 3) (Mult (Sub (Num 4) (Num 5)) (Num 7))
```



Option A



Option B



Option C

2.2 Interpreter and Parser

Now, please open `Interp.hs` under the `src` folder.

In the 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 2. Following the logic of the above `evaluate` function, extend it to support negation and exponentiation operations in the file `Interp.hs`. (You can use the `^` operator like `2 ^ 3` to calculate "2 to the power of 3" and the `negate` function like `negate 3` in Haskell. Have a try in GHCi!)

It is cumbersome to have to write long `Exp`s 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 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 (Num 8) (Num 2))
```

Writing parsers can sometimes be unbearably tedious. Good news is that there are lots of *parser generator* tools that can help automate the work. In this tutorial, we will use Happy, a parser generator for Haskell. Let's install Alex (the Lexer) and Happy by running the following command:

```
stack install alex happy
```

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 code bundle, we have included a Happy grammar file named `Parser.y` in `src` directory. 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 `src` 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 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. Let's do it!

Question 3. 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.)

2.3 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, under the `src` folder, trying:

```
> ghci Declare.hs
......
*Declare> e2
Add (Num 3) (Mult (Sub (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 4. Implement the variant of the `show` function.

To define this function, do the following:

1. Go to the file `Declare.hs` and comment out `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 = showExp
   ```

   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:

   ```
   showExp :: Exp -> String
   showExp = error "TODO: Question 4"
   ```

   You need to replace this definition of `showExp` 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))
```

2.4 Error Handling

Let's go to the `Interp.hs` again.

Though our tiny interpreter is good at doing arithmetic calculations, it fails 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 *total*. 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 5. In the file `Interp.hs`, complete the definition of `evaluate2` with error handling. Also create a function `calc2` that is similar to `calc` but 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. It will either continue with the successful result or handle the failure (in our case, just propagate the error message).

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

1. When divided by zero (e.g., $2 \div 0$).
2. To the power of 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 "(1 + 2) / (3 - 3)"
Left "Divided by zero: (3 - 3)"

*Interp> calc "(2 - 3) ^ (2 - 4)"
Left "To the power of a negative number: (2 - 4)"
```

2.5 Inference Rules

In the lecture, we have also seen how to define operational semantics through inference rules, in which we write $e \to n$ to indicate "e evaluates to n". A judgement like this is thought to be true only if the pair (e, n) is in the relation defined by our inference rules, which means there exists a derivation with $e \to n$ at the root.

$$\dfrac{\dfrac{}{3 \to 3}\,\text{EN} \qquad \dfrac{\dfrac{\dfrac{}{4 \to 4}\,\text{EN} \qquad \dfrac{}{5 \to 5}\,\text{EN}}{4 - 5 \to -1}\,\text{E}- \qquad \dfrac{}{7 \to 7}\,\text{EN}}{(4 - 5) \times 7 \to -7}\,\text{E}\times}{3 + (4 - 5) \times 7 \to -4}\,\text{E}+$$

An example of a derivation is given above. The derivation shows how we can assign a meaning (i.e. result) to e_2. Typically, we start writing each derivation bottom-up (i.e. starting with the original expression at the root) and leave the right-hand side of the arrow blank. The right-hand side will be the semantic result for that expression, which will be filled in a top-down manner. At this point, there is only one rule we can apply, by looking at its conclusion: $E+$. Thus, we can write the line above the expression, and split the derivation into two, corresponding to both sub-expressions. Again, we do not fill the right-hand sides, yet. Informally, when applying these rules, we are saying: *assuming that I can derive the semantics for some e_1 and e_2, then I can also derive that for $e_1 + e_2$.*

We repeat this process until we reach the leaves of the derivation, that is, no more hypothesis is needed. Now, we can easily fill in the missing right-hand sides, by starting at the leaves, and going down the derivation tree. By looking at each rule, we can see how to combine each result in the premises in order to fill the result in the conclusions. For instance, the top-most $E-$ combines both results of its premises 4 and 5 by subtraction, denoted as the result -1. We can keep applying this process, until we reach our original expression. Notice that, when executing our `evaluate` functions, Haskell will perform this same derivation for us, even though we only get to see the final result.

Question 6. Write the derivation for the expression e_3.

If this is the case, then the inference rules are syntax directed

Question 7. Based on our inference rules, for an expression, can there be more than one derivation? Try to explain why.

As we can see, there is a corespondence between the inference rules and our evaluator. When writing a derivation, we are going through the recursive calls of our `evaluate` function.

Recall that in the lecture, we have the rule $E+$:

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

The following rule is another equivalent version. We assume the knowledge of arithmetic equality, so you don't need to write any reasoning for a correct arithmetic equation in the form of $n = n_1 + n_2$ (e.g. $5 = 2 + 3$).

$$\frac{e_1 \to n_1 \qquad e_2 \to n_2 \qquad n = n_1 + n_2}{e_1 + e_2 \to n}\ \mathrm{E}+',$$

Question 8. Assuming we replace the inference rule $E+$ by $E+'$, please write the derivation for the e_2.