Middle East Technical University — Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring '2018-2019

Homework 2

Due date: 7 April 2019, Sunday, 23:55

1 Objectives

This assignment aims to assist you to expand your knowledge on functional programming by the help of Haskell programming language.

2 Problem Definition

In this assignment, you will be taking one step further in the abstract representation of a given expression by handling erroneous case regarding *only* the types of operands. Moreover, by expanding the definition of the AST type in our program, we combine variable mapping and the expression itself in an AST. This new approach will give us a chance to observe not only variable scopes in an expression but also two different evaluation strategies: `eager` and `normal-order` evaluation.

2.1 Handling Errors

Before going through the issues about binding a variable to an expression, let's examine some erroneous expression examples we have to deal with.

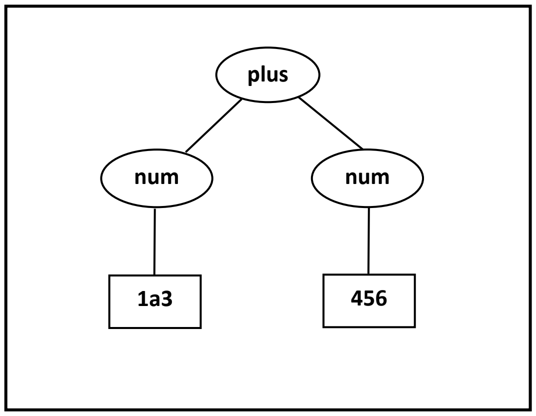

Figure 1: An example of invalid expression

You can see the problem here, right? The AST wants us to cast the value "1a3" as a number, but the second character does not look like a digit. What will we do now? We have to return an error message indicating the cause of the error. We can say that *"the value '1a3' is not a number!"*. The exclamation mark can be too much, but we need to show how disappointed we are. So, let's keep it.

Another problem might occur if we want to add a string to a number. For this example, we use the AST in Figure 2.

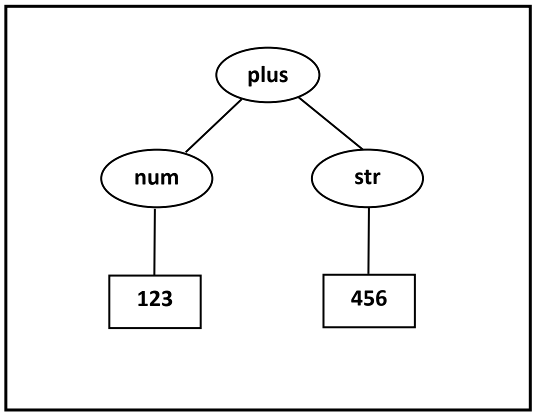

Figure 2: Another example of invalid expression

In this case, the error message must give the information about invalidity of the operand types. Therefore, we will use a message like: *"plus operation is not defined between num and str!"*

Here are the rules regarding the faulty cases in an expression:

- The values which includes one or more non-numeric characters cannot be the operand of the **"num"** operator. Only exception for this case is the leading '-' character for negative numbers.
- We will assume that the string representing the real numbers like 3.14 cannot be referred as Int type for the sake of simplicity.
- There will be no invalid string. Namely, any value in the scope of this assignment, can be the operand of the **"str"** operator.
- You should remember that **"plus"**, **"times"** and**"negate"** operators are applicable only on numbers.
- Similarly, **"cat"** and **"len"** operators are applicable only on strings.
- The error messages of binary operators should be constructed according to which child has invalid type for the operation. For example, *"plus operation is not defined between num and str!"* and *"plus operation is not defined between str and num!"* are different messages which indicate the left and the right causes the error respectively. If both types are not valid for the operator, for example in "plus" operation, the following error message should be returned: *"plus operation is not defined between str and str!"*
- Priority in errors takes place in `post` order. That is, an error which occurs in the left child of an operator has the most priority for messaging. After that we check the right child, if the operator is binary. Finally, we check the type(s) of values coming from operand(s) for being proper to operate or not.

2.2 Binding A Variable in AST & Scoping

Here we introduce the "let" node in the AST which is responsible for mapping a variable to an expression in the overall expression. See the following figure for better understanding:

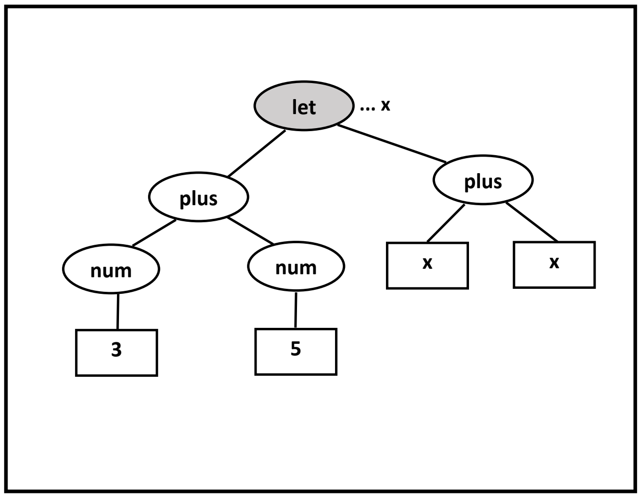

Figure 3: Example of AST with let node

AST in the Figure 3, represents the expression "let x be (3+5) in (x+x)" which is equal to 16 [1]. Now that we introduce the let node, we are able to map the variables to not only values but also expressions.

There is one more advantage of the let node. By the help of this node, we are able to define the scope of the variable. Consider the following AST in Figure 4:

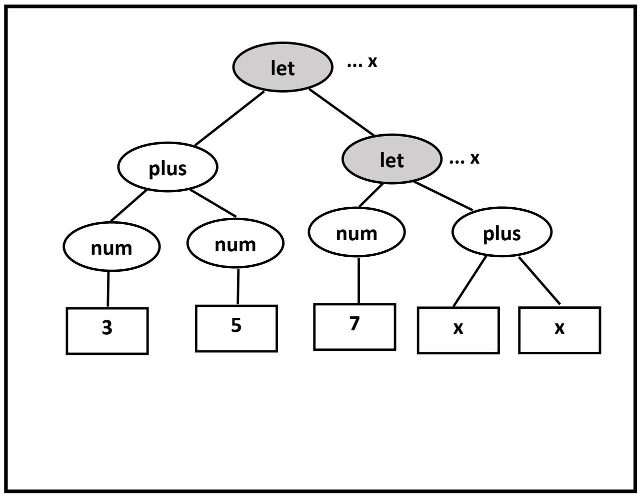

Figure 4: Example of AST with binding of same variable in different scopes

This AST represents the expression "let x be (3+5) in let x be 7 in (x+x)". In this expression same variable name is used in two different bindings. We know that the local variable with the same name overwrites the global one. Hence, the deeper let node overwrites the binding of the other let node, the expression will be inferred as (7 + 7) and therefore, the result will be 14.

[1] Actually, this is not the exact formal representation, but we prefer it for the sake of clarity.

2.3 Eager vs Normal-Order Evaluation

As we map a variable to an expression, there are different approaches about how to substitute this variable in the overall expression. In the scope of this assignment, we will examine the two of them: **eager** and **normal-order**. In the eager approach, we do the calculation for the value of variable before the substitution. In this way, the AST in Figure 3 is represented as below after the substitution:

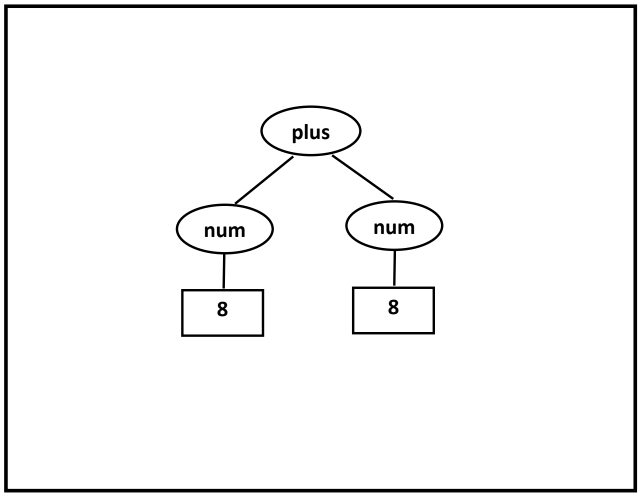

Figure 5: The same AST within the Figure 3, after the substitution with eager approach

On the other hand, in the normal-order evaluation, we directly substitute the variable with the expression itself. By this approach, the same AST will become like in the following figure, after the substitution:

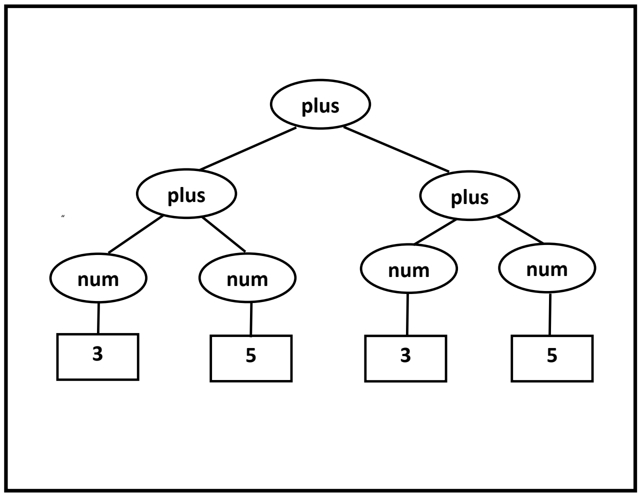

Figure 6: The same AST within the Figure 3, after the substitution with normal-order approach

Although they lead the same result, eager and normal-order evaluations often take different number of steps in the calculation. It may seem that eager evaluation is more efficient since it requires less steps. But, do not jump into the conclusion, that is not the case for all expressions. For example, in the weird expression we have seen earlier, “let x be (3+5) in let x be 7 in (x+x)”, (3+5) is calculated in the eager approach even if we don’t need it. In the normal-order evaluation, we can ignore this step, since the calculation is done after the substitution.

In this assignment, you will implement both approaches and observe the difference between them in terms of the number of steps they take in the calculation of given expression. Here are the rules for this observation:

- Casting operators, “num” and “str” don’t require any step. We will assume that they are applied without any effort.
- Substitution operation does not require any step by itself. We will only count the steps of mapped expression in the substitution.
- The rest of the operators take one step in the calculation.

3 Specifications

Before listing the functions you will implement, we should update the definition of the AST type so that we can deal with the bindings in an AST:

```
data ASTDatum = ASTSimpleDatum String | ASTLetDatum String deriving (Show, Read)

data AST = EmptyAST | ASTNode ASTDatum AST AST deriving (Show, Read)
```

Here, we also introduce the ASTDatum type with two kinds of value constructor. `ASTSimpleDatum` will be used for variables, values and operators in the AST. Second one, `ASTLetDatum` will be used for the indication of bounded variables in the AST. According to new definitions, we represent the AST in Figure 3 as the following (newline characters are added to ease the reading):

```
(ASTNode (ASTLetDatum "x") (ASTNode (ASTSimpleDatum "plus")
(ASTNode (ASTSimpleDatum "num") (ASTNode (ASTSimpleDatum "3") EmptyAST EmptyAST)
EmptyAST) (ASTNode (ASTSimpleDatum "num") (ASTNode (ASTSimpleDatum "5") EmptyAST
EmptyAST) EmptyAST)) (ASTNode (ASTLetDatum "x") (ASTNode (ASTSimpleDatum "num")
(ASTNode (ASTSimpleDatum "7") EmptyAST EmptyAST) EmptyAST)
(ASTNode (ASTSimpleDatum "plus") (ASTNode (ASTSimpleDatum "x") EmptyAST EmptyAST)
(ASTNode (ASTSimpleDatum "x") EmptyAST EmptyAST))))
```

Moreover, we have to define a type for the result of an AST since we aim to handle erroneous cases as well:

```
data ASTResult = ASTError String | ASTJust (String, String, Int) deriving (Show,
    Read)
```

ASTError will carry the error message while ASTJust will show the value of the result, the type of the result and the number representing how many steps it takes to evaluate the AST for the non-erroneous cases.

As we know the types in our homework, we can move on to functions which you are expected to implement.

3.1 isNumber (10 Points)

The first function that you will implement is called `isNumber` and here is the type declaration for this function:

```
isNumber :: String -> Bool
```

It takes a String and returns True or False according to rules for valid numbers given in Section 2.1.

3.2 eagerEvaluation (45 Points)

The second function is called `eagerEvaluation` and has the following signature:

```
eagerEvaluation :: AST -> ASTResult
```

It takes an AST and returns the result of the **eager** evaluation as the type of ASTResult.

3.3 normalEvaluation (45 Points)

The last function you will implement is `normalEvaluation` and it is given with the type declaration below:

```
normalEvaluation :: AST -> ASTResult
```

It takes an AST and returns the result of the **normal-order** evaluation as the type of ASTResult.

4 Sample I/O

```
*Hw2> isNumber "1a3"
False
*Hw2> isNumber "456"
True
*Hw2> isNumber "-123"
True
*Hw2> isNumber "abc"
False
*Hw2> isNumber "123.4"
False
```

```
*Hw2> eagerEvaluation (ASTNode (ASTSimpleDatum "plus") (ASTNode (ASTSimpleDatum "
    num") (ASTNode (ASTSimpleDatum "1a3") EmptyAST EmptyAST) EmptyAST) (ASTNode (
    ASTSimpleDatum "num") (ASTNode (ASTSimpleDatum "456") EmptyAST EmptyAST)
    EmptyAST))
ASTError "the value '1a3' is not a number!"

*Hw2> eagerEvaluation (ASTNode (ASTLetDatum "x") (ASTNode (ASTSimpleDatum "plus") (
    ASTNode (ASTSimpleDatum "num") (ASTNode (ASTSimpleDatum "3") EmptyAST EmptyAST)
    EmptyAST) (ASTNode (ASTSimpleDatum "num") (ASTNode (ASTSimpleDatum "5") EmptyAST
     EmptyAST) EmptyAST)) (ASTNode (ASTSimpleDatum "plus") (ASTNode (ASTSimpleDatum
    "x") EmptyAST EmptyAST) (ASTNode (ASTSimpleDatum "x") EmptyAST EmptyAST)))
ASTJust ("16","num",2)

*Hw2> eagerEvaluation (ASTNode (ASTLetDatum "x") (ASTNode (ASTSimpleDatum "plus") (
    ASTNode (ASTSimpleDatum "num") (ASTNode (ASTSimpleDatum "3") EmptyAST EmptyAST)
    EmptyAST) (ASTNode (ASTSimpleDatum "num") (ASTNode (ASTSimpleDatum "5") EmptyAST
     EmptyAST) EmptyAST)) (ASTNode (ASTLetDatum "x") (ASTNode (ASTSimpleDatum "num")
     (ASTNode (ASTSimpleDatum "7") EmptyAST EmptyAST) EmptyAST) (ASTNode (
    ASTSimpleDatum "plus") (ASTNode (ASTSimpleDatum "x") EmptyAST EmptyAST) (ASTNode
     (ASTSimpleDatum "x") EmptyAST EmptyAST))))
ASTJust ("14","num",2)
```

```
*Hw2> normalEvaluation (ASTNode (ASTSimpleDatum "plus") (ASTNode (ASTSimpleDatum "
    num") (ASTNode (ASTSimpleDatum "123") EmptyAST EmptyAST) EmptyAST) (ASTNode (
    ASTSimpleDatum "str") (ASTNode (ASTSimpleDatum "456") EmptyAST EmptyAST)
    EmptyAST))
ASTError "plus operation is not defined between num and str!"

*Hw2> normalEvaluation (ASTNode (ASTLetDatum "x") (ASTNode (ASTSimpleDatum "plus")
    (ASTNode (ASTSimpleDatum "num") (ASTNode (ASTSimpleDatum "3") EmptyAST EmptyAST)
     EmptyAST) (ASTNode (ASTSimpleDatum "num") (ASTNode (ASTSimpleDatum "5")
    EmptyAST EmptyAST) EmptyAST)) (ASTNode (ASTSimpleDatum "plus") (ASTNode (
    ASTSimpleDatum "x") EmptyAST EmptyAST) (ASTNode (ASTSimpleDatum "x") EmptyAST
    EmptyAST)))
ASTJust ("16","num",3)

*Hw2> normalEvaluation (ASTNode (ASTLetDatum "x") (ASTNode (ASTSimpleDatum "plus")
    (ASTNode (ASTSimpleDatum "num") (ASTNode (ASTSimpleDatum "3") EmptyAST EmptyAST)
     EmptyAST) (ASTNode (ASTSimpleDatum "num") (ASTNode (ASTSimpleDatum "5")
    EmptyAST EmptyAST) EmptyAST)) (ASTNode (ASTLetDatum "x") (ASTNode (
    ASTSimpleDatum "num") (ASTNode (ASTSimpleDatum "7") EmptyAST EmptyAST) EmptyAST)
     (ASTNode (ASTSimpleDatum "plus") (ASTNode (ASTSimpleDatum "x") EmptyAST
    EmptyAST) (ASTNode (ASTSimpleDatum "x") EmptyAST EmptyAST))))
ASTJust ("14","num",1)
```

5 Regulations

1. **Programming Language:** You must code your program in Haskell. Your submission will be tested with `ghc/ghci` on CengClass system. You are expected to make sure your code loads successfully in `ghci` interpreter on the CengClass system.

2. **Implementation:** You have to code your program by only using the functions in Prelude module (the standard module of the Haskell). Namely, you **cannot** import any module in your program.

3. **Late Submission:** You have been given 10 late days in total and at most 3 of them can be used for an assignment. After 3 days you get 0, no excuse will be accepted.

4. **Cheating: We have zero tolerance policy for cheating**. People involved in cheating (any kind of code sharing and codes taken from internet included) will be punished according to the university regulations.

5. **Discussion:** You must follow the CengClass for discussions and possible updates on a daily basis.

6. **Evaluation:** Your program will be evaluated automatically using "black-box" technique so make sure to obey the specifications. Erroneous expressions will be only used in order to test the way that you handle the type checking. Apart from that you don't have to concern about invalid ASTs.

6 Submission

Submission will be done via CengClass system. You can either download the template file, make necessary additions and upload the file to the system or edit using the editor on the CengClass and save your changes.