Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring '2018-2019

Homework 1

Due date: 24 March 2019, Sunday, 23:55

1 Objectives

This assignment aims to assist you to discover the functional programming world by the help of Haskell programming language.

2 Problem Definition

So far, you have only concerned about the syntax of programming languages you code with. They are called `concrete syntax`, since they are formally defined with literals and symbol characters. In this homework, we will think more deeply, regarding what would be the case if we have to define an expression without considering any real programming language. But wait, it has been already thought by cool CS people before and they have come up with the idea of a structure called `Abstract Syntax Tree`. In this homework, we will observe that how easily we can get an expression written in Haskell language and the result of the expression for any given Abstract Syntax Tree.

Let's continue with the definition of Abstract Syntax Tree:

> *"An Abstract Syntax Tree, or AST for short, is an ordered tree whose leaves are variables or values, and whose interior nodes are operators whose arguments are its children."* [1]

It's good that we don't have to define anything from scratch. We put the operators like ``` ``plus'' ``` or ``` ``times'' ``` into the parent nodes of the nodes representing the operands. If we apply these operations in the correct order we get the correct representation of any expression in the form of AST. Leaf nodes will be the variables or values in the expression. We have to give a clue to AST, about the type of the values in an expression. Therefore, we use unary operators like ``` ``num'' ``` or ``` ``str'' ``` representing the numbers and strings respectively.

Consider the basic expression, 333 + 667. To write it as an AST we would need a root node with a ``` ``plus'' ``` operator. Since we work directly with values, we also need child nodes which are called ``` ``num'' ```, indicating that this ``` ``plus'' ``` operation will apply to two numbers. Finally, the leaf nodes of "333" and "667" are inserted as the children of these ``` ``num'' ``` nodes and we get AST as shown in Figure 1.

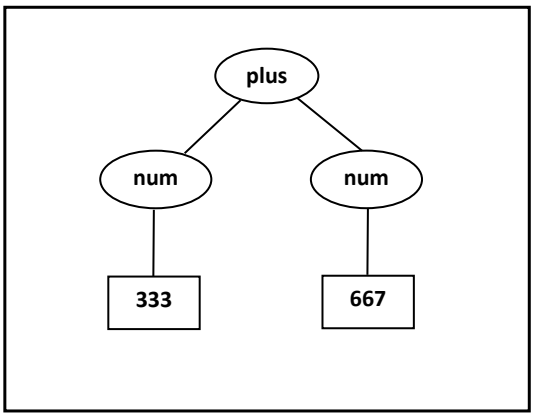

Figure 1: AST for the expression of 333 + 667

Arity of an AST node is not limited by definition but we restrict ourselves in binary case in the scope of this homework. That is, in this homework, we deal with the unary and binary operators only. These operators are defined as following:

- **"plus" :** takes two numbers and applies addition operation.
- **"times" :** takes two numbers and applies multiplication operation.
- **"negate" :** takes a number and reverses its sign.
- **"cat" :** takes two strings and applies concatenation operation.
- **"len" :** takes a string and calculates the length of it.
- **"num" :** takes a value and indicates that it is a number. In the case of Haskell, we will assume that this number is an instance of `Int` data type.
- **"str" :** takes a value and indicates that it is a string.

As you can infer from the operators above we deal with two data types as operands, num (as `Int`) and str (as `String`), for the sake of simplicity.

Here is another expression like ((333+667)*y) which includes "y" as a variable. If we want to show this expression in the AST form we get the tree shown in the Figure 2.

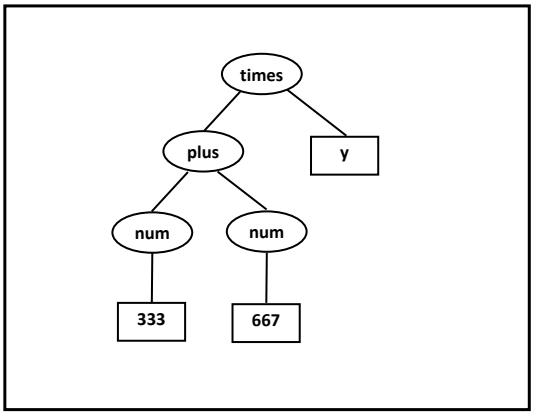

Figure 2: AST for the expression of ((333+667)*y)

As you can see, we don't state the type of a variable in the AST. Instead, we will use an external mapping indicating the types and values of variables in the given AST when we need to calculate the result of the given expression. For example, if we want to calculate this expression for y as 14 and get the value of 14000, we'll state it in the form of 3-tuple, ("y", "num", "14"). Therefore, we'll need a list of 3-tuples to substitute the values of the variables in a given AST to get its result.

For example, if we want to get the result of the AST given below, we will provide a list like [("a", "str", "hello_"), ("b", "str", "world!")] and we get the string "hello_world!".

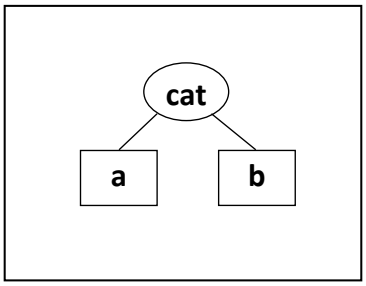

Figure 3: AST for the expression of (a++b)

Here is a final example for the expression, (-(length ("C" ++ "Eng"))) covering the rest of the operators. We will see the details of our program in the next section.

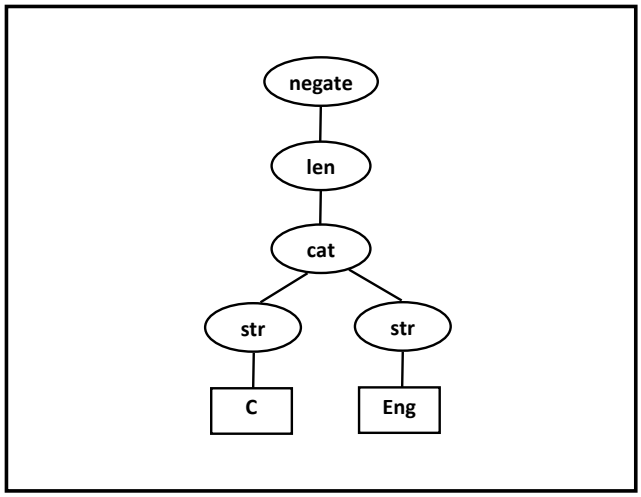

Figure 4: AST for the expression (-(length ("C" ++ "Eng")))

3 Specifications

Before listing the functions you will implement, let's examine the AST type we define as below:

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

It is defined like any other binary tree data type in Haskell. It derives Show and Read typeclasses to see its string representation and to cast the type from its string representation respectively.
According to this type, we represent the AST in the Figure 1 as:

```
(ASTNode "plus" (ASTNode "num" (ASTNode "333" EmptyAST EmptyAST) EmptyAST) (ASTNode
    "num" (ASTNode "667" EmptyAST EmptyAST) EmptyAST))
```

To deal with the list of 3-tuples stating the values of variables in AST, we defined an alias like the following:

```
type Mapping = [(String, String, String)]
```

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

3.1 writeExpression (40 Points)

First function which converts the expression represented in AST to Haskell expression is called `writeExpression` and has a signature like this:

```
writeExpression :: (AST, Mapping) -> String
```

Namely, it takes a tuple consisting of an AST and a Mapping and returns a String as a result.

You will use parenthesis for any operation in AST to avoid ambiguity and here are the rules for representing the expression in Haskell:

- You will use '+' for `plus`
- You will use '*' for `times`
- You will use '-' for `negate`
- You will use '++' for `cat`
- You will use 'length' function for `len`
- You can write the numbers in AST directly.
- You will use surrounding ' \" ' to represent a string.
- If the AST includes variables, the expression will start with `let` keyword. After listing all variables and their mapping in the form of <variable>=<value> with ';' between them, you will write `in` keyword and finally the expression itself. (If the AST includes only one variable, ';' will not be used.)

See the Section 4 to understand the rules better. Make sure that you obey these rules and do not try to beautify it in order not to lose any points in black-box testing.

3.2 evaluateAST (60 Points)

Second and last function you are going to implement is called as `evaluateAST`. It takes a tuple consisting of an AST and a Mapping and returns a tuple of the AST after the substitution of variables and a String as the result of the evaluation. You can see the type declaration of this function below:

```
evaluateAST :: (AST, Mapping) -> (AST, String)
```

For the evaluation of the given AST, you have to substitute its variables with their type indicated with "num" and "str" operators. Note that same variable may occur in different nodes of the AST. We will assume that all of them will take the same value in the substitution. After the substitution, you can evaluate the AST according to the operators defined in Section 2.

We will assume that unary operators have operand only in the left child and its right child will always be EmptyAST during the evaluation.

In the next section, you will see the sample inputs and outputs for these two functions. They may not cover all of the cases but they can ease your understanding about how they work.

4 Sample I/O

```
*Hw1> writeExpression ((ASTNode "plus" (ASTNode "num" (ASTNode "333" EmptyAST
    EmptyAST) EmptyAST) (ASTNode "num" (ASTNode "667" EmptyAST EmptyAST) EmptyAST)),
     [])
"(333+667)"

*Hw1> writeExpression ((ASTNode "times" (ASTNode "plus" (ASTNode "num" (ASTNode "
    333" EmptyAST EmptyAST) EmptyAST) (ASTNode "num" (ASTNode "667" EmptyAST
    EmptyAST) EmptyAST)) (ASTNode "y" EmptyAST EmptyAST)), [("y", "num", "14")])
"let y=14 in ((333+667)*y)"

*Hw1> writeExpression ((ASTNode "cat" (ASTNode "a" EmptyAST EmptyAST) (ASTNode "b"
    EmptyAST EmptyAST)), [("a", "str", "hello_"), ("b", "str", "world!")])
"let a=\"hello_\";b=\"world!\" in (a++b)"

*Hw1> writeExpression ((ASTNode "negate" ((ASTNode "len" (ASTNode "cat" (ASTNode "
    str" (ASTNode "C" EmptyAST EmptyAST) EmptyAST) (ASTNode "str" (ASTNode "Eng"
    EmptyAST EmptyAST) EmptyAST)) EmptyAST)) EmptyAST), [])
"(-(length (\"C\"++\"Eng\")))"
```

```
*Hw1> evaluateAST ((ASTNode "plus" (ASTNode "num" (ASTNode "333" EmptyAST EmptyAST)
     EmptyAST) (ASTNode "num" (ASTNode "667" EmptyAST EmptyAST) EmptyAST)), [])
(ASTNode "plus" (ASTNode "num" (ASTNode "333" EmptyAST EmptyAST) EmptyAST) (ASTNode
     "num" (ASTNode "667" EmptyAST EmptyAST) EmptyAST),"1000")

*Hw1> evaluateAST ((ASTNode "times" (ASTNode "plus" (ASTNode "num" (ASTNode "333"
    EmptyAST EmptyAST) EmptyAST) (ASTNode "num" (ASTNode "667" EmptyAST EmptyAST)
    EmptyAST)) (ASTNode "y" EmptyAST EmptyAST)), [("y", "num", "14")])
(ASTNode "times" (ASTNode "plus" (ASTNode "num" (ASTNode "333" EmptyAST EmptyAST)
    EmptyAST) (ASTNode "num" (ASTNode "667" EmptyAST EmptyAST) EmptyAST)) (ASTNode "
    num" (ASTNode "14" EmptyAST EmptyAST) EmptyAST),"14000")

*Hw1> evaluateAST ((ASTNode "cat" (ASTNode "a" EmptyAST EmptyAST) (ASTNode "b"
    EmptyAST EmptyAST)), [("a", "str", "hello_"), ("b", "str", "world!")])
(ASTNode "cat" (ASTNode "str" (ASTNode "hello_" EmptyAST EmptyAST) EmptyAST) (
    ASTNode "str" (ASTNode "world!" EmptyAST EmptyAST) EmptyAST),"hello_world!")

*Hw1> evaluateAST ((ASTNode "negate" ((ASTNode "len" (ASTNode "cat" (ASTNode "str"
    (ASTNode "C" EmptyAST EmptyAST) EmptyAST) (ASTNode "str" (ASTNode "Eng" EmptyAST
     EmptyAST) EmptyAST)) EmptyAST)) EmptyAST), [])
(ASTNode "negate" (ASTNode "len" (ASTNode "cat" (ASTNode "str" (ASTNode "C"
    EmptyAST EmptyAST) EmptyAST) (ASTNode "str" (ASTNode "Eng" EmptyAST EmptyAST)
    EmptyAST)) EmptyAST) EmptyAST,"-4")
```

5 Evaluating the Expression in Haskell

If you wish to evaluate the expression you write via `writeExpression` function on your own computer, you can use the Haskell interpreter by installing an additional package called "hint". Before reading the rest of this section, you should know that this part will **not** be graded, but it can be a way to prove that our expressions are valid.

To install that package, the steps you should follow is given below.

```
huseyin@huseyin:~$ sudo apt install cabal-install
...
huseyin@huseyin:~$ cabal install hint
```

After installing the package, you should run ghci and import "Language.Haskell.Interpreter" module and call the runInterpreter function like shown below.

```
huseyin@huseyin:~$ ghci
...
> import Language.Haskell.Interpreter
> runInterpreter $ setImports["Prelude"] >> eval "(333+667)"
Right "1000"

> runInterpreter $ setImports["Prelude"] >> eval "let y=14 in ((333+667)*y)"
Right "14000"

> runInterpreter $ setImports["Prelude"] >> eval "let a=\"hello_\";b=\"world!\" in
    (a++b)"
Right "\"hello_world!\""

> runInterpreter $ setImports["Prelude"] >> eval "(-(length (\"C\"++\"Eng\")))"
Right "-4"
```

You can see from the examples above, the results of evaluations are in the form of `Right <result>` which indicates the evaluation is done without any error.

6 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. **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.

3. **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.

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

5. **Evaluation:** Your program will be evaluated automatically using "black-box" technique so make sure to obey the specifications. No erroneous input will be used. Therefore, you don't have to consider the invalid expressions.

7 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.

References

[1] R. Harper, *Practical foundations for programming languages.* Cambridge University Press, 2016.