Middle East Technical University

Department of Computer Engineering

CENG 242

Programming Language Concepts
2023-2024 Spring
Programming Exam 3

Due: 21 April 2024 11:59
Submission: **via ODTUClass**

1 General Specifications

- There are five questions in this Programming Exam, for each question you will either implement a given Haskell function or a given Haskell type class.
- Definitions of the data types, signatures of the functions, their explanations and specifications are given in the following section. Read them carefully.
- Make sure that your implementations comply with the function signatures.
- You may define helper function(s) as needed.

2 Functions

2.1 Shunting-yard Algorithm

This function converts mathematical expressions from infix notation to postfix notation. It achieves this transformation by taking three arguments, all of which are lists of strings:

- `expression`: This list represents the infix expression you want to convert. It contains tokens like variable names, numbers, operators, and parenthesis.
- `stack`: This list acts as a temporary storage for operators during the conversion process. Initially, it might be empty.
- `queue`: This list ultimately holds the expression in postfix notation. Initially, it might be empty as well.

In essence, shuntingyard takes an infix expression, analyzes the order of operations using the operator precedence rules, and rearranges the tokens into a postfix format, suitable for efficient evaluation in Haskell. Here is it's definition in Haskell:

```
shuntingyard :: [String] -> [String] -> [String] -> [String]
```

Here is its sample execution of the `shuntingyard` function as below:

```
Prelude> shuntingyard ["5", "+", "(", "4", "*", "x", ")"] [] []
["5", "4", "3", "*", "+"]
```

In detail, here is a psedeucode of Shunting-yard algorithm

- The algorithm takes a list of tokens (numbers, variables, operators, parentheses) as input.
- It initializes two data structures:
  - `output_queue`: A queue to store the postfix expression.
  - `operator_stack`: A stack to store operators temporarily.
- It iterates through each token in the input list.
- If the token is an operand (number or variable), it's directly added to the output queue.
- If the token is an opening parenthesis (, it's pushed onto the operator stack.
- If the token is a closing parenthesis), all operators on the stack with higher or equal precedence are popped and added to the output queue until an opening parenthesis (is encountered. The opening parenthesis is then popped and discarded.
- If the token is an operator, a loop iterates through the operator stack while the top operator has higher or equal precedence compared to the current operator. These higher precedence operators are popped and added to the output queue. Finally, the current operator is pushed onto the stack.
- Once all tokens are processed, any remaining operators on the stack are popped and added to the output queue. The function returns the final postfix expression stored in the output queue.

see Graphical illustration for details.

Specifications :

- you can consider `expression` as an expression string, like "3 + 5 * x", parsed into its tokens.
- A token might be
  - a parenthesis (either opening or closing),

  - an operator (unary minus (-), sine (sin), cosine (cos), addition (+), mult. (*)),
  - an operand (constant if is string representing a float number, variable otherwise).

- the operator precedence is as below:
  - unary minus (-)
  - sine and cosine trigonometric functions
  - multiplication
  - addition
  - opening and closing parenthesis

Note that: unary minus has the highest precedence, and parentheses has the lowest precedence.

3 Data Types

The provided Haskell code defines a data type named `Expression`. This data type allows you to represent various mathematical polynomials in your program. It includes options for variables (represented by strings), constants (floating-point numbers), trigonometric functions (sine and cosine applied to other expressions), negation, addition, and multiplication. Essentially, this code equips you with the building blocks to construct and manipulate mathematical expressions within your Haskell code. Its definition is as follows:

```
data Expression = Variable String
    | Constant Float
    | Sine Expression
    | Cosine Expression
    | Negation Expression
    | Addition Expression Expression
    | Multiplication Expression Expression
```

Here is the construction of the expression "5 + (4 * x)" with `Expression` data type:

```
Addition (Constant 5.0) (Multiplication (Constant 4.0) (Variable "x"))
```

Expressions can be considered as inductive (recursive) data type and it's components (operators) can grouped by their sub-expression (operand) numbers. That is:

- nullary operators (`Variable`, `Constant`),
- unary operators (`Negation`, `Sine`, `Cosine`),
- binary operators (`Addition`, `Multiplication`)

In the following sections, you will implement some features for those operator collections.

4 Type Classes

Haskell leverages type classes for shared functionalities across data types. This section explores three built-in type classes: `Show`, `Eq`, and `Num` as well as a custom type class `Differential`.

4.1 Show

This type class enables converting data types into readable strings for printing or displaying values. Common types like `Int` and `String` have built-in `Show` instances. For custom data types (like `Expression`), you need to define a `Show` instance specifying how to convert each constructor (e.g., `Variable`, `Constant`) into a string.

Here is how to instantiate `Show` type class for value constructors of `Expression`:

- Those taking no operand (like Variable and Constant) must be represented together with their value constructor name, i.e. `Variable 'x'` or `Constant 3.0`.
- Unary operators use their mathematical symbol like `-Constant 3.0`, or `sin Constant 3.0`.
- Binary operators encapsulates within parentheses along with their operands e.g. `(Constant 5.0 + Constant 3.0)`

Note that: Negation has no white-space before its operand. Operators but Negation has single white-spaces. Only binary operators use paranthesis. Only Nullary operators use their value constructor name.

4.2 Eq

The `Eq` typeclass enables comparison of values for equality. Common types like Int and String have built-in `Eq` instances. For custom data types (like `Expression`), you need to define an `Eq` instance specifying how to compare values of each constructor (e.g., Variable, Constant) for equality. This allows you to use comparison operators like == and /= with your custom data types.

Here is how to instantiate `Eq` type class for value constructors of `Expression`:

- Nullary operators of the same value constructor (i.e. Constant a vs. Constant b) are equal to each other if they have the same values (i.e. a = b).
- Unary operators of the same value constructor are equal to each other if they have the same operands inductively.
- Binary operators of the same value constructor are equal to each other if their left-hand-side operands are the same and their right-hand-side operands are the same inductively.

Note that: Don't check abelianity (commutativity) of binary operators.

4.3 Num

The `Num` type class provides functionalities for numeric operations on data types. Built-in types like `Int` and `Float` already have `Num` instances defined.

For custom data types (like `Expression`), defining a `Num` instance allows you to construct your data from built-in operations (in our case, it corresponds to arithmetic operations on expressions). These operations might include:

- Negation (unary minus)
- Addition (+)
- Multiplication (*)

Additionally, using polymorphism of constructors, constant expressions might be simplified.

General cases Here is how to instantiate `Eq` type class for value constructors of `Expression`:

- `fromInteger`: constructs a `Constant` from a Float
- `negate`: Constructs a `Negation` from an `Expression`
- `(+)`: Constructs an `Addition` from two `Expression`
- `(*)`: Constructs an `Multiplication` from two `Expression`

Special Cases Here are some additional considerations of constructors of the type class `Num`, especially with value constructors `Constant`s.

- **Negation**: The `negate` function (if defined for the Num type class) negates the value of a Constant when applied.
- **Addition by zero**: Addition (`(+)`) exhibits a simplification behavior. When one operand is a Constant 0, the expression effectively becomes the other operand. (This aligns with the concept of adding zero to any number.)
- **Multiplication by one**: Multiplication (`(*)`) demonstrates similar simplification. If one operand is a Constant 1, the expression reduces to the other operand. (This reflects multiplying by one.)
- **Multiplication by Zero**: A special case exists for multiplication ((*)). If one operand is a Constant 0, the entire expression is considered equal to Constant 0.0 due to the multiplicative property of zero.

These additional considerations highlight potential optimizations or simplifications that might be implemented within the `Num` type class definition for the `Expression` data type.

4.4 Differential

The `Differential` is a custom type class providing functionalities for calculating the derivative of expressions with respect to a variable. This allows symbolic differentiation within your Haskell programs. Its definition is as follows:

```
class Differential a where
    diff :: a -> a -> a
```

The `diff` function defined in the `Differential` type class takes two arguments of the same type (a). The first argument represents the expression for which we want to find the derivative. The second argument specifies the variable with respect to which we want to differentiate the expression. For simplicity the second argument will be always a `Variable`. The function itself returns a new expression representing the derivative.

For custom data types (like `Expression`), defining a `Differential` instance specifies how to differentiate each constructor of the data type with respect to another variable. The provided instance showcases differentiation for the `Expression` data type defined earlier.

Differentiation Rules The provided instance defines differentiation rules for various constructors:

- **Constant**: The derivative of a constant value (Constant c) with respect to any variable is always zero (Constant 0).
- **Variable**: The derivative of a variable (Variable v) with respect to itself (Variable dv) is one (Constant 1). Otherwise, the derivative is zero (Constant 0).
- **Sine**: The derivative of the sine of an expression (Sine expr) with respect to a variable (Variable dv) is the cosine of the expression multiplied by the derivative of the original expression.
- **Cosine**: The derivative of the cosine of an expression (Cosine expr) with respect to a variable (Variable dv) is the negative sine of the expression multiplied by the derivative of the original expression.
- **Negation**: The derivative of the negation of an expression (Negation expr) with respect to a variable (Variable dv) is simply the negation of the derivative of the original expression.
- **Addition**: The derivative of the sum of two expressions (Addition lhs rhs) with respect to a variable (Variable dv) is the sum of the derivatives of the individual expressions.
- **Multiplication**: The derivative of the product of two expressions (Multiplication lhs rhs) with respect to a variable (Variable dv) involves the sum of two terms:
  - The derivative of the first expression multiplied by the second expression.
  - The first expression multiplied by the derivative of the second expression.

Note that: You must **not** apply special cases of the type class `Num`. That is, you should **not** simplify equations.

5 Regulations

1. **Implementation and Submission:** The template file named "PE3.hs" is available in the Virtual Programming Lab (VPL) activity called "PE3" on OdtuClass. At this point, you have two options:

   - You can download the template file, complete the implementation and test it with the given sample I/O (and also your own test cases) on your local machine. Then submit the same file through this activity.
   - You can directly use the editor of VPL environment by using the auto-evaluation feature of this activity interactively. Saving the code is equivalent to submit a file.

   If you work on your own machine, make sure that your implementation can be compiled and tested in the VPL environment after you submit it.
   There is no limitation on online trials or submitted files through OdtuClass. The last one you submitted will be graded.

2. **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.

3. **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.

- **Important Note:** The given sample I/O's are only to ease your debugging process and NOT official. Furthermore, it is not guaranteed that they cover all the cases of required functions. As a programmer, it is your responsibility to consider such extreme cases for the functions. Your implementations will be evaluated by the official test cases to determine your ***actual*** grade after the deadline.