Middle East Technical University | Department of Computer Engineering

# CENG 242

Programming Language Concepts

Spring '2019-2020

Homework 2

Due date: 12 April 2020, Sunday, 23:55

## 1 Objectives

This assignment aims to greatly improve your functional programming skills by posing a variety of challenges well adapted to the functional style, which benefit from the use of what you have learned so far about the Haskell language.

## 2 Scenario

### 2.1 Introduction

You are a computer engineer in a company where you and your colleagues are working on developing a tiny embedded processor specialized for simple arithmetic [1]. This processor can only perform addition, multiplication and negation. This processor has its own set of instructions, and you want to implement a small expression language on top which you can easily compile into the processor's instruction set. The expressions that you aim to work with are fairly simple, only including the given operations and integral constants along with strings for representing variables, one example could be `a*b+(2+-7)`.

You, personally, will be working on optimizing this compiler to minimize the number of operations done when computing an expression. Being experienced software engineers, you know that compiler optimization usually involves lots of transformations on the intermediate form of the compiled code and that the parsing of small languages can be nicely implemented using parser combinators. This makes your problem an ideal use case for functional programming languages. As an avid fan of the features Haskell provides, you decide to implement your prototype in Haskell!

### 2.2 Expression Tree

The first step is coming up with an internal representation for these operations. You decide along with your team to use a representation where leaves are the values on which the operations are applied, and you have two types of internal nodes: one for unary operations and another for binary operations. Since this structure can represent such operations on arbitrary types, you decide to make it polymorphic. Here is the basic definition in code:

[1] I received some complaints about the lack of scenario in the first homework, so here's a homework with (some kind of) scenario!

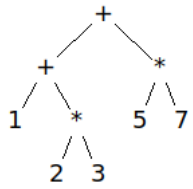
```haskell
data UnaryOperator = Minus
data BinaryOperator = Plus | Times

data Expression a = Leaf a
                  | UnaryOperation UnaryOperator (Expression a)
                  | BinaryOperation BinaryOperator (Expression a) (Expression a)
```

One question that might come to mind, because addition and multiplication can be done on an arbitrary number of operands, is "Why use a binary tree rather than n-ary?". This is because the binary structure can easily be mapped into instructions for your embedded processor, due to instructions usually working on two operands. An example of how that could work (names starting with R are registers) can be seen in Figure 1. [2]:

1+2*3+(5*7)

(a) Expression




(b) Tree Form

```
mov 1, R0
mov 2, R1
mul 3, R1
add R1, R0
mov 5, R2
mul 7, R2
add R2, R0
```

(c) Processor Instructions

Figure 1: Expression Compilation

Now that the logic behind this decision is out of the way, the next step is defining the leaves of the tree we will work with. Since we defined our values as either integral constants or variables, we define our leaf data type and an alias for the types of expressions we will be working with:

```haskell
data Value = Variable String | Constant Int
type ExprV = Expression Value
```

You can find these definitions in the `Expression.hs` file.

## 2.3 Parsing

Directly constructing expression trees is no fun at all, so you need a way to parse expression trees from text inputs. Thankfully, one of your buddies who is working on the parsing part of the language[3] quickly wrote a very basic expression parser for you using the shunting yard algorithm. The functionality is found in the `Parser` module (which is supported by the `Tokenizer` module) in the form of a single function, `parse`:

[2] These are not real assembly instructions, although they are similar to x86-64, but I hope you get the point!

[3] "Do you know him?", you say? Well, of course I know him. He's me!

```
ghci> :l Parser
[1 of 3] Compiling Expression       ( Expression.hs, interpreted )
[2 of 3] Compiling Tokenizer        ( Tokenizer.hs, interpreted )
[3 of 3] Compiling Parser           ( Parser.hs, interpreted )
Ok, three modules loaded.
ghci> parse "3"
Leaf (Constant 3)
ghci> parse "3+x"
BinaryOperation Plus (Leaf (Constant 3)) (Leaf (Variable "x"))
ghci> parse "3+5*2+-var"
BinaryOperation Plus (BinaryOperation Plus (Leaf (Constant 3)) (BinaryOperation Ti
mes (Leaf (Constant 5)) (Leaf (Constant 2)))) (UnaryOperation Minus (Leaf (Variabl
e "var")))
ghci> parse "(7+8)*(x+12)"
BinaryOperation Times (BinaryOperation Plus (Leaf (Constant 7)) (Leaf (Constant 8)
)) (BinaryOperation Plus (Leaf (Variable "x")) (Leaf (Constant 12)))
```

Now that this problem is out of the way and you can easily construct expression trees, it is time to move on with your job!

# 3 Functions to Implement

## 3.1 foldAndPropagateConstants - 20 points

Your first target is optimizing sequences of assignments. Given a sequence of `(String, ExprV)` pairs in a list, you want to evaluate constant subexpressions as much as possible, and if one of the variables evaluates to a constant, substitute the appearance of that variable in later assignments with a constant. If a variable that has not been defined previously appears, leave it as is (you can think that those are values obtained during runtime via device I/O). The same applies to variables that have previously been defined but could not be fully folded due to containing an undefined variable. Here are a few examples:

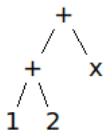
```
foldAndPropagateConstants :: [(String, ExprV)] -> [(String, ExprV)]
-- assignments -> assignments folded with substitutions if possible
```

```
ghci> foldAndPropagateConstants [("x", parse "1+2+3")]
[("x",Leaf (Constant 6))]
ghci> foldAndPropagateConstants [("x", parse "1+2+3"), ("y", parse "5*x + 7")]
[("x",Leaf (Constant 6)),("y",Leaf (Constant 37))]
ghci> foldAndPropagateConstants [("x", parse "1+2+3"), ("y", parse "5*x + 7"), ("z
", parse "y+var")]
[("x",Leaf (Constant 6)),("y",Leaf (Constant 37)),("z",BinaryOperation Plus (Leaf
(Constant 37)) (Leaf (Variable "var")))]
ghci> foldAndPropagateConstants [("a", parse "x+3*5"), ("b", parse "(7+5)*a")]
[("a",BinaryOperation Plus (Leaf (Variable "x")) (Leaf (Constant 15))),("b",Binary
Operation Times (Leaf (Constant 12)) (Leaf (Variable "a")))]
```

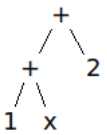
Note that you should evaluate subtrees to constants if and only if the whole subtree is composed of constants. For example, the following three expressions are equivalent, but only the first tree structure

will have some constant folding applied (this is also illustrated in figure 2):

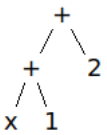
```
ghci> foldAndPropagateConstants [("a", parse "1+2+x")]
[("a",BinaryOperation Plus (Leaf (Constant 3)) (Leaf (Variable "x")))]
ghci> foldAndPropagateConstants [("a", parse "1+x+2")]
[("a",BinaryOperation Plus (BinaryOperation Plus (Leaf (Constant 1)) (Leaf (Variab
le "x"))) (Leaf (Constant 2)))]
ghci> foldAndPropagateConstants [("a", parse "x+1+2")]
[("a",BinaryOperation Plus (BinaryOperation Plus (Leaf (Variable "x")) (Leaf (Cons
tant 1))) (Leaf (Constant 2)))]
```




(a) 1+2 can be folded to 3




(b) No folding




(c) No folding

Figure 2: Algebraically Equivalent Expressions and Folding Behavior

The reason for this is to ease your implementation, you do not have to play with the expression by using commutativity/reassociating etc. Only fold subexpressions if they are composed entirely of constants (or variables that can be substituted by constants).

In summary:

- Go through each assignment in the list order (left to right).
- Evaluate constant subexpressions in the right hand side of the assignment. This may reduce the variable to a single constant in case it does not contain any unknown variables. While evaluating, you should apply both the binary operators (`+`, `*`) and the unary operators (`-`).
- If the expression contains variables that have previously been reduced to constants, substitute them with that constant.
- If the expression contains variables that are either undefined or could not be fully reduced, leave those variables as they are. You should still attempt to fold as much of the expression tree as possible.

To make the problem a bit easier, the same variable will not be assigned to more than once; i.e. the same LHS will not reappear further down the list as the LHS of another assignment. Consequently, the RHS of an assignment will not contain the LHS, e.g. `("a", parse "a+5")` is not valid.

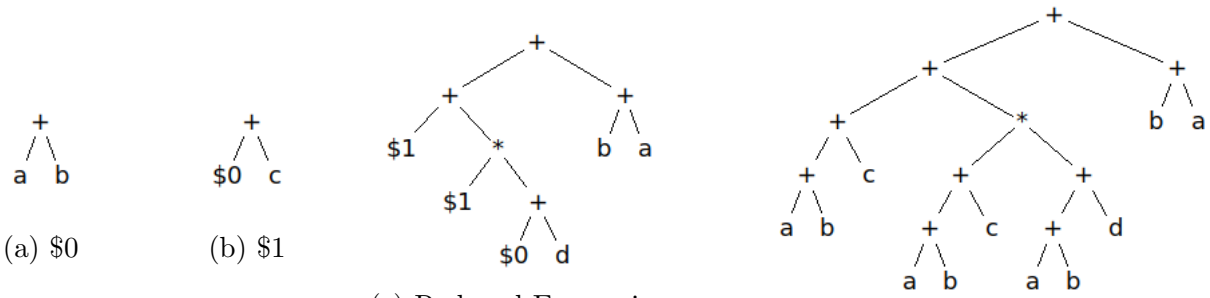
## 3.2 assignCommonSubexprs - 30 points

The next step is the elimination of common subexpressions. The idea is this: If a subexpression is repeated multiple times in an expression, it should be replaced by a variable to reduce the number of operations performed. Thus, given an expression, your aim is to find repeating subexpressions and to substitute them with a new variable; returning a sequence of variables which can be substituted in left-to-right manner to re-obtain the original expression.

To reduce the complexity of the problem, consider the fact that a common subexpression of any height can be eliminated by substituting each operation on leaves with its own variable in an iterative manner. e.g. if you have an expression like `(1+2+3)+(1+2+3)`, you can first substitute `v0=1+2` to obtain `(v0+3)+(v0+3)`, and then substitute `v1=v0+3` to obtain `v1+v1`. This is in contrast to substituting the whole subexpression using `v0=1+2+3` to obtain `v0+v0`. Beware that some expressions might not contain repeated subexpressions. Below are a few examples (another example is shown in 3):

```
assignCommonSubexprs :: ExprV -> ([(String, ExprV)], ExprV)
-- mainExpr -> (common subexpr. variables, reducedMainExpr)
```

```
ghci> assignCommonSubexprs $ parse "1+2"
([],BinaryOperation Plus (Leaf (Constant 1)) (Leaf (Constant 2)))
ghci> assignCommonSubexprs $ parse "(x+1)+(x+1)"
([("$0",BinaryOperation Plus (Leaf (Variable "x")) (Leaf (Constant 1)))],BinaryOpe
ration Plus (Leaf (Variable "$0")) (Leaf (Variable "$0")))
ghci> assignCommonSubexprs $ parse "(-x+1)+(-x+1)"
([("$0",UnaryOperation Minus (Leaf (Variable "x"))),("$1",BinaryOperation Plus (Le
af (Variable "$0")) (Leaf (Constant 1)))],BinaryOperation Plus (Leaf (Variable "$1
")) (Leaf (Variable "$1")))
ghci> assignCommonSubexprs $ parse "(x+1+2)+(x+1+2)"
([("$0",BinaryOperation Plus (Leaf (Variable "x")) (Leaf (Constant 1))),("$1",Bina
ryOperation Plus (Leaf (Variable "$0")) (Leaf (Constant 2)))],BinaryOperation Plus
 (Leaf (Variable "$1")) (Leaf (Variable "$1")))
ghci> assignCommonSubexprs $ parse "(x+1+2)+(x+1+2)+(x+1)"
([("$0",BinaryOperation Plus (Leaf (Variable "x")) (Leaf (Constant 1))),("$1",Bina
ryOperation Plus (Leaf (Variable "$0")) (Leaf (Constant 2)))],BinaryOperation Plus
 (BinaryOperation Plus (Leaf (Variable "$1")) (Leaf (Variable "$1"))) (Leaf (Varia
ble "$0")))
```




(a) $0 (b) $1 (c) Reduced Expression (d) Initial Expression

Figure 3: A Common Subexpression Elimination Example

In summary:

- Find operations on leaves (subexpressions of height 2) that repeat more than once
- Create a new variable for each repeating operation, and subtitute and the operation with that variable.
- Repeat the above steps until you cannot find any more repeating subexpressions.

- Ensure that your assignments can be substituted left to right. e.g. if some variable `$0` appears in `$1`, then `$0` should occur before `$1` in the variable list.

You do not have to detect equivalent subexpressions such as `a+b` and `b+a`. Only detect subexpressions that are exactly the same. Also, you can name your variables in any way you want, however their name should start with a non-alphanumeric character so that they do not clash with variables that might already exist in an expression, and obviously the name should not repeat in other assignments; each one has to be distinct. What matters is that you capture all the subexpressions correctly and that assignments can be substituted left to right using your given names. Also, do not simplify the expression tree any way other than substituting variables such as folding constants (as in the first function), removing additions with zero etc.

## 3.3 reducePoly - 50 points

Finally, you observe that since the only operations you have available are addition, multiplication and negation, any single-variable expression is actually a polynomial! Now, your goal is to simplify any expression of a single variable, say $x$, into polynomial form $a_0 + a_1x + a_2x^2 + ... + a_nx^n$, which definitely simplifies the evaluation of the arbitrarily complicated input expression. [4]

The final polynomial that you will construct will be equivalent to parsing `a0+a1*x+a2*x*x ...`. Obviously, factors with zero coefficients should not appear. Also, you need to perform two basic simplifications. The first one is that unit (1) coefficients should not appear in the reconstructed expression, and the second one is that you should replace a multiplication by (-1) with a unary minus on the leftmost $x$. e.g. $-x^3$ is equivalent to parsing `(-x)*x*x`, rather than `-1*x*x*x`. Once again, a few examples follow (along with a more complicated visual example, in figure 4):

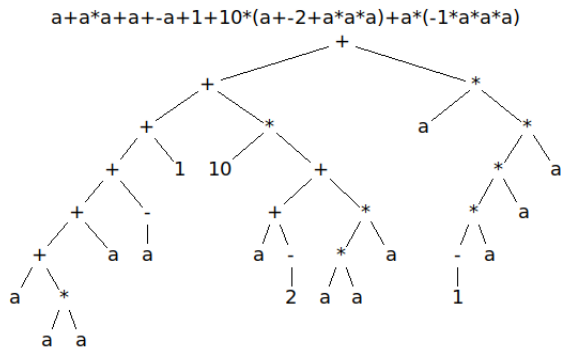
```
reducePoly :: ExprV -> ExprV
-- single variable expression -> reduced to polynomial
```

```
ghci> reducePoly $ parse "2+3"
Leaf (Constant 5)
ghci> reducePoly $ parse "2+x+x+x+-x"
BinaryOperation Plus (Leaf (Constant 2)) (BinaryOperation Times (Leaf (Constant 2)
) (Leaf (Variable "x")))
ghci> reducePoly $ parse "1+a*(a+1)"
BinaryOperation Plus (BinaryOperation Plus (Leaf (Constant 1)) (Leaf (Variable "a"
))) (BinaryOperation Times (Leaf (Variable "a")) (Leaf (Variable "a")))
ghci> reducePoly $ parse "(2*x+2)*(2*x+2)"
BinaryOperation Plus (BinaryOperation Plus (Leaf (Constant 4)) (BinaryOperation Ti
mes (Leaf (Constant 8)) (Leaf (Variable "x")))) (BinaryOperation Times (BinaryOper
ation Times (Leaf (Constant 4)) (Leaf (Variable "x"))) (Leaf (Variable "x")))
```

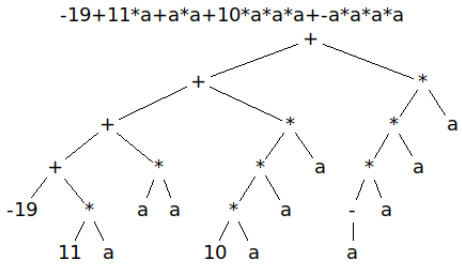
There will be no inputs containing multiple variables. However do note that you need to construct the polynomial with the name of the variable contained in the input. If the expression contains variable `var`, reconstruct your polynomial with `var`; if it's `y`, use `y` etc. It is possible for an input to contain no variables, in which case it will simply evaluate to a constant.

Unlike the parser outputs, negative coefficients should be negative literals, and not positive literals with the unary minus applied to them.

[4] The reconstructed tree is not optimal! The best method to evaluate a polynomial would be to use Horner's method, but we will not implement it in the scope of this homework.




(a) A Single Variable Expression




(b) After Reduction

Figure 4: A Polynomial Reduction Example

# 4 GUI

Do you know what's almost as bad as trying to create tree structures in text? That's right: trying to visualize trees in your mind by looking at their textual representation. Oof! Thankfully, your buddy working on the parsing stuff [3] did you a solid [5] and implemented a functional but crude graphical interface which you can use to visualize your trees. You can find a small user manual in the accompanying PDF `hw2-gui-man.pdf`, which contains both installation instructions and an explanation of the GUI's features. This is definitely not necessary for you to complete the homework, even less so than the parser, but it's a nice thing to have for certain! [6] Here's a picture from the GUI (in figure 5):

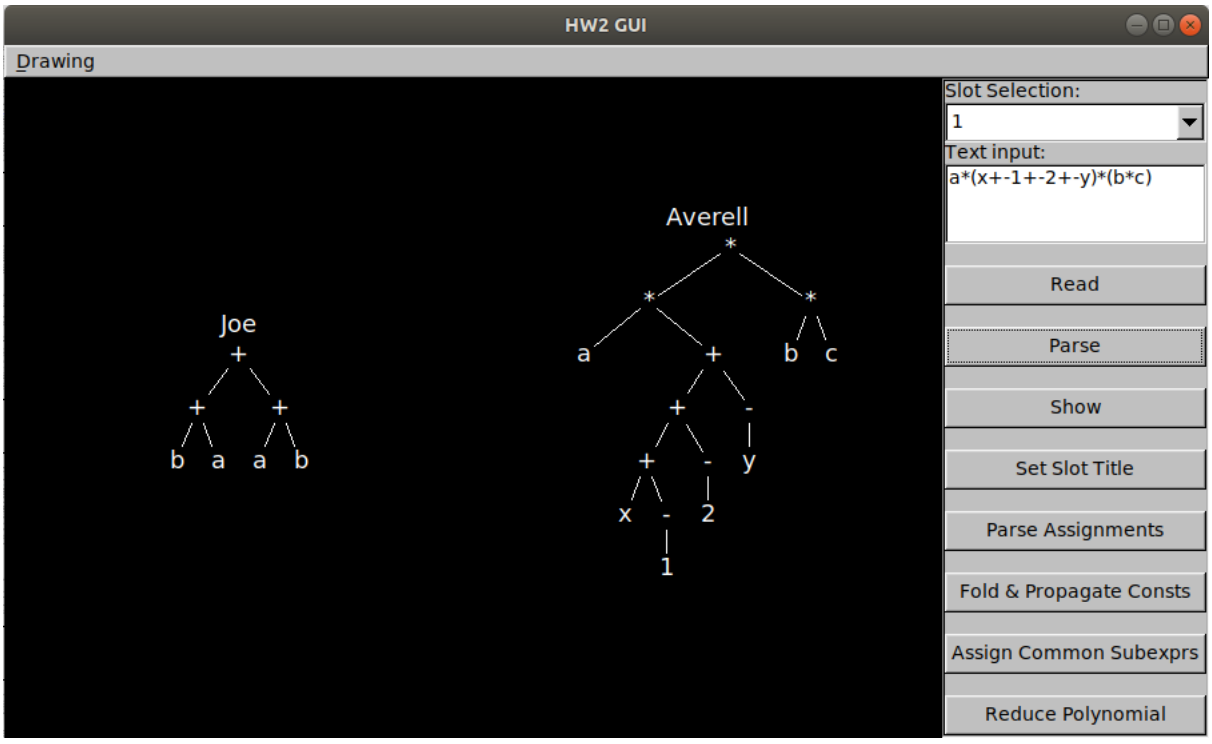



Figure 5: A Screenshot of the GUI

[5] You really owe that person some sort of **bever**age! Kidding.

[6] It's so nice that I used it to create figures for this PDF. The Haskell way of lazyness!

# 5 Tips

- Once again, try to make maximal use of standard library functions to make your job easier. You can use everything from the `Prelude`, `Data.List` and anything else you might want to import such as `Data.Maybe`.
- Get familiar with folds, which are `foldl` and `foldr` (along with a few more variants found in `Data.List`). These are great for iterating over lists and will improve your Haskell experience both in the homework and (probably) the lab exam.
- While the binary expression tree is nice for compilation, it is not the best representation to work with for every problem. Feel free to extend the homework with your own data types and work with them. Converting your `ExprV` values to a different type, working on that type and then reconstructing an `ExprV` result could be easier than trying to solve the problem in the `ExprV` domain. (Nested tip: polynomials can be represented nicely with linked lists!)
- Once again, the input will be within the constraints described in the function descriptions. Remember that the parser and GUI is not related to your grade at all, and are simply there to make your life easier. Your functions just work on expressions.
- There is no running time constraint on the functions. The expressions that your code will be tested with will be fairly small, so you should be able to get through even with exponentially complex solutions, even though that's not too great!

# 6 Submission

Submission will be done via cengclass. Submit a single Haskell source file named `HW2.hs`. Keep the signatures and module declaration the same as the template file. You can modify the other source files for your own fun, but they will not affect your grading and should not be submitted. Feel free to check them out for inspiration, though.

# 7 Grading

Each function will be tested and graded **independently**. However, you still need to make sure your code file **as a whole** can be compiled by `ghc`; so if there are functions you do not intend to implement, make them return a default value you like. The types of the functions **must** obey the provided specifications.

If your file has a different name than expected or you have multiple files, 5 points will be deducted from your final grade. If your code does not compile (but the compilation errors can be fixed trivially), this will also result in a 5 point deduction. These deductions will *not* reduce your grade below zero.

# 8 Regulations

1. **Programming Language:** You must write your code in Haskell. Your submission must be compilable with `ghc`/`ghci` on the ineks.
2. **Late Submission:** You have been given 10 late days in total and at most 3 of them can be used for one assignment. After 3 days, submissions will no longer be accepted and you will receive a grade of 0, no matter the circumstances. The extended deadline is therefore 15 April 2020, Wednesday, 23:55 **sharp**.
3. **Cheating: We have a zero tolerance policy for cheating**. People involved in cheating will be punished according to the university regulations.

4. **Newsgroup:** You must follow the course category on COW for discussions and possible updates on a daily basis.

5. **Evaluation:** Your code will be evaluated automatically using the "black-box" testing technique, so make sure to obey the specifications.