Homework Assignment 3

Functional and Logic Programming, 2023

Due date: Thursday, June 01, 2023 (01/06/2023)

Bureaucracy

- Submission is in pairs, but solo submission is also allowed. We very much suggest you pair-up, as solving this exercises with another person greatly enhances your learning (as well as being more fun!).
- To submit, create a zip file named `HW3_<id1>_<id2>.zip` where `<id1>` and `<id2>` are the submitters IDs.
  - Or `HW3_<id>.zip` if submitting alone.
- Make sure your submission compiles successfully. Submissions which do not compile will receive a 0 grade!
  - We will be using the following command to compile the file: `ghc -Wall -Werror HW3.hs`.
- You may submit the assignment after the due date even without approval (e.g., excluding reserve duty, serious illness, or other cases covered by the student administration).
  - You will be penalized for **5 points for every late day**.
  - The **maximum** extension allowed by this is **3 days**.
- For any late submissions, with or without approval, please E-mail your submission directly to ofir.yaniv@post.runi.ac.il.

General notes

- The instructions for this exercise are split between this file and `HW3.hs`. This file offers a more high-level overview of the exercise, as well as offering a few hints. The Haskell file details all the required functionality for this assignment.
- You may not modify the `import` statement at the top of the file, nor add new `import`s.
  - If you are unsure what some function does, you can either ask HLS or Hoogle.
  - Hoogle also support module lookups, e.g., `Prelude.notElem`.
  - Do be aware however, that some functions from the standard library are more general than what we have learned so far in class.

    - And some cases their definition may not be entire clear just yet!

- The exercises and sections are defined linearly. That is, it is a good idea to use previously defined functions (either from the same section or ones before it). It is also a good idea to use functions you saw in class; some of them are already imported, and some of them you would have to define yourself.
  - Do not be alarmed by the large amount of functions! Many of them are simple one-liners, and were designed to aid you in solving the more complex functions.
  - In general, you may define as many helper functions as you wish.
- Try to write elegant code, as taught in class. Use point-free style, η-reductions, and function composition to make your code shorter and more declarative. Prefer `foldr` over manual recursion where possible, and use functions like `map` and `filter` when appropriate. Although it is not required in the homework assignments, non-elegant code *will* be penalized in the test, so this is a good exercise. HLS and hlint can be very helpful in this.
  - Do note that in some cases, hlint may suggest functions which are not imported, or which you are trying to implement right now!
- If possible, please ask your questions first in Piazza, as this will allow all students to take part in the discussion.

Section 1: Expanded Digital Circuits

In the tutorial, we implemented a simplifier for a language based on Boolean circuits. In this assignment, we will expand the language to support three new features:

1. Boolean expressions. An expression is either a literal, a variable, or a Boolean function of other expressions.
2. Defining new variables. These override the previews value of a variable, and existing only within their scope.
3. Unconditional blocks. Similar to {...} blocks in a language like Java, variable definitions exist only inside their containing block. Semantically speaking, these are equivalent to `if (true)` {...}.

The modified tree, or abstract syntax tree (AST), now looks like this:

```
type Variable = String
data Expression
  = Not Expression | Or Expression Expression | And Expression Expression
  | Var Variable | Literal Bool deriving (Show, Eq)
data Statement =
  Return Expression |
  Block [Statement] |
  If Expression [Statement] |
  IfElse Expression [Statement] [Statement] |
  Define Variable Expression
  deriving (Show, Eq)
```

The rules for defining variables are:

- Just like before, we assume all variables have some initial value defined before starting the program.
- Variable `Define`, only apply to the statement's current scope (and its sub-scopes, of course). Outside the scope, the variable is reverted to its old value.
  - In other words, if a variable is defined inside an **`if`** statement, **`else`** statement, or an unconditional block, its assigned value will be "forgotten" when exiting the block's scope.
  - A top-level definition applies for the remainder of the program.
- Definitions may use the old variable definition, e.g., `x = x || y` is represented in our tree as `Define "x" (Or (Variable "x") (Variable "y"))`.

1.1 Pretty printing

Before we begin simplifying, it would be very useful for debugging if we could print programs in our language. To do that, we want to support pretty printing expressions, statements, and entire programs. To support overloading, we'll create a **`class`** for pretty printing, which should be implemented for all three types.

```
class PrettyPrint a where
  prettyPrint :: a -> String
```

We begin with pretty printing expressions, using the following rules:

- A pretty printed expression will always be a single line.
- `Literal`s are printed in lower case: `true`/`false`.
- `Var`s are printed as is.
- We will only add parenthesis when needed.
  - So `!x` does not need parenthesis, nor does `x || y`, but `!(x || y)` does.
  - In general, a top level expression—e.g., in a variable definition or return statement—will never need parenthesis.

  Here are a few examples of pretty printed expressions:

```
prettyPrint $ Not $ Var "x"
"!x"
prettyPrint $ Or (Var "x") (Var "y")
"x || y"
prettyPrint $ And (Var "x") (Not $ Var "y")
"x && !y"
prettyPrint $ Or (Var "x") (Not $ And (Var "y") (Not $ Var "z"))
"x || !(y && !z)"
```

The next step is pretty printing a single `Statement`. Note however, that a single statement can contain multiple statements in itself, however those statements will always be **indented**. A few guidelines, based on the "One True Brace Style":

- Indentation should use **two** spaces.
- The `if` condition follows the normal expression printing rules, and will always be parenthesized.
- An `if`/`else` statement will always have curly braces, and will use
- The `if`/`else` block will always be in a new line, and will always be indented.
- An unconditional block is printed similar to an `if` statement, except it doesn't have the `if` part.

Here is a comprehensive example of pretty printing a statement:

```
prettyPrint $
 If (And (Var "x") (Not $ Var "y")) [
    Define "z" (Var "x"),
    Block [
      Define "y"
        (Literal False),
      Define "y"
        (And (Var "y") (Var "z"))
    ],
    Define "a"
      (Or (Var "z") (Var "y")),
    Block [],
    IfElse (Var "a")
      [Return $ Literal True]
      [Return $ Var "z"]
 ]
```

```
if (x && !y) {
  z = x
  {
    y = false
    y = y && z
  }
  a = z || y
  {
  }
  if (a) {
    return true
  } else {
    return z
  }
}
```

Hints:

- Define a helper function for keeping track of the current level of indentation.
- The function `unlines :: [String] -> String` from `Prelude` joins a list of `String`s using `\n`, similar to the following Python code:

```
"\n".join(["1", "2", "3"])
```

Tip: If you just evaluate or call `prettyPrint` from within `GHCi`, you will get a string with literal `\n` instead of new lines. Use `putStrLn` from within `GHCi` to avoid this:

```
p = If (Var "x") [Return $ Literal True]
prettyPrint p
"if (x) {\n    return true\n}\n"
putStrLn $ prettyPrint p
if (x) {
  return true
}
```

Lastly, we need to pretty print an entire program, or `[Statement]`. Hint: This should be a simple one-liner.

1.2 Simplifying the new AST

Since we added expressions, the first simplification step is to support simplifying expressions. We will use the following simplification rules, applied inductively:

- If the value of a `Var` is known, it should be replaced with a `Literal`.
- `Not` of a `Literal` should return the negated `Literal`
- If any operand of `Or` is `true`, it should be replaced with a `Literal true`

- If any operand of `And` is `false`, it should be replaced with a `Literal false`

Next, we will simplify `Statements`. For the most part, most `Statement` don't require any extra simplifications over what was shown in class:

- The expressions of `Return` or `Define` are already simplified by `simplifyExpression`.
- `If`/`IfElse` behave the same as in the tutorials, with one difference: to ensure definition scopes, we will use `Block`.
  - So `if (true) b` becomes a `Block b`.
    - Note that `if (true)` can either be written as part of the original program, or reached via simplifications!
  - Likewise, `if (false) b1 else b2` becomes `Block b2`.
- As mentioned, `Block` behave similar to `if (true)` blocks.

However, since our new language now sports variable definitions, statements are no longer standalone, but can actually change the current `Scope`! For example, a definition `x = true` changes the known value of `x` for the current scope. Because of that, we can no longer use `concatMap`, since we need to *carry* the updated scope from one `Statement` to another. We will use `foldl'` for the iteration over the `[Statement]`, since we need access to the initial empty scope at the start of the iteration. Note that variable definitions only apply within their scope! Below is the skeleton for `simplify`:

```
simplify :: [Statement] -> [Statement]
simplify = simplifyWithScope empty

simplifyWithScope :: Scope -> [Statement] -> [Statement]
simplifyWithScope s = reverse . snd . foldl' (uncurry go) (s, []) where
  go scope statementsSoFar statement =
    let (newScope, simplified) = simplifyStatement scope statement
     in (newScope, simplified ++ statementsSoFar)
  -- Implement this!
  simplifyStatement :: Scope -> Statement -> (Scope, [Statement])
```

Below is an example of a pre and post simplified code:

```
if (x) {
  z = x || y
  {
    if (z) {
      y = !z
    }
    z = y
    if (z) {
      return false
    } else {
      return z
    }
  }
}
```

1: pre-optimized

```
if (x) {
  z = true
  {
    {
      y = false
    }
    z = y
    if (z) {
      return false
    } else {
      return false
    }
  }
}
```

2: post-optimized

Section 2: Basic typeclasses

2.1 Abstract data types

Implement `Show`, `Eq`, and `Ord` instances for `Tree`, treating it as an abstract Set. You may assume there are no repeats in the trees, and that they are already ordered.

- Two trees should be equal if they hold the same values.
- Printing a tree should return its elements sorted, with wrapping braces ({}).
- Trees should be ordered lexicographically (similar to lists), based on their sorted set.

Examples:

```
single e = Tree Empty e Empty
tree1 = Tree Empty 1 $ Tree Empty 2 $ single 3
tree2 = Tree (single 1) 2 (single 3)
tree1 == tree2
True
show tree1
{1,2,3}
show tree2
{1,2,3}
tree1 `compare` tree2
EQ
```

2.2 Typeclass constraints

`nub` removes duplicates from a list, maintaining the original order.

```
nub [2,1,3,3,4,1,2]
[2,1,3,4]
```

Tip: it is not possible to implement this better than $O(n^2)$, because you only have `Eq`!

Next, we'll deal with sorting a list. We can either sort a list whose elements are `Ord` (`sort`) , or we use a mapping function to extract an `Ord` value from the list elements (`sortOn`). This should be done $O(n \log(n))$, of course.

```
sort [3, 1, 2, 3]
[1,2,3,3]
data Person = Person Int String deriving Show
age  (Person a _) = a
name (Person _ n) = n
persons = [Person 40 "Taylor", Person 42 "Isaac", Person 37 "Zac"]
sortOn (Down . name) persons
[Person "Zac" 37,Person 40 "Taylor",Person 42 "Isaac"]
sortOn (Down . age) persons
[Person 42 "Isaac",Person 40 "Taylor",Person "Zac" 37]
```

Hints:

- You can implement one sort in terms of another.
- In Haskell, `Data.Set` is sorted, and `Data.Map` is sorted by its keys. So to sort in $O(n \log(n))$ time, you can insert element into a `Data.Set` or `Data.Map` and then extract the elements using `elems` (both structures have this function), or `assocs` which returns a list of keys and values (only exists in `Map`, obviously). However, note that sets don't allow repeats, and maps don't allow repeats in keys.
- The following `data` type may prove useful:

```
-- Eq/Ord is based on the first argument only!
data Arg a b = Arg a b
instance Eq a => Eq (Arg a b) where
  (Arg a1 _) == (Arg a2 _) = a1 == a2
instance Ord a => Ord (Arg a b) where
  (Arg a1 _) <= (Arg a2 _) = a1 <= a2
```

- `Data.Map` supports updating on insert using `insertWith`, or creating an entire `Map` using fromListWith. For `Data.Set` you would have to implement similar functionality yourselves.