CMPT 383 Comparative Programming Languages

Programming Assignment 1

This assignment is due by 11:59pm PT on Tuesday Feb 11, 2025. Please submit it to Canvas.

Requirements:

- This assignment must be your own work. No collaboration is permitted.
- You can only use library functions from the following modules: `Prelude`, `System.IO`, `System.Environment`, `Data.List`, and `Data.Map.Strict`. Detailed information of these modules can be found on the website `https://hoogle.haskell.org`

Late policy:
Suppose you can get n (out of 100) points based on your code and report

- If you submit before the deadline, you can get all n points.
- If you submit between 11:59pm PT Feb 11 and 11:59pm PT Feb 12, you get $n - 10$ points.
- If you submit between 11:59pm PT Feb 12 and 11:59pm PT Feb 13, you get $n - 20$ points.
- If you submit after 11:59pm PT Feb 13, you get 0 points.

(100 points) A formula in propositional logic can be a boolean constant (`Const`) with value `True` or `False`, a boolean variable (`Var`) such as $x_1, x_2, \ldots$, or the composition of formulas using logic connectives $\neg$ (`Not`), $\wedge$ (`And`), $\vee$ (`Or`), $\rightarrow$ (`Imply`), and $\leftrightarrow$ (`Iff`).

For a formula ϕ, a variable assignment is a mapping that maps each variable in ϕ to a truth value in {`True`, `False`}. Given a formula ϕ and a variable assignment, the formula ϕ evaluates to a truth value based on the following truth tables (where T stands for `True` and F stands for `False`).

ϕ_1	$\neg\phi_1$
T	F
F	T

(a) `Not`

ϕ_1	ϕ_2	$\phi_1 \wedge \phi_2$
T	T	T
T	F	F
F	T	F
F	F	F

(b) `And`

ϕ_1	ϕ_2	$\phi_1 \vee \phi_2$
T	T	T
T	F	T
F	T	T
F	F	F

(c) `Or`

ϕ_1	ϕ_2	$\phi_1 \rightarrow \phi_2$
T	T	T
T	F	F
F	T	T
F	F	T

(d) `Imply`

ϕ_1	ϕ_2	$\phi_1 \leftrightarrow \phi_2$
T	T	T
T	F	F
F	T	F
F	F	T

(e) `Iff`

For example, consider a concrete formula ϕ being $x_1 \wedge \neg x_2$. ϕ evaluates to `True` if the variable assignment is x_1 = `True` and x_2 = `False`. Also, ϕ evaluates to `False` if the variable assignment is x_1 = `True` and x_2 = `True`.

A formula ϕ is said to be *satisfiable* if there exists a variable assignment under which ϕ evaluates to `True`. Otherwise, the formula is said to be *unsatisfiable*. In general, the satisfiability of a formula can be checked using the truth table method. Specifically, we can list all possible variable assignments of a formula, and then check if any variable assignment can make the formula evaluate to `True`.

For example, check the satisfiability of $x_1 \wedge \neg x_2$.

x_1	x_2	$\neg x_2$	$x_1 \wedge \neg x_2$
T	T	F	F
T	F	T	T
F	T	F	F
F	F	T	F

Here, $x_1 \wedge \neg x_2$ is satisfiable because there exists a variable assignment $x_1 = T$ and $x_2 = F$ under which the formula evaluates to T.

As another example, check the satisfiability of $\neg(x_1 \to (x_2 \to x_1))$.

x_1	x_2	$x_2 \to x_1$	$x_1 \to (x_2 \to x_1)$	$\neg(x_1 \to (x_2 \to x_1))$
T	T	T	T	F
T	F	T	T	F
F	T	F	T	F
F	F	T	T	F

Here, $\neg(x_1 \to (x_2 \to x_1))$ is unsatisfiable, because there is no variable assignment that can make the formula evaluate to T.

You need to write a Haskell program to check the satisfiability of formulas in propositional logic. The program must be in a form that GHC can compile (i.e., you need a `main`). It needs to take one command-line argument that denotes the path to the formula file. You can assume each line of the file contains a formula to check, and the program needs to print to the console telling whether each formula is satisfiable (print `SAT`) or not (print `UNSAT`).

Representation of Formulas

We can use strings for variable names

```haskell
type VarId = String
```

A propositional formula can be represented as a value of the following `Prop` type in Haskell

```haskell
data Prop = Const Bool
          | Var VarId
          | Not Prop
          | And Prop Prop
          | Or Prop Prop
          | Imply Prop Prop
          | Iff Prop Prop
          deriving (Eq, Read, Show)
```

Representation of Variable Assignments

Since a variable assignment maps variables to a truth value, we can import the `Map` data structure and relevant functions from library `Data.Map.Strict`

```haskell
import qualified Data.Map.Strict as Map
```

and define the variable assignment as

```haskell
type VarAsgn = Map.Map VarId Bool
```

Inclusion of Definitions

Note that you need to use exactly the same definition of `VarId`, `VarAsgn`, `Prop`, and their deriving clauses as written in this document. Otherwise, you will lose points because the grading scripts may not work as expected.

Detailed Steps

1. Write a function `findVarIds :: Prop -> [VarId]` that returns a list of **distinct** variable names in a propositional formula.
2. Write a function `genVarAsgns :: [VarId] -> [VarAsgn]` that returns **all** possible variable assignments given a list of variable names. Hint: given n variable names, there are 2^n variable assignments in total. You might want to use `Map.empty` and `Map.insert` for the map data structure.
3. Write a function `eval :: Prop -> VarAsgn -> Bool` that computes the truth value of a formula given a variable assignment.
4. Write a function `sat :: Prop -> Bool` that returns whether a formula is satisfiable or not.
5. Write a simple function `readFormula :: String -> Prop` that reads a formula string (e.g., `Iff (Var "x1") (Var "x2")`) to its corresponding value of type `Prop`. Hint: look at the deriving clause of `Prop`.
6. Write a function `checkFormula :: String -> String` that takes a formula string as input and produces a string as input indicating whether the formula is satisfiable. Specifically,
   - If the formula is satisfiable, output string `SAT`.
   - If the formula is unsatisfiable, output string `UNSAT`.
7. Write a `main` function to handle IO and put everything together.

Sample Input and Output

Suppose we have a formula file called `formulas.txt` that contains the following four lines:

```
Const True
Iff (Var "x") (Var "y")
And (Var "x1") (Not (Var "x2"))
Not (Imply (Var "p") (Imply (Var "q") (Var "p")))
```

After compiling, we can run the executable and get

```
$ ./P1_SFUID formulas.txt
SAT
SAT
SAT
UNSAT
```

Deliverable

A zip file called `P1_SFUID.zip` that contains at least the followings:

- A file called `P1_SFUID.hs` that contains the source code of your Haskell program. You can have multiple source files if you want, but you need to make sure `ghc P1_SFUID.hs` can compile.
- A report called `P1_SFUID.pdf` that explains the design choices, features, tests, issues (if any), and anything else that you want to explain about your program.