CMPT 383 Comparative Programming Languages

Programming Assignment 2

This assignment is due by 11:59pm PT on Wednesday Mar 9, 2022. Please submit it to Canvas.

Requirements:

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

Late policy:
Suppose you can get n (out of 10) 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 Mar 9 and 11:59pm PT Mar 10, you get $n-1$ points.
- If you submit between 11:59pm PT Mar 10 and 11:59pm PT Mar 11, you get $n-2$ points.
- If you submit after 11:59pm PT Mar 11, you get 0 points.

(10 points) Consider the following grammar G_0 for formulas in propositional logic

$$
\begin{array}{rcl}
\mathit{Formula} & ::= & \text{`}\texttt{T}\text{'} \mid \text{`}\texttt{F}\text{'} \mid \mathit{Ident} \\
& \mid & \text{`}\texttt{(}\text{'}\ \mathit{Formula}\ \text{`}\texttt{)}\text{'} \\
& \mid & \text{`}\texttt{!}\text{'}\ \mathit{Formula} \\
& \mid & \mathit{Formula}\ \text{`}\texttt{/\textbackslash}\text{'}\ \mathit{Formula} \\
& \mid & \mathit{Formula}\ \text{`}\texttt{\textbackslash/}\text{'}\ \mathit{Formula} \\
& \mid & \mathit{Formula}\ \text{`}\texttt{->}\text{'}\ \mathit{Formula} \\
& \mid & \mathit{Formula}\ \text{`}\texttt{<->}\text{'}\ \mathit{Formula}
\end{array}
$$

Here, *Formula* is the start symbol. `T` stands for the constant `True`, and `F` stands for the constant `False`. `Ident` denotes variable names, starting with a lower-case letter, followed by zero or more alphanumeric characters (letters or digits). `!` denotes the logical not, `/\` denotes the logical and, `\/` denotes the logical or, `->` denotes the logical implication, and `<->` denotes the logical iff. The **precedence** of different operators (from high to low) is as follows: `(), !, /\, \/, ->, <->`. All binary operators are **right-associative**.

In this assignment, you need to write a parser in Haskell to parse strings in the language of G_0. Given such a string, the parsing result should be a value of the following type

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

As indicated by the names of data constructors, `Const` denotes a boolean constant, `Var` denotes a variable, `Not` denotes the logical not, `And` denotes the logical and, `Or` denotes the logical or, `Imply` denotes the logical implication, and `Iff` denotes the logical iff. For example,

- `T` should be parsed into `Const True`
- `t` should be parsed into `Var "t"`
- `x1 /\ x2` should be parsed into `And (Var "x1") (Var "x2")`
- `x1 /\ x2 \/ x3` should be parsed into `Or (And (Var "x1") (Var "x2")) (Var "x3")`

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

Handling Whitespaces

In general, a whitespace means a space character or a control character that is similar to a space, such as `\t`, `\r`, `\n`. The complete set of whitespace characters is defined by the `isSpace` function from `Data.Char`.

When writing a grammar like G_0, we can assume there are zero or more whitespace characters surrounding each symbol in the grammar. For example, "`T`" is a string in the language of G_0. "` T `" with preceding and trailing whitespaces is also considered a string in the language of G_0. However, we cannot assume there is any whitespace "inside" the symbol with quotation marks in the grammar. For example, `/\` should be considered as one symbol as a whole. No whitespace is allowed between `/` and `\`, because adding whitespaces between `/\` splits it into two symbols.

You need to follow the above convention when writing a grammar. You also need to handle whitespaces in your parser. As a hint, the `token` function that we learned in class can handle whitespaces.

Detailed Steps

1. Rewrite grammar G_0 to G_1 such that G_1 enforces the intended precedence. Include G_1 in your report. As a hint, here are some possible productions in G_1:

$$\begin{array}{rcl} \textit{Formula} & ::= & \textit{Formula}\ \text{`}\texttt{<->}\text{'}\ \textit{Formula} \mid \textit{ImpTerm} \\ \textit{ImpTerm} & ::= & \ldots \\ & \ldots & \\ \textit{Factor} & ::= & \text{`}\texttt{(}\text{'}\ \textit{Formula}\ \text{`}\texttt{)}\text{'} \mid \text{`}\texttt{T}\text{'} \mid \text{`}\texttt{F}\text{'} \mid \textit{Ident} \end{array}$$

2. Rewrite grammar G_1 to G_2 such that G_2 enforces right-associativity of all binary operators. Include G_2 in your report.

3. Reuse the code that we have learned about `Parser`, including the `Parser` definition, the `parse` function, instances of `Functor`, `Applicative`, `Monad`, `Alternative` (imported from `Control.Applicative`), basic parsing primitives, and so on.

4. Write a parser `constant :: Parser Prop` (exact name) that can parse `T` and `F`.

5. Write a parser `var :: Parser Prop` (exact name) that can parse variables.

6. Write a parser `formula :: Parser Prop` (exact name) that can parse all possible formulas in the language of G_2.

7. Write a function `parseFormula :: String -> String` (exact name) that takes a formula string (e.g., `x1 /\ x2`) as input and generates a string as output representing the parsing result. Specifically,
   - If the parsing succeeds and a value `v` of type `Prop` is obtained, generate the output using `show v`.
   - If the parsing fails, output string "`Parse Error`". Note that non-exhaustive consumption of the input formula string should be considered as a parsing failure.

8. Write a `main` (exact name) to handle IO and put everything together.

The program must be in a form that GHC can compile. It needs to take one command-line argument denoting the path to the formula file. Each line of the file contains a formula string to parse, and the program needs to print the result of `parseFormula` on each string to the console.

Sample Input and Output

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

```
T
t
x1 /\ x2
x1 /\ x2 \/ x3
/\ x1
```

After compiling, we can run the executable and get

```
$ ./p2_firstname_lastname formulas.txt
Const True
Var "t"
And (Var "x1") (Var "x2")
Or (And (Var "x1") (Var "x2")) (Var "x3")
Parse Error
```

Deliverable

A zip file called `p2_firstname_lastname.zip` that contains at least the followings:

- A file called `p2_firstname_lastname.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 p2_firstname_lastname.hs` can compile.
- A report called `p2_firstname_lastname.pdf` that includes the grammars G_1, G_2, and explains the design choices, features, issues (if any), and anything else that you want to explain about your program.