301398647 CMPT 383: Programming Assignment 2 Report Evan Mangat

Step 1:

Using what was shown on slide 14 of Lecture 12-Parsing2

G_1 =

Formula ::= Formula ‘<->’ Formula | ImpTerm

ImpTerm ::= ImpTerm ‘->’ ImpTerm | OrThis

OrThis ::= OrThis ‘\/’ OrThis | AndThis

AndThis ::= AndThis ‘/\’ AndThis | NotThis

NotThis ::= ‘!’ NotThis | Factor

Factor ::= ‘(‘ Formula ‘)’ | ‘T’ | ‘F’ | Ident

Step 2:

Using what was shown on slide 15 of Lecture 12-Parsing2

G_2 =

Formula ::= ImpTerm ‘<->’ Formula | ImpTerm

ImpTerm ::= OrThis ‘->’ ImpTerm | OrThis

OrThis ::= AndThis ‘\/’ OrThis | AndThis

AndThis ::= NotThis ‘/\’ AndThis | NotThis

NotThis ::= ‘!’ NotThis | Factor

Factor ::= ‘(‘ Formula ‘)’ | ‘T’ | ‘F’ | Ident

Step 3:

Not too much to be said here it was simply reformatting and copying the code from slides 11-2-Parsing1, and 12-Parsing2.

Though there was the notable addition of the char:: function not present in the slide, however it was used in the string :: function taken from the slides.

This function simply returns the parsed char, it will be very helpful for Step 4.

Step 4:

While the eventual solution to this step was very simple (3 Lines including the type signature), I found actually generating the concept in code was quite difficult in comparison to the other steps. I actually chose to do step 5 and 6 before step 4 to help conceptualize. Step 4 is the "base" level of the chain composed of these 3 steps. So I worked backwards. There was no template in the slides to follow for this question. Eventually the idea to use an alternate to parse in this instance clicked as I was reading the 11-2 Slides on Alternatives. There will only ever be 2 choices between True and False. Thus, the choice to go with this implementation.

Step 5:

Upon initial glance I thought this might be the hardest function to implement. Luckily much of the heavy lifting of this functions complexity is handled by the identifier function we were provided to deal with whitespace. Since the job of isolating vars is taken care of, all that's left to be created is returning the string as type Prop as required by the type signature. From the definition of Prop, Var's are Prop's of type String so we return Var(c). Another integral part of the function is the added <|> constant, which connects this to Step 4, since this is what allows the chain in Step 6 to include 'T' and 'F' as described in G_2's final production: Factor ::= '(' Formula ')' | 'T' | 'F' | Ident

Step 6:

This was the easiest to implement of the later steps. This is because the blueprint is given on Slide 17 of lecture 12-Parsing2. All that is required of us is to combine that blueprint with our G_2 rather than the grammar on the slide and that is all. A minor difficulty I encountered while doing this step was at the end, the slide didn't demonstrate how to handle productions that terminated in multiple possibilities in code. On the slide it was just natural, but we have 'T' | 'F' | Ident. To solve this issue, I ended up just linking to Ident (var function) and then within the var function linking to constant (handles 'T' and 'F'). This was after much difficulty trying to find a way to link both at the end of Step 6 in:

```
factor :: Parser Prop
factor = do symbol "("
            f <- formula
            symbol ")"
            return f
          <|> var
```

Step 7:

The 2nd hardest function I found to implement. This was simply because like Step 4, there was nothing in the slides to base off. In order to find a solution, I had reviewed the checkFormula code from P1 and thought about how the question was presented in the P2.pdf. It reminded me of a case based implementation with the bullet points, so that is what I went for. If it encounters a valid item of type prop in accordance with the top of the chain of Step 6 with formula, it will output that prop with the correct return value determined by the chain from Step 6 -> Step 5 -> Step 4. If the parse value is invalid or any other situation, it will return "Parse Error". Originally I was actually scared I made a mistake because the output had excess "\" in it. However, I realized it works fine in terminal and this only occurs in ghci.

Main:

Very little to say here. I essentially re-used the main code from P1 while exchanging checkFormula for parseFormula. We get arguments, start reading the file with the first line being the head arg and save it in input. Input is broken up by line with lines and saved into the formulas variable. Each line in formulas is then mapped with parseFormula for a result.

Sample run of program. formulas.txt contains the following test case from P2.pdf:

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

The result:

```
PS C:\Users\evanm> ghc ./P2_Evan_Mangat.hs
PS C:\Users\evanm> ./P2_Evan_Mangat formulas.txt
Const True
Var "t"
And (Var "x1") (Var "x2")
Or (And (Var "x1") (Var "x2")) (Var "x3")
Parse Error
PS C:\Users\evanm>
```