Design and implementation

Answer to *** questions:

In the "Expr ::= Expr Oper Expr" there's Left recursive happens, so we managed to alter it.

First we list them by sequence of priority and got these for classes:

> *1.Expr ::= 'not' Expr | Expr1*
> *2.Expr1 ::= Expr2 | Expr1 oper1 Expr1*
> *3.Expr2 ::= Expr3 | Expr2 oper2 Expr2*
> *4.Expr3::= Term | Expr3 oper3 Expr3*

Then we deal with left-associative and got this:

> *1.Expr::= 'not' Expr | Expr1*
> *2.Expr1 ::= Expr2 | Expr2 oper1 Expr2*
> *3.Expr2 ::= Expr3 | Expr2 oper2 Expr3*
> *4.Expr3 ::= Term | Expr3 oper3 Expr3*

Then we deal with Left recursion and get this:

> *1.Expr ::= 'not' Expr | Expr1*
> *2.Expr1 ::= Expr2 | Expr2 oper2 Expr2*
> *3.Expr2 ::= Expr3 Expr2'*
> *Expr2' ::= oper2 Expr2 | epsilon*
> *4.Expr3 ::= Term Expr3'*
> *Expr3' ::= oper3 Expr3 | epsilon*

Then to make it run faster to parse we managed to put the operate out like this:

> *1.Expr::= 'not' Expr | Expr1*
> *2.Expr1 ::= Expr2 Oper1*
> *Oper1 ::= == Expr2 | != Expr2| < Expr2 | <= Expr2 | > Expr2 | >= Expr2 | in Expr2| not in Expr2|epsilon*
> *3.Expr2 ::= Expr3 oper2*
> *oper2 ::= + Expr3 | - Expr3 | epsilon*
> *4.Expr3 ::= Term oper3*
> *Oper3 = * Term | / Term | % Term | epsilon*

And then we implemented the parser as the grammer.

<++ will stop after it met with the first match
<|> won't stop untile it finished all the matching

We mostly used <++ in the task because it will only return one result, so it will be easier to judge in parseString.

implementation

These are some parts in the code we think important

We first implemented those parser as the assignment says except the Expr and the Oper part. For most of the compositions of the Expr we implemented them as parsers separately, like parseNum, parseString..etc.

For the parseIdent, we first have a list of reserved words, and then in the parser we split the Ident into two parts, the first letter and rest letters. As the first letter can't be digit, we just match with letter and "_".Then we match the rest letters and combine them, check if it's in the reserved word list.

For the parseNum, we also do that in first-rest pattern to get rid of the case like ' - 1 '. As there might have mutiple 0 or -0 at the beginning, we just first match the first letter, if it's '0' or "-0" just return the Exp 0, in that way there will not have such as 007 in the final parse result.

For the comment part, we made a function called skipmore. In the warmup part to get rid of the blank lines we used the ReadP built-in function skipSpaces. That function can match and remove all the spaces, but in this assignment we need to not only deal with spaces, but also with comments. In the skipmore function it first uses look function to check the following string, if it's head is '#' then that means it's a comment.Then the function will continue to match string until it reads the char '\n', then it will return a Parser() and stop parsing.

For the list comprehension part, as there are words "for in" and "if" to handle, there needs to be spaces or some parentheses after the word, we implemented a function purestring to handle these words.In the function it first matches the input string, and then look for the following string. If there are parentheses or spaces it will return empty into parse tree to continue parsing the following Exprs. If there's no brackets or spaces it will fail with an error.

In another big part of the assignment which is parsestring we currently just finished the basic parsing work. We implemented a function stringhelper, it will recursively read string and match unicode char if it's not ' or . And then concat it and run the function again until it get the ' which means the end of a string, then it will return the whole string into the parsing tree.

As we spent most of the time in warmup part to understand it and make it perfect, we barely have time to finish the Assignment. There's still two parts which is parse "Expr Oper Expr" and special cases of string left. However we have got the idea of them, the Expr we want to alter the original grammer into the one we have above and try to figure it out. About the string part we are gonna to add some Judge conditions into the stringhelper function to complete it.

Assessment

Completeness

We completed all the tasks in the assignment.

Correctness

We made a lot of test cases and also runned the code on onlineTA, both works well.

Efficiency

We made the operator seprately as a function to fasten the matching speed,if it parse the left component it just go into the function to check the operator instead of repeatly parse the first component.

Robustness

If there's ambiguous grammer being input the string will not be all consumed ,and in parseString it will return a Left implement error. We wanted this to be na error but onlineTA seems need it to be a Left value, so we changed it to Left.It will receive all exception situations so the code will not crush easily.

Maintainability

We have some helper function such as stringhelper, that will help to deal with some parsing and if needs to be changed it's easy to change them. .We made the code line as short as possible to make it readable.Also we add some comments about grammer on the functions we declared to make it easier to understand