CSE 4713 / 6713 — Programming Languages
Assignment #2

For programming assignment #2, your task is to write a recursive descent parser in C++. You should study the example parsers, and complete a parser that can successfully parse the provided source code files. This project will require that you build a parser for the *supersub* language. The grammar for this language is defined by the rules in EBNF given in table 1. The first phase of the project will be to build a recursive descent parser to recognize this language and detect syntax errors. Once you have the ability to parse the input, then you will need to construct parse trees for expressions in the language. These expressions will be stored in a dictionary for future processing that will be described in assignment #3.

Table 1 is also includes the first sets which are the sets of valid tokens that will begin a string generated by that given rule. The first sets can be used to select which rule needs to be called to further process the subsequent rules. Generally there will be a procedure created for each rule in table 1. There procedures will call each other recursively to break the input stream into parts as defined by the grammar. Notice that the definition of the <A> rule defines a list of identifiers. However, this definition alone is ambiguous without the disambiguation comment that says to use left association. This means that care will need to be taken when constructing the parse trees for this rule to ensure that left associativity is preserved.

In this project you will be required to build a parse tree to represent the expressions generated by the <A> rule. Thus the functions that implement <F>, <A>, and <I> will return an expression tree. Similarly the rules for <E>, <T>, and <R> will return integers. In this case the calculations of the integer values of the expression will be calculated as the integer expressions are parsed.

The base source files provided with the project include a parseTree.h file that defines a `Expression` class that you will used to represent the parse trees generated by these rules. The leaves of the parse tree will either be defined by a `VARIABLE` or `COMBINATOR` type, while the nodes of the tree will be defined by either the substitution operation (`SLASHDOT`) or by the application listing operator (`APPLICATION`) defined by rule <A>. Note the `SLASHDOT` node uses the name field and the right expression, while the `APPLICATION` operator uses the left and right expression fields. The nodes of the tree only make use name field. Routines to construct these elements of parse trees are provided with the functions:

- `makeVariable(string name)`
- `makeCombinator(string name)`
- `makeSlashDot(string name, expressionP argument)`
- `makeApplication(expressionP left, expressionP right)`

The Expression class includes the methods for converting the expression into a text string, for performing a deep copy of the expression tree, and for updating the left and right fields (performing a deletion of the field changes). These utilities will be used more extensively in the third assignment. The current assignment will make use of the String converter to provide a standardized way to print out expressions for comparison against the reference implementation.

In addition to the Expression class, the `expressionDictionary` class is provided to allow the storage of expressions by given names. This will be used in this assignment to store expressions that are given in `combinator` commands. These commands will be used to save an expression for later processing using the `expressionDictionary`. A `Print()` method is provide to print out the dictionary contents which will be performed by the `dictionary` command.

Assignment 2

Table 1: Grammar production rules

Grammar Productions:	**First Token Set:**
P → {S}	{“combinator”, “evaluate”, “dictionary”, “set”}
S → (J | K | L | M) “;”	{“combinator”, “evaluate”, “dictionary”, “set”}
J → “combinator” [“evaluate”] A	{“combinator”}
K → “evaluate” A	{“evaluate”}
L → “dictionary”	{“dictionary”}
M → “set” ID E	{“set”}
A → I { I } % **left associative**	{ ID, CID, “\”, “{“, “(” }
F → “\” ID “.” A | “{” F “}”	{ “\” , “{”}
I → CID | ID | “(” A “)” | F	{ ID, CID, “\”, “(”, “{” }
E → T {(+ | -) T } % **left associative**	{ (, INTLIT }
T → R {(* | /) R } % **left associative**	{ (, INTLIT }
R → INTLIT |(E)	{ (, INTLIT }

Terminal symbols:

ID – identifier, CID – c-identifier, INTLIT – integer literal

Non-Terminal symbols:

Functions that implement non-terminals F, A, and I should return expression since these will be building parse trees for the expressions that they represent. Functions that represent E, T, and R should return int since they will evaluate integer expressions.

Error Conditions Detected:

Syntax error: display an appropriate error message, report line of error, and quit execution

Execution requirements:

For valid syntax the program should behave as follows: When an evaluate command is executed the program should print out "`Eval expression:`" followed by the string that comes from the parse tree method `String()`. For the `combinator` command (regardless if the optional `evaluate` option is given, the expression tree should be inserted into the dictionary that is already in the skeleton code as `combinatorDictionary` using the `insertItem()` method. The name the expression is inserted under is the identifier that is parsed as part of the `combinator` statement. The `set` statement will simply print out "`let`" and then the identifier name followed by the computed integer value that is the computed integer value of the expression that follows. The dictionary command will simply call the `Print()` method of the dictionary after first printing out the string "`Combinator Dictionary is:`". Example input and output files are provided with the example. Some example test inputs are provided for you to exercise your project. You should create additional tests as these will not necessarily be comprehensive. There are a number of `err#.in` files that will be used to test your parsers ability to correctly identify syntax errors. Your program should be able to correctly identify syntax errors and the line on which they occur. The second test is the `sample1.in` and `sample2.in`. Your program should be able to produce the output file for these tests (given in `sample1.out` and `sample2.out`) exactly before submitting to the grader. A second output file for one of the test cases called `sample1.diag` provides additional print statements from a working recursive parser. You can use this to help debug your implementation. NOTE: Your program must read input from standard input and write to standard output to be counted as correct.

Helping programs: To get a better idea of how to implement this assignment several examples have been created. First a subdirectory in the assignment-2 skeleton code called associationExample gives examples of how to build parse trees for different types of associations. This is provided to help you understand how to build a left associative tree for the A production rule. To get a better idea of how to build parse trees the example from chapter 4 has been extended to generate parse trees of expressions. This code should give you an idea of how to build the rules that construct parse trees for this assignment. Finally, there is a more complex example of a parser given in assign-2-example that has similar characteristics to the assignment grammar.

Note, the grader has changed from last assignment. The grader for this assignment is Kelly Ervin, ke301@msstate.edu. The process will be the same as last assignment in that when you get the program working send it to Kelly for testing. If it passes the internal tests he will give you the OK to submit to mycourses. The grading will be the same as the previous assignment with 80% of the grade being assigned by the program correctness and the remaining 20% being based on code beauty. Like the last assignment, a 10 point deduction will be assigned for every 24 hour period it is late after the due date, with weekends counting as one 24 hour period.