CSE 756, Project 2: simpleC Parser

You need to build a parser for a small subset of C, using CUP and your scanner from Project 1. Expand your solution for Project 1 – modify your simpleC.cup and simpleC.flex to handle the specific subset of C described below. Use the Main class provided at the course web page, instead of MyLexer from Project 1. Do not change this driver class – its output will be used for grading. The project is due by **April 17 (Tuesday)**, 11:59 pm.

Input Language

The input file will start with a (possibly empty) sequence of variable declarations for global variables. Each declaration is of the form ***type declarator*, *declarator*, …;** The type is **`double`**, **`float`**, or **`int`**. The list contains one or more declarators (separated by commas) and ends with a semicolon. Each declarator is a sequence of

1. Zero or more **`*`**
2. An **`identifier`**
3. Zero or more array dimensions; a dimension could be **`[]`** or **`[ integer_literal ]`**
4. Optional initializer of the form **= *expression*** (expressions are described below)

For example, the input file could contain the following declarations for global variables:

```
int a = 5/6, *b, c[1], **d[][5];
float e[100];
```

Note that C allows initialization of arrays at declarations: e.g., **`int a[3]={5,6,7}`**. To simplify life, you can assume that the input program will never contain such array initializations. Your parser does not need to check for their existence because they will never occur.

After the sequence of declarations, the input file contains a single function of the form ***type*** **`identifier(){ … }`** where the identifier is the function name and the return type is **`double`**, **`float`**, or **`int`**. The body of the function is a non-empty sequence of

- variable declarations for local variables
- statements

The declarations of local variables are similar to the declarations of global variables. The declarations and the statements can appear in arbitrary order.

There are several categories of statements. *Expression statements* are of the form ***expression*;** (ending with a semicolon); expressions are defined in more detail below. *Block statements* are of the form **{ … }** where inside the block we have an arbitrary non-empty sequence of variable declarations and statements, possibly including nested blocks. *Iteration statements* are of the form described in Section 6.8.5 of the ANSI C document. You need to implement while-loops, do-while-loops, and the simpler form of for-loops (where the for-loop does **not** contain a declaration). The body of a loop can be any statement, including a block statement. A *return statement* is **`return`** ***expression*;** (ending with a semicolon). Since we do not have a **`void`** return type for the function, a return statement **`return;`** is not allowed.

An *expression* is similar to the *assignment-expression* defined in Section 6.5.16 of the ANSI C document. However, you need to implement a strict subset of the possible C expressions defined by *assignment-expression*. Specifically, you need to implement **all and only** the following operators: **=**, **+=**, **-=**, **>>=**, **<<=**, **>**, **>=**, **<**, **<=**, **<<**, **>>**, **+**, **-**, *****, **/**, **++**, **--**, **[]** (i.e., array subscript operator), and parentheses **( )**. In addition, the expressions can contain identifiers, integer constants, and floating point constants. The expressions do not contain other operators, or constructs such as functions calls. Note that

operators have certain precedence and associativity, as encoded in the grammar in Section 6.5 of the ANSI C document. Furthermore, some of them serve multiple roles: for example, **+**, **-**, and ***** could be binary or unary, and **++** could be pre-increment or post-increment. Examine carefully the grammar in Section 6.5; if you have any questions as to whether to implement a certain kind of expression, talk to me.

Output

The output of your parser should be a single String value that is returned back to **`Main`**. In **`Main`**, this value is printed to **`System.out`**. Each nonterminal in your grammar should be declared a "**`non terminal String`**" in simpleC.cup. All productions should include **`{: RESULT = …; :}`** to compute the value corresponding to this particular parse tree node. The value is pretty printing of the subtree rooted at this node (even thought you will not explicitly construct a parse tree, implicitly there is one). At the root of the tree, the String value is pretty printing of the entire program. The result should be a valid C program that can be compiled (by **`gcc`** on **`stdsun`**) and executed. The behavior of this program should be exactly the same as the behavior of the original program. Examples of such output are available on the web page (out1.c, out2.c).

Details

- Feel free to consult the grammar in the ANSI C document, but keep in mind that the language that you need to implement is not exactly C. You are supposed to write a parser for the *exact language* described in this handout: every valid input program for this language should be executed correctly, and every invalid input program for this language should be rejected with an error message.
- The actions in the grammar should **not** print directly to **`System.out`**: they should simply construct the String values that eventually will become part of the String that is printed by **`Main`**.
- The keywords **`extern`** and **`void`** are not part of the language defined for this lab, and you should remove them from your scanner.
- A number of additions will be necessary for the scanner: e.g., adding **<=**, **>=**, etc. Make sure to submit the modified simpleC.flex.
- Your submission must work correctly on the two test programs (modfft1/2.c) provided on the web page.
- The sample output on the web page is pretty-printed using indentation. You are not required to add indentation – but if you want to have it, feel free to add it.
- Use java.lang.Number for the type of the tokens for integer/floating-point literals. This is a supertype of java.lang.Long/Integer/Float/Double. When constructing the string representation for pretty printing, make sure you take into account the type of the literal (**`1.23`** vs. **`1.23F`** and **`123`** vs. **`123L`**). Given the way the scanner works, input constants in octal or hexadecimal representation will be transformed into decimal constants when generating the tokens, and will be printed as such. This is the behavior you should implement.
- There are various *semantic checks* that would normally happen after parsing. For example, each identifier occurring inside an expression must be in the scope of some declaration. As another example, the left operand of the assignment operator cannot be a constant value (i.e., we cannot have **4=5** as a valid assignment expression). Such constraints are not expressed via the context-free grammar of the language, and therefore should **not** be your concern when performing parsing.

Project Submission

On or before 11:59 pm on the due date, you should submit two files: simpleC.cup and simpleC.flex. They should work correctly with the provided **`Main`** class. Submit your project using

```
submit c756aa lab2 simpleC.flex simpleC.cup
```

If the timestamp on your electronic submission is **12:00 am on the next day or later**, you will receive 10% reduction per day, for up to three days. If your submission is later than 3 days after the deadline, it will not be accepted and you will receive zero points for this project. If you resubmit your project, this will override any previous submissions and only **the latest** submission will be considered – **resubmit at your own risk**.

Academic Integrity

The project you submit must be your own work. Minor consultations with others in the class are OK. The work on the project should be your own: all the design, programming, and testing should be done independently. Submissions that show excessive similarities will be taken as evidence of cheating and dealt with accordingly.