COSC 4785
Compiler Construction I, Fall 2022
Programming Project 3, Starting with Bison

# 1 Introduction

This is the beginning of the syntactic analysis. We are going to try to ensure that our input conforms to the rules of the grammar. To do this, we will combine *flex* with *bison*. *Bison* is the new version of the older program **yacc** ( "yet another compiler compiler" ). As far as I know, *bison* is not an acronym for anything, its just GNU version of *yacc*. You will have to modify your second program to work with this new software. You can find the newest version of the *bison* manual online at http://www.gnu.org/software/bison/manual/ and I have a pdf copy that matches the department's software version of 3.0.4 available on the course homepage. Basically, the *bison* parser (a function called `yyparse()`) repeatedly calls `yylex()`. `yylex()` passes the numeric token and then `yyparse()` tries to figure out if that token matches the grammar.

There are a number of moving parts here. I suggest that you take some time and read over the first few chapters of the *bison* manual. Please IGNORE the information about **GLR** parsers. Any use of **GLR** parsers will result in a 0 grade.

We can certainly have more discussion on the use of *bison* in class if you have specific questions. Otherwise I expect that you can figure all out by yourself.

This programming assignment will be worth 125 points, 25 points for the lexical (*flex*) part and 100 points for the syntactic (*bison*) part.

# 2 Assignment

You will

1. Use *bison* and *flex* to implement a syntactic analyzer for a **portion** of the Decaf language that you are being given in this document.
2. Ambiguity. The Decaf grammar is ambiguous. You will have to use associativity rules and operator precedence to tell *bison* how to resolve the ambiguity. The dangling else problem is handled correctly by *bison*. It will give a shift/reduce warning and indicate it plans to shift. This is the correct action since it causes an else to be matched with the nearest unmatched if. Note that there are couple of ways to make this warning go away and still parse the grammar correctly.

3. If you put in some particular precedence OR associativity and *bison* tells you that it is meaningless or being ignored, REMOVE IT.

4. Errors. Your analyzer is expected to recognize **syntactic errors** but not semantic errors (like type checking and multiply defined variables). You will get your chance to do that later. That means do not worry about implementing symbol tables right now. Bison does provide error productions. A recommendation from the text is that error recovery be associated with "major" nonterminals that generate expressions, statements, methods, blocks and classes. As part of this assignment you must decide the set of nonterminals for which error productions are going to be specified. This will partially be trial and error. The goal is to choose a set that will allow you to generate meaningful messages and allow the parser to recover from an error without missing too many other errors. There is a trade-off between reporting as many errors as possible (very fine-grained set of nonterminals) and keeping the parser from getting hopelessly confused (coarse-grained selection). There are really no rules, it is mostly a matter of experience. That is one of the goals of this assignment, give you that experience.

   Be careful. Add the error productions one at a time. You may find adding one causes the generation of a large number of shift/reduce warnings. Do not let that large number of warnings continue.

   Note that **your** parser errors will try to include information (like filename and line number) on the input. This allows the user to better determine the cause of the error. "Parse error at end of input" may be all you can do about some errors but others should be more meaningful. Make sure that the line numbers are correct. Accurate column numbers will be part of a "perfect" score but will not be major deduction.

   **Lexical errors** should be printed out by the lexical analyzer (`yyFlexlExer.yyLex()`) and include the appropriate line and column number. Error tokens should **not** be passed to the parser. This may (probably will) cause parse errors but passing them to the parser would not be helpful either.

   I want a document explaining exactly what you are doing to handle all these errors. This document should detail the decisions you made on the specific error handlers/productions.

5. **Parse tree**. Your analyzer will internally build a parse tree. Although this code will be modified in later assignments to produce an abstract syntax tree, creating the code now will be a great start for later. That means that you will create a **data structure** that can be used to represent a parse tree. If you do not understand this type of data structure, you should REALLY ask before the due date of the assignment.

   We are programming in C++. This should be easy. Just create one or more or MANY classes that are all based on a very simple "base" class for the elements in the tree.

# 3 Output

As we are only using part of the grammar for this assignment, the "tree" you print won't be complete, but that is okay. Basically each variable declaration and each expression will be its own tree. BUT make sure that they are printed in the order in which they occur.

If you print out anything other than what I specify, I **will** make deductions. I need to be able to automate testing to some degree and extraneous output is a serious problem. Your output will consist of:

1. A set of error messages beginning at the left margin. Each error message should consist of

   (a) the line number where the error was detected,

   (b) the complete contents of the line, and

   (c) an *informative* error message describing the nature of the error.

   If the program has no errors, nothing should be printed for this part. An error message might look like

   ```
   345: x = read{};
                ^
   Invalid character
   ```

   In this case I managed to get the column number and placed the caret at the point on the line where I detected the error. YOU DO NOT HAVE TO HAVE COLUMN NUMBERS PRINTED YET. But you do need to try to figure out what the error is.

2. The parse tree. This is not a true tree structure. Each interior node of the parse tree should be printed by showing the production to which it was expanded. For instance if the node is *ClassDeclaration*, the production could be (nonterminals should be enclosed in angle brackets, <>):

   ```
   <ClassDeclaration> --> class identifier <ClassBody>
   <ClassBody> --> { ...
   ```

   The nodes should be printed in pre-order. This means that the productions that are output will form a leftmost derivation (read the text). Leaf nodes should not be printed, since they will occur on the right hand side of the appropriate productions. The exception to the leaf nodes are ID and NUMBER. Even though the grammar treats these as terminals, they must have a value associated with them so I expect to see something like `ID --> count` or `NUMBER --> 234`.

The two sections of the output should be distinct. That is the error output should occur first (as the input is processed) and the parse tree should follow. Do NOT print any headers or special comments to separate the sections. A blank line or two is plenty, but that is all. The output from the lexical assignment will be eliminated **except** for error messages with their corresponding line and column number.

# 4 What to turn in

1. Your source code **will** have comments including a heading in each file. See the C++ Style Guide on the main WyoCourses page. These should identify the file and explain any non-obvious operations. Feel free to comment as necessary to help yourself but do not comment every line.

2. The program will only read from standard input and write to standard output. Once created I should get the correct output by doing something simple like

   ```
   buckner %>./program3 < inputfile
   ```

3. If you want to add other options that is fine but do not print any "helpful" messages. DO NOT make me have to read your program OR some README file to figure out how to compile and execute the program. For example I had better be able to JUST type *make* and get something like

   ```
   $> make
   bison --report=state -W -d program3.ypp
   flex++ --warn program3.lpp
   g++ -ggdb -Wall -Wno-sign-compare program3.cpp program3_lex.cpp program3.tab.cpp\
           -o program3
   ```

4. Upload a single *tar* file to WyoCourses. The *tar* file will be named **program3.tar** OR **program3.tgz** if you compress it with gzip. (See the *tar* man page.) Do not archive a directory, just the individual files. The *tar* file will contain:

   - **program3.lpp**, the *flex* input file.
   - **program3.ypp**, the *bison* input file.
   - **program3.EXT**, the **documentation** of your error handling. In this case **EXT** is one of pdf, odt, doc, or docx. Remember that pdf documents must have embedded fonts so they can be correctly read on Linux.
   - **program3.cpp**, the main program.
   - Any other **.cpp** and **.hpp** files that are required for your implementation.
   - Do NOT turn in any of the files created by *g++*, *flex++* or *bison*. Those should be CORRECTLY re-created when I *make* the executable.

- A *make* file named **Makefile**. And it will have at least an additional "clean" target that removes the executable, the files generated by *flex* and *bison*, and object files, if any. It should also remove **core.*** files. The executable you create will be named **program3**.
- Do not place files in separate subdirectories or anything strange like that.
- Do not include test files, samples, or anything that is not absolutely required to create the executable or is not specified by this document.
- You may also include a **program3_readme.txt** that is plain text (readable in vim or other Linux text editor WITHOUT special settings, which means written on Linux NOT ON WINDOWS) and has any information that you think is pertinent about your submission. I should feel free to ignore this but I promise I will read it.

# 5 Range of input

I said earlier that this assignment would only cover a part of the decaf grammar. This is so that you do not become completely bogged down in details. Refer to discussions about ambiguity and a possible need to expand the given grammar. I have attached a modified version of the grammar that will work for this assignment. Do not add things to this grammar that are not part of the original grammar. I will grant that the semicolon following an expression is not in the grammar BUT for this assignment it is needed. And you need to be able to handle a semicolon by itself. In Java, C, and C++ that indicates an empty statement or variable declaration. Of course putting one in where it is not wanted will cause an error. But something like

```
int a;;
```

should NOT be an error.

Start at the bottom and work up. You have be able to recognize/handle an identifier before you can deal with a variable declaration. DO NOT type in a ton of rules and then try to fix all the huge number of errors. Get the simplest things working and build SLOWLY.

Use cout or cerr statements for troubleshooting, debugging, testing. REMOVE ALL THESE BEFORE TURNING THE CODE IN!!!!

Our modified syntax starts on the NEXT PAGE.

Modifed syntax only for Program 3

In this the "elements" production is temporary. Just for this assignment. It will be removed from the next. It is so that we can have more than one type of thing in the input.

The "SumOperators", "ProdOperators", "RelationalOperators", and "UnaryOperators" are JUST LABELS in the table. They ARE NOT meant to be part of your implementation.

| | | |
|---|---|---|
| program | → | elements |
| | \| | program elements |
| elements | → | vardec |
| | \| | exp SEMI |
| vardec | → | type ID SEMI |
| | → | ID ID SEMI |
| | → | type multibrackets ID SEMI |
| | → | ID multibrackets ID SEMI |
| exp | → | name |
| | \| | NUMBER |
| | \| | null |
| | \| | READ LPAREN RPAREN |
| | \| | newexp |
| | \| | LPAREN exp RPAREN |
| | \| | name LPAREN RPAREN |
| | | *SumOperators* |
| | \| | exp PLUS exp |
| | \| | exp MINUS exp |
| | \| | exp OR exp |
| | | *ProdOperators* |
| | \| | exp TIMES exp |
| | \| | exp DIV exp |
| | \| | exp MOD exp |
| | \| | exp AND exp |
| | | *RelationOperators* |
| | \| | exp EQ exp |
| | \| | exp NE exp |
| | \| | exp GE exp |
| | \| | exp LE exp |
| | \| | exp GT exp |
| | \| | exp LT exp |
| | | *UnaryOperators* |
| | \| | MINUS exp |
| | \| | PLUS exp |
| | \| | NOT exp |

| | | |
|---|---|---|
| newexp | → | NEW ID LPAREN RPAREN |
| | \| | NEW simpletype |
| | \| | NEW simpletype bracketexps |
| | \| | NEW ID bracketexps |
| | \| | NEW simpletype bracketexps multibrackets |
| | \| | NEW ID bracketexps multibrackets |
| type: | → | simpletype |
| simpletype | → | INT |
| bracketexps | → | bracketexp |
| | \| | bracketexps bracketexp |
| bracketexp | → | LBRACK exp RBRACK |
| multibrackets | → | LBRACK RBRACK |
| | \| | multibrackets LBRACK RBRACK |
| name | → | THIS |
| | \| | ID |
| | \| | name DOT ID |
| | \| | name bracketexps |