COSC 4785
Compiler Construction I
Programming Project 4, Finishing the Grammar

1 Introduction

Now that you something working with *bison*, you need to complete the task of parsing the entire Decaf grammar. This should be relatively easy. At least the learning curve will not be as steep. Make sure that your resulting program recognizes all the aspects of the Decaf grammar, and nothing more. Please do not put in "but I thought" productions that change the grammar.

2 Assignment

You will

1. Use *bison* and *flex* to implement a syntactic analyzer for all of the Decaf language that you have been given.
2. Ambiguity. The Decaf grammar is ambiguous. You will then 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 error and indicate it plans to shift. This is the correct action since it causes an else to be matched with the nearest unmatched if.
3. **Note** that the grammar has a change. The first $< NewExpression >$ production is changed.
4. You will probably need to rewrite the productions to eliminate the notational conventions such as * (Kleene closure) and + (Positive closure) that are not recognized by *bison*.
5. 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. So 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 might 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.

Lexical errors should be printed out by the lexical analyzer (`FlexlExer.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.

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. If you can figure out accurate column numbers that would be good as well but is not required.

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.

6. **Parse tree**. Your analyzer will 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.

3 Output

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 (if possible), 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 DO THAT. 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 would 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 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 should 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 (like your name, date, course at the beginning). 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 read from standard input and write to standard output. Once created I should get the correct output by doing something simple like Later assignments will handle multiple files.

   ```
   buckner %>./program4 < 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 the README file to figure out how to compile and execute the program. For example I had better be able to

   ```
   $> bison --report=state -W -d program4.y
   $> flex++ --warn program4.lpp
   $> g++ -ggdb -std=c++11 -Wall  program4.cpp program4_lex.cpp\
      program4.tab.c -o program4
   ```

4. Upload the following to WyoCourses.

   - **program4.lpp**, the *flex* input file.
   - **program4.y**, the *bison* input file.
   - **program4.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 and can be correctly read on Linux.
   - **program4.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 *flex* or *bison*. Those should be created when I create the executable.
   - If there are a large number of files and you have a **make** file, send that along. It can be named either "Makefile" or "makefile".
   - Do not place files in separate subdirectories or anything strange like that.
   - You may also attach a **program4_readme.txt** that is plain text (readable in vim or other Linux text editor WITHOUT special settings) and has any information that you think is pertinent.

5 Grammar

1. $Program \rightarrow ClassDeclaration^{+}$
2. $ClassDeclaration \rightarrow$ **class identifier** $ClassBody$
3. $ClassBody \rightarrow$ { $VarDeclaration^{*}$ $ConstructorDeclaration^{*}$ $MethodDeclaration^{*}$ }
4. $VarDeclaration \rightarrow Type$ **identifier** ;
5. $Type \rightarrow SimpleType \mid Type$ []
6. $SimpleType \rightarrow$ **int** $\mid$ **identifier**
7. $ConstructorDeclaration \rightarrow \epsilon \mid$ **identifier** ($ParameterList$) $Block$
8. $MethodDeclaration \rightarrow ResultType$ **identifier** ($ParameterList$) $Block$
9. $ResultType \rightarrow Type \mid$ **void**
10. $ParameterList \rightarrow \epsilon \mid Parameter < , Parameter >^{*}$
11. $Parameter \rightarrow Type$ **identifier**
12. $Block \rightarrow$ { $LocalVarDeclaration^{*}$ $Statement^{*}$ }
13. $LocalVarDeclaration \rightarrow Type$ **identifier** ;

14. *Statement* → ;
| *Name* = *Expression* ;
| *Name* (*Arglist*) ;
| **print** (*Arglist*) ;
| *ConditionalStatement*
| **while** (*Expression*) *Statement*‘
| **return** *OptionalExpression* ;
| *Block*

15. *Name* → **this**
| **identifier**
| *Name* . **identifier**
| *Name* [*Expression*]

16. *Arglist* → ϵ
| *Expression* $<$, *Expression* $>^*$

17. *ConditionalStatement* → **if** (*Expression*) *Statement*
→ **if** (*Expression*) *Statement* **else** *Statement*

18. *OptionalExpression* → ϵ
| *Expression*

19. *Expression* → *Name*
| **number**
| **null**
| *Name* (*ArgList*)
| **read** ( )
| *NewExpression*
| *UnaryOp Expression*
| *Expression RelationOp Expression*
| *Expression SumOp Expression*
| *Expression ProductOp Expression*
| (*Expression*)

20. *NewExpression* → **new** *identifier* (*Arglist*)
| **new** *SimpleType* $<$ [*Expression*] $>^*$ $<$ [] $>^*$

21. $UnaryOp \rightarrow + \mid - \mid !$

22. $RelationOp \rightarrow == \mid != \mid <= \mid >= \mid < \mid >$

23. $SumOp \rightarrow + \mid - \mid ||$

24. $ProductOp \rightarrow * \mid / \mid \% \mid \&\&$

5.1 Operator Precedence

Operator precedence is as follows (from lowest to highest):

1. RelationOp
2. SumOp
3. ProductOp
4. Unaryop

Within each group of operators, each operator has the same precedence.

5.2 Operator Associativity

All operators are left associative. For example, **a + b + c** should be interpreted as **(a + b) + c**.

6 Lexical conventions

1. Identifiers are unlimited sequences of letters, numbers and the underscore character. They *must* begin with a letter or underscore. They may contain upper and lower case letters in

any combination.

2. The *number* is an unsigned sequence of digits. It may be preceded with a unary minus operator to construct a negative number.

3. The decaf language has a number of reserved **keywords**. These are words that cannot be used as identifiers. As the language is also case sensitive, versions of the keywords containing uppercase letters can be used as identifiers. It is a compile time error to use a keyword as an identifier. Keywords are separated from **identifiers** by the use of whitespace or punctuation. The keywords are as follows:

int	**void**	**class**	**new**
print	**read**	**return**	**while**
if	**else**	**this**	

4. The following set of symbols are operators in the decaf language:

[	]	**{**	**}**
!=	==	<	>
<=	>=	**&&**	||
!	+	**-**	*****
/	%	;	,
(	)	=	//
.			

5. Whitespace is space, tab and newline. Other than using it to delimit keywords (and the newline for a
comment) it is not required. It will **not** be passed to the parser.