Decaf PP2: Syntax Analysis

Date Due: **02/23/2018 Friday 11:59pm**

1 Goal

In this programming project, you will extend the Decaf compiler to handle the syntax analysis phase, the second task of the front-end, by using *bison* to create a parser (PP1 is the lexical analysis). The parser will read Decaf source programs and construct an Abstract Syntax Tree (AST). If no syntax errors are found, your compiler will print the completed AST at the end of parsing. At this stage, you are not responsible for verifying semantic rules, just the syntactic structure. The purpose of this project is to familiarize you with the tools and give you experience in solving typical difficulties that arise when using them to generate a parser.

There are two challenges to this assignment: the first is all about yacc/bison: taking your SLR knowledge, coming up to speed on how the tools work, and generating a Decaf parser. The second challenge will come in familiarizing yourself with our provided code for building an AST tree. The code is just a starting skeleton to get you going. Your job will be to fully flesh it out over the course of the semester. Our sense is that in the long run you will be glad to have had our help, but you will have to first invest the time to come up to speed on someone else's code, so be prepared for that in this project.

2 Syntactical Structure of Decaf

The reference grammar given in the Decaf language handout defines the official grammar specification you must parse. The language supports global variables and functions, classes and interfaces, variables of various types including arrays and objects, arithmetic and boolean expressions, constructs such as if, while, etc. First, read the grammar specification carefully. Although the grammar is fairly large, most of is not tricky to parse.

3 Starter files

Starting files for this project are available in Blackboard. As always, be sure to read through the files we give you to understand what we have provided. The starting PP2 project contains the following files (the boldface entries are the ones you will definitely modify, you may modify others as needed):

Makefile	builds project
main.cc	main() and some helper functions
scanner.h	declares scanner functions and types
scanner.l	our scanner for Decaf
parser.h	declares parser functions and types
parser.y	skeleton of a yacc parser for Decaf
ast.h/.cc	interface/implementation of base AST node class
ast_type.h/.cc	interface/implementation of AST type classes
ast_decl.h/.cc	interface/implementation of AST declaration classes
ast_expr.h/.cc	interface/implementation of AST expression classes
ast_stmt.h/.cc	interface/implementation of AST statement classes
errors.h/.cc	error-reporting class to use in your compiler
list.h	simple list template class
location.h	utilities for handling locations, yylloc/yyltype
utility.h/.cc	interface/implementation of various utility functions
samples/	directory of test input files
solution/	solution executable of pp2

Use *make* to build the project. The makefile provided will produce a parser called *dcc*. It reads input from stdin, writes output to stdout, and writes errors to stderr. You can use standard UNIX file redirection to read from a file and/or save the output to a file:

```
% dcc < samples/program.decaf >& program.outanderr
```

You will need a scanner. You can either use the one you created or the one we provide. The *scanner.l* file is our lex specification for a Decaf scanner (with integrated comment handling and no preprocessor to keep things clean). Feel free to look through the code to learn how we did things, it's always interesting to see how different people solve the same problem. Same as last time, we put a solution executable of pp2 under the *solution* directory to help you understand the expected behavior of the parser.

4 Using bison

- The given *parser.y* input file contains a very incomplete skeleton which you must complete to accept the correct grammar. Your first task is to add the rules for each of the Decaf grammar features. You do not need the actions to build the AST yet, as it may help to first concentrate on getting the rules written and conflicts resolved before you add actions.

- Running yacc/bison on an incorrect or ambiguous grammar will report shift/reduce errors, useless rules, and reduce/reduce errors. To understand the conflicts being reported, scan the generated *y.output* file that identifies where the difficulties lie. Take care to investigate the underlying conflict and what is needed to resolve it rather than adding precedence rules like mad until the conflict goes away.

- Your parser should accept the grammar as given in the Decaf specification document, but you can rearrange the productions as needed to resolve conflicts. Some conflicts (if-then-else, overlap in function versus prototype) can be resolved in a multitude of ways (re-writing the productions, setting precedence, etc.) and you are free to take whatever approach appeals to you. All that you need to ensure is that you end up with an equivalent grammar.

- All conflicts and errors should be eliminated, i.e. you should not make use of bison's automatic resolution of conflicts or use *%expect* to mask them. Note: you might see messages in *y.output* like: "Conflict in state X between rule Y and token Z resolved as reduce." This is fine – it just means that your precedence directives were used to resolve a conflict.

5 Building the AST

- There are several files of support code (the generic list class, and the five AST files with various AST node classes). Before you get started on building the parse tree, read through these carefully. The code should be fairly self-explanatory. Each node has the ability to print itself and, where appropriate, manage its parent and lexical location (these will be of use in the later projects). Consider the starting code yours to modify and adapt in any way you like.

- We included limited comments to give an overview of the functionality in our provided classes but if you find that you need to know more details, don't be shy about opening up the *.cc* file and reading through the implementation to figure it out. You can learn a lot by just tracing through the code, but if you cannot seem to make sense of it on your own, you can contact us through Piazza or come to office hours.

- You can add actions to the rule as you go or wait until all rules are debugged and then go back and add actions. The action for each rule will be to construct the section of the AST corresponding to the rule reduced for use in later reductions. For example, when reducing a variable declaration, you will combine the Type and Identifier nodes into a VarDecl node, to be gathered in a list of declarations within a class or statement block at a later reduction.

- Be sure you understand how to use symbol locations and attributes in yacc/bison: accessing locations using @n, getting/setting attributes using $ notation, setting up the attributes union, how attribute types are set, the attribute stack, *yylval* and *yylloc*, so on. There is information on how to do these things (and much more) in the on-line references (http://dinosaur.compilertools.net/#flex and http://dinosaur.compilertools.net/#bison).

- Keep on your toes when assigning and using results passed from other productions in yacc. If the action of a production forgets to assign $$ or inappropriately relies on the default result ($$ = $1), you usually don't get warnings or errors, instead you are rewarded with entertaining runtime nastiness from using an unassigned variable.

- We expect you to match our output on the reference grammar, so be sure to look at our output and make good use of *diff -w*. At the end of parsing, if no syntax errors have been reported, the entire parse tree is printed using an in-order walk. Our AST classes are all configured to properly print themselves in the expected format, so there is nothing new you need to do here. If you have wired up the tree in the correct way, the printed version should match ours, line for line.

6 Testing

In the starting project, there is a *samples* directory containing various input files and matching *.out* files which represent the expected output that you should match. *diff* is your friend here!

Note that, *the provided test files do not test every possible case!* Examine the test files and think about what cases aren't covered. Make up lots of test cases of your own. Run your parser on various incorrect files to make sure it finds the errors. What formations look like valid programs but are not? What sequences might confuse your processing of expressions or class definitions? How well does your error recovery strategy stand up to abuse?

Remember that syntax analysis is only responsible for verifying that the sequence of tokens form a valid sentence given the definition of the Decaf grammar. Given that our grammar is somewhat "loose", some apparently nonsensical constructions will parse correctly and, of course, we are not yet doing any of the work for verify semantic validity (type-checking, declare before use, etc.). The following program is valid according to the grammar, but is obviously not semantically valid. It should parse correctly.

```
string binky()
{
      neverdefined b;

if (1.5 * "Stanford")
b / 4;
}
```

7 Hints

Some hints to help you code:

1. Our reference systems are the fox servers (fox01.cs.utsa.edu to fox06.cs.utsa.edu). Your assignments will be graded on these servers.
2. You can find The documents for *Bison* at http://dinosaur.compilertools.net/#bison. It is worthwhile to spend some time on *Bison*'s document before you start coding.
3. Additionally, there are some useful online *Flex/Bison* tutorials at here and here.
4. Google is always your best friend. You can find answers to almost all of your questions online.
5. The *parser.y* file is compiled by *flex* command, which generates two files, *y.tab.c* and *y.tab.h*. Essentially, *Bison* translates the *.y* file into C source files, which are later compiled to generate the scanner. Consequently, except the patterns, you will write mostly C code in the *parser.y* file.
6. A *yacc/bison* source file has similar structure as *flex*:

```
%{
   /* section 1: header files, macros and declarations that are directly
      copied into y.tab.c */
%}
   /* section 2: Yacc/Bison declarations */
%%
   /* section 3: Grammar rules */
%%
   /* section 4: user subroutines (C functions) that are directly copied
      into y.tab.c */
```

7. A rule defines a grammar production and the action associated with the production. That is, a rule is,

   ```
   grammar_production          {actions for the production in C code}
   ```

   For example, for variable declaration grammer, we may have a rule,

   ```
   VarDecl :  T_Int T_Identifier ';'   { $$ = new VarDecl(new Identifier(@2, $2),
                                                          Type::intType); }
           ;
   ```

   This rule creates a new AST tree node for the variable declaration with the identifier name (string value), the location of the identifier, and type of the identifier. The "@2" is the location variable (i.e., *yylloc*) of the second term (*T_Identifier*. The "$2" represents the value of of the second term. The "$$" is the value associated with the non-terminal symbol *VarDecl* after this particular reduction. For terminal tokens, their values are from the variable *yylval* set in the scanner. For non-terminal symbols, their values are set to the "$$" (which also refers to *yylval*) using corresponding actions. In general, "@n" and "$n" give the location and value of the *n*'th term of the right hand of a production.

   This example grammar rule is also provided in *parser.y*.

8. In the *Yacc/Bison* declaration part, you need to specify the type for *yylval*. *yylval* should be a C union of all possible types of the values of terminal and non-terminal symbols. The declaration part should also include list of terminal tokens and non-terminal types (symbols). Most of the terminal tokens are given in the *parser.y*, but you need to expend the list of non-terminal types.

9. The *Yacc/Bison* declaration part should also include operator precedence and associativity. You should read the document on precedence of *Bison* at here. Pay attention to unary operators, in particular, the negative sign.

10. As stated previously, your job is to write rules to insert correct tree nodes into the AST tree. The class declarations of each type of nodes are already provided to you (the *ast_∗* files).

11. Shift/reduce and reduce/reduce conflicts are probably the most annoying issues in this project. It is better to read about these conflcts before coding at here and here.

12. *Bison* also generates a *y.output* file which contains all the states of the resulting LALR parser. You can manually run the LALR algorithm with these states when debugging weird errors. You can also use "-t" with *Bison* to enable debug traces.

8 Grading

This project is worth 60 points and points will be allocated for correctness. We will run your program through the given test files from the samples directory as well as other tests of our own, using $diff - w$ to compare your output to that of our solution.

9 Deliverables

Electronically submit your entire project to Blackboard. You should submit a tar.gz of the project directory. Be sure to include a brief README file, which is your chance to let us know about optional features you added for bonus points.

Because we grade the submissions using scripts, it is important that everyone uses the same directory structure as the original *pp2.tar.gz*, as specified in Section 3. More specifically, inside your tar.gz, there should be a directory named "pp2", and all of your source files should be within the directory "pp2" without any sub-directories. There is no need to attach the *samples* directory. There is a 40% penalty for anyone who fails to follow this directory structure.