CS131 Compilers: Programming Assignment 2
Due Saturday, April 17, 2020 at 11:59pm

Fu Song Yiwei Yang Yangbiao Ji Cunhan You

1 Policy on plagiarism

These are individual homework. While you may discuss the ideas and algorithms or share the test cases with others, at no time may you read, possess, or submit the solution code of anyone else (including people outside this course), submit anyone else's solution code, or allow anyone else to read or possess your source code. We will detect plagiarism using automated tools and will prosecute all violations to the fullest extent of the university regulations, including failing this course, academic probation, and expulsion from the university.

2 Overview of the Project

Project assignments 1-4 will direct you to design and build a compiler for the Classroom Object-Oriented Language (cool), designed by Alexander Aiken for use in an undergraduate compiler course project. Assignments will cover the frond-end of the compiler: lexical analysis, parsing, semantic analysis, and intermediate code generation. Each assignment should be solved using C++ programming language and will ultimately result in a working compiler phase which can interface with other phases.

In this assignment, you will write a syntax analyzer, also called a parser, using a syntax analyzer generator Bison and the cool support package for manipulating trees. The parser needs as input the output of your completed lexer. In case that you do not complete the lexer, you might use provided reference lexer instead. The output of your parser will be an abstract syntax tree (AST for short) constructed using semantic actions of the parser generator. The AST will be used for semantic analysis in next project assignment.

You certainly will need to refer to the syntactic structure of cool, found in Figure 1 of The Cool Reference Manual (as well as other parts). We first should download and learn the documentation for Bison which is available online. The C++ version of the tree package is described in "A Tour of

Cool Support Code". You will need the tree package information for this and future assignments.

This assignment consists of three parts:

1. get familiar with cool support code for manipulating trees,
2. install and get familiar with Bison `https://www.gnu.org/software/bison/`,
3. write a syntax analyzer, also called a scanner, using a syntax analyzer generator Bison.

3 Files and Directories

To get started, go to `http://s3l.shanghaitech.edu.cn:8081/compiler/PA2`. The files that you will need to modify are:

- cool.y. This file contains a start towards a parser description for cool. You will need to add more rules. The declaration section is mostly complete; all you need to do is add type declarations for new non-terminals. (We have given you names and type declarations for the terminals.) The rule section is very incomplete.
- bison_test_good.cl and bison_test_bad.cl. These files test a few features of the grammar. You should add tests to ensure that bison_test_good.cl exercises every legal construction of the grammar and that bison_test_bad.cl exercises as many types of parsing errors as possible in a single file.
- Makefile: this file describes how to generate the binaries for parsing. You should not need to modify it. If you would like to understand Makefile, read `https://www.gnu.org/software/make/manual/make.html`.

Note that you will not hand in your modified bison bison_test_good.cl and bison_test_bad.cl files. However, it is important to make good tests to ensure that your parser is working properly.

Although these files are incomplete as given, the parser does compile and run. To build the parser, you must type

$$make\ parser$$

in the directory PA2/src. This will start the compilation process and link the support code needed for this phase into your working directory. Start the syntax analyser by typing

$$lexer\ input\_file \mid parser$$

4 Parser Output

Your semantic actions should build an AST using the cool support code tree package, whose interface is defined in */PA2/cool-support/include/cool-tree.h*. The Tour section of the cool manual contains an extensive discussion of the tree package for cool abstract syntax trees. You will need most of that information to write a working parser. Read the Tour section carefully: it contains explanations, caveats, and other details that will help you avoid a number of pitfalls in understanding and using the AST classes.

The root (and only the root) of the AST should be of type *program*. For programs that parse successfully, the output of parser is a listing of the AST.

For programs that have errors, the output is the error messages of the parser. We have supplied you with an error reporting routine that prints error messages in a standard format; please do not modify it. You should not invoke this routine directly in the semantic actions; Bison automatically invokes it when a problem is detected. For constructs that span multiple lines, you are free to set the line number to any line that is part of the construct.

5 Error Handling

You should use the *error* pseudo-nonterminal to add error handling capabilities in the parser. The purpose of *error* is to permit the parser to continue after some anticipated error. It is not a panacea and the parser may become completely confused. See the Bison documentation for how best to use error. Your test file bison_test_bad.cl should have some instances that illustrate the errors from which your parser can recover. To receive full credit, your parser should recover in at least the following situations:

1. If there is an error in a class definition but the class is terminated properly and the next class is syntactically correct, the parser should be able to restart at the next class definition.

2. Similarly, the parser should recover from errors in features (going on to the next feature), a let binding (going on to the next variable), and an expression inside a {...} block.

3. Do **not** worry if the lines reported by your parser do not exactly match the reference compiler. Also, your parser need only work for programs contained in a single file. **Don't** worry about compiling multiple files.

6 Testing the Parser

You will need a working scanner *lexer* to test the parser. You may use either your own scanner or reference scanner *lexer*. Don't automatically assume that the scanner (whichever one you use!) is bug free latent bugs in the scanner may cause mysterious problems in the parser. Bison produces a human-readable dump of the LALR(1) parsing tables in the *cool.output* file. Examining this dump may sometimes be useful for debugging the parser definition.

You should test this compiler on both good and bad inputs to see if everything is working. Remember, bugs in your parser may manifest themselves anywhere. Your parser will be graded using our lexical analyzer. Thus, even if you do most of the work using your own scanner you should test your parser with the our reference scanner before turning in the assignment.

7 Notes

1. Your parser input must **NOT** generate any warning about having ANY shift-reduce or reduce-reduce conflicts. All conflicts must be resolved, either by redesigning the grammar rules or by using bison's precedence declarations.

2. You may use precedence declarations, but only for expressions. Do not use precedence declarations blindly (i.e., do not respond to a shift-reduce conflict in your grammar by adding precedence rules until it goes away). If you find yourself making up rules for many things other than operators in expressions and for *let*, you are probably doing something wrong.

3. The cool *let* construct introduces an ambiguity into the language (try to construct an example if you are not convinced). The manual resolves the ambiguity by saying that a *let* expression extends as far to the right as possible. Depending on the way your grammar is written, this ambiguity may show up in your parser as a shift-reduce conflict involving the productions for *let*.

   This problem has a simple, but slightly obscure, solution. We will not tell you exactly how to solve it, but we will give you a strong hint. We implemented the resolution of the *let* shift-reduce conflict by giving low precedence to the token that controls the precedence of the relevant production.

   Since your compiler uses pipes to communicate from one stage to the next, any extraneous characters produced by the parser can cause errors; in particular, the next stage may not be able to parse the AST your parser produces.

4. You must declare Bison "types" for your non-terminals and terminals that have attributes. For example, in the skeleton *cool.y* is the declaration:
   %type ⟨program⟩ program
   This declaration says that the non-terminal program has type ⟨program⟩. The use of the word "type" is misleading here; what it really means is that the attribute for the non-terminal *program* is stored in the *program* member of the *union* declaration in *cool.y*, which has type *Program*. By By specifying the type
   %type ⟨member_name⟩ X Y Z ...
   you instruct Bison that the attributes of non-terminals (or terminals) *X*, *Y*, and *Z* have a type appropriate for the member *member_name* of the union. All the union members and their types have similar names by design. It is a coincidence in the example above that the non-terminal *program* has the same name as a union member. It is critical that you declare the correct types for the attributes of grammar symbols; failure to do so virtually guarantees that your parser won't work. You do not need to declare types for symbols of your grammar that do not have attributes.

   The g++ type checker complains if you use the tree constructors with the wrong type parameters. If you ignore the warnings, your program may crash when the constructor notices that it is being used incorrectly. Moreover, Bison may complain if you make type errors. Heed any warnings. Don't be surprised if your program crashes when bison or g++ give warning message

8 When, What and How to Hand in

1. When: Make sure that the final version you submit does not have any debug print statements and that your syntax specification is complete.

2. What: You have to hand in all files that you modify in this assignment. That is cool.y. Don't copy and modify any part of the support code! In addition to looking at the test results, we will manually look at your code. 5% of the grade is for commenting your code. Another 5% is for appropriate usage of Bison features.

3. How: Every commit to the gitlab will grade your code automatically. If you don't want grading every time, you can develop your code in the developing branch. You are allowed to push after 3.20.