Programming Assignment III
Due Thursday, October 12

1   Introduction

In this assignment you will write a parser for Cool. The assignment makes use of two tools: the parser generator **bison** and a package for generating and manipulating trees. You will not use use the generating part of the package, but only the manipulators of a (already generated) tree definition we will give you. The output of your parser will be an abstract syntax tree (AST). The AST will be constructed using **bison**'s semantic actions.

You certainly will need to refer to the syntactic structure of Cool, found in Figure 1 of the CoolAid, as well as other portions of the reference manual. There is a chapter on **bison** in the course reader. There is also a section in the textbook on **yacc**, which is a close relative of **bison**. The tree package is described in the *Tour* handout. You will need the tree package information for this and future assignments.

There is a lot of information in this handout and you need to know most of it to write a working parser. Please read the handout thoroughly.

2   Files and Directories

To get the assignment type

```
gmake -f ~cs164/assignments/PA3/Makefile
```

in a directory named **PA3**. This command copies a number of files to your directory, some of them with read-only permission. As usual, you should not modify files that are read-only. If you insist on making your own versions of these files, you are on your own. If your code doesn't compile with the original files (including the original **Makefile**) you will receive little credit, because your program will fail all of the automatically run test cases. Please read and follow the directions in the **README** file.

The files that you 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 section before the first %% is mostly complete; all you need to do is add %type declarations for new nonterminals. (We have given you names and %type declarations for the terminals.) The rule section is very incomplete.

- **good**.cl and **bad**.cl
  These test a few features of the grammar. You should add tests to ensure that **good**.cl exercises every legal construction of the grammar and that **bad**.cl exercises as many types of parsing errors as possible in a single file. Explain your tests in these files and put any overall comments in the **README** file.

- **README**
  As usual, this file will contain the write-up for your assignment. Explain your design decisions, your test cases, and why you believe your program is correct and robust. It is part of the assignment to explain things in text as well as to comment your code. Make sure that the name, student ID, and login of each group member is in the **README** file.

The files that your code will link with and include, but that you should not modify, are:

- parsetest.cc
  This file contains a simple driver for the parser.

- cool-tree.aps
  This file is a specification of the Cool tree language written in a special language. It defines a type system and describes the kinds of nodes used for the abstract syntax trees.

- cool.h, cool-tree.h, cool-tree.handcode.h, and tree.h
  The header file cool.h defines basic types used by the compiler. The other files define the routines that let you create and examine abstract syntax trees and lists of trees. It may interest you that the second file was automatically generated from cool-tree.aps.

- dumptype.cc
  This file implements a routine for printing out ASTs. The algorithm used illustrates the use of virtual functions to recursively traverse of an AST; you will need to write algorithms in a similar style in future assignments.

- handle_files.cc
  This file implements a simple mechanism for parsing multiple input files.

- parsetest.a
  This is the object code for handle_flags.cc and handle_files.cc.

- lexer.a
  This is the object code for the coolc lexical analyzer.

- mylexer.a
  This is an alternative to lexer.a. Use this library if you want to run the parser with your own lexical analyzer (see README for details).

Important: All software supplied with this assignment is supported on both the HP and DEC machines. Remember to run gmake clean if you switch architectures. To make sure your code will link correctly on both architectures, you should place cs164/bin at the beginning of your PATH variable.

3   Testing the Parser

You will need a working scanner to test the parser. You may use either your own scanner or the coolc scanner. See the README file for instructions on how to build a working parser either way. 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.

Note that parsetest has a -p flag for debugging the parser; using this flag causes lots of information about what the parser is doing to be printed on stdout. In addition, the -l flag is available for generating debugging output from the scanner. Using these two flags together is sometimes useful for isolating strange behavior in the parser.

Once you are confident that your parser is working, try make mycoolc to build a complete compiler that includes your parser. 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 coolc scanner before turning in the assignment. As always, type make turnin to turn in your assignment; see the README file.

4   Parser Output

Your semantic actions should build an AST for the input. The root (and only the root) of the AST should be of type Program. The semantic action for the start symbol assigns the root of the AST to a global variable ast_root (also of type Program, of course). For programs that parse successfully, the output of parsetest 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 a function yyerror in cool.y that prints error messages in a standard format; please do not modify this function. You should not invoke yyerror directly in the semantic actions; bison automatically invokes yyerror when it detects a problem.

Your parser need only work for programs contained in a single file—don't worry about compiling multiple files.

5   Error Handling

You should use bison's 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. In your README, describe which errors you attempt to catch. Your test file bad.cl should have some instances that illustrate the errors from which your parser can recover. To receive full credit, you parser should recover in at least the following situations:

- 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.

- 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 begin-end block.

    Do not be overly concerned about the the line numbers that appear in the error messages your parser generates. If your parser is working correctly, the line number will generally be the line where the error occurred. For erroneous constructs broken across multiple lines, the line number will probably be the last line of the construct.

6   The Tree Package

There is an extensive discussion of the APS tree package for Cool abstract syntax trees in the *Tour*. You will need most of that information to write a working parser.

7   Remarks

You may use bison's precedence declarations, but only for expressions. Do not use precedence declarations blindly—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 things other than operators in expressions, you are probably doing something wrong.

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 <class_> class
```

This declaration says that the non-terminal **class** has type **<class_>**. The use of the word "type" is misleading here; what it really means is that the attribute for the non-terminal **class** is stored in the **class_** member of the **union** declaration in **cool**.y, which has type **Class_**. 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 **class** 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 messages.