Cairo University
Faculty of Computers & Information
Computer Science Department
Year 2014 – 2015
Second Term

Assignment-2: Cool Parser

Based on Compilers Assignments at Stanford University

1. Introduction

In this assignment you will write a parser for Cool. The assignment makes use of two tools: the parser generator (i.e. bison) and a package for manipulating trees. The output of your parser will be an abstract syntax tree (AST). You will construct this AST using semantic actions of the parser generator. You certainly will need to refer to the syntactic structure of Cool, found in Figure 1 of The Cool Reference Manual. You can find it there: http://www.cs.virginia.edu/~cs415/cool-manual/node39.html.Yet, you may need to check other parts in the manual for completeness.

The documentation for bison is available online. You can find it there: https://www.gnu.org/software/bison/manual/bison.html

The C++ version of the tree package is described in the Tour of Cool Support Code handout. You can find it there: http://web.stanford.edu/class/archive/cs/cs143/cs143.1112/materials/handouts/cool-tour.pdf

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.*

You must work individually or in teams of up to 3 members.

2. Files and Directories

To get started, create a directory where you want to do the assignment and execute the following commands in that directory:

```
make -f /usr/class/cs143/assignments/PA3/Makefile
```

Important: you will need to make one slight change to our code before it will link. Please comment out line 29 of the file parser-phase.cc (which you should not otherwise modify), so that it looks like:

```
//int curr_lineno;    // needed for lexical analyzer
```

This command will copy a number of files to your directory. Some of the files will be copied read-only (using symbolic links). You should not edit these files. In fact, if you make and modify private copies of these files, you may find it **impossible** to complete the assignment. See the instructions in the README file. The only file that you will need to modify is:

- `cool.y`.

This file contains a start towards a parser description for Cool. The declaration section is mostly complete, but you will need to add additional type declarations for new nonterminals you introduce. We have given you names and type declarations for the terminals. You might also need to add **precedence** declarations. The rule section, however, is rather incomplete. We have provided some parts of some rules. You should not need to modify this code to get a working solution, but you are welcome to if you like. However, **do not assume** that any particular rule is complete.

- `good.cl and bad.cl`

These files test a few features of the grammar. Feel free to modify these files to test your parser.

- `README`

This file contains detailed instructions for the assignment as well as a number of useful tips.

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. By default, the `coolc` scanner is used; to change this behavior, replace the `lexer` executable (which is a symbolic link in your project directory) with your own scanner. 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.

You will run your parser using `myparser`, a shell script that "glues" together the parser with the scanner. Note that `myparser` takes 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`. `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 `coolc` scanner before turning in the assignment.

4. Parser Output

Your semantic actions should build an AST. 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. 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 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 one-for-all solution and the parser may become completely confused. See the bison documentation for how best to use `error`. To receive full credit, your 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 `{...}` block.

Do not be overly concerned about 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 C++ version of the tree package for Cool abstract syntax trees in the `Tour of the Cool Support Code` documentation. You will need most of that information to write a working parser. The link is provided in the first page.

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.

7. A Demo

1. Create a folder named "`phase2`".
2. Open the terminal and run the following commands:
   ```
   $ cd phase2
   $ make -f /usr/class/cs143/assignments/PA3/Makefile
   ```

3. Open `cool.y` file and familiarize yourself to its content. `cool.y` file contains some very important notes that you'll need to complete your parser. It defines some basic grammar rules to enable you to parse an empty class. You'll need to complete the remaining yourself.

4. We'll try the `cool.y` file as it's. Close it and create a new file named "`mytest.cl`". Write the following content to it.
   ```
   class A{

   };

   class B{

   };
   ```

5. In terminal, run the following commands
   ```
   $ make parser
   $ ./myparser mytest.cl
   ```

6. An AST of the program is printed to the terminal where the root is the program, and the children are the classes A, B.

7. You're to complete the implementation of the Cool grammar rules in `cool.y` file to be able to parse complete Cool codes.

8. You may need to look at `README` file as well as the comments in `cool.y` file for better understanding of what's going on.

8. How to Submit

Submissions and discussions are in lab time. No online submissions required. The assignment is published on `4 April` week labs, and will be discussed on `11 April` week labs.

9. What to Submit

Be ready with the following:

- A folder containing your modified `cool.y` file, as well as the rest of the generated files.
- At least 3 small test files. Name them `test1.cl`, `test2.cl`, .. etc.

Good Luck,