Programming Assignment II
Due Tuesday, April 26, 2022 at 11:59pm

1 Introduction

In this assignment you will write a parser for Cool. The assignment makes use of two tools: a parser generator called 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 (as well as other parts). The documentation for bison is available online, and the tree package is described in the *Tour of Cool Support Code* 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.*

As before, you may work individually or as a pair for this assignment.

2 Files and Directories

To get started, log in to one of the *myth* machines and run one of the following commands:

```
/usr/class/cs143/bin/pa_fetch PA2 <project_directory>
```

The project directory you specify will be created if necessary, and will contain a few files for you to edit and a bunch of symbolic links for things you should not be editing. (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 files that you will need to modify are:

- 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 the 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. For your own understanding, put comments in these files to explain your tests (although the test files will not be graded).

- README

  This file contains detailed instructions for the assignment as well as a number of useful tips. You should edit this file to include your SUNet ID(s) (see Section 9 for details).

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

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. 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.

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 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:

- 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* section of the Cool documentation. You will need most of that information to write a working parser.

7 Remarks

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).

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`. If you run into such a conflict, you might want to consider solving the problem by using a bison feature that allows precedence to be associated with productions (not just operators). See the bison documentation for information on how to use this feature.

Since the mycoolc compiler uses pipes to communicate from one stage to the next, any extraneous characters produced by the parser can cause errors; in particular, the semantic analyzer may not be able to parse the AST your parser produces. **Since any prints left in your code will cause you to lose many points, please make sure to remove all prints from your code before submitting the assignment.**

8 Implementation Notes

- 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 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.

- Running bison on the initial skeleton file will produce some warnings about "useless nonterminals" and "useless rules". This is because some of the nonterminals and rules will never be used, but these *should* go away when your parser is complete.
- 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.

9 What to Turn In

Before submitting, please ensure you have done the following:

- Please make sure that the final version you submit does not have any debug print statements.
- Please remove the instructions section in the README. This section is the one that starts at the beginning, and runs till the following line (inclusive of this line)

  ```
  ---8<------8<------8<------8<---cut here---8<------8<------8<------8<---
  ```

  This section of the README is only to provide you additional guidance about which files you need to edit, and you must remove it before actually submitting the assignment.
- Ensure that you have edited the README so that the first line includes your username in the following format:

  ```
  user: <your_sunet_id>
  ```

  If you are working in a group of two, there should be TWO separate user lines- the submission script will fail if both the SUNet IDs are contained on the same line. In particular, if two students with SUNet IDs abcdef and bcdefg work together, their README should mention the SUNet IDs in the following way:

  ```
  user: abcdef
  user: bcdefg
  ```

  The important thing to note is that the following will NOT work:

  ```
  user: abcdef, bcdefg
  ```

Your SUNet ID is the name you use to log in to the myth machines (not your 8-digit student ID number), and can be queried with the `whoami` command.

- Once the README is ready, run the following from your project directory:

  ```
  make submit
  ```

  This will package up most of the files in your project directory (it will ignore things like core dumps and Emacs backup files) and copy them into the submission folder, along with a time stamp.

  The last version of the assignment you submit will be the one graded. Each submission overwrites the previous one. However, we reserve the right to retain older submissions for reference in case of any disputes. Remember that you have three late days to use for the entire quarter; please refer to the late policy page on the course website for details.

  The burden of convincing us that you understand the material is on you. Obtuse code may have a negative effect on your grade, so take the extra time to make your code readable.

- **Important:** We will only accept assignments turned in via the submit script. Please test-submit your assignment at least 24 hours before the deadline to be sure that the script is working for you, even if you are not finished. It is your responsibility to check for problems ahead of time.