Machine Problem 2

CS 426 — Compiler Construction
Fall Semester 2014

Handed Out: September 9, 2014. Due: September 30, 2014, 10:00 P.M.

In this MP, you will write a recursive descent parser for *cool* in C++. The output of your program will be an abstract syntax tree (AST) if the program is syntactically correct, else a series of error messages.

1 Provided Files

Familiarize yourself with the files in *mp2/src*, which contains the main source files for MP2. These files include:

- *Makefile:* this file describes how to generate the binaries for parsing your source files. You should not need to modify it.
- *cool-parser.cc:* a skeleton parser that you will need to modify completely to write your parser.
- *parser_test_good.cl:* A very simple test file for your parser that doesn't produce any error. You should write your own test files to make sure your parser is working correctly.
- *parser_test_bad.cl:* A simple test file for your parser, that produces errors. Make your parser detect these errors and generate error messages for them. Write your own test files with multiple errors in each file to test your parser's error reporting ability.

The files that you will need to modify are:

- *cool-parser.cc*
  This file contains a skeleton parser for *cool*. Right now, it parses programs which have empty class definitions (no features).

  The function `cool_parse()` should parse the input and generate the AST. The root of the AST should be of type `Program` and must be assigned to the variable `ast_root` (this has been done for you).

  Functions `lookNextToken()` and `consumeNextToken()` can be used to look at and consume next tokens. You can use these functions as provided.

  Feel free to modify anything in this file, but make sure it works as expected with other provided files.
- *parser_test_good.cl* and *parser_test_bad.cl*
  These files test a few features of the grammar. You should add tests to ensure that *parser_test_good.cl* exercises every legal construction of the grammar and that *parser_test_bad.cl* exercises as many types of parsing errors as possible in a single file.

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

The supplied file *parser-phase.cc* contains the `main` program for the parser. You may not modify this file, but it may help you to see how your parser will be called.

2 Compiling and running the parser

To build the parser, type

$ make parser

in the directory *mp2/src*. This will link some more files of support code to your directory and compile your skeleton. Your parser needs as input the output of your completed lexer. So first complete the lexer (MP1), then you can start working on your parser. You can test your parser with a cool program by typing

$ lexer input_file | parser

We will provide a reference binary for *parser* one week before the due date (September 23, 2014) and an extensive set of tests a few days before the due date (September 26, 2014). Try to use them only after you are really done with your own testing. Your goal should be that the reference binaries uncover no new bugs.

3 Parser Output

Your parser should build an AST using the *cool* support code tree package, whose interface is defined in *mp2/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 `yyerror()` that prints error messages in a standard format; please do not modify it.

You should invoke it each time you encounter an error while reading a token. Details of error reporting are described in the next section.

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

4 Error Handling

The given skeleton parser terminates parsing on encountering an error. You should add error handling capabilities in the parser. Your test file *parser_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:

- 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.

4.1 Reporting Errors

Two types of error reports must be generated.

- Generic error report : Use the function `yyerror()` to report any parser errors while reading a token. You should not call this function multiple times with the same token.
- Verbose error report : Use the function `printerr()` to provide detailed error information, for example, "Class name should begin with a capital letter" if you encounter an OBJECTID instead of a TYPEID in class definition. The commandline flag -v prints verbose errors, while the flag -q suppresses it (default does not print verbose errors).

To receive full credit, you should atleast report detailed errors for 8 - 10 different errors.

5 Testing the Parser

Don't automatically assume that the scanner is bug free — latent bugs in the scanner may cause mysterious problems in the 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.

6 Notes

- You need to convert the given grammar from *Figure 1* of the *Cool Manual* to ensure it is not left recursive.

- You need to ensure the AST generated follows the precedence rules defined in the *Cool Manual*.
- 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.
- 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.

7 Further ways to use your Makefile

- *make* builds the parser.
- *clean* removes all compiled files.
- *realclean* is like clean, but also removes all links. Use this if you really want to start with just your source files.

Documentation on *make* can also be downloaded from the course web page.

8 What and how to hand in

You have to hand in *cool-parser.cc* that you modify in this MP.

Don't copy and modify any part of the support code! The provided files are the ones that will be used in the grading process.

Important: We will also use our version of the lexer to test and grade your parser. So make sure your parser also works with our lexer.

The details on the hand in procedure are posted on the Assignments page on the course website.