Machine Problem 1

CS 426 — Compiler Construction
Fall Semester 2015

Handed Out: August 31, 2015. Due: September 14, 2015, 5:00 p.m.

The machine problems for this semester will deal with implementing a compiler for the Classroom Object-Oriented Language (*cool*). Your compiler will need to cover the full set of *cool* instructions and detect and report errors in the input files if there are any. Correct programs will be translated into *llvm*.

This MP deals with the front end only. So your programs have to process input files in *cool* and build the abstract syntax tree (AST). The assignment is divided into three parts: (1) get familiar with *cool*, its support code, and the files we provide for this MP; (2) write a scanner or lexical analyser (lexer); and (3) write a parser.

1 Cool and Provided Files

1. To get familiar with *cool* read the documentation *CoolAid: The Cool Reference Manual*, available under the Resources link on the course web page.

   Write a simple example program in *cool* to get familiar with the language. You don't have to hand in this program and you won't get points for it, but understanding the language is important for the rest of the MP.

   If you need further examples, there is a link to a set of example *cool* programs, together with a README file and expected output files. Read the README file to learn what the example programs are about, and read the comments in the example files for further help.

2. Download and unpack the *mp1* directory from the Project link on the class web site. This directory contains all the code that you will need for this MP.

3. Become familiar with the *cool* support code, which provides several data structures that you will use in writing your *cool* compiler, including all the AST classes. You will compile this code and link against it for each of the parts of this project. Download the document *A Tour of the Cool Support Code* from the course web page and read and understand it. If you want to examine the support code source files you can find them in the directories *cool-support/include* (header files) and *cool-support/src* (implementation files) within the directory *mp1*.

4. Familiarize yourself with the files in *mp1/src*, which contains the main source files for MP1. These files include:

   - *Makefile:* this file describes how to generate the binaries for scanning and parsing your source files. You should not need to modify it.
   - *cool.flex:* a skeleton flex input file that you will need to extend to write your lexical analyser.
   - *cool.y:* a skeleton bison input file that you will need to extend to write your parser.
   - *flex_test.cl:* a very simple cool program for the first scanner test. As this file doesn't cover all elements of *cool*, you need to write your own examples to make sure your lexer processes the full set of *cool* tokens.

- *bison_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.
- *bison_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 compiler consists of several phases. Each phase will be compiled into its own binary file. It will take its input from standard input (except the lexer which takes a commandline input filename) and write to standard output. So in the end to start your compiler you will call

lexer input_file | *parser* | *semant* | *cgen* > *output_file*

For this assignment you are making the lexer and parser. Reference binaries for *lexer*, and *parser* will be provided four days before the due date, but try to use them only after you are really done with your own testing. You should test your compiler passes extensively with your own tests. We believe that writing a comprehensive and well thought out set of tests is a critical part of good programming practice. Your goal should be that the reference binaries uncover no new bugs.

2 Scanner

Write a lexical scanner for *cool* using *flex*. To do this, first read the documentation on *flex*, which you can download from the Resources section of the course web page.

2.1 Files

The files that you will need to modify are:

- *cool.flex*
  This file contains a skeleton for a lexical description for *cool*. You can actually build a scanner with this description but it does not do much. You should read the *flex* manual to figure out what this description does do. Any auxiliary routines that you wish to write should be added directly to this file in the appropriate section (see comments in the file).

- *flex_test.cl*
  This file contains some sample input to be scanned. It does not exercise all of the lexical specification but it is nevertheless an interesting test. It is not a good test to start with, nor does it provide adequate testing of your scanner. Part of your assignment is to come up with good testing inputs and a testing strategy. (Don't take this lightly – good test input is difficult to create, and forgetting to test something is the most likely cause of lost points during grading.)

  You should modify this file with tests that you think adequately exercise your scanner. Our *flex_test.cl* is similar to a real *cool* program, but your tests need not be. You may keep as much or as little of our test as you like.

Note that you will not hand in your modified *flex_test.cl* file. However, it is important to make a good test to ensure that your lexer is working properly.

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

2.2 Compiling and running the scanner

To build the lexer type

make lexer

in the directory *mp1/src*. This will start the compilation process and link the support code needed for this phase into your working directory.

Start the lexical analyser by typing

lexer input_file

2.3 Scanner Results

You should follow the specification of the lexical structure of *cool* given in Section 10 and Figure 1 of the CoolAid. Your scanner should be robust—it should work for any conceivable input. For example, you must handle errors such as an EOF occurring in the middle of a string or comment, as well as string constants that are too long. These are just some of the errors that can occur; see the manual for the rest.

You must make some provision for graceful termination if a fatal error occurs. Core dumps are unacceptable.

Programs tend to have many occurrences of the same lexeme. For example, an identifier generally is referred to more than once in a program (or else it isn't very useful!). To save space and time, a common compiler practice is to store lexemes in a *string table*. We provide a string table implementation.

All errors will be passed along to the parser. The *cool* parser knows about a special error token called *ERROR* which carries an error message to communicate errors from the lexer to the parser. There are several requirements for reporting and recovering from lexical errors.

Most of these situations should be reported by returning an error token with some human-readable message describing the problem as the error string.

- When an invalid character (one which can't begin any token) is encountered, a string containing just that character should be returned as the error string. Resume lexing at the following character.
- When a string is too long, or contains invalid characters, report that. Lexing should resume after the end of the string. Do not produce a string token before the error token.
- If a string contains an unescaped newline, report that, and resume lexing at the beginning of the next line — we assume the programmer simply forgot the close-quote. Do not produce a string token before the error token.
- If a comment remains open when EOF is encountered, report that. Do **not** tokenize the comment's contents simply because the terminator is missing. (This applies to strings as well.)
- If you see "`*)`" outside a comment, report this as an unmatched comment terminator, rather than tokenzing it as `*` and `)`.

- Do **not** test whether integer literals fit within the representation specified in the *cool* manual — simply use the *add_** functions, which create a *Symbol* with the entire literal's text as its contents, regardless of its length. *Symbol* (a typedef defined in stringtab.h) is a pointer to *Entry*, which is a wrapper around a string.

There is an issue in deciding how to handle the special identifiers for the basic classes (*Object*, *Int*, *Bool*, *String*), *SELF_TYPE*, and *self*. However, this issue doesn't actually come up until later phases of the compiler—the scanner should treat the special identifiers exactly like any other identifier.

Your scanner should maintain the variable *curr_lineno* that indicates which line in the source text is currently being scanned. This feature will aid the parser in printing useful error messages.

Finally, note that if the lexical specification is incomplete (some input has no regular expression that matches) then the scanner generated does undesirable things. Make sure your specification **is** complete.

2.4 Notes

- Each call on the scanner returns the next token and lexeme from the input. The value returned by the function *cool_yylex* is an integer code representing the syntactic category: whether it is an integer literal, semicolon, the *if* keyword, etc. The codes for all tokens are defined in the file *cool-parse.h*. The second component, the semantic value or lexeme, is placed in the global union *cool_yylval*, which is of type YYSTYPE. The type YYSTYPE is also defined in *cool-parse.h*. The tokens for single character symbols (e.g., ";" and ",", among others) are represented just by the integer (ASCII) value of the character itself. All of the single character tokens are listed in the grammar for *cool* in the CoolAid.
- For class identifiers, object identifiers, integers and strings, the semantic value should be a *Symbol* stored in the field *cool_yylval.symbol*. For boolean constants, the semantic value is stored in the field *cool_yylval.boolean*. Except for errors (see below), the lexemes for the other tokens do not carry any interesting information.
- We provide you with a string table implementation, which is discussed in detail in *A Tour of the Cool Support Code* and documentation in the code. For the moment, you only need to know that the type of string table entries is *Symbol*.
- When a lexical error is encountered, the routine *cool_yylex* should return the token *ERROR*. The semantic value is the string representing the error message, which is stored in the field *cool_yylval.error_msg* (note that this field is an ordinary string, not a symbol). See previous section for information on what to put in error messages.

3 Parser

Use *bison* to generate a parser for *cool* and build the AST.

First download the documentation on *bison* from the course web page and read it. Then take a look at the file *cool.y* and understand it. This is the skeleton for the second phase of your compiler — the parser.

3.1 Files

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 nonterminals. (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.

Note that you will not hand in your modified *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.

3.2 Compiling and running the parser

To build the parser, type

```
make parser
```

in the directory *mp1/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, then you can start your parser. Use *bison_test_good.cl* to test your skeleton parser with a working cool program by typing

```
lexer bison_test_good.cl | parser
```

3.3 Parser Output

Your semantic actions should build an AST using the *cool* support code tree package, whose interface is defined in *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.

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

3.4 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:

- 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.

3.5 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. *bison* produces a human-readable dump of the LALR(1) parsing tables in the *cool.output* file (see the *bison* manual). Examining this dump is frequently 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.

3.6 Notes on the Parser

- 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.
- 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.

  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. The ambiguity will 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.

- 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.

  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.

4 Further ways to use your Makefile

- *make* builds the lexer and 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.

5 What and how to hand in

You have to hand in all files that you modify in this MP. That is

- *cool.flex*
- *cool.y*

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 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 will be posted on Piazza and the project page on the course website.