Lab 0 – Cool Frontend (Bonus)

ECE 479K — Compiler
Spring Semester 2024

Out: Jan 20th   Due: End of the semester

In this lab you will write (most) of the compiler front-end for the Classroom Object-Oriented Language (*Cool*). Your implementation will process input files in Cool and build the abstract syntax tree (AST), or report syntax errors. The assignment is divided into three parts: (1) get familiar with Cool, its support code, and the files we provide for this lab; (2) write a lexical analyser (lexer, also called scanner); and (3) write a parser.

1 Cool and Provided Files

Compilers work in passes (phases) where the output of one pass is the input to the following pass. For simplicity, you will use pipes to pass outputs to inputs and write each pass as an executable. That is, the lexer will accept a file as its input (a filename) and produce its output onto STDOUT. The parser will accept input from STDIN and produce the AST onto STDOUT. Later passes will operate similarly from STDIN to STDOUT. For this lab, you will call:

```
lexer input_file | parser > outputfile
```

The outputfile contains the printout of the AST.

You will be implementing the `lexer` and `parser` for this assignment. Here's how we strongly recommend you to get started:

1. Get familiar with the Cool language by reading the documentation CoolAid: The Cool Reference Manual, and write a simple example program in Cool. 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 lab.
2. Download the starter file `lab0_starter.tgz` from the lab tab on Canvas. This tarball contains all the code that you will need for Lab0. For detailed instructions on how to use this code and setup the environment for doing this lab, refer to this doc.
3. Get 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. A Tour of the Cool Support Code walks you through an earlier version of the Cool support code. While it is a good starting point, you should also read the code itself and the comments in it, in the directories `cool-support/include` (header files) and `cool-support/src` (implementation files) in the code handout.
4. Familiarize yourself with the files in `src/`, which contains the main source files for lab0. 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 simple Cool program to test your lexer on. 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.

2 Lexer

Write a lexical analyser for Cool using *Flex*. In addition to what is covered in the lectures, refer to the Flex manual for detailed usage.

2.1 Files

The files that you will need to modify are:

- `src/cool.flex`

  This file contains a skeleton for a lexical description for Cool. Without any changes it does compile and produce a `lexer`, but it does not do much. You should read the comments in this file together with the Flex manual to figure out what to 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).

- `src/flex_test.cl`

  This file is a sample Cool program that you can use to test your lexer. It is an interesting test case, but it is not a good test to start with, nor does it provide adequate testing. 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.)

  Note that you will not hand in your modified `flex_test.cl` file. You should modify this file in whichever way you think will adequately test your lexer.

The Cool support file `cool-support/src/lex_main.cc` contains the `main` program for the lexer. You may not modify this file, but this information may help you to see how your lexer will be called.

2.2 Compiling and Running the Lexer

To build the lexer, type `make lexer` in the directory `src/`. This will invoke Flex to convert `cool.flex` into a C file, compile all the C/C++ files needed, and link them together into the final executable `lexer`.

To run the lexer, simply type `lexer input_file`. `lexer -h` shows the usage of lexer including more available options, but they won't be necessary.

2.3 Scanner Behavior

You should follow the specification of the lexical structure of Cool given in Section 10 and Figure 1 of the CoolAid. Your lexer 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*. Our lab0 code handout includes 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 lexer should treat the special identifiers exactly like any other identifier.

Your lexer 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 lexer generated does undesirable things. Make sure your specification **is** complete.

2.4 Notes

- Each call on the lexer 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

You have two options for implementing the parser – the second phase of your compiler – and build the AST. The recommended approach is to use *Bison* to generate a Cool parser. Bison is the standard tool for this purpose and can generate a parser for LR languages. As an alternative, you may write your own top-down (recursive descent) parser from scratch. This will allow you to understand more of the code that you write, but will not expose you to the more powerful and standard Bison tool. The rest of this section is written with the assumption that you will choose to use Bison, and the following section will address the from-scratch alternative.

Read the Bison documentation ; take a look at the `src/cool.y`, the skeleton of your parser file, and understand it.

We expect you to use *Bison* to generate a parser for Cool. But if you are really adventurous, you can write a recursive descent parser for Cool in C++. See Section 3.7 if you want to go this route.

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 (we have provided names and type declarations for the terminals); All you need to do there is add type declarations for new nonterminals. The rule section is very much incomplete and you need to finish it.
- `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 `src/`. Your parser needs the output of your completed lexer as input, so first complete the lexer before starting your parser.

- `make` (also `make all`) builds both the lexer and the parser, and `make clean` removes all compiled files.

Use `bison_test_good.cl` in the handout (a valid Cool program) to test your parser 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 the 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.

3.7 Write a Recursive Descent Parser from Scratch

If you choose to write the parser from scratch, your parser should have the same command line interface as the Bison-based parser and report/survive the same errors. Write your recursive descent parser in a `cool_extra_parser.cc` file under `src/`. It should be self-contained and can only include `stdlib` headers and cool support files.

- At submission and grading time, we'll only compile `cool_extra_parser.cc` and only include headers and link against `stdlib` and unmodified cool support files. You can add new `make` rules to `src/Makefile` for local testing purposes, and they will be ignored on submission.

At submission time, notify your TA that you built your own parser instead of using *Bison*. We will test this parser on the exact same test cases as the Bison parser and use the reference lexer for parser input.

4 What to Hand In

You need to submit exactly 2 files that you modify in this lab:
- `cool.flex`
- `cool.y`

If you hand wrote your parser, submit `cool_extra_parser.cc` instead of `cool.y`.

Hand in and grading will be done on gradescope. We will post a more detailed instruction for hand-in when the lab deadline approaches.

Do not modify any part of the support code! Your modifications on these files will be ignored. The provided files are the ones that will be used in the grading process.

We will use the reference lexer to test and grade your parser, so make sure your parser also works with the reference lexer.

If you used LLM-based code-generating tool in your submission:
- If you used a prompted code generator, such as GPT-4, annotate every block of code that was assisted by the code generator (even if you further edited the code generator's output). You should label and number each such code block with comments (e.g., `/* LLM Block 1 */`) and clearly indicate the start/end line of the block. Comment at the end of the file on (1) the tool you used (GPT-3.5, GPT-4, etc.) and (2) the full prompt you gave to the tool, for each code block.

- If you used an unprompted code generator, such as GitHub Copilot, comment at the beginning of each file the name of the tool you used, and indicate any large block of code (≥ 5 lines) you got from the tool.
- We will manually check these comments and may try to reproduce some of the code generation results with your prompts.