CS-GY 6413/CS-UY 3943 — Programming Assignment 2

Jeff Epstein

Introduction

We will write a compiler for the Cool language. Each of several programming assignments will cover one component of the compiler: scanner, parser, semantic analyzer, and code generator. When combined, these components will result in a complete compiler. This assignment covers the scanner (also known as the lexical analyzer, or lexer).

Before starting, please make sure that you're using a suitable programming environment, as discussed in previous assignments. If you haven't yet installed the necessary tools, you can do so with this command from your Debian or Ubuntu terminal:

```
sudo apt-get install build-essential flex bison spim
```

You should complete the assignment in the C++ language, using the `flex` scanner generator. Please refer to the `flex` manual, available on Classes.

You may find online regex analyzers, such as Regexr and iHateRegex, to be useful. However, please be aware that implementations of regexes may differ in the syntax and features offered.

This assignment assumes that you are already familiar with the Cool language. Please refer to the Cool manual, available on Classes, for details on the language.

Collaboration policy Collaboration of any kind is prohibited on this assignment. All submitted work must represent the effort of a single student.

Assignment

Your task is to implement a scanner that can lexically analyze Cool source code, and produce a sequence of tokens. You should use the language description given in the Cool manual, in particular the section "Lexical Structure." You may also refer to the provided example programs.

You will write the scanner using `flex`, a tool for generating scanners. Using `flex`, you can specify patterns that break input into *lexemes* and categorize them as a particular *token*. For example, the lexeme `while` occurring in the input identifies a token `WHILE`. The lexeme `+` identifies a token `'+'`. The lexeme `1234` identifies a token `INT_CONST`. Your scanner description, written in `flex`, will be compiled into C++ code, which will then be compiled into an executable program.

Emitting tokens For each pattern in your `flex` file, you can write an *action* enclosed in curly braces (`{}`). Each action is normal C++ code. A `return` statement occurring in the action should indicate the token that has been matched by the given pattern. Each scanning rule should simply (a) consume the characters from the input stream, (b) if appropriate the type of token, set `yylval`, and (c) `return` the type of token. Do not print anything directly to the output: do not call `printf`, `cout<<`, etc.

Single-character symbols such as `;`, `(`, and `+` are identified by tokens whose name is their own character value. For example, the semicolon token can be matched like this:

```
";"              { return ';'; }
```

Multi-character symbols have specially-defined tokens. The following `flex` code and action will cause the occurrence of the symbol `=>` to be classified as the token `DARROW`:

```
"=>"              { return (DARROW); }
```

All available named token types are given in the `yytokentype` type in `cool-parse.h`.

In the case of `DARROW`, only one symbol matches that token type. Other tokens, however, may match different lexemes: `1234`, `0`, and `52` all match the `INT_CONST` token; `""`, `"\n"`, and `"hello world"` all match `STR_CONST`; and `foo`, `bar_baz`, and `x1234` all match `OBJECTID`. In these and other cases, the scanner should emit not only the token type, but also the lexeme (i.e. the corresponding text from the source code) that matched. To do so, your action must assign an appropriate value in the `yylval` union. In particular, `yylval` provides the following members that you should use. You do not have to use any other members of `yylval` for this assignment.

- `yylval.boolean` — Store an integer representing a boolean value (either 0 or 1) here when indicating a boolean literal by returning the token type `BOOL_CONST`.
- `yylval.symbol` — Store a *symbol* (of type `Entry*`) representing the matched lexeme here when returning the token types `TYPEID`, `OBJECTID`, `STR_CONST`, or `INT_CONST`.

  To get an appropriate symbol value, we use *tables*, which provide a common repository of lexemes. Use of a table potentially helps save memory, by allowing us to refer to its position in the table, rather than repeating its value. An implementation of the appropriate tables has been provided for you (see `stringtab.h`). Three tables are to be used in your scanner: `idtable`, for storing identifiers; `inttable` ↪ , for storing integer literals; and `stringtable`, for storing string literals. In each case, use the `add_string` method to insert a value into the table. The value returned by `add_string` is a reference to the entry in the table, and it is this value that should be stored in `yylval.symbol`.

  For example, you could use the following pattern and action to match integer literals:

  ```
  [0-9][0-9]*      { yylval.symbol = inttable.add_string(yytext,yyleng);
                     return (INT_CONST); }
  ```

  Here, the `yytext` variable provided by `flex` is the matching lexeme in the source code, and `yyleng` is the number of character in that lexeme.

- `yylval.error_msg` — Store a normal C string (of type `char*`) here when indicating a lexical error by returning the token type `ERROR`.

Strings Strings in the Cool language occur between double quotes ("). However, the value that you enter into `stringtable` should not include the double-quotes. That is, the double quotes are not part of the value of the string. Your scanner is responsible for removing the double quotes.

Your scanner is also responsible for other transformations on the input text. For example, if the programmer types these eight characters in their Cool source code:

$$\boxed{\texttt{"}}\,\boxed{\texttt{a}}\,\boxed{\texttt{b}}\,\boxed{\backslash}\,\boxed{\texttt{n}}\,\boxed{\texttt{c}}\,\boxed{\texttt{d}}\,\boxed{\texttt{"}}$$

your scanner would return the token `STR_CONST` whose semantic value is these 5 characters:

$$\boxed{\texttt{a}}\,\boxed{\texttt{b}}\,\boxed{\backslash\texttt{n}}\,\boxed{\texttt{c}}\,\boxed{\texttt{d}}$$

where $\boxed{\backslash\text{n}}$ represents the literal ASCII character for newline (ASCII value 10).

Error handling Your scanner should be robust: it should work for any conceivable input, even for inputs that are not valid Cool programs. 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 (i.e. segfaults) or uncaught exceptions are unacceptable.

Note that your scanner is responsible only for *lexical* analysis of the input. Other errors, such as syntax errors (for example, an `if` without an associated `then`) or semantic errors (for example, a reference to an undefined identifier) will be handled by future phases of the compiler, and your scanner should not produce an error.

If your scanner encounters a lexical error in its input, it should pass this error on to the parser by emitting a special error token. Specifically, your scanner should assign an appropriate string to `yylval.error_msg` and then return `ERROR` from a pattern's action. Do not print anything directly to the output. When possible, your scanner should continue scanning after the error.

Here are requirements for error handling:

- When an invalid character (one that can't begin any token) is encountered, a string containing just that character should be returned as the error string. Resume scanning at the following character.
- If a string contains an unescaped newline, report that error as "Unterminated string constant" and resume scanning at the beginning of the next line. We assume the programmer simply forgot the close-quote.
- When a string is too long, report the error as "String constant too long" in the error string of the `ERROR` token. If the string contains invalid characters (i.e., the null character), report this as "String contains null character". In either case, scanning should resume after the end of the string. The end of the string is defined as either:
  - the beginning of the next line if an unescaped newline occurs after these errors are encountered; or
  - after the closing quotation mark otherwise.
- If a comment remains open when end-of-file is encountered, report this error with the message "EOF in comment". Do not tokenize the comment's contents simply because the terminator is missing. Similarly for strings, if end-of-file is encountered before the close-quote, report this error as "EOF in string constant".
- If you see `*)` outside of a comment, report this error as "Unmatched `*)`", rather than tokenzing it as * and).

Other notes 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. To do this, match newlines in the source code and update the variable accordingly.

Ignore the token `LET_STMT`: it is used only by the parser phase.

Your lexical specification should be *complete*, i.e. every input should match some regular expression. If `flex` encounters input that is not matched by any pattern, it may not produce desired results. The following pattern and action, placed at the end of the pattern section of your file, will match any character that isn't otherwise matched (possibly resulting in bad tokens):

```
.  { return yytext[0]; }
```

Setup

Download two files: `pa2.zip`, which contains starter code specifically for this assignment; and `refimpl.zip`, which contains a reference implementation of the Cool compiler. Extract the content of both zip files into separate directories.

There are a few files that are of interest of us:

- `cool.flex` — This file contains a skeleton for a lexical description for Cool. There are comments indicating where you need to fill in code, but this is not necessarily a complete guide. Part of the assignment is for you to make sure that you have a correct and working lexer. Except for the sections indicated, you are welcome to make modifications to our skeleton. You can actually build a scanner with the skeleton description, but it does not do much. Any auxiliary routines that you wish to write should be added directly to this file in the appropriate section (see comments in the file).
- `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 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.
- `README` — This file contains detailed instructions for the assignment as well as a number of useful tips. You should also edit this file to include the write-up for your project. You should explain design decisions, why your code is correct, and why your test cases are adequate. It is part of the assignment to clearly and concisely explain things in text as well as to comment your code.

You don't need to modify any other file.

Running the `make` command from that directory will compile your code. If compilation is successful, it will produce an executable file `lexer`.

Testing

There are several ways you can test your scanner. In all cases, it's up to you to design inputs that adequately test all features of your scanner: don't rely only on the provided test file.

Direct testing You can call your scanner directly. The following command will call your compiled `lexer` executable, giving it the file `test.cl` as input:

```
./lexer test.cl
```

Or, equivalently:

```
make dotest
```

The output of your scanner should be a sequence of tokens, one per line, marked with the line number where the token occurred and augmented by any additional data relevant to that token, such as an identifier's name. The provided code takes care of formatting the output: all you have to do is give the scanning rules.

You can compare the output of your scanner with that of the scanner provided as part of the reference implementation. To call the provided scanner, assuming you've unzipped the reference implementation into a folder named `refimpl`, use this command:

```
../refimpl/lexer test.cl
```

Testing as part of compiler The provided reference implementation of Cool contains complete versions of all four phases of the compiler. You can replace the reference version of the scanner with your version, and then verify that the compiler still works correctly.

To replace the scanner, just copy your compiled `lexer` into the directory where you've extracted the reference implementation:

```
cp lexer ../refimpl/lexer
```

Now run the compiler `coolc` on your test program. If your scanner is correct, the subsequent phases of compilation should run normally, and you can run the compiled test program.

Rules

Your code should compile cleanly with a 5.x or newer version of `gcc` and the provided `Makefile`.

You may use any standard C++ header files, but you may not use any third-party libraries.

Do not add any additional files to the program. Modify only those files indicated in `README`.

Use good coding style, including informative variable names, consistent indentation, and clear structure. Your code should be robust and generally free of bugs. As a special case, please note that heap leaks are acceptable, as long as they don't impact the functioning of the program.

You are not obligated to use comments. However, confusing, unclear, and uncommented code may result in a reduced grade. The grader must be able to understand your code.

Submission

Test your code thoroughly before you submit.

In addition to your code, you must answer the questions at the end of the provided `README` file. Write your answers directly in the `README` file.

Run the command `make zip` and check the content of the resulting zip file to make sure that it contains all the code you wrote. Submit only the zip file on Gradescope. Do not add additional files to the zip file.

Acknowledgments

This sequence of assignments is based on work by Alex Aiken and Westley Weimer.