Programming Assignment II

Due Friday, October 10, 2014

1. Overview

Programming assignments II-V will direct you to design and build a compiler for Cool. Each assignment will cover one component of the compiler: lexical analysis, parsing, semantic analysis, and code generation. Each assignment will ultimately result in a working compiler phase which can interface with other phases.

For this assignment you are to write a lexical analyzer, also called a scanner, using a lexical analyzer generator (the Java tool is called `jflex`). You will describe the set of tokens for Cool in an appropriate input format and the analyzer generator will generate the actual Java code for recognizing tokens in Cool programs. You must work in a group for this assignment (where a group consists of one or two people).

2. Files and Directories

To get started, create a directory where you want to do the assignment and execute one of the following commands in that directory. You should type:

```
make -f ~/cool/assignments/PA2J/Makefile source
```

(notice the “J" in the path name). This command will copy a number of files to your directory. Some of the files will be copied read-only (using symbolic links). You should not edit these files. 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.lex`

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 jlex 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).

- `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. 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.

Although these files as given are incomplete, the lexer does compile and run (make lexer). There are a number of useful tips in the README file.

3. Setting Up the Development Environment

To setup your development environment you should accomplish two main tasks: creating and populating your IDE project, and setting up a repository to facilitate collaboration with your partner. This instructions will use Eclipse Luna as the IDE and GitHub as the source code repository, but you are welcome to user a different setup. Is will not be acceptable not to use a source code repository such as CVS, SVS, or Git. However, we strongly recommend you to use GitHub as it is becoming an industry standard that is often used to track a computer scientists software development contributions and experience.

After completing Lab 2 you should be familiar with the Eclipse IDE and the JFlex/CUP plugin. Create a new project named `CoolCompiler` to complete all you work pertaining to the lexical analysis phase and other phases of the compiler. You should copy all the java source files from the directory created above for your PA2 work. You can simply drag and drop the files from the Ubuntu file manager into your Eclipse `src` directory within the `CoolCompiler` project.

The next step consists of setting up a GitHub repository and connecting your Eclipse project to this repository. If you haven't already (It's about time!) you should register for a free GitHub account and create a new public repository for your compiler project. If you decide to name your repository "AceCompiler" GitHub will assign it a URL similar to the following. Copy the URL to your clipboard to use it in the next step.

https://github.com/<YourGitUsername>/AceCompiler.git

At this point you should add your parner's GitHub username to the list of collaborators on the GitHub Settings for your repository. Give appropriate permissions to allow your partner to pull/push code from/to GitHub as he contributes to the development process.

To connect an Eclipse project with your repository you should install the EGit Eclipse plugin. The latest version (Luna) of Eclipse comes with this plugin already installed. In Eclipse change to the Git perspective:

Window -> Open Perspecive -> Other … -> Git

Now you must clone the GitHub repository. Do not worry that you do not have anything on the repository yet. Select the "Clone Repository" icon of menu item on the "Git Repositories" panel on the left of the Git perspective. You can then paste the repository URL from your clipboard. Once you paste the URL all the rest of the parameters will be filled automatically. We do not recommend that you modify these parameters unless you are absolutely sure of what you are doing. Click Next repeatedly until the repository is finally cloned. You will see a new repository in your "Git Repositories" panel.

Now you should associate or "Share" your Compiler project with your Git repository. Open the Java perspective, right-click on your project and select Team -> Share Project. Select Git and click **Next**. Select your cloned Git repository from the drop-down menu and click Finish. Back on the Java perspective your project should now show the name of the repository next to it.

The next step consist of uploading your Java source files to the GitHub repository. This is a two step process that first commits your files to the local Git repository and then pushes your local repository to GitHub. Remember that Git is a distributed source control system, so you will be able to commit changes to your repository even when your laptop has no Internet connection. Propagating your local repository to GitHub, or any remote Git repository for that matter, will require a push and thus such as connection.

To commit your files to the local repository, right click on the project and select Team -> Commit. On the upper text box of the dialog box you should type a message describing the changes that you are committing. On the lower part of the dialog box you should see a list of all the files that have suffer modifications. You should choose among modified files, those that you wish to commit based on your knowledge about the state of these files. You should normally only commit files that are ready to be shared with others and/or put into a software release. You should never commit evidently broken code. This first time that you commit the PA2 files, all the files will appear on the commit dialog since all of them are new. After you finish selecting your files hit the Commit button. Your local Git repository should now have updated copies of the committed files.

To update the GitHub repository you should right-click on the project and select Team -> Push to Upstream. A dialog box will pop-up asking for your GitHub credentials. Provide the credentials and hit OK. Once the push is completed you can go to the GitHub website and verify that your project now appears in the repository.

You can know tell your partner to clone the repository on his/her Eclipse environment. He/She will no have to push, but rather pull the repository from GitHub into his/her local repository.

Remember that you can commit working code first to you local repository even when disconnected in order to make sure that you keep your latest working copy separate from your working files, which are typically broken with ongoing changes as you work on your project. To share your changes with your partner you must push and ask him/her to pull from GitHub.

4. Scanner Results

You should follow the specification of the lexical structure of Cool given in Section 10 and Figure 1 of the CoolAid manual. 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 or uncaught exceptions 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. See the following sections for the details.

All errors will be passed along to the parser. The Cool parser knows about a special error token called ERROR, which is used to communicate errors from the lexer to the parser. There are several requirements for reporting and recovering from lexical errors:

- 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, that should be reported. Lexing should resume after the end of the string.
- 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.
- 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 create a Symbol with the entire literal's text as its contents, regardless of its length.

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.

Your scanner should convert escape characters in string constants to their correct values. For example, if the programmer types these eight characters:

"ab\ncd"

your scanner would return a token whose semantic value is these 5 characters:

ab¶cd

In this example, we use ¶ to represent the ascii code for newline. In JLex, you can produce this code by typing \n.

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

5. Notes for the assignment

- The codes for all tokens are defined in the file `TokenConstants.java`. Each call on the scanner returns the next token and lexeme from the input. The semantic value or lexeme, can be obtained by calling the function `yytext()`, which returns a Java String object. All of the single character tokens are listed in the grammar for Cool in the CoolAid.
- Each action for the lexemes in the last section of `cool.lex` should return an object of type `Symbol` with its `value` field containing the semantic value or lexeme unless it is not necessary (for example, a state-related action). The only parameter of the constructor for `Symbol` specifies the token type (integer constant, object id, etc) as specified in `TokenConstants.java` and in the API.

Example:

```
Symbol ret = new Symbol(TokenConstants.BOOL_CONST);
ret.value = boolvalue;
return ret;
```

- 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. The type of string table entries is `Symbol`. Before returning a string constant, integer constant or an identifier, you must add it to the provided string tables in `AbstractTable.stringtable`, `AbstractTable.inttable` and `AbstractTable.idtable`, respectively.

  Example for an identifier:

```
AbstractSymbol lex_val = AbstractTable.idtable.addString(yytext());
Symbol ret = new Symbol(TokenConstants.OBJECTID);
ret.value = lex_val;
return ret;
```

- When a lexical error is encountered, the action should return a `Symbol` of type `TokenConstants.ERROR`. Its `value` field should contain a `StringSymbol` with the error message. See previous section for information on what to put in error messages.

  Example:

```
String err_msg = new String("Unterminated string");
StringSymbol error = new StringSymbol(err_msg, err_msg.length(), 0);
Symbol ret = new Symbol(TokenConstants.ERROR);
ret.value = error;
return ret;
```

6. Testing the Scanner

There are at least two ways that you can test your scanner. The first way is to generate sample input files and scan them running your lexer from Eclipse as a Java Application. This prints out the line number and the lexeme of every token recognized by your scanner.

The other way, when you think your scanner is working, is to try running `./mycoolc` to invoke your `lexer` together with all other compiler phases (which we provide). This will be a complete Cool compiler that you can try on the sample programs and your program from Assignment I.

You can also look at the reference lexer's output and try to match yours to that:

```
~/cool/bin/lexer test.cl
```

Your output should be the same as the reference lexer. Do not output any unnecessary code or debug strings or your lexer will not work with the other phases of the compiler.

7. Turning In the Assignment

1. Make sure your code is in `cool.lex` and that it compiles and works.
2. Your test cases should be in `test.cl`.
3. Include any other relevant comments in the `README` file and answer any questions that appear in it.
4. Make sure you push all the Java, JFlex and README file to your GitHub repository.
5. When you are ready to submit your PA2 for grading tag your project with the label "v1.0" and add the professor's GitHub account (bvelez) as a collaborator with just pull access. You can learn more about Git taging at http://git-scm.com/book/en/Git-Basics-Tagging.
6. IMPORTANT: Test that your tag retrieves the correct project files by pulling the code into a new dummy project and making sure that it runs as expected.
7. After this test, you may proceed to work on other phases of the project and even push new code to your repository without affecting the version that you submitted for grading PA2. That's the beauty of source code version control, Git and GitHub.