CS 331 Compilers **Fall 2017**

Programming Project 1: Lexical Analyzer (Scanner)

Prof. Szajda *Due Thursday, September 21, 11:59:59 pm*

1 Overview of the Programming Project

Programming projects I – IV will direct you to design and build a compiler for Cool. Each project will cover one component of the compiler: lexical analysis, parsing, semantic analysis, and code generation. Each project will ultimately result in a working compiler phase which can interface with other phases. You must implement the project in Java.

For this project, you are to write a lexical analyzer, also called a *scanner*, using a *lexical analyzer generator*. (The Java tool is called `JLex`.) 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.

On-line documentation for all the tools needed for the project will be made available on the course web site. This includes manuals for `JLex` (used in this project), the documentation for `java_cup` (used in the next project), as well as the manual for the spim simulator. You must work in pairs for this project – your combined project must list both participants.

2 Using `JLex`

I could provide an introduction to `JLex`, but I won't, because all I would be doing is paraphrasing the `JLex` manual, which is both very readable and relatively short (about 20 pages of not-very-dense text, many sections of which can be skipped). At this point in your CS major, each of you has the skills to read a tech manual.

The most important part of your lexical analyzer will be the rules section. A rule in `JLex` (and every other scanner generator I've run across) specifies an action to perform if the input matches the regular expression or definition at the beginning of the rule. The action to perform is specified by writing regular Java source code. For example, assuming that `OBJECTSYM` is a regular expression describing the object identifier syntactic category in Cool (corresponding to `TokenConstants.OBJECTID`), the rule:

```
{OBJECTSYM} {
    return new Symbol{TokenConstants.OBJECTID,
                      AbstractTable.idtable.addString(yytext()));
}
```

records the value (lexeme) of the identifier in the `idtable` and returns the appropriate token code. Understanding this line of code takes a bit of digging. First, you should take a look at the documentation in the `AbstractTable.java` source file to learn what the `idtable` is, and what the `addString` method does. Also, this code is returning an object of type `Symbol` (as should pretty much all of your Java code in this assignment). You won't, however, find the source for the `Symbol` class in your cool project files. Instead, this is a class imported from the `java_cup.runtime` package. Java CUP (Constructor of Useful Parsers) is a Java based program that, as the name implies, constructs useful parsers from relatively simple specifications. While you don't have to know much about CUP for this project (though will for the next), you do have to know this: the CUP generated parser expects to receive objects of class `Symbol` from the lexer, and CUP generally returns a `Symbol`. The API for Java CUP is linked off the course web page – you should check out the `Symbol` API to learn about the constructors for `Symbol`. Finally, `yytext()` is a method that returns a `String` representing the matched portion of the input character stream. This is a method that is created in the generated Java lexer code (so you can use it in your specifications, because it will be available in the generated Java code).

An important point to remember is that if the current input (i.e., the result of the function call to `yylex()`) matches multiple rules, JLex picks the rule that matches the largest number of characters. For instance, if you define the following two rules

```
[0-9]+ { // action 1}
[0-9a-z]+ {// action 2}
```

and if the character sequence 2a appears next in the file being scanned, then action 2 will be performed since the second rule matches more characters than the first rule. If multiple rules match the same number of characters, then the rule appearing first in the file is chosen.

When writing rules in `JLex`, it may be necessary to perform different actions depending on previously encountered tokens. For example, when processing a closing comment token, you might be interested in knowing whether an opening comment was previously encountered. One obvious way to track state is to declare global variables in your declaration section, which are set to true when certain tokens of interest are encountered. `JLex` also provides syntactic sugar for achieving similar functionality by using state declarations such as:

```
%Start COMMENT
```

which can be set to true by writing `BEGIN(COMMENT)`. To perform an action only if an opening comment was previously encountered, you can predicate your rule on `COMMENT` using the syntax:

```
<COMMENT> {
   // the rest of your rule ...
}
```

There is also a special default state called `YYINITIAL` which is active unless you explicitly indicate the beginning of a new state. You might find this syntax useful for various aspects of this project, such as error reporting. I strongly encourage you to read the full documentation on `JLex` written by Elliot Berk linked to on the course web page before writing your own lexical analyzer.

3 Files and Directories

You are assumed to be working within the virtual OS provided, and thus all of the files and directories discussed here are contained in that virtual OS.

To get started, create a directory where you want to do the project and execute the command

```
make -f /usr/class/cs143/assignments/PA2J/Makefile
```

Note that even though this is the first programming assignment, the directory name is PA2. Future assignments will also have directories that are one more than the assignment number, so please remember this! This situation arises because some versions of the compilers course that Professor Aiken taught had a small initial assignment, which he called assignment 1. Since we are skipping that first assignment (Professor Aiken typically skips it now as well), the assignment numbers as they appear in the directory structure are all one ahead of the actual assignment number. Since I didn't want to fiddle too much with the VM image, I've left these as is.

Notice also the "J" in the path name above. Professor Aiken allows his students to do projects in either C++ or Java. Because of this, the directory structure in the VM contains provisions both for C++ versions of the project (which uses `Flex`, not `JLex`). If you forget the "J" in the path, your command will work, but you will be running a command that puts the C++ versions of files into your directory! Not what you want.

The correct 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. 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 the skeleton. You can actually build a scanner with the skeleton 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. (Dont 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. The program `test.cl` is similar to a real Cool program, but your tests need not be. You may keep as much or as little of the 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.

Although these files are incomplete as given, the lexer does compile and run. **To build the lexer, you must type `make lexer`!**

4 Scanner Results

In this assignment, you are expected to write `JLex` rules that match on the appropriate regular expressions defining valid tokens in Cool (as described in Section 10 and Figure 1 of the Cool manual) and perform the appropriate actions, such as returning a token of the correct type, recording the value of a lexeme where appropriate, or reporting an error when an error is encountered. Before you start on this assignment, make sure to read Section 10 and Figure 1 of the Cool manual, then study the different tokens defined in `TokenConstants.java`. Your implementation needs to define `JLex` rules that match the regular expressions defining each token defined in `TokenConstants.java` and perform the appropriate action for each matched token. For example, if you match on a token `BOOL_CONST`, your lexer has to record whether its value is true or false. Similarly if you match on a `TYPEID` token, you need to record the name of the type. Note that not every token requires storing additional information. For example, only returning the token type is sufficient for some tokens (e.g., keywords).

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 – uncaught exceptions are unacceptable.

4.1 Error Handling

All errors should be passed along to the parser. You lexer should not print anything. Errors are communicated to the parser by returning a special error token called `ERROR`. (Note, you should ignore the token called `error` [in lowercase] for this assignment; it is used by the parser in the next assignment.) There are several requirements for reporting and recovering from lexical errors:

- 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 lexing at the following character.
- If a string contains an unescaped newline, report that error as `''Unterminated string constant''` and resume lexing 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 in the `ERROR` token. If the string contains invalid characters (i.e., the null character), report this as `''String contains null character''`. In either case, lexing should resume after the end of the string. The end of the string is defined as either
  1. the beginning of the next line if an unescaped newline occurs after these errors are encountered; or
  2. after the closing `''` otherwise.
- If a comment remains open when EOF 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 an EOF is encountered before the close-quote, report this error as `‘‘EOF in string constant’’`.

- If you see `‘‘*)’’` outside a comment, report this error as `‘‘Unmatched *)’’`, rather than tokenzing it as `*` and `)`.
- Recall from lecture that this phase of the compiler only catches a very limited class of errors. Do not check for errors that are not lexing errors in this assignment. For example, you should not check if variables are declared before use. Be sure you understand fully for which errors the lexing phase of a compiler does and does not check for before you start.

4.2 String Table

Programs tend to have many occurrences of the same lexeme. For example, an identifier is generally 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. A string table implementation is provided here. See the following sections for the details.

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.

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.

4.3 Strings

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

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

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 $\boxed{\backslash\texttt{n}}$ represents the literal ASCII character for newline.

Following specification on page 15 of the Cool manual, you must return an error for a string containing the literal null character. However, the sequence of two characters

$$\boxed{\backslash\texttt{n}}\ \boxed{\texttt{0}}$$

is allowed but should be converted to the one character

$$\boxed{\texttt{0}}.$$

4.4 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.

You should ignore the token `LET_STMT`. It is used only by the parser (PA3). Finally, note that if the lexical specification is incomplete (some input has no regular expression that matches), then the scanner generated by `JLex` do undesirable things. *Make sure your specification is complete!*

5 Auxiliary States

If you have not already inferred this, let me make it clear at this point that you are going to have to create auxiliary states to help determine which actions should take place under certain circumstances. For example, you will likely need a `COMMENT` state. Why? Well, recall how the lexer works in general. It begins in a state called `YYINITIAL`. It then begins checking the input string one character at a time. When the longest match (as defined by the set of regular expressions) is found, this token is generated (and sent to the parser), and the matching lexeme is removed from the input string. At this point, whatever state the lexer was in, it is returned to state `YYINITIAL`, and processing begins with the remainder of the input string. So, suppose we are in state `YYINITIAL` and the beginning of the input stream is `(*`. This signals the beginning of a comment. If you do not transition to a special comment state at this point, and the following characters are `if` your lexer might think it has found a keywork (and return the associated token), when it has not. Because the `if` is inside a comment, it should be ignored. This implies that your lexer, on encountering `if`, has to take different actions, depending on what state it is in. This is only one example – there are many more. Bottom line: you need to think through what states you will need, then define them, and the behaviors associated with them, in your `JLex` file.

6 Misc. Notes

- Each call on the scanner returns the next token and lexeme from the input. The value returned by the method `CoolLexer.next_token` is an object of class `java_cup.runtime.Symbol`. This object has a field representing the syntactic category of a token (e.g., integer literal, semicolon, the `if` keyword, etc.). The syntactic codes for all tokens are defined in the file `TokenConstants.java`. The component, the semantic value or lexeme (if any), is also placed in a `java_cup.runtime.Symbol` object. The documentation for the class `java_cup.runtime.Symbol` as well as other supporting code is available on the course web page. Examples of its use are also given in the skeleton.
- For class identifiers, object identifiers, integers, and strings, the semantic value should be of type `AbstractSymbol`. For boolean constants, the semantic value is of type `java.lang.Boolean`. Except for errors (see below), the lexemes for the other tokens do not carry any interesting information. Since the `value` field of class `java_cup.runtime.Symbol` has generic type `java.lang.Object`, you will need to cast it to a proper type before calling any methods on it.
- The project provides you with a string table implementation, which is defined in `AbstractTable.java`. For the moment, you only need to know that the type of string table entries is `AbstractSymbol`.

- When a lexical error is encountered, the routine `CoolLexer.next_token` should return a `java_cup.runtime.Symbol` object whose syntactic category is `TokenConstants.ERROR` and whose semantic value is the error message string. See Section 4 for information on how to construct error messages.

7 Testing the Scanner

There are at least three ways that you can test your scanner. The first way is to generate sample inputs and run them using `lexer`, which prints out the line number and the lexeme of every token recognized by your scanner. The second way, when you think your scanner is working, is to try running `mycoolc` to invoke your lexer together with all other compiler phases (which are provided). This will be a complete Cool compiler that you can try on any test programs. The third way, and the one that I use, is to use the code insertion facilities that `JLex` provides. In particular, `JLex` allows one to insert Java code at various points in the generated Java lexer code. Using this, you can insert customized error or behavior messages, so, for example, you can have your lexer print out every token it generates and the lexeme associated with that token. You can also (as I have done) allow your code to respond to a debug flag that is hard coded in the lexer. If the debug flag is set, then my lexer prints out every action in great detail (augmenting the information provided by method 1). Turning off the debug flag effectively causes my debug code to be ignored.

8 Transfering Work Between the VM and Your Machine

The instructions I'm giving here are for a Mac. There are similar steps you need to take if you're running VirtualBox on a PC. The detailed instructions are listed under `Sharing` in the VirtualBox reference manual. (Note that the manual speaks of having to install/enable `Guest Additions`. These are already installed and enabled in the VM to the degree required.)

So, the steps:

1. Create a shared folder on your computer. (In Mac, this is done via the sharing preferences in the System Preferences app.)
2. In the VirtualBox menu, go to `Machine->Settings` and then look for the `Sharing` tab. Once there, the dialog box leads you through choosing the path (on the host machine, not the VM) to the shared folder. When setting this up, be sure to choose the `Automount` option.
3. Reboot the Bodhi OS. This is done by pressing the `Bodhi` button on the lower left of the OS screen, choosing `System`, then `Reboot`.
4. After the reboot, the shared folder will be mounted at the location `/media/sf_XXX`, where `XXX` stands for the name of the shared folder on the host. So, e.g., if you named the file `transfer_folder`, then the shared file will be located at `/media/sf_transfer_folder`.
5. The last little bit of nastiness: the shared folder is owned by root. I don't know the root password for the Bodhi VM, so you can't change the owner of the file, etc. But you can do anything you want (including moving files into and out of the directly, listing directory contents, etc.) by putting `sudo` before every command. So, for example, to copy the file foo.tar from the directory `/home/compilers/my_work` to the shared folder `transfer_folder`, and assuming you are currently in `/home/compilers/my_work`, you run the command

```
sudo cp foo.tar /media/sf_transfer_folder
```

Obviously, running the transfer from within your Mac or PC should not require the sudo stuff.

9 What to Turn In

I will give instructions on exactly what to turn in and exactly how to turn it in in a revision to this document in a couple of days. You will be turning in via Box, but the exact folder to which you should submit has not yet been determined.