

Programming Assignment I

Due Friday, April 10, 2009 at 11:55pm

1 Overview

Programming assignments I–IV (or 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. You will have an option of doing your projects in C++ or Java.

For this assignment, you are to write a lexical analyzer, also called a *scanner* or *lexer*. You will be writing the lexer “by hand” rather than using a *lexical analyzer generator* such as *lex*, *flex*, or *jlex*. You will write code to approximate the function of a DFA by scanning for Cool tokens and then output the results in the appropriate format. The description of the output and the list of tokens will be provided for you.

On-line documentation for all the tools needed for the project will be made available on the ECS142 web site. You may work either individually or in pairs for this assignment. Pairs are encouraged.

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*. For the C++ version of the assignment, you should type

```
gmake -f /home/cs142/s09/cool/assignments/PA2/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—please don’t get confused! This situation arises because we are skipping the usual first assignment in this offering of the course. For Java, type:

```
gmake -f /home/cs142/s09/cool/assignments/PA2J/Makefile
```

(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:

- *lertest.cc* / *Lexer.java*

This file contains starter code to read text file input and call the handwritten lexer below. This file probably will not need to be modified.

- *cool-hand-lex.cc* (in the C++ version) / *CoolHandLex.java* (in the Java version)

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. Although you will be writing a hand lexer you may find the the flex/jlex manual helpful to understand some of the defined datastructures and other assorted definitions.

- `cool-parse.h` / `TokenConstants.java`
This file contains the complete list of token types that your lexer will need to recognize. See the README file in the project directory for more details.
- `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.
- `life.cl`
This file contains another not so simple cool program.
- `life.l`
This file contains sample output from running `life.cl` through the reference lexer.
- `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 (`gmake lexer`).

3 Scanner Results

You should follow the specification of the lexical structure of Cool given in Section 10 and Figure 1 of the Cool 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.

3.1 Error Handling

All errors should be passed along to the parser. Your 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 PA3.) 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 tokenizing it as * and).

3.2 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*. We provide a string table implementation for both C++ and Java. 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.

3.3 Strings

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

"	a	b	\n	c	d	"
---	---	---	----	---	---	---

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

a	b	\n	c	d
---	---	----	---	---

where

\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

$\boxed{\backslash} \boxed{0}$

is allowed but should be converted to the one character

 $\boxed{0}$.

3.4 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).

4 Testing the Scanner

There are at least two 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 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.

5 What to Turn In

When you are ready to turn in the assignment, type `gmake submit-clean` in the directory where you have prepared your assignment. This action will remove all the unnecessary files, such as object files, class files, core dumps, Emacs autosave files, etc. Following `gmake submit-clean`, use the `handin` utility to submit each modified file. e.g. `handin cs142 PA2 cool-lex-hand.cc` or `handin cs142 PA2J CoolLexHand.java`

The last submission you do will be the one graded. Each submission overwrites the previous one. Remember that late assignments are not accepted. If in doubt, submit early and often. The burden of convincing us that you understand the material is on you. Obtuse code, output, and write-ups will have a negative effect on your grade. Write safe code (e.g., don't use `strcpy`), comment appropriately, and modularize well.