Programming Assignment 1
Due Mar 10, 2023 Midnight

Programming Assignments for this semester will guide you to design and implement a working compiler for the Classroom Object-Oriented Language (*Cool*). One or two assignments cover each component of the compiler: lexical analyzer, parser, and semantic analyzer, IR generator, and IR optimization pass. Combined together, you will have a working compiler that translates the complete set of *Cool* features into *llvm* IR, detects and reports errors with debugging information, and performs IR-level optimizations. You can verify your *Cool* compiler by compiling the resulting *llvm* IR with the *llvm* back-end and testing the binary.

For this assignment, you are going to **implement a lexical analyzer or scanner** using a scanner generator tool called `flex`. You will have to define the set of tokens for *Cool* in a correct format, and `flex` will generate the actual C++ code for identifying tokens in *Cool* programs. Your lexer should output a sequence of tokens that correctly represents the source program, and gracefully handle lexical errors.

Documentation for all the tools needed for the project is or will be made available on the course website, including the *Cool* reference manual, links to online manuals for *flex* and *bison* (for next PA), as well as online documentation for *llvm*. It is highly recommended that you read the *Cool* reference manual, get familiar with the language, and write simple *Cool* programs too which you can also use for testing your compiler.

1 Files and Directories

To get started, download and unpack the *pa1* directory under Resources from the course website. This directory contains all the files that you will need for this PA. The files you will need to modify are:

- `cool.flex`
  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 flex 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.

Although these files are incomplete as given, the lexer does compile and run. To build the lexer, type

make or *make lexer*

in the directory *pa1/src*. This will start the compilation process and link the support code needed for this phase into your working directory. Execute the lexical analyser by typing

lexer input_file

If you need further examples, there is a link to a set of example *Cool* programs under Resources at the course website, together with a README file and expected output files. Read the README file to learn what the example programs are about, and read the comments in the example files for further help.

2 Scanner Result

In this assignment, you are expected to write *flex* 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 if needed, or reporting an error when an error is encountered.

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.

2.1 Error Handling

All errors should be passed along to the parser. Errors are communicated to the parser by returning a special error token called ERROR which carries a detailed error message. 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.
- When a string is too long, report the error as `"String constant too long"` in the error string in the ERROR token. When the string contains the null character, report this as `"String contains invalid character"`. Do not produce a string token before the error token.
- When a string contains an unescaped newline, report this as `"Unterminated string constant"`. 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 this error with the message `"EOF in comment"`. Do *not* tokenize the comment's contents simply because the terminator is missing. Similarly, for string, 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 `)`.

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 does not 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.

Recall from lectures that this phase of the compiler only catches a very limited class of errors. **Do not try to check for errors that that are not lexing errors in this assignment.**

Finally, note that the lexical specification is incomplete (some input has no regular expression that matches) then the scanner generated produces undesirable result. Make sure your specification is complete.

3 Notes

- Each call on the scanner 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 of 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 itslef. 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 print in error messages.

4 Testing the Scanner

You can write your own sample Cool programs or use the Cool example programs provided to run them using lexer, which will print out the line number and the lexeme of every token recognized by your lexer. From the output, you can check if all the lexemes are correctly recognized. You can also try running the reference parser included in the assignment to invoke your lexer together with it.

lexer input_file | parser

If the parser fails to interact with lexer and emits error messages, it is very likely that your lexer is incomplete or has a bug unless you were testing a program with intentional lexical errors. For later assignments, we will provide a full reference compiler so that you can test your implementation together with them by running a compiled binary and verifying its execution result.

5 What and How to Turn In

It is your responsibility to ensure that the final version you submit does not have any debug print statements and that your lexical specification is complete (every possible input has some regular expression that matches). You have to turn in the *pa1* directory with your modified version of `cool.flex` after compressing it again with

tar -cvf pa1_[your student id].tar pa1

You can upload the compressed file to the board for assignment submission at the course website. Please do not copy or modify any part of the support code. The provided files are the ones that will be used in the grading process.