1 Scanner Results

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 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 cool-parse.h. Your implementation needs to define Flex/Jlex rules that match the regular expressions defining each token defined in cool-parse.h 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 like 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. Core dumps or uncaught exceptions are unacceptable.

1.1 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 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 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 what errors the lexing phase of a compiler does and does not check for before you start.

1.2 String Table

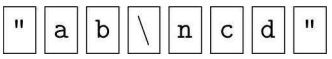
Programs tend to have many occurrences of the same lexeme. For example, an identifier is generally referred to more than once in a program (or else it isn't very useful!). To save space and time, a common compiler practice is to store lexemes in a *string table.* We provide a string table implementation 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.

1.3 Strings

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



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

00

is allowed but should be converted to the one character

.

1.4 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 scanners generated by both flex and jlex do undesirable things. *Make sure your specification is complete.*

2 Notes for the flex

- 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 (e.g., integer literal, semicolon, if keyword, etc.). The codes for all tokens are defined in the file cool-parse.h. The second component, the semantic value or 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 ",") are represented just by the integer (ASCII) value of the character itself. All of the single character tokens are listed in the grammar for Cool in the Cool manual.

- 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 in 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 the previous section for information on what to put in error messages.