We want to build a lexical analyzer for cool. In order to do that we have to have a DFA that helps us specify which class to assign to every token. What flex does is to convert RE to NFA and NFA to DFA so we will make an RE and give it to flex and it will make the DFA for us.

I will explain types of classes and their implementation:

Key words:

As said in cool-manual, keywords of cool are:

class, else, false, fi, if, in, inherits, isvoid, let, loop, pool, then, while, case, esac, new, of, not, true

In cool some types of keywords are case insensitive which means it doesn't matter whether they are lower case or upper case.

For example: ELSE [Ee][lL][sS][eE]

This means "Else" and "elSE" are the same things for lexer.

And after reading it lexer should return ELSE. The code is like this:

{ELSE} return(ELSE);}

for Booleans (NOT and TRUE) we will use: cool_yylval.boolean

example:

{TRUE} {cool_yylval.boolean = 1; return(BOOL_CONST);}

Special notation:

There are 3 of them: DARROW, ASSIGN, LE.

For DARROW:

{DARROW} {return (DARROW);}

Integers:

[0-9]+ means one or more int characters concatenated together.

And for integers semantic value should be a symbol stored in the field cool_yylval.symbol:

{INTCONST} {cool_yylval.symbol = inttable.add_string(yytext);return (INT_CONST);}

Identifiers:

Type identifiers begin with a capital letter:

TYPEID [A-Z][a-zA-Z0-9_]*

Semantic value is stored in a symbol in the field cool_yylval.symbol:

{TYPEID} {cool_yylval.symbol = stringtable.add_string(yytext);return (TYPEID);}

Object identifiers are te same they just begin with a lower case letter.

Symbols:

Symbols are : ( ) . @ ~ * / + - < = { } : , ;

And we will return themselves. For example:

"(" return '(';

Comments:

There are two types of comments: first type is a comment between 2 sets of dashes: --comment—. This is only in one line

And second type is like this:

(* comment

Comment

Comment *)

So between (* and *)

In the second type there are also nested comments and we will keep count with comment_no which is initially wet to 0. With every new (* seen, we will increment it.

“—“.* {}

here it reads all characters but new line. So it read until end of line and does not do anything.

"*)" {cool_yylval.error_msg = "Unmatched *)"; return (ERROR);}

Here when lexer reads “*)”, it has not read any “(*” yet (start of comment) there should be an error.

"(*" {BEGIN(comment); ++comment_no ;}

Here lexer reads “(*” so we will set is as a new comment and increment comment_no for nested comments.

<comment>"(*" {comment_no++;}

If lexer reads (* after a comment, it is a nested comment and we will increment comment_no

```
<comment>"*)"        {
                     comment_no--;
                     if (comment_no == 0)
                          BEGIN(INITIAL);
                     else if(comment_no <0){
                          cool_yylval.error_msg="Unmatched *)";
                          comment_no=0;
                          BEGIN(INITIAL);
                          return ERROR;
                     }
                }
```

If it reads *) after a comment, there are two possibilities. if number of “*)”s are less than number of “(*”s no error will occur but if they are more then we will have “Unmatched *)” error. If number of “(*”s and “*)”s are equal, then it means comment is over and goes to initial state.

```
<comment>"\n"        {curr_lineno++;}
```

Curr_lineno shows which line of code lexer is reading.

```
<comment><<EOF>>  {BEGIN(INITIAL); cool_yylval.error_msg = "EOF in comment"; return ERROR;}
```

When reading a comment, if we reach end of file we will go to initial state and return the error: “EOF in comment”.

String:

A string begins with: “. So after that we will BEGIN(string) and go to next state. Now if lexer sees “ again. We should first check if length of string is not too long. If it is, we will return an erro. If its not, we will add the string to stringtable:

```
<string>"\""            {
                        BEGIN(INITIAL);
                        if
(string_buf_ptr>(string_buf+MAX_STR_CONST)){
                              *string_buf_ptr = '\0';
                              cool_yylval.error_msg = "String constant
too long";
                        return (ERROR);
                        }
                        else if (!null_found) {
                              cool_yylval.symbol =
stringtable.add_string(string_buf);
                              return (STR_CONST);
                        }
```

If string reaces end of file, we will return EOF error.

With null_found we will check if string has null characters and return error if needed.