Compilers-II(CS3423)

Assignment I - Lexical Analyzer

Name: Aditya Bacharwar
Roll no: Es21btech11003

Compilation Steps :

1. Main source file name : **lex_source_program.l**
2. Command to Invoke flex tool on main source file : "**flex lex_source_program.l"**
   - Flex takes your lexer specification written in the Lex language and generates C code for the lexer based on those rules.
   - Flex generates the file, lex.yy.c which contains the C code for the lexer.
3. Command to compile the lex.yy.c file using gcc C-compiler : "**gcc lex.yy.c**"
   - It outputs a lexer executable file **'a.out'**
4. To run the executable file 'a.out' with input file as command line argument :
   "**./a.out 'relative path to input file'** "
   - The lexer reads the contents of this file and performs lexical analysis on it.
   - The outputs two files : 1) seq_tokens_i.txt in 'TK' folder 2) C_i.txt in 'TC' folder
   - **seq_tokens_i.txt** - sequence of tokens generated while doing lexical analysis
   - **C_i.txt -** Corresponding C-code for the input code in our language
   - Example command : ./a.out test_cases/1.txt

Known Short-comings for my Implementation:

1. Whenever there is an error, it only shows the line number at which the error occurred. It doesn't provide any other details about the error that has occurred.
2. Negative integers are considered as identifiers.
3. Not able to parse with proper indexing syntax as done in C-language by square braces instead using curly braces
   (example : string a = "hello world\n";
   In C accessing 'e' in the string will be like a[1]
   But my lexer parses it as a{1})
4. Strings are not properly defined for escape characters - unable to escape double quotes in between the string

Definitions:

1. digits [0-9]+

    - This allows for any digit from 0 to 9 to occur one or more times, allowing all possible combinations of allowed numbers to be made.
2. Letter [A-Za-z]
    - Allowing all the uppercase letters and lowercase letters in our language to be parsed in our language.
3. WhiteSpace [\t\n]+
    - This allows us to parse multiple whitespaces, tabs and newlines occurring in continuation.

4. Label pp{digits}+
    - Defined according to our language, every new line start with pp followed by some number
5. char_string and string \"[^\"]\"
    - Every string constant is enclosed in double quotes and can have any character except for double quotes between them; for char_string, it is enclosed in single quotes.
6. punctuation, special symbol, special_operator_symbol
    - These are just defined a basic classes defined in our language
7. operator (“ “{operator_symbol}” ”)
    - According to our language, it is mandatory for operators to have whitespace before and after them
8. reserved
    - This definition has explicitly defined all the reserved_keywords the are present in our language. Allowing each to individually occur at respective places where they are required
9. data types
    - Similar the reserved, this has all the available datatypes present in our language explicitly declared with a space after them as it is necessary to have a space after datatype else it can be considered a identifiers
10. Identifiers
    - Can start with a special symbol followed by one or more number of digits and letters (ex : _2)
    - Or, can start with a digits again followed by one or more number of digits and letters containing at least one letter in variable name (ex : 1x)
    - Else with start with a letter followed by zero or more letters or digits (ex : x3wy)

```
/* definitions*/
digits [0-9]+
letter [A-Za-z]
whitespace [ \t\n]+
label "pp"{digits}+
char_string \'[^\']\'
string \"[^\"]*\"
punctuation [;:,]
special_symbol [\[\]\(\)]
special_operator_symbol [+\-\*@\\/:_=#]
operator_symbol [+\-\*/_=]
operator (" "{operator_symbol}" ")
reserved "gteq"|"gt"|"lteq"|"lt"|"and"|"neq"|"or"|"jump to"|"eq"|"otherwise"|"in case
that"|"do"|"return"
datatype "integer_2 "|"string "|"character_1 "|"null "
identifier
{special_operator_symbol}({letter}|{digits})+|{digits}*{letter}+({digits}|{letter})*
/* rules */
%%
```