COVER PAGE

CS323 Programming Assignments

Fill out all entries 1 - 7. If not, there will be deductions!

1. Names [1. Andrew Lee] Section [21737]

   [2. Cesar Carrillo] Section [21737]

   [3. Alan Campos] Section [21737]

2. Assignment Number [1]

3. Due Date [10/06/2024]

4. Submission Date [10/06/2024]

5. Executable File name [a.out]
(A file that can be executed without compilation by the instructor, such as .exe, .jar, etc - NOT a source file such as .cpp)

6. Names of the testcase files - input test file output test file

test 1. [	test1.txt]	[result1.txt]
test 2. [	test2.txt]	[result2.txt]
test 3. [	test3.txt]	[result3.txt]

7. Operating System [Linux, MacOS]
(Window – preferred)

To be filled out by the Instructor:

Comments and Grade:

Problem: Create a lexical analyser using finite state machines to analyze Rat2F code.

Execution

The script is executed as follows:

>> ./a.out

This will analyze a *test.txt* file by default, producing output file *result.txt.* The output file contains the ordered tokens with their corresponding lexemes.

Design

Lexer Function

lexer() is the core of the lexical analyzer. It processes each lexeme and categorizes it based on the FSM results and the defined lists of keywords, operators, and separators, and makes sure to ignore properly formatted comments. It then sets the token type accordingly (e.g., keyword, identifier). If unrecognized, then sets the token type to unknown.

Keywords, Operators, and Separators Definition

The program first defines three lists containing the Rat24F language keywords, operators, and separators:

```
char opList[11] = {'+','=','-','*','/','%','<','>','!','&','|'};
char sepList[9] = {'{','}','[',']','(',')',',',';',':'};
string keywords[18] = {"if", "for", "else", "return", "get", "put", "true", "false", "int", "real", "while", "do", "case", "break", "switch", "endl", "cout", "cin"};
```

There is a separate section within *lexer()* to catch double operators such as ==, !=, etc by utilizing the .peek() member function in order to check the next character.

Finite State Machines

Three finite state machine functions validate identifiers, integers, real numbers, and strings. Each function follows specific rules based on the Rat24F language syntax.

- ***fsmID*()**: First checks if the input matches a keyword in order to filter out keywords. Then determines if a string is a valid identifier by checking if it starts with a letter and contains only letters and digits thereafter. An identifier must end with a letter.
  Regular expression: l (l | d)* l | l
  NFSM: 0 = Starting State. 1 = Initial letter. 2 = Middle characters. 3 = Final letter.
  S0 - l → S1

S1 - l | d → S2
S2 - l → S3
S2 - d → S2
S3 - d → S2
Accepting State(s): 1, 3

- ***fsmDigit():*** Checks if the given input is a valid integer or real number.
  Integer regular expression: d+
  Real regular expression: d+ . d+
  NFSM: 0 = Starting State. 1 = Pre-period digits/Integer. 2 = Post-period digits/Real.
  3 = Period
  S0 - d → S1
  S1 - d → S1
  S1 - . → S3
  S2 - d → S2
  S3 - d → S2
  Accepting State(s): 1 for integers, 2 for reals

- ***fsmString():*** Checks for quotation marks used properly for given input to be a string.
  Regular expression: " (l | d)* "
  NFSM: 0 = Starting State. 1 = Middle characters. 2 = Success. 3 = Failure
  S0 - " → S1
  S1 - " → S2
  S2 - anything → S3
  Accepting State(s): 2

Main Function

The main function opens the required files and initializes the token and lexeme strings. It then runs *lexer()* until the end of file, and calls *print()* the changed token and lexeme strings after a successful *lexer()* call. The function then closes all files and ends.

Print Function

This function was mainly implemented to reduce repetitive code during development and testing, but also holds as a placeholder for future additions to this Rat24F compiler. The function prints the output to a result.txt and then clears the current lexeme.

Data Structures

1. Character List Arrays: keywords, operators, and separators lists store Rat24F language elements. These lists are used for direct matching with each lexeme.

2. Strings: The program processes each lexeme as a string and uses string operations in order to replace the token type and output a token and lexeme string to a result.txt file.

3. File Objects: File objects are used for reading the input code and writing the output results.

Limitations

None

Shortcomings

None