Brian Alexander
CWID: 806093548
9/28/2019

CPSC 323 Compilers - Lexer Documentation

I. Problem Statement

Ideally our lexer will be able to open a text file of any length, compare the characters one at a time in sequence, and produce a list of valid and invalid tokens that have been entered in that text file. Additionally, if there are any invalid tokens in the text file, it will be helpful to have the location of that error specified. Lastly, if there is an error, we can hopefully recover from that error and continue parsing the text file.

The proposal to solve this problem is to use Python to read in and parse the text file. After creating a state transition table using the desire tokens and expected input characters, it will be possible to implement an algorithm that traverses the text file and identifies the tokens one at a time as it parses the document.

II. How To Use My Program

My program is provided as a Python 3 script. It takes one command-line argument which is the path of the text file (or name if they are in the same directory) you wish to be parsed. For example, if the script is named lexer.py and the file name is test.txt, with both files in the same directory, the command would be python3 lexer.py test.txt. The results of the program will be displayed in the console.

III. Program Design

At the top of my lexer file there are many constants defined that will be used throughout the program to help increase readability and functionality. Firstly, I define constants for all of the different state types (ST), for example ST_OPERATOR = 0. The included states are: operator, separator, identifier, keyword, int, real, comment, decimal, space, and error. Next I add an array, **get_token_string**, which can be indexed by the different states to turn them back into a human-readable format. For example, **get_token_string[ST_OPERATOR]** will evaluate to the string 'OPERATOR'. In the same way I also defined character types (CT). These are representative of all the different characters that can be accepted by the lexer. The included character types are: alpha, digit, whitespace, bang (!), dollar ($), operator (*+-=/><%), and separator (()}{[],.:;).

The next major part of my application is the transition table. It is implemented using Python dictionaries, which are hash tables, but could be implemented using a standard 2D array. I chose the dictionary method for improved readability. In the transition table we define how the state flows from one state to another based on a given character input.

There are only two functions in my application, **get_char_type** and **lexer**. **get_char_type** is a simple function that accepts one argument. The argument is the character that has been read from the file. Using a series of **if** statements, the function will determine which character type that character belongs to and return the appropriate code.

The meat of the logic and processing occurs in the **lexer** function. It also accepts one argument, which is the path where the file is located. The function has some internal variables worth noting: **token** starts as an empty string that is concatenated to while constructing a token, **tokens** is an array that will

hold all of the valid tokens as they are found, **illegal_tokens** is an array that will hold all illegal tokens if they are found, **current_state** holds the current state of the machine defaulting to ST_SPACE at creation, and **line_number** holds the current line number while parsing that is used to help identify invalid tokens.

The function initiates a loop that reads one character at a time until the end of the file is reached. While it is looping it performs its construction and classification algorithms to identify all of the tokens, including those which are illegal.

It does this classification using a two step process. First it determines the new state (**new_state**) that the machine will be in given the new character it has read in using the state transition table (**transition_table**). Based on that determination it has two choices it can make. One is to concatenate the new character onto the token and the other is to append the already built token to the array and start a new token using the presently found character.

It also handles a few edge cases. For example, all identifiers are classified as keywords until they can be definitively labeled as identifiers (for example with the presence of a $ or digit). Before appending a keyword, therefore, it is important to check that the keyword belongs to our list of defined keywords. Another edge case is that of spaces and comments, which should be ignored. These are never added to our token list.

The running script was designed to be as modular as possible. Instead of using hardcoded paths to filenames it accepts one command-line argument, which is the path to the file that we wish to have lexed. Upon running, the **lexer** function is called with the path from the command-line specified as its only argument. The **lexer** function returns a tuple of arrays. The first array is all of the correctly identified tokens and their token type. The second array is all of the illegal tokens that were found in the text file. It then loops through both arrays to display their contents.

Please see Section VII for the finite state machine, regular expressions, and Thompson's construction.

IV. Limitations

None

V. Shortcomings

Although I do not consider it a shortcoming, it has come to my attention that the specification for the project is that the lexer returns one token at a time, unlike mine which returns an array of all of the tokens once it has finished the whole document. This is a simple issue that can be fixed by changed with a few lines of code and a different structuring, but I do not have the heart to attempt it now and risk having my script not working. An example I can think of off the top of my head uses a class based setup with a **Lexer** class and a **get_token** method.

Addendum: I implemented the Lexer class with a get_token method that returns one token at a time. It essentially operates in the same way. Now the class has member variables named _file, line_number, token, and current_state that manage the state of the lexer.

VI. Additional Features

Brian Alexander
CWID: 806093548
9/28/2019

As written about above in the design section, I was able to implement error handling that captures illegal tokens and continues parsing. It is also line number aware and can thusly include the line number where any token was found, however it was only included to show the line number where illegal tokens were found to assist in debugging.

VII. FSM, and REs, and Thompson's Construction

Below is the state transition table I used in creating my lexer. Given some current state and an input character type, the table will output a new state that can be used to carry forward the algorithm.

State Transition Table		**Input Character Type**							
		alpha	digit	.	$	space	oper	separ	!
	int	error	int	deci	error	space	oper	separ	comm
	real	error	real	error	error	space	oper	separ	comm
	deci	error	real	error	error	error	error	error	error
Current	space	key	int	error	error	space	oper	separ	comm
State	separ	key	int	error	error	space	oper	separ	comm
	oper	key	int	error	error	space	error	separ	comm
	key	key	iden	error	iden	space	error	separ	comm
	iden	iden	iden	error	iden	space	oper	separ	comm
	comm	comm	comm	comm	comm	comm	comm	comm	space
	error	error	error	error	error	space	oper	separ	error

Below are the regular expressions used to define the tokens used in my lexer.

Token Type	**Regular Expression**	**Plain English**
Identifier	(a+)($|a|d)*	one or more letters, followed by zero or more $'s, letters, or digits in any order
Real	(d+).(d+)	one or more digits, followed by a period, followed by one or more digits
Integer	d+	one or more digits
Operator	* | + | - | = | / | > | < | % |	one of any characters in the given set
Separator	' | (|) | } | { | [|] | , | . | : | ; | s	one of any characters in the given set
Comment	! c* !	one exclamation point, followed by any number of non-exclamation-point

Brian Alexander
CWID: 806093548
9/28/2019

		characters, followed by an exclamation point. (Not reported to the user.)
Space	s+	one or more spaces. (Not reported to the user.)

Here is an illustration for a Thompson's construction resolving identifiers, reals, and integers. Please note that the lowercase 'a' specifies all alphabetic characters, 'd' represents all digit characters, '$' represents all '$' characters, '.' represents all '.' characters, and the epsilons are for epsilon transitions. Identifiers are resolved in the color blue, reals in purple, and integers in orange. Starting state is 'T' and accepting state is 'Z'.