COS**341** 2020

PROJECT Specification: **Task Booklet 00**

Introduction

During the project you will first implement a lexer and a parser, (other components later). For these tasks you will be using a ‘made up’ language which will be referred to as Student's Programming Language (SPL). SPL was created for you by your professor for educational purposes.

Your task for this specification (00) will be to implement a lexer that can tokenize text from an input file.

Admissible Words of **SPL**

The next 3 slides will define the admissible words of the SPL language (formatted in blue). The slides will define what keywords, operators, separators, identifiers and literals are supported by SPL.

Note: The characters □ and # will be used to represent the space (0x20) and newline (0xA) characters respectively.

Keywords

and	or	not
add	sub	mult
if	then	else
while	for	eq
input	output	halt
num	bool	string
proc	T	F

Operators and Separators

Description	Word
Less than operator	<
Greater than operator	>
Space separator	□
Newline separator	#
Opening parenthesis	(
Closing parenthesis	)
Opening brace	{
Closing brace	}
Assignment operator	=
Comma	,
Semi-colon	;

Literals and Identifiers

Description	Word Definition (Regular Expression)	Valid Examples	Invalid Examples
Integer literals	(-?[1-9][0-9]*|0)	0 -42 901	014 --9
String literals	"[a-z0-9□]{0,8}"	"2hello□" "" "□□"	"123456789" "al
User-defined identifiers	[a-z][a-z0-9]*	h3 xyz a2b	3a

Note: See chapter 1.1.1 in the text book for regular expression shorthand's.

Note: user-defined names *cannot* be the same as a keyword.

Your Task

Given a **.txt** file of unstructured text your program must scan the file and save a new file representing the linked list of tokens it found. This should be done if the input contains only the admissible words of SPL. If not then you must output a descriptive error of what is wrong with the given input and where the failure occurred.

Step-by-step Implementation

1. Write the **NFA on paper**: construction algorithm = in the book.
2. Convert your NFA to a **DFA on paper**: conversion algorithm = in the book.
3. Convert your DFA to a **Min-DFA on paper**: conversion algorithm: see book.
4. **Implement your Min-DFA** in software (using IF-THEN-ELSE statements)

Remember that your implemented Min-DFA is only the "core of the lexer, however not yet the lexer itself.
"Choose wisely" a suitable lexing strategy (first match or longest match);
"Embed" your Min-DFA core into your chosen lexing strategy (algorithm);
"Wisely utilise" blank_space to decide when to re-set the Min-DFA back to start;
In case that back-tracking might be needed: use a stack to buffer the characters

Input

Your program must be able to take a file name as an argument. Your lexer must then scan the given file character by character tokenizing the input.

Output

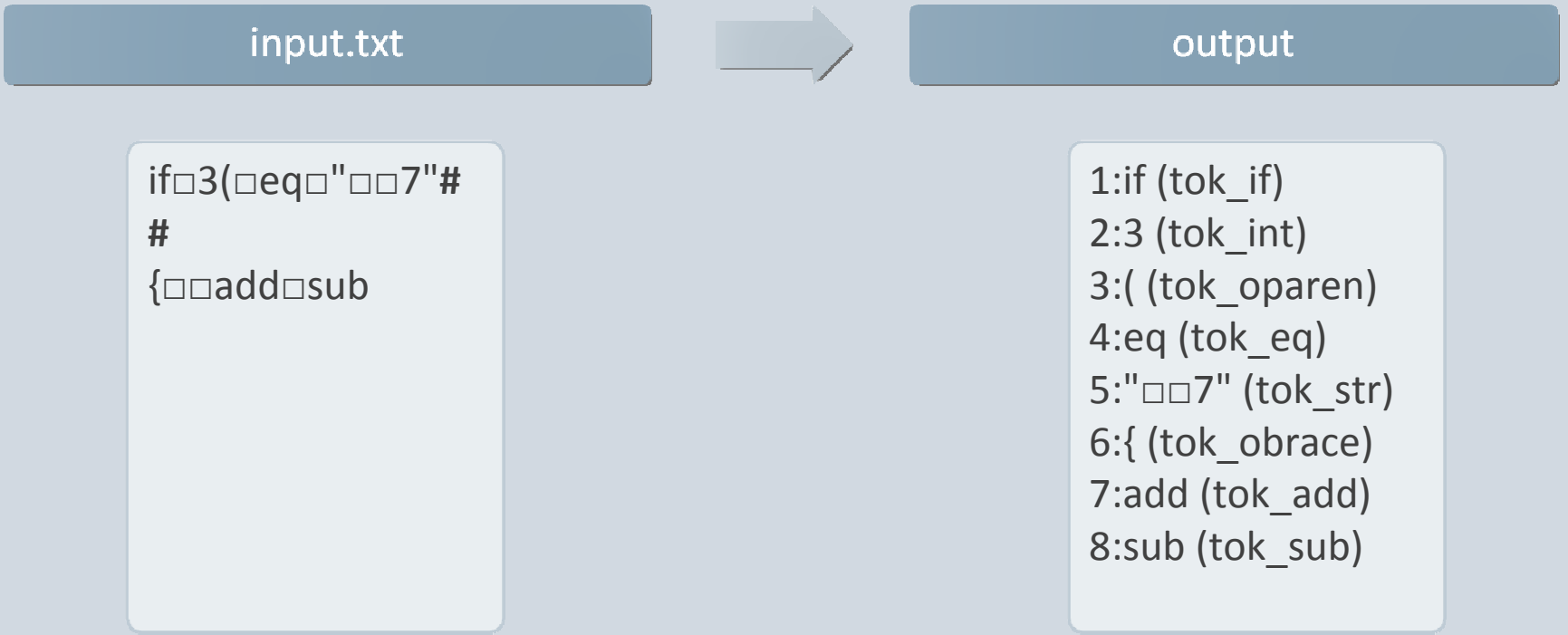
If your lexer encounters something that it cannot tokenize it must output a descriptive error of where and why the error has occurred. For example the following could be the output that your lexer prints:

Lexical Error [line: 2, col: 13]: '**@**' is **not a valid character**

Lexical Error [line: 4, col: 3]: '***"universit***' strings have at most 8 characters

Output

If no errors occur you must create a persistent file with all the tokens that the lexer was able to scan from the input file. For example:



Handling Whitespace

□ characters **can** appear **within string** tokens. They can also appear between tokens (for example to separate two numbers from each other). However, the □ symbols should *not* be regarded as proper “tokens” in their own right; their purpose is mostly auxiliary.

Also the line-break **#** characters should not appear as tokens in their own right; nor should they occur within other tokens. They must only be treated as separator characters.

Additional Notes

- In the next following part of this project you will read and parse the tokens from the file you save from this part of the project.
- Plagiarism is not allowed! You or your group may not use any code written by someone not within your own group.
- With regard to the software development language that you have chosen (e.g. Java or C++): you may not use any utility-libraries or built-in language features that automatically provide regular expression functionalities. You must develop the Min-DFA with pen and paper before you implement it.
- You may not use any automatic lexer generator tools. Everything must be hand coded by yourself, from scratch.
- You are encouraged to discuss the work to be done with your group members, but ensure that you as an individual understand the full process. For this purpose it is also recommended that you implement your own full lexer for your own understanding (and not use any lexer generator tools).