Assignment 7

The goal of this assignment is to create two programs: a tokenizer and a recognizer. The tokenizer will read an input file line by line and convert the textual input into an ordered collection of tokens and lexemes. The recognizer will use recursive descent to parse the output of the tokenizer and determine if the given tokens form a valid program.

There is no specified programming language for this assignment. You will need to choose which language you want to implement these programs in. More details about language selection (including bonuses and penalties!!) are provided later in the manual. Because this assignment is language agnostic, all program input and output will be done via file I/O with filepaths provided as command line arguments. Another consequence is differing file extensions from assignment to assignment. When describing the submission files, this assignment manual will use the asterisk (*) as a wildcard character. i.e. Tokenizer.* means the filename must be tokenizer but the file extension can be anything.

## Common.*

Common.* is an optional but recommended file. The purpose of Common.* is to define any and all includes / imports, constants, globals, enums, structs / objects which are shared (or common) between both Tokenizer.* and Recognizer.* . Common.* will be stored in the same directory / package as both Tokenizer.* and Recognizer.* . Although Tokenizer.* and Recognizer.* are two distinct programs which compile independently of each other, it is very likely they will share some commonality.

Specifically, I have two recommendations for Common.* . First, I recommend defining a token enum, where each enum value is one of the distinct token classes defined in the provided lexical structure. Having an enum to represent your tokens will greatly increase programmer efficiency without sacrificing security. Second, each lexeme generated by Tokenizer.* will have a corresponding token class. Because each lexeme will have an associated token class, I recommend defining a struct / simple object to implement / enforce that association. Said struct / object would have two data fields, one for the lexeme and one for the associated token class. You can name this struct / object anything you'd like but if you're looking for inspo I use "Lex" as the name in my implementation.

## Tokenizer.*

Tokenizer.* will read in two command line arguments when run. The first command line argument is the filepath of the input file and the second is the filepath of the output file. Your program will **need** to take in these two command line arguments for Gradescope accept it. No other methods for acquiring the filepaths are allowed.

When run, Tokenizer.* will read in all characters from the input file. It will convert these characters into lexemes, then associate each lexeme with a token class.

NOTE: Each line of every input file is guaranteed to be 256 characters or less.

The given lexical structure is free format and the position of characters in the text file does not affect their meaning. Alphanumeric lexemes will be delimited by both whitespace and by symbol lexemes. It's my recommendation that you construct lexemes character by character. i.e. iterate over every individual character in the input file and determine if the current character is part of an alphanumeric lexeme, whitespace, or part of a symbol lexeme. The type of character you identify will determine the next action your tokenizer will take.

NOTE: Because both whitespace and symbol lexemes can be delimiters, standard split functions do not provide the functionality required for this task and should really be avoided.

Adding complexity is some symbol lexemes are multiple characters in length. Because a multi-character symbol lexeme can be a delimiter, there will be circumstances where the subsequent character in the file will need to be examined to determine which symbol lexeme is present.

After generating a lexeme, it must be associated with a token class. You can do this association after a lexeme is generated or you can generate all lexemes then associate all generated lexemes with their respective token classes. While either approach is valid, I recommend the latter approach. Breaking lexeme generation apart from token association simplifies the overall program structure.

Associating each lexeme with a token class is fundamentally a string comparison problem, as the provided lexical structure specifies what strings are valid lexemes for each token class.

Two token classes in the provided lexical structure are defined via regular expressions. This means any string which matches the specified regex is part of that token class. The exception is any strings explicitly defined in the lexical structure, which are reserved words. i.e. "return" **would** be an IDENTIFIER token as it matches the regex provided for IDENTIFIER in the given lexical structure. But "return" is explicitly defined as a RETURN_KEYWORD token earlier in the lexical structure. To avoid any confusion, you ought to compare lexemes against the token classes in the order defined in the lexical structure. i.e. check if the generated lexeme is a reserved word before checking if it's an IDENTIFIER token.

NOTE: While identifying if a given string matches a specified regular expression is trivial in most languages, it is unsurprisingly more obnoxious to implement in C. As such, on Canvas I provide C methods for identifying if a given char* matches a specified regular expression.

After all lexemes have been generated and associated with a token class, this information is written to the output file. The format is as follows. Each token / lexeme pair is written on its own line. The token is written first and the token format must match the format in the provided lexical

structure. After the token a single space character will be written, then the lexeme in its entirety, then a newline character. An example output for tokenTest0.txt is provided below for context:
VARTYPE int
IDENTIFIER main
LEFT_PARENTHESIS (
RIGHT_PARENTHESIS )
LEFT_BRACKET {
RIGHT_BRACKET }

NOTE: All input files for this project will be tokenizable. We don't need to consider handling invalid lexemes when building Tokenizer.*

## Recognizer.*

Recognizer.* will read in two command line arguments when run. The first command line argument is the filepath of the input file and the second is the filepath of the output file. Your program will **need** to take in these two command line arguments for Gradescope accept it. No other methods for acquiring the filepaths will be accepted.

When run, Recognizer.* will read in a list of tokens and their associated lexemes from the input file. The output file from Tokenizer.* will be used as the input file for Recognizer.* . Recognizer.* will determine if the ordered set of tokens from the input file is legal in the language defined by the given EBNF grammar. The purpose of our recognizer is to apply the given grammar rules and report any syntax errors.

To accomplish this task Recognizer.* will implement a recursive decent parser based upon the provided EBNF grammar. The implemented parser must be a recursive decent predictive parser which utilizes single-symbol lookahead, consuming each token one at a time. Parsers which utilize multi-symbol lookahead will not be accepted.

NOTE: When implementing a recursive decent parser, each grammar rule is defined as its own function. But each function is analyzing shared data: the set of all tokens and the current token being consumed. I recommend making this data globally defined rather than passing the data to each function individually. Doing so will greatly reduce program complexity.

An input is deemed to be syntactically invalid if a token or non-terminal was required by the current EBNF grammar rule but not present. If a syntax error is found, parsing should halt immediately and your program should report an error by printing an error message to the output file. If a token was expected but not present, the error must specify: Which grammar rule had the error, which number token we were examining, the expected token, and the actual token. An example format is as follows:

Error: In grammar rule body, expected token #6 to be RIGHT_BRACKET but was IDENTIFIER

If a non-terminal was expected but not present, the error must specify: Which grammar rule had the error and the expected non-terminal. An example format is as follows:

Error: In grammar rule function, expected a valid body non-terminal to be present but was not.

NOTE: Parsing should halt immediately when a syntax error is found. But how? The vast majority of syntax errors will be discovered several function calls deep. In this case, trying to gracefully return from each invoked function so the program can terminate normally is the wrong approach. Doing so creates a lot of complexity related to code structure and execution flow. Instead, when an error is detected, your program should print its error message to the output file, then terminate the program via a call to the exit function. While this assignment is language agnostic, almost all modern programming languages implement an exit function in some form or another. Java has System.exit(), Python sys.exit(), and C implements exit(). A quick DuckDuckGo search will provide details for almost any other programming language. My recommendation is to create a helper function called error or something similar. This helper function will be invoked whenever a syntax error is detected, be passed all information necessary to print the correct error message to the console, write that error to the output file, then call the exit function to terminate the program.

NOTE II: The exit function should have an argument value of 0, regardless of which language you're using. If the integer argument provided is non-zero, it signals to the operating system an “abnormal termination”. But reporting a syntax error then terminating is a normal course of action for our program, so the argument value provided to the exit function should be 0.

Given a grammatically valid input, every given token must be parsed. If the top-level grammar rule (function) is invoked and concludes without error, this indicates that (for a correctly defined recognizer) the ordered set of input tokens were syntactically valid. But there's a catch. If the top-level grammar rule concludes without error, the given set of tokens is only valid if all tokens have been consumed. i.e. if the first 10 tokens are a syntactically valid input but the input file contained 13 tokens, this constitutes a syntax error. This must be identified and reported with the following error:

Error: Only consumed 10 of the 13 given tokens

If all input tokens are consumed and no syntax errors reported, Recognizer.* will output "PARSED!!!" to the output file. Recognizers which do not consume every given token for a grammatically valid input will not be accepted.

## Language Choice

As mentioned, you are allowed to select whichever language you'd like to implement this project! But to keep things balanced, I am providing an *incomplete* language "tier-list" of sorts, with point bonuses and penalties applied to each language tier. So, without further ado:
F Tier (-5 points):

> Javascript, PHP, C++
>
> (yes i will take off 5 points if you develop the project in one of these languages. seriously. these are not the languages you're looking for!!)

B Tier (+0 points)

> Java, Python

A Tier (+10 points)

> C

S Tier (+20 points)

> Common Lisp
>
> (Note: to get these bonus points your code needs to actually implement a tokenizer and recognizer. i.e. submitting a mostly incomplete assignment in an S-Tier language will result in 0 bonus points awarded. The distinction between non-functional and mostly incomplete is up to my discretion, so check with me if you're unsure)

If you wish to implement the program in a language not listed please reach out and let me know! I will need to confirm that I can have Gradescope compile whatever language you'd like to use / update Gradescope to support the language. I will decide which tier any requested language will be placed in and will do so before the assignment deadline.

NOTE: The F Tier languages are not supported on Gradescope by default. You'll need to reach out and ask if you **really** wanna use them.

## Final Notes

A set of text files will be provided for testing. There will be nine files which have syntactically valid inputs and nine which have syntactically invalid inputs. All tests are lexicographically valid. Explanations for each test are also provided. These 18 files are identical to the input data on Gradescope. If your program can successfully parse all 18 files, it should pass all tests on Gradescope.

There is no recommended development environment for this assignment and you are welcome to use whatever application you are most comfortable with.

Provided EBNF grammar:

```
function        --> header body
header          --> VARTYPE IDENTIFIER LEFT_PARENTHESIS [arg-decl] RIGHT_PARENTHESIS
arg-decl        --> VARTYPE IDENTIFIER {COMMA VARTYPE IDENTIFIER}
body            --> LEFT_BRACKET [statement-list] RIGHT_BRACKET
statement-list  --> statement {statement}
statement       --> while-loop | return | assignment
while-loop      --> WHILE_KEYWORD LEFT_PARENTHESIS expression RIGHT_PARENTHESIS body
return          --> RETURN_KEYWORD expression EOL
assignment      --> IDENTIFIER EQUAL expression EOL
expression      --> term {BINOP term} | LEFT_PARENTHESIS expression RIGHT_PARENTHESIS
term            --> IDENTIFIER | NUMBER
```

Provided lexical structure:

```
LEFT_PARENTHESIS    --> (
RIGHT_PARENTHESIS   --> )
LEFT_BRACKET        --> {
RIGHT_BRACKET       --> }
WHILE_KEYWORD       --> while
RETURN_KEYWORD      --> return
EQUAL               --> =
COMMA               --> ,
EOL                 --> ;
VARTYPE             --> int | void
IDENTIFIER          --> [a-zA-Z][a-zA-Z0-9]*
BINOP               --> + | * | != | == | %
NUMBER              --> [0-9][0-9]*
```

Grading Rubric:

| Category | Points |
|---|---|
| Tokenizer.* tests | 40.5 |
| Recognizer.* tests | 40.5 |
| Code is well commented | 10 |
| Code is well formatted | 9 |
| Total | 100 |