Kennesaw State University

College of Computer and Software

Department of Computer Science

CS 4308 - Concepts of Programming Languages - Section W01

Project Deliverable 1

Leonard Hayes | lhayes32@students.kennesaw.edu

12/5/2019

**Initial Problem Statement**

I was given a subset of grammar for the Julia language. This subset does not have a complete compiler to check the grammar and I was tasked with completing the first step of implementing the compiler, creating a lexical analyzer to examine the code and find and identify the lexemes given a token list created from the grammar given.

**Summary and Purpose**

The purpose of this report is to detail and describe the process of making the Scanner. This Scanner will read the input file and do a full lexical analysis. This entails reading the code and scanning for lexemes based on the token list. By the end it will print out each lexeme as well as the token associated with it found in the program This Scanner is also special because as it scans there are checks that my code will do to ensure that the code is correct. If there is an error, my code will print an error statement at the line and index it was found at.

**Description**

In building my Julia Scanner, there are several portions that the code is divided into to achieve the functionality that I was looking for. In this section we will talk about each class relevant to the Scanner and how it is implemented.

Definition of Classes

There are two classes that were used in creating the Scanner: The Scanner class and the Token Class.

**Scanner:** The scanner class has three tasks: Hold the token list for the Julia Grammar, scan lexemes and place their corresponding tokens into a tree-like list, and check each incoming lexeme for correctness.

**Token:** Stores information regarding the Token, this includes the type of token, its id, and it’s corresponding value as a lexeme.

Scanner Class

Upon creating an instance of the Scanner class, the object will inherit all of the Scanner class’ variables. The scannert of several variable types, such as collections, independent character types, tracking indices tracking strings, and two dictionaries: token and keywords.

The independent character type variables are used in the scanning portion to decipher the symbol in the **read\_symbols(self, symbol)** method. In side of this method, the index tracking variables, local\_index and global\_index, are used to follow and keep track of where we are in the file and detect where there are errors. After each space character, the token is interpreted then added to the tokenCollection. The interpretation, done by the adding and checking functions, make sure the symbol or keyword is in either the token or keywords dictionary. This is done by using the re/regex class and using the search function to compare the symbols and tokens amongst each other using the regex “[]” format. When a match is found, the token portion is split into two parts, the id number and the name of the token. These two and the token’s lexeme are stored as variables in the token class. These token class variables are then stored in both the tokenCollection and line\_o\_tokens lists. When a newline character is detected, it is added to the token tree and reset. The token tree is used for the parser so I won’t go into detail on what it is for now.

Token Class

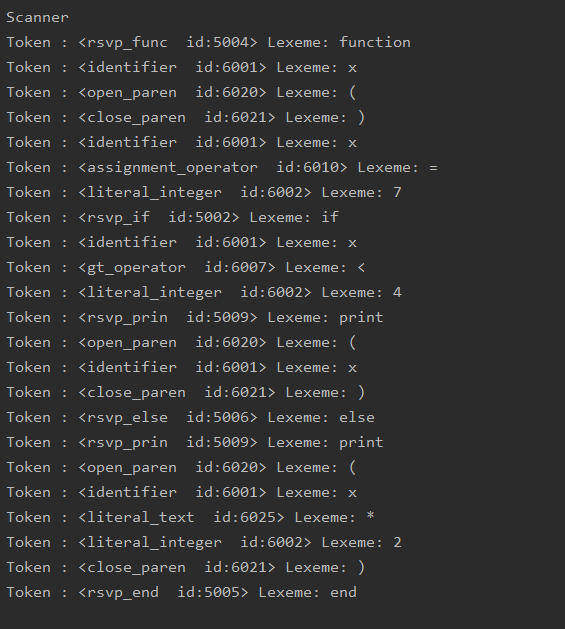
The token class is simply and object that stores information about the token and it’s corresponding lexeme. Upon initialization it has three parameters: type which is the token’s name, id which is the number id tagged to the token, and the actual lexeme that makes the token. The last value, operator, corresponds to the type of operator given the token. If the token is an equal sign then the operator value is assignment, if it an arithmetic operator then arithmetic, and if the operator is relational the operator value is relational.

The Token Class holds one function that prints out the token in a special format when needed. This format is: <Token name and identifier> Lexeme.

**Token Chart**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **TOKEN NAME** | **TOKEN ID** | **LEXEME** |  | **TOKEN NAME** | **TOKEN ID** | **LEXEME** |
| identifier | 6001 | [a-zA-z][a-zA-Z0-9\_]\* |  | rsvp\_while | 5001 | while |
| literal\_integer | 6002 | [0-9]+ |  | rsvp\_if | 5002 | if |
| end\_comment | 6003 | [=][#] |  | rsvp\_for | 5003 | for |
| le\_operator | 6004 | [>][=] |  | rsvp\_func | 5004 | function |
| lt\_operator | 6005 | [>] |  | rsvp\_end | 5005 | end |
| ge\_operator | 6006 | [=][<] |  | rsvp\_else | 5006 | else |
| gt\_operator | 6007 | [<] |  | rsvp\_true | 5007 | true |
| eq\_operator | 6008 | [=][=] |  | rsvp\_fals | 5008 | false |
| ne\_operator | 6009 | [!][=] |  | rsvp\_prin | 5009 | print |
| assignment\_operator | 6010 | [=] |  |  |  |  |
| unadd\_operator | 6011 | [+][a-zA-z][a-zA-Z0-9\_]\* |  |  |  |  |
| unsub\_operator | 6012 | [-][a-zA-z][a-zA-Z0-9\_]\* |  |  |  |  |
| add\_operator | 6013 | [+] |  |  |  |  |
| sub\_operator | 6014 | [-] |  |  |  |  |
| mul\_operator | 6015 | [^/|^\*][\*][^/|^\*] |  |  |  |  |
| div\_operator | 6016 | [^/|^\*][/][^/|^\*] |  |  |  |  |
| pow\_operator | 6017 | [\\^] |  |  |  |  |
| literal\_quote | 6018 | [\"] |  |  |  |  |
| literal\_comma | 6019 | [,] |  |  |  |  |
| open\_paren | 6020 | [(] |  |  |  |  |
| close\_paren | 6021 | [)] |  |  |  |  |
| white\_space | 6022 | [\t]+|[\r]+|[\f]+|[ ]+ |  |  |  |  |
| dot\_pts | 6023 | [.] |  |  |  |  |
| colon | 6024 | [:] |  |  |  |  |
| litreral \_text | 6025 | [^\"\\\\\\r\\n]\*(?:\\\\.[^\"\\\\\\r\\n]\*)\* |  |  |  |  |
| begin\_comment | 2026 | [#][=][^\"\\\\\\r\\n]\*(?:\\\\.[^\"\\\\\\r\\n]\*)\* |  |  |  |  |
| line\_comment | 6027 | [#][^\"\\\\\\r\\n]\*(?:\\\\.[^\"\\\\\\r\\n]\*)\* |  |  |  |  |
| mod\_operator | 6028 | [%] |  |  |  |  |
| or\_operator | 6029 | [|][|] |  |  |  |  |
| and\_operator | 6030 | [&][&] |  |  |  |  |
| others | 6031 | .+ |  |  |  |  |

**Screenshot Sample**

****

**Limitations and Design**

The hardest part of designing the scanner was researching and defining what kind of issues could the compiler face at the scanner level. I decided the main problem to solve in this portion was “Is the file at least correctly written?” While trying to solve the problem, it was easy to make checking methods focus on what came before the symbol so that we incrementally check how the file is written and does it make sense to continue without throwing an error. After getting those down, I was able to write add methods to add the lexeme into a token, then into a list containing all of the previous tokens. The scan was just the cherry-on-top, which simply was what read the symbol and called the other functions. Tracking the symbol was also important because it could tell exactly where the issue is located. When used in the main file, all that is needed is a file reader that stores the contents of the file into a variable, then a loop that calls **read\_symbols()** on each character in the file.

It must be stated though that due to time constraints I found it easier to write all of this in one file rather than to split it up in separate files to make it easy to check for errors. Also there is an important note to make that in a previous scanner made with my old group we thought that lexemes and tokens were completely seperate things. In this go-around, I was able to find that tokens are just the formal definition of lexemes in a sense. This helped me in printing my output and ensuring it was correctly labeled.

**Comments and Conclusion**

When creating the scanner, it was a little difficult to begin due to time constraints and seemingly starting from the beginning. It became a lot easier when talking to other groups about different implementation styles and what they could’ve done better on there’s. From there I was able to gather that the best method was the three step scan-check-add method to ensure that not only was I collecting the correct items, but that they were supposed to be there so I didn’t run into any more problems down the line. I also chose to use a dictionary to store my tokens because of how simple it is to use and really defines the entire token, where an array can only manage certain parts of the token. This portion of the project helped me understand the difference between lexemes and tokens, as well as how the lexical analyzer works to get the list it needs for the parser.

**References**

Julia Documentation. “Documentation” JuliaLang,

https://docs.julialang.org/en/v1/manual/documentation/index.html. Accessed 18 October

2019

Regular Expression Documentation. “Lesson: Regular Express” Oracle,

https://docs.oracle.com/javase/tutorial/essential/regex/. Accessed 18 October 2019

Coder, Guy.( 2013, February 19) “What is the difference between a token and a lexeme?” [Best

Answer]. Message posted to https://stackoverflow.com/questions/14954721/

what-is-the-difference-between-a-token-and-a-lexeme/14958865

Building a Lexer Tutorial. “building a lexer in python – a tutorial “ Python Members Club,

https://www.pythonmembers.club/2018/05/01/building-a-lexer-in-python-tutorial/

.Accessed 11/25/19