Kennesaw State University

College of Computer and Software

Department of Computer Science

CS 4308 - Concepts of Programming Languages - Section W01

Project Deliverable 2

Leonard Hayes | lhayes32@students.kennesaw.edu

12/5/2019

**Initial Problem Statement**

We were given four test files of the Julia Language, this subset consisted of small functions that do simple tasks. The task was to complete the second step of implementing a compiler, creating a parser to ensure the expressions and statements are implemented correctly and placing the contents into tree.

**Summary and Purpose**

The purpose of this report is to describe the process of creating the parser. This parser scans the contents of the file, does a full lexical analysis, then does a syntax analysis.

**Description**

To implement the parser, and to get it working as followed, I focused on how the grammar in the task was broken down. I noticed how syntax was denoted as either blocks, statements, and expression and decided to create files centered around those three elements.

The parse tree is given in the code follows the grammar provided to us in the documentation. As the code runs through the file, it will generate and print the parse tree for its immediate block then continue to the next one. The parse tree is designed in a top down fashion, going from the statement to the expression and ending on the individual identifiers and literal integers. All of the individual trees can be seen in the “**Input Data and Results”** diagram.

There are three files that make up the entire parsing process: **Decoder**, **Decoder\_Statement**, **Decoder\_Expression**. Decoder focuses on blocks and **Decoder\_Statement and Decoder\_Expression**  deal with statements and expressions respectively. It starts with the **decode\_block(boolean)** function which simply goes through the tree begins to handle lines based on it’s beginning statement (i.e. for, while, etc.) and will only break when the length is shorter than 1, the lexeme is “end”, or the lexeme is an else statement and the given boolean is true. If it passes these checks, then the decoder will create a Statement Variable from the block using the Statement Class.

The Statement class consists of two methods, it’s init function and its **print\_encoded\_statement()** function. The **print\_encoded\_statement()** function will print the grammar of the keyword then print the expression and in some cases, the block that the keywords may hold (i.e. if, else, while). When the class is first initialized, it requires the statement and what type of statement it is based on the keyword or identifier at the beginning of the line. Based on the type, the class will take the expression part of the statement and add it to the expressions collection. When all is collected, the statement is stored in a block variable in the decoder to be printed out later.

The Expression class handles the expressions that can be found in the statements. The class initializes with the given type of the expression and the collection of tokens within the expression statement in order. The method will always check what the assignment operator is, and if that is incorrect or “None” it will throw an error. Otherwise the expression will be stored in the expressions collection in the statement class. The Expression class also consists of three functions, **print\_encoded\_expression(),**  **write\_tokens()** and **print\_infix().** All of these functions are used to print out the tokens both as what they are and their lexemes in infix order respectively.

To summarize, the parser recursively scans block, statement, then expression to collect all data needed then print it out to the user.

Challenges

Many challenges in this design stemmed from formulating how to break it down, hence the choice of block-statement-expression. Then came the issue of what made statements statements and what made expressions expressions. From there it became a little easier to build the parser but still ran into some difficulties when splitting it between multiple files. Also I noticed that some of the given files were in prefix where my code only reads in infix, which I believe is ok because julia is normally written in infix. One of the final issues I had was with how to for loop was handled. In the Julia grammar, iterative loops are created with <assignment statement> : <assignment> whereas in the Julia documentation, for loops are shown to work with <assignment> : <literal\_integer>, because for loops go from what there set to on the left side up to the integer on the right side. After talking to the professor, she found it permissible for my Julia program to be written that way.

**Input Data and Results**

|  |  |  |
| --- | --- | --- |
| **Input** | **In-Code Translation** | **Result** |
| Function (Program) | <rsvp\_func> | Function id () <block> end |
| Block of code | <block> | <statement> | <statement><block> |
| For | <rsvp\_for> | For id = <iter> <block> end |
| If | <rsvp\_if> | If <boolean\_expression><block>else<block> end |
| while | <rsvp\_while> | While <boolean\_expression><block> end |
| statement | <statement> | <rsvp\_if> | <assignment\_statement> | <rsvp\_while> | <rsvp\_print> | <rsvp\_for> |
| print | <rsvp\_print> | print(<arithmetic\_expression>) |
| iter | “ITER” | <assignment\_statement> | <literal\_integer> |
| <boolean\_expression> | “BOOLEAN” | <arithmetic\_expression><relative\_op><arithmetic\_expression> |
| <relative\_op> | “relative” | Le\_operator | lt\_operator | ge\_operator | gt\_operator | eq\_operator | ne\_operator |
| <arithmetic\_expression> | “ARITHMETIC” | <id> | <literal\_integer> | <arithmetic\_expression><arithmetic\_op><arithmetic\_expression> |
| <arithmetic\_op> | “arithmetic” | Add\_operator | sub\_operator | mul\_operator | div\_operator |

**Syntax Error Detection**

A quick note. Most of the common errors are found in the scanner because in my code the scanner ensures the general code is correct before sending it to the parser. It’s easier to do this way so the parsing of the **tokenTree** runs much smoother and unexpected errors can be prevented.

|  |  |  |
| --- | --- | --- |
| **Error** | **Julia File** | **Screen Shot** |
| Double Space detection | function a ( )  for i = 3 : 5  if != i 4  ... |  |
| Incorrect Boolean expressions | function a ( )  for i = 3 : 5  if != i  ... |  |
| Incorrect iteration usage | function a ( )  for i = 3 : j  if i != 4 |  |
| No function detection | functio a ( )  for i = 3 : 9  if i != 4  ... |  |
| Misplaced symbols | function a ( )  for i = 3 9  if i != 4  ... |  |

**Limitations and Design Improvements**

The only real limitations was the amount of time I had. I was able to get around it by talking with other groups and asking questions where I had trouble to speed up the process.With more time I would’ve made copious more comments and also made the program a bit neater by giving specific variables for statement and expression types rather than relying on strings.

**Screenshots**

|  |  |
| --- | --- |
| **Input** | **Output** |
|  |  |
|  |  |
|  |  |
|  |  |

**Conclusion**

So far, this part has taken the most time for this project. Parsing is such an interesting topic but can be difficult when it comes to designing and interpreting a new language. Talking with other groups really helped me nail down what I wanted and how to do it. That along with several forums online that helped me understand more about the parser and how exactly to decode each part. In the end I did learn a lot more with this project and it was very beneficial overall.

**References**

Julia Documentation. “Control Flow” JuliaLang,

https://docs.julialang.org/en/v1/manual/control-flow/. Accessed November 25, 2019

Effbot. “Simple Top-Down Parsing” Fredrik Lundh.

http://effbot.org/zone/simple-top-down-parsing.htm. Accessed November 26, 2019

Wiki Haskell. “Parsing expressions and statements” Haskell,

https://wiki.haskell.org/Parsing\_expressions\_and\_statements. Accessed November 26,

2019