Kennesaw State University

College of Computing and Software Engineering

Department of Computer Science

CS 4308 – Concepts of Programming Languages – Section 01

Project Deliverable 3

Steven Mayo

December 9, 2020

# Initial Problem Statement

We were tasked with creating an interpreter on a subset of Ada. While the exact scope of our interpreter was up to our discretion, it at least needed to be able to perform binary arithmetic operations on integers, assign a value to a variable, and print the value.

# Summary and Purpose

The purpose of this report is to detail the process of making our interpreter, as well as demonstrate its output.

# Description

The lexical analyzer was the first stage of our project. We realized that we could implement this using rply, a python module for making lexical analyzers. Using it, all we had to do to create a lexical analyzer was use regular expressions to define tokens. Then rply would scan the input for these tokens and create a list that kept track of their token name, their text, and where the token was recognized. We did this in lex.py Tokens defined first would override tokens defined later if the regular expressions contained one another, so we had to make sure to define our tokens in descending order of specificity. Once we had compiled all the tokens we thought we would need, we were ready to move on to the parser.

We had a lot of trouble getting started on the parser. Despite knowing roughly what it needed to do, (take a list of tokens and check if they were in a syntactically correct order) we were very confused about the first steps to take in doing that. While we were meandering, we occupied time by creating a grammar in Backus-Naur Form that covered the necessary subset of Ada. Rply does contain functionality for making rules for parsers, but the documentation for it was extremely sparse. We did not understand what actions using it would do.

We split up to do research about how this could be done. Noah and George looked for different modules to use to create parsers. I happened to be the person who found the format we ended up using, however. While reviewing the textbook’s chapter on parsing, I noticed the passage “A **recursive-descent** parser is a coded version of a syntax analyzer based directly on the BNF description of the syntax of language” (Sebesta, 2019 Page 305) This gave me the idea to make a recursive descent parser.

I managed to get a prototype of how the parser would work done by the next time we met up. This only recognized print statements and string literals, but it was enough to demonstrate the format we would use. For each rule in the grammar we had made, I defined a function. The function would use a custom class, TokenStatus, to mark the current token it was trying to parse from. The function would check the name of the current token, and then the next, and then the next, to see if the tokens matched the rule’s definition. If the rule was matched, the function would return a TokenStatus that was past the end of the rule. If the rule was not matched, the function would either throw an error or return the TokenStatus to its position before entering the method.

This worked out well for a while. Noah and George understood it, and were able to work off of it. George fleshed out the definition of string expressions by adding the option of concatenation, and Noah listed out the relational operators and the type names. However, we had two more problematic features to add that changed the structure of the parser: variable declaration/assignment and order of operations.

Variable declaration/assignment wasn’t a particularly problematic addition

Some planned elements were not finished, such as support for if statements, but some optional elements were completed, such as Boolean operators.

Next, for each rule in the grammar, we began creating a method for our parser

We didn’t need to make extensive changes to turn our parser into an interpreter. In fact, while testing to make sure that our parser worked correctly, we made all of our operators work. That way, we could input expressions that required correct order of operations in order to evaluate correctly. If the variable was assigned the correct value, we knew our parser was traversing the expression in the right order.

We made the parser save and return values by assigning a value to TokenStatus.value inside methods. Then, when a rule needs the value returned by another rule, it gets that value from TokenStatus.value. If we had realized that Python easily supports returning multiple items in a tuple, we would have done that instead.

We did make a few changes for this deliverable, though. We added a Boolean that would toggle whether the parser would output any messages about the syntax structure as it parses, called ‘parsing’. We also added the functionality for Put\_Line to print things, which it only does while not parsing.

We turned the parser into a class that could be instantiated and run multiple times. This way, we could run through it once while printing the parser’s output, and then run through it again printing the interpreter’s output.

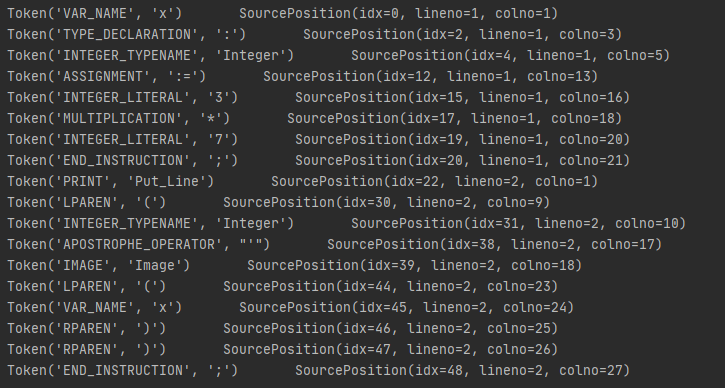
# Results

## Assigning and printing a value:

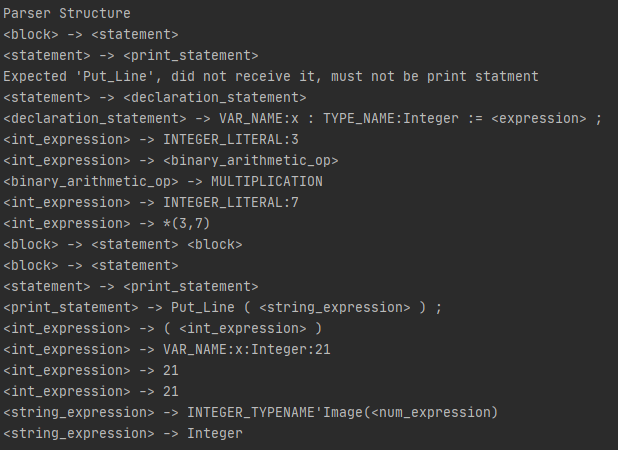
Input used:

x : Integer := 3 \* 7;  
Put\_Line(Integer'Image(x));

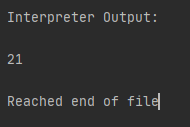
Lexical Analyzer Output:



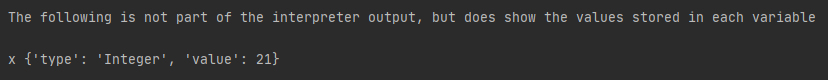
Parser Output:



Interpreter:



From p.show\_declared\_vars()

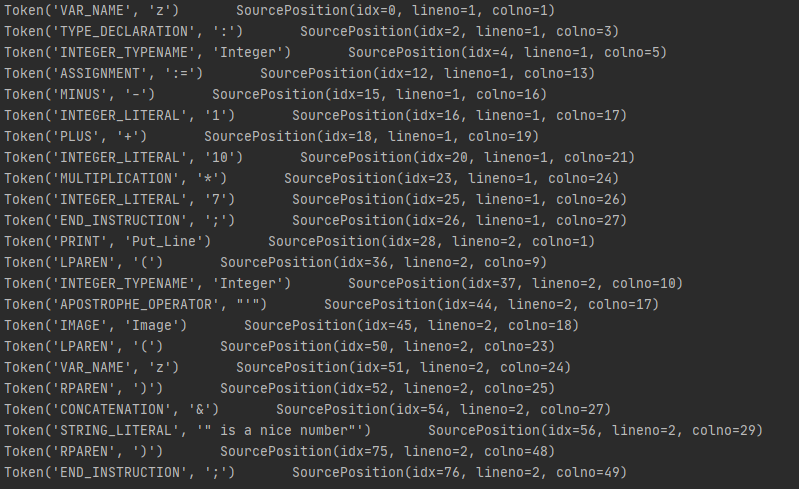


## Order of Operations

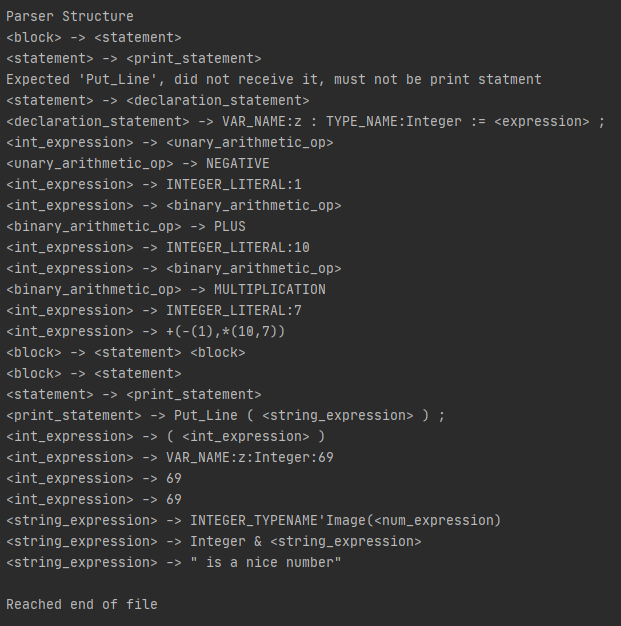
Input:

z : Integer := -1 + 10 \* 7;  
Put\_Line(Integer'Image(z) & " is a nice number");

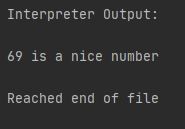
Lexical Analyzer Output:



Parser Output:



Interpreter Output:



From p.show\_declared\_vars()



# References

The references used are in APA format.

Gaynor, A. (2014). RPLY¶. Retrieved December 08, 2020, from <https://rply.readthedocs.io/en/latest/>

Sebesta, R. W. (2019). 4.3.2. In *Concepts of programming languages* (pp. 304-305). NY, NY: Pearson.

Ada Programming/Type System. (2018, May 8). Retrieved December 09, 2020, from <https://en.wikibooks.org/wiki/Ada_Programming/Type_System>