Programming Languages & Compilers - ENGR 3960

Assignment 2 – Minimp Interpreter

Created by

**Daniel Smullen**

**Joseph Heron**

**Jonathan Gillett**

24/02/2013

Document Revision 1.0

Contents

[Abstract Syntax Tree Generation 3](#_Toc352420948)

[Semantic Analysis 3](#_Toc352420949)

[Data Structure for Function/Variable Names and Scope 3](#_Toc352420950)

[Interpretation, Execution, and Invalid Semantics 5](#_Toc352420951)

[Displaying Results 5](#_Toc352420952)

[Testing 5](#_Toc352420953)

[Appendix A 6](#_Toc352420954)

# Abstract Syntax Tree Generation

Using the scanning and parsing methodology created in Assignment 1, our file was augmented to generate nodes for the abstract syntax tree as tokens are parsed through in the code. The grammar was rewritten to be, and numerous corrections to the grammar were implemented.

Also part of the parsing and AST generation was functionality for determining scopes using indentation and de-indentation. This proved to be a particular challenge. While it is easy to find indentations, finding the lack thereof is much harder. A stack was used to determine the depth of indentation of statements along a line. Missing indentations pop elements off the stack in order to return to the previous scope. This can even work for several simultaneous de-indentations, returning to the lowest scope between two lines. The difference between the current indentation level and the next indentation level is used to determine the actual change in scope. When the AST is generated and the symbol table is generated, this is used to record the scope of variables during semantic analysis.

# Semantic Analysis

Recursively interpreting each node in the AST allows for analysis of the structure of the code to determine whether the semantics of parsed statements are valid. Interpretation and execution of these statements is handled simultaneously.

The challenging aspect of this analysis is the recursive nature. Each node along the AST must also have its own interpretation implementation, and stringing these together while maintaining scope proved particularly challenging.

## Data Structure for Function/Variable Names and Scope

A multi-key hash table is used to map two keys to an entry –variable/function name, and the scope. This allows for names which are recurring in different scopes. Items are removed from the symbol table when they are no longer in scope. A method inside the class (called) gathers every variable in a scope less than the scope just exited, and deletes them from the hash table. This is critical to the semantic analysis, to ensure that scope violations do not occur.



Figure : Multi-Key Hash Table

|  |
| --- |
| i = 0  x = 5  while (i < x):  j = 0  while j < i:  print j  j = j+1  i=i+1 |

Figure : Source with Multiple Scopes

Figure 1 shows a pictorial representation of the initial state of the code in Figure 2. This demonstrates that there are separate scopes, and these can be found by mapping the entries within each scope to their values. Because the variable name and also the scope value maps to a particular value, scope 1 could potentially reuse the variable x, j could not be accessed within scope 0.

# Interpretation, Execution, and Invalid Semantics

Every node of the AST has an method. For each method, we perform validations for the semantics. This is handled differently for each type of node in the AST. One recurring pattern for validation is to determine whether a variable is used in the correct scope. This is done by verifying whether it exists in the symbol table inside the correct scope – only variables in a scope higher or in the current scope can be accessed. Another recurring validation is determining whether the correct number of arguments has been used in a function call. This is done by checking the node, and determining whether it has the correct number of children. Addition and subtraction, for example, are similar as they are binary operations but do not require the usage of an. In their semantic analysis, the validation process ensures they have two children.

## Displaying Results

Results, based on the grammar, can only be displayed using the built in function. The interpretation and execution for this method simply uses a call to which outputs the text to the console. This is used for displaying the results of evaluations within the code, and was used diagnostically within our testing for output.

The basic functionality of the AST component of the interpreter outputs the AST as an array, in an extension of the basic functionality. This was used for diagnostic output during development in order to determine whether the code was parsed accurately as a correct AST. This was output in a tree view as well, though this functionality already existed in Assignment 1.

# Testing

Our testing made use of a shell script which executes the parser for several test suites that we developed in anticipation of more specific tests for the evaluation of the assignment.

The default test suite executes our own tests, created during Assignment 1. These tests address each of the statements in the grammar, and are a carry-over from Assignment 1, with a few additions.

The Assignment 1 test suite includes the tests originally developed for Assignment 1’s evaluation, including corrected versions of the test. Also included were the original uncorrected tests, which should fail based on subtleties of the grammar, during lexical and semantic analysis.

The Assignment 2 test suite includes the tests provided for this iteration of the assignment. These were example source files provided with more complex examples that also test the evaluation of output in complex ways. These incorporate nested statements, nested scopes, and recursion.

**All test cases, covering every grammar and usage of nested statements were captured and recorded, and tested successfully. Please refer to Appendix A.**

# Appendix A

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test File** | **Test Suite** | **Test Description** | **Minimp Grammars Covered** | **Nested Statements** | **Includes Multiple Scopes** |
| mult\_1.mim |  | Tests multiple mathematical expressions combined that is assignment for a variable | AdditionFactor  AdditionalExpression  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  Statement | No |  |
| mult\_2.mim |  | Tests an if condition statement with assignment and mathematical operations | AdditionFactor  AdditionalExpression  AnotherStatement  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  IfStatement  Indent  IndentCopy  IndentEnter  MoreStatements  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  Sequence  Statement | Yes (2 levels) |  |
| mult\_3.mim |  | Tests an if condition, while, and print statement as well as mathematical expressions assigned to a variable | AdditionFactor  AdditionalExpression  AnotherStatement  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  IfStatement  Indent  IndentCopy  IndentEnter  MoreStatements  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  PrintStatement  Program  Sequence  Statement  WhileStatement | Yes (2 levels) |  |
| mult\_4.mim |  | Tests nested if condition statements, an if condition statement is nested inside the scope of an if condition statement | AdditionalExpression  AdditionFactor  AnotherStatement  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  IfStatement  Indent  IndentCopy  IndentEnter  MoreStatements  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  Sequence  Statement |  |  |
| print\_1.mim |  | Tests print statement with an integer value | AdditionFactor  AdditionalExpression  Comparison  ElementAccess  ElementExpression  Expression  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  PrintStatement  Program  Statement | No |  |
| print\_2.mim |  | Tests print statement with an identifier that has been assigned an integer | AdditionFactor  AdditionalExpression  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  PrintStatement  Program  Statement | No |  |
| print\_3.mim |  | Tests print statement with a mathematical expression using two integers | AdditionFactor  AdditionalExpression  Comparison  ElementAccess  ElementExpression  Expression  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  PrintStatement  Program  Statement | No |  |
| print\_4.mim |  | Tests print statement with a mathematical expression with addition and multiplication | AdditionFactor  AdditionalExpression  Comparison  ElementAccess  ElementExpression  Expression  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  PrintStatement  Program  Statement | No |  |
| while\_1.mim |  | Tests a simple while statement where a variable is incremented by one for each iteration | AdditionFactor  AdditionalExpression  AnotherStatement  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  Indent  IndentEnter  MoreStatements  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  Sequence  Statement  WhileStatement | Yes (2 levels) |  |
| while\_2.mim |  | Tests a nested while statement a while statement is nested inside the scope of a parent while statement | AdditionFactor  AdditionalExpression  AnotherStatement  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  Indent  IndentEnter  MoreStatements  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  Sequence  Statement  WhileStatement | Yes (3 levels) |  |
| factorial.mim |  | A generalized test that combines and tests many of the grammars. Involves defining a factorial function that computes the factorial iteratively using a while statement | AdditionFactor  AdditionalExpression  AnotherStatement  ArgList  AssignmentStatement  CallStatement  Comparison  DefStatement  ElementAccess  ElementExpression  Expression  ExpressionList  FunctionCallStatement  Indent  IndentEnter  MoreArgs  MoreExpressions  MoreStatements  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  PrintStatement  Program  ReturnStatement  Sequence  Statement  WhileStatement | Yes (3 levels) |  |
| multtab.mim |  | A complex test that tests multiple levels of indentation to verify that mutl-tab source files are correctly parsed. It involves almost all of the statements, nested if condition statements, print statements, and while statements | AdditionFactor  AdditionalExpression  AnotherStatement  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  ExpressionList  FunctionCallStatement  IfStatement  Indent  IndentCopy  IndentEnter  MoreExpressions  MoreStatements  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  PrintStatement  Program  Sequence  Statement  WhileStatement | Yes (4 levels) |  |
| comparison\_1.mim |  | Tests the < and > comparison operators with multiple if condition statements | AdditionFactor  AdditionalExpression  AnotherStatement  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  IfStatement  Indent  IndentCopy  IndentEnter  MoreStatements  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  Sequence  Statement | Yes (2 levels) |  |
| comparison\_2.mim |  | Tests the <= and >= comparison operators with multiple if condition statements | AdditionFactor  AdditionalExpression  AnotherStatement  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  IfStatement  Indent  IndentCopy  IndentEnter  MoreStatements  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  Sequence  Statement | Yes (2 levels) |  |
| comparison\_3.mim |  | Tests the == comparison operator using an if condition statement | AdditionFactor  AdditionalExpression  AnotherStatement  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  IfStatement  Indent  IndentCopy  IndentEnter  MoreStatements  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  Sequence  Statement | Yes (2 levels) |  |
| math\_1.mim |  | Tests the mathematical addition expression | AdditionFactor  AdditionalExpression  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  Statement | No |  |
| math\_2.mim |  | Tests a combination of the mathematical addition and subtraction expressions | AdditionFactor  AdditionalExpression  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  Statement | No |  |
| math\_3.mim |  | Tests the mathematical multiplication expression | AdditionFactor  AdditionalExpression  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  Statement | No |  |
| math\_4.mim |  | Tests the mathematical division expression | AdditionFactor  AdditionalExpression  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  Statement | No |  |
| math\_5.mim |  | Tests a combination of the mathematical multiplication and division expressions | AdditionFactor  AdditionalExpression  AssignmentStatement  CallStatement  Comparison  ElementAccess  ElementExpression  Expression  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  Statement | No |  |
| func\_1.mim |  | Tests a simple function definition statement, the function returns an integer | AdditionFactor  AdditionalExpression  AnotherStatement  ArgList  Comparison  DefStatement  ElementAccess  ElementExpression  Expression  Indent  IndentEnter  MoreStatements  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  ReturnStatement  Sequence  Statement | Yes (2 levels) |  |
| func\_2.mim |  | Tests a simple function definition statement with one argument, the function returns the argument provided | AdditionFactor  AdditionalExpression  AnotherStatement  ArgList  Comparison  DefStatement  ElementAccess  ElementExpression  Expression  Indent  IndentEnter  MoreArgs  MoreStatements  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  ReturnStatement  Sequence  Statement | Yes (2 levels) |  |
| func\_3.mim |  | Tests a function definition statement with more arguments, the function calculates the sum of the arguments and returns the value | AdditionFactor  AdditionalExpression  AnotherStatement  ArgList  AssignmentStatement  CallStatement  Comparison  DefStatement  ElementAccess  ElementExpression  Expression  Indent  IndentEnter  MoreArgs  MoreStatements  MultiplicationExpression  MultiplicationFactor  PrimitiveExpression  Program  ReturnStatement  Sequence  Statement | Yes (2 levels) |  |