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

Using the scanning and parsing methodology created in Assignment 1, our $.jjt$ 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 $LL(1)$, 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 $myNode$ class (called $removeScope()$) 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.

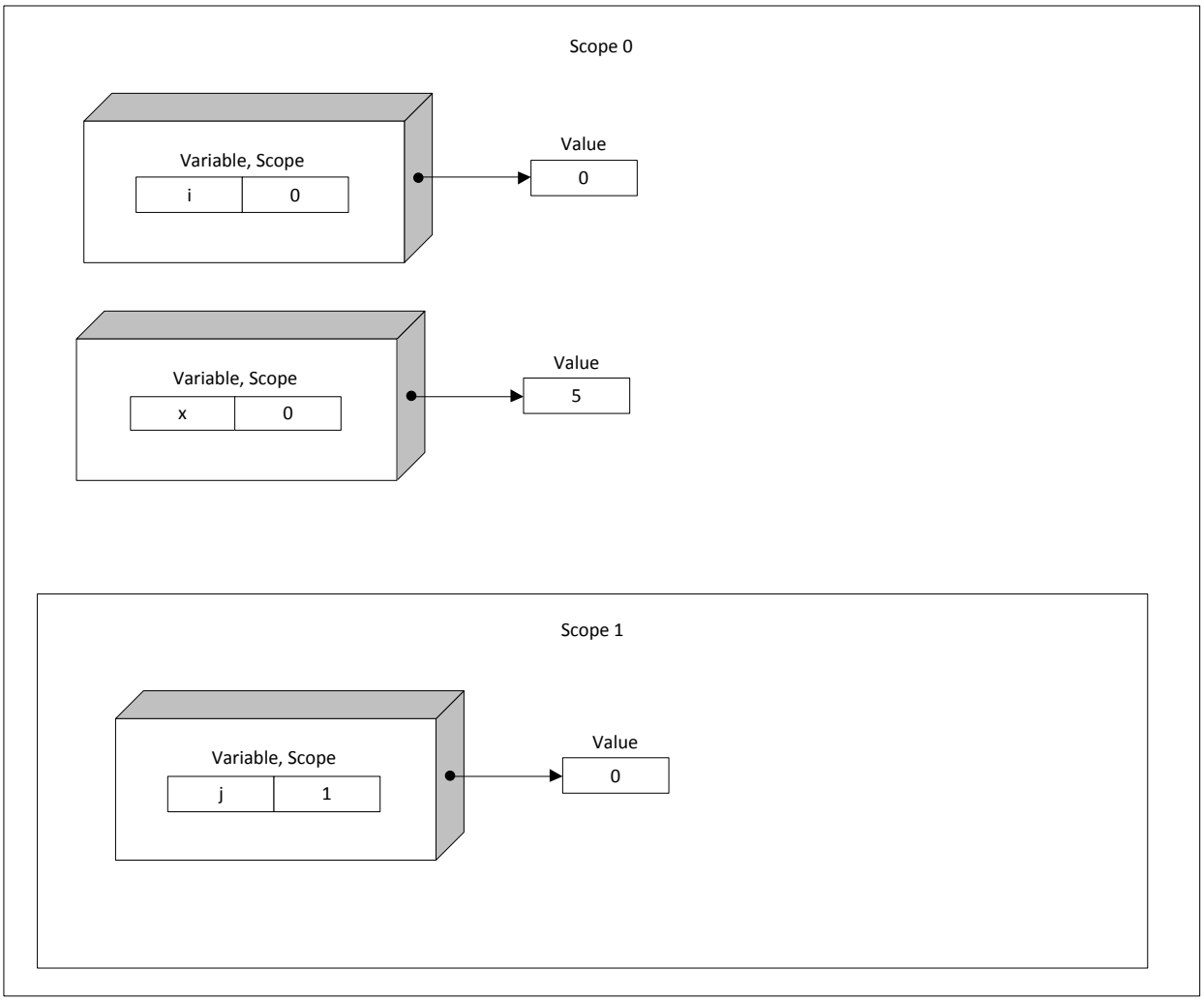

Figure 1: Multi-Key Hash Table

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

Figure 2: 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 $interpret()$ 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 $ArgsList$ 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 $ArgsList$. 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 $print()$ function. The interpretation and execution for this method simply uses a call to $System.out.println()$ 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	Includes Multiple Scopes
comparison_1.mim	Default	Tests the < and > comparison operators with multiple if condition statements	Yes (2 levels)
comparison_2.mim	Default	Tests the <= and >= comparison operators with multiple if condition statements	Yes (2 levels)
comparison_3.mim	Default	Tests the == comparison operator using an if condition statement	Yes (2 levels)
comparison_4.mim	Default	Tests printing the results of each combination of comparison operators for both true and false results.	No
comparison_5.mim	Default	Tests the == comparison operator using a variable in one if condition statement.	Yes (2 levels)
element_1.mim	Default	Tests printing a list of integer values.	No
element_2.mim	Default	Tests printing a list where the first element is an arithmetic expression and the remainder are integer values.	No
element_3.mim	Default	Tests printing a list of arithmetic expressions involving addition, multiplication, and division.	No
factorial.mim	Default	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	Yes (3 levels)
factorial_rec.mim	Default	A test that involves defining a factorial function which is then executed recursively to compute the factorial of a value.	Yes (3 levels)
func_1.mim	Default	Tests a simple function definition statement, the function returns an integer	Yes (2 levels)
func_2.mim	Default	Tests a simple function definition statement with one argument, the function returns the argument provided	Yes (2 levels)
func_3.mim	Default	Tests a function definition statement with more arguments, the function calculates the sum of the arguments and returns the value	Yes (2 levels)

func_4.mim	Default	Tests a function definition statement with an argument, where the argument provided to the method when it is called is a mathematical expression.	Yes (2 levels)
math_1.mim	Default	Tests the mathematical addition expression	No
math_2.mim	Default	Tests a combination of the mathematical addition and subtraction expressions	No
math_3.mim	Default	Tests the mathematical multiplication expression	No
math_4.mim	Default	Tests the mathematical division expression	No
math_5.mim	Default	Tests a combination of the mathematical multiplication and division expressions	No
math_6.mim	Default	Tests a complex combination of both multiplication and division in one statement.	No
math_7.mim	Default	Tests a complex combination of all mathematical operations (addition, subtractions, multiplication, and division) combined into one statement.	No
mult_1.mim	Default	Tests multiple mathematical expressions combined that is assignment for a variable	No
mult_2.mim	Default	Tests an if condition statement with assignment and mathematical operations	Yes (2 levels)
mult_3.mim	Default	Tests an if condition, while, and print statement as well as mathematical expressions assigned to a variable	Yes (2 levels)
mult_4.mim	Default	Tests nested if condition statements, an if condition statement is nested inside the scope of an if condition statement	Yes (3 levels)
multtab.mim	Default	A complex test that tests multiple levels of indentation to verify that multi-tab source files are correctly parsed. It involves almost all of the statements, nested if condition statements, print statements, and while statements	Yes (4 levels)
print_1.mim	Default	Tests print statement with an integer value	No
print_2.mim	Default	Tests print statement with an identifier that has been assigned an integer	No

print_3.mim	Default	Tests print statement with a mathematical expression using two integers	No
print_4.mim	Default	Tests print statement with a mathematical expression with addition and multiplication	No
variable_1.mim	Default	Tests printing a variable that has been initialized.	No
variable_2.mim	Default	Tests printing a variable that has been initialized, then the variable is added to itself.	No
variable_3.mim	Default	Tests printing a variable that has not been initialized, this results in a semantic error for an undefined variable.	No
while_1.mim	Default	Tests a simple while statement where a variable is incremented by one for each iteration	Yes (2 levels)
while_2.mim	Default	Tests a nested while statement a while statement is nested inside the scope of a parent while statement	Yes (3 levels)
RightFunction.mimp	Assignment 1	Tests executing a correct method which returns the square of the argument passed. Then tests printing the memory address of a square method.	Yes (2 levels)
RightFibonacci.mimp	Assignment 1	Tests executing a correct recursive Fibonacci method using binary tail recursion.	Yes (3 levels)
Rightif.mimp	Assignment 1	Tests executing a correct if statement where a one of a zero is printed.	Yes (2 levels)
RightList.mimp	Assignment 1	Tests printing a correct list with integer values and mathematical expressions.	No
RightWhile.mimp	Assignment 1	Tests a correct while loop where a variable used in the iteration is compared to a max iterations variable.	Yes (2 levels)
WrongFunction.mimp	Assignment 1	A test to ensure that the parser catches invalid syntax for the square method definition.	Yes (2 levels)
WrongIf.mimp	Assignment 1	A test to ensure that the parser catches invalid syntax for one if statement that is missing a colon.	Yes (2 levels)
WrongList.mimp	Assignment 1	A test to ensure that the parser catches invalid syntax for a list that is using parenthesis rather than square brackets.	No
WrongWhile.mimp	Assignment 1	A test to ensure that the parser catches invalid syntax for a while statement that is missing a colon.	No

ExampleFibonacci	Assignment 2	Tests executing a recursive Fibonacci method using binary tail recursion.	Yes (3 levels)
ExampleNestedFunction	Assignment 2	Tests executing a complex test involving a function with a nested function defined within the function. The function is then executed and the nested function is returned, which is then passed by reference as an argument to a method which executes the function passed by reference.	Yes (3 levels)
ExampleNestedLoops	Assignment 2	A tests which involves a method that contains a nested while statement. A while statement is nested inside the scope of a parent while statement, each while statement has a counter variable declared within their scope.	Yes (4 levels)
ExamplePrint	Assignment 2	A test which involves printing the results of multiple types of comparison operations involving variables as well as the return value that is returned by a method.	Yes (2 levels)
ExampleSquare	Assignment 2	Tests executing a correct method which returns the square of the argument passed. Then tests printing the memory address of a square method.	Yes (2 levels)