Programming Languages & Compilers - ENGR 3960
Assignment 1 – Minimp Language

Created by

Daniel Smullen

Joseph Heron

Jonathan Gillett

Contents

Scanning and Parsing

In order to create the lexer and parser component of the assignment, `JavaCC` using `JJTree` was required. This was used to perform automatic code generation, which implements a specified grammar.

Using `JJTree` & `JavaCC`

`Minimp` grammar was provided in Bachus-Naur form. In order for `JJTree` and `JavaCC` to implement the grammar programmatically, the grammar was translated from Bachus-Naur form into the Java-like grammar that `JJTree` supports.

Discussion

The original grammar provided contained errors. As a result, the grammars could not be translated. In order to determine a solution for this, LL1 parse tables were generated on given grammars by hand. These were used in conjunction with reference materials on python grammar to attempt to repair the errors and create a correct grammatical structure. This data was submitted for review by the instructor, and updated assignment files were received which contained additional errors. The same process was undertaken to find and eliminate the errors. Ultimately, *ExpressionList* on line 23 was found to still be invalid and was corrected to accept only integers, expressions (in parentheses), or identifiers.

JJTree Usage

`JJTree` accepts a context-free grammar as input, and generates a *.jj* file which is used within `JavaCC`. This process generates the node classes used to create abstract syntax trees. This process also generates abstract syntax tree classes for each grammar. The output is stored within the same folder as the *.jj* file, and processed by `JavaCC`.

Discussion

Several issues were encountered supporting indentation for `minimp`. The grammar was updated, and code was introduced to track the scope of nested statements based on indentation using a counter. Keeping track of the level of indentation allows the parser to determine whether a code block is one statement with a scope, or whether it should be broken up into several statements within several scopes.

JavaCC Usage

`JavaCC` was used to generate the tokenizer and parser automatically, given the *.jj* file input from `JJTree`.

Discussion

The updated grammar inside `AnotherStatement()` had some issues with choosing the wrong branch in the parsing tree when several layers of indentation deep in `if` statements. Parsing does not complete; a logical error occurs as an empty statement will eventually be returned. Why the parser returns an empty statement was unable to be determined.

Compilation & Testing

`Mimpc` is the main program, which accepts the source file input and parses it. If errors occur during this process, error codes are returned which provide additional diagnostic information.

Error Codes

The following error codes were implemented:

Error	Description	Code
VALID_SOURCE_FILE	Source file is valid, everything is OK.	0
INVALID_SOURCE_FILE	The source file specified is invalid, contains errors, or is corrupt.	-1
FILE_NOT_FOUND	The file specified is not found.	-2
INVALID_ARGUMENTS	The arguments specified to `mimpc` are invalid.	-3

Creating mimpc.java

All the generated classes were compiled using the Java compiler after automatic code generation. This created a class which tokenizes and parses an input file. A wrapper class was required to accept an input file for tokenizing and parsing, which uses the generated tokenizer, parser, and abstract syntax tree classes.

Discussion

After having completed debugging of all the automatically generated classes, compilation occurred without incident.

Testing

If the tokenizing and parsing is successful (file is valid), an abstract syntax tree is generated.

If tokenizing or parsing is unsuccessful, an error occurs which describes where the file is nonconforming to the specified tokens or grammar.

Test Cases

Test cases were created that covered all valid token types. Additional test cases were created that cover all 26 specified grammars. Created complex cases were also created that cover multiple grammars, including nested grammars.

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 Description	Minimp Grammars Covered	Nested Statements
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, prinrt statemetns, 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) |