Programming Languages & Compilers - ENGR-3960U

Assignment 3

Created by

Joseph Heron

Daniel Smullen

Jonathan Gillett

04/10/2013

Document Revision 1.0

Table of Contents

Design Methodology

This design phase requires that code generation functionality is added to the compiler. The compiler already supports lexical analysis, semantic analysis, abstract syntax tree generation, and Java interpretation. It is the artifacts of these processes that allow us to generate machine code. Analysis of the abstract syntax tree allows for specific nodes to execute the generation of MIPS assembly code whose operands are stored within registers. Evaluation of these commands and the allocation and correct usage of their operands is made possible by directed traversal of the abstract syntax tree.

In brief summary, Table 1 shows the commands and structures that were successfully implemented (and tested), and is located in Appendix A.

Temporary Register Usage

A stack was used to keep track of the temporary registers used. This has direct applicability for all areas of the compiled commands and structures, as most statements require the usage of one or more temporary registers for storage and manipulation of data. This is also important for scope information, as a stack structure (already previous implemented during the interpretation phase of the compiler) is used to maintain scope within structures inside the code to be compiled.

As registers are used for storing data, elements are added to the stack corresponding to each register. This is done in order to keep track of register usage, and ensure there is no overflow. As statements such as mathematical operations) use the registers temporarily and complete execution of their commands, the temporary registers are then popped from the stack, and freed for usage again.

Code examples where the stack usage is evident can be found inside `Node.java`.

Discussion and Future Improvements

If all registers are exhausted as the stack grows too large, the SPIM processor emulator is vulnerable to crashing or logical errors, because our code does not implement more complex safeguards for register usage. The Java stack in the compiler itself can grow larger than the amount of registers available, and registers may be overwritten. There is potential for future optimization of the compiler by storing values in the SPIM memory stack instead of temporary registers for operations that require more registers than are available, but this comes with many hardships and tremendous overhead implementation, as well as a performance trade-off (because the values would be in memory, rather than registers).

Printing Output

In order to print values, output is stored in a temporary output register. An assembly system call results in an interrupt which executes the actual print statement and forces the output to the console. Printing output works for all data types supported within *minimp* – that is to say, Boolean and integer values can be printed to the screen. This also means that the result of evaluated complex expressions can also be printed. Boolean values required special handling, as the data type is not implicitly supported for output to the screen. In instances where Boolean values for 1 were detected, output was replaced with the string `true`, and `false` for 0.

Test examples where output is printed for Boolean data types can be found in `comparison_4.mim`, and almost every test incorporates output of integer values to the screen.

The ability to evaluate print statements was one of the first instructions implemented in the compiler, as it was useful for debugging output.

Mathematical Operations

All integer arithmetic operations – addition, subtraction, multiplication, and integer division – were supported in our compiler. Variables can also be used in arithmetic operations, in combination with explicitly defined values. These can be chained together to generate complex expressions, and parentheses are supported for enforcement of order of operations in complex expressions.

Evaluation of the operations utilizes temporary registers to perform the calculations, which utilizes the temporary register stack. If the first operand in an operation is a constant value, it is stored in a register. Otherwise, the register that the value is already in is used to output the calculations. The expression is evaluated, and the handle to the register containing the output is returned. This can be used in conjunction with a print statement to output the actual values stored in the register, or the register's stored value can be manually validated.

Comparisons

Comparisons are conducted in a similar manner to mathematical operations. The evaluation of the comparison is conducted, and the handle to the register that stores the result of the comparison is returned. This register will contain either a 0 or 1, (Boolean false or true, respectively) depending on the resultant evaluation of the comparison.

Variables

Variables are stored in an assembly data member with a handle conformant to the format seen in Figure 1. This is done so that the variable can keep track of its own name (or handle), and its own scope. This is critical to the implementation of the scope abstraction within the *minimp* code, as the SPIM processor emulator (and any microprocessor for that matter) has no concept of scope implementation.

As variables are utilized in arithmetic operations, they are loaded into registers, and their temporary data member handle is returned.

Throughout the *minimp* code parsing, as new variable entities are used, they are added to an internal list in the compiler, so that they can all be output as data members in the assembly code upon final assembly code generation.

Some additional data members are reserved for various strings that are used throughout the remainder of the assembly code – `true` and `false` for Boolean values, for example. There is no risk of collision between these names and the reserved words due to the format of the data structure, as any members in the *minimp* code will always be appended with an underscore and scope value during compilation into assembly data members.

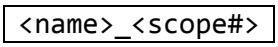

Figure 1 - Variable Data Structure

Boolean Values

Boolean values utilize 1s and 0s in place of human-readable `true` and `false` strings and are fully recognised as valid data types in the same way as integer values. The result of comparison operators is, by default, a Boolean data type. As such, complex comparisons can be conducted and the resultant values can be used in other complex chains of expressions. Boolean values can also be used within lists.

In assembly code generation and execution, Boolean values are treated in the same way as integer values, with limited exception – they have support for human-readable output through usage of predefined data members that store the appropriate strings for retrieval when necessary. This adds minimal overhead requirements to the data member definitions section of outputted assembly code to store these strings.

Element Lists

Element lists, otherwise known as vectors, represent lists of several values that are separated by commas, and are denoted by the usage of special square brackets. The semantic evaluation of these structures is primarily conducted during interpretation and abstract syntax tree generation, as these values are iterated across. Support for these lists works the same way as any complex expression, as complex expressions can be substituted for any single list member, and the structure overall remains the same. As such, the only additional facility required for lists is printing the output in a specially formatted way, and the data storage overhead requirement for several list members as opposed to a single data element.

These data types also have additional strings which are included in assembly code output in order for print statements to be executed. These print statements require lists to be human-readable, and as such support for the square braces and the comma separators is needed. Each of these special characters are included as data members in assembly code, and are output before, between, and after the elements in the data structure, respectively.

If Structures

These structures required special abstractions in assembly code generation to allow for their logic to function correctly. Assembly code generation for if statements, when they are encountered in *minimp* code, creates a label data member (seen in Figure 2) in assembly code which increments for each subsequent if statement. The else component of the structure is similarly labeled. The conditions for the structure are evaluated, and as a result the assembly code will cause the program counter to jump to the corresponding else statement label (stored in that statement's label data member) if the condition is false. At the end of the assembly code corresponding to the structure's if and else bodies, another label data member is created so that after the statement code body is completed execution, the else code body can be jumped past. As a result, the code is executed in the normal sequence that is to be expected for *minimp* code.

```
else_<instance count>
```
Figure 2 - If Structure Label Data Members' Structure

While Loops

Logical implementation of while loops in assembly code generation is similar to that of if structures. Again, the while loop begins with an incrementing label (seen in Figure 3). When the condition for the loop becomes false, and the loop is broken, the evaluation of the condition will result in the execution jumping to the end of the body of the while code, stored in the corresponding label. The very last statement inside the loop, however, jumps to the beginning of the loop structure, resulting in the looping behavior.

```
while_<instance count>
```
Figure 3 - While Loop Label Data Members' Structure

Testing

Testing was required throughout the implementation process of code generation, and regression tests were kept during the implementation of new functionality in order to ensure that the overarching structure of *minimp* was preserved as each new function was added.

These tests can be found inside the `test/` folder.

36 tests were created, in addition to the tests provided for official evaluation. These tests comprehensively execute every implemented statement (using every grammar listed) in several situations, providing confidence that they will execute for all required circumstances.

Automated black-box and regression testing was used to successfully uncover deficiencies in the grammar, and the evaluation tests that were provided, as well as provide validation for the correct functionality of the compiler.

Tests Provided for Evaluation

It must be noted that throughout all phases of the compiler implementation, tests which were provided for evaluation underwent necessary remediation in order to make them conformant to the *minimp* grammar specification. As a result, these tests pass, but in their original provided format they must fail as they contain invalid syntax.

Automated Test Framework

An automated test framework using a shell script executes the compiler on all of the tests provided, and automatically executes the generated assembly code files in the SPIM processor emulator. The resultant output is validated against exemplar output, and if differences are discovered, the test fails. If the resultant output is identical to the expected output, the test passes. All of the tests created, as well as the tests provided for evaluation (which were remediated in order to conform to the correct *minimp* grammar rules) pass. The visual output of the automated test framework can be seen in Figure 5 - All Test Results, located in Appendix B.

MIPS Assembly Code Output & SPIM Output Validation

The shell script used during testing is capable of automatically testing the assembly code output generation process, and executing the automatically generated code. The ability for us to validate this is given through usage of the `diff` command, using exemplars (noted with .out file extensions containing expected output) and the output from the SPIM processor emulator. The test framework automatically compares them.

The final output assembly code from all automated tests can be found in the `asm/` folder.

This means that the initial *minimp* code is parsed, assembly code is generated, the generated code is executed, and the results of execution are compared against expected outputs. In order for this to be facilitated, all tests required judicious usage of the print statement, so that output would be made visible. Direct introspection into the registers in the SPIM emulator was also conducted for manual validation, and the values present in the registers was checked against expected values, and translated into human-readable decimal values.

Appendix A

Table 1 - Implemented Commands and Structures

	Required	Implemented in *Minimp*
Sequence of Instructions	Yes	Yes
Integer Values	Yes	Yes
Boolean Values	No	Yes
Variables	Yes	Yes
Assignment Operator	Yes	Yes
Addition Operator	Yes	Yes
Subtraction Operator	No	Yes
Multiplication Operator	No	Yes
Division Operator	No	Yes
Equality Operator	Yes	Yes
Greater Than Operator	No	Yes
Less Than Operator	No	Yes
Greater Than Or Equal Operator	No	Yes
Less Than Or Equal Operator	No	Yes
Conditional Statements	Yes	Yes
Loop Statements	Yes	Yes
Function Definitions	No	No
Function Calls	No	No
Return Statements	No	No
Print Statement	No	Yes
Lists	No	Yes

Appendix B

All tests which contain supported statements/commands pass during execution of the automated test framework. Please refer to Table 1 - Implemented Commands and Structures in Appendix A for information about which commands are currently supported.

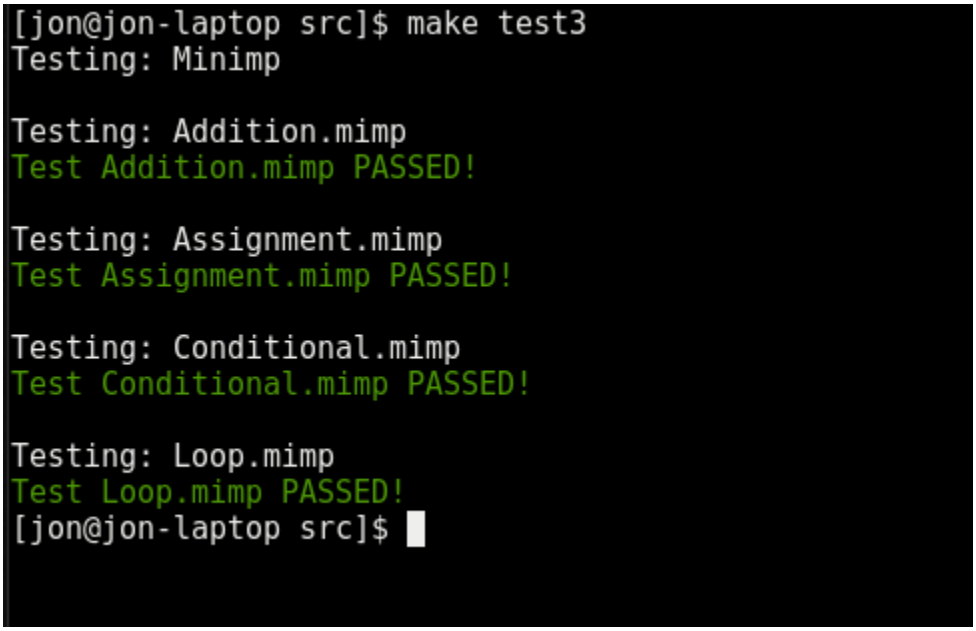

Figure 4 - Assignment 3 Evaluation Tests (Remediated) Results

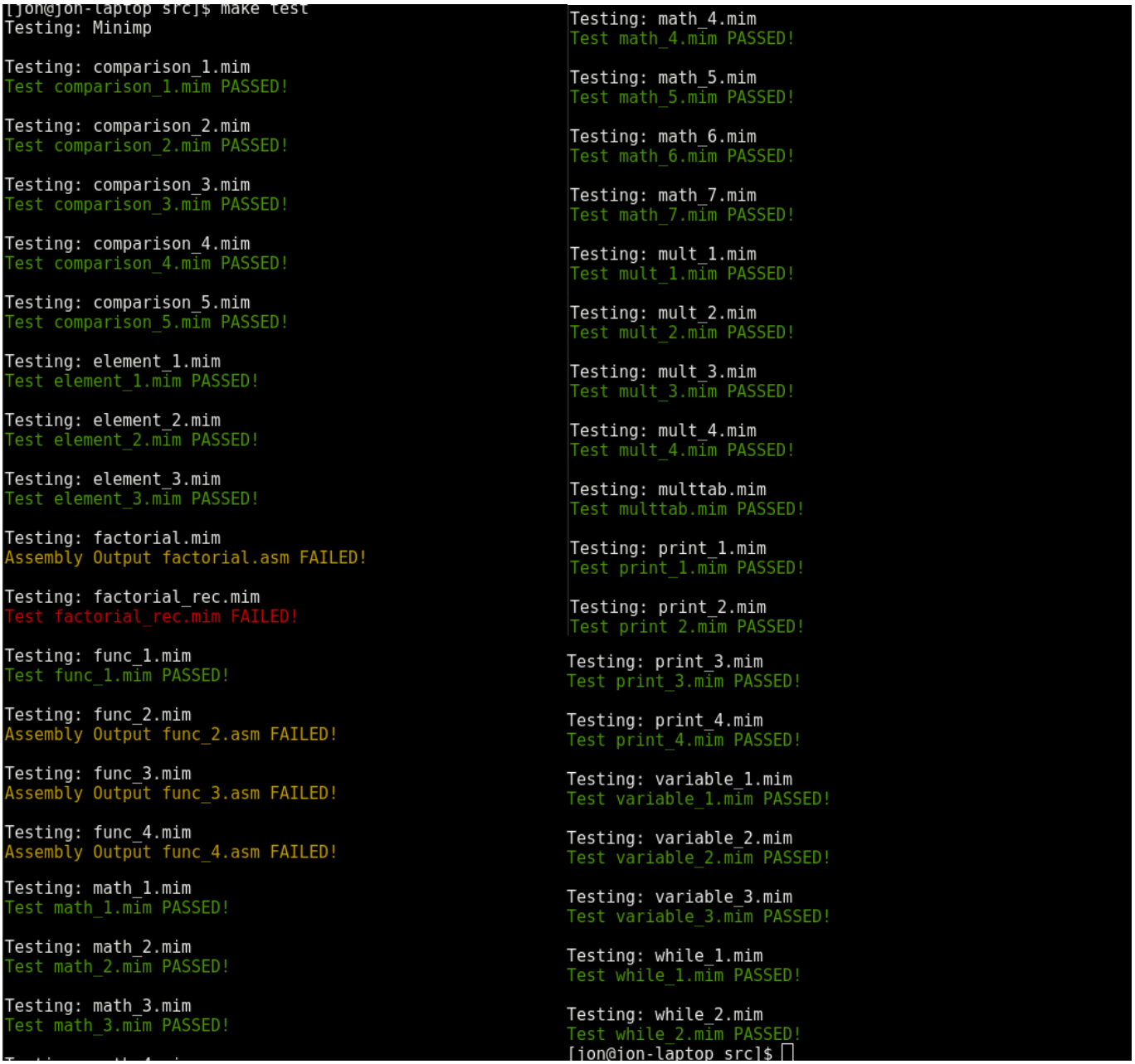

Figure 5 - All Test Results

Appendix C

This appendix contains a visual representation of the changes that were made to the provided evaluation tests in order to make them conformant to the *minimp* grammar, and therefore suitable for execution. As such, statements which were eliminated from the specification have been implemented again.

These results of automated testing using these fixed tests are seen in Figure 4 - Assignment 3 Evaluation Tests (Remediated) Results in Appendix B.

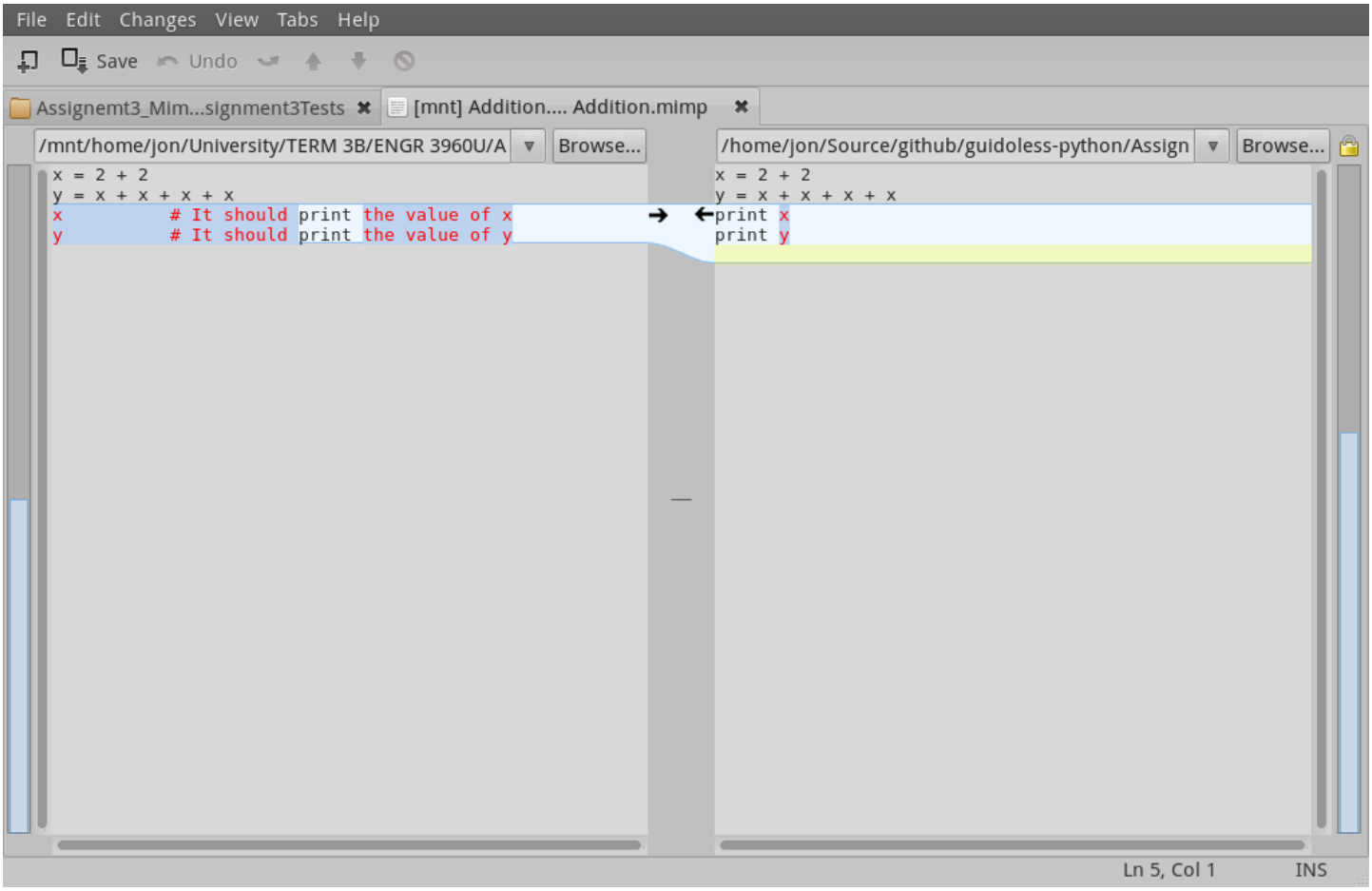

Figure 6 - Modified Addition.mimp Test Visualization

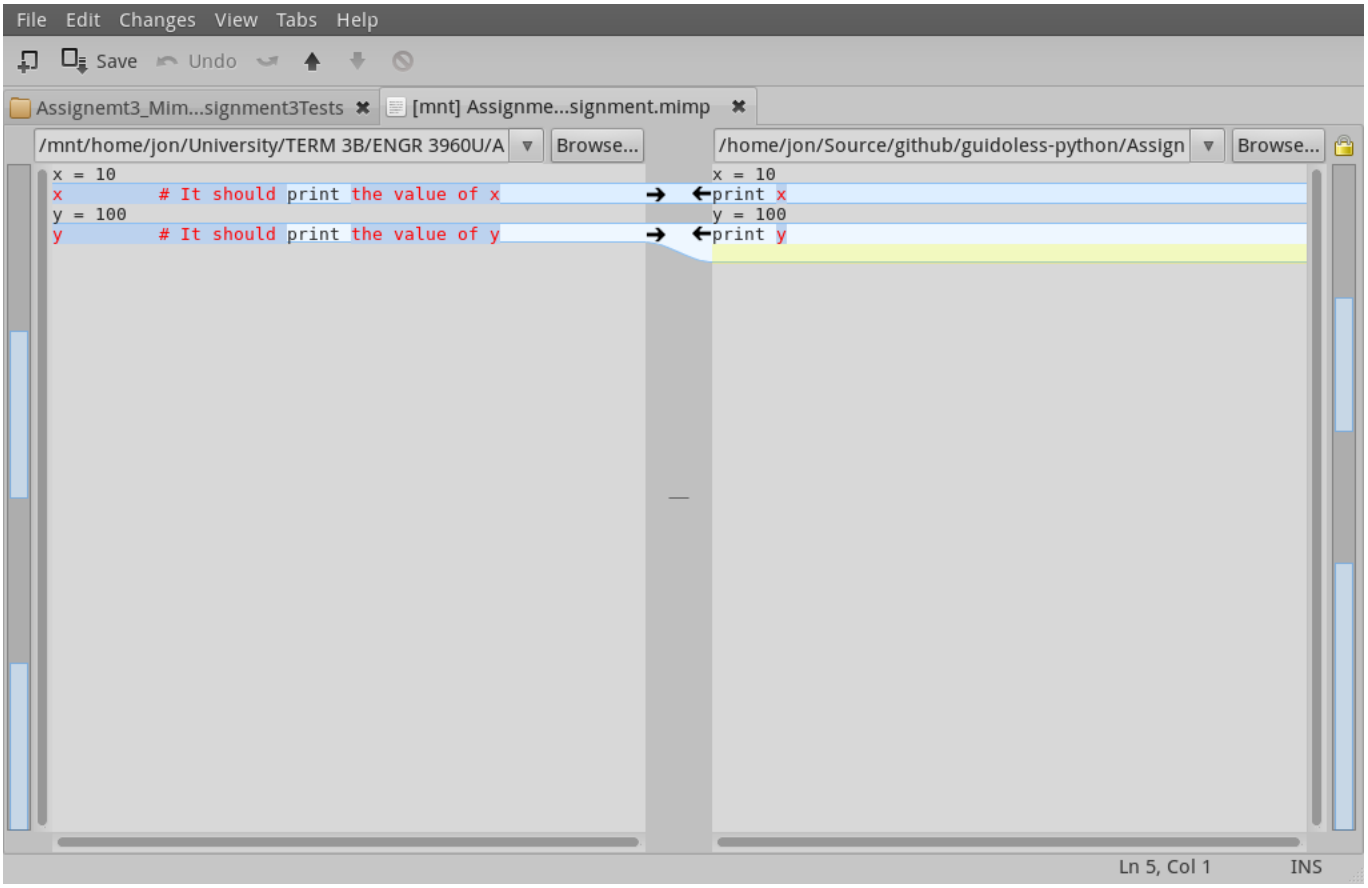

Figure 7 - Modified Assignment.mimp Test Visualization

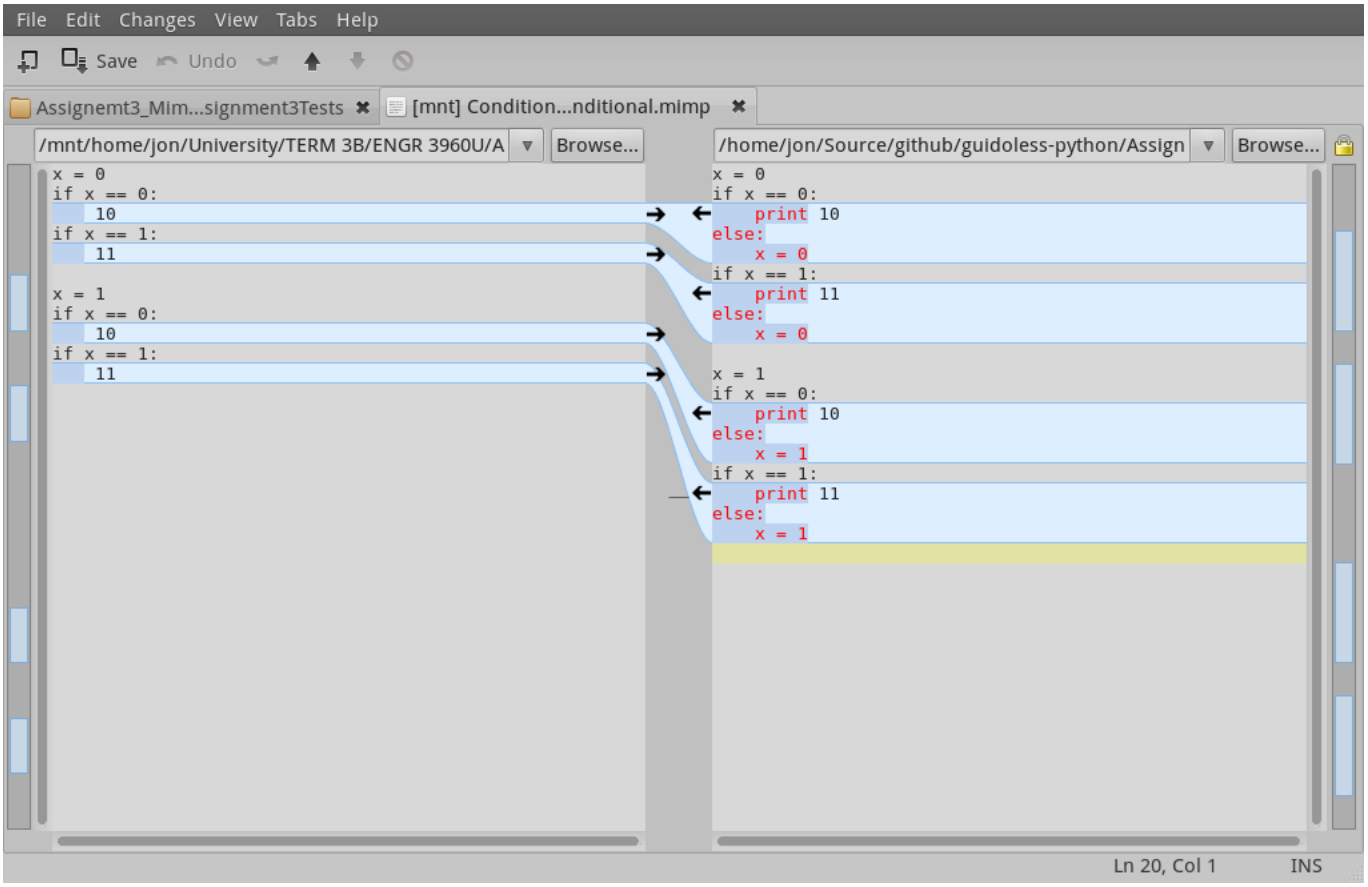

Figure 8 - Modified Conditional.mimp Test Visualization

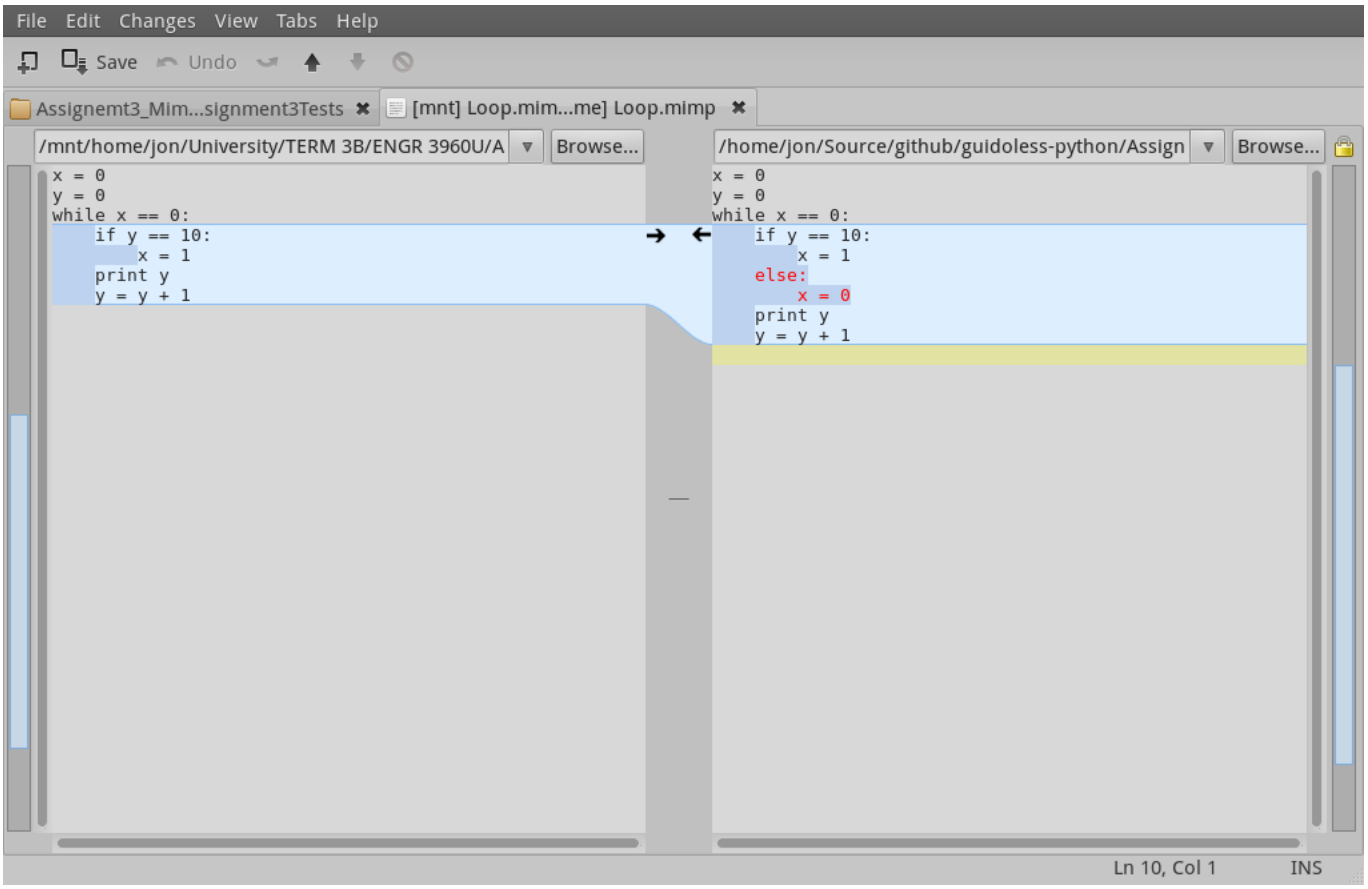

Figure 9 - Modified Loop.mimp Test Visualization