Compiler Theory & Practice Course Assignment

CPS 2000

247696(M)

Edward Fleri Soler

Table of Contents

1. Introduction

In this report we shall go through and understand the processes and steps involved in creating a programming compiler to execute basic high-level language instructions. The language we shall be dealing with is MiniLang, which is based on similar grammar to common high-level languages such as Java or C.

Compilers work in a number of stages, and are generally split into two parts being the front-end and the back-end. We shall first tackle the front end of the compiler; namely the lexer, and we shall then move on to the back side of the compiler, including the parser, semantic analyser and interpreter. We shall also explore the concept of a read-execute-print-loop (REPL) and observe the effectiveness of the generated compiler on different test cases.

2. Lexical Analysis

This compiler is designed to read in input from a text file, and compile the text as a whole, outputting a result and other data related to the compilation process. The first step in this complex process is the tokenisation of data. The text data within the given text file must first be formalised before it may undergo any further processing. This process is dealt with by the lexer, which is a program class forming part of the front end namespace.

2.1 Token Groups

The lexer implemented in this assignment is based upon an Extended Backus-Naur Form for the MiniLang language. The provided EBNF breaks down tokens into productions, which make up the syntax of the language. This EBNF was used to deduce the acceptable phrases, characters and operators which may be used within a MiniLang command. These tokens were listed down on paper, and token groups were devised, allowing for similar tokens to be handled in the same manner during the first steps of the compiler. The following is a list of token types devised and implemented in this system:

- TOK_Type
- TOK_BooleanLiteral
- TOK_Digit
- TOK_StringLiteral
- TOK_Identifier
- TOK_MultiplicativeOP
- TOK_AdditiveOP
- TOK_RelationalOP
- TOK_Assignment
- TOK_Keyword
- TOK_Punctuation
- TOK_EOF

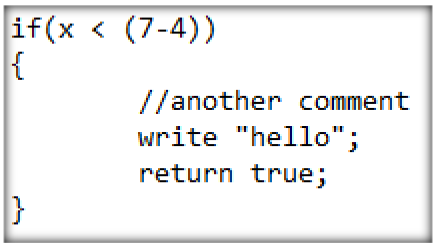
A **Token** class was implemented so as to create a blueprint for these tokens. Each individual token object holds a token type **TOK_TYPE**, a string **name** and an integer **value**. The name and value variables will later be used to store the contents of each token. Either one variable or the other will be used, and this is dependent upon the token type. Token getter and setter functions were also implemented so as to indirectly access or alter the token objects.

2.2 Lexer

The next step in the lexical analysis process involves the conversion from text into tokens. The **Lexer** class was implemented within the front end namespace, and held a number of functions responsible for handling the text data. A vector of type Token; **tokenVector**, was also declared within the class so as to hold all the program tokens. The **readFromFile** function was implemented to accept a string name for a file as input, access the file in storage and read all the text data from the file into a single string which is then returned.

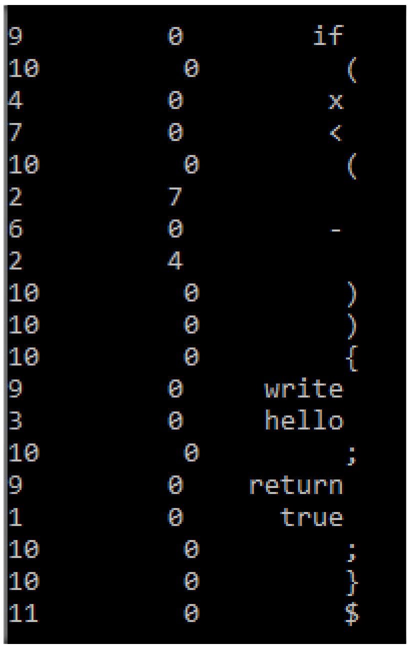
The string variable is then passed on to the **getTokens** function, which reads the text character by character, grouping characters together and assigning a specific token in their name. This function acts like a sieve, testing each character for a specific value, ignoring whitespaces or comments and grouping meaningful characters into word phrases. The respective token, together with a meaningful value is created in place of the text and stored within the vector. Once the whole string variable has been read, an End-Of-File token is inserted at the back of the vector, to mark the end of the token list.

The **getNextToken** and **lookNextToken** functions were also defined within this class. These functions will be made use of by the parser, by feeding the next token in the vector to the parser on each request, gradually forming a syntax tree on which the program is based. For testing purposes, a **tokenTester** function was also devised, to output all generated tokens onscreen. The following is an example input, followed by the tokenised output:



```
if(x < (7-4))
{
        //another comment
        write "hello";
        return true;
}
```

Figure 1. Lexer test input



```
9          0        if
10          0         (
4          0         x
7          0         <
10          0         (
2          7
6          0         -
2          4
10          0         )
10          0         )
10          0         {
9          0     write
3          0     hello
10          0         ;
9          0    return
1          0      true
10          0         ;
10          0         }
11          0         $
```

Figure 2. Lexer test output

In figure 2 above, we may observe the tokens generated by the input in figure 1. The first column depicts the token type of each token, ranging from 1 to 11, the second column depicts the integer value of each token, while the third column depicts the string value of each token. One may also observe that whitespaces and comments are completely ignored when tokenising the data.

3. Parsing

The next step of compilation involves taking all the tokens and parsing them into a meaningful syntax tree. The EBNF provided was used when creating the syntax tree by means of a top-down parsing method.

3.1 Nodes

A number of **Node** classes were defined within the back end namespace, so as to create nodes for each vertex in the syntax tree. An **ASTNode** acted as the parent class for all other node classes, which inherit a vector of **ASTNode** pointers and override two pure virtual functions.

The **ASTNode** pointer **children** was declared to hold pointers to child nodes for each element in the tree. One of the two functions was the **addChild** function, which would accept nodes of various types as argument, and would add these nodes to the **children** vector. Each node overrides the **addChild** function with one or many other **addChild** functions, catering for a specific node type which the given node could accept as a child. The other function which each node held is the **accept** function. This function is important as it allows for a visitor setup to be in place. This function will be heavily used throughout the remainder of the compilation steps.

3.2 Parser

The next phase involves actually generating the syntax tree. This solution is defined within the **Parser** class. This class, and its functions, is the first step in obtaining a data structure which could be interpreted. A parse function was defined for each individual node which could exist in the tree; example: **parseStatementNode**, **parseExpressionNode**, and so on. These functions will come into play once parsing has begun.

The **parsing** function is the first step in this process. When it is called, the lexer is invoked to process the program text and generate the vector of tokens. The parsing then begins by creating a program node, which shall act as the root of the syntax tree. Any node within the tree shall therefore be a descendent of this node. The control flow then enters a loop, which only terminated when the next token, consumed via the **getNextToken** lexer function, is an End-Of-File token, or an exception is thrown further down the parser. With each cycle of the loop, a statement node is defined, added as a child of the program node, and in-turn parsed by its own parsing function (**parseStatementNode**).

The parsing of a statement then begins by interpreting what the appropriate next production is by observing the next token in the vector. According to token type or value, the respective production from the EBNF provided is determined, with a node of the respective type (example **ASTVariableNode**) being created and added as a child of the statement node, and the parse function of the respective type (example **parseVariableNode)** being called to further parse the statement.

This cycle continues down a production, until a token which does not fit in with the current production is reached; in which case an exception is thrown, or the end of the production is reached. Cases which involve the possibility of repeated sequences, such as the production of expressions, were dealt with by means of a while loop, where the possible intermediary tokens are accepted, and the following token is tested again to see whether to cycle another time, or exit the statement. Certain characters within a production do not result in the generation of new tokens, but simple play a structural role in the production of a statement.

One such example is an open bracket '(' or a semi-colon ';' which are required for certain productions, but are simply read and discarded.

While working upon the parser for this assignment, one problem which came up was the fact that this EBNF contains one production which is not LL(1), but LL(2). A non-terminal **Factor** could produce either a **Literal** or a **Function Call**. However, a function call also starts with a literal terminal, resulting in ambiguity. Therefore, solely when dealing with factors, it was required to look two tokens ahead, so as to decide which production to follow.

Another similar problem was realised when handling digits within the program. Both the **Integer Literal** terminal and the **Real Literal** terminal started with an unknown number of digits. It was therefore required to read in tokens until the next non-digit character was read, so as to decide whether to create an integer node or a real node. This was performed by means of a while-loop, which looped through each token, storing the digits in variables. This loop cycled until the next read character was neither a digit nor a period '.'. If throughout this consumption a period was encountered, then a real node was created and populated with the respective value; otherwise an integer node was created.

If at any point throughout the parsing process, an invalid token is read, then an exception is thrown, and parsing stops. Otherwise, if no exception is thrown, the result is a complete syntax tree, which may be further tested for semantics and ultimately executed. The root node is then returned, to be passed on to further functions.

Examples and test cases for the parsing section of the compiler shall be discussed in the following section.

4. Printing of the Abstract Syntax Tree

Following the generation of the abstract syntax tree, a print function shall be implemented so as to observe the structure of the syntax tree. The use of this feature was ideal when coding, testing and debugging the remainder of the program.

4.1 Visitor Class

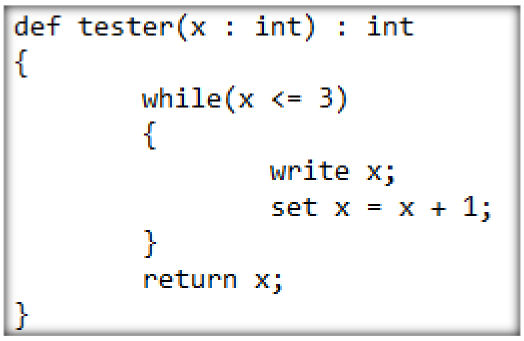
A **Visitor** class was implemented within the back end namespace of the program. This class consisted solely of pure virtual functions, which shall later be inherited and overriden by a child class of the **Visitor**. A function definition was defined for each node type, so as to accept specific node pointers as arguments. The **accept** function (discussed above) within each node class simply calls the visitor function for the respective node, and allows for the contents of each node to be accessed. This in turn will allow for data to be gathered and output on screen.

4.2 Print Info Visitor

The **PrintInfoVisitor** class is derived from the **Visitor** class defined above, and shall be used to access each node's values/children, and output them onscreen in XML format. Each pure virtual function within the **Visitor** class is overridden in this class, and an integer **indent** is also defined to indent each tag within the XML a certain number of tabs for ease of viewing.

The visitor class for each non-terminal node outputs the node name on screen within an opening tag, invokes the **accept** function for each of its child nodes (resulting in these nodes being visited), followed by the closing off of the node with a closing tag. Each terminal node, on the other hand, simply outputs opening and closing tags around the integer, Boolean or string value which they held.

By calling the visitor function of the root node through a **PrintInfoVisitor** object, the whole of the syntax tree is visited in a depth-first search fashion. The following is a sample of input text and the respective XML tags generated following the parsing and visiting of each node:



```
def tester(x : int) : int
{
        while(x <= 3)
        {
                write x;
                set x = x + 1;
        }
        return x;
}
```

Figure 3. XML printing test input

```
<Program>
  <Statement>
    <Function Declaration>
      <Identifier>tester</Identifier>
      <Formal Parameters>
        <Formal Parameter>
          <Identifier>x</Identifier>
          <Type>int</Type>
        </Formal Parameter>
      </Formal Parameters>
```

```
<Type>int</Type>
<Block>
  <Statement>
    <While Loop>
      <Expression>
        <Simple Expression>
          <Term>
            <Factor>
              <Identifier>x</Identifier>
            </Factor>
          </Term>
        </Simple Expression>
        <Relational Op><=</Relational Op>
        <Simple Expression>
          <Term>
            <Factor>
              <Literal>
                <Integer Literal>3</Integer Literal>
              </Literal>
            </Factor>
          </Term>
        </Simple Expression>
      </Expression>
      <Block>
        <Statement>
          <Write Statement>
            <Expression>
              <Simple Expression>
                <Term>
                  <Factor>
                    <Identifier>x</Identifier>
                  </Factor>
                </Term>
              </Simple Expression>
            </Expression>
          </Write Statement>
        </Statement>
        <Statement>
          <Assignment>
            <Identifier>x</Identifier>
            <Expression>
              <Simple Expression>
                <Term>
                  <Factor>
                    <Identifier>x</Identifier>
                  </Factor>
                </Term>
                <Additive Op>+</Additive Op>
                <Term>
                  <Factor>
```

```
                          <Literal>
                            <Integer Literal>1</Integer Literal>
                          </Literal>
                        </Factor>
                      </Term>
                    </Simple Expression>
                  </Expression>
                </Assignment>
              </Statement>
            </Block>
          </While Loop>
        </Statement>
        <Statement>
          <Return>
            <Expression>
              <Simple Expression>
                <Term>
                  <Factor>
                    <Identifier>x</Identifier>
                  </Factor>
                </Term>
              </Simple Expression>
            </Expression>
          </Return>
        </Statement>
      </Block>
    </Function Declaration>
  </Statement>
</Program>
```

Figure 4. XML printing test output

Observing figures 3 and 4 above, it is apparent that both the parsing and the XML printing sections of the program successfully execute. Although the XML output might look complicated at first, on closer observation, the pattern following through each production is very clear and understandable. It is easy to trace through the origins of a terminal node, as well as to observe sibling nodes within the syntax tree. Multiple tests have been run on sample codes of different length and complexity for the above section of the code, and each test has concluded successfully.

Following the testing of the above code, it was also made clear that binary expressions such as multiplication and addition required no special treatment, as the EBNF implemented enforced precedence rules through the production orders.

5. Semantic Analysis

The next stage in the compilation process is semantic analysis. This step ensures that the processed code is semantically correct and does not violate any access or type rules. Scopes are implemented in this step, where a new scope is generated every time a new block is entered, ensuring that no two variables with the same name may exist within the same scope, and retaining the values of variables and functions for interpretation at a later stage.

5.1 Symbol Table

The symbol table is the basis for all semantic and scope testing. It acts as a store for all variables and function definitions within a current scope. The symbol table is built off of a stack **scopeStack** with a mapping on each level. Each mapping represents a scope, and the stack may hold a number of scopes, nested within each other. Each scope is made up of a mapping from a string, being the variable or function name, to a symbol table entry **stentry**.

A symbol table entry holds data regarding the respective variable or function. Therefore, symbol table entries are built off of a struct, which holds a Boolean value **func** which defines whether the current entry is a function definition or not. This struct also holds a **funcval** structure **fv** and a **varval** structure **vv**. A **varval** is itself a struct, which holds type **t** which defines the type of the current variable, and 4 variables which can hold data of their respective type (string, integer, float or Boolean). A **funcval** is made up of a **varval** which stores the return type of the function, a vector of variables to store the parameters, a vector of strings to store the parameter names, and an **ASTBlock** pointer, to store a pointer to the block node of a function. The type definition of a variable is **ptype**, which could either be real, string, integer or Boolean.

Therefore, any single entry can be distinguished in type, tested for data type, and have its value or type accessed or altered. This is made possible by a number of defined class functions. The **scopeStack** may also have a new level added or removed, each time a block is left or entered respectively. Further class functions also allow for **stentry** type variables to be added to a scope, tested for existence, type or parameter type, and retrieved.

This symbol table class, together with its functions, shall therefore form the foundation for semantic testing and storage of data.

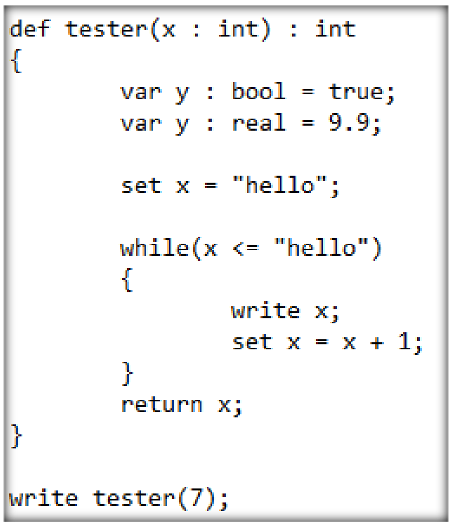
5.2 Semantic Analysis Visitor

The **SemanticAnalyserVisitor** class is derived from the **Visitor** class, and makes use of the visitor design pattern to traverse the syntax tree in a depth first fashion, and test for semantic soundness. This class overrides the visitor functions defined it its parent class, with each node type once again holding its own visitor function. A symbol table variable **st** is also defined so as to store and test data as explained below. Further **ptype** and **varval** variables are defined so as to hold the data type and data value of recently visited nodes.

In a similar fashion to the XML printer explained previously, the semantic analyser begins testing from the root program node, visiting child nodes from left to right, depth-first. The main test being carried out is that of type checking. On each occurrence of a literal, expression, declaration or assignment, the types of the involved constituents are tested for agreement. Example, in the case of a variable declaration, a variable declared as Boolean, may only be assigned a Boolean value. Therefore, at variable declaration, variable assignment, or the involvement of the said variable in any expression, the variable type and possible value must be fetched from the symbol table and tested for compliance. Similar tests take place with operators, ensuring that the operands on each side of an operator match with the type of operator.

Type checking must also takes place when assigning actual parameters to formal parameters during a function call. At this point in the generation of the compiler, a decision was taken to only allow operations or assignments to take place between operands of the exact same type. Therefore, typecasting from real to integer or vice versa is not implemented, meaning that these two data types may not be involved within the same expression or its constituents.

Other tests conducted during this traversal include the testing of existence of variables. When defining a variable, a test must be conducted to ensure that a variable with the same identifier does not exists within the current scope of the symbol table. On assignment, a test must also be made to check whether the current variable has been previously defined. Also, when dividing two numbers, a test must be made on the denominator to ensure that it is not 0. If any errors or inequalities are encountered during this traversal, an error is output onscreen, defining the type of error encountered, and an exception is thrown. On the following page is a test case which we shall use to observe the efficiency of the semantic analyser:



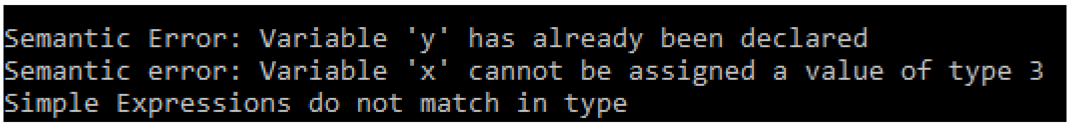
```
def tester(x : int) : int
{
        var y : bool = true;
        var y : real = 9.9;

        set x = "hello";

        while(x <= "hello")
        {
                write x;
                set x = x + 1;
        }
        return x;
}

write tester(7);
```

Figure 5. Semantic Analysis test input



```
Semantic Error: Variable 'y' has already been declared
Semantic error: Variable 'x' cannot be assigned a value of type 3
Simple Expressions do not match in type
```

Figure 6. Semantic Analysis test output

Observing figure 5, one may note a number of semantic inconsistencies within the sample code. First of all, the variable **y** has been declared twice, resulting in an error. Being a hard-type language, this sort of inconsistency should raise an error. The next problem is that the variable **x**, which is a formal parameter of type integer, has been assigned the string value "hello". This should raise a type error due to the inconsistency in variable type and expression type. Finally, within the test of the while-loop, the integer variable **x** is once again being compared to a string by means of a relational operator. Relational operators may only compare integer or real values, and therefore this is an illegal expression.

Now, observing figures 6 above, we may note that all three of these semantic errors have been caught by the compiler. The first error report matches with the first discussed error regarding the repeated declaration of **y**. The second error report is with respect to the assignment of a string value to an integer variable. Finally, the last error reported is a response to the relational comparison of an integer variable with a string.

Seeing that all semantic errors within the test case were caught, it may be concluded that the semantic analyses for this compiler is fully functional in testing for semantic completeness.

6. Code Interpretation

The next and final stage in the compilation process is the interpretation of the code. This step involves traversing the syntax tree, computing all operations such as expressions or assignments, taking decisions on loops or if-statements and handling write statements, outputting results on screen.

The **Interpreter** class, within the back end namespace, is derived from the **SemanticAnalyserVisitor** class, and holds visitor functions for each node type. The visitor functions will, once again, be used to access data in each node of the tree, which shall allow for decisions to be taken and the correct path through code to be selected. Most interpreter visitor functions are similar to the visitor functions within the semantic analyser, however some functions have added features to handle interpretation. The visitor function for an if-statement begins by evaluating the Boolean result of its decision expression, and based on this result, decides which path to follow. If this expression results in a 'true' value, then the first block is entered. Otherwise, if an else block exists, then that block is entered.

The same principle is applied to while-loops, with the loop block being executed each time the expression evaluates to true, and only stops cycling once the expression evaluates to false. The symbol table and scope is implemented in the exact same manner as the semantic analyser, with changes in value to variables being recorded.

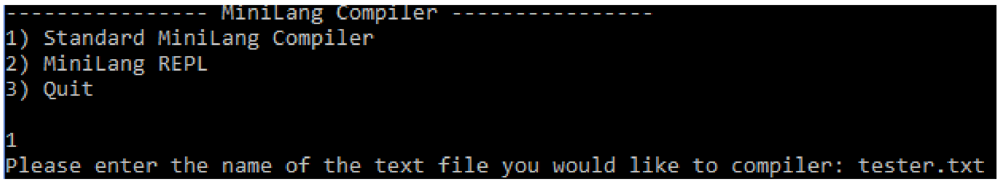
Finally, when visiting a write node, the expression on the right hand side of the statement is first evaluated, and then output on screen. These statements therefore account for all the data viewed on screen following the compilation of a program. In the following section, we shall discuss how the system as a whole was put together, and we shall observe compiler test cases.

7. Bringing it all together

Above, we have discussed the various vital components that make up the MiniLang compiler. We shall now see how these components were brought together to create a fully functional compiler. We shall then observe test programs which were used for debugging and improvements on the program.

7.1 MiniLang Compiler Main

The **CompilerTester** class holds the main function for execution in this solution. A basic command line interface main menu was set up to allow the user to select between ordinary compilation or REPL compilation. Ordinary compilation begins with a prompt for the file name to be entered. This filename is then passed onto the **parse** function discussed above, which tokenises the text and generates an abstract syntax tree. The root node of this syntax tree is then returned and passed on to the **printInfoVisitor** class functions so as to print the XML tags for the tree. The same root node is then passed onto the **semanticAnalyserVisitor** class functions for semantic analysis, before finally being interpreted by means of the **Interpreter** visitor functions, resulting in the complete compilation of the program.



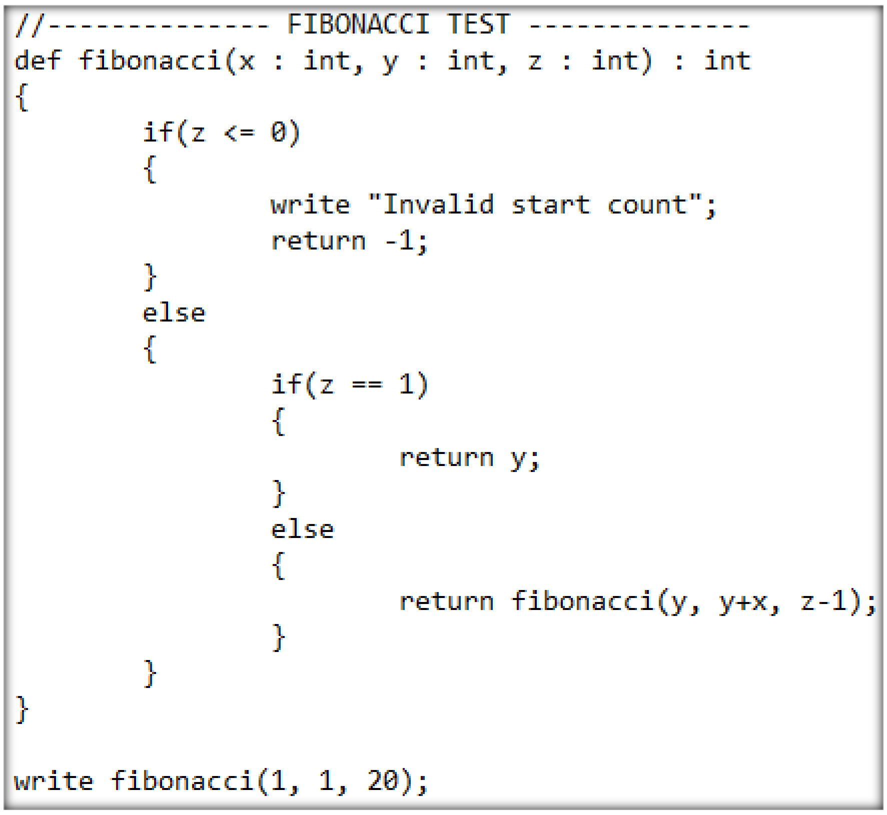
```
---------------- MiniLang Compiler ----------------
1) Standard MiniLang Compiler
2) MiniLang REPL
3) Quit

1
Please enter the name of the text file you would like to compiler: tester.txt
```

Figure 7. MiniLang Compiler Main Menu

7.2 Compiler Testing

We shall now look at two sample programs fed to the compiler for testing purposes, both during the generation of the program and the debugging and improvement phase:



```
//-------------- FIBONACCI TEST --------------
def fibonacci(x : int, y : int, z : int) : int
{
        if(z <= 0)
        {
                write "Invalid start count";
                return -1;
        }
        else
        {
                if(z == 1)
                {
                        return y;
                }
                else
                {
                        return fibonacci(y, y+x, z-1);
                }
        }
}

write fibonacci(1, 1, 20);
```

Figure 8. Fibonacci test input program

The above code defines a function called **fibonacci** which is designed to compute the nth Fibonacci value. This program was considered ideal for testing as it involves a variety of different statements and concepts, such as tail recursion. The function is defined to accept three parameters, the first two being the starting values for the Fibonacci sequence, while the last dictating the nth number to compute. The compiler program was executed, and the file name of this file was passed as input. The following is the complete program output on execution:

```
---------------- Syntax Errors ----------------

---------------- Program Abstract Syntax Tree in XML format ----------------


<Program>
  <Statement>
    <Function Declaration>
      <Identifier>fibonacci</Identifier>
      <Formal Parameters>
        <Formal Parameter>
          <Identifier>x</Identifier>
          <Type>int</Type>
        </Formal Parameter>
        <Formal Parameter>
          <Identifier>y</Identifier>
          <Type>int</Type>
        </Formal Parameter>
        <Formal Parameter>
          <Identifier>z</Identifier>
          <Type>int</Type>
        </Formal Parameter>
      </Formal Parameters>
      <Type>int</Type>
      <Block>
        <Statement>
          <If Statement>
            <Expression>
              <Simple Expression>
                <Term>
                  <Factor>
                    <Identifier>z</Identifier>
                  </Factor>
                </Term>
              </Simple Expression>
              <Relational Op><=</Relational Op>
              <Simple Expression>
                <Term>
                  <Factor>
                    <Literal>
                      <Integer Literal>0</Integer Literal>
                    </Literal>
                  </Factor>
                </Term>
              </Simple Expression>
            </Expression>
            <Block>
              <Statement>
                <Write Statement>
                  <Expression>
                    <Simple Expression>
                      <Term>
                        <Factor>
                          <Literal>
                            <String Literal>Invalid start count</String Literal>
                          </Literal>
                        </Factor>
                      </Term>
                    </Simple Expression>
                  </Expression>
                </Write Statement>
              </Statement>
              <Statement>
                <Return>
                  <Expression>
                    <Simple Expression>
                      <Term>
                        <Factor>
                          <Unary Op [-]>
                            <Expression>
                              <Simple Expression>
                                <Term>
                                  <Factor>
                                    <Literal>
                                      <Integer Literal>1</Integer Literal>
                                    </Literal>
                                  </Factor>
                                </Term>
                              </Simple Expression>
                            </Expression>
                          </Unary Op>
                        </Factor>
                      </Term>
                    </Simple Expression>
                  </Expression>
                </Return>
              </Statement>
            </Block>
            <Block>
              <Statement>
```

```
<If Statement>
  <Expression>
    <Simple Expression>
      <Term>
        <Factor>
          <Identifier>z</Identifier>
        </Factor>
      </Term>
    </Simple Expression>
    <Relational Op>==</Relational Op>
    <Simple Expression>
      <Term>
        <Factor>
          <Literal>
            <Integer Literal>1</Integer Literal>
          </Literal>
        </Factor>
      </Term>
    </Simple Expression>
  </Expression>
  <Block>
    <Statement>
      <Return>
        <Expression>
          <Simple Expression>
            <Term>
              <Factor>
                <Identifier>y</Identifier>
              </Factor>
            </Term>
          </Simple Expression>
        </Expression>
      </Return>
    </Statement>
  </Block>
  <Block>
    <Statement>
      <Return>
        <Expression>
          <Simple Expression>
            <Term>
              <Factor>
                <Function Call>
                  <Identifier>fibonacci</Identifier>
                  <Actual Parameters>
                    <Expression>
                      <Simple Expression>
                        <Term>
                          <Factor>
                            <Identifier>y</Identifier>
```

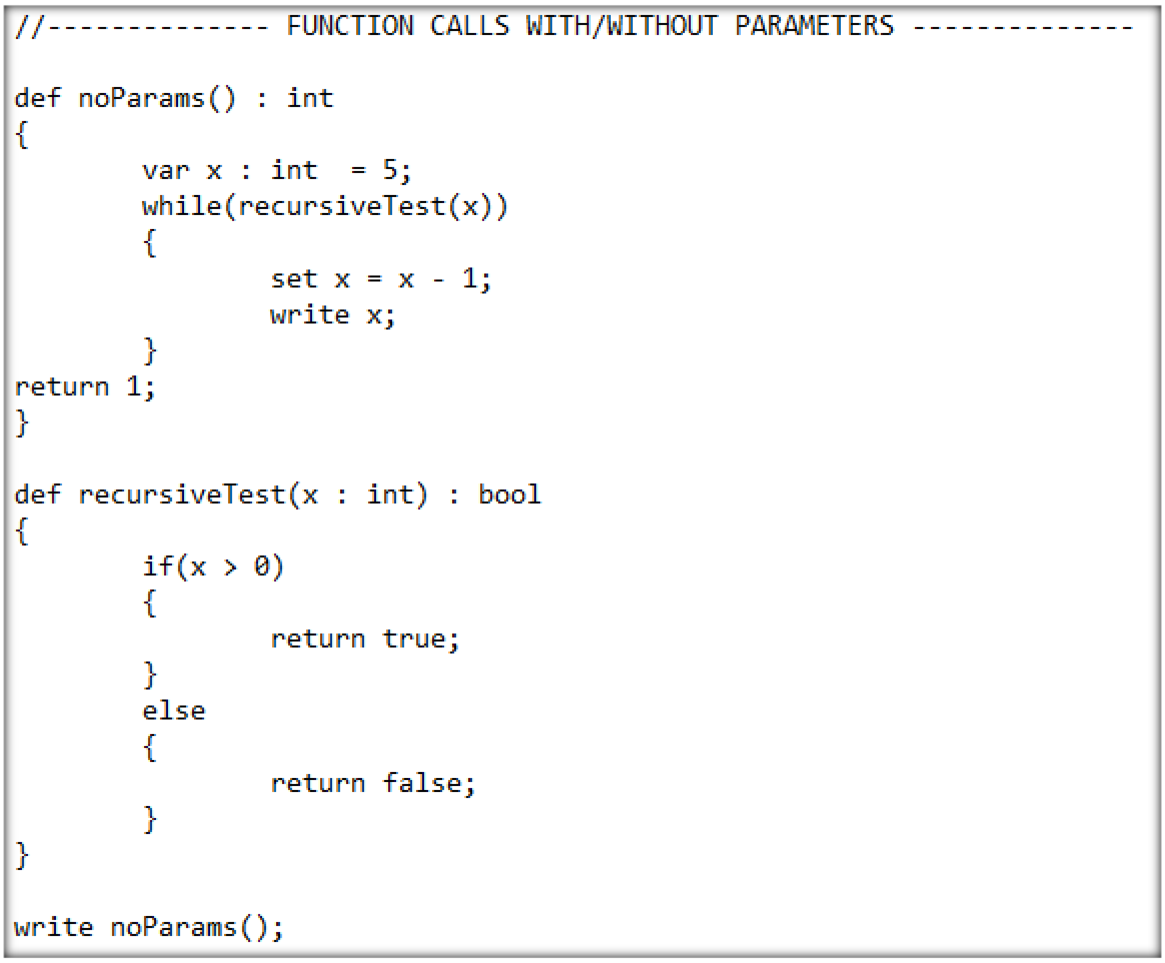
```
                                  </Factor>
                                </Term>
                              </Simple Expression>
                            </Expression>
                            <Expression>
                              <Simple Expression>
                                <Term>
                                  <Factor>
                                    <Identifier>y</Identifier>
                                  </Factor>
                                </Term>
                                <Additive Op>+</Additive Op>
                                <Term>
                                  <Factor>
                                    <Identifier>x</Identifier>
                                  </Factor>
                                </Term>
                              </Simple Expression>
                            </Expression>
                            <Expression>
                              <Simple Expression>
                                <Term>
                                  <Factor>
                                    <Identifier>z</Identifier>
                                  </Factor>
                                </Term>
                                <Additive Op>-</Additive Op>
                                <Term>
                                  <Factor>
                                    <Literal>
                                      <Integer Literal>1</Integer Literal>
                                    </Literal>
                                  </Factor>
                                </Term>
                              </Simple Expression>
                            </Expression>
                          </Actual Parameters>
                        </Function Call>
                      </Factor>
                    </Term>
                  </Simple Expression>
                </Expression>
              </Return>
            </Statement>
          </Block>
        </If Statement>
      </Statement>
    </Block>
  </If Statement>
</Statement>
```

```
    </Block>
  </Function Declaration>
</Statement>
<Statement>
  <Write Statement>
    <Expression>
      <Simple Expression>
        <Term>
          <Factor>
            <Function Call>
              <Identifier>fibonacci</Identifier>
              <Actual Parameters>
                <Expression>
                  <Simple Expression>
                    <Term>
                      <Factor>
                        <Literal>
                          <Integer Literal>1</Integer Literal>
                        </Literal>
                      </Factor>
                    </Term>
                  </Simple Expression>
                </Expression>
                <Expression>
                  <Simple Expression>
                    <Term>
                      <Factor>
                        <Literal>
                          <Integer Literal>1</Integer Literal>
                        </Literal>
                      </Factor>
                    </Term>
                  </Simple Expression>
                </Expression>
                <Expression>
                  <Simple Expression>
                    <Term>
                      <Factor>
                        <Literal>
                          <Integer Literal>20</Integer Literal>
                        </Literal>
                      </Factor>
                    </Term>
                  </Simple Expression>
                </Expression>
              </Actual Parameters>
            </Function Call>
          </Factor>
        </Term>
      </Simple Expression>
      </Expression>
    </Write Statement>
  </Statement>
</Program>
---------------- Output ----------------

10946
```

As may be observed from the above output, the program was found to have no syntax or semantic errors, resulted in the successful generation of an XML output, and outputted the correct value of **10946**. This reinforces the fact that this compiler can handle all forms of valid input correctly, and correctly handles scope levels and concepts such as recursion.

We shall now perform one more test, so as to ensure that the compiler can properly handle function declarations and calls with and without parameters:



```
//-------------- FUNCTION CALLS WITH/WITHOUT PARAMETERS --------------

def noParams() : int
{
        var x : int  = 5;
        while(recursiveTest(x))
        {
                set x = x - 1;
                write x;
        }
return 1;
}

def recursiveTest(x : int) : bool
{
        if(x > 0)
        {
                return true;
        }
        else
        {
                return false;
        }
}

write noParams();
```

Figure 9. Input program to correctly test function declarations/calls

The above program involves two function definitions; **noParams** which takes no arguments and returns an integer, and **recursiveTest** which takes one integer and returns a Boolean. The above code shall serve to test the recursive handling of this compiler, as well as function declarations/calls:

```
---------------- Syntax Errors ----------------

---------------- Program Abstract Syntax Tree in XML format ----------------

<Program>
  <Statement>
    <Function Declaration>
      <Identifier>noParams</Identifier>
      <Type>int</Type>
      <Block>
        <Statement>
          <Variable Declaration>
            <Identifier>x</Identifier>
            <Type>int</Type>
            <Expression>
              <Simple Expression>
                <Term>
                  <Factor>
                    <Literal>
                      <Integer Literal>5</Integer Literal>
                    </Literal>
                  </Factor>
                </Term>
              </Simple Expression>
            </Expression>
          </Variable Declaration>
        </Statement>
        <Statement>
          <While Loop>
            <Expression>
              <Simple Expression>
                <Term>
                  <Factor>
                    <Function Call>
                      <Identifier>recursiveTest</Identifier>
                      <Actual Parameters>
                        <Expression>
                          <Simple Expression>
                            <Term>
                              <Factor>
                                <Identifier>x</Identifier>
                              </Factor>
                            </Term>
                          </Simple Expression>
                        </Expression>
                      </Actual Parameters>
                    </Function Call>
                  </Factor>
                </Term>
              </Simple Expression>
            </Expression>
            <Block>
              <Statement>
                <Assignment>
                  <Identifier>x</Identifier>
                  <Expression>
                    <Simple Expression>
                      <Term>
                        <Factor>
                          <Identifier>x</Identifier>
                        </Factor>
                      </Term>
                      <Additive Op>-</Additive Op>
                      <Term>
                        <Factor>
                          <Literal>
                            <Integer Literal>1</Integer Literal>
                          </Literal>
                        </Factor>
                      </Term>
                    </Simple Expression>
                  </Expression>
                </Assignment>
              </Statement>
              <Statement>
                <Write Statement>
                  <Expression>
                    <Simple Expression>
                      <Term>
                        <Factor>
                          <Identifier>x</Identifier>
                        </Factor>
                      </Term>
                    </Simple Expression>
                  </Expression>
                </Write Statement>
              </Statement>
            </Block>
          </While Loop>
        </Statement>
        <Statement>
          <Return>
            <Expression>
              <Simple Expression>
```

```
              <Term>
                <Factor>
                  <Literal>
                    <Integer Literal>1</Integer Literal>
                  </Literal>
                </Factor>
              </Term>
            </Simple Expression>
          </Expression>
        </Return>
      </Statement>
    </Block>
  </Function Declaration>
</Statement>
<Statement>
  <Function Declaration>
    <Identifier>recursiveTest</Identifier>
    <Formal Parameters>
      <Formal Parameter>
        <Identifier>x</Identifier>
        <Type>int</Type>
      </Formal Parameter>
    </Formal Parameters>
    <Type>bool</Type>
    <Block>
      <Statement>
        <If Statement>
          <Expression>
            <Simple Expression>
              <Term>
                <Factor>
                  <Identifier>x</Identifier>
                </Factor>
              </Term>
            </Simple Expression>
            <Relational Op>></Relational Op>
            <Simple Expression>
              <Term>
                <Factor>
                  <Literal>
                    <Integer Literal>0</Integer Literal>
                  </Literal>
                </Factor>
              </Term>
            </Simple Expression>
          </Expression>
          <Block>
            <Statement>
              <Return>
                <Expression>
                <Simple Expression>
                  <Term>
                    <Factor>
                      <Literal>
                        <Boolean Literal>true</Boolean Literal>
                      </Literal>
                    </Factor>
                  </Term>
                </Simple Expression>
              </Expression>
            </Return>
          </Statement>
        </Block>
        <Block>
          <Statement>
            <Return>
              <Expression>
                <Simple Expression>
                  <Term>
                    <Factor>
                      <Literal>
                        <Boolean Literal>false</Boolean Literal>
                      </Literal>
                    </Factor>
                  </Term>
                </Simple Expression>
              </Expression>
            </Return>
          </Statement>
        </Block>
      </If Statement>
    </Statement>
  </Block>
</Function Declaration>
</Statement>
<Statement>
  <Write Statement>
    <Expression>
      <Simple Expression>
        <Term>
          <Factor>
            <Function Call>
              <Identifier>noParams</Identifier>
            </Function Call>
          </Factor>
        </Term>
      </Simple Expression>
    </Expression>
  </Write Statement>
</Statement>
```

```
</Program>
---------------- Output ----------------

4
3
2
1
0
1
```

Observing the above output, one may note that the program was correctly executed, starting with the value 5, which is passed onto the second function to test whether it is greater than 0. It is then reduced by 1, and output on screen. This cycle results in the digits 4 thru 1 being output on screen, followed by the number 1 which is returned by the initial function called through the write statement. This test therefore provides concrete grounds for the functionality of this compiler.

8. Read-Execute-Print-Loop

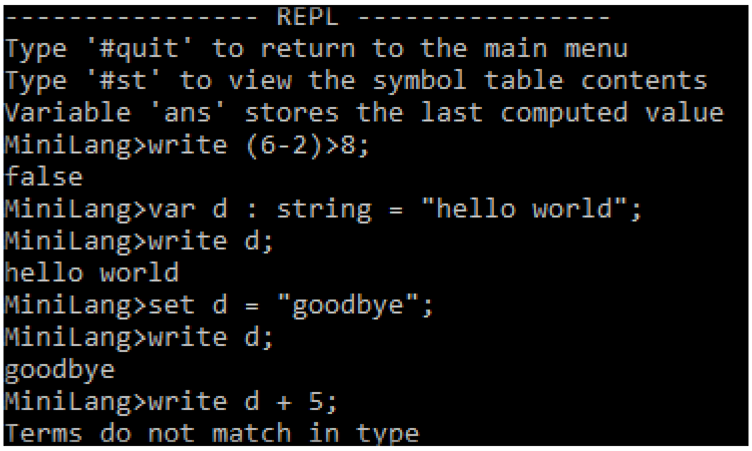
The Read-Execute-Print-Loop (REPL) is a command-line interface designed to handle program statements for compilation. This feature implements most of the above components in an adaptive manner, tending to individual statements at a time. This feature also has added features activated by means of keywords.

8.1 Command Execution

If the REPL option is selected from the main menu, the user is guided towards a command-line interface, where they may insert a single statement at a time. The input statement undergoes the same lexical analysis as a normal program would, however the interpreter and parser for the REPL are slightly altered. They are designed to take in a statement node rather than a program node. Therefore, instead of looping through child nodes and handling statement nodes till an End-Of-File, the REPL handles a single statement at a time, completing the whole compilation process, and then accepting another statement. This means that certain productions such as function declarations or function returns have been left out of the REPL completely, as they are invalid commands.

While designing the REPL, a decision was taken to completely omit the semantic analysis phase from this part of the program, as this could be handled by the interpreter due to the smaller size of input, as well as the difference in the manner of function of the symbol table.

Also, the XML printer was ignored for this section, as it was considered unimportant on such small operations.



```
---------------- REPL ----------------
Type '#quit' to return to the main menu
Type '#st' to view the symbol table contents
Variable 'ans' stores the last computed value
MiniLang>write (6-2)>8;
false
MiniLang>var d : string = "hello world";
MiniLang>write d;
hello world
MiniLang>set d = "goodbye";
MiniLang>write d;
goodbye
MiniLang>write d + 5;
Terms do not match in type
```

Figure 10. Testing of REPL functionality

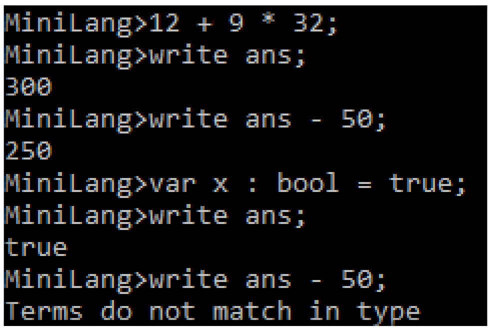
In figure 10 above, we may observe the functionality of the REPL. As was explained, statements are accepted one line at a time, but the same functionality as a normal compiler is present. In the first statement, an expression is evaluated and the result is output on screen. In the second and third statement, a variable **d** is declared to be of type string and is initialised to store “hello world”. We then see that this variable is successfully output on screen. This variable is then altered by means of the **set** keyword, which once again is successful. Finally, we observe that semantic analysis is still enforced when an attempt to perform an operation between a string and an integer is caught and stopped.

8.2 Added Functionality

Apart from the above mentioned functionality, the REPL hosts added features for convenience and testing purposes. The main design of the REPL is based on the idea of a single symbol table with a life time equal to the time spent in the REPL section. Therefore, upon entry into this section, a stack level is added to create the REPL symbol table, and this is only destroyed upon exiting the REPL. Therefore, any variable declarations or assignments throughout the REPL use shall be stored within the symbol table. Therefore, variables may first be declared in one statement each, and then used in another statement, with values being updated upon change.

One feature which is possible through this design, is the inclusion of a generic variable **ans** which stores the most recent result computed. Therefore, a simple expression such as ‘4 +

7' may be computed with or without a variable declaration, variable assignment, write statement or so on; and the value is still stored in **ans**, allowing the user to make use of this value later on. Variable **ans**, however, is still subjected to all other syntax and semantics rules, with type checking still being enforced upon on it.

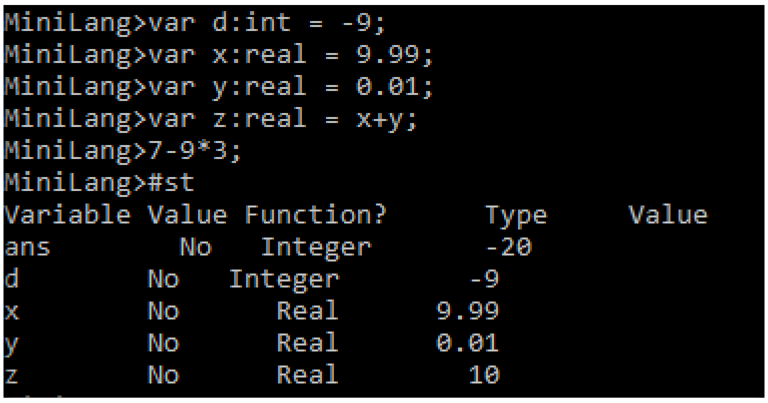


```
MiniLang>12 + 9 * 32;
MiniLang>write ans;
300
MiniLang>write ans - 50;
250
MiniLang>var x : bool = true;
MiniLang>write ans;
true
MiniLang>write ans - 50;
Terms do not match in type
```

Figure 11. Testing of ans functionality

Observing figure 11 above, we may understand how the **ans** variable is working. We see that it not only stores the value of the last computed expression, but it is also still subject to semantic analysis.

Another added feature is the use of the **#st** keyword. This keyword results in the printing of the contents of the symbol table onto the screen. As was mentioned above, a single symbol table is kept throughout the life-span of the REPL. Therefore, when **#st** is entered, the contents are simply output on screen, displaying the name, whether the variable is a function or not, variable-type and value (if applicable). This feature allows the user to better understand what is currently happening in the back end, and was highly useful when testing and debugging the program.



```
MiniLang>var d:int = -9;
MiniLang>var x:real = 9.99;
MiniLang>var y:real = 0.01;
MiniLang>var z:real = x+y;
MiniLang>7-9*3;
MiniLang>#st
Variable Value Function?      Type      Value
ans         No    Integer       -20
d        No    Integer        -9
x        No       Real      9.99
y        No       Real      0.01
z        No       Real        10
```

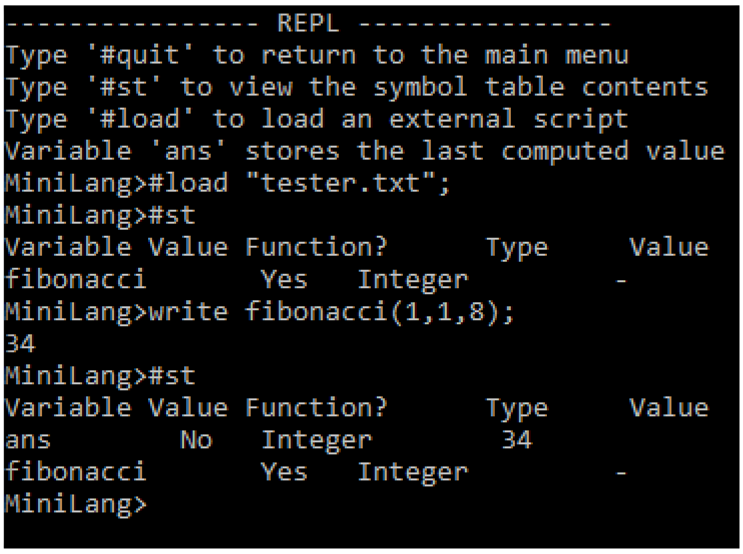
Figure 12. Testing of #st functionality

Figure 12 above showcases the functionality of the **#st** keyword. We may observe that the 4 declared variables, as well as the **ans** special variable appear within the symbol table, with their respective name, function-type, type and value. The inclusion of function definitions in the symbol table shall be discussed below.

The final added feature is the inclusion of a **#load** keyword. This keyword allows for an external script holding one single function definition to be loaded into the REPL, allowing for that function to be called upon and used. The decision to limit the possible statements within the loaded script to only one, was taken so as to ensure simplicity and computability. On text input, if the keyword is found to be a substring within the input, then a dedicated class function is called to handle the loading of the script.

The **InterpreterREPLScript** class performs a similar job to the ordinary interpreter, however it specifically handles the loading of a function into the REPL scope. This class is built out of visitor functions, but only contains a handful of functions; just the functions required to create a function variable to be stored within the symbol table. Once called, the first statement (function declaration) of the syntax tree is interpreted, generating a variable storing the function name, return type, parameters and a pointer to the rest of the body. The remainder of the body is not interpreted at this stage. Therefore, following the loading of a script, the function has been included in the symbol table but no actual execution has taken place.

The next step would be to input a function call with the given parameters, if any. This function call is handled as a normal REPL statement, accessing the function definition from the symbol table, updating the parameter values, and executing the body, returning the result generated by the function.



```
---------------- REPL ----------------
Type '#quit' to return to the main menu
Type '#st' to view the symbol table contents
Type '#load' to load an external script
Variable 'ans' stores the last computed value
MiniLang>#load "tester.txt";
MiniLang>#st
Variable Value Function?      Type     Value
fibonacci       Yes   Integer        -
MiniLang>write fibonacci(1,1,8);
34
MiniLang>#st
Variable Value Function?      Type     Value
ans        No   Integer        34
fibonacci       Yes   Integer        -
MiniLang>
```

Figure 13. Testing of #load functionality

Figure 13 above is a screenshot of a test case for the **#load** keyword. The previously used **fibonacci** function which accepts 3 parameters of type integer and returns an integer, is stored within the tester.txt text file. This time round, the function definition is not followed by any write statement, as this would result in an illegal script. Once the script is loaded, we call the **#st** keyword to load the symbol table. We observe that the function variable has been entered into the symbol table, with a return type of integer but a negligible value (as it does not store a fixed value).

We then proceed to generate a function call following a write statement, with parameters to produce the 8th Fibonacci number. The correct result is successfully output on screen, and **ans** has also been updated within the symbol table. This test, together with the previously conducted tests, prove that the REPL section of the compiler is fully functional and can handle correct, as well as incorrect statements.