Juan Carlos Hernandez Puebla
Professor Steinmetz
CSC 450
April 28, 2016

Test Document

This document describes the tests implemented for the Scanner, Parser, Symbol Table, and Syntax Tree.

Scanner

The Scanner was tested though various txt files, which tested for correct identity of valid and invalid tokens, as well as its token type and lexeme. This was done by iterating through the tokens found in the test file and asserting the values for token, token type, and lexeme. The first test file named input.txt tested for general recognition of different Token Types of the Pascal language. The Token Types included were ID, EQUALS, LESS_THAN, and NUM. The Scanner correctly asserted the expected values for token, token type, and lexeme. Included in this test file was the # character, which is a symbol not part of the Pascal language. Hence, it asserted that the token be false, the token type be null, and the lexeme be "#". The test that followed tested for more specific lexical conventions. For example, the second test named badComment.txt, tested for a bad written Pascal comment. The test file included a second { symbol within the initial { symbol and the ending } symbol. This represents an error and so the token was asserted to false , the token type was asserted to null, and the lexeme was "{". The next test was for a well written Pascal comment, where the entire comment should be ignored. Since nothing was written after the comment, the Scanner has reached the end of the file. Therefore, the token was asserted to false , the token type to null, and the lexeme to null. These assertions are made every time the Scanner reaches the end of a test file.

Parser

The Parser was tested by taking in a Pascal written program file as its argument. Then the Parser invoked its program function. If the program was recognized as a valid Pascal program, then a successful parse message was printed. The first test file named simplest.pas tested for the simplest valid Pascal program possible. It contained no declarations and no sub program declarations, just the following.

```
program f;

begin

end
.
```

This file was successfully identified as a valid written Pascal program. The next test file named singleDeclaration.pas, added a single declaration to the simplest valid program. The Parser successfully identified this file it as a valid program. The next test named declarations.pas, added more declarations of both integer and real types. The Parser identified it as a valid program as well. Eventually the Parser was tested with Pascal programs that included declarations, subprogram declarations, and compound statement, all together.

After the valid written programs were tested, then invalid Pascal programs were used to test the Parser. The first invalid test file named badDeclarations.pas tested for an invalid declaration due to its

standard type being declared as a double. The Parser was able to identify the non matching Token Type for the standard type function. It also successfully printed out the line of where this error occurred. The rest of the test files tested for incorrect implementations of subprogram declarations and incorrect implementation of a compound statement.

Symbol Table

The Symbol Table was tested by adding different program names, function names, procedure names, and variable names to it. These names were then tested for their correct type. For example, initially a program name such as “Simple” was added to the Symbol Table. This represents a valid addition to the table since this identifier had not been added to the Symbol Table earlier as any other type. Multiple duplicate names were added as different types and only the first initial addition for the identifier passed as a valid addition. The rest of the additions failed since that identifier had already been added to the table as a different type. This was also the case for the same names that were added as the same type. It failed to be added to the table since it was already in the table.

Next, the correct type associated for each name was tested for. For example, the name “Simple” was tested for if it was a program name, function name, procedure name, and variable name. The only case that passed as true was for program name since it was initially added to the table as a program name. The rest of the cases rightfully passed as false cases. This was done for the various names of various types added earlier to the table. Some names that were never added to table was also tested for. All of the checkups of types correctly passed as false since they were never added to the table.

Syntax Tree

The Syntax Tree was tested initially by manually putting together the different components that make up a pascal program. After this, the syntax tree was tested for correct generation once it had been integrated to the Parser. One particular test case was the sample.pas program, which is noted at the

bottom. For the initial test case, individual nodes such as a Variable Node was created manually for what would be declared variables in the program. The same was done for other nodes such as Compound Statement Node, Operation Node, and Value Node. Once the sample Pascal program was manually constructed, it was tested for its correct corresponding syntax tree. Therefore, a string was typed out for what was expected for the result of the syntax tree to be. Then it was compared to the actual output. This test successfully passed since the expected and actual results were equal to each other.

Next, the various nodes of the Syntax tree were integrated to the Parser. Once again, the sample Pascal program was tested for correct output of its syntax tree. Although, this time the program its self was taken in as a file and parsed by the parser. As the Parser parsed the sample.pas file, it built its syntax tree until it reached the end of the program. The expected syntax tree was typed out and compared to the actual syntax tree produced. Once again this test case successfully passed.

The next case that was tested for was for a program that included an if statement, an else statement, and a while statement. This source program is listed after the first sample program below. Both the expected result and actual output were compared and were found to be equivalent. The last test involved the following expression 5 * 7 - 6 + 3 * (2 / 4). One more, the expected and actual output were equivalent. The source code for this program is listed below as well.

Test Case 1:

```
program sample;

var  dollars,yen,bitcoins:integer;

begin
   dollars := 1000000;
   yen := dollars * 102;
   bitcoins := dollars / 400
end
.
```

Test Case 2:

```
program sampleifwhile;
```

```
var  dollars,yen,bitcoins:integer;

begin
   dollars := 1000000;
   yen := dollars * 102;
   bitcoins := dollars / 400;

if dollars >= 10000 then

dollars := 1000

else

dollars := 8000;

while dollars >= 7000 do

dollars := dollars - 500

end
.
```

Test Case 3:

```
{5 * 7 - 6 + 3 * (2 / 4)}
program factortest;

var  expression:integer;

begin
   expression := 5 * 7 - 6 + 3 * (2 / 4)
end
.
```