# **CPSC 254 Lab 5 Group 2-3 Test Cases**

## R28 <Primary>

## R27 <Factor>

## R26.1 <Term'>

## R26 <Term>

## R25 <Expression>

## R25.1 <Expression'>

## R24 <Relop>

## R23 <Condition>

## R22 <While>

* The output for the <While> rule is a while (<Condition>) <Statement>
* This function returns true when there exists “(“, a <Statement>, and “)” lexeme in the token holder vector.
* This can be tested by creating a token\_holder vector consisting of token\_323 variables with and without the qualifiers and seeing what is returned.
* The expected outcome for a vector consisting of the appropriate lexemes is true and false every other time.
* If the return value is true, the statement “Test Passed” is printed out, whereas if false is returned, “Test Failed” is displayed.

## R21 <Scan>

## R20 <Print>

* Print returns true when it outputs a put(<Expression>);
* In order to test if this works we need to declare a few token\_323 variables with specific token and lexme types.
* the first thing we should do is update the token using .token\_update(“Keyword”)
* Then we can update the lexme using .lexme\_update(“put”)
* We then repeat this for (expression) using (“;”)
* Then we can push\_back all off the tokens and call the test using procedure\_Print passing in the token and location variables
* We can then output if the test failed or passed

## R19 <Return>

* Return returns true if it returns return <expression> ;
* So in order to test this, lets test if it works properly first and foremost, just using a vector with “return” and “;”
* this function creates a vector of token 323 called all\_tokens to store the following tokens:
  + return\_token which contains the value “return” pushed by lexeme\_update identified by the qualifier “Keyword” in token\_upd
  + ate
  + semicolon\_token which contains the value “;” pushed by lexeme\_update identified by the qualifier “Operator” in token\_update
* The vector all\_tokens stores the previous token values
* The function then checks whether the values are stored properly by calling procedure\_Return(all\_tokens, loc)
* If the value returns true, we print out the message “Test passed: …” etc, and the same for if the test fails, we print “Test failed:..” etc.

## R18 <If>The output from the <If> rule is either a “if (<Condition>) <Statement> fi” or “if (<Condition>) <Statement> else <Statement> fi”

* For the “if (<Condition>) <Statement> fi” result:
  + Input: reading in a text file that contains a complete and valid “if” statement.
  + Expected output: for the procedure\_If() function to return true if the “if” statement contains a condition and at least one statement.
* For the “if (<Condition>) <Statement> else <Statement> fi” result:
  + Input: reading in a text file that contains a complete and valid “if else” statement.
  + Expected output: for the procedure\_If() function to return true if the “if else” statement has at least 2 statements, one for the “if” and another for the “else”
* These test cases should be sufficient since it tests the “if” statement on its own and as well as testing it if it is followed by an “else” statement.

## R17 <Assign> The output from this rule is “<Identifier> = <Expression>;”

* The <Assign> function returns a bool expression “return\_holder” which is true when there is an output string pushed back into the list of <Identifier> = <Expression>.
* The input is the location of the reading token and the all\_tokens vector made up of token\_323 variables.
* This can be tested by creating a vector of token\_323 variables and seeing if true is returned when there is no lexeme in the token\_holder that contains “;”, an Expression, or both.
* The expected output is true when there is a ; and Expression and a false when there is not.
* If the return value is true, the statement “Test Passed:...” is printed out, whereas if false is returned, “Test Failed:...” is printed out.

## R16 <Compound>

* Compound returns true when it returns a statement list
* So in order to test this, let’s declare a statement list using some token\_323 variables.
* We can use the built in functionality of a token list to update the token and lexme variables respectively. We can also pass in any types of variables such as Keyword, int, Identifier, x, Delimiter, etc.
* After that we will then store these tokens in the vector<token\_323> list using the push\_back() function.
* We can then initialize a location variable which holds the current location of the values.
* If everything goes well, we can call the function and pass in the vector<token\_323> list and location variable.
* Then simply just return true if the output is true and return false when the output is false.

## R15 <Statement>

## R14 <Statement List>

## R13 <IDs>

* <IDs> outputs <Identifier> and <IDs>
* So in order to test this, lets test if it works by making sure it takes some identifier string “a,b,c”.
* In order to test this, this test function creates a vector of token 323 called all\_tokens to store the following tokens:
  + ids\_token which contains the value “a,b,c” pushed by lexeme\_update identified by the qualifier “IDs” in token\_update
* The vector all\_tokens stores the previous token values
* The function then checks whether the values are stored properly by calling procedure\_IDs(all\_tokens, loc)
* If the value returns true, we print out the message “Test passed: …” etc, and the same for if the test fails, we print “Test failed:..” etc.

## R12 <Declaration>

* The output from the <Declaration> rule are <Qualifier> and <IDs>
* For the <Qualifier> and <IDs> result:
  + Input: reading in a text file that contains a qualifier (a keyword applied to a data type) such as “const int” and IDs with valid and unique names like num1, car\_model, account\_3.
  + Expected output: for the procedure\_Declaration function to return true. The vector should contain tokens for a valid qualifier and accepted ID names.
* These test cases should be sufficient since it tests applying a keyword to a data type to create a qualified data type. Also, testing a variety of formats for valid ID names with a combination of letters, numbers, and using underscores.

## R11 <Declaration List>

## R10 <Opt Declaration List>

The output for <Opt Declaration List> is either a <Declaration List> or <Empty>

* If the function procedure\_Declaration\_List returns true, then the function results in pushing back the output string into the all\_tokens vector or into an Empty one otherwise.
* The input for the <Opt Declaration List> is a vector of token\_323 variables and loc.
* To test this, the test function creates a vector comprised of token\_323 variable with and without the necessary lexeme or other requirements and seeing if it returns the proper value. True is returned for an appropriate string and False for an incorrect one.
* If the return value is true, the statement “Test Passed” is printed out, whereas if false is returned, “Test Failed” is printed out.

## R9 <Body>

* Body returns true when it returns a statement list
* So in order to test this, let’s declare a statement list using some token\_323 variables.
* We can use the built in functionality of a token list to update the token and lexme variables respectively. We can also pass in any types of variables such as Keyword, int, Identifier, x, Delimiter, etc.
* After that we will then store these tokens in the vector<token\_323> list using the push\_back() function.
* We can then initialize a location variable which holds the current location of the values.
* If everything goes well, we can call the function and pass in the vector<token\_323> list and location variable.
* Then simply just return true if the output is true and return false when the output is false.

## R8 <Qualifier>

## R7 <Parameter>

* <Parameter> outputs an <ID> or a <Qualifier>
* In order to test the <ID> result:
  + We create a token object that holds “IDs” and we update the lexeme with some name “a,b,c”
  + It’s than later pushed back to a token vector all\_tokens
* In order to test the <Qualifier> result:
  + The same logic from before applies, we create a token object that holds “Qualifier” this time, and we update the lexeme with the string “const”
  + It’s also pushed back onto all\_tokens
* The function then tests the results by running procedure\_Parameter using the all\_tokens vector.
* The result is than checked conditionally, if it returned successfully, we get a Test Passed, or Test failed if failed to return properly. This tells us whether the function works properly.

## R6 <Parameter List>

## R5 <Opt Parameter List>

* The output from the <Opt Parameter List> rule is either a <Parameter List> or <Empty>
* For the <Empty> result:
  + Input: reading in an empty text file. Or providing an empty vector of tokens.
  + Expected output: for the procedure\_Opt\_Parameter\_List function to return true and for there to be no tokens/ an empty vector.
* For the <Parameter List> result:
  + Input: reading in a text file that has example parameters such as (int num1, float sum) or generating tokens for the vector in the function.
  + Expected output: for the procedure\_Opt\_Paramter\_List function to return true. The tokens in the vector would contain the variables’ data type and name.
* These test cases should be sufficient since it covers both outputs from empty to a list of parameters.

## R4 <Function>

## R3 <Function Definitions>

## R2 <Opt Function Definitions>

## R1 <Rat20F>