CSCI 3081W Fall 2014 - Writing Assignment 1

Name : Wen Chuan Lee
ID : leex7095 [4927941]
Section : 006
Section TA : Mr. Andrew Hall

Introduction

Throughout the first two iterations of developing a functional translator that would take a file written in a domain-specific language, the Forest Cover Analysis Language (FCAL) and translate it to a C++ file, many observations and experiences regarding code development were gained. The use of automated testing strategies in a project with a deadline was definitely useful. However, as useful as it was, it was not sufficient to be the only form of testing used in the course of developing this project. This paper discusses many issues that arose during the development of the translator: whether the automated testing served it's purpose appropriately; the cost effectiveness of certain test cases; the testability of the program developed; along with ample explanations and examples taken from the project's source code. Each section contains a brief overview in the first paragraph, followed by a more in-depth discussion of the topic. A conclusion is then provided to summarize the entire project and experiences.

Automated Testing Strategies

Our automated testing strategy was mostly sufficient and appropriate for the project. The automation provided did work well with the development of the scanner and parser. However, manual testing was also required to complement the automated test cases as the automated test cases had a weakness in which stepping through the code using automated testing tools was impossible.

The tests provided were largely functional tests on various levels. Functional tests are defined as black box tests that will pass an input to a particular function and check the expected output, it does not know about the actual internals of each function. An example of the functional tests given are in the RegexTests.cpp file as well as the ScannerTests.cpp. The given tests assert for simple but essential items such as if regexes written actually matched the given input text, as well as if the list of terminals generated in the scanner were correctly placed into a linked-list. A good example of this would be the bad_syntax_good_tokens() test that would test if the scanner works as it should through reading in FCAL source code and generating a list of tokens generated. (code snippet is on the next page)

```
void test_scan_bad_syntax_good_tokens ( ) {
   const char *filename = "../samples/bad_syntax_good_tokens.dsl" ;
   char *text =  readInputFromFile ( filename )  ;
   TS_ASSERT ( text ) ;
   Token *tks = s->scan ( text ) ;
   TS_ASSERT (tks != NULL) ;
   tokenType ts[] = {       intConst, intConst, intConst, intConst,
                            stringConst, stringConst, stringConst,
                            floatConst, floatConst, floatConst,
                            matrixKwd, semiColon, comma, colon,
                            leftCurly, leftParen, rightCurly, rightParen,
                            plusSign, star, dash, forwardSlash,
                            equalsEquals, lessThanEqual, greaterThanEqual,
                            notEquals, assign, variableName, variableName,
                            variableName, variableName, variableName, variableName, variableName,
                            intKwd, floatKwd,  stringKwd,  whileKwd,
                            endOfFile    } ;
   int count = 39;
   TS_ASSERT ( sameTerminals ( tks, count, ts ) ) ;
}
```

--- The test case above effectively checks for all token types to ensure program works correctly by matching the right regular expressions (regexes) and generating the right tokens.

The automated tests definitely assisted in pushing out a faster and more robust program in a short time as we are now given various test cases to test that the program written is actually robust and error-free, enabling a shorter release cycle. Through the use of Makefiles which automated the whole process of compiling and running the test cases, faster and more frequent testing was achieved. A total of 2 weeks were allocated for both iterations, with the requirements for iteration 1 being given on Sept 30 and due on the Oct 14, and specifications for iteration 2 given on Oct 14 and due on Oct 28. The time given was sufficient for both iterations as the requirements were fulfilled by the given due date with the program passing the given and additional written test cases, ensuring that an each iteration with no known bugs was released. However, if even more time was supplied, the project could have been even more optimized in terms of code execution as well as organization of the files. The total amount of time spent for iteration 1 was roughly over 15 hours cumulatively, while iteration 2 took only over 8 hours with the testing included. The reason for this could be that the base of the program was laid out properly in the first iteration and allowed for swift improvements to the previous codebase, having automated test cases also greatly sped up testing and rapid debugging of the program for both iterations as well.

The test cases while helpful were not completely inclusive of every single test scenario that was required of the program to get working. More test cases had to be written to ensure that components of the program (Scanner and Parser) did not incorrectly match regex and

generate tokens as well as parsing the results. Below is an example of the test cases that had be written in addition to the provided ones.

```
void compare_terminals(const char* inString, tokenType tT){
    Token *tks = s->scan(inString);
    TS_ASSERT (tks != NULL) ;
    tokenType ts[] = { tT, endOfFile } ;
    TS_ASSERT ( sameTerminals ( tks, 2, ts ) ) ;
  }

  void test_terminal_intKwd () { compare_terminals(" Int", intKwd);}
  void test_terminal_floatKwd () { compare_terminals(" Float", floatKwd);}
  void test_terminal_stringKwd () { compare_terminals(" Str", stringKwd);}
  void test_terminal_boolKwd () { compare_terminals("  Bool", boolKwd);}
  void test_terminal_trueKwd () { compare_terminals("  True", trueKwd);}
  void test_terminal_falseKwd () { compare_terminals("  False", falseKwd);}
  void test_terminal_matrixKwd () { compare_terminals("  Matrix", matrixKwd);}
```

--- Figure showing test cases that checked for the expected generation of terminals. A helper function was also written to eliminate duplicate code with trivial differences.

However, the test cases provided were not sufficient as if the program only passes the given test cases, many other untested cases remain and bugs might still exist in the program. An example of this was the regex for matching strings, there was a bug in the regular expression that would also incorrectly match the "[" character, causing issues with the parsing when the translator ran on other FCAL programs. Fortunately, as design principles where adhered to the bug was found and fixed quickly. It was however indicative that while the test cases greatly helped test and provide confidence that the program functions, it was not sufficient to test for correctness.

Regarding test automation, the automated tools used in this project were the CxxTest suite as well as makefile automation. The use of CxxTest suite was considered as restricted to test execution. As for the makefile automation, it was used extensively in both compiling the actual program as well as running the test cases for it. By executing a makefile target, the test cases could be quickly executed. These tools were sufficient for the project as it allowed for very fast testing (executing the command "make run-tests" would execute all the tests, approximately 100 of them in total in a few seconds).

However, the use of manual testing was employed in this project as well. It is not the core of testing, but rather it serves the purpose of complementing the automated tests. In the first few revisions of iteration 1, in which the program compiles and executes the test cases successfully, but did not terminate and entered an infinite loop, the use of the C++ print statement (also known as 'cout') was used to narrow down the error in the *Scan* function. The portion of the code that caused an infinite loop was not evident as the test cases could not show where this occurred. The use of the GNU Project Debugger (GDB) was proposed but was

considered to be time consuming as stepping through the code that would jump to other functions (which were correct) to produce the infinite loop would be ineffective and time consuming.

```
for(int i = 0; i < endOfFile; i++)
{
        numMatchedChars = matchRegex (&regArray[i], text);
        if (numMatchedChars > maxNumMatchedChars) {
                maxNumMatchedChars = numMatchedChars ;
                term = static_cast<tokenType>(i);
                std::cout<<"Found match. term is semicolon"<<term<<std::endl; printf("Text matched: %.
*s\n",numMatchedChars,text);
        }
}
printf ("Regexes successfully completed matching \n");
```

--- Figure showing an example of debugging with print statements.

The manual approach to testing was also used nearing the end of iteration 1, in which at least 80% of the test cases passed with the remaining failing due to a small error. As such print statements such as cout to print the entire list of tokens allowed for manual review of the list of tokens to specifically pinpoint errors. In cases such as these it was much more helpful to print out the entire list than have Cxx-Test only report saying that the assertion failed. This is shown below:

```
while (currToken->next != NULL) {
          cout << "Current Token :" << currToken-> terminal<< "    ";
          cout << "Lexeme : " << currToken-> lexeme << endl;
          currToken = currToken -> next;
        }
        currToken = tokens;
         pr = parseProgram( ) ;
    }
    catch (string errMsg) {
        cout << "Error at: " << errMsg << endl;
         pr.ok = false ;
         pr.errors = errMsg ;
         pr.ast = NULL ;
    }
```

--- Figure above displaying a snippet of the *parse* function in *parser.cpp*, in which print statements are used to view the entire mapping of tokens that are parsed, and where the error might be.

Costs of Repetitive Testing throughout Development

The definition of 'cost-effective' we want to consider when evaluating the test cases is how repetitive it is, which is also the number of times the test cases will be executed. The more times a test case will be executed, the potential cost effectiveness increases, making the test case an ideal candidate for automation. The set of tests that are considered most cost effective is the 48 test cases written for *scanner* in iteration 1. This is because the test cases were re-executed every time the entire set of test cases are run. The least cost effective test cases would be the test cases written for the parser to ensure correctness, as it was executed fewer number of times. There were no test cases that are considered to be not cost effective at all since all test cases were reused. The test cases are executed in a bottom-up approach in which the program is improved as more test cases are added in. There is a high return on investment on testing the *scan* function as it is the main component of the program that scans and matches text, and the makefile is configured to always execute the scanner test cases each time testing is run.

As the provided test cases only test for very basic functionality as well as the entire function, the test cases were executed every time code for a basic functionality was written, and after the entire function was completed. The number of times the test cases were executed for both iteration 1 and 2 was approximately 40 times per iteration. This estimate was based on the fact that the tests were executed 20 times upon setting up the basic functionality of the code as well as the remaining times to debug the entire program when coding was completed. In short, most of the test cases were executed in the beginning to ensure for program correctness. The provided test cases were executed about 100 times in total throughout the 2 iterations. The test cases were very reusable and allowed for frequent retesting of the functions previously written. This provided confidence that future changes to the program did not introduce errors to the previous functions. This was especially useful during a redesign of the code structure with the aim of optimizing the program and making the code much more readable.Test cases were a vital part in making sure that the program still works as intended after introducing the changes. The following figure shows how to execute all the test cases written in a single command with the use of a makefile.

```
~/3081repo/group-repo/repo-group-VictoriousSecret/project/src$ make run-tests
```

Program Testability

The concept of testability depends on the design of the software to be test-friendly, as well as the design of the environment to allow for automated testing. The components of the translator built so far are considered to be designed for testability. This is largely due to the fact that suggested design guidelines were adhered to.

Furthermore, the translator was built in a Unix environment in which the GNU C-Compiler (GCC) was installed and the CxxTest suite was made available for use allowed for

a conducive testing environment. This made compiling and testing more convenient than using a Windows system.

Designing and implementing a test-friendly translator was achieved through following the design principle of modularity, where the entire program is broken up into smaller functions that form the building blocks of the *scan* function. This allowed for independent testing of each function. For example, in *scanner.cpp*, testing the main *scan* function first required testing of these smaller functions:

- consumeWhiteSpaceAndComments
- matchRegex
- constructors for Scanner and Token

In addition to the modularity of the translator, separate files for individual testing of the scanner, the regexes, and the parser. With the scanner and parser being developed separately, an isolated test of each layer could be done, as well as an almost-exhaustive test of error conditions. Mock-ups of certain functions of the code (such as the passing a string directly to the *scan* function instead of depending on *readInput*) can be used to test parts of the translator already written. This allowed for isolated testing, which reduced the scope of where the error may occur, and systematically testing all errors.

The Scanner does not have assertions to allow for self-diagnosis and proper detection of error conditions. While both components of the program would not crash and fail during execution, there was no self-diagnosis. The Parser however contains a try-catch block that would attempts to initialize a scanner and attempt to retrieve the linked-list of tokens, and would print out an error message upon failure. There were no conditional compile directives in place as well as the program is meant to only compile and execute on Unix machines, or machines with G++ compiler installed. However, the program handled exceptions by returning a clearly erroneous result. For example, the *readInput* function will return a null object if the file is not available. Another example of this is the lexicalError token produced when the *scanner* function encounters characters that do not match any of the regexes at all. This is shown in the code snippet below.

```
if(term == lexicalError){
        maxNumMatchedChars = 1;}
std::string lex (text, maxNumMatchedChars);
```

An aspect of the program that may be considered as a weakness is that both the scanner and parser do not print out error messages. While there exists no incomplete or incomprehensible error messages, useful error messages are missing from both components, as per the example above, it would be very easy to notify the user that an error matching the text was found, simply by adding a “cout” statement. While currently the only means of knowing if an error has occurred when the Parser attempts to parse a lexicalError token, it is sufficient for our use case as the error is not serious enough to stop the entire program. The proposed

improvement to the code snippet above, in which we notify the user of an error, or even stop the entire execution of the program is shown on the next page.

```
if(term == lexicalError){
        std::cout << "ERROR: Character does not match any Regex." << std::endl
        maxNumMatchedChars = 1;
        exit (EXIT_FAILURE); }
std::string lex (text, maxNumMatchedChars);
```

Conclusion

In the two iterations used to build a translator that consisted of a scanner and a parser, our automated testing strategy was considered sufficient and appropriate for the project. The automation served the purpose allowing for test cases that are fast, isolated and repeatable. However, manual testing was also required to complement the automated test cases as the automated test cases had a weakness in which stepping through the code using automated testing tools would be very time-consuming and almost impossible.

The most cost effective test cases are the test cases provided for the *scanner* in iteration 1. This is because the test cases always re-executed every time the entire set of test cases are run, making it the most executed test case among the others. The least cost effective set of test cases would be the test cases written for the parser. This is because they have been executed less times and are newly made given the current stages of development of this program.

The project was designed for testability from ground up. The approach taken in designing and implementing the translator made it test-friendly by adhering to the principles of modularity, in which the entire program is broken up into smaller functions that are the building blocks of the scan function. Isolated testing was performed on each individual part of the program before testing the main function (*scan* and *parse*), reducing the scope where error that may occur. However, an aspect of the program that may be considered as a weakness is that both the scanner and parser do not print out error messages. A simple modification can be made to the portion of the code that checks for a LexicalError token to notify the user or even stop further execution of the program.

Automated testing strategies definitely accelerated the development of the project as it would not be feasible to execute the test cases manually every time. Development was also smoother as the project was designed for testability from ground up and having cost-effective test cases. Therefore, automated testing should be considered compulsory where possible, complemented with manual testing such as adding print statements to step through the code where errors cannot be directly pinpointed from the automatic test cases. Employing the correct strategy to automated testing is vital in the success or failure of a project.