Andrew Eissen

UMUC CMSC 330

7/12/18

Project 1: Recursive Descent Parser in Java

Section I: Project description

The CMSC 330 Advanced Programming Languages recursive descent parser project requires that the author implement a recursive descent parser capable of accepting a user-input text file, converting that text file's contents into a listing of properly formatted tokens/lexemes, and parsing the results into a Swing-based GUI. Making use of the grammatical conventions provided in the project rubric, the program must check all tokens and their associated progression (as stored within an `ArrayList`) to ensure that all elements are properly represented and displayed in the GUI.

Section II: Deliverables

Included in the Project 1 packet is this documentation file in `.pdf` form, a set of five (5) plaintext files used in the examples herein to test the program's response to different types of legal and illegal grammar, and the four (4) Java files necessary for the program to run. These files are `Calculator.txt`, `AllOptionsUsed.txt`, `Nest.txt`, `ExcessSpaces.txt`, `ErrorTest.txt`, `Application.java`, `RecursiveDescentParser.java`, `Type.java`, and `Token.java`. Additionally, included in a separate folder are the six (6) high quality full-sized screenshot images featured in this document for ease of viewing. The author understands that images imbedded in Word documents can appear grainy if not sufficiently large and has thus elected to include the screenshots themselves along with the other deliverables.

As the author chose to hand-write the program in Sublime Text 3 and thus did not make extended use of an integrated development environment for most of the project's duration, no ancillary folders related to the running of this program in the Eclipse or Netbeans IDEs are included. The author recommends a simple console-based compilation of the main file "`Application.java`" via the following console commands: "`javac Application.java`" and "`java Application`".

Section III: Development phases

As with all projects, the author went about constructing the program in a several phase approach. He began first with the implementation of the rubric requirements in a poor but workable format, producing a rough working codebase capable of properly executing the required operations but which was rife with inefficient, duplicate code. The second phase concerned the necessary optimizations, introducing improved code via the application of proper Java development principles like the use of reflection, consolidation of copy/pasted code into methods, and cleaner, more easily followed layout changes and evolutions.

Finally, in the third phase, the author went about unit testing the program's reactions to both properly formatted and improperly arranged user input in the form of different text files' content. This phase saw the eradication of several longstanding bugs that had persisted since the first phase, such as bugs forbidding two panel widgets from existing together on the same panel without the latter's layout manager affecting the former's. Other bugs fixed during this period of testing included a bug that created `Type.UNKNOWN` tokens for excess whitespace characters encountered between file content tokens, as well as a bug that improperly nested panels in a subtly incorrect manner.

Section IV: Design and approach

Of foremost importance to the development of the program was the issue of expected usability and user-friendliness. To make the program conducive to easy usage by users of all experience levels,

the author elected to build a primary GUI displaying a set of options buttons and a status log that would allow the user to select new files to parse without having to recompile and rerun the program multiple times. Further, this status GUI included a button allowing the user to clear old log history displaying tokens and errors from previously parsed files. The idea for a status log preserving the program's output messages came from a series of JavaScript mass-editing plugins the author designed for use with the MediaWiki PHP platform that required a status log for the displaying of operations and associated errors encountered. The concept has similarly been applied to projects from the author's other CMSC classes, including the author's CMSC 335 SeaPort project series.

Due to the presence of this `JFrame`-based GUI, the GUI results of successful parsing operations were displayed in a `JDialog` rather than a second `JFrame`, as the use of multiple `JFrame`s is a buggy design approach that often results in one `JFrame`'s exiting operation applying to all `JFrame` windows currently open. Furthermore, the author's ancillary research on the matter on the StackExchange network indicated that the use of multiple `JFrame`s is further seen as bad form by the vast majority of Swing developers. The use of the `JDialog` instead of a frame did not alter the output of the program at all, as `JDialog`s are generally capable of displaying all manner of Swing elements without issue, including those widgets likely to be encountered in the program grammar.

Concerning the lexer or tokenizer, the author waffled back and forth a few times as to the best way to approach assembling tokens from file contents. Originally, the author went with a regular expression-driven approach, making use of a regex string and the `String.split` function to divide individual lines of the file into `String` tokens. However, as the author read that comparisons between `String`s in `switch` blocks and `if` statements are more expensive operations than similar operations undertaken on primitive types, the author abandoned this in favor of a different tokenization paradigm. Instead, the author implemented an approach that divided individual file lines into `char` arrays,

iterating over their lengths to determine where tokens began and ended in relation to others. A pair of functions taken from the "C Program Formatter Written in Java" module in the CMSC 330 course materials were added to evaluate the resultant `char` composites and determine the most applicable `Type` for each. However, while this `char` approach worked, the related method was somewhat messier and more difficult to follow than that used in the previous regex approach.

As far as the resultant tokens themselves were concerned, the author originally went with a simple `ArrayList` of `String` tokens. However, once again inspired by the aforementioned module reading, he eventually created an `enum` class of all potential token types, fittingly called `Type`, along with a separate `Token` class that contained fields for `Type`, token `String` content, and the line on which the token was first encountered by the lexer (this field was used primary for error messages added to the status log). Though this approach was perhaps a bit more complex than the simple `String` comparison-driven approach employed during the first draft of the program, the author believes its implementation has enhanced readability and made the parsing operations easier to trace and follow by hand.

The author initially approached the parser part of the program logically, making use of a global counter integer that would be incremented as a set of `if...else` statements are encountered and passed. However, though this approach remained the most workable, it often resulted in a spaghetti code mess of statement blocks that were difficult to read and follow manually during the debugging phase. Wherever possible, the author attempted to remove copy/pasted duplicate code in these methods, making use of separate methods enhanced by Java reflection to consolidate code of similar purpose together. The author experimented with a number of other approaches making use of loops and the like, but ultimately decided that the method described above was the most ideal given the one week time constraint. If the project were like the aforementioned SeaPort project, extending over the

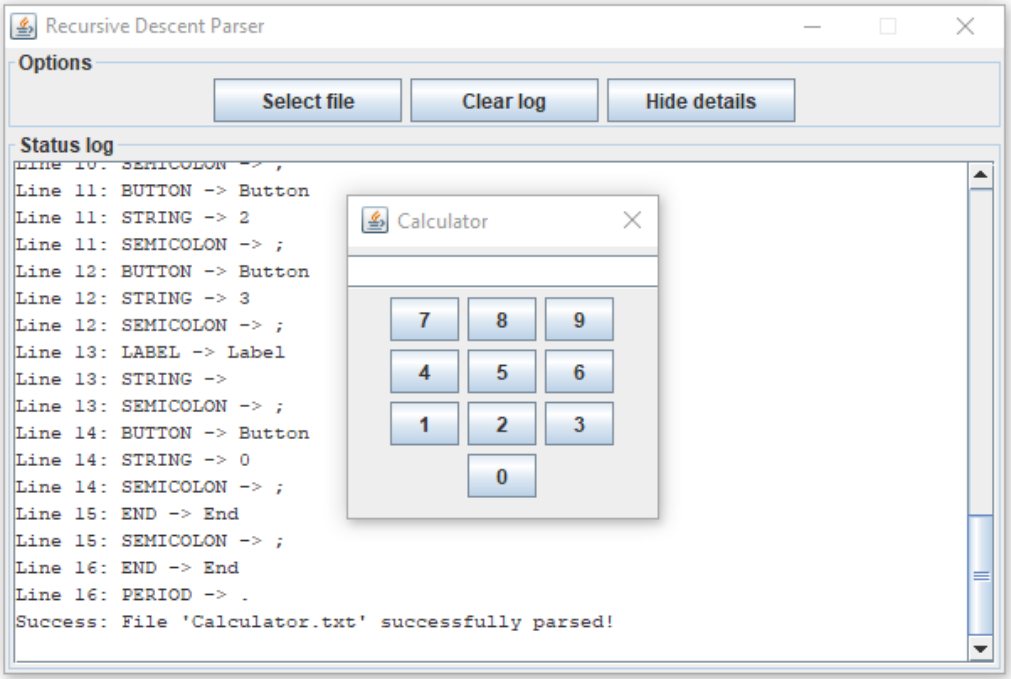
lifetime of the class, the author would likely have spent more time experimenting with other workable, more readable formats.

Section V: Test cases

The author made use of several files to test the output of the program in the course of its construction, five of which are included in this set of example test cases. Between all five test files, as required by the rubric, the example cases test all layout managers, widgets, and recursive aspects alike to show the program's reactions to different types of proper and improper input. Furthermore, included are a few examples of the program's reaction to user-pressed options buttons and the like. However, as these are not the primary functionality of the program, the author encourages the reader to experiment with the options provided in the status GUI in his/her own time.

***Test Case 1**: Rubric example*

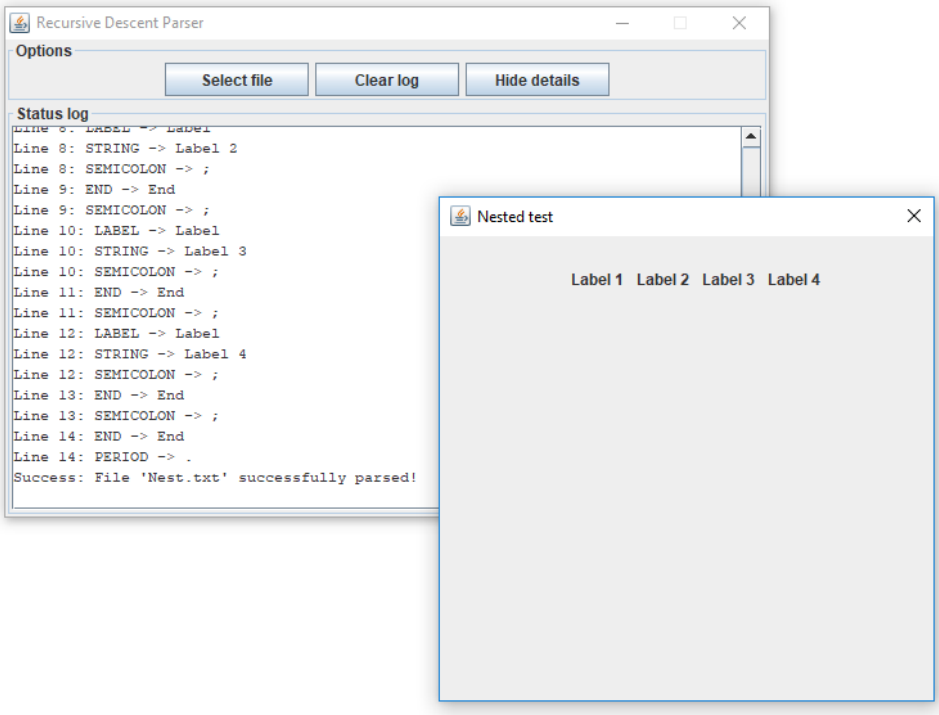
```
Window "Calculator" (200, 200) Layout Flow:
 Textfield 20;
 Panel Layout Grid(4, 3, 5, 5):
 Button "7";
 Button "8";
 Button "9";
 Button "4";
 Button "5";
 Button "6";
 Button "1";
 Button "2";
 Button "3";
 Label "";
 Button "0";
 End;
End.
```



The first example used by the author to test the output of the program was that provided in the Project 1 assignment rubric itself, the “Calculator” example. This example makes use of both the `FlowLayout` and extended `GridLayout` managers, augmented via the inclusion of several buttons, a space-filling `JLabel`, and a `JTextField`. As expected, the program displays a properly developed `JDialog` that matches the desired appearance provided in the rubric.

***Test Case 2**: Nested panels example*

```
Window "Nested test" (400, 400) Layout Flow:
  Panel Layout Flow:
    Panel Layout Flow:
      Panel Layout Flow:
        Panel Layout Flow:
          Label "Label 1";
        End;
        Label "Label 2";
      End;
      Label "Label 3";
    End;
    Label "Label 4";
  End;
End.
```

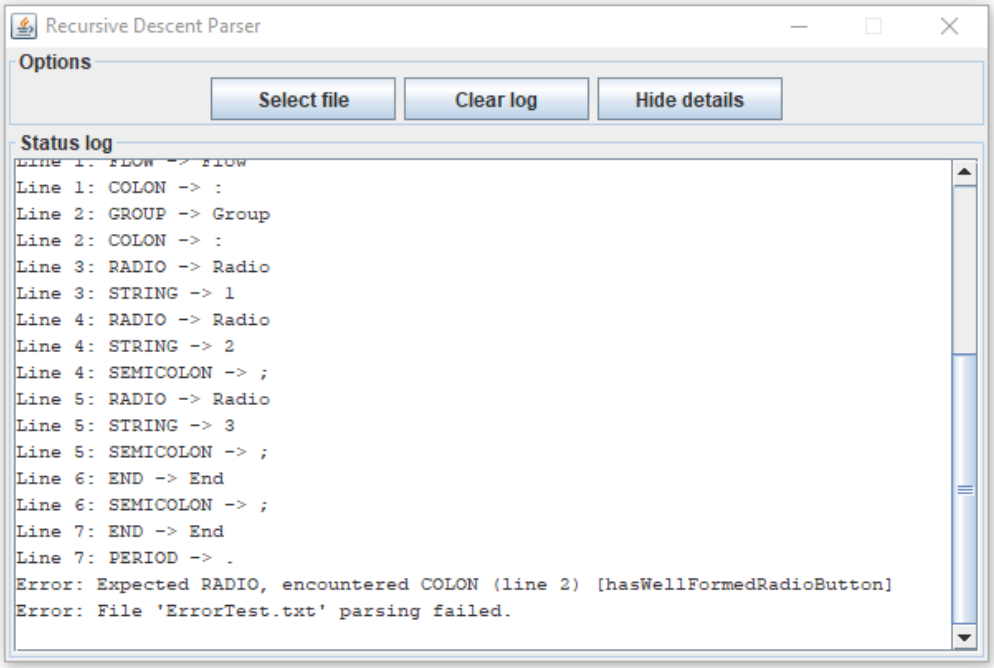


The second example included to test the program’s reaction to properly formatted input employs a set of nested `JPanel`s, all arranged within each other. Originally, this test case revealed an unnoticed bug that persisted for several days. This bug assembled a GUI like that shown above but did not technically nest the panels, instead arranging them deceptively beside each other as though they were properly nested. It wasn’t until the author tested the panel arrangements via the addition of black borders to all `JPanel`s that this bug was uncovered. This bug fix actually uncovered a solution to

another bug that had plagued the program during the first days, related to a newly added panel's layout manager affecting the layout of previously assembled panels. The current iteration properly handles nested elements in accordance with the rubric requirements, as illustrated by this test case.

***Test Case 3**: Broken input*

```
Window "Broken grammar" (300, 200) Layout Flow:
  Group:
    Radio "1"
    Radio "2";
    Radio "3";
  End;
End.
```

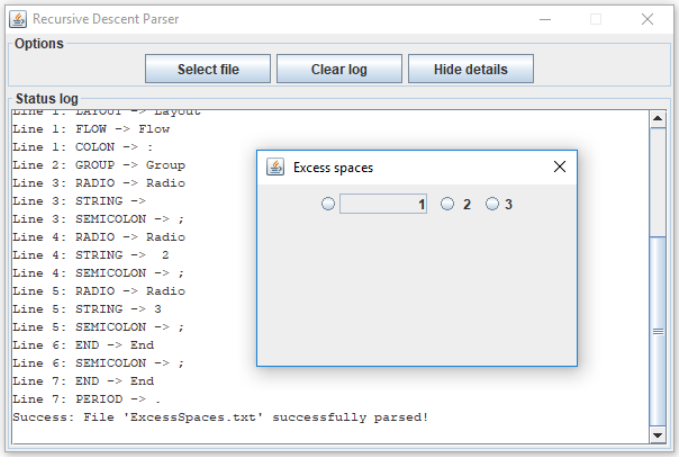


The third test case included herein displays the program's reaction to malformed or improper input. This file, titled "`ErrorTest.txt`," includes a pair of errors in its contents, namely an unneeded colon after the `Group` token and a missing semicolon on line 3 after the `String` "1." As per the project rubric requirements, only the first error encountered is reported in the status GUI log, formatted as "`Expected RADIO, received COLON`" followed by the line on which the error was found and the name of the method in which the error was encountered. Originally, the program displayed all related errors in other methods, a fact that was helpful in tracing the progression of faulty token types during the debugging phase. However, this functionality was eventually removed in favor of the first

message only being displayed, a decision made to bring the program more closely in line with the rubric requirements mentioned above.

***Test Case 4**: File with excess whitespace between tokens*

```
Window        "Excess spaces"          (300,         200)               Layout                  Flow    :
  Group
        Radio                     "                     1";
    Radio                    "2"                                                                    ;
              Radio "3"       ;
End               ;
End                                                .
```

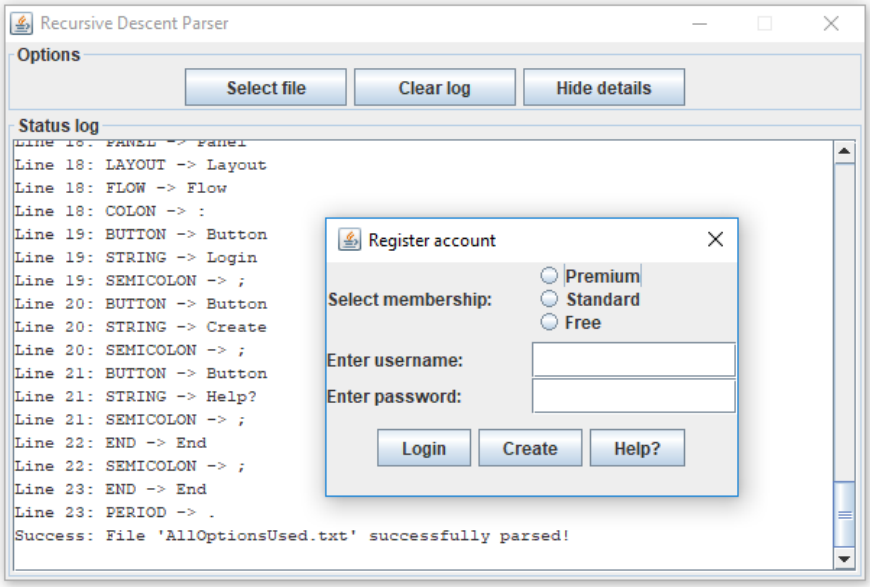


The fourth test case concerns the program's reaction to properly assembled, grammatically correct files that suffer from an excess of whitespace between tokens, words and symbols alike. In this example, the only excess whitespace retained is the set of spaces that appear in the first radio button's associated label on line 3. As the author made the assumption that all formatting in `String` labels is presumably part of the file creator's design strategy, nothing within quotation marks is edited or redacted by the program.

Prior to the running of similar test cases, the program logged such whitespace characters as `Type.UNKNOWN` tokens, a bug which necessitated proper removal of whitespace from potential tokens prior to the determination of their type. This was done via proper application of the `.trim()` method for `String`s, which enabled the author to ensure that only composite `String`s bereft of prefix or postfix whitespace were evaluated and tokenized.

***Test Case 5**: All widgets, layouts, and features included*

```
Window "Register account" (300, 200) Layout Grid (3, 1, 5, 5):
  Panel Layout Grid (1, 2, 5, 5):
    Label "Select membership:";
    Panel Layout Grid (3, 1):
      Group
        Radio "Premium";
        Radio "Standard";
        Radio "Free";
      End;
    End;
  End;
  Panel Layout Grid(2, 2):
    Label "Enter username:";
    Textfield 10;
    Label "Enter password:";
    Textfield 10;
  End;
  Panel Layout Flow:
    Button "Login";
    Button "Create";
    Button "Help?";
  End;
End.
```



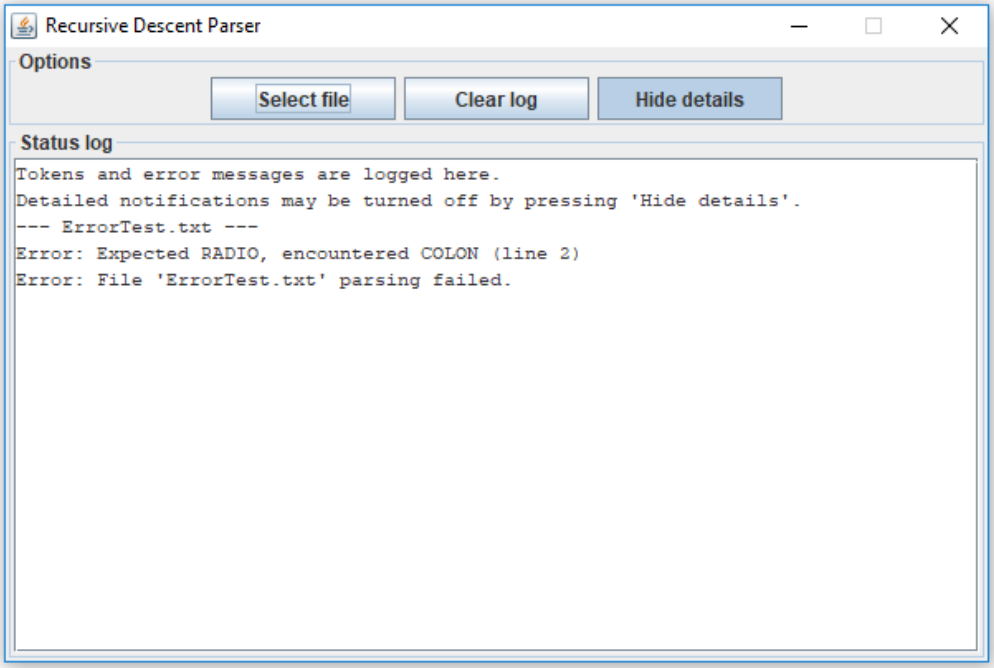
The fifth test case file, entitled "`AllOptionsUsed.txt`," is used to test all the possible widget combinations, nesting levels, and layout managers legally permitted in the project grammar, in accordance with the rubric requirement. The example itself is a login/registration modal of sorts constructed with Swing, designed to appear as a somewhat legitimate exercise in the program's ability to construct GUIs of actual usability beyond simple contrived examples.

The text file in question includes `JTextField`s, `JLabel`s, `JButton`s, `JPanel`s, and `JRadioButton`s in addition to instances of the standard `GridLayout`, extended four-parameter `GridLayout`, and `FlowLayout` managers. Furthermore, nesting is illustrated via a grouping of radio buttons within a panel within another panel, and multiple nested widgets are permitted to simultaneously occupy places within the same panel. The implementation of this latter feature caused the author many headaches, as it was originally impossible for a pair of panels, for instance, to simultaneously be nested together within the same outer panel without the first nested inner panel's

layout manager being negatively affected by the second's. This was one of the principle bugs encountered during the course of this assignment and required much testing to properly rectify.

***Test Case 6**: Status log output with "Hide details" button pressed*

```
Window "Broken grammar" (300, 200) Layout Flow:
  Group:
    Radio "1"
    Radio "2";
    Radio "3";
  End;
End.
```



The sixth test case is a "bonus" of sorts, included to illustrate one of the defined configuration options available to the user. If overwhelmed by the sheer number of included token types and values, error messages, and updates logged in the panel over the course of the program's lifetime, the user may simply press the "Hide details" toggle button included in the options buttons panel on the top of the status GUI. Pressing this button ensures that the program posts only the most relevant messages in the log, such as a simple "Success" vs. "Failure" message indicating whether or not the file contents were parsed properly and the first error message without the method name in brackets. This minimalist approach was not required by the rubric, but was added by the author simply for enhanced user-friendly value.

Section VI: Lessons learned and potential improvements

The primary lesson learned during the completion of the project was the use of reflection to assist in the construction of Swing components depending on the nature of the included method parameters. Though the author is a fan of reflection and has successfully applied its principles to the major projects of other classes, such as the aforementioned SeaPort project from CMSC 335, he has never used the concept in the creation of a method capable of constructing different GUI components from method parameters. Further reflection methods like "`getConstructor`" and "`Class.forName`," previously unknown to the author, proved to be very useful in consolidating code that would otherwise have remained in a messy, copy/pasted mess.

As far as potential future updates are concerned, the author harbors a desire to rewrite the so-called “spaghetti code collection,” an infamous set of several format progression-tracing methods in the heart of the `RecursiveDescentParser` class’s parser section. These methods make use of a great number of near identical `if…else` blocks that trace the tokens included in the class’s `ArrayList` of `Token`s. Under present circumstances, the methods do work as intended in accordance with the project rubric; however, the author would have liked to experiment further and perhaps develop a more readable, more optimized alternative. As stated previously, the author did experiment with several loop-based approaches, but was unable to find a workable alternative.

Furthermore, another improvement the author would consider would be the removal of lexer-related code to its own specific class, possibly called `Lexer.java`, that would further divide the various partitions of the program based on their specific roles in file content evaluation. However, the author decided to hold off on any such further divisions in the course of writing the program, believing that keeping all the operational code in one file would make tracing the progression of the program’s operations easier during the debugging phase.