University of Nevada, Reno

CITRIN: The C++ Interactive Interpreter

A thesis submitted in partial fulfillment
of the requirements for the degree of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

by

CHRISTOPHER A. SALLS

Dr. Michael Leverington, Thesis Advisor

December, 2013

Abstract

Many students, especially those learning their first programming language, struggle to understand the essential concepts of C++. The purpose of CITRIN is to create a tool that will help students to learn to program effectively with C++. The structure of C++ is more intuitive with CITRIN, which can show the execution of the program and not simply the output. By showing the execution of the program in a line by line analysis, CITRIN can show how programming concepts apply to the code that students write by giving immediate and detailed feedback.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Concept and Specification

Concept

The concept behind CITRIN included many different goals and ways to accomplish them. The main goal of the project is to help beginner programmers better understand the programming language of C++ and to make it seem as easy as possible. The project will complete this goal by creating an easy to use C++ interpreter that will walk beginners through their code, showing when different actions take place such as a variable change, and will help them fix their code by locating errors.

To keep the interpreter easy to use it will run on a simple and well-designed graphical user interface that allows for the easy viewing of important data. Users will be able to open an existing code file, or can write new code using the programs interface. Once they have a piece of code ready they can then interpret it using two options which include running through it start to finish or using a step, line by line, function. The interpreter will highlight which line it is currently dealing with and will display the current states of variables and output. The interpreter will also allow for users to save code, use a quick tutorial, and point users to errors.

The interpreter and graphical user interface are both written using java and were developed using eclipse. The graphical interface also includes a text editor which is implemented using the swing library. The interpreter is designed to run and evaluate C++ code, and the program should be able to run on any platform that can run a java executable.

There are two main groups of intended users. The first, mentioned earlier, is beginner coders and students in entry level programming classes. The interpreter will help them better understand what is happening when their code is running so that they can improve as programmers. The second group of intended users would be teachers. This tool has great potential for use in the classroom. Teachers can run programs on it and walk through the code line by line to show examples of concepts they are teaching in class. This will allow for students to learn aspects of coding faster and with less hassle.

There are several other C++ interpreters on the market today many of which have good qualities but also have aspects that hold them back. Most of the interpreters do a very good job of executing C++ and most of its functionality, but cost too much, are too large or run slowly, or have confusing and hard to understand graphical interfaces that slow down the interpreters use because the user must learn how to interact with the interpreter. These issues are what lead to the idea behind CITRIN.

CITRIN came about to combine the good aspects of previous interpreters and to leave behind the bad. CITRIN has a very straightforward graphical user interface as to not confuse the user. The code, output, and other details are all easily viewable in a three window interface. The program is also streamlined with no unnecessary add-ons that keep its size very manageable and allow for the interpreting to run quickly and smoothly.

This project has great open source value. It can be used as an educational tool across the nation in Universities and in the hands of anyone interested in learning to code. Keeping it open source will make this product much more accessible and makes it more attractive for schools that operate on a budget. Having it open source also means that teachers can work with the interpreter to fit it to their needs.

There is always room for our product to be further developed past the timeframe of this semester as well. The code is templated and documented so that it can be easily added to and changed. This was done so that as more C++ functionality is required for the interpreter it can be easily integrated. This was also an influence in keeping it open source. This project gets us good experience with working in a group which is a necessity in the professional world. Having the code well documented and organized also shows that we as a group are ready to deal with change.

Software Requirements

Table 1: Functional Requirements

R01	[1]	A user shall be able to type and edit C++ code.
R02	[1]	A user shall be able to select text.
R03	[1]	A user shall be able to cut, copy and paste text.
R04	[1]	A user shall be able to find and replace text.
R05	[1]	A user shall be able to undo and redo multiple levels of previous changes.
R06	[1]	A user shall be able to save and open text files.
R07	[1]	CITRIN shall provide a GUI for editing the code.
R08	[1]	CITRIN shall be able to interpret for loops.
R09	[1]	CITRIN shall be able to interpret C++ code and run code in a console.

R10	[1]	CITRIN shall be able to interpret declarations and assignments.
R11	[1]	CITRIN shall be able to interpret math operations.
R12	[1]	CITRIN shall be able to interpret if, else statements.
R13	[1]	CITRIN shall be able to interpret while loops.
R14	[1]	CITRIN shall give simple descriptions of syntax errors.
R15	[1]	CITRIN shall provide a GUI for running the code.
R16	[1]	A user shall be able to step through the code line by line.
R17	[1]	A user shall be able to set breakpoints in the code.
R18	[1]	CITRIN shall display current values of variables.
R19	[1]	CITRIN shall display the current state of boolean expressions.
R20	[1]	CITRIN shall be able to interpret do while loops.
R21	[2]	CITRIN shall provide syntax highlighting of C++ code.
R22	[2]	CITRIN shall be able to interpret iostream functions.
R23	[2]	CITRIN shall be able to interpret ifstream functions.
R24	[2]	A user shall be able to set the speed for the code to run through line by line.

R25	[2]	CITRIN will show which variables are in scope and which are inactive.
R26	[3]	CITRIN shall show the state of the file that is being used for iostream or ifstream.
R27	[2]	CITRIN shall be able to detect a change in a specified variable value.
R28	[3]	CITRIN shall have a function stack for execution history.
R29	[3]	CITRIN shall be able to detect if a specified variable is in the current statement.
R30	[3]	CITRIN shall have a tutorial to teach users how to use the program.
R31	[3]	CITRIN shall have events for callbacks.
R32	[3]	CITRIN shall define simple built-in events for callbacks.
R33	[3]	CITRIN shall be able to detect the change in boolean expressions.
R34	[3]	CITRIN shall be able to save interpreter settings (such as which variable to watch).
R35	[3]	CITRIN shall be able to save the current execution context.

Table 2: Nonfunctional Requirements

T01	CITRIN shall be implemented in Java.
T02	CITRIN shall be easily extendable.
T03	CITRIN shall have a simple user interface.
T04	CITRIN shall run on Windows XP and later.
T05	CITRIN shall run on Mac OS X.
T06	CITRIN shall produce correct results.
T07	CITRIN shall give warnings for undefined behavior.

Use Case Modeling

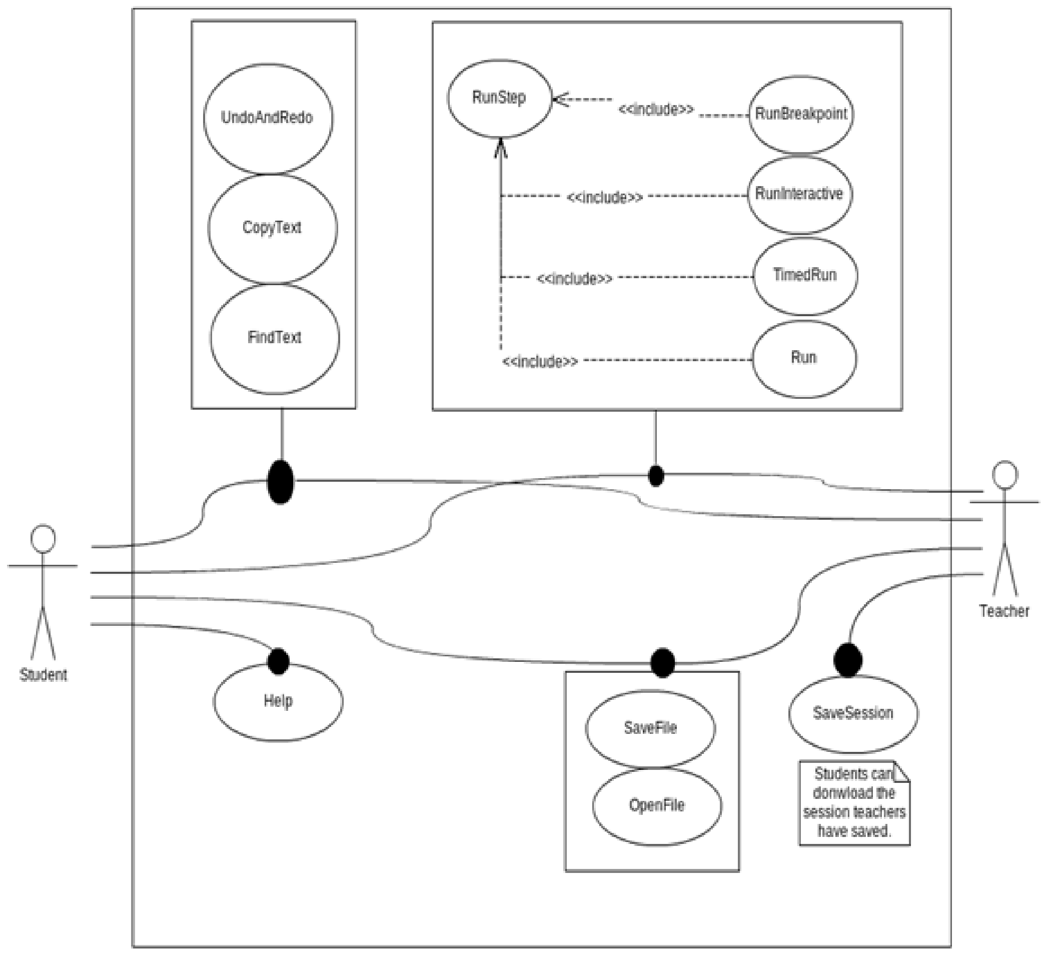


Figure 1: Use Case Diagram

Table 3: Use Case Descriptions

UC01	CopyText	The user highlights a section of text then uses either a keyboard shortcut or the menu to copy (or cut) the text. Then the user can use the paste command to copy the text to the cursor's location.
UC02	FindText	The user searches for a string of text by selecting the find text option. The text skips to the next line of code containing the search string which is highlighted.
UC03	UndoAndRedo	The user types a piece of code then uses a keyboard shortcut multiple times to undo the last few changes, then the user can use the redo command multiple times to reverse the undo's.
UC04	SaveFile	The user can save the code to file by selecting the save option. If it has not been saved before it would ask for a file name.
UC05	Help	The user can follow a tutorial on how to use the program or get information on the available commands and buttons

UC06	RunStep	The user will write the code and then step through the code line by line by choosing this option.
UC07	RunBreakpoint	The user will write the code first, set a breakpoint, and then choose this option to have the code run until it reaches that breakpoint.
UC08	RunInteractive	This option will allow the user to see what the program does as each line is written.
UC09	Run	This option runs through the entire written program without stopping.
UC10	SaveSession	The user can save the current session such as current execution context and any interpreter settings changed.
UC11	TimedRun	Steps through the code one line at a time. The user may adjust the speed at which steps are taken.
UC12	OpenFile	The user can open a previously saved code file and then may run that code or edit it.

Detailed Use Cases

Use Case: CopyText
ID: 1
Brief Description: The user can copy text and paste it to another location.
Primary Actors: User.
Secondary Actors: None.
Preconditions: There is some text that the user wants to copy (or cut) and paste to another location.
Main Flow: 1. The user highlights the section of text to be copied or cut. 2. The user uses a keyboard shortcut or menu option to copy or cut the text. 3. The user uses a keyboard shortcut or menu option to paste the text in the desired location.
Postcondition(s): The desired text has been pasted into the correct location.
Alternative flows: None.

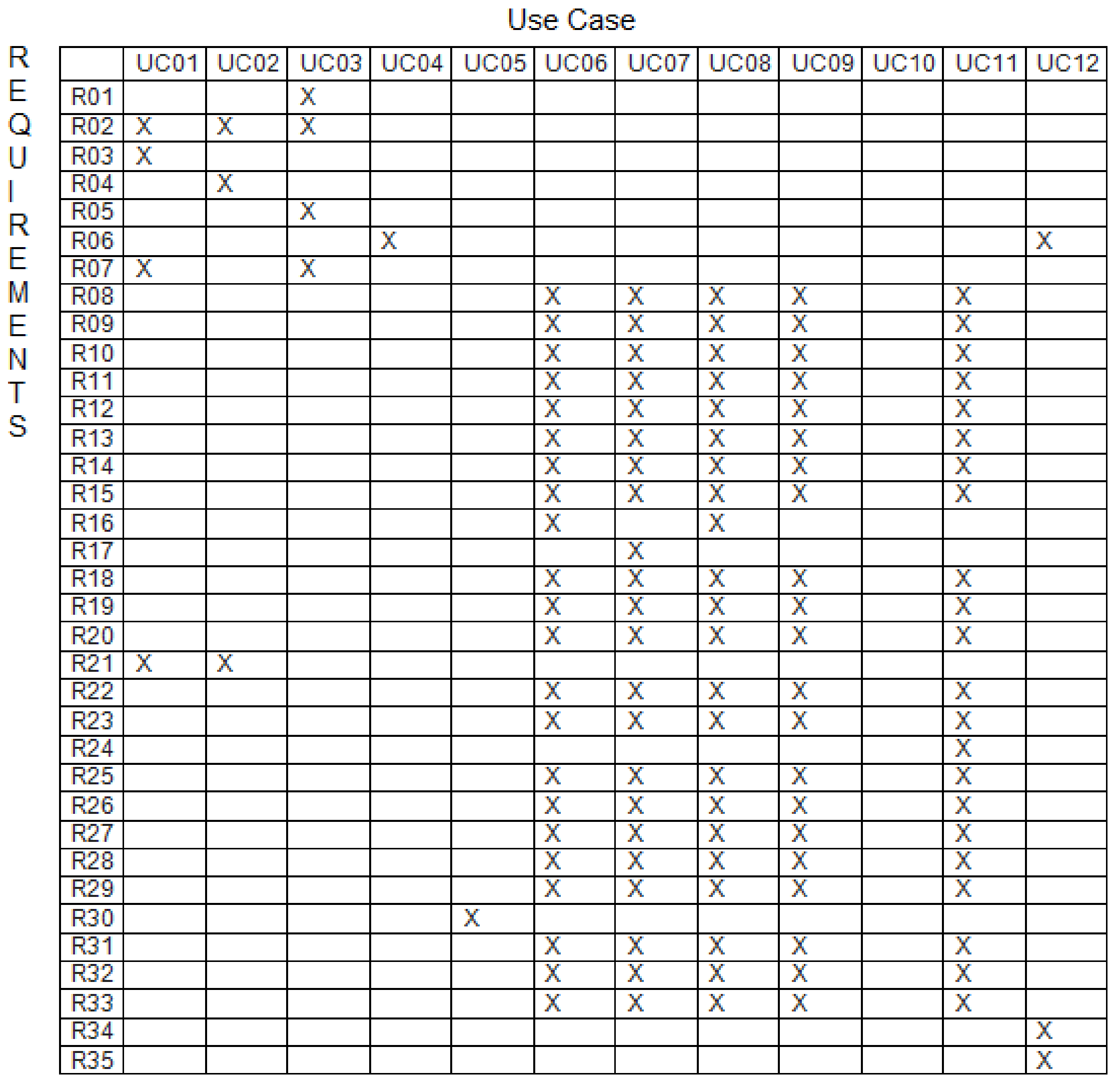
Use Case: FindText
ID: 2
Brief Description: Finds all the instances of some text in the file.
Primary Actors: User.
Secondary Actors: None.
Precondition(s): There is some text in the file.
Main Flow: 1. The user types in the word or phrase that he is looking for. 2. If an instance of the word or phrase is found 2.1. The word or phrase is highlighted. 2.2. The user can go to the next instance of the word or phrase. 2.3. The system will continue going to the next instance of the word or phrase until the user chooses to stop. 3. Else 3.1. The system tells the User that no matches were found.
Postcondition(s): None.
Alternative flows: None.

Use Case: RunBreakpoint
ID: 7
Brief Description: CITRIN interprets the code until it reaches the breakpoint.
Primary Actors: User
Secondary Actors: None.
Preconditions: 1. The user has written a complete program. 2. The user has set a breakpoint where he wants the program to stop.
Main Flow: 1. The user runs the code using the interpreter. 2. The program will stop at the breakpoint.
Postcondition(s): 1. The output screen will show what the output is at that break point. 2. The states screen will show the states of the variables, boolean expressions, or file iostream, ifstream at that break point.
Alternative Flows: None.

Use Case: SaveSession 1st Scenario: Demo Creation and Reloading
ID: 10
Primary Actors: Instructor
Secondary Actors: None.
Preconditions: The instructor has demo source code.
Primary Scenario: 1 The instructor runs the source code on the interpreter with desired input until the program reaches the state to show the demo with break/next/step. 2 The instructor hits the SaveSession button to save both of the execution context and the interpreter settings. 3 The students receives the saved session to get right into the demo condition.
Postconditions: Demo Session

Use Case: SaveSession 2nd Scenario: Debugging Session
ID: 10
Primary Actors : Student
Secondary Actors: None.
Preconditions: The students have bugs they cannot fix on their own.
Secondary Scenario : 1 The student runs the code on the interpreter and saves the session where the code is misbehaving. 2 The instructor can see the saved session to help them.
Postconditions : Bug Fixed

Requirement Tracing



Use Case

REQUIREMENTS

	UC01	UC02	UC03	UC04	UC05	UC06	UC07	UC08	UC09	UC10	UC11	UC12
R01			X									
R02	X	X	X									
R03	X											
R04		X										
R05			X									
R06				X								X
R07	X		X									
R08						X	X	X	X		X	
R09						X	X	X	X		X	
R10						X	X	X	X		X	
R11						X	X	X	X		X	
R12						X	X	X	X		X	
R13						X	X	X	X		X	
R14						X	X	X	X		X	
R15						X	X	X	X		X	
R16						X		X				
R17							X					
R18						X	X	X	X		X	
R19						X	X	X	X		X	
R20						X	X	X	X		X	
R21	X	X										
R22						X	X	X	X		X	
R23						X	X	X	X		X	
R24											X	
R25						X	X	X	X		X	
R26						X	X	X	X		X	
R27						X	X	X	X		X	
R28						X	X	X	X		X	
R29						X	X	X	X		X	
R30					X							
R31						X	X	X	X		X	
R32						X	X	X	X		X	
R33						X	X	X	X		X	
R34												X
R35												X

Figure 2: Requirement Traceability Matrix

CHAPTER 2

Project Design

High Level and Medium Level Design

The GUI subsystem is the entry point for the user. There are GUI controls and an editor pane through which the user can interact with CITRIN. Other subsystems, such as the Interpreter and the Document Manager receive the message to do some action such as running the Interpreter on one statement, on entire source code, or opening and saving source code. The interpretation, especially in interactive mode, involves intricate display of data to the user, such as making the shadowed variable dimmed. So the interpreter collects data in the Symbol Table to help display the C++ data to the user at a later time. The document data is a repository for information, such as position of displayed text, color being used for the display, and lock condition for write access to the document.

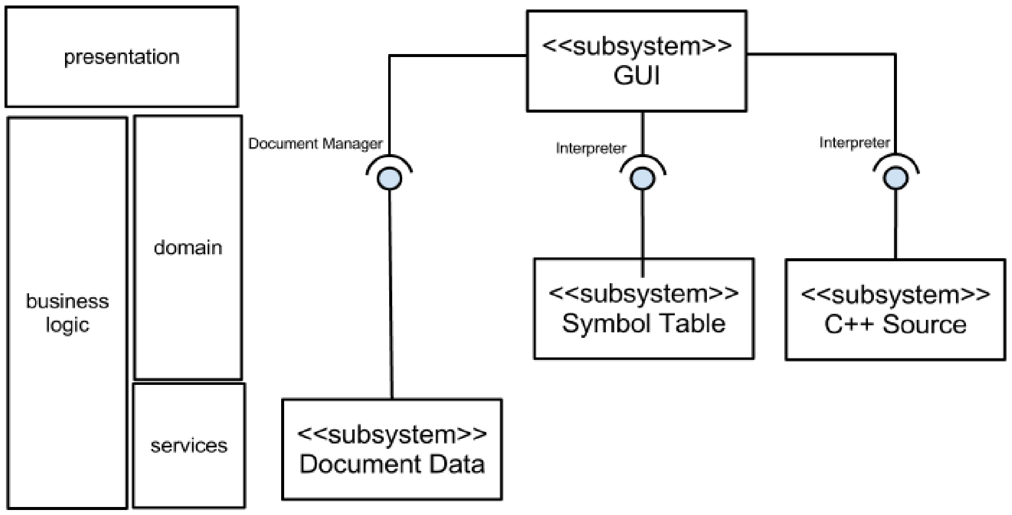


Figure 3: High Level Design Diagram

Classes

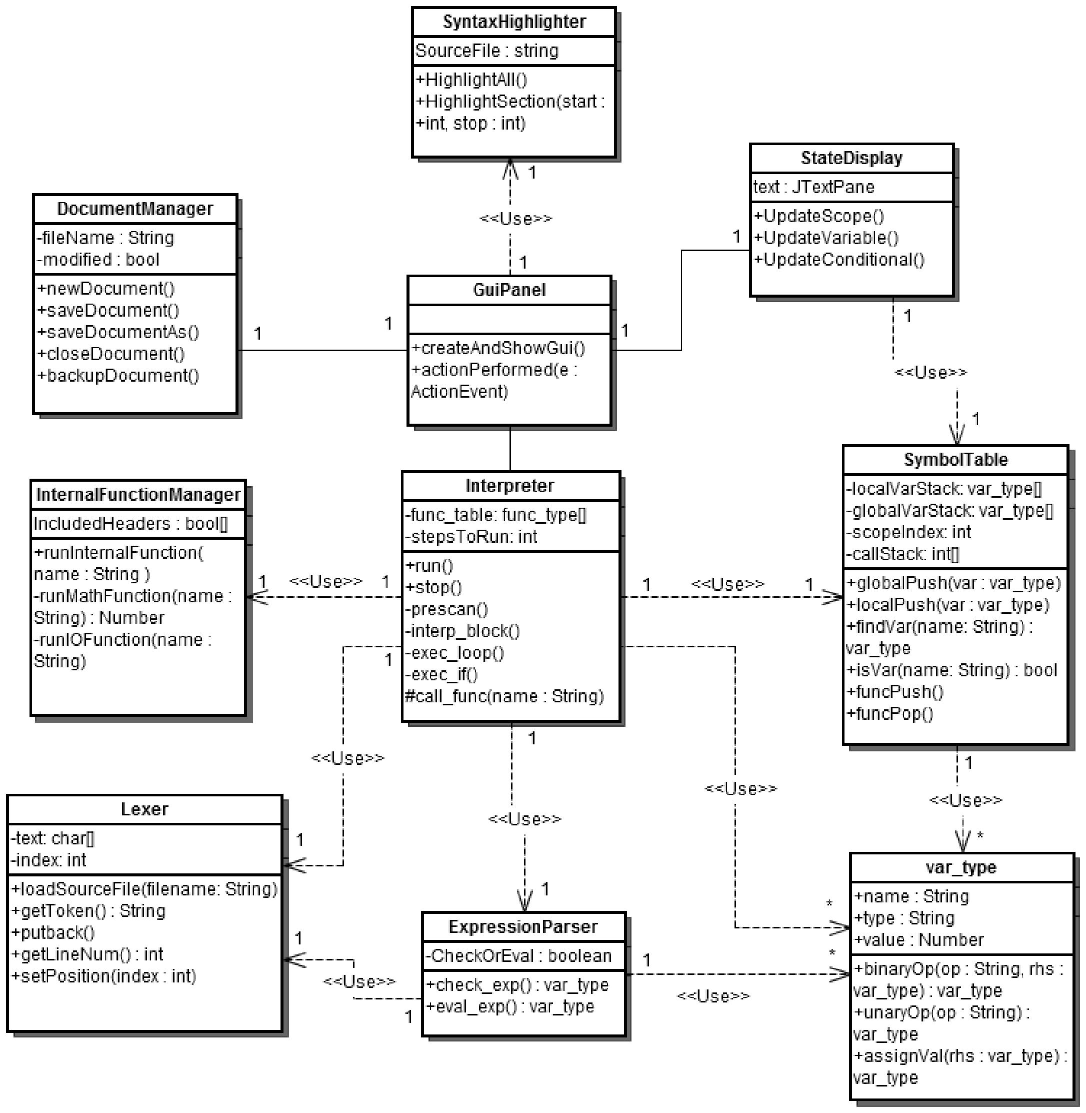


Figure 4: Class Diagram

Class: DocumentManager	This class manages creating new documents, opening documents, saving documents, closing documents, and creating duplicates.
newDocument()	Creates a new file.
openDocument()	Opens a document that the user selected and displays it in the GUI.
saveDocument()	Saves the file. If it’s the first time being saved, ask the user for a name for the file and a location in which to save it. Otherwise it will update the file with the new content.
saveDocumentAs()	Saves the file. Asks the user for a name for the file and a location in which to save it.
closeDocument()	Closes the current document being worked on.
backupDocument()	Makes a copy of the document and saves it.

Class: GUIPanel	GUI interface which handles the creation of the GUI. This class will implement ActionListener.
createAndShowGui()	Creates the panel(s) for the GUI and displays it.
actionPerformed(ActionEvent e)	Finds out what was activated and what action to take.

Class: SyntaxHighlighter	This class highlights portions of the text based on the category of the terms. It will highlight some syntax errors and make the source code more readable.
highlightAll()	Highlights the entire document. Should be called when a document is loaded.
highlightSection()	Highlights a section of the text. This should be used to avoid the overhead of highlighting the entire document when possible.

Class: StateDisplay	This class manages the display of the values of variables and conditional statements.
updateVariable()	Updates the value displayed for a variable.
updateConditional()	Updates the displayed result of a conditional
updateScope()	This function should be called when entering or leaving a scope. It updates the display to only show variables visible within the current scope.

Class: InternalFunctionManager	This class deals with internal functions in place for users.
runInternalFunction()	Determines the function being called, checks if it was included, and makes the function call

Class: Interpreter	This class implements the main structure for interpreting code. It will call functions and interpret blocks of code.
run()	Starts the interpreter running. The interpreter will use wait-notify logic to determine if it should continue running.
prescan()	This function scans the code to record the locations of user defined functions and set up global variables.
execLoop()	Checks the loops conditional statement and executes the loop as long as it is true.
interpretBlock()	Interprets a block of code recursively
execIf()	Checks the result of the conditional statement and then interprets either the “if” or “else” block of code accordingly.

Class: ExpressionParser	This class is used to analyze and evaluate expressions
eval_exp()	Evaluate an expression of of user code, returns a var_type containing the result
check_exp()	Checks an expression and returns the type of the result

Class: Lexer	This class reads in tokens and used to determine the type of token.
getToken()	Reads in the next token from a point in the source code file.
getLineNum()	Gets the line number of the current position in the source file.
setPosition()	Sets the position in the source code file to begin reading from.
isVariable()	Determines if a token is a variable.
tokenType()	This function returns the type of a token.

Class: SymbolTable	This class manages the variable stacks, scope and finding variables
localPush()	Pushes a variable onto the local variable stack
findVar()	Finds a variable within the symbol table and returns a reference to it
funcPush()	Updates the scope index to move old local variables out of scope
funcPop()	Updates the scope index, deleting local variables that were in the scope popped off

Class: VarType	This class holds a single variable
binaryOp()	Evaluates a binary operator with this variable and one other
unaryOp()	Evaluates a unary operator applied to this variable
assignVal()	Assigns a value to a variable, if it can

Data Structures:

<u>Symbol Table</u>:

C++ variables are categorized by scope level and they are accessible if they are in the current scope or the ancestor scopes. Each scope except for the global one has one and only one parent scope and 0 or more children scopes. This can be achieved easily by Tree data structure. The Scope class is implemented with an associative array with the symbol (variable) name as its key, and the Symbol Table class is implemented by a Tree of Scope class.

Each entry in the Scope class is a symbol with information needed by the parser and the interpreter: symbol name, start position in the C++ code, the length of symbol, and the data the symbol represents such as int, float, and array of primitive types.

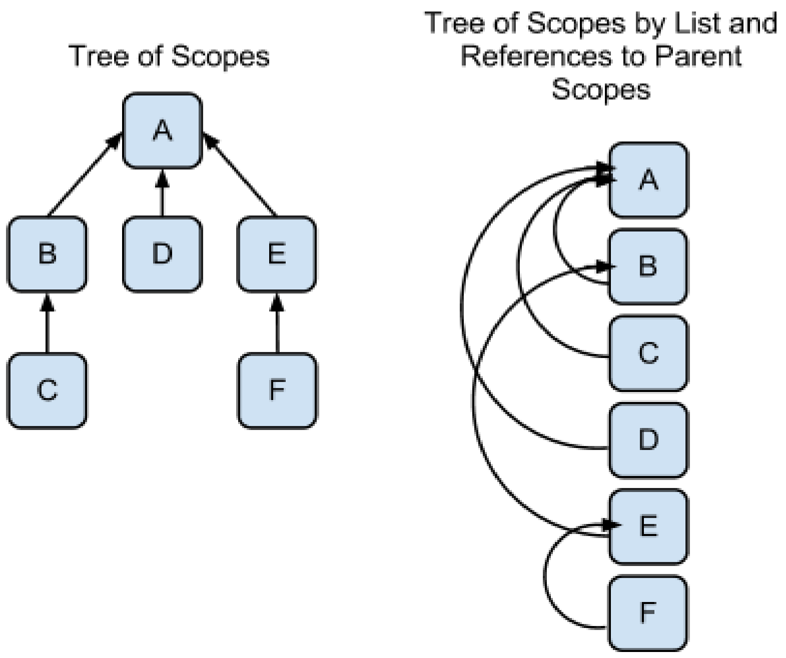


Figure 5: Example Tree of Scopes

Detailed Design:

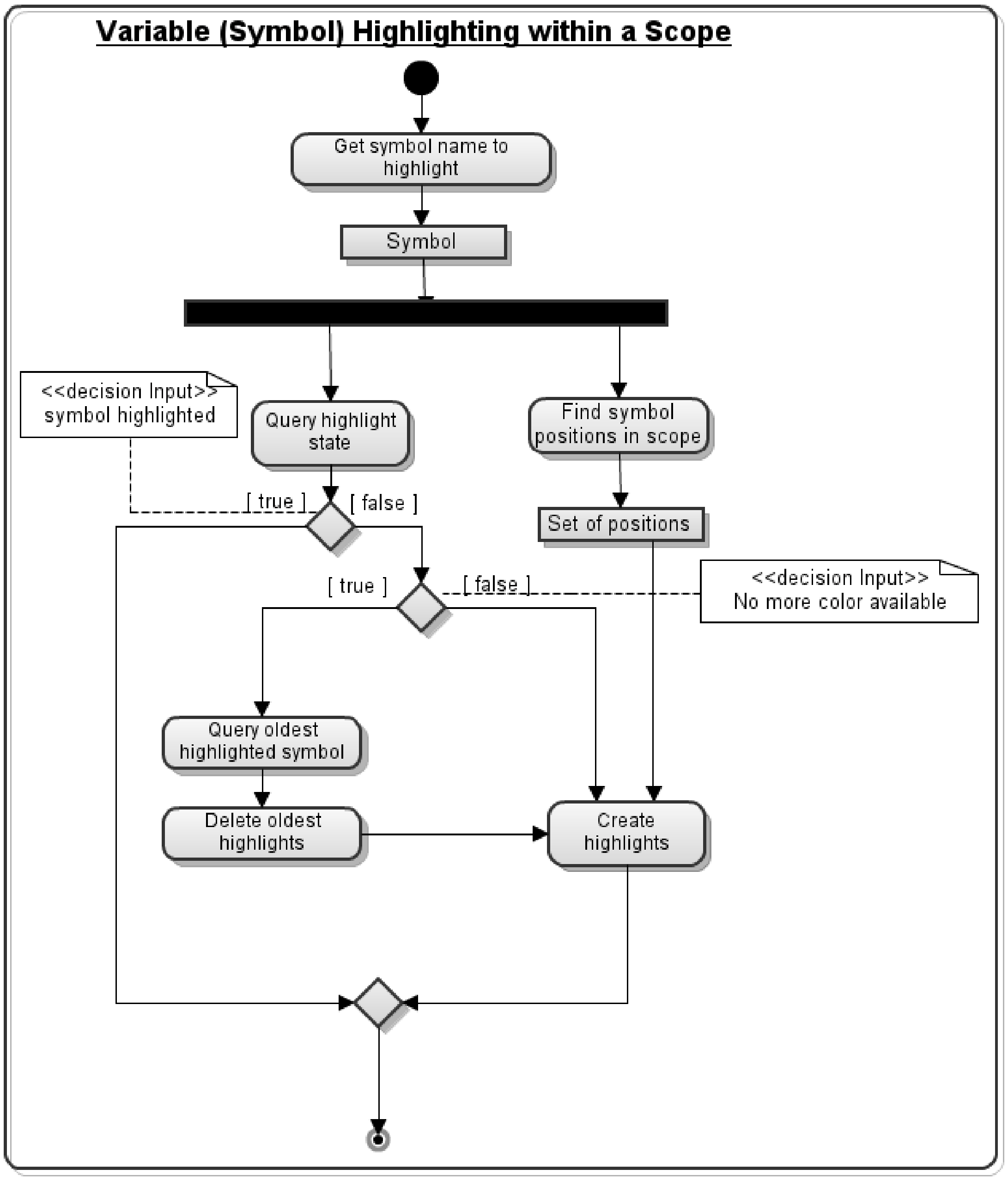

Variable (Symbol) Highlighting within a Scope
Get symbol name to highlight
Symbol
<<decision Input>> symbol highlighted
Query highlight state
Find symbol positions in scope
[true]
[false]
Set of positions
[true]
[false]
<<decision Input>> No more color available
Query oldest highlighted symbol
Delete oldest highlights
Create highlights

Figure 6: Variable Highlighting Sequence

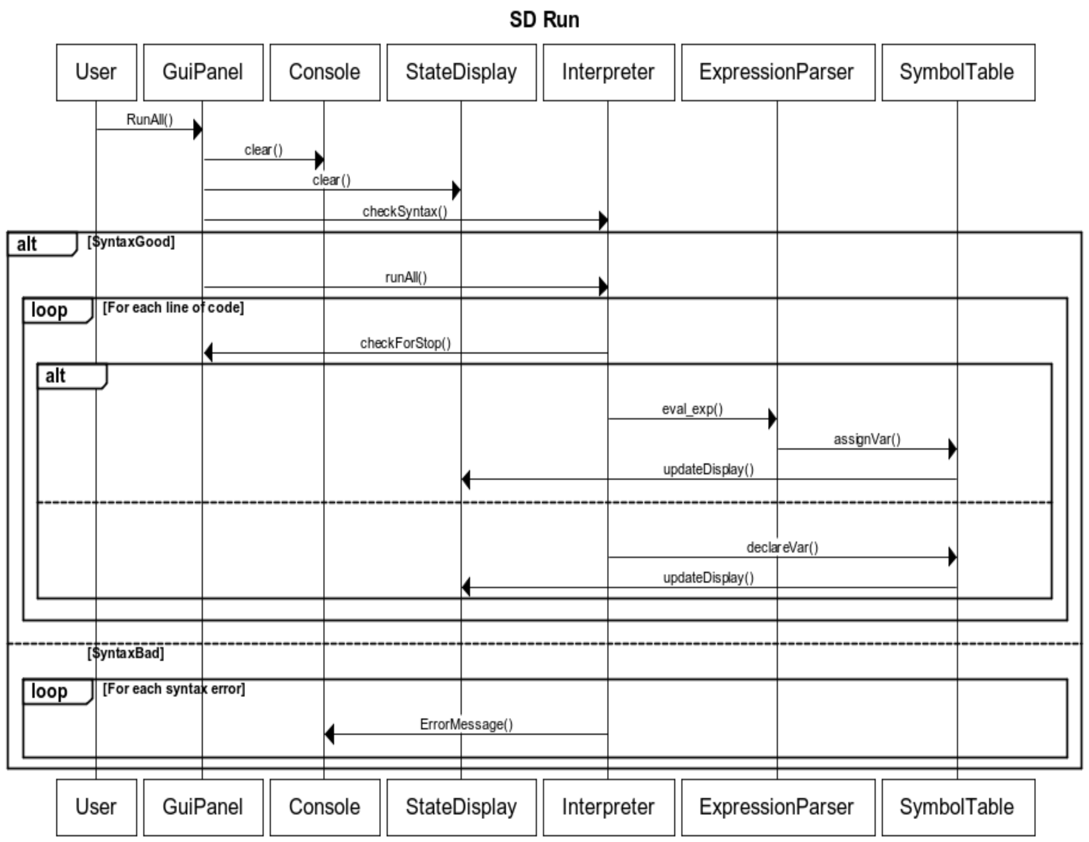


Figure 7: Sequence Diagram: Run All

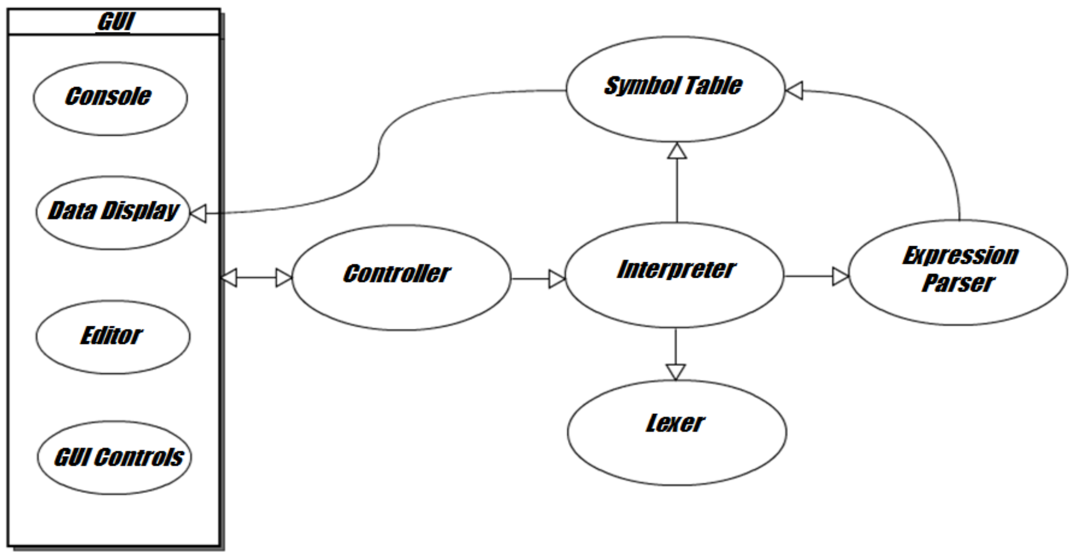


Figure 8: Architecture Design – CITRIN is modeled with a Model View Controller Pattern. The user interacts with the GUI subsystem.

Pseudocode : Interpreting One Statement

Precondition: Syntax errors are already checked and do not exist.

```
var_type Call(String)

1. Get the arguments to the function call
     1.1 Read in open parenthesis
     1.2 if no arguments, i.e. next token is ')'
          1.2.1 return
     1.3 while there are still arguments
          1.2.1 value = evaluate expression
          1.2.2 add value to argument list
2. Find matching function
     2.1 Find functions with the same name and number of params
     2.2 For each matching function
          2.2.1 If each argument is the best
               2.2.1.1 choose this function to call
     2.3 If no function chosen throw error
3. For each parameter
     3.1 push parameter onto stack
4. Set location  to start of function
5. Interpret block
6. Return return value of function
```

User Interface Design:

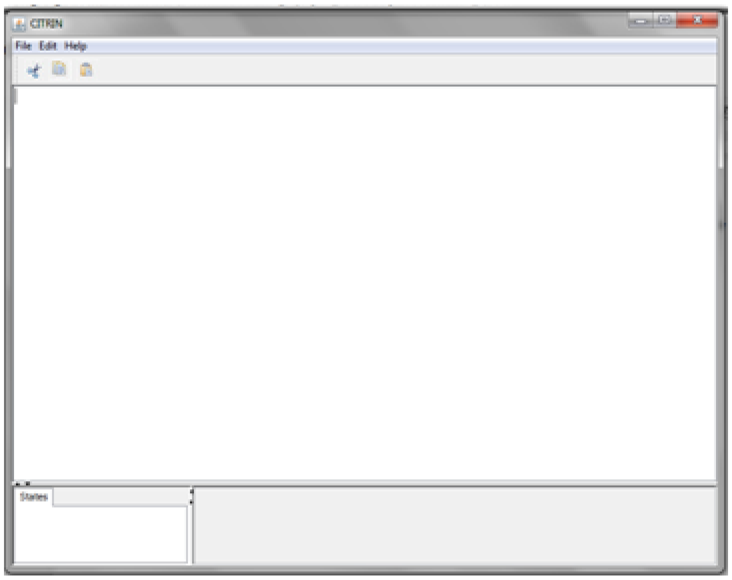


Figure 9: Main User Interface - Provides area to write code and view output and states. The menu bar allows users to open, save, and run code.

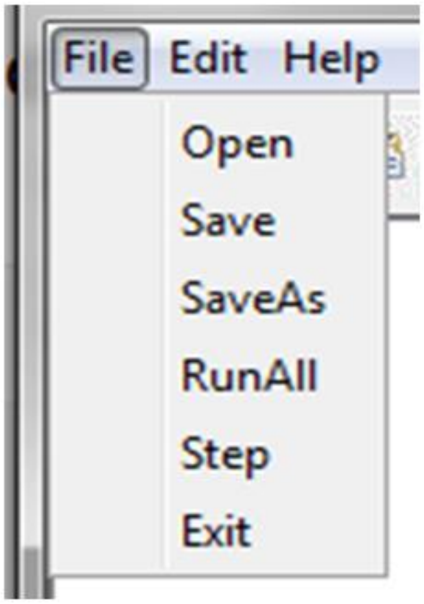


Figure 10: File Option - This menu allows users to create new code, open existing code, and save code. It also allows for the interpreter to move line by line or exit.

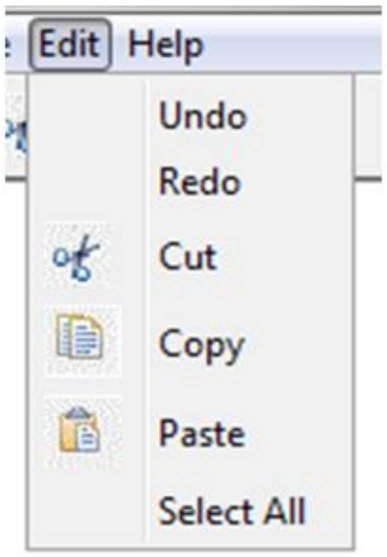
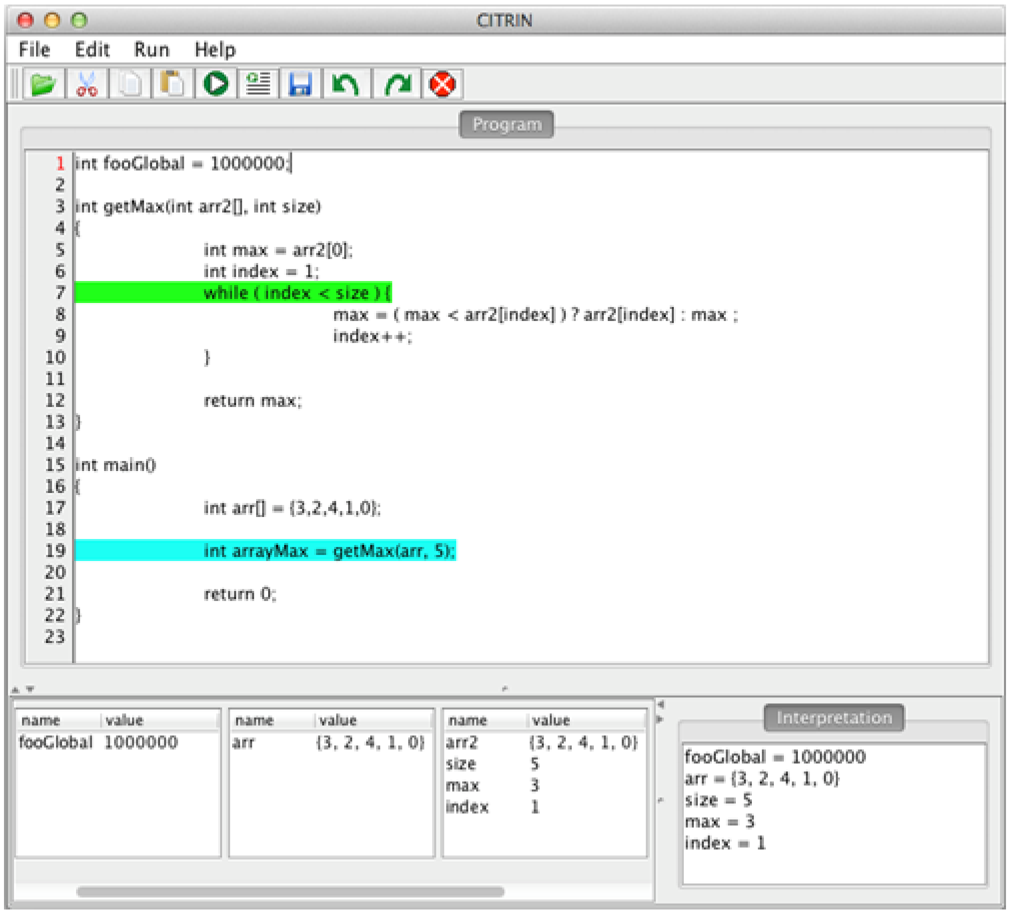


Figure 11: Edit Option - This option allows users to cut, copy, paste, delete, find, and replace text as well as actions to be undone or redone.

CHAPTER 3

Completed Work

Software Requirements Implemented



```
int fooGlobal = 1000000;

int getMax(int arr2[], int size)
{
        int max = arr2[0];
        int index = 1;
        while ( index < size ) {
                max = ( max < arr2[index] ) ? arr2[index] : max ;
                index++;
        }

        return max;
}

int main()
{
        int arr[] = {3,2,4,1,0};

        int arrayMax = getMax(arr, 5);

        return 0;
}
```

name	value
fooGlobal	1000000

name	value
arr	{3, 2, 4, 1, 0}

name	value
arr2	{3, 2, 4, 1, 0}
size	5
max	3
index	1

```
fooGlobal = 1000000
arr = {3, 2, 4, 1, 0}
size = 5
max = 3
index = 1
```

Figure 12: Screen Shot of CITRIN Running Code

Table 4: Functional Requirements Implemented

R01 [1] A user shall be able to type and edit C++ code.
R02 [1] A user shall be able to select text.
R03 [1] A user shall be able to cut, copy and paste text.
R05 [1] A user shall be able to undo and redo multiple levels of previous changes.
R06 [1] A user shall be able to save and open text files.
R07 [1] CITRIN shall provide a GUI for editing the code.
R08 [1] CITRIN shall be able to interpret for loops.
R09 [1] CITRIN shall be able to interpret C++ code and run code in a console.
R10 [1] CITRIN shall be able to interpret declarations and assignments.
R11 [1] CITRIN shall be able to interpret math operations.
R12 [1] CITRIN shall be able to interpret if, else statements.
R13 [1] CITRIN shall be able to interpret while loops.
R14 [1] CITRIN shall give simple descriptions of syntax errors.
R15 [1] CITRIN shall provide a GUI for running the code.
R16 [1] A user shall be able to step through the code line by line.

R17 [1] A user shall be able to set breakpoints in the code.
R18 [1] CITRIN shall display current values of variables.
R19 [1] CITRIN shall display the current state of boolean expressions.
R20 [1] CITRIN shall be able to interpret do while loops.
R24 [2] A user shall be able to set the speed for the code to run through line by line.
R25 [2] CITRIN will show which variables are in scope and which are inactive.
R27 [2] CITRIN shall be able to detect a change in a specified variable value.
R28 [3] CITRIN shall have a function stack for execution history.
R29 [3] CITRIN shall be able to detect if a specified variable is in the current execution statement.
R31 [3] CITRIN shall have events for callbacks.
R32 [3] CITRIN shall define simple built-in events for callbacks.
R33 [3] CITRIN shall be able to detect the change in Boolean expressions.

Table 5: Nonfunctional Requirements Implemented

T01 CITRIN shall be implemented in Java.
T02 CITRIN shall be easily extendable.
T03 CITRIN shall have a simple user interface.
T04 CITRIN shall run on Windows XP and later.
T05 CITRIN shall run on Mac OS X.
T06 CITRIN shall produce correct results.
T07 CITRIN should give warnings for undefined behavior.

Future Work

- Improvements to GUI, editor and console
- Set up communication lines of CITRIN by the observer pattern without hardcoding message from one class to another.
- Console input and output - CITRIN should interpret cin and cout statements and write that to the console.
- File input and output
- Support for classes. This would make CITRIN valuable for all of the introductory C++ courses.
- Add functionality for pointers and dynamic memory.

- Add reference operator functionality
- Add optimizations to improve the speed that code can be interpreted at.
- Interactive Mode - Interpret a program as it is being written. This would be important for students to have instant gratification as they work on coding.
- Add tutorial - one idea is that make a tutorial class which observes the interpreter events and show something graphical such as pop-up messages to show what's just happened as the result of interpretation

Software Overview

<u>CitrinInterrupter</u>

CITRIN uses threads for interpreters, but they do not exit in an orderly manner sometimes. This class keeps track of threads and collects all of them upon some mishaps.

<u>CitrinObserver/CitrinObservable</u>

CitrinObserver.java contains two interfaces CitrinObserver and CitrinObservable. Some classes cannot inherit from existing Observable/Observer Java library class since Java does not allow multiple inheritance, which is solved by using the interfaces as mixins.

<u>Console</u>

This is a subclass of JTextArea intended as an input/output console. It is currently used to for displaying the interpretation results.

Controller

This class acts as a go-between for the UI components and the interpreter components.

Editor

This is a subclass of JTextArea. It is CITRIN's text editor, where the user types his/her code.

ExpressionEvaluator

This is the expression parser. It is a custom recursive descent parser. It has two public methods eval_expr and check_expr. eval_expr evaluates all function calls and operators, while check_expr evaluates the types of an expression. check_expr will completely evaluate constant expressions, but will only evaluate the type of a non-constant expression.

DataDisplay

An observer class that is meant to listen to SymbolTableNotifier so that the changes on the C++ variables and scopes will be automatically reflected on the corresponding GUI panel. This keeps variables scope by scope.

GuiPanel

This is a facade class that creates all the GUI components as well as CITRIN internal classes upon the startup of CITRIN. Many callback actions for GUI events are also defined in class.

Interpreter

This class interprets the code by reading in a token and deciding what action to take. It includes the ability to scan the code for syntax errors which reuses as many of the interpretation modules as possible. It should be run in a new thread and its interfaces are thread safe. It should be refactored into smaller modules.

Keyword

An enumeration of keywords and variable types.

Lexer

Class for reading in tokens from the input file. Skips whitespace and comments, while tracking the line number.

StopException

Exception used when the execution of the interpreter is terminated during a run.

SymbolTable

This acts as a C++ variable repository. Offers public interface to manipulate scopes and store symbols in order to ease the task of Interpreter and to provide other components of the CITRIN with access to the variables according to the current scope.

SymbolTableNotifier

This is a subclass of SymbolTable which acts an observable and publishes SymbolTableEvent such as ScopePoppped/SymbolInserted.

SyntaxError

An exception that is thrown when a syntax error is encountered.

TextLineNumber

Provides line numbering for the text editor. It highlights the line number of the current line.

Token_type

An enumeration of the types of tokens.

Var_type

A class that packages variables, and information about them. Contains methods for primitive operators and type checking.

Code Contribution

Yuta Matsumoto

SymbolTable, SymbolTableNotifier and their drivers

DataDisplay and its driver

CitrinObserver / CitrinObservable

CitrinInterrupter

GuiPanel

Automated Test of Interpreter on pre-written C++ files

Some Integration Work

Chris Salls

Most of Interpreter class

ExpressionEvaluator

Lexer

Threading

Integration

Jessica Hall

GuiPanel

RunMultipleSteps

TimedRun

Line highlighting

Undo/Redo

Shaun Davidson

GuiPanel

RunBreakPoint

TextLineNumber integration

Some Interpreter work

Integration

Editor

Resources

CITRIN Website: https://sites.google.com/site/citrinproject/

CITRIN on Github: https://github.com/citrin2013/citrin

CITRIN Introduction Video: http://www.youtube.com/watch?v=-IaTiksZOBY

Annotated References

Bergin, Joseph. "Graphical User Interface Programming for Multi-Platform Applications in Java 2. or GUI Programming in Java for Everyone (Version 3)." Pace University. September 26, 2001.

This article provides information on programming a graphical user interface (GUI) in Java. The article goes over basic GUI development. Bergin does this by showing the steps necessary to create a Java application. The article provides examples of code using the Java Foundation Classes (JFC) and Swing.

Bolton, David. "About Compilers and Interpreters." Retrieved October 25, 2012, from http://cplus.about.com/od/introductiontoprogramming/a/compinterp.htm

This article describes the differences between compilers and interpreters. It discusses the advantages and disadvantages of each and which languages work well with them. Interpreting is described as editing, debugging, or running the program. It is said to be a far faster process and helps novice programmers edit and test their code. However, the program itself will run much slower than a compiler since each line of code must be re-read and processed.

Campbell, Matt. "Ch - A C/C++ Interpreter." Retrieved October 29, 2012, from www.mactech.com/articles/mactech/Vol.19/19.09/CInterpreter/index.html

This article discusses CH a C/C++ Interpreter. It says that CH is a simple interpreter that allows for fast execution of C++ code. All that is necessary is the program file. It can interpret on two levels. It can interpret code on the fly as it is written, or several lines can be run at once. The interpreter is also shown to be expandable because it supports new standards and is able to execute many functions.

Ertl, M. “The Structure and Performance of Efficient Interpreters” Journal of Instruction Level Parallelism. Volume 5. November, 2003.

This paper describes some techniques for creating an efficient interpreter and provides some comparisons of techniques and existing interpreters. The beginning of the paper is an introduction that covers why efficient interpreters are important and some comparisons with native code compilers. It covers using branch prediction to improve performance in the instruction flow pipeline rather than waiting for the result of the branch to be known. Another relevant section of the paper is the part titled “Interpreter Writers” where the author gives some closing advice for anyone writing an interpreter.

Hanson, David “Compact recursive-descent parsing of expressions” Journal of Software: Practice and Experience. Volume 15, Issue 12, pages 1205-1212. October 30, 2006.

This article covers a newer more compact method of recursive descent parsing using a table. The introduction briefly talks about the use recursive descent parsing when writing a compiler by hand, instead of using a parser generator. Recursive descent parses operator precedence with a series of recursive calls. Usually n+1 procedures are used with n levels. However, this article describes how to implement a parser using only 2 procedures, using a table instead. This may be a cleaner way to implement our parser and make it easily extensible.

Naumann, Axel “Introducing Cling.” Retrieved November 3, 2012 from root.cern.ch/drupal/sites/default/files/AxelNaumann-cling-GoogleTech.pdf

This presentation at a Google Tech Talk, describes the Cling C++ interpreter. Cling is a very complete, command line based interpreter, which uses Clang as a backend. The presentation describes the use of interpreters over time. Where to get it and what it supports. Naumann goes on to describe the design of the interpreter, including the wrapping of expressions and determination of declarations or expressions. He also describes how the just in time compiler handles dynamic scopes.

Schildt, Herbert. *C: The Complete Reference* 2000. McGraw Hill.

Schildt discusses how to create your own C interpreter. It goes through the basics of dealing with token categories and describes what features need to be provided for an interpreter. The use of library functions is explained and there is a list of methods for improving the overall speed and usability of an interpreter. These methods can be extended to create a C++ interpreter.

Wu. C. Thomas. An Introduction to Object-Oriented Programming with Java. McGraw Hill 2006.

This book is an introduction to programming in Java and gives the basic concepts behind Java. This book has the basic concepts behind Java and it has sections on how to program GUIs and the different components and layouts that are involved in writing a GUI in Java. This reference book for Java will be helpful because the project is being written in Java. The information about how to create GUIs is especially helpful as a GUI has been created for this program and this book provides good information on how to create a GUI and the components that make up a GUI.

Yu-Cheng Chou, Stephen S. Nestinger, Harry H. Cheng. “Ch MPI: Interpretive Parallel Computing in C.” IEEE Computing in Science and Engineering. Vol. 12, No. 2, March/April 2010, pp. 54-67.

This publication discusses the advantages to using the CH MPI as an interpreter to allow users to work in a C based scripting environment. The article discusses how this advantage would allow developers to create fast working prototypes of code without having to worry about compiling and recompiling after making modifications. This allows for users to find and deal with errors on a much quicker basis.

Glossary of Terms

Abstract Syntax Tree (AST): Tree representation of the abstract syntactic structure of source code.

Boolean: A data type representing a value of either true or false.

Breakpoint: A point in a program that, when reached, triggers some special behavior useful to the process of debugging. They are used to pause program execution or dumb the values of some or all program variables. Breakpoints may be part of the program itself or they may be set by the programmer as part of an interactive session with a debugging tool for scrutinizing the program's execution.

Callback is a piece of executable code that is passed as an argument to some other code. The other code is expected to execute the argument at some time.
This is achieved by "Observer Pattern" in OOP with a dependency relationship instead of passing a pointer to a function.

Call Stack: a stack data structure that stores information about the active subroutines of a computer program. This kind of stack is also known as an execution stack, control stack, run-time stack, or machine stack, and is often shortened to just "the stack". A call stack is used for several related purposes, but the main reason for having one is to keep track of the point to which each active subroutine should return control when it finishes executing

Compiler: A computer program by which a high-level programming language is converted into machine language that can be acted upon by a computer.

Console: A window or frame where text can be entered and displayed. Simple C++ programs only interact with the user via text input and output through the console.

Expression in a programming language is a combination of explicit values, constants, variables, operators, and functions that are interpreted according to the particular rules of precedence and of association for a particular programming language, which computes and then produces (*returns*, in a stateful environment) another value.

Graphical User Interface: A human-computer interface that uses windows, icons and menus that can be manipulated by a mouse and keyboard.

Hotkey: Also known as a keyboard shortcut. By pressing a set of keys simultaneously, the user can trigger an operation within the program.

Integrated Development Environment: A set of programs run from a single user interface.

Interpreter: Hardware, or software that transforms one statement at a time of a program written in a high-level language into a sequence of machine actions and executes the statement immediately before going on to transform the next statement.

Ifstream: In this project, ifstream is what the students will see as data is being read from a file into their programs and it will either be good or bad depending on the state of the file.

Input: The data that the user inputs into the program using methods such as the keyboard and console, or an ifstream from a file.

Iostream: In this project, the iostream that is seen on the screen will mean that the student is writing data to a file and depending on the state of the file, the iostream will either be good or bad.

Observer Pattern is a software design pattern in which an object maintains a list of its observers, and notifies them automatically of any state changes.
Java Swing library uses this pattern, in which the interface observers implement is called ActionListener, and the subjects are GUI controls such as GUI buttons. This is the link to the Java tutorial of ActionListener :

Output: Output does not show the state that the code is in instead it shows the data that the user's program outputs to the console.

Parser: A parser analyzes a sequence of tokens to determine the grammatical structure of a target language.

Recursive Descent Parser is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input. One procedure is associated with each nonterminal of a grammar. The sequence of procedure calls during the analysis of an input string implicitly defines a parse tree for the input, and can be used to build an explicit parse tree, if desired.

Scanner: A program or function which converts a sequence of characters into a sequence of tokens.

Scope: The context within source code in which a variable or function name is valid.

State: The state of the boolean expression or variable. A boolean expression can either be in a true or false state. The state of a variable is whatever is in the variable at the time whether it is garbage or an assigned value.

State of the file: The state of the file will be good meaning that data can now be read from it or written to the file. Or the state will be bad meaning data cannot be read from the file or written to it. The state of the file can depend on if the file exists, is empty, or if the file is open or closed.

Statement is the smallest standalone element of an imperative programming language. A program written in such a language is formed by a sequence of one or more statements. A statement will have internal components (e.g., expressions).

Strategy Pattern is a software design pattern in which strategies are composed of in another object called context without using inheritance but strategies satisfy the same interface so that context composing a strategy can switch to another strategy at run-time by changing what it composes of.

Symbol Table: is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location.

Syntax: The grammatical rules and structural patterns governing the ordered use of appropriate words and symbols for issuing commands or writing code in a particular software application or programming language.

Token: A string of characters that will be taken by a scanner as a unit.

Top-down Parsing is a parsing strategy where one first looks at the highest level of the parse tree and works down the parse tree by using the rewriting rules of a formal grammar. Also see recursive descent parsing above.

Tutorial: A set of instructions given to a user that deals with the use of a software package.

Variable: A named memory location in a program in which data can be stored and read from.

Variable Shadowing occurs when a variable within a certain scope has the same name as a variable declared in an outer scope.