



THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC 3002

Introduction to Computer Science: Programming Paradigms

---

## Project Report: Jiu Zhang IDE

---

刘泉	118010187
刘润	118010188
舒欣	118020362
卓然	118010475
刘芷萱	118010202

May 31<sup>st</sup>, 2020

## Contents

1 Introduction	- 3 -
2 Related Work	- 3 -
3 Our Work	- 4 -
3.0 Superiority of our project	- 4 -
3.1 Editor and GUI Design	- 5 -
3.1.1 Code Editor	- 6 -
3.1.2 Output Window	- 6 -
3.1.3 Input Field	- 6 -
3.1.4 Menu Bar and Text Rendering	- 7 -
3.1.5 Project Tree (File Tree)	- 7 -
3.1.6 Run Button and Code Integration	- 8 -
3.2 Compiler	- 9 -
3.2.1 Analysis part	- 9 -
3.2.2 Convert part	- 9 -
3.2.3 Results Display	- 10 -
3.3 MIPS Assembler	- 10 -
3.4 Linker	- 12 -
3.4.1 Function	- 12 -
3.4.2 Some changes from proposal	- 12 -
3.4.3 Code implementation	- 12 -
3.4.4 Demo	- 13 -
3.5 MIPS Simulator	- 13 -
4 Contribution Evaluation	- 14 -
5 Reflections	- 15 -
5.1 Quan Liu	- 15 -
5.2 Run Liu	- 16 -
5.3 Zhixuan Liu	- 16 -

5.4 <i>Ran Zhuo</i>	- 18 -
5.5 <i>Xin Shu</i>	- 20 -
6 Appendix	- 21 -
6.1 <i>Compiler demo</i>	- 21 -
6.2 <i>Linker demo</i>	- 24 -

# 1 Introduction

We choose to design and implement the classical Chinese programming (文言文编程). Inspired by “Nine Chapters on Arithmetics” (九章算术) and the wenyan-lang on github, we designed a coding language called Jiu Zhang, which is written in traditional Chinese, and wrote some sample files to solve some classical mathematical problems, such as “Chicken and Rabbit Cage Problem” (鸡兔同笼问题). With that being said, we include the editor, compiler, assembler, linker and simulator in our final outcome, and integrate these parts into a Jiu Zhang (九章) GUI for display.

There are several reasons we choose to do classical Chinese programming. First, it allows us to use the natural language grammar to write codes, which is quite amazing and convenient. Second, writing codes in Chinese allows us to break the routine that codes must be written in English.

There are several changes from the proposal in our final implementation. First, we did not build a debugger since the person who was assigned to write it has quit the course in the midway. Second, we have made some changes in the language syntax part, which will be further explained in the compiler section.

## 2 Related Work

As we sought for inspirations, a story of classical Chinese programming aroused our interest. It leads us to a GitHub project released in 2019, which is completed by a senior Computer Science student in Carnegie Mellon University. His project introduces the idea of programming in classical Chinese and a new programming language designed by himself under this thought. Although it is defective and not complete enough, we still find it constructive and meaningful. In brief, that is where we first got our ideas from.

Wenyan:

```
吾有一數。曰三。名之曰「甲」。  
為是「甲」遍。  
吾有一言。曰「問天地好在。」。書之。  
云云。
```

Equivalent JavaScript:

```
var n = 3;  
for (var i = 0; i < n; i++) {  
    console.log("問天地好在。");  
}
```

Figure 1. “Hello World” Written in Wenyan-lang GitHub Project

Then we dug into Chinese mathematical classics like “Nine Chapters on Arithmetics” (九章算术) and “Zhou Bi Suan Jing” (周髀算经), and found the languages they use to describe math problems are concise and beautiful. Therefore, we decided to refer to these classics when designing our own programming language.

In the process of realizing our project, we mainly obtained the necessary external resources through the following three channels. Firstly, we imported Zhixuan Liu’s work of MIPS assembler and simulator in the course CSC3050, and adding new features like error handling to them. Secondly, we went to the internet and read the codes from programmers’ communities like GitHub and CSDN to get some ideas for reference. Thirdly, we also learned a lot from textbooks like *Fundamentals of Compiling* and *C++ GUI Programming with Qt 4*.

## 3 Our Work

### *3.0 Superiority of our project*

Our project is the world's first IDE that transforms the profound ancient Chinese classic "Nine Chapters on Arithmetics" (九章算术) into a modern programming language (named Jiu Zhang). It supports coding with classical Chinese, and thus breaks the conventional English style.

As an original readable Chinese programming language, Jiu Zhang is easy to understand and manage even for people without much programming knowledge. Using our language, users can not only experience the refined beauty of Chinese during the programming process, but also enjoy the user-friendly grammar. Moreover, our IDE allows users to interact with their code in a Graphic User Interface instead of the console window.

#### **Some function highlights:**

- Support complex statement: nested "while", "if" statement; mixed "if" "while" statement; nested function; recursion function
- Multi file compilation
- Display the number line in real time
- Highlight syntax automatically
- Research key words

- Adjustable font
- Reports errors

### 3.1 Editor and GUI Design

Once our project compiled, an editor interface will be displayed. This Graphic User Interface with various functions is an integration of all our efforts. It conforms to the usual pattern of a code editor and has a clean design, which makes it quite user-friendly.

In this editor, we allow users to create their own projects and perform coding using classical Chinese (文言文) based on the grammar we declared. They can interact with our editor through the input and output windows, finding their bugs and get the results of their programs.

Basically, our editor interface is composed of six parts, which are the Code Editor, the Output Window, and Input Field on the right; the Menu Bar above; and the Project Tree (or say File Tree) and the Run Button on the left. This report will discuss each part in turn, and a sample image of the editor is shown below.

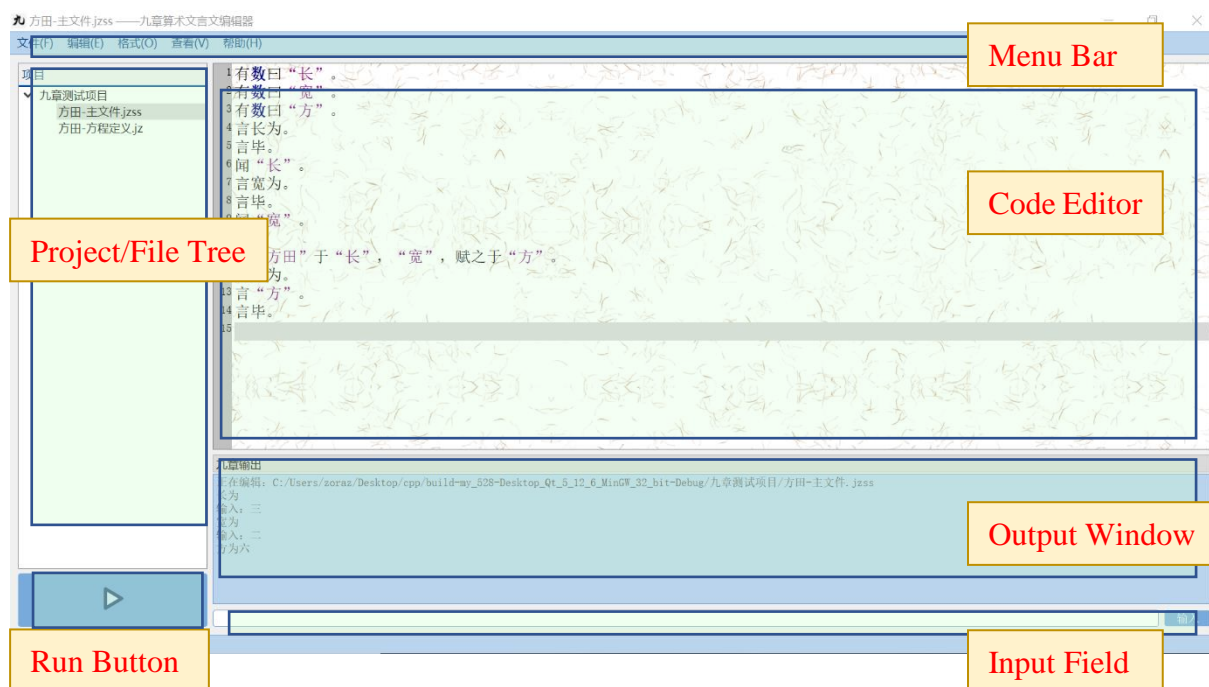


Figure 2. Display Pattern of Jiu Zhang Editor

### 3.1.1 Code Editor

**Text editing area:** We define a class called CodeEditor, which inherit the QPlainTextEdit class provided by Qt Creator. Based on the QPlainTextEdit class, the text is displayed in WYSIWYG format and supports basic operations like cut, copy, and paste.

Through using the new class, we display line numbers on the left and update the width of the area of line number as the number of digits change. Also, we highlight the current line as grey.

**Highlighter:** We used the highlighter example given by Qt for reference, and designed our own highlight mode of Jiu Zhang code. To be precise, the data types (数 for integer, 判 for bool) and function definition (法 for function) is set bold in dark blue. The variables defined by users is dark magenta. The loop identifiers (当, 乃继, 乃离, 乃止) and conditional statements (若, 否) are blue italic, while the comments (注: ……。) are green.

The highlight rules are described by regular expressions.

### 3.1.2 Output Window

**Output display:** This area is defined as a QTextEdit and set to be read-only. Each time the user chooses to load a certain file, the output window will be cleared and show the absolute path of the current file. It also shows the output information of the compiled project (characters after “言”). Besides, when the program asks for an input, this window will print “输入: ” and wait for users to click on the input button.

**Bug display:** If the project written by the user fails to run, certain types of errors will be displayed in this area, so that users can modify their codes accordingly. The method of catching error messages will be further discussed in later parts.

### 3.1.3 Input Field

The input field is made up of a QPushButton and a QLineEdit widget. Only when the simulator asks for inputs, the input button will be enabled and the contents in the line editor will then be passed to the simulator.

### 3.1.4 Menu Bar and Text Rendering

The menu bar is created using classes including QMenuBar, QMenu, QAction, QDialog, QHBoxLayout, and some other widgets. There are five menus in the menu bar, which are the file menu, the editor menu, the format menu, the view menu, and the help menu. They have realized various functions like creating a new file, opening or saving a project, setting the font of code editor, finding a certain string, and so on. When implementing the menu bar, we used the text editor example given by Qt for reference.

In the view menu, there is an option called “渲染” (render in English). When the user clicks on this option, it will generate a window displaying your current code in a beautiful pattern. We use QLabel to show the rendered version of codes, which are turned into classical Chinese with a font called “隶书”. The background is set to be a picture of rice paper.

### 3.1.5 Project Tree (File Tree)

**Tree layout:** The layout of the project and file names is implemented based on QTreeWidget. We set the widget to accept focus by clicking, and connect another function to respond to double-click. If the user double-clicks the tree item of a project, all the Jiu Zhang files under this project will be loaded and added to the tree layout. If the user double-clicks a file, the contents of this file will be loaded and displayed in the code editor, while the output contents as well as the window title of the whole GUI will also be changed according to the name of the current file.

Meanwhile, if users right click on the project tree, a menu consisting of various actions will be displayed. Through the linked menus, users can add new projects or files, delete or rename the chosen project and file, close or run the chosen project.



Figure 3. Respective Tree Menus of Projects and Files



To be specific, when the user wants to create a new project, the GUI will first pop up a dialog box and ask him or her to choose a folder to put the new project in. Then, it will pop up an input dialog box to ask for the name of the project. After that, a new project will be created under the tree as well as in the user's computer. Besides, adding new file only requires a file name. Moreover, when the user clicks on the Close or Delete option, a message box will pop and ask for confirm. The run option in the menu performs the same function as the run button below the tree, which will be detailly discussed in the next part (3.1.6).

**File operations:** We get the project directory by a file dialog which allows user to choose a folder. Then we store the file path as data in the tree item of a project, and ask for the data when we want to operate files using the file path.

### *3.1.6 Run Button and Code Integration*

Once the run button is clicked, the current project written by the user will start compiling. The running process is an integration of all the code contributed by our group members, and it can be divided into 9 steps.

- 1) Check whether the current project is saved, and pop up a message box as reminder if not.
- 2) Automatically find the path of the project, despite whether the current focus is a project or a file, and accordingly find all the Jiu Zhang files under this project.
- 3) Create a temporary work folder under the project folder for storage of all the intermediate files generated.
- 4) Load the absolute paths and contents of all the Jiu Zhang files under this project, and put the two kinds of information separately into two vectors. The two vectors respectively store the file path as a std string and the contents as a std wstring. Put the file with suffix “.jzss” at the beginning of two vectors.
- 5) Compile the Jiu Zhang code in wstrings and generate a corresponding MIPS (.asm) file in the temporary work folder for each subfile.
- 6) Assemble the MIPS files to generate “.o” files written in machine code.
- 7) Link the “.o” files together by solving unsolved labels and generate an integrated “.jexe” file written in machine code. (Besides, a “.trans” file is generated as a byproduct.)
- 8) Simulate the operation mode of MIPS language and execute the program, interacting with the user if required.
- 9) Throw and catch errors in the running process, and print them out in the output window.

## ***3.2 Compiler***

The compiler is designed to convert the Jiu Zhang language file .jzss/.jz into assembly language file .asm. The Compiler consists of two parts:

### ***3.2.1 Analysis part***

The raw code is first processed by lexical analyzer and syntax analyzer to produce a whole syntax tree representing the basic syntax structure of the code.

In the lexical analysis part, the idea of finite automata is used. The code is read character by character, and meaningful character groups are identified to be tokens. Variable names are replaced by unique IDs. A token stream is created in this part.

In the syntax analysis part, the syntax analyzer receives the token stream and separates the stream into single statements according to the "sep" token. By identifying the feature token, like "if" and "while", the statement is processed accordingly and generates the corresponding syntax node. Nodes are connected with each other logically. In the end, the analyzer returns the root node of the syntax tree to the next part.

### ***3.2.2 Convert part***

In the convert part, each root node received from syntax analyzer is converted to one output file in MIPS assembly language. To achieve this, the program are designed with three function: traversal, conversion and generating header.

The syntax tree firstly get into a recursion function to achieve traversal. The traversal is in both preorder and post order, which is determined by the type of node operated. For instance, "if" node do operations both before and after recursions.

When nodes are processed to generate MIPS code, variable' IDs are replaced by memory addresses. Variables in function are replaced by the memory offset in order to support nested function. In addition, the judge and loop statement are assigned unique IDs to achieve nested statement.

To facilitate multi files compilation, in the linkerHeader, the function defined and function undefined with line number are shown.

```
1  .linkerHeader
2
3  func_defined:
4  func3
5  func7
6
7  func_undefined:
8  func4@22
9
10 var_defined:
```

Figure 4. Display “.linkerHeader”

The Compiler also has the error handling ability:

- 错误：xx行：未识别的字符。 (Unknown character)
- 错误：xx行：缺少”。 (Lacks ")
- 错误：xx行：“ 使用错误。 (Wrong ")
- 错误：xx行：语法错误。 (Wrong Syntax)
- 错误：xx行：重复定义变量。 (Repeat Definition)
- 错误：xx行：没有if的else。 (Else lacks if)
- 错误：xx行：函数外返回。 (Return outside function definition)
- 错误：xx行：定义函数语法错误。 (Wrong function definition)
- 错误：文件缺乏end。 (Lacks end)
- 错误：使用未定义变量。 (Unknown Variable)

### 3.2.3 Results Display

We have provided a demo of a project with three .asm files. That project shows how to call functions in library file. The main function in the first file call function3 in the second file (library file). The function3 call the function7 in the third file (library file).

The figures are attached in Appendix part for your reference.

## 3.3 MIPS Assembler

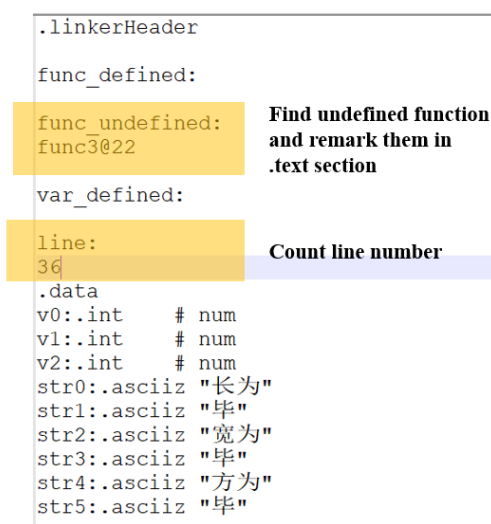
In this part, MIPS assembler receives a vector from compiler, with all the MIPS code file

path in it. The main work MIPS assembler does is to convert the MIPS codes to binary codes, and delivers them to linker.

Assembler will traverse all the MIPS file that compiler gives. For each file:

Firstly, assembler catches all the undefined functions announced in the .linkerHeader section. Then scans through the .text section, finds all the function labels and matches them with their address.

Secondly, creates a “.o” file and remains the .linkerHeader section unchanged. Counts the line number of .text section and puts it in .linkerHeader section.



```
.linkerHeader
func_defined:
func_undefined:
func3@22
var_defined:
line:
36
.data
v0:.int    # num
v1:.int    # num
v2:.int    # num
str0:.asciiz "长为"
str1:.asciiz "毕"
str2:.asciiz "宽为"
str3:.asciiz "毕"
str4:.asciiz "方为"
str5:.asciiz "毕"
```

Figure 5. Display “.o” file

Thirdly, covert the MIPS code to machine code line by line. To do so, each MIPS line is split into elements and stored in a vector called lineElements, together with their line address for converting the branch and jump instructions. Assembler uses many maps to detect each instructions and output the machine code in the corresponding format in the .o file.

What’s more, assembler keeps the function labels and undefined functions for linker.

[illegible]

Figure 6. Display “.o” files Before Being Linked

### 3.4 Linker

### 3.4.1 Function

The linker's function is to link the functions that are undefined in the main file with library files by code reallocation and address calculation, which are either written by the users or provided by us.

### 3.4.2 Some changes from proposal

We decide not to export the constants or static variables from the library such as `pi`, natural log, since we could let the users to input these numbers by themselves.

### 3.4.3 Code implementation

First, we get a vector with file addresses, with the first being main file and others being the library files. We open the main file, whose suffix is `.o`, and then copy the `.data` and `.text` section of main file in a file with the suffix `.trans`, which means that it is a midway-generated file, and calculate the number of codes in the main file, which is `mainLine`.

Second, we open every library file using for loop, copy the function codes in it and remove the if: and endif: labels since these labels are of no use. At the same time, we keep a

counting variable count to calculate the current address, so that we could know each function's address in the integrated file.

Finally, we replace the undefined function labels in the machine code with corresponding address and remove the function labels.

#### 3.4.4 Demo

We have provided several demos of the running results of linker. The func3 is undefined in main.asm file, but is defined in libraryFile1.asm, however, during func3's definition it needs func7, which is defined in libraryFile2.asm, so linkedFile.jexe is the result of linking these three files.

They are attached in Appendix part for your reference.

### 3.5 MIPS Simulator

In this part, simulator execute the machine code from linker and interacts with users in Editor GUI. Simulator can be mainly divided into three parts.

1) Memory malloc, initial register, and data stored.

For each project, simulator first malloc a 6MB memory (which will be released when the project finished executing). This malloced memory imitates the real memory storage (figure number). Simulator also initializes a general register. Then, simulator puts the data in .data section into data segment in malloced memory and puts machine codes in .text section into text segment.

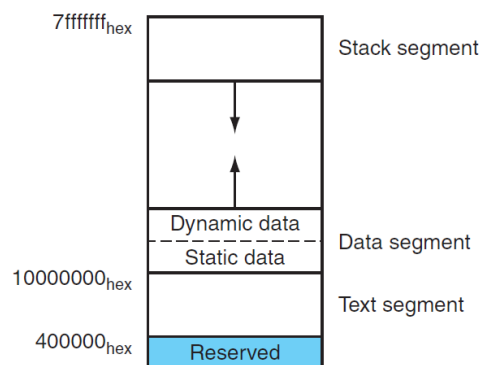


Figure 7. Memory Framework

## 2) Machine cycle

Simulator initializes a pointer, call pc. It enables simulator execute machine codes line by line as well as branch and jump to the target machine code. Different types of machine codes correspond to different processing functions.

## 3) Syscall and Editor GUI connections

Jiu Zhang supports Chinese strings input and output. Therefore, simulator turns Chinses mathematical string input in to corresponding integer. Simulator also convers integer to Chinses characters and displays them in GUI. As for output Chinses string, simulator transfers the coding format of Chinses characters for them to display in GUI interface.

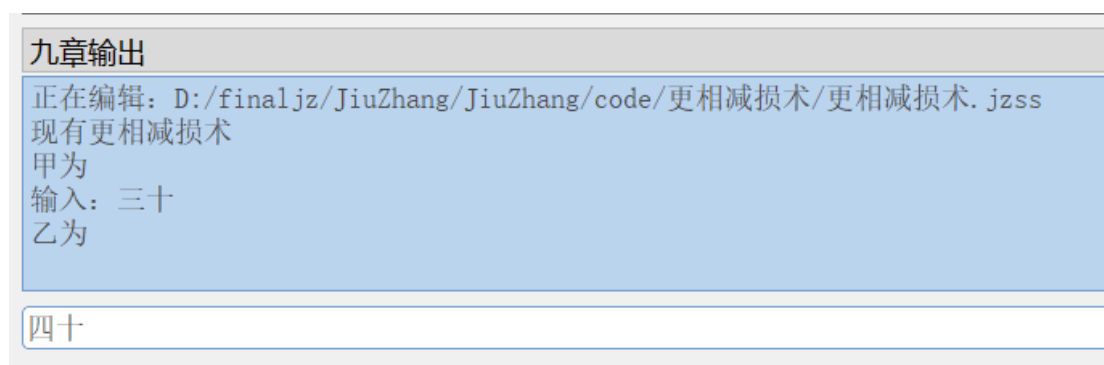


Figure 8. Output and Input Fields of GUI

# 4 Contribution Evaluation

There are five members of our team. We are

刘泉	118010187	Team representative
刘润	118010188	Team leader
舒欣	118020362	
卓然	118010475	
刘芷萱	118010202	

The table below shows our actual contributions:

Name	Percentage	Contributions
刘泉	21%	<ol style="list-style-type: none"><li>1. Compiler: Generate MIPS codes to support complex functions</li><li>2. Connect compiler with assembler and linker</li><li>3. Test functions</li></ol>
刘润	22.75%	<ol style="list-style-type: none"><li>1. Compiler: Lexical Analysis &amp; Syntax Analysis &amp; Jiu Zhang Syntax.</li><li>2. Editor: Debug</li><li>3. Assembler: Debug</li><li>4. Simulator: Debug</li></ol>
刘芷萱	22%	<ol style="list-style-type: none"><li>1. Editor GUI layout framework, preliminary implementation of menu bar function, textedit rows displays, render</li><li>2. Assembler</li><li>3. Simulator</li><li>4. Linker code optimization and debug</li></ol>
卓然	21.25%	<ol style="list-style-type: none"><li>1. Editor: GUI design, File processing &amp; Project tree, Text render, Highlighter, Menu bar optimization</li><li>2. Code Integration: Link between GUI and compiler, Preliminary implementation of input and output functions</li></ol>
舒欣	10.5%	<ol style="list-style-type: none"><li>1. Looked for reference data for linker</li><li>2. Figured out the working principle of linker</li><li>3. Wrote the code of linker.</li></ol>

## 5 Reflections

### 5.1 Quan Liu

#### 1. Debug in team cooperation

In the test phase, the codes written by different teammates were connected together. I found the output of Boolean type variable, which should have line breaks, are shown in one line in the final result. Firstly, I comment out all the codes that are possible to cause bug, but the result is still not correct. Then, I try replaced Boolean type variable by Int type variable, which are same in some way. I found Int type variable can run normally. Therefore, I review my program again and again to check the reason causing the difference. Finally, I realized that the bug exists in my teammate's program, which is next one to connect with mine. This



process takes me several hours.

## **2. Time arrangement in team cooperation**

I was responsible for compiler's second part, most work began after teammates finished the compiler's first part. It made me starting my code lately. However, I realized now that some works are no need to wait to start. For instance, my work is to achieve two functions: traversal and conversion. The conversion part has less connection with previous program. Therefore, it is the better choice to start from conversion part, instead of starting traversal part, which is the first part in logical order.

## **3. Learn from CSDN**

Learn to traverse map; Learn to write recursion function in MIPS code; use stack point(\$sp) to allocate memory in MIPS code; Learn to use "throw" to report error.

## **4. Learn from class**

The knowledge related to tree: traversal in preorder and postorder; use "new" to allocate memory and use "delete" to release memory after using function; use structure; use recursion function; use C++11 language for(auto it : map); use (info\* &); use map.count() to judge whether the key is in the map.

## **5.2 Run Liu**

Designing a compiler, one important topic in Computer Science, is an extremely challenging task, especially for me, a Data Science student. In CUHK(SZ), there is a course about compiler construction, CSC4180. As a result, it took me a lot of time to search and learn related knowledge about compiler, because from the classroom I learned only the most basic part of C++. It was such a relief when the project was finished.

## **5.3 Zhixuan Liu**

In the following content, I will briefly put forward a representative problem I encountered of this project, and attach my corresponding solution strategies. In the end, I will

summarize what I have learned from this project that is beyond our course and has benefited me a lot.

### **Simulator and GUI connections:**

Firstly, I set up a QTextedit for the program input and output part, and set it to a terminal-like style. In other words, the QTextedit interface was designed to interact with the user. I have tried to find the position of the cursor in QTextedit, determine the number of lines where the cursor is, and set that the user can only enter the content when the cursor is on the last line. However, I found that this method cannot extract the instructions input by the user properly. Then, I tried to nest the terminal in the editor GUI. However, this method made the interface look messy, and when the Chinese characters were output, they were all garbled.

Finally, I decided to separate the user input and output areas. For input, the user can input Chinese numerals. When the user clicks the input button, the Chinese characters entered by the user will be recognized and converted into Chinese numerals. Only when the syscall instruction is run, the converted number will be stored in the general register.

However, when dealing with output instructions, I encountered another problem. The first is how the simulator modifies the display content of the GUI. Because editor and simulator are implemented in different cpp files, and the instance is only defined in main.cpp. I was unable to call and modify the GUI display content in simulator.cpp. I tried to write the main.h file, or set the editor GUI instance as a global variable, but neither of them succeeded. Finally, I added the pointer of the editor GUI instance to all the function parameters, so that the simulator can modify the display part of the editor.

When the simulator and GUI are successfully connected, I found that only UTF-8 encoded files can run successfully, and ANSI encoded files cannot output Chinese correctly. After searching information, I found that the UTF-8 Chinese storage uses 3 bytes while the ANSI Chinese storage uses 2 bytes. The compiler has a problem when calculating the memory address where the Chinese data is stored. After modifying the Chinese character placeholder, the simulator can output on the GUI interface.

When processing user input, I used a while loop to wait for the user to change the variable, which caused the program to collapse. I tried to give the while loop an interval, but I

found that this is a single-threaded program, and such a function cannot be achieved.

Eventually, I found a `processEvent` in Qt that captured user behavior.

### **What I learned:**

Through this project, I learned the basic design of the Qt GUI, I know the various encoding formats and conversions of Chinese, and I have a deep understanding of the working principle and implementation of the linker.

## ***5.4 Ran Zhuo***

In the process of writing an GUI editor, I ran into various difficulties and adopted different methods to deal with them. I will first discuss the tree connection problem detailly, and then mention other problems briefly. In the end, I will give a summary of what I learned.

### **Tree connection:**

When I have established the basic layout of our GUI program and put each component in its position, I found that the components are still independent fragments. As for the file tree, I could only add or delete items in the tree widget, but could not open the file directly in the code editor, or operate the actual files and folders in the computer.

I want to allow users to manage files and projects conveniently in the editor interface, instead of simply playing with the tree nodes and making no difference outside the tree widget. However, at first, I had no idea how to connect the file tree on the left to the code editor as well as the computer storage. Neither could I find a similar example for reference.

To deal with the matter, I analyzed the demand and split it into several parts. Generally I took three steps to analyze.

First, I decided all the options to be listed in the right-click menu of the tree widget, which includes (1) creating new projects, (2) adding new files, (3) renaming or (4) deleting the chosen project or file, (5) closing or (6) running the chosen project.

Second, I divided each function to smaller pieces of problems. For example, to add a new file successfully, I need to (1) take an input from the user as the new file name, (2) refresh the file tree on the left, (3) add the file under the computer directory according to the

document location of the project folder, and (4) open the empty new file in the code editor.

Third, after splitting tree connection into more than 20 small targets, I dealt with the targeted problems one by one. Finally, I got through the connection problem successfully.

### **Other problems:**

At the very beginning, Zhixuan and I read the material attached (Data Structures for Text Sequences.pdf) but still had no idea how to start. We discussed with each other and found sample projects provided by Qt. We referred to the samples and developed our own class CodeEditor based on QPlainTextEdit.

When designing the style of our interface, first I tried to use `setStyleSheet` to modify the style of each widget. However, it was too troublesome, and I learned to import existing style documents instead.

When I wrote the highlighter of our language, I spend quite some time learning to use regular expression.

Once we wanted to generate nice pictures just like ancient books by rendering Jiu Zhang code. However, we could not find a way to lay the text from right to left, from top to bottom. So we modified it to a second-best scheme, changing the font and adding a nice background.

Code integration is one of the most difficult problems we solved. When we collected all the effective code from our group members and tried to combine them together, we ran into a whole bunch of questions and found it hard to debug. We devoted a lot of time to communicating with other group member and improving the consistence of our code, and finally got it through.

### **What I learned:**

Through this project, I learned to use Qt to create a GUI program, refer to Qt Assistant for help, search for answers in Stack Overflow, and to cooperate with others and write code as a group. I also learned to take advantage of existing materials, obtained a better knowledge of MIPS language, and got a general idea of using regular expressions and designing a new programming language.

## 5.5 *Xin Shu*

### **Difficulties:**

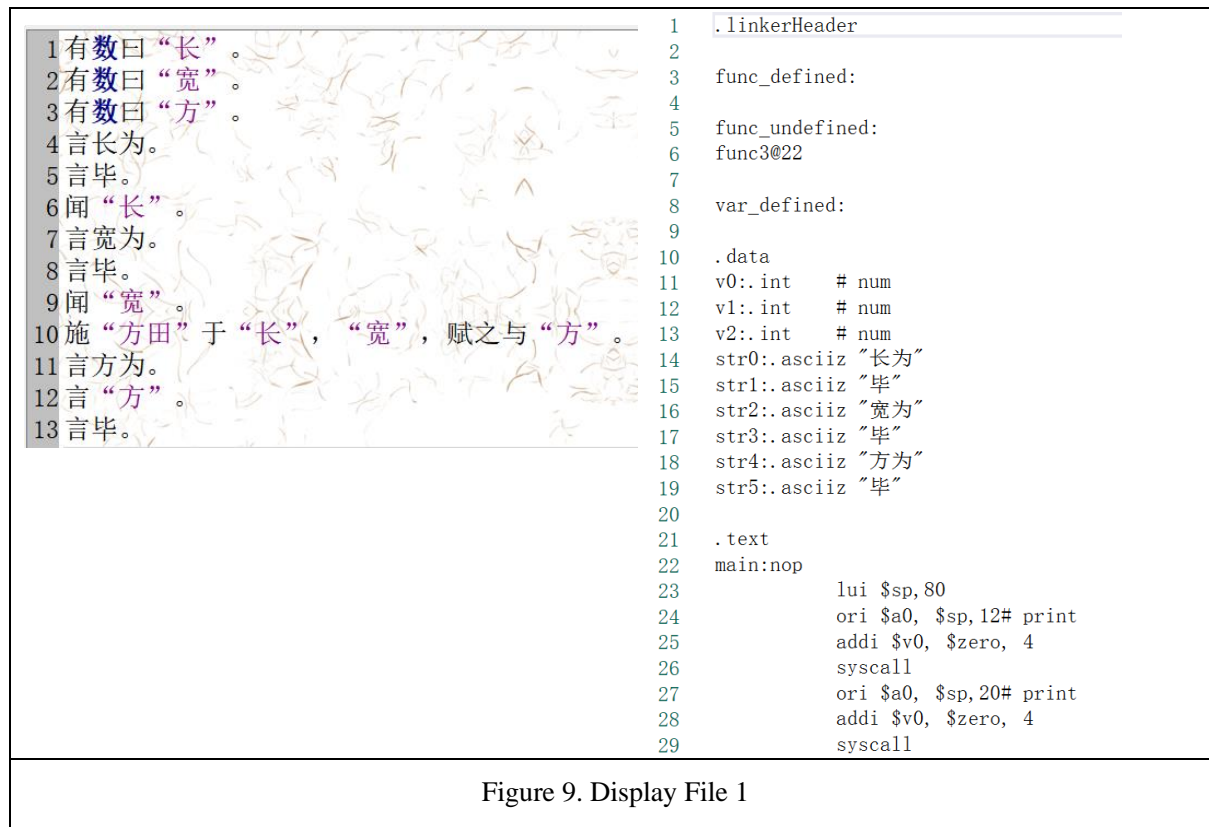
1. To solve the scenario where main function calls function1 in the library file\_1, and function1 calls function2 in library file\_2, originally we decided to use the recursion, but then we came to a more simple solution, which is to copy all the function codes defined in the library files into main file, then calculate the corresponding function addresses in the integrated file. This way, we could handle multiple references easily.
2. Originally, we are wondering how to integrate the data section of library files, however, since we choose not to support and export the static variable, we cancel the data section of library files and only support exporting functions.
3. Since compiler will generate some invalid main function codes in library file, we need to cancel them.

### **The connection and unification between different parts is important.**

Since linkers receives the files assembled and generates one file totally linked, it is really important to coordinate with the compiler, assembler and simulator, and came to our own solution. To fulfill this, originally, we defined an ELF (Executable and Linkable File) format, with “.linkerHeader” being the information section transferred from compiler to assembler, and finally to linker. However, out of simplicity and convenience, finally we decided to reference the undefined functions, both in the main file and the library files, with just labels, in this way we could combine the main file and library files and easily calculate the new addresses of j and jal machine instruction, therefore we do not need a .linkerHeader section. It is true that during the preparation period, we were continuously altering our solutions. It is just this process of creation and modification that makes our solution simpler and more effective.

## 6 Appendix

### 6.1 Compiler demo



1 有法曰“方田”，关乎数“甲”，数“乙”。	1 .linkerHeader
2 有数曰“丙”。	2
3 以“甲”乘“乙”，赋之于“丙”。	3 func_defined:
4 施“赋税”于“丙”，赋之与“丙”。	4 func3
5 得“丙”。	5
6 乃止。	6 func_undefined:
	7 func7@0
	8
	9 var_defined:
	10
	11 .data
	12
	13 .text
	14 main:nop
	15 lui \$sp, 80
	16 addi \$v0, \$zero, 10
	17 syscall
	18 func3:nop
	19 lui \$t0, 1
	20 sub \$sp, \$sp, \$t0
	21 sw \$ra, 0(\$sp)
	22 sw \$a0, 4(\$sp)
	23 sw \$a1, 8(\$sp)
	24 lw \$t1, 4(\$sp)#multi
	25 lw \$t2, 8(\$sp)
	26 mult \$t3, \$t1, \$t2
	27 sw \$t3, 12(\$sp)
	28 lw \$a0, 12(\$sp)
	29 jal func7 #usefunction
	30 lw \$ra, 0(\$sp)
	31 sw \$v0, 12(\$sp)
	32 lw \$t0, 12(\$sp)
	33 ori \$v0, \$t0, 0
	34 lui \$t0, 1
	35 add \$sp, \$sp, \$t0
	36 jr \$ra

Figure 10. Display File 2

1 有法曰“赋税”，关乎数“甲”。  
2 以“甲”减三，赋之于“甲”。  
3 得“甲”。  
4 乃止。

```
1  .linkerHeader
2
3  func_defined:
4  func7
5
6  func_undefined:
7
8  var_defined:
9
10 .data
11
12 .text
13 main:nop
14     lui $sp, 80
15     addi $v0, $zero, 10
16     syscall
17 func7:nop
18     lui $t0, 1
19     sub $sp, $sp, $t0
20     sw $ra, 0($sp)
21     sw $a0, 4($sp)
22     lw $t1, 4($sp)#sub
23     li $t2, 3
24     sub $t3, $t1, $t2
25     sw $t3, 4($sp)
26     lw $t0, 4($sp)
27     ori $v0, $t0, 0
28     lui $t0, 1
29     add $sp, $sp, $t0
30     jr $ra
```

Figure 11. Display File 3







- 26 -