CSC3002

Introduction to Computer Science: Programming Paradigms

Project Report

The System Binary Tools

Huayue Li 118010138

Yuhao Feng 118010068

Guodong Shi 118010257

Longxiang Li 118010144

Xinyu Fang 118010061

Haoyang Hong 118010096

Course Coordinator: Prof. Rui Huang

The Chinese University of Hong Kong, Shenzhen

Spring 2020

Introduction

An integrated development environment (IDE), is everything a programmer needs to get their work done. An IDE is a good integration of editing, files, management, debugger and other functions, which makes the development easier and faster. We choose the system binary tools as our project topic and implement all the things required including editor, compiler, simulator, linker, loader, assembler and debugger. The reason why we choose this topic is that after the group discussion, we are all interested in it and think we can learn a lot from the process of programing of this topic. In addition, we also learn some related knowledge this semester so that we consider this as a great opportunity to practice.

Related work

The basic operation skills are taught in class. Moreover, to solve some critical problems, we learned extra knowledge from the Internet as well as electronic books. Some related online courses are also helpful. Also, some thoughts on Github are worth learning.

Contributions

Team Member	Work Division
Huayue Li	Editor
Guodong Shi	Compiler: Building the Abstract Syntax Tree
Yuhao Feng	Compiler: MIPS Assembly Code Generation
Longxiaong Li	Linker, Assembler
Xinyu Fang	Loader, Simulator
Haoyang Hong	Debugger

Cooperation between teammates is also necessary and all our team members try the best to accomplish this project and make it as complete and perfect as we can. Then, we will divide our report into eight parts and illustrate them separately.

Editor

Introduction:

For editor, there is no change from the proposal. For some basic functions of editor like open, save, and paste, because their logic of codes is common and easy, so we just use the code which we learnt from a course on the Internet with some modifications to satisfy our requirements. And there are also some functions which are not important for our project like the font settings in the code, to make sure the integrity of the whole code of our project, we do not delete them and you can ignore them because they are not important for our project. For the most important part of the editor: run and debugger, we use our own logic of code to finish them.

Related work:

For editor, we get our ideas from some courses on the Internet which taught how to design basic editor (without run and debug functions). And we get the ideas of run and debug functions through talking in our group. The necessary external resources are gotten from textbooks and the Internet.

Our work:

For the editor, we design many basic functions as follows:

For the operations of file: open, save, save as, save all, close and so on.

For the operations of edit: cut, paste, run and so on.

For the operations of window: transform of windows.

For the operations of debug: debug one step and debug all steps.

We design many buttons in the window of our editor and create many functions which are connected to these buttons so that when the buttons are clicked, the functions will be called to implement corresponding functions.

For example, for the run function, we design a button of run, when the user clicks this button, the program will call the function run and then call the function of

compile…. Through the whole process, the program can translate the code in the editor into the real result and show it in the terminal of Qt.

(贴图)

Higher grade: the functions of our editor are relative comprehensive which include all kinds of aspects like file, edit, window and so on.

The window of our editor is clear and beautiful.

Reflections:

When we design the run function of our editor, we encounter the difficulty that our editor can only run one file but not many files together. We talk about this problem and finally choose to use the save all instead of save in the run function to overcome this problem. When we design the debug function of our editor, we encounter the difficulty that our editor cannot update the information of debug in our debug window. We talk about this problem and finally choose to move the functions of debug from mainwindow.cpp file to a new debug the cpp file to overcome this problem.

From this project, we have learnt a lot of editor design using qt creator. We leant how to use ui file and how to connect our code to the widgets like buttons in the editor.

Compiler: Building the Abstract Syntax Tree

Introduction

This is the first half of compiler, including the design of a simple language, lexical analysis and syntax analysis. For syntax analysis, I learned something new from the book *Introduction to Compiler Design,* and build an abstract syntax tree using basic knowledge.

Our work

Lexical analysis is completely using the knowledge learned from class. Source code are read line by line, and the lexer will judge distinguish the labels or symbols. If the key word ‘main’ is detected, then a syntax tree with the label of main will be built. The root node of syntax tree has a vector containing all the children nodes, and those nodes contain the syntax tree of each line individually. Another case is when the key word is ‘def’, then it would be stored in a new syntax tree which represents function definition. All of roots will be stored in a vector so that they can be used to generate final code. The following codes show the process of main as an example:

```
// Do the lexical analysis line by line
void lexer(string file){
    ifstream infile;
    infile.open(file.c_str());
    string line;
    bool isMain;

    while (getline(infile, line)){
        if (line == "main"){
            isMain = true;
            Node *mainTree = new Node;
            mainTree -> type = "main";
            mainTree -> expr1 = new Node;
            mainTree -> expr2 = new Node;
            mainTree -> expr1 -> type = "NA";
            mainTree -> expr2 -> type = "NA";
            roots.push_back(*mainTree);
        }
```

In order to deal with arithmetic expression, I defined a function called symbolCheck. This function can check whether the code contains symbols like ‘+’, ‘-’ or others, and decide the type of that code. If it contains symbols, then it is expression. Otherwise it is a factor.

```
//Check symbols
void symbolCheck(string line, Node *&node, string symbol){
    string lexp, rexp;
    lexp = line.substr(0, line.find(symbol));
    rexp = line.substr(line.find(symbol)+1);
    node->expr1 = new Node;
    node->expr2 = new Node;
    node->expr1->data = lexp;
    if(existed("+", lexp) || existed("-", lexp) || existed("*", lexp) || existed("/", lexp)){
        node->expr1->type = "expression";
        buildTree(node->expr1->data, node->expr1);
    }
    else node->expr1->type = "factor";
    node->expr2->data = rexp;
    if(existed("+", rexp) || existed("-", rexp) || existed("*", rexp) || existed("/", rexp)){
        node->expr2->type = "expression";
        buildTree(node->expr2->data, node->expr2);
    }
    else node->expr2->type = "factor";
    node->symbol = symbol;
}
```

Since our language can support at most 2 parameters for each function, varCon is designed to connect each the name of functions with their types. And the process of detect the amounts of parameters are included.

```
// Connect the type of variable and the name of variable
void varCon(string func, string line, Node *&node ){
    if(existed(",", line)){
        //2 parameters
        string var1 = line.substr(line.find('}')+1, line.find(',')-line.find('}')-1);
        string var2 = line.substr(line.find(',')+1);
        node->expr1 = new Node;
        node->expr1->type = func;
        node->expr1->expr1 = new Node;
        node->expr1->expr2 = new Node;
        varType(var1, node->expr1->expr1);
        varType(var2, node->expr1->expr2);
    }
    else if (line.find('}') == line.size()-1){
        // no parameter
        node->expr1 = new Node;
        node->expr1->type = func;
    }
    else{
        // 1 parameter
        string var = line.substr(func.size()+2);
        node->expr1 = new Node;
        node->expr1->type = func;
        node->expr1->expr1 = new Node;
        varType(var, node->expr1->expr1);
    }
}
```

The main part of building the abstract syntax tree is in the function buildTree(). This function will be called each time the code is divided into smaller parts, and therefore the tree can be built.

Reflections

We also learned some new skills through though the process. For instance, we learned to use external variable, which perfectly solved the problems of redefinition. The most important thing is that we have a deeper understanding to the design of language. We learned the basic construction of syntax tree from the book *Introduction to compiler,* and build the abstract syntax tree by defining a node, connecting children leaves with parents. The connection between function types and their values can be paired in the tree, which makes it clear to use them in the later semantic analysis.

Some difficulties are unavoidable. For example, since the original way to do the lexical analysis is to read through each line of code only once, it is difficult to judge

whether the brackets will affect the expressions. So, we try another method to solve the arithmetic expressions containing brackets. For those lines of code which contain brackets, we read them character by character so that we can use a stack to store those brackets, and judge whether arithmetic symbols are within the brackets. Another difficulty is when connecting with editor, we find that our compiler sometimes cannot compile again after the first click of 'run' button. Later, we find out that it is because the mistake of memory management. As a solution, we change the style of releasing the memory space.

Compiler: MIPS Assembly Code Generation

Introduction:

This part is the second half of the complier part, which is mainly focused on translating the previously generated abstract syntax tree (**AST**) into MIPS assembly code. The MIPS assembly code contains the data part (containing static data) and text part (containing instructions), and can be executed by the linker. This part is called code generation. It is comprised of "**CodeGen.h**" and "**CodeGen.cpp**".

Related work:

My ideas for the compiler was inspired by an online video course provided by Stanford University. I got the general idea of a compiler and simplified the structure for our project. In this code generation part, all the codes are original, and all the algorithms were designed by ourselves. I also read the book, *Computer Organization and Design: The Hardware-Software Interface*, as a reference of MIPS assembly language standard. We designed our own data structure to store the complicated AST, for higher efficiency and simpler implementation.

Our work:

The core algorithm applied in this part is post-order tree traversal. This guarantees the code was translated from the leaves of the tree to the root, and for each

node, the two children are both finished processing before the node starts processing. This fulfills all dependencies in the program. For the static data part, a traversal is performed to find all the variable declarations and strings inside the “print” function. All the integers use the **.word** format, and are set to 0 as default. All the strings use the **.asciiz** format. All the static data starts from 0x500000 in our virtual memory. The address of these data was calculated respectively and stored for further use. For the text part, a traversal will be performed on each line of the code, which corresponding to each element in the vector leaves of the root node. The MIPS assembly code will be generated in appropriate sequence. This requires special attention to register allocation, memory access, function calling and parameter passing. If there are multiple functions defined in the code, the codes will be generated correspondingly, with a label printed before each section.

Let me explain some functions used in this part. The function “**translation**” is the topmost function in this part, which manages I/O. The function “**loadStatic**” will traverse the AST and load every static data into vector “**staticData**”. I structed a type “**tag**” to store each static data. The function “**writeStatic**” will translate all the tags stored in the vector “**staticData**” into MIPS assembly code. The function “**writeCode**” is responsible for translating all the operations into MIPS assembly code, namely, the .text part. In the function “**writeCode**”, the core function is “**treeTrav**”. Function “**treeTrav**” will traverse the tree below a given root, and translate the tree into assembly operations. Since each line of code in the source program represents an element in the vector **root.leaves**, each leaf in **root.leaves** will be traversed separately, using a for loop. If there are multiple functions defined in the source program, each of them will be traversed and labeled separately.

I also want to elaborate some key parts. The first part is register allocation. Since $t0-$t9 are the temporary registers used for temporary storage, they need to be allocated properly and produces no conflict. I use a stack to store these registers, and can be popped when there is need. The stack is in the writeCode function, and its name is **tempRegs**. After a register finishes its duty, it will be pushed back into the

stack as soon as possible. The AST traversal has a return type of string, and this string is used to pass registers. For the child nodes, the name of the register storing the calculation result will be returned to its parent node, so that the parent node can directly use the register for arithmetic calculation. This approach makes the AST traversal simple and concise.

For the function part, some specific logics are applied to avoid conflict. Our compiler supports at most two parameters for each function. These parameters are bound with registers. Parameter 1 was bound to **$s0** and parameter 2 was bound to **$s1**. After a function is called, the data will be sent directly into these registers. Inside the function, manipulation on these parameters will be interpreted as manipulation on **$s0** and **$s1**. The advantage is that it eliminates the conflict between the parameter name and static data, in case that the parameter name is the same with some previously declared variable.

Moreover, the code generation part also supports some simple syntax error detection. For example, it will notify the user when printing a variable which is not declared before, or the arithmetic expression contains some illegal characters (like English characters). However, it does not mean that it supports a complete debugging function.

Reflections:

This project gives me a more profound understanding of translation, not only the translation between computer languages. I think the translation between human languages are also similar. The first step is to decompose the sentences into tokens, and analyze them to put them in a specific **"container"** (like the **AST** used in this project). This step is to simplify the meaning of this sentence, because different sentences can have the same meaning, which is the same as computer language. The second part is to examine the elements inside the **"container"**, and rearrange them using the syntax of the target language. The most difficult part of building the complier is to design a suitable **"container"**, so that it can represent all the data and

relationships in a concise way, and facilitates translation in both ways. Our compiler is relatively simple and primitive. Further optimization can focus on reducing the size of the assembly code or use simpler(faster) operations to replace complicated(slower) operations, to make the assembly code execution more efficient.

Assembler

An assembler is a program that translates assembly language into machine language. During the final running process, the assembler receives the **ASM file** from the compiler and generate an **object file** (which contains the machine code and related information) for the linker. In general, the target codes generated by assembler can only become executable after transferred by linker.

Implementation

The implementation of our assembler partially refers to the idea given in *Computer Organization and Design: The Hardware/Software Interface (5th Edition)* (Patterson & Hennessy, 2014). In our assembler, the assembly language is chosen to be MIPS due to its regularity and practicability. During the assembling process, the ASM file will be scanned twice. The first is for recording the address of all the symbols, and the second is for the actual assembling. During the assembling, all the information will be recorded in an ***ObjInfo*** object. (See the Appendix I for the description of class ***ObjInfo.***)

In order to make it more convenient to implement the following tools in the IDE, the content in the **object file** are extended according to ELF format. More specifically, one object file contains *Header*, *Text segment*, *Data Segment*, *Relocation information* and *Symbol table*. The contents and access methods for each particular part are shown as follows.

Header

It contains the information of the size of the total data and the total text in one file. This is obtained from the first two lines of comments in the ASM file.

Text Segment

It contains the machine codes and their corresponding virtual address. The method to get the machine code has been descripted in the proposal. Briefly speaking, MIPS instructions are divided into three formats (R/I/J) and they are further subdivided according to the specific type and numbers of their arguments. In this case, the transformation for each particular sub-format becomes a simple matching and replacement.

Data Segment

The content in this section is the same as that in the *.data* section in the corresponding ASM file.

Symbol tables

The names and the addresses of all the data and labels are recorded in this section. During the first scanning, whenever the assembler detects a label or a piece of data, it records the name string and the current address by program counter.

Relocation information

This section offers information for the relocation operation in the **linker**. The content in this section are the virtual address of the instructions which are needed relocated. More specifically, these instructions contain **jump** instructions, **branch** instructions and **memory operations** (loading/storing). For the first two kinds, during the second scanning, whenever the assembler encounters an unresolved symbol, it records the current virtual address, the opcode of the instructions and the dependent symbols. For memory operations, since the dependent symbols for memory operations cannot obtained directly from MIPS assembling language,

Linker

In modern programing, it is common that a program will be split into different files in the advantage of specialization and cooperation. Such a coding style necessitates an additional step to integral. The **linker** is such a tool to implement the step. When multiple object files are passed from the assembler to the linker, the linker will combine them into a single object file and pass it to the loader. One mentionable thing is that, in our IDE, the information transmission between linker and assembler is based on the ***ObjInfo*** object rather than file in order to simplify the operation.

Implementation

The implementation of our assembler partially refers to the idea given in *Computer Organization and Design: The Hardware/Software Interface (5th Edition)* (Patterson & Hennessy, 2014) and the tutorial offered by Hakim Weatherspoon.

The linker implemented in our project is a MIPS static linker. As descripted in the proposal, the linker will resolve all the unresolved symbols. In this project, the linker will shift the virtual address based on current data base or text base and combine multiple ***ObjInfo*** objects into one ***ObjInfo*** object. With the relocation information and the symbol table, the resolving and relocating operation can be simply implemented by matching and replacement.

Reflections

In the hypothetical implementation in the proposal, the mechanism of the linker is divided into three parts, which are to find all the library routines, to resolve the reference, and to relocate the instructions. For the first part, since library calls are not referred to and the linker is oriented to the MIPS assembling languages, there is no need to find the library routines. For the same reason, we chose to design a static linker rather than a dynamic linker. For the next two parts, it seems confusing because they are kind of repetitive operations with the operations in the assembler. Therefore,

we choose to expand the content to avoid the redundant operations. These experiences verify the importance of the ad hoc approach and cooperation.

Loader

Introduction

A loader is implemented in our project to mainly loading programs. It places programs into memory and prepares them for execution. To load a program, a memory space of 132 Bytes are allocated to store 34 registers, and a memory space of 10MB is allocated to store machine codes and data. Then, reading the contents of the object file containing the program instructions and related data into memory. Other essential preparatory tasks, for example, initialize all these registers, will be carried out at the end of the loading process. Once loading is complete, the loader will pass control to simulator.

Related Work

The general idea for the loader is from the book *Linker and Loader* written by Akshay Khatri. In the book, the author introduced the working mechanism and history of loader, which helps a lot on the structure of our loader. Nonetheless, all the related samples in the book didn't use C++, so we implement the loader totally by ourselves.

Virtual Address and Relative Virtual Address

Virtual address(VA) and relative virtual address(RVA) are needed to be dealt with when using a loader. Since in our system, each register only takes 32 bits, so we store 32 bits VA instead of 64 bits physical address in registers. A RVA is the virtual address of an object from the file once it is loaded into memory, minus the base address of the file image. Our loader has built the right connection between VA, RVA and physical address, which greatly improves convenience.

Reading Code into Memory

Machine code in text section are loaded into memory starting from VA 400000h, and data in data section are loaded starting from VA 500000h. File information like text size and data size are also stored in memory.

Initialization

Before starting to execute a program, loader need to initialize 34 registers. 30 registers, including hi, lo and register 0 to register 27, are initially set to zero. However, special registers like $gp, $sp, $fp needed to be treat differently. The following shows the corresponding three VA:

i. $gp: a00000h

ii. $sp:a00000h

iii. $fp:500000h

Simulator

Introduction

A simulator is included in our project. It can run the machine code in allocated memory, enabling the computer to execute programs written for a specific operating system.

Related Work

In the tutorials of CSC3050, TA gives us a brief introduction on how to implement a simulator, so we follow that and implement our own simulator.

Our Work

The preparation work has already been done in the loader, so the simulator can handle with execution part directly. The process of executing a program using the simulator can be divided into three parts.

1. Write functions for all the instructions to simulate the implementation of instructions. Our simulator supported 70 the most frequently used instructions, which is enough for our project.
2. Using if-else if structure to judge the name of each instruction and then call the function corresponding to it.

3. Write all the functions for the special instruction “syscall”. To be noticed, when $v0 == 10 or $v0 == 17, code *exit(0)* is replaced with *break*, because *exit(0)* will shut down the editor, which is obviously not expected.

In addition, there is something to pay attention to about program counter. In most case, the integer stored in program counter is automatically added to four after an instruction is executed. However, to assure the next line to be run is correct, special measures are taken to cope with the value of program counter when executing a few special instructions. These instructions are *beq, bgez, bgezal, bgtz, blez, bltzal, bltz, bne, j, jal, jalr, jr.*

Debugger

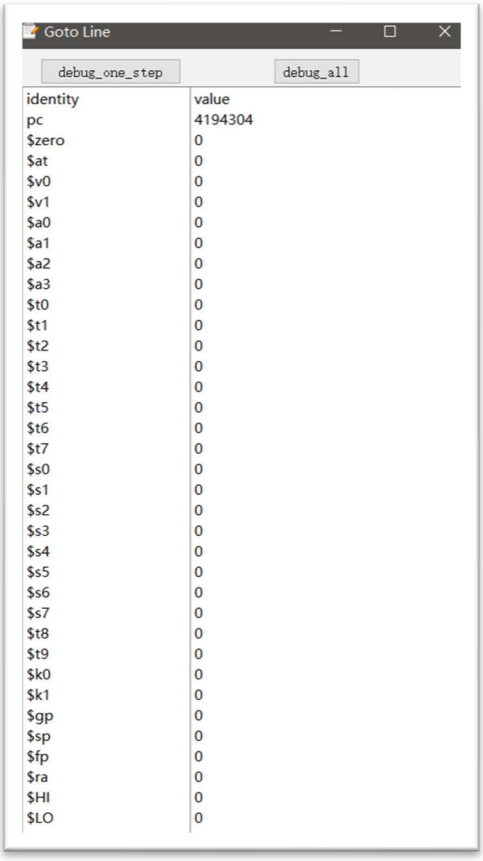
Introduction

Debugger is a computer program used to test and debug other programs. The main use of a debugger is to run the target program under controlled conditions that permit the programmer to track its operations in progress and monitor changes in computer resources that may indicate malfunctioning code. A debugger can be categorized into two categories: machine-level and source-level. Our project only focuses on the source-level debugger.

Function

1. Display the call stack of the source program, including the chain of procedure and function calls and the names of values of all the local variables of each routine in the call chain.
2. Display or set the value of a particular variable.
3. Resume execution of the source program.
4. Terminate execution of the source program immediately.

Our debugger basically accomplishes these tasks in a popped-out window. For instance, run the test file and press the debugger button on the right top of the tool bar.

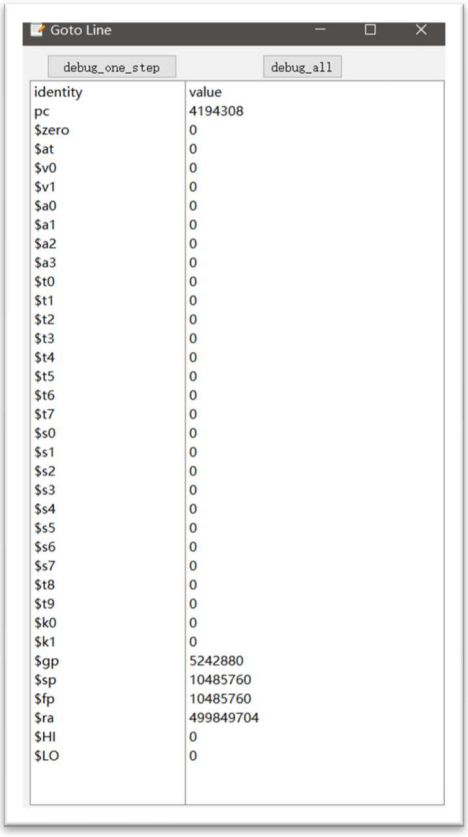


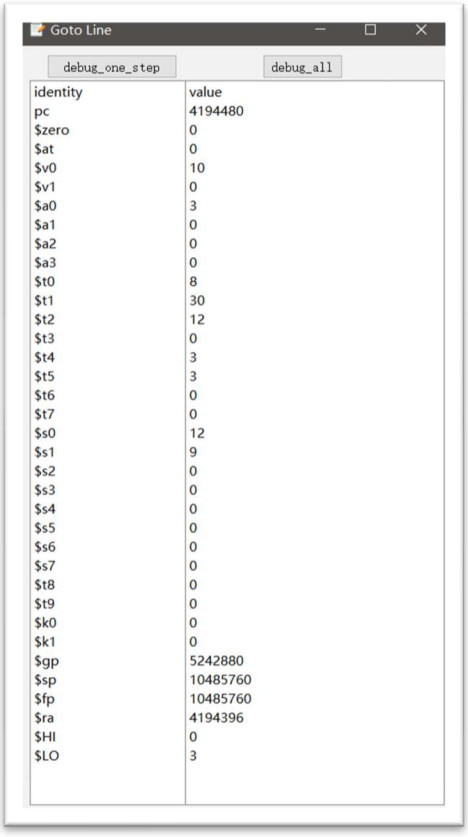
Then, the following results will show up.

Our debugger is oriented to the source code. The two blocks show the identity and value of the registers respectively. The two buttons on the top of the window are named debug_one_step and debug_all respectively on the top of the window.

The output as the left picture shows is the initialized value. When pressing debug_one_step, the debugger will show the corresponding value after one-line-forward execution (left one shows). When press debug_all, it will show the value after executing all the lines in MIPS form.

Contribution and Reflection





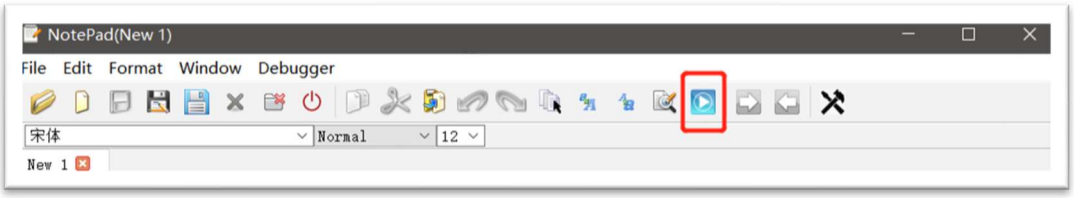
One of the tough parts we encountered during the construction process is the implementation of the debug_one_setp. This part needs the help of two teammates who are in charge of constructing editor and simulator. The problem to get the value of the registers is quite complicated. To solve this problem, we set these registers in debugger class. Then, the value of the registers will update according to the return value from the simulator functions.

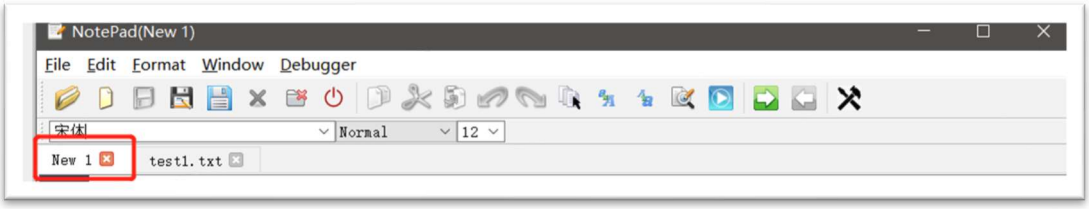
How to test our project

After running the codes:

To test the single-file function, import “test1.txt” and delete the “New 1” file. Then, press the “run” button, the blue arrow shaped one. The result will be displayed in the Qt window.

To test the multi-file function, import “test2.txt”, “test3.txt” and follow the same procedures showed above. Then the result will be displayed in the Qt window.





As for the test of the debugger function, import “test1.txt” and delete the “New 1” file. Then, press the debugger button on the right top of the tool bar. Then you can press “debug_one_step” button or “debug_all”. Then the result will be displayed in the right side of the popped-out window.