CSC3002

Introduction to Computer Science: Programming Paradigms

**Project Report**

**Topic** **Two**:

The System Binary Tools

**Group 36 members:**

Yuzhe Jin 118010128

Wentao Guo 117010078

Yuan Gao 118010075

Yuhao Du 118010053

Jiaqi Liu 117010165

Hanshen Jing 117010112

**Outline**

1. **Introduction**
2. **Related Work**
3. **Our Work**
4. Simplified C language definition
5. Compiler

a) Lexical analysis (lexical.cpp & lexical.h)

b) Syntax & semantic analysis

b1) symbolItem.h

b2) midCode.cpp & midCode.h

b3) grammar.cpp & grammar.h

b4) mipsCode.cpp & mipscode.h

1. Assembler & Simulator
2. Editor
3. Debugger
4. **Contributions**
5. **Reflections**
6. **Instruction**

Our group chooses the project topic two, the system binary tools. Among the listed system tools, we have chosen to implement editor, compiler, assembler, simulator and debugger. Because of time limitation, we decided to focus on the compilation in one file, thus the loader and linker were not implemented in the final product. However, we added a debugger based on the MIPS code to improve the integrity of the IDE.

1. **Related Work**

We gained the idea of how to implement a compiler from the following book:

*DIY Compiler and Linker*, Wang Bojun & Zhang Yu

We followed its ideas about lexical, syntax and /\*semantic\*/ analysis which are based on LL(1) grammar.

We learnt the structure and functions of MIPS code and the concepts of assembler and simulator from the following book:

*Computer Organization and Design (5th)*, David A. Patterson & John L. Hennessy

We imported Jin Yuzhe’s second project in CSC3050 to be the prototype of the assembler and simulator in this project.

1. **Our Work**
   1. **Simplified C language definition**

The simplified C (SC) language we defined at last is slightly different from the one mentioned in the proposal. The supported data type, flow control, function, operation and reserved word are shown in Table 1.

|  |  |
| --- | --- |
| Table 1. Supported language in SC | |
| Data type | int, char, void, array (char or int) |
| Flow control | if, else, for, do, while, return |
| Function | printf(), main(), self-defined functions |
| Operation | +, -, \*, /, “(double quote), ‘(single quote), {, }, [, ], (, ), >, <, !=, ==, >=, <=, = |
| Reserved word | Const, int, char, void, main, if, else, do, while, for, printf, return |

Note:

1. When declaring an array, the size of the array must be declared and the content of the array cannot be assigned directly. E.g. int array1[100]; char array2[100].

2. The printf() function supports single-variable or double-variable input. The single-variable input can be int, char or string. Int and char input can be variable name or constant value, however, string input can only be constant string like “something”. When passing two parameters to the function, for example, printf(“1=”, 1), the first parameter must be a constant string and the second one must be of int type. Note that the int type variable can also be a return value of function, e.g. printf(add(1, 2)), where int add() was declared before.

3. The entry of the program should be the function, void main().

4. The self-defined functions could only return int or void.

5. A demonstration of the notes mentioned above is shown in Figure 1.

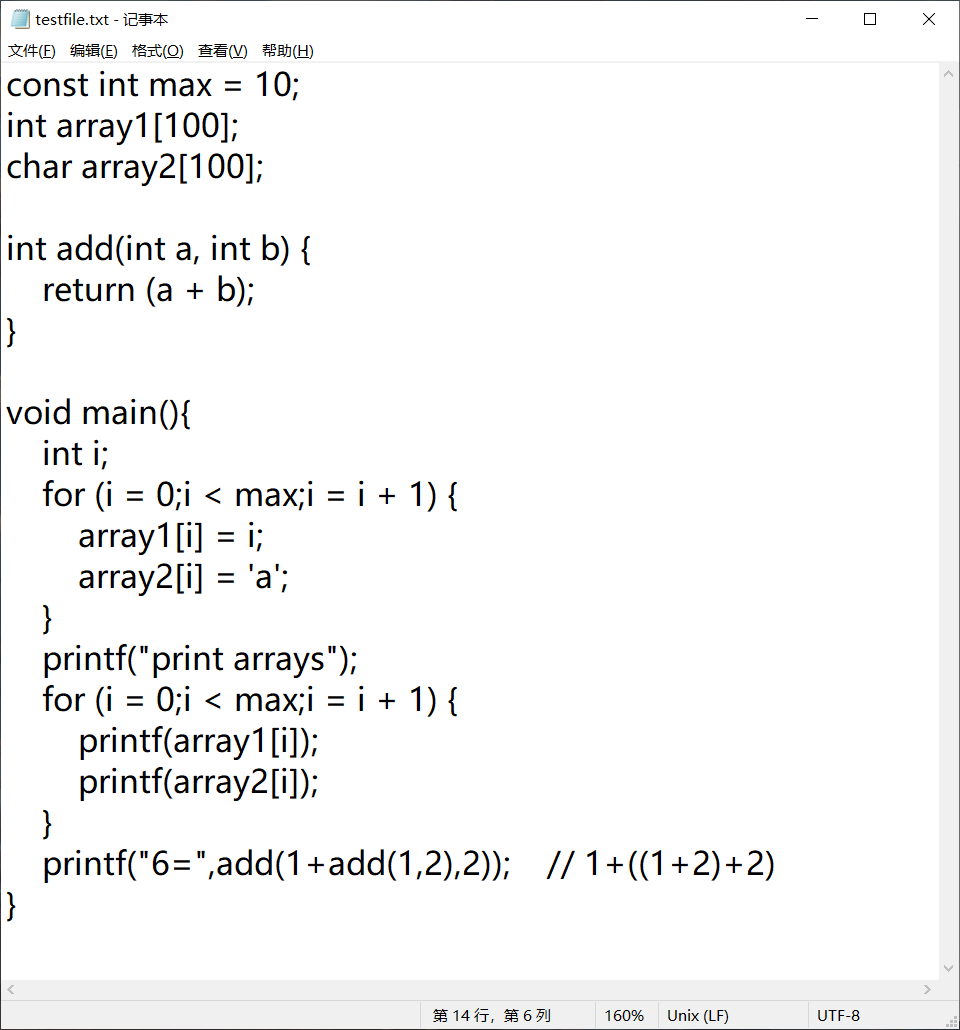


Figure 1. Demonstration of the SC

* 1. **Compiler**

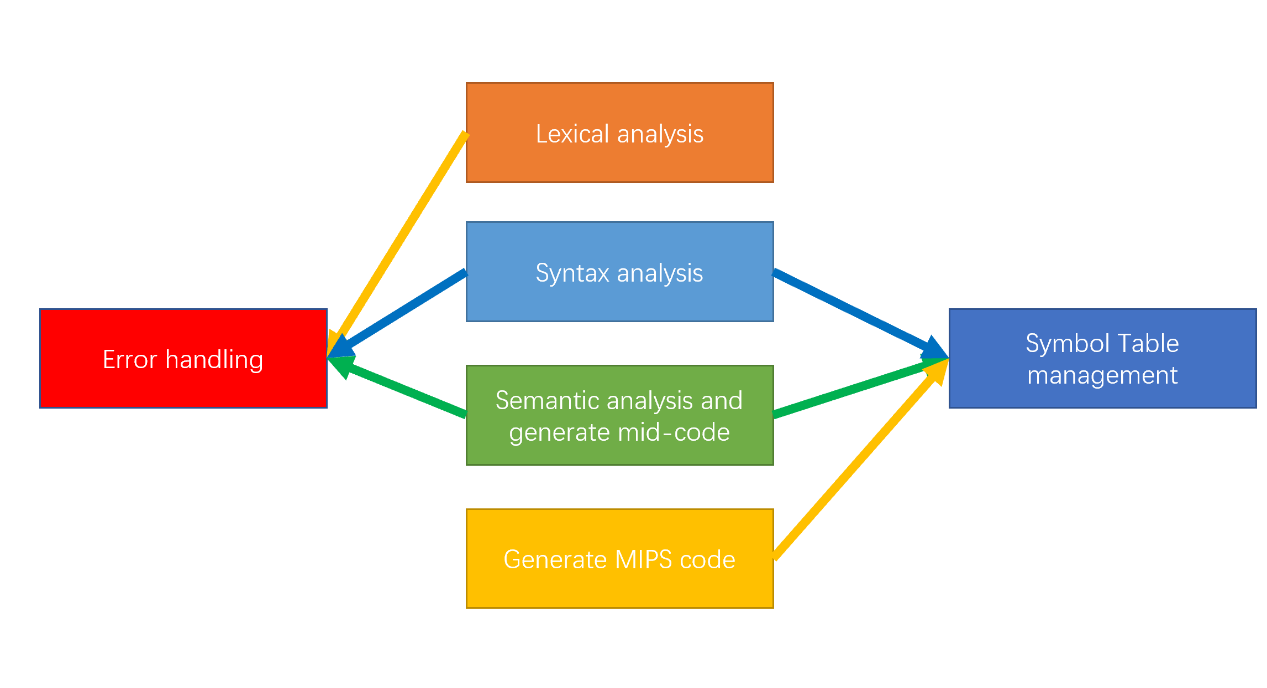
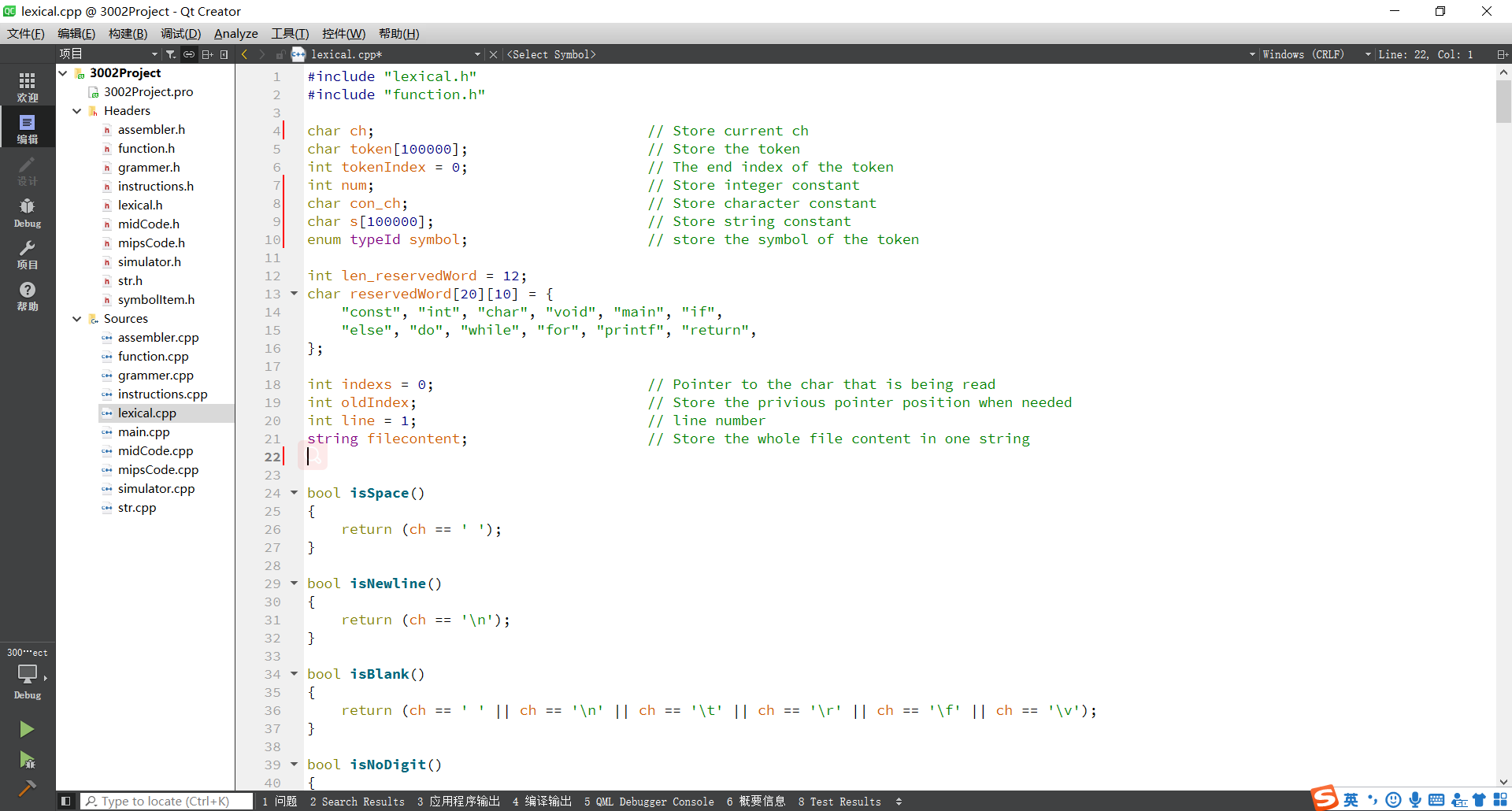
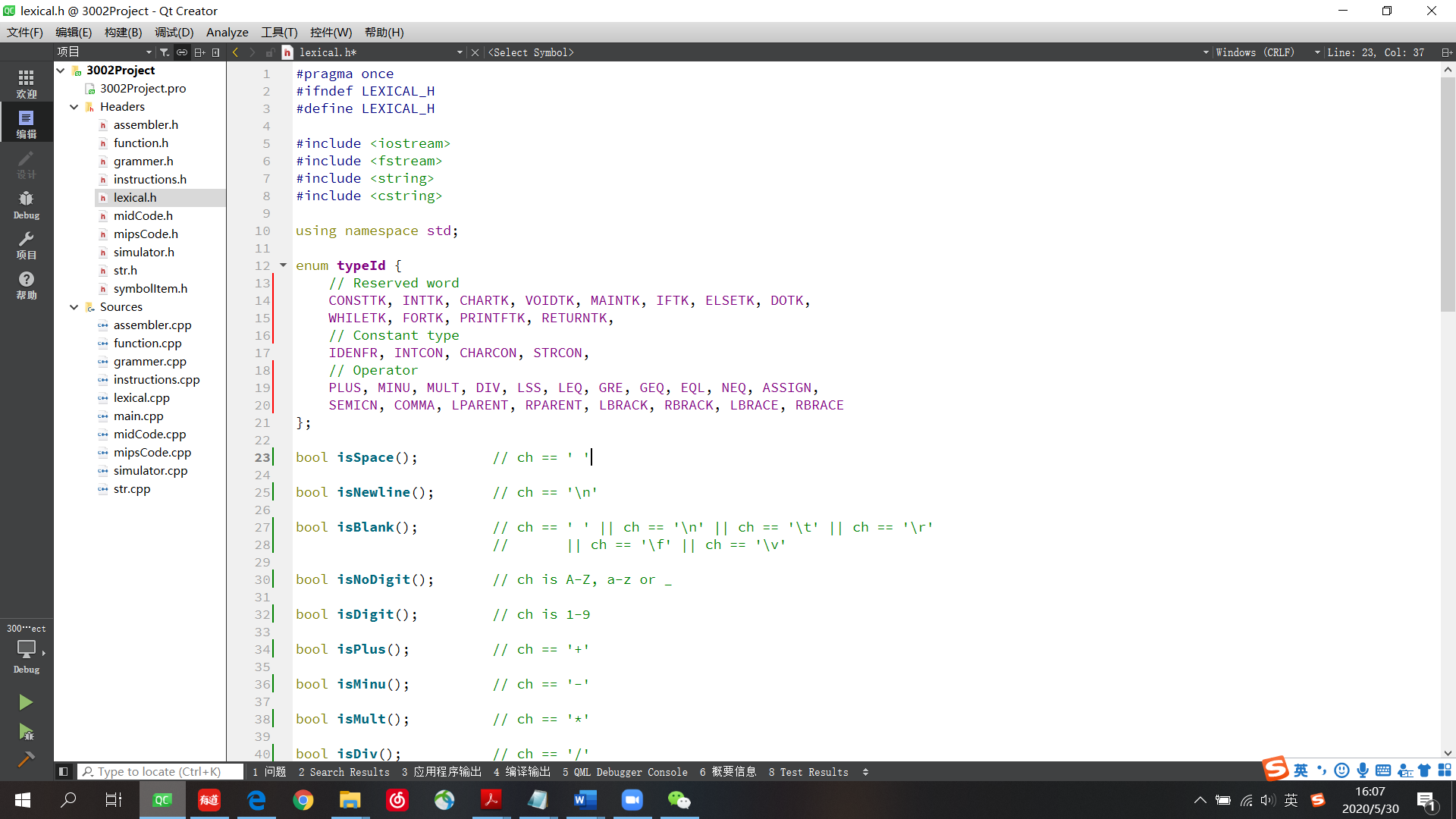


Figure 2. General logic of the compiler

1. **Lexical analysis (lexical.cpp & lexical.h)**

As the compilation process is based on LL(1) grammar, the lexical analysis should be done word by word from the file’s beginning to its end. In general, the lexical analyzer passes the information of one token to the syntax analyzer each time we call. It gets one token with its symbol while moving the pointer to the following character. The following variables are used to store the text or token’s information.





To simplify the process of reading the text, we first convert the text content into a string, where we use the variable, *indexs*, to access the corresponding character. Then we use get\_ch() to read the string character by character, which would be called by the function, getsym(), to parse the following token.





Before starting to parse token, we first call the function, preprocess(), to clear all the blank spaces and comments before any character appears.



Then, if reading an operator or punctuator, we let *symbol* equal to its corresponding value in *typeId*.

If reading a integer, we call the function, parseNum(), to finish collecting the integer and let *symbol* equal to *INTTK*.



If reading an alphabetic character, we call parseWord() to finish reading the whole token and see if it is a reserved word or an identifier.

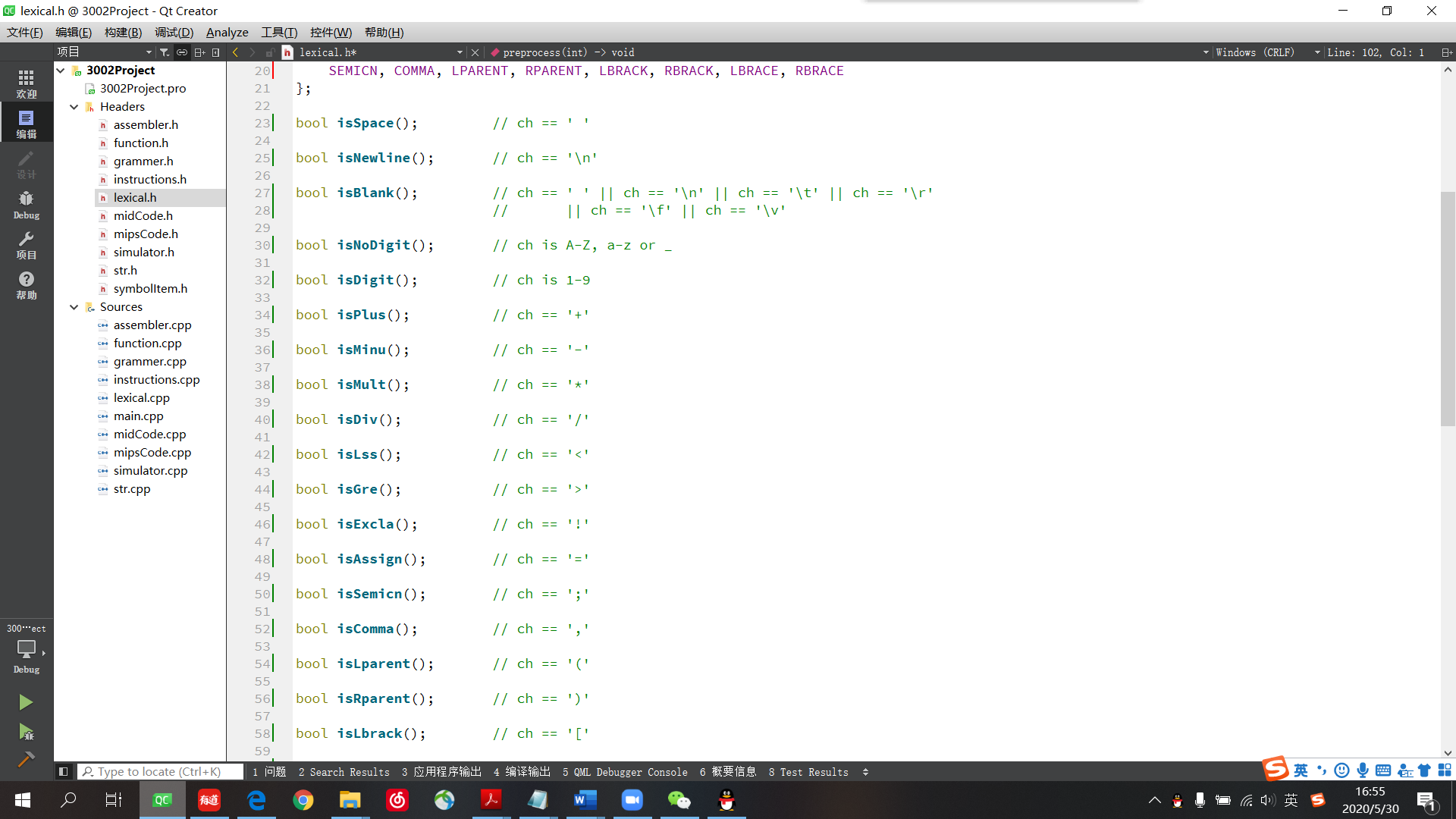


If reading a single quote or double quote, we call the function parseChar() or parseString() to collect the following text information.





Those are the basic logic of the lexical analyzer. Figure 3 shows all the functions appearing in the lexical.cpp.





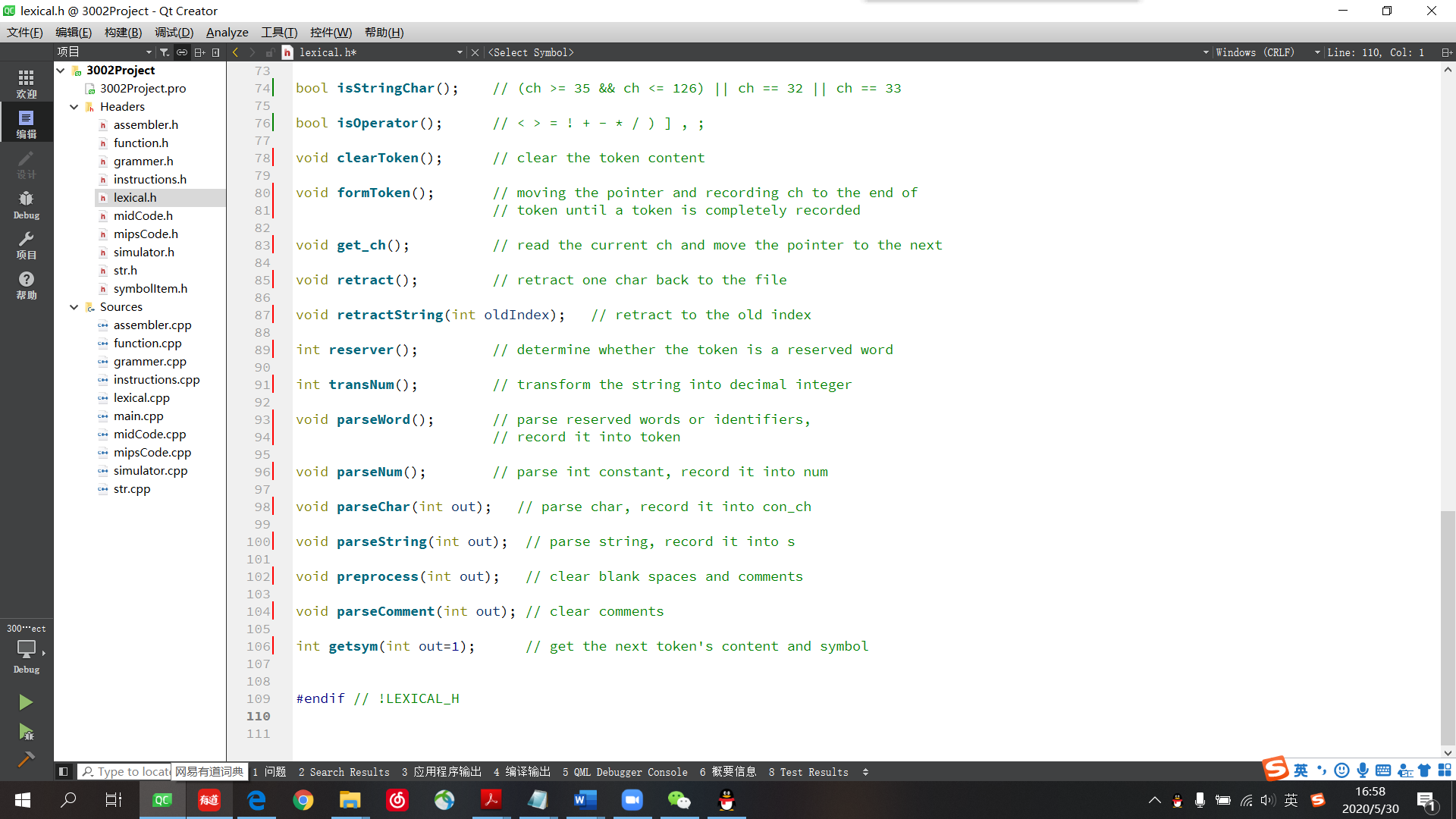
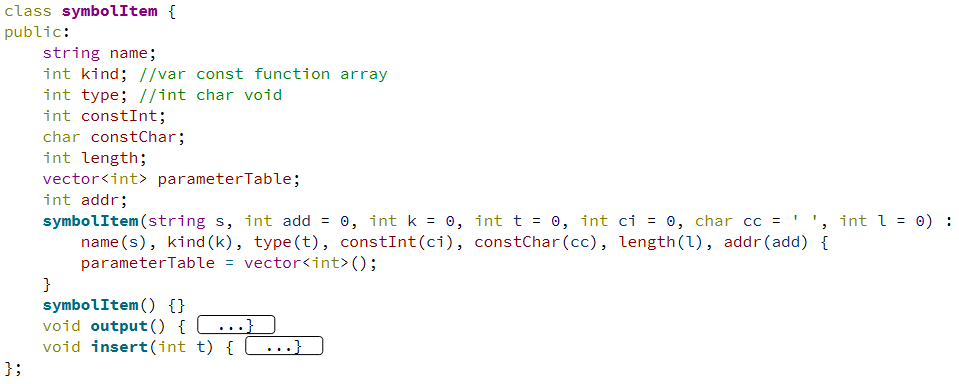


Figure 3. All the functions in lexical.cpp

1. **Syntax & semantic analysis**

**b1) symbolItem.h**

The symbolItem class is designed to store the information of every variable and function. Here is the code and the explanation of every parameter.



name is the name of the variable, constant, array and function;

kind is the type of them (variable, constant, array, function);

type is the type of the function return value;

constInt and constChar is the value of the constant;

length is the length of the array;

parameterTable stores the type of the parameters in functions;

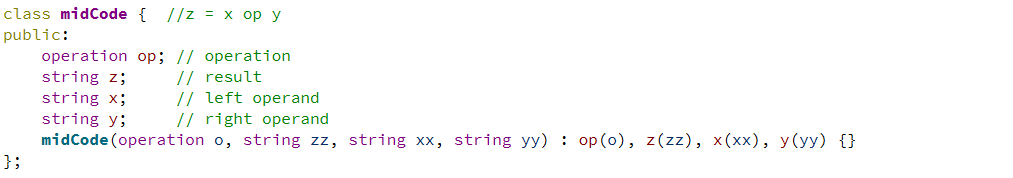
addr is the address of variables and arrays.

**b2) midcode.cpp & midcode.h**

Mid code is a mediate language between the SC and the MIPS code, which has a much closer relationship with the MIPS code. Here we first introduce the operations in mid code, which are shown in Table 2.

|  |  |  |
| --- | --- | --- |
| Table 2. Mid code operations | | |
| Operations | Operands | Comments |
| PLUSOP | mc.z = mc.x + mc.y | + |
| MINUOP | mc.z = mc.x – mc.y | - |
| MULTOP | mc.z = mc.x\* mc.y | \* |
| DIVOP | mc.z = mc.x / mc.y | / |
| LSSOP | mc.z = (mc.x < mc.y) | < |
| LEQOP | mc.z = (mc.x <= mc.y) | <= |
| GREOP | mc.z = (mc.x > mc.y) | > |
| GEQOP | mc.z = (mc.x >= mc.y) | >= |
| EQLOP | mc.z = (mc.x == mc.y) | == |
| NEQOP | mc.z = (mc.x != mc.y) | != |
| ASSIGNOP | mc.z = mc.x | = |
| GOTO | GOTO mc.z | Unconditional jump |
| BZ | BZ mc.z (mc.x = 0) | Conditional jump with false |
| BNZ | BNZ mc.z (mc.x = 1) | Conditional jump with true |
| PUSH | PUSH mc.z (mc.y) | Pass parameters |
| CALL | CALL mc.z | Call function |
| RET | RET mc.z | Return |
| RETVALUE | RETVALUE mc.z = mc.x | Return value |
| SACN | SCAN mc.z | Read |
| PRINT | PRINT mc.z mc.x | Print |
| LABEL | mc.z: | Label |
| FUNC | FUNC mc.z mc.x () | Function |
| PARA | PARA mc.z mc.x | Parameters |
| GETARRAY | mc.z = mc.x[mc.y] | Read from array |
| PUTARRAY | mc.z[mc.x] = mc.y | Assign the array |
| EXIT | EXIT | Exit main function |
| INLINEEND | INLINEEND mc.z mc.x | Inline function end |
| INLINERET | INLINERET mc.z | Inline function return value |

Notice that every line of mid code is composed of four compositions, so we designed a class to store all of that information.

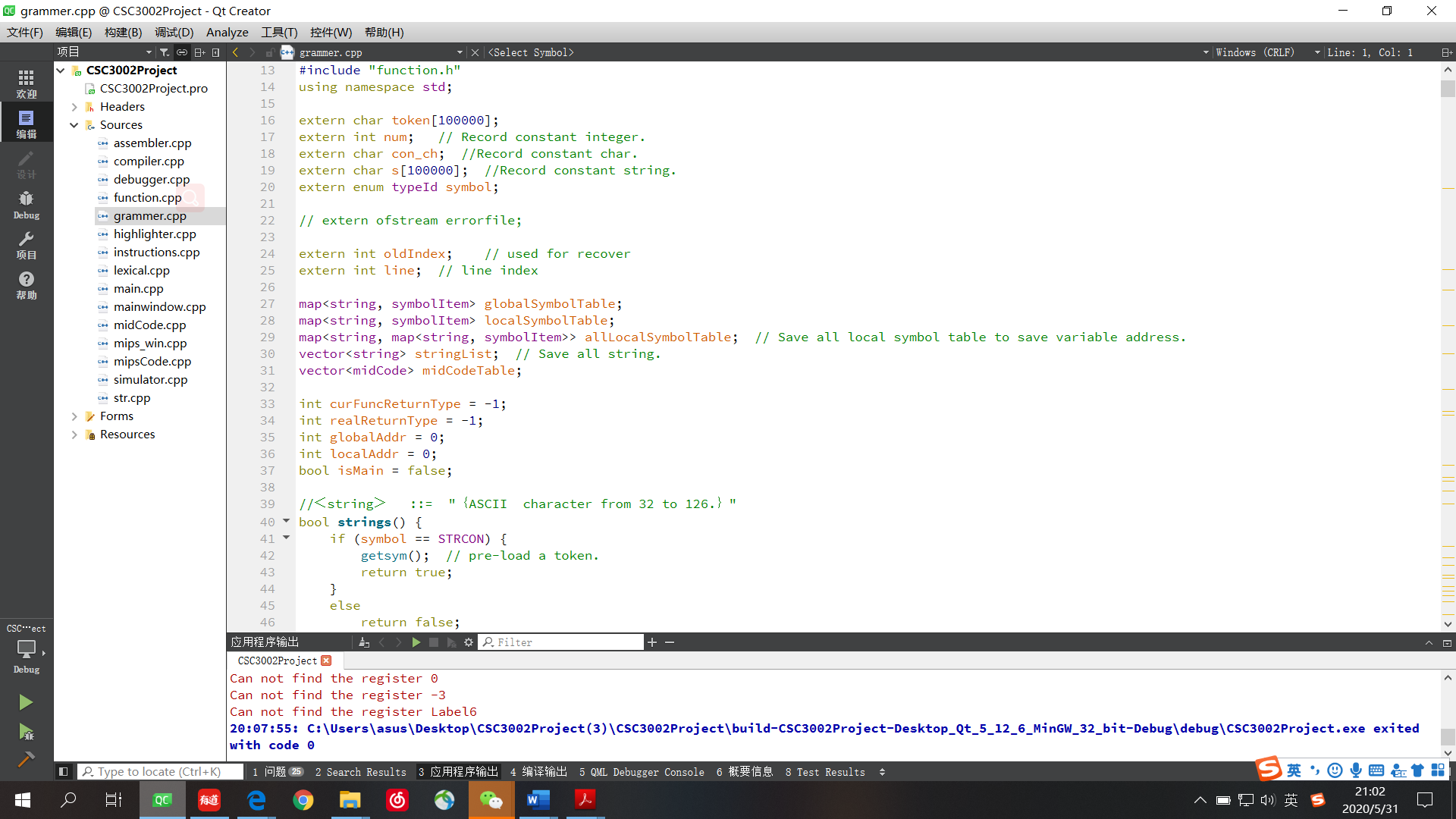


In grammar.cpp, lines of mid code will be generated after the syntax and semantic analysis. So, we write a function to integrate those lines of code and output a file called midCode.txt for further compilation (converting to MIPS).



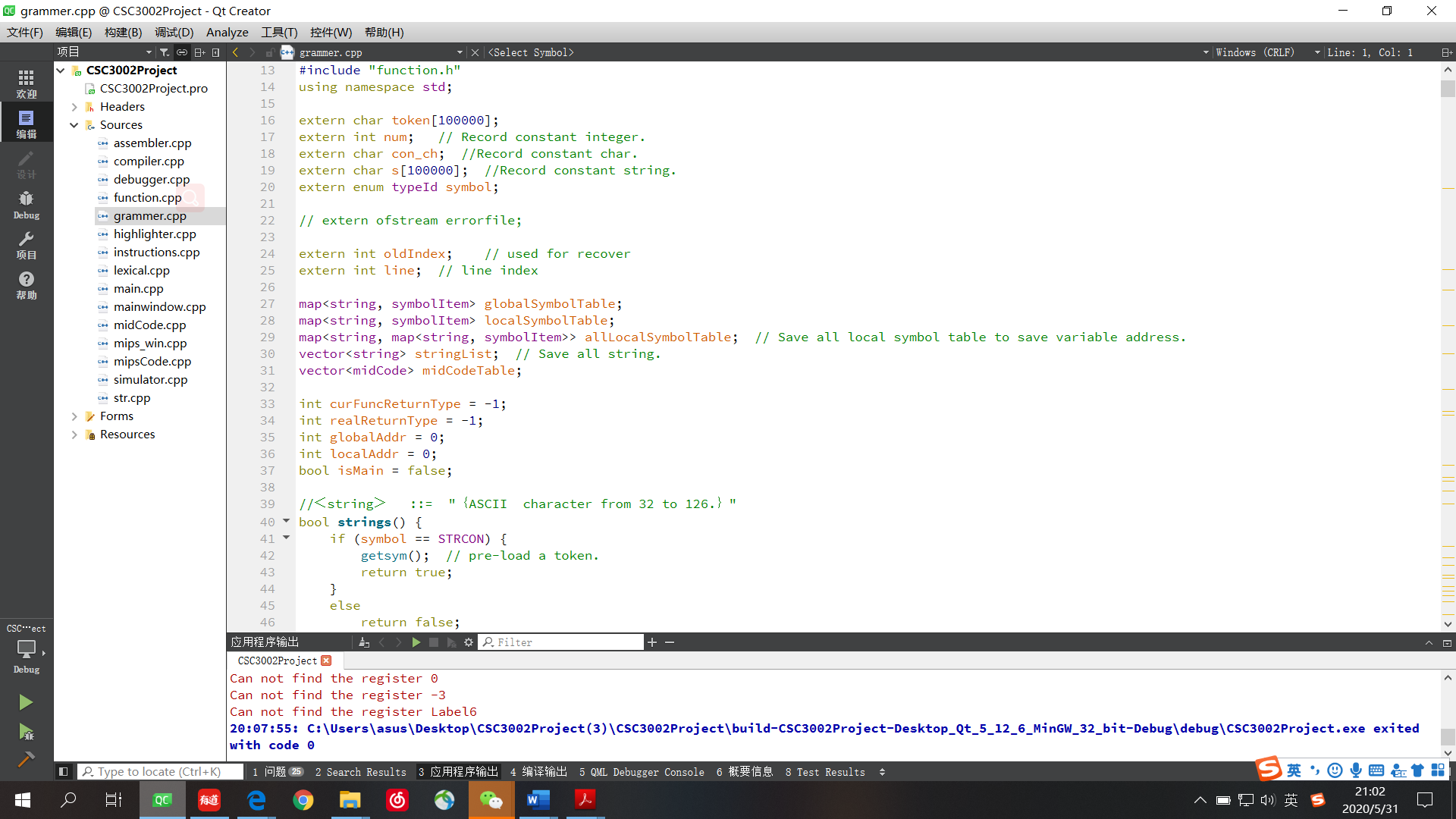
**b3) grammar.cpp & grammer.h**

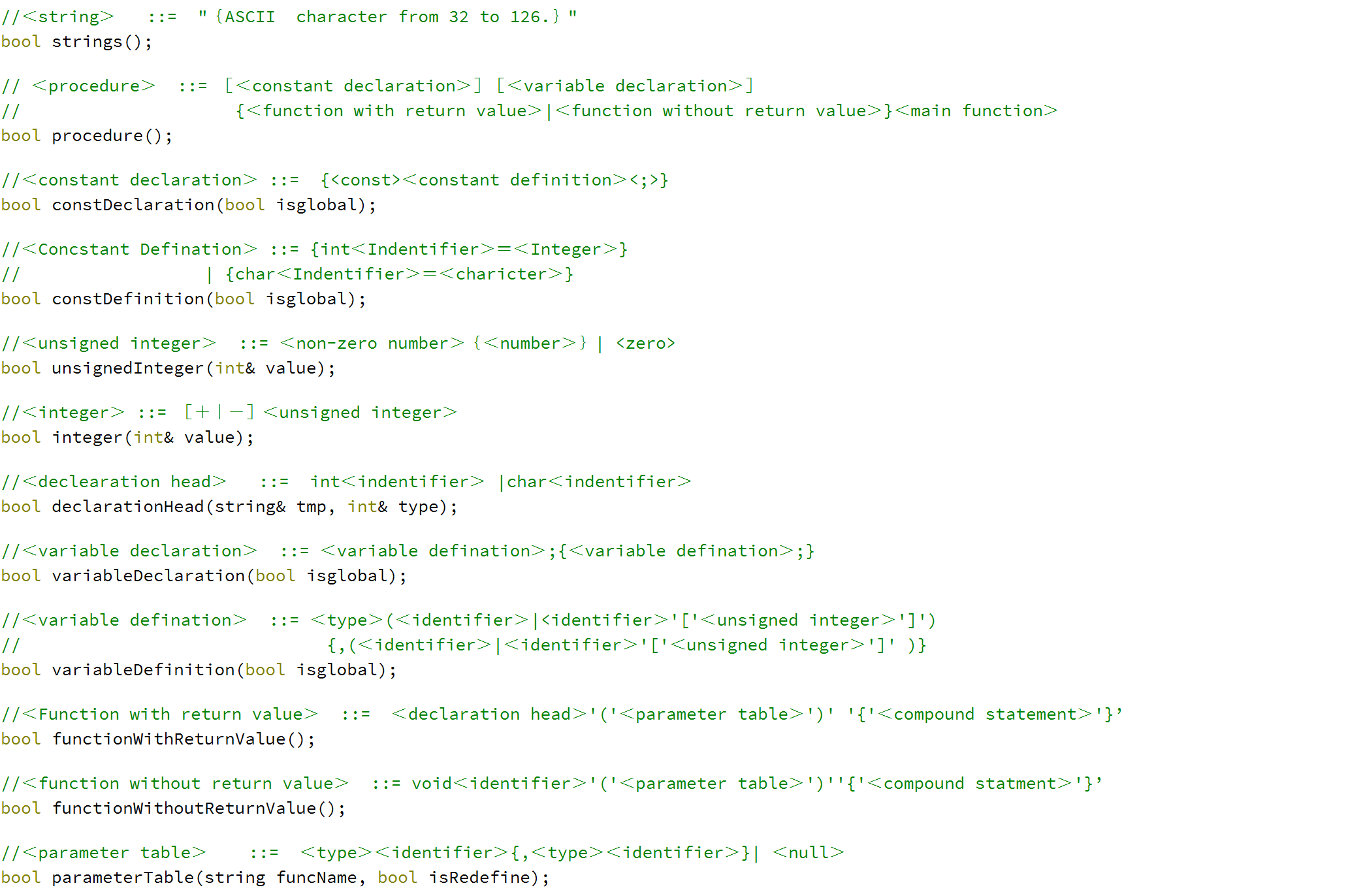
In this project, we choose top-to-down and deterministic method to do syntax analysis. We define several grammar rules for Simplified C language and using recursion to do semantic analysis. It will analysis the current token and judge if there exist a grammar rule that can match the token. If there is no grammar matched, it will raise error and output the line number of error code. If it matches the grammar rules, a line of mid code will be generated and stored in the following data structure for further code generation.



When analyzing the identifiers, it needs to pre-load a token. If the next token is left bracket, then the current grammar should be function definition. In this case the analyzer should be rollback to the token before int, while result in the confliction of variable definition and definition of function with return value. To solve this problem, we design a function void retractString(int oldIndex) to go back to a certain position.

After analyzing the identifiers, we need to store them in some data structure for constructing the data section and function symbols in MIPS code. Here are the structures we used to store all the functions and global and local variables.



The whole pack of functions appearing in grammar.cpp are shown in Figure 4. 

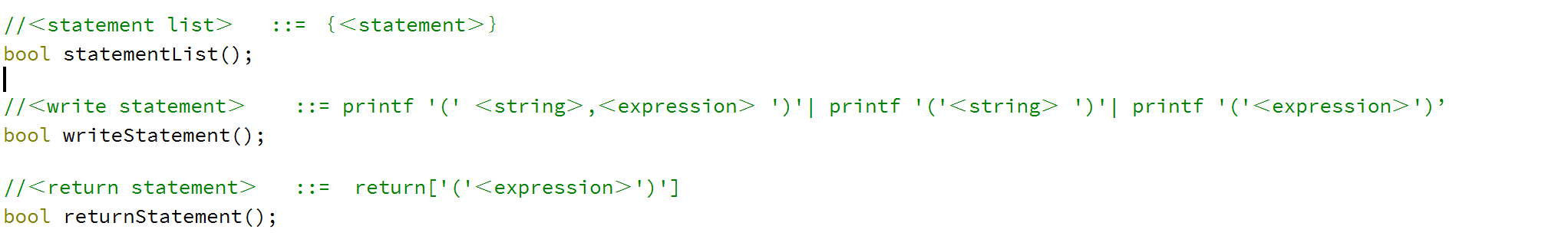
 

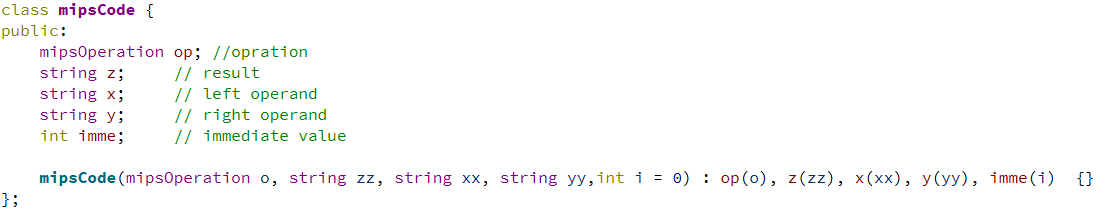
Figure 4: grammar rules for Simplified C language.

**b4) mipsCode.cpp & mipsCode.h**

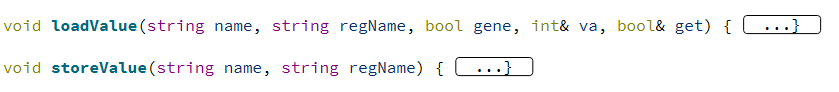
The final step is to convert mid code to MIPS code. The supported instructions MIPS code are shown in Table 3.

|  |  |  |
| --- | --- | --- |
| Table 3. MIPS instructions | | |
| Instructions | Operands | Comments |
| add | add rd, rs, rt | rd = rs + rt |
| addi | addi rt, rs, imm | rt = rs + imm |
| sub | sub rd, rs, rt | rd = rs – rt |
| mult | mult rs, rt | rs \* rt |
| div | div rs, rt | rs / rt |
| mflo | mflo rd | move from lo to rd |
| mfhi | mfhi rd | move from hi to rd |
| sll | sll rd, rt, shamt | shift left |
| beq | beq rs, rt, label | branch to label if rs==rt |
| bne | bne rs, rt, label | branch to label if rs!=rt |
| bgtz | bgtz rs, label | branch to lable if rs > 0 |
| bgez | bgez rs, label | branch to lable if rs >= 0 |
| bltz | bltz rs, label | branch to lable if rs < 0 |
| blez | blez rs, label | branch to lable if rs <= 0 |
| j | j target | unconditional jump to target |
| jal | jal target | jump and link |
| jr | jr rs | jump to the address in rs |
| lw | lw rt, address | load word |
| sw | sw rt, address | store word |
| syscall | syscall | system call |
| ori | ori rt, rs, imm | rt = rs | imm |
| lui | lui rt, imm | Load the lower halfword of imm into the upper halfword of rt. |
| dataSeg | .data | data segment |
| textSeg | .text | text segment |
| asciizSeg | .asciiz | store the strings |
| label | label | store the labels |

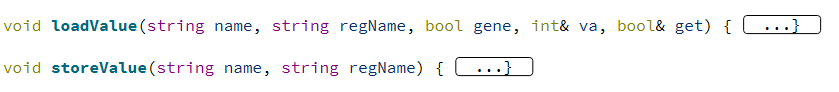
Every line of MIPS code can be divided into five parts, so we also designed a class for MIPS to store all of those data.



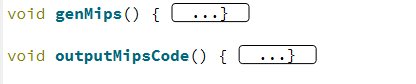
And here are the functions that finally translates mid code to MIPS code:



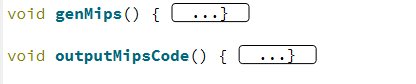
(Load the value from the main memory to the registers)



(Store the value from the registers to the main memory)



(From the middle code to MIPS code)



(Output the MIPS code to the mips.asm file)

* 1. **Assembler & Simulator**

1. Simulate memory

First, malloc 6MB memory using malloc() function. It will return an void pointer. This pointer will be regraded as the base address of the simulator memory, which is 0x0040000. The relationship between the real address and the “fake address” (in MIPS simulator) is as follow:

We will use this formula to transform the real address and fake address.

1. Maintain memory space

Define an int array: **int registers[34];** //registers[32]=$hi and registers[33]=$lo

Then initialize the register:

**registers[0] = 0; //$zero**

**registers[28] = 0x00500000; //gp**

**registers[29] = 0x00A00000; //$sp**

**registers[30] = 0x00A00000; //$fp**

1. Maintain a code PC

PC is define as a *int* pointer because every MIPS instruction will take 32 bit. It is initialized with base address we malloced in Step 1.

1. Maintain a code section.

We will mainly use the program in Project 1 to assemble the MIPS instructions into binary form. Notice that in Project 2 the start address of code is 0x00400000 instead of 0x00000000. Thus some places of program have been adjusted. Also I add some new output functions so that the program can assemble the binary and store it into memory directly rather than output a text file first.

1. Maintain a data section.

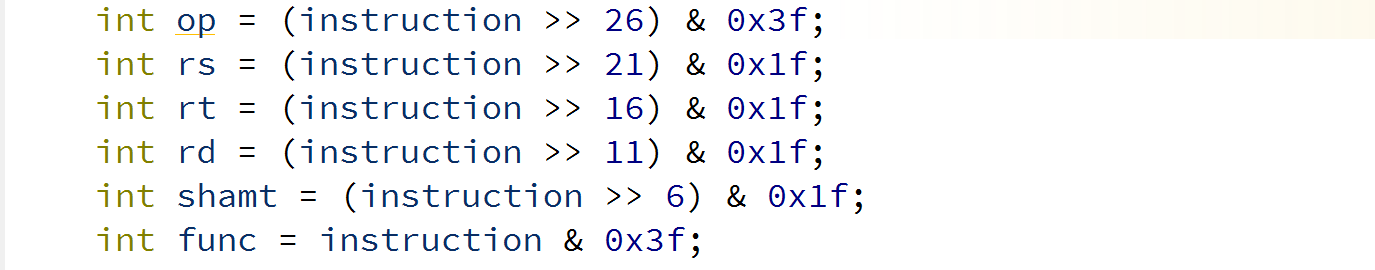
Load the data. Every data will occupy the full block (4 bytes).

1. Maintain a stack section.

&sp = 0x00A00000;

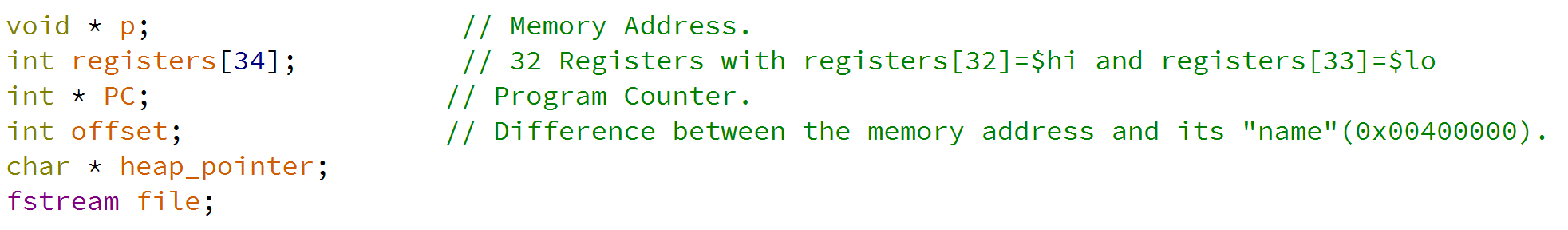
1. Simulate the execution.

By dereference the PC, we can take the binary code line by line. I mainly use the shaft operation (>> and <<) and & operation to split the instruction according to the number of bits of op, rs, rt, rd and so on.



Next, using “switch … case …”, I can call the corresponding functions.

1. End simulate
2. When assemble the MIPS code, the assemble function will return the end address of the MIPS code. Thus if PC reaches the end address of exit()/exit2() is called the program will stop.
3. Important Global variables



* 1. **Editor**

The editor in this project is a main window which is for users to operate and present the result. To be more precise, there are several parts of the user interface. The final editor is presented in Figure 5.

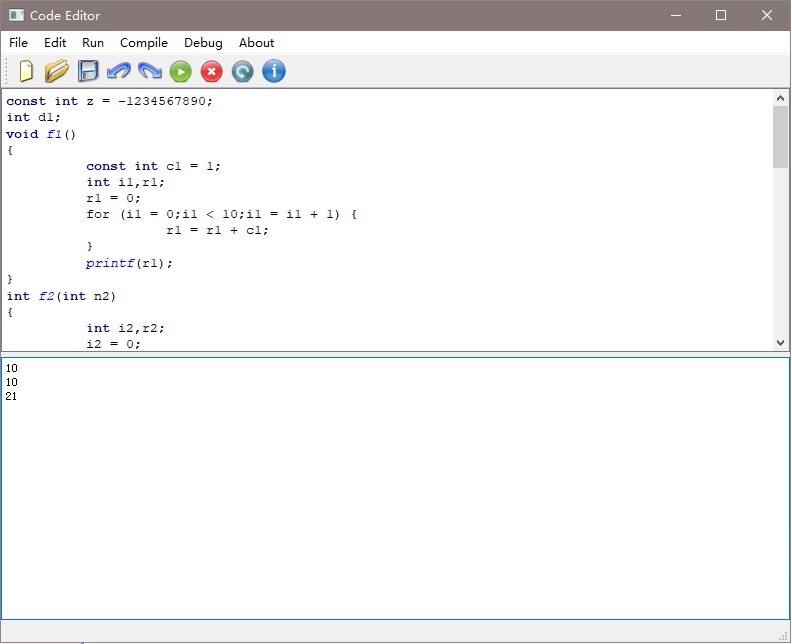


Figure 5. Editor

The top part is the title bar. And it consists of the title of the window and the control buttons. The next part is the menu bar. All commands this project contains are listed here. And all functions are divided into six sections, *File, Edit, Run, Compile, Debug, and About*. Amid each section, after clicking the button, we could get more detailed commands. The specific notation and interpretation are shown in Table 4.

|  |  |  |
| --- | --- | --- |
| Table 4. Buttons in the Editor | | |
| Main Commands | Second level | Interpretation |
| *File* | *New* | This command is to clear all current codes in the editor. And it would start a new script. |
| *Open* | This command is to make users open their own files in the editor. |
| *Save* | This command is to save all codes in the editor as an *.txt* file, which is the input of the compiler. |
| *Save as* | The function of this command is basically the same as *Save*. The addition is that the codes will be saved in a new file at any address. |
| *Refresh* | This command is to reboot the editor. Its target is to remove all codes, outputs, and value of assigned variables. |
| *Exit* | This command is to quit this editor. |
| *Edit* | *Cut* | This command is to cut the selected codes, which are abandoned or removed to other place. |
| *Copy* | This command is to copy the selected codes. |
| *Paste* | This command is to paste codes from last cutting or copying at cursor. |
| *Redo* | This command is to recover the contents or action that are deleted by *Undo*. |
| *Undo* | This command is to cancel, annul, or reverse last action or its effect. |
| *Run* | *Run in terminal* | This command is to immediately execute codes in the editor. |
| *Compile* | *Compile to other codes* | This command is to initiate another window showed in figure-3. This window would provide both the middle codes and MIPS of current codes in the editor. |
| *Debug* | *Debugger* | The details of this command would be introduced in section 3.5. |
| *About* | *Information* | This is an announcement to users. It declares the course and developers involved. |

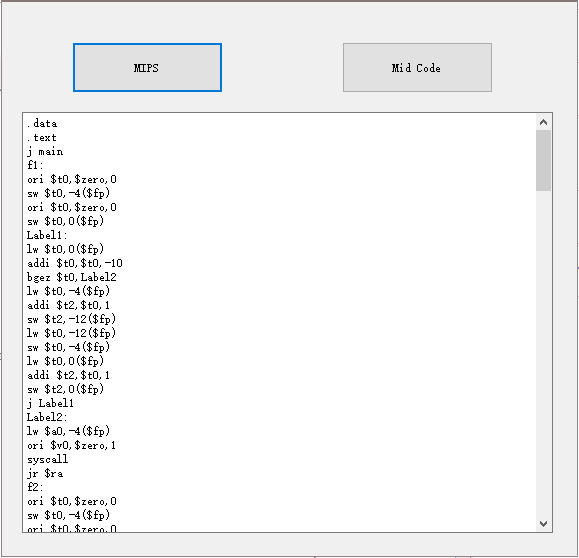
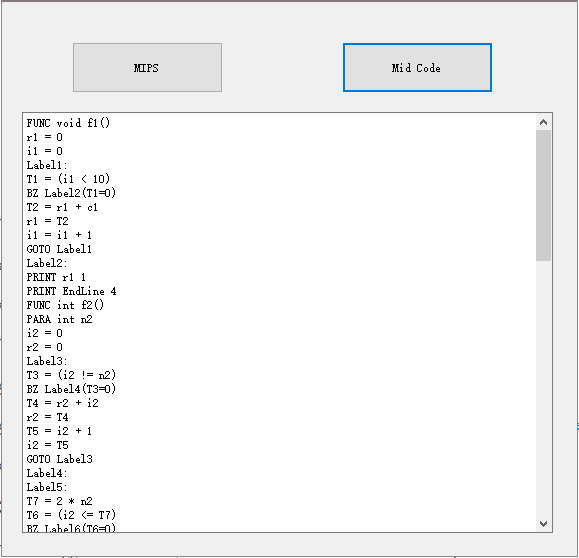
 

Figure 6. User Interface of "*Compile to other codes"*

The third part is the shortcut-key bar, including nine icons. The shortcut keys are still achieving the former functions. From left to right, it represents new, open, save, undo, redo, run in terminal, refresh, and information. Below shortcut keys, there are two blank zones. The top one is regarded as editing zone, in which the code typed or the file opened. The other is the terminal where the outputs are shown. This editor performs a simple syntax highlighting function, which means that the codes in editing zone will be highlighted in different color and format according to C++ syntax. The involved key words are *char, class, const, double, enum, explicit, friend, inline, int, long, namespace, operator, private, protected, public, short, signals, signed, slots, static, struct, template, typedef, typename, union, unsigned, virtual, void, volatile,* and *bool*.

**3.5 Debugger**

This debugger is initiated by the button at menu bar of editor. It would initiate a new window like compile. The demonstration is presented as Figure 7.

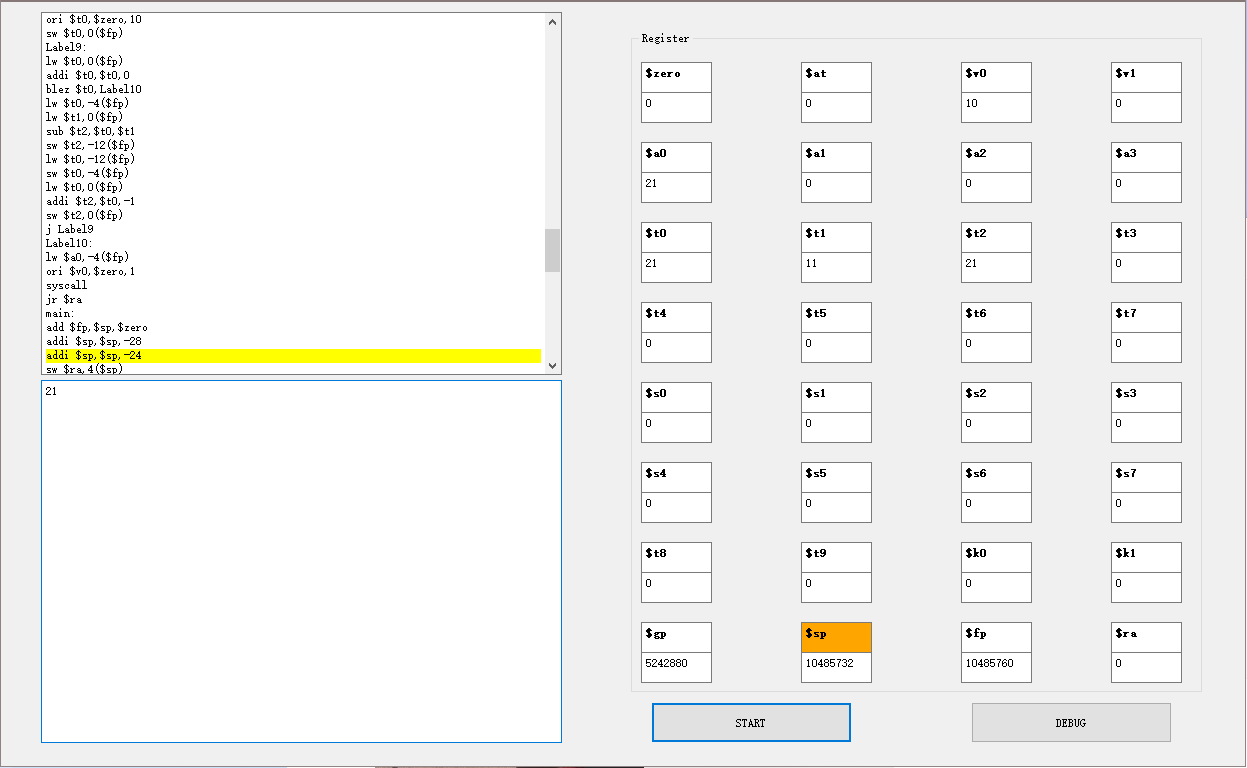


Figure 7. User Interface of Debugger

There are two textboxes on the left side of this window. The one on the top is to display the corresponding MIPS codes, the other one is the terminal. The first step is to click the START button. The MIPS codes will be displayed, and at the same time, the first instruction line to be debugged will be highlighted. Then, we could continue debugging by clicking the button DEBUG for some certain lines of MIPS until there is no other line to be debugged. And each time the DEBUG button is clicked, the code line in the debugging process will be highlighted.

To be more specific, when the button START is clicked, the debugger calls functions. In the function: int init\_debugger(), the memory and registers will be initialized and the MIPS code will be convert to binary form using assembler, just as same as simulator. Also, it will scan through the MIPS code and initialize two maps by using the following instructions:

map<int, int> address2line & map<int, string> line2instruction

Here address2line will map the instruction address to the line index of instruction in MIPS file, and line2instruction will map the line index to the binary form of instruction. The first instruction (usually “j main”) will be highlighted.

Next, user can use button DEBUG to run the MIPS instruction line by line. The debugger will execute the highlight instructions and update the target register value. The background color of input register will be changed to yellow. The background color of output register will be orange. In that case, user can check the value of each register and find if there is any bug in the MIPS file.

1. **Contributions**

|  |  |
| --- | --- |
| Name | Work |
| Yuzhe Jin | Syntax and semantic analysis, assembler and simulator, debugger, debugging |
| Wentao Guo | Editor’s buttons, Debugger |
| Yuan Gao | Lexical analysis, debugging |
| Yuhao Du | Mid and MIPS code generation, debugging |
| Jiaqi Liu | Editor’s main window |
| Hanshen Jing | Editor’s buttons |

1. **Reflections**

In implementing the compiler, we have encountered so many troubles especially when comprehending syntax and semantic analysis for LL(1) grammar. Thus, we have to drop some of the language characteristics that were mentioned in the proposal:

1. We do not support short and struct for data types.
2. We do not support continue and break for loop statements.
3. We do not support \*(deference) and & for data operations.

Also, we have not optimized our program, and have not figure out the reason of a bug in the compiler, which is that sometimes some strings may not be able to print out.

However, we understand more about program language’s syntax and grammar and gain more practical skills about group programming. When we were combining the editor and the compiler, there was a tremendous bug that the editor can only compile once for any file. Later we found out that it was because we did not restore the global variables in the compiler. It taught us a lesson that we must think about the other parts in advance in group programming.