

# Small-C Compiler Implementation

## Report of CS215 Course Project 2

Author: 左楠 ID: 5120109159

Depart. of Computer Science and Engineering, SJTU  
Shanghai, People's Republic of China  
devinz@sjtu.edu.cn

**Abstract**—This is the report of course CS215 project 2, which is aimed to implement a simple compiler that can check and translate a high-level language Small-C to an assembly language MIPS. In project 1, we have already implemented the lexical analyzer and the syntax analyzer with the assist of two powerful tools - *flex* and *yacc*. In project 2, I tackled the more challenging task by coding in C++ and solved all the problems on my own.

**Index Terms**—compiler principles, Small-C language, MIPS, syntax-directed translation, intermediate code, code generation.

### I. INTRODUCTION

The whole project can be divided into two major parts: one is the front end of the compiler, including the lexical analysis and syntax analysis that we have done in project 1, as well as semantic analysis and intermediate code generation. The other one is the back end of the compiler, namely machine-code generation, including instruction selection, register allocation and so forth.

In the front end implementation, I established a parse tree at the same time with syntax analysis, and then traversed the parse tree in a top-down way to realize syntax-directed translation and intermediate code generation. My intermediate code largely takes the form of three-address code mentioned in the textbook, but still has some disparity from that one, since I hope to pass more information to the back end of my compiler. Besides the IR code, the symbol table produced in the front end is another fruitful result that is passed to the back end, and was realized by using hashing technique in my implementation.

In the back end implementation, I established other data structures to realize the idea of register allocation mentioned in the textbook (register descriptor and address descriptor). The back end of my compiler can analyze the IR code by using string manipulation, and accomplish the code generation task by consulting the register descriptors and address descriptors (stored in the symbol table).

In the end of the project, my compiler can pass all the three simple test cases provided by the mentor, as well as some special tests designed by myself.

### II. INTERMEDIATE-CODE GENERATION

#### A. Symbol Table Implementation

To illustrate my symbol table design, I should first introduce how I classify the identifiers in Small-C language. There are three kinds of identifiers in Small-C: *variables*, *structures* and *functions*. An integer variable is doubtless a *variable*, so is an n-dimension integer array. Fields of a structure type object are also *variables* - that is to say, every structure object will be considered as a collection of integer variables in my implementation. By using such classification, I established my symbol table based on three hash tables: one is for *variables*, another is for *structures*, and the other one is for *functions*.

Structure *Var* is a data structure that can record the information of a *variable* in the symbol table. Field *scope* records the scope of that *variable*: for global *variables* (including integers, arrays and structure fields), it is an empty string; for local variables declared somewhere in a function definition (including the *main* function), it will be a string ended with the function identifier, starting with a list of block labels that the *variable* lies in, separated by char '@'. For example, if integer *x* is a local variable defined in block BLK\_3, BLK\_3 lies in BLK\_1, and BLK\_1 is the body of function *int funct()*, then the scope of such *x* should be "BLK\_3@BLK\_1@*funct*". By using string manipulation techniques and the method introduced in the next section, we can record and recognize the scopes of all the *variables* defined in Small-C. Structure *tag* is help to record more information about the *variable*: for an integer or integer array, it will be a special string "?"; for a field of a structure object, the tag will be the identifier of that structure object; for a parameter defined in a function, this field will be a string starting with char '#' followed by the position of that parameter. Field *id* is the identifier string of this variable (integer name, field name or parameter name). Field *size* is a head of a linked-list that records the sizes of each dimension if it is an array, in order to help translate the index of an array variable in IR code generation. For example, array *int arr[100][200][50]* has a size list containing 100, 200 and 50. Field *next* is a pointer to implement open hashing. Apart from the fields mentioned above, the remaining three fields are aimed to help code

generation in the back end: *mem* marks whether the data of that variable in memory is currently valid; *reg* uses a 32-bit integer to mark whether the 32 registers of MIPS register file contain that variable; and *addr* records the memory address of that variable (address of the first word for an array). Global variables are stored in the static data section, so *addr* is the relative address from the first global variable; local variables are stored in function stacks, and thus their *addrs* are relative addresses from the base of the stack, where *\$fp* points to.

Structure *Strt* helps to record a *structure* type in the symbol table. Fields *scope*, *id* and *next* are the same as those mentioned in *Var*. Field *fld* is the head of a linked-list that contains the field names of that structure.

Structure *func* helps to record a *function* in the symbol table. Field *id* and *next* are all the same as mentioned above. Field *argc* records the number of the arguments that should be passed when such a function call occurs. Field *spc* records the space that this function needs in the stack section in memory.

For all the three structure types, I established corresponding open hash tables respectively, to record *variables*, *structures* and *functions*, and these three hash tables are encapsulated into a class named *Hash*. My *smallc.y* program declared a *Hash* object named *tbl* to work as the symbol table. In parse tree traversal, it inserts a new element by using methods *insVar*, *insStrt*, and *insFunc*, whenever a new identifier is legally declared; and whenever an identifier shows up in an expression, it will consult the hash tables to check whether it exists in the symbol table by using methods *srchVar*, *srchStrt*, and *srchFunc*. In this way, I could easily implement type checking, including finding the variables and functions re-declared, and variables and functions that are used without declared. Apart from these, my class *Hash* also has methods to check whether integers or one-dimension arrays are legally initialized or assigned by traversal corresponding parse tree nodes. It can also check whether the number and type of arguments passed to the function coincide with the parameters in function definition. And in code generation, it can also inform the back end the space that a function needs in the stack section and return a *Var* type pointer of the corresponding symbol table entry when the compiler encounters a variable string (*tag-id* pair). Therefore, we can say the symbol table object *tbl* is the most active role in my whole program from the front end to the back end.

### B. Scope and Address Determination

Apart from the symbol table class *Hash* mentioned in the previous section, a handful of global variables are strongly active in the front end. For example, *nymHelp* helps the compiler to name a new temporary variable when it needs by using a string starting with “@tmp\_” followed by a natural number. A similar case is *lblHelp*, which helps the compiler generate a block name or label in branches or loops, by using a string starting with “BLK\_” followed by a number or starting with “Label\_” followed by a number.

The most important global variable in my program actually is the string *scp*, which represents the current scope in both the front end work and the back end work. As mentioned above, in the front end, for global scope, *scp* is empty; whenever the

compiler enters a function *scp* will become the function name; and whenever it enters or leaves a block, it will insert or remove the block label in the head of *scp* string. When inserting a variable newcomer, my symbol table will search for the variable that has the exactly same scope, tag and id to check out re-declaration; but in expression translation, it will find the variable entry that has the same tag and id with the longest scope suffix - that means the most recently declared. By using this method, we can recognize the scope of a variable and also mark the scope of a segment of the IR code to inform the back end of the compiler. The back end will use a stack (a class *StrStack* in my implementation) to help determine the current scope in code generation since variable scopes have relationship similar to something like brackets.

Another important global variables are *addrLocal* and *addrGlobal*, which help to determine the memory address of a variable newcomer, and help to calculate the space a function needs. I must say sorry for the reason that these two variables are not correctly named. Actually, *addrLocal* helps to calculate both local and global variables in the procedure of IR code generation, and *addrGlobal* is just its assistant. In global scope, *addrLocal* will increment itself to mark the position of each global variable. Whenever it enters a function (attention! not a statement block), *addrGlobal* backups the current value of *addrLocal*, and *addrGlobal* resets to zero and start to calculate the position of local variables stored in a stack. When the function definition finishes, *addrLocal* is assigned to *addrGlobal* again to continue help determine the positions of global variables. The space allocated for the static data section is 4 times the *addrLocal* record (every integer has 4 bytes) plus 20 (spare space); and the space allocated for a function is 4 times the *addrLocal* result plus 8 (4 bytes for *\$fp* storage, 4 bytes for *\$ra* storage). In this way, we can provide all the information of space allocation and data read/write to the back end of the compiler.

### C. Semantic Analysis Achievement

Actually, my compiler can do all the semantic analysis mentioned in the project instruction. The implementation of this is based on my symbol table as well as other functions defined in *smallc.y*. I would list these achievements here combined with the screen-shots of test cases in the appendix I of this report.

A variable that is used without declared will be checked out, as illustrated in FIG I.

A function that is called without defined will be checked out as illustrated in FIG II.

A structure that is used without declared will be checked out as illustrated in FIG III.

A variable that is re-declared in the same scope will be checked out, but in different scopes are correctly legal, as illustrated in FIG IV.

A function that is re-declared will be checked out as illustrated in FIG V.

A structure that is re-declared in the same scope will be checked out, but in different scopes are allowed, as illustrated in FIG VI.

Reserved words cannot be taken as identifier in FIG VII. Actually, this work was done by *flex* lexical analyzer generator.

A program must has an entrance named *main* function, illustrated in FIG VIII. This was implemented by calling *tbl.srchFunct* method to search the main function at the end of the translation.

Number and type of the arguments passed to a function will be checked by the symbol table class, illustrate in FIG IX.

Operator [] can only be used in an array in FIG X. Actually, my compiler can do array type checking of all dimensions. That means a 3-dimension array has different type with a 2-dimension array. This was realized by checking the *size* field of the structure *Var*.

Dots can only used in struct type objects, as illustrated in FIG XI, due to the way I look at structure fields mentioned above.

FIG XII and FIG XIII illustrated statements “break” and “continue” can only be placed in a for-loop. The implementation of this will be introduced in the next section.

FIG XIV illustrated that some types of right-values are illegal. This was done in expression translation discussed in the next section.

The condition of an IF statement cannot be empty, but those of a for loop can be. This was illustrated in FIG XV.

The last screen-shot shows that only integer type variables can involve in expressions, and this was actually realized by the syntax analyzer generator *yacc*.

#### D. More Translation Details

As mentioned above, the syntax-directed translation and IR code generation is based on the traversal of the parse tree established in syntax analysis. The traversal is actually done by both the functions in *smallc.y* and methods of the symbol table class. In this section, I will introduce some important details about translation functions defined in *smallc.y*.

The first one is about how I deal with “break” and “continue” statements. This work was done when the parse tree node labeled as STMTBLOCK, STMTS and STMT. Two more arguments are passed to the functions that traverse those nodes, one is named *preCont*, and another is named *preNext*. *preCont* is a string, actually a label, that indicates where should we “goto” if there is a “continue” statement inside this section of Small-C code. If it is NULL, seeing a “continue” statement will be considered as a semantic error, and I call it “CONT Error”. I use the same method to test “break” statement with argument *preNext*, which indicates where should we jump if a “break” shows up in this section of Small-C code.

The second thing I wanna introduce is how I translated expressions. This seems like an easy task, but in reality it took me a whole day to debug!! I designed a function named *transExp* to traverse a node marked as EXP, another named *transExps* to traverse a node marked as EXPS. An EXP can either be reduced to an EXPS or an empty statement, and in some situations it cannot be empty in reality (like the condition of an IF statement). In *transExp*, I pass a string argument called *dst*, which indicates what variable that expects to be assigned to the outcome of this expression, empty string if there’s no

such destination variable. If *dst* is not empty (it must be a temporary variable produced outside this function), it shows that we expect this EXP must be reduced to an EXPS to get an outcome, otherwise a semantic error has occurred. If *dst* is empty, and at the same time EXP has a sub-node EXPS, we get the outcome of this EXPS and get a temporary variable for *dst* by using tool function *getDst* to be assigned to this result. This guarantees that *transExp* can always get a temporary variable that contains the outcome of this node, which much facilitate the work of the back end (even though this may produce useless temporary variables and waste register usage and memory space). Function *transExps* has int reference argument *&val*, string argument *\*rev*, and bool reference argument *&cst* to return the outcome of the current expression. If *cst* returns true, it indicates *val* gets a constant outcome; otherwise *rev* gets a variable outcome string (tag-id pair in IR code). In this function, we first recursively translate the sub-expressions and judge the *cst* by their outcomes and the production rules. If we can get a constant outcome, we must calculate and return the constant result in *val*; otherwise we will generate corresponding three-address instructions, get necessary temporary variables and assign to *rev*. If this expression is a leaf of the expression tree, we will call function *transAtom* to deal with it. The function *transAtom* has the same arguments and corresponding functions as *transExps*. However, *transAtom* has a bool return value that indicates whether this atom of expression can be considered as a left-value. This helps us check out semantic error when a right-value appears on the left side of an assignment statement.

The last thing about this section is the way I translate the index of an array variable - a function named *transArrs*. In IR code generation, I change all n-dimension array variables into one-dimension form, the new index is calculated by a hidden expression:  $base + \dots((i_1 * n_2 + i_2) * n_3 + i_3) * \dots$ . This expression can be iteratively calculated in a loop and by calling function *transExps*, and thus constant index can also be resolved in the compiling stage.

### III. MACHINE CODE GENERATION

Since all about scope determination and address calculation has been discussed in the previous sections, things that can be stated here are far less. In my program, instruction selection is just string manipulation, and there’s nothing to talk about that. What I should introduce here lies in the work of register allocation.

#### A. Register Descriptor

In code generation, I designed more data structures to facilitate register allocation. Class *Reg* represents a register descriptor that records all the valid variable data that stored in the corresponding register. Attribute *idx* is a register index in MIPS architecture, and helps to map this descriptor to a register name in *regname[]*. Attribute *ref* is the head of a list of variable symbol table entries (stored as *Var\**). In register allocation, a register descriptor will search, insert or remove variable pointers properly when we wanna issue store-word or

load-word instructions, or a register is chosen to be a destination register in an operation. These actions are coded strictly according to the rules introduced in the textbook.

Another thing is about the implementation of the *getReg(l)* function introduced in the textbook. In my compiler, I chose register from 8 to 24 as registers that can contain variables, and register 25 is specially allocated to constant values when they are needed. Function *getVarReg* gets a register index for a variable, and function *getCstReg* allocate register \$t9 to a constant value. In those functions, load-word and store-word instructions are issued properly when they are necessary. In expression code generation (including binary operator expressions, unary operator expressions and assignment), we first call the functions above to get registers for the operands and issue the MIPS arithmetic operation instruction, and finally changes the descriptor of the destination register.

The last thing about register descriptors is that they can only work in a basic block. So in code generation, whenever I meet a head of a new basic block (like a label, a jump instruction, a function call or a return instruction), I will clear all the register descriptors and store all the register data if necessary.

### B. Array Content Sentinels

A fatal bug about the register allocation method introduced above is that we cannot always get a *Var\** symbol table entry when some data in memory is needed. For example, when we need an array variable *arr[100]*, we can calculate its memory address but we cannot get a symbol table entry since the whole array is stored in the same symbol table entry. To tackle this problem, I introduced another set of three *Var\** sentinels to represent such an array variable operand, which is called *arrHelp[]*. Another integer *arrFlag* marks the next sentinel that can be used - if it is 3, all the three sentinels are all occupied. In this way, we can look at array variables the same way as we look at an int variable, but we must remember to write back these array elements when the instruction is finished, since in the next instruction, these sentinel pointers may be used again.

### C. Function Call Procedure

The last thing about MIPS code generation is the work that I do whenever a function call happens. This procedure is written in a function named *genFuncCall*, which is done by the caller itself.

First, I read all the “param” instructions and issue corresponding assignment instructions to store all the arguments at the bottom of the new stack.

Then, since there will be a head of basic block when *jump* and *link* appears, I clear all the register descriptors.

Next step, I store \$fp and \$ra in the top of the old stack for backups, and change \$fp and \$sp to new values to enter the new stack.

As you have expected, a *jal* instruction is issued here.

After *jal* instruction, I restore \$fp, \$sp and \$ra back to its original values. We are now in the old stack again.

Last but not least, I assign the data \$v0 to the variable waiting for the return value. This is once more an assignment statement, which needs *getVarReg* and descriptor actions.

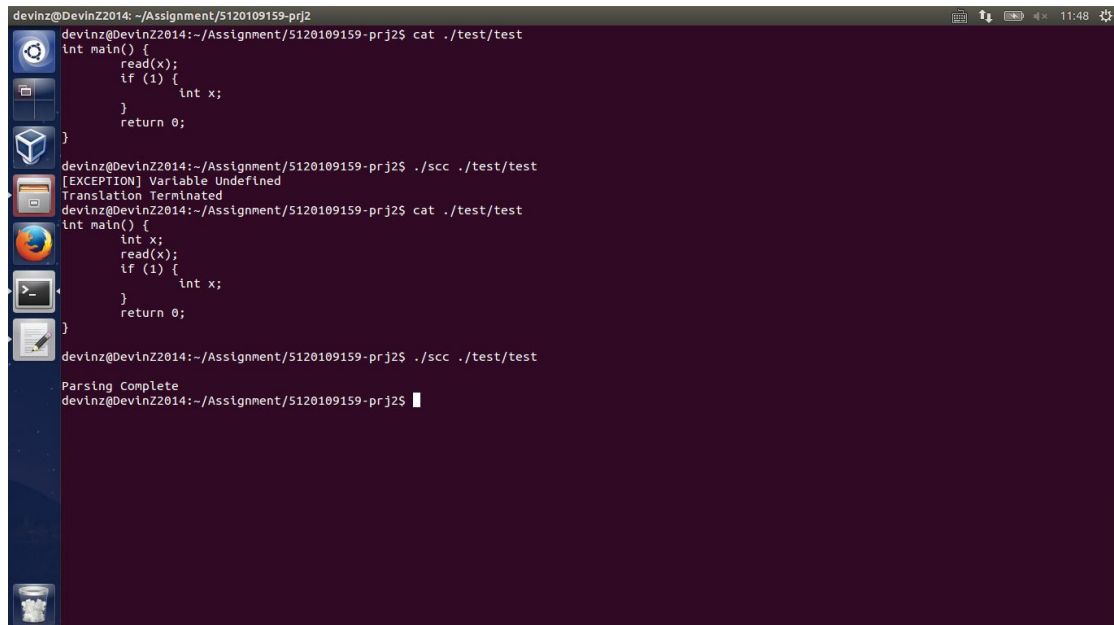
## IV. PROJECT INSPIRATIONS

To sum up, I should say, this is a very challenging but also exciting project. The amount of work and difficulty of this project surpass all the projects that I have ever done (since I am still a neophyte in coding). In this project, I really got a lot of harvests, including a refresher in C++ and data structure, some lessons taught from coding and debugging, and also a deeper knowledge of compiler principles and MIPS architecture. Now I feel more than lucky to have chosen this course, though I was really intimidated by the overwhelming task at first.

At the end of this report, I wish to express gratitude to all the mentors of this project for their helpful instructions and hard work. Thanks to Prof. Wu for his excellent teaching in course CS308. Also, thanks to all my friends that offered me helpful assists and suggestions.

## APPENDIX I: SCREEN-SHOTS OF SOME TEST CASES

FIG I. VARIABLE UNDEFINED CHECKING

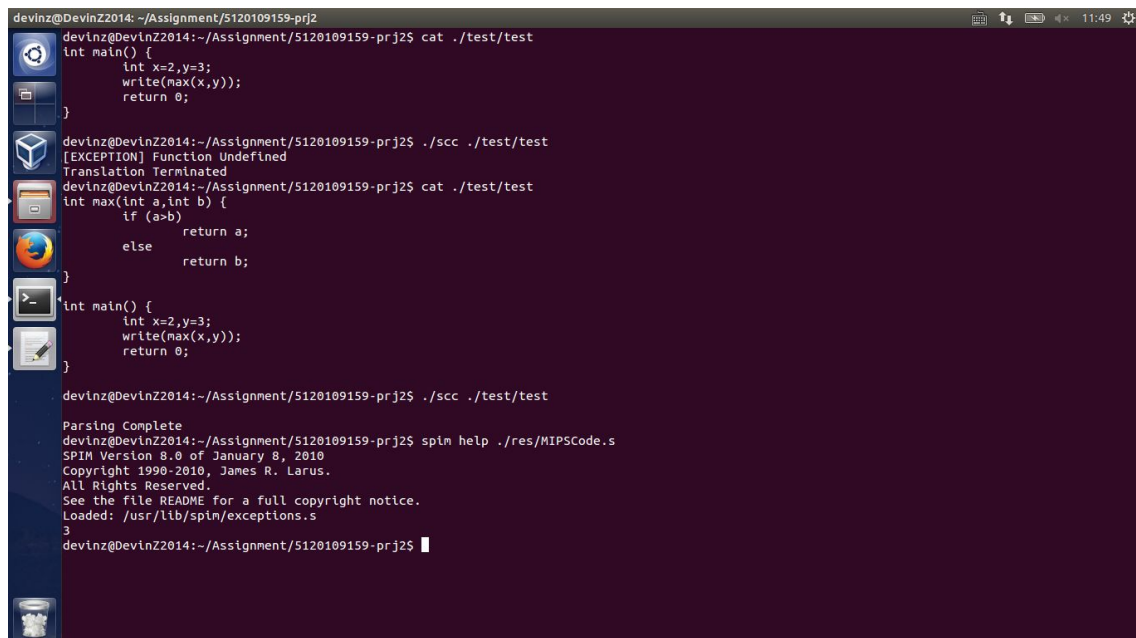


```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    read(x);
    if (1) {
        int x;
    }
    return 0;
}

devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] Variable Undefined
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    int x;
    read(x);
    if (1) {
        int x;
    }
    return 0;
}

devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
Parsing Complete
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

FIG II. FUNCTION UNDEFINED CHECKING



```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    int x=2,y=3;
    write(max(x,y));
    return 0;
}

devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] Function Undefined
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int max(int a,int b) {
    if (a>b)
        return a;
    else
        return b;
}

int main() {
    int x=2,y=3;
    write(max(x,y));
    return 0;
}

devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
Parsing Complete
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ spim help ./res/MIPSCode.s
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
3
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

FIG III. STRUCTURE UNDEFINED CHECKING

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    struct student a;
    return 0;
}

devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] Structure Undefined
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
struct student {
    int id, grade;
};

int main() {
    struct student a;
    read(a.grade);
    write(a.grade);
    return 0;
}

devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
Parsing Complete
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ spin help ./res/MIPSCode.s
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spin/exceptions.s
100
100
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

IV. VARIABLE REDEFINED CHECKING

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int x = 2333;
int x;

int main() {
    write(x);
    return 0;
}

devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] Variable Redefined
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int x = 2333;

int main() {
    int x;
    read(x);
    if (2<x) {
        int x;
        read(x);
        write(x);
    }
    write(x);
    return 0;
}

devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
Parsing Complete
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ spin help ./res/MIPSCode.s
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spin/exceptions.s
10
1000
1000
10
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

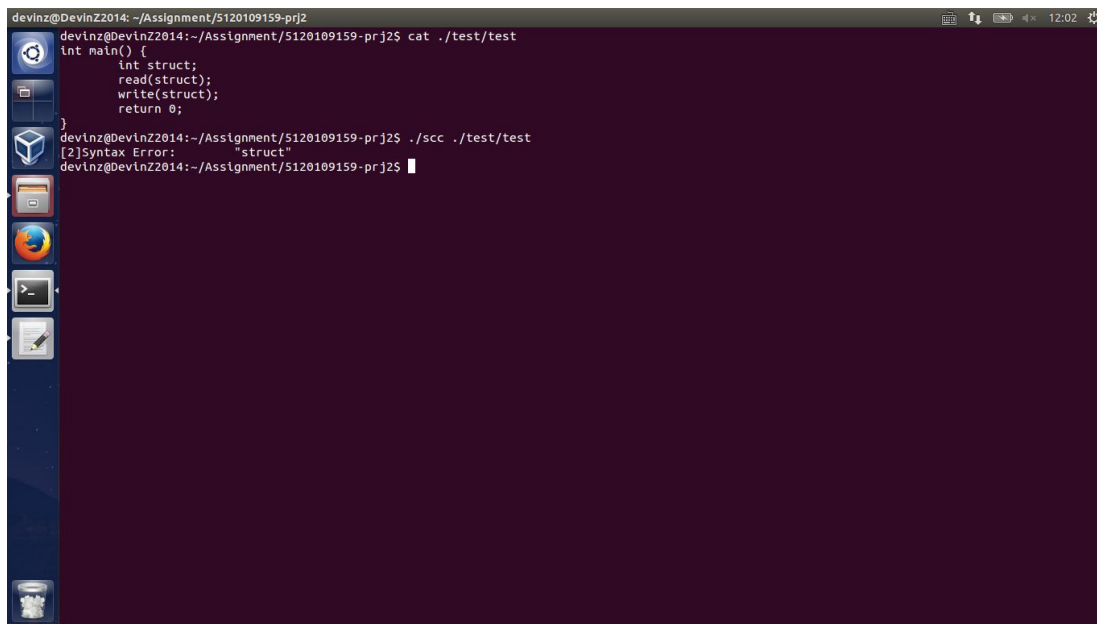
## V. FUNCTION REDEFINED CHECKING

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int funct(int x) {
    return 2*x+1;
}
int funct(int y) {
    return 5*x-1;
}
int main() {
    int x;
    read(x);
    write(funct(x));
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] Function Redefined
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int funct(int x) {
    return 2*x+1;
}
int main() {
    int x;
    read(x);
    write(funct(x));
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
Parsing Complete
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ spin help ./res/MIPSCode.s
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
15
31
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

## VI. STRUCTURE REDEFINED CHECKING

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
struct student {
    int id, grade;
};
struct student {
    int id;
}b,c;
int main() {
    struct student a;
    read(a.grade);
    write(a.grade);
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] Structure Redefined
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
Parsing Complete
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
struct student {
    int id, grade;
};
int main() {
    struct student {
        int id,grade,age;
    }b,c;
    struct student a;
    read(a.grade);
    write(a.grade);
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
Parsing Complete
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

## VII. RESERVED WORDS CANNOT USED AS IDENTIFIERS



A terminal window with a dark purple background and a blue sidebar on the left containing icons for a file manager, web browser, and other applications. The terminal text shows a user named 'devinz' at a machine named 'DevinZ2014' in the directory '~/Assignment/5120109159-prj2'. The user first runs 'cat ./test/test', which displays the contents of a C file: 

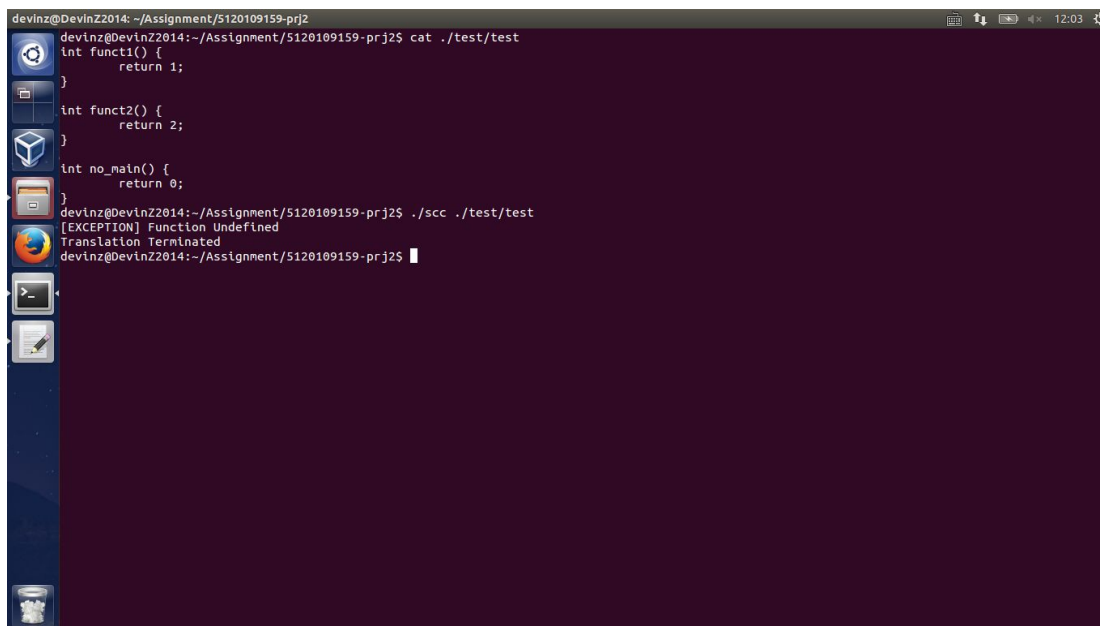
```
int main() {  
    int struct;  
    read(struct);  
    write(struct);  
    return 0;  
}
```

 Then, the user runs './scc ./test/test', which results in a syntax error: 

```
[2]Syntax Error:      "struct"  
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2  
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test  
int main() {  
    int struct;  
    read(struct);  
    write(struct);  
    return 0;  
}  
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test  
[2]Syntax Error:      "struct"  
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

## VIII. PROGRAM ENTRANCE CHECKING



A terminal window with a dark purple background and a blue sidebar on the left. The terminal text shows the same user and directory as the previous screenshot. The user runs 'cat ./test/test', which displays a C file with three functions: 

```
int funct1() {  
    return 1;  
}  
  
int funct2() {  
    return 2;  
}  
  
int no_main() {  
    return 0;  
}
```

 Then, the user runs './scc ./test/test', which results in a program entrance error: 

```
[EXCEPTION] Function Undefined  
Translation Terminated  
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2  
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test  
int funct1() {  
    return 1;  
}  
  
int funct2() {  
    return 2;  
}  
  
int no_main() {  
    return 0;  
}  
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test  
[EXCEPTION] Function Undefined  
Translation Terminated  
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```



## IX. ARGUMENT CHECKING

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int max(int a,int b) {
    if (a>b)
        return a;
    else
        return b;
}

int main() {
    int x = 10,y=5;
    write(max(x));
    return 0;
}

devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] ARGS Error
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int max(int a,int b) {
    if (a>b)
        return a;
    else
        return b;
}

int main() {
    int x = 10,y=5;
    write(max(x,y));
    return 0;
}

devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
Parsing Complete
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ spm help ./res/MIPSCode.s
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
10
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

## X. ARRAY INDEX CHECKING

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    int x;
    read(x[0]);
    return 0;
}

devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] ARRS Error
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    int arr1[5] = {1,2,3,4,5};
    int arr2[5][5];
    int arr3[5][5][5];
    arr2[3] = arr1[3];
    return 0;
}

devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] ARRS Error
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    int arr1[5] = {1,2,3,4,5};
    int arr2[5][5];
    int arr3[5][5][5];
    int i,j,k;
    for (i=0;i<5;++i) {
        for (j=0;j<5;++j) {
            arr2[i][j] = arr1[i]+i;
            for (k=0;k<5;++k)
                arr3[i][j][k] = arr2[i][j]+j;
        }
    }
    write(arr3[4][4][4]);
    return 0;
}

devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
Parsing Complete
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ spm help ./res/MIPSCode.s
SPIM Version 8.0 of January 8, 2010
```

## XI. DOT USED IN NON-STRUCT VARIABLES

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
struct Node {
    int data;
};

int main() {
    int x;
    read(x.data);
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] Variable Undefined
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
struct Node {
    int data;
};

int main() {
    struct Node x;
    read(x.data);
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
Parsing Complete
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

## XII. BREAK IN WRONG PLACES

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    int i;
    for (i=0; i<100; ++i) {
        if (i%2) continue;
        write(i);
        if (i==8) break;
    }
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
Parsing Complete
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ spim help ./res/MIPSCode.s
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
0
2
4
6
8
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    int i;
    if (1) {
        break;
    }
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] BREAK Error
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

### XIII. CONTINUE IN WRONG PLACES

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    int i;
    for (i=0; i<100; ++i) {
        if (i%2) continue;
        write(1);
        if (i==8) break;
    }
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
Parsing Complete
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ spin help ./res/MIPSCode.s
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spin/exceptions.s
0
2
4
6
8
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    int i;
    if (1) continue;
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] CONT Error
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

### XIV. RIGHT-VALUE ASSIGNED

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    int x,y,z=2;
    x+y = z*2;
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] L-val Error
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
struct Node {
    int data;
};
int main() {
    int x,y,z=2;
    a = 2*z;
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] Variable Undefined
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

## XV. IF CONDITIONS AND FOR CONDITIONS

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    int x,y,z=2;
    if () {
        x = y = z;
    }
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] IF Condition Error
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    int x,y,z=2;
    for (;;) {
        x = y = z;
        if (++z>100)
            break;
    }
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
Parsing Complete
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

## XVI. NON-INT INVOLVED IN EXPRESSIONS

```
devinz@DevinZ2014: ~/Assignment/5120109159-prj2
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    struct person {
        int id;
    } a;
    int y;
    y = 2*a+1;
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] Variable Undefined
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
    int arr[100];
    int y;
    y = 2*arr+1;
    return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] ARRS Error
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

## APPENDIX II: INTERMEDIATE CODE AND MIPS CODE EXAMPLES

THE OUTCOME OF TEST2 PROVIDED BY THE TA

// “./res/InterCode”

```

goto    _gcd
gcd:
&BLK_0@gcd
? @tmp_1    =    !    ? y
? @tmp_0    =    ? @tmp_1
ifFalse ? @tmp_0    goto    Label_1_NEXT
? @tmp_2    =    ? x
return ? @tmp_2
Label_1_NEXT:
? @tmp_4    =    ? y
? @tmp_6    =    ? x    %    ? y
? @tmp_5    =    ? @tmp_6
_funcall _gcd
param ? @tmp_4
param ? @tmp_5
? @tmp_7    =    call    gcd    2
? @tmp_3    =    ? @tmp_7
return ? @tmp_3
#BLK_0@gcd

```

\_gcd:

```

main:
&BLK_2@main
read ? a
read ? b
? @tmp_9    =    ? a
? @tmp_10   =    ? b
_funcall _gcd
param ? @tmp_9
param ? @tmp_10
? @tmp_11   =    call    gcd    2
? @tmp_8    =    ? @tmp_11
write ? @tmp_8
? @tmp_12   =    0
return ? @tmp_12
#BLK_2@main
_main:
_END_

```

//“./res/MIPSCode.s”

```

.data
GLB_VAR:
.space 28
endL:
.asciiz "\n"
.text
.globl main

main:
j _gcd

gcd:
#BLK_0@gcd:
lw $t0, -4($fp)
li $t9, 0
seq $t1, $t0, $t9
move $t2, $t1
sw $t1, -12($fp)
sw $t2, -8($fp)
li $t9, 0
beq $t2, $t9, Label_1_NEXT
lw $t0, -0($fp)
move $t1, $t0

```

```

move $v0, $t1
jr $ra
Label_1_NEXT:
lw $t0, -4($fp)
move $t1, $t0
lw $t2, -0($fp)
rem $t3, $t2, $t0
move $t4, $t3
#begin calling gcd
sw $t1, -4($sp)
sw $t4, -8($sp)
sw $t1, -24($fp)
sw $t3, -32($fp)
sw $t4, -28($fp)
sw $fp, 0($sp)
sw $ra, 4($sp)
addi $fp, $sp, -4
li $t9, 40
sub $sp, $sp, $t9
jal gcd
addi $sp, $fp, 4
lw $fp, 0($sp)
lw $ra, 4($sp)

```

```

        move    $t0 ,    $v0
                #end calling gcd
        move    $t1 ,    $t0
        move    $v0 ,    $t1
        jr      $ra
_gcd:
                #:

        _main:
        move    $t2 ,    $t9
        move    $v0 ,    $t2
        jr      $ra
        #:

MAIN_STACK_POSITION:
        li      $fp ,    0x7ffffff8
        li      $t9 ,    20
        sub     $sp ,    $fp ,    $t9
_REAL_MAIN:
                #BLK_2@main:
                #begin read ? a
        li      $v0 ,    5
        syscall
        move    $t0 ,    $v0
                #end read ? a
                #begin read ? b
        li      $v0 ,    5
        syscall
        move    $t1 ,    $v0
                #end read ? b
        move    $t2 ,    $t0
        move    $t3 ,    $t1
                #begin calling gcd
        sw      $t2 ,    -4($sp)
        sw      $t3 ,    -8($sp)
        li      $t9 ,    0
        sw      $t0 ,    GLB_VAR($t9)
        li      $t9 ,    4
        sw      $t1 ,    GLB_VAR($t9)
        sw      $t2 ,    -4($fp)
        sw      $t3 ,    -8($fp)
        sw      $fp ,    0($sp)
        sw      $ra ,    4($sp)
        addi    $fp ,    $sp ,    -4
        li      $t9 ,    40
        sub     $sp ,    $sp ,    $t9
        jal     gcd
        addi    $sp ,    $fp ,    4
        lw      $fp ,    0($sp)
        lw      $ra ,    4($sp)
        move    $t0 ,    $v0
                #end calling gcd
        move    $t1 ,    $t0
                #begin write ? @tmp_8
        move    $a0 ,    $t1
        li      $v0 ,    1
        syscall
        li      $v0 ,    4
        la      $a0 ,    endL
        syscall
                #end write ? @tmp_8
        li      $t9 ,    0

```