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 *addr*s 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 (atttention! 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 funciton 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 funciton 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 recuced 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: $base + ...((i_1*n_2+i_2)*n_3+i_3)*...$ 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(I)* 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.

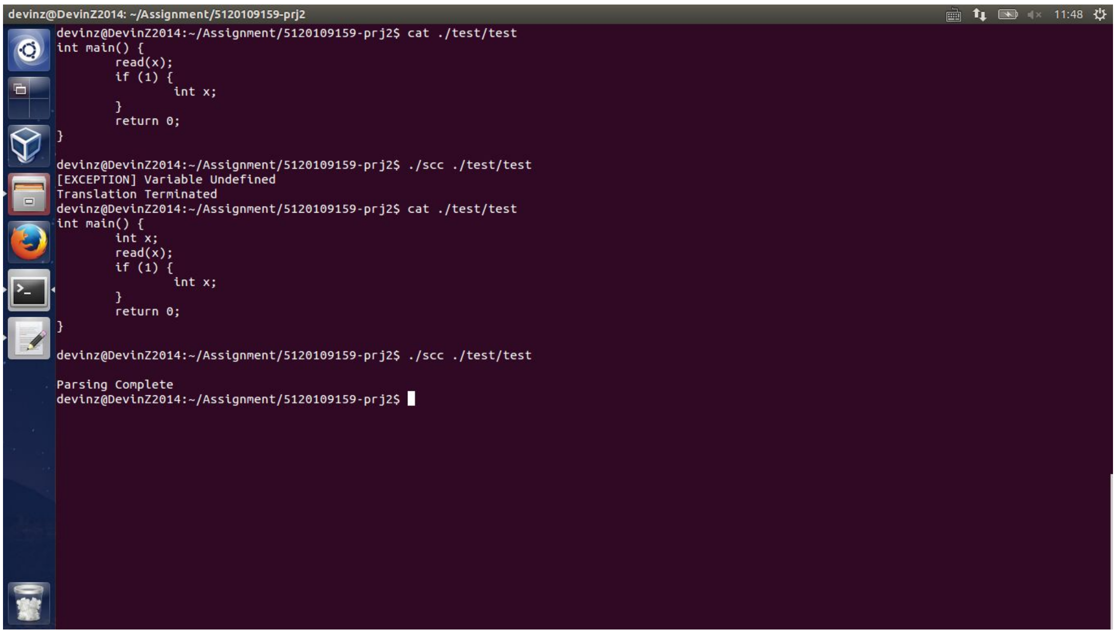
IV. Project Inspirations

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.

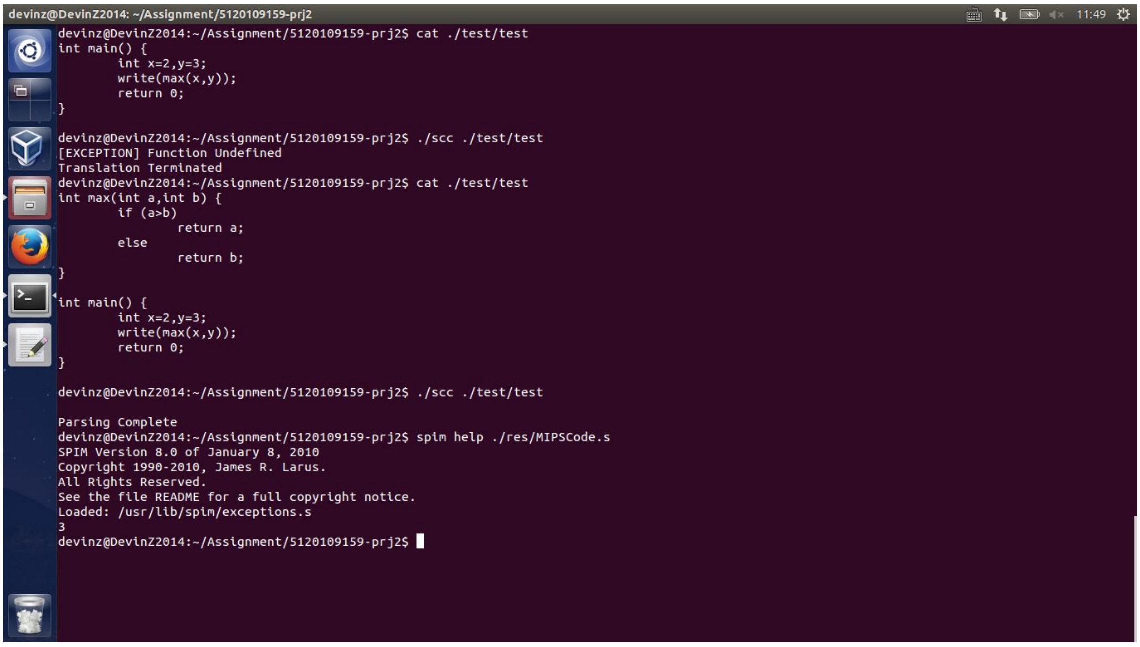
APPENDIX I: SCREEN-SHOTS OF SOME TEST CASES

FIG I. VARIABLE UNDEFINED CHECKING



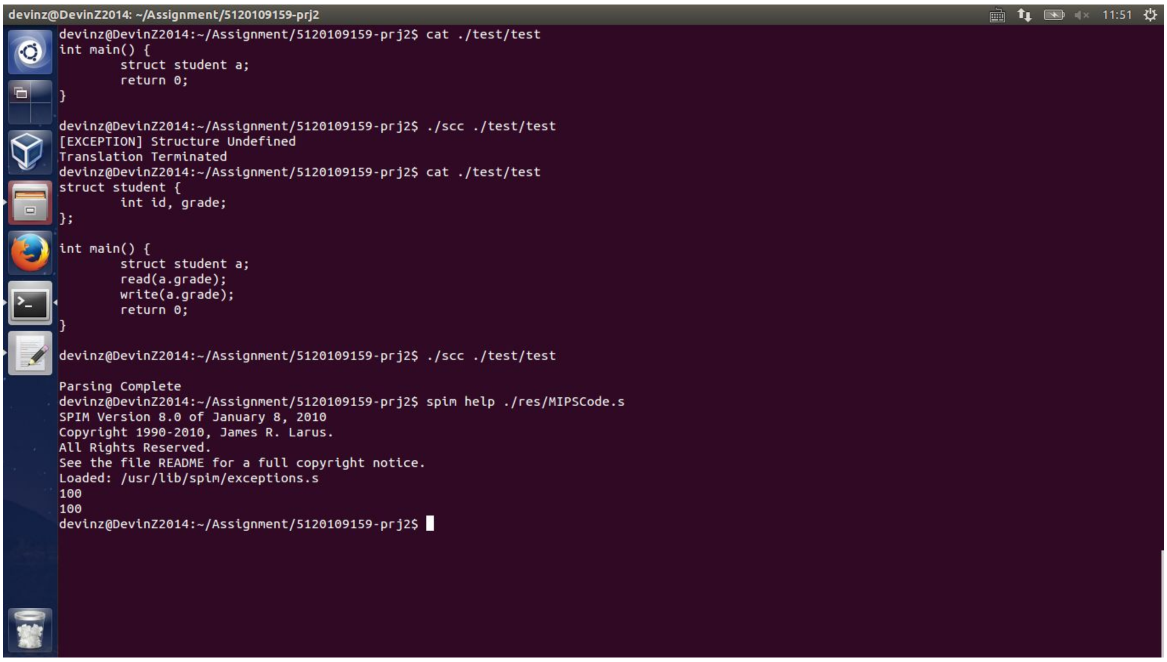
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
        read(x);
        if (1) {
                int x;
        }
        return 0;
}

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



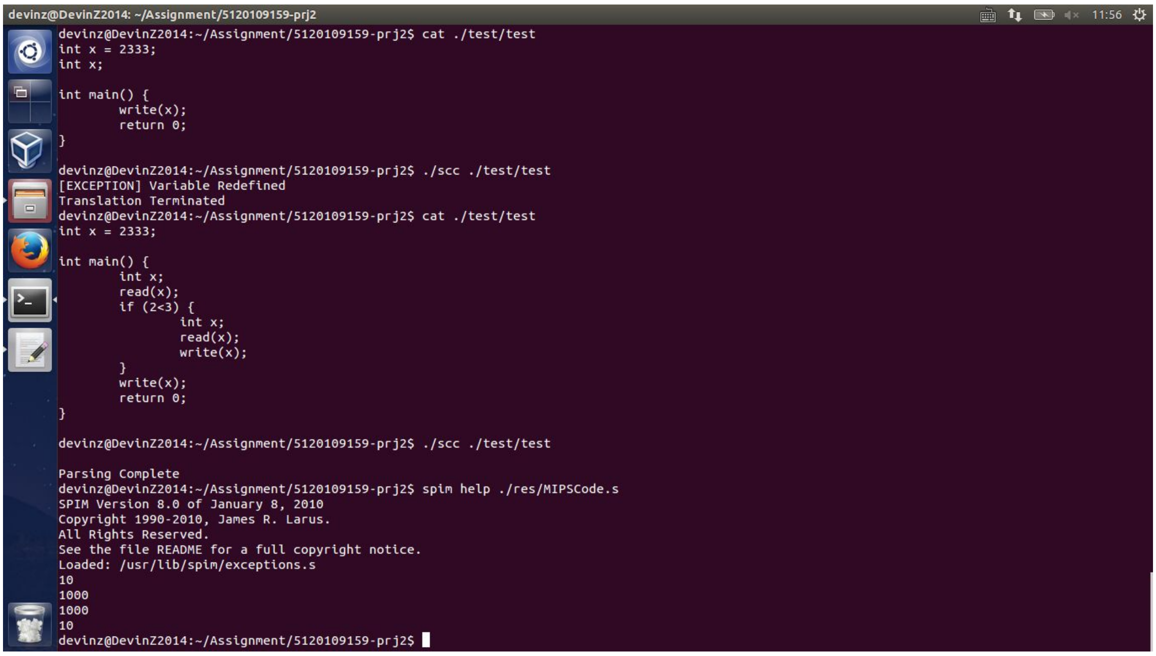
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
        int x=2,y=3;
        write(max(x,y));
        return 0;
}

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



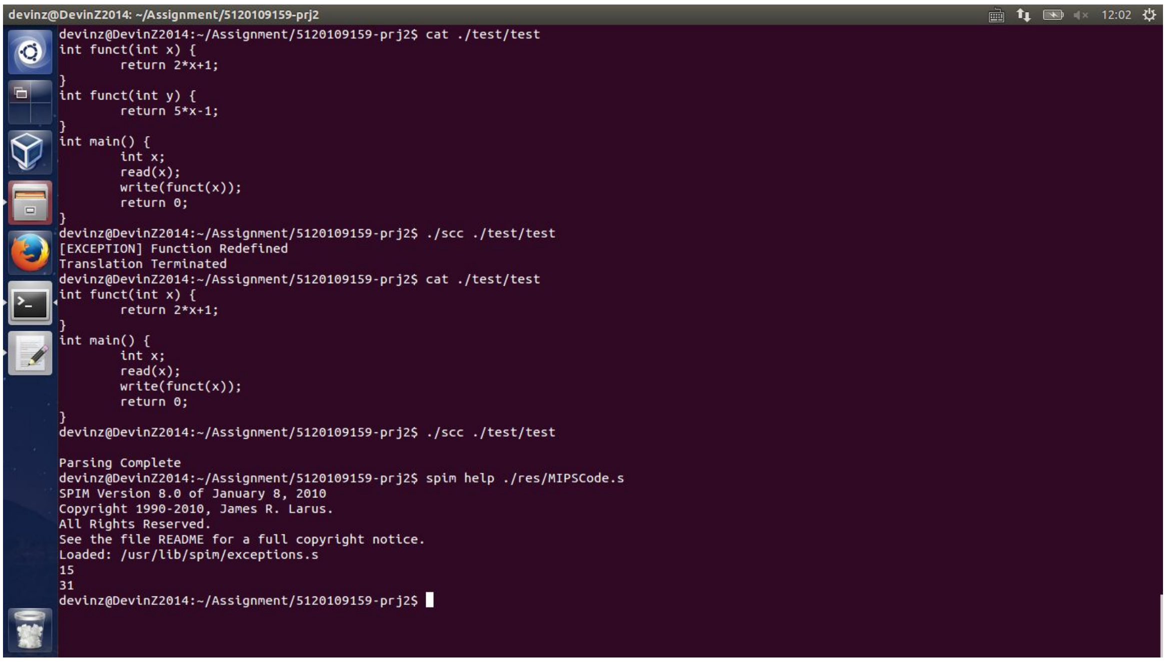
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
        struct student a;
        return 0;
}

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$ 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
100
100
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

IV. VARIABLE REDEFINED CHECKING



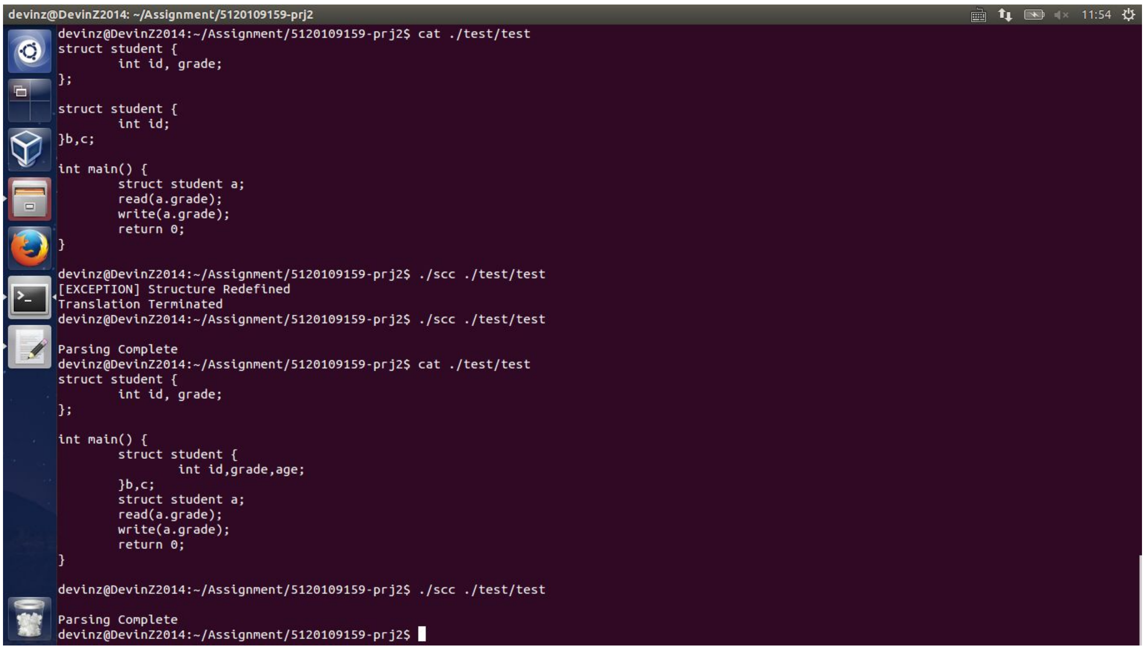
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int x = 2333;
int x;

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<3) {
                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$ 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
10
1000
1000
10
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

V. Function Redefined Checking



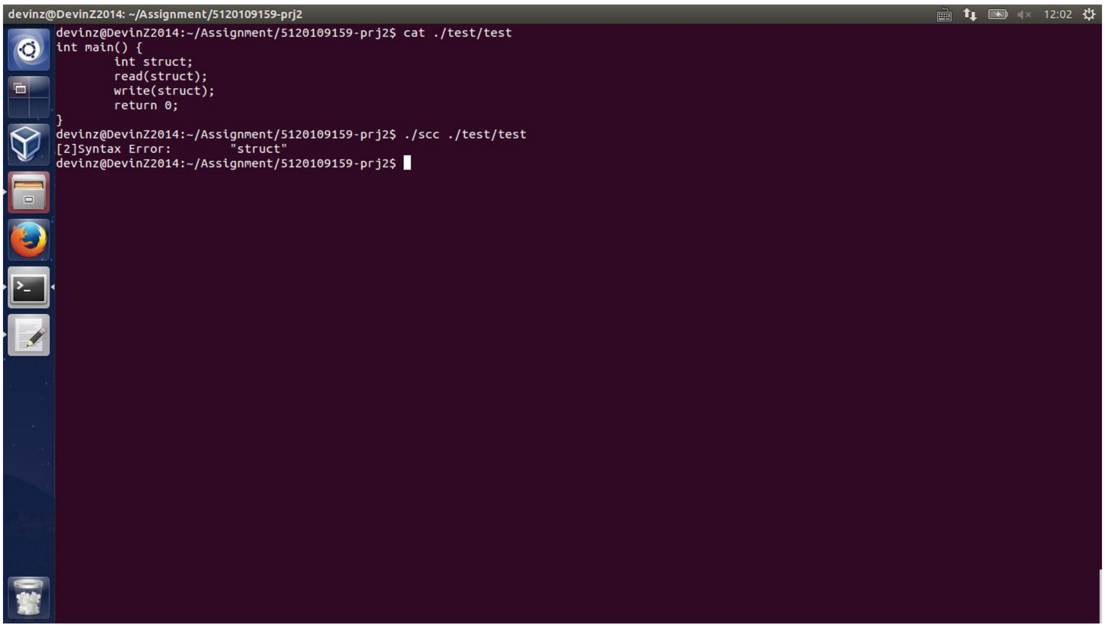
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int funct(int x) {
        return 2*x+1;
}
int funct(int y) {
        return 5*x-1;
}
int main() {
        int x;
        read(x);
        write(funct(x));
        return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[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$ 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
15
31
devinz@DevinZ2014:~/Assignment/5120109159-prj2$
```

VI. Structure Redefined Checking



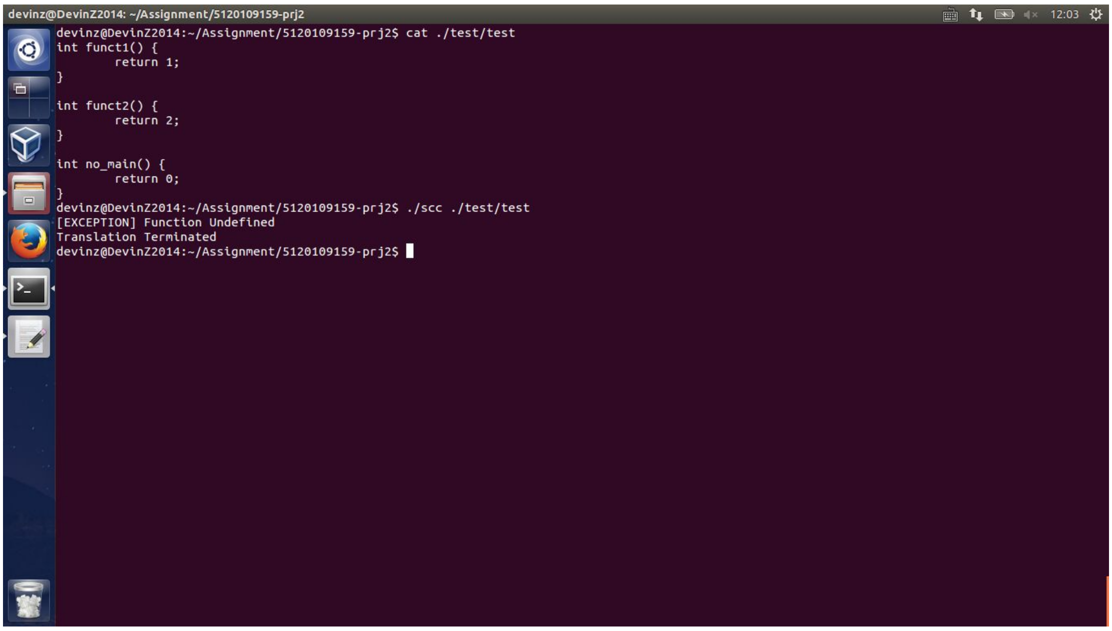
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
struct student {
        int id, grade;
};

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



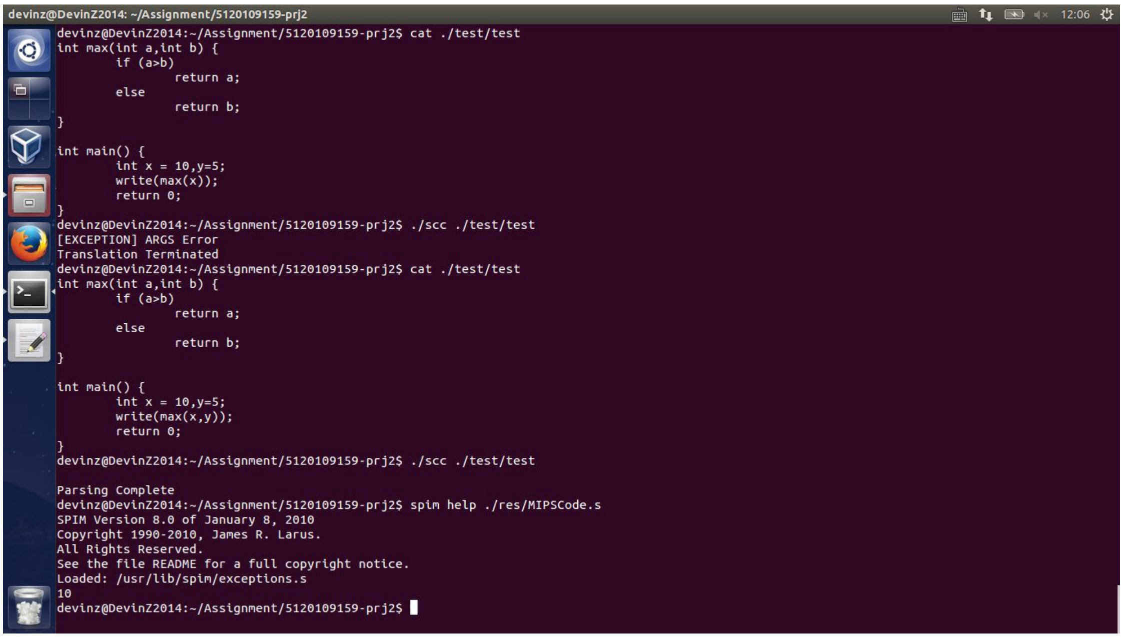
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
        int struct;
        read(struct);
        write(struct);
        return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[2]Syntax Error:         "struct"
devinz@DevinZ2014:~/Assignment/5120109159-prj2$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

VIII. Program Entrance Checking



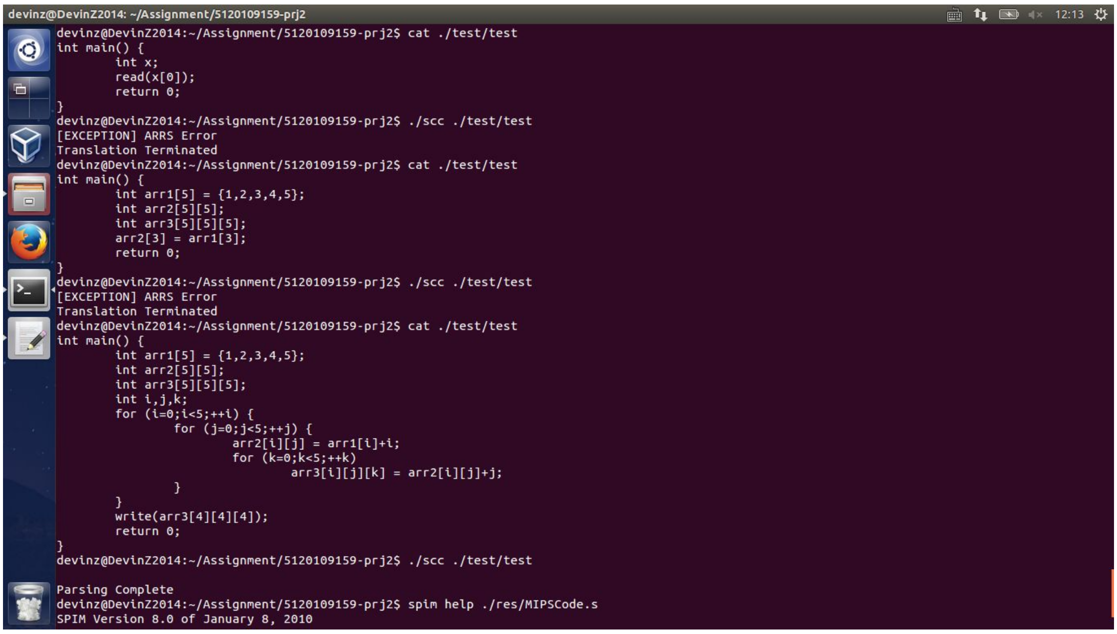
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int funct1() {
        return 1;
}

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



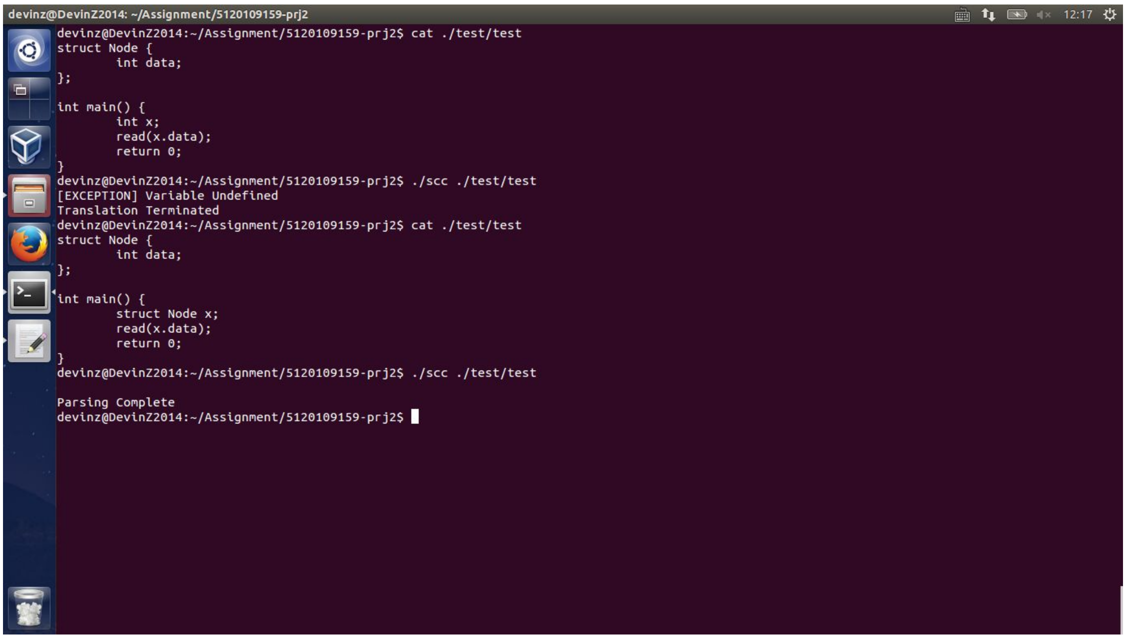
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int max(int a,int b) {
        if (a>b)
                return a;
        else
                return b;
}

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$ 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
10
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ █
```

X. Array Index Checking



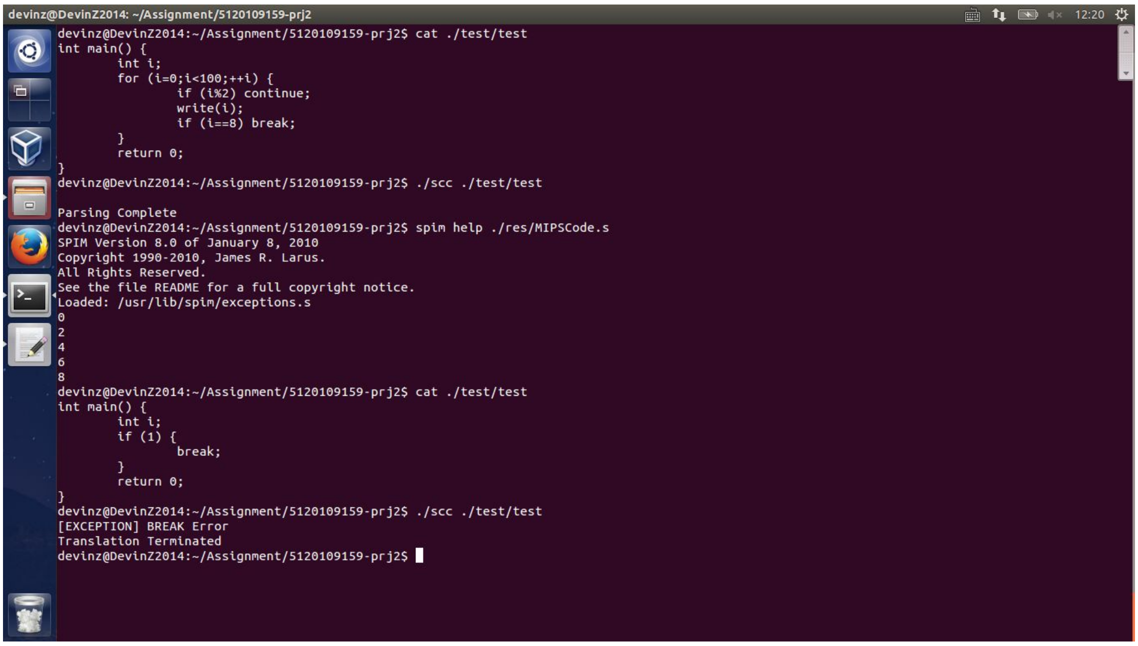
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
        int x;
        read(x[0]);
        return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] ARRS Error
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
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$ spim help ./res/MIPSCode.s
SPIM Version 8.0 of January 8, 2010
```

XI. Dot Used In Non-Struct Variables



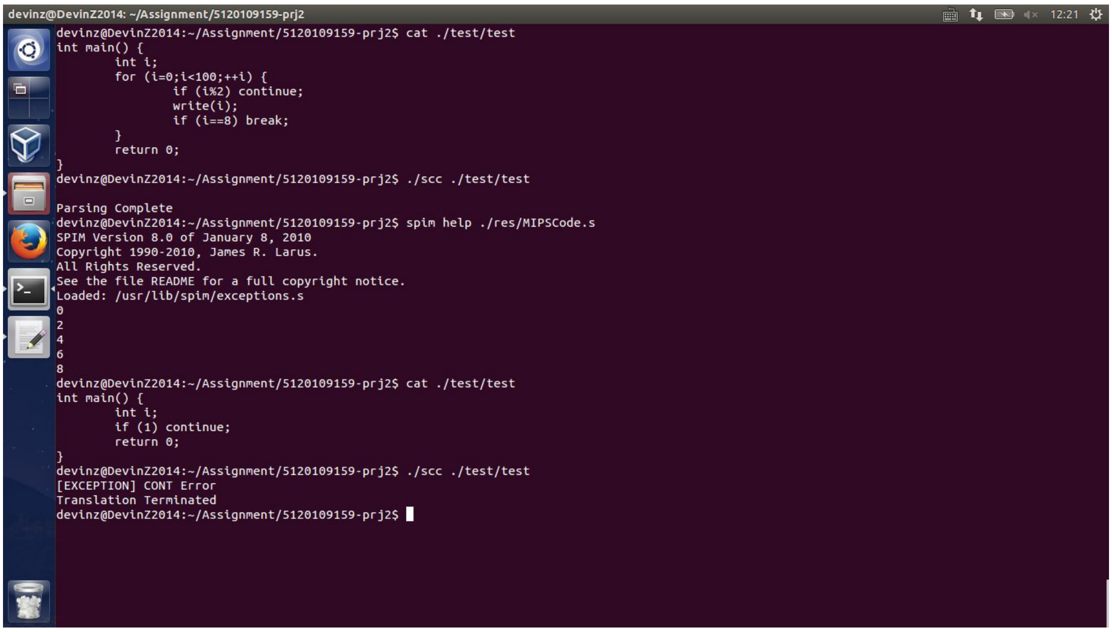
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
struct Node {
        int data;
};

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



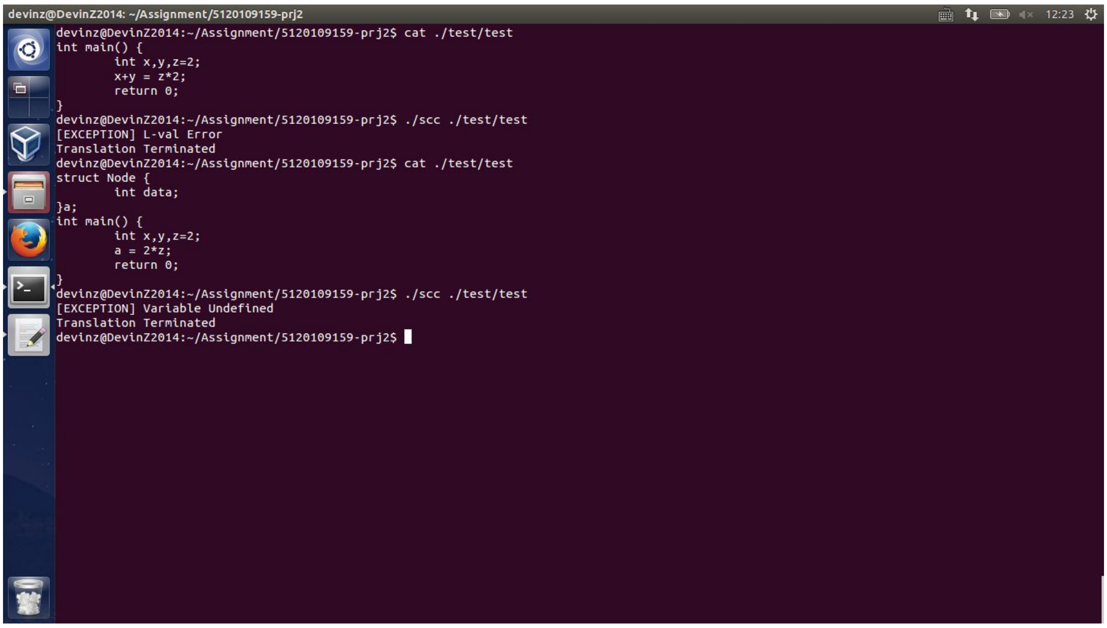
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
        int i;
        for (i=0;i<100;++i) {
                if (i%2) continue;
                write(i);
                if (i==8) break;
        }
        return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./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



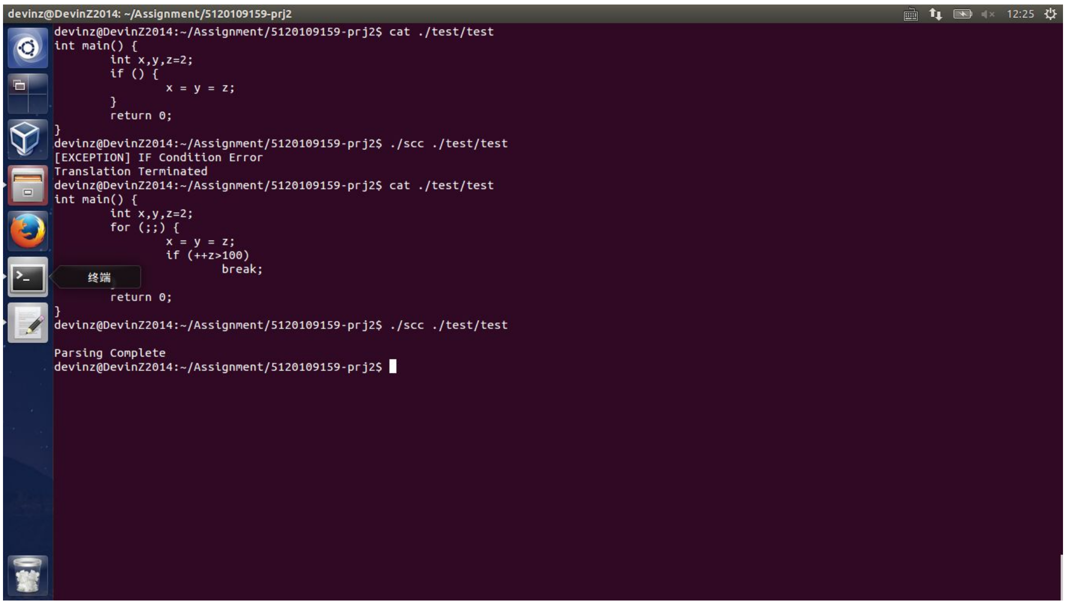
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
        int i;
        for (i=0;i<100;++i) {
                if (i%2) continue;
                write(i);
                if (i==8) break;
        }
        return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./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) 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



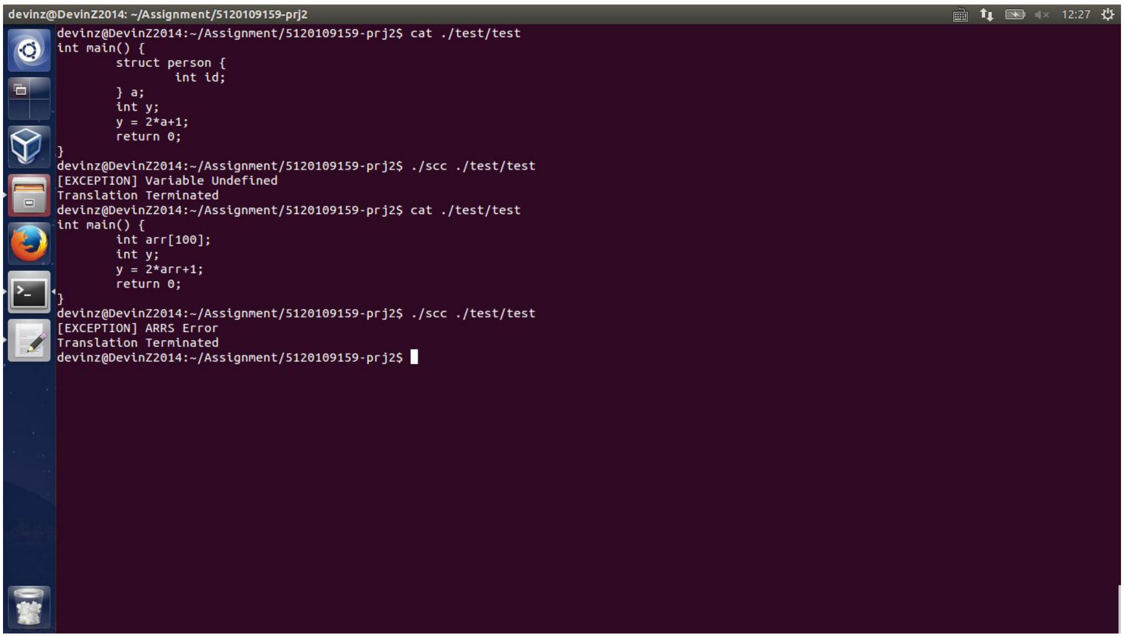
```
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ cat ./test/test
int main() {
        int x,y,z=2;
        x+y = z*2;
        return 0;
}
devinz@DevinZ2014:~/Assignment/5120109159-prj2$ ./scc ./test/test
[EXCEPTION] L-val Error
Translation Terminated
devinz@DevinZ2014:~/Assignment/5120109159-prj2$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

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

Appendix III Original Source Code

0. “makefile”:

```
all: y.tab.o lex.yy.o
	g++ y.tab.o lex.yy.o   -ly -ll -o ./scc
y.tab.o: y.tab.c ./src/def.h
	g++ -c y.tab.c -o y.tab.o
lex.yy.o: ./src/def.h lex.yy.c
	g++ -c lex.yy.c -o lex.yy.o
lex.yy.c: ./src/smallc.l ./src/def.h
	flex ./src/smallc.l
y.tab.c y.tab.h: ./src/smallc.y ./src/def.h
	yacc ./src/smallc.y -v -d
clean:
	rm   lex.* y.tab.*
```

1. “src/def.h”:

```
/* Structs and Classes */
#ifndef _HEADER_H
#define _HEADER_H
#define MAX 65535
#define NUM 9967
#define BLK 128
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <fstream>
#include <iostream>
using namespace std;

// NODE OF PARSE TREE
struct Node {
    char *data;             // yytext
    int prod;         // production rule
    Node *left,*right;
    Node (char *str,int p);
    ~Node();
};

// ARRAY SIZE:
struct Size {
    int data;         // size of one dimension
    Size *next;             // for linked-list
    Size(int sz);
    ~Size();
};

// VARIABLE (ARRAY):
struct Var {
    char *scope;    // "<lbl_list><funct>"; "" for
global
    char *tag;        // "?"; "<strt>" for flds;
"#<pos>" for paras
    char *id;         // variable (array) or field
identifier
    Size *size;             // linked-list of array sizes
    Var *next;        // for chaining-hash
    int addr;         // MIPS virtual address (data or
stack)
    bool mem;               // whether stored in
memory
    int reg;          // whether stored in regs
    Var();
    ~Var();
    bool backUp() const;
};

// STRUCTURE FIELD:
struct Fld {
    char *name;             // field identifier
    Fld *next;        // for linked-list
    Fld(char *id);
    ~Fld();
};

// STRUCTURE:
struct Strt {
    char *scope;    // "<lbl_list><funct>"; "" for
global
    char *id;         // structure identifier
    Fld *fld;         // linked-list of fields
    Strt *next;             // for chaining-hash
    Strt();
    ~Strt();
};

// FUNCTION:
struct Func {
    char *id;         // function identifier
    int argc;         // number of paras
    int spc;          // size of stack
    Func *next;             // for chaining-hash
    Func();
    ~Func();
};

// HASHING FOR SYMBOL TABLE:
class Hash {
    Var varTbl[NUM];            // variables
    Strt strtTbl[NUM];          // structures
    Func funcTbl[NUM];          // functions
    int getIndex(char *id) const;
    void getVarSize(Size* sz,Node *var);
    void getStrtFld(Fld* fld,Node *sdefs);
    void getStrtFldHelp(Fld* &fld,Node *sdecs);
    void addStrtVarsHelp(Fld *fld,Node *vars);
    void getFuncParas(int &argc,Node *paras);
    void transArrs(Size *sz,Node *arrs,char *buf)
const;
	bool testArgc(int argc,Node *args) const;
	void calArgs(char *id,int argc,Node *args)
const;
	public:
	void insVar(char *tag,char *id,Node
*var=NULL);
	void insStrt(char *id,Node *sdecs,Node
*vars);
	void addStrtVars(char *id,Node *vars);
	void insFunc(char *id,Node *paras);
	void setFuncSpc(char *id);
	void srchVar(char *tag,char *id,Node
*arrs,char *idx) const;
	void srchFunc(char *id,Node *args,char *str)
const;
	void callMain() const;
	int getStackSz(char *id) const;
	Var *search(char *tag,char *id) const;
};

// SCOPE DETERMINATION:
struct StrNode {
	char data[BLK];		// scope name
	StrNode *next;		// for linked-stack
	StrNode();
	~StrNode();
};
class StrStack {
	StrNode *head;
	public:
	StrStack();
	~StrStack();
	void push(char *str);
	bool pop(char *str);
};

// REGISTER DESCRIPTOR:
struct Ref {
	Var *var;		// variable pointer
	Ref *next;		// for linked-list
	Ref(Ref *n=NULL);
	~Ref();
};
class Reg {
	int idx;		// register number
	Ref *ref;		// variables that it stores
	void insert(Var *var);
	bool remove(Var *var);
	void store(Var *var);
	public:
	Reg();
	~Reg();
	void setIdx(int i);
	bool search(Var *var) const;
	bool empty() const;
	bool bestDst(Var *var) const;
	int getCost() const;
	void clearArrHelp(Var *var);
	void dstAct(Var *var);
	void load(Var *var);
	void spill();
	void clear(bool st);
	void showHelp();
};

#endif
```

2. “src/smallc.l”:

```
/* Lexical Analyzer */
%{
#include "./src/def.h"
#include "y.tab.h"
int lineCnt = 1;
%}

Dig	[0-9]
Int	{Dig}+|0[Xx][0-9A-Fa-f]+|0[0-7]+
Id	[_a-zA-Z][_a-zA-Z0-9]*
%%

{Int}		{
	yylval.node = new Node(yytext,0);
	return INT;
}
"int"	{
	yylval.node = new Node(yytext,0);
	return TYPE;
}
"struct"	{
	yylval.node = new Node(yytext,0);
	return STRUCT;
}
"return"	{
	yylval.node = new Node(yytext,0);
	return RETURN;
}
"if"	{
	yylval.node = new Node(yytext,0);
	return IF;
}
"else"		{
	yylval.node = new Node(yytext,0);
	return ELSE;
}
"break"	{
	yylval.node = new Node(yytext,0);
	return BREAK;
}
"continue"		{
	yylval.node = new Node(yytext,0);
```

```
        return CONT;
}
"for"{
        yylval.node = new Node(yytext,0);
        return FOR;
}
{Id} {
        yylval.node = new Node(yytext,0);
        return ID;
}
";"     {
        yylval.node = new Node(yytext,0);
        return SEMI;
}
","     {
        yylval.node = new Node(yytext,0);
        return COMMA;
}
"."     {
        yylval.node = new Node(yytext,0);
        return DOT;
}
"("     {
        yylval.node = new Node(yytext,0);
        return LP;
}
")"     {
        yylval.node = new Node(yytext,0);
        return RP;
}
"["     {
        yylval.node = new Node(yytext,0);
        return LB;
}
"]"     {
        yylval.node = new Node(yytext,0);
        return RB;
}
"{"     {
        yylval.node = new Node(yytext,0);
        return LC;
}
"}"     {
        yylval.node = new Node(yytext,0);
        return RC;
}
"!"     {
        yylval.node = new Node(yytext,0);
        return NOT;
}
"++"            {
        yylval.node = new Node(yytext,0);
        return DOUBLE_PLUS;
}
"--"    {
        yylval.node = new Node(yytext,0);
        return DOUBLE_MINUS;
}
"~"     {
        yylval.node = new Node(yytext,0);
        return BIT_NOT;
}
"*"     {
        yylval.node = new Node(yytext,0);
        return MULT;
}
"/"     {
        yylval.node = new Node(yytext,0);
        return DIV;
}
"%"     {
        yylval.node = new Node(yytext,0);
        return MOD;
}
"+"     {
        yylval.node = new Node(yytext,0);
        return PLUS;
}
"<<"            {
        yylval.node = new Node(yytext,0);
        return SL;
}
">>"            {
        yylval.node = new Node(yytext,0);
        return SR;
}
">"     {
        yylval.node = new Node(yytext,0);
        return GT;
}
"<"     {
        yylval.node = new Node(yytext,0);
        return LT;
}
">="            {
        yylval.node = new Node(yytext,0);
        return NLT;
}
"<="            {
        yylval.node = new Node(yytext,0);
        return NGT;
}
"=="            {
        yylval.node = new Node(yytext,0);
        return EQ;
}
"!="    {
        yylval.node = new Node(yytext,0);
        return NE;
}
"&"     {
        yylval.node = new Node(yytext,0);
        return BIT_AND;
}
```

```
"^"  {
       yylval.node = new Node(yytext,0);
       return BIT_NOR;
}
"|"   {
       yylval.node = new Node(yytext,0);
       return BIT_OR;
}
"&&"       {
       yylval.node = new Node(yytext,0);
       return AND;
}
"||"  {
       yylval.node = new Node(yytext,0);
       return OR;
}
"+="       {
       yylval.node = new Node(yytext,0);
       return PLUS_ASSIGN;
}
"-=" {
       yylval.node = new Node(yytext,0);
       return MINUS_ASSIGN;
}
"*="{
       yylval.node = new Node(yytext,0);
       return MULT_ASSIGN;
}
"/=" {
       yylval.node = new Node(yytext,0);
       return DIV_ASSIGN;
}
"%="       {
       yylval.node = new Node(yytext,0);
       return MOD_ASSIGN;
}
"&="       {
       yylval.node = new Node(yytext,0);
       return AND_ASSIGN;
}
"^="       {
       yylval.node = new Node(yytext,0);
       return NOR_ASSIGN;
}
"|=" {
       yylval.node = new Node(yytext,0);
       return OR_ASSIGN;
}
"<<="    {
       yylval.node = new Node(yytext,0);
       return SR_ASSIGN;
}
">>="    {
       yylval.node = new Node(yytext,0);
       return SL_ASSIGN;
}
"-"   {
       yylval.node = new Node(yytext,0);
       return MINUS;
}
"="  {
       yylval.node = new Node(yytext,0);
       return ASSIGN;
}
[\n]  lineCnt++;
[ \t]+       /* eat up whitespace */
.      return UNREC;
%%
```

3. “src/smallc.y”:

```
/* Parser and Code Generator */
%{
#include "./src/def.h"
extern int yychar,lineCnt;
extern char *yytext;
extern FILE *yyin;

///////// GLOBAL VARIABLES ////////
ifstream fin;           // File Reader
ofstream fout;          // File Writer
Node *root = NULL;      // Parse Tree Root
Hash tbl;             // Symbol Table Object
char scp[BLK];              // current scope
int nymHelp = 0;   // help to generate a tmp var
int lblHelp = 0;      // help to generate a label
int addrLocal = 0;  // help to calculate local/global
address
int addrGlobal = 0;       // help to record local
address
Reg regfile[32];      // 32 Reg objects in MIPS
int regUsed = 0;    // registers used in a
three-addr expression
Var *arrHelp[3];     // sentinel Var pointers for
array variables
int arrFlag = 0;      // state of arrHelp
const char* regname[32] =
{"$zero","$at","$v0","$v1","$a0","$a1","$a2","$
a3",

       "$t0","$t1","$t2","$t3","$t4","$t5","$t6","$t
7",

       "$s0","$s1","$s2","$s3","$s4","$s5","$s6","$
s7",

       "$t8","$t9","$k0","$k1","$gp","$sp","$fp","$
ra"};

//////// FUNCTION DECLARATIONS ////////
int yylex();
static void print_tok();
void yyerror(const char *s);
void transError(char *msg);
Node *trBuild(char *str,int num,Node* arr[],int p);
void show(Node *sub);
void newTmpVar(char *val);
bool getDst(char *dst);
void newLabel(char *buf);
char *getId(Node *sub);
int getInt(Node *sub);
void itoa(char *str,int val);
int atoi(char *str);
bool sameTag(char *a,char *b);
bool sameScp(char *a,char *b);
int ham(int num);
void translate();
void transExtDefs(Node *sub);
void transExtDef(Node *sub);
void transDefs(Node *sub);
void transVarDecs(Node *sub);
void transInit(Node *var,Node *init);
void transInitHelp(Node *args,char *id,int sz);
void transExp(Node *sub,char *dst);
void transExps(Node *sub,bool &cst,char *rev,int
&val);
int cstHelp(int op,int val1,int val2=0);
bool transAtom(Node *sub,bool &cst,char *rev,int
&val);
void transStspec(Node *stspec,Node *vars);
void transFunc(Node *func,Node *stmtBlk);
void transStmtBlk(Node *stmtBlk,char
*preNext=NULL,char *preCont=NULL);
void transStmts(Node *stmts,char
*preNext=NULL,char *preCont=NULL);
void transStmt(Node *sub,char
*preNext=NULL,char *preCont=NULL);
void genCode();
void genStart();
void genCodeHelp();
void genInst(int argc,char *argv[]);
void genFuncCall(char *stackSize);
void genAssign(char *argv[]);
void genBinOp(char *argv[]);
void genUniOp(char *argv[]);
Var *getTerm(char *str);
int getVarReg(Var *var,bool dst);
void getCstReg(char *cst);
void issue(bool store,int idx,Var *var);
void descriptorClear(bool st);
void descriptorShow();
%}

%union              { Node *node; };
%token <node>    INT TYPE STRUCT RETURN IF
ELSE BREAK CONT FOR ID SEMI COMMA LC RC
UNREC
                    ASSIGN PLUS_ASSIGN
MINUS_ASSIGN MULT_ASSIGN DIV_ASSIGN
MOD_ASSIGN
                    AND_ASSIGN NOR_ASSIGN
OR_ASSIGN SR_ASSIGN SL_ASSIGN
                    OR AND BIT_OR BIT_NOR
BIT_AND EQ NE GT LT NLT NGT SL SR PLUS MINUS
                    MULT DIV MOD NOT
DOUBLE_PLUS DOUBLE_MINUS BIT_NOT DOT LP
RP LB RB
%type <node>        PROGRAM EXTDEFS EXTDEF
SEXTVARS EXTVARS STSPEC FUNC PARAS
STMTBLOCK
                    STMTS STMT DEFS SDEFS
SDECS DECS VAR INIT EXP EXPS ARRS ARGS
%start              PROGRAM
%nonassoc           LOWER_THAN_ELSE
%nonassoc           ELSE
%right              ASSIGN PLUS_ASSIGN
MINUS_ASSIGN MULT_ASSIGN DIV_ASSIGN
MOD_ASSIGN
                    AND_ASSIGN NOR_ASSIGN
OR_ASSIGN SR_ASSIGN SL_ASSIGN
%left               OR
%left               AND
%left               BIT_OR
%left               BIT_NOR
%left               BIT_AND
%left               EQ NE
%left               GT LT NLT NGT
%left               SL SR
%left               PLUS MINUS
%left               MULT DIV MOD
%right              NOT DOUBLE_PLUS
DOUBLE_MINUS BIT_NOT
%left               DOT LP RP LB RB
%%

PROGRAM: EXTDEFS {
            Node *arr[1] = {$1};
            $$ =
trBuild((char*)"PROGRAM",1,arr,1);
            root = $$;
            }
            ;

EXTDEFS: EXTDEF EXTDEFS {
            Node *arr[2] = {$1,$2};
            $$ = trBuild((char*)"EXTDEFS",2,arr,1);
            }
            | {
            $$ =
trBuild((char*)"EXTDEFS",0,NULL,2);
            }
            ;
EXTDEF: TYPE EXTVARS SEMI {
            Node *arr[3] = {$1,$2,$3};
            $$ = trBuild((char*)"EXTDEF",3,arr,1);
            }
```

```
        | STSPEC SEXTVARS SEMI {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"EXTDEF",3,arr,2);
        }
        | TYPE FUNC STMTBLOCK {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"EXTDEF",3,arr,3);
        }
        ;
SEXTVARS: ID COMMA SEXTVARS {
        Node *arr[3] = {$1,$2,$3};
        $$ =
trBuild((char*)"SEXTVARS",3,arr,1);
        }
        | ID {
        Node *arr[1] = {$1};
        $$ =
trBuild((char*)"SEXTVARS",1,arr,2);
        }
        | {
        $$ =
trBuild((char*)"SEXTVARS",0,NULL,3);
        }
        ;
EXTVARS: VAR ASSIGN INIT COMMA EXTVARS {
        Node *arr[5] = {$1,$2,$3,$4,$5};
        $$ = trBuild((char*)"EXTVARS",5,arr,1);
        }
        | VAR COMMA EXTVARS {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"EXTVARS",3,arr,2);
        }
        | VAR ASSIGN INIT {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"EXTVARS",3,arr,3);
        }
        | VAR {
        Node *arr[1] = {$1};
        $$ = trBuild((char*)"EXTVARS",1,arr,4);
        }
        | {
        $$ =
trBuild((char*)"EXTVARS",0,NULL,5);
        }
        ;
STSPEC: STRUCT ID LC SDEFS RC {
        Node *arr[5] = {$1,$2,$3,$4,$5};
        $$ = trBuild((char*)"STSPEC",5,arr,1);
        }
        | STRUCT LC SDEFS RC {
        Node *arr[4] = {$1,$2,$3,$4};
        $$ = trBuild((char*)"STSPEC",4,arr,2);
        }
        | STRUCT ID {
        Node *arr[2] = {$1,$2};
        $$ = trBuild((char*)"STSPEC",2,arr,3);
        }
```

```
        ;
FUNC: ID LP PARAS RP {
        Node *arr[4] = {$1,$2,$3,$4};
        $$ = trBuild((char*)"FUNC",4,arr,1);
        }
        ;
PARAS: TYPE ID COMMA PARAS {
        Node *arr[4] = {$1,$2,$3,$4};
        $$ = trBuild((char*)"PARAS",4,arr,1);
        }
        | TYPE ID {
        Node *arr[2] = {$1,$2};
        $$ = trBuild((char*)"PARAS",2,arr,2);
        }
        | {
        $$ = trBuild((char*)"PARAS",0,NULL,3);
        }
        ;
STMTBLOCK: LC DEFS STMTS RC {
        Node *arr[4];
        arr[0] = $1;
        arr[1] = $2;
        arr[2] = $3;
        arr[3] = $4;
        $$ =
trBuild((char*)"STMTBLOCK",4,arr,1);
        }
        ;
STMTS: STMT STMTS {
        Node *arr[2];
        arr[0] = $1;
        arr[1] = $2;
        $$ = trBuild((char*)"STMTS",2,arr,1);
        }
        | {
        $$ = trBuild((char*)"STMTS",0,NULL,2);
        }
        ;
STMT: EXP SEMI {
        Node *arr[2] = {$1,$2};
        $$ = trBuild((char*)"STMT",2,arr,1);
        }
        | STMTBLOCK {
        Node *arr[1] = {$1};
        $$ = trBuild((char*)"STMT",1,arr,2);
        }
        | RETURN EXP SEMI {
        Node *arr[3]={$1,$2,$3};
        $$ = trBuild((char*)"STMT",3,arr,3);
        }
        | IF LP EXP RP STMT ELSE STMT {
        Node *arr[7] =
{$1,$2,$3,$4,$5,$6,$7};
        $$ = trBuild((char*)"STMT",7,arr,4);
        }
        | IF LP EXP RP STMT %prec
LOWER_THAN_ELSE {
        Node *arr[5] = {$1,$2,$3,$4,$5};
        $$ = trBuild((char*)"STMT",5,arr,5);
        }
        | FOR LP EXP SEMI EXP SEMI EXP RP
STMT {
        Node *arr[9] =
{$1,$2,$3,$4,$5,$6,$7,$8,$9};
        $$ = trBuild((char*)"STMT",9,arr,6);
        }
        | CONT SEMI {
        Node *arr[2] = {$1,$2};
        $$ = trBuild((char*)"STMT",2,arr,7);
        }
        | BREAK SEMI {
        Node *arr[2] = {$1,$2};
        $$ = trBuild((char*)"STMT",2,arr,8);
        }
        ;
DEFS: TYPE DECS SEMI DEFS {
        Node *arr[4] = {$1,$2,$3,$4};
        $$ = trBuild((char*)"DEFS",4,arr,1);
        }
        | STSPEC SDECS SEMI DEFS {
        Node *arr[4] = {$1,$2,$3,$4};
        $$ = trBuild((char*)"DEFS",4,arr,2);
        }
        | {
        $$ = trBuild((char*)"DEFS",0,NULL,3);
        }
        ;
SDEFS: TYPE SDECS SEMI SDEFS {
        Node *arr[4] = {$1,$2,$3,$4};
        $$ = trBuild((char*)"SDEFS",4,arr,1);
        }
        | {
        $$ = trBuild((char*)"SDEFS",0,NULL,2);
        }
        ;
SDECS: ID COMMA SDECS {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"SDECS",3,arr,1);
        }
        | ID {
        Node *arr[1] = {$1};
        $$ = trBuild((char*)"SDECS",1,arr,2);
        }
        ;
DECS: VAR ASSIGN INIT COMMA DECS {
        Node *arr[5] = {$1,$2,$3,$4,$5};
        $$ = trBuild((char*)"DECS",5,arr,1);
        }
        | VAR COMMA DECS {
        Node *arr[3];
        arr[0] = $1;
        arr[1] = $2;
        arr[2] = $3;
        $$ = trBuild((char*)"DECS",3,arr,2);
        }
        | VAR ASSIGN INIT {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"DECS",3,arr,3);
        }
        | VAR {
        Node *arr[1] = {$1};
        $$ = trBuild((char*)"DECS",1,arr,4);
        }
        ;
VAR: ID {
        Node *arr[1] = {$1};
        $$ = trBuild((char*)"VAR",1,arr,1);
        }
        | VAR LB INT RB {
        Node *arr[4] = {$1,$2,$3,$4};
        $$ = trBuild((char*)"VAR",4,arr,2);
        }
        ;
INIT: EXP {
        Node *arr[1] = {$1};
        $$ = trBuild((char*)"INIT",1,arr,1);
        }
        | LC ARGS RC {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"INIT",3,arr,2);
        }
        ;
EXP: EXPS {
        Node *arr[1] = {$1};
        $$ = trBuild((char*)"EXP",1,arr,1);
        }
        | {
        $$ = trBuild((char*)"EXP",0,NULL,2);
        }
        ;
EXPS: EXPS ASSIGN EXPS {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"EXPS",3,arr,1);
        }
        | EXPS PLUS_ASSIGN EXPS {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"EXPS",3,arr,2);
        }
        | EXPS MINUS_ASSIGN EXPS {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"EXPS",3,arr,3);
        }
        | EXPS MULT_ASSIGN EXPS {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"EXPS",3,arr,4);
        }
        | EXPS DIV_ASSIGN EXPS {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"EXPS",3,arr,5);
        }
        | EXPS MOD_ASSIGN EXPS {
```

```
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,6);
}
| EXPS AND_ASSIGN EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,7);
}
| EXPS NOR_ASSIGN EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,8);
}
| EXPS OR_ASSIGN EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,9);
}
| EXPS SR_ASSIGN EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,10);
}
| EXPS SL_ASSIGN EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,11);
}
| EXPS OR EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,12);
}
| EXPS AND EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,13);
}
| EXPS BIT_OR EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,14);
}
| EXPS BIT_NOR EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,15);
}
| EXPS BIT_AND EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,16);
}
| EXPS NE EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,17);
}
| EXPS EQ EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,18);
}
| EXPS GT EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,19);
}
| EXPS LT EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,20);
}
| EXPS NGT EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,21);
}
| EXPS NLT EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,22);
}
| EXPS SL EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,23);
}
| EXPS SR EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,24);
}
| EXPS PLUS EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,25);
}
| EXPS MINUS EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,26);
}
| EXPS MULT EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,27);
}
| EXPS DIV EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,28);
}
| EXPS MOD EXPS {
Node *arr[3] = {$1,$2,$3};
$$ = trBuild((char*)"EXPS",3,arr,29);
}
| DOUBLE_PLUS EXPS %prec NOT {
Node *arr[2] = {$1,$2};
$$ = trBuild((char*)"EXPS",2,arr,30);
}
| DOUBLE_MINUS EXPS %prec NOT {
Node *arr[2] = {$1,$2};
$$ = trBuild((char*)"EXPS",2,arr,31);
}
| MINUS EXPS %prec NOT {
Node *arr[2] = {$1,$2};
$$ = trBuild((char*)"EXPS",2,arr,32);
}
| NOT EXPS %prec NOT {
Node *arr[2] = {$1,$2};
$$ = trBuild((char*)"EXPS",2,arr,33);
}
| BIT_NOT EXPS %prec NOT {
Node *arr[2] = {$1,$2};
$$ = trBuild((char*)"EXPS",2,arr,34);
```

```
        }
        | LP EXPS RP {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"EXPS",3,arr,35);
        }
        | ID LP ARGS RP {
        Node *arr[4] = {$1,$2,$3,$4};
        $$ = trBuild((char*)"EXPS",4,arr,36);
        }
        | ID ARRS {
        Node *arr[2] = {$1,$2};
        $$ = trBuild((char*)"EXPS",2,arr,37);
        }
        | ID DOT ID {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"EXPS",3,arr,38);
        }
        | INT {
        Node *arr[1] = {$1};
        $$ = trBuild((char*)"EXPS",1,arr,39);
        }
        ;
ARRS: LB EXP RB ARRS {
        Node *arr[4] = {$1,$2,$3,$4};
        $$ = trBuild((char*)"ARRS",4,arr,1);
        }
        | {
        $$ = trBuild((char*)"ARRS",0,NULL,2);
        }
        ;
ARGS: EXP COMMA ARGS {
        Node *arr[3] = {$1,$2,$3};
        $$ = trBuild((char*)"ARGS",3,arr,1);
        }
        | EXP {
        Node *arr[1] = {$1};
        $$ = trBuild((char*)"ARGS",1,arr,2);
        }
        ;
%%

int main(int argc,char **argv)
{
    ++argv; --argc;
    if (argc>0)
        yyin = fopen(argv[0], "r");
    else
        yyin = stdin;
    if (!yyparse()) {
        translate();
        cout<<"\nParsing Complete\n";
        genCode();
    }
    return 0;
}

//////////////////////////////////
//          TOOLS FOR EASY WORK          //
//////////////////////////////////


//////// PARSING ERROR MESSAGES ////////

static void print_tok()
{
    if (yychar>=255)
        cerr<<"\""<<yytext<<"\""<<endl;
    else
        cerr<<"\""<<yychar<<"\""<<endl;
}

void yyerror(const char *s)
{
    cerr<<"["<<lineCnt<<"]Syntax Error:\t";
    print_tok();
}

void transError(char *msg)
{
    cerr<<"[EXCEPTION] "<<msg<<endl;
    cerr<<"Translation Terminated"<<endl;
   exit(1);
}


//////// Tree Building ////////


Node *trBuild(char *str,int num,Node* arr[],int p)
{
    Node *val = new Node(str,p);
    if (num>0) {
        val->left = arr[0];
        Node *itr = val->left;
        int i=1;
        for (;i<num;i++) {
            itr->right = arr[i];
            itr = itr->right;
        }
    }
    return val;
}

void show(Node *sub)
{
    if (sub==NULL) return;
    else if (sub->prod==0)
        cout<<sub->data<<' ';
    else if (sub->left!=NULL) {
        Node *itr = sub->left;
        while (itr!=NULL) {
            show(itr);
            itr = itr->right;
        }
    }
}


//////// Other Tools ////////

// Generate a Temporary Variable:
void newTmpVar(char *val)
{
    bzero(val,BLK);
    strcpy(val,"? @tmp_");
    itoa(val,nymHelp++);
    tbl.insVar((char*)"?",val+2,NULL);
}

// Generate a L-val of an EXP:
bool getDst(char *dst)
{
    if (!strlen(dst)) {
        newTmpVar(dst);
        return true;
    } else
        return false;
}

// Generate a New Label:
void newLabel(char *buf)
{
    bzero(buf,BLK);
    strcpy(buf,"Label_");
    itoa(buf,lblHelp++);
}

// Get Leftmost ID:
char *getId(Node *itr)
{
    while (itr->prod!=0)
        itr = itr->left;
    return itr->data;
}

// Test and Decode INT:
int getInt(Node *sub)
{
    int base, pos = 0, val = 0;
    char *str = sub->data;
    if (str[pos]=='0') {
        if (str[++pos]=='X'||str[pos]=='x') {
            base = 16; pos++;
        } else
            base = 8;
    } else
        base = 10;
    for (;pos<strlen(str);pos++) {
        val *= base;
        if (str[pos]>='0'&&str[pos]<='9')
            val += str[pos]-'0';
        else if (str[pos]>='A'&&str[pos]<='F')
            val += str[pos]-'A'+10;
        else
            val += str[pos]-'a'+10;
        if (val<0)
            transError((char*)"INT out of Range");
    }
    return val;
}

// Convert int into char*
void itoa(char *str,int val)
{
    int tmp = val,len = 0;
    while (tmp>0) {
        tmp /= 10;
        len++;
    }
    int pos = strlen(str);
    if (len==0) str[pos] = '0';
    else for (int i=len-1;i>=0;i--) {
        str[pos+i] = (val%10)+'0';
        val /= 10;
    }
}

// Convert char* into int
int atoi(char *str)
{
    int val=0, pos=0;
    while (str[pos]>='0'&&str[pos]<='9')
        val = val*10+(str[pos++]-'0');
    return val;
}

// Whether of Same Tag:
bool sameTag(char *a,char *b)
{
    if (!strcmp(a,b))
        return true;
    else if (a[0]=='#'&&!strcmp(b,"?"))
        return true;
    else
        return false;
}

// Whether Valid in Current Scope:
bool sameScp(char *a,char *b)
{
    if (!strlen(a))
        return true;
    else if (!strcmp(a,b))
        return true;
    char *c = strstr(b,a);
    if (c!=NULL)
        return (*(c-1)=='@');
    else
        return false;
}

// Hamming Code of a Binary:
int ham(int num)
{
    num =
(num&0x55555555)+((num>>1)&0x55555555);
    num =
(num&0x33333333)+((num>>2)&0x33333333);
    num =
(num&0x0F0F0F0F)+((num>>4)&0x0F0F0F0F);
    num =
(num&0x00FF00FF)+((num>>8)&0x00FF00FF);
    num =
(num&0x0000FFFF)+((num>>16)&0x0000FFFF);
    return num;
}


////////////////////////////////////////
//        TRANSLATION BY TRAVERSAL        //
////////////////////////////////////////


// Traverse PROGRAM:
void translate()
{
    fout.open("./output/InterCode");
   if (!fout) {
        cerr<<"Cannot open file \"";
      cerr<<"./output/InterCode\""<<endl;
      exit(0);
   }
    bzero(scp,BLK);
    transExtDefs(root->left);
   tbl.callMain();
    fout<<"_END_"<<endl;
    fout.close();
    delete root;
}

// Traverse EXTDEFS:
void transExtDefs(Node *sub)
{
    if (sub->prod==1) {
        transExtDef(sub->left);
        transExtDefs(sub->left->right);
    }
}

// Traverse EXTDEF:
void transExtDef(Node *sub)
{
    Node *sub1 = sub->left;
    Node *sub2 = sub1->right;
    Node *sub3 = sub2->right;
    if (sub->prod==1)
        transVarDecs(sub2);
    else if (sub->prod==2)
        transStspec(sub1,sub2);
    else if (sub->prod==3)
        transFunc(sub2,sub3);
}

// Traverse DEFS:
void transDefs(Node *sub)
{
    if (sub->prod==1)
        transVarDecs(sub->left->right);
    else if (sub->prod==2)

    transStspec(sub->left,sub->left->right);
    if (sub->prod!=3)

    transDefs(sub->left->right->right->right);
}

// Traverse EXTVARS or DECS
void transVarDecs(Node *sub)
{
    if (sub->prod==5) return;
    else if (sub->prod%2==1)

    transInit(sub->left,sub->left->right->right);
    tbl.insVar((char*)"?",getId(sub),sub->left);
    if (sub->prod==1)

    transVarDecs(sub->left->right->right->right
->right);
    else if (sub->prod==2)
        transVarDecs(sub->left->right->right);
}


// Traverse INIT:
void transInit(Node *var,Node *init)
{
    if (init->prod==1) {
        if (var->prod!=1)
            transError((char*)"INIT Error");
        char dst[BLK];
        bzero(dst,BLK);
        strcat(dst,"? ");
        strcat(dst,getId(var));
        transExp(init->left,dst);
    } else {
        if (var->prod!=2||var->left->prod!=1)
            transError((char*)"INIT Error");
        char *id = getId(var);
        int sz = getInt(var->left->right->right);
        transInitHelp(init->left->right,id,sz);
    }
```

```
}

// Traverse ARGS in INIT:
void transInitHelp(Node *args,char *id,int sz)
{
//      to test: int arr[num] = {a1,a2,...,an};
        for (int i=0;i<sz;i++) {
                char dst[BLK];
                bzero(dst,BLK);
                strcat(dst,"? ");
                strcat(dst,id);
                strcat(dst," [");
                itoa(dst,i);
                strcat(dst,"]");
                transExp(args->left,dst);
                if (args->prod==1)
                        args = args->left->right->right;
                else break;
        }
        if (args->prod==1)
                transError((char*)"INIT Error");
}

// Traverse EXP:
void transExp(Node *sub,char *dst)
{
//      if dst is decided:
//              generate an assignment statement
        if (sub->prod==1) {
                int val; bool cst;
                char rev[BLK];
                bzero(rev,BLK);
                transExps(sub->left,cst,rev,val);
                if (strlen(dst)) {
                        if (cst)

        fout<<'\t'<<dst<<"\t=\t"<<val<<endl;
                        else

        fout<<'\t'<<dst<<"\t=\t"<<rev<<endl;
                } else if (cst)
                        itoa(dst,val);
                else
                        strcpy(dst,rev);
        } else {
                getDst(dst);
                fout<<'\t'<<dst<<"\t=\t1"<<endl;
        }
}

// Traverse EXPS:
void transExps(Node *sub,bool &cst,char *rev,int
&val)
{
//      decide cst according to production rule
//      if cst is true:
//              val returns a constant
```

```
//      else:
//              rev returns a variable
        bool cst1, cst2;
        int val1, val2;
        char x[BLK], y[BLK];
        bzero(x,BLK);
        bzero(y,BLK);
        if (sub->prod==1) {                                     //
x = y

        transExps(sub->left->right->right,cst,rev,va
l);
                if (!transAtom(sub->left,cst1,x,val1))
                        transError((char*)"L-val Error");
                if (cst)

        fout<<'\t'<<x<<"\t=\t"<<val<<endl;
                else

        fout<<'\t'<<x<<"\t=\t"<<rev<<endl;
        } else if (sub->prod<12) {                      // x
op= y

        transExps(sub->left->right->right,cst2,y,val
2);
                if (!transAtom(sub->left,cst1,x,val1))
                        transError((char*)"L-val Error");
                fout<<'\t'<<x<<"\t=\t"<<x<<'\t';
                char op[5];
                bzero(op,5);
                strcpy(op,sub->left->right->data);
                op[strlen(op)-1] = '\0';
                if (cst2)
                        fout<<op<<'\t'<<val2<<endl;
                else
                        fout<<op<<'\t'<<y<<endl;
                cst = false;    // otherwise l-val error
                strcpy(rev,x);
        } else if (sub->prod<30) {                      // x op
y
                transExps(sub->left,cst1,x,val1);

        transExps(sub->left->right->right,cst2,y,val
2);
                if (cst1&&cst2) {
                        cst = true;
                        val =
cstHelp(sub->prod,val1,val2);
                } else if (cst1) {
                        cst = false;
                        getDst(rev);

        fout<<'\t'<<rev<<"\t=\t"<<val1<<'\t';

        fout<<sub->left->right->data<<'\t'<<y<<
endl;
                } else {
```

```
            cst = false;
            getDst(rev);

    fout<<'\t'<<rev<<"\t=\t"<<x<<'\t';
            if (cst2)

    fout<<sub->left->right->data<<'\t'<<val2
<<endl;
            else

    fout<<sub->left->right->data<<'\t'<<y<<
endl;
        }
    } else if (sub->prod<32) {          // ++x
or --x
        Node *sub1 = sub->left->right;
        if (!transAtom(sub1,cst1,rev,val1))
            transError((char*)"L-val Error");//}
        cst = false; // otherwise l-val error
        fout<<'\t'<<rev<<"\t=\t"<<rev;
        if (sub->prod==30)
            fout<<"\t+\t1"<<endl;
        else
            fout<<"\t-\t1"<<endl;
    } else if (sub->prod<35) {          // op x

    transExps(sub->left->right,cst1,x,val1);
        if (cst1) {
            cst = true;
            val = cstHelp(sub->prod,val1);
        } else {
            cst = false;
            getDst(rev);
            fout<<'\t'<<rev<<"\t=\t";

    fout<<sub->left->data<<'\t'<<x<<endl;
        }
    } else if (sub->prod==35)          //
(exps)
       transExps(sub->left->right,cst,rev,val);
   else                                 //
atom
        transAtom(sub,cst,rev,val);
}

// Calculate Constant EXP:
int cstHelp(int op,int val1,int val2)
{
    switch (op) {
    case 2: return (val1+val2);
    case 3: return (val1-val2);
    case 4: return (val1*val2);
    case 5: return (val1/val2);
    case 6: return (val1%val2);
    case 7: return (val1&val2);
    case 8: return (val1^val2);
    case 9: return (val1|val2);
    case 10: return (val1>>val2);
    case 11: return (val1<<val2);
    case 12: return (val1||val2);
    case 13: return (val1&&val2);
    case 14: return (val1|val2);
    case 15: return (val1^val2);
    case 16: return (val1&val2);
    case 17: return (val1!=val2);
    case 18: return (val1==val2);
    case 19: return (val1>val2);
    case 20: return (val1<val2);
    case 21: return (val1<=val2);
    case 22: return (val1>=val2);
    case 23: return (val1<<val2);
    case 24: return (val1>>val2);
    case 25: return (val1+val2);
    case 26: return (val1-val2);
    case 27: return (val1*val2);
    case 28: return (val1/val2);
    case 29: return (val1%val2);
    case 30: return (val1+1);
    case 31: return (val1-1);
    case 32: return (-val1);
    case 33: return (!val1);
    case 34: return (~val1);
    }
}


// Translate Atoms of an Expression:
bool transAtom(Node *sub,bool &cst,char *rev,int
&val)
{
//  the function will consult the Symbol Table to
//      identify variables, struct fields or
functions
//      and return its name by an int or a string
//  the return value indicates whether it can be
a l-val
//  if EXPS is an INT:
//      cst is true, rev invalid, val returns the
INT value
//  else:
//      cst is false, rev returns "<tag> <var>",
val invalid
    bzero(rev,BLK);
    cst = false;
    if (sub->prod==39) {        // INT
        cst = true;
        val = getInt(sub->left);
        return false;
    } else if (sub->prod==38) { // ID DOT ID
        char *tag = getId(sub->left);
        strcpy(rev,tag);
        strcat(rev," ");
        char *id =
getId(sub->left->right->right);
```

```
		strcat(rev,id);
		tbl.srchVar(tag,id,NULL,NULL);
		return true;
	} else if (sub->prod==37) { // ID ARRS
		strcpy(rev,(char*)"? ");
		char *id = getId(sub->left);
		strcat(rev,id);
		Node *arrs = sub->left->right;
		tbl.srchVar((char*)"?",id,arrs,rev);
		return true;
	} else if (sub->prod==36) { // funct(ARGS)
		char *id = getId(sub);
		Node *args = sub->left->right->right;
		if (!strcmp(id,(char*)"read")) {
			if
(args->prod!=2||args->left->prod==2)
				transError((char*)"ARGS
Error");
			else if
(!transAtom(args->left->left,cst,rev,val))
				transError((char*)"READ
Error");
			else

	fout<<"\tread\t"<<rev<<endl;
		} else if (!strcmp(id,(char*)"write")) {
			if
(args->prod!=2||args->left->prod==2)
				transError((char*)"ARGS
Error");
			else {
				getDst(rev);
				transExp(args->left,rev);

	fout<<"\twrite\t"<<rev<<endl;
			}
		} else
			tbl.srchFunc(id,args,rev);
	} else
		return false;
}

// Traverse STSPEC & (SEXTVARS|SDECS):
void transStspec(Node *stspec,Node *vars)
{
	char id[BLK];
	bzero(id,BLK);
	if (stspec->prod==2)
		getDst(id);
	else
		strcpy(id,stspec->left->right->data);
	if (stspec->prod==1)

	tbl.insStrt(id,stspec->left->right->right->rig
ht,vars);
	else if (stspec->prod==2)
		tbl.insStrt(id,stspec->left->right->right,vars)
;
	else if (stspec->prod==3)
		tbl.addStrtVars(id,vars);
}

// Traverse FUNC:
void transFunc(Node *func,Node *stmtBlk)
{
// mark scope, calulate space and set labels
	char *id = getId(func);
	fout<<endl;
	if (strcmp(id,(char*)"main"))
		fout<<"\tgoto\t_"<<id<<endl;
	strcpy(scp,id);
	addrGlobal = addrLocal;
	addrLocal = 0;
	tbl.insFunc(id,func->left->right->right);
	fout<<id<<':'<<endl;
	transStmtBlk(stmtBlk);
	tbl.setFuncSpc(id);
	addrLocal = addrGlobal;
	fout<<"_"<<id<<":\n"<<endl;
	bzero(scp,BLK);
}


// Traverse STMTBLOCK:
void transStmtBlk(Node *stmtBlk,char
*preNext,char *preCont)
{
	char scptmp[BLK];
	bzero(scptmp,BLK);
	strcpy(scptmp,scp);
	bzero(scp,BLK);
	strcpy(scp,(char*)"BLK_");
	itoa(scp,lblHelp++);
	strcat(scp,"@");
	strcat(scp,scptmp);
	fout<<'&'<<scp<<endl;
	transDefs(stmtBlk->left->right);
	transStmts(stmtBlk->left->right->right,preN
ext,preCont);
	fout<<'#'<<scp<<endl;
	strcpy(scp,scptmp);
}


// Traverse STMTS:
void transStmts(Node *stmts,char *preNext,char
*preCont)
{
	if (stmts->prod==1) {

	transStmt(stmts->left,preNext,preCont);

	transStmts(stmts->left->right,preNext,preC
ont);
	}
}

// Traverse STMT:
void transStmt(Node *sub,char *preNext,char
*preCont)
{
	switch (sub->prod) {
	case 1: {			// EXP SEMI
		char dst[BLK];
		bzero(dst,BLK);
		transExp(sub->left,dst);
		break;
	}
	case 2:				// STMTBLOCK

	transStmtBlk(sub->left,preNext,preCont);
		break;
	case 3: {			// RETURN EXP SEMI
		char dst[BLK];
		bzero(dst,BLK);
		getDst(dst);
		transExp(sub->left->right,dst);
		fout<<"\treturn\t"<<dst<<endl;
		break;
	}
	case 4: {			// IF LP EXP RP STMT
ELSE STMT
		char
dst[BLK],Else[BLK],Next[BLK],scpTmp[BLK];
		bzero(dst,BLK);
		getDst(dst);
		newLabel(Next);
		bzero(Else,BLK);
		strcpy(Else,Next);
		strcat(Next,"_NEXT");
		strcat(Else,"_ELSE");
		Node *tmp = sub->left->right->right;
	   if (tmp->prod==2)
		   transError((char*)"IF Condition
Error");
		transExp(tmp,dst);

	fout<<"\tifFalse\t"<<dst<<"\tgoto\t"<<Els
e<<endl;
		tmp = tmp->right->right;
		transStmt(tmp,preNext,preCont);
		fout<<"\tgoto\t"<<Next<<endl;
		fout<<Else<<":\n";

	transStmt(tmp->right->right,preNext,preCo
nt);
		fout<<Next<<":\n";
		break;
	}
	case 5: {			// IF LP EXP RP STMT
		char dst[BLK],Next[BLK],scpTmp[BLK];
		bzero(dst,BLK);
		getDst(dst);
		newLabel(Next);
		strcat(Next,"_NEXT");
		Node *tmp = sub->left->right->right;
	   if (tmp->prod==2)
		   transError((char*)"IF Condition
Error");
		transExp(tmp,dst);

	fout<<"\tifFalse\t"<<dst<<"\tgoto\t"<<Ne
xt<<endl;

	transStmt(tmp->right->right,preNext,preCo
nt);
		fout<<Next<<":\n";
		break;
	}
	case 6: {			// FOR LP EXP SEMI
EXP SEMI EXP RP STMT
		char
dst[BLK],For[BLK],Cont[BLK],Next[BLK],scpTmp[
BLK];
		bzero(dst,BLK);
		newLabel(For);
		bzero(Next,BLK);
		strcpy(Next,For);
		bzero(Cont,BLK);
		strcpy(Cont,For);
		strcat(For,"_FOR");
		strcat(Next,"_NEXT");
		strcat(Cont,"_CONT");
		Node *tmp = sub->left->right->right;
		transExp(tmp,dst);
		fout<<For<<":\n";
		tmp = tmp->right->right;
		bzero(dst,BLK);
		getDst(dst);
		transExp(tmp,dst);

	fout<<"\tifFalse\t"<<dst<<"\tgoto\t"<<Ne
xt<<endl;
		tmp = tmp->right->right;

	transStmt(tmp->right->right,Next,Cont);
		fout<<Cont<<":\n";
		bzero(dst,BLK);
		transExp(tmp,dst);
		fout<<"\tgoto\t"<<For<<endl;
		fout<<Next<<":\n";
		break;
	}
	case 7:				// CONT SEMI
		if (preCont==NULL)
			transError((char*)"CONT Error");
		fout<<"\tgoto\t"<<preCont<<endl;
```

```
                break;
        case 8:                                // BREAK SEMI
                if (preNext==NULL)
                        transError((char*)"BREAK Error");
                fout<<"\tgoto\t"<<preNext<<endl;
        }
}


//////////////////////////////////////
//              MIPS CODE GENERATION                //
//////////////////////////////////////


// Entrance of Code Generation:
void genCode()
{
        fin.open("./output/InterCode");
        if (!fin) {
                cerr<<"Cannot open file \"";
             cerr<<"./output/InterCode\""<<endl;
             exit(2);
       }
        fout.open("./output/MIPSCode.s");
       if (!fout) {
                cerr<<"Cannot open file \"";
             cerr<<"./output/MIPSCode.s\""<<endl;
             exit(3);
       }
        genStart();
        bzero(scp,BLK);      // current scope reset
        // Var Pointer Sentinels Allocation:
        arrHelp[0] = new Var();
        arrHelp[0]->tag = new char[BLK];
        arrHelp[1] = new Var();
        arrHelp[1]->tag = new char[BLK];
        arrHelp[2] = new Var();
        arrHelp[2]->tag = new char[BLK];
        for (int i=0;i<32;i++)
                regfile[i].setIdx(i);
        genCodeHelp();
        // Var Pointer Sentinels Recycling:
        delete arrHelp[0];
        delete arrHelp[1];
        delete arrHelp[2];
        fout.close();
        fin.close();
}

// Start of the MIPS Code:
void genStart()
{
        fout<<"\t.data"<<endl;
        fout<<"GLB_VAR:"<<endl;
        fout<<"\t.space\t"<<(addrLocal+20)<<end
l;
        fout<<"endL:"<<endl;
        fout<<"\t.asciiz\t\"\\n\""<<endl;
        fout<<"\t.text"<<endl;
        fout<<"\t.globl main"<<endl;
        fout<<"main:"<<endl;
}

// Scan Three-Address Instructions:
void genCodeHelp()
{
        char buffer[BLK];
        char scptmp[BLK];
        StrStack stack;
        bzero(buffer,BLK);
        char *inst[12];
        while (true) { // one line per loop
                fin.getline(buffer,BLK);
                if (!strlen(buffer))
                        fout<<endl;
                else if (!strcmp(buffer,(char*)"_END_"))
                        break;
                else if (!strcmp(buffer,(char*)"main:"))
{

        fout<<"_MAIN_STACK_POSITION:"<<endl;


        fout<<"\tli\t$fp ,\t0x7ffffff8"<<endl;
                        char cst[BLK];
                        bzero(cst,BLK);

        itoa(cst,tbl.getStackSz((char*)"main"));
                        getCstReg(cst);

        fout<<"\tsub\t$sp ,\t$fp ,\t"<<regname[25]
<<endl;
                        fout<<"_REAL_MAIN:"<<endl;
                } else if (buffer[0]=='#') {
                        stack.pop(scp);
                        fout<<"\t\t\t\t\t\t#"<<scp<<":\n";
                } else if (buffer[0]=='&') {
                        stack.push(scp);
                        bzero(scp,BLK);
                        strcpy(scp,buffer+1);
                        fout<<"\t\t\t\t\t\t#"<<scp<<":\n";
                } else if
(!strncmp(buffer,(char*)"Label_",6)){
                        descriptorClear(1);
                        fout<<buffer<<endl;
                } else if (buffer[0]!='\t') {
                        fout<<buffer<<endl;
                } else {
                        int pos = 0;
                        inst[pos] = strtok(buffer+1,"\t");
                        while (inst[pos])
                                inst[++pos] =
strtok(NULL,"\t");
                        genInst(pos,inst);
        }
    }
}

// Generate MIPS Instructions:
void genInst(int argc,char *argv[])
{
    char *tag=NULL,*id=NULL;
    if (!strcmp(argv[0],(char*)"goto")) {
    // Unconditional Jump:
        descriptorClear(1);
        fout<<"\tj\t"<<argv[1]<<endl;
    } else if (!strcmp(argv[0],(char*)"ifFalse")) {
    // Conditional Jump:
        int x = getVarReg(getTerm(argv[1]),0);
        descriptorClear(1);
        getCstReg((char*)"0");
        fout<<"\tbeq\t"<<regname[x]<<" ,\t";

    fout<<regname[25]<<" ,\t"<<argv[3]<<e
ndl;
    } else if (!strcmp(argv[0],(char*)"read")) {
    // Function read(int &x)
        fout<<"\t\t\t#begin read
"<<argv[1]<<endl;
        fout<<"\tli\t$v0 ,\t5"<<endl;
        fout<<"\tsyscall"<<endl;
        Var *var = getTerm(argv[1]);
        int z = getVarReg(var,1);

    fout<<"\tmove\t"<<regname[z]<<" ,\t$v0"
<<endl;
        regfile[z].dstAct(var);
        fout<<"\t\t\t#end read
"<<argv[1]<<endl;
    } else if (!strcmp(argv[0],(char*)"write")) {
    // Function write(int x)
        fout<<"\t\t\t#begin write
"<<argv[1]<<endl;
        int x = getVarReg(getTerm(argv[1]),0);

    fout<<"\tmove\t$a0 ,\t"<<regname[x]<<e
ndl;
        fout<<"\tli\t$v0 ,\t1"<<endl;
        fout<<"\tsyscall"<<endl;
        fout<<"\tli\t$v0 ,\t4"<<endl;
        fout<<"\tla\t$a0 ,\tendL"<<endl;
        fout<<"\tsyscall"<<endl;
        fout<<"\t\t\t#end write
"<<argv[1]<<endl;
    } else if
(!strcmp(argv[0],(char*)"_funct_call_"))
        genFuncCall(argv[1]);
    else if
(argc==3&&!strcmp(argv[1],(char*)"="))
        genAssign(argv);
    else if
(argc==4&&!strcmp(argv[1],(char*)"="))
        genUniOp(argv);
    else if
(argc==5&&!strcmp(argv[1],(char*)"="))
        genBinOp(argv);
    else if (!strcmp(argv[0],(char*)"return")) {
    // Instruction RETURN VAR
        int x = getVarReg(getTerm(argv[1]),0);

    fout<<"\tmove\t$v0 ,\t"<<regname[x]<<e
ndl;
        descriptorClear(1);
        fout<<"\tjr\t$ra"<<endl;
    }
}

// Issue a Function Call:
void genFuncCall(char *id)
{
    fout<<"\t\t\t#begin calling "<<id<<endl;
    // Store the Arguments:
    char inst[BLK];
    int cnt = 0;
    while (true) {
        bzero(inst,BLK);
        fin.getline(inst,BLK);
        if (strncmp(inst,(char*)"\tparam\t",7))
            break;
        else {     // deal with the (cnt)th
argument
            int x =
getVarReg(getTerm(inst+7),0);

    fout<<"\tsw\t"<<regname[x]<<" ,\t-";

    fout<<(4*(++cnt))<<"($sp)"<<endl;
        }
    }
    descriptorClear(1);
    // store register $fp and $ra
    fout<<"\tsw\t$fp ,\t0($sp)"<<endl;
    fout<<"\tsw\t$ra ,\t4($sp)"<<endl;
    // move pointers $fp and $sp
    fout<<"\taddi\t$fp ,\t$sp ,\t-4"<<endl;
    int stackSize = tbl.getStackSz(id);
    char cst[BLK];
    bzero(cst,BLK);
    itoa(cst,stackSize);
    getCstReg(cst);
    fout<<"\tsub\t$sp ,\t$sp ,\t"<<regname[25]
<<endl;
    // jump and link
    fout<<"\tjal\t"<<id<<endl;
    // restore $sp, $fp and $ra
    fout<<"\taddi\t$sp ,\t$fp ,\t4"<<endl;
    fout<<"\tlw\t$fp ,\t0($sp)"<<endl;
    fout<<"\tlw\t$ra ,\t4($sp)"<<endl;
    // Deal with the Return Value
    char *rev = strtok(inst+1,"\t");
    Var *var = getTerm(rev);
    int y = getVarReg(var,1);
    fout<<"\tmove\t"<<regname[y]<<" ,\t$v0\
n";
    regfile[y].dstAct(var);
    fout<<"\t\t\t#end calling "<<id<<endl;
}

// Decode z = x op y:
void genBinOp(char *argv[])
{
    // Allocate Registers:
    int x,y,z;
    Var *a = getTerm(argv[2]);
    if (a!=NULL)
        x = getVarReg(a,0);
    else {
        getCstReg(argv[2]);
        x = 25;
    }
    regUsed |= (1<<x);
    Var *b = getTerm(argv[4]);
    if (b!=NULL)
        y = getVarReg(b,0);
    else {
        getCstReg(argv[4]);
        y = 25;
    }
    regUsed |= (1<<y);
    Var *c = getTerm(argv[0]);
    z = getVarReg(c,1);
    regUsed |= (1<<z);
    // Decode the Operator:
    char *op = argv[3];
    if (!strcmp(op,(char*)"||")
        ||    !strcmp(op,(char*)"|"))
        fout<<"\tor\t";
    else if (!strcmp(op,(char*)"&&")
        ||    !strcmp(op,(char*)"&"))
        fout<<"\tand\t";
    else if (!strcmp(op,(char*)"<<"))
        fout<<"\tsll\t";
    else if (!strcmp(op,(char*)">>"))
        fout<<"\tsrl\t";
    else if (!strcmp(op,(char*)"^"))
        fout<<"\txor\t";
    else if (!strcmp(op,(char*)"=="))
        fout<<"\tseq\t";
    else if (!strcmp(op,(char*)"!="))
        fout<<"\tsne\t";
    else if (!strcmp(op,(char*)"<"))
        fout<<"\tslt\t";
    else if (!strcmp(op,(char*)">"))
        fout<<"\tsgt\t";
    else if (!strcmp(op,(char*)"<="))
        fout<<"\tsle\t";
    else if (!strcmp(op,(char*)">="))
        fout<<"\tsge\t";
    else if (!strcmp(op,(char*)"+"))
        fout<<"\tadd\t";
    else if (!strcmp(op,(char*)"-"))
        fout<<"\tsub\t";
    else if (!strcmp(op,(char*)"*"))
        fout<<"\tmulo\t";
    else if (!strcmp(op,(char*)"/"))
        fout<<"\tdiv\t";
    else if (!strcmp(op,(char*)"%"))
        fout<<"\trem\t";
    else
        fout<<"\tBIN_OP\t";
    // Issue the Instruction
    fout<<regname[z]<<" ,\t";
    fout<<regname[x]<<" ,\t";
    fout<<regname[y]<<endl;
    regfile[x].clearArrHelp(a);
    regfile[y].clearArrHelp(b);
    regfile[z].dstAct(c);
}

// Decode z = op x:
void genUniOp(char *argv[])
{
    // Allocate Registers:
    int x,z;
    Var *a = getTerm(argv[3]);
    if (a!=NULL)
        x = getVarReg(a,0);
    else {
        getCstReg(argv[3]);
        x = 25;
    }
    regUsed |= (1<<x);
    Var *c = getTerm(argv[0]);
    z = getVarReg(c,1);
    regUsed |= (1<<z);
    // Decode the Operator:
    char *op = argv[2];
    if (!strcmp(op,(char*)"-"))
        fout<<"\tneg\t";
    else if (!strcmp(op,(char*)"~"))
        fout<<"\tnot\t";
    else if (strcmp(op,(char*)"!"))
        fout<<"\tUNI_OP\t";
    else {
        getCstReg((char*)"1");
        fout<<"\tsub\t"<<regname[z]<<" ,\t";

    fout<<regname[25]<<" ,\t"<<regname[x]
<<endl;
        regfile[x].clearArrHelp(a);
        regfile[z].dstAct(c);
        return;
    }
    // Issue the Instruction:
    fout<<regname[z]<<" ,\t";
    fout<<regname[x]<<endl;
    regfile[x].clearArrHelp(a);
    regfile[z].dstAct(c);
}


// Decode z = x:
void genAssign(char *argv[])
{
    // Allocate Registers:
    int x,z;
    Var *rval = getTerm(argv[2]);
    if (rval!=NULL)
        x = getVarReg(rval,0);
    else {
        getCstReg(argv[2]);
        x = 25;
    }
    regUsed |= (1<<x);
    Var *lval = getTerm(argv[0]);
    z = getVarReg(lval,1);
    regUsed |= (1<<z);
    // Issue the Instruction:
    fout<<"\tmove\t"<<regname[z]<<" ,\t"<<r
egname[x]<<endl;
    regfile[x].clearArrHelp(rval);
    regfile[z].dstAct(lval);
}

// Decode an Operand String <tag id>:
Var *getTerm(char *str) {
// return tbl pointer for non-array vars
// return sentinel pointer for array vars
// return NULL for an immediate
    if (str[0]<0||str[0]>'9') {
        char buf[BLK];
        bzero(buf,BLK);
        strcpy(buf,str);
        char *tag = strtok(buf," ");
        char *id = strtok(NULL," ");
        char *arr = strtok(NULL,"]");
        if (arr==NULL)
            return tbl.search(tag,id);
        else {
            Var *base = tbl.search(tag,id);
            arrHelp[arrFlag]->scope =
base->scope;
            bzero(arrHelp[arrFlag]->tag,BLK);
            arrHelp[arrFlag]->id = base->id;
            arrHelp[arrFlag]->addr =
base->addr;
            if (arr[1]<'0'||arr[1]>'9')

    strcpy(arrHelp[arrFlag]->tag,arr);
```

```
                else

    arrHelp[arrFlag]->addr+=4*atoi(arr+1);
                arrHelp[arrFlag]->mem = true;
                arrHelp[arrFlag]->reg = 0;
                return arrHelp[arrFlag++];
            }
    } else
        return NULL;
}

// Get a Register for a Variable:
int getVarReg(Var *var,bool dst)
{
    if (!dst) {     // As a Src Operand:
        // a reg currently storing var
        for (int i=8;i<25;i++) {
            if (regfile[i].search(var))
                return i;
        }
        // get an empty register
        for (int i=8;i<25;i++) {
            if (regfile[i].empty()) {
                regfile[i].load(var);
                return i;
            }
        }
        // a reg that stores other values
        int min = MAX, pos = 0, cost;
        for (int i=8;i<25;i++) {
            if (regUsed&(1<<i))
                continue;
            cost = regfile[i].getCost();
            if (cost==0) {
                regfile[i].load(var);
                return i;
            } else if (cost<min) {
                min = cost;
                pos = i;
            }
        }
        regfile[pos].spill();
        regfile[pos].load(var);
        return pos;
    } else {        // As a Dst Operand:
        // a reg that is empty or only stores var
        for (int i=8;i<25;i++) {
            if (regfile[i].bestDst(var))
                return i;
        }
        // a reg that stores other values
        int min = MAX, pos = 0, cost;
        for (int i=8;i<25;i++) {
            if (regUsed&(1<<i))
                continue;
            cost = regfile[i].getCost();
            if (cost==0)
```

```
                return i;
            else if (cost<min) {
                min = cost;
                pos = i;
            }
        }
        regfile[pos].spill();
        return pos;
    }
}

// Store an Immediate in $t9:
void getCstReg(char *cst)
{
    fout<<"\tli\t"<<regname[25];
    fout<<" ,\t"<<cst<<endl;
}

// Issue a SW or LW Instruction:
void issue(bool store,int idx,Var *var)
{
    if ((var==arrHelp[0]||var==arrHelp[1]
        ||var==arrHelp[2])&&strlen(var->tag))
{
        // if address is not in var->addr:
        //      get address in $t9
        char buf[BLK];
        bzero(buf,BLK);
        strcpy(buf,var->tag);
        char *tag = strtok(buf," ")+1;
        char *id = strtok(NULL," ");
        Var *idxVar = tbl.search(tag,id);
        int x = getVarReg(idxVar,0);
        // addr = 4*index+addrOfBase
        fout<<"\tsll\t"<<regname[25]<<" ,\t";
        fout<<regname[x]<<" ,\t2"<<endl;

    fout<<"\taddi\t"<<regname[25]<<" ,\t";

    fout<<regname[25]<<" ,\t"<<var->addr<
<endl;
        if (!strlen(var->scope)) {      // data
section
            if (store)
                fout<<"\tsw\t";
            else
                fout<<"\tlw\t";

    fout<<regname[idx]<<" ,\tGLB_VAR(";
            fout<<regname[25]<<")\n";
        } else {                        // stack

    fout<<"\tsub\t"<<regname[25]<<" ,\t$fp";

    fout<<" ,\t"<<regname[25]<<endl;
            if (store)
                fout<<"\tsw\t";
```

```
            else
                fout<<"\tlw\t";
            fout<<regname[idx]<<" ,\t(";
            fout<<regname[25]<<")\n";
        }
    } else if (!strlen(var->scope)) {
        // Data Section Address in var->addr
        char cst[BLK];
        bzero(cst,BLK);
        itoa(cst,var->addr);
        getCstReg(cst);
        if (store)
            fout<<"\tsw\t";
        else
            fout<<"\tlw\t";
        fout<<regname[idx]<<" ,\tGLB_VAR(";
        fout<<regname[25]<<")\n";
    } else {
        // Stack Address in var->addr
        if (store)
            fout<<"\tsw\t";
        else
            fout<<"\tlw\t";
        fout<<regname[idx]<<" ,\t-";
        fout<<var->addr<<"($fp)\n";
    }
}

void descriptorClear(bool st)
{
    for (int i=8;i<25;i++)
        regfile[i].clear(st);
}

void descriptorShow()
{
    for (int i=8;i<25;i++) {
        cout<<regname[i]<<":\n";
        regfile[i].showHelp();
    }
}


////////////////////////////////////////////
//          IMPLEMENTATION of 'header.h'          //
////////////////////////////////////////////


//////// PARSE TREE NODE ////////


Node::Node(char *str,int p) {
    data = strdup(str);
    left = NULL;
    right = NULL;
    prod = p;
}
```

```
Node::~Node() {
        free(data);
        delete left;
        delete right;
}


//////// SYMBOL TABLE ELEMENTS ////////

Size::Size(int sz) {
        data = sz;
        next = NULL;
}

Size::~Size() {
        delete next;
}

Var::Var() {
        scope = NULL;
        tag = NULL;
        id = NULL;
        size = new Size(0);
        next = NULL;
        addr = 0;
        mem = true;
        reg = 0;
}

Var::~Var() {
        free(scope);
        free(tag);
        free(id);
        delete size;
        delete next;
}

bool Var::backUp() const {
        if (!mem)
                return (ham(reg)>1);
        else
                return true;
}

Fld::Fld(char *id) {
        name = strdup(id);
        next = NULL;
}

Fld::~Fld() {
        free(name);
        delete next;
}

Strt::Strt() {
        scope = NULL;
        id = NULL;
        fld = new Fld((char*)"");
        next = NULL;
}

Strt::~Strt() {
        free(scope);
        free(id);
        delete fld;
        delete next;
}

Func::Func() {
        id = NULL;
        argc = 0;
        spc = 0;
        next = NULL;
}

Func::~Func() {
        free(id);
        delete next;
}


//////// HASHING FOR SYMBOL TABLE ////////


// Hashing Function (division method):
int Hash::getIndex(char *id) const {
        int val = 0;
        for (int i=0;i<strlen(id);i++)
                val = (val*128+id[i])%NUM;
        return val;
}

// Insert a Variable If Declared:
void Hash::insVar(char *tag,char *id,Node *var) {
        int pos = getIndex(id);
        Var *itr = &varTbl[pos];
        while (itr->next!=NULL) {
                if (!strcmp(itr->next->scope,scp)
                &&   sameTag(itr->next->tag,tag)
                &&   !strcmp(itr->next->id,id))
                        transError((char*)"Variable
Redefined");
                itr = itr->next;
        }
        itr->next = new Var();
        itr = itr->next;
        itr->scope = strdup(scp);
        itr->tag = strdup(tag);
        itr->id = strdup(id);
        itr->addr = addrLocal;
        if (var!=NULL)
                getVarSize(itr->size,var);
        int tmp = 4;
```

```
    Size *sz = itr->size->next;
    while (sz!=NULL) {
        tmp *= sz->data;
        sz = sz->next;
    }
    addrLocal += tmp;
}

// Traverse VAR and Generate Size List:
void Hash::getVarSize(Size* sz,Node *var) {
    if (var->prod==2) {
        Size *tmp = new
Size(getInt(var->left->right->right));
        tmp->next = sz->next;
       sz->next = tmp;
        getVarSize(sz,var->left);
    }
}

// Insert a Structure If Declared:
void Hash::insStrt(char *id,Node *sdefs,Node
*vars) {
    int pos = getIndex(id);
    Strt *itr = &strtTbl[pos];
    while (itr->next!=NULL) {
        if (!strcmp(itr->next->scope,scp)
        &&   !strcmp(itr->next->id,id))
            transError((char*)"Structure
Redefined");
        itr = itr->next;
    }
    itr->next = new Strt();
    itr = itr->next;
    itr->scope = strdup(scp);
    itr->id = strdup(id);
    getStrtFld(itr->fld,sdefs);
    addStrtVarsHelp(itr->fld,vars);
}

// Traverse SDEFS:
void Hash::getStrtFld(Fld* fld,Node *sdefs) {
    if (sdefs->prod==1) {
        getStrtFldHelp(fld,sdefs->left->right);

    getStrtFld(fld,sdefs->left->right->right->rig
ht);
    }
}

// Traverse SDECS and Generate Fld List:
void Hash::getStrtFldHelp(Fld* &fld,Node *sdecs)
{
    fld->next = new Fld(getId(sdecs->left));
    fld = fld->next;
    if (sdecs->prod==1)

    getStrtFldHelp(fld,sdecs->left->right->right)
```

```
;
}

// Add Structure Variables:
void Hash::addStrtVars(char *id,Node *vars) {
    int pos = getIndex(id);
    Strt *itr = strtTbl[pos].next;
    while (itr!=NULL) {
        if (sameScp(itr->scope,scp)
        &&   !strcmp(itr->id,id)) {
            addStrtVarsHelp(itr->fld,vars);
            return;
        }
        itr = itr->next;
    }
    transError((char*)"Structure Undefined");
}

// Traverse SEXTVARS or SDECS:
void Hash::addStrtVarsHelp(Fld *fld,Node *vars) {
    if (vars->prod!=3) {
        char *tag = getId(vars);
        Fld *itr = fld->next;
        while (itr!=NULL) {
            insVar(tag,itr->name,NULL);
            itr = itr->next;
        }
        if (vars->prod!=2)

    addStrtVarsHelp(fld,vars->left->right->right)
;
    }
}

// Insert a Function If Declared:
void Hash::insFunc(char *id,Node *paras) {
    int pos = getIndex(id);
    Func *itr = &funcTbl[pos];
    while (itr->next!=NULL) {
        if (!strcmp(itr->next->id,id))
            transError((char*)"Function
Redefined");
        itr = itr->next;
    }
    itr->next = new Func();
    itr = itr->next;
    itr->id = strdup(id);
    itr->argc = 0;
    if (paras!=NULL)
        getFuncParas(itr->argc,paras);
}

// Traverse PARAS and Add Parameters:
void Hash::getFuncParas(int &argc,Node *paras)
{
    if (paras->prod!=3) {
        char buf[BLK];
```

```
            bzero(buf,BLK);
            buf[0] = '#';
            itoa(buf,argc++);

        insVar(buf,getId(paras->left->right),NULL);
            if (paras->prod!=2)

        getFuncParas(argc,paras->left->right->right
->right);
        }
}

// Record Stack Size of a Function:
void Hash::setFuncSpc(char *id) {
        int pos = getIndex(id);
        Func *itr = funcTbl[pos].next;
        while (itr!=NULL) {
            if (!strcmp(itr->id,id)) {
                itr->spc = addrLocal;
                break;
            }
            itr = itr->next;
        }
}

// Idenitify Variables or Struct Fields:
void Hash::srchVar(char *tag,char *id,Node
*arrs,char *idx) const {
//      when consulting struct fields or funct paras,
//          arrs==NULL, idx==NULL, no need to
test ARRS
//      otherwise
//          test ARRS and append idx with index
var/cst
        int pos = getIndex(id);
        Var *itr = varTbl[pos].next;
        while (itr!=NULL) {
            if (sameScp(itr->scope,scp)
            &&   sameTag(itr->tag,tag)
            &&   !strcmp(itr->id,id)) {
                if (arrs==NULL) return;
                else if (arrs->prod==2) {
                    if (itr->size->next!=NULL)
                        transError((char*)"ARRS
Error");
                    return;
                } else {
                    if (itr->size->next==NULL)
                        transError((char*)"ARRS
Error");
                    char buf[BLK];
                  bzero(buf,BLK);
                    Size *sz =
itr->size->next->next;
                    transArrs(sz,arrs,buf);
                    strcat(idx,(char*)" [");
                    strcat(idx,buf);
```

```
                    strcat(idx,(char*)"]");
                    return;
                }
            }
            itr = itr->next;
        }
        transError((char*)"Variable Undefined");
}

// Traverse and Test ARRS
void Hash::transArrs(Size *sz,Node *arrs,char
*rev) const {
      bool cst, cst1;
      int val, val1;
      char rev1[BLK];
      // get EXP_0 as initial idx
      if (arrs->prod==2)
          transError((char*)"ARRS Error");
      Node *exp = arrs->left->right;
      if (exp->prod==2)
          transError((char*)"ARRS Error");
       transExps(exp->left,cst,rev,val);
       if (!cst) {
           char dst[BLK];
           bzero(dst,BLK);
           getDst(dst);

       fout<<'\t'<<dst<<"\t=\t"<<rev<<endl;
           strcpy(rev,dst);
       }
      arrs = exp->right->right;
      // idx = idx*SIZE_k+EXP_k (iterately)
      while (sz!=NULL) {
          // multiplication: idx *= SIZE_k
          if (cst)
              val = val*(sz->data);
          else {
              fout<<'\t'<<rev<<"\t=\t"<<rev;
              fout<<"\t*\t"<<sz->data<<endl;
          }
          // get EXP_k in rev1 or val1
          if (arrs->prod==2)
              transError((char*)"ARRS Error");
          exp = arrs->left->right;
          if (exp->prod==2)
              transError((char*)"ARRS Error");
          bzero(rev1,BLK);
          transExps(exp->left,cst1,rev1,val1);
          // addition: idx += EXP_k
          if (cst&&cst1)
              val += val1;
          else if (cst){
              cst = false;
              strcpy(rev,rev1);
              fout<<'\t'<<rev<<"\t=\t"<<val;
              fout<<"\t+\t"<<rev1<<endl;
          } else {
            cst = false;
            fout<<'\t'<<rev<<"\t=\t"<<rev;
            if (cst1)
                fout<<"\t+\t"<<val1<<endl;
            else
                fout<<"\t+\t"<<rev1<<endl;
        }
        // for next loop
        arrs = exp->right->right;
        sz = sz->next;
    }
    if (arrs->prod!=2)
        transError((char*)"ARRS Error");
    if (cst)
        itoa(rev,val);
}

// Identify and Translate Function Call:
void Hash::srchFunc(char *id,Node *args,char
*dst) const {
    int pos = getIndex(id);
    Func *itr = funcTbl[pos].next;
    while (itr!=NULL) {
        if (!strcmp(itr->id,id)) {
            if (!testArgc(itr->argc,args))
                transError((char*)"ARGS
Error");
            else if (strcmp(id,(char*)"main")) {
                calArgs(itr->id,itr->argc,args);
                getDst(dst);

    fout<<'\t'<<dst<<"\t=\tcall\t"<<id;

    fout<<'\t'<<itr->argc<<endl;
            } else if
(args->prod!=2||args->left->prod!=2)
                transError((char*)"ARGS
Error");
            return;
        }
        itr = itr->next;
    }
    transError((char*)"Function Undefined");
}

// Traverse ARGS to Test Number of Arguments:
bool Hash::testArgc(int argc,Node *args) const {
    if (argc==0)
        return (args->prod==2 &&
args->left->prod==2);
    for (int i=0;i<argc-1;i++) {
        if (args->prod==2)
            return false;
        args = args->left->right->right;
    }
    return args->prod==2;
}
```

```
// Traverse ARGS to Translate Arguments:
void Hash::calArgs(char *id,int argc,Node *args)
const {
    char buffer[30][BLK];
    for (int i=0;i<argc;i++) {
        bzero(buffer[i],BLK);
        getDst(buffer[i]);
        transExp(args->left,buffer[i]);
        if (args->prod!=2)
            args = args->left->right->right;
    }
    fout<<"\t_funct_call_\t"<<id<<endl;
    for (int i=0;i<argc;i++)
        fout<<"\tparam\t"<<buffer[i]<<endl;
}

// Test Whether Entrance Exists:
void Hash::callMain() const {
    char buf[BLK];
    bzero(buf,BLK);
    Node *args = new Node((char*)"ARGS",2);
    args->left = new Node((char*)"EXP",2);
    srchFunc((char*)"main",args,buf);
}

// Consult the Stack Size of a Function:
int Hash::getStackSz(char *id) const {
    int pos = getIndex(id);
    Func *itr = funcTbl[pos].next;
    while (itr!=NULL) {
        if (!strcmp(itr->id,id))
            break;
        itr = itr->next;
    }
    return itr->spc;
}

// Search a Variable in Code Generation:
Var *Hash::search(char *tag,char *id) const {
    int pos=getIndex(id),maxLen=-1,len;
    Var *itr=varTbl[pos].next, *rev=NULL;
    while (itr!=NULL) {
        len = strlen(itr->scope);
        if (sameScp(itr->scope,scp)
        &&   sameTag(itr->tag,tag)
        &&   !strcmp(itr->id,id)
        &&  len>maxLen) {
            rev = itr;
            maxLen = strlen(itr->scope);
        }
        itr = itr->next;
    }
    return rev;
}
```

```
//////// SCOPE DETERMINATION ////////


StrNode::StrNode() {
        bzero(data,BLK);
        next = NULL;
}

StrNode::~StrNode() {
        delete next;
}


StrStack::StrStack() {
        head = new StrNode();
}

StrStack::~StrStack() {
        delete head;
}

void StrStack::push(char *str) {
        StrNode *tmp = new StrNode();
        strcpy(tmp->data,str);
        tmp->next = head->next;
        head->next = tmp;
}

bool StrStack::pop(char *str) {
        if (head->next==NULL)
                return false;
        bzero(str,BLK);
        StrNode *tmp = head->next;
        strcpy(str,tmp->data);
        head->next = tmp->next;
        tmp->next = NULL;
        delete tmp;
        return true;
}


//////// REGISTER DESCRIPTOR ////////


Ref::Ref(Ref *n) {
        var = NULL;
        next = n;
}

Ref::~Ref() {
        var = NULL;
        delete next;
}

Reg::Reg() {
        ref = new Ref();
}
```

```
Reg::~Reg() {
        delete ref;
}

void Reg::setIdx(int i) {
        idx = i;
}

void Reg::insert(Var *var) {
        Ref *r = new Ref(ref->next);
        r->var = var;
        ref->next = r;
}

bool Reg::remove(Var *var) {
        Ref *itr = ref;
        while (itr->next!=NULL) {
                if (itr->next->var==var) {
                        Ref *r = itr->next;
                        itr->next = r->next;
                        r->next = NULL;
                        //delete r;
                        return true;
                }
                itr = itr->next;
        }
        return false;
}

bool Reg::search(Var *var) const {
        Ref *itr = ref->next;
        while (itr!=NULL) {
                if (itr->var==var)
                        return true;
                itr = itr->next;
        }
        return false;
}

// Whether It's Empty:
bool Reg::empty() const {
        if (idx<8||idx>24)
                return false;
        else
                return ref->next==NULL;
}

// Empty or Only Storing DstVar:
bool Reg::bestDst(Var *var) const {
        if (idx<8||idx>24)
                return false;
        else if (ref->next==NULL)
                return true;
        else if (ref->next->next==NULL)
                return ref->next->var==var;
        else
        return false;
}

//
int Reg::getCost() const {
    int cost = 0;
    Ref *itr = ref->next;
    while (itr!=NULL) {
        if (!itr->var->backUp())
            cost++;
        itr = itr->next;
    }
    return cost;
}

// Release src Sentinel Var Pointers:
void Reg::clearArrHelp(Var *var) {
    if (var==arrHelp[0]||var==arrHelp[1]||
        var==arrHelp[2])
        remove(var);
}

// Changes on Chosen as a DstReg:
void Reg::dstAct(Var *var) {
    // Other vars' Address Descriptor:
    Ref *itr = ref->next;
    while (itr!=NULL) {
        itr->var->reg &= (~(1<<idx));
        itr = itr->next;
    }
    // Modify the Register Descriptor:
    delete ref->next;
    ref->next = NULL;
    if (var==arrHelp[0]||var==arrHelp[1]
        ||var==arrHelp[2])
        store(var);
    else
        insert(var);
    var->reg = (1<<idx);
    var->mem = false;
    // Modify global marks
    regUsed = 0;
    arrFlag = 0;
}

// Instruction: LW reg[idx], <var_addr>
void Reg::load(Var *var) {
    // Other vars' Address Descriptor:
    Ref *itr = ref->next;
    while (itr!=NULL) {
        itr->var->reg &= (~(1<<idx));
        itr = itr->next;
    }
    // Modify the Register Descriptor:
    delete ref->next;
    ref->next = NULL;
    insert(var);
```

```
    // var's Own Address Descritpor:
    var->reg |= (1<<idx);
    issue(0,idx,var);
}

// Instruction: SW reg[idx], <var_addr>
void Reg::store(Var *var) {
    // var's Address Descriptor:
    var->mem = true;
    issue(1,idx,var);
}

// Backup All Vars that It Stores:
void Reg::spill() {
    Ref *itr = ref->next;
    while (itr!=NULL) {
        if (!itr->var->backUp())
            store(itr->var);
        itr = itr->next;
    }
}

// Clear the Register Descriptor:
void Reg::clear(bool st) {
//  write back vars if st==1
    Ref *itr = ref->next;
    while (st&&itr!=NULL) {
        if (!itr->var->mem)
            store(itr->var);
        itr->var->reg &= (~(1<<idx));
        itr = itr->next;
    }
    delete ref->next;
    ref->next = NULL;
}

void Reg::showHelp()
{
    Ref *itr = ref->next;
    while (itr!=NULL) {
        Var *var = itr->var;
        cout<<'\t'<<var->scope<<endl;
        cout<<'\t'<<var->tag<<endl;
        cout<<'\t'<<var->id<<endl;
        itr = itr->next;
    }
}
```