Report of the Compiler Project2

Zebing Lin 5120309085

December 19, 2014

1 Introduction

In this project, we are required to design and implement a simplified compiler, which translates the given language smallc source codes into **MIPS** assembly codes. I finished this project following five steps:

1. Construct the abstract syntax tree.
2. Check semantic errors.
3. Translate the parse tree into an intermediate representation.
4. Optimize and allocate registers.
5. Interpret the intermediate representation into MIPS code.

This report will introduce my design and implementation of the five parts.

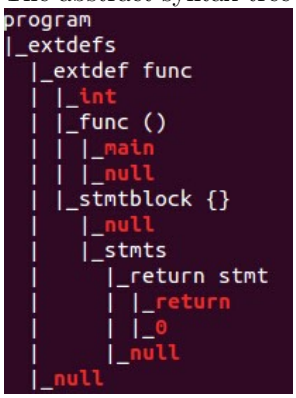
2 Syntax Tree Construction

This part is implemented in **def.h**, **ast.h** and **smallc.y**.

Compiler would do lexical analysis first, and the result of which would be the input of lexical analysis, then we can construct the abstract syntax tree through syntax analysis. For example, for the input

```
int main() {
    return 0;
}
```

The abstract syntax tree would be like this:



I defined a struct type **TreeNode** in **def.h**. It has several members. The explanation of each member is given in the following codes.

```
typedef struct TreeNode {
    TreeNode type; //type of the tree node
    int line_num; //the corresponding line number of the treenode
    char* data; //description of the tree node
    int size, capacity; //size is the number of children this node has
    //capacity is the maximum number of children it can have
    struct TreeNode** children; //its children
} TreeNode;
```

Some of the work has already been done in project1. Bison will parse the source code for you. To construct the parse tree, I implemented a function named *create_node()*,which assigns values to its members while creating its children. Since the number of children is unfixed, so using MACROs in *stdarg.h* is convenient. In **smallc.y** I just add syntactic actions to every production rule while the return value is assigned to $$. To check the intermediate result, I also implemented *print_ast()* to print the abstract syntax tree in the terminal.

The abstract syntax tree indicates the inheritance relation between non-terminal symbols and terminal symbols. The syntactic actions gather information needed for the next steps. The semantics analysis part and the code generation part would both be conducted by traversing from the root of the tree.

3 Semantics Analysis

This part is mainly implemented in **semantics.h**.

The semantics analysis part is to check the semantic errors of the source codes. **I have dealt with all the possible errors mentioned in the slides and some other possible errors I thought by myself.** I would like to introduce my implementation in following three parts:

- The changes I made to the productions in *smallc.y* to prevent some semantic errors.
- The implementation of my symbol table.
- The specific methods I used to check the semantic errors.

I have a function named *report_err()* to report the line number and the specific information of the errors with *exit(1)* in the end.

3.1 Changes to the production rules

To make my semantic analysis easier, I modified the semantic rules. Mainly it is to prevent the case EXP $\rightarrow \epsilon$. Obviously, statements like *if (); else;* is forbidden, so I changed the production rule "STMT $\rightarrow$ IF LP EXP RP STMT" to "STMT $\rightarrow$ IF LP EXPS RP STMT". Likely, the return type would never be void, and compiler would never accept an initialization like *int a = ;* , so I modified the corresponding *EXP* to *EXPS*.

3.2 Symbol Table

Symbol table os the data structure to hold information about the variables of the source programs. The symbol table records the type declarators, variables and the information of parameters and return type of functions. Considering the symbol table needs to hold a lot of information, I defined a struct to hold the info.

```
struct SymbolTable {
    map <char*, char*,ptr_cmp> table;
    map <char*, vector<char *>, ptr_cmp > struct_table;
    map <char*, vector<char *>, ptr_cmp > struct_id_table;
    map < char*, int, ptr_cmp > struct_name_width_table;
```

```
    map <char*, int, ptr_cmp> width_table;
    //records the num of "int"s, the actual width should be 4 times of that
        map <char*, vector<int>, ptr_cmp> array_size_table;
        int parent_index;
    //which record the index of his parent in the upper level;
    //-1 means it's the gloable scope
} env[MAX_LEVEL][MAX_DEPTH];
```

For efficiency of inquiry, I used the <map> in STL and I used char* rather than string to store the data of each node, so redefining comparison is necessary. I would introduce the codes line by line.

3.2.1 map <char*, char*,ptr_cmp> table

It maps variable names to types. There are two types, either a struct or an int.

3.2.2 map <char*, vector<char *>, ptr_cmp > struct_table

It maps variables of a struct to a vector which contains the name of the members of the struct varaible.

3.2.3 map <char*, vector<char *>, ptr_cmp > struct_id_table

It maps the name of the type of the struct to a vector which contains the name of the members of this type of struct. Notice that it's different from the second table, for example:

```
    struct a {
        int num1;
        int num2;
    } b;
```

The second table maps the struct variable "b" to a vector which contains num1 and num2, while this one maps the name of the struct type "a" to a vector which contains num1 and num2. This one is mainly deal with the "STSPEC → STRUCT ID" case.

3.2.4 map <char*, int, ptr_cmp > struct_name_width_table

It maps name of struct variables to the number of members of this struct, which is later used in the code generation part.

3.2.5 map <char*, int, ptr_cmp > width_table

It maps name of integers and arrays to how many 4 bytes they occupy.

3.2.6 map <char*, vector<char *>, ptr_cmp > array_size_table

It maps name of arrays to a vector which contains how many 4 bytes of the width each dimension. For example, for an array a[5][8], it will contain a vector which contains 8 and 1, which multiplies 4 is the width of each dimension.

3.2.7 parent_index

It records the location of the upper namespace, since a C program can have multiple namespaces.

3.2.8 env[MAX_LEVEL][MAX_DEPTH]

It will contain all the symbol tables of each namespace. Due to the properties of the C language, a program could have many levels. When never we met a new {, which means we have entered a new namespace, we have to construct a new symbol table of this namespace, level increases and the number of namespaces(or depth) of the original namespace has to be updated; when we meet } and leave the current namespace, we have to get back to the original namespace using *parent_index* where it's embedded, which means level decreases.

3.3 Specific Techniques of Error Checking

3.3.1 Variables and functions have to declared before usage

For variables usage, I have a function named *semantics_check_id()* to search for the name of the variable. At the declaration stage, the info has already been stored. So this function just search its current namespace to find if names match, otherwise it will go to the parent namespace, until the global scope. If still not found, it will call *report_err()*.

Functions are dealt differently. Functions can only be defined in the global scope, so I have *func_table* to map function names to the vector of the number of parameters. Notice the it's a vector here, since we have to take **overloaded function** into consideration. The **overloaded function** would be covered later. When a function is called, we have to check whether the function name exist and whether number of parameters match.

3.3.2 Variables and functions should not be redeclared

Likely, in the declaration scope, we use *semantics_check_id()* to find whether this name has already been declared as struct, also we have to check the *func_table* to see whether there are functions with the same name.

3.3.3 Reserved words can not be used as identifiers

I have a function named *isReserved()* to check whether name of id coincides with reserved words.

3.3.4 Program must contain a function int main() to be the entrance

I have a bool variable named *bool_main*, which is initially false, when we meet the definition of the *main()* function, we will modify it to be true. After the traversal of the syntax tree, we will check *bool_main*, if it's false, we will report error.

3.3.5 The number of variables passed should match definition of vector

I have a *func_cnt_table* to record the number of arguments of a function call, and I will check if it matches the definition, which could be done via *func_table*. It's worth mentioning that there may be cases that the parameters of a function are also function calls, so whenever there is a function call, we have to store the number of parameters temporarily and restores afterwards.

3.3.6 Use [] operator a non-array variable is not allowed

I have member named *width_table*, along with a function named *get_width_semantics()*, to search for variables and return how many 4 bytes it has. So when we meet the [] operator, we check whether the variable is a struct first, if not, we get its width, if the width is one, then it's not an array, error.

However, there is a defect in this. When you declare an array of 4 bytes like *int a[1]*, my implementation is unable to distinguish it from an int.

3.3.7 The . operator can only be used to a struct variable

First we use the function *find_struct_id* to check whether this variable has been declared as struct, also we get the namespace of its declaration, then we refer to *struct_id_table* to get the vector <char*>, and check whether the id after dot is a member of the struct.

3.3.8 break and continue can only be used in a loop

I defined a boolean variable named *in_for*, when we enter a for loop, it's modified to be true, when we exit the loop, it's modified to be false. Then when we meet break and continue, we check *in_for* first, if not in a loop, then report error.

3.3.9 Right-value can not be assigned by any value to expression

In this simplified C language, there are limited kinds of expressions that could be a left value, which includes "ID DOT ID", "ID ARRS", "EXPS ASSIGN EXPS" and etc.. So I have a function named *check_left_value_exps()* to check whether the EXPS is within the kinds. When we are doing assigning actions or calling *read()*, we call the check left value function.

In the intermediate representation part, when the expression must be a left value, I also check that the corresponding register should an address, otherwise I would report lvalue error.

3.3.10 The condition of if statement should be an expression with int type

We first check whether the EXPS would be a struct, then we call the *get_width_semantics()* to get the bytes of the id of the EXPS, to see whether it's an array pointer or an int.

3.3.11 The condition of for should be an expression with int type or ϵ

Same as the upper one.

3.3.12 Only expression with type int can be involved in arithmetic

Same as the upper one.

3.3.13 Some other

I also ensured that:

1. Size of array can't be negative.
2. Members of a struct can't have same names.
3. Parameters of a function can't have same names.

4 Intermediate Representation

This part is implemented in **translate.h**. After semantic checking, we can assume that there are no semantic errors, and translate the parse tree into an intermediate representation(IR). The IR is supposed to be independent of the details of both source language and target language. I refered to the tiger book and choose quadruple (three-address code) as my IR, for further optimization and realization.

My IR is relatively similar to machine language, the major difference is that I assume registers are infinite here, and whenever there is a variable declaration I would assign it a register to represent the address of it in the memory. The advantage of this is that apart from the declaration part, the other parts would only contain operations concerned with register and constants, which brings convenience to later optimization.

The specific design of my IR is listed here.

Table 1: My IR Design

Quadruple	Explanation
label, n	n is an integer
goto, n	n is an integer
func, s	s is a string
call s	s is a string
ret	function return
or,a,b,c	a = b — c
xor,a,b,c	a = b ^ c
and,a,b,c	a = b & c
sll,a,b,c	a = b << c
srl,a,b,c	a = b >> c
add,a,b,c	a = b + c
sub,a,b,c	a = b - c
mul,a,b,c	a = b * c
div,a,b,c	a = b / c
rem,a,b,c	a = b % c
neg,a,b	a = b
lnot,a,b	a = !b
not,a,b	a = ~b
beqz,a,b	if a==0 goto label b
bnez,a,b	if a!=0 goto label b
bgez,a,b	if a>=0 goto label b
bgtz,a,b	if a>0 goto label b
blez,a,b	if a<=0 goto label b
bltz,a,b	if a<0 goto label b
li,a,b	a=b, b is constant
lw,a,b,c	a= *(b+c), c is constant
sw,a,b,c	*(b+c) = a, c is constant
move,a,b	a=b

4.1 Definitions

```
struct Address{
    RegType type;//type of the argument
    string name; //for the "label" and "call" case
    int value; //register
    int real; //the real allocated register in MIPS
    int needload;
    //indicate whether load op is necessary
    generation
    int needclear;
    //indicate that whether this op should be cleared
    MIPS generation
};

struct Quadruple{
    string op; // name of operation
    int active; //whether it's active, which is used in optimization
    int flag; //to indicate that whether its arguments should be revised
    Address arguments[3];//the three arguments of the quadruple
};
```

4.2 Whole picture of My Design

Registers can store addresses, but also values. So it's important that we distinguish the two. We can check quickly find out whether a register stores value or address by **RegisterState**, when the two types convert to each other, we have to modify it.

You can see there is a type in the Address struct, which is an attribute of the register. It includes:

- ADDRESS_LABEL, which means it's not a register but a label number
- ADDRESS_CONSTANT, which means it's not a register but a constant number
- ADDRESS_TEMP, a register
- ADDRESS_NAME, which means it's name, so we have to refer the name member

I have three vectors here, the type of which is quadruple, IR,GIR and MIR.

- IR corresponds to the main part of my intermediate representation.
- GIR corresponds to the instructions of declarations of global variables, since global variables declarations could be in different places.
- MIR corresponds to assignment actions and initialiation of global variables.

I also have some vectors with type int. **RegisterOffset** contains the info of offsets of the register. **LabelBreak** and **LabelCont** contain the label numbers of break and continue actions. **vs_reg** is used together with the vector <string> *vs_id*, which associates identifiers with the registers that store the addresses. **Sequential scanning is OK** because the semantic analysis has been done, and the case that variable used is in another namespace has already been excluded.

I also have some int variables:*current_sp,local_register_count,function_begin_sp*, the meanings of them are consistent with their name.

I defined many quadruple generation functions, which assigns values to the member of the struct according to the semantics. You can see them in **translate.h**.

4.3 Translation of EXPS

The function *translate_exps()* has a parameter "reg", which means the value of the expression would be stored in reg.

4.3.1 Ordinary Arithmetic Expressions

The function *new_register()* is called first, and we call *translate_exps()* recursively, and we use *catch_calue_self()* to catch the value and stores into itself, and then we do the arithmetic action and store the value in *reg*.

4.3.2 Logical Expressions

The result of logical expressions is either 0 or 1. So I have the following IR:
li reg,1
jump judgement here
li reg,0
label l
Depending on the type, I use beqz, bnez, bgez, bgtz, blez and bltz, to judge whether to jump to label l. If a jump happens, then the value of reg is 1, otherwise it's 0.

4.3.3 Unary Expressions

It's a bit special when we meet "++" or "–". We have to use a temporary register to hold the original address of reg, and alter the value of reg and store back to its address.

4.3.4 Assignment Expressions

We defined a auxiliary function named *translate_assignment()*, which catches value and stores value. Operations like "+=" are nothing difficult, just combine arithmetic expressions and "=" expression together.

4.3.5 Integers and Arrays

We judge whether it's an array by its width. If it is, we will call the *translate_arrs()*, which will calculate the offset, by adding the offset we get to the location of array element, thus we can attain its value.

4.3.6 Struct Expressions

It's very similar to the array case. I have an auxiliary function named *cal_offset()* to location the struct definition and get the offset, then we can get the localtion of the member, thus we can attain its value.

4.4 Function Definitions and Function Calls

4.4.1 Function Definitions

We have to set *function_sp* to *current_sp* first and set *local_register_count* to 0.
We store the value of the parameters on the stack, sequentially.
Then we translate the stmtblock.
Since the number of local registers is unknown before the translation of the stmtblock, for those instructions associated with the stack pointer, they have to be modified afterwards.
Finally, we generate "ret" and restore the *current_sp*. The return value is stored immediately after the local variables in the stack.

4.4.2 Function Calls

If it's not in the main function, we have to store $ra in the stack and restore it afterwards. The stack pointer should also be restored after the calling procedure.

4.5 Declarations

Get the width of the variables and reserve spaces on the stack(for local variables) or in the global scope.

4.6 Statements

4.6.1 If Statement

The logic here is like what we have been taught:
if (a) then b; else c -> (a->L1) — b — goto L2 — label: L1 — c — label: L2
If the judgement of "if" is true then we will execute b and we met "goto L2" then we just skip the else part. Otherwise, we will directly jump to L1, skipping the expression following "if".

4.6.2 For Statement

The logic here is like what we have been taught:
for(a;b;c)d; -> a — (b->L3) — label L1 — d — label L2 — c — (!b->L1) — label L3
We do a and then judge b, if false we go to L3 directly. Else we enter the loop, execute d and judge b, if true we go to L1 to excute the loop again, else go to L3. For *continue* and *break* statements, we have two vectors to store the label number they should go to, in case of loops embedded in loops. In this single loop, *break* will goto L3 while *continue* will goto L2.

4.6.3 Read and Write Statement

Read is to assign the value of $v0 into the corresponding address of a variable, while *Write* is to get the value of an variable and move it to $a0. I use -2 and -3 and the register number of these two to Distinguish them from other registers.

I have already defined *scanf_one* and *printf_one* corresponding to *read()* and *write()* and the codes will be directly copied into the MIPS code. It's worth mentioning that if these two statements are called in a function that is not main, it's necessary to store and restore $ra, otherwise your function would be unable to get back to main().

4.7 Overloaded Functions

The thing I did is the same as a real compiler does: rename the function. In **semantics.h** I modify the overloaded function' name according to the number of parameters it has when it's defined, and when the overloaded function is called, we will check the number of arguments and make the calling functions' name consistent with the previously modified name. Then, inside my toy compiler, the overloaded functions are just totally different from each other, and they can be safely translated like normal functions.

5 Optimization

This part is implemented in **optimize.h**.

I did some common subexpression elimination and dead code elimination while ensuring correctness.

I have two arrays *prev_pos* and *next_pos* along with a function *update_pos()* to update the arguments of my quadruples.

5.1 Optimize_1()

It dealt with cases like:
move t2,t1
move t3,t2
Clearly we can eliminate redundancy here. But we have to avoid removing quadruples that involves registers for later use. In this example, if t2 is reassigned later, then we can reduce our IR.

5.2 Optimize_2() and Optimize_3()

It dealt with cases like:
li t1,1
move t2,t1
Also we have to guarantee that the value t1 is not for later use.
The code is quite straightforward, you can go and have a look.

6 Machine-code Generation

The register allocation part is done in **optimize.h**, and the interpretation part is done in **interprete.h**.

6.1 Instruction Selection

Most of the instruction selections are done in the IR part. However, there are many things we can do in this part.

- Everywhere that needs 0, we can use $0 instead.
- Check labels and branches, if the branch destination is immediately following it, delete the branch.
- For all arithmetic expressions that involves constants, interpret them to intermediate instructions
- Check successive lws and sws. If the instruction in front of sw is lw and the rt and address are the same, then the sw can be deleted; likely for the lw case.
- Use sll when you times 4.

I tried my best to implement every principle above, but just finished part of them, due to limited time. Implementation could be found in my codes.

6.2 Register Allocation

I used the linear scan algorithm. To be specific, registers from $11 to $25 are allocated, if not enough, $8, $9 and $10 are reserved for use. In fact we can always use lw and sw operations to make things work even with just 3 registers. The algorithm scans all the live ranges in a single pass, and allocates registers in a greedy fashion. I imitated the methods of tiger book while referring to the slides given by the TA.

I calculate the *use* and *def* of each quadruple, and iteratively update the *in* and *out* of each quadruple according to the BFS flow gram, and finally allocate registers for each variable using the linear scan algorithm. The implementation is in **optimize_4()**.

6.3 Interpretation to MIPS

Since my IR is close to machine code and the register allocation part has been done, interpretation from IR to MIPS is quite straightforward. You can see my implementation in **interprete.h**.

7 Conclusion

This project is quite challenging and demanding, but finally I finished it, though not perfectly. I think my coding ability and engineering skills surely improved. It's really important to think thoroughly before writing long codes, or you will get in a mess.

Thanks to the help of TA and my classmates.

8 References

Shanghai Jiao Tong Univ. Project 1: Lexical and Syntax Analyzer[EB/OL]. http://www.cs.sjtu.edu.cn/~fwu/teaching/cs308.html

Shanghai Jiao Tong Univ. Introduction to Lex[EB/OL]. http://www.cs.sjtu.edu.cn/~fwu/teaching/cs308.html

Shanghai Jiao Tong Univ. Introduction to Yacc[EB/OL]. http://www.cs.sjtu.edu.cn/~fwu/teaching/cs308.html

Jeff Lee. ANSI C Yacc grammar[EB/OL].http://www.lysator.liu.se/c/ANSI-C-grammar-y.html#translation-unit

Jeff Lee. ANSI C grammar, Lex specification[EB/OL].http://www.lysator.liu.se/c/ANSI-C-grammar-l.html

Unknown. Writing an Interpreter with Lex, Yacc, and Memphis[EB/OL].memphis.compilertools.net/interpreter.html

Lam M, Sethi R, Ullman J D, et al. Compilers: Principles, Techniques, and Tools[J]. 2006.

Andrew W. Appel Modern Compiler Implementation in C[J]. 2006.