CS215 compiler project

Small-C Compiler

Chao Gao

5142029014

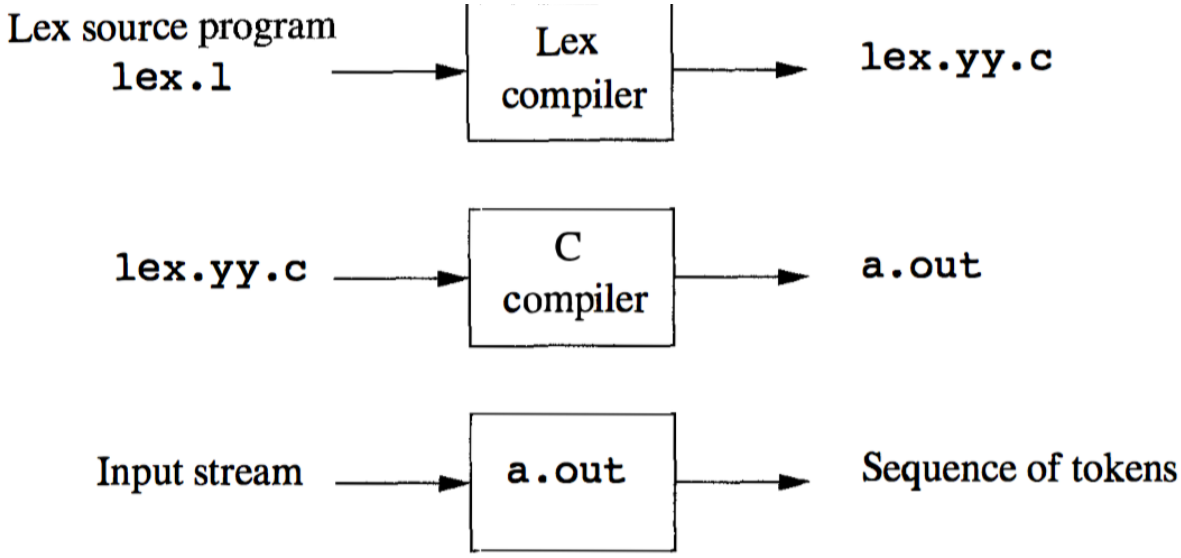
dimon-gao@sjtu.edu.cn

2016.1.17

1. Introduction

For this project, I am going to implement a simplified compiler, for a given programming language, namely small C, which is a simplified C-liked language containing only the core part of C programming language. Finally I get the compiler to translate small C codes to MIPS assembly codes.
The project was done on the Linux environment called Ubuntu 14.04.
The implementation procedure is divided into 5 parts.
- lexical analyzer
- syntax analyzer for syntax tree generation
- semantic analyzer and intermediate code generation
- optimizations
- MIPS code generation

2. lexical analyzer



This part is mainly implemented in **small.l**.
For this part, I will use flex as the tool of lexical analyzer and I can work out the lexical analysis under the framework with only specifying the regular expression and the according action.

line-number

Line number can be used in the error checking, for anyone wants to use a compiler, it is necessary to tell where the bug occurs.

```
[\n] {linenum = linenum+1;}
```

number

I will only handle positive numbers here, as for minus sign, I will handle it in semantic checking. Besides, a number starts with 0X,0x is valid, the English characters will only appear in the numbers start with 0X, 0x.

```
([0-9]*|0[xX][0-9a-fA-F]+)  { yylval.string = strdup(yytext); return INT; }
```

read and write

Two special functions added to smallC.

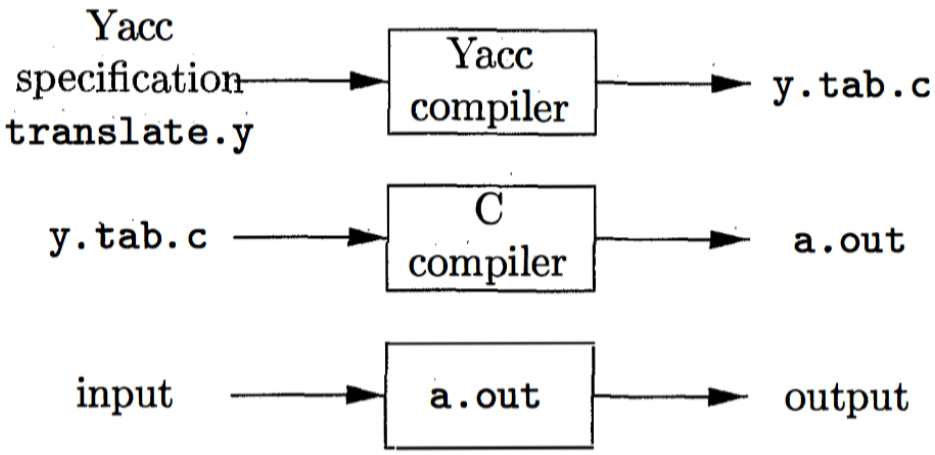
```
"read" {yylval.string = strdup(yytext); return READ;}
"write" {yylval.string = strdup(yytext); return WRITE;}
```

normal symbols

Since the logic are almost the same for all symbols, here just show some examples.

```
"&" {yylval.string = strdup(yytext); return '&';}
"!" {yylval.string = strdup(yytext); return '!';}
"~" {yylval.string = strdup(yytext); return '~';}
"-" {yylval.string = strdup(yytext); return '-';}
```

3. syntax analyzer



This part is mainly implemented in **small.y, node.h, tree.h**.
For this part, the result I get from the lexical analyzer will be the input of the syntax analyzer. Then I can construct the abstract syntax tree.
Define a struct type **TreeNode** in node.h. The explanation of each member is given in the following codes.

```
typedef struct TreeNode {
    TreeNodeType type; //type of the treenodes
    int line_num;      //the linenumber
    char* data;
    int size, capacity; //num of children, maximum num of children
    struct TreeNode** children;
} TreeNode;
```

The abstract syntax tree indicates the information needed for the next steps. The semantic analysis part and code generation part will traverse from the root of the tree.

In this part, we can handle some grammar problems.

precedence of IF statement
The grammar given above may induce one or two reduce-reduce/shift-reduce conflicts, you need to assign precedence of some expressions manually to

eliminate these conflicts. For example, “IF LP EXP RP STMT” should have lower precedence than “IF LP EXP RP STMT ELSE STMT”.

```
%nonassoc  IFX
%nonassoc ELSE
...
| IF LP EXPS RP STMT %prec IFX { $$ = create_node(linenum,_STMT, "if stmt", 2, $3,$5); }
| IF LP EXPS RP STMT ELSE STMT %prec ELSE { $$ = create_node(linenum,_STMT, "if stmt", 3,
$3,$5,$7);}
```

Here one thing has to be mentioned, since situation like if(EXP) will not be acceptable if EXP is empty, so change EXP to EXPS(not empty), avoiding the empty situation.

the dimension of array no more than 2

```
ARRS: LB EXP RB ARRS { $$ = create_node(linenum,_ARRS, "arrs []", 2, $2,$4); }
| {$$ = create_node(linenum,_NULL, "null", 0);}
```

read and write

```
| READ LP EXPS RP SEMI{$$ = create_node(linenum,_STMT,"read stmt",1, $3);}
| WRITE LP EXPS RP SEMI{$$ = create_node(linenum,_STMT,"write stmt",1, $3);}
```

error message

show the line number of error and its error text.

```
void yyerror(char *s)
{
    fflush(stdout);
    fprintf(stderr,"\n[Line %d]: %s %s\n",linenum,s,yytext);
}
```

4. semantic analyzer & intermediate representation

This part is mainly implemented in **semantics.h, intermediate.h.**
In this part, I am going to do semantic analysis and syntactic checking to examine potential semantic errors after generating a parse tree.

symbol table

symbol table is an important data structure to store the information of the source program.

```
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 gloabl scope
} env[MAXSIZE][MAXSIZE];
```

semantic checking

declaration

Variables and functions should be declared before usage, and they should not be re-declared. Reversed words can not be used as identifiers.
For variables, I have the function to search for the name, if it is not be found in current scope, it will go to the parent scope until the global scope.

Also, we can find out if this name has already be declared.

```
bool semantics_check_id(char* s,int num) {
    if (isReserved(s)) {
        report_err("Reserved words can not be used as identifiers", s,num);
        return false;
    }
    if (env[level][cnt[level]].table.find(s)!=env[level][cnt[level]].table.end()){
//already exist
                report_err("redeclearation of",s,num);
                return false;
    }
    else return true;
}
```

For functions, refer to **func_table** to see whether the function name exists.
Also, check func_table to see if it is re-declared.

function
Program must a function int main() to be the entrance.
define a flag variable **bool_main** to indicate if we meet the definition of main function.

The number and type of variables passed should match the definition of the function.
Define a table to record the number of arguments of a function, and I can check if it matches the definition.

operator
Use [] operator to a non-array variable is not allowed.
Use the size of the variable to indicate if it is an array variable.

The . operation to a non-array variable is not allowed.

Use find_struct_id function to check if this variable has been declared as struct. Then refer to struct_id_table to check the variable after dot is a member of

```
if (find_struct_id(p->children[0]->data,NULL)) {
                report_err(p->children[0]->data,"is a struct",p->children[0]->line_num);
        }
```

struct.

break and continue

Break and continue can only be used in a for-loop.
Define a flag variable to indicate whether it is in a loop.

right value

Right-value can not be assigned by any value or expression.
Use check_left_value_exps() function to check whether the expression can be left value. The code is as follows.

```
void check_left_value_exps(TreeNode *p) {
    if (!strcmp(p->data,"=")) {
        check_left_value_exps(p->children[0]);
        semantics_check_exps(p->children[1]);
    }
    else if (!strcmp(p->data,"exps arr")||!strcmp(p->data,"exps struct")||!strcmp(p-
>data,"exps ()")) {
        return;
    }
    else {report_err("lvalue required as left operand assignment",NULL,p->line_num);}
}
```

operator type checking

The condition of if statement should be an expression with int type.
The condition of for should be an expression with int type or empty.
Only expression with type int can be involved in arithmetic.
First check if the expression is a struct, then we find out whether it is an array pointer or an int.

others

The size of an array can not be negative.

There are no members with the same name in a struct.
There are no parameters with the same name in a function.

intermediate code generation

This is the most complex part of the whole project in my opinion. After semantic analysis, we can assume that there are no semantic errors, so I can translate the parse tree into an intermediate representation(IR), which acts as a bridge between high-level language and machine language. Here I choose quadruple(three-address code) as my IR.

```
struct Quadruple{
    string op; // name of operation
    int active; //whether it's active
    int flag; //whether the arguments should be revised
    Address arguments[3];
};
```

The following is my three-address code format.

op	arg1	arg2	destination	explanation
move	a		b	a=b
label			n	integer n
goto			n	jump
func	s			function
call	s			call function
add	b	c	a	a=b+c
sub	b	c	a	a=b-c
mul	b	c	a	a=b*c

op	arg1	arg2	destination	explanation
div	b	c	a	a=b/c
rem	b	c	a	a=b%c
lnot	b		a	a=!b
not	b		a	a=~b
or	b	c	a	a=b | c
xor	b	c	a	a=b^c
and	b	c	a	a=b&c
srl	b	c	a	a=b>>c
all	b	c	a	a=b<<c
bgtz	a		b	if a>0 goto b
bgez	a		b	if a>=0 goto b
beqz	a		b	if a==0 goto b
bnez	a		b	if a!=0 goto b
blez	a		b	if a <=0 goto b
bltz	a		b	if a<0 goto b
li	b		a	a=b
lw	b	c	a	a=*(b+c)
sw	b	c	a	*(b+c)=a

function call

For function calls, if it is 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.

read and write statement

These two I handle them additionally. 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.

Also I define two codes corresponding to read and write, and the code will be directly copied into the MIPS code.

5. optimization

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

Just some specific cases

```
move t1, t2
move t3, t1
```

```
li t1, 1
move t2, t1
```

we can have some dead code eliminations if t1 is not for late use or t1 is resigned value later.

6. MIPS code generation

This part is mainly implemented in **optimize.h**, **codegen.h**.

instruction selection
In this part, we assume that there are infinite registers. What we need to do is to choose suitable instructions and translate them. Actually, this part can also regard as the optimization.

- if the brach address is the following instruction, delete the branch
- use $0 instead of 0
- for expressions that have constants, translate them into intermediate instructions.

register allocation
In this part, I use the brute force algorithm, registers from $11 to $ 25 are allocated, if not enough, $8, $9, $10 are reversed for use. I just scan all the live registers and allocate for each variable.

Input and output

For two specific function read() and write(), I have mentioned them in the previous parts, define two functions and directly print them to MIPS code.

```
printf("__printf_one:\nli $v0, 1\nsyscall\njr $ra\n__scanf_one:\nli $v0,5\nsyscall\njr $ra\n\n");
```

7. Conclusion

This project is really a hot potato. Every time you should think clearly before implementation, and divide the big projects into small parts. When you write a large scale of codes, it is likely that you get lots of bugs, the best way is to debug step by step.

Thanks Prof.Jiang for the impressive teaching about compiler principle and TAs for the guidance of the compiler project.