Project Report for Compiler Principle

Yuxiao Zhang
5130309009
Computer Science
Shanghai Jiao Tong University
Email: 1377478547@qq.com

***Abstract*— This project implements a simple complier for small C language. We do several steps to translate small C code to LLVM instruction with the help of FLEX and YACC. In project 1 we have implement lexical analysis and syntax analysis, so this report will mainly introduce how to generate intermediate code using the syntax trees we have built before.**

I. Introduction

In this project we need to use what we have learned to build a simple complier for Small C language. There are some basic steps: lexical analysis, syntax analysis and intermediate code generation. We don't need to generate machine code cause our goal is to print out LLVM instruction.

The first and second part has been finished in project 1. We do the lexical analysis using FLEX and send the token to YACC to finish the parsing task. The main idea is to create a node for every terminals and non-terminals and push it into stack when parsing the grammar.

So we can build a syntax tree for our result. Next I will introduce how to generation LLVM instruction by translating this syntax tree.

II. Baisc Idea

There are several choice for me to do the translation. The simplest way is to add sematic action while parsing. In this way we even don't need to use the syntax tree! However, since YACC uses buttom-up parsing, it's not an easy task to directly generate code in the way of button-up. So I choose the most direct way: top down translation.

So what I do is to start from root node and translate in preorder way. Every time I choose the right functions according to children's situation of the node. That's the basic idea.

III. Implementation

A. Function

With this idea, I write several functions for some important parsing grammar, the function name basically means the corresponding grammars:

```
void syntax_program(struct node* root);
void syntax_extdefs(struct node* t);
void syntax_extdef(struct node* t);
void syntax_extvars(struct node* t);
void syntax_declareout(struct node* t);
void syntax_function(struct node* t);
void syntax_paras(struct node* t);
void syntax_para(struct node* t);
void syntax_stmtblock(struct node* t);
void syntax_defs(struct node* t);
void syntax_def(struct node* t);
void syntax_decs(struct node* t);
void syntax_declarein(struct node* t);
void syntax_stmts(struct node* t);
void syntax_stmt(struct node* t);
char* syntax_exp(struct node* t);
void syntax_argsout(struct node* t);
void syntax_argsin(struct node* t);
void syntax_argsfunc(struct node* t);
void syntax_extdefstruct(struct node* t);
void syntax_decstructid(struct node* t);
void syntax_extdefstructid(struct node* t);
void syntax_extvarsstructid(struct node* t);
void syntax_extdefstructop(struct node* t);
void syntax_defsstructop(struct node* t);
void syntax_defstructop(struct node* t);
```

After implementing these function correctly, we can just finish our task by calling syntax_program(struct node* root) in the main function.

B. Detail logic

There are mainly 2 parts in a function:

- Judge the condition of node and decide which function to go next. For example, we need to find out whether a variable is a single INT type or array type. If it is a INT type we need to go to ID parts and if it is an array type we need to go to ARRS parts.
- Doing translation, including printing codes, selecting register, manipulating symbol table and changing the status of some flags.

When we generate code, we need to deal with ID carefully, so I create a symbol table. Each element in symbol table is a structure called symbolnode:

```
struct symbolnode
{
   char* symbolstr;
   char type;
```

```
        char* arrsize;
        char* structstr;
        int block;
        int structnum;
        int isvalid;
         };
```

Which include some information such as name, type, size of array (if the ID is an array), structure name (if the ID is a structure), number of parameter in structure (if the ID is a structure), number of block level and whether it is valid in some cases.

C. Special case

- Hex and Oct number: I deal with it in the simplest way – transform those number into Dec when doing lexical analysis:
-

```
({digit}*)|(0x({hexdigit}*)) {flag=1;strcpy(terval,"");
if(yytext[0]=='0' && yytext[1]=='x'){sprintf(terval,"%d",str-
   tol(yytext,NULL,16));}
else if(yytext[0]=='0')
   {sprintf(terval,"%d",strtol(yytext,NULL,8));}
else {sprintf(terval,"%d",strtol(yytext,NULL,10));}
strcat(terval," (INT)");/*fflush(stdout);*//*printf("int: %s\n",
   yytext);*/return(INT);
}
```

- UMINUS and MINUS: Actually, I separate them in lexical analysis. The basic logic is to set a flag. We can see that if an '-' is BINARYOP, it can only appears exactly after some symbols like ID,INT, ')', ']'. So when we find such tokens we set the flag to 1, when we find other tokens we set the flag to 0. If we find '-', we can check the flag, if flag is 1 we return UMINUS, else we return MINUS.
- Structure part: We need to make some changes when we declare struct, especially in symbol table.
- Multilevel blocks: That's why I record the number of block levels. So we can detect such error:
  Int main(){
  {int d=1;}
  d=0;
  return 0;
  }
- Array assignment: for some special cases such as a[4]={2,3,4,5}, I just translate into
  a[0]=2;
  a[1]=3;
  a[2]=4;
  a[3]=5;

IV. Error Detection

After testing all the testcases, I doing some basic error detection. The first part is lexical error, the second part is syntax error and the third part is semantic error.

A. Lexical error

This type of error is easy to detect, since we can find this kind of error when doing lexical analysis. I set a counter in FLEX file to record the line number. Every time when we meet '\n', we increase the counter. If we find some string cannot be recognized as a token, we can return an error and print the line number.

B. Syntax error

This part is also simple, I detect several kinds of syntax error.

The first kind of error is we forget some symbol such as semicolon or comma. We can find these error by checking the status exactly before SEMI or COMMA in the parser, if the parser fail to parse in these places, we know that the parser fail to find a semicolon or a comma.

The second kind of error is that the bracket is not matched. This is hard because we can hardly know where a left bracket or a right bracket supposed to be. So I detected these errors in an not very accurate way. I just count the number of left and right bracket, if the number is not match, I return this kind of error.

C. Semantic error

When I allocate a variable, first I will check the symbol table to see if this variable has been allocated. If so I throw a **multiple definition** error. Else I will put it into the symbol table, with some information such as name, global or local and the block level. When the program use an variable, I check the symbol table to find whether this variable has been allocated, if not I will throw an **undefined-variable** error.

I also do the elimination of variables. When a variable leave its scope, I set the valid bit to 0, which means the variable is not valid anymore.

It's a little difficult to get the line number of these kinds of error, so I just add some information into a node, so I can save line number for all the ID nodes. In this way we can simply return the line number of the problematic ID node.

V. Code Optimization

In order to decrease the amount of code, I also do some simple optimization. That is to do some calculation when generating intermediate code. Take the following code as an example:

a=2+3/5-4*6;

If we don't do the optimization, we need to generate many register to store many middle- time results, such as 3/5, 4*6 and so on. And that also needs many instructions to do these things. So I just calculate the result and directly store the result into the variable. This is not hard since we can check the operands of one expression. If all the operands is INT rather than address, we can do the calculation and return an INT, here is an example for ADD:

```
 char* op1 = (char*)malloc(sizeof(char)*60);
op1 = syntax_exp(t->child[0]);
char* op2 = (char*)malloc(sizeof(char)*60);
op2 = syntax_exp(t->child[2]);
```

```
		if(op1[0]!='%' && op1[0]!='@' && op2[0]!='%' &&
op2[0]!='@')
		{
		  char *num=(char *)malloc(sizeof(char)*10);
		  sprintf(num, "%d",
strtol(op1,NULL,10)+strtol(op2,NULL,10));
		  return num;
		}
		else
		{
	char *num=(char *)malloc(sizeof(char)*10);
	sprintf(num, "%d", regnum++);
	char* tmpReg = (char*)malloc(sizeof(char)*60);
	strcpy(tmpReg,"%r");
	strcat(tmpReg,num);

	printf("  %s = add nsw i32 %s, %s\n",tmpReg,op1,op2);
	return tmpReg;
		}
```

ACKNOWLEDGMENT

Thanks Dr. Jiang for giving us such a project to learn complier and to get familiar with FLEX, YACC and LLVM. Thanks our TA for giving us guidance.