Project Report for CS215

Implementing a Lexical-Syntax Analyzer & Code Generator

Renxuan Wang
5130309363
Computer Science
Shanghai Jiao Tong University
Email: 290484989@qq.com

***Abstract*—This report is for CS215 project 1&2, which is about how I implemented a compiler which creates a parse tree for a small C program (project 1), and further translate into a LLVM assembly program (project 2), which can be run on LLVM. The project consists of an 80-line Flex and a 2000-line Yacc program. This report is written in the IEEE format. Lexical analysis, syntax analysis, semantic analysis, code generation and optimizations will be covered gradually.**

***Index Terms*—compiler, LLVM, code generation.**

I. Lexical Analysis

This part is done in Project 1. The lexical analyzer reads in the Small-C source code, and recognize tokens according to regular definitions. There are two details should be mentioned:

- I use a variable called *isBinary* to indicate whether a "-" symbol is a binary operator or a unary operator. When we encounter an integer, a ")", a "]" or an ID, set *isBinary* to 1 since a "-" following these lexemes must be a binary operator. When we encounter other lexemes, we set it to 0. And when we encounter a "-", we return MINUS if *isBinary=0* and return SUB if is*Binary=1*.
- We need a string to pass IDs, the specific types (in our project, only *int*) and the value of integers from Lex to Yacc. I used a string variable called *passing* to do such works.

II. Syntax Analysis

This part is also done in Project 1. The syntax analyzer receives tokens from Lex and build a parse tree. Every node on the parse tree has a label and a pointer pointing to its children. The main method is appending syntax actions to the grammars. For example:

EXTDEF : SPEC EXTVARS SEMI
{root=buildnode("EXTDEF"); insertNonTerm(root,stack[top-1]); insertNonTerm(root,stack[top]); insertTerm(root,"SEMI"); stack[--top]=root;}

where *buildnode(s)* creates a node labeled *s*, *insertNonTerm (node$_1$, node$_2$)* set *node$_2$* a child of *node$_1$*, *insertTerm (node, s)* creates a node labeled *s* as a child of *node*, and *stack* is a stack to support bottom-up parsing.

In this case, EXTDEF derives two non-terminals and a terminal, so we need to build a new node, fetch two nodes from the stack (they are put into the stack earlier, when parsing SPEC and EXTVARS) and make them the children of EXTDEF. The terminal SEMI should also be the child of it. After that, we should remove SPEC and EXTVARS out of the stack, and put EXTDEF into the stack for latter parsing.

The rest part of syntax analysis is almost the same, except that we need to pay some attention to ID, integer and type. As mentioned in Sec. I, we should use *passing* to be the label of node ID, integer and type. And for ε nodes, I used nodes labeled "NULL".

What's more, I changed the grammar a little. For each ID, I added one extra non-terminal FID (fake ID). In specific, I substituted all ID in the grammar with FID, and FID points to a node ID. Because when I didn't do this, I encountered a problem when parsing *struct struct_name variable_name*. In this case, only *variable_name* can be passed from Lex. This is very strange, and I haven't figured out why until now. But after I made such a change, this problem disappeared.

To meet the requirement of project 2, the production rule for READ and WRITE should be added. They are added as below:

EXP -> …
| WR
WR -> WRITE LP EXP RP
| READ LP EXP RP

To print the parse tree, I use preorder traversal, and use "-" to mark the level of each node. A node of level n will have n "-" preceding it.

III. Semantic Analysis

From now on, the contents are all about Project 2.

A. Symbol

Each ID has a symbol in symbol table. When we want to declare a new ID, we create a symbol. When we want to use an ID, we need to check whether it's already in the symbol table. I define the struct symbol as follows:

```
struct symbol
{
    char* word;
    char type;
    int level;
```

```
    char valid;
    char* arrSize;
    char* structName;
    int structMem;
};
```

word is the name of the ID. *level* is the level where an ID is declared. For global variables (structures, arrays, functions), the level is 0. For variables declared in {{…}}, its level is 2, and so on. *type* is for 3 core conceptions, label, address and register. Label is for jump. Address, as divide to global(@) and local(%), is used for load/store. *valid* indicates whether the symbol is still valid. The rest 3 attributes are for array and structure.

B. Symbol Table

The way I managed the symbol table may seems a little strange. My symbol table is a 53*15 two dimensional array, and is indexed by the first letter of an ID. That is, a->symbolTable[0][*], ... , z->symbolTable[25][*], A->symbolTable[26][*], ... , Z->symbolTable[51][*], _->symbolTable[52][*] and it ends. This is a simple way, yet competent for this project.

C. Semantic Analysis

Variables (structures, arrays, functions) should be declared before usage and should not be re-declared. And according to my test of GCC compiler, a name cannot be shared by a variable, a structure, an array and a function.

The above requirements can be achieved by symbol table. When we want to declare a new variable, we use its first letter to get the right entry, and check whether there is a valid symbol of the same level having the same name. If so, we should report the error and exit, otherwise we can create a new symbol safely. When we want to use a variable, we need to check whether it's already in the symbol table. We get the entry, and try to find a valid symbol whose level is **less or equal to** the current level. You may ask what if the following case occurs:

```
{int a;{int a; a=1;}}
```

well, as we can observe, the inner variable must be declared later. So we only need to check in the inverted order, and the first valid symbol we find is what we want. I should underscore that when we leave a statement block, all the variables declared in it should be set to invalid.

To summarize, I accomplished requirement 1, 2, 6 and 7 of semantic analysis.

IV. Code Generation

Instead of bottom-up parsing, I first build a parse tree, then traverse the tree to generate LLVM assembly program. Although it requires some extra work compared to bottom-up, it is more explicit and closer to the way we human think. Correspondingly, I defined a series of functions. Roughly each of them correspond to a non-terminal. The way they work is similar to the recursive descent parsing in our textbook. For example:

```
void Extdefs(struct node* t) //external defininitions
{
    if (t->child[1]) //EXTDEF EXTDEFS
    {
        Extdef(t->child[0]);
        Extdefs(t->child[1]);
     }
}
```

it's quite straightforward. The way we decide which production rule to choose is shown in this example: we use its children nodes to help. In this case, it's whether the node have two children, and in some other cases, we will check the label of children nodes. After that, we apply corresponding functions on its children nodes. Of course, there are a lot of things worth discussion.

A. Global vs. Local

The LLVM IR code for global declaration and local declaration are different. However, the grammar doesn't show the difference. So I split DEC into two functions, one is *DecExt()* and the other is *DecIn()*.

B. Cases Mergence

In some cases, although a non-terminal have different production rules, we have to attempt a judgment earlier. One typical example is DEC. There are four cases for DEC: (1) a; (2) a=1; (3) a[n]; (4) a[n] = {…}. The LLVM IR codes for them are different. So if we merge the 4 cases into *Dec()* function instead of using another function *Var()*, it will be much more convenient.

C. Parameters

In *int function(int x)*, x is defined in PARAS function. However, it needs to be loaded when entering the STMTBLOCK. To deal with this case, I used *paraFlag* to denote whether we have parameters, *paraNum* to denote how many parameters, and *paraArrry* to store the value of parameters.

D. STMTBLOCK

How do we know when to print '{' and '}' when encounter STMTBLOCK? At first, I print them as long as encountering STMTBLOCK, but soon find it illegal. Then I add a variable called *entryDepth*, and increase it when it goes deeper into STMTBLOCK, decrease it when gets out. Only when it is zero print '{' and '}'.

E. Decimal Conversion

We have 0xD7 and -0327 in our test cases, which means we should do decimal conversion for octal and hex integers. I did this in Lex, with the help of *strtol()* function.

F. Register vs. Immediate

Whether *Exp()* returns a register or an immediate number should be handled carefully. If it's an immediate number, we need to first allocate an address to store it, then load it to a register. If it's a register, we can just use it.

There are so many other noticeable details, but I just omit the rest.

V. Optimizations

A. Error detection

My program have the following error detections: syntax errors, re-declaration, undeclared reference. For syntax errors, I insert the detection into the process of parsing. For example:
EXTVARS : DEC {errortype=9;} COMMA {errortype=10;} EXTVARS {errortype=0;}

I defined 10 syntax errors in total, and wrote the coresponding output information in *yyerror()*. The output information is a missing symbol, unrecognized word or imcomplete expression with their linenumber.

For re-declaration and undeclared reference, I put the error detection in the semantic functions. The process is already illustrated in Sec. III. And if an error occurs, I will output "Error." to the output file and the specific error type to the command line as demanded.

B. Register Allocation

We have Ershov algorithm in our textbook, but it's too hard for me to implement. We have infinite registers, so we don't even need to do any optimazation. However, I reset the register number to 0 everytime the program goes into FUNC.

VI. Test Results

To test the program, the command should be:
cd (directory)
make
./scc input_file output_file
lli output_file

I have tested with all the test cases, and the results are all correct. But those test cases are all right programs. So I came up with some test cases myself which are wrong programs to show my error detection, and list the outputs in Table I. The programs are in the appendix.

TABLE I. My Own Test Cases

Test Case #	Output
1	Error: a } may be missing at line 1
2	Error: Multiple declaration of "a"
3	Error: "b" is not a defined variable!
4	Error: "b" is not a defined variable!
5	Error: Multiple declaration of "a"

Acknowledgment

Thanks our teacher Dr. Jiang for giving us such a project to learn more about compiler principles. Thanks our T.A. for providing guide.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques, & Tools, 2rd edition.

Appendix

Test case 1:
```
int main(){
```

Test case 2:
```
int a(){return 0;}
int a=7;

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

Test case 3:
```
int a;
int main()
{
    b=1;
    return 0;
}
```

Test case 4:
```
int main()
{
   { int b=1;}
    b=0;
    return 0;
}
```

Test case 5:
```
int main()
{
    int a[2]={1,2};
    int a=1;
}
```