Project2 Report

Shen Jiyuan

5130309194

1 Introduction

In this project, we will implement a code generator to translate intermediate representation, which is produced by the syntax analyzer implemented in project I, into LLVM instructions. The code generator should return a LLVM assembly program, which can be run on LLVM. After finishing this project, we will get a compiler that can translate Small-C source programs to LLVM assembly programs.

Overview: The following statements will be exactly the same as the implementation process on the codes for my project, in other words, we will see a procedure including a simple and direct translation from intermediate representation to LLVM assembly format first and semantic analysis.

2 Project1 Review

Lex and Yacc are used to intermediate parse tree in project I. 'Node.h' defines one structure for all nodes named treeNode, which includes three attributes (label, code, *child). Both the l-file and y-file files will include this head file and use the treeNode structure for yylval to deliver terminals. After the lexeme analyzing in lex, the terminals are delivered to yacc to create the parse tree through defined grammar rules. And in this project, we will put all definitions into yacc file. Hitherto, what we have got is a parse tree rooted in *root.

3 Project2 Preparation

Environment: Ubuntu 14.04

Lex Installation: $sudo apt-get install flex

Yacc Installation: $sudo apt-get install bison

LLVM Installation: $sudo apt-get install clang $sudo apt-get install llvm

4 Structure Sketch

Translation from intermediate representation to assembly language LLVM requires us to do a traversal on the intermediate parse tree by pre-ordering. And through each grammar, corresponding translation actions should be applied and implemented. In order for clearer descriptions, I divide the implementation into various fragments.

Tips: In order to translate the tree to a LLVM language format, we should first have a look at the object codes. All testcases are in '.sc' files, we could copy the tests into c-files, and use ' $clang -emit-llvm filename.c -S -o outputname.ll ' to generate the LLVM file. And such files can be used as references for translation.

More: There are three crucial concepts: lable, address and register. Lable is for jump operation; Address comprises global @ and local %, which is used for load /store; Register mainly deals with intermediate calculation and load/store subject.

4.1 Basic Framework

The traversal for translation on the parse tree rooted in *root requires function definitions for each left non-terminal in grammar sentences, and functions for right non-terminals are called in the function body. Therefore, each non-terminal will have a function definition exactly. For instance, the _PROGRAM function for the PROGR AM non-terminal and so on.

4.2 Variable declaration and initialization

Adding two attributes to node structure in head file: whether it is a left value and the current working space (set 0 if it is global, and a different integer for each function.). At the beginning, the calling and returning processes between functions should be carefully considered. We should choose the right returning types and make sure the program will enter the right functions.

A variable stack for the entire program which contains the name, working space and so on for every variable declared. For array, we have another important attributes

to record – the number of the elements. Notes: 'stack' elements are in structure treeN ode defined in the head file; 'top' records the number of variables in the stack.

Especially specified, the symbol structure is used for symbol table as elements, which is considered as type when using structure.

4.3 Function declaration and calling

When entering a function, first we should update the working space for all the nodes below and push all the variables (containing parameters and all others declared in the callee) into the variable stack. The function _PARAS returns names for all the parameters of the function. Because the values of parameters should not be changed in the callee, we should change the name when the parameters are used. I use another attribute in the variable stack (isPara) to record whether a variable is a parameter. Finally, when returning from a function, the variables with same working space should be popped.

4.4 Expressions

Because the test cases only contain several kinds of expressions, I handle these things in my code and abandon others. By observing the assembly code produced by ./clang, I think the _EXP function should return the name of the register which contains the value of the expression. If one expression contains a operator with assigning effects, the situation becomes more complicated. At this point, we must set the isLeft attribute for subexpressions on the left of this expression. And the returning value of the left subexpression should be a register which stores the address of it. Dealing with the boolean values, we should be careful about the types. Because expressions like “x>y” should return a boolean value. I don't recode the type (“i1”, “i32”) and deal with the type issues by using “icmp ne i32, 0” command which changes a i32 into a i1.

4.5 Input/Output

The read and write functions are respectively scanf() and printf() functions in C

language. As these two functions have been realized by us, we should check the name of functions in cEXP to make sure we will see them as special. Two functions are defined specifically to check the IO operations.

4.6 Branches

There are two types of branches: for and if. Both should be handled in _STMTS function. Remember that the number of labels for a certain series of branches should be the same. So we should record the number when we are in a same branch.

5 Register Allocation

It is entirely such a first implementation here to recount the register every time we encounter with the FUNC rule.

6 Semantic Analysis

In the semantic checking stage, the implementation is completed step by step:

(In contrast to project 1, we do not have an off-the-shelf tool to rely on. Instead, all the codes should be written by ourselves. There are a number of things we can do as follows.)

5.1 Variables and functions should be declared before usage.

We will check if variables and functions have been defined when using them. Therefore, this checking will be coded in the called parts. For variables, we should add codes in 'EXP : ID ARRS', and also notice that the scope could not be checked as the push and pop operations are completed properly. For functions, as the same idea for stack in the translation stage, we add a list for all declared functions, and each element in the *list is in structure funcT which contains two attributes(return type and funcName). We should add codes in 'EXP : ID LP ARGS RP' (Because 'args' also includes EXP rules). The two implementations are both added in the function '_EXP'.

5.2 Variables and functions should not be re-declared.

This checking is different from the last one, the implementations should be added to the definition parts. For variables, codes are added to 'VAR : ID'. For functions, codes are added to 'FUNC : ID LP PARAS RP'.

5.3 Reserved words cannot be used as identifiers.

When defining identifiers, we could apply the simple comparisons between all key words and the current identifier. Reserved words for small-c language are lexeme we have figured in lex.

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

In order to check whether there is a function int main() in the program, we just need to check the 'funcT *list'. Because the elements are defined in structure with attributes return type and function name. And look through all test cases, the function number is little, which means we could just apply the sequential check.

5.5 *break* and *continue* can only be used in a for-loop.

Because *break* and *continue* can not be used in other statements, we can check all rules in the function _STMT except the for-loop.

5.6 Right-value can not be assigned by any value or expression.

There is no need to check the 'VAR ASSIGNOP INIT' rule, and we could only judge the first 'EXP' in the rule 'EXP ASSIGNOP EXP' if its 'isLeft' is true or false.

7 Key Points

7.1 The dfs traversal algorithm implemented for the parse tree needs parameters that seem too large costs for delivering through returned structure. I defined integer values as global standard to justify the algorithm.

7.2 Considering the print problem with the brackets that required for STMT rule. It is simple if we make use of the same principle for the last one: one integer value. When entering a statement the value increases by one, and it decreases by one when outing. Therefore the printing action is done when the value exactly equals to zero.

7.3 Actually when it comes to the load problem, it is obvious that we should load values that are not left value. Yet here I simply push every variable into the stack, and use the variable 'isLeft' to represent whether it should be load in the printed file.

8 Babbling

What I have to concede is that this project is implemented in a limited time for which my fallacious distribution and regulation account. And take this opportunity I am willing to show my appreciation to Dr. Jiang and all TAs.