Compiler Principles
Project 2 - Code Generation

5130309228 YUAN YAO

January 24, 2016

1 Introduction

In this project, we are required to implement a code generator to translate the intermediate representation, which is produced by the syntax analyzer implemented in project 1, into LLVM instructions. My code generator could return a LLVM assembly program, which can be run on LLVM.

2 Environment

Since there is much frustration during my configuration of llvm by using svn and ./configure, I installed it by apt-get and the same with clang. The environment is as follow:

Host OS	Mac OS X
Virtual Machine	VM ware
Guest OS	Ubuntu 14.04.3
IR	llvm&clang-3.5

Table 1: Environment

3 Implementation

The implementation is separated into three parts in logical order. The first part is Machine Code Generation, then Semantic Analysis, finally Optimization.

3.1 Code Generation

Code generation is based on the tree that have been generated in project1. So the easiest way to implement the code generation is to go through the whole tree from the root. All the implementation are packed into one function : **CodeGen** in file "CodeGeneration.c" and is called in the main function in "Small.y".

CodeGen()

This function pack all the code and can be called to generate the llvm IR for one C file. In this function, it initialize parameter queue which would be introduced later, call another function: transalte() and free the space allocated in the end.

```
void codeGen(Node * root,FILE * fout){

        //initialize paraQueue
        ParaQueue * paraQueue ;
          ...

        translate(root,paraQueue);

        //free memory allocated
        free(paraQueue->queue);
        free(paraQueue);
}
```

translate() and translate_exp()

These two function, as their names, are used to translate the syntax tree into llvm IR. They use each node's data attribute to match the rules. If they match each other, then write some IR codes on the output file.
Particularly, translate_exp() has a return value whose type is "char *" and would be very useful for other translations. That's why I extract this translation function.But since "exps" contain "exp" and is used in some rules like

stmts:FOR LP exps SEMI exps SEMI exps RP stmt . This is because the expression in for can be void but it's not valid in if statements. So I made tranlate() function also have a return value.

Parameter Queue

I implement a parameter queue to store the parameters. That's because they will be used after they are read. So we have to provide an array to save them and pop them when they are called.
The reason that I choose queue is that it save much codes and space.It just need to be defined once and then can be used many times because after leaving each function statement block, the queue will be cleaned automatically. And as the CodeGen() part mentioned, this data structure is intantiated in CodeGen() function. Its definition and implementation of enqueue and dequeue are in the file "paraQueue.h"

Symbol Table

Similar to parameters, every variables declared should be store in the symbol table.Each time they are used, they would be required to access their type, name or maybe the structure they belong. The meanings of each attribute are as follow:

Attributes	Meaning
Type	Is the variable global or local
Name	The name of the variable
StrMem	Which attribute in the structure
StrName	The variable's structure's name
ArrSize	The size of the array variable

Table 2: Symbol's Definition

In this part, I use a very quick data structure: AVL Tree to realize a symbol table. It will be introduced in the optimization part.

3.2 Semantic Analysis

Since not every program that matches the grammar is a semantically correct program, after generating a parse tree, we need to do semantic analysis and syntactic checking to examine whether there is any semantic error.
It is much easier to examine these errors during code generation because we may have know much information, thus my error handling codes are all in CodeGeneration.c. Then the followings are the error conditions I have handled.

Not declared variable

The symbol table is useful in handling this kind of errors since every time the program want to find a symbol in the symbol table, it must fail if there is no such variable. Then we can assert this variable has not been declared.

No re-declared variable

Similar to the former one, the only different thing is to check the error in the insert function. If there already have the same variable, the insert function would return null instead of a normal root's address. Thus we can handle this situation. (The root is the symbol avl tree's root.)

Reserved words

This part is a little different, and I implement in yacc file "Small.y". I create a function named checkWord() in checkReserved.c so it can be used easily. For each identifier read by parser, it will immediately check if it is valid and if not the program will throw the message, then exit the process.

Main function

To check if there is a main function, just define a flag named "ifMain", when the main function is defined, the flag will be set as 1. Each time the code generator reaches at the end of the translation, it would check ifMain and print error message if it is still 0.

Break and continue in for loop

Use a flag for for-loop called forLayer. Every time translate the stmt of the loop, add the flag. After translation of the stmt part, subtract the flag. If the flag equals to 0 when meet the break sentence, that means break is not in the for-loop's stmt.It is an error.

3.3 Optimization

AVL Tree for symbol table

I implement an optimization to accelerate the running time. That is to use AVL tree as a symbol table.
Actually, the most searching algorithms take O(logn) time. They are binary search, binary search tree and etc. So I firstly tried to apply BST on the symbol table, but it cannot confirm O(logn) time for every search since it may not be balanced.
Then I figure out AVL tree, it only takes O(logn) time to insert a new node, and also it ensure every search operation takes O(logn) time.
The implementation is in the avlTree.h file, it also contain some error handling codes.

4 Conclusion

This project is much harder for me, it need much time and patience because I spent nearly 10 hours to debug my code.But it also practise us much more than other projects, I have figured out many methods for debugging the codes.
Besides, it also let us know exactly how a compiler works and how amazing llvm is.
Thanks to all the classmates and TAs who have helped me more or less.