Compiler Report

Guang Yang

May 30, 2014

Abstract

In this report, I present the my work of programming the compiler in the last two months. I design the compiler from five phases–Lexer & Parser,Semantic,Intermediate Code, Code Generater and Optimization. Seen from the final test, it works well.

Key words: compiler,report,pattern,Antlr

1 Introduction

The annual compiler project is near the end, I have learned a lot during these two months of programming the compiler. Thanks to all my teaching assistants for their hard work and all the students who answered my questions and help me complete this challenge. Here are some conclusions about my work during these two months, I will report them by phases below.

2 Implementation

The whole program is divided into five parts but some parts seem to be very close and have strong relation.So in my implementation, I nearly combined the Intermediate Code phase and Code Generater into one phase and that took much of time to implement, nevertheless, it has some advantages in the final Optimizer phase.Here are the detailed progress.

1. Lexer & Parser

We begin the first step by analysis the semantic of the input code, so we need to divide the code into partition–Lexer and use the grammar to parse it into a grammar tree so that it may facilitate the analysis.

In this phase, I choose Antlr 3.5.1 as Lexer & Parser. Antlr 3.5.1 has the function to rewrite semantic trees and can easily generate a nice AST. Due to this reason, I use it in the first phase and this save me much time indeed. But when I do the semantic check I get to know the defects of this tool. Just because of its tree-rewriting function, it takes hundredfold time than Antlr 4.0 or JFlex & Cup to run(especially in the data $expr$). So I refer to the guide for the way to solve it and finally find a new option $memorize = true$ to reduce the time it takes.This option forces the Antlr to memorize the tree it has already scanned and skip these scanned trees when backtracking.It does works. At first it takes $70s$ or so to compile the data $expr$, but after this option it only takes about $1s$–it does works.

2. Semantic

In the second step, we need to do the static grammar check for the input code such as type check and arguments check and so on. And in fact, we can do some optimization in this phase just like constant propagation and constant folding. The hardest problem in this phase is the standard of grammar which require our program to fit in these various conditions.

I have to say that there are so many cases that I can't match them all which results my data-oriented program. Even passing all the data takes me much of time. I think I should have planned a very robust architecture so that I can save much time which have been spent on the debugging. In this phase, I get the string from the Parser and construct an AST from the grammar. Here I don't use the treewalker which is built-in the Antlr, but try to add some virtual tokens which represent the Lex token and lift them as the tree roots. Finally I can grab these virtual tokens and construct the AST. Then I assign a class for each grammar token and perform the check in their own class which I think to be a very convenient pattern.

To record the information passed by its children, I use a class called $returrecord$.

```
public final class returnrecord
{
    public boolean lvalue;      //tag for leftvalue
    public boolean constant;    //tag for constant
    public type rtype;          //type for this token
    public Object value;        //constant value
    public location loc;        //location for next phase
}
```

Designing this class simplify the work for many occasions especially in the grammar $postfix$ and $unary\_expression$.

What's need to say is that due to the grammar pattern of $Logical\_and\_expression$ and $Logical\_or\_expression$, I save much time when dealing with the short-cut. In Antlr, I don't lift the token to form an binary tree for these two expression so they remained plain which heavily simplify the work.

As for the *table*, I use the implementation provided by $xjia$. I put the type of a identifier as the value of it, and when the identifier being used later, I may get the type from the nearest table and set it type before checking.But to solve the problem of distinguishing *namespace* and *scope* , I redesign the *env* part. I use linklist to store these two table separately and call them *structtable* and *functable* respectively.

The implementation of *struct* becomes one of the hardest challenges, but I think I have worked well in dealing with the *struct* and I passed all the bonus data except for GC and rear function declaration.

3. Intermediate Code

In this step, we may do the last work of the front end of the compiler.Compiler is separated in two parts, the front end communicates with the input and output the intermediate code for the backend while the backend deals with the intermediate code and translates them into the MIPS code which can be execute.

It's hard to separate this phase from the next Code Generater phase, so the design in this phase seems to be very important.Firstly, I have a good pattern of the quadruples. But I think the implementation of *array* and *struct* which involved the address are hard to do in the next phase, so I check this in this phase. I use the class *location* to represents the address of every variables including intermediate variables.

```
public class location
{
    public Object contain;      //constant value
    public int offset;          //memory offset
    public String type;         //location type
    public int number;          //register number
    public boolean global;      //tag for global variable
    public boolean address;     //tag for address
}
```

The *location* can be an register, a memory address or a constant, and I also use a boolean tag to chart whether this represents an address. By using this method, I can handle lots of problems like *multiarray* or *multipointer* even *struct*. And I can also use them in the last phase to trace the information I want.

In detail, I want to talk about the implementation of *array* and *struct*. For each array I spare 4 more bits to store the pointer of it. So when I refer the array, I take the pointer to do calculation and locate the address of the data. To distinguish *multiarray* and *multipointer* from locating the address, I need to determine whether the result is a pointer while checking the postfix []. For struct I store them in the memory and don't allocate them a pointer as what I do for the array. When dealing with a struct, I put the offset of every element of the struct in the table, so I can easily locate the address of each element. But this method seems to be clumsy when operating copy. I have to copy the data of a struct word by word when assignment need to be done.

For passing the parameters, I use stack to pass these variables. And for return value, I allocate a new space after the global space to store them.

For common variables ,it has nothing new to say. I have just put the location in the table and merge the location and type into a new class. But for constant, I nearly precalculate them all. I could have done this optimization for all the pointer, but I encountered a problem at passing the parameter. So I gave up the optimization for the pointer and only completed this for all the constant.

4. Code Generater

In this phase, we can output the final MIPS code. Many optimization can be done here. And generally the translator would work here to translate the inexecutable intermediate code into SPIM code. Actually the problem of address should be handled here instead of the former phase.

Just because I nearly skip the intermediate code phase or in other word, skip the code generater phase, I have to merge lots of work in every semantic class. But it still have some advantages. When I run my two-register SPIM, I can pass two third of the data(performance) due to my pattern of code. I can get rid of many useless code in this step and reduce my final code even before the optimization so I have a very simple SPIM code structure which produces very nice performance.

5. Optimizer

In the last phase, we may operate some optimization on our code to reduce the instruction number when running. It's a phase that can do lots of powerful things which may heavily simplify the output code and ensure a good performance.

I use many simple but powerful optimization in this phase such as many peephole optimization, dead code elimination in addition with the output optimization. Before the register allocation, I have already use some constant optimization to eliminate useless code. So after register allocation ,I have only five data need to be passed.

In register allocation, I analysis the stream for the whole code. Due to the increase of the register number, every function has its own registers and collision will never occur. I use inverse order to calculate the stream equation so the time of iteration significantly reduced. Then I use linear scan to allocate the register and for most data 10 registers are enough. When dealing with the spilling registers, I set up a set to record them. And in my architecture, global variables are always spilled for correctness. Taken spilling into account, I always allocate a memory address for each register regardless of whether it is spilled. And for global variables, I use the SPIM instruction *.data* to allocate spaces for them and manage them independently.

I find that peephole optimization is powerful for the structure which are not very concise. Every peephole optimization can reduce the instruction number for even one to one point five million which I haven't expected. And output optimization is also powerful for such data which has lots of output. In summary, even I haven't wrote some complex optimization, my SPIM code still has a better performance for most data.

3 pros & cons

As a conclusion to my compiler, I think it performs good in the test, but still have many features to supplement. Due to my architecture, my SPIM code seems to be very concise but it increases the time to operate an optimization. So I have only chosen some optimizations which can be operated after the generation of SPIM code. If I operated some powerful optimization like common expression elimination and loop unrolling, I have the confidence to challenge the performance bonus. From the results, at least, my design pattern does reduces the instruction number to a large extent.

4 Conclusion

During the last two months of programming the compiler, I have learned a lot on how to program a large project. This project not only develops my debugging skill and also trains my architectural skill. I have recognized that if having a perfect architectural, one would save lots of time and heavily increase the efficiency. So planning the architectural in advance always be a good choice.At last, I may express my appreciation to all of my teaching assistances and classmates who help me weather the storm. Thanks!