Compilers Course Project Report

Dong Xie
Shanghai Jiao Tong University, 2011 ACM Class
xiedong1993@gmail.com

Abstract

This article describes the overall framework and implementation details of the project for compiler course in 2013 spring semester. This project is a reduced C compiler implemented by full featured Java with the support of JFlex and CUP. To generate better assembly codes, this compiler implements numbers of optimizations such as local copy/constant propagation, dead code elimination, common subexpression elimination, etc. In the final test on my own PC, the compiler showed a quite fair performance on all given test cases.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers; D.4.2 [*Storage Management*]: Allocation/deallocation strategies

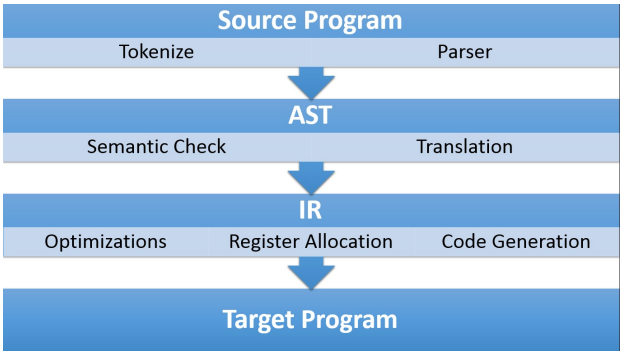
General Terms Compiler Design, Compiler Optimization, Storage Management

Keywords Compiler, AST, IR, Memory Allocation, Optimization

1. Introduction

The task of this project is to design a compiler from a reduced version of C (grammar defined here) to SPIM assembly code. Target codes will be run in statspim to estimate performance of the compiler.

My compiler framework follows the classic mode, which consists of lexical analysis(tokenizer), syntax analysis(parser), semantic Check, IR generation, register allocation, code generation and numbers of optimization. The following picture shows the overall framework of my compiler.



Then, this article will describe each part respectively. I will show the main structure of each part and list main problems in implementation with corresponding solutions.

2. Lexical Analysis, Grammar Analysis

In my compiler, the work of lexical analysis and grammar analysis is implement by codes generated by JFlex and CUP. JFlex is a java lexical analyzer generator, while CUP is a parser generator. These tools can generate a precise lexical analyzer and a parser according to the well-defined rule.

Source program will be first parsed by the parser with lexical analysis. If it does not have any grammar problems, it will converted to AST(Abstract Symbol Tree). Otherwise, the parser will report errors.

2.1 Structure of AST

AST is a tree build up by abstract symbols which determines the orders of operations. There are four basic kinds of abstract symbols: declarations, statements, expressions and others.

- **Declarations:** Declarations are used to mark declarations for a new variable, a new type or a new function. It is the most top level structure in programs and its functions.
- **Statements:** Statements are used to mark statements in functions including select statement, iteration statement, etc. Statements and declarations make up the outermost level of functions.
- **Expressions:** Expressions are used to make all kinds of expressions including binary operation, assign, constants, etc. It may be the most underlying meaningful elements in programs
- **Others:** Such as initializer, declators, parameters, etc. They are the components of declarations, statements and expressions which are defined to make dealing the structure much easier.

For continuous same kinds of abstract symbols, I took list structure but not tree structure because this will make the iteration on these symbols more efficiently.

2.2 Problems and Solutions

- To deal with string const and char const, we must have new states for regular expressions cannot match them correctly.
- In the lexical analysis, we can simply replace escaped chars to original ones. Do it for this can be rather troublesome if you deal with it later.
- Parsers generated by CUP may be have conflicts, one way to resolve them is to statement precedence for some specific symbols.
- Because of the mechanism of CUP, the parser cannot run correctly when simply match typedef-name to ID, I implement it

by using a environment table to keep defined typedef-name, then we can distinguish typedef-name and ID. But this solution still has problems. For once an identifier is defined as a typedef-name, it will always be tokenize as a typedef-name but not a ID.

3. Semantic Check

The main task of semantic checking is to check that for each node in AST if types of its sons meets its demand. If there is something wrong, semantic check should report errors and interrupt the compilation procedure. On the contrary, semantic check can work out all type information, which means we don't need to do it again in IR translation.

Since there are no such things like variable-length arrays, we can calculate memory size for all types which will make memory allocation in IR translation be much easier.

3.1 Problems and Solutions

- Structs and unions can have pointers even if it has not been defined, so we need a kind of "fake" type NAME, and when it is necessary this type should linked with the true one.
- In most cases, integers, chars and pointers can be treated as same, but there are also plenty of exceptions. So, it is important to try it in gcc.

4. IR Translation

Since source program has passed parser and semantic check, we can consider it is correct. Now we need to translate AST to Intermediate-Code.

4.1 IR Structure

All variables in IR should be an address in memory, in my structure there are three type of variables: Temp(somewhere in memory), Label(somewhere in code), Const(just an immediate). Basic elements in IR is quads, each quads will have the same assembly rule when it is translated to assembly code:

- **Binary Operations**: Format is ***dst = left op right***, where left must be a Temp, while right can be a Temp or a Const.
- **Unary Operations**: Format is ***dst = op src***, where src must be a Temp. Contains reference and dereference operation.
- **Move, MoveL**: Format is ***dst = src***, where src can be a Temp or a Const or a Label.
- **Goto**: Unconditional jump
- **Branch, ifFalse(beqz)**: Format is ***branchop left right label***, where left must be a Temp, while right can be a Temp or a Const.
- **CallFunc, CallProc**: Format is ***(ret) = call funclabel [...]***, where all parameters should be temp.
- **Enter, Leave**: Quad appears at the head and tail of a function, which is used to allocate and release stack spaces.
- **Retrun**: Quad returns a value to upper level.
- **Store, StoreB, StoreZ(z = 0)**: Format is ***x[y] = z***, where y must be a Const and x, z must be a Temp.
- **Load, LoadB**: Format is ***x = y[z]***, where z must be a Const and x, y must be a Temp.
- **DataFrag**: Spaces in data segment for string constants.
- **LABEL**: Just a label. :)

4.2 Problems and Solutions

- Since we need to support pointers and register allocation, every Temp need a position in the memory and we can find this address quickly and easily. The solution is for every function, we allocation memory for Temp inside it from the stack, and give every Temp an identical index. Thus, every Temps address can be calculated quickly and nicely as ***sp[index * wordSize]***.
- Keep types of expressions from semantic checking can help to checkout some special cases easily and save compiling time greatly.
- Use an assembly written function ***printf*** would be a great waste, inline it in IR level helps.
- Pay attention to address mode, it is a good solution to distinguish arrays and pointers precisely. (For memory allocation and addressing mode can be very different)
- 3 kinds of expressions should be dealt with specially, they are ***logical*** and, ***logical or*** and ***logical not***.

5. Register Allocation, Memory Allocation

Before generating assembly code, the most important thing is how to allocate register and memory. On one hand if we do not use register, the target program would be rather slow. On the other hand, if we use register, we need to notice that some special memory access (reference and dereference operation) may make values in register expire.

At the very beginning, I just write a simple linear scan register allocation, and then I start to think about how to allocate memory. Let us just take the classical way, we put global Temps in the global space, other Temps in the stack space. And when we need spaces for arrays, structs or unions, we simply allocate requested size on the heap space. However, following problems are raised, and here we give solutions in the same time.

5.1 Problems and Solutions

- Global variables are shared among functions. When a function changed one of these global variables, how can the other functions have the correct values? This problem appears because our register allocation is locally in the function. Thus, in different functions, register for same global variable could be different. Therefore, we now have three choice. The first one is to spill all global Temps so that all operations on global variable will be done the same on memory. The second one is to allocate register globally on global variables, thus, there will be no problem when you jumped from one function to another. The last one is that when entering functions or finishing a function call, load all global variables needed, and when a global variable is not needed any longer in the function, then save it, or before function call, save all global variable remains. Comparing these three method, the first one is clearly the slowest, and the second one may cause a lack of registers within function. So, I choose the third way at last.
- When pointer operation is called (reference or dereference), how to keep values in register and values in the memory same? Of course, we cannot spill all Temp. Another way is to store all changed register to memory. But if we consider user only do safe pointer operation, we only need to spill all Temps referenced for user would not access some other space.
- Where should I put extra ($parametercount > 4$) parameters in? Of course we can just do as the classic way to save in the stack. But if we save it in data segment and use a system register to keep it, we can save some time(less memory access time).

6. Code Generation

This is the last step before the target program can run. If all preparations are done well in last steps, this part would be quite easy for we just need to use some fixed rules to translate quads to assembly. Some tricks such as formatted strings can help us to implement this period succinctly.

It is worth noting that in order to make the target program run normally, few function labels should be renamed such as toplevel and main. Besides, we also need to implements a few default functions.

In addition, I implement a continues label eliminator and a useless datafrag eliminator to make the target program more beautiful and tidier.

7. Optimization

Implemented optimization shows as following: (All listed optimization can correctly optimize given test cases)

- **Local Copy/Constant Propagation:** Propagate copy and const within basic blocks, judge by a two-way hash set. Need to work with dead code eliminator for taking effects.
  P.s. I suppose that local copy propagation in project tiger by Xiao Jia has some mistake, and my version is a refine of that.
- **Dead Code Eliminator:** Eliminate dead code within a function. Judge if a quad is dead code on the result of liveness analysis.
- **Common Subexpression Eliminator:** Propagate same subexpressions within a function, replace them by its first correct calculated destination. Then you can send it to dead code eliminator and expect a fairy well result. For the correctness of the optimization, a available subexpression analyzer is needed. (For my IR is not SSA mode)
  P.s. Also, I suppose that common subexpression elimination in project tiger by Xiao Jia has some mistake, and my version is a refine of that.
- **Shortcut Expression Optimization:** For shortcut expressions (e.g. ***a && b && c && d && e***), add a fall label to make wherever the branch is taken, PC will jump to the outermost level at once.
- **Peephole Optimization:**
  - **Constant Operation Calculator:** Calculate all binary/unary operations that can be calculated while compiling.
  - **Branch Compression:** Replace ***x = a cmpop b*** and ***ifFalse x label*** by ***Branchop a b label***.
  - **Store Zero Folder:** Replace ***x = 0*** and ***y[z] = x*** by ***y[z] = 0***.
  - **Strength Reduction:** Eliminate times or divide 1 and refine times or divide power of 2.
  - **Goto Compression:** If the target of Branch or Jump is a Jump, then compress it.
  - **Naive Assembly Eliminator:** Eliminate self move and continuous ***move***, ***li*** or ***lw***

8. Conclusion

As of today, I successfully finish this reduced C compiler. In these past 2 months, I tried my best to code my first compiler. This is a very precise experience to me, and I have gained a lot from this project. To tell the truth, I think I can do better if I have more time. Thus, I suppose I would continue this project and make it better when I have got some spare time.

At last, I would like to thanks TAs, my classmates and everyone who helped me on this project. Thanks very much! :)

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi Jeffrey D. Ullman. *Compilers Principles, Techiques, & Tools.* China Machine Press

[2] Andrew W. Appel *Modern Compiler Implementation in Java.* Published by Cambridge University Press