Report

Document for Compiler 2013

Li Yit
5110309044

ACM 11, SJTU, Shanghai
E-Mail : lrank@sjtu.edu.cn

Abstract

This is a report for the project of mine in Class Compiler 2013.

The purpose of this class is to understand how the compiler work and details of the area of the compiler by finishing this project.

The task of this project to design a compiler which translate a C-like language into MIPS assembler code and imply the compiler. This document supports the details that how I achieve this job.

Generally, a process of imply a compiler is divided into four steps: parsing, semantic checking, intermediate-code generation, and code generation.

I will talk about the each step one by one, and thing correlated to them, such as the details that how I got to imply this project, the problem I met and the solutions of some of them.

Keywords Compiler2013, C-like language, MIPS assembler

1. Introduction

1.1 Background

This is my sophomore year in ACM 11, and conventionally, we wound be taught a class entitled 'Compiler Principles, Techniques and Tools' with a experiment project, in which we need to finish a project of compiler that I mentioned in the abstract. This year, the compiler class is canceled this semester. However, the project is reserved by some reason. In another word, we only had to consult TAs who were in charge of this project on everything we would ask about.

In fact, these TAs are excellent teachers for me because of their professional knowledge and their jobs.

1.2 Programming Environment

The core of this project is implement. So, programming environment is given:

I chose the programming language of JAVA, (C or C++ could be ok, but it could make the project a little more difficult than JAVA), and I use ECLIPSE Juno as my platform with JAVA SE 1.6 inside.

2. Parsing - Syntactic Analysis

Parsing is the very first thing we need to do of analysis a C program. Actually, TAs had already designed a grammar for our C-like language. And the grammar was well-designed and it was very close to C language.

2.1 Token Seize

We consider that the minimal element of a program is a token, such as a keyword, a variable, a operator and etc. That means a program consists of many tokens.

Therefore, the first thing we need to do is obtaining separated or free tokens. We use regular expression to match tokens one by one. However, it is not convenient to write a automata by hand. Helpfully, I use JFlex, which does not work quite well but still enough to some extent, to get these individual tokens.

2.2 Syntactic Analysis and AST

Combining the tokens legally, where legal mean the combination can be accepted by the grammar, we will make the expressions, and combine the expressions we make statements, and at last we make the program. That is to say, the grammar itself is a rule of generating of a language. Also helpfully, I use the tool named Cup, which can help us to combine tokens from free ones and identify whether the program is grammar-legal or not.

We write the rule or Grammar into Cup, and if the input is accepted by the Cup, the input is grammar-legal.

And after the process of Cup, we can get a parse tree as we want.

When asked how I design my Abstract Syntax Tree, I just cut out some unnecessary things on the parse tree, and nothing special. It can handle some of them when I do the parsing work.

2.3 Problems and Solutions

2.3.1 Typedef Problem

In parsing process, the most problem I met is about the things of typedef, which allows programmer to define their own new type. That makes a question that when I should consider an ID as the ID itself, where ID represents a name of a variable here, and when I should consider an ID as a typeID, which could be a name of user-defined type.

This persecuted me very much. Actually, many of us could not solve this problem, and TA Xiao Jia made responses to our confusion, submitting a repository entitled typedef_example on GitHub.

However, this could not help a lot. In his solution, once we consider a variable as a typeID, it will not be able to be considered as an ID again, however things are different in our language. This made a real bug of the design of our language, and we have tolerate the conflicts in the next work.

A reasonable solution is to set the environment when making the parsing to decide whether an ID is actually an ID or a typeID. This solution is troublesome for needing hundreds lines of codes and having to modify the JFlex and Cup.

2.3.2 Parsing Tree Optimization

We can do some optimizations on the CST, such as cutting the redundancy brunches, merging the brunches, cutting the useless nodes, and etc. These optimizations will be helpful to the following steps.

3. Semantic Check - Semantic Analysis

An illegal program can match the grammar, but it could not make sense from the view of semantic. Therefore, we need semantic checking.

3.1 Be Compatible with GCC

Because we were making a compiler for C-like language, so, our TAs chose GCC as the standard of the semantic checking.

As we all known, GCC is very popular amount the programmers who use C. However, it has some strange semantics, too strange to think at least in my view, about some ambiguous cases.

It is a hard work to make sure my compiler to be compatible with GCC. The general cases are easy-considering, but to solve the others, we had to test every cases on GCC and make a list of hundreds of rules and add them into my semantic checking one by one. This is a real big and unpleased work.

3.2 Redesign and Rebuild

The first time I realized that it might not be perfect for the design of my framework, for instance, the way I organize relationships of the types, the way I manage the environments, and etc. It was a mighty work for me to redesign everything and to reprogram the whole project.

' Rome was not built in a day. ' Re-dos often happen as TAs told us. I could not take everything into consideration at the every first time, and the more code I wrote, the more problems I could found.

Therefore, keeping writing and thinking is a good choice.

4. Translate - IR Generation

The third part of my project is translate, which translate the origin code into intermediate-code as its name represented. Also, I need to design a set of intermediate codes myself.

A well-worked intermediate-code need to transfer the useful information from the AST to intermediate-code. Therefore, in this part, we decompose the tree, reorganize the elements, and integrate them into linear list of intermediate codes.

It is not easy as I thought, maybe it is easier that we just translate the origin code into the MIPS assembler because it is very ease to lost the information and mix the relationships between items up and down. All of these I talk about are actual problems I met. Being unfamiliar with MIPS assembler, I suffered a lot because I did not know whether the intermediate code I designed was practical or not.

4.1 Details and Problems of my IR

Here are details of IR:

- Tcode : The base class of all the IRs;
- InCode : The three address code(3AC), where dest, operator, src1, src2 are recorded.

  In the 3AC, it is worth to be careful about the temp variable is taken by a address or a real value. This is important for the following code generating;
- Label : The Label actually generate a label in MIPS;
- Define : The Define code means we define a new variable or intermediate one, where the name, type, offset and level will be recorded;
- Callcode : Used to call a function;
- TReturn : Used to end a call of a function, as the pair of Callcode;
- PARAcode : Used to pass the parameters to the function before calling it;
- Gotocode : A real goto code;
- NotIfcode : Used as iteration expression, here I omit the Ifcode because they are the same in function, and one will be enough.

This design of IRs is not complex, but it is not perfect and not enough as well. therefore, I created some vectors, such as ArrayList, to contain the information i missed. I still think this IR generating is the worst part of mine, though I change it many times.

It still has the space for performance optimization.

5. Codegen - Code Generation

In this part, I need to translate the IR codes into the correspondent MIPS codes.

5.1 Access to Variables

At first, I chose to put both global and local variables in the stack. This is available. But, at last considering the limit of stack and the easy access of global variable, I changed it and put the global ones out of the stack. However, there are no great differences. When it come to me, that when a variable need a very large space, such as an array, it could be unable to access the variable by 'load or store \$d, (im)\$s', where the im is limited in scale, I chose to set a label for every global variables for easy access. Therefore, I separated the global and local variables in memory, where the local ones will be put in stack.

Stack organization is a important part of codegen. Calling a function will cause use of the stack. Table 1 gives a diagram of a stack use caused by a function-calling.

Table 1. Layout of the stack

Old function (Higher memory address)
Function return value (\$fp + 4)
Parameter 1 (\$fp)
Parameter 2
. . .
Parameter n
Local variable 1
Local variable 2
. . .
Local variable m (\$sp calculated = \$fp - (n + m) $\times$ 4)
Size of this function in stack (\$sp - 4)
New function (Lower memory address)

As we seen, the size of stack the \$fp - \$sp + 8 bits.

You can choose to store the address in the stack and put the value in the heap or global memory. I think it has advantages when it is involved the point operation. You wound not make a confusion by that way.

However, I still chose to store the value in the stack. It is more intuitive but I must keep my mind clear. Because I divided the global and the local variables into different memory space (the parameters were seen as the local ones), I have to distinguish them when I access to them.

- global

  When I define a global variable, I will set a global label. So, I load label into the $gp, and calculate the array or struct offset. then, I get the accurate address of the global variable.

- local (and parameters)

  All the local variables are settled in the stack space related to the current function. So, Directly access to the locals by $fp and the offset, which has calculated when defined.

- other

  When the variable is a string, I barely load or store the address of the beginning of the string, which generally stored in the global memory.

  This is like the first method of storing the address in the variable.

However, it is not enough. In my first consideration, I diff the array and pointer, which makes a hidden bomb myself because it obey the fact that array can access by point. I can fix it by set some flags, but when it comes to 2-dimension or high-dimension pointer, it does not work again. I have to commit another failure in my design.

5.2 Printf Design

Standard functions such as malloc and printf need to be designed by ourselves. Function malloc is not difficult and I will pass it. I only show the details of design of the function printf:

Considered the performance of the code, I wrote it in MIPS completely by hand.

1. parameters copy, this is no different with calling other function, in which the first parameter is a string, which is store with the global variable as a string constant.
2. check every char of the string one by one and print the parameter if matched by %, or print the current char if not. When matched the 0, printf ends and return;
3. In MIPS, we have print_int and print_str. So, how to print a char? My solution is to new a space of two bytes called print_buf, the first position hold the char I want to print, and the second position will be zero all the time. Then I just call print_str to print the print_buf.

5.3 Register Allocation

Register from $t5 to $t9 are free to use.

Considering my poor time and that to pass the testcases has the highest priority, I only finish the random register allocation, which does not preform very well.

Graph coloring is a little trouble to write.

Linear scan is a good choice because it is quite easy to understand and to practise.

6. Bonus

- Pretty printer

  When I get my tokens, I have a set of rules to print these tokens to make them in a pretty form. Therefore, I can de-token, and print the C-like program back. However, it may not be the very origin program because my token will change them.

- Higher-order functions

  My solution is to consider function as a kind of type, and the basic type is pointer. I transfer nothing but pointers, which could be easier to access to what I want, even a function. Of course, I need to change the grammar to support the type of function.

  Actually, it worked somehow though I did not do enough tests. Unfortunately, I just considered it as an alpha version, and it did not appear in my last version.

- Garbage collection

  Once I took it seriously, but I found myself can not make a complete garbage collection because of the abuse of pointer in this C-like language. I can make the arithmetic operation to a pointer, that is to say, I can access to a memory by almost any other pointers. This can make the garbage collection unreliable.

  Therefore, I gave it up. But if I face a JAVA-like language, I think I am competent to this job.

- Optimization

  All of the optimizations are mentioned before. I have not done much about it.

- Refine Wikipedia or translate into Chinese

  Underworking. I will keep refining it and put the result on the my Github. Maybe the article is not display well because of the coding is different in Windows where I wrote them.

- User-friendly error information display

  If a input go wrong, I can show whether the error is caused by the grammar, or by the semantic. Both of them can show the wrong position of grammar or semantic.

  It is easy to achieve that. Merely record the positions of every tokens when parsing. Then, print the types of errors and their position after grammar and semantic checking.

7. Failure

Actually, I have not done this job well. I found this bugs in my program, I am sorry that I did not have time to modified them.

1. Printf module can not support well.
2. Pointer processing problem. I have not released Pointer and Array are the same thing in C program, that made it down.

8. Impressions and Feelings

This project named compiler 2013 is biggest project that I have even written. I really impressed on it and learned a lot, not only the knowledge of compiler, C-language, code writing and etc, but also some of corporation, hardworking and life.

1. The first thing I had mentioned above - ’ Rome was not built in a day. ’ Be patient and be ready for doing it once more before you succeed.
2. The second thing is after learning LC3 in the class Introduction to CS in my freshman year, I thought I grasp the assembler, C and C++ language. But now, I still have a long way to go. What I got before is only a tip of the iceberg.
3. The third thing, when in busy, managing the time as well. Everything is important though some of them are more.
4. More.

A. Appendix

- My Repository of Compiler2013

  It could keep updating.

  https://github.com/lrank/compiler2013.git

- Repository for Refining Wikipedia on my gihub

  https://github.com/lrank/Refine-Wikipedia-Compiler.git

- Repository of Tiger by stfairy

  Thank you for providing such a excellent template of compiler and it really helps us a lot.

  https://github.com/stfairy/tiger.git

Acknowledgments

1. Thank all of the TAs in Class Compiler 2013 for their work and consults.
2. Thank Wikipedia for supporting the opportunity of the bonus.
3. Thank all the classmates who provide assistance to me.

References

[1] Alfred V. Aho, Monica S.Lam, Ravi Sethi, Jeffrey D. Ullman *Compilers Principles Techniques and Tools (2nd Edition)*

[2] Andrew W. Appel *Modern Compiler Implementation in Java*