Java Implementation for C Compiler

Kaichun Mo, *Shanghai Jiao Tong University*

***Abstract*—This is the report for my compiler project in spring of 2014.**

***Keywords*—Compiler Report, Syntactic Analysis, Semantic Analysis, Parser, Intermediate Code, Register Allocation, Optimizations, Function Inline, Dead Code Elimination**

I. Introduction

THIS is the report for my compiler project in spring of 2014. In this report, I mainly discuss about the implementation details when dealing with syntactic analysis, semantic analysis, translation to intermediate code and many reasonable optimizations.

II. Syntactic and Semantic Analysis

A. Lexical Analysis

I use *Jflex* tool to generate my lexer. Special attention should be devoted to how to deal with string constant and how to ignore comments and includes.

1) Ignorance of Includes: I just use DFA to match includes and just ignore them.

```
IncludeComment = ("#" {White}* "include" {White}* "<"
 [a-zA-Z.]* ">")|("#" {White}*"include" {White}* "\""
 [a-zA-Z.]* "\"")
```

where

```
White = [ \t\f]
```

2) Ignorance of Comments and Recognize String Constant: Just set up new yystate to record all the content in comments or strings. Nothing special.

B. Parser

I use *Java_Cup* tool to generate my parser. Special attention should be devoted to how to deal with $*$ or $+$.

For the grammar below.

$$A \rightarrow B*$$

I use two different translation methods to ease the later coding.

1) Collect as a List: Look at the following code.

```
A ::= B_list:l B:e {: l.list.add(e); RESULT = l; :}
    | B:e          {: RESULT = new B_list(e);     :}
    ;
```

Using this way, we can collect all the B's in a LinkedList, which can make it easier to sequentially traverse all the B's.

2) Collect as a tree: Another way is to generate the elements as a tree.

```
A ::= B_list:l B:e {: RESULT = new A(l, e); :}
    | B:e          {: RESULT = new A(e);    :}
    ;
```

This method are widely used when dealing with binary expression. Since the operands of a binary expression has potential priority during execution, presenting binary expression as a tree is a better option.

C. Semantic Analysis

The job of this phase is to give each variable a type and consequently check type matching when necessary aiming to guarantee the validity of the program. The job is easy to do besides the following challenges.

1) Type for Array: Consider the following declaration for array a.

```
int a[10][20];
```

We should set

$$a\text{: Array(Array(int, 20), 10)} \ (\surd)$$

instead of

$$a\text{: Array(Array(int, 10), 20)} \ (\times)$$

2) Operation between Pointer and Integer: I set up a list of rules to judge whether a pointer type variable can do specific operation with an integer. For example,

- pointer-pointer ($\surd$)
- pointer-integer ($\surd$)
- pointer+pointer ($\times$)
- integer-pointer ($\times$)
- etc.

3) Nested Definition for Struct: Since the struct does not mean the beginning of a new environment, it's not valid that

```
struct A
{
        struct A
        {
                int x;
        } y;
};
```

After noticing this problem, it can be solved easily.

4) Struct Used before Declared: Since the following used before fully declared is valid, we need to deal with this problem specially.

```
struct A
{
        int x;
        struct A *next;
};
```

I solve this problem simply by rechecking. During the definition for struct A, I firstly give the variable *next* a type named `Name`. Immediately after the definition for A is finished, I just simply recheck all the fields in struct A and modify all `Name` type variable to its regular type with A involved.

5) Constant Computation: Operations between constants can be executed during compile time. For example,

```
int a[8+8-1];
```

where $8 + 8 - 1 = 15$ is necessary when determining the size of *a*.

III. Intermediate Representation

I generate my own intermediate representation aiming at to ease the translation from IR to final MIPS code as well as retaining a relatively large possibility for back-end optimizations.

I create a Java class named `IRStmt` as the super-class for all IR statements. While, a class named `Temp` is created to store all temporary variables in the IR code and a class named `Label` is created to remember the information for labels, including function labels and normal branch labels.

Similar to the MIPS code style, my IR statements are classified to the following 27 quadruples.

A. Conditional Branch Instructions

1) Beq: Contain two `Temps` *reg1*, *reg2* and one `Label` *label* which means *beq reg1, reg2, label*.

2) Bne: Contain two `Temps` *reg1*, *reg2* and one `Label` *label* which means *bne reg1, reg2, label*.

3) Beqz: Contain one `Temp` *reg* and one `Label` *label* which means *beqz reg label*.

4) Bnez: Contain one `Temp` *reg* and one `Label` *label* which means *bnez reg label*.

B. Expression Computation Instructions

1) BinaryOp: Contain one binary operator *op* and three `Temps` *res*, *left*, *right* which means *BOp res, left, right*, where *BOp* are all possible binary operation instruction, just as `add`, `sub`, `sll`, etc.

2) BinaryOpI: Contain one binary operator *op*, two `Temps` *res*, *left* and one immediate number `right` which means *BOpI res, left, right*, where *BOpI* can be `addi`, `andi`, `xori`, etc.

3) UnaryOp: Contain one unary operator *op* and two `Temps` *res*, *reg* which means *UOp res, reg*, where *UOp* are all possible unary operation instruction, just as `neg`, `not`, etc.

4) Compare: Contain one comparison operator *op* and three `Temps` *res*, *left*, *right* which means *COp res, left, right*, where *COp* can be `slt`, `sgt`, `sltu`, etc.

5) CompareI: Contain one comparison operator *op*, two `Temps` *res*, *left* and one immediate number *right* which means *COpI res, left, right*, where *COpI* can only be `slti` and `sltiu`.

C. Unconditional Jump Instructions

1) Jump: Contain one `Label` *label* which is the target of this jump instruction. This IR statement can be directly translate to MIPS code *j label*.

2) JumpReg: Contain one `Temp` *add* which stores the target of this jump to register instruction. This IR statement can be directly translate to MIPS code *jr reg*.

3) Jal: Contain one `Label` *label* which is the target of this jump and link instruction. This IR statement can be directly translate to MIPS code *jal label*.

4) JalR: Contain one `Temp` *reg* which stores the target of this jump and link to register instruction. This IR statement can be directly translate to MIPS code *jalr reg*.

D. Load/Store/Move Instructions

1) LoadA: Contain one `Temp` *res* and one `Label` *label* which stores the address of the data. This IR statement corresponds to MIPS code *la res, label*.

2) LoadI: Contain one `Temp` *res* and one immediate number *number* which corresponds to MIPS code *li res, number*.

3) LoadW: Contain two `Temps` *res*, *add* and one immediate number *offset* which corresponds to MIPS code *lw res, offset(add)*.

4) LoadB: Contain two `Temps` *res*, *add* and one immediate number *offset* which corresponds to MIPS code *lb res, offset(add)*.

5) Move: Contain two `Temps` *res*, *ori* which corresponds to MIPS code *move res, ori*.

6) StoreW: Contain two `Temps` *res*, *add* and one immediate number *offset* which corresponds to MIPS code *sw res, offset(add)*.

7) StoreB: Contain two `Temps` *res*, *add* and one immediate number *offset* which corresponds to MIPS code *sb res, offset(add)*.

E. Function Call Related Instructions

1) FuncCall: Contain one `Temp` *res* to store the possible return value of this function. This class also contain `FuncEntry` *entry* which contains all the information about this function.

2) Param: This class is designed to prepare the function arguments before function call. For example, if `a=f(b,c)` shows in the C code, I will generate the following IR codes.

Param reg_b
Param reg_c
reg_a = f()

This method can help me handle function parameters better when translating to final MIPS code.

3) getParam: This class is designed to receive the function parameters in. Using the same example above, in code segment for f, I will generate the following IR codes first.

```
f:
  getParam t1
  getParam t2
```

to get the parameter b and c into temporary `Temp` *t1* and *t2* respectively.

4) Return: This class contains a single `Temp` *res* which means *move $v0, res* to return the function return value in MIPS code.

F. Other Useful Instructions

1) LabelStmt: This class contains one `Label` *label* just to print *label:* in IR code.

2) Syscall: This class just corresponds to MIPS code *syscall*. This class can be used when inlining the function *printf*.

3) Nop: This class just stands for the ending of a function. When I meet a `Nop` when doing MIPS translation, I can begin dealing with the final procedure of this function, just as load the original *fp* and *ra* back, adjust the *sp*, *fp*, etc.

IV. Final MIPS Translation

MORE methods should be mentioned when translating from IR code to the final MIPS code.

A. Using MIPS Directives

For better using the features provided by MIPS, I put all the global variable in data segment. This can help me better handle the global variables from one side, but this idea bring me some troubles when doing register allocation for global variables from another side.

I create a class named `Directives` as a super-class for all directives. I use 5 classes to handle this problem.

Directives Classes

1) DotAsciiz: For global string constant, just like

```
char *s="Hello, world!"
```

I use this class to store the string as follows.

```
L1:
    .asciiz "Hello, world!"
```

2) DotByte: `.byte` can be used to store a global character.

3) DotWord: `.word` can be used to store some consequent numbers.

4) DotSpace: `.space` can be used to apply for some blank continuous space.

5) DotLabel: This class is similar to `LabelStmt` in above quadruples. The only difference is that this class extends `Directives` rather than `IRStmt`.

Using the directives can help me better handle the global variables. When I need its address, I can use `la reg, label` to receive its address and use `lw reg, label` to get its value in final MIPS code.

B. Dealing with Struct Return and Struct Parameter Passing

1) Struct Return: If a function return a struct or union, I take the following method to deal with this problem. I pass one more parameter to this function, which is the start address pointing at a blank space of this struct size. When the function finishes computing the result, it chooses to directly write the answer in this space, instead of return the struct instance back.

2) Struct Parameter Passing: I can just put the whole struct content at the parameter area as what we do to normal parameters. And, what we need to do more is just to tell the function the information about each parameter's size and its start address in the memory.

C. Dealing with Printf and Malloc

At first, I write two MIPS function `printf` and `malloc` to implement these two needed functions. But, to optimize the performance, it's compelling to inline these two functions.

1) Malloc: When we need to allocate some space for a variable, we can use the following codes to do that.

```
li $v0, 9
li $a0, size
syscall
move reg, $v0
```

where the `Temp` *reg* store the address of this space.

2) Printf: When we encounter with the following code.

```
printf("1:%d, 2: %c end.",a,b);
```

We can automatically split this into three instructions during compile time, if the print format is a string constant.

```
print_str("1:")
print_num(a)
print_str(", 2: ")
print_char(b)
print_str(" end.")
```

where each little function can be implemented easily using `syscall`.

Thus, using this idea, it's quite easy to inline `printf` as much as possible. But, if `"%0nd"` occurs or the print format is not constant, we cannot inline `printf`, I just simply call my MIPS `printf` procedure when suffering those situations.

V. Optimizations

TO strengthen my compiler, I implement some classical optimizations as well as some little but powerful ones. Here are they.

A. Register Allocation

This is the most powerful optimization before generating final MIPS code. I use *Linear Scan*[4] Method to do this. It is a general and compulsory optimization when implementing compilers. There is no need to describe the procedure. The method can be easily access from any compiler book.

I use 19 registers when doing linear scan, and I only allocate for local variables. Since I do not use SSA form IR, to guarantee the correctness, I do the allocation procedure in every basic block and connect the alive sections for each temporary variable together as the alive section of this variable. For example, if variable *t1* has two separate sections $[3, 10]$ and $[15, 30]$, I take $[3, 30]$ as the alive section for *t1*. This looks like a waste of resources, but it works well if the program do not need too many variables.

B. Function Inline

I implement function inline for the funtions satisfies all the following properties.

- Its size is of small scale.
- It does not have any declared variables.

- It does not call any other functions.
- Its whole parameters and return value are not of struct or union type.

I follow the procedure below to do the function inline.
Input: IR codes
Output: optimized IR codes
Procedure:

- Compute out all the functions that can be inlined and include them in set S.
- Locate all function calls whose corresponding function is $s \in S$.
- Rename all temporary variables in s in order to eliminate the name conflicts after inlining.
- Replace all formal parameter variables in s with real variables in f, where f is the function that calls s.

For example, the following code can do function inline.

```
f:
  getParam $t1
  addi $t2, $t1, 1
  Return $t2
g:
  Param $t3
  $t4=f()
```

The optimized IR code is as follows.

```
g:
  addi $t5, $t3, 1
  $t4=$t5
```

Using the function inline method, my MIPS code can be more efficient.

C. Global Variable Optimization

Since my dealing with global variables is not so good and all the global variables need load before use, I come up with some ideas to optimize this problem.

1) Optimization in Basic Block: In one basic block, one global variable can be used more than one time. Since its address is fixed, I can just retain the first occurrence of `LoadA` instructions and delete all others. But after this simple modification, I need to modify all the registers storing its address to the one register survived. For example, I can optimize the following codes to a better way.

```
la $t0, L1
lw $t1, $t0
la $t2, L1
lw $t3, $t2
```

After the optimization, I get a better code below.

```
la $t0, L1
lw $t1, $t0
lw $t3, $t0
```

After using this optimization, I find the performance get better if there are relatively large number of global variables involved.

2) Optimization in Functions: If a function has many global variables involved and it does not have function calls to another functions, I can just load all the global variables' addresses at the first place when entering this function. This optimization works considerably for a small scale of programs in the test set.

D. Dead Code Elimination

Dead code can be detected during any phase of coding my compiler.

1) Useless Variables: When doing register allocation, we can find some useless temporary variables. We can just simply remove them from the IR code without affecting the result. The useless variable should satisfy the following property.

- The variable is defined, but it is never used.

For example, when we encounter with

```
i++;
```

we just simply translate it into

```
move $t0, reg
addi reg, reg, 1
```

where *reg* stores the original value of *i* and *$t0* is created for possible subsequent use the original value of *i*. But, since *$t0* is defined but never used, my compiler just ignore the `move` statement. This makes sense, because the purpose of this code is just to increase the value of *i*.

2) Redundant Statement: If the following code occurs,

```
if(0==1)  [stmt1]
else  [stmt2]
```

there is no way [stmt1] can be executed. Thus, just remove the garbage code and keep the useful code can save more execution time.

After the optimization, we get

```
[stmt2]
```

which is more efficient and concise.

E. Other trivial Optimization

1) Pseudo-instructions Elimination: Sometimes, using pseudo-instructions can waste more time. For example,

```
seq $t0, $t1, $t2
```

are translated to four real instructions in SPIM simulator. But after thinking a little bit, I find that using two instructions can implement `seq`, which contributes more efficiency.

```
xor $t0, $t1, $t2
sltiu $t0, $t0, 1
```

The same optimizations can be done to `sne`, `sge` and `sle`.

2) Short-circuit Optimization: When we encounter with

```
a && b && c
```

it's very natural to do the translation as follows.

```
    t1 = 1
    beqz a goto L1
    t3 = 1
    beqz b goto L3
    beqz c goto L3
    j L4
L3:
    t3 = 0
L4:
    beqz t3 goto L1
    j L2
L1:
    t1 = 0
L2:
```

But the efficiency is terrible, compared to the following optimized code.

```
    t2 = 1
    beqz a goto L1
    beqz b goto L1
    beqz c goto L1
    j L2
L1:
    t2 = $0
L2:
```

which uses Short-circuit Optimization.

Acknowledgement

THIS report uses IEEE latex template `bare_jrnl.tex`, which is a template designed for computer science related journal papers.

References

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

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

[3] James R. Larus, *Assemblers, Linkers, and the SPIM Simulator*.

[4] Massimiliano Poletto and Vivek Sarkar, *Linear Scan Register Allocation*.