Compiler Report [*]

Compiler for C programming language

Chen Nanxin
Shanghai Jiaotong University
bobchennan@gmail.com

Abstract

This paper is a report of the compiler project. .

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

General Terms Algorithms, Performance, Design, Compiler

Keywords Parser, Semant, Translate, MIPS, Computer Language, C programming language, Functional programming

1. Parser

This includes syntactic analysis and lexical analysis.

1.1 Lexical analysis

Read the documents for JFlex. In this section we need to delete other information such as includes and comments. There are a few things to be mention.

In C programming language we can use typedef to simplify some codes. But in JFlex it's hard to distinguish the identifier and typedef name. At first I mantain a table for querying which cannot deal with this condition:

```
typedef int x;
x x;
```

when we meet the second "x" in "x x;", we will look after the symbol "x" in the table which tells us that "x" is a typedef name. So we will fail in the synactic analysis because "x x;" is translated into "typedef-name typedef-name;". So we can not solve this problem only in lexical analysis.

1.2 Syntactic analysis

Read the documents for JCup. In this section we need to modify the grammar at first because JCup doesn't support some metacharacters such as "+" and "*". But if you use ANTLR, the only thing you need to do is to copy and paste the grammar in your codes.

[*] This work was supported by cnx.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

PLDI 05 May 20–30, 2013, Shanghai, China.
Copyright © 2013 ACM x-xxxxx-xxx-x/xx/xxxx. . . $15.00

In order to deal with the problem, we need to distingush the variable definitions. It can be done in syntactic analysis, so we can handle this condition:

```
typedef int x;
x y, x;
```

But we will fail in this condition:

```
typedef int x;
x x;
```

Why would this happen? Because of the algorithm, JCup needs to read one more token. It means when JCup tells us it's a variable definition, "x" was already taken as a typedef-name. In yacc(a syntactic analysis tool for C programmer), we can hack it to modify the pre-readed symbol. But it's hard for JCup users to hack it because of the jar format.

Therefore we should use other method. A method which is used by Wuhang is to look for the last two tokens. This works well for almost cases.

Another core issue is to generator the AST(Abstrac Syntax Tree). It can be done using semantic actions. I just merge some rules in grammar and translate the symbols in grammar into some classes. For example, there are something looks like this:

```
postfix-expression: primary-expression postfix*
postfix: '[' expression ']'
       | '(' arguments? ')'
       | '.' identifier
       | '->' identifier
       | '++'
       | '--'
```

I have merged the definiton of postfix-expression and postfix into one rule and translated into this classes:

- Array-expression
- Function-expression
- Dot-expression
- Pointer-expression
- Inc-expression
- Dec-expression

Also I have read Yacc is dead: An update which introduces a new algorithm for parsing. This is a intuitive algorithm which may hit exponential complexity in the worst cases. But it's easy to understand and easy to implement using funcitonal programming. Memoization and Laziness need to be used for the infinite descent on some recursive context-free language. There are some Java implements in Java-Parser-Derivatives and Parsing with Derivatives(a series of posts on derivative-related techniques).

2. Semantic check

This is the most boring part of the compiler. All we need to do is to improve our C programming language skills and try our best to think about all the details in C programming language. There are some difficulties in semantic check:

- Pointer types
- Relation between char type and int type
- incomplete type and function prototype
- Lvalue. For instance, an array is not a modifiable lvalue.

I also have finished a source code indenter, formatter and beautifier for the C programming language. It's quite easy to use. Just add the name of the saving file for formatted code after the command line and you will get the code after formatted.

3. Translation

This part is similar with semantic check because we only need to visit every node in the AST and try to translate it into intermediate language. At last my intermediate language includes 17 forms:

- Address operation expression: Calculate the address of the expression. This expression must be L-value. For example, a=&b.
- Binop expression: Calculate the result of the binop expression. For example, a=b+c.
- Call expression: Call the function definited in somewhere. For example, a=gcd(b,c).
- Enter label: The label marks the beginning of the function.
- Goto expression: Goto the label which marks somewhere. For example, goto L1.
- If expression: Goto the label if the temp not equals to zero.
- IfFalse expression: Goto the label if the temp equals to zero.
- LABEL: Marks the label here. For example, "L1:".
- Leave label: The label marks the ending of the function.
- Load expression: Load the value in somewhere in the memory. For exmaple, a=b[c].
- Malloc expression: Call malloc function.
- Move expression: An assign. For example, a=b.
- Return expression: Return some value for functions.
- Store expression: Store the value into somewhere in the memory. For example, a[b]=c.
- Storez expression: Store zero into somewhere in the memory. This is a special case for store expression which can uses $zero.
- Unary expression: Calculate the unary operator expression. For example, a=~b.

These forms are easily to translate into assembly language. But there are also some things we need to do or we can do to optimize our intermediate code:

- We can do register allocation which will greatly reduce the instructions. I use linear scan to do register allocation because linear scan algorithm is more convenient to implement than graph coloring. At last I don't use $fp register.
- We can translate the code such like "a=b*4" and "b=a/4" into bit operations. Four times are always using for address calculation.
- For successive and operation or or operation, we should implement short-circuit evaluation. For code "a=b&&c&&d", I will translate to these intermediate codes(if *a* maps to R1):
  1. IfFalse b goto L1
  2. IfFalse c goto L1
  3. IfFalse d goto L1
  4. R2=1
  5. goto L2
  6. L1:R2=0
  7. L2:R1=R2

All above and some other optimizations are implemented in my compiler. For instance, variable-length arrays are supported in my compiler.

The next issue is that how we store and visit the arrays. There are two types to store the arrays:

- Just using continuous space to store the arrays and calculate the offset of the element.
- using extra space to store pointers which are pointed to the position of the element or pointer.

These two ways are both OK in general, but in some cases all of these should be used. For example, the second way need to be used in the following case:

```c
int i;
int* a[11];
for(i=0;i<11;++i)
	a[i]=malloc(11*sizeof(int));
a[0][1]=2;
```

Because a[0], a[1], a[2]...a[10] may not be stored in the successive space so you should caculate the offset for two times. Also it's useful in order to support the pointer of the array, such like

```c
typedef int intarray[11];
int a[11][11];
intarray *p=&(a[0]);
```

Another problem is that how to do register allocation for global variables. The easiest way to solve this problem is not to assign register for any global variables. This will cost a lot which makes a lot of loads and stores. However, we can use some better ideals. We can treat global variables as local variables to do register allocation. And we must be careful that the global variables need to be loaded at the beginning and reloaded after the function calls which may modify their values. Also when we meet function calls we must store the values to update the memory. In addition, providing some registers only for global variables is a good ideal because in some cases we don't use up all the registers. These means conduct well in the test of the eight queens.

Finally my assembly code looks like this:

- .data
  - args: store arguments when the number of arguments exceed four.
  - disp: store global variables here.

    - gc_sp_limit: store the stack pointer and maybe used in garbage collection.
    - stores string literals here.
- .text
    - Functions such as main.
    - speicial work for global variables.
    - Function definition using mips such as printf and so on.

For one function definited in C programming language, the assembly code looks like this:

- function-name label
- adjust the stack pointer to store registers
- main body
- adjust the stack pointer to resume registers

This is the test code(eight queens):

```
#include <stdio.h>
int N = 8;
int row[8], col[8];
int d[2][8 + 8 - 1];
int ans;
void search(int c) {
  if (c == N) {
    ++ans;
  }     else {
    int r;
    for (r = 0; r < N; r++) {
      if (row[r] == 0
        && d[0][r+c] == 0
        && d[1][r+N-1-c] == 0) {
        row[r] = d[0][r+c] = d[1][r+N-1-c] =
              1;
        col[c] = r;
        search(c+1);
        row[r] = d[0][r+c] = d[1][r+N-1-c] =
              0;
      }
    }
  }
}
int main() {
  search(0);
  printf("%d\n",ans);
  return 0;
}
//92
```

It runs about 49W instructions for this code.

For instance, the main() function translates into assembly code like this:

```
main:
          sw $sp, gc_sp_limit
          la $gp, disp
          la $v1, args
          addiu $sp, $sp, -80
          sw $ra, 76($sp)
          sw $t0, 4($sp)
          jal PROGRAM
          li $t0, 0
          move $a0, $t0
          jal search
     la $t1, L34
          move $a0, $t1
          lw $k0, 32($gp) # load for spilling
          move $a1, $k0
          jal printf
          move $t0, $v0
          li $v0, 0
          j __main
__main:
          lw $t0, 4($sp)
          lw $ra, 76($sp)
          addiu $sp, $sp, 80
          jr $ra
```

Acknowledgments

Logical, Mathematics, Computer language

References

[1] Matthew Might, David Darais, and Daniel Spiewak. An easy way to do general parsing. I*n Parsing with Derivatives(ICFP 2011)* .