

编译原理 Lab4

202220013 徐简

项目环境:

1. GNU LINUX Realease: Ubuntu
2. GCC version: 7.5.0
3. GNU Flex version: 2.6.4
4. GNU Bison version: 3.0.4

实现的功能

目标代码生成

- 完成了必做要求
- 对手册提供的测试样例进行了测试模拟

代码实现

实验4的代码，除了debug时修改了之前的一些漏洞之外，集中在 `mips.c`

- 在语义分析部分，将 `read` 函数和 `write` 函数初始化进符号表

```
void initTable()
{
    for (int i = 0; i < HASHSIZE + 1; ++i) {
        hashTable[i] = NULL;
    }
...
    item->name = "read";
...
    insert(item);
...
    item->name = "write";
...
    insert(item);
}
```

- 在目标代码生成部分，首先将头部信息输出到文件中

```
#define PRECODE "..."
FILE* f = fopen(filename, "w");
fprintf(f, PRECODE);
```

- 指令选择部分，遍历中间代码，进行模式匹配

```

extern InterCodes* head;
InterCodes* p = head->next;
while (p) {
    p = p->next;
    ...
}
fclose(f);

```

- 寄存器分配采用了朴素寄存器分配算法

将所有的变量或者临时变量都放在内存里，每次生成代码的时候，将将变量读入寄存器中，计算完成后，再将结果写回内存。

- 为了实现这样的机制，引入了记录数据偏移量的数组 `varOffset`，根据类型遍历中间代码，得到 `varOffset`。

```

//初始化
int n = varCount + tmpCount - 1;
varOffset = (int*)malloc(sizeof(int) * n);
for (int i = 0; i < n; i++) {
    varOffset[i] = -1;
}
//计算VarOffset

int allocVar(InterCodes* begin)
{
    int fpoffset = 4;
    InterCodes* p = begin;
    while (p && p->code.kind != FUNCTION) {
        ...
        ...
        p = p->next;
    }
    return fpoffset;
}

```

- 最后再遍历中间代码完成翻译，用偏移量完成正确的变量读取和写入

测试

- 手册测试样例运行截图如下
- Test1.cmm

● Test2.cmm

```

Enter an integer:7
1
1
2
3
5
8
13

00000..[00440000]
: lw $a0 0($sp) # argc
: addiu $a1 $sp 4 # argv
: addiu $a2 $a1 4 # envp
: sll $v0 $a0 2
: addu $a2 $a2 $v0
: jal main
: nop
: li $v0 10
: syscall # syscall 10 (exit)
li $v0, 4
la $a0, _prompt
syscall
li $v0, 5
syscall
jr $ra
li $v0, 1
syscall
li $v0, 4
la $a0, _ret
syscall
move $v0, $0
jr $ra
addi $sp, $sp, -4
sw $fp, 0($sp)
move $fp, $sp
addi $sp, $sp, -84
li $t0, 0
sw $t0, -4($fp)
; 30: lw $t1, -4($fp)
; 31: move $t0, $t1
; 32: sw $t0, -8($fp)
; 33: li $t0, 1
; 34: sw $t0, -12($fp)
; 35: lw $t1, -12($fp)
; 36: move $t0, $t1
; 37: sw $t0, -4($fp)

R21 [$s5] = 0 [00400074] 8fc9ffffc lw $9, -4($30)
R22 [$s6] = 0 [00400078] 00094021 addu $8, $0, $9 ; 30: lw $t1, -4($fp)
R23 [$s7] = 0 [0040007c] afc8ffff8 sw $8, -8($30) ; 31: move $t0, $t1
R24 [$t8] = 0 [00400080] 34080001 ori $8, $0, 1 ; 32: sw $t0, -8($fp)
R25 [$t9] = 0 [00400084] afc8ffff4 sw $8, -12($30) ; 33: li $t0, 1
R26 [$k0] = 0 [00400088] 8fc9ffff4 lw $9, -12($30) ; 34: sw $t0, -12($fp)
R27 [$k1] = 0 [0040008c] 00094021 addu $8, $0, $9 ; 35: lw $t1, -12($fp)
; 36: move $t0, $t1
; 37: sw $t0, -4($fp)

Copyright 1990-2012, James R. Larus.
All Rights Reserved.
SDTM is distributed under a BSD license

```

● Test2.cmm

```

Enter an integer:7
5040
|


00..[00440000]
lw $a0 0($sp) # argc
addiu $a1 $sp 4 # argv
addiu $a2 $a1 4 # envp
sll $v0 $a0 2
addu $a2 $a2 $v0
jal main
nop
li $v0 10
syscall # syscall 10 (exit)
$v0, 4
$a0, _prompt
call
$v0, 5
syscall
$ra
$v0, 1
syscall
$v0, 4
$a0, _ret
syscall
move $v0, $0
$ra
addi $sp, $sp, -4
$fp, 0($sp)
move $fp, $sp
addi $sp, $sp, -40
$t0, 8($fp)
$t0, -4($fp)

R21 [$s5] = 0 [00400074] 8fc9ffffc lw $9, -4($30)
R22 [$s6] = 0 [00400078] 00094021 addu $8, $0, $9 ; 30: lw $t1, -4($fp)
R23 [$s7] = 0 [0040007c] afc8ffff8 sw $8, -8($30) ; 31: move $t0, $t1
R24 [$t8] = 0 [00400080] 34080001 ori $8, $0, 1 ; 32: sw $t0, -8($fp)
R25 [$t9] = 0 [00400084] afc8ffff4 sw $8, -12($30) ; 33: li $t0, 1
R26 [$k0] = 0 [00400088] 8fc9ffff4 lw $9, -12($30) ; 34: sw $t0, -12($fp)
R27 [$k1] = 0 [0040008c] 00094021 addu $8, $0, $9 ; 35: lw $t1, -12($fp)
; 36: move $t0, $t1
; 37: sw $t0, -4($fp)

Copyright 1990-2012, James R. Larus.
All Rights Reserved.
SDTM is distributed under a BSD license

```

使用方法

使用makefile完成编译、运行：

- `make`, `make clean`
- 测试使用命令`./parser ../Test/test1.cmm out1.s`。中间代码会被保存到当前目录的`outxxxx.s`中

实验思考

- 为了确保跑通代码生成，只完成了最基础的寄存器分配方法，效率很低，只用到了两个寄存器，而且需要大量的寄存器内存读取操作，这在实际的应用中是不允许发生的
- 时间原因，只对手册的样例进行了测试，所以不知道对于有更加复杂的数据结构和函数调用的情况下，当前的代码能否胜任