编译实习报告

黄道吉 元培学院 1600017857

概述

在这学期的编译实习课中，我实现了一个可以将MiniC翻译到RiscV64的编译器，前者是标准C语言的子集。在课程要求之外，我实现了以下的功能，在后续的部分有详细的说明

- 报错系统
- 简单优化
- 简单的语法拓展

分阶段编译器的实现

MiniC2Eeyore

本报告简要说明设计MiniC到Eeyore的编译程序的功能，设计思路和拓展。

这一部分的代码能够将单个符合MiniC的BNF文法的程序，翻译成符合Eeyore文法的程序，输入输出都在stdin/out中完成，在线评测系统中可以通过所有测试样例。翻译程序大致分为两个阶段，首先根据BNF文法在内存中构建语法树，随后深度优先遍历语法树输出对应Eeyore程序。

在课程要求之外的部分，我实现了下列的功能

- 报错系统
- 对生成代码的简单优化
- 已实现的语法拓展
    - 多行注释
    - 支持for，do-while语句

报错系统

- 报错的情况包括: 函数多次声明且参数不同，函数多次定义，同一作用域内多次定义同名变量，使用未声明变量/函数，传参检查
- 报错的语句一般形如下面，显示行号和出错变量

```
printf("error: func %s already declared with diff. # of param. in line %d, exit\n",
root->name, yylineno);
```

语义分析

语义分析阶段会识别所有token，包括运算符和表达式

- 标识符识别尽量按照C标准实现
- 标识符名，数字值由局部变量传递到语法分析阶段
- 支持单行和多行注释，忽略掉\r\t\n等符号

- 语义分析阶段报错同样显示行号和错误类别

变量表

变量表被简单组织为栈，用数组实现

- 每一个表项记录C中的名字，Eeyore中下标，函数参数个数，变量类型和深度
- 深度是全局变量，初始值为0
  - 每遇到一个"{"加一，并标记现在变量表的位置
  - 遇到"}"退栈，将所有这个作用域内定义的变量删除，只需要简单的将栈顶定位之前记录的位置，深度减一
  - 定义变量时，从变量表的栈顶向下寻找，如果找到同作用域内的同名变量，报错退出，否则新增一个表项
  - 查找变量时，从变量表的栈顶向下寻找，找到最近作用域的变量。因为退栈机制，所以变量表中变量深度一定递增，所以不会找到其他同深度的同名变量(e.g.{int a;}{int a;}，处理第二个a时，第一个a已经退栈)

节点设计

节点定义在minic.h文件中

- children数组存储子节点，next数组存储兄弟节点(e.g.下一条语句)
- NODE_TYPE, STMT_TYPE, EXP_TYPE, OP_TYPE分别标注节点的相应信息
  - 确定节点类型可以通过node -> stmt / exp -> op确定
  - *_type的具体取值在minic.h中，取值统一命名，如双目运算符+的节点类型为

```
node.node_type = NODE_EXP
node.exp_type = EXP_BINOP
node.op_type = OP_ADD
```

- op_type中运算符的顺序和op_table一致，方便将op_type作为下标查表输出，减少编码困难
- 类似变量表，存储节点名，Eeyore中下标，label下标，函数参数，数组大小，数字的值
  - 后四个变量分别在不同类型节点出现，合并为一个field
  - 合并操作由#define 实现...保持编码明确...也是偷懒

节点组织

表达式由表达式树实现，算术符号优先级在.y文件头有定义

语句节点的组织类似以下例子

```
int foo(int a, int b){
    if(a == 0){
        // A
    }else{
        // B
        while(b == 0){
            // C
        }
```

```
        // D
    }
    // E
    return 0;
}
```

节点的组织类似

- foo
  - foo.children[1]: a
    - a.next: b
  - foo.children[0]: if // stmt_list as a linked list
    - if.children[0]: a == 0
    - if.children[1]: A
    - if.children[2]: B // also a linked list
      - B.next: while
        - while.children[0]: b == 0
        - while.children[1]: C
      - while.next: D
      - D.next: NULL
    - if.next: E
    - E.next: return
  - foo.next: // next function or stmt or NULL

通过遍历节点树，按照编译原理课内方法翻译语句

简单优化

通过观察公开的测试样例，对下面的会产生冗余代码的部分优化，尽量减少NUM/ID -> EXP的规约过程

- a = 0, b = a, a[i] = 0
  - 对于赋值语句直接赋数字或者标识符的，通过额外走深一层语法树，直接翻译
  - 对于a = 0，由var t0 = 0, a = t0简化为a = 0
- a = 1 + 1, a = 1 + a, a = 1 + b + c
  - 对于双目运算符，用类似的方法直接带入

经过这样的优化，生成的Eeyore行数会减少，下面是一个简单的例子

```
# e.g. a = 0
t0 = 0       ==>  a = 0
a = t0
```

for do-while 循环

我参考CSAPP书中推荐的循环翻译方式，增加了对for和do-while循环的支持

- 将样例中的while语句保持语义的替换成for和do-while之后，运行结果没有变化

- 由于for循环中出现了没有分号结尾的stmt，我额外定义了stmt_wo_colon方便在for语句中使用

下面是一些例子

```
int i;
for(i = 0; i < 5; i = i + 1){
    a = 0;
}

do{
    a = 0;
    i = i + 1;
}while(i < 5);
```

```
// for
T1 = 0
l0:
var t0
t0 = T1 < 5
if t0 == 0 goto l1
T2 = 0
var t1
t1 = T1 + 1
T1 = t1
goto l0
l1:
// do-while
l2:
T2 = 0
var t2
t2 = T1 + 1
T1 = t2
var t3
t3 = T1 < 5
if t3 != 0 goto l2
```

二义性

实现的语法编译时会出现移进规约冲突的警告，但对于没有错误的C文件，处理不会出现问题

- 表达式规定优先级，对于同级运算符，规定运算顺序之后不会出现错误结果
- 对if else的二义性和没有加括号的if语句
    - 原BNF文法可以处理正确的if-else语句
    - {stmt} -> stmt保证了有没有括号对处理没有影响

工程细节

编码过程中尽力保持代码(对自己的)可读性

- 包括详细的changelog，todolist
- 变量统一命名: 下划线分割单词，除非意义明确不缩写

- 对重复使用的代码
  - 如果是批量处理，尽量使用数组/循环实现，e.g. 遍历children，op_type
  - 功能明确的部分单独编写函数，e.g. 变量表中查位置
- 利用git控制版本，方便回滚代码

EEyore2Tigger

这一部分的报告包括Eeyore到Tigger的程序的功能，拓展。

这一部分的代码能够将单个符合Eeyore的BNF文法的程序转换成符合Tigger文法的程序。除了下面列出来的功能，其他的数据结构和实现方法和翻译Minic类似。

在课程要求之外，我实现了以下的功能

- 简单优化：常量表达式消除
- 线性扫描寄存器分配算法
  - 在服务器上有几个不同的.y文件备份，线性扫描算法在文件名包含"lin"的文件中

常量表达式消除

在翻译双目运算符时，如果每一个运算数都是数字，可以翻译过程中直接计算出来，具体实现参考下面的例子

```
int get_result(int a, int b, Op_Type op){
    int res = 0;
    switch(op){
        case OP_EQ:{
            res = a == b;
            break;
        }
        // more cases
    }
    return res;
}
```

线性寄存器分配

我将每一个语句看作一个基本块，记录每一条语句上活跃的变量

- 预处理
  - 构建数据流：主要是扫描if和goto语句，找到对应的下一/二条语句；其他语句的下一条语句默认是下一行语句
  - 活性分析：记录每一条语句的use和define
    - 对数组的每一次访问都看作使用整个数组
    - 按照之前构造的数据流倒序分析活性，对每一个函数的语句迭代直到收敛
- 分配寄存器：每一个变量对应一个寄存器
  - 从前向后扫描每一条语句，如果有变量没有分配寄存器，分配一个空缺的寄存器，没有空缺的，踢掉最后活跃时间最晚的一个变量
  - 变量使用 *不* 按照RISC规范，所有寄存器都看作caller saved，调用函数时保存活跃的变量，返回后写回寄存器

Tigger2RISCV

这一阶段的工作主要是逐行翻译tigger代码，生成对应的riscv64代码

- 其间有遇到栈空间过大超过常数数据范围的情况，通过特判常数范围解决

各阶段整合

下面是一些整合各个阶段的编译器的方法

- bash脚本：第一个参数为输入文件名，第二个为输出文件名

```
eeyore < $0 > tmp.eyr
tigger < tmp.eyr > tmp.tgr
riscv < tmp.tgr > $1
rm ./tmp.eyr
rm ./tmp.tgr
```

- 可执行文件包装上述脚本

```
system(/* bash script */);
```

测试

我的实现在计测平台上分别取得了100/98/98的分数

- Eeyore2Tigger两个没有通过的样例是sort5和sort7
    - 在Minic2Eeyore阶段曾经出现过代码不识别\r的问题，也是在sort*的样例出现问题，不清楚是这个阶段的问题还是上一个阶段遗留的问题

另外，为了在提交前检查代码准确性，我设计了一些样例观察编译器的行为

- 出于节省篇幅，我压缩了下面代码中的部分换行。
- 下面代码主要用来检查基本算术表达式，分级符号表，数组做参数和内置函数

```
int test_arithmic(int a, int b, int c){
    a=b+c; a=b-c; a=b*c; a=b/c;
    a = -b; a = !b;
    a = b && c; a = b || c; a = b > c; a = b < c;
    a = b != c; a = b % c; a = b == c;
    return a;
}

int test_vtb(){ // vtb: var table
    int a;
    a = 0;
    if(a == 0){ int a; a = 0; }
    a = 0;
    if(a == 0){ int a; a = 0
```

```
    }else{ int a; a = 0;
    }
    a = 0;
    while(a == 0){ int a; a = 0; }
    a = 0;
    return a;
}

int test_array(int a[5]){
    a[0] = 1; int b; b = a[0];
    return b;
}

int test_buildin_func(){
    int b;
    b = getint(); b = putint(b); b = getchar(); b = putchar(b);
    return 0;
}
```

我另外比较了riscv-gcc和我的编译器编译同一段c代码的结果，将前者作为标准可以看出我的实现的不足的地方

- 活用x0寄存器：gcc的结果在保存0值时利用了x0/zero寄存器，但我没有做特判
    - 这个问题在eeyore2tigger阶段出现，或许可以在tigger2risc阶段通过窥孔优化解决这个问题，而不是简单逐行翻译
- 维护栈首尾指针：gcc的结果维护了s0作为栈顶指针，这样对栈顶变量的查找可以更加方便，也能绕过常数不得超过2048的限制
- 使用灵活的指令集：gcc使用了sext.w, addiw等(伪)指令生成代码，也使用了多样的跳转(伪)指令
- 常量传播：gcc对常量传播做的非常好，我在minic2eeyore, eeyore2tigger阶段都做了一些简单的常量传播和常量表达式消除，但囿于常量数值范围的限制，在riscv里面实现成先加载常数到寄存器中，再进行算术运算的形式
    - 可以考虑特判范围解决

下面是简单的示例

```
# asm from gcc
addi    sp,sp,-2032
sd  ra,2024(sp)
sd  s0,2016(sp)
addi    s0,sp,2032
sw  zero,-28(s0)
sw  zero,-24(s0)
sw  zero,-20(s0)
j   .L2


addiw   a5,a5,1
```

```
# asm from me
li s11,-2064
add sp,sp,s11
```

```
li s11,2060
add s11,s11,sp
sw ra,0(s11)
lw t0,2008(sp)
li t0,0
sw t0,2008(sp)
lw t0,2004(sp)
li t0,0
sw t0,2004(sp)
lw t0,2000(sp)
li t0,0
sw t0,2000(sp)



li s11,1
add t0,t1,s11
```

- 此外，我构建了一些循环样例观察到gcc在O3的情况下会进行
  - 死代码消除
  - 循环展开，乃至计算简单的循环结果，如下例

```
# c++: for(int j = 0; j < 10000; j += 1){i += 1;}
li      s1,8192
addi    s1,s1,1808
add     s1,a0,s1
```

这些都是进一步拓展我的编译器的可行方向。

错误和改正

在实现线性扫描寄存器分配算法时，我参考了作者的原文和课上提供的材料，但一开始的实现中误认为线性扫描算法可以on-the-fly分配寄存器，只能通过90+的样例。

- 这种实现只有在变量数目过多，并且有使用到很久之前的变量时才会出现错误，自己构造的样例没有覆盖到这些情况
- 在回顾了算法之后我发现了这个错误，把代码修正成了现在的样子

在翻译到riscv后，一度增加了两个错误，是因为在设计栈时没有节约空间，出现栈大小超过2048的情况

- 这样对sp的增减操作超过riscv中11位临时数的表示范围
- 在特判栈大小之后解决了问题

总结

这个学期的编译实习课程是任务量不小的一门课程，它的学习曲线一开始十分陡峭，即便是对于已经学过编译原理的我来说(或许是因为已经忘记了正课内容)。但在入门了flex&bison之后，剩下的内容就不存在工程上的难题了，只有如何正确的理解和实现翻译算法而已

- 或许在课程开端有系统的介绍flex和bison的使用方法会更加新手友好一些。这个学期有同学在课程群内分享了相关的书，我在其中学习了很多具体的使用方法（e.g. 运算符优先级，自定义变量类型(yystype)），可能把类似的书目放到课程网站上会好一点。

- 但这也不单单是编译实习可以改进的地方，我的编译技术正课上完全略过了flex的部分。在这里，正课和实习课内容似乎有脱节。
- 另一个小的地方是内置函数的部分，在eeyore和tigger模拟器上确实有原生支持，但到riscv那里是需要自己编译一个lib链接过去的。会产生的问题包括 1. 如何编译一个lib 2. 内置函数名字是什么 e.g. f_putint or putint。但这些都可以通过简单的试错解决

这个学期中我也受任务驱动重温了编译原理乃至ICS的内容，也因为需要分配寄存器去读了线性分配寄存器算法的原文。这是很有收获的一个学期，即便对于专业方向和编译无关的我来说，在这个学期得到的工程经验将是能够应用到未来任何需要编写代码的场景的。单人维护数千行代码，设计数据结构，差错，重构和优化的经验都是本科以来任务最艰巨也是最有成就感的。

在这里我要再次感谢刘老师一个学期以来详尽负责的教学。在学习曲线最陡峭的地方，老师和每一个人面谈解决实习中遇到的问题。同样还需要感谢尽责细心的助教师兄/师姐，他们在面测时详细的检查代码，讨论我设计的数据结构可行性，提出改进的方案，在课下也在线不厌其烦的解决繁杂的问题。