从 MiniC 到 RiscV

编译实习课程报告

1500012739 特古斯

2018 年 1 月 12 日

目录

1 综述

这学期的编译实习的任务是从 MiniC(简化版的 C)→ 中间代码 Eeyore→ 中间代码 Tigger→RiscV 伪汇编，每个阶段都有一定的检查以确保此阶段的正确性和进度的进展。从零开始做一个编译器的确是非常有成就感的一件事，我也在其中很大的提升了自己的代码能力。

这学期的编译实习与以往不同，第一次采用 MiniC 作为课程的语言平台。相对于原来的 MiniJava，MiniC 的语法更加简单，剔除了面向对象的内容，简化了许多的工作。但与此同时，课程设计者在课程设计也中有些缺陷和考虑欠妥的地方，经过今年的实践，我也对课程有一些建设性的提议。

2 实验平台

在整个实现过程中我全部采用了 Lex+Yacc 工具链进行翻译，因为翻译过程遵循同样的模式，修改起来也比较方便。我也曾经尝试过用 Python 用正则表达式匹配进行中间代码优化，不过后来由于过于繁琐还是直接用 Yacc 生成的代码优化了。

Yacc 是一个采用 LALR(1) 语法分析的工具，输入 BNF 并添加语义规则就可以生成需要的代码，一般都要与 Lex 结合。Lex 则是一个用正则表达式分析语言，把每个符号分解为 Token 进一步输入 Yacc。

下面是对每个部分方法详述。

3 MiniC

这部分的任务是分析 MiniC 的语法并翻译成 Eeyore。MiniC 的语法比 C 简单了许多，不过原来的文档提供的 BNF 有许多漏洞，我也在基础之上加上了许多的小优化，下面是我实现的 BNF。

3.1 BNF

⟨*Goal*⟩ `::= DefnDeclList*`

⟨*DefnDeclList*⟩ `::= (VarDefn|FuncDefn|FuncDecl)*`

⟨*VarDefn*⟩ `::= Type Identifier ';'`
`| Type Identifier '['`⟨*INTEGER*⟩`']' ';'`
`| Type Identifier '=' Expression ';' //`声明时赋值
`| Type Identifier '['`⟨*INTEGER*⟩`']' '=' Expression ';'`

⟨*VarDecl*⟩ `::= Type Identifier`
`| Type Identifier'['`⟨*INTEGER*⟩`?']'`

⟨*FuncDefn*⟩ ::= `Type Identifier '(' ( VarDecl ( ',' VarDecl )* )? ')' '{' (FuncDecl | Statement)* '}'`

⟨*FuncDecl*⟩ ::= `Type Identifier '(' ( VarDecl (',' VarDecl)*)?')' ';'`

⟨*Type*⟩ ::= `'int'`

⟨*Statement*⟩ ::= `'{' (Statement)* '}'`
| `'if' '(' Expression ')' Statement ('else' Statement)?`
| `'while' '(' Expression ')' Statement`
| `Identifier '=' Expression ';'`
| `Identifier '[' Expression ']' '=' Expression ';'`
| `Expression // 无左值表达式`
| `'return' Expression ';'`

⟨*Expression*⟩ ::= `Expression ( '+' | '-' | '*' | '/' | '%' ) Expression`
| `Expression ( '&&' | '||' | '<' | '==' | '>' | '!=' ) Expression`
| `Expression '[' Expression ']'`
| ⟨*INTEGER*⟩
| `Identifier`
| `( '!' | '-' ) Expression`
| `Identifier '(' (Expression (',' Expression)*)? ')' // 参数支持表达式`
| `'(' Expression ')'`

⟨*Identifier*⟩ ::= ⟨*IDENTIFIER*⟩

3.2 特点

- **增加了对新语法的支持**

  与原来的 BNF 比较，增加了

  - 无返回值表达式，如直接调用函数
  - 表达式传入任何可传入的位置（如函数参数）
  - 声明时直接赋值

  同时支持在函数中声明与定义函数，作用域为最近的域。

- **对代码进行检查**

  - 未声明变量或函数直接报错，输出出错行数并停止翻译
  - 重复定义变量采用第一次定义的变量
  - 函数参数数量不匹配直接报错，输出出错行数
  - 语法错误直接报错，输出出错行数

3.3 实现细节

```
typedef struct environment
  {
    struct environment* pre;//前向指针
    map<string,string> symTable;//符号表
    map<string,string> declList;//已声明函数
    map<string,string> funcPara;//函数参数
    int varCnt;
  } Env;
```

Listing 1: 环境结构

整个过程使用了 On-the-fly 动态生成的方法，同时考虑到并不需要特别多的依赖关系，每个非终结符号都可以用 string 代表它所代表的变量，整个过程非常的简洁。因为这个原因，符号表中只需存储 MiniC 中每个变量名和对应的 eeyore 临时变量名，用一个 map 表示符号表即可。而不同的环境用一个链表串起来，在每个环境中存储的信息只有符号表、已声明的变量集合（用来判断是否有未声明的变量）和函数参数。

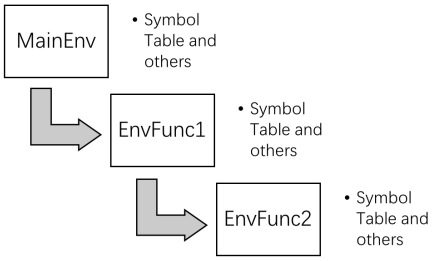

图 1: 环境示意图

3.4 编写中遇到的问题

首先是没有理解 Eeyore 中变量的命名方式，一开始直接用 T 开头后面加乱七八糟的字母来方便调试，结果一直跑不通…才发现后面必须是标准的数字。

还有就是两方符号的识别与命名：每次识别到新的符号，直接把所有符号先转换为内部的独有变量，不过这样也带来了一些问题：对 Identifier 来说没有上下文信息，不好判断是在什么地方进行的，因此需要全局变量来辅助，这样也造成了一些结构上比较丑陋的地方，不过总体来说利大于弊。

为了翻译上的方便，我一开始使用了大量的临时变量存储中间结果，这也给后来的优化埋下了很多伏笔。

3.5 实现过程中的小 Tips

因为 On-the-fly 生成时不存在显式的语法树，有些必须的辅助信息（如判断是否为函数参数）需要作为继承属性输入到下层的非终结符号中，而我把每个符号的类型都设置成了 string……因此我就用了各种方法

对栈中进行传递，比如全局变量和临时的环境。

总体来看，在 MiniC 翻译中用很简洁的 On-the-fly 生成取得了很好的效果，对代码错误也有一定的自动改正能力，不过生成的 Eeyore 代码还有很多的冗余，这将在之后的中间代码过程中逐步进行优化。

4 Eeyore

从 Eeyore 翻译到更底层的 Tigger 是整个编译器的核心。在这个过程中，我们需要进行 Eeyore 代码的翻译和优化，并进行寄存器分配，最终生成 Tigger 代码。

这阶段最大的挑战是要初步分析出每个函数的栈空间、将无限的变量分配到有限的寄存器和代码更细粒度的分解。

4.1 BNF

因为需要能在 Eeyore 模拟器上正确运行，这里的 BNF 和原来没有什么变化。

⟨*Declaration*⟩ ::= 'var' ⟨*INTEGER*⟩? Variable

⟨*FunctionDecl*⟩ ::= Function '[' ⟨*INTEGER*⟩ ']' '\n' ((Expression | Declaration)'\n')* 'end'
Function

⟨*RightValue*⟩ ::= Variable | ⟨*INTEGER*⟩

⟨*Expression*⟩ ::= Variable '=' RightValue OP2 RightValue
| Variable '=' OP1 RightValue
| Variable '=' RightValue
| Variable '[' RightValue ']' = RightValue
| Variable = Variable '[' RightValue ']'
| 'if' RightValue LogicalOP RightValue 'goto' Label
| 'goto' Label
| Label ':'
| 'param' RightValue
| Variable '=' 'call' Function
| 'return' RightValue

⟨*Identifier*⟩ ::= ⟨*IDENTIFIER*⟩

⟨*Variable*⟩ ::= ⟨*VARIABLE*⟩

⟨*Label*⟩ ::= ⟨*LABEL*⟩

⟨*Function*⟩ ::= ⟨*FUNCTION*⟩

4.2 特点

4.3 实现

相对于原来复杂的语法树，Eeyore 代码没有显式的多层嵌套，只有在分支的时候有两种分叉。在分析 Eeyore 代码时直接将代码存为链式结构，之后再对整个代码数组进行分析即可。

生成内部表示代码的数组之后，对每一个函数进行活性分析，产生每个变量的活跃区间，然后对代码进行优化，得到优化后的代码后用线性扫描进行寄存器分配，最终生成 Tigger 代码。

```
struct variable{
  int id,st,ed,isGlobal,isArray,glbID;\\从Eeyore文件中分析得出
  int _pos,_reg,_mem,_active;\\在活性分析中用到
  int pos,reg,mem,active;\\在代码生成中用到
  string name;

  variable(string _name,int _id,int memID = 0):id(_id),name(_name)\\构造函数
  {
    mem = memID;isArray = 0;
    st = 63333;ed = -1;_pos = 0;_reg = 0;
    if(mem == 0) {isGlobal = 1;glbID = glbID_cnt++;}
    else isGlobal = 0;
  };
  variable() {};
};
```

Listing 2: 变量结构

```
struct block{
  int type;\\代码类型
  string arg1, arg2, arg3, arg4;\\不同参数
  vector<int> pre;\\语句的前驱
  bitset<MAXVARS> def,use,live;
  block(int _type, string _arg1 = "", string _arg2 = "", string _arg3 = "",
     string _arg4 = ""):
  type(_type),arg1(_arg1),arg2(_arg2),arg3(_arg3),arg4(_arg4) {};
};
```

Listing 3: 代码块结构

```
struct myfunction{
  string name;
```

```
  int stackSize,varCnt;\\变量数和栈空间
  myfunction(string _name,int _varCnt)
  {
    name = _name;
    varCnt = _varCnt;
    stackSize = 12;
  };
  myfunction(){};
};
```

Listing 4: 函数结构

4.4 活性分析

我做的是关于函数域中的活性分析。首先要分析出代码的结构以确定每个语句的前驱与后继，这里除了 if 有两个后继，goto 有确定的后继，其它都是确定的一个后继。这样我们就可以计算出每个语句的前驱和后继，之后从最后一条语句开始计算 live 变量（用 bitset 存储），有改变的话就把前驱放在 queue 中，用一个类似 BFS 的算法不断遍历，直到 queue 中没有语句为止，也就是收敛了。确定了每个语句的活跃变量之后，我直接粗暴地把每个变量的活跃区间设为最前面活跃到最后活跃的长度（否则会造成非常麻烦的情况，课上也讨论过这个问题）。

4.5 优化代码

总的来说这部分很多是给之前 Eeyore 填坑……我的优化步骤是：单步窥孔优化 → 复写传播 → 无用代码消除 →if 表达式优化 → 表达式计算

- 窥孔优化：前一步生成的代码有许多 $b = a, c = b$ 性质的语句；我的处理方法是是判断 b 的活跃区间是否只在这两句话，如果是的话直接删除并替换即可；
- 复写传播：如果前面有同样引用一个变量的话，直接替换掉，能给之后死代码消除提供空间
- 无用代码消除：遇到无用的变量或表达式，直接删除即可（用活跃区间判断也可）
- if 表达式优化：同样是优化前一步 Eeyore 生成的坑，原本在 if 中只有 $ifa == 0goto$ 类型的，在这里把逻辑表达引入了 if 语句之中，也节省了语句和寄存器
- 表达式计算：如果一个表达式是二元或单元常数运算，那么编译器直接算出来就可以

4.6 寄存器分配

现在大家主要都是采用的线性扫描算法，与原本的图染色相比代码好写了许多，性能根据观察也没有很大的损失，同时性能也有很大提高 ($O(n)$)。具体的算法就是根据活跃区间结束位置排序，贪心地将每个寄存器分配给相应的变量，当变量不活跃时释放寄存器，给下一个变量留出空间。对于必须溢出的变量我直接把它放到了一个固定的寄存器中，与原本的实现没有太大的损失（其实构造这么复杂的例子还是很难的）。

在 RiscV 的 23 个可用寄存器中，为了提高性能和解决奇怪的 Tigger 语法问题（Reg 和 Immediate 计算），我将三个寄存器固定了用途：

- s8 专门存储数字 4，处理数组地址问题（取值时都会乘 4）
- s9 专门存储立即数 1，做与寄存器的二元运算
- s10 专门存储立即数 2，做与寄存器的二元运算（其实可以删除因为立即数与立即数计算已经被优化了）
- s11 专门存储临时的地址，处理数组地址问题

4.7 代码生成

理论上直接把每个代码按规则翻译就好了……但是写代码的时候这时候出的 Bug 最多。给每个变量分配寄存器之后就是线性扫描代码，在扫描过程中对每一句都根据变量是否活跃的状态 store/load 相应的寄存器，主要的困难就是调用函数是对栈帧的处理。调用函数时，记录之前的 param 命令，将 caller-saved 寄存器的值 load 到内存/栈中，然后把参数传到寄存器中（顺序不能变化）。每个函数开头还需要保存 callee-saved 寄存器，为了减小开销我们都只是保存/恢复活跃的寄存器。其他的细节都在代码里，在此就不再赘述了。

4.8 优化效果

这里我使用了自己写的 qsort 和原本课程设计者提供的 wseq，分别统计优化前和优化后的 Tigger 代码长度和运行时用 translate 得出的实际指令数。

源文件	Tigger 优化前	Tigger 优化后	RiscV 优化前	RiscV 优化后	Tigger 模拟器运行时间
qsort	271	236	48608	48490	1.49->1.07
wseq	459	434	-	-	-

wseq 测试量过大，没有实际在模拟器上测试。RiscV 实际运行指令数相对改进不大的缘故，应该是运行时环境占了大多数指令（尤其是系统调用的 printf 函数什么的）。总体来看优化还是能起到比较显著的效果。

5 Tigger

从 Tigger 到 RiscV 相对来说比较简单，根据表格提供的代码一条一条翻译过去就好。有一些命令没有提供，我填补之后的结果附在了后面。

在翻译过程中我发现用移位操作代替乘法能够减小指令强度，尤其是对于大量对数组地址乘 4. 于是我直接采用了 slli 左移两位代替。不过现在我对 RiscV 的基本指令还不是特别理解，伪指令在手册中并没有特别多的说明，还需要自己发掘基本指令的关系，希望以后的课程上可以稍微增多讲解。

6 测试

编译器完成后麻烦的是各种测试…首先课程设计者提供了一部分测试代码，不过有一些问题：

- 首先是代码规范和原来的不一致，在修改了 BNF 之后才能够正确运行
- 样例数据量太大，wseq 在 Tigger 模拟器上跑需要数十分钟
- 没有标准的 Eeyore 和 Tigger 代码作参考，给调试造成了一些不便

我使用了自己写的一版 qsort 进行初步评测，代码如下。在测试中遇到的问题在解决之后很多都不记得了……一般都是自己的小 Bug 或功能不完善造成的。

```
int a[10000];
int c;
int getint();
int putint(int x);
int display(int array[100], int n)
{
    int i;
    int o;
    i = 1;
    while (i < n + 1) {
      int x;
      x = array[i];
      o = putint(x);
      i = i + 1;
    }
    return 1;
}
```

```
int quicksort(int array[100], int maxlen, int begin, int end)
{
    int i;
    int j;
    if(begin < end)
    {
        i = begin + 1;
        j = end;
        while(i < j)
        {
            if(array[i] > array[begin])
            {
                int t;
                t = array[i];
                array[i] = array[j];
                array[j] = t;
                j = j - 1;
            }
            else
            {
                i = i + 1;
            }
        }
        if(array[i] > array[begin] - 1)
        {
            i = i - 1;
        }

        int t;
        t = array[begin];
        array[begin] = array[i];
        array[i] = t;

        int o;
        o = quicksort(array, maxlen, begin, i);
        o = quicksort(array, maxlen, j, end);

        return 1;
    }
```

```
    else {
      return 1;
    }
}

int main()
{
    int n;
    int array[10000];
    n = getint();
    int i;
    i = 1;
    while (i < n + 1) {
      array[i] = getint();
      i = i + 1;
    }

    int o;
    o = display(array, n);

    int st;
    st = 1;
    int ed;
    ed = n;
    o = quicksort(array, n, st, ed);
    o = display(array, n);
    return 0;
}
```

6.1 对测试样例的建议

- 提供充足的小样例我在一开始做的时候完全是一头雾水，lex 和 yacc 如何使用都不清楚，而且要把完整的语法实现出来再调试都非常困难。所以提供几个个小样例和对应的 Eeyore 代码在模拟器中对拍，让同学入门会比较方便。
- 全面完善样例在自己测试过程中突然发现了一个问题——机测上能通过的代码本地却有问题：纠结了一番发现是函数传递数组的问题，这个特性在原来的 MiniC 中存在，然而机测中并没有发现……我的建议是直接把 sort 里面的数组放在函数中传递。还有多寄存器的问题现在的样例做的比较简陋（直接用 26 个赋值语句解决问题）。

7 对课程的建议

毕竟是第一年的课程改革，这学期的课程的确有一些不完善之处。但总体来说在老师和助教的帮助下，整个过程还是非常顺利完成了。

7.1 关于开发环境

现在很大的问题是测试与开发环境的融合。辛苦助教了一学期用原始的 FTP+ 脚本 + 邮件进行测试，对教学双方都不是很方便。每节课很多时间都是浪费在同学与助教交流如何测试和测试结果之类的问题，消耗了大量的时间。

一开始就碰到了 Mac 和 Linux 环境下 makefile 的问题，Mac 下正常运行的 Linux 却不通过，后来研究是编译选项的问题，然后又在文件输入输出统一上浪费了很多时间与精力。