实验4 中间代码生成

202011081001 卢皓宇 2022年12月9日

实验要求

在词法分析、语法分析和语义分析程序的基础上，将C--源代码翻译为中间代码。

要求将中间代码输出成**线性结构（三地址代码）**，使用虚拟机小程序（附录B）来测试中间代码的运行结果。

基本要求

1. 对于正确的测试样例，输出可以在虚拟机上运行的中间代码文件。
2. 在程序可以生成正确的中间代码（“正确”是指该中间代码在虚拟机小程序上运行结果正确）的前提下，效率也是最终评分的关键因素。

附加要求

1. 修改前面对C--源代码的假设2和3，使源代码中：
    1. 可以出现结构体类型的变量（但不会有结构体变量之间直接赋值）。
    2. 结构体类型的变量可以作为函数的参数（但函数不会返回结构体类型的值）。
2. 修改前面对C--源代码的假设2和3，使源代码中：
    1. 一维数组类型的变量可以作为函数参数（但函数不会返回一维数组类型的值）。
    2. 可以出现高维数组类型的变量（但高维数组类型的变量不会作为函数的参数或返回类值）。

实验分工

- 卢皓宇：基于样例完善先前的语义分析过程，基于样例2和附加样例2的要求实现代码
- 姜潮涌：确定需要的翻译模式，并基于样例1和附加样例1的要求实现代码
- 段欣然：基于样例协助完善先前的语义分析过程，设计优化方法，减少“空label”编写语法树查找函数 `searchTree()` 以实现在调用变量时确认其是否为数组变量、是否需要计算偏移量/添加*&等运算符等
- 杜隆清：编写样例，测试代码鲁棒性并提出修改建议

实验环境

- Bison: v3.5.1
- Flex: v2.6.4
- 虚拟机程序irsim
- 本地程序编写与测试
    - Windows Subsystem for Linux 2: Ubuntu 20.04 LTS
    - gcc: 9.4.0
- 云主机测试
    - BNUCLOUD云主机: Ubuntu 18.04 LTS
    - gcc: 7.5.0

实验设计

中间代码的表示

我们选用树形表示中间代码。具体而言，我们采用DFS遍历语法树，也就是对整个代码进行翻译，并将生成中间代码所必须的变量符号存入语法树的对应结点中。

与以链表为代表的线性IR相比较，在我们看来，树形IR的组织依赖于语法树的构造，而不需要单独存储完整的中间代码

- 在通过树形结构对上下文进行组织的过程中，包括声明变量、函数以及运算符等在内的符号已经通过child和bro指针连接起来了
- 在此基础上，我们要做的就是在语法树结点中完善三地址码表示所需要的、在源代码中不会显式书写的**中间变量**、**跳转标签**等
- 这样，语法树就包含了所有构建中间代码所需要的所有必要信息，接下来只要对不同类型的结点制定不同的输出中间代码的规则，然后遍历语法树输出即可

为了方便调试、降低中间代码生成与语法语义分析过程的耦合度，我们选择在完成词法、语法和语义分析，并确认没有任何错误后，再执行中间代码生成过程。

我们以最典型的Exp部分产生式的中间代码生成为例，介绍我们翻译的思路

- Exp这部分的语法树递归调用比较常见，总体来说叶结点就包括ID、INT、FLOAT三类，所以对这3个产生式自然是直接翻译（直接将变量符号名/立即数存入返回值`tmp`中）；单目运算符中，负数形式也是一样

```
if(head->child->bro == NULL) {
    if(!strcmp(head->child->name, "ID")) {
        // fprintf(f, "%s ", head->child->id);
        tmp = (char*)malloc(sizeof(head->child->id));
        sprintf(tmp, "%s", head->child->id);
        // return head->child->id;
    }
    else if(!strcmp(head->child->name, "INT")) {
        // fprintf(f, "#%d ", head->child->intValue);
        tmp = (char*)malloc(sizeof(head->child->intValue+3));
        sprintf(tmp, "#%d", head->child->intValue);
    }
}
else if(head->child->bro->bro == NULL) {
    //没有写NOT Exp
    if(!strcmp(head->child->name, "MINUS")) {
        if(!strcmp(head->child->bro->child->name, "INT")) {
            // fprintf(f, "#-%d ", head->child->bro->child->intValue);
            tmp = (char*)malloc(sizeof(head->child->bro->child->intValue+3));
            sprintf(tmp, "#-%d", head->child->bro->child->intValue);
        }
        //else  好像也没有这种情况？
    }
}
```

- 双目运算符，则递归翻译左右两个操作数。注意`translate_Exp()`函数的返回值是char*，这个设置是为了便于确定多项式在转化为多次二元运算翻译时所需要的中间变量而设置的，以乘法为例

```
else if(!strcmp(head->child->bro->name, "STAR")) {
    char* tmp1 = translate_Exp(head->child, f);
    // fprintf(f, "* ");
    char* tmp2 = translate_Exp(head->child->bro->bro, f);
    fprintf(f, "t%d := %s * %s\n", tcnt, tmp1, tmp2);
    tmp = (char*)malloc(sizeof(tcnt) + 3);
    sprintf(tmp, "t%d", tcnt);
    strcpy(head->id, tmp);
    tcnt++;
}
```

也就是递归翻译左右两个操作数、并将其返回值存入临时变量`tmp1`和`tmp2`中，将两者相乘计算结果存入中间变量`t(cnt)`中。其中`cnt`是一个全局的整型变量，每当需要设置中间变量时就可调用其作为中间变量编号；整个`translate_Exp()`函数的返回值即此处的`tmp`。也就是每一个Exp翻译的结果变量符号

- 有参函数和无参函数的调用本身比较简单粗暴，就是先“后序”遍历Args产生式，以倒序打印传递的参数；再更新中间变量编号，将调用函数计算结果存入中间变量即可

```
else if(head->child->bro) { // Exp: ID LP Args RP
    translate_Args(head->child->bro->bro, f);

    fprintf(f, "t%d := CALL %s\n", tcnt, head->child->id);
    tmp = (char*)malloc(sizeof(tcnt) + 3);
    sprintf(tmp, "t%d", tcnt);
    tcnt++;
}

void translate_Args(struct node* head, FILE *f) {
    if(head->child->bro) { // Args : Exp COMMA Args
        translate_Args(head->child->bro->bro, f);
    }
    char* tmp1 = translate_Exp(head->child, f);
    // fprintf(f, "yes%s\n", tmp1);
    if(searchTree(root, tmp1)) {
        // printf("%d\n", cnt);
        // for(int i = 0; i < cnt ; i++) {
        //  printf("%d\n", arr[i]);
        // }

        fprintf(f, "ARG &%s\n", tmp1);
    }
    else {
        fprintf(f, "ARG %s\n", tmp1);
    }
    cnt = 0, ed = 0;  // 下次使用search前需要做
}

void translate_Args(struct node* head, FILE *f) {
    if(head->child->bro) { // Args : Exp COMMA Args
        translate_Args(head->child->bro->bro, f);
    }
    char* tmp1 = translate_Exp(head->child, f);
    // fprintf(f, "yes%s\n", tmp1);
```

```
    if(searchTree(root, tmp1)) {
        fprintf(f, "ARG &%s\n", tmp1);
    }
    else {
        fprintf(f, "ARG %s\n", tmp1);
    }
    cnt = 0, ed = 0;  // 下次使用search前需要做
}
```

这里出现了一个搜索语法树以确定传入参数是否为数组类型的变量的函数`searchTree()`，在后面的部分我们会详细展开，这里先不赘述。因为我们选择语法/语法分析结束后再开始生成中间代码，所以在生成中间代码时，我们不能直接搜索中间代码（#include的问题真的挺麻烦）。因此，语法树在此充当了符号表的功能——毕竟它也包含了代码语义的上下文信息，且经过语法/语义检查，我们也能确定原代码合法、所需要的东西一定是可以被搜到的。

> 实验四的最大意义：给实验三的代码debug~~（光是解除实验三的代码对实验四样例的报错就de了好几天~~

一些优化

由于时间紧迫且能力有限，我们最主要的效率优化来自于不为已声明的变量赋别名，这样源代码中重复调用同一变量、立即数时，就不需要另外赋一个中间变量，因而减少了instruction count。

除此之外，我们还对`if-else`和`while`的label机制做了一些优化，减少了朴素方式下可能存在的“空label”问题。主要是优化了`if-else`的翻译过程，代码如下。

```
    else if(!strcmp(head->child->name, "IF")) {
        sprintf(head->id, "label%d", r);

        fprintf(f, "IF ");
        translate_Exp(head->child->bro->bro, f);
        fprintf(f, " GOTO label%d\n", r);
        back1 = r; r += 1;
        if(head->child->bro->bro->bro->bro->bro && !strcmp(head->child->bro->bro->bro->bro->bro->child->name, "IF")) { // else if 优化
            int tmp = flagif;
            flagif = 0;
            translate_Stmt(head->child->bro->bro->bro->bro->bro->bro, f);
            fprintf(f, "GOTO label%d\n", BACK);
            flagif = tmp;
        }
        else if(head->child->bro->bro->bro->bro->bro != NULL) {
            translate_Stmt(head->child->bro->bro->bro->bro->bro->bro, f);
            fprintf(f, "GOTO label%d\n", r);
            BACK = r; r += 1;
        }
        else {
            fprintf(f, "GOTO label%d\n", r);
            BACK = r; r += 1;
        }
        fprintf(f, "LABEL label%d :\n", back1);
        translate_Stmt(head->child->bro->bro->bro->bro, f);
        if(flagif)
        fprintf(f, "LABEL label%d :\n", BACK);
```

```
28
29      }
```

主要思想为优先翻译所有`if/else if`，在所在条件下分别`go to`跳转，然后翻译`else`，这样的操作可以使进入相应程序块运行结束后跳转到该`if else`结构结束的跳转公用一个label，而`else`不使用label（包括`else if`少占用一个label）。所有最后我设置了一个回溯的写法，特点是将整个`if else`结构倒序翻译。

经过思考总结，我们不难得知这部分优化的本质可以看作特判`else if`语句并做特殊处理。

设计高维数组搜索函数

该函数的目的除了判断变量是否为数组之外，还需传出各个维度的大小，我们的方案是深搜遍历语法树直到找到对应变量名，判断是否为数组后用全局变量记录各维大小。代码如下。

```
1  int searchTree(struct node* head, char* varName) {
2      if(ed) return 0;
3      int res = 0;
4      // printf("yes%s %d\n", head->name, st);
5      if(st && !strcmp(head->name, "LB")){
6          // printf("here\n");
7          arr[cnt++] = head->bro->intValue;
8      }
9      if(st && !strcmp(head->name, "SEMI")){
10         // printf("there\n");
11         st = 0;ed = 1;
12         return res;
13     }
14     if(!head->child){
15         if(head->bro)
16             res |= searchTree(head->bro, varName);
17             return res;
18     }
19     if(head->child->type == STRING_TYPE) {
20         // printf("dxr %s %s\n", head->child->id, varName);
21         if(!strcmp(head->child->id, varName)) {
22             if(head->bro) {
23                 if(!strcmp(head->bro->name, "LB")){
24                     st = 1;
25                     if(head->bro)
26                         res |= searchTree(head->bro, varName);
27                     return 1; // 是数组
28                 }
29                 else{
30                     res |= searchTree(head->child, varName);
31                     if(head->bro)
32                         res |= searchTree(head->bro, varName);
33                     return res;
34                 }
35             }
36             else{
37                 // if(st && )
38
39                 res |= searchTree(head->child, varName);
40                 // printf("here\n");
```

```
                if(head->bro){
                    // printf("%s\n", head->bro->name);
                    res |= searchTree(head->bro, varName);
                }

                return res;
            }
        }
        else{
            res |= searchTree(head->child, varName);
            if(head->bro){
                    // printf("%s\n", head->bro->name);
                res |= searchTree(head->bro, varName);
            }
            return res;
        }
    }
    else {
        res |= searchTree(head->child, varName);
        if(head->bro)
            res |= searchTree(head->bro, varName);
    }
    // printf("yes%s\n", head->child->name);
    return res;
}
```

其中`res`标识是否为数组，`st`标识变量为数组开始记录各维大小，`ed`标识搜索结束，进行剪枝。结果保存在`arr`数组中，维度为`cnt`。

实验结果

样例1

```
./complier test1.cmm
```

```
FUNCTION main :
READ n
IF n > #0 GOTO label1
IF n < #0 GOTO label2
WRITE #0
GOTO label3
LABEL label2 :
t1 := #-1
WRITE t1
GOTO label3
LABEL label1 :
t2 := #1
WRITE t2
LABEL label3 :
RETURN #0
```

样例2

```
./compiler test2.cmm
```

```
FUNCTION fact :
PARAM n
IF n == #1 GOTO label1
t0 := n - #1
ARG t0
t1 := CALL fact
t2 := n * t1
RETURN t2
GOTO label2
LABEL label1 :
RETURN n
LABEL label2 :

FUNCTION main :
READ m
IF m > #1 GOTO label3
result := #1

GOTO label4
LABEL label3 :
ARG m
t3 := CALL fact
result := t3

LABEL label4 :
WRITE result
RETURN #0
```

附加样例1

```
./compiler ext1.cmm
```

```
FUNCTION add :
PARAM temp
t0 := *temp
t1 := temp + #4
t2 := *t1
t3 := t0 + t2
RETURN t3

FUNCTION main :
DEC opobj 8
op := &opobj
*op := #1
t4 := op + #4
*t4 := #2
ARG op
t5 := CALL add
```

```
n := t5

WRITE n
RETURN #0
```

附加样例2

```
./compiler est2.cmm
```

```
FUNCTION add :
PARAM temp
t0 := *temp
t1 := temp + #4
t2 := *t1
t3 := t0 + t2
RETURN t3

FUNCTION main :
DEC op 8
DEC r 8
i := #0
j := #0
LABEL label1 :
IF i < #2 GOTO label2
 GOTO label3
LABEL label2 :
LABEL label4 :
IF j < #2 GOTO label5
 GOTO label6
LABEL label5 :
t4 := j * #4
t5 := &op + t4
t6 := i + j
*t5 := t6

t7 := j + #1
j := t7

 GOTO label4
LABEL label6 :
t8 := i * #4
t9 := &r + t8
ARG &op
t10 := CALL add
*t9 := t10

t11 := i * #4
t12 := &r + t11
t13 := *t12
WRITE t13
t14 := i + #1
i := t14

j := #0
```

```
46
47   GOTO label1
48  LABEL label3 :
49  RETURN #0
```

实验反思

1. 树形IR感觉跟线性IR最大的区别在于，不会有专门的数据结构存储相对更完备的、构造中间代码所需要的信息，因而在生成中间代码时，其实需要更多从上下文（child-bro指针访问）判断翻译模式、计算数组/结构体偏移量等，写起来相对比较麻烦
2. 由于实验进展比较紧张，没有能较好地将生成树形IR与打印树形IR的代码分离成两个独立的过程。但实现效果没有问题，已经存入语法树的中间变量、跳转标签等也会在最后的实验5中得以利用