编译原理实验 3-火炬详解

ywy_c_asm

2023.5

特别致谢：制造了网页版 IR 虚拟机的 czj 佬　Orz

目录

1　火炬概述　2

2　这个实验要干什么？如何实现它？　2

3　语法树结点与基础数据结构　2

4　表达式的编译　4

5　函数定义的编译　8

6　变量定义的编译　9

7　控制及一般语句的编译　10

1 火炬概述

我提供的这个版本的火炬是从 lab1 的代码基础上改的，跟我自己的实现完全不同（好吧可能有的地方有点类似但是我确实是重新从头写了代码……这下就不用担心撞代码问题力），只能实现最起码的要求（**所有变量不同名，无全局变量，只有一维数组，数组不参与传参，没有结构体**），并且我还把 float 给删掉了（这应该也没啥问题，应该不会真拿 float 给你测），因此**所有的数据类型皆为 int**。如果你使用过我 lab2 的火炬，你会发现我把 lab2 的各种数据结构都删了，这个版本的代码根本没有 malloc，因为实现的极度精简，根本不需要它们。

主要代码都在 cmm_analyser.c 里，有效代码仅一两百行（就 3 个主要函数：CalcExp、CondExp、Compile）。剩下的文件基本上都直接继承自我的 lab1。make.sh 用于编译（如果出现权限错误，右键它然后在属性里选中“允许执行”，或者你手敲一遍这里面写的命令也行）。此外我的可执行文件只提供了文件输出，使用方式为：

```
./cmm [输入文件] [输出文件]
```

（另外：大家如果真的要使用火炬，记得……删一下注释……尤其是那个“By 谁谁谁”的……thx）

2 这个实验要干什么？如何实现它？

这个实验显而易见就是给你语法树，对其做和 lab2 类似的语义分析，生成中间代码。如果你仔细研究了实验要求里给出的可以使用的中间代码，你一定会发现**临时变量**是一个关键点，如果不考虑优化的话，我们甚至可以十分随便地且无限量地分配和使用它们！例如，对于下面的一个简单语句，有简单和繁琐两种不同的生成方式：

```
int a=b + c;
```

简单版中间代码（仅生成一个临时变量）：

```
a := b + c
```

繁琐版中间代码：

```
t1 := b
t2 := c
t3 := t1 + t2
a := t3
```

你会发现，尽管后一种方式繁琐，但是它好实现！我们只需要递归地对每一个子表达式都分配一个临时变量即可！这我们后面再提。总而言之，临时变量你想怎么用就怎么用，我们这里直接统一用一个整数编号 x 表示一个名称为 tx 的临时变量。

此外，还有 label 这种东西，它其实只会在 if/while 中被用到（甚至我们还不用考虑回填的问题，无脑分配标签即可！），我们也统一用一个整数编号 x 表示一个名称为 labelx 的标号。总而言之，简单地在全局维护分配了多少整数编号即可，保证每次分配的都不同就行。

我们其实主要处理 4 种语句：表达式（不含条件表达式，纯计算的）、局部变量定义、函数定义、控制语句（if/while）。它们在语法树上的定位方式和我 lab2 的火炬基本一致，总之就是在语法树结点上标记产生式，使用递归/非递归的方式解析之类的。不过，我们根本不需要处理错误，因此可以省略很多东西，比如：不需要知道变量/表达式的类型（反正也没有多维数组/结构体，全都是 int 或者 int 一维数组），不需要维护函数定义表，不需要知道调用的这个函数的参数列表长啥样，不需要解析类型……等等。因此，实验 3 远比实验 2 好写，特别是在实验 2 已经解析了语法树上的各语法成分的基础上而言。

3 语法树结点与基础数据结构

关于我的语法树结点我在实验 2 火炬里已经提了很多了，实验 3 跟它一模一样，这里重新再提一遍。

我的语法树结点是定义成这样的：

```
typedef struct _node{          //表示一个语法树上的结点（终结符/非终结符）
    int isTerminal;            //是否为终结符
    int type;                  //类型，对于终结符和非终结符有不同的意义
                               //若结点为终结符则取值为cmm.y里定义的token值
                               //（例如：INT、FLOAT……）
                               //若结点为非终结符则取值为Unterm_....
    int subtype;               //对于非终结符确定其是哪个产生式（用于lab3）
    int line;                  //语法树结点的行号（lab3中无用）
    int relop;
    //对于RELOP非终结符（>,<,>=,<=,==,!=）确定它是哪一种，词法分析时设置（lab3新引入的）
    union{                     //终结符的值
        int int_val;           //INT终结符的值
        float float_val;       //FLOAT终结符的值（在lab3中无用）
        char str_val[33];      //ID终结符或者TYPE终结符的字符串值
    };
    struct _node* next;        //右兄弟指针
    struct _node* child;       //最靠左的儿子指针
}Node;
```

当然，你的实验 1 的语法树结点可能有不太一样的定义，不过都大同小异。我用的是左儿子右兄弟表示法，child 表示第一个左儿子，next 表示右兄弟。因此，我们在解析语法树时常常需要这样访问某个儿子：

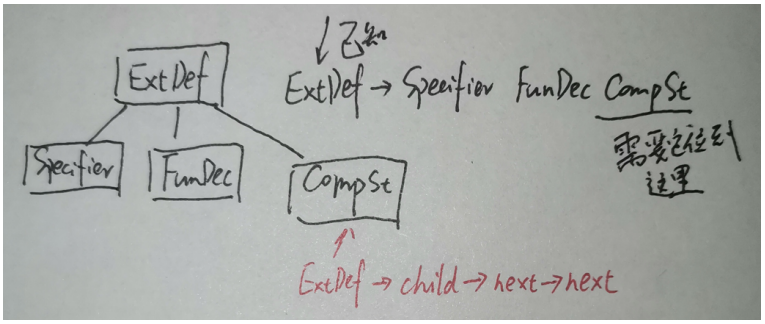

图 1: 从一个定义函数的 ExtDef 结点定位它表示函数体的 CompSt 儿子

Node 结构体中的结点属性基本上都是从实验 1 搬过来的，有些跟实验 2 甚至无关。这里唯一新增的就是 mustright，其实不把它定义在结点里也行，它就是所谓的“综合属性”，用于标记 Exp 表达式结点是否只能作为右值，在递归解析表达式子树时使用（后面会看到）。

除此之外，type 和 subtype 也很重要，它们是在 Bison 语法分析时就得到的，type 表示终结符/非终结符的类型，但是光有这个还不够（对于实验 2 来说），毕竟一个非终结符可能由多种产生式归约得到，因此 subtype 就标记了这是这个非终结符的第几个产生式。举个例子：C–文法定义了产生式 VarDec -> ID | VarDec LB INT RB，对于一个 VarDec -> VarDec LB INT RB 的结点，它的 type 为 Unterm_VarDec，subtype 就是 1（而不是 0），这样我们就可以很方便的知道各个儿子的构成。

此外，str_val 可以让我们得知这个 ID 结点或者 TYPE 结点的字符串值，从而得知类型或者名称。

关于用到的数据结构，根本没我 lab2 里那么花里胡哨，甚至都没有 String、Vector、HashMap 那些，也没有 Type/Func 这些实体数据结构，更没有 malloc 之类的玩意，我只定义了一个基于数组的简单变量表：

```
//超简单变量表
char* varnames[100];     //变量名字符串
int vars[100];           //变量的临时编号（对于数组表示其地址）
```

```
int varptr=0;

int find_var(char* str){     //在变量表里暴力查变量编号
    for(int i=0; i < varptr; i++)
        if(strcmp(str,varnames[i]) == 0)
            return vars[i];
    return 0;
}
```

这个变量表可以按照变量名存变量对应的临时变量编号，每定义一个变量（或者声明一个参数），我们都给他分配唯一的一个临时变量编号。注意：对于数组变量，这个临时变量是它的地址，后面我们再提。

此外，为了能够自由且随意地分配新的临时变量编号或者标号，我使用从 1 开始的全局的分配编号，分配时不断让它们 ++：

```
int gid=1,glabel=1; //用于分配临时变量编号和label的
```

4 表达式的编译

注意：这里所解析的表达式，并不作为条件表达式，条件表达式有特殊的处理方式，在后面 if/while 的部分阐述。当然，普通表达式也是可以作为条件的（非 0）。

在前面我们也说了，为了简化实现，我们完全可以在递归解析表达式时，对每个子表达式都分配一个用于存中间值的临时变量。不过，其实可以稍加简化一些，首先，对于已经定义的变量，直接使用它对应的临时变量即可。其次，对于整常数，也不用开一个临时变量直接存它，可以直接令该常数表达式对应这个常数，省一个临时变量。

看两个例子：

```
a=b + 1;
```

我们的程序会把它编译成这样：

```
t1 := b + #1     //在加法产生式结点处生成（当然实际上不会生成叫"b"的临时变量，只是示意）
a := t1          //在赋值产生式结点处生成
```

以及：

```
int b=-1 + (a - c);
```

我们的程序会把它编译成这样：

```
t1 := #0 - #1        //负号是一个运算符，不算是常数的一部分，"1"才是常数INT非终结符
t2 := a - c
t3 := t1 + t2
b := t3
```

在我的实现中，为了简化，一个表达式的解析结果仅简单地用一个整数表示，既可以是临时变量的编号，也可以是整常数值。你可能会问：如何区分它们呢？考虑到我们的临时变量编号都是从 1 开始的（>=1），并且整常数都是 >=0 的，因此我将整常数都取负，这样如果表达式对应的整数 x<=0，就说明是一个值为-x 的整常数，否则是一个叫 tx 的临时变量。你看，这样也不需要结构体啥的，一个整数就能全部表示，是不是特别简单？

由于我们经常要在中间代码里使用表达式的解析结果作为操作数，它可能是临时变量或者常数，我用了函数 Trans 来生成它作为操作数的字符串（当然，如果我们在一些场合下确定这个 var 就是临时变量，也可以不用 Trans 转换字符串）：

```
char* Trans(int var,int k){ //将可能为临时变量或者常量的编号var翻译为字符串
    //那个k不用细究，为了好写加的
    static char str[3][30];
    if(var <= 0)
        sprintf(str[k],"#%d",-var); //常数
    else
        sprintf(str[k],"t%d",var);  //临时变量的编号
    return str[k];
}
```

现在来看表达式的编译，我的函数 CalcExp 解析表达式结点，生成计算代码，并返回结果对应的临时变量编号或者整常数。先看前一部分：

```
int CalcExp(Node* tree,int isleft){
    //对于非条件的Exp生成计算代码并返回临时变量
    //如果是常数x，则返回-x（此时返回值<=0)
    //isleft=1则表示左值，对于数组，返回的是数组元素地址（赋值的时候对其解引用赋值）
    //除非是处理赋值表达式的左值，否则大部分情况下令isleft=0即可
    switch(tree->subtype){
    case 0:     //Exp->ID，查变量表
        return find_var(tree->child->str_val);
    case 1:     //Exp->INT
        return -tree->child->int_val;
    case 3:     //Exp->LP EXP RP
        return CalcExp(tree->child->next,0);
    case 4:     //Exp->ID LP RP
    case 5:{    //Exp->ID LP Args RP，函数调用
        char* funcname=tree->child->str_val;    //函数名
        Node* args=tree->child->next->next;
        static int arglist[30];
        int ptr=0;
        if(!args->isTerminal){  //Args非空
            //Args → Exp COMMA Args | Exp
            while(1){   //以非递归方式不断把Exp剥离下来，得到传入的各个参数表达式
                Node* exp=args->child;
                arglist[ptr++]=CalcExp(exp,0);  //解析表达式
                if(args->subtype == 0)  //Args->Exp
                    break;
                args=args->child->next->next;   ////Args → Exp COMMA Args
            }
        }
        //特殊处理输入输出函数
        if(strcmp(funcname,"read") == 0){
            int ret=gid++;
            fprintf(foutput,"READ t%d\n",ret);
            return ret;
```

```
        }else if(strcmp(funcname,"write") == 0){
            fprintf(foutput,"WRITE %s\n",Trans(arglist[0],0));
            return 0;    //write不参与运算，随便返回点啥
        }
        for(int i=ptr - 1; i >= 0; i--) //倒着传入参数
            fprintf(foutput,"ARG %s\n",Trans(arglist[i],0));
        int ret=gid++;  //存返回值的临时变量
        fprintf(foutput,"t%d := CALL %s\n",ret,funcname);
        return ret;
    }
```

首先用一个 switch 判断归约 Exp 结点的产生式。对于 Exp->ID 就直接从变量表里查这个普通变量（一定不是数组）对应的临时变量编号即可。对于 Exp->INT 直接返回常数。对于 Exp->LP Exp RP 直接剥掉外层括号往下递归。

然后就是处理函数调用的部分了，这里需要简单特殊处理一下 read()/write(x)。对于一般函数调用，使用非递归方式（这个在 lab2 的火炬里已经多次使用了）把 Args 的每个 Exp 都剥离出来，得到传入的每个参数表达式，先生成它们的计算代码，知道去哪些临时变量里找它们的值，由于我们要倒着传参，因此用一个数组暂存，然后倒过来把它们都 Arg 进去，然后对调用返回值分配一个临时变量暂存即可。

你会发现，由于他保证不会有语法错误，我们根本不需要真的知道是不是定义过了叫这个名的函数，它的参数列表是啥，返回类型是啥，我们不关心，反正一定是对的！所以也不用开函数表，simply 往“这个名字”里无脑传参然后调用即可。

然后再看访问（一维 int）数组变量元素的部分：

```
    case 6:{     //Exp->Exp LB Exp RB
        //保证是一维数组
        char* name=tree->child->child->str_val; //变量名，第一个Exp一定是Exp->ID
        int baseaddr=find_var(name);     //数组【基地址】变量编号
        int addr=gid++; //用于计算元素地址的临时变量
        int index=CalcExp(tree->child->next->next,0);    //数组下标
        fprintf(foutput,"t%d := %s * #4\n",addr,Trans(index,0));     //假定类型一定是int
        fprintf(foutput,"t%d := t%d + t%d\n",addr,addr,baseaddr);
        if(isleft)
            return addr;     //数组元素为左值，赋值时取地址即可，没必要取值
        int ret=gid++;
        fprintf(foutput,"t%d := *t%d\n",ret,addr);  //否则，用在表达式中，取地址处的元素值
        return ret;
    }
```

这里我需要强调一下，我们作出了这样的规定，如果之前定义了一个数组变量，考虑到仅对其创建一个用 DEC 分配空间的临时变量是**不能表示地址**的，因此我们每次在分配完空间后会立刻创建另一个取数组地址的临时变量，把这个存在变量表里，之后对于数组都使用这个表示数组基地址的临时变量。

所以，这里对于数组元素的访问，首先需要根据基地址（根据名字在变量表中查出来的代表这个数组的临时变量）和下标进行地址计算，即“基地址 + 下标 *4”。算出这个地址后，一般来说，就可以直接对这个地址解引用，用另一个临时变量存元素的值，作为这个表达式的值。这里的处理比较繁琐，其实可以优化很多，不过也就这样吧。

但是，有一种特殊情况，就是如果这个取数组元素的表达式用在表达式的左值，比如这样：

```
a[1]=233;
```

那么，我们就不能返回表示这个元素的值的临时变量，因为我们应该在更上层的赋值表达式处，生成一条“*addr := exp”的地址赋值指令，所以这种情况下应该返回计算数组元素地址的临时变量。为了区分出这种情况，考虑到在简单样例中没有赋值语句嵌套的情况，我们通过 CalcExp 的 isleft 参数告知这个表达式是否位于赋值语句的左侧，这样在处理数组元素的时候就可以在 isleft=1 时返回元素地址了。上面那个赋值表达式会被编译成这样：

```
t1 := #1 * #4          //在访问数组元素结点处生成
t1 := t1 + a           //在访问数组元素结点处生成
*t1 := #233            //在赋值表达式结点处生成
```

然后再看剩下的处理一般运算符的部分：

```
    case 8:{     //Exp->MINUS Exp
        int re=CalcExp(tree->child->next,0);
        int ret=gid++;       //进行运算的临时变量
        fprintf(foutput,"t%d := #0 - %s\n",ret,Trans(re,0));
        return ret;
    }
    case 10:     //Exp->Exp STAR Exp
    case 11:     //Exp->Exp DIV Exp
    case 12:     //Exp->Exp PLUS Exp
    case 13:{    //Exp->Exp MINUS Exp
        char op=(tree->subtype == 10) ? '*' : (
                (tree->subtype == 11) ? '/' : (
                (tree->subtype == 12) ? '+' : '-'));
        int a=CalcExp(tree->child,0),b=CalcExp(tree->child->next->next,0);
        int ret=gid++;  //进行运算的临时变量
        fprintf(foutput,"t%d := %s %c %s\n",ret,Trans(a,0),op,Trans(b,1));
        return ret;
    }
    case 17:{    //Exp->Exp ASSIGNOP Exp, 赋值
        int re=CalcExp(tree->child->next->next,0);
        int le=CalcExp(tree->child,1);  //isleft=1, 若左边为数组则返回存地址的临时变量
        if(tree->child->subtype == 6)    //左边是数组，返回的是地址，解引用赋值
            fprintf(foutput,"*t%d := %s\n",le,Trans(re,0));
        else
            fprintf(foutput,"t%d := %s\n",le,Trans(re,0));
        return le;  //这个无所谓，反正样例的赋值不会嵌套
    }
    default:{
        printf("Error when calc exp\n");
        exit(-1);
    }
    }
}
```

这里没啥好说的，主要注意一下赋值表达式的左值解析的时候要传入 isleft=1。

5 函数定义的编译

这部分在处理非表达式语句的主递归函数 Compile 里。对于 ExtDef，由于不可能有全局变量，它必定是函数定义。看代码：

```
void Compile(Node* tree){
    //对一般语法树递归生成中间代码
    if(tree->isTerminal)
        return;
    switch(tree->type){
    case Unterm_ExtDef:{
        //ExtDef → Specifier FunDec CompSt
        //这里仅处理函数定义，并且specifier一定为INT
        Node* fundec=tree->child->next;
        /*
            VarDec → ID
            FunDec → ID LP VarList RP | ID LP RP
            VarList → ParamDec COMMA VarList | ParamDec
            ParamDec → Specifier VarDec
        */
        Node* varlist=fundec->child->next->next;
        fprintf(foutput,"FUNCTION %s :\n",fundec->child->str_val);
        if(!varlist->isTerminal){   //varlist不为空，进行参数声明
            while(1){   //VarList → ParamDec COMMA VarList | ParamDec，遍历所有ParamDec
                Node* paramdec=varlist->child;
                int arg=gid++;  //为参数分配一个临时变量
                fprintf(foutput,"PARAM t%d\n",arg);
                Node* vardec=paramdec->child->next;
                varnames[varptr]=vardec->child->str_val;    //假定数组不作为参数，只有ID
                vars[varptr++]=arg; //连同名字存入变量表
                if(varlist->subtype == 1)   //VarList->ParamDec
                    break;
                varlist=varlist->child->next->next; //VarList → ParamDec COMMA VarList
            }
        }
        Node* compst=tree->child->next->next;
        Compile(compst);    //递归解析函数体
        return;
    }
```

这里其实非常简单，就是以非递归方式把 VarList 中每个 ParamDec 都剥下来，它一定不是数组，就是简单变量（参数），分配临时变量，用 PARAM 声明，插入变量表即可。也不用检查冲突定义啥的。然后递归解析函数体就行了。

6 变量定义的编译

这部分在 Compile 接下来处理 CompSt 的部分，因为没有全局变量，变量定义只会在 CompSt 的 DefList 里出现（不考虑函数参数声明的话）。来看 CompSt 解析的部分：

```
case Unterm_CompSt:{
    //CompSt → LC DefList StmtList RC
    /*
    StmtList → Stmt StmtList | e
    Stmt → Exp SEMI | CompSt | RETURN Exp SEMI | IF LP Exp RP Stmt
        | IF LP Exp RP Stmt ELSE Stmt | WHILE LP Exp RP Stmt
    DefList → Def DefList | e
    Def → Specifier DecList SEMI
    DecList → Dec | Dec COMMA DecList
    Dec → VarDec | VarDec ASSIGNOP Exp
    VarDec → ID | VarDec LB INT RB
    */
    Node* deflist=tree->child->next;
    Node* stmtlist=tree->child->next;   //注意DefList和StmtList可能是空的
    if(!deflist->isTerminal && deflist->type == Unterm_DefList){
        stmtlist=stmtlist->next;
        //处理deflist中的变量定义，所有变量定义都是这里处理的（没有全局变量）
        while(deflist != NULL){     //枚举DefList中的每一行Def
            Node* def=deflist->child;
            Node* declist=def->child->next;
            while(1){               //枚举DecList中的每一个Dec（定义的单个变量）
                Node* dec=declist->child;
                Node* vardec=dec->child;
                int var=gid++;      //新分配一个临时变量
                if(vardec->subtype == 0){   //VarDec->ID
                    varnames[varptr]=vardec->child->str_val;
                    vars[varptr++]=var;
                    if(dec->subtype == 1){  //Dec → VarDec ASSIGNOP Exp
                        //处理赋初值表达式
                        int re=CalcExp(dec->child->next->next,0);
                        fprintf(foutput,"t%d := %s\n",var,Trans(re,0));
                    }
                }else{  //VarDec->VarDec LB INT RB, 假定一定是int一维数组
                    int size=vardec->child->next->next->int_val;
                    fprintf(foutput,"DEC t%d %d\n",var,size * 4);   //分配空间
                    int addr=gid++;
                    fprintf(foutput,"t%d := &t%d\n",addr,var);
                    //注意：var仅表示这个数组，而addr表示其首地址
                    //以后var这个编号就没用了，用数组的时候直接用表示地址的临时变量ad
                    //反正以后用到数组一定是访问其元素，这一定会计算地址
```

```
                    varnames[varptr]=vardec->child->child->str_val; //第一个 VarDec一
                    vars[varptr++]=addr;
                }
                if(declist->subtype == 0)    //DecList->Dec
                    break;
                declist=declist->child->next->next; //DecList → Dec COMMA DecList
            }
            deflist=deflist->child->next;
        }
    }
    if(!stmtlist->isTerminal && stmtlist->type == Unterm_StmtList)
        Compile(stmtlist);      //直接递归处理 stmtlist
    return;
}
```

同样是用非递归方式的循环，将 Def（共用同一种类型的一行变量定义）从 DefList 里一个一个一个的剥离下来，也不用关心 Specifier 是啥，只有 int。然后再来一重循环把 Dec（单个变量定义）从 DecList 里一个一个一个的剥离下来。对于 Dec，它可能是普通变量，就直接给它分配临时变量编号，也不一定要真的生成代码，除非是有赋初值，那么用 CalcExp 解析初值赋给它就行了。无论如何，都要在变量表里插入它的临时变量。

如果 Dec 是数组变量定义，因为一定是一维数组，所以也不用存初始容量（访问元素时不关心它），用 DEC 给它分配空间，然后就像之前说的，再分配一个临时变量取地址，用这个表示地址的临时变量代替这个数组，将其插入变量表。

7 控制及一般语句的编译

这里是 Compile 里处理 Stmt 以及剩下的部分（没想到吧，全部代码就这么点！）：

```
case Unterm_Stmt:{
    /*
    Stmt → Exp SEMI | CompSt | RETURN Exp SEMI | IF LP Exp RP Stmt
            | IF LP Exp RP Stmt ELSE Stmt | WHILE LP Exp RP Stmt
    */
    switch(tree->subtype){
    case 0: //Stmt->Exp SEMI
        CalcExp(tree->child,0); break;
    case 1: //Stmt->CompSt
        Compile(tree->child); break;
    case 2:{//Stmt->RETURN Exp SEMI
        int ret=CalcExp(tree->child->next,0);
        fprintf(foutput,"RETURN %s\n",Trans(ret,0));
        break;
    }
    case 3: //IF LP Exp RP Stmt
    case 4:{//IF LP Exp RP Stmt ELSE Stmt
        //保证Exp一定是ID或者Exp RELOP Exp这样的简单条件表达式
        /*
```

```
            [计算条件表达式]
            IF [条件为真] GOTO true_label
            [else语句体]
            GOTO end_label
        LABEL true_label:
            [if语句体]
        LABEL end_label:
        */
        int true_label=glabel++;     //True语句体入口
        int end_label=glabel++;      //if后续语句
        Node* condexp=tree->child->next->next;
        CondExp(condexp,true_label);     //条件跳转
        //如果有else，把else语句体放在前面，这样省跳转
        if(tree->subtype == 4)  //IF LP Exp RP Stmt ELSE Stmt
            Compile(tree->child->next->next->next->next->next->next);    //生成else的s
        fprintf(foutput,"GOTO label%d\n",end_label);
        //无论是不是有else，条件不成立时必须跳过true语句块
        fprintf(foutput,"LABEL label%d :\n",true_label);
        Compile(tree->child->next->next->next->next);
        fprintf(foutput,"LABEL label%d :\n",end_label);
        break;
    }
    case 5:{     //WHILE LP Exp RP Stmt
        /*
        LABEL begin_label:
            [计算条件表达式]
            IF [条件为真] GOTO true_label
            GOTO end_label
        LABEL true_label:
            [循环体]
            GOTO begin_label
        LABEL end_label:
        */
        int begin_label=glabel++;    //循环入口
        int true_label=glabel++;     //循环体入口
        int end_label=glabel++;      //循环后续语句
        fprintf(foutput,"LABEL label%d :\n",begin_label);
        Node* condexp=tree->child->next->next;
        CondExp(condexp,true_label);     //条件跳转
        fprintf(foutput,"GOTO label%d\n",end_label);     //结束循环
        fprintf(foutput,"LABEL label%d :\n",true_label);
        Compile(tree->child->next->next->next->next);
        fprintf(foutput,"GOTO label%d\n",begin_label);
        fprintf(foutput,"LABEL label%d :\n",end_label);
```

```
            break;
        }
        }
        return;
    }
    default:{   //其它非终结符直接无脑递归遍历
        for(Node* ptr=tree->child; ptr != NULL; ptr=ptr->next)
            Compile(ptr);
    }
    }
}
```

对于 Stmt，它可能是普通语句/普通表达式/return/while/if，对于前两种情况无脑递归即可，对于 return 简单地生成一条 RETURN 指令即可。

对于 while 和 if，我们需要生成一个具有条件跳转和标号的结构。考虑到条件跳转指令的特殊结构，我们不能直接简单地把条件表达式使用 CalcExp 解析出来值然后判断它是否非 0。考虑到实验样例的简单性，我们不考虑那些用 and、or 等连接的复杂条件表达式，仅考虑以下两种：

1. 普通表达式，这个可以直接用 CalcExp 计算出值，然后 IF exp != #0 GOTO label。

2. 形如 A RELOP B 的条件表达式，这个需要用 CalcExp 解析 A 和 B，然后显式的在条件跳转指令中指定 relop（这个在词法分析时已经被记录在语法树结点上了）。

由于 if 和 while 都要解析这两种条件表达式，我使用了函数 CondExp 来解析条件表达式并生成条件转移指令：

```
void CondExp(Node* condexp,int true_label){
    //处理 if/while 使用的条件表达式，若真则跳转到指定标签
    //保证条件表达式一定为 Exp RELOP Exp 或者普通 Exp（判断非 0）
    if(condexp->subtype == 14){ //Exp->Exp RELOP Exp
        int a=CalcExp(condexp->child,0);
        int b=CalcExp(condexp->child->next->next,0);
        char* rel;
        int relop=condexp->child->next->relop;  //词法分析时区分不同比较运算符
        switch(relop){
        case RELOP_EQU: rel="=="; break;
        case RELOP_NEQ: rel="!="; break;
        case RELOP_GE:  rel=">="; break;
        case RELOP_LE:  rel="<="; break;
        case RELOP_GT:  rel=">";  break;
        case RELOP_LT:  rel="<";  break;
        }
        fprintf(foutput,"IF %s %s %s GOTO label%d\n",Trans(a,0),rel,Trans(b,1),true_label
    }else{  //普通 Exp，先计算然后 simply 判断是否非 0
        int cond=CalcExp(condexp,0);
        fprintf(foutput,"IF %s != #0 GOTO label%d\n",Trans(cond,0),true_label);
    }
}
```

对于 if-else，我生成这样的“else 语句块在前放置”的结构，这样节省了跳转指令：

```
    [计算条件表达式]
    IF [条件为真] GOTO true_label
    [else语句体]    (这个可省略)
    GOTO end_label
LABEL true_label:
    [if语句体]
LABEL end_label:
```

对于 while，我生成这样的结构：

```
LABEL begin_label:
    [计算条件表达式]
    IF [条件为真] GOTO true_label
    GOTO end_label
LABEL true_label:
    [循环体]
    GOTO begin_label
LABEL end_label:
```

是不是非常简单？当然，如果你想的话，可以这样优化（我没在这个版本中使用，不过我自己的版本是这么干的）：把条件转移指令由“条件成立则跳转”改成“条件不成立则跳转”……

EOF