南开大学
编译原理实验报告（三）

题目：中间代码与目标代码生成

专业：软件工程

小组：贯彻习大大重要回信精神

任课教师：马玲

2017 年 12 月

一．小组成员及分工

公岩松（1511419）：相关数据结构及接口的封装；
郭迎港（1511423）：语句部分的代码生成；
孟宇航（1511443）：表达式部分的代码生成；

本次实验的思路、架构是全组多次讨论的集体智慧的结晶，在测试、debug方面也充分体现了结伴编程的思想。

二．实验内容

1. 利用 flex、bison，对于一个符合语法的 C 程序，给出对应的三地址码；
2. 利用得到的三地址码，生成目标代码（C）.

支持的语法：

1. 数据类型：int、char、float、double 等 ；
2. 基本语句：赋值语句、if 语句、while 语句、for 语句等；
3. 基本运算：四则运算、取模运算、位运算、关系运算、逻辑运算、++、--、+=、 -=、/=、*=等；
4. 代码块部分：复合语句（支持作用域）；
5. 注释：单行注释、多行注释 ；
6. 其他：输入输出语句、多维指针、强制类型转换等。

三．实现方法

（一）.三地址码的定义

考虑到三元式或间接三元式的维护具有一定复杂性，我们选用四元式来定义三地址码，即有 op, arg1, arg2, result 四个字段，例如对于以下代码片段：

```
int i,n=1;
for(i=0;i<5;i++)
{
    n++;
}
write(n);
```

对应的四元式输出为：

1	=	1		n
2	=	0		i
3	<	i	5	&0
4	FOR	&0		10
5	GOTO			8
6	a++	i		&1
7	GOTO			3
8	a++	n		&2
9	GOTO			6
10	WRITE	n		

其中需要特别说明的是，以“&”开头的为临时变量；为了区分前++、后++，两者对应的 op 分别为++a 和 a++，前--、后--同理；对于 IF、WHILE、FOR 等条件跳转语句的四元式，其含义为当作为参数的 bool 表达式值为 false 时跳转到 result 中的语句，对应上图第 3、4 个四元式，即如果“i<5”为 false，则跳至第 10 条四元式，也就是跳出循环；对于指针的目标代码声明我们做了特殊处理，例如对于**p 而言，它在四元式中的 result 不再是一个临时变量，而是“**p”本身。

跳转语句的 result 即跳转的目标理论上应该是一个继承属性，但为了能在自底向上的语法分析过程中一次生成四元式而不是再次遍历语法树回填，我们实际上依赖了一个可变的综合属性。我们为上次作业中的每个语法树节点增加了一个字段 next，表示该节点之后的一条四元式的标号。这是一个综合属性，父节点的 next 依赖子节点的 next。而在归约过程中，所有跳转语句都是后来插入的四元式，并在插入时对受影响的节点的 next 以及受影响的之前插入的跳转四元式的 result 进行更新。下面以最为复杂的 for 循环的一条产生式为例，进行具体的说明。

```
iteration_statement : FOR LCURVE expression_statement expression_statement expression RCURVE statement
```

当规约至此条产生式时，$3、$4、$5、$7 对应的四元式都已经生成完毕，之后跳转语句的四元式的插入过程如下图所示：

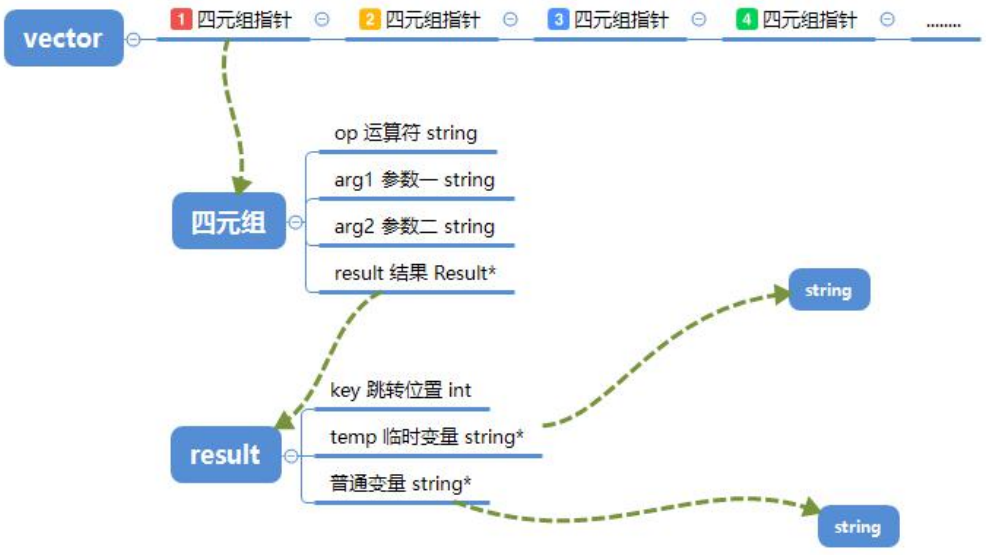
（1） （2） （3）

（4） （5）

（二）.相关数据结构

1、四元组

四元组包含 op，arg1，arg2，result 四个部分前三个部分都可以用 string 来表示，最后一个部分有三种可能，一是 string 的变量名，一是 string 的临时变量名，一是 int 的跳转位置。为了能精确跳转到某个位置，四元式必须以某种顺序的形式存放在内存中，并且根据前文中我们采用的算法，需要有少量的查询并插入新的四元式组的操作，有大量 push_back 类型的操作。为了区别不同类型的四元式，并且能够使运行效率尽可能的高，尽可能节省空间。我们决定采用以下形式表示四元组：



为了达到一边生成四元式一边生成目标代码的目的，我们在四元式类中加入了存放每个四元式对应的 C 代码的 vector<string>字段（图中未予显示）。

从四元式组需要预留的接口有：

```
//加结果是临时变量的四元式
string append(string op, string arg1, string arg2, bool flag = false);
//加结果是变量的四元式
string append(string op, string arg1, string arg2, string symbol, bool flag = false);
//加结果是跳转的四元式
int append(string op, string arg1, string arg2, int key, bool flag = false);
//修改一个为key的四元式
bool modifyKey(int number, int i);
//返回下一个插入的编号
int nextKey();
//输出代码
void outputCode();

//插入结果是跳转的四元式
int insert(int position, string op, string arg1, string arg2, int key, bool flag = false)
```

其中值得注意的部分是，当我们插入结果是临时变量的四元式时，我们会根据运算符和参数，得到临时变量的类型，并把临时变量储存在临时变量表中。另外，上图中的 flag 标记跳转指令是向前跳转还是向后跳转，二者在处理上有细微的差别。

2、声明段

这个数据结构主要用来储存声明和一些头信息，只是一些 vector<string>。主要在当临时变量表和符号表在离开一个作用域时会把此作用域的所有符号或者临时变量按照一定命名规则放入声明段中。这样做有两个好处：

第一， 不会造成变量名混淆的问题。

第二， 有利于我们做临时变量复用的代码优化。

3、临时变量表

临时变量表继承了符号表。

临时变量表同符号表，有以下几个功能，上一次的实验报告中有详细的说明，这里简单列举：

A，区分作用域，在子作用域中可覆盖父作用域的符号。

B，当出了一个不会再用到的作用域时，作用域中的符号全部删除。

C，不会出现重定义的问题。

D，只可查询到当前作用域能够访问的未被覆盖的符号。

为了能够实现代码优化，我们在此基础上增加了一个功能，自动找到可以复用的临时变量名，并且使新的临时变量复用它。为此我们维护了一个哈希表。

（三）.目标代码生成

在我们的解决方案中，目标代码生成和四元式的生成是同时进行的，都在自底向上的一趟语法分析中，当每一条四元式通过 insert 或 append 进四元式组时

都会将其对应的 C 代码插入之前提到的 vector<string>中。最后在输出时，先输出声明段中的信息，再按序输出每个四元式的 vector<string>（在最前面加上 LABEL 信息以便跳转）。

（四）.代码优化

我们一共采用了两种简单的代码优化手段。其一是对代数恒等式的优化，例如 x+0、x-0、x*1、x/1 这样的式子不单独产生四元式，并在语法分析阶段提前完成对常量的计算。另外就是不冲突的临时变量的复用，这依赖于之前的符号表数据结构对作用域的良好支持。以下面这段代码为例：

```
int main()
{
    int i,a,b=1;
    if(b<5)
    {
        a=b+1;
    }
    for(i=0;i<5;i++)
    {
        a=a+5;
    }
    return 0;
}
```

其对应的四元式如下：

=	1		b
<	b	5	&0
IF	&0		6
+	b	1	&1
=	&1		a
=	0		i
<	i	5	&1
FOR	&1		15
GOTO			12
a++	i		&2
GOTO			7
+	a	5	&3
=	&3		a
GOTO			10

我们可以看到对临时变量&1 的复用，这种复用在面对复杂的代码时可以大大减少临时变量的使用数量。