第四部分

毋哲

四元式的设计

任务：

四元式是我们选择的本编译程序所编译的源程序的中间形式。我们要知道其实完全可以不使用中间形式而一次生成目标代码。但是使用中间形式有很多的优点。首先，中间代码不需要考虑机器的特性，因此可以很容易的将生成中间代码的编译程序移植到别的机器上，这时只需要再为中间代码开发一个到目标机指令的翻译器即可。其次，使用中间代码可以更好地对编译程序进行优化，在后面的章节中，我们将会看到使用中间代码进行优化的方法。

什么时候需要开始考虑设计四元式呢？

四元式是源程序的一种等价的中间表示形式。因此，在语义上，四元式与源程序是等价的。因此，在做语义分析之前，就一定要根据文法，设计出所需要用到的四元式。而且语义分析的结果就是生成四元式，因此也就必须要在语义分析之前设计好四元式。

四元式变量的命名

通过对编译原理的学习，我们了解在源代码的中间形式中，会出现一类在源程序中不会出现的变量，临时变量。如语句 d=a+b+c，可能生成如下形式的四元式

+,a,b,t1

+,t1,c,t2

=,t2,,d

因此，对临时变量的命名是一个需要解决的问题。比较推荐的方法是使用一个特殊标识符再在其后面加上一个编号来表示。如：T1,&1 等。这里不推荐使用字母加数字的方法是因为很有可能原程序中含有名字相同的变量。因此使用源程序中不能作为标识符首字母的字符来当做临时变量的标识符是一个比较好的方法。

其次还有一个需要考虑的地方，也是大家一开始不容易想到的地方就是对源程序中的变量名进行改变。为什么要这么做呢？以目标语言为 X86 汇编语言举例。当我们生成了汇编语言目标代码以后，我们要是用汇编器对生成的代码进行汇编。而汇编语言中也是存在保留字的，如 C，因此当我们源程序中有 C 这个变量时，生成的汇编语言中就需要对其进行更名。虽然说这个步骤可以留到在生成目标代码阶段（汇编语言）再进行处理，但是由于我们希望我们的程序可以移植，因此对于那些不能保证有对保留字进行更名处理的编译器的后端，就会出现错误。因此，我们希望编译器的前端可以提供保证变量名不会是保留字的功能。其实在生成中间代码阶段改变变量名很容易，只需要在所有变量名前加一个‘_’字符即可。

是否必要为每一种语句设计一套四元式？

对于四元式，我想每个人都希望使用尽量少的四元式来表示源程序的中间形式。但是，一开始，我们很难抽象出一些公共的四元式来供所有的源程序语句使用，这就需要我们对源程序中一些感觉比较复杂的语句进行分析，来找出它们的共性。

我们首先来看如下三种语句：

if， while, for

我们很容易发现，对于这三种语句，都有对条件语句进行判断的语句，所以我们设想可

否使用同一种判断语句的四元式来实现上面三种语句。

对于 if 语句，假设有下列语句：

if (a<b) a=1; else b=1;

我们将其转换为四元式，很容易想到首先要进行条件表达式的判断（假设我们已经设计出了条件表达式的四元式），可得如下四元式：

<,a,b,,

之后就需要根据条件表达式的结果进行跳转，这里我们可以使用这样一个四元式，JWNT,,label,,表示当上一句判断表达式为假时(jump when not true)，跳转到 label 标识的语句。这里我们使用当条件为假时跳转，目的是使得四元式执行的顺序与源程序更相近。因此此时得到的四元式为：

```
    <,a,b, ,
    JWNT, ,L1, ,
    =,a, ,1
L1:  =,b, ,1
```

这时观察四元式，假如表达式为真，则不跳转，这时当进行完 a=1 后还会进行 b=1。这种情况是不符合语义的，因此我们需要在执行完 else 之前的语句后，跳转到 else 之后语句列的结束，这里用到一个无条件跳转四元式 JMP。因此，此时得到的四元式为：

```
    <,a,b, ,
    JWNT, ,L1, ,
    =,a, ,1
    JMP, ,L2, ,
L1:  =,b, ,1
L2:
```

上述语句就表示了一个简单 if 语句的四元式。

下面再观察 while 语句。假设有 while 语句：

```
while (a<b)
{
    a=a+1;
    b=b-1;
}
```

根据上述设计 if 四元式的思路，可以很容易地设计出 while 语句的四元式，而且除了比较四元式和运算四元式以外，也只需要使用 JWNT 四元式和 JMP 四元式即可。

```
L2:  <,a,b, ,
    JWNT, ,L1, ,
    +,a,1,a;
    -,b,1,b;
    JMP, , L2, ,
L1:
```

其实 for 语句也可以只使用比较、计算、JWNT 和 JMP 四元式便可被完全描述出。对于 for 语句四元式的流程，请大家自己思考。

下面给出我设计出的四元式供参考：

说明：第一个单词（第 1 元）表示操作名称，X1,X2,X3 分别表示第 2 元，第 3 元和第 4 元。

基本操作符

LABEL	X1,X2,X3	设立一个标签，第 3 个元为标签标识；未使用第 2 元和第 4 元
BEGIN	X1,X2,X3	函数开始，第 3 个元是函数名；未使用第 2 元和第 4 元
END	X1,X2,X3	函数结束，第 3 个元是函数名；未使用第 2 元和第 4 元
SCANF	X1,X2,X3	从标准输入读，第 3 个元是读入的变量；未使用第 2 元和第 4 元
PRINTF	X1,X2,X3	向标准输出写，第 2 个元是字符串，第 3 个元是表达式；未使用第 4 元
RET	X1,X2,X3	返回，第 3 个元是要返回的变量；未使用第 2 元和第 4 元
JWNT	X1,X2,X3	条件为假时跳转，第 3 个元是要跳到的标志；未使用第 2 元和第 4 元
JMP	X1,X2,X3	无条件跳转，第 3 个元是要跳到的标志；未使用第 2 元和第 4 元
VARPOP	X1,X2,X3	形参赋值，第 3 个元是形参标志符；未使用第 2 元和第 4 元
VARPUSH	X1,X2,X3	传实参，第 3 个元是实参的变量；未使用第 2 元和第 4 元
CALL	X1,X3,X3	函数调用，第 2 个元是函数名，第 3 个元是返回值的地址，第 4 个元是函数调用参数个数；

运算操作符

+	X1,X2,X3	加法运算，第 2，3 元是操作数，第 4 元是值存储地址；
-	X1,X2,X3	减法运算，第 2，3 元是操作数，第 4 元是值存储地址；
*	X1,X2,X3	乘法运算，第 2，3 元是操作数，第 4 元是值存储地址；
/	X1,X2,X3	除法运算，第 2，3 元是操作数，第 4 元是值存储地址；
Opp	X1,X2,X3	求相反数，第 2 个元是操作数，第 4 个元是目的变量；未使用第 3 元
=	X1,X2,X3	赋值，第 2 个元是操作数，第 4 个元是目的变量；未使用第 3 元

比较运算符

>	X1,X2,X3	大于比较，第 2，3 元是操作数；未使用第 4 元
>=	X1,X2,X3	大于等于比较，第 2，3 元是操作数；未使用第 4 元
<	X1,X2,X3	小于比较，第 2，3 元是操作数；未使用第 4 元
<=	X1,X2,X3	小于等于比较，第 2，3 元是操作数；未使用第 4 元
!=	X1,X2,X3	不等于比较，第 2，3 元是操作数；未使用第 4 元
==	X1,X2,X3	等于比较，第 2，3 元是操作数。未使用第 4 元

强制转换运算符

ctoi	X1,X2,X3	字符类型强制转化为整形，第 2 个元是操作数，第 4 个元是目的变量；未使用第 3 元
ctof	X1,X2,X3	字符类型强制转化为浮点型，第 2 个元是操作数，第 4 个元是目的变量；未使用第 3 元
itoc	X1,X2,X3	整形强制转化为字符类型，第 2 个元是操作数，第 4 个元是目的变量；未使用第 3 元
itof	X1,X2,X3	整形强制转化为浮点型，第 2 个元是操作数，第 4 个元是目的变量；未使用第 3 元
ftoc	X1,X2,X3	浮点型强制转化为字符类型，第 2 个元是操作数，第 4 个元是目的变量；未使用第 3 元
ftoi	X1,X2,X3	浮点型强制转化为整形，第 2 个元是操作数，第 4 个元是目的变量；未使用第 3 元

语义分析

语义分析到底是在做什么？

在我周围的同学进行编译程序编写的时候，当设计出四元式以后就无从下手了，大家对语义分析的作用都不是很了解。在课程中学到的语法制导翻译技术有点过于抽象，而且大多数同学学得也不扎实，无法掌握其中的要领，因此总能听到同学们在问，语义分析到底干些什么？

根据我个人对语义分析的理解，我认为语义分析无非是在做两件事。1，检查语义错误（所使用的变量是否有定义等）。2，生成四元式。由于生成的四元式与源程序等价，因此，我们可以看作是将一种语言翻译成另一种语言，所以被称之为“语义”分析。我想如果这样解释一下的话，语义分析会比较好理解一些。

需要完全将文法写成属性翻译文法才能做语义分析吗？

有些同学在学了编译原理课以后，感觉语义分析实在复杂，要把语法首先改成语法制导翻译文法才能进行语义分析。而在改文法的过程中，又不知应该如何设置动作符号和各种属性，而且即使设置好了，也不一定正确，更可能给编程带来很大的困难（实现上的困难和理解上的困难）。

其实大可不必将文法改成语法制导属性翻译文法再进行语义分析。其实对于每一个动作，其实可以很自然地将其插入到语法分析中。比如：当判断出当前是定义变量的时候，当

识别出一个标识符，就可以检查这个标识符的有效性（动作），并将其插入到符号表中（动作）。如果判断出当前是一个表达式，就需要查符号表获得每个变量的类型（动作，属性传递），然后将结果返回给调用这个表达式的语法分析函数（属性传递）。

下面以一个具体的例子来解释如何不通过属性翻译文法来进行语义分析。

```
//<条件>的语法分析函数
int Condition()//<条件>子函数，语义分析的目的是生成条件四元式
{
    ExpressCentence();//<表达式>子函数。条件四元式中需要获得这个表达式的信息
    if (WordBack.type<=2010 && WordBack.type>=2005)//判断是表达式后有无比较符号
    {
        RearWork();//读一个单词
        ExpressCentence();//进入另一个<表达式>子函数。条件表达式中也需要这个表达式的信息。
        //此时，两个表达式的语义信息都已获得，所以应该在此处生成条件四元式。
        return 1;
    }
    //若条件中只有一个表达式，则需要将这个表达式与 0 进行比较，在此处生成相应的四元式。
    return 1;
}
```

根据上面红色字的分析，我们可以很容易的得到带有语义分析的<条件>子函数，这个函数可以生成条件四元式。

```
//<条件>的语法、语义分析函数
int Condition()//<条件>子函数
{
    int ConditionType;
    struct CharAndType *Exa,*Exb,*Exc;
    Exa=ExpressCentence();//<表达式>子函数
    if (WordBack.type<=2010 && WordBack.type>=2005)
    {
        ConditionType=WordBack.type;
        ReadWord();
        Exb=ExpressCentence();
        ConditionMiddle(ConditionType,Exa,Exb);//条件语句四元式
        return 1;
    }
    Exc=(struct CharAndType *)malloc(sizeof(struct CharAndType));
    strcpy(Exc->str,"0");
    Exc->type=INTPIG;
    ConditionMiddle(BUDENGYUPIG,Exa,Exc);
    return 1;
}
```

于是，我们在没有使用属性翻译文法的前提下，正确地进行了语义分析。我个人认为这是一种相对高效，也较准确的思路去进行语义分析。

属性翻译文法的属性传递

当大家学习属性翻译文法的时候，印象最深的可能就是对各种属性的传递了。我们到底什么时候应该传递属性？什么时候向上传，什么时候向下穿？都应该传递些什么属性呢？着看起来貌似是一个令人十分头疼的问题。但是如果大家掌握了上一例中利用对语义的理解进行语义分析的方法后，对属性传递的问题其实也是很好解决的。

我们还是那<条件>子函数举例：

```
//<条件>的语法、语义分析函数
int Condition()//<条件>子函数
{
    int ConditionType;
    struct CharAndType *Exa,*Exb,*Exc;
    Exa=ExpressCentence();//<表达式>子函数，需要获取表达式信息，因此<表达式>子函数需要向上传递分析出的表达式信息。
    if (WordBack.type<=2010 && WordBack.type>=2005)
    {
        ConditionType=WordBack.type;
        ReadWord();
        Exb=ExpressCentence();//同理需要获得第二个表达式的信息
        ConditionMiddle(ConditionType,Exa,Exb);//条件语句四元式，生成四元式的过程需要知道比较类型和两个表达式的信息，因此需要向下传递信息。
        return 1;
    }
    Exc=(struct CharAndType *)malloc(sizeof(struct CharAndType));
    strcpy(Exc->str,"0");
    Exc->type=INTPIG;
    ConditionMiddle(BUDENGYUPIG,Exa,Exc);//同理条件语句四元式
    return 1;
}
```

用一种思路来概括属性传递的方法就是：别的函数中需要什么属性，这个函数当分析出这些属性后就要将其传递给上层函数。

对于例子中的表达式。我们只需要知道表达式的类型和保存其结果的标识符即可，因此 Exa,Exb,Exc 只需要有两个字段，一个是类型，一个是标识符或常量值。而生成四元式时，除了需要知道表达式的信息以外，还需要知道比较的类型，因此在向下传递属性的过程中，我们还需要传递比较的类型，即例子中 ConditionMiddle 调用的第一个参数。

通过上述方法，我们可以以一个比较理性的思路来解决属性传递不好判断的问题。

语义分析中的类型转换

类型转换是我们编译器需要提供的功能，而且只要在类型不匹配的时候，就需要进行强制类型转换。因此，强制类型转换主要有 6 种。字符转整型，字符转浮点型，整型转字符，整型转浮点型，浮点型转字符，浮点型转整型。

那我们又是在什么时候需要考虑进行强制类型转换呢？当然就是在类型不同的两个变量进行运算（比较运算或算术运算时）的时候要进行强制类型转换。一个比较容易被忽略的地方是在进行函数调用时，使用的参数类型和定义的不一致时，也需要进行强制类型转换。

我个人认为强制类型转换在生成四元式时考虑是要比在语义分析时进行考虑要方便得多。因为在语义分析中，各种跟运算有关的操作既多又杂，难免无法将每一个需要进行强制类型转换的地方包含全，而且会增加代码的重复。而交给四元式生成的函数处理强制类型转换，一是简化了语义分析的复杂度，二是修改起来也比较方便，而且强制类型转换的代码比较容易移植，比如说加法，减法，乘法，除法完全可以使用一套强制类型转换的函数，省去了不少编程方面的麻烦事。