编译原理笔记（二）

语义分析和中间代码生成

语义分析的任务

- 语义检查：一致性检查（变量类型）和越界检查（数组越界、重复定义）
- 语义处理：对说明语句，填写符号表，进行存储分配；对执行语句，生成中间代码

语法制导翻译：在语法分析过程中，根据每个**产生式对应的语义子程序**（语义动作）进行翻译（生成中间代码）

语法分析是建立语法树，语义分析是遍历语法树，语法制导翻译就是遍历和建立同时完成

中间代码，三地址形式，三地址语句表

Key：归约之后子结点就不存在了，将之后需要用到的子结点的信息在父结点进行保留

> **⑥ 无条件转移语句 goto L**
> **⑦ 条件转移语句 if x rop y goto L； if a goto L**
> **⑧ 过程参数语句param x和过程调用语句 call P, n**
> **⑨ 过程返回语句 return**

四元式

(op, arg1, arg2, result)

语义变量

i.NAME：和终结符i相关联，表示与i对应的普通变量的标识符字符串（即变量名）（？与终结符对应的不一定是标识符，标识符可能含多个字母）

E.PLACE：和非终结符E相关联，表示与E对应的变量在符号表中的位置（普通变量）或整数编码（临时变量）

语义函数

newtemp()：有点像C语言的malloc，创建一个临时变量，并返回临时变量的整数编码。

一棵语法树中可能有多个E，也有多个E.place，但是在形成语法树的过程中，每个E的位置都是唯一确定的，所以是可以区分的，语义子程序会知道自己用的是哪个E

说明语句

将NAME、TYPE、OFFSET（该变量在符号表中的地址偏移量，具体的符号T的OFFSET=OFFSET+T.WIDTH）等说明信息存放到符号表中，只定义，不emit中间代码（？词法分析的时候不得到了二元式（类编码，属性），符号表分名字域和信息域，词法分析的时候是登记名字，但这里好像又重新写了一张表）

增加开始符号S和非终结符M，用于OFFSET初始化:

> S→MD

> M→ε {OFFSET:=0}

赋值语句

给普通变量赋值

```
A->i:=E
{ P=entry(i.NAME);
if(P!=0) emit(P,=,E.PLACE);
else error();}
```

切分成二元表达式，创建temp变量

```
E->E1 op E2
{E.PLACE=newtemp();
emit(E.PLACE，E1.PLACE，op，E2.PLACE);}
```

可能还要考虑操作数的类型和类型转换

emit(P，=，E.PLACE)

非终结符与普通变量关联

```
E->i
{P=entry(i.name);
if(P!=0) E.PLACE=P;
else error();}
```

赋值语句，每次运算归约一次（:=赋值运算也要单独归约），产生中间代码，按运算顺序切分

控制语句

作为控制语句的布尔表达式

布尔表达式为真为假时的出口是指goto语句的地址，根据goto转移到新地址L

反填backpatch(address, L)，也是填ip=address的goto语句中的L

```
B→i1 rop i2
{ B.T := ip;
  emit (if i1 rop i2 goto 0);
  B.F := ip; //emit之后ip+1了
  emit (goto 0)}
```

无条件转移语句

向前转移

> lable->i: {i.NAME=L时，将L加入符号表，类型栏为“标号”，定义否为“已”，地址为ip当前值（其后第一个三地址语句的地址）}
>
> goto L; {查符号表，取出L的地址xxx，生成三地址语句 goto xxx}

向后转移

为什么形成链？把转移目标不确定但会转移到相同地址的三地址语句用链式存储，方便回填

goto 向后转移到相同的label，label暂时未知

控制语句中套控制语句，B.F出口未知

分情况，L是否在符号表中，是否已定义？

goto L

① 若L不在符号表中，则填入它，置“类型”为“标号”，“定义否”为“未”，生成三地址语句goto 0；

② 若L已在符号表中，“类型”栏为“标号”，“定义否”为“已”，则生成三地址语句goto L；

③ 若L已在符号表中，“类型”为“标号”，“定义否”为“未”，则生成三地址语句goto 0并更新标号L的引用链。

label→i:（此时i.NAME=L）

① 若i不在符号表中，则填入它，置“类型”为“标号”，“定义否”设为“已”，“地址”为ip的当前值；

② 若i已在符号表中，“类型”不为“标号”，或“定义否”不为“已”，则报“名字重定义”错误；

③ 若i已在符号表中，“类型”为“标号”，则把”定义否”改为“已”，把“地址”栏中的链首（设为q）取出，同时把当前ip填入，最后执行backpatch(q,ip)，将x为链首的链上所有三地址语句的转移目标都填为ip的当前值。

S.CHAIN：语义变量，记录由S生成的一串三地址语句链的链头，第一个goto语句的地址。

拉链（合链）

backpatch(S.CHAIN, ip)，找到地址为S.CHAIN的goto 0语句，把0改成ip，以S.CHAIN为链首的链子上的goto 0语句都改

(p) goto 0
……
(q) goto p
……
(r) goto q
链头：S.CHAIN = r

t1 = r，t2 = w:
(p) goto 0　(u) goto 0
(q) goto p　(v) goto u
(r) goto q　(w) goto v
执行merg(t1,t2)后，得到新链t3：
(p) goto 0　(u) goto r
(q) goto p　(v) goto u
(r) goto q　(w) goto v

执行backpatch(t3,120)后:
(p) goto 120　(u) goto 120
……　……
(q) goto 120　(v) goto 120
……　……
(r) goto 120　(w) goto 120

条件转移

> if B then S1 else S2

后面要用的就存在父结点，两个非终结符的语义变量是不是一样的

合链 M->SN

如果S.CHAIN和N.CHAIN存的内容是一样的，那就合到M.CHAIN

赋值语句S和谁合链都是他本身，S.CHAIN=0

S.CHAIN，想当前这个语句里面如果有控制语句，会转移到哪里，不是直接表示语句的非终结符，就去想他从哪里合并来的，存的到底是什么内容

if then走完之后要生成转到else后面的goto语句

末尾有；的区别，这里；表示else后面的语句块结束了，即B.F已知，反填回去

```
// 每次归约到S语句前面
B-> i rop j
{
    B.T=ip;
    emit(if B goto 0);
    B.F=ip;
    emit(goto 0);
}

M->if B then
{
    backpatch(B.T,ip); // ip指向下一条三地址语句，下面就该翻译then后面的语句的内容了
    M.CHAIN=B.F; // B归约到M，信息保存到M
}

S1->...
```

```
{
    // 若S1是赋值语句
    //赋值语句本身不形成链，CHAIN记为0，合并的时候相当于单链合并
    S1.CHAIN=0;
    // 若S1包含控制语句C，则有C.F=B.F，当前的B.F存在M.CHAIN中
    S1.CHAIN=M.CHAIN ???
}

N->MS1 else
{
    q=ip;
    emit(goto 0);
    backpatch(M.CHAIN, q);
    N.CHAIN=merge(M.CHAIN, S1.CHAIN); //画出流程图可知，S1为假时也转移到B.F，意义一致，可以合并
}

S2->... //同S1

S->NS2
{
    S.CHAIN=merge(N.CHAIN, S2.CHAIN);
    // backpatch(S.CHAIN, ip);
    // 如果S2后面有;，表示else后的语句块结束，则回填S.CHAIN
}

```

> while B do S

不归约 while B，B的地址要存

```
W → while
{
    W.code=ip;
}

/* 归约到B */

D → W B do
{
    backpatch(B.T, ip);
    D.CHAIN=B.F;
    D.CODE=W.CODE;
}

/* 归约到S1 */

S → D S1
{
    /*
    // 这样行吗?
    S.CHAIN=merge(D.CHAIN, S1.CHAIN);
    backpatch(S.CHAIN, D.CODE);
    emit(goto D.CODE); //向前转移，直接转，不用回填
    */
    backpatch(S1.CHAIN , D.CODE);
```

```
    emit (goto D.CODE); //向前转移，直接转，不用回填
    S.CHAIN = D.CHAIN; //没遇到; 不确定do语句有没有结束，暂时不能回填S.CHAIN(B.F)

}
```

结尾没有分号就把整个分支或者循环的出口保存在S.CHAIN中，如果有就表示语句块结束，出口确定，backpatch(S.CHAIN, ip)

怎么知道每次归约到哪里?

每个语句块归约一次，终结符（字符串）跟在那边无所谓，可以按语义划分，for i:=E1 | step E2 | until E3 | do S1，一般是归约到语句块之前

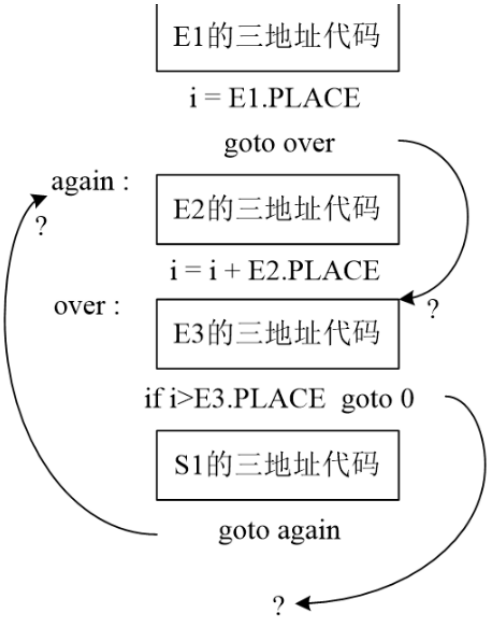
while和repeat需要保存返回地址到CODE，所以单独归约一次

> for i:=E1 step E2 until E3 do S1

先赋初值，然后判断、执行，然后递加、判断、执行循环

文法: S → for i:=E1 step E2 until E3 do S1
语义为:

```
      i=E1;
      goto over;
again : i=i+E2
over:  if(i<=E3){
      S1;
      goto again;}
```




```
F1 → for i:=E1
{P=entry(i.NAME);
emit ( P, =, E1.PLACE );
F1.PLACE := P;    //保存i在符号表中的位置（指向i的指针，后面还要i递增
F1.CHAIN: = ip;   //指向goto 0，期待回填，所以保存。
emit ( goto 0 ); //over地址位置
F1.AGAIN := ip;} //指向下一条三地址语句

F2 → F1 step E2
{F2.AGAIN := F1.AGAIN;
F2.PLACE := F1.PLACE;
emit (F2.PLACE  = F1.PLACE + E2.PLACE ,)
backpatch ( F1.CHAIN , ip )}

F3 → F2 until E3
{F3.AGAIN := F2.AGAIN;
 F3.CHAIN := ip;  //保存下一条三地址语句ip，用于回填
emit ( if F2.PLACE > E3.PLACE goto 0 ) ;}//出口未知
```

```
S → F3 do S1
{emit ( goto F3.AGAIN );
backpatch ( S1.CHAIN , F3.AGAIN );
S.CHAIN := F3.CHAIN;} //记录整个for循环的出口
```

> for i:=1 to N do S1

```
F→for i:=1 to N do
{ F.place:=entry(i);
   emit(:=,'1',-, F.place);
    F.again:=ip;                        //F+1
    emit(j<=,entry(i),N,F.again+2);
    F.CHAIN:=ip;                  //F+2
    emit(j,-,-,0);}

S→F S1
{     backpatch(S1.CHAIN,ip);
      emit (+,F.place,'1',F.place);
      emit(j,-,-,F.again);
      S.CHAIN:=F.CHAIN  }
```

T 语义翻译：

每一步归约都要检查子结点后面要用的信息是否保存到父结点中了，PLACE、CHAIN、CODE、OVER、AGAIN

先画流程图，画出拉链，对着流程图检查中间代码

归约的顺序是从上到下，从内层往外层，比如while B do S，先归约出B，然后归约while B do，再归约S，while B do S，按while语句的框架归约

最后归约；，回填最外层的出口

一般会给具体的句子来归约，给句型需要考虑非终结符可能会产生转移语句，因此形成拉链，需要回填CHAIN

好像语义分析的例子都是根据我们自己的产生式在归约?

T：直接写出中间代码

先算一下一共生成多少条，分支语句生成两条，赋值语句切分，else前面加goto

中间代码优化

局部/全局优化：基本块内/外

基本块的划分

1.入口语句：

✓程序的第一个语句；

✓能由转向语句（条件语句或无条件语句）转移到的语句；

✓紧跟在条件语句之后的语句；

2.出口语句：

✓转向语句；

✓停止语句；

入口语句的第三条规则，应该跟在转向语句之后的都是入口语句吧？

划分基本块时先找出所有入口语句，然后根据入口语句找块

画出语句块的程序流图，按照代码逻辑顺序、转移、分支即可

块内优化：

- 合并已知量，常数计算
- 删除公共表达式
- 删除无用赋值，比如重复赋值
- 删除死代码，条件恒为真或假

全局优化

只讨论循环优化

❑ **循环的定义：**
循环是程序流图中有唯一入口结点的强连通子图。

① **入口结点**:子图中满足下列条件的结点n：n是流图的结点，在子图外有一结点m，它有一有向边m→n引向结点n；

② **强连通子图**:任意两个结点之间有路径可互相到达。

1.代码外提：
在循环中，对x:=op y 或 x:=y op z一类的运算，如果y,z均为循环不变量，则该代码可以提到循环外，只计算一次；

2.强度削弱：
在循环中，若变量i有唯一定值，i:=i±c，则称i为基本归纳变量。
若变量j:=c_1*i±c_2，i是基本归纳变量，则称j为同族归纳变量。
将同族归纳变量j的乘法变成加法，称为强度削弱，即改成j:=j±c_1*c，同时在循环外赋初值j:=c_1*i±c_2，c_1*c是固定值，乘法只用算一次，之后都是加法

3.删除归纳不变量：
将循环的控制条件由依赖于基本归纳变量改成依赖同族归纳变量，则可将基本归纳变量删除。例如：j = 10 * i + 5，判断条件为 i > 10 ，则将 i > 10 改为 j > 105，同时删除i相关的语句。

T：简答题，或者给代码做优化，看下PPT上的例子

强度削弱的时候注意不仅要改写和i直接相关的，还要注意间接相关的，比如t1=10*i，t2=t1+10

目标代码生成

目标代码尽量短，充分利用机器的寄存器

一般要把操作数先从内存取到寄存器中再计算，有的指令格式可能允许直接从内存中取数，这样就涉及到置换算法，怎么分配寄存器效率最高

T：写出目标代码，合理分配寄存器

尽量重复利用寄存器，中间结果也需要保存

T：固定分配寄存器节省的代价计算

在等号左边的变量如果被固定分配，就可以节省一条写回内存的指令，该指令的执行代价是2，所以是2*LIVE

在等号右边的变量如果被固定分配，就可以节省一条从别的寄存器取数的指令，该指令执行代价为1（取指令时访问一次内存），所以是USE

运行时存储空间的组织

活动记录是一个程序单元的数据空间

活动记录前三个地址存放的内容：返回地址，动态链接（根据调用关系），静态链接（根据代码的嵌套关系）

动态链接：调用它的外层程序单元

静态链接：代码外层

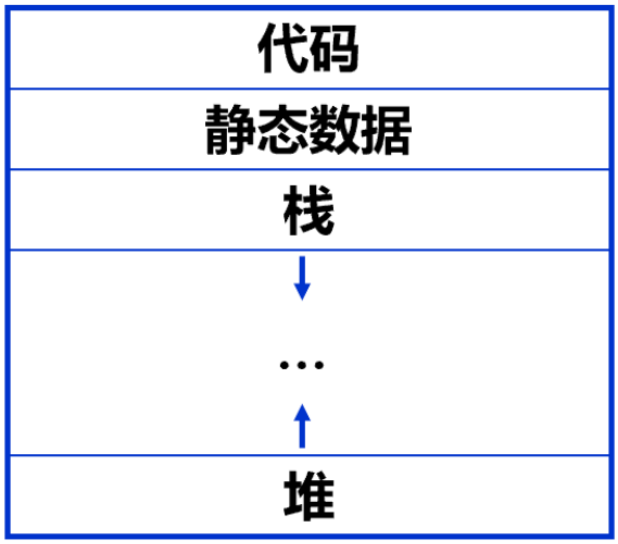
变量类型

- 静态
- 半静态：变量在单元每次激活时动态地绑定刚建立的活动记录，变量的长度在编译时可确定
- 半动态：变量长度在单元激活时才能确定，比如动态数组；编译时在活动记录中建立描述符，描述符的大小在编译时可决定（因为数组的维数是可确定的）；在单元激活时，才分配它们的空间；
- 动态：描述符及空间大小在编译时不能决定。编译时在活动记录中为动态变量设置二个指针，一个指向该变量的描述符，另一个指向该变量的存储空间。某些语言中可变的变量，比如维数可变的数组，链表和树。

分配模式

- 静态分配，变量与存储区的绑定关系在编译时建立，不允许递归调用
- 栈式分配：半静态和半动态变量
- 堆分配：堆是根大于或小于子节点的完全二叉树，动态变量

运行时的空间组织



非局部环境的引用

都是找最近外层中的定义

静态作用域规则，最近外层是指代码嵌套中的外层

动态作用域规则，最近外层是指调用的外层，此时静态链接和动态链接一致

在嵌套深度为nd的程序单元p中引用在嵌套深度为nt的程序单元t中说明的变量x

$d = n_t - n_p$

`f(d)= (d==0) ? current : D[f(d-1)+2]`

使用静态链接，找到t的活动记录首地址Dt=f(d)

当A调用B时

$d = n_A - n_B$

Dt=f(d+1)

D[free+2]=f(d+1)

半静态callP语句的翻译（必考）

动态链接：指向调用单元的活动记录开始位置

静态链接：

free指向栈顶地址+1，current指向记录的开始位置（从0开始计数）

生成记录

```
/* 填写前三条内容 */
D[free]=ip+k;  //D[curve]表示某地址单元的内容，假设调用语句之后的第k条位置为返回地址
D[free+1]=current;
D[free+2]=f(d+1);  //静态作用域规则
/* 移动指针 */
current=free;
free=current+L;
ip=P的代码段首地址;      //激活被调用单元
```

释放记录

```
free=current;
current=D[current+1];  //转向动态链接指向的调用单元
ip=D[free];  // ip是代码段的指针，此时应该指向返回语句
```

半静态变量的栈式分配

活动记录由两部分组成：一部分是编译时可静态确定的部分(描述符：首地址、类型、维度），另一部分是动态数组的存储区。

1.程序单元被激活时，在栈顶单元分配活动记录的第一部分；分配后free指针指向栈顶；

2.遇到每个动态数组说明，将确定的信息填入内情向量，并计算出它的长度L；然后在栈顶分配L个空间，即free:=free+L

参数类型

形参：被调用单元的参数

实参：调用单元的参数

传参方式

传值，传值得结果，传指针

选择或填空：

编译的定义

产生语言，注意i，j，k是否相等，>0还是>=0?

文法的二义性，某个句子能对应两棵不同的语法树

规范推导和归约

设计准则

算符文法和算符优先文法的定义

简答题：

编译的流程，变量的属性（4），数据类型和例子（6），控制结构，语言的类型，算符文法的优先级怎么判断，中间代码优化（块内、块间，也可能考大题）

分析题：

写出规范推导，画树，找那几个部分

算法题：

语法分析考两道题，预测分析法（先消左递归和公共左因子）

语义分析20分，记忆三种结构，画出流程图，把要用的信息保留到父节点，反填，无条件转移，一般是写中间代码，可能考填空

空间分配，call P和return的翻译

选择 2*10

填空 10

简答 25

语法树 10

预测分析

SLR(1)

中间代码，语义子程序 （循环可能会写成不一样的形式）continue和break 怎么划分？

类型转换的两种方式：扩展和收缩

类型的举例

各阶段的输入、输出

递归下降分析的概念

值调用，引址调用