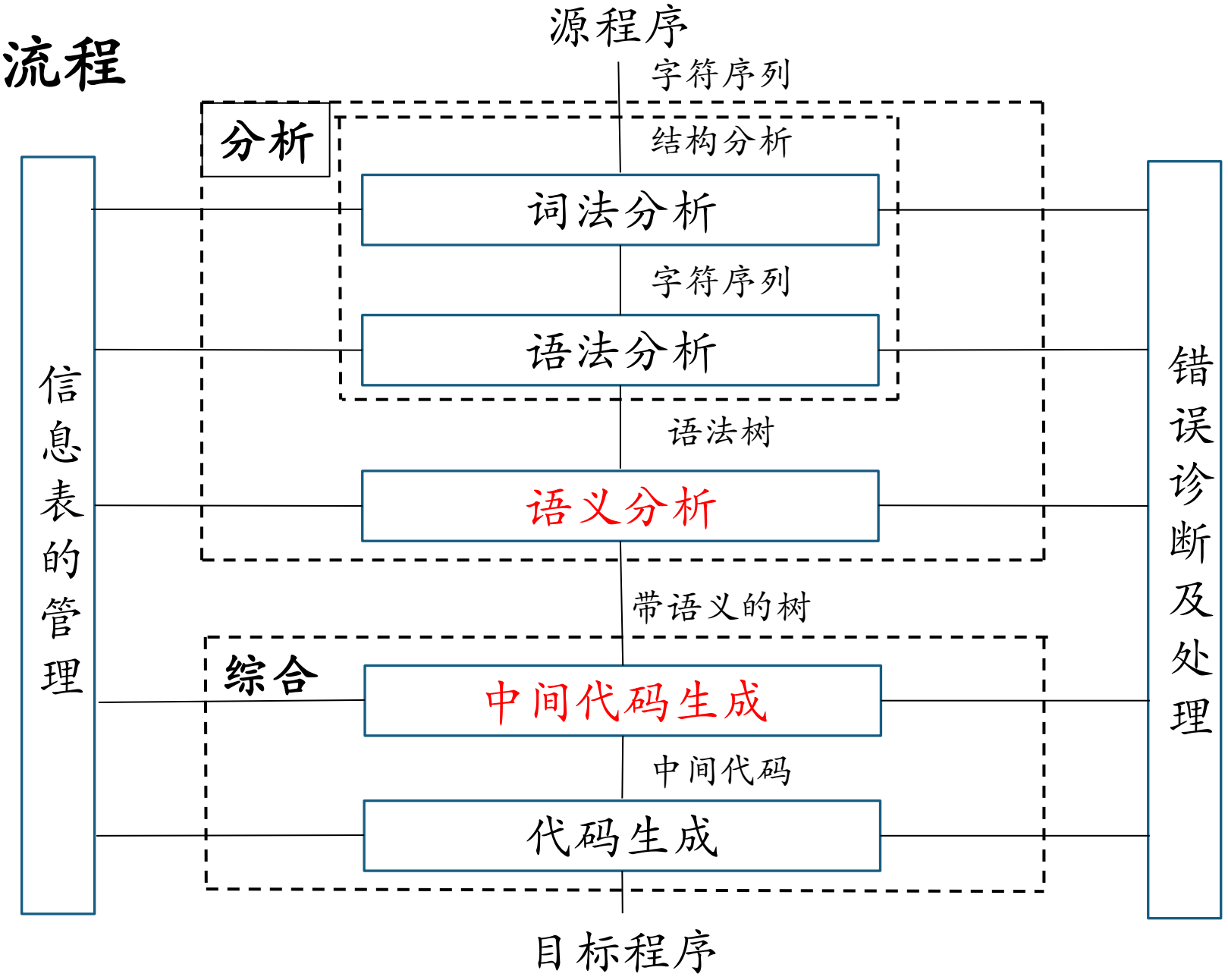
编译流程


源程序
字符序列
分析
结构分析
词法分析
字符序列
语法分析
语法树
语义分析
带语义的树
综合
中间代码生成
中间代码
代码生成
目标程序
信息表的管理
错误诊断及处理

语义分析

一个源程序经过词法分析、语法分析之后，表明该源程序在单词书写上是正确的，并且符合程序语言所规定的语法结构。

词法分析以及语法分析并未对程序内部的逻辑含义加以分析，因此编译程序接下来的工作是语义分析。

语义分析是审查源程序有无语义错误，为代码生成阶段收集信息。

语义分析的任务是对结构上正确的源程序进行上下文有关性质的审查，即进行类型审查。

类型审查是审查每个算符是否具有语言规范允许的运算对象，当不符合语言规范时，编译程序应报告错误。

第一次对源程序的语义作出解释，引起源程序质的变化。

语义的类型

编译中的语义包含以下两个类型：

静态语义：

静态语义是对程序约束的描述，这些约束无法通过抽象语法规则来妥善地描述，实质上就是语法规则的良形式条件，它可以分为类型规则和作用域/可见性规则两大类

例如：类型检查：如操作数类型匹配

控制流检查：控制语句转向点是否合法？goto、break、…

一致性检查：数组维数是否正确、变量是否已经定义、…

动态语义：

程序单位描述的计算

例如：除零溢出错误、数组下标越界、无效的指针等。

静态语义分析

静态语义分析基本功能有：类型检查、控制流检查、唯一性检查、关联名字检查、名字的作用域分析等等。

类型检查。验证程序中执行的每个操作是否遵守语言的类型规则，编译程序必须报告不符合类型规则的信息。必要时进行相应的类型转换。

控制流检查。控制流语句必须使控制转移到合法的地方。例如，在 C 语言中 break 语句使控制跳离包含该语句的while、for 或 switch 语句。如果不存在包含它的语句，则报错。

静态语义分析

静态语义分析基本功能有：确定类型、类型检查、控制流检查、唯一性检查、关联名字检查、识别含义等等。

一致性检查。在很多场合要求对象只能被定义一次。例如Pascal语言规定同一标识符在一个分程序中只能被说明一次，同一case语句的标号不能相同，枚举类型的元素不能重复出现等等。

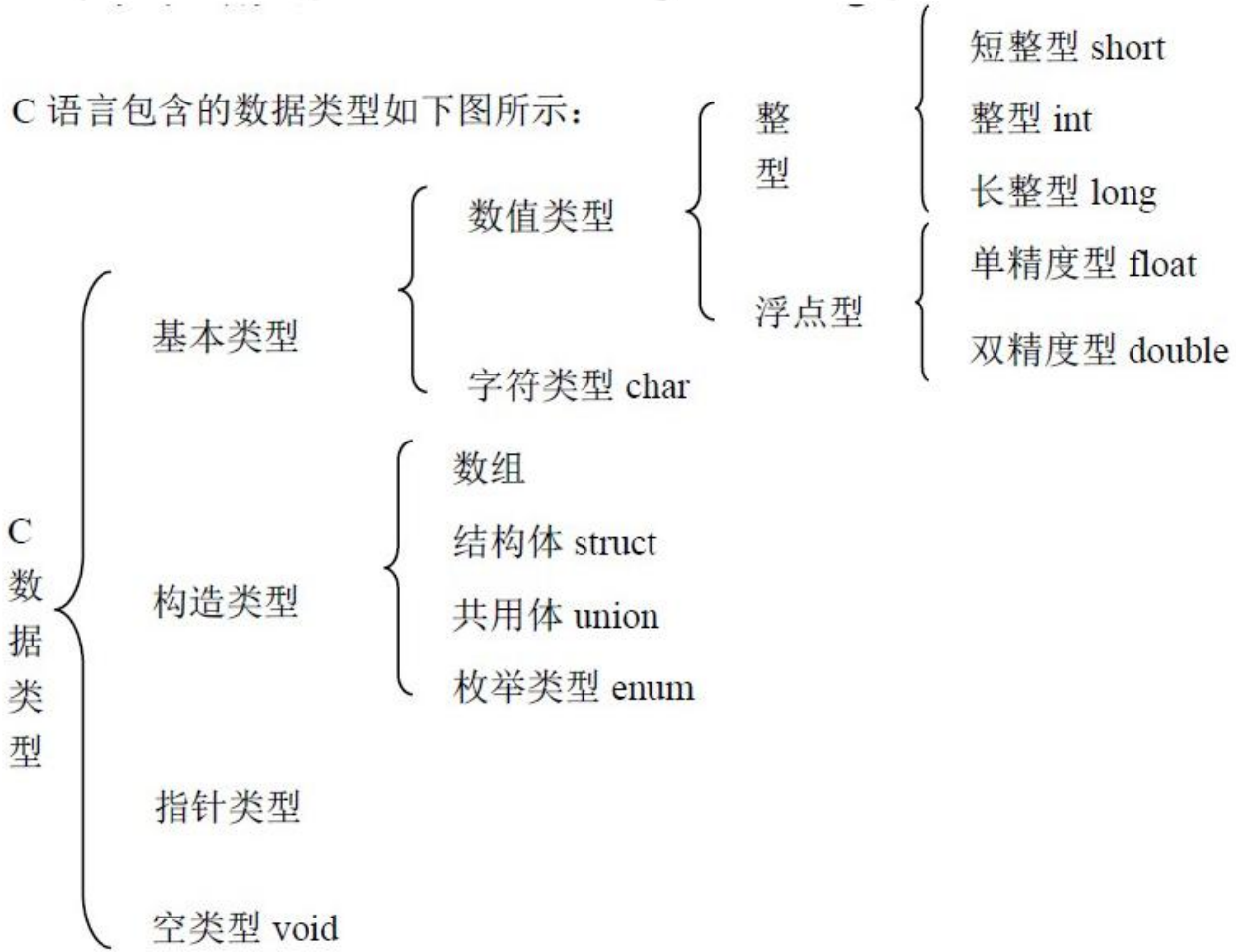
关联名字检查。有时，同一名字必须出现两次或多次。例如，Ada 语言程序中，循环或程序块可以有一个名字，出现在这些结构的开头和结尾，编译程序必须检查这两个地方用的名字是相同的。

名字的作用域分析。分析某个变量或者子程序的作用域。

数据类型

- 基本(初等)数据类型：数值数据，逻辑数据，字符数据，指针类型等
- 复合数
- 抽象数

C 语言包含的数据类型如下图所示：



作用域

Global scope

在任何函数和类定义之外的区域。

所声明的标识符具有全局作用域。

Class scope

特指类定义的作用域。

所声明的类类型具有全局作用域。

所声明的类成员具有类作用域。

作用域类型

Function scope

特指函数形参表中参数的作用域。

所声明的函数具有全局作用域,或类作用域,或其它局部作用域。

所声明的形参具有函数作用域。

Local scope

由定界符{ }/begin end分隔，定义在过程(函数)体或其它局部作用域(分程序)内。

所声明的标识符具有局部作用域。

语义分析的描述

属性文法：描述语义规则。

语法制导翻译：在语法分析的同时，执行语义子程序：

- 检查静态语义
- 翻译(生成)中间(目标)代码

静态语义分析的环境

符号表：

为语义分析提供类型、作用域等信息。

为代码生成提供类型、作用域、存储类别、存储（相对）位置等信息。

语义分析的实现工具

Yacc(Bison)对语法制导翻译的支持：

Yacc(Bison)对语义值的支持

通过$伪变量访问文法符号的语义值。

Yacc(Bison)对语义动作的支持

在语法规则的动作中调用语义子程序：计算语义值，诊断语义错误，生成中间代码等。

符号表

* 符号表是一种供编译器用于保存有关源程序构造的各种信息的数据结构。
* 符号表中通常保存如下信息：

  符号名，变量的名字、函数的名字、过程的名字

  符号类型，常量、变量、数组、整型、布尔型

  地址码，常量或变量等的地址

  层次信息，程序结构上被定义的层次

  行号信息，标识符在程序中的行号

  存储类别，关键字或变量定义出现的位置

  存储位置，变量在存储区的位置

符号表的作用

* 作用：将信息从声明的地方传递到实际使用的地方

* 声明阶段：将信息存进符号表中
* 赋值阶段：取出符号表中的信息

符号表的形式

* 每个名字对应一个表项
* 一个表项包括名字域和信息域

名字	属性信息

* 属性域

  包含多个子域及标志位

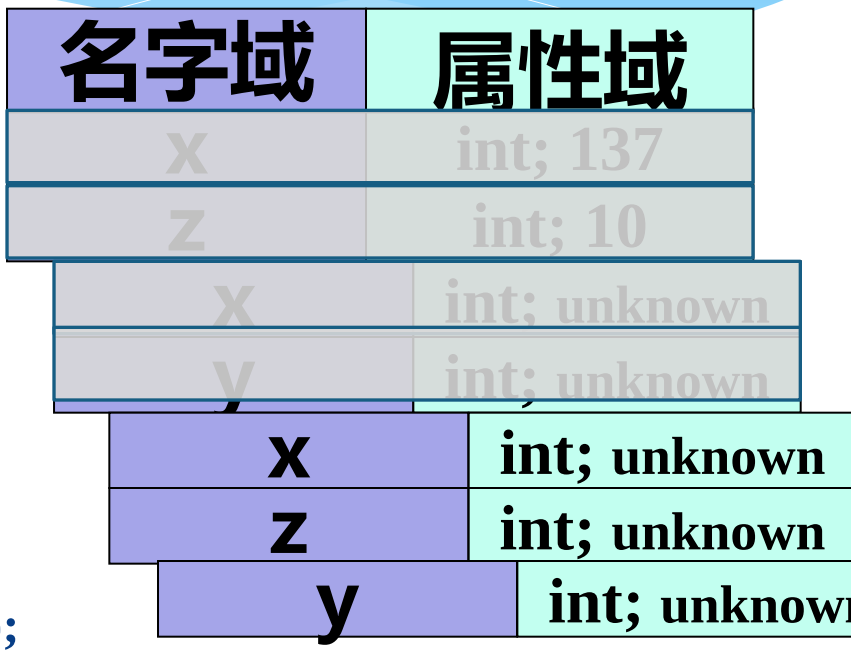
  类型、值、存储大小、相对地址、

  形参标志、说明标志、赋值标志

符号表

核心问题：对于抽象语法树中的被引用的标识符
如何在符号表中找出其对应的属性?

```
0: int x = 137;
1: int z = 10;
2: int MyFunction(int x, int y) {
3:         printf("%d,%d,%d\n", x, y, z);
4:         {
5:                 int x, z;
6:                 z = y;
7:                 x = z;
8:                 {
9:                         int y = x;
10:                        printf("%d,%d,%d\n", x, y, z);
11:                }
12:        }
13:        z = x + z;
14:
15:        printf("%d,%d,%d\n", x, y, z);
16: }
```



声明：录入符号表
引用：访问符号表

如何管理符号表?

符号表

```
0: int x = 137;
1: int z = 10;
2: int MyFunction(int x, int y) {
3:         printf("%d,%d,%d\n", x, y, z);
4:         {
5:                 int x, z;
6:                 z = y;
7:                 x = z;
8:                 {
9:                         int y = x;
10:                        printf("%d,%d,%d\n", x, y, z);
11:                }
12:        }
13:        z = x + z;
14:
15:        printf("%d,%d,%d\n", x, y, z);
16: }
```

名字域	属性域
x	int; 137
z	int; 10
x	int; unknown
y	int; unknown
x	int; unknown
z	int; unknown
y	int; unknown

语义分析的方法

* 语法制导翻译：

  在语法分析时不建立抽象语法树，而是直接生成中间代码，适用于文法比较简单的语言

* 抽象语法树：

  以抽象语法树为基础进行语义分析及中间代码生成

  在语法分析过程中建立抽象语法树是目前编译器的标准步骤。

属性文法

1968年，Knuth首次提出了属性文法（也称为属性翻译文法）这一概念。

这种文法以上下文无关文法为基础，为文法中的终结符和非终结符配备了若干相关的“值”（也称为属性）。这些属性代表与文法符号相关的信息。

属性文法A(attribute grammar)是一个三元组:A=(G,V,F),其中

G:是一个上下文无关文法,

V:有穷的属性集,每个属性与文法的一终结符或非终结符相连,

F:关于属性的计算规则.每个规则与一个产生式相联.而此规则只引用该产生式左端或右端的终结符或非终结符相联的属性。

属性

属性有不同的类型，可以象变量一样地被赋值.赋值规则附加于语法规则之上。赋值与语法同时进行，赋值过程就是语义处理过程。

在推导语法树的时候，诸属性的值被计算并通过赋值规则层层传递。有的从语法规则左边向右边传，有的从右边向左边传。语法推导树最后完成时，就得到开始符号的属性值。也就是整个程序的语义。

属性分为两种：

继承属性(inherited attribute)和综合属性(synthesized attribute)

继承属性的计算规则由顶向下
综合属性的计算规则由底向上

综合属性

在分析树节点N上的非终结符A的综合属性是由N上的产生式所关联的语义规则来定义的。

A是产生式的左部

节点N上的综合属性只能通过N的子节点或N本身的属性值来定义。

例，表达式文法E→E+E|(E)|i中，E.val是一个综合属性。则有：

E→E1+E2　　{E.val := E1.val + E2.val}

E→(E)　　{E.val:=E1.val}

E→i　　{E.val:=i.lex}

继承属性

在分析树节点N上的非终结符B的继承属性是由N上的父节点上的产生式所关联的语义规则来定义的。

B是产生式的右部某一个符号。

节点N上的继承属性只能通过N的父节点、N本身和N的兄弟节点上的属性值来定义。

例如：添加标识符类型的语义描述。

type为一个继承属性.

产生式	语义规则
D→TL	L.type:=T.type
T→int	T.type:=integer
T→real	T.type:=real
L→L1,id	L1.type:=L.type Addtype(id.entry, L.type)
L→id	Addtype(id.entry, L.type)

属性计算方法

树遍历的属性计算方法

假设语法树已经建立，并且树中已带有开始符号的继承属性和终结符的综合属性。然后以某种次序遍历语法树，直至计算出所有属性。最常用的遍历方法是深度优先，从左到右的遍历方法。如果需要的话，可使用多次遍历（或称遍）。

一遍扫描的处理方法

与树遍历的属性计算文法不同，一遍扫描的处理方法是在语法分析的同时计算属性值，而不是语法分析构造语法树之后进行属性的计算，而且无需构造实际的语法树。

因为一遍扫描的处理方法与语法分析器的相互作用，它与下面两个因素密切相关：（1）所采用的语法分析方法（2）属性的计算次序。

S-属性文法和L属性文法

终结符号可以具有综合属性，但是不能有继承属性。终结符号的属性值由词法分析器提供。

非终结符既可以有综合属性，也可以有继承属性。但文法始符号的所有继承属性用于属性计算前的初始值。

如果一个属性文法只含有综合属性，则称为S-属性文法。

如果对于文法每个产生式$A \rightarrow X_1X_2 \ldots X_n$，其每个语义规则中的每个属性或者是综合属性，或者是$X_j(1 \leq j \leq n)$的一个继承属性且这个继承属性仅依赖于：

(1)产生式X_j在左边符号$X_1, X_2, \ldots X_{j-1}$的属性；

(2)A的继承属性。

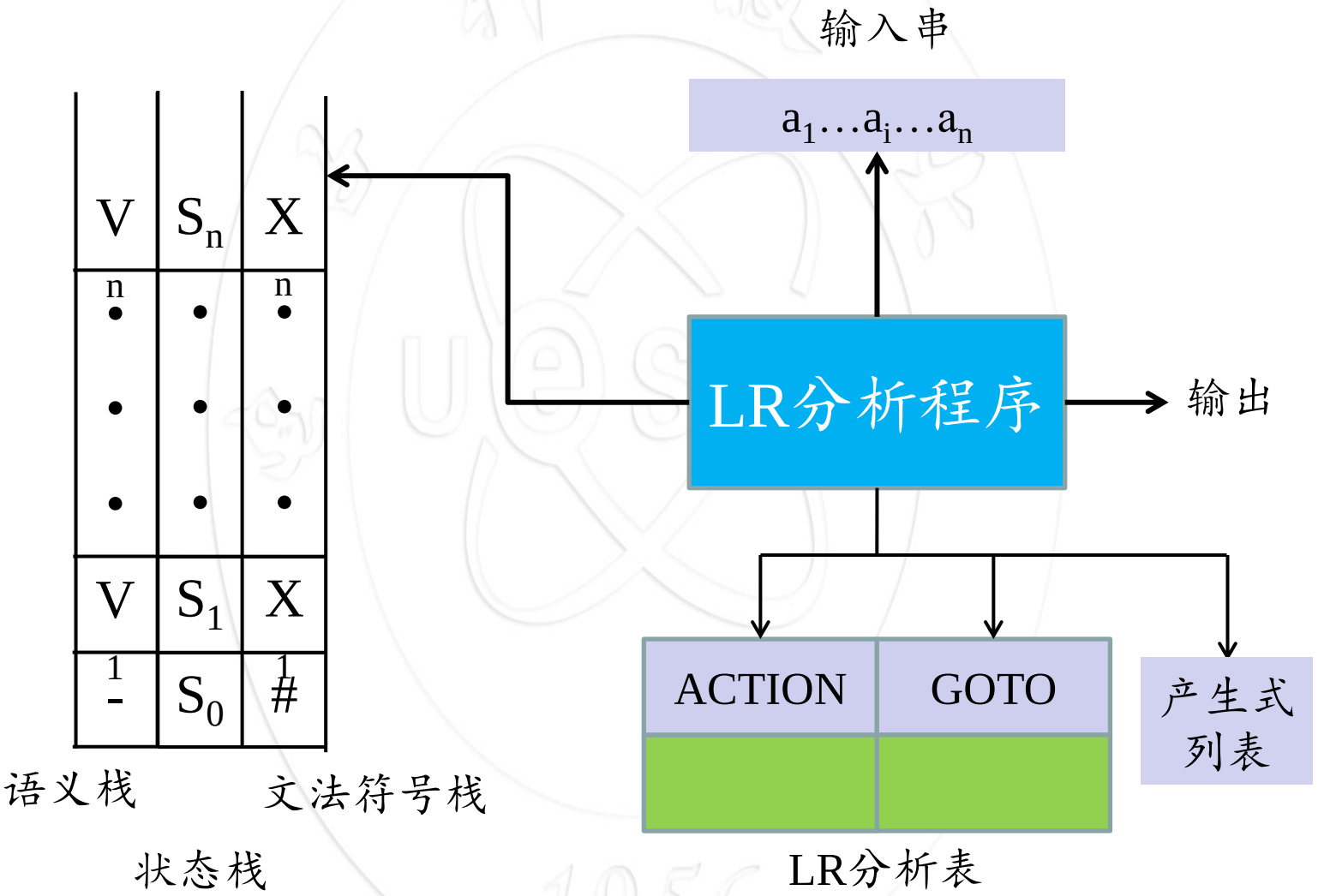
则称为L-属性文法。

S-属性文法的自下而上计算

S-属性文法只含有综合属性，因此可以在分析输入符号串的同时自下而上进行计算。分析器可以保存与栈中文法符号有关的综合属性值，每当进行规约时，新的属性值就由栈中正在规约的产生式右边符号的属性值来计算。

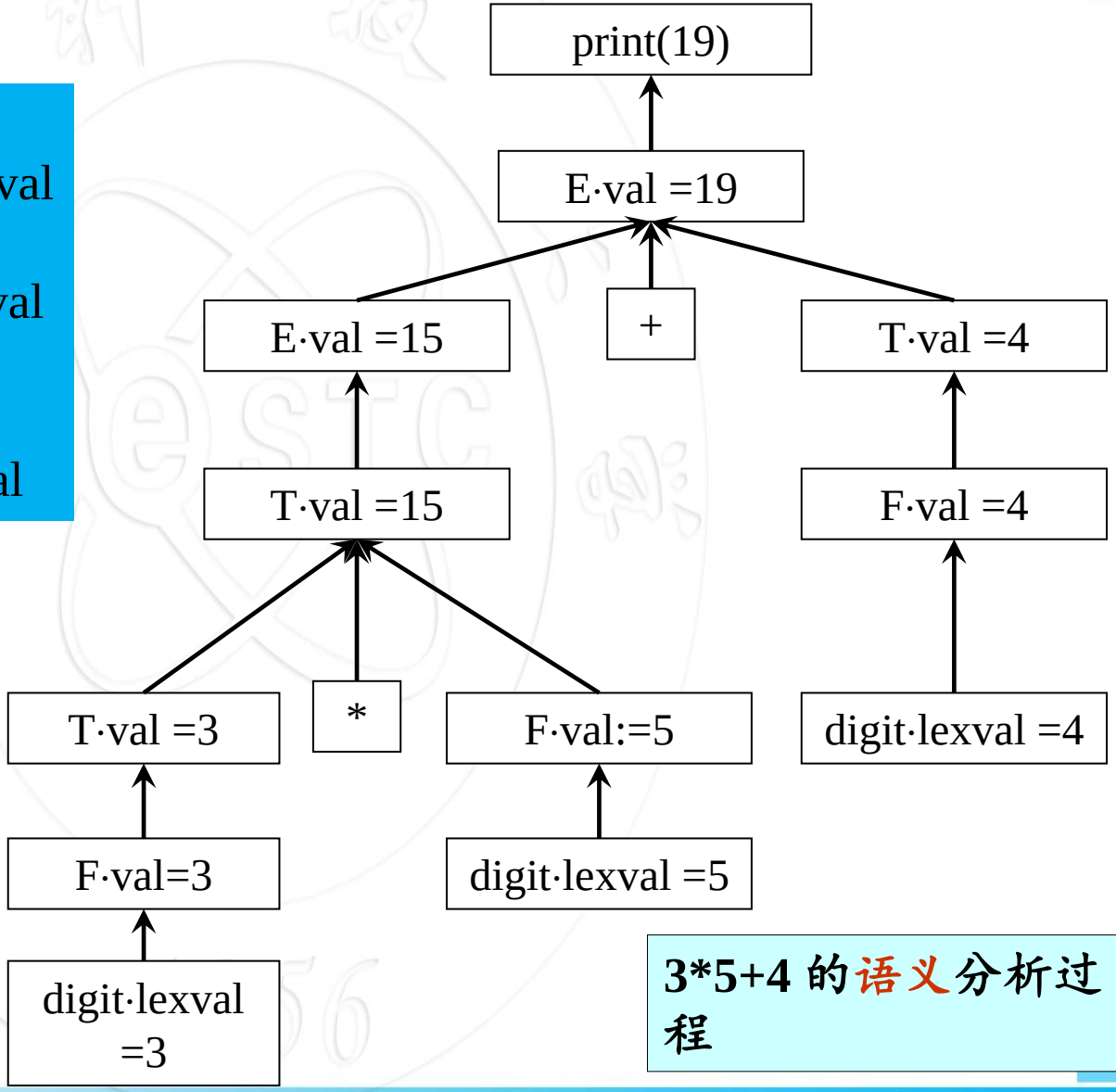
S-属性文法的翻译器通常可借助于LR分析器实现，对LR分析器进行改造，在对输入串进行语法分析的同时对属性进行计算。

S-属性文法的自下而上计算



S-属性文法的自下而上计算

$L \rightarrow E$	print(E.val)
$E \rightarrow E_1+T$	$E.val=E_1.val+T.val$
$E \rightarrow T$	E.val=T.val
$T \rightarrow T_1*F$	$T.val=T_1.val*F.val$
$T \rightarrow F$	T.val=F.val
$F \rightarrow (E)$	F.val=E.val
$F \rightarrow digit$	F.val=digit.lexval



3*5+4 的语义分析过程

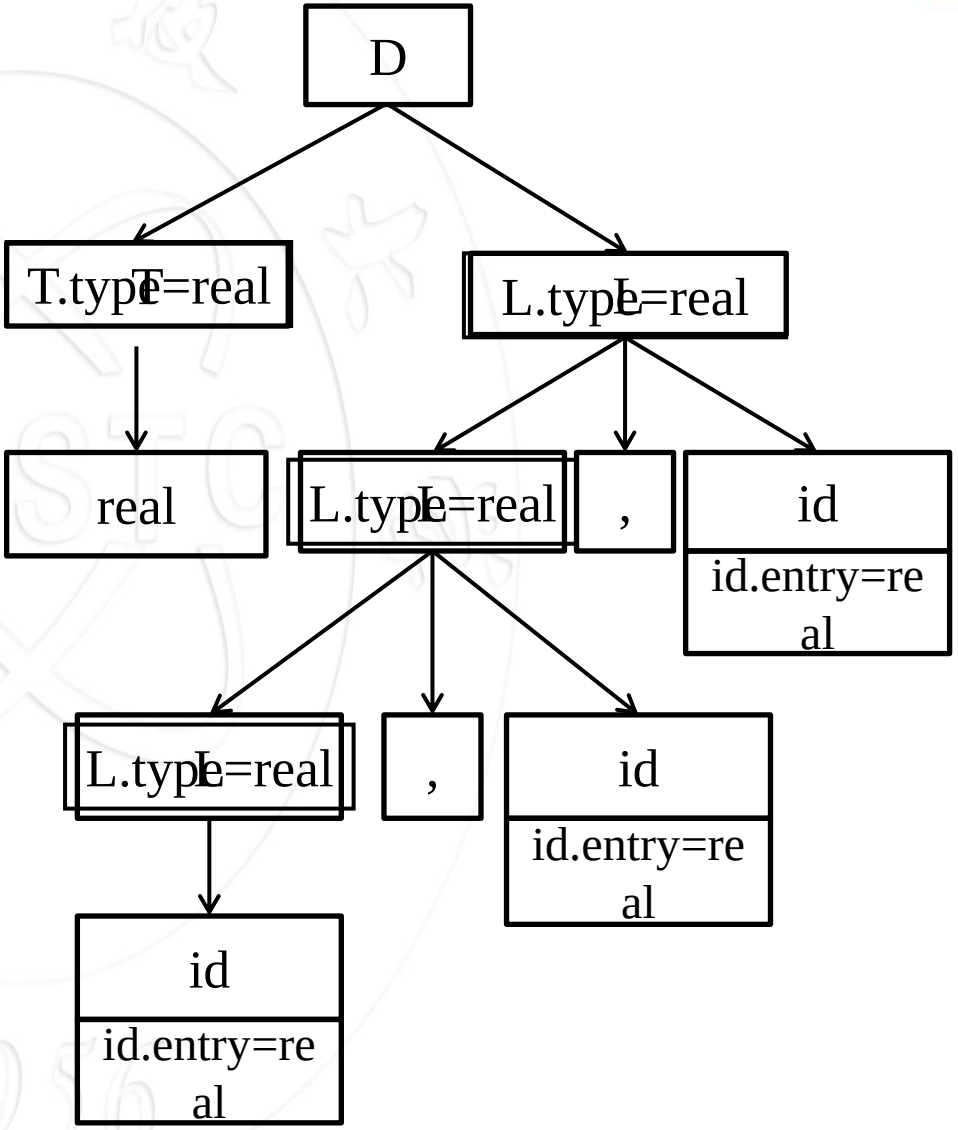
L-属性文法的自上而下计算

L-属性文法允许一次遍历就计算出所有属性值。

LL(1)自上而下语法分析过程，可以看成是深度优先建立语法树的过程，因此，可以在自上而下语法分析的同时实现L-属性文法的计算。

L-属性文法的自上而下计算

产生式	语义规则
D→TL	L.type:=T.type
T→int	T.type:=integer
T→real	T.type:=real
L→L1, id	L1.type:=L.type Addtype(id.entry, L.type)
L→id	Addtype(id.entry, L.type)



real id1, id2, id3 的语义分析过程

语法制导翻译

语法制导翻译是一种由源程序的语法结构所驱动的属性规则计算方法。

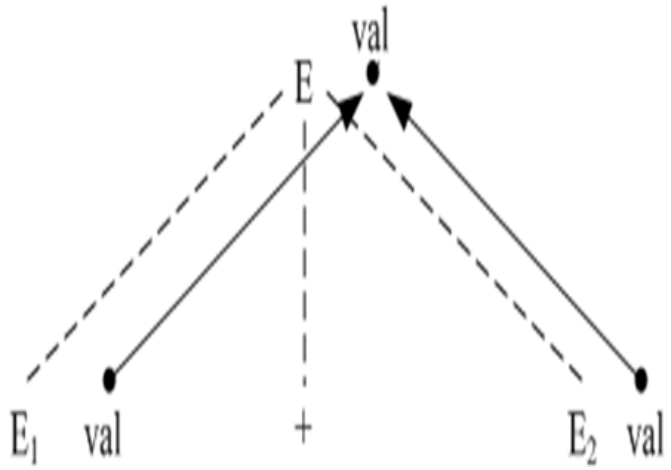
首先对输入符号串建立语法树，接着分析属性的依赖关系，当属性和属性计算规则设计的合适时，即语义规则的计算能够在一遍扫描语法树的情况下可以完成，便可以在语法分析的同时进行语义处理。这个思路为编译程序生成系统提供了一种模型。

依赖图

在语法树中，节点的综合属性和继承属性之间相互依赖，可用依赖图来进行描述。

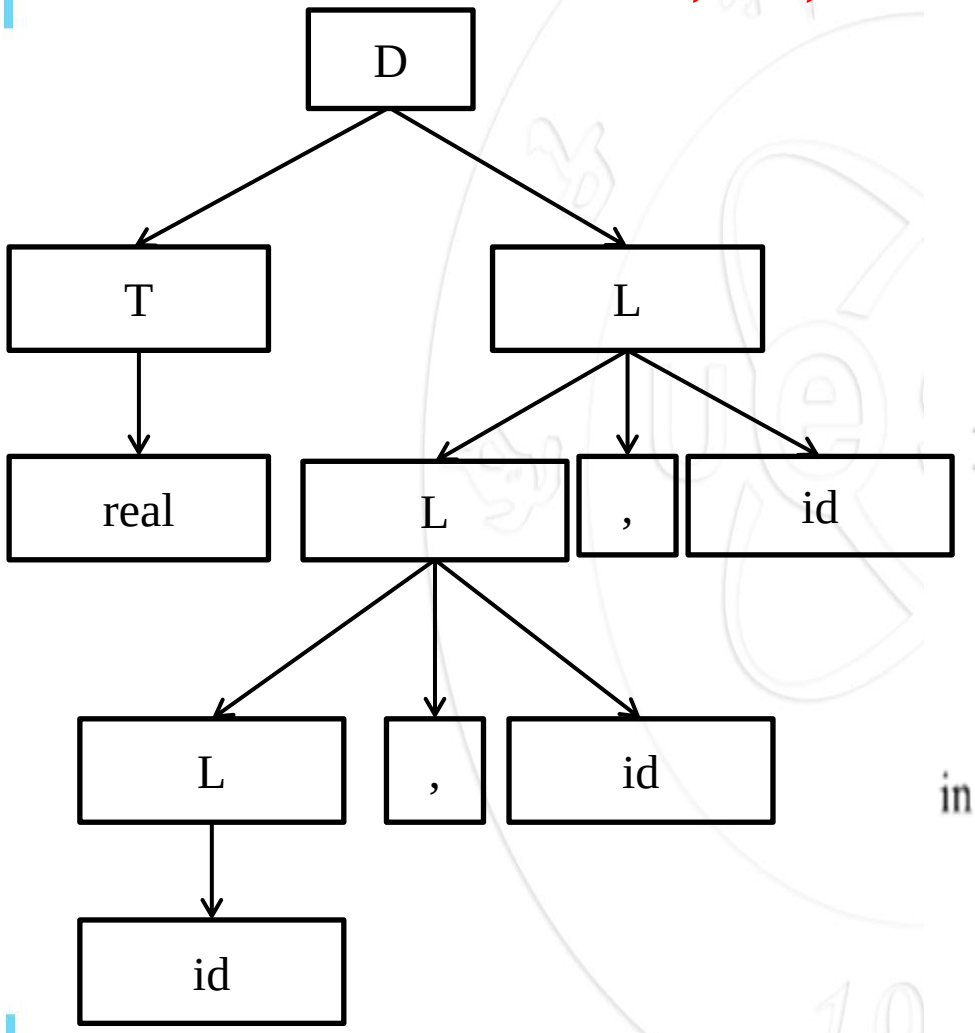
依赖图是一个有向图。如果属性a依赖于属性b，则从属性b的节点有一条有向边连接到属性b的节点。

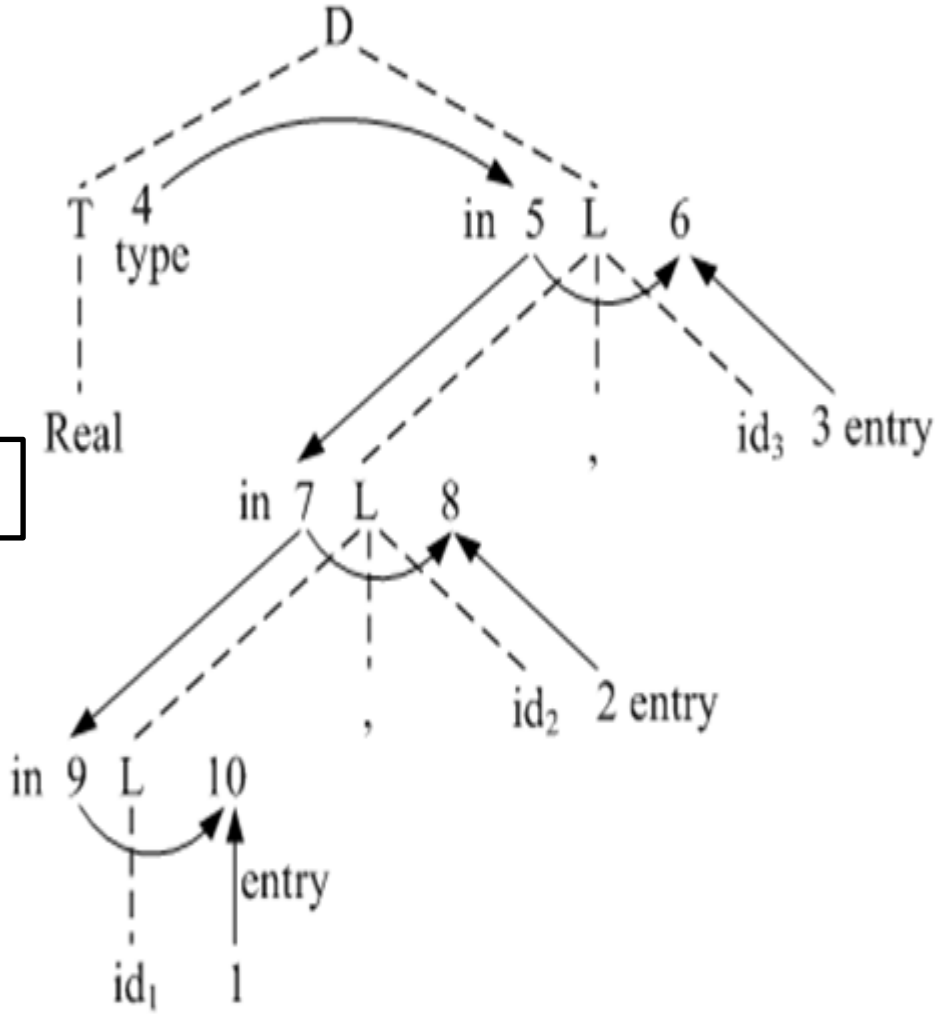
E.val是从E1.val和E2.val综合得出



依赖图

real id1, id2, id3





依赖图

语义规则执行顺序：

(1) a4=real;

(2) a5=a4;

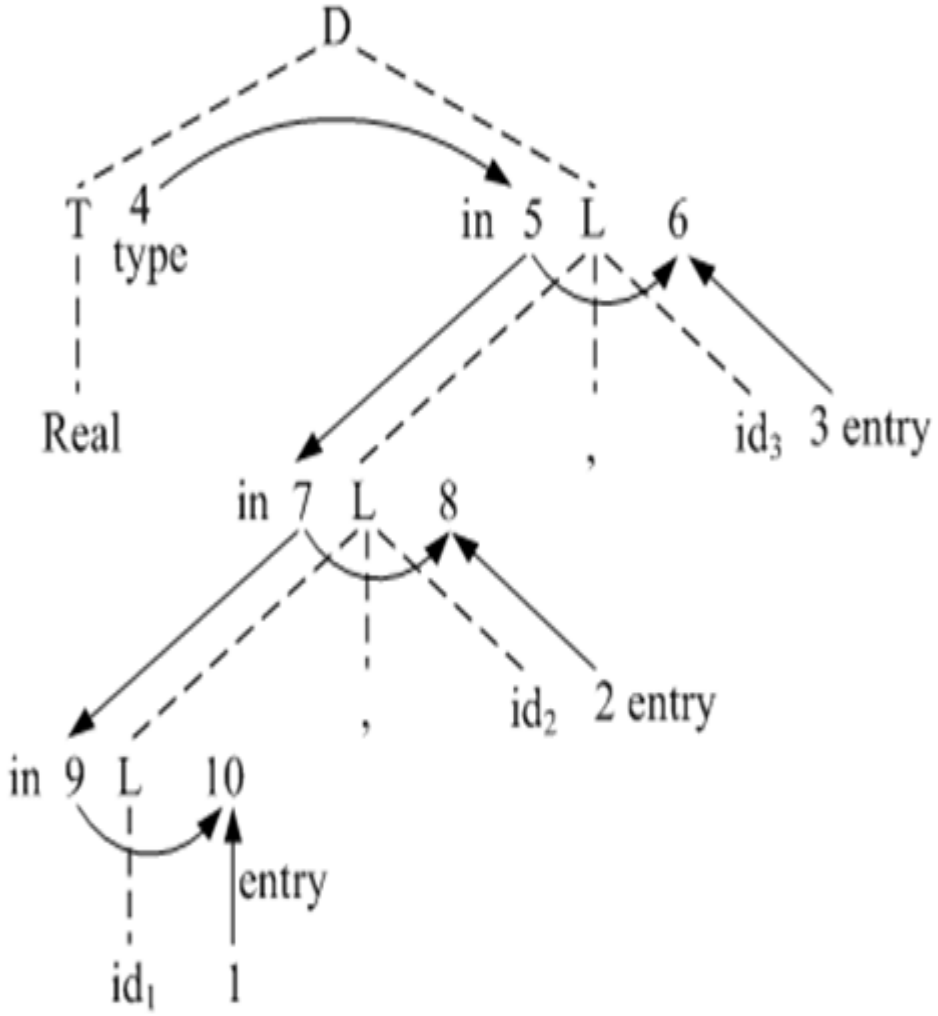
(3) addtype(id3.entry, a5);

(4) a7=a5;

(5) addtype(id2.entry, a7);

(6) a9=a7;

(7)addtype(id1.entry, a9)



3*5+4 的语法分析过程

$L \rightarrow E$

$E \rightarrow E_1+T$

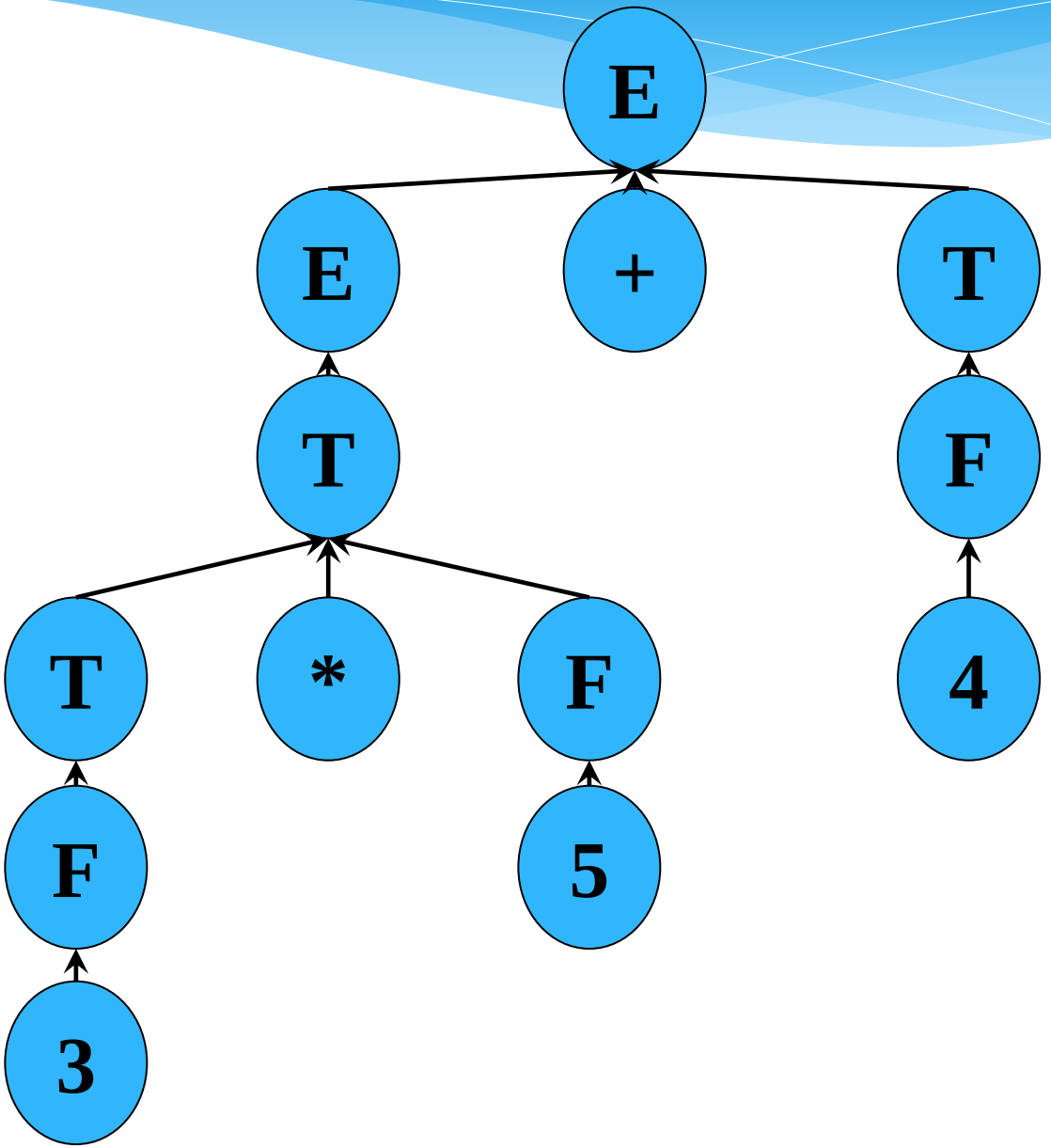
$E \rightarrow T$

$T \rightarrow T_1*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow digit$



计算器程序的语法制导翻译(直接计算结果)

$L \rightarrow E$	{print(E.val); }
$E \rightarrow E_1+T$	{E.val = E_1.val + T.val ; }
$E \rightarrow T$	{E.val = T.val ;}
$T \rightarrow T_1*F$	{T.val = T_1.val * F.val ;}
$T \rightarrow F$	{T.val = F.val ;}
$F \rightarrow (E)$	{F.val = E.val ;}
$F \rightarrow digit$	{F.val = digit.lexval ;}

语义分析--语义子程序

* 该子程序描述了一个产生式所对应的翻译工作。
* 这些工作包括：生成中间代码，查、填有关的符号表，语义检查和报错，修改编译程序某些工作变量的值等）。
* 在语法分析过程中，每当一个产生式用于匹配或进行归约时，就调用该产生式所对应的语义子程序，以完成既定的翻译任务。

语义子程序

* 语义子程序

完成语义检查、进行语义处理

检查：类型、控制流、一致性

处理：记录信息、生成中间代码

语义子程序的核心是生成中间代码

以AST为基础的翻译 示例

$L \rightarrow E$	{ L.past = E.past; }
$E \rightarrow E_1 + T$	{E.past = newExpr('+', E_1.past, T.past) ; }
$E \rightarrow T$	{E.past = T.past ;}
$T \rightarrow T_1 * F$	{T.past = newExpr('*', T_1.past, F.past) ;}
$T \rightarrow F$	{T.past = F.past ;}
$F \rightarrow (E)$	{F.past = E.past ;}
$F \rightarrow$ digit	{F.past = newNum(digit.lexval) ;}

$L \rightarrow E$ { L.past = E.past; }

$E \rightarrow E_1+T$ {E.past = newExpr('+', E_1.past, T.past) ; }

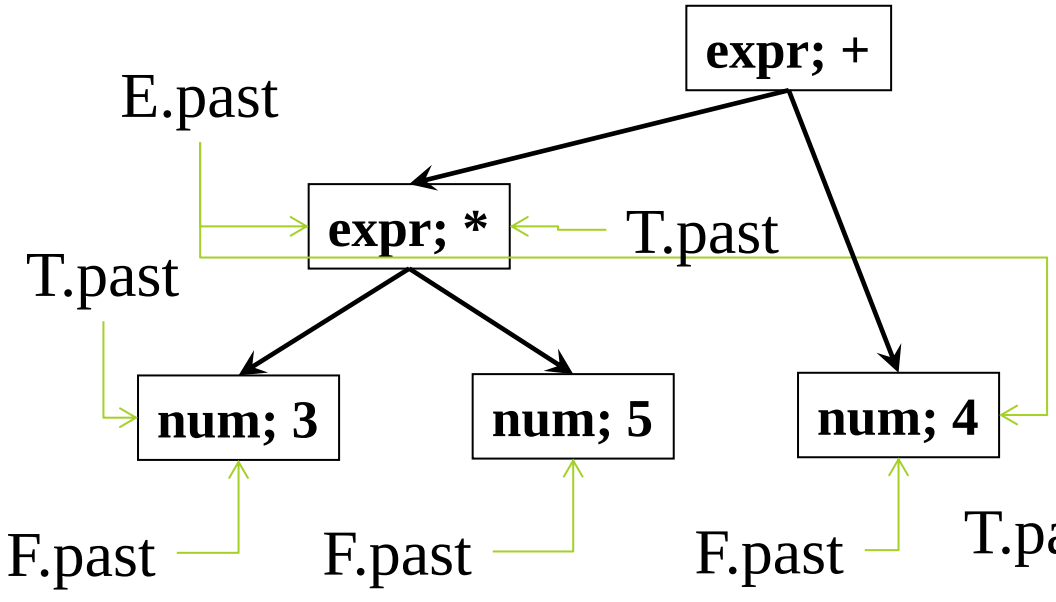
$E \rightarrow T$ {E.past = T.past ;}

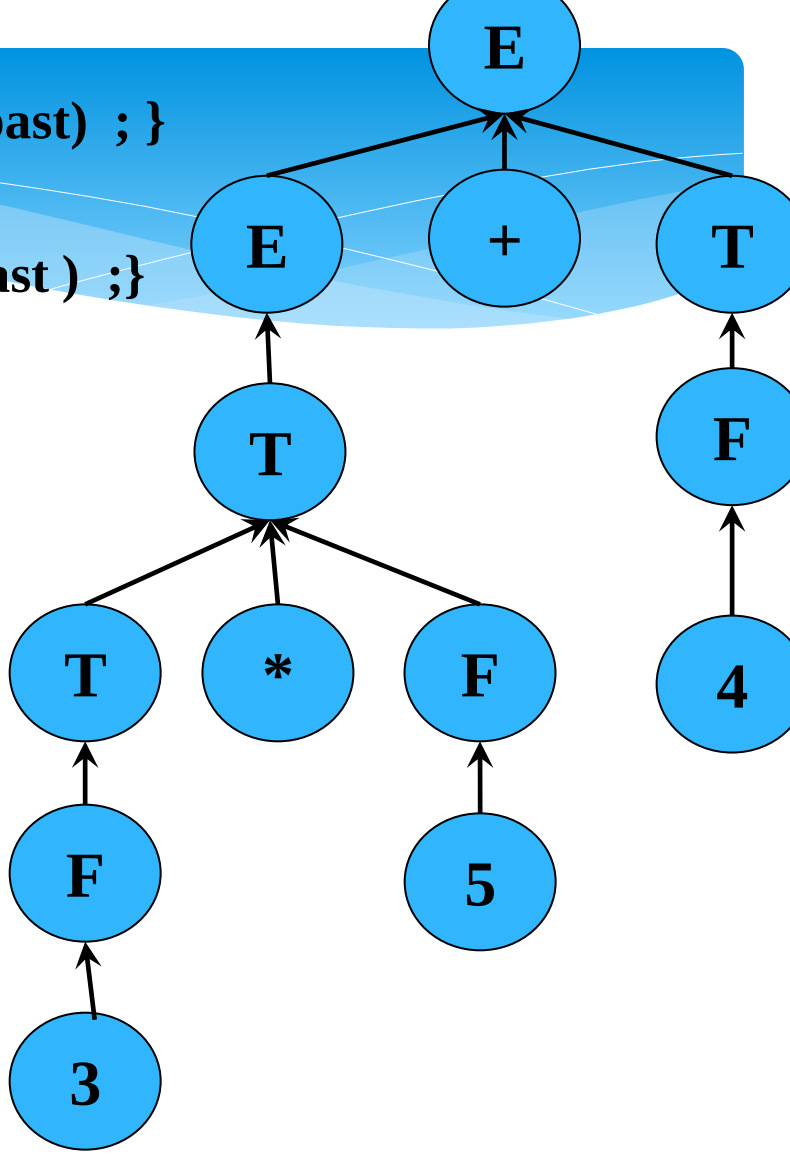
$T \rightarrow T_1*F$ {T.past = newExpr('*', T_1.past, F.past) ;}

$T \rightarrow F$ {T.past = F.past ;}

$F \rightarrow (E)$ {F.past = E.past ;}

$F \rightarrow$ digit {F.past = newNum(digit.lexval) ;}





3*5+4 AST 建立过程

语义变量和语义函数

语义值的表示

X.type、X.category、X.address、…

常用语义变量:

1) i.name: 语义变量，与终结符 i 相关联

表示 i 对应的标识符字符串

语义变量和语义函数

2) E.place: 语义变量，与非终结符E相关联

表示E对应的变量在符号表的位置

或整数编码(临时变量)

采用变量名或临时变量名表示

产生式：$E \rightarrow E_1+E_2$

emit(E.place, E_1.place, op, E_2.place)

语义变量和语义函数

3) newtemp():语义函数

产生一个新的临时变量

返回其地址(整数编码)

符号表的位置

采用临时变量名表示：t1, t2, …

4) entry(i.name): 语义函数

对变量 i 查符号表，返回i在符号表中的位置

语义变量和语义函数

5) **emit(RESULT, ARG1, oper, ARG2)** 或

emit(oper, ARG1, ARG2, RESULT)

语义函数

产生四元式并填入四元式表中

同时四元式编号增加1

中间代码概述

中间代码(Intermediate Code, Intermediate representation, Intermediate language) 是源程序的一种内部表示，其复杂性介于源语言和目标机语言之间。

中间代码的作用：

(1) 使编译程序的逻辑结构更加简单明确

(2) 利于进行与目标机无关的优化

(3) 利于在不同目标机上实现同一种语言

中间代码的形式：

逆波兰式、四元式、三元式、间接三元式、树

中间代码的层次

中间代码按照其与高级语言和机器语言的接近程度，可以分成以下三个层次：

高级：最接近高级语言，保留了大部分源语言的结构。

中级：介于二者之间，与源语言和机器语言都有一定差异。

低级：最接近机器语言，能够反映目标机的系统结构，因而经常依赖于目标机。

中间代码的层次

源语言（高级语言）	中间代码（高级）	中间代码（中级）	中间代码（低级）
float a[10][20]; a[i][j+2];	t1 = a[i, j+2]	t1 = j + 2 t2 = i * 20 t3 = t1 + t2 t4 = 4 * t3 t5 = addr a t6 = t5 + t4 t7 = *t6	r1 = [fp - 4] r2 = [r1 + 2] r3 = [fp - 8] r4 = r3 * 20 r5 = r4 + r2 r6 = 4 * r5 r7 = fp – 216 f1 = [r7 + r6]

中间代码的形式

逆波兰式

也称为后缀式，由波兰逻辑学家Lukasiewicz提出的一种表示表达式的方法，该方法把操作数放在前面，把运算符放在后面。

a+b => ab+　　a+b+c => ab+c+　　a * (b + c / d)　　abcd / + *

图表示法

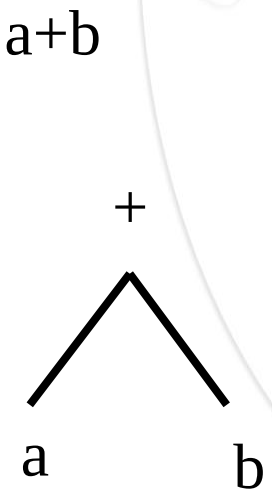
主要包含抽象语法树和DAG两种

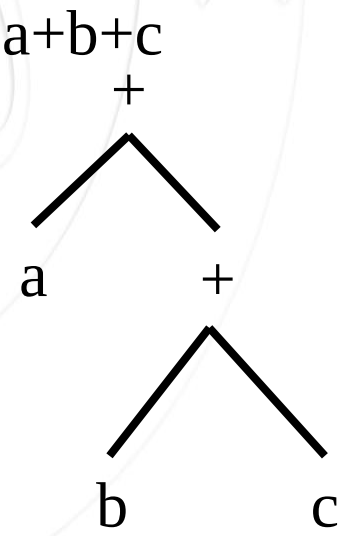
中间代码的形式

抽象语法树

在语法树中去除了那些对翻译不必要的信息，从而获得更为高效的中间代码表示。

抽象语法树构造原则：操作符和关键字都不作为叶子节点，而是作为内部节点出现。



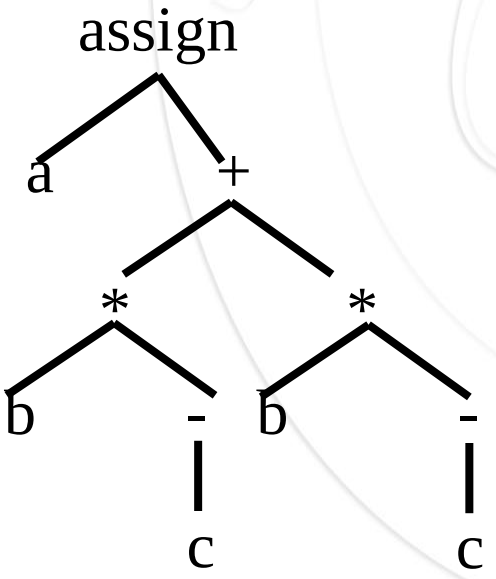


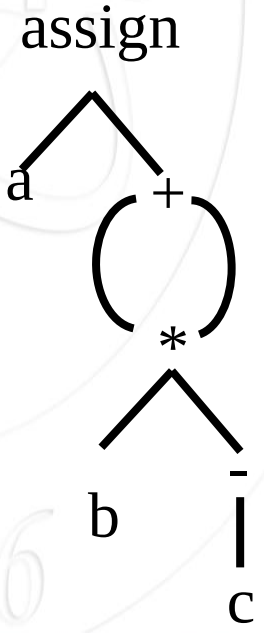
中间代码的形式

DAG(Directed Acyclic Graph)

无循环有限图，对于表达式中的每一个子表达式，DAG中都有一个相应的节点，一个内部节点代表一个操作符，其子节点代表操作数。与抽象语法树不同的是，DAG中代表公共子表达式的节点具有多个父节点。

例：a = b * -c + b * -c





中间代码的形式

三地址代码

三地址代码形式最多包含3个地址，两个用来表示操作数，一个用来存放结果。一般形式为：

$$x := y \ op \ z$$

其中，y和z表示名字、常数或编译过程中产生的临时变量。x表示名字或临时变量，op表示各种运算符。每个语句的右边只能有一个运算符。

由于每个语句右边只能有一个运算符，因此需要用多条语句来表示一个抽象语法树或者DAG。

中间代码的形式

三地址代码

例：a = b * -c + b * -c

T1 = -c
T2 = b * T1
T3 = -c
T4 = b*T3
T5 = T2 + T4
a = T5

T1 = -c
T2 = b * T1
T5 = T2 + T2
a = T5

三地址代码

三地址代码-四元式

一般有3种方式来表示三地址语句：四元式、三元式、间接三元式。

一个四元式是具有四个域的记录结构，一般形式为：

(序号) (op, arg1, arg2, result)

表示对arg1和arg2执行op所指定的运算并将结果放到result中。

T1 = -c	(1) (uminus, c, -, T1)
T2 = b * T1	(2) (*, b, T1, T2)
T3 = -c	(3) (uminus, c, -, T2)
T4 = b*T3	(4) (*, b, T3, T4)
T5 = T2 + T4	(5) (+, T2, T4, T5)
a = T5	(6) (=, T5, -, a)

三地址代码

三地址代码-三元式

一个三元式是具有三个域的记录结构，一般形式为：

(序号) (op, arg1, arg2)

表示对arg1和arg2执行op所指定的运算，通过计算临时变量值的代码位置来引用该临时变量，从而减少临时变量带来的时空开销。

T1 = -c	(1) (uminus, c, -)
T2 = b * T1	(2) (*, b, (1))
T3 = -c	(3) (uminus, c, -)
T4 = b*T3	(4) (*, b, (3))
T5 = T2 + T4	(5) (+, (2), (4))
a = T5	(6) (=, a, (5))

源程序的基本组成单位

关键字：表示语句性质，反映语句结构；

标识符：表示程序中的各种实体；如变量名、常量名、过程名、函数型，文件名，数组名等

标识符的类型反映了标识符的语义特征属性，是翻译的语句。因此需要记录与符号表(namelist)中。

在整个编译过程中动态地采集、记录、变更、引用。

语句翻译的设计要点

1、确定语句的目标结构；

2、确定中间代码；

3、根据目标结构和语义规则，构造语义子程序(转换翻译程序)；

4、涉及的具体实现技术。

语句翻译的设计要点

确定语句的目标结构；

if e then S

1	e.code
•••	
i	测e.code
i+1	J_F, , , m
i+2	S.code
•••	
n	

高级语言的语句分类

说明语句—定义各类名字的属性

说明语句翻译：将所定义的名字的各种属性登记到符号表中

可执行语句—用于完成指定的功能，涉及到指令代码

可执行语句翻译：首先应根据各源语句的语法结构和语义设计出它的目标代码结构，找出源与目标的对应关系，给出对信息(数据表示)描述和从源到目标的变换算法。语义子程序则根据变换方法进行翻译。

说明语句

说明语句用于定义变量，大多数高级语言都要求变量先定义后使用。说明语句主要说明类型等属性信息。在翻译时主要动作是查看/填写符号表

对每个局部名字，我们都将在符号表中建立相应的表项，并填入有关的信息如类型、在存储器中的相对地址等。相对地址是指对静态数据区基址或活动记录中局部数据区基址的一个偏移量。

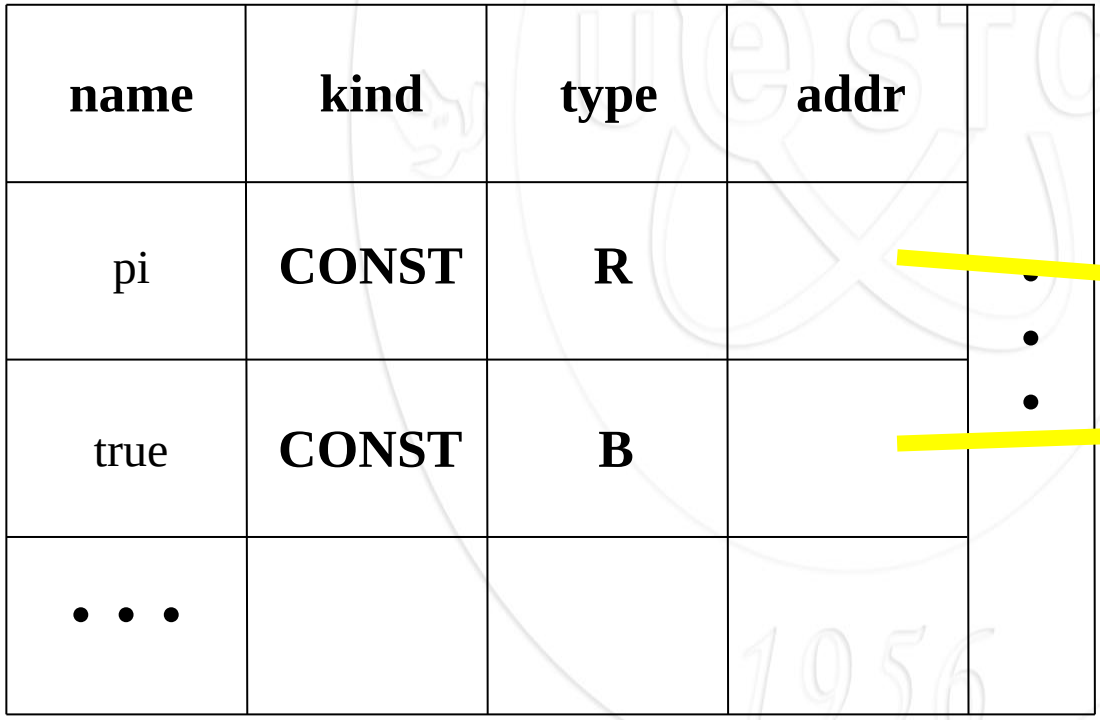
当产生中间代码地址时，对目标机一些情况做到心中有数是有好处的。例如，假定在一个以字节编址的目标机上，整数必须存放在4的倍数的地址单元，那么，计算地址时就应以4的倍数增加。

常量说明语句的翻译

一般形式： #define 标识符 常量

CONST pi = 3.14; true = 1;

name	kind	type	addr
pi	CONST	R	
true	CONST	B	
• • •			



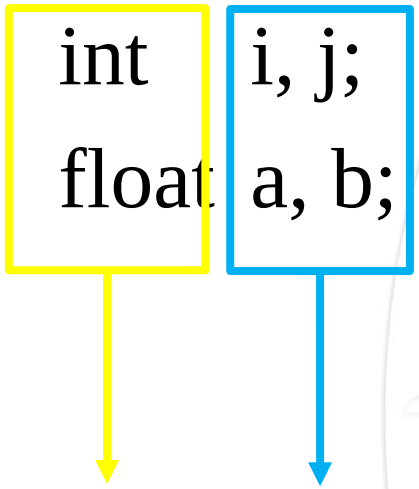
	value
1	
2	

常量说明语句的语义子程序

CONST_DEF →CONST
<con_list>;
<con_list> →<con_list>;CD
<con_list> →CD
CD →id=num

```
num{
        num.ord=look_con_table(num.lexval);
          id.ord=num.ord;
         id.type= num.type;
         id.kind=constant;
         add(id.entry; id.ord; id.type; id.kind) }
```

简单说明语句



符号表

name	kind	type	addr	
i	V	I	0	• • •
j	V	I	4	
a	V	R	8	
b	V	R	16	
• • •				

D → int id
D → float id
D → D, id;

简单说明语句的翻译方案

D → int id {fill(ENTRY(id), int); D.AT = int}
D → float id {fill(ENTRY(id), float); D.at = float}
D → D, id; {fill(ENTRY(id), D1.AT);
D.AT = D1.AT}

其中：

D.AT：设为非终结符D的语义变量，记录说明语句规定的量的某种属性。

fill(P, A): 函数。把属性A填入P所指的符号表入口的相应数据项中。

ENTRY(i): 函数。给出i所代表的量在符号表中的入口。

复合类型说明语句

语义处理包含：

(1) 结构成员与该结构相关

(2) 结构的存储方式：一个结构的所有成员项连续存放

(3) 结构的引用是结构成员的引用，不能整体引用；

(4) 所有成员信息单独记录;

```
struct data{
        int year;
        int
month;
        int day;
        int c[10];
} today, yesterday;
```

复合类型说明语句

```
struct date{
        int year;
        int month;
        int day;
        int c[10];
} today, yesterday;
```

name	kind	type	length	offset
year	V	I	4	0
month	V	I	4	4
day	V	I	4	8
c	array	I	40	12
• • •				52

name	kind	type	addr
date	struct	• • •	
• • •			

过程中的说明语句

在C、Pascal及FORTRAN等语言的语法中，允许在一个过程中的所有说明语句作为一个组来处理，把它们安排在一组数据区中。从而我们需要一个全程变量如offset来跟踪下一个可用的相对地址的位置。

过程中说明语句的翻译

P→D	{offset 0}
D→D;D	
D→id:T	{enter(id.name,T.type,offset); offset offset+T.width}
T→integer	{T.type integer; T.width 4}
T→real	{T.type real; T.width 8}
T→array[num]of T_1	{T.type array(num.val,T_1.type); T.width um.val×T_1.width}
T→↑T_1	{T.type pointer(T_1.type); T.width 4}

赋值语句

赋值语句是用来赋给某变量一个具体值的语句。在程序中，赋值语句是最基本的语句。

赋值语句的功能是完成等号右端的表达式所规定的运算，并将计算结果赋给左端的变量。

赋值语句中的表达式的类型可以是整型、实型、数组和记录。

语义变量及函数定义

定义的一些语义变量和过程：

GENCODE(OP,ARG1,ARG2,RESULT)：语义过程，产生一个四元式，并填入四元式表，编号自动增1。

NEWT：函数，返回一个临时变量序号。在翻译可执行语句的过程中，可能需要使用临时变量，假定用NEWT过程来申请临时变量T_i，每申请一次，i增1。

ENTRY(i)：函数，查找变量i的入口地址；检查id.name是否在符号表中，如在则返回一指向该登陆项的指针，否则返回NULL

E.PLACE：与给终结符E相联系的语义变量，可能是某变量的入口地址，或者为临时变量序号。

NXQ：即将要生成的四元式的编号。

简单赋值语句

简单赋值语句：只有简单变量，且变量类型相同

例如： x = 2;

简单赋值语句的代码结构

id = value;

“=”左右部需要做类型检查和转换

inttoreal();

简单赋值语句

赋值语句的一般形式：

id = aexpr;

x = a * b + c;

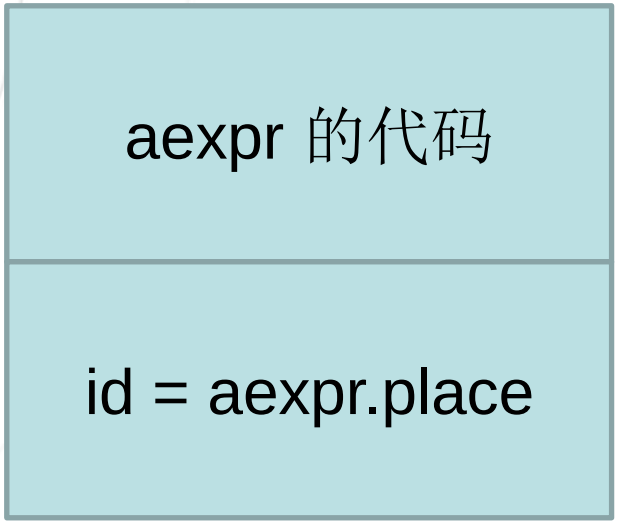
$S \rightarrow id=E$

$E \rightarrow E_1 + T \mid T$

$T \rightarrow T_1 * F \mid F$

$F \rightarrow -F \mid id \mid (E)$

赋值语句的处理集中在表达式的处理上！



赋值语句的翻译

编号	产生式	语义规则
（1）	$S \rightarrow id=E$	{gencode(=, E.place, _, entry(id)}
（2）	$E \rightarrow E_1 + T$	{E.place = newtemp(); gencode(+, E1.palce, T.place, E.place)}
（3）	$E \rightarrow T$	{E.place = newtemp; E.place = T.place}
（4）	$T \rightarrow T_1 * F$	{T.place = newtemp(); gencode(*, T1.palce, F.place, T.place)}

赋值语句的翻译

编号	产生式	语义规则
（5）	T → F	{T.place = newtemp; T.place = F.place}
（6）	F → -F	{F.place = newtemp; gencode(@, F.place, _, F1.place)}
（7）	F → id	{F.place = newtemp; F.place = entry(id)}
（8）	F → (E)	{F.place = newtemp; F.place = E.place}

布尔表达式的作用及形式

布尔表达式的作用

（1）用作控制语句中的条件表达式

（2）用在逻辑赋值语句中

布尔表达式的形式

（1）单个布尔量（布尔变量或关系表达式）

（2）布尔量的运算：not, and, or，优先级比关系运算高

（3）关系运算：>, <, ==, <=, >=

布尔表达式的文法描述

(1) <BE> → <BE> or <BT>

(2) <BE>or → <BE> or

(3) <BE> → <BT>

(4) <BT> → <BT> and <BF>

(5) <BT> and → <BT> and

(6) <BT> → <BF>

(7) <BF> → not <BF>

(8) <BF> → (<BE>)

(9) <BF> → <AE> rop <AE>

(10) <BF> → id rop id

(11) <BF> → id

布尔量的代码结构

每个布尔量(布尔变量或关系表达式)的目标结构有两个出口：

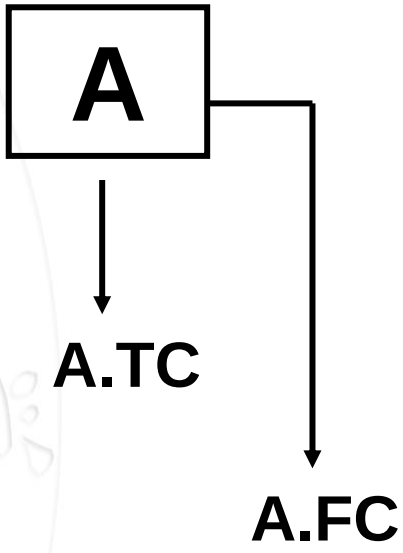
真出口指向为真时应跳转的位置；

假出口指向为假时应跳转的位置。

布尔量翻译为两条四元式：

(jnz, A, _, P)：真出口，当A为真时跳转到四元式P

(j, _, _, q)：假出口，无条件跳转到四元式q



关系表达式也翻译为两条四元式：

(jrop, i1, i2, P)：真出口，当i1 rop i2为真时跳转到P

(j, _, _, q)：假出口，无条件跳转到四元式q

布尔量的代码结构

在翻译布尔表达式时，后面的语句还未确定，那么P和q如何确定？

解决办法：回填技术

生成了真、假出口两个链，两个0就是待填部分。等到以后翻译到后面再填入。

填入；不知时填0，

的转移去向相同，但又不 具体位置，应该

出口相同的四元式连

(1)(jnz, A, _, 3)
(2)(j, _, _, 0)
(3)(jnz, B, _, 5)
(4)(j, _, _, 2)
(5)(j>, C, D,)
(6)(j, _, _, 4)

当扫描到and时，对A进行归约，产生两个四元式1，2，其中真出口已知，为3

假出口未知，填入0

当扫描到B后的and时，

假出口仍未知，但它与A的假出口相同，则链

假出口仍未知，但它与上两个布尔量的假出口相同，则链接起来，填4

布尔量的代码结构

A and B and C > D:

(1)(jnz, A, _, 3)
(2)(j, _, _, 0)
(3)(jnz, B, _, 5)
(4)(j, _, _, 2)
(5)(j>, C, D, 0)
(6)(j, _, _, 4)

生成了真、假出口两个链，两个0就是待填部分。等到以后翻译到后面再填入。

布尔量的代码结构

将P1,P2为链首两个四元式链合并在一起，可以用下述函数返回合并后的四元式链首：

```
merge(P1，P2)
{
  if P2 == 0)  return (P1);
  else  {
        P := P2;
        while (四元式P的第4分量内容不为0) do
        P = 四元式P的第4分量内容;
        把P1填进四元式P的第4分量;
        return (P2);
    }
}
```

布尔量的代码结构

假出口链上的四元式应转向相同的位置，所以一旦知道转向的真实位置，就应返填，返填是将已知位置填入链上的所有四元式的第四个分量。

用backpatch，把已知位置t填入以P为链首的四元式中：

```
BACKPATCH(P，t)
{
  Q=P;
  while (Q!=0) do {
     m = 四元式Q的第4分量内容;
     把t填进四元式Q的第4分量;
     Q = m;
    }
}
```

布尔量的翻译方案

NXQ表示当前产生的四元式的编号

编号	产生式	语义规则
（11）	<BF> → id	{ <BF>.TC=NXQ; <BF>.FC=NXQ+1; gencode(jnz，entry(i)，_，0); gencode(j，_，_，0) }
（10）	<BF> → id rop id	{ <BF>.TC=NXQ; <BF>.FC=NXQ+1; gencode(jrop,$i^{(1)}$.PLACE,$i^{(2)}$.PLACE,0) gencode(j，_，_，0) }
（9）	<BF> → <AE> rop <AE>	{ <BF>.TC=NXQ; <BF>.FC=NXQ+1; gencode(jrop,<AE>$^{(1)}$.PLACE,<AE>$^{(2)}$.PLACE,0) ncode(j，_，_，0) }

单个的布尔量

关系运算

带算术运算的关系运算

布尔量的翻译方案

编号	产生式	语义规则
（8）	<BF> → not<BF>	{ <BF>.TC=<BF>.FC; <BF>.FC=<BF>.TC }
（7）	<BF> → (<BE>)	{ <BF>.TC=<BE>.TC; <BF>.FC=<BE>.FC }
（5）	<BT>^{and} → <BT> and	{ backpatch(<BT>.TC, nextquad); <BT>^{and}.FC=<BT>.FC }
(4)	<BT>→<BT>^{and} <BF>	{ <BT>.TC=<BF>.TC; <BT>.FC=merge(<BT>.FC, <BF>.FC) }
(2)	<BE>^{or}→<BE> or	{ backpatch(<BE>.FC, nextquad); <BE>^{or}.TC=merge(<BE>.TC, <BT>.TC) }
(1)	<BE>→<BE>^{or} <BT>	{ <BE>.FC=<BT>.FC; <BE>.TC=merge(<BE>.TC, <BT>.TC) }

产生式6: <BT>→<BF>，3: <BE>→<BT>的翻译与7相似，都是将右边的真假出口直接赋值到左边

(5)(4)构成and逻辑运算

(2)(1)构成or逻辑运算

布尔表达式的中间代码

举例：a > b and c > d or e < f

100: if a > b goto103
101: t1 = 0
102: goto 104
103: t1 = 1
104: if c > d goto 107
105: t2 = 0
106: goto 108
107: t2 = 1
108: t3 = t1 and t2

109: if e < f goto 112
110: t4 = 0
111: goto 113
112: t4 = 1
113: t5 = t3 or t4

控制流语句

控制流语句：改变程序执行顺序，引起程序执行发生跳转的语句。

- 程序设计语言中出现频繁的语句；
- 为可执行语句，要产生相应的目标代码；
- 控制流程的变换，依靠代码中的跳转指令与对应跳转的语句标号。

条件语句和循环语句

条件语句的一般形式

IF STATEMENT G:

S → if (Er) S1 |

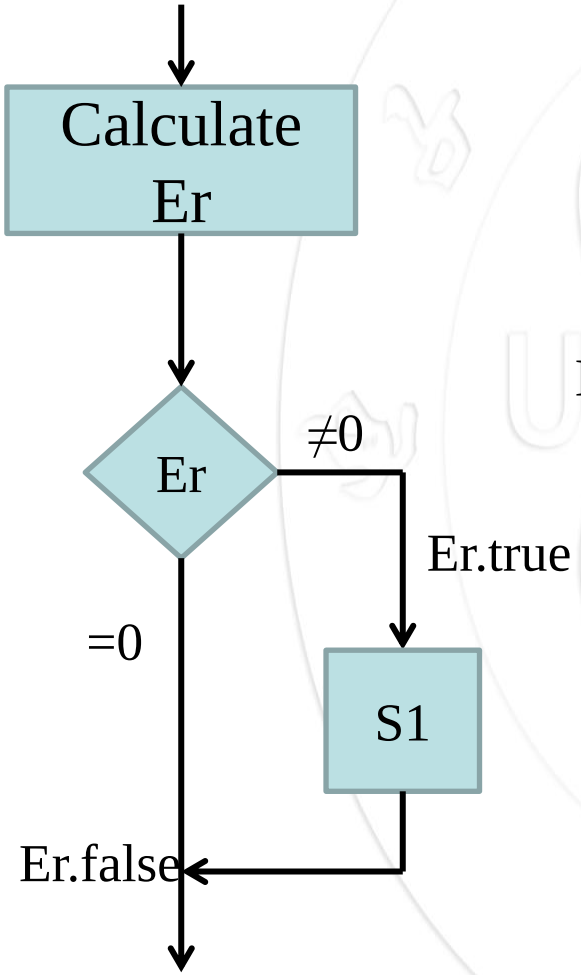
if (Er) S1 else S2 |

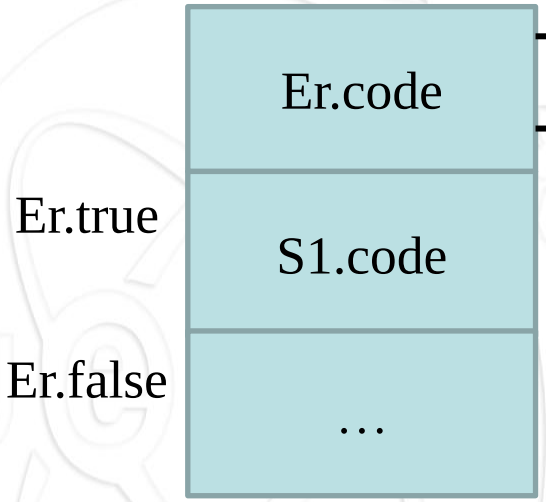
while (Er) S1

其中：Er 是条件表达式（有语义值Er.true 和 Er.false）

S1, S2: 语句

条件语句的计算流程

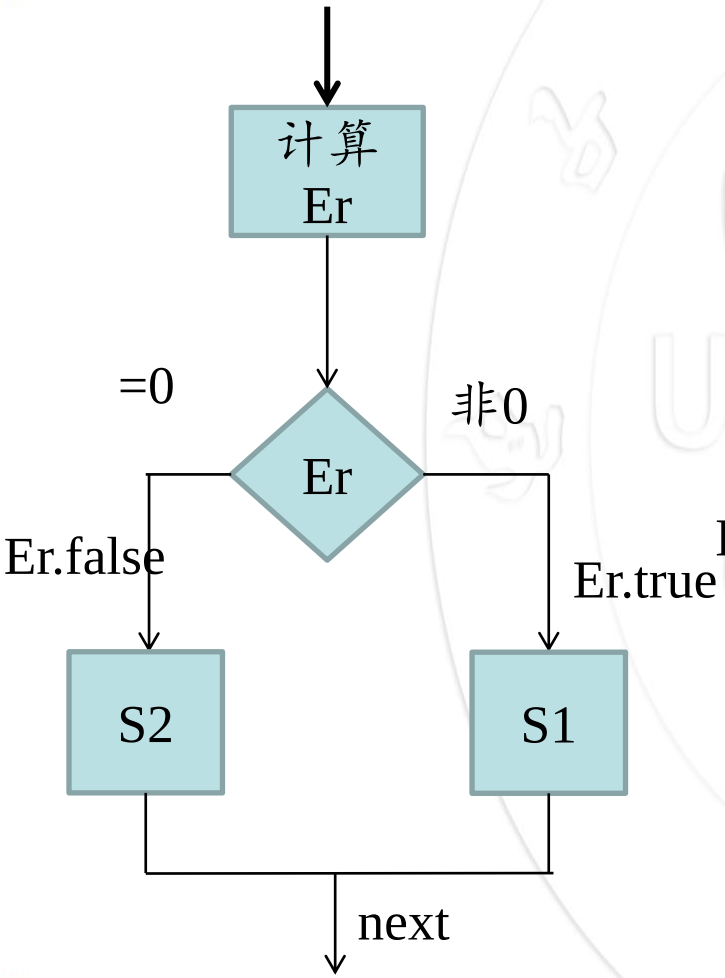


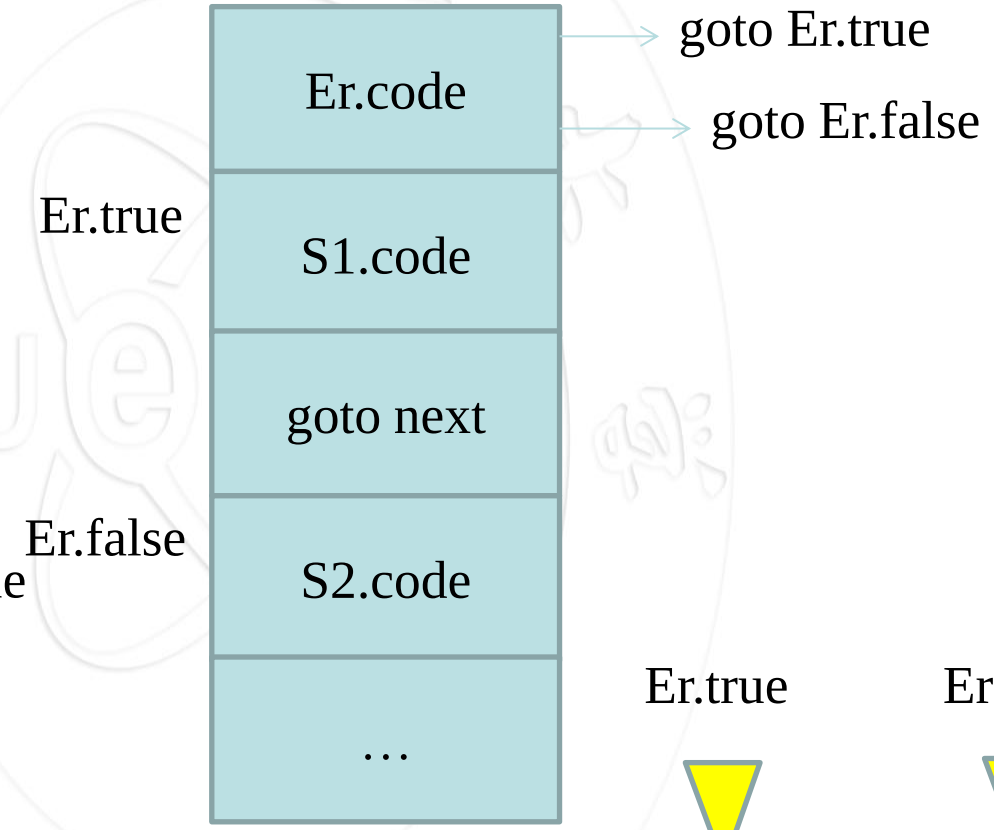


Er.true Er.false

$S \rightarrow \text{if (Er) } S1$

条件语句的计算流程





Er.true　　Er.false　next

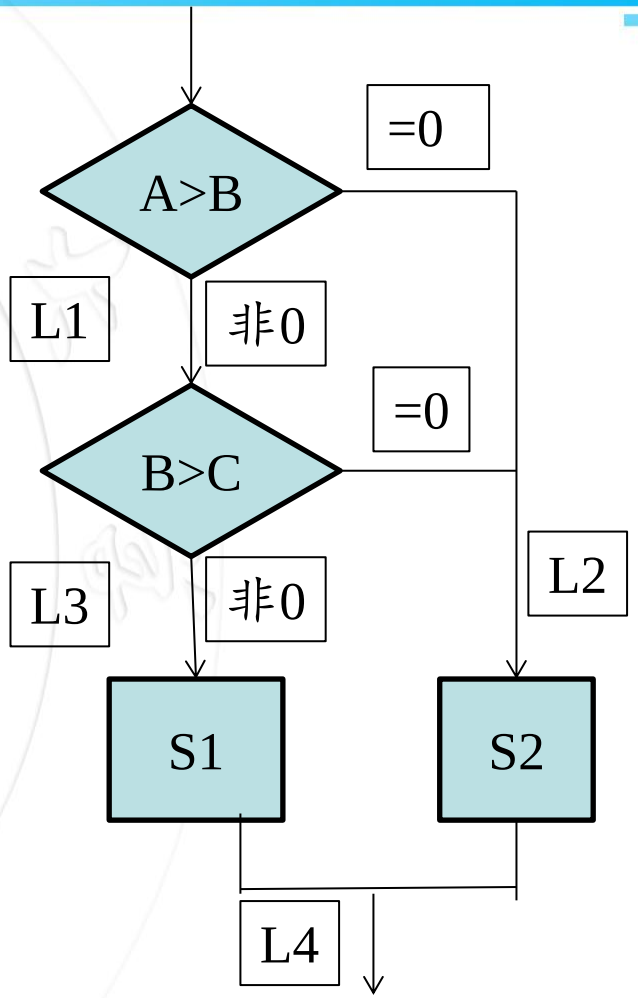
$S \rightarrow \text{if (Er) } S1 \text{ else } S2$

条件语句的计算流程

if (A > B && B > C) S1 else S2

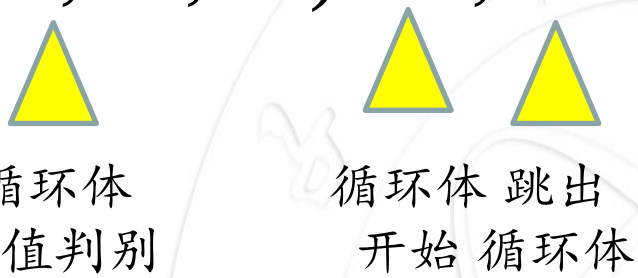
L1 L2 L3 L4

100: if A > B goto 102
101: goto 105
102: if B > C goto 105
103: goto 104
104: S2
105: S1
106: goto 100
107:



循环语句的一般形式

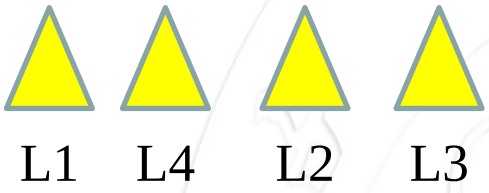
for (e1; e2; e3) S;

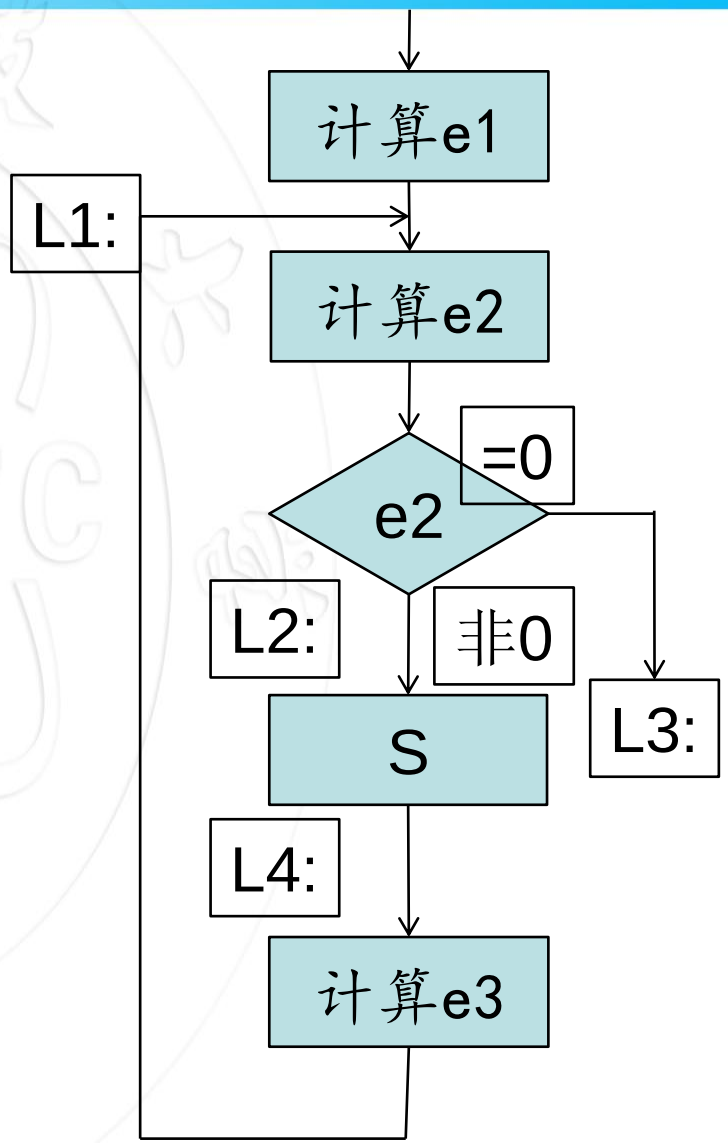


While (E) do S;

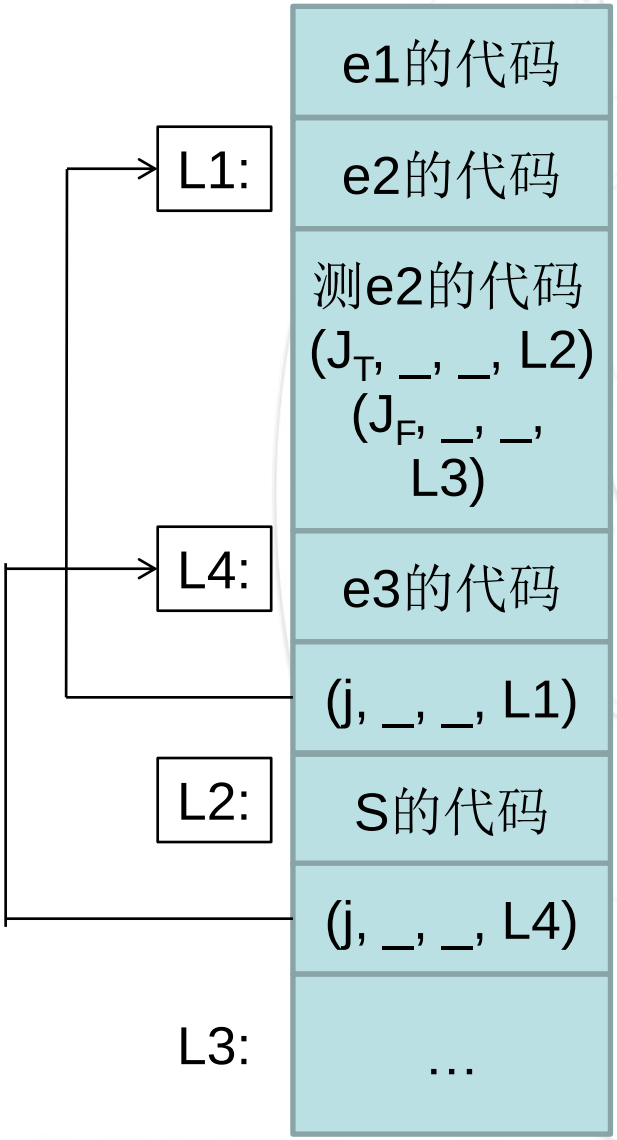
for语句的翻译

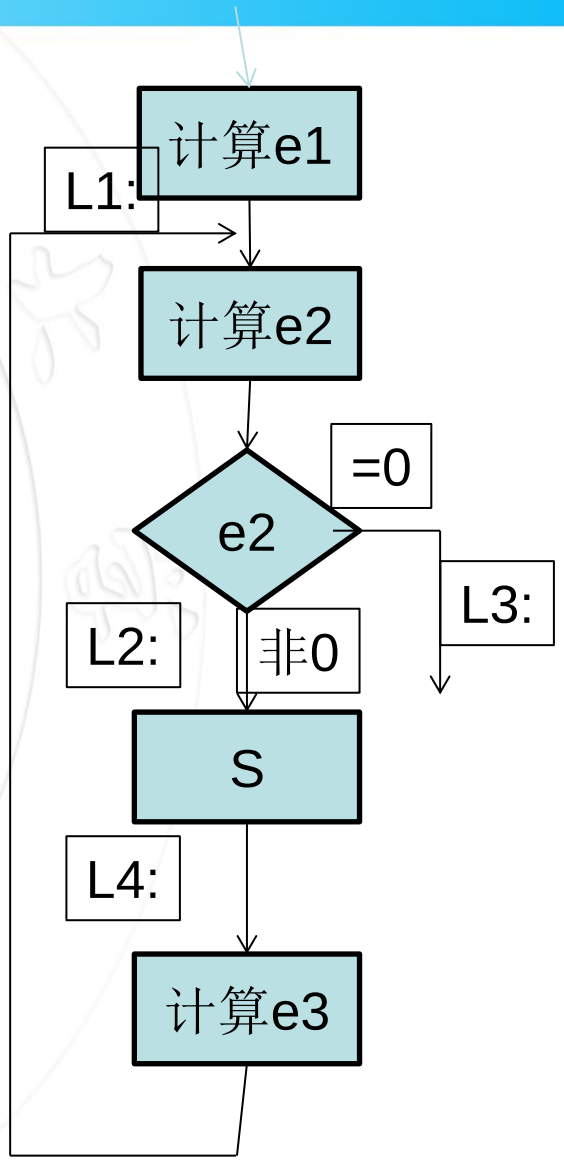
for(e1; e2; e3) S;





for语句的翻译


e1的代码
L1:
e2的代码
测e2的代码
(J_T, _, _, L2)
(J_F, _, _, L3)
L4:
e3的代码
(j, _, _, L1)
L2:
S的代码
(j, _, _, L4)
L3:
…

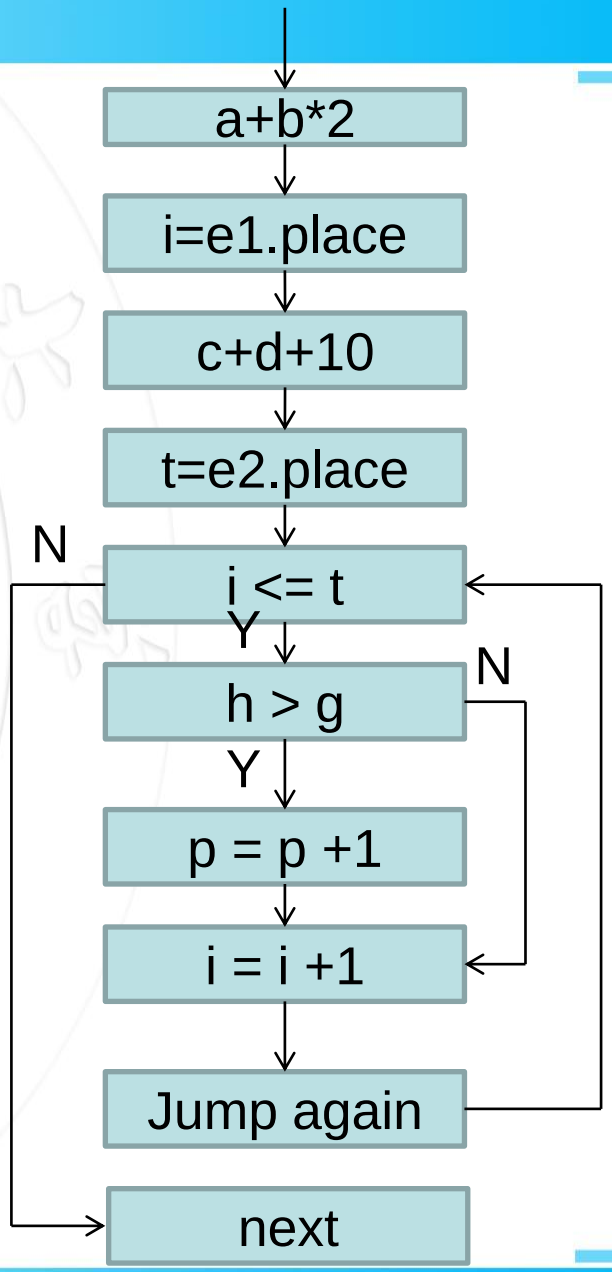

计算e1
L1:
计算e2
=0
e2
L3:
L2:
非0
S
L4:
计算e3

for语句的翻译

例子：for i=a+b*2 to c+d+10 do

if h > g then p = p +1;

a+b*2	(1)	(*, b, 2, T1)
	(2)	(+, a, T1, T2)
i=e1.place	(3)	(=, T2, _, i)
c+d+10	(4)	(+, c, d, T3)
	(5)	(+, T3, 10, T4)
t=e2.place	(6)	(=, T4, _, t)
i>t	(7)	(j>, i, t,)

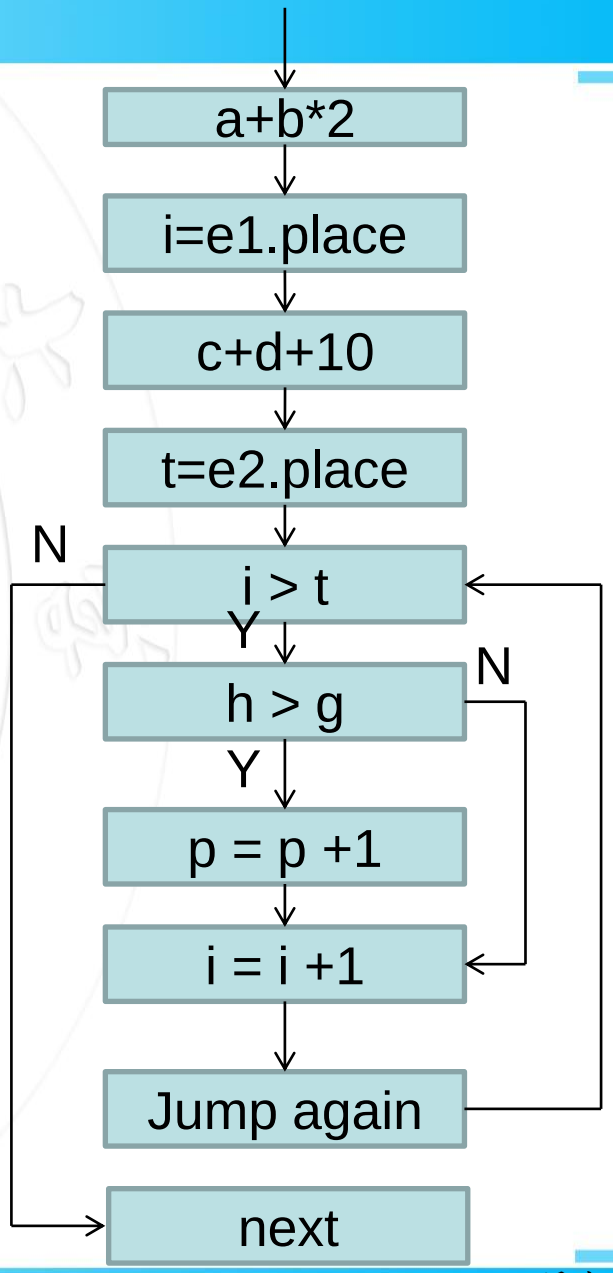


for语句的翻译

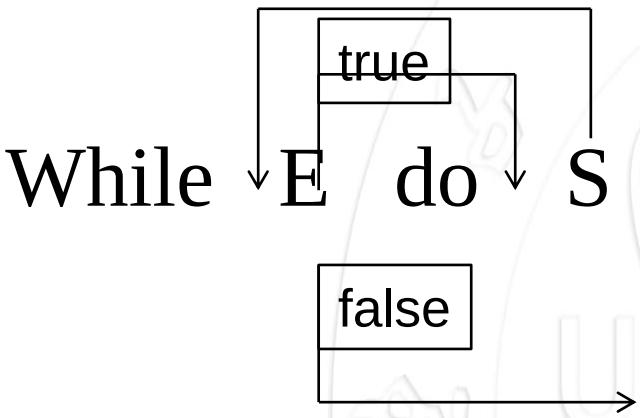
例子：for i=a+b*2 to c+d+10 do

if h > g then p = p +1;

h>g	(8)	(j>, h, g, 10)
	(9)	(j, _, _, 12)
p=p+1	(10)	(+, p, 1, T5)
	(11)	(=, T5, _, p)
i=i+1	(12)	(+, i, 1, T6)
	(13)	(=, T6, _, i)
jump again	(14)	(j, _, _, 7)
next	(15)	



while语句的翻译



①布尔表达式E的真出口为S的第一个四元式地址。

②E的假出口导致程序控制离开while-do语句，然而这个转移目标地址在整个while-do语句翻译完也未必明确。将该四元式的地址作为S的语义值S.chain保存下来，以便在外层环境中伺机回填。

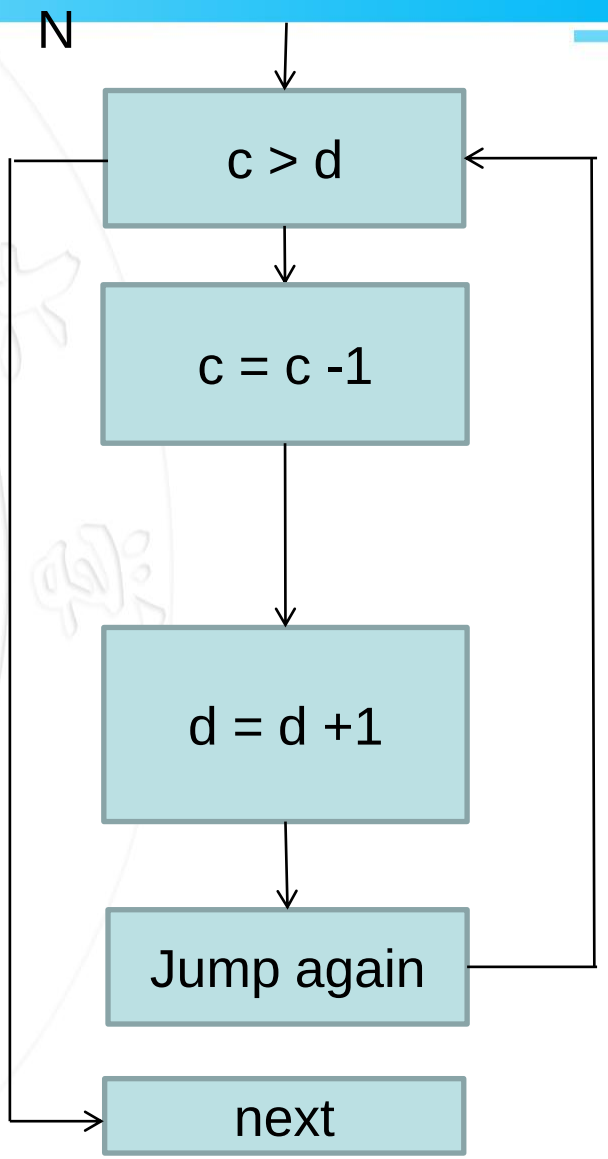
③在S的代码最后应有一条无条件转移四元式，转向测试布尔表达式E，构成循环。需引进新的语义变量.quad，用于记录E的第一个四元式地址。

while语句的翻译

例子：while c > d do

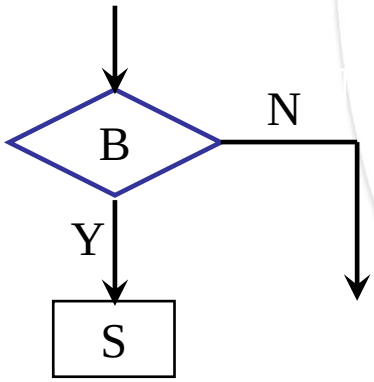
begin

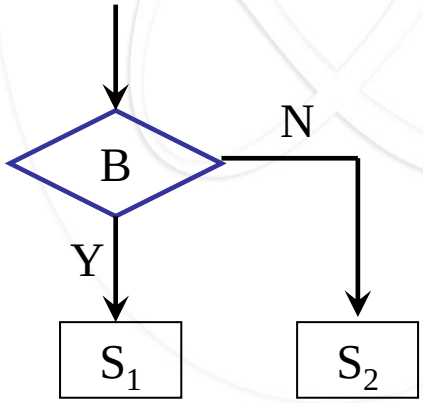
c > d	(1)	(j>, c, d, 3)
	(2)	(j, _, _, 8)
c = c + 1	(3)	(+, c, 1, T1)
	(4)	(=, T1, _, c)
d = d + 1	(5)	(+, d, 1, T2)
	(6)	(=, T2, _, d)
Jump again	(7)	(j, _, _, 1)
next	(8)	

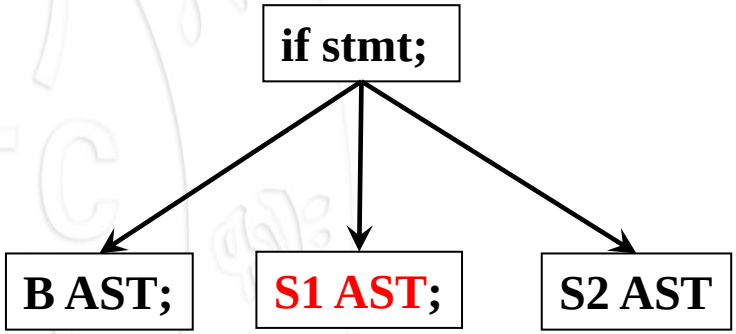


$S \rightarrow$ **if B then** S_1

$S \rightarrow$ **if B then** S_1 **else** S_2

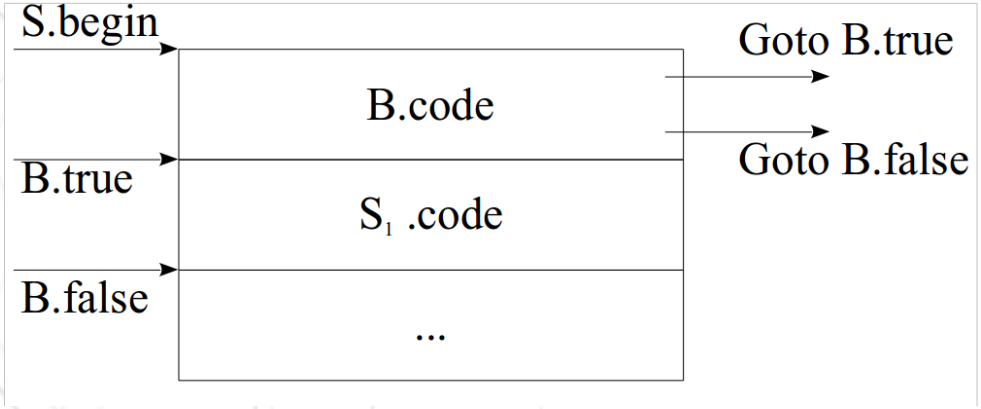






if a<b then a:=a+b

```
100   if a< b goto  102
101   goto  104
102   t1 = a + b
103   a = t1
104
```

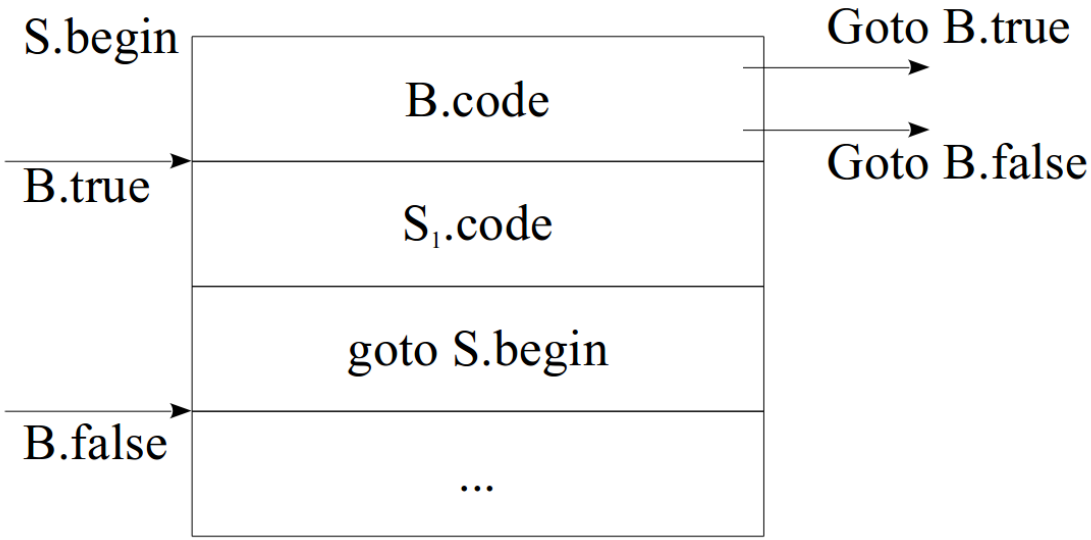


if a<b then a:=a+b else a:=a-b

```
100:   if a < b goto 102
101:   goto 105
102:   t1 = a + b
103:   a = t1
104:   goto 107
105:   t2 = a - b
106:   a = t2
107:   XXX
```

```
100:   if a > b goto 104
101:   t1 = a + b
102:   a = t1
103:   goto 106
104:   t2 = a - b
105:   a = t2
106:   XXX
```

S→while B do S_1



while a<b do if c<d then e:=f+g

```
100:    if a < b goto 102
101:    goto 107
102:    if c < d goto 104
103:    goto 100
104:    t1 = f + g
105:    e = t1
106:    goto 100
107:
```

本章小结

* 语义分析的任务
* 语义分析的分类
* 符号表
* 语法制导翻译
* 赋值语句的翻译
* 控制语句的翻译