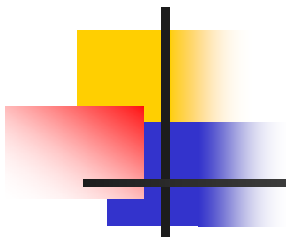




编译原理

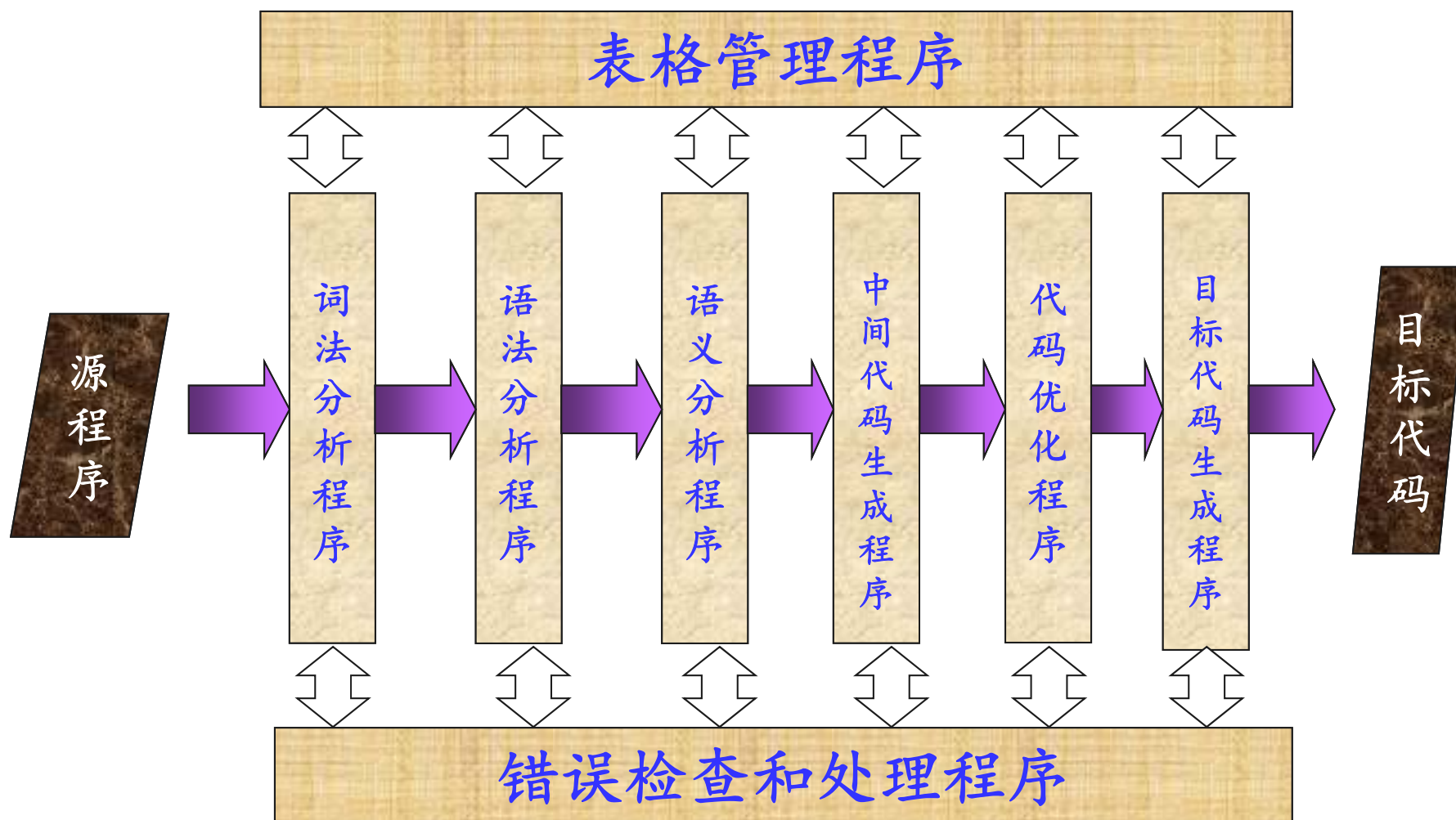
武汉大学计算机学院
编译原理课程组



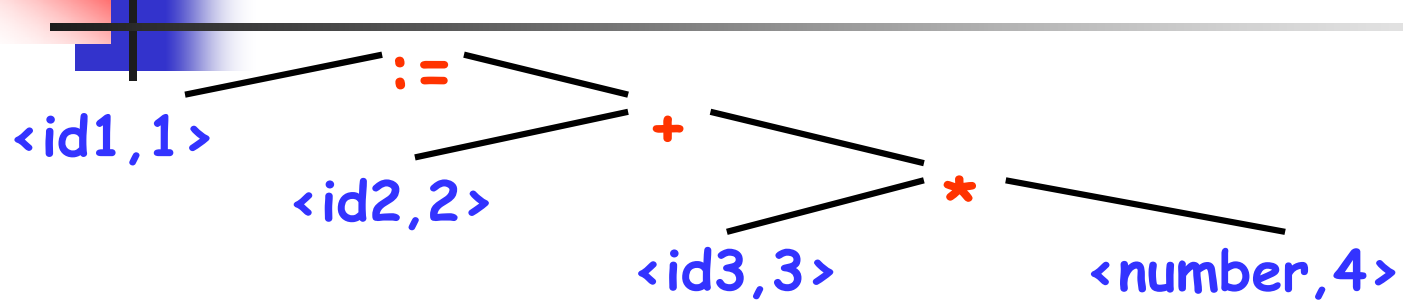
前述内容回顾

- 基本思想
- 存在的问题
- 解决方法
- LR分析方法
- 二义性文法的LR分析

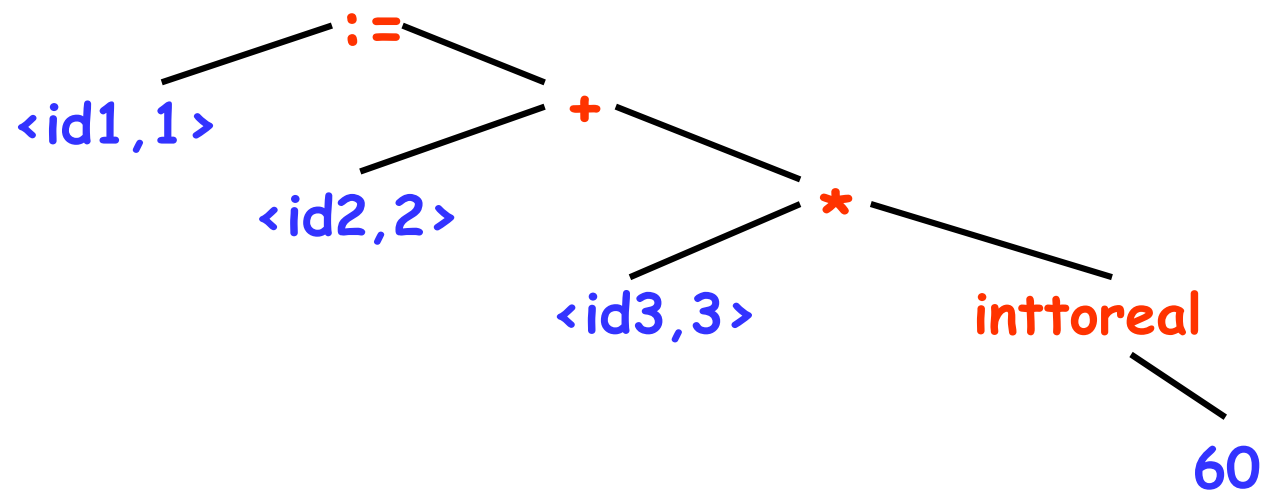
编译程序的结构



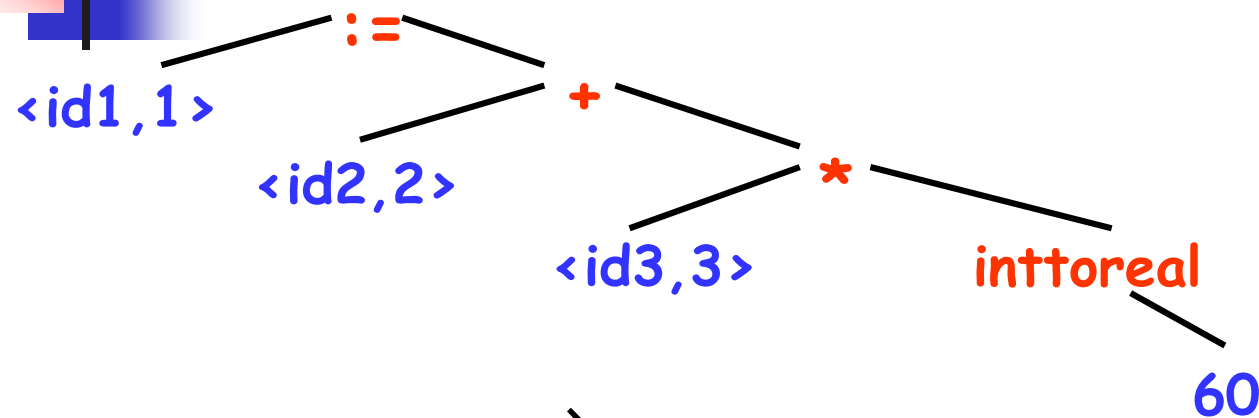
编译过程——语义分析



Semantic analyzer



编译过程——中间代码生成



Intermediate code generator

(1)	(inttoreal,	60	-	t1)	
(2)	(*	,	id3	t1	t2)
(3)	(+	,	id2	t2	t3)
(4)	(:=	,	t3	-	id1)

id1 := id2 + id3 * 60

四元式



语义分析

语法正确并不能保证含义(语义)正确。

依据语言的**语义规则**对语法分析得到的语法结构进行**静态语义检查**(确定类型、类型和运算合法性检查、识别含义与相应的语义处理及其它一些静态语义检查),并用另一种内部形式表示出来,或者直接用目标语言表示出来。



语义分析

程序的含义涉及两方面：数据结构的含义与控制结构的含义。

数据结构的含义——名字的含义(类型正确性检查)。

控制结构的含义——语言自身定义(形式化与非形式化)。

【例1】 $G[E]: E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid i$

【例2】 $\langle \text{赋值语句} \rangle \rightarrow \langle \text{变量} \rangle \langle \text{赋值符号} \rangle \langle \text{表达式} \rangle$ $V := e$
 $V = e$



语义分析

语义分析的基本功能：

- ◆ 确定类型——数据类型(词法分析)
- ◆ 类型检查——运算合法性、运算对象类型一致性或相容性
- ◆ 识别含义——语法成分的含义(中间代码、目标代码)
- ◆ 其他静态语义检查——控制流检查等



语义分析

一般情况下，语义分析仅产生中间代码，因为：

- ◆ 词法分析与语法分析简单、比例小，有利于难点分解；
- ◆ 有利于中间代码优化；
- ◆ 有利于程序的移植；
- ◆ 有利于任务的分解、人员的组织。



语义分析

语义是上下文有关的，进行形式化很困难。

尚无公认的、广泛被接受与流传的语义形式化系统用于描述程序设计语言的语义。

尚未形成可用于编译程序构造的、系统的形式化语义算法或典型技术。

语法制导翻译技术（SDTS: Syntax Directed Translation Scheme）有利于语义分析与目标代码生成的形式化走向实用。



第8章 语法制导翻译

- ◆ 属性文法
- ◆ 目标代码结构
- ◆ 中间代码
- ◆ 控制语句的翻译



8.1 语法制导翻译

1. 属性文法(Attribute Grammar)

1968年, Knuth (高德纳)

对文法中的非终结符号或者终结符号引入一些属性, 描述相应语言结构的语义值(性质)。

属性可以是需要表达或涉及的任何内容, 如名字的类型、名字的值、名字的存储地址、生成的代码等。



8.1 语法制导翻译

1. 属性文法(Attribute Grammar)

翻译文法/增量式文法

$$A \rightarrow (\alpha | \{f(\dots);\})^*$$

为产生式附加语义子程序，用于计算文法符号的属性值。可以是查填符号表的操作、打印出错信息的操作、生成代码的操作等。

属性值的计算，由语法分析过程中产生的语法分析树相应结点的环境推导出来。

8.1 语法制导翻译

1. 属性文法(Attribute Grammar)

翻译文法/增量式文法 $A \rightarrow (\alpha | \{f(\dots);\})^*$

基础文法 $G' [E]$:

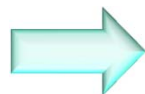
$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) \mid \text{num} \mid i$



增量式文法 $G' [E]$:

$E \rightarrow TE'$

$E' \rightarrow +T \{ \text{writecode}('+'); \} E' | \varepsilon$

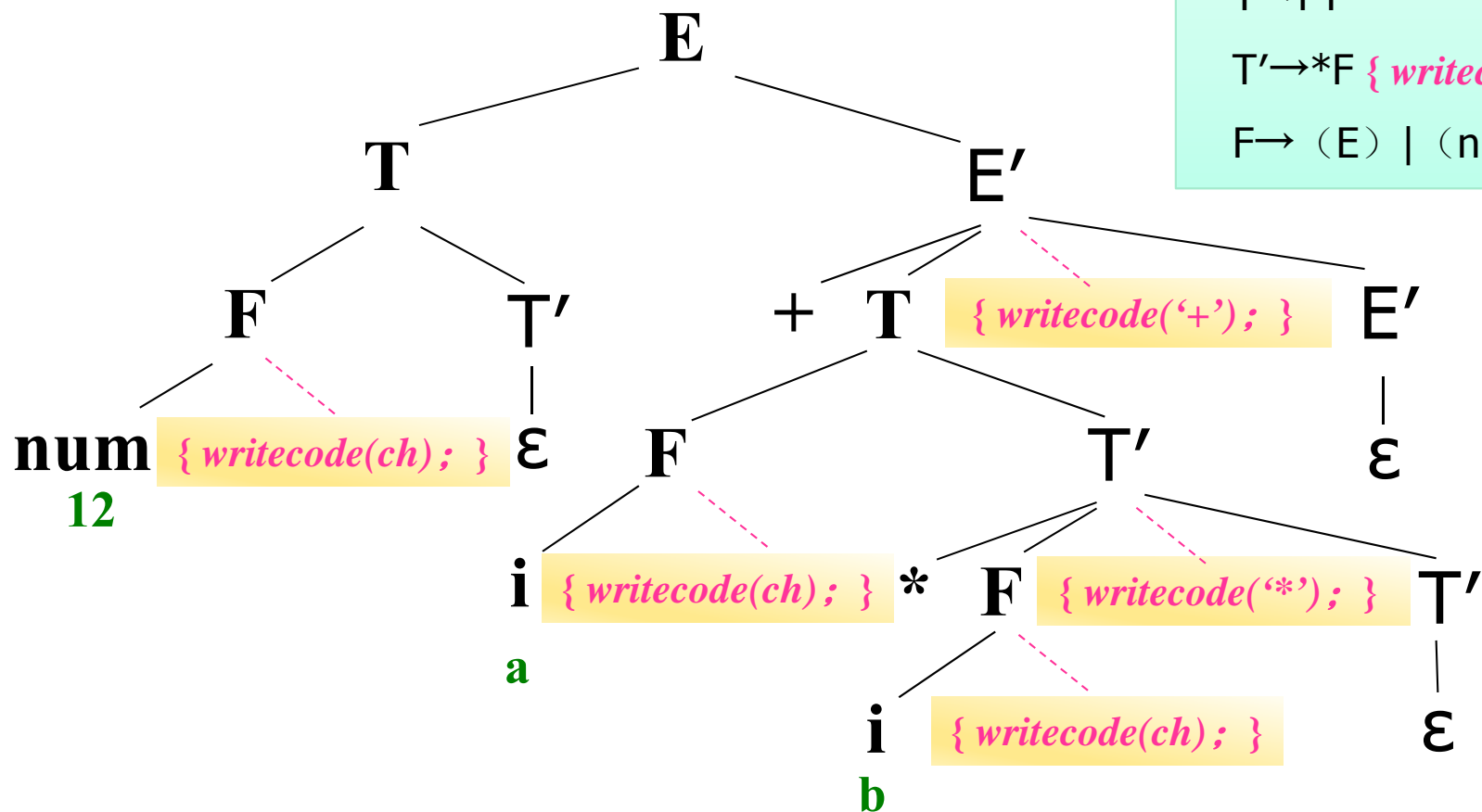
$T \rightarrow FT'$

$T' \rightarrow *F \{ \text{writecode}('*'); \} T' | \varepsilon$

$F \rightarrow (E) \mid (\text{num} | i) \{ \text{writecode}(\text{ch}); \}$

翻译文法/增量式文法

例：表达式 $12+a*b$ 的AST



增量式文法 $G' [E]$:

$E \rightarrow TE'$

$E' \rightarrow +T \{ \text{writecode}('+'); \} E' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *F \{ \text{writecode}('*'); \} T' \mid \epsilon$

$F \rightarrow (E) \mid (\text{num}|i) \{ \text{writecode}(ch); \}$

Annotated Parse Tree

注释树

附注语法分析树

自下而上归约，打印出表达式的逆波兰式 $12ab*+$



8.1 语法制导翻译

1. 属性文法(Attribute Grammar)

对某个上下文无关文法，为每个文法符号指定一组属性，且为文法中的每个产生式附加一段属性计算方法——语义规则/语义动作/语义子程序，则称该文法为属性文法。

属性代表与文法符号相关的信息；属性值可以在语法分析过程中计算和传递；属性加工过程即语义的处理过程。为每个产生式配备的计算属性的计算规则，即语义规则。



8.1 语法制导翻译

1. 属性文法——举例

$G[L]$:

$L \rightarrow E_n$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{digit}$

语法制导定义

SDD(Syntax-Directed Definition)

{ print(E.val); }

{ E.val:=E₁.val+T.val; }

{ E.val:=T.val; }

{ T.val:=T₁.val*F.val; }

{ T.val:=F.val; }

{ F.val:=E.val; }

{ F.val:=digit.lexval ; }



8.1 语法制导翻译

2. 语法制导翻译的基本思想

在语法分析的过程中，依随分析的过程，根据每个产生式添加的语义动作进行翻译。一旦某个产生式被选用于推导或归约，就执行其后相应的语义动作，完成预定的翻译工作。

语法分析与语义分析穿插进行，语法分析引导语义分析。



8.1 语法制导翻译

2. 语法制导翻译的基本思想——举例

$G[L]$:

简单算术表达式求值

$L \rightarrow E_n$	{ 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; }
$F \rightarrow \text{digit}$	{ F.val := digit.lexval ; }

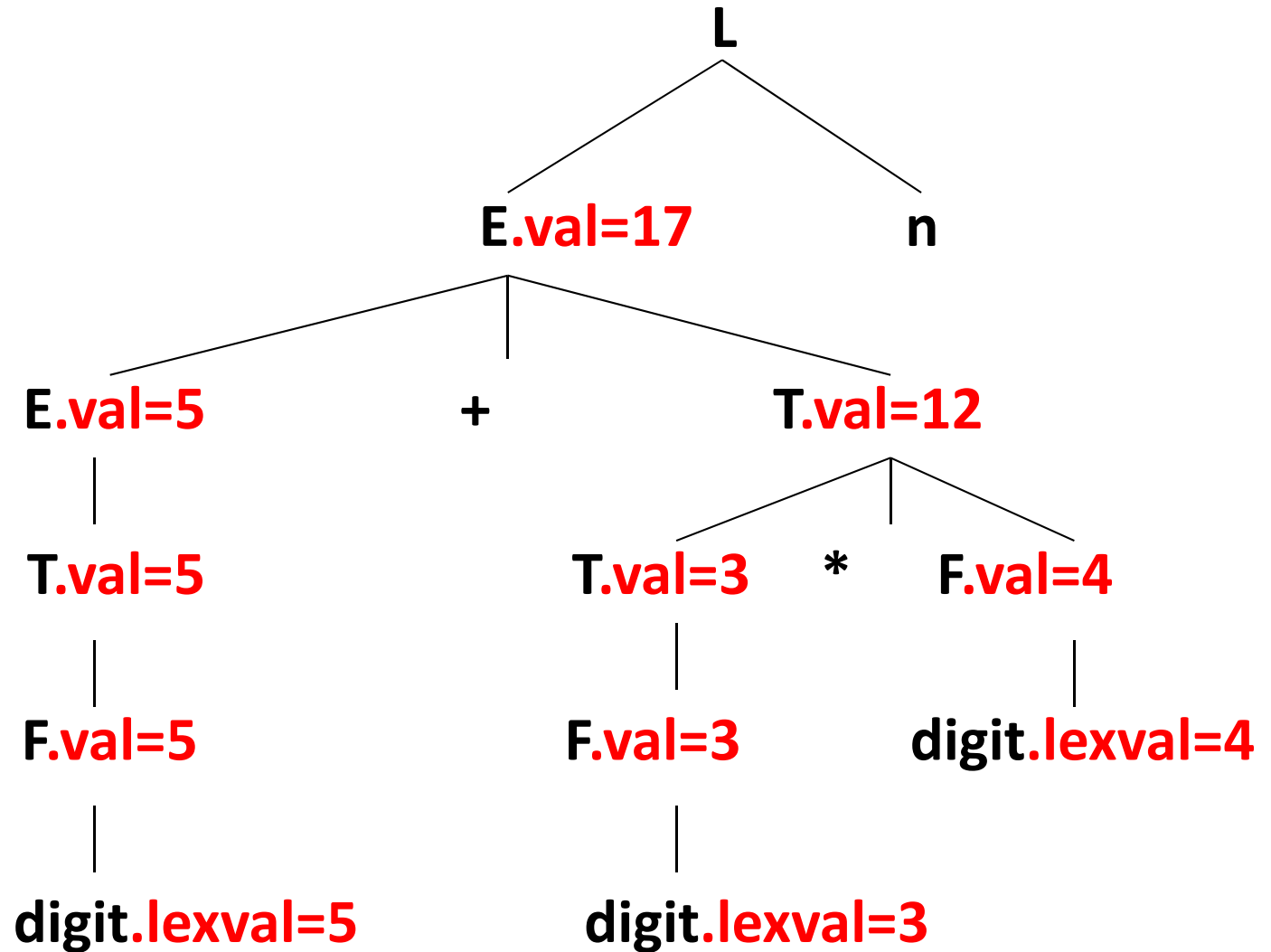
综合属性/归约型
(Synthesized Attribute)

依赖于
子结点的属性

综合属性/归约型 (Synthesized Attribute)

G[L]:

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

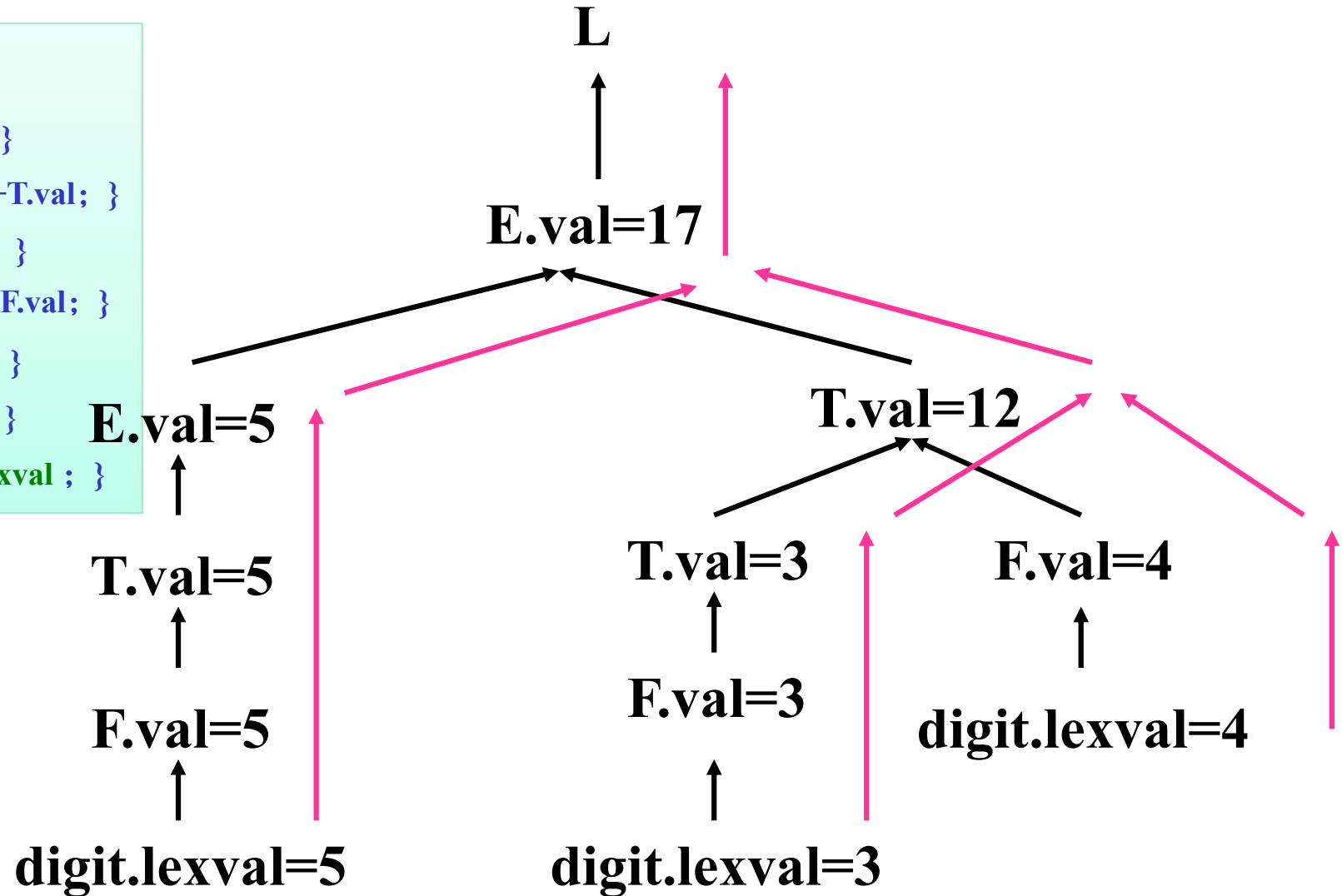


Annotated parse tree for 5+3*4

综合属性/归约型 (Synthesized Attribute)

G[L]:

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



bottom up, from the leaves to the root.



8.1 语法制导翻译

2. 语法制导翻译的基本思想——举例

$G[D]$:

$D \rightarrow TL$

{ $L.in := T.type$; }

$T \rightarrow \text{int}$

{ $T.type := \text{integer}$; }

$T \rightarrow \text{real}$

{ $T.type := \text{real}$; }

$L \rightarrow L_1, \text{id}$

{ $L_1.in := L.in$; }

$\text{addtype}(\text{id.entry}, L.in)$; }

$L \rightarrow \text{id}$

{ $\text{addtype}(\text{id.entry}, L.in)$; }

继承属性/推导型
(Inherited Attribute)

依赖于
父结点/兄弟结点的属性

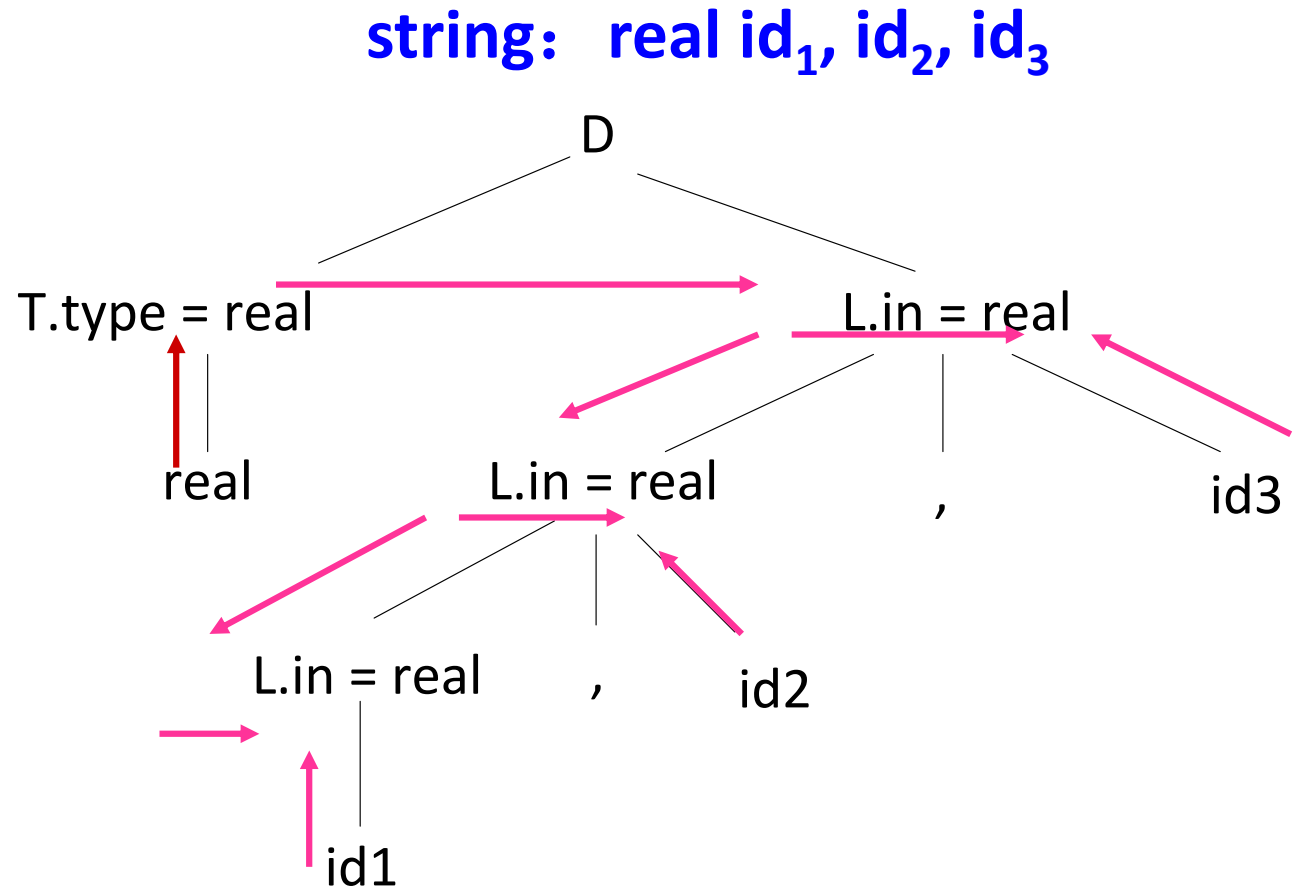
继承属性/推导型 (Inherited Attribute)

G[D]:

$D \rightarrow TL$ { $L.in := T.type$; }
 $T \rightarrow int$ { $T.type := integer$; }
 $T \rightarrow real$ { $T.type := real$; }
 $L \rightarrow L_1, id$ { $L_1.in := L.in$; }
 $addtype(id.entry, L.in)$; }
 $L \rightarrow id$ { $addtype(id.entry, L.in)$; }



Top down, from the root to the leaves.



Symbol **T** is associated with a **synthesized** attribute *type*.
Symbol **L** is associated with an **inherited** attribute *in*.



8.1 语法制导翻译

2. 语法制导翻译的基本思想——举例

$G[P]:$

简化的变量说明的翻译

$P \rightarrow MD;S$

$M \rightarrow \epsilon \quad \{ \text{offset}:=0 \}$

$D \rightarrow D;D$

$D \rightarrow \text{id}:T \quad \{ \text{enter}(\text{id.name}, T.\text{type}, \text{offset})$

$\text{offset}:=\text{offset}+T.\text{width} \}$

$T \rightarrow \text{integer} \quad \{ T.\text{type}:=\text{integer}; T.\text{width}:=4 \}$

$T \rightarrow \text{real} \quad \{ T.\text{type}:=\text{real}; T.\text{width}:=8 \}$



8.2 目标代码结构

1. 语法成分的翻译

◆ 说明语句 ——用于定义各种名字的属性

把所定义名字的各种属性都登记到符号表中

◆ 可执行语句 ——用于完成指定的功能

从源结构到目标结构的变换



8.2 目标代码结构

2. 赋值语句

语法形式:

〈赋值语句〉 → 〈变量〉〈赋值符号〉〈表达式〉

$V := e$

$V = e$

目标代码结构:

计算左部变量V的地址的目标代码

计算表达式e值的目标代码

将e值送V单元的指令



8.2 目标代码结构

3. if 语句

语法形式:

$\langle \text{if语句} \rangle \rightarrow \text{if } \langle \text{布尔表达式} \rangle \text{ then } \langle \text{语句} \rangle |$
 $\text{if } \langle \text{布尔表达式} \rangle \text{ then } \langle \text{语句} \rangle \text{ else } \langle \text{语句} \rangle$

目标代码结构: $\text{if } B \text{ then } S_1 \text{ else } S_2$

计算布尔表达式B的目标代码

B值假(0)转 L_1

语句 S_1 的目标代码

无条件转 L_2

L_1 : 语句 S_2 的目标代码

L_2 : 后继语句



8.2 目标代码结构

4. 循环语句

语法形式:

〈while语句〉 \rightarrow while 〈布尔表达式〉 do 〈语句〉

目标代码结构: while B do S

L_1 : 计算布尔表达式B值的目标代码
B值假(0)转 L_2
语句S的目标代码
无条件转 L_1
L_2 :



8.3 中间代码

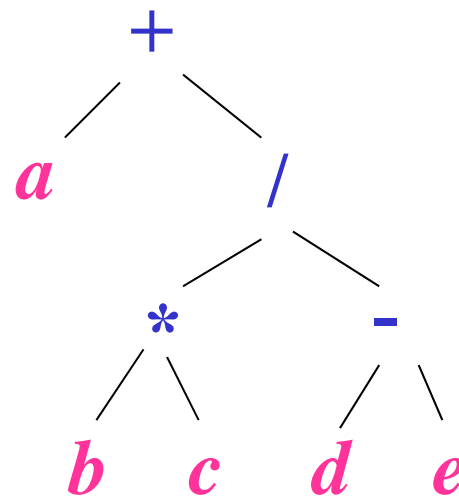
常见的中间代码形式有：逆波兰表示、四元式、三元式和树形表示(抽象语法树)等等。

8.3 中间代码

1. 树型表示

抽象语法树 (AST: Abstract Syntax Tree)

例：表达式 $a+b*c/(d-e)$ 的AST



层次结构分析

内部结点：运算

其子结点：该运算的分量

特点：结构紧凑，容易构造，结点数少，计算机内表示方便。



8.3 中间代码

2. 逆波兰表示——波兰逻辑学家J. Lukasiewicz

运算符直接跟在其运算量(操作数)的后面

—— 后缀(Post Fix)表示法。

逆波兰表示易于生成代码。

同一层中(括号算作新一层)运算符按其优先级别的次序出现。

中缀式

$a*(b+c)$

$a*(b+c*d)$

逆波兰式

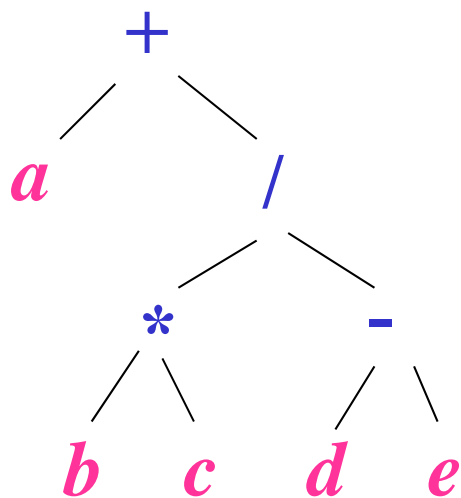
$abc+*$

$abcd*+*$

8.3 中间代码

2. 逆波兰表示

例：表达式 $a+b*c/(d-e)$ 的逆波兰表示、抽象语法树（AST）



$abc*de-/+$

逆波兰表示法的求值运算简单，一遇到运算符即可开始运算。



8.3 中间代码

2. 逆波兰表示

例1：求 $x+y \leq z \vee a > 0 \wedge (8+z) > 3$ 的逆波兰表示。

例2：表达式 $a*b-c-d\$e\$f-g-h*i$ 中，运算符的优先级由高到低依次为 $-$ 、 $*$ 、 $\$$ ，且均右结合，求其逆波兰表示。

逆波兰表示的特点：

第一，无括号；

第二，运算符出现的顺序就是实际的运算顺序；

第三，运算对象出现的顺序与中缀形式一致。

逆波兰表示法可由表达式的表示推广到其它语法成分的表示。



8.3 中间代码

2. 逆波兰表示

逆波兰表示法可由表达式的表示推广到其它语法成分的表示。

赋值语句： $\langle \text{变量} \rangle := \langle \text{表达式} \rangle$ 的逆波兰表示为：

$\langle \text{变量}' \rangle \langle \text{表达式}' \rangle :=$

其中， $\langle \text{变量}' \rangle$ 、 $\langle \text{表达式}' \rangle$ 分别表示

$\langle \text{变量} \rangle$ 、 $\langle \text{表达式} \rangle$ 的逆波兰形式。



8.3 中间代码

2. 逆波兰表示

条件语句： IF <表达式> THEN <语句1> ELSE <语句2>

的逆波兰表示为：

<表达式'> L_1 jumpf <语句1'> L_2 jump <语句2'>

其中，<表达式'>、<语句1'>、<语句2'>分别表示<表达式>、<语句1>、<语句2>的逆波兰形式。而 L_1 表示<语句2'>的开始处， L_2 表示IF语句的后继语句的逆波兰表示的开始处，<表达式'> L_1 jumpf表示若<表达式'>的值为假，则转至 L_1 处， L_2 jump表示无条件转至 L_2 处。



8.3 中间代码

2. 逆波兰表示

$k := 100;$

if $k > i + j$

then $k := k - 1$

else $k := i * 2 - j * 2;$



8.3 中间代码

2. 逆波兰表示

k:=100;

if k>i+j

then k:=k-1

else k:=i*2-j*2;

1	2	3	4	5	6	7	8
k	100	:=	k	i	j	+	>
9	10	11	12	13	14	15	16
18	jf	k	k	1	-	:=	27
17	18	19	20	21	22	23	24
j	k	i	2	*	j	2	*
25	26	27	28	29	30	31	32
-	:=						



8.3 中间代码

3. 四元式表示 ($\langle \text{运算符} \rangle, \langle \text{运算量1} \rangle, \langle \text{运算量2} \rangle, \langle \text{结果} \rangle$)

表达式 $-(a+b)/(c-d)-(a+b*c)$

+	a	b	T_1
-	T_1		T_2
-	c	d	T_3
/	T_2	T_3	T_4
*	b	c	T_5
+	a	T_5	T_6
-	T_4	T_6	T_7



8.3 中间代码

3. 四元式表示

四元式表示法可由表达式的表示推广到其它语法成分的表示。

k:=100;	(1) ($:=$, 100, $_$, k)	(7) (jump, $_$, $_$, (12))
if k>i+j	(2) (+, i, j, T_1)	(8) (*, i, 2, T_4)
then k:=k-1	(3) (>, k, T_1 , T_2)	(9) (*, j, 2, T_5)
else k:=i*2-j*2;	(4) (jumpf, T_2 , $_$, (8))	(10) (-, T_4 , T_5 , T_6)
	(5) (-, k, 1, T_3)	(11) ($:=$, T_6 , $_$, k)
	(6) ($:=$, T_3 , $_$, k)	(12)



8.3 中间代码

4. 三元式表示 ($\langle \text{运算符} \rangle, \langle \text{运算量1} \rangle, \langle \text{运算量2} \rangle$)

表达式 $-(a+b)/(c-d)-(a+b*c)$

(1) $(+, a, b)$

(2) $(-, (1), _)$

(3) $(-, c, d)$

(4) $(/, (2), (3))$

(5) $(*, b, c)$

(6) $(+, a, (5))$

(7) $(-, (4), (6))$



8.3 中间代码

4. 三元式表示

三元式表示法可由表达式的表示推广到其它语法成分的表示。

k:=100;	(1) (:= , 100, k)	(7) (jump , _, (12))
	(2) (+ , i, j)	(8) (* , i, 2)
if k>i+j	(3) (> , k, (2))	(9) (* , j, 2)
then k:=k-1	(4) (jumpf , (3), (8))	(10) (- , (8), (9))
else k:=i*2-j*2;	(5) (- , k, 1)	(11) (:= , (10), k)
	(6) (:= , (5), k)	(12)



8.3 中间代码

4. 三元式表示

三元式与四元式的比较。

优点：

无须引进临时变量，占用存储空间少。

不足之处：

由于三元式相互引用(通过序号)太多，不便于实现代码优化。

8.4 语句的翻译

1. 赋值语句的翻译

语法形式:

$S \rightarrow i := E$

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow -E$

$E \rightarrow (E)$

$E \rightarrow i$

$V := e$

目标代码结构:

计算左部变量 V 的地址的目标代码

计算表达式 e 值的目标代码

将 e 值送 V 单元的指令



8.4 语句的翻译

1. 赋值语句的翻译

引进语义变量和语义过程：

lookup(name)：查符号表，若在，返回表项位置，否则null。

entry(name)：获得name在符号表中的位置。

newtemp：回送一个代表新临时变量名(T_1, T_2, \dots 等)的整数码。

E.place：存放E值的变量在符号表的入口或整数码(临时变量)。

GEN(op, arg₁, arg₂, result)：生成四元式(op, arg₁, arg₂, result)并填进四元式表中。



8.4 语句的翻译

1. 赋值语句的翻译

属性文法：

$S \rightarrow i := E$ { GEN($:=$, E.place, __, entry(i)) }

$E \rightarrow E_1 + E_2$ { E.place := newtemp; GEN(+, E₁.place, E₂.place, E.place) }

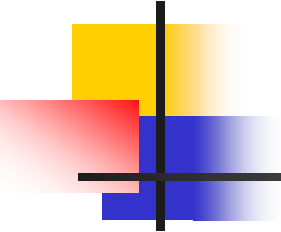
$E \rightarrow E_1 * E_2$ { E.place := newtemp; GEN(*, E₁.place, E₂.place, E.place) }

$E \rightarrow -E_1$ { E.place := newtemp; GEN(-, E₁.place, __, E.place) }

$E \rightarrow (E_1)$ { E.place := E₁.place }

$E \rightarrow i$ { E.place := entry(i) }

$A := -B * (C + D) \quad \#$



8.4 语句的翻译

算术表达式到逆波兰表示的语法制导翻译

翻译文法：

$E \rightarrow E + T$ { print(+)

$E \rightarrow E - T$ { print(-)

$E \rightarrow T$

$T \rightarrow T * F$ { print(*)}

$T \rightarrow T / F$ { print(/)}

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow i$ { print(id)}

$a + b * c \quad \Rightarrow \quad abc * +$



8.4 语句的翻译

2. 布尔表达式的翻译

语法形式:

$E \rightarrow E \text{ and } E$

$E \rightarrow E \text{ or } E$

$E \rightarrow \neg E$

$E \rightarrow (E)$

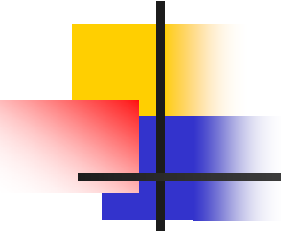
$E \rightarrow i$

$E \rightarrow i \text{ rop } i$

作用:

□ 求逻辑值

□ 作控制条件



8.4 语句的翻译

2. 布尔表达式的翻译

$$A \vee B \wedge C = D$$

两种计值方法:

- ◆ 逐步求值法: 算出每一个运算分量的值
- ◆ 短路表达式求值: 优化措施, 利用布尔运算符的性质

把 $A \vee B$ 解释成 if A then true else B

把 $A \wedge B$ 解释成 if A then B else false

把 $\neg A$ 解释成 if A then false else true

假如函数过程不产生副作用, 则上述两种方法等价。



8.4 语句的翻译

2. 布尔表达式的翻译

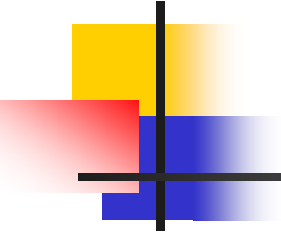
可将布尔表达式翻译成仅含如下三种形式的四元式序列：

$(j_{nz}, A_1, _, p)$ —— A_1 为“真”，转向四元式 p

(j_{rop}, A_1, A_2, p) —— $A_1 \text{ rop } A_2$ 为“真”，转向四元式 p

$(j, _, _, p)$ —— 无条件转向四元式 p

例如： $x := A > B \vee C$



8.4 语句的翻译

2. 布尔表达式的翻译

◆ 逐步求值法

- (1) $(>, A, B, T1)$
- (2) $(\vee, T1, C, T2)$
- (3) $(:=, T2, -, X)$

例如: $x := A > B \vee C$

◆ 短路表达式求值

- (1) $(j_>, A, B, (5))$
- (2) $(j_{nz}, C, -, (5))$
- (3) $(:=, 'false', -, x)$
- (4) $(j, -, -, (6))$
- (5) $(:=, 'true', -, x)$
- (6)



8.4 语句的翻译

2. 布尔表达式的翻译

《陈火旺》P188 表7.7, 例7.3

多遍扫描法：综合属性和继承属性求值

第一遍扫描：自下而上，综合属性求值

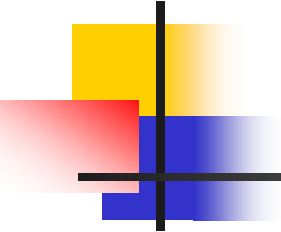
E. code

第二遍扫描：自上而下，继承属性求值

E. true

E. false

这里所有的标号的具体值都无法在产生代码时确定，需再扫描一遍源程序的分析树/语法树，才能计算出标号的具体值，故为多遍。



8.4 语句的翻译

2. 布尔表达式的翻译

单遍扫描法：真、假出口链与地址回填

在自下而上的分析中，一个布尔表达式E的真假出口往往不能在产生指令的同时就填上，只好把这个未完成的指令的地址(如四元式的编号)作为E的语义值暂存起来，待到整个表达式的指令产生完毕后再来回填这个未填的转移目标——地址回填。

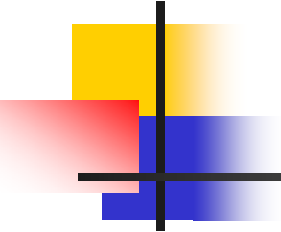


8.4 语句的翻译

2. 布尔表达式的翻译

增加语义变量：

对每个E，赋予两个语义值： $E.truelist$ 和 $E.falselist$ ，分别记录表达式E所对应的四元式需要回填“真”、“假”出口的四元式的地址所构成的链。



8.4 语句的翻译

2. 布尔表达式的翻译 —— 引进语义过程和语义变量

nextquad: 指向下一个将要形成但尚未形成的四元式。初值为1。

每当执行一次GEN, nextquad的值自动累加增1。

make list(i): 创建一个仅含i的新链表, 其中i是四元式数组的一个下标(标号); 函数返回指向这个链的指针。

merge(P_1 , P_2): 把以 P_1 和 P_2 为链首的两条链合并为一, 返回合并后的链首 P_1 。

backpatch(P, t): 过程“回填”, 把P所链接的每个四元式第四区段都填为t。



8.4 语句的翻译

2. 布尔表达式的翻译

修改文法:

$$E \rightarrow E \text{ and } E$$
$$E \rightarrow E \text{ or } E$$
$$E \rightarrow \neg E$$
$$E \rightarrow (E)$$
$$E \rightarrow i$$
$$E \rightarrow i \text{ rop } i$$
$$E \rightarrow E_1 \text{ and } M E_2$$
$$E \rightarrow E_1 \text{ or } M E_2$$
$$E \rightarrow \neg E$$
$$E \rightarrow (E)$$
$$E \rightarrow i$$
$$E \rightarrow i_1 \text{ rop } i_2$$
$$M \rightarrow \varepsilon$$

- | | |
|--|--|
| (1) $E \rightarrow E_1$ or $M E_2$ | { <code>backpatch(E_1.falselist, M.quad)</code> ;
E .truelist:=merge(E_1 .truelist, E_2 .truelist);
E .falselist:= E_2 .falselist; } |
| (2) $E \rightarrow E_1$ and $M E_2$ | { <code>backpatch(E_1.truelist, M.quad)</code> ;
E .truelist:= E_2 .truelist;
E .falselist:=merge(E_1 .falselist, E_2 .falselist); } |
| (3) $E \rightarrow \text{not } E_1$ | { E .truelist:= E_1 .falselist;
E .falselist:= E_1 .truelist; } |
| (4) $E \rightarrow (E_1)$ | { E .truelist:= E_1 .truelist;
E .falselist:= E_1 .falselist; } |
| (5) $E \rightarrow id_1 \text{ relop } id_2$ | { E .truelist:=makelist(nextquad);
E .falselist:=makelist(nextquad+1);
gen('j'relop.op', 'id ₁ .place', 'id ₂ .place', '0');
gen('j, -, -, 0'); } |
| (6) $E \rightarrow id$ | { E .truelist:=makelist(nextquad);
E .falselist:=makelist(nextquad+1);
gen('j _{nz} ' ' , ' id.place ' , ' '-' ' , ' 0');
gen('j, -, -, 0'); } |
| (7) $M \rightarrow \epsilon$ | { M .quad:=nextquad; } |

$a < b$ or $c < d$ and $e < f$



8.4 语句的翻译

2. 布尔表达式的翻译

$a < b$ or $c < d$ and $e < f$

地址回填

.....

(100) (j<, a, b, 0)

(101) (j, _, _, 102)

(102) (j<, c, d, 104)

(103) (j, _, _, 0)

(104) (j<, e, f, 0)

(105) (j, _, _, 0)

(106)



8.4 语句的翻译

3. 条件语句的翻译

语法形式:

〈if语句〉 \rightarrow if 〈布尔表达式〉 then 〈语句〉 else 〈语句〉

目标代码结构: if E then S_1 else S_2

计算布尔表达式E的目标代码

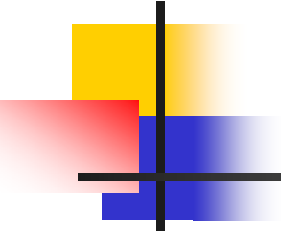
E值假(0)转 L_1

语句 S_1 的目标代码

无条件转 L_2

L_1 : 语句 S_2 的目标代码

L_2 : 后继语句



8.4 语句的翻译

3. 条件语句的翻译

《陈火旺》P193 表7.8, 例7.5

多遍扫描法：综合属性和继承属性求值

控制流语句的多趟翻译模式

参见《陈火旺》第7章 7.4节、7.5节

8.4 语句的翻译

3. 条件语句的翻译

单遍扫描法：地址回填

SDTS基本思想：改造文法，并添加语义子程序

$S \rightarrow T \text{ else } S_2$

Sub1: 生成 S_2 的中间代码;
设置位置标号 L_2

$T \rightarrow I \text{ then } S_1$

Sub2: 生成 S_1 的中间代码;
生成无条件转 L_2 的代码;
设置 L_1

$I \rightarrow \text{if } E$

Sub3: 生成计算 E 的中间代码;
生成若 E 为false则转 L_1 的中间代码

计算布尔表达式 E 的中间代码

E 值假(0) 转 L_1

语句 S_1 的中间代码

无条件转 L_2

L_1 : 语句 S_2 的中间代码

L_2 : 后继语句

举例:

if $A > B \vee C$ then $B := B + 1$ else $A := A * B$

8.4 语句的翻译

4. 控制语句的翻译 地址回填 插入指令 语法制导 ——修改文法

语法形式:

- | | |
|--|--|
| (1) $S \rightarrow \text{if } E \text{ then } S \text{ else } S$ | (1) $S \rightarrow \text{if } E \text{ then } M_1 S_1 N \text{ else } M_2 S_2$ |
| (2) $S \rightarrow \text{if } E \text{ then } S$ | (2) $N \rightarrow \varepsilon$ |
| (3) $S \rightarrow \text{while } E \text{ do } S$ | (3) $M \rightarrow \varepsilon$ |
| (4) $S \rightarrow \text{begin } L \text{ end}$ | (4) $S \rightarrow \text{if } E \text{ then } M S_1$ |
| (5) $S \rightarrow A$ | (5) $S \rightarrow \text{while } M_1 E \text{ do } M_2 S_1$ |
| (6) $L \rightarrow L; S$ | (6) $S \rightarrow \text{begin } L \text{ end}$ |
| (7) $L \rightarrow S$ | (7) $S \rightarrow A$ |
| | (8) $L \rightarrow L_1; M S$ |
| | (9) $L \rightarrow S$ |

- (1) $S \rightarrow \text{if } E \text{ then } M_1 S_1 N \text{ else } M_2 S_2$

```

      { backpatch(E.truelist, M1.quad) ;
        backpatch(E.falselist, M2.quad) ;
        S.nextlist := merge(S1.nextlist, N.nextlist, S2.nextlist) ; }

```
- (2) $N \rightarrow \epsilon$

```

      { N.nextlist := makelist(nextquad) ; gen('j, -, -, 0') ; }

```
- (3) $M \rightarrow \epsilon$

```

      { M.quad := nextquad ; }

```
- (4) $S \rightarrow \text{if } E \text{ then } M S_1$

```

      { backpatch(E.truelist, M.quad) ;
        S.nextlist := merge(E.falselist, S1.nextlist) ; }

```
- (5) $S \rightarrow \text{while } M_1 E \text{ do } M_2 S_1$

```

      { backpatch(S1.nextlist, M1.quad) ;
        backpatch(E.truelist, M2.quad) ;
        S.nextlist := E.falselist ; gen('j, -, -, ', M1.quad) ; }

```
- (6) $S \rightarrow \text{begin } L \text{ end}$

```

      { S.nextlist := L.nextlist ; }

```
- (7) $S \rightarrow A$

```

      { S.nextlist := makelist( ) ; }

```
- (8) $L \rightarrow L_1 ; M S$

```

      { backpatch(L1.nextlist, M.quad) ;
        L.nextlist := S.nextlist ; }

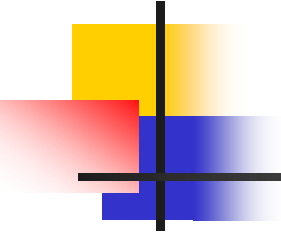
```
- (9) $L \rightarrow S$

```

      { L.nextlist := S.nextlist ; }

```

while $a < b$ or $c < d$ and $e < f$ **do** if $(c < d)$ then $x := y + z$; $x := 1$;



8.4 语句的翻译

4. 控制语句的翻译

while $a < b$ or $c < d$ and $e < f$ **do** if ($c < d$) then $x := y + z$; $x := 1$;

.....

(100) ($j <$, a , b , **106**)

(101) (j , $_$, $_$, **102**)

(102) ($j <$, c , d , **104**)

(103) (j , $_$, $_$, **111**)

(104) ($j <$, e , f , **106**)

(105) (j , $_$, $_$, **111**)

(**106**) ($j <$, c , d , 108)

(107) (j , $_$, $_$, 100)

(108) ($+$, y , z , T_1)

(109) ($:=$, T_1 , $_$, x)

(110) (j , $_$, $_$, 100)

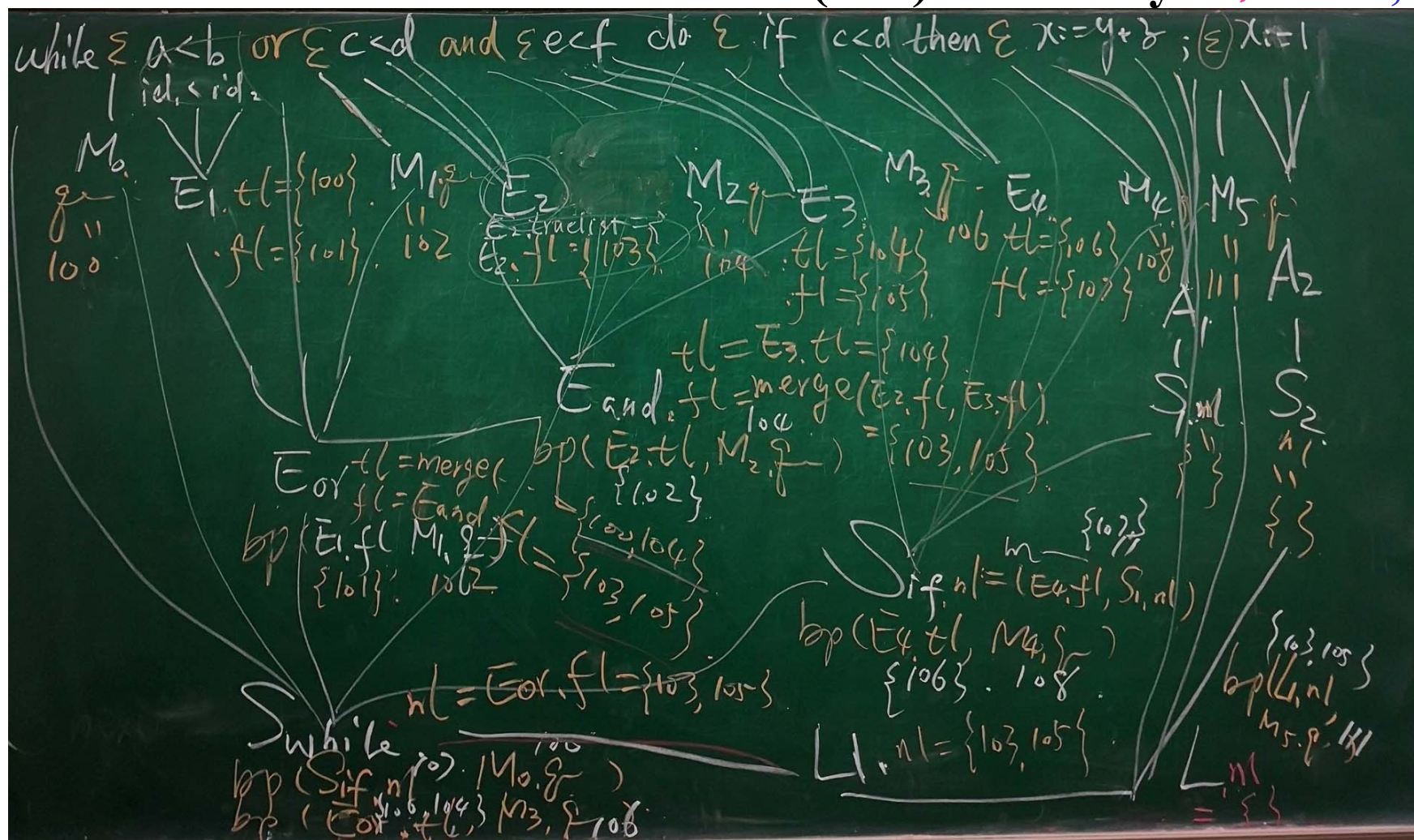
(**111**) ($:=$, **1**, $_$, x)

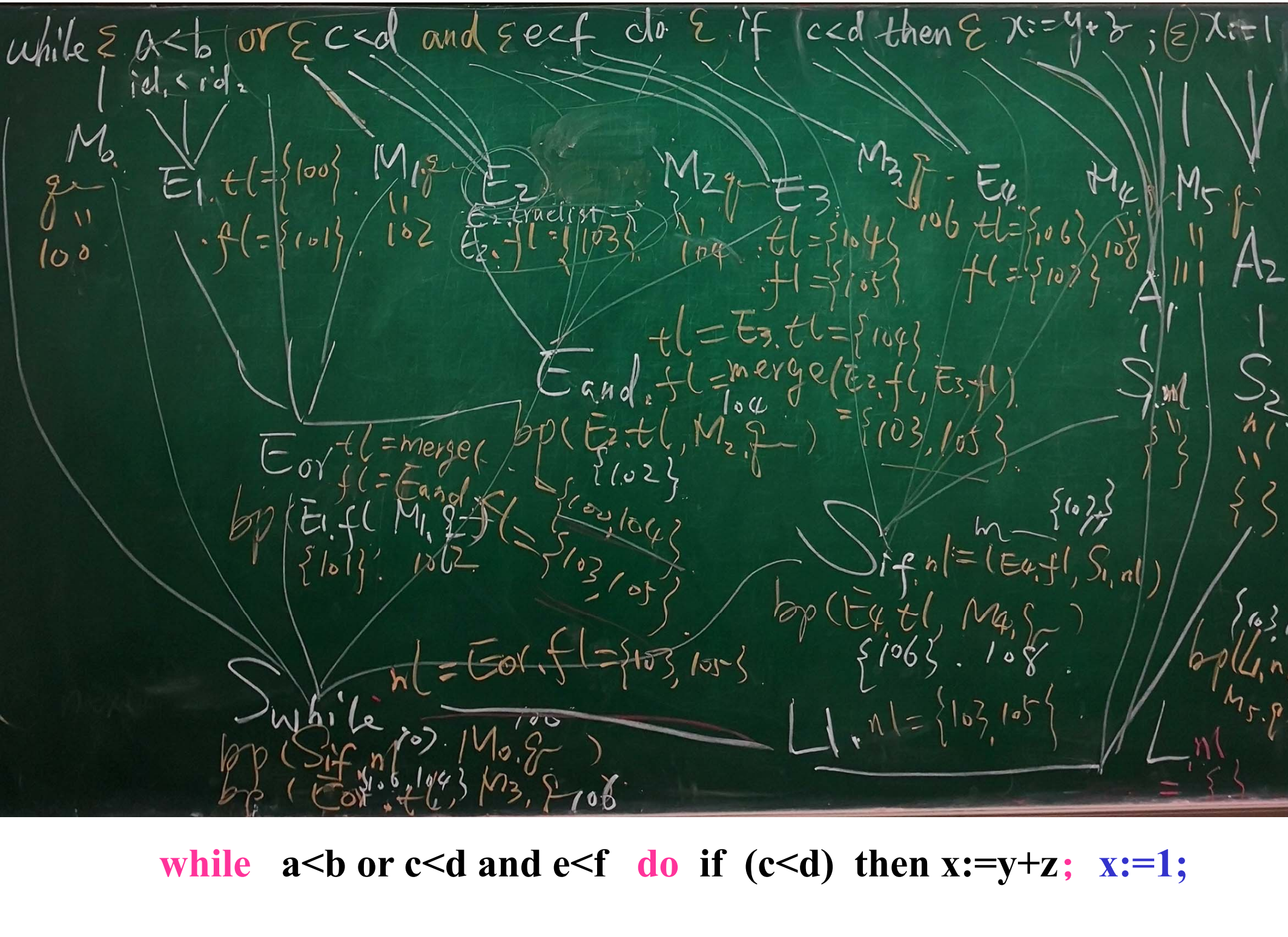
(112)

8.4 语句的翻译

4. 控制语句的翻译

while $a < b$ **or** $c < d$ **and** $e < f$ **do** **if** $(c < d)$ **then** $x := y + z$; $x := 1$;





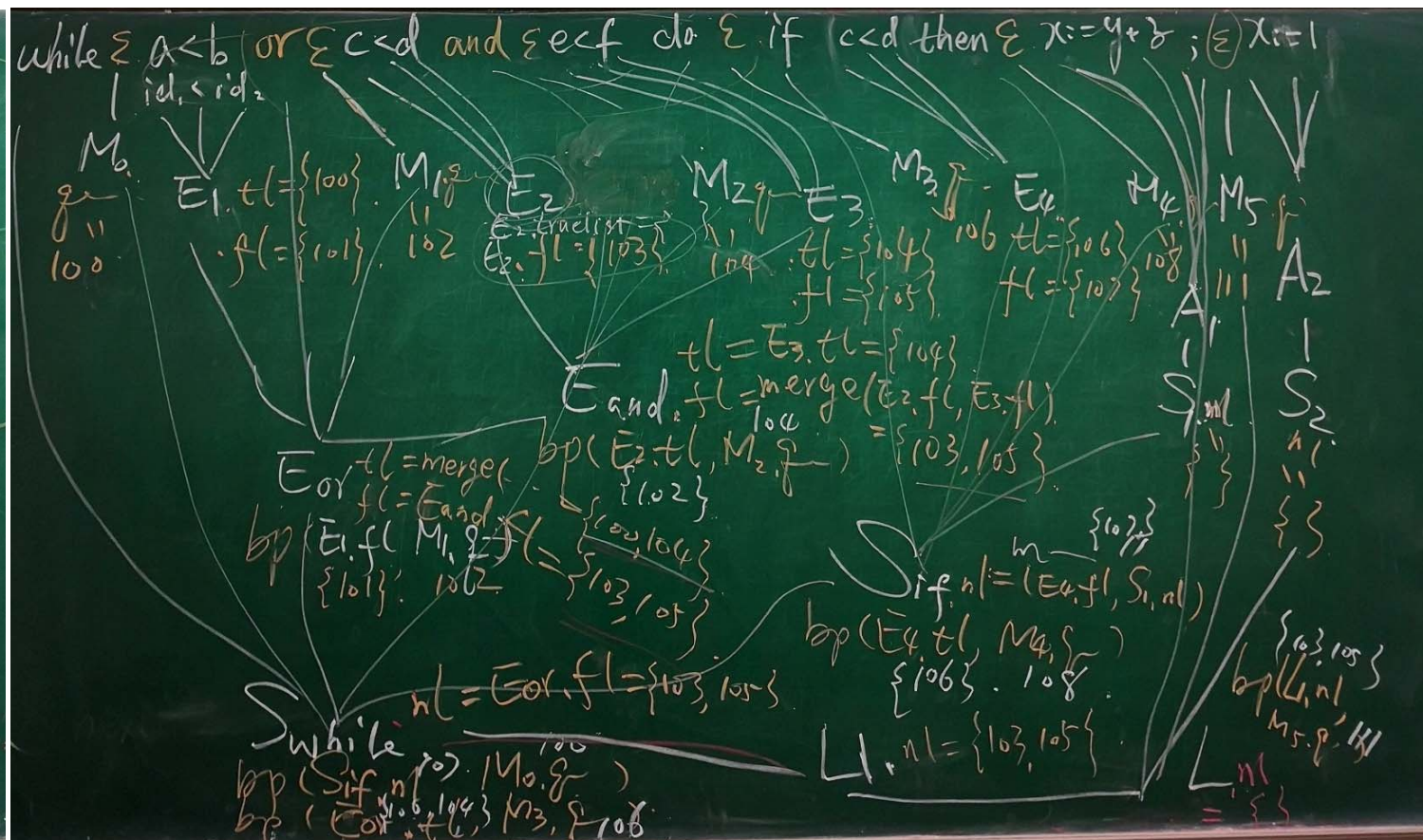
while $a < b$ or $c < d$ and $e < f$ **do** if $(c < d)$ then $x := y + z$; $x := 1$;

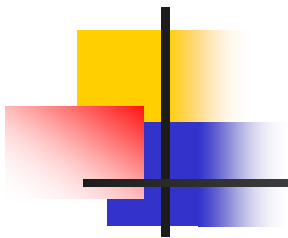
8.4 语句的翻译

4. 控制语句的翻译

while $a < b$ or $c < d$ and $e < f$ **do** if $(c < d)$ then $x := y + z$; $x := 1$;

$E_1.tl = \{100\}$ ($j < a, b, 100$)
 $E_1.fl = \{101\}$ ($j, -, -, 101$)
 $E_2.tl = \{102\}$ ($j < c, d, 102$)
 $E_2.fl = \{103\}$ ($j, -, -, 103$)
 $E_3.tl = \{104\}$ ($j < e, f, 104$)
 $E_3.fl = \{105\}$ ($j, -, -, 105$)
 $E_4.tl = \{106\}$ ($j < c, d, 106$)
 $E_4.fl = \{107\}$ ($j, -, -, 107$)
 (108) ($+, y, z, T_1$)
 (109) ($:=, T_1, -, x$)
 (110) ($j, -, -, 109$)
 (111) ($:=, 1, -, x$)
 (112) ($:=, 1, -, x$)





本章内容回顾

- ◆ 属性文法
- ◆ 目标代码结构
- ◆ 中间代码
- ◆ 控制语句的翻译



下章内容简介 —— 第9章

运行时的存储组织与分配