第78章 语法制导翻译和中间代码生成

7.1 属性文法

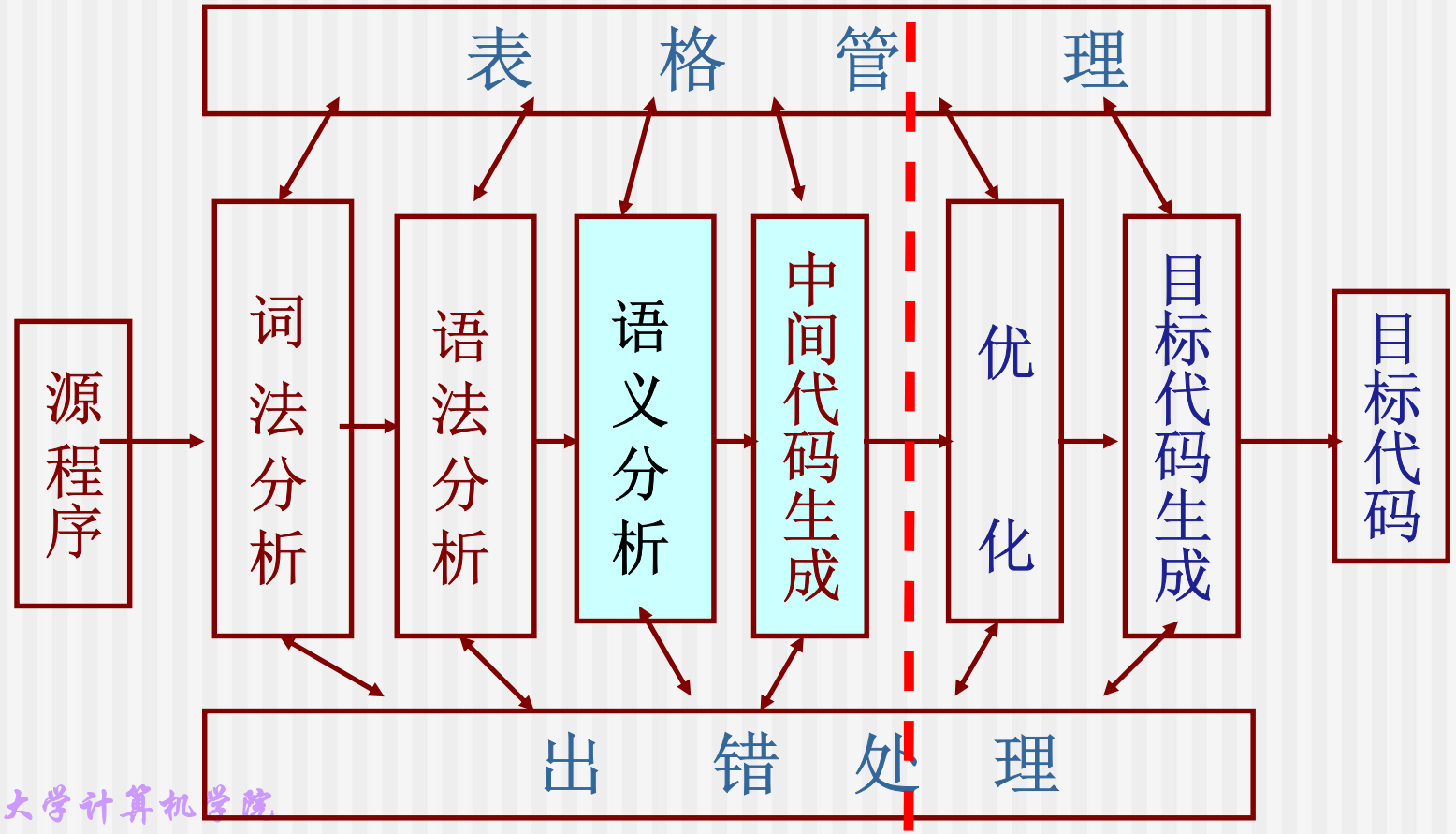
7.2 基于属性文法的语义计算

8.3 中间代码的形式

广东工业大学计算机学院

引言

- 在词法分析和语法分析之后，编译程序要做的工作就是进行静态语义检查和翻译。



语义处理的功能

- 编译程序的任务是把源程序翻译成目标程序，而且目标程序必须和源程序的语义等同。
- 编译中的语义处理是指两个功能：
- 第一，审查每个语法结构的静态语义，即验证语法结构合法的程序是否真正有意义。
- 第二，如果静态语义正确，语义处理则要执行真正的翻译：
  - **(1)** 将源程序翻译成程序的一种中间表示形式(中间代码)
  - **(2)** 将源程序翻译成目标代码。

静态语义分析的内容

- 静态语义分析通常包括（书**P203**）：
- ① 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系统的过程，编译程序必须报告不符合类型系统的信息。
- ② 控制流检查。控制流语句必须使控制转移到合法的地方。
- ③ 一致性检查。在很多场合要求对象只能被定义一次。
- ④ 相关名字检查。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。
- ⑤ 名字的作用域分析。

中间代码

- 所谓中间代码，也称中间语言，是复杂性介于源程序语言和机器语言的一种表示形式。
- 有的编译程序直接生成目标代码，有的编译程序采用中间代码。一般来说，快速编译程序直接生成目标代码，没有将中间代码翻译成目标代码的额外开销。
- 但是为了使编译程序结构在逻辑上更为简单明确，常采用中间代码，这样可以将与机器相关的某些实现细节置于代码生成阶段仔细处理，并且可以在中间代码一级进行优化工作使得代码优化比较容易实现。

语义分析的特点

- 语义分析不像词法分析和语法分析那样可以分别用正规文法和上下文无关文法描述。
- 由于语义是上下文有关的，因此语义的形式化描述是非常困难的。
- 虽然语义的形式化工作已经有相当的进展，但由于语义的复杂性，使得语义分析不能像语法那样规范，
- 到目前为止，语义的形式化描述并没有语法的形式化描述那样成熟，使得语义的描述处于一种自然语言的描述或者半形式化描述的状态。
- 而没有基于数学抽象的形式化描述，就很难设计出基于数学模型的统一算法来实现语义分析器的自动生成。

常见的语义分析方法

目前较为常见的是用属性文法作为描述程序语言语义的工具，并采用语法制导翻译的方法完成对语法成分的翻译工作。

- 直观上讲，语法制导翻译法就是为每个产生式配上一个语义子程序，在语法分析过程中，在选用某个产生式的同时，执行该产生式所对应的语义子程序来进行翻译的一种办法。

主要内容

- 本章主要内容：
- **1.** 引入属性文法和语法制导翻译方法的基本思想
- **2.** 介绍几种典型的中间代码形式
- **3.** 讨论一些常用语法成分的翻译

本课内容

- **7.1 属性文法**
- **7.2 语法制导翻译概论**
- **8.3 中间代码的形式**

8.1 属性文法

1、文法的属性

属性是指与文法符号的类型和值等有关的一些信息，在编译中用属性描述处理对象的特征。

【例】

- 判断变量 **X** 的类型是否匹配，要用 **X** 的数据类型来描述；
- 判断变量 **X** 是否存在，要用 **X** 的存储位置来描述；
- 而对变量 **X** 的运算，则要用 **X** 的值来描述；
- 因此，语义分析阶段引入 **X** 的属性，
  如 **X.type**、**X.place**、**X.val** 等来分别描述变量 **X** 的类型、存储位置以及值等不同的特征。

Knuth（克努特）在1968年首先提出属性文法

2、属性文法（属性翻译文法）

- 在上下文无关文法的基础上，为每个文法符号(终结符或非终结符)引入若干属性值（称为属性）。
- 这些属性代表与文法符号相关信息，属性可以是类型、值、代码序列、符号表内容（入口）等。
- 属性可计算或传递。属性的加工过程即是语义处理的过程。
- 对于文法的每个产生式都配备了一组属性的计算规则，称为语义规则。

属性文法定义

- 一个属性文法是一个三元组**A = (G, V, F):**
  - **G:** 上下文无关文法;
  - **V:** 属性的有穷集。
  - **F:**表示属性的断言或谓词的有穷集。
- 每个属性与某个文法符号 **X** (终结符或非终结符)相关联，用“文法符号.属性”表示这种关联。
- 每个断言与文法的某产生式相联，写在{ }内。属性的断言又称语义规则，它所描述的工作可以包括属性计算、静态语义检查、符号表的操作、代码生成等，有时写成函数或过程段。

属性文法定义

- 语义规则（Semantic Rule）

  在属性文法中，每个产生式 $A \rightarrow \alpha$ 都关联一个语义规则的集合，用于描述如何计算当前产生式中文法符号的属性值或附加的语义动作

- 属性文法中允许如下语义规则
  - 复写（copy）规则，形如

    X.a := Y.b
  - 基于语义函数（semantic function）的规则，形如

    b:=f(c_1, c_2, …, c_k) 或 **f(c_1, c_2, …, c_k)**

    其中，**b,c_1, c_2, …, c_k**是该产生式中文法符号的属性
- 实践中，语义函数的形式可以更灵活

【例】一个简单表达式文法

$E \rightarrow T^{(1)} + T^{(2)} \mid T^{(1)}$ or $T^{(2)}$

$T \rightarrow$ num | true | false

可以得到关于类型检查的属性文法：

$E \rightarrow T^{(1)} + T^{(2)}$	$\{T^{(1)}.type=int$ and $T^{(2)}.type=int\}$
$E \rightarrow T^{(1)}$ or $T^{(2)}$	$\{T^{(1)}.type=bool$ and $T^{(2)}.type=bool\}$
$T \rightarrow$ num	$\{T.type=int\}$
$T \rightarrow$ true	$\{T.type=bool\}$
$T \rightarrow$ false	$\{T.type=bool\}$

与每个非终结符E相连的有属性Type，要么是int，要么是bool.

与非终结符E的产生式相联的断言指明：两个N的属性必须相同。

✧ 属性文法举例2

– 识别语言 $L = \{ a^i b^j c^k \mid i, j, k \geq 1 \}$
不含限定条件，但显示 $a^n b^n c^n$ $(n \geq 1)$ 是合法的

产生式	语义动作
$S \rightarrow ABC$	{ if (A.num=B.num) and (B.num=C.num) then print("Accepted!") else print("Refused!") }
$A \rightarrow A_1 a$	{ A.num := A_1.num + 1 }
$A \rightarrow a$	{ A.num := 1 }
$B \rightarrow B_1 b$	{ B.num := B_1.num + 1 }
$B \rightarrow b$	{ B.num := 1 }
$C \rightarrow C_1 c$	{ C.num := C_1.num + 1 }
$C \rightarrow c$	{ C.num := 1 }

属性的分类：综合属性

- 在一个属性文法G中，对应于每个产生式A→a都有一套与之相关联的语义规则，每条规则的形式为

  b := f (c1, c2, … ck)

- **f**是一个函数。**b**和**c1, c2, … ck**是文法符号的属性。

- **(1)** 如果**b**是**A**的一个属性，并且**c1, c2, … ck**是产生式右部文法符号的属性，则称**b**是**A**的综合属性。

(0)L→E	{print(E.val)}
(1)E→E_1+T	{E.val：＝E_1.val+T.val}
(2)E→T	{E.val：＝T.val}
(3)T→T_1*F	{T.val：＝T_1.val×F.val}
(4)T→F	{T.val：＝F.val}
(5)F→(E)	{F.val：＝E.val}
(6)F→digit	{F.val＝digit.lexval}

c1　c2

综合属性b

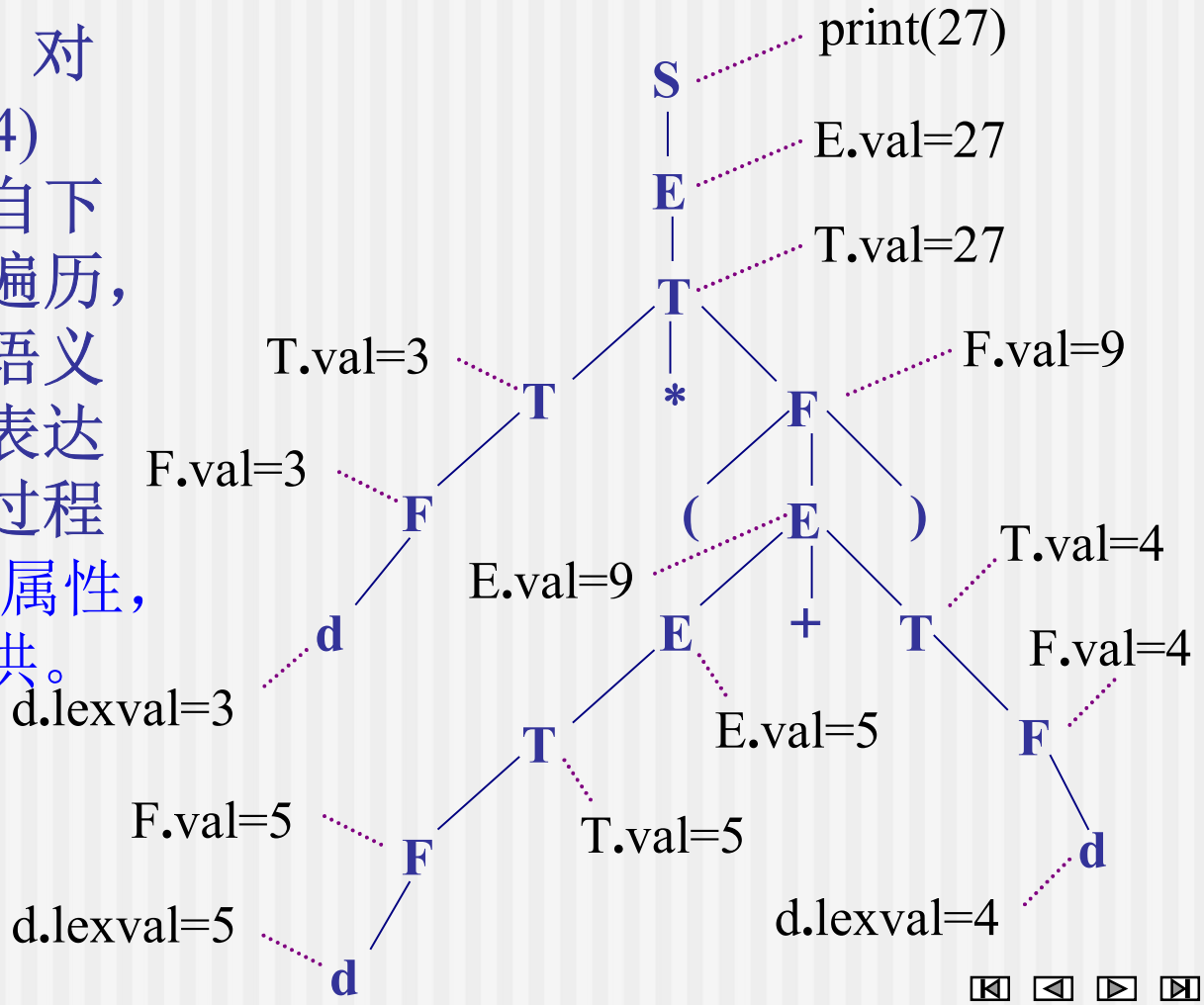
f的具体表达式

属性文法

✧ 综合属性代表自下而上传递的信息

– 接上页的例子，对表达式 3＊(5+4) 的分析树进行自下而上（后序）遍历，并执行相应的语义规则，得到该表达式的一种求值过程

•终结符只有综合属性，由词法分析器提供。



属性的分类：继承属性

- 在一个属性文法G中，对应于每个产生式A→a都有一套与之相关联的语义规则，每条规则的形式为

  b := f (c1, c2, … ck)

- f是一个函数。b和c1, c2, … ck是文法符

- (2)如果b是产生式右边某个文法符号L的一个属性，并且c1, c2, … ck是A或产生式右边任何文法符号的属性，则称b是L的继承属性。

即文法产生式右部符号的某些属性根据其左部符号的属性和右部的其它符号的某些属性计算而得。

产生式	
(1) D→TL	L.in := T.type
(2) T→int	T.type := integer
(3) T→real	T.type := real
(4) L→L¹, id	L¹.in := L.in addtype(id.entry, L.in)
(5) L→id	addtype(id.entry, L.in)

继承属性b

属性的分类：继承属性

- 继承属性用于“自上而下”传递信息。(见书**P163**）

  继承属性由相应语法树中结点的父结点属性计算得到，沿语法树向下传递，由根结点到分枝(子)结点，它反映了对上下文依赖的特性。继承属性可以很方便地用来表示程序语言上下文的结构关系。
- 非终结符可有综合属性也可有继承属性.但文法开始符号没有继承属性。

本课内容

- **7.1** 属性文法
- **7.2** 基于属性文法的语义计算
  - **S-**属性文法
  - **L-**属性文法
- **8.3** 中间代码的形式

基于属性文法的语义计算

✧ 基于属性文法的语义计算

计算方法分两类：

– 树遍历方法

通过遍历分析树进行属性计算（语法分析遍之后）

– 单遍的方法

语法分析遍的同时进行属性计算

基于属性文法的语义计算

✧ 单遍的方法

– 语法分析遍（也需建立语法分析树）的同时进行属性计算

- 自下而上方法
- 自上而下方法

– 只适用于特定文法

本课程只讨论如下两类属性文法：

- S-属性文法
- L-属性文法

单遍的方法

- 在语法分析过程中，随着分析的步步进展，每当使用一条产生式进行推导（对于自上而下分析）或归约（对于自下而上分析），就执行该产生式所对应的语义动作，完成相应的翻译工作。
- 单遍的方法，是编译器采用的是一遍扫描源代码的方式，也就是在语法分析的同时计算属性值。不论对自上而下分析或自下而上分析都适用
  - ❿S-属性文法适合一遍扫描的自下而上分析。
  - ❿L-属性文法适合一遍扫描的自上而下分析；

1. S -属性定义(S-属性文法)

- **S-属性文法**：仅包含综合属性的属性文法。
- 例如：算术表达式求值的属性文法：

产生式	语义规则
(0)L→E	{print(E.val)}
(1)E→E_1+T	{E.val：=E_1.val+T.val}
(2)E→T	{E.val：=T.val}
(3)T→T_1*F	{T.val：=T_1.val×F.val}
(4)T→F	{T.val：=F.val}
(5)F→(E)	{F.val：=E.val}
(6)F→digit	{F.val=digit.lexval}

- **S-属性文法**的翻译器通常借助**LR**分析器来实现

2. L-属性定义(L-属性文法)

- **L-属性文法**：既包括综合属性，又包括继承属性。即对于所有 $A \rightarrow X_1\ X_2 \cdots X_n$，至少下面条件之一成立：
- **(1)** 每一个属性都是一个综合属性；
- **(2)** 计算 X_i 的属性时，仅使用**A**的属性和 $X_1, X_2, \cdots, X_{i-1}$ 的属性。
- 注意：
- **(1)** 在**L-**属性文法中，其属性可用深度优先的顺序从左至右计算。
- **(2) S-**属性文法是的**L-**属性文法特例。

L-属性文法举例

- **(1)** 每一个属性都是一个综合属性；
- **(2)** 计算X_i的继承属性时，仅使用**A**的属性和$X_1, X_2, \cdots, X_{i-1}$的属性。

产生式	语义规则
$A \rightarrow LM$	L.i:=l(A.i) M.i:=m(L.s)
$A \rightarrow QR$	A.s:=f(M.s) R.i:=r(A.i) Q.i:=q(R.s) A.s:=f(Q.s)

- 表中的属性文法不是**L-**属性文法。
- 文法符号**Q**的继承属性依赖于它右边文法符号**R**的属性。

本课内容

- **7.1** 属性文法
- **7.2** 语法制导翻译概论
- **8.3** 中间代码的形式
- **8.4** 简单赋值语句的翻译

中间代码的形式

- 中间代码：一种介于源语言和目标语言之间的中间语言形式。

- 编译程序所使用的中间代码有多种形式(书**P208**)。常见的有：
- 逆波兰记号表示
- 三元式表示**TAC**
- 四元式表示
- 树形表示**AST**

四元式表示

- 四元式是一种比较普遍采用的中间代码形式：

  算符**op**，运算对象**ARG1**，运算对象**ARG2**，运算结果**RESULT**

- 例如**a := b * c + b * d**的四元式表示：

  （1）（*，b，c，t_1）
  （2）（*，b，d，t_2）
  （3）（+，t_1，t_2，t_3）
  （4）（:=，t_3，－，a）

  （1）（*，b，c）
  （2）（*，b，d）
  （3）（+，（1），（2））
  （4）（：＝，（3），a）

- 也就是说，四元式之间的联系是通过临时变量实现的。

四元式表示的另一种形式

- 有时把四元式序列写成(以a := b * c + b * d为例):

（1）（*，b，c，t_1）	（1）t_1 := b*c
（2）（*，b，d，t_2）	（2）t_2 := b*d
（3）（+，t_1，t_2，t_3）	（3）t_3 := t_1+t_2
（4）（:=，t_3，—，a）	（4）a := t_3

- 另外，把（jump，—，—，L）写成goto L，把（jrop，B，C，L）写成if B rop C goto L。
- 四元式的通用性：四元式表示很类似于三地址指令，有时称之称为“三地址代码”(书上称为TAC)。
- 这种表示可看作一种虚拟三地址机的通用汇编码，即这种虚拟机的每条“指令”包含操作符和三个地址，其中两个地址存放运算对象，另一个地址是存放结果。

中间代码表示举例

- 请将表达式**-(a+b)*(c+d)-(a+b)**表示成四元式序列：

四元式表示：

(1) (+, a, b, t1)　　**(2)** (+, c, d, t2)

(3) (*, t1, t2, t3)　　**(4)** (-, t3, /, t4)

(5) (+, a, b, t5)　　**(6)** (-, t4, t5, t6)

本课内容

- **7.1** 属性文法
- **7.2** 语法制导翻译概论
- **8.3** 中间代码的形式
- **8.4** 简单赋值语句的翻译

简单赋值语句的翻译

- 简单赋值语句

  是指不含复杂数据类型（如数组，记录等）的赋值语句。
- 赋值语句的语义审查包括：
  1. 每个使用性标识符是否都有声明？
  2. 运算符的分量类型是否相容？
  3. 赋值语句的左右部的类型是否相容？
- 赋值语句的翻译目标：

  在赋值语句右部表达式产生的四元式序列后加一条赋值四元式（三地址码）。

一些常用的属性和过程

一些常用的属性：

id.name：表示变量的名称。

E.place：表示存放**E**值的变量名在符号表的登录项地址，或一整数码**(**若此变量是一个临时变量**)**。

一些常用的过程：

lookup()： 语义审查函数。**Lookup(id.name)**表示审查**id.name**是否出现在符号表中。

emit()：生成一条指令（新书命令**gen**）。例如：

emit(E.place ':=' E1.place '+' E2.place)

newtemp()：分配一个工作单元。

赋值语句的翻译

- 下面列出了把赋值语句翻译成四元式的语义描述。这里的语义工作包括对变量进行“先定义后使用”的检查。
- (pl0P442)

赋值语句	翻译(四元式语义描述)
(1) *S*→*id* := *E*	{ p := lookup(id.name); if p ≠ nil then emit(p ':=' E.place) else error }
(2) E→E1+E2 (书P447)	{ E.place := newtemp; emit(E.place ':=' E1.place '+' E2.place) }
(3) E→E1*E2	{ E.place := newtemp; emit(E.place ':=' E1.place '*' E2.place) }

赋值语句的翻译(续1)

- 下面列出了把赋值语句翻译成四元式的语义描述。这里的语义工作包括对变量进行“先定义后使用”的检查。

赋值语句	翻译(四元式语义描述)
(4) E→－E1	{ E.place := newtemp; emit(E.place ':=' ' uminus' E1.place) }
(5) E→(E1)	{ E.place := E1.place }
(6) E→id	{ p := lookup(id.name); if p ≠ nil then E. place := p else error }

简单赋值语句的翻译扩展讨论

- 实际上，在一个表达式中可能会出现各种不同类型的变量或常数作混合运算。

(2) E→E1+E2	{ E.place := newtemp; emit(E.place ':=' E1.place '+' E2.place) }

- (1) 如不能接受不同类型的运算对象的混合运算，则应指出错误；
- (2) 如能接受混合运算，则应进行类型转换的语义处理。
- 假如上例中的表达式可以有混合运算，id可以是实型量或者整型量，并且约定必须首先将整型量转换为实型量。

增加语义变量和算符

- 为进行类型转换的语义处理，增加语义变量和运算符：
- **E.type**：表示**E**的类型信息，**E.type**的值或为**int**或为**real**。
- 一目算符**itr**：将整型运算对象转换为实型。
- 此外，为区别整型加(乘)和实型加(乘)，把+（*）分别写作$+^{i}$（$*^{i}$）和$+^{r}$（$*^{r}$）。

E→E1*E2的扩充翻译讨论

- 对于产生式E→E1*E2，由于E1和E2有可能是int类型或为real类型，所以可能有4种情况：
- (1) E1：int　　E2：int
- (2) E1：real　　E2：real
- (3) E1：int　　E2：real
- (4) E1：real　　E2：int

需要将E1转换为real类型

需要将E2转换为real类型

E→E1*E2的扩充翻译方案

- 产生式　　语义动作
- E→E1*E2　{ E.place := newtemp；
- 　　if E1.type = int AND E2.type = int then
- 　　begin
- 　　emit(E.place, ‘:=‘，E1.place，’ *i’,
- 　　E2.place);
- 　　E．type：=int
- 　　end
- 　　……
- 　　}

生成一条类似这样的中间代码：E.id := E1.id * E2.id

E→E1*E2的扩充翻译方案(续1)

- 产生式　　语义动作
- E→E1*E2　　{ ……

```
else if E1.type = real AND
                        E2.type = real then
begin
    emit (E.place, ':=',
                E1.place, '*r', E2.place);
    E.type := real
end
```

- }

E→E1*E2的扩充翻译方案(续2)

- 产生式　　语义动作
- E→E1*E2　　{ …… /*E1.type = int and E2. type = real*/
- ■

```
else if E1.type = int then
begin
    t := newtemp;
    emit(t,  ':=', 'itr',  E1.place);
    emit(E.place,  ':=', t, '*r',
                                E2.place);
    E.type：=real
end
}
```

把E1转换成real类型，生成四元式： (itr, E1.place, -, t)

E→E1*E2的扩充翻译方案(续3)

- 产生式　　语义动作
- E→E1*E2　　{ ……

```
else /*E1.type = real and E2. type = int*/
begin
    t := newtemp;
    emit(t,    ':=', 'itr',  E2.place);
    emit(E.place, ':=', E1.place, '*r',
                                        t);
    E.type := real
end
```

- }

把E2转换成int类型，生成四元式：　(itr, E2.place, -, t)

作业

- 整章一起布置