

编译原理

第七章 语法制导翻译及中间代码生成

方徽星

扬州大学 信息工程学院 (505)

fanghuixing@yzu.edu.cn

2018 年 6 月

内容提要

1 语法制导翻译

- 语法制导定义
- S 属性定义的自下而上计算
- L 属性定义的自上而下计算
- L 属性定义的自下而上计算

2 中间代码生成

- 中间语言
- 声明语句
- 赋值语句
- 布尔表达式和控制流语句

1.1 语法制导定义 (Syntax-Directed Definition)

- SDD 是一个附带有**属性**及**语义规则**的**上下文无关文法**
 - ☞ 每个文法符号有一组属性, b
 - ☞ 每个产生式 ($A \rightarrow \alpha$) 有一组语义规则, $b = f(c_1, c_2, \dots, c_k)$

1.1 语法制导定义 (Syntax-Directed Definition)

- SDD 是一个附带有**属性**及**语义规则**的**上下文无关文法**
 - ☞ 每个文法符号有一组属性, b
 - ☞ 每个产生式 ($A \rightarrow \alpha$) 有一组语义规则, $b = f(c_1, c_2, \dots, c_k)$
- b 称为 A 的**综合属性 (synthesized attribute)**:
 - ☞ 如果 b 是 A 的属性, 且
 - ☞ c_1, c_2, \dots, c_k 是产生式右部文法符号的属性或 A 的其他属性

1.1 语法制导定义 (Syntax-Directed Definition)

- SDD 是一个附带有**属性**及**语义规则**的**上下文无关文法**
 - ☞ 每个文法符号有一组属性, b
 - ☞ 每个产生式 ($A \rightarrow \alpha$) 有一组语义规则, $b = f(c_1, c_2, \dots, c_k)$
- b 称为 A 的**综合属性 (synthesized attribute)**:
 - ☞ 如果 b 是 A 的属性, 且
 - ☞ c_1, c_2, \dots, c_k 是产生式右部文法符号的属性或 A 的其他属性
- b 称为 X 的**继承属性 (inherited attribute)**:
 - ☞ 如果 b 是产生式右部某个文法符号 X 的属性, 且
 - ☞ c_1, c_2, \dots, c_k 是 A 的属性或右部文法符号的属性

1.1 语法制导定义 (Syntax-Directed Definition)

- SDD 是一个附带有**属性**及**语义规则**的**上下文无关文法**
 - ☞ 每个文法符号有一组属性, b
 - ☞ 每个产生式 ($A \rightarrow \alpha$) 有一组语义规则, $b = f(c_1, c_2, \dots, c_k)$
- b 称为 A 的**综合属性 (synthesized attribute)**:
 - ☞ 如果 b 是 A 的属性, 且
 - ☞ c_1, c_2, \dots, c_k 是产生式右部文法符号的属性或 A 的其他属性
- b 称为 X 的**继承属性 (inherited attribute)**:
 - ☞ 如果 b 是产生式右部某个文法符号 X 的属性, 且
 - ☞ c_1, c_2, \dots, c_k 是 A 的属性或右部文法符号的属性
- b 依赖于 c_1, c_2, \dots, c_k

1.1 语法制导定义 (Syntax-Directed Definition)

- **终结符只有综合属性**，属性值由词法分析器提供，SDD 中没有计算终结符号属性值的语义规则
- **属性文法**：语义规则函数没有副作用的 SDD，其规则仅仅通过其他属性值和常量值来定义一个属性值

产生式	语义规则
1 $L \rightarrow E \mathbf{n}$	$L.val = E.val$
2 $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3 $E \rightarrow T$	$E.val = T.val$
4 $T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
5 $T \rightarrow F$	$T.val = F.val$
6 $F \rightarrow (E)$	$F.val = E.val$
7 $F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit.lexval}$

1.1 语法制导定义 (Syntax-Directed Definition)

- 仅仅使用综合属性的语法制导定义称为 **S 属性定义**
- 对于 S 属性定义，分析树各结点属性的计算可以自下而上地完成，通过计算语义规则而得到结点的属性
- 每个结点的属性值都标注出来的分析树称为**注释分析树**，计算各结点属性值的过程叫做分析树的**注释**或**修饰**

1.1 语法制导定义 (Syntax-Directed Definition)

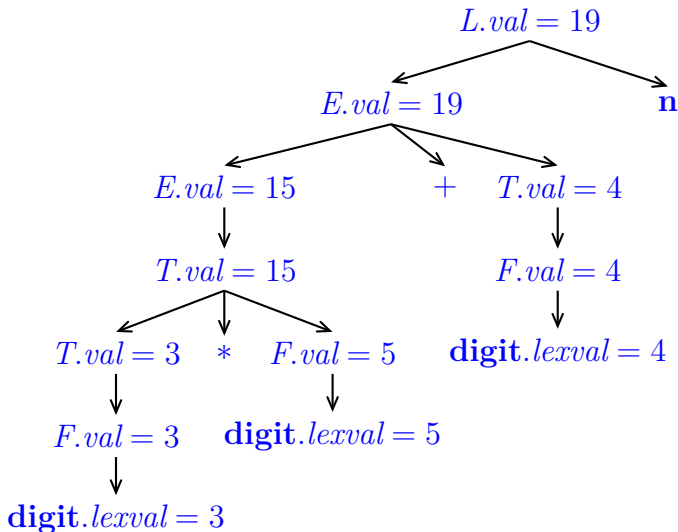


图: $3 * 5 + 4n$ 的注释分析树

1.1 语法制导定义 (Syntax-Directed Definition)

- 在分析树中, 结点的**继承属性**是由它的**兄弟**结点、**父**结点和**自己**的属性来定义的
- 非终结符 T 有综合属性 $type$, L 有继承属性 in , $addType$ 把类型信息加到符号表中各个标识符的条目中:

产生式	语义规则
1 $D \rightarrow TL$	$L.in = T.type$
2 $T \rightarrow \mathbf{int}$	$T.type = integer$
3 $T \rightarrow \mathbf{real}$	$T.type = real$
4 $L \rightarrow L_1, \mathbf{id}$	$L_1.in = L.in \quad addType(\mathbf{id}.entry, L.in)$
5 $L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.in)$

$\mathbf{id}.entry$: 在词法分析过程中得到的一个指向某个符号表对象的值

1.1 语法制导定义 (Syntax-Directed Definition)

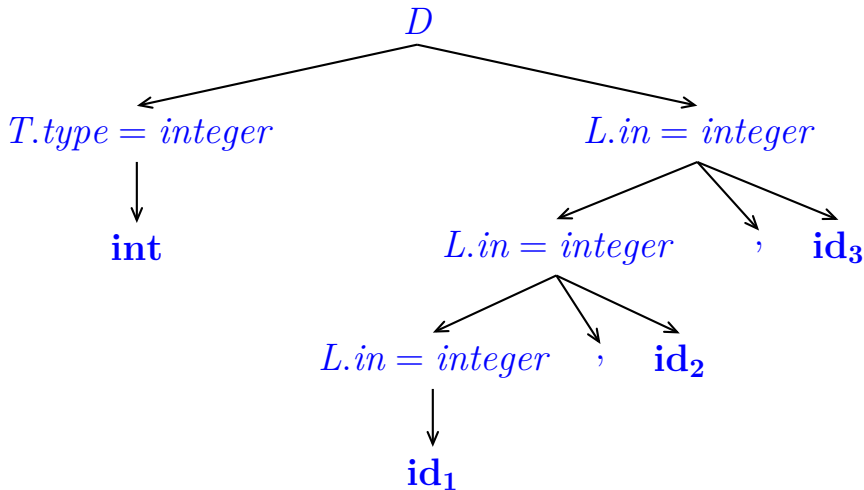
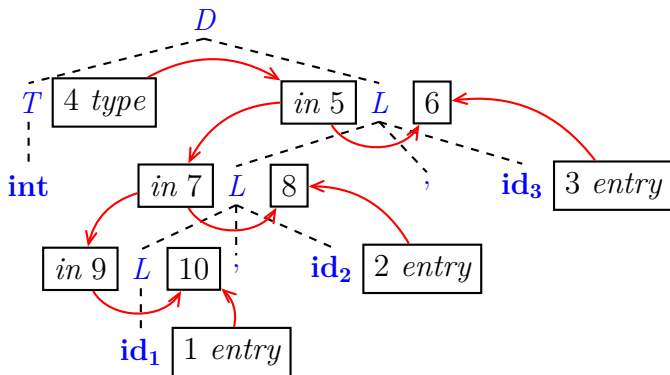


图: `int id1, id2, id3` 的注释分析树

1.1 语法制导定义 (Syntax-Directed Definition)

- 分析树结点属性之间的依赖关系可以使用**依赖图**进行描述
- 分析树中每个结点属性在依赖图中有的结点与之对应
- 属性之间的依赖，通过依赖图中相应结点的有向边来体现



1.1 语法制导定义 (Syntax-Directed Definition)

- 拓扑排序是有向无环图中的结点的一种排序

☞ $(m_i \rightarrow m_j) \implies m_i \text{ 先于 } m_j$

☞ 由拓扑排序可以确定依赖图中属性的计算顺序

☞ 可保证由规则 $b = f(c_1, \dots, c_k)$ 计算 b 时, 属性 c_i 已计算出来

1.1 语法制导定义 (Syntax-Directed Definition)

- 拓扑排序是有向无环图中的结点的一种排序
 - ☞ $(m_i \rightarrow m_j) \implies m_i \text{ 先于 } m_j$
 - ☞ 由拓扑排序可以确定依赖图中属性的计算顺序
 - ☞ 可保证由规则 $b = f(c_1, \dots, c_k)$ 计算 b 时, 属性 c_i 已计算出来
- 翻译过程可以按如下步骤完成

1.1 语法制导定义 (Syntax-Directed Definition)

- 拓扑排序是有向无环图中的结点的一种排序
 - ☞ $(m_i \rightarrow m_j) \implies m_i$ 先于 m_j
 - ☞ 由拓扑排序可以确定依赖图中属性的计算顺序
 - ☞ 可保证由规则 $b = f(c_1, \dots, c_k)$ 计算 b 时, 属性 c_i 已计算出来
- 翻译过程可以按如下步骤完成
 - ☞ 首先根据文法构造**输入的分析树**

分析树方法

1.1 语法制导定义 (Syntax-Directed Definition)

- 拓扑排序是有向无环图中的结点的一种排序
 - ☞ $(m_i \rightarrow m_j) \implies m_i$ 先于 m_j
 - ☞ 由拓扑排序可以确定依赖图中属性的计算顺序
 - ☞ 可保证由规则 $b = f(c_1, \dots, c_k)$ 计算 b 时, 属性 c_i 已计算出来
- 翻译过程可以按如下步骤完成
 - ☞ 首先根据文法构造**输入的分析树**
 - ☞ 再由分析树构造属性依赖图 (**因此依赖图是动态构建的**)

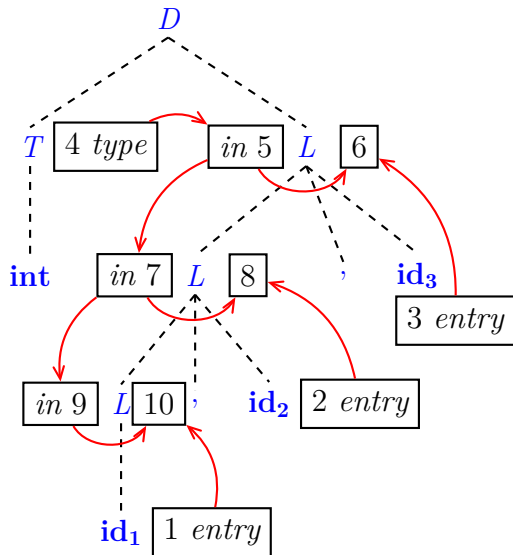
分析树方法

1.1 语法制导定义 (Syntax-Directed Definition)

- 拓扑排序是有向无环图中的结点的一种排序
 - ☞ $(m_i \rightarrow m_j) \implies m_i$ 先于 m_j
 - ☞ 由拓扑排序可以确定依赖图中属性的计算顺序
 - ☞ 可保证由规则 $b = f(c_1, \dots, c_k)$ 计算 b 时, 属性 c_i 已计算出来
- 翻译过程可以按如下步骤完成
 - ☞ 首先根据文法构造**输入的分析树**
 - ☞ 再由分析树构造属性依赖图 (**因此依赖图是动态构建的**)
 - ☞ 对依赖图中的结点进行拓扑排序, 得到语义规则的计算顺序
 - ☞ 按照计算顺序得到输入串的翻译

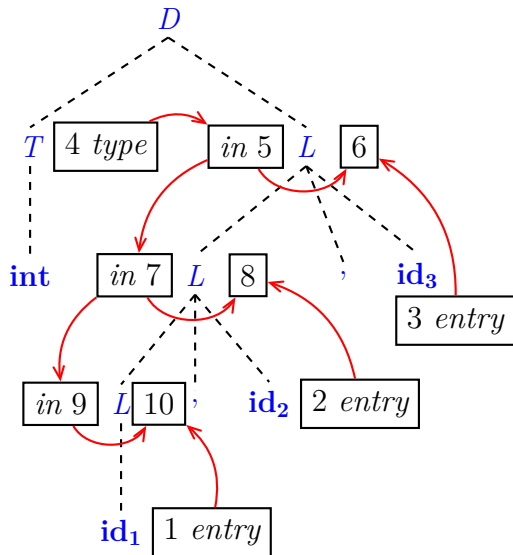
分析树方法

1.1 语法制导定义 (Syntax-Directed Definition)



结点 1,2,3 的属性 *entry* 由
词法分析器提供

1.1 语法制导定义 (Syntax-Directed Definition)



属性计算

序号为 n 的结点属性记为 a_n

☞ $a_4 = integer$

☞ $a_5 = a_4$

☞ $addType(id_3.entry, a_5)$

☞ $a_7 = a_5$

☞ $addType(id_2.entry, a_7)$

☞ $a_9 = a_7$

☞ $addType(id_1.entry, a_9)$

结点 1,2,3 的属性 *entry* 由词法分析器提供

1.1 语法制导定义 (Syntax-Directed Definition)

- 分析树方法缺点
 - ☞ 若依赖图有环，则失败
 - ☞ 速度慢，计算顺序是动态和即时确定的

1.1 语法制导定义 (Syntax-Directed Definition)

- 分析树方法缺点
 - ☞ 若依赖图有环，则失败
 - ☞ 速度慢，计算顺序是动态和即时确定的
- 如何改进？

1.1 语法制导定义 (Syntax-Directed Definition)

- 分析树方法缺点

- ☞ 若依赖图有环，则失败
- ☞ 速度慢，计算顺序是动态和即时确定的

- 如何改进？

- ☞ **基于规则的方法：**

- 在编译前，手工或工具对产生式的语义规则进行分析，得到计算顺序
- 缺点：对依赖关系复杂的 SDD 很难事先确定计算顺序

1.1 语法制导定义 (Syntax-Directed Definition)

- 分析树方法缺点

- ☞ 若依赖图有环，则失败
- ☞ 速度慢，计算顺序是动态和即时确定的

- 如何改进？

- ☞ **基于规则的方法：**

- 在编译前，手工或工具对产生式的语义规则进行分析，得到计算顺序
- 缺点：对依赖关系复杂的 SDD 很难事先确定计算顺序

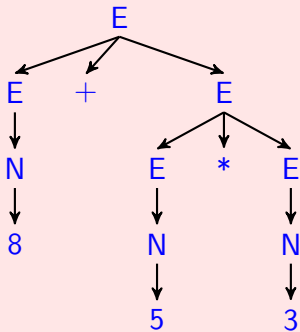
- ☞ **忽略规则的方法：**

- 根据编译器生成工具的计算策略限定编译器设计者所提供的语义规则的形式
- 缺点：限制了 SDD 的形式

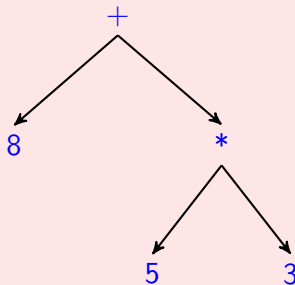
1.2 S 属性定义的自下而上计算

- 以构造抽象语法树为例，熟悉 S 属性定义及在自下而上分析过程中完成属性计算

语法分析树



抽象语法树



1.2 S 属性定义的自下而上计算

- $mkLeaf(id, entry)$: 建立标记为 id 的标识符结点, 该结点另一个域的值为 $entry$, 为符号表中该标识符条目的指针

产生式	语义规则
1 $E \rightarrow E_1 + T$	$E.nptr = mkNode('+', E_1.nptr, T.nptr)$
2 $E \rightarrow T$	$E.nptr = T.nptr$
3 $T \rightarrow T_1 * F$	$T.nptr = mkNode('*', T_1.nptr, F.nptr)$
4 $T \rightarrow F$	$T.nptr = F.nptr$
5 $F \rightarrow (E)$	$F.nptr = E.nptr$
6 $F \rightarrow id$	$F.nptr = mkLeaf(id, id.entry)$
7 $F \rightarrow num$	$F.nptr = mkLeaf(num, num.val)$

综合属性 $nptr$ 是用来记住函数调用返回的指针

1.2 S 属性定义的自下而上计算

- $mkLeaf(id, val)$: 建立标记为 **num** 的整数结点, 该结点另一个域的值为 val , 为该整数的值

产生式	语义规则
1 $E \rightarrow E_1 + T$	$E.nptr = mkNode('+', E_1.nptr, T.nptr)$
2 $E \rightarrow T$	$E.nptr = T.nptr$
3 $T \rightarrow T_1 * F$	$T.nptr = mkNode('*', T_1.nptr, F.nptr)$
4 $T \rightarrow F$	$T.nptr = F.nptr$
5 $F \rightarrow (E)$	$F.nptr = E.nptr$
6 $F \rightarrow id$	$F.nptr = mkLeaf(id, id.entry)$
7 $F \rightarrow num$	$F.nptr = mkLeaf(num, num.val)$

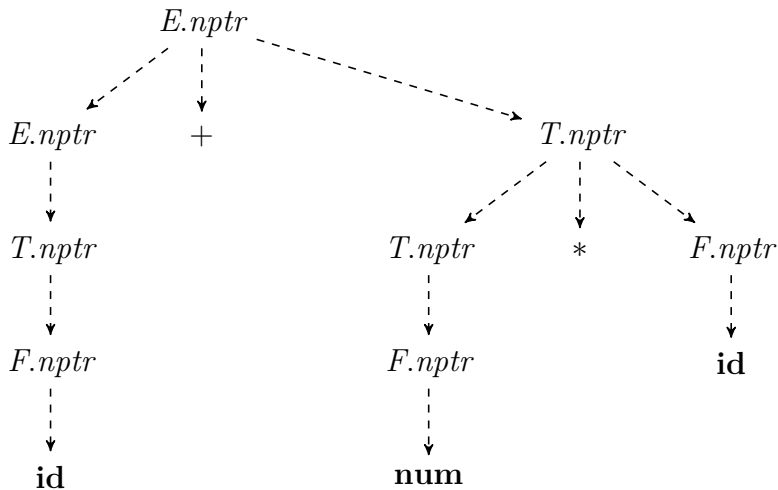
1.2 S 属性定义的自下而上计算

- $mkNode(op, left, right)$: 建立标记为 op 的算法结点, $left$ 和 $right$ 分别是左右子树的指针

产生式	语义规则
1 $E \rightarrow E_1 + T$	$E.nptr = mkNode('+', E_1.nptr, T.nptr)$
2 $E \rightarrow T$	$E.nptr = T.nptr$
3 $T \rightarrow T_1 * F$	$T.nptr = mkNode('*', T_1.nptr, F.nptr)$
4 $T \rightarrow F$	$T.nptr = F.nptr$
5 $F \rightarrow (E)$	$F.nptr = E.nptr$
6 $F \rightarrow id$	$F.nptr = mkLeaf(id, id.entry)$
7 $F \rightarrow num$	$F.nptr = mkLeaf(num, num.val)$

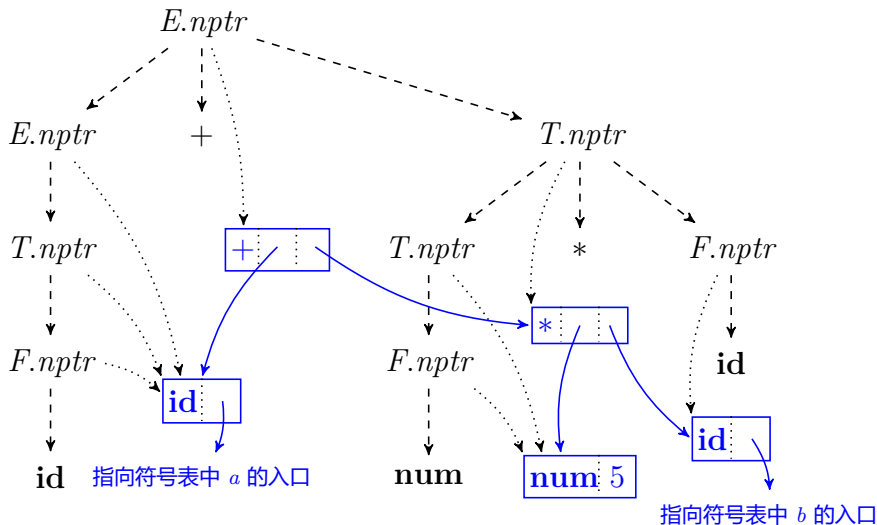
1.2 S 属性定义的自下而上计算

考虑表达式 $a + 5 * b$, 构造注释分析树和抽象语法树



1.2 S 属性定义的自下而上计算

考虑表达式 $a + 5 * b$, 构造注释分析树和抽象语法树



1.2 S 属性定义的自下而上计算

对于输入表达式: $a + 5 * b$, 实际执行的函数调用如下:

☞ $p_1 = mkLeaf(\mathbf{id}, extry_a)$

1.2 S 属性定义的自下而上计算

对于输入表达式: $a + 5 * b$, 实际执行的函数调用如下:

☞ $p_1 = mkLeaf(\mathbf{id}, extry_a)$

☞ $p_2 = mkLeaf(\mathbf{num}, 5)$

1.2 S 属性定义的自下而上计算

对于输入表达式: $a + 5 * b$, 实际执行的函数调用如下:

☞ $p_1 = mkLeaf(\mathbf{id}, entry_a)$

☞ $p_2 = mkLeaf(\mathbf{num}, 5)$

☞ $p_3 = mkLeaf(\mathbf{id}, entry_b)$

1.2 S 属性定义的自下而上计算

对于输入表达式: $a + 5 * b$, 实际执行的函数调用如下:

☞ $p_1 = mkLeaf(\mathbf{id}, entry_a)$

☞ $p_2 = mkLeaf(\mathbf{num}, 5)$

☞ $p_3 = mkLeaf(\mathbf{id}, entry_b)$

☞ $p_4 = mkNode('*', p_2, p_3)$

1.2 S 属性定义的自下而上计算

对于输入表达式: $a + 5 * b$, 实际执行的函数调用如下:

☞ $p_1 = mkLeaf(\mathbf{id}, entry_a)$

☞ $p_2 = mkLeaf(\mathbf{num}, 5)$

☞ $p_3 = mkLeaf(\mathbf{id}, entry_b)$

☞ $p_4 = mkNode('*', p_2, p_3)$

☞ $p_5 = mkNode('+', p_1, p_4)$

1.2 S 属性定义的自下而上计算

- 综合属性可由**自下而上的语法分析器**在分析输入的同时计算
 - 👉 **栈**中增加一个域来**保存**文法符号的**综合属性**
 - 👉 **归约**时计算**产生式头**符号的综合属性

	\dots	\dots
$top \rightarrow$	Z	$Z.z$
$top - 1 \rightarrow$	Y	$Y.y$
$top - 2 \rightarrow$	X	$X.x$
	\dots	\dots
	state	val

1.2 S 属性定义的自下而上计算

若产生式 $A \rightarrow XYZ$ 的语义规则是 $A.a = f(X.x, Y.y, Z.z)$

- 归约前, 属性 $Z.z$ 的值存在 $stack[top].val$, ...
- 归约时, A 存在 $stack[top-2].state$, 综合属性的值 $A.a$ 放入 $stack[top-2].val$, 然后 $top := top - 2$

$top \rightarrow$	Z	$Z.z$
$top - 1 \rightarrow$	Y	$Y.y$
$top - 2 \rightarrow$	X	$X.x$

	state	val

1.2 S 属性定义的自下而上计算

使用 LR 分析器实现计算器

产生式	语义规则
1 $L \rightarrow E \mathbf{n}$	$L.val = E.val$
2 $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3 $E \rightarrow T$	$E.val = T.val$
4 $T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
5 $T \rightarrow F$	$T.val = F.val$
6 $F \rightarrow (E)$	$F.val = E.val$
7 $F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit.lexval}$

1.2 S 属性定义的自下而上计算

使用 LR 分析器实现计算器

产生式	语义动作
1 $L \rightarrow E \mathbf{n}$	$\{print(stack[top - 1].val); top = top - 1;\}$
2 $E \rightarrow E_1 + T$	$\{stack[top - 2].val = stack[top - 2].val + stack[top].val; top = top - 2;\}$
3 $E \rightarrow T$	
4 $T \rightarrow T_1 * F$	$\{stack[top - 2].val = stack[top - 2].val \times stack[top].val; top = top - 2;\}$
5 $T \rightarrow F$	
6 $F \rightarrow (E)$	$\{stack[top - 2].val = stack[top - 1].val; top = top - 2;\}$
7 $F \rightarrow \mathbf{digit}$	

1.2 S 属性定义的自下而上计算

翻译表达式 $8 + 5 * 2n$

输入	state	val	所用产生式
$8 + 5 * 2n$	-	-	
$+5 * 2n$	8	8	
$+5 * 2n$	F	8	$F \rightarrow \text{digit}$
$+5 * 2n$	T	8	$T \rightarrow F$
...
n	$E + T * F$	$8 + 5 * 2$	$F \rightarrow \text{digit}$
n	$E + T$	$8 + 10$	$T \rightarrow T * F$
...
	L	18	$L \rightarrow E n$

1.3 L 属性定义的自上而下计算

- 若考虑继承属性呢？

1.3 L 属性定义的自上而下计算

L 属性定义

如果语法制导定义 \mathcal{D} 的**每个**产生式 $A \rightarrow X_1 X_2 \dots X_n$ 的**每条**语义规则 \mathcal{R} 满足:

- ① \mathcal{R} 计算的属性是 A 的综合属性, 或
- ② \mathcal{R} 计算的是 X_j 的继承属性, 且仅依赖 $X_{i < j}$ 的属性和 A 的继承属性

则称 \mathcal{D} 是 L 属性的

S 属性定义属于 L 属性定义

1.3 L 属性定义的自上而下计算

L 属性定义

产生式	语义规则
1 $D \rightarrow TL$	$L.in = T.type$
2 $T \rightarrow \mathbf{int}$	$T.type = integer$
3 $T \rightarrow \mathbf{real}$	$T.type = real$
4 $L \rightarrow L_1, \mathbf{id}$	$L_1.in = L.in \quad addType(\mathbf{id}.entry, L.in)$
5 $L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.in)$

1.3 L 属性定义的自上而下计算

L 属性定义

产生式	语义规则
1 $D \rightarrow TL$	$L.in = T.type$
2 $T \rightarrow \text{int}$	$T.type = integer$
3 $T \rightarrow \text{real}$	$T.type = real$
4 $L \rightarrow L_1, \text{id}$	$L_1.in = L.in \quad addType(\text{id.entry}, L.in)$
5 $L \rightarrow \text{id}$	$addType(\text{id.entry}, L.in)$

第 1 行的语义规则 $L.in = T.type$ 归约时对 $L.in$ 进行赋值，但在此之前，第 4 行和第 5 行中需要用到 $L.in$ 的值

1.3 L 属性定义的自上而下计算

- 可以使用**翻译方案**来描述语义规则的执行时机

1.3 L 属性定义的自上而下计算

- 可以使用**翻译方案**来描述语义规则的执行时机
 - 🗨 翻译方案的语义动作放在括号 `{ }` 内

1.3 L 属性定义的自上而下计算

- 可以使用**翻译方案**来描述语义规则的执行时机
 - ☞ 翻译方案的语义动作放在括号 $\{\}$ 内
 - ☞ 可以插入到产生式右部的任何位置

1.3 L 属性定义的自上而下计算

- 可以使用**翻译方案**来描述语义规则的执行时机
 - 翻译方案的语义动作放在括号 $\{\}$ 内
 - 可以插入到产生式右部的任何位置

如果

$$A \rightarrow \alpha\{C\}\beta,$$

则语义动作 C 的执行

- ① 在 α 的推导 (或向 α 的归约) 结束之后,
- ② 在 β 的推导 (或向 β 的归约) 开始之前

1.3 L 属性定义的自上而下计算

翻译方案:

$$E \rightarrow TR$$

$$R \rightarrow \text{addop } T \{ \text{print}(\text{addop.lexeme}); \} R_1 \mid \varepsilon$$

$$T \rightarrow \text{num } \{ \text{print}(\text{num.val}); \}$$

把加和减算符的中缀表达式翻译成后缀表达式的

如果输入是 $8 + 5 - 2$, 该翻译方案的输出是 $8\ 5\ +\ 2\ -$

1.3 L 属性定义的自上而下计算

- 只有综合属性时，为每条语义规则建议一个赋值动作，把动作放在产生式最右端即可得到翻译方案

1.3 L 属性定义的自上而下计算

- 只有综合属性时，为每条语义规则建议一个赋值动作，把动作放在产生式最右端即可得到翻译方案
- 如果含有继承属性，则

1.3 L 属性定义的自上而下计算

- 只有综合属性时，为每条语义规则建议一个赋值动作，把动作放在产生式最右端即可得到翻译方案
- 如果含有继承属性，则
 - 产生式右部符号的继承属性必须在先于这个符号的动作中计算(插在该符号的左端位置)

1.3 L 属性定义的自上而下计算

- 只有综合属性时，为每条语义规则建议一个赋值动作，把动作放在产生式最右端即可得到翻译方案
- 如果含有继承属性，则
 - ☞ 产生式右部符号的继承属性必须在先于这个符号的动作中计算(插在该符号的左端位置)
 - ☞ 一个动作不能引用该动作右边符号的综合属性

1.3 L 属性定义的自上而下计算

- 只有综合属性时，为每条语义规则建议一个赋值动作，把动作放在产生式最右端即可得到翻译方案
- 如果含有继承属性，则
 - ☞ 产生式右部符号的继承属性必须在先于这个符号的动作中计算(插在该符号的左端位置)
 - ☞ 一个动作不能引用该动作右边符号的综合属性
 - ☞ 左部非终结符号的综合属性只能在它所引用的所有属性都计算完成后才能计算，放在产生式最右端

1.3 L 属性定义的自上而下计算

L 属性定义

产生式	语义规则
1 $S \rightarrow B$	$B.ps = 10$ $S.ht = B.ht$
2 $B \rightarrow B_1 B_2$	$B_1.ps = B.ps$ $B_2.ps = B.ps$ $B.ht = \max(B_1.ht, B_2.ht)$
3 $B \rightarrow B_1 \text{ sub } B_2$	$B_1.ps = B.ps$ $B_2.ps = \text{shrink}(B.ps)$ $B.ht = \text{disp}(B_1.ht, B_2.ht)$
4 $B \rightarrow \text{text}$	$B.ht = \text{text.h} \times B.ps$

ps 是 B 的继承属性, 且仅依赖产生式头的继承属性

1.3 L 属性定义的自上而下计算

产生式

$$1. S \rightarrow B$$

语义规则

$$B.ps = 10$$

$$S.ht = B.ht$$

翻译方案

$$S \rightarrow \{ B.ps = 10; \}$$

B

$$\{ S.ht = B.ht \}$$

1.3 L 属性定义的自上而下计算

产生式

$$2. B \rightarrow B_1 B_2$$

语义规则

$$B_1.ps = B.ps$$

$$B_2.ps = B.ps$$

$$B.ht = \max(B_1.ht, B_2.ht)$$

翻译方案

$$B \rightarrow \{ B_1.ps = B.ps; \}$$

$$B_1$$

$$\{ B_2.ps = B.ps; \}$$

$$B_2$$

$$\{ B.ht = \max(B_1.ht, B_2.ht); \}$$

1.3 L 属性定义的自上而下计算

产生式

3. $B \rightarrow B_1 \text{ sub } B_2$

语义规则

$$B_1.ps = B.ps$$
$$B_2.ps = shrink(B.ps)$$
$$B.ht = disp(B_1.ht, B_2.ht)$$

翻译方案

$$B \rightarrow \{ B_1.ps = B.ps; \}$$
$$B_1 \text{ sub}$$
$$\{ B_2.ps = shrink(B.ps); \}$$
$$B_2$$
$$\{ B.ht = disp(B_1.ht, B_2.ht); \}$$

1.3 L 属性定义的自上而下计算

产生式

4. $B \rightarrow \text{text}$

语义规则

$B.ht = \text{text}.h \times B.ps$

翻译方案

$B \rightarrow \text{text}$

$\{ B.ht = \text{text}.h \times B.ps; \}$

1.3 L 属性定义的自上而下计算

文法中存在左递归的 SDD

产生式	语义规则
1 $E \rightarrow E_1 + T$	$E.nptr = mkNode('+', E_1.nptr, T.nptr)$
2 $E \rightarrow T$	$E.nptr = T.nptr$
3 $T \rightarrow T_1 * F$	$T.nptr = mkNode('*', T_1.nptr, F.nptr)$
4 $T \rightarrow F$	$T.nptr = F.nptr$
5 $F \rightarrow (E)$	$F.nptr = E.nptr$
6 $F \rightarrow \mathbf{id}$	$F.nptr = mkLeaf(\mathbf{id}, \mathbf{id}.entry)$
7 $F \rightarrow \mathbf{num}$	$F.nptr = mkLeaf(\mathbf{num}, \mathbf{num}.val)$

1.3 L 属性定义的自上而下计算

存在左递归的翻译方案

产生式	语义动作
1 $E \rightarrow E_1 + T$	$\{E.nptr = mkNode('+', E_1.nptr, T.nptr); \}$
2 $E \rightarrow T$	$\{E.nptr = T.nptr; \}$
3 $T \rightarrow T_1 * F$	$\{T.nptr = mkNode('*', T_1.nptr, F.nptr); \}$
4 $T \rightarrow F$	$\{T.nptr = F.nptr; \}$
5 $F \rightarrow (E)$	$\{F.nptr = E.nptr; \}$
6 $F \rightarrow \text{id}$	$\{F.nptr = mkLeaf(\text{id}, \text{id.entry}); \}$
7 $F \rightarrow \text{num}$	$\{F.nptr = mkLeaf(\text{num}, \text{num.val}); \}$

需要消除左递归

1.3 L 属性定义的自上而下计算

- 考虑一般性的左递归问题：

$$A \rightarrow A_1 Y \quad \{A.a = g(A_1.a, Y.y);\}$$

$$A \rightarrow X \quad \{A.a = f(X.x);\}$$

```
1 Val A()  
2 {  
3     A1.a = A();  
4     Y.y = Y();  
5     A.a = g(A1.a, Y.y);  
6  
7     return A.a;  
8 }
```

```
1 Val A()  
2 {  
3     X.x = X();  
4     A.a = f(X.x);  
5  
6     return A.a;  
7 }
```

1.3 L 属性定义的自上而下计算

消除左递归之后:

$$A \rightarrow XR$$

$$R \rightarrow YR \mid \varepsilon$$

```
1 Val A()  
2 {  
3   X.x = X();  
4   /* A.a = f(X.x) */  
5   R.in = f(X.x);  
6   R.out = R(R.in);  
7   A.a = R.out;  
8   return A.a;  
9 }
```

```
1 Val R(R.in)  
2 {  
3   R.out = R.in;  
4   return R.out;  
5 }
```

```
1 Val R(R.in)  
2 {  
3   Y.y = Y();  
4   /* A.a = g(A1.a, Y.y); */  
5   R1.in = g(R.in, Y.y);  
6   R1.out = R(R1.in);  
7   R.out = R1.out;  
8   return R.out;  
9 }
```

1.3 L 属性定义的自上而下计算

消除左递归之后

$$\begin{aligned} A &\rightarrow X \quad \{R.in = f(X.x); \} & R &\quad \{A.a = R.out; \} \\ R &\rightarrow Y \quad \{R_1.in = g(R.in, Y.y); \} & R_1 &\quad \{R.out = R_1.out; \} \\ R &\rightarrow \varepsilon \quad \{R.out = R.in; \} \end{aligned}$$

消除左递归之前

$$\begin{aligned} A &\rightarrow A_1 Y \quad \{A.a = g(A_1.a, Y.y); \} \\ A &\rightarrow X \quad \{A.a = f(X.x); \} \end{aligned}$$

1.3 L 属性定义的自上而下计算

使用 i 代替 in , s 代替 out , 换名之后

$$\begin{aligned}A &\rightarrow X \quad \{R.i = f(X.x); \} & R &\quad \{A.a = R.s; \} \\R &\rightarrow Y \quad \{R_1.i = g(R.i, Y.y); \} & R_1 &\quad \{R.s = R_1.s; \} \\R &\rightarrow \varepsilon \quad \{R.s = R.i; \}\end{aligned}$$

消除左递归之前

$$\begin{aligned}A &\rightarrow A_1 Y \quad \{A.a = g(A_1.a, Y.y); \} \\A &\rightarrow X \quad \{A.a = f(X.x); \}\end{aligned}$$

1.3 L 属性定义的自上而下计算

- 语法制导的预测翻译器构造算法

- ☞ 输入：语法制导的翻译方案

- ☞ 输出：语法制导翻译器的代码

- ☞ 方法：修改预测分析器的构造技术

为每个非终结符 A 构造一下函数

- ① A 的每个继承属性声明为该函数的一个形式参数
- ② A 的综合属性作为函数返回值
- ③ 在函数中为 A 产生式中的其他每个文法符号的每个属性声明一个局部变量
- ④ A 函数的代码框架和预测分析过程一致

1.3 L 属性定义的自上而下计算

- 语法制导的预测翻译器构造算法

- 👉 输入：语法制导的翻译方案

- 👉 输出：语法制导翻译器的代码

- 👉 方法：修改预测分析器的构造技术

与产生式相关联的代码按如下方法生成：

- ① 对于有属性 x 的终结符号 X ，把 x 的值保存在为 $X.x$ 声明的变量中，然后调用匹配过程
- ② 对于非终结符号 B ，产生赋值 $c = B(b_1, \dots, b_k)$ ， b_i 是对应 B 继承属性的变量， c 是代表 B 综合属性的变量
- ③ 对于每个语义动作，把代码复制到函数体中，把对属性的引用改成相应的局部变量的引用

1.3 L 属性定义的自上而下计算

基于如下翻译方案构造预测翻译器

$$E \rightarrow T \{ R.i = T.nptr; \} \quad R \{ E.nptr = R.s; \}$$
$$R \rightarrow +T \{ R_1.i = mkNode('+', R.i, T.nptr); \} \quad R_1 \{ R.s = R_1.s; \}$$
$$R \rightarrow \varepsilon \{ R.s = R.i; \}$$
$$T \rightarrow F \{ W.i = F.nptr; \} \quad W \{ T.nptr = W.s; \}$$
$$W \rightarrow *F \{ W_1.i = mkNode('*', W.i, F.mptr); \} \quad W_1 \{ W.s = W_1.s; \}$$
$$W \rightarrow \varepsilon \{ W.s = W.i; \}$$
$$F \rightarrow (E) \{ F.nptr = E.nptr; \}$$
$$F \rightarrow \mathbf{id} \{ F.nptr = mkLeaf(\mathbf{id}, \mathbf{id}.entry); \}$$
$$F \rightarrow \mathbf{num} \{ F.nptr = mkLeaf(\mathbf{num}, \mathbf{num}.val); \}$$

1.3 L 属性定义的自上而下计算

为每个非终结符构造函数

$\text{syntaxTreeNode} * E()$

$\text{syntaxTreeNode} * R(\text{syntaxTreeNode} * i)$

$\text{syntaxTreeNode} * T()$

$\text{syntaxTreeNode} * W(\text{syntaxTreeNode} * i)$

$\text{syntaxTreeNode} * F()$

1.3 L 属性定义的自上而下计算

```
1  syntaxTreeNode* R (syntaxTreeNode* i) {
2      syntaxTreeNode *nptr, *i1, *s1, *s;
3      char addoplexeme;
4      if (lookahead == '+') {
5          addoplexeme = lexval;
6          match('+');
7          nptr = T();
8          i1 = mkNode(addoplexeme, i, nptr);
9          s1 = R(i1);
10         s = s1;
11     }
12     else s = i;
13     return s;
14 }
```

i 对应继承属性, s 对应综合属性

1.3 L 属性定义的自上而下计算

考虑如下文法：

$$D \rightarrow L : T$$

$$T \rightarrow \text{integer} \mid \text{char}$$

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

在第一个产生式中，非终结符 L 从它右边的 T 获得类型信息，因此基于此文法的 SDD 不是 L 属性定义的，需修改文法

1.3 L 属性定义的自上而下计算

修改后新的文法:

$$D \rightarrow \text{id } L$$

$$L \rightarrow , \text{id } L \mid : T$$

$$T \rightarrow \text{integer} \mid \text{char}$$

此时, 类型作为 L 的综合属性 $L.type$, 可以有翻译方案:

$$D \rightarrow \text{id } L \quad \{addType(\text{id}.entry, L.type);\}$$

$$L \rightarrow , \text{id } L_1 \quad \{L.type = L_1.type; addType(\text{id}.entry, L_1.type);\}$$

$$L \rightarrow : T \quad \{L.type = T.type;\}$$

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

$$T \rightarrow \text{char} \quad \{T.type = char;\}$$

此翻译方案仅含有综合属性

1.4 L 属性定义的自下而上计算

- 在自下而上分析框架中实现 L 属性定义的方法,
- 使用自下而上的方法来完成任何可以用自上而下方式完成的翻译过程

1.4 L 属性定义的自下而上计算

- 通过修改文法将语义动作变换到产生式的右端
 - 引入新的 (标记) 非终结符号 M 和产生式 $M \rightarrow \varepsilon$
 - 每个嵌入动作由不同的非终结符号 M 代表
 - 如果 M 在某个产生式

$$T \rightarrow \alpha \{c\} \beta$$

中替换了语义动作 c , 对 c 进行修改得到 c' , 并且将 c' 关联到 $M \rightarrow \varepsilon$ 上, 动作 c' :

- 将动作 c 需要的属性作为 M 的继承属性进行复制
- 按照 c 中的方法计算各属性, 但是将计算得到的这些属性作为 M 的综合属性

1.4 L 属性定义的自下而上计算

将如下含有嵌入动作的翻译方案变换成无嵌入动作的方案：

$$E \rightarrow TR$$

$$R \rightarrow +T \{print(' + '); \} R_1 \mid -T \{print(' - '); \} R_1 \mid \varepsilon$$

$$T \rightarrow \mathbf{num} \{print(\mathbf{num.val}); \}$$

变换成：

$$E \rightarrow TR$$

$$R \rightarrow +TMR_1 \mid -TNR_1 \mid \varepsilon$$

$$T \rightarrow \mathbf{num} \{print(\mathbf{num.val}); \}$$

$$M \rightarrow \varepsilon \{print(' + '); \}$$

$$N \rightarrow \varepsilon \{print(' - '); \}$$

1.4 L 属性定义的自下而上计算

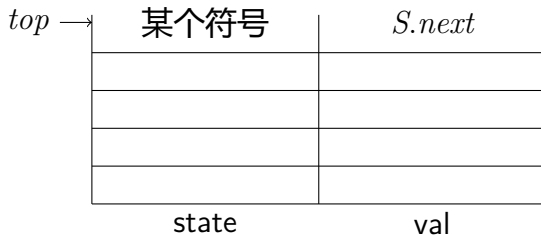
考虑如下翻译方案 (部分):

$$\begin{aligned} S \rightarrow & \text{while} \\ & (\\ & \quad \textcircled{1} \{ L_1 = \text{new}(); L_2 = \text{new}(); C.\text{false} = S.\text{next}; C.\text{true} = L_2; \} \\ & \quad C \\ &) \\ & \textcircled{2} \{ S_1.\text{next} = L_1; \} \\ & S_1 \\ & \{ S.\text{code} = \text{label} \parallel L_1 \parallel C.\text{code} \parallel \text{label} \parallel L_2 \parallel S_1.\text{code}; \} \end{aligned}$$

用 M 替换第①个语义动作, 用 N 替换第②个语义动作

1.4 L 属性定义的自下而上计算

在分析以 while 开始的串之前



1.4 L 属性定义的自下而上计算

将空串归约为 M 之后栈的情况

$top \rightarrow$	M	$C.true, C.false, L_1, L_2$
	(
	while	
	某个符号	$S.next$
	state	val

执行的代码:

$$L_1 = new(); L_2 = new(); C.true = L_2; C.false = stack[top-3].next$$

1.4 L 属性定义的自下而上计算

将空串归约为 N 之后栈的情况

$top \rightarrow$	N	$S_1.next$
)	
	C	$C.code$
	M	$C.true, C.false, L_1, L_2$
	(
	while	
	某个符号	$S.next$
	:	:
	state	val

执行的代码: $S_1.next = stack[top - 3].L_1$

内容提要

1 语法制导翻译

- 语法制导定义
- S 属性定义的自下而上计算
- L 属性定义的自上而下计算
- L 属性定义的自下而上计算

2 中间代码生成

- 中间语言
- 声明语句
- 赋值语句
- 布尔表达式和控制流语句

2.1 中间语言

- 抽象语法树和 C 语言都可以作为源程序的一种中间表示
- 中间表示-1: 后缀表示

后缀表示

表达式 E 的后缀表示可以定义如下:

👉 如果 E 是变量或者常数, 那么 E 的后缀表示就是其自身, x vs x , 12 vs 12

2.1 中间语言

- 抽象语法树和 C 语言都可以作为源程序的一种中间表示
- 中间表示-1: 后缀表示

后缀表示

表达式 E 的后缀表示可以定义如下:

- 如果 E 是变量或者常数, 那么 E 的后缀表示就是其自身, x vs x , 12 vs 12
- 如果 E 是形式为 $E_1 \text{ op } E_2$ 的表达式, 其中 op 是任意二元算符, 则 E 的后缀表示是 $E'_1 E'_2 \text{ op}$, 其中 E'_1 和 E'_2 分别是 E_1 和 E_2 的后缀表示, $1 + 2 - 3$ vs $1\ 2\ 3\ -\ +$

2.1 中间语言

- 抽象语法树和 C 语言都可以作为源程序的一种中间表示
- 中间表示-1: 后缀表示

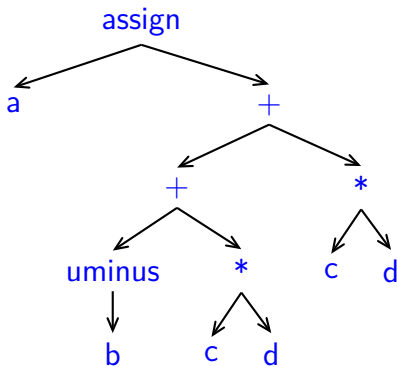
后缀表示

表达式 E 的后缀表示可以定义如下:

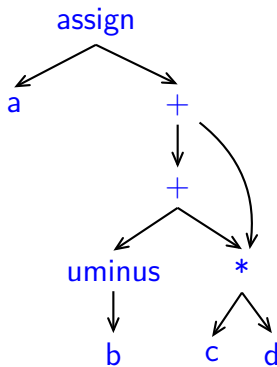
- 如果 E 是变量或者常数, 那么 E 的后缀表示就是其自身, x vs x , 12 vs 12
- 如果 E 是形式为 $E_1 \text{ op } E_2$ 的表达式, 其中 op 是任意二元算符, 则 E 的后缀表示是 $E'_1 E'_2 \text{ op}$, 其中 E'_1 和 E'_2 分别是 E_1 和 E_2 的后缀表示, $1 + 2 - 3$ vs $1 \ 2 \ 3 \ - \ +$
- 如果 E 是 (E_1) , 则 E 的后缀表示是 E'_1 的后缀表示 $(1 - 2) * 3$ vs $1 \ 2 \ - \ 3 \ *$

2.1 中间语言

- 中间表示-2: 有向无环图 (Directed Acyclic Graph, DAG)



(a) 抽象语法树



(b) 有向无环图

图: $a = (-b + c * d) + (c * d)$ 的图形表示

2.1 中间语言

- 中间表示-2: 有向无环图 (Directed Acyclic Graph, DAG)

产生式	语义规则
1 $S \rightarrow \mathbf{id} = E$	$S.nptr = mkNode('assign', mkLeaf(\mathbf{id}, \mathbf{id}.entry), E.nptr)$
2 $E \rightarrow E_1 + E_2$	$E.nptr = mkNode('+', E_1.nptr, E_2.nptr)$
3 $E \rightarrow E_1 * E_2$	$E.nptr = mkNode('*', E_1.nptr, E_2.nptr)$
4 $E \rightarrow -E_1$	$E.nptr = mkUNode('uminus', E_1.nptr)$
5 $E \rightarrow (E_1)$	$E.nptr = E_1.nptr$
6 $E \rightarrow \mathbf{id}$	$E.nptr = mkLeaf(\mathbf{id}, \mathbf{id}.entry)$

有向无环图也可以使用上述 SDD 进行构造
构造结点的函数检查是否有相同结点存在

2.1 中间语言

- 中间表示-3: 三地址代码, 每条指令/语句的一般形式为

$$x = y \text{ op } z$$

其中, x, y, z 是名字、常数或临时变量, op 代表算符

- 每条指令通常含有 3 个地址: 两个运算对象地址和一个结果地址

$$x + y * z$$

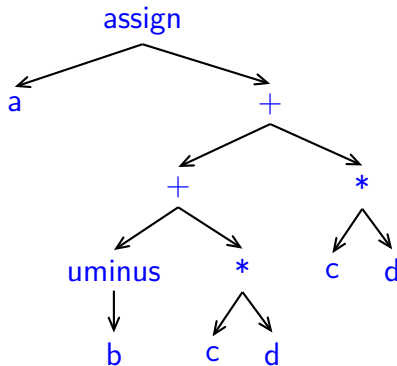
将被翻译成

$$t_1 = y * z$$

$$t_2 = x + t_1$$

其中 t_1 和 t_2 是临时名字

2.1 中间语言

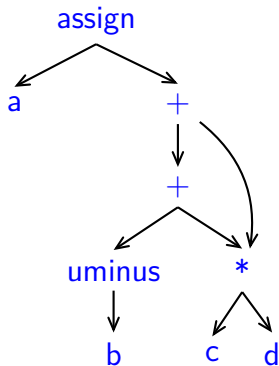


(a) 抽象语法树

$t_1 = -b$
 $t_2 = c * d$
 $t_3 = t_1 + t_2$
 $t_4 = c * d$
 $t_5 = t_3 + t_4$
 $a = t_5$

(b) 对应抽象语法树的代码

2.1 中间语言



(c) 有向无环图

$$\begin{aligned} t_1 &= -b \\ t_2 &= c * d \\ t_3 &= t_1 + t_2 \\ t_4 &= t_3 + t_2 \\ a &= t_4 \end{aligned}$$

(d) 对应有向无环图的代码

2.1 中间语言

常见三地址指令形式:

- 形如 $x = y \text{ } op \text{ } z$ 的赋值指令, 其中 op 是双目算术/逻辑运算符, x, y, z 是地址

2.1 中间语言

常见三地址指令形式:

- 形如 $x = y \text{ op } z$ 的赋值指令, 其中 op 是双目算术/逻辑运算符, x, y, z 是地址
- 形如 $x = op \ y$ 的赋值指令, 其中 op 是单目运算符: 单目减、逻辑非、转换运算

2.1 中间语言

常见三地址指令形式:

- 形如 $x = y \text{ op } z$ 的赋值指令, 其中 op 是双目算术/逻辑运算符, x, y, z 是地址
- 形如 $x = op \ y$ 的赋值指令, 其中 op 是单目运算符: 单目减、逻辑非、转换运算
- 形如 $x = y$ 的复制指令, 它把 y 的值赋给 x

2.1 中间语言

常见三地址指令形式:

- 形如 $x = y \text{ op } z$ 的赋值指令, 其中 op 是双目算术/逻辑运算符, x, y, z 是地址
- 形如 $x = op \ y$ 的赋值指令, 其中 op 是单目运算符: 单目减、逻辑非、转换运算
- 形如 $x = y$ 的复制指令, 它把 y 的值赋给 x
- 无条件转移指令 $goto \ L$, 下一步要执行的指令是带标号 L 的三地址指令

2.1 中间语言

常见三地址指令形式:

- 形如 $x = y \text{ op } z$ 的赋值指令, 其中 op 是双目算术/逻辑运算符, x, y, z 是地址
- 形如 $x = op \ y$ 的赋值指令, 其中 op 是单目运算符: 单目减、逻辑非、转换运算
- 形如 $x = y$ 的复制指令, 它把 y 的值赋给 x
- 无条件转移指令 $goto \ L$, 下一步要执行的指令是带标号 L 的三地址指令
- 形如 $if \ x \ goto \ L$ 的条件转移指令, 如果 x 为真时, 下一步将执行带有标号 L 的指令; 否则下一步执行序列中的后一条指令

2.1 中间语言

常见三地址指令形式:

- 形如 `if False x goto L` 的条件转移指令, 如果 x 为假时, 下一步将执行带有标号 L 的指令; 否则下一步执行序列中的后一条指令

2.1 中间语言

常见三地址指令形式:

- 形如 `if False x goto L` 的条件转移指令, 如果 x 为假时, 下一步将执行带有标号 L 的指令; 否则下一步执行序列中的后一条指令
- 形如 `if x relop y goto L` 的条件转移指令, 对 x 和 y 应用一个关系运算符 (如: $<$, $==$, $>=$)

2.1 中间语言

常见三地址指令形式:

- 形如 `if False x goto L` 的条件转移指令, 如果 x 为假时, 下一步将执行带有标号 L 的指令; 否则下一步执行序列中的后一条指令
- 形如 `if x relop y goto L` 的条件转移指令, 对 x 和 y 应用一个关系运算符 (如: $<$, $==$, $>=$)
- 形如 `param x` 的指令进行参数传递

2.1 中间语言

常见三地址指令形式:

- 形如 `if False x goto L` 的条件转移指令, 如果 x 为假时, 下一步将执行带有标号 L 的指令; 否则下一步执行序列中的后一条指令
- 形如 `if x relop y goto L` 的条件转移指令, 对 x 和 y 应用一个关系运算符 (如: $<, ==, >=$)
- 形如 `param x` 的指令进行参数传递
- 形如 `call p, n` 的指令进行过程调用, n 表示实参个数

2.1 中间语言

常见三地址指令形式:

- 形如 `if False x goto L` 的条件转移指令, 如果 x 为假时, 下一步将执行带有标号 L 的指令; 否则下一步执行序列中的后一条指令
- 形如 `if x relop y goto L` 的条件转移指令, 对 x 和 y 应用一个关系运算符 (如: $<$, $==$, $>=$)
- 形如 `param x` 的指令进行参数传递
- 形如 `call p, n` 的指令进行过程调用, n 表示实参个数
- 形如 `y = call p, n` 的指令进行函数调用

2.1 中间语言

常见三地址指令形式:

- 形如 `if False x goto L` 的条件转移指令, 如果 x 为假时, 下一步将执行带有标号 L 的指令; 否则下一步执行序列中的后一条指令
- 形如 `if x relop y goto L` 的条件转移指令, 对 x 和 y 应用一个关系运算符 (如: $<$, $==$, $>=$)
- 形如 `param x` 的指令进行参数传递
- 形如 `call p, n` 的指令进行过程调用, n 表示实参个数
- 形如 `y = call p, n` 的指令进行函数调用
- 返回指令 `return y`, 其中 y 表示返回值

2.1 中间语言

常见三地址指令形式:

- $x = y[i]$, 将位置 y 之后的第 i 个存储单元中存放的值赋给 x

2.1 中间语言

常见三地址指令形式:

- $x = y[i]$, 将位置 y 之后的第 i 个存储单元中存放的值赋给 x
- $x[i] = y$, 将位置 y 中存放的值赋给 x 之后的第 i 个存储单元中

2.1 中间语言

常见三地址指令形式:

- $x = y[i]$, 将位置 y 之后的第 i 个存储单元中存放的值赋给 x
- $x[i] = y$, 将位置 y 中存放的值赋给 x 之后的第 i 个存储单元中
- $x = \&y$, 将 x 的右值设置为 y 的左值

变量: $[L, R] \wedge L \neq \varepsilon \wedge R \neq \varepsilon$

常量: $[L, R] \wedge L = \varepsilon$

Pre : $\{ x : [L_x, R_y], y : [L_y, R_y] \}$

$x = \&y$

Post : $\{ x : [L_x, L_y], y : [L_y, R_y] \}$

2.1 中间语言

常见三地址指令形式:

- $x = y[i]$, 将位置 y 之后的第 i 个存储单元中存放的值赋给 x
- $x[i] = y$, 将位置 y 中存放的值赋给 x 之后的第 i 个存储单元中
- $x = \&y$, 将 x 的右值设置为 y 的左值
- $x = *y$, 将 x 的右值设置为存储在位置 y 中的值

变量: $[L, R] \wedge L \neq \varepsilon \wedge R \neq \varepsilon$

常量: $[L, R] \wedge L = \varepsilon$

Pre : $\{ x : [L_x, R_y], y : [L_y, R_y], z : [L_z, R_z], L_z = R_y \}$

$x = *y$

Post : $\{ x : [L_x, R_z], y : [L_y, R_y], z : [L_z, R_z] \}$

2.1 中间语言

常见三地址指令形式:

- $x = y[i]$, 将位置 y 之后的第 i 个存储单元中存放的值赋给 x
- $x[i] = y$, 将位置 y 中存放的值赋给 x 之后的第 i 个存储单元中
- $x = \&y$, 将 x 的右值设置为 y 的左值
- $x = *y$, 将 x 的右值设置为存储在位置 y 中的值
- $*x = y$, 将 y 的右值放入 x 指向的存储单元中

变量: $[L, R] \wedge L \neq \varepsilon \wedge R \neq \varepsilon$ **常量:** $[L, R] \wedge L = \varepsilon$

Pre : $\{ x : [L_x, R_x], y : [L_y, R_y], z : [L_z, R_z], L_z = R_x \}$
 $*x = y$
Post : $\{ x : [L_x, R_x], y : [L_y, R_y], z : [L_z, R_y] \}$

2.1 中间语言

- 中间表示-4: 静态单赋值形式 (Static Single-Assignment Form, SSA)

所有赋值指令都是对不同变量的赋值

使用不同的下标区分变量

$$p = a + b$$

$$q = p - c$$

$$p = q * d$$

$$p = e - p$$

$$q = p + q$$

(a) 三地址代码

$$p_1 = a + b$$

$$q_1 = p_1 - c$$

$$p_2 = q_1 * d$$

$$p_3 = e - p_2$$

$$q_2 = p_3 + q_1$$

(b) 静态单赋值形式

2.1 中间语言

同一个变量可能在不同的控制流路径上被赋值：

```
1  if(flag)
2      x = -1;
3  else
4      x = 1;
5
6  y = x * a;
```

如果在条件语句的两个分支上使用不同的名字，
那么在赋值 $y = x * a$ 中应该为 x 使用哪个名字呢？

2.1 中间语言

使用 ϕ 函数

```
1  if(flag)
2       $x_1 = -1$ ;
3  else
4       $x_2 = 1$ ;
5
6   $x_3 = \phi(x_1, x_2)$ ;
7   $y = x_3 * a$ ;
```

如果控制流通过条件为真的部分,

$\phi(x_1, x_2)$ 返回 x_1 ; 否则返回 x_2

2.2 声明语句

- 过程中声明的翻译方案, P 是一个表示过程的文法符号:

$$P \rightarrow \{offset = 0\} D; S$$
$$D \rightarrow D; D$$
$$D \rightarrow \mathbf{id} : T \quad \{enter(\mathbf{id.lexeme}, T.type, offset);$$
$$offset = offset + T.width; \}$$
$$T \rightarrow \mathbf{integer} \quad \{T.type = integer; T.width = 4; \}$$
$$T \rightarrow \mathbf{real} \quad \{T.type = real; T.width = 8; \}$$
$$T \rightarrow \mathbf{array}[\mathbf{num}] \mathbf{of} T_1 \quad \{T.type = array(\mathbf{num.val}, T_1.type);$$
$$T.width = \mathbf{num.val} \times T_1.width; \}$$
$$T \rightarrow \uparrow T_1 \quad \{T.type = pointer(T_1.type); T.width = 4; \}$$

其中全局变量 $offset$ 表示相对地址; $enter$ 表示建立符号表条目; $width$ 表示字节数目

2.2 声明语句

- 过程嵌套的声明翻译方案, tS 为符号表栈; oS 为相对地址栈:

$$P \rightarrow M D; S \quad \{addWidth(top(tS), top(oS)); pop(tS); pop(oS); \}$$
$$M \rightarrow \varepsilon$$
$$D \rightarrow D_1; D_2$$
$$D \rightarrow \text{proc id}; N D_1; S$$
$$D \rightarrow \text{id} : T$$
$$N \rightarrow \varepsilon$$

$addWidth$: 把当前的相对位置存入符号表;

2.2 声明语句

- 过程嵌套的声明翻译方案, tS 为符号表栈; oS 为相对地址栈:

$$P \rightarrow M D; S \quad \{ addWidth(top(tS), top(oS)); pop(tS); pop(oS); \}$$
$$M \rightarrow \varepsilon \quad \{ t = mkTable(nil); push(t, tS); push(0, oS); \}$$
$$D \rightarrow D_1; D_2$$
$$D \rightarrow \text{proc id}; N D_1; S$$
$$D \rightarrow \text{id} : T$$
$$N \rightarrow \varepsilon$$

$mkTable$: 建立新的符号表, 新符号表中含有上一层的符号表的指针;

2.2 声明语句

- 过程嵌套的声明翻译方案, tS 为符号表栈; oS 为相对地址栈:

$$P \rightarrow M D; S \quad \{addWidth(top(tS), top(oS)); pop(tS); pop(oS); \}$$
$$M \rightarrow \varepsilon \quad \{t = mkTable(nil); push(t, tS); push(0, oS); \}$$
$$D \rightarrow D_1; D_2$$
$$D \rightarrow \mathbf{proc} \ \mathbf{id}; N \ D_1; S \quad \{t = top(tS); addWidth(t, top(oS)); pop(tS); pop(oS); enterProc(top(tS), \mathbf{id}.lexeme, t); \}$$
$$D \rightarrow \mathbf{id} : T$$
$$N \rightarrow \varepsilon$$

enterProc: 为子过程建立符号表条目;

2.2 声明语句

- 过程嵌套的声明翻译方案, tS 为符号表栈; oS 为相对地址栈:

$$P \rightarrow M D; S \quad \{addWidth(top(tS), top(oS)); pop(tS); pop(oS); \}$$
$$M \rightarrow \varepsilon \quad \{t = mkTable(nil); push(t, tS); push(0, oS); \}$$
$$D \rightarrow D_1; D_2$$
$$D \rightarrow \mathbf{proc} \ \mathbf{id}; N D_1; S \quad \{t = top(tS); addWidth(t, top(oS)); pop(tS); \\ pop(oS); enterProc(top(tS), \mathbf{id}.lexeme, t); \}$$
$$D \rightarrow \mathbf{id} : T \quad \{enter(top(tS), \mathbf{id}.lexeme, T.type, top(oS)); \\ top(oS) = top(oS) + T.width; \}$$
$$N \rightarrow \varepsilon$$

enter: 建立新的符号表条目;

2.2 声明语句

- 过程嵌套的声明翻译方案, tS 为符号表栈; oS 为相对地址栈:

$$P \rightarrow M D; S \quad \{ addWidth(top(tS), top(oS)); pop(tS); pop(oS); \}$$
$$M \rightarrow \varepsilon \quad \{ t = mkTable(nil); push(t, tS); push(0, oS); \}$$
$$D \rightarrow D_1; D_2$$
$$D \rightarrow \mathbf{proc} \ \mathbf{id}; N D_1; S \quad \{ t = top(tS); addWidth(t, top(oS)); pop(tS); \\ pop(oS); enterProc(top(tS), \mathbf{id}.lexeme, t); \}$$
$$D \rightarrow \mathbf{id} : T \quad \{ enter(top(tS), \mathbf{id}.lexeme, T.type, top(oS)); \\ top(oS) = top(oS) + T.width; \}$$
$$N \rightarrow \varepsilon \quad \{ t = mkTable(top(tS)); push(t, tS); push(0, oS); \}$$

2.2 声明语句

- 记录类型表达式



$$T \rightarrow \mathbf{record} \ L \ D \ \mathbf{end} \quad \{ T.type = record(top(tS)); \\ T.width = top(oS); pop(tS); pop(oS); \}$$
$$L \rightarrow \varepsilon \quad \{ t = mkTable(nil); push(t, tS); push(0, oS); \}$$

record: 记录类型构造器

2.3 赋值语句

- 在处理赋值语句中的名字时，需要在符号表中查找它的定义，获取它的属性，
- 然后在三地址代码中使用它在符号表中位置的指针

$$\begin{aligned} S \rightarrow \mathbf{id} := E \quad & \{ p = \text{lookup}(\mathbf{id.lexeme}); \\ & \mathbf{if}(p \neq \text{nil}) \text{ emit}(p, ' = ', E.place); \mathbf{else error}; \} \\ E \rightarrow E_1 + E_2 \quad & \{ E.place = \text{newTemp}(); \\ & \text{emit}(E.place, ' = ', E_1.place, ' + ', E_2.place); \} \end{aligned}$$

-  E 的属性 $place$ 用来记住符号表条目的地址，函数 newTemp 新建一个临时变量的名字，把名字存入符号表，并返回该条目的地址；
-  过程 emit 将其参数写到输出文件中，其参数构成一条三地址指令

2.3 赋值语句

- 在处理赋值语句中的名字时，需要在符号表中查找它的定义，获取它的属性，
- 然后在三地址代码中使用它在符号表中位置的指针

$$S \rightarrow \text{id} := E \quad \{ p = \text{lookup}(\text{id.lexeme}); \\ \text{if}(p \neq \text{nil}) \text{emit}(p, '=', E.place); \text{else error}; \}$$
$$E \rightarrow E_1 + E_2 \quad \{ E.place = \text{newTemp}(); \\ \text{emit}(E.place, '=', E_1.place, '+', E_2.place); \}$$
$$E \rightarrow -E_1 \quad \{ E.place = \text{newTemp}(); \\ \text{emit}(E.place, '=', 'uminus', E_1.place); \}$$
$$E \rightarrow (E_1) \quad \{ E.place = E_1.place; \}$$
$$E \rightarrow \text{id} \quad \{ p = \text{lookup}(\text{id.lexeme}); \\ \text{if}(p \neq \text{nil}) E.place = p; \text{else error}; \}$$

2.3 赋值语句

- **一维数组**的元素一般是顺序存放，如果每个数组元素的宽度是 w ，那么一维数组 A 的第 i 个元素从地址

$$base + (i - low) \times w \quad (1)$$

开始，其中 low 是下标的下界， $base$ 是分配给该数组的地址，即 $base$ 是 $A[low]$ 的地址，表达式 (1) 可以重写为：

$$i \times w + (base - low \times w)$$

则可以在编译时完成表达式后一部分的计算，从而减少运行时的计算

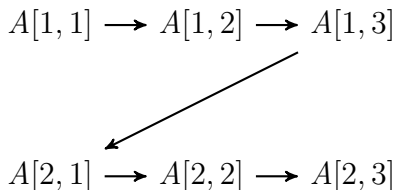
2.3 赋值语句

- **二维数组:**

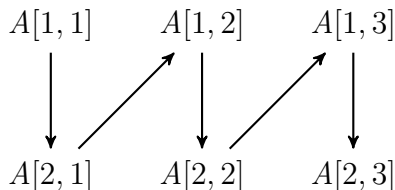
- ☞ 行为主，一行接一行
- ☞ 列为主，一列接一列

- 对于一个 2×3 的数组 A

以行为主



以列为主



2.3 赋值语句

- 以行为主的二维数组, $A[i, j]$ 的地址可以由公式:

$$base + ((i - low_1) \times n_2 + (j - low_2)) \times w$$

计算, 其中 low_1 和 low_2 分别是这两维的下界, n_2 是第二维的大小; 如果 $high_2$ 是 j 的上界, 则 $n_2 = high_2 - low_2 + 1$

- 若 i, j 都是编译时不能确定的值, 则公式可以重写为

$$((i \times n_2) + j) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$$

该表达式的后一项可以在编译时计算

2.3 赋值语句

- k 维数组引用 $A[i_1, i_2, \dots, i_k]$ 的三地址代码需要对表达式

$$(\dots((i_1 \times n_2 + i_2) \times n_3 + i_3) \dots) \times n_k + i_k$$

进行计算，使用递推过程：

$$e_1 = i_1$$

$$e_m = e_{m-1} \times n_m + i_m$$

来完成，直到 $m = k$ 为止

除了第一个下标表达式 i_1 外，对其他下标表达式，需要产生乘和加两条三地址指令

2.3 赋值语句

含有数组表达式的赋值文法，如果 L 是简单名字，则产生正常的赋值；否则产生由 L 的两个属性确定的存储单元的索引赋值：

$$S \rightarrow L := E \quad \{ \text{if}(L.offset == \mathbf{null}) \text{ emit}(L.place, '=', E.place); \\ \text{else emit}(L.place, '[', L.offset, ']', '=', E.place); \}$$

$L.offset$ 为空时，表示简单名字

2.3 赋值语句

数学表达式的代码和之前的一样：

$$\begin{aligned} E \rightarrow E_1 + E_2 \quad & \{ E.place = newTemp(); \\ & emit(E.place, '=', E_1.place, '+', E_2.place); \} \\ E \rightarrow (E_1) \quad & \{ E.place = E_1.place; \} \end{aligned}$$

2.3 赋值语句

如果 E 产生数组元素 L , 可用索引得到存储单元 $L.place[L.offset]$ 的内容:

```
 $E \rightarrow L$   {if( $L, offset = \text{null}$ )  $E.place = L.place$ ;  
             else begin  
                  $E.place = newTemp()$ ;  
                  $emit(E.place, '=', L.place, '[', L.offset, '']$ );  
             end}
```

2.3 赋值语句

L 产生数组表达式时, $L.place$ 为新的临时变量赋值为不变部分已经计算好的值; $L.offset$ 对应需要动态计算的部分:

$$\begin{aligned} L \rightarrow Elist] \quad & \{ L.place = newTemp(); \\ & emit(L.place, ' = ', invariant(Elist.array)); \\ & L.offset = newTemp(); \\ & emit(L.offset, ' = ', Elist.place, ' * ', width(Elist.array)); \} \end{aligned}$$

- $Elist$ 的综合属性 $array$ 用于传递符号表中数组名条目的指针
- 函数 $invariant(array)$ 从 $array$ 所指向的符号表中的条目读取静态可计算的值
- 函数 $width(array)$ 从 $array$ 所指向的符号表中的条目读取单个数组元素的字节数量
- $Elist.place$ 用来保存根据 $Elist$ 的下标表达式计算的值

2.3 赋值语句

L 产生简单名字时:

$$L \rightarrow \mathbf{id} \quad \{ L.place = id.place; L.offset = \mathbf{null}; \}$$

$offset$ 为空表示简单名字

2.3 赋值语句

多于一个下标表达式的情况:

$$\begin{aligned} Elist \rightarrow Elist_1, E \quad & \{ t = newTemp(); m = Elist_1.ndim + 1; \\ & emit(t, '=', Elist_1.place, '*', limit(Elist_1.array, m)); \\ & emit(t, '=', t, '+', E.place); \\ & Elist.array = Elist_1.array \\ & Elist.place = t; \\ & Elist.ndim = m; \} \end{aligned}$$

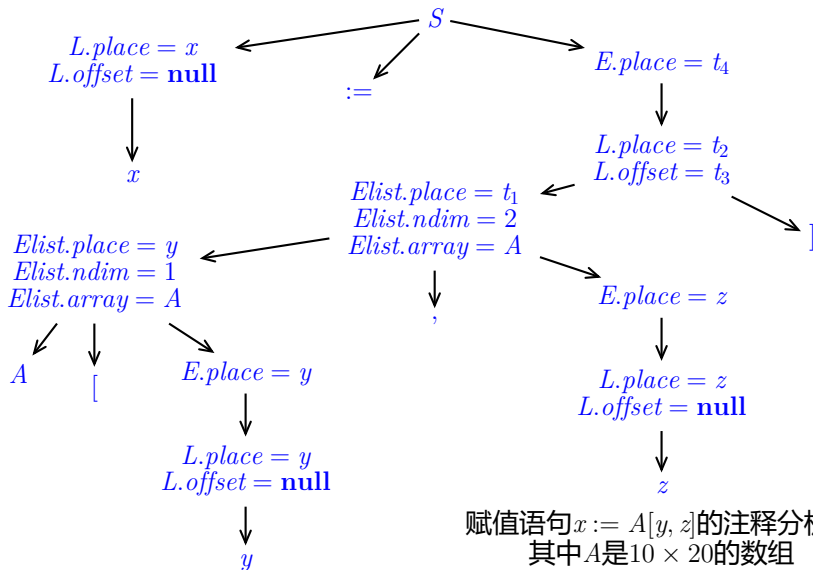
- $Elist.ndim$ 记录已经分析过的下标表达式的个数
- $limit(array, j)$ 返回 $array$ 所指向的数组中的第 j 维的大小
- $Elist_1.place$ 对应递推公式中的 e_{m-1}
- $Elist.place$ 对应递推公式中的 e_m

2.3 赋值语句

最后，对应递推公式中 $m = 1$ 的情况

$$\begin{aligned} Elist \rightarrow \mathbf{id}[E] \quad & \{ Elist.place = E.place; \\ & Elist.ndim = 1; \\ & Elist.array = \mathbf{id}.place; \} \end{aligned}$$

2.3 赋值语句



2.3 赋值语句

赋值语句 $x := A[y, z]$ 的三地址指令序列如下:

$$t_1 = y * 20$$

$$t_1 = t_1 + z$$

$$t_2 = c \quad /* \text{预先计算好的数组静态地址} */$$

$$t_3 = t_1 * 4 \quad /* \text{每个元素有 4 个字节} */$$

$$t_4 = t_2[t_3]$$

$$x = t_4$$

$$w = 4, n_1 = 10, n_2 = 20$$

$$((i \times n_2) + j) \times w + (base - ((low_1 \times n_2) + low_2) \times w)$$

2.3 赋值语句

考虑存在类型转换的情况

$$x = y + i * j$$

x, y : real, i, j : integer

$$t_1 = i \text{ int} * j$$

$$t_2 = \text{inttoreal } t_1$$

$$t_3 = y \text{ real} + t_2$$

$$x = t_3$$

```
1 E → E1 + E2
2 {
3   E.place = newTemp();
4   if (E1.type==integer && E2.type==integer) begin
5     emit(E.place, '=', E1.place, 'int+', E2.place);
6     E.type = integer;
7   end else if (E1.type==integer && E2.type==real) begin
8     u = newTemp();
9     emit(u, '=', 'inttoreal', E1.place);
10    emit(E.place, '=', u, 'real+', E2.place);
11    E.type = real;
12  end else . . .
13 }
```

2.4 布尔表达式和控制流语句

- 布尔表达式文法:

$$B \rightarrow B \text{ or } B \mid B \text{ and } B \mid \text{not } B \mid (B) \mid E \text{ relop } E \mid \text{True} \mid \text{False}$$

其中, **or** 和 **and** 是左结合的, **or** 的优先级最低, 然后是 **and**, 最后是 **not**

2.4 布尔表达式和控制流语句

- 短路计算, 完成部分计算就可以知道真假, 把 B_1 **or** B_2 定义成

if B_1 then **True** else B_2

把 B_1 **and** B_2 定义成

if B_1 then B_2 else **False**

如果 B_i 有副作用 (如修改某个全局变量), 则完全计算和短路计算的结果可能有差异

2.4 布尔表达式和控制流语句

- 短路计算, 完成部分计算就可以知道真假, 把 B_1 **or** B_2 定义成

if B_1 then **True** else B_2

把 B_1 **and** B_2 定义成

if B_1 then B_2 else **False**

如果 B_i 有副作用 (如修改某个全局变量), 则完全计算和短路计算的结果可能有差异

- 翻译语句 `if($x < 100$ or $x > 200$ and $x \neq y$) $x = 0$;`

```
1      if x<100 goto L2
2      if False x>200 goto L1
3      if False x!=y goto L1
4 L2:  x=0
5 L1:
```

2.4 布尔表达式和控制流语句

考虑控制流语句的文法：

$$\begin{aligned} S \rightarrow & \text{if } B \text{ then } S_1 \\ & | \text{if } B \text{ then } S_1 \text{ else } S_2 \\ & | \text{while } B \text{ do } S_1 \\ & | S_1; S_2 \end{aligned}$$

其中 B 是布尔表达式.

2.4 布尔表达式和控制流语句

- 继承属性 $B.true/B.false$ 是 B 为真/假时, 控制流应该转向的标号
- SDD 中的符号 \parallel 是串连接运算符
- 函数 $newLabel$ 用于产生新标号

控制流语句的语法制导定义

产生式	语义规则
$S \rightarrow \text{if } B \text{ then } S_1$	$B.true = newLabel()$
	$B.false = S.next$
	$S_1.next = S.next$
	$S.code = B.code$
	$\parallel gen(B.true, ':')$ $\parallel S_1.code$

2.4 布尔表达式和控制流语句

- 标号 $S.next$ 是继承属性

控制流语句的语法制导定义

产生式	语义规则
	$B.true = newLabel()$
	$B.false = newLabel()$
$S \rightarrow \text{if } B \text{ then}$	$S1.next = S.next;$
$\quad S_1$	$S2.next = S.next;$
else	$S.code = B.code$
$\quad S_2$	$\parallel gen(B.true, ':') \parallel S_1.code$
	$\parallel gen('goto', S.next)$
	$\parallel gen(B.false, ':') \parallel S_2.code$

2.4 布尔表达式和控制流语句

- *begin* 是局部变量，存放 while 语句的第一条指令的标号

控制流语句的语法制导定义

产生式	语义规则
	$begin = newLabel()$
	$B.true = newLabel()$
	$B.false = S.next$
$S \rightarrow \text{while } B \text{ do } S_1$	$S_1.next = begin$
	$S.code = gen(begin, ':') \parallel B.code$
	$\parallel gen(B.true, ':') \parallel S_1.code$
	$\parallel gen('goto', begin)$

2.4 布尔表达式和控制流语句

- 最后，串行组合语句

控制流语句的语法制导定义

产生式	语义规则
$S \rightarrow S_1; S_2$	$S_1.next = newLabel()$ $S_2.next = S.next()$ $S.code = S_1.code \parallel gen(S_1.next, ':') \parallel S_2.code$

2.4 布尔表达式和控制流语句

布尔表达式的语法制导定义

$$B \rightarrow \text{True}$$
$$B.code = gen('goto', B.true)$$
$$B \rightarrow \text{False}$$
$$B.code = gen('goto', B.false)$$
$$B \rightarrow E_1 \text{ relop } E_2$$
$$B.code = E_1.code \parallel E_2.code$$
$$\parallel gen('if', E_1.place, \text{relop.op}, E_2.place, 'goto', B.true)$$
$$\parallel gen('goto', B.false)$$

2.4 布尔表达式和控制流语句

布尔表达式的语法制导定义

$$B \rightarrow (B_1)$$

$$B_1.true = B.true$$

$$B_1.false = B.false$$

$$B.code = B_1.code$$

$$B \rightarrow \text{not } B_1$$

$$B_1.true = B.false$$

$$B_1.false = B.true$$

$$B.code = B_1.code$$

2.4 布尔表达式和控制流语句

布尔表达式的语法制导定义

$$B \rightarrow B_1 \text{ and } B_2$$
$$B_1.true = newLabel()$$
$$B_1.false = B.false$$
$$B_2.true = B.true$$
$$B_2.false = B.false$$
$$B.code = B_1.code$$
$$\parallel gen(B_1.true, ':')$$
$$\parallel B_2.code$$
$$B \rightarrow B_1 \text{ or } B_2$$
$$B_1.true = B.true$$
$$B_1.false = B.newLabel()$$
$$B_2.true = B.true$$
$$B_2.false = B.false$$
$$B.code = B_1.code$$
$$\parallel gen(B_1.false, ':')$$
$$\parallel B_2.code$$

2.4 布尔表达式和控制流语句

- 考虑布尔表达式 $a < b$ **or** $c < d$ **and** $e < f$, 则可以翻译得到如下代码

```
1      if a<b goto Ltrue
2      goto L1
3  L1:  if c<d goto L2
4      goto Lfalse
5  L2:  if e<f goto Ltrue
6      goto Lfalse
```

2.4 布尔表达式和控制流语句

考虑语句:

```
1 while a<b do
2     if c<d then
3         x:=y+z
4     else
5         x:=y-z
```

代码:

```
1 Lbegin:
2     B.code
3 LBtrue:
4     S1.code
5     goto Lbegin
```

其中:

- $B = a < b$
- $S_1 = \text{if } c < d \text{ then } x := y + z \text{ else } x := y - z$

2.4 布尔表达式和控制流语句

```
1 while a<b do
2     if c<d then
3         x:=y+z
4     else
5         x:=y-z
```

```
1 Lbegin:
2     if a<b goto LBtrue
3     goto LBfalse
4 LBtrue:
5     S1.code
6     goto Lbegin
7 LBfalse:
```

其中:

- L_B^{false} 是 while 语句的下一条语句的标号

2.4 布尔表达式和控制流语句

```
1 while a<b do
2     if c<d then
3         x:=y+z
4     else
5         x:=y-z
```

```
1 Lbegin:
2     if a<b goto LBtrue
3     goto LBfalse
4 LBtrue:
5     IfB.code
6 LIfBtrue:
7     SubS1.code
8     goto Lbegin
9 LIfBfalse:
10    SubS2.code
11    goto Lbegin
12 LBfalse:
```

- IfB 是 $c < d$; Sub_{S₁} 是 $x := y + z$; Sub_{S₂} 是 $x := y - z$

2.4 布尔表达式和控制流语句

```
1 while a<b do
2     if c<d then
3         x:=y+z
4     else
5         x:=y-z
```

```
1 Lbegin:
2     if a<b goto LBtrue
3     goto LBfalse
4 LBtrue:
5     if c<d goto LIfBtrue
6     goto LIfBfalse
7 LIfBtrue:
8     SubS1.code
9     goto Lbegin
10 LIfBfalse:
11     SubS2.code
12     goto Lbegin
13 LBfalse:
```

2.4 布尔表达式和控制流语句

```
1 while a<b do
2     if c<d then
3         x:=y+z
4     else
5         x:=y-z
```

```
1 Lbegin:
2     if a<b goto LBtrue
3     goto LBfalse
4 LBtrue:
5     if c<d goto LIfBtrue
6     goto LIfBfalse
7 LIfBtrue:
8     t1=y + z
9     x=t1
10    goto Lbegin
11 LIfBfalse:
12    SubS2.code
13    goto Lbegin
14 LBfalse:
```

2.4 布尔表达式和控制流语句

```
1 while a<b do
2     if c<d then
3         x:=y+z
4     else
5         x:=y-z
```

```
1 Lbegin:
2     if a<b goto LBtrue
3     goto LBfalse
4 LBtrue:
5     if c<d goto LIfBtrue
6     goto LIfBfalse
7 LIfBtrue:
8     t1=y + z
9     x=t1
10    goto Lbegin
11 LIfBfalse:
12    t2=y - z
13    x=t2
14    goto Lbegin
15 LBfalse:
```

其实 $L_{B \bullet false}$ = while 语句的下一条语句的标号

2.4 布尔表达式和控制流语句

- 考虑如下 switch-case 语句：

```
1  switch (E) {  
2      case  $V_1$ :  $S_1$   
3      case  $V_2$ :  $S_2$   
4          ...  
5      case  $V_{n-1}$ :  $S_{n-1}$   
6      default:  $S_n$   
7  }
```

- 计算表达式 E 的值
- 在 $n - 1$ 个常量 V_1, V_2, \dots, V_{n-1} 中寻找和 E 的值相同的常量
- 若有匹配的常量 V_i , 则执行 S_i ; 否则执行 S_n

2.4 布尔表达式和控制流语句

```
1      E的代码, 其中t用来存储E的值
2      if t!=V1 goto L1
3      code for S1
4      goto next
5 L1:   if t!=V2 goto L2
6      code for S2
7      goto next
8 L2:   ...
9      ...
10 Ln-2: if t!=Vn-1 goto Ln-1
11      code for Sn-1
12      goto next
13 Ln-1: code for Sn
14 next:
```

问题：在处理分支的时是不知道下一个分支的

2.4 布尔表达式和控制流语句

```
1      E的代码, 其中t用来存储E的值
2      goto test
3 L1:  code for S1
4      goto next
5 L2:  code for S2
6      goto next
7      ...
8 Ln-1: code for Sn-1
9      goto next
10 Ln:  code for Sn
11      goto next
12 test: if t=V1 goto L1
13      if t=V2 goto L2
14      ...
15      if t=Vn-1 goto Ln-1
16      goto Ln
17 next:
```

```
1 case t V1 L1
2 case t V2 L2
3 ...
4 case t Vn-1 Ln-1
5 case t t Ln
6 next:
```

case 三地址指令

2.4 布尔表达式和控制流语句

- 考虑如下包含过程调用语句的文法

$$\begin{aligned} S &\rightarrow \text{id}(Elist) \\ Elist &\rightarrow Elist, E \\ Elist &\rightarrow E \end{aligned}$$

```
1 param E1.place  
2 param E2.place  
3 ...  
4 param En.place  
5 call id.place, n
```

翻译方案 (使用队列 q 存储参数)

$$S \rightarrow \text{id}(Elist) \quad \{ \text{对 } q \text{ 中每个 } E.place, \\ \text{执行 } emit('param', E.place); \\ \text{然后 } emit('call', id.place, len(q)); \}$$
$$Elist \rightarrow Elist, E \quad \{ \text{把 } E.place \text{ 放入队列 } q \text{ 末尾}; \}$$
$$Elist \rightarrow E \quad \{ \text{初始化队列 } q, \text{ 把 } E.place \text{ 放入 } q \text{ 末尾}; \}$$

小结

- 语法制导定义的概念和形式, S 属性及 L 属性定义
- 语法制导翻译方案, 语义动作可以放在产生式体的任何位置
- 抽象语法树的例子
- 基于语法制导定义和翻译生成中间代码
- 三地址码比较重要