

# 编译原理笔记

上官凝

2021 年 3 月 12 日

# 目录

<b>1 词法分析</b>	<b>4</b>
1.1 词法记号及属性、描述和识别 . . . . .	4
1.1.1 词法记号和属性 . . . . .	4
1.1.2 串和语言 . . . . .	4
1.1.3 正则表达式 . . . . .	5
1.1.4 状态转换图 . . . . .	5
1.1.5 关于 FLEX 的正则表达式 . . . . .	6
1.2 有限自动机 . . . . .	6
1.2.1 NFA 和 DFA . . . . .	6
1.2.2 正则表达式到 NFA . . . . .	6
1.2.3 NFA 到 DFA . . . . .	7
1.2.4 DFA 的化简 . . . . .	7
<b>2 语法分析</b>	<b>8</b>
2.1 上下文无关文法 . . . . .	8
2.1.1 CFG 推导 . . . . .	8
2.1.2 文法的二义性 . . . . .	8
2.1.3 悬空 else 文法 . . . . .	9
2.1.4 左递归 . . . . .	9
2.2 自顶向下分析方法 . . . . .	9
2.2.1 递归下降分析方法 . . . . .	9
2.2.2 消除左递归 . . . . .	10
2.2.3 有左因子的文法 . . . . .	10
2.2.4 复杂的回溯代价太高 . . . . .	10
2.3 预测分析法 . . . . .	11
2.3.1 “第一个”集合和“跟随”集合 . . . . .	11
2.3.2 LL(1) 文法 . . . . .	11
2.4 递归下降的预测分析 . . . . .	12
2.4.1 预测分析表 . . . . .	12
2.5 自底向上分析方法 . . . . .	13

2.5.1 移进 (shift)-归约 (reduce) 分析技术 . . . . .	13
2.6 LR(k) 分析技术 . . . . .	14
2.7 SLR . . . . .	14
2.7.1 产生规则 . . . . .	14
2.7.2 有效项目 . . . . .	15
2.7.3 SLR 分析表 . . . . .	15
2.7.4 SLR(1) 文法 . . . . .	16
2.7.5 判定满足 SLR 文法输入串 . . . . .	16
2.8 规范的 LR 分析方法 . . . . .	16
2.8.1 构造 LR(1) 项目集规范族 . . . . .	16
2.8.2 构造规范的 LR 分析表 . . . . .	17
2.8.3 LALR . . . . .	17
<b>3 语法制导的翻译 . . . . .</b>	<b>18</b>
3.1 语法制导的定义 . . . . .	18
3.1.1 综合属性和继承属性 . . . . .	18
3.1.2 注释分析树和属性依赖图 . . . . .	19
3.1.3 具有受控副作用的语义规则 . . . . .	20
3.2 语法制导的翻译方案 . . . . .	20
3.3 S 属性定义的自下而上计算 . . . . .	21
3.3.1 语法树 . . . . .	21
3.3.2 S 属性的 SDD . . . . .	21
3.3.3 后缀 SDT 的语法分析栈实现 . . . . .	21
3.4 L 属性定义的自上而下计算 . . . . .	23
3.4.1 产生式内部带有语义动作的 SDT . . . . .	23
3.4.2 从 SDT 中消除左递归 . . . . .	23
3.5 L 属性定义的 SDT . . . . .	24
3.5.1 将 L 属性的 SDD 转换为 SDT . . . . .	25
3.5.2 在递归下降语法分析过程中进行翻译 . . . . .	25
3.5.3 边扫描边生成代码 . . . . .	26
3.5.4 L 属性的 SDD 和 LL 语法分析 . . . . .	26
3.5.5 L 属性的 SDD 的自底向上语法分析 . . . . .	26
<b>4 中间代码生成 . . . . .</b>	<b>28</b>
4.1 中间语言 . . . . .	28
4.1.1 三地址代码 . . . . .	28
4.2 . . . . .	28
4.2.1 符号表中的名字 . . . . .	28
4.2.2 表达式中的运算 . . . . .	29
4.2.3 数组元素的寻址 . . . . .	30

4.3 布尔表达式和控制流语句 . . . . .	30
4.3.1 布尔表达式 . . . . .	30
4.3.2 短路代码 . . . . .	31
4.4 控制流语句的翻译 . . . . .	31
4.4.1 if 语句中间代码生成的 SDD . . . . .	32
4.4.2 带有 else 的语句的中间代码生成 SDD . . . . .	33
4.4.3 while 语句的中间代码生成 SDD . . . . .	33
4.4.4 顺序结构中间代码生成 SDD . . . . .	33
4.5 布尔表达式的控制流翻译 . . . . .	34
4.5.1 或运算 . . . . .	34
4.5.2 且运算 . . . . .	34
4.5.3 非运算 . . . . .	34
4.5.4 关系运算 (relop) . . . . .	35
4.6 回填 . . . . .	35
<b>5 类型检查</b>	<b>37</b>
5.1 类型表达式 . . . . .	37
5.1.1 自动生成类型表达式 . . . . .	37
5.1.2 构造类型表达式的 SDT . . . . .	37
5.2 类型等价 . . . . .	38
5.3 类型检查 . . . . .	38
<b>6 与类型相关的中间代码生成</b>	<b>40</b>

# 第 1 章 词法分析

词法分析器的任务是把构成源程序的字符流翻译成词法记号流。其目的是将输入字符串识别为有意义的子串

1. 子串的种类 (Name)
2. 可帮助解释和理解该子串的属性 (Attribute)
3. 可描述具有相同特征的子串的模式 (Pattern)

## 第 1.1 节 词法记号及属性、描述和识别

### 1.1.1 词法记号和属性

词法记号由记号名和可选的属性值构成的二元组，经常使用记号名来引用记号。一个记号的模式描述属于该记号的词法单元的形式。词法单元是源程序中匹配一个记号模式的字符序列，它由词法分析器识别为该记号的一个实例。为什么需要属性？概括的说，记号名影响语法分析的决策，属性影响记号的翻译。

### 1.1.2 串和语言

语言表示字母表上的一个串集，属于该语言的串称为该语言的句子或字，注意  $\emptyset$  和  $\{\epsilon\}$  这样的抽象语言也符合这个定义。其运算参考字符串的运算（连接）和集合的运算（交并补 etc）

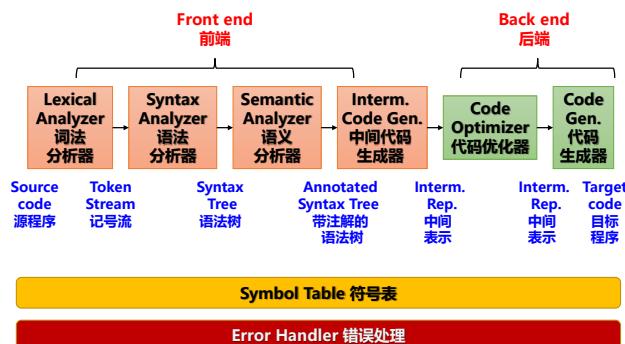


图 1.1: 编译器总览

### 1.1.3 正则表达式

约定：

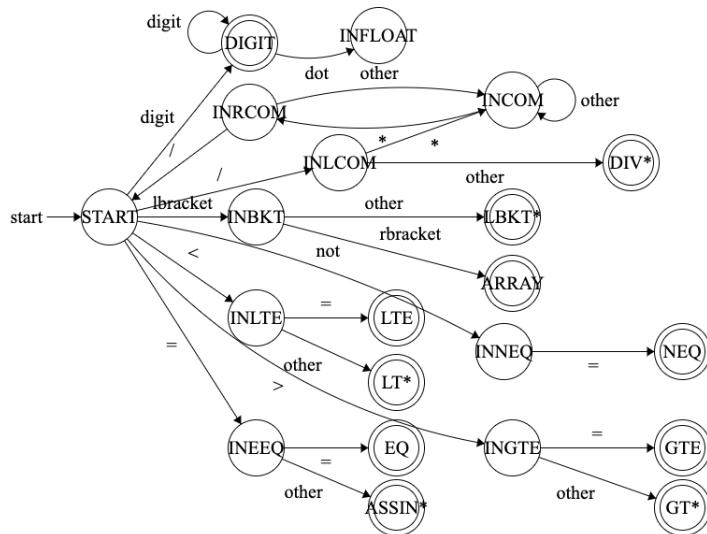
1. 闭包运算 ( $*$ ) 有最高的优先级并且是左结合的运算
2. 连接运算（两个正规式并列）的优先级次之且也是左结合的运算
3. 选择运算 ( $|$ ) 的优先级最低并且也是左结合的运算
4. 零个或一个实例 ( $r?$ ) 表示  $r | \epsilon$
5. 字符组 ( $[abc]$ ) 表示  $a | b | c$
6. 缩写字符组 ( $[a - z]$ ) 表示  $a | b | \dots | z$

可以对正规式命名，并用这些名字来引用相应的正规式。可以采用自底向上的方式来构建正规定义

### 1.1.4 状态转换图

(打个广告：在我的主页上面放了一个在线画 FSM 的网址连接)

大概长这样（这个是分析各种运算符以及注释的一个转换图，数字识别没有画完整）(IN 是指 intermediate，中间态)：



在词法分析时，会出现“移进归约冲突”（这里不是这么叫的，我只是借鉴一下这种说法）。就是识别到一个 token，它可能是一个关键字，也有可能是其他 token 的前缀。这时候应用最长匹配规则，即 lookahead，如果不符合词法规则（状态机无法跳转）就回退。

值得注意的是，词法分析器对程序采取非常局部的观点，即词法分析器只会管给定的程序（一个字符串）是不是能拆分成符合词法定义的 token 流。这些 token 在词法分析器看来是相互无关的。

### 1.1.5 关于 FLEX 的正则表达式

众所周知，实验会用到 FLEX，在写 .l 文件的时候要写正则表达式，大概规则如下：

1. . 匹配任何单个字符，除 \n.
2. - 表示匹配范围，如： a-z，表示匹配 a-z 之间的任何字符
3. \* 匹配前面表达式的零个或多个拷贝。
4. [] 匹配括号内的任意字符的字符类，第一个符号是 ^，表示匹配除括号中的字符以外的任意字符，如 [^/] 表示除斜杠以外的所有字符。
5. () 表示里面的模式被允许匹配多少次。
6. \ 用于转义字符
7. + 匹配前面表达式一次或多次出现。
8. ? 匹配前面表达式零次或 1 次出现。
9. | 匹配前面表达式或随后表达式
10. " " 引号中的每个字符解释为字面意思
11. {} 指示一个模式可能出现的次数，后面可以跟 \* 或者 +

以及，可以参考这个文章

## 第 1.2 节 有限自动机

### 1.2.1 NFA 和 DFA

NFA (不确定的有限自动机) 可以把同样的符号标记在出自同一个状态的多条边上，而 DFA (确定的有限自动机) 是 NFA 的特殊情况，其中：任何状态下都没有  $\varepsilon$  转换；且对任何状态 s 和任何输入符号 a，最多只能有一条表示为 a 的边离开 s

### 1.2.2 正则表达式到 NFA

基本模型有  $a | b$ 、 $ab$ 、 $s^*$  和  $(s)$ ，分别对应以下的图：

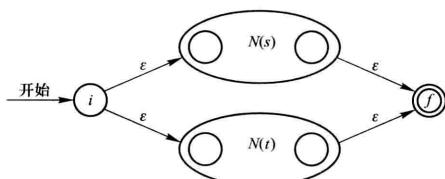


图 2.16 识别正规式  $s|t$  的 NFA

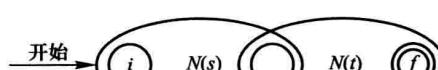
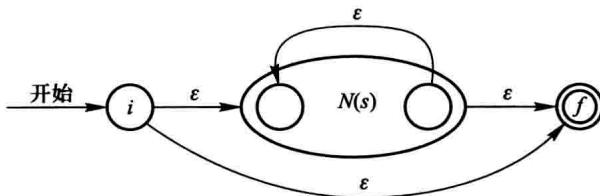


图 2.17 识别正规式  $st$  的 NFA

图 2.18 识别正规式  $s^*$  的 NFA

### 1.2.3 NFA 到 DFA

NFA 的状态是 DFA 的状态集合（闭包）。主要的有两步：

1. 递归计算出来包含状态 0 (start 状态) 的闭包（就是经过  $\epsilon$  能到达的所有状态）
2. 这个闭包在接收某个输入之后可能到达的新的状态闭包

然后对新的闭包迭代第二步，直到没有新的了

### 1.2.4 DFA 的化简

观察 DFA 的状态转换表，将对于任意输入符号，其跳转都一样的几个状态合并为同一状态，并选取一个代表。首先要**分开接受状态和非接受状态**

# 第 2 章 语法分析

输入：词法分析获得的记号序列；输出：程序的语法树。需要一种可以描述合法记号序列的语言、一种可以区分合法和非法的记号序列的方法。正则表达式不能用于描述配对或嵌套的结构，这是因为有限自动机不能记录重复访问同一状态的次数。

## 第 2.1 节 上下文无关文法

上下文无关文法 CFG 是由一个四元组  $(V_T, V_N, S, P)$  组成的：

1. 终结符：就是“句型”最终需要化成的形式，里面的元素都是确定的符号，也称为 *token*。在谈论编程语言的文法时，记号名时终结符的同义词。
2. 非终结符：可以认为是推导过程中的“中间变量”，或“形式参量”，用自下而上的角度来说，非终结符是终结符或/和非终结符的规约
3. 开始符号：是语法树的 *root*
4. 产生式的有限集合：生成规则，用  $\rightarrow$

### 2.1.1 CFG 推导

CFG 推导是“从文法推出文法所描述的语言中所包含的合法串集合的动作”，也就是说，从开始符号开始，利用生成规则进行迭代替换，得到句型。“上下文无关”是指在推导过程中，每一步  $\alpha A \beta \Rightarrow \alpha \gamma \beta$  仅依赖于  $A \rightarrow \gamma$ ，而与  $\alpha, \beta$  无关。对于语言、文法、句型、句子，语言是句子的集合，句子是实例。最左推导  $\Rightarrow_{lm}$  和最右推导（规范推导） $\Rightarrow_{rm}$ 。一步推导  $\Rightarrow$  和零步或多步推导  $\Rightarrow^*$  以及一步或多步推导  $\Rightarrow^+$ 。

### 2.1.2 文法的二义性

文法的某些句子存在不止一种最左（最右）推导，或者不止一棵分析树，则该文法是二义的。一般是由优先级和结合性的不确定导致的。所以可以通过定义优先级（将多个选项拆分为不同非终结符的嵌套推导规则来认为定义等级划分）来解决这个问题。这样，更接近于开始符号的非终结符就不能直接推导到终结符，更接近于终结符的不能从开始符号终结推导。左推导

优先级从高到低，右推导相反比如将第一个式子换成下面的几个：

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

$$E \rightarrow E + E \mid F$$

$$F \rightarrow F * F \mid G$$

$$G \rightarrow (G) \mid \text{id}$$

### 2.1.3 悬空 else 文法

悬空 else 文法是一个经典的例子：



2020/9/24

Cheng @ Compiler Fall 2020, USTC

56



2020/9/24

Cheng @ Compiler Fall 2020, USTC

56

图 2.1: 悬空 else 文法及其二义性

图 2.2: 悬空 else 文法二义性的消除

应当注意的是，文法的二义性并不意味着语言是二义的。只有当产生一个语言的所有文法都是二义的时，才称为是二义的。而且也可以构造允许二义文法的分析器，但要附带消除二义性但规则（剪枝回溯）。值得看到的是，定义无二义文法可能会失去简洁性。

### 2.1.4 左递归

一个文法是左递归的，如果他有非终结符 A，对于某个串  $\alpha$ ，存在推导  $A \Rightarrow^+ A\alpha$ 。自上而下的分析方法不能用于左递归文法。

## 第 2.2 节 自顶向下分析方法

### 2.2.1 递归下降分析方法

包括一个输入缓冲区和向前看指针 lookahead，自左向右扫描输入串；设计一个辅助过程 match()，将 lookahead 指向的位置与产生式迭代生成的终结符进行匹配，如匹配，将 lookahead 挪到下一个位置。就是对于一个可能的串，从 root 开始构建一棵语法的生成树。递归下降也有着自己的一些问题，比如可能进入无限循环。下面是递归下降的三个问题，

### 2.2.2 消除左递归

左递归分为直接左递归和间接左递归。直接左递归有形如  $A \rightarrow A\alpha$  的产生式，间接左递归是通过有限次迭代（类似于证明序列）得到（非直接左递归首先变成直接左递归，然后消除他）。消除左递归的通用方法如下所示：首先将 A 的产生式组合在一起：

$$A \rightarrow A\alpha_1 | A\alpha_2 | \cdots | A\alpha_m | \beta_1 | \beta_2 | \cdots | \beta_n$$

其中  $\beta_i$  都不以 A 开始， $\alpha_i$  非空，然后用

$$\begin{aligned} A &\rightarrow \beta_1 A' | \beta_2 A' | \cdots | \beta_n A' \\ A' &\rightarrow \alpha_1 A' | \alpha_2 A' | \cdots | \alpha_m A' | \varepsilon \end{aligned}$$

代替 A 的产生式。注意后面加上了一个  $\varepsilon$ ，作为一个终结符结束递归

### 2.2.3 有左因子的文法

有左因子的 (left-factored) 文法形如  $A \rightarrow \alpha\beta_1 | \alpha\beta_2$ 。在自上而下的分析中，当不清楚应该用非终结符 A 的那个选择来替换它时，可以通过重写 A 产生式来推迟这种决定，推迟到看到足够多的输入，能帮助正确决定所需选择为止。如上式等价于

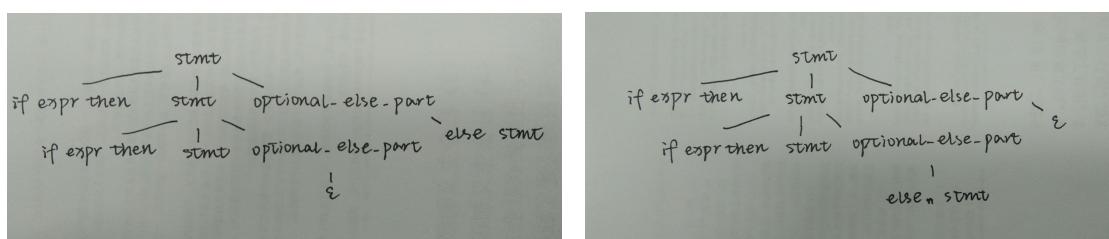
$$A \rightarrow \alpha A'$$

$$A \rightarrow \beta_1 | \beta_2$$

需要注意的是，提取左因子的重写并不能解决二义性，比如之前提到的悬空 else 文法，提取左因子之后成为

$$\begin{aligned} \text{stmt} &\rightarrow \text{if expr then stmt optional\_else\_part} | \text{other} \\ \text{optional\_else\_part} &\rightarrow \text{else stmt} | \varepsilon \end{aligned}$$

这个文法对于如  $\text{if expr then if expr then stmt else stmt}$  这个表达式时，有如下两种最左推导的分析树



### 2.2.4 复杂的回溯代价太高

非终结符有可能有多个产生式，由于信息缺失，无法准确预测选择哪一个，考虑到往往需要对多个非终结符进行推导展开，因此尝试的路径可能呈指数级爆炸

## 第 2.3 节 预测分析法

Predictive Parsing 与递归下降法类似，但是没有若干尝试，没有回溯，采用的通过向前看一些记号来预测需要用到的产生式的方法。此方法接收 LL(k) 文法，即自左向右读取(left-to-right)，最左推导(leftmost derivation)，基于对 k 个记号向前看的推导。在实际应用中，LL(1) 用的最广泛。

### 2.3.1 “第一个”集合和“跟随”集合

LL(1) 文法中有两个最重要的集合，即 FIRST( $\alpha$ ) 和 FOLLOW( $\alpha$ )。

$$FIRST(\alpha) = \{a \mid \alpha \Rightarrow^* a \dots\} \quad \alpha \in V_N, a \in V_T$$

即，可以从  $\alpha$  推导出的句型的首部词的集合

$$FOLLOW(A) = \{a \mid S \Rightarrow^* \dots Aa \dots\} \quad \alpha \in V_N, a \in V_T$$

即，可能在推导过程中紧跟在 A 右边的终结符号的集合



图 2.3: 确定给定推导文法的 FIRST 集合



图 2.4: 确定给定推导文法的 FOLLOW 集合

### 2.3.2 LL(1) 文法

则有 LL(1) 文法的定义为：对于任何两个产生式  $A \rightarrow \alpha \mid \beta$  都有

1.  $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
2. 若  $\beta \Rightarrow^* \epsilon$ ，那么  $FOLLOW(A) \cap FIRST(\alpha) = \emptyset$

对于第二条，假设  $FIRST(A) \cap FIRST(\alpha) = \{a\}$ ，即

$$\begin{aligned} S &\Rightarrow^* \dots Aa \dots \\ A &\Rightarrow^* aa' \end{aligned}$$

又由于  $\beta \Rightarrow^* \epsilon$  所以当遇到  $Aa$  时，既可以用  $A \rightarrow \alpha$  来展开，也可以用  $A \rightarrow \beta, \beta \Rightarrow^* \epsilon$  来消去 A。

LL(1) 文法有一些明显的性质，如没有公共左因子（提取左因子）、不是二义的（规则重写）、不含左递归（化为直接左递归，拆分）

## 第 2.4 节 递归下降的预测分析

所谓预测分析是指能根据当前的输入符号为非终结符确定采用哪一个选择（预读取  $k$  个 token）。在这一部分，主要是要求写出一个预测分析的程序伪代码。伪代码主要是包括两个部分，即  $match$  函数和对于每一个非终结符的函数。以如下的文法为例：

$$\begin{aligned} type &\rightarrow simple \uparrow \text{id} \mid \text{array} [simple] \text{ of } type \\ simple &\rightarrow \text{integer} \mid \text{char} \mid \text{num} \text{ dotdot num} \end{aligned}$$

其递归下降预测分析器（伪代码）为（部分）：

```
void match(terminal t)
{
    if (lookahead == t) lookahead = nextToken();
    else error();
}

void type()
{
    if ((lookahead == integer) || (lookahead == char)
        || (lookahead == num))
        simple();
    else if (lookahead == '\uparrow')
    {
        match ('\uparrow');
        match (id);
    }
    else if (lookahead == array)
    {
        /* code */
    }
    else error();
}
```

具体来说，就是每一个非终结符  $T$ ，需要构造过程  $voidT()$ ，在其中对  $T$  的产生式右端  $\alpha$  进行匹配。在匹配的判断中使用  $FIRST$  集合。匹配之后执行的动作：非终结符调用他自己的过程，终结符  $a$  调用  $match(a)$

### 2.4.1 预测分析表

对文法的每一个产生式  $A \rightarrow \alpha$ ，执行以下两个步骤：

1. 对  $FIRST(\alpha)$  的每个终结符  $A$ ，把  $A \rightarrow \alpha$  加入  $M[A, a]$ ，注意这里的  $\alpha$  是一个表达式

2. 对产生式  $A \rightarrow \alpha$ , 考虑  $FIRST(\alpha)$  和  $FOLLOW(A)$ , 如果  $\varepsilon$  在  $FIRST(\alpha)$  中, 对  $FOLLOW(A)$  的每个终结符  $b$  (包括  $\$$ ) , 把  $A \rightarrow \alpha$  加入  $M[A, b]$

值得注意的是, 填入表格的  $A \rightarrow \alpha$  是一条推导文法, 如  $E \rightarrow TE'$  或者  $T' \rightarrow \varepsilon$  注: 多重定义条目意味着文法左递归或者是二义的, 但是可以通过删除部分条目来消除

## 第 2.5 节 自底向上分析方法

**自左向右读取!** 基本方法是归约 (右推导的逆过程), 用到句柄 (可归约串, 可能不唯一) 的概念, 思路是针对输入串, 尝试根据产生式规则归约 (reduce) 到文法的开始符号, 是一个比自顶向下更一般化的方法。归约是每一步, 特定子串被替换为相匹配的某个产生式左部的非终结符。

对于句柄, 句柄是“该句型中和某产生式右部匹配的子串, 并且把它归约成该产生式左部的非终结符, 代表了最右推导的逆过程的一步”。首先句柄是字串而不一定是一个完整的串, 它是在归约过程中某一步产生的句型中的一部分。从另一个方向来看, 句柄是对应者自顶向下推导的过程中分析树的非树叶的结点。句柄的右边仅包含终结符 (这是因为归约是最右推导的逆过程)。这里“句柄的右边”指的是在这一步规约的句型中, 位置在此句柄右边的串。如果文法二义, 那么句柄可能不唯一。

比如说, 在如下的最右推导 (逆过程就是归约) 中:

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow Abc \mid b \\ B &\rightarrow d \\ S \Rightarrow_{rm} aABe &\Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abcd \end{aligned}$$

在句型  $aAbcde$  中, 这一步用于规约的句柄为  $Abc$ , 即“句型中的字串”, 同时为产生式  $A \rightarrow Abc$  的右部匹配。

### 2.5.1 移进 (shift)-归约 (reduce) 分析技术

1. 两个空间: 栈 (用来保存已经扫描过的文法符号)、缓冲区 (用来保存还未分析的文法符号)
2. 四个状态: 移进 (shift, 将下一个输入符号放到栈顶, 以形成句柄)、归约 (reduce, 句柄替换为对应的产生式的左部非终结符)、接收 (accept, 分析成功)、报错 (error)

移进-归约技术需要解决一些问题, 如:

1. 移进-归约冲突 (如何决策选择移进 (构造句柄) 还是归约)
2. 进行归约时, 确定右句型中将要归约的子串 (识别句柄)
3. 归约-归约冲突 (进行归约时, 如何确定选择哪一个产生式)

## 第 2.6 节 LR( $k$ ) 分析技术

【回顾】自顶向下和自底向上：

1. 自顶向下 (Top-down)

- (a) 针对输入串，从文法的开始符号出发，尝试根据产生式规则推导 (derive) 出该输入串。
- (b) LL(1) 文法及非递归预测分析方法
- (c) left-to-right scan + leftmost derivation

2. 自底向上 (Bottom-up)

- (a) 针对输入串，尝试根据产生式规则归约 (reduce) 到文法的开始符号。
- (b) LR( $k$ ) 文法及其分析器
- (c) left-to-right scan + rightmost derivation

一个文法，如果能为他构造出所有条目都唯一都 LR 分析表，就说他是 LR 文法。Say，LR 语法分析器的关键在于构造 LR 分析表：

1. 计算所有可能的状态

- (a) 每一个状态描述了语法分析过程中所处的位置
- (b) 可确定正在分析的产生式集合
- (c) 可确定句柄形成的中间步骤

2. 明确状态之前的跳转关系

3. 明确状态与输入之间对应的移进或者归约操作

LR 分析器的格局是二元组，其第一个成分是栈的内容，第二个成分是尚未扫描的输入。活前缀是最右句型的前缀，该前缀不超过最右句柄的右端。分析表的转移函数实际上是识别活前缀的 DFA，规约函数是对句柄

## 第 2.7 节 SLR

文法  $G$  的 LR(0) 项目（简称项目）是在其右部的某个地方加点，表示分析过程的状态的产生式。点的左边是已经看见（读入）的部分，右边是期待看见的部分。或者说，左边是历史信息，右边是展望信息。

### 2.7.1 产生规则

为了构造出指定文法的 SLR 分析表，首先重写此文法：

- ① 增加一个新的开始符号（产生拓广文法），即如果之前文法 G 的开始符号是 S，那么增加开始符号  $S'$  和产生式  $S' \rightarrow S$ ，新增产生式的目的是用来指示分析器什么时候应该停止分析并宣布接收输入
- ② 将多路选择的产生式拆分开，比如  $E \rightarrow E + T \mid T$  拆分成  $E \rightarrow E + T$  和  $E \rightarrow T$  两个产生式，这样对判断状态集有好处
- ③ 将所有的产生式编号，从  $S' \rightarrow S \ (0)$  开始

然后从新的开始产生式开始，用下面两条规则构造项目集的闭包：(和书上写的不一样)

1. 由上一个状态转换过来的产生式（点向后移动一个位置）
2. 如果某一条产生式的点在一个非终结符前面，那么看这个非终结符的所有产生式，并将其初始产生式假如到闭包中

解释一下上面的规则：

- ① 我自己定义一个概念叫初始产生式，意思是点加在产生式右部的最左端。（其实就是书上 P76 写的非核心项目加上  $S' \rightarrow \cdot S$ ，如产生式  $E \rightarrow E + T$ ，其初始产生式就是  $E \rightarrow \cdot E + T$ ）
- ② 在生成状态  $I_0$  时，从拓展文法的开始产生式  $S' \rightarrow S$  的初始产生式  $S' \rightarrow \cdot S$  开始
- ③ 状态之间的跳转并不需要对前状态的所有元素（产生式）都满足，只表达一种对于此项目集可能的跳转模式
- ④ 跳转状态，意味着接收了某种输入
- ⑤ 点在不同位置的同一产生式视作两个不同的项目，需要单独加进去。比如本来我有一个产生式  $E \rightarrow TE$ ，现在在闭包里面有  $E \rightarrow T \cdot E$ ，我还要加进去  $E \rightarrow \cdot TE$

## 2.7.2 有效项目

如果说  $S' \Rightarrow^* \alpha Aw \Rightarrow \alpha\beta_1\beta_2w$ ，那么就说项目  $A \rightarrow \beta_1\beta_2$  对活前缀  $\alpha\beta_1$  是有效的。一个项目对于好几个活前缀都可以是有效的，比如若  $\beta_2 \neq \varepsilon$  则应当移进（继续读取才能规约为 A）；若  $\beta_2 = \varepsilon$ ，则已经可以直接归约了。

## 2.7.3 SLR 分析表

SLR 分析表分为左右两部分，左边是动作 (action) 表，右边是转移 (goto) 表。二者的行标分别为终结符加 \$ 和非终结符，列标为项目集（状态）。action 表分为三种情况：

1. 对于没有读取完的产生式项目（点在右部中间，形如  $A \rightarrow \alpha \cdot a\beta$ ， $a$  为终结符， $\alpha, \beta$  可以为  $\varepsilon$ ），设这个项目所在的状态为  $I_i$ ，读取  $a$  之后会跳转到  $I_j$ ，即  $I_i \xrightarrow{a} I_j$ ，那么就在分析表的第 i 行第 a 列填入  $s_j$
2. 对于读取完毕的产生式项目，设这个项目所在的状态为  $I_i$ ，产生式的编号为  $j$ ，那么就在分析表的第 i 行第 a 列填入  $r_j$ ，其中 a 是 FOLLOW(A) 中的所有元素

3. 对于包含项目  $S' \rightarrow S\cdot$  的状态  $i$ , 在第  $i$  行第  $\$$  列填入  $acc$

转移表就是说如果状态  $i$  在接收非终结符  $A$  之后转移到状态  $j$ , 那么就在第  $i$  行第  $A$  列填入  $j$

#### 2.7.4 SLR(1) 文法

一个上下文无关文法  $G$ , 通过上述算法构造出 SLR 语法分析表, 且表项中没有移进/归约或者归约/归约冲突, 那么  $G$  就是 SLR(1) 文法。1 代表了当看到某个产生式右部时, 只需要再向前看 1 个符号就可决定是否用该式进行归约。通常可以省略 1, 写作 SLR 文法。

#### 2.7.5 判定满足 SLR 文法输入串

根据 SLR 文法分析表, 使用“文法符号栈-输入缓冲区-对应行为”三联表的形式, 尝试读取并分析输入串, 直到 ACC 或者 ERR

### 第 2.8 节 规范的 LR 分析方法

在识别活前缀 DFA 的状态中, 增加信息, 可以排除一些不正确的归约操作, 故规范的 LR 分析方法增加了前向搜索符: 一个项目  $A \rightarrow \alpha \cdot \beta$ , 如果真的用这个产生式进行规约之后, 期望看到的符号是  $a$  (换句话说, 在自底向上分析的过程中, 对产生式右部  $\alpha\beta$  进行规约为  $A$  之后, 在  $A$  的右边应当出现的符号)。LR(1) 项目是一个二元组 (SLR 中的项, 搜索符), 形式的定义为  $[A \rightarrow \alpha \cdot \beta, a]$ 。当  $\beta$  不为空的时候,  $a$  不起作用, 当  $\beta$  为空的时候, 如果下一个输入符号为  $a$ , 那么将按照  $A \rightarrow \alpha$  进行规约, 故有  $a$  的集合是  $FOLLOW(A)$  的子集。

#### Def 2.8.1. 对活前缀有效

称 LR(1) 项目  $[A \rightarrow \alpha \cdot \beta, a]$  对活前缀  $\gamma$  有效, 当且仅当如下情形: 如果存在着推导  $S' \Rightarrow_{rm}^* \delta Aw \Rightarrow_{rm} \delta\alpha\beta w$ , 其中  $\gamma = \delta\alpha$ ,  $a$  是  $w$  的第一个符号, 或者  $a$  是  $\$$  且  $w$  是  $\varepsilon$

#### 2.8.1 构造 LR(1) 项目集规范族

先声明一个我自己瞎起的名字: 产生式左边的叫左值, 右边的都叫右值。构造方法和 SLR 的构造方法类似:

1. 不妨设拓广文法的新开始产生式为  $S' \rightarrow S$ , 那么从  $S' \rightarrow \cdot S, \$$  开始
2. 【扩充闭包】在一个状态 (项目集) 中, 对他现有的每一个项目  $[A \rightarrow \alpha \cdot B\beta, a]$ , 进行如下构造:
  - (a) 【左值】考虑点后面的非终结符  $B$
  - (b) 【右值】对于拓展文法中的所有  $B$  在左边的产生式  $B \rightarrow \gamma$  (和 SLR 考虑的一样)
  - (c) 【搜索符】考虑  $FIRST(\beta a)$  中的每一个终结符  $b$ , 并将  $[B \rightarrow \cdots \gamma, b]$  不重复地加入到状态集合中

3. 【找全状态】规则和 SLR 一样，只是要注意，不仅点的位置不同就不同，搜索符不同也不同

回顾一下 FIRST 集合的确定



10/11/2020

Cheng @ Compiler Fall 2020, USTC

57



10/11/2020

Cheng @ Compiler Fall 2020, USTC

63

图 2.5: 确定给定推导文法的 FIRST 集合

图 2.6: 确定给定推导文法的 FOLLOW 集合

## 2.8.2 构造规范的 LR 分析表

表格形式与 SLR 类似，但是规则稍有变动：

1. 当点在右值中间时 ( $[A \rightarrow \alpha \cdot a\beta, b]$ )，根据  $I_i \xrightarrow{a} I_j$ ，将  $s_j$  填入 i 行 a 列。注意这种情况下，搜索符  $b$  是没有用的。在这种产生式的形式下， $b$  的唯一用处是在求闭包的时候，求  $FIRST(\beta a)$
2. 当点在右值最右端时，不再看 **FOLLOW(A)**，而是对于  $I_i$  中的所有  $[A \rightarrow \alpha \cdot, a]$ ，将  $r_j$  (j 为产生式编号) 填入 i 行 a 列 (其实就是看了 FOLLOW 的一个子集)
3. acc 和 goto 不变

## 2.8.3 LALR

就是在规范 LR 的基础上合并同心项目集 (i.e. 略去搜索符之后他们是相同的集合)。并按照规范 LR 的规则构造 LALR 分析表。合并同心项目集可能会引起冲突，但不会引起新的移进-归约冲突

# 第3章 语法制导的翻译

这一章开始赋予抽象的符号系统以实际的含义，进入到生成中间代码的阶段。

语法和文法就像数理逻辑中的(语法)和(语义)，编译原理的语法是对前面文法分析树在具体程序语言(前提集)上的“赋值”，使得终结符和非终结符有了自己的“属性”。做一组不是很恰当的类比：语法=语法，语义规则=赋值，串=逻辑式，翻译=逻辑式的真假，文法符号=个体变元，属性=指派

## 第3.1节 语法制导的定义

语法制导(SDD)的定义是带属性和语义规则的上下文无关文法。SDD为CFG中的文法符号设置语义属性，用来表示语法成分对应的语义信息。语法制导翻译是使用上下文无关文法(CFG)来引导对语言的翻译，是一种面向文法的翻译技术。

**如何表示语法信息** 为CFG中的文法符号设置语义属性，用来表示语法成分对应的语义信息

**如何计算语义属性** 文法符号的语义属性值是用与文法符号所在产生式(语法规则)相关联的语义规则来计算的：对于给定的输入串x，构建x的语法分析树，并利用与产生式相关联的语义规则来计算分析树中各结点对应的语义属性值

在语法制导中，用的是基础的上下文无关文法，每一个文法符号有一组属性，每一个文法产生式 $A \rightarrow \alpha$ 有一组形式如 $b = f(c_1, c_2, \dots, c_k)$ 的语义规则，不一定这个产生式中的每一个元素的(每一个)属性都需要写进去。如果b是左值的属性， $c_1, \dots, c_k$ 是右值的属性或者左值的其他属性，那么就叫b为综合属性。**终结符只能有综合属性，属性值无需计算，由词法分析给定**

### 3.1.1 综合属性和继承属性

仅仅使用综合属性的语法制导定义称为S属性定义。但是不是所有情况都这么简单，这时候就会用到继承属性(比如在消除了左递归的文法里面)

总的来说，综合属性将属性自底向上传，而继承属性自上而下传。而对于如下产生式

$$T \rightarrow FT'$$

$$T' \rightarrow *FT'_1$$

在计算 T 的综合属性  $T.syn$  时, 由于乘法运算符和操作数都隐藏在了  $T'$  里面, 而这个隐藏内容在后续产生式  $T' \rightarrow *FT'_1$  里面, 那么就需要将 F 的属性保存于  $T'$  的继承属性里面, 传递到下一个“计算周期”, 得到结果之后再返回到上一级并传递给 T。如下:

产生式	语义规则
$T \rightarrow FT'$	$T'.inh = F.val, T.val = T'.syn$
$T' \rightarrow *FT'_1$	$T'_1.inh = T'.inh \times F.val, T'.syn = T'_1.syn$

### 3.1.2 注释分析树和属性依赖图

根据语法分析, 构建出来语法分析树, 然后在每一个节点根据其属性值进行计算。当这些属性都是综合属性时, 计算可以自底向上地完成。

如果分析树上的一个节点的属性 b 依赖于某个节点的属性 c, 那么 b 的语义规则的计算必须在 c 之后。这种依赖关系可以用一种称为**依赖图**的有向图来描述。

**虚拟属性** 在构造分析树的依赖图之前, 先为由过程调用组成的语义规则引入虚拟综合属性  $b$ , 使得每一条规则都能写成  $b = f(c_1, c_2, \dots, c_k)$  的形式。在语义分析时, 每一个产生式都有一个综合属性 (至少在向上返回的时候会用到的)。在后续推理过程中, 逐渐 (按需) 加入继承属性。

属性依赖图的组成是: 分析树上的每个结点的每一个属性, 都在依赖图上构造一个结点, 如果属性 b 依赖于属性 c, 那么从 c 到 b 连一条有向边。

1. 属性值为点 (vertex): 分析树中每个标号为 X 的结点的每个属性 a 都对应着依赖图中的一个结点
2. 属性依赖关系为边 (edge): 如果属性  $X.a$  的值依赖于属性  $Y.b$  的值, 则依赖图中有一条从  $Y.b$  的结点指向  $X.a$  的结点的有向边

**属性计算次序** 由语法制导定义规范的翻译可以准确地按照以下步骤 (分析树方法) 完成:

1. 根据基础文法构造输入的分析树
2. 构造属性依赖图
3. 对依赖图进行拓扑排序, 得到语义规则对计算次序
4. 按这个次序计算属性得到输入串的翻译

属性计算依赖于拓扑排序, 那么在有向图中, 什么时候拓扑排序不存在呢: ① 当图中出现环的时候 ② SDD 的属性之间存在循环依赖关系 (有互锁?)

解决方案: 使用某些特定类型的依赖图不存在环的 SDD: S 属性的 SDD 和 L 属性的 SDD 一个没有副作用的 SDD 称为属性文法

1. 属性文法增加了语义规则描述的复杂度

2. 如: 符号表必须作为属性传递
3. 为了简单起见, 我们可以把符号表作为全局变量, 通过副作用函数读取或者添加标识符

### 3.1.3 具有受控副作用的语义规则

在实践中, 翻译过程会出现一些副作用 (什么是副作用呢? 就是当一个操作/函数除了产生其计算结果外, 还对其他部分产生了影响: 一个桌上计算器可能打印出一个结果; 一个代码生成器可能把一个标识符的类型加入到符号表中)。对于 SDD, 我们在属性文法和翻译方案之间找到一个平衡点。(why?) 属性文法<sup>1</sup>没有副作用, 并支持任何与依赖图一致的求值顺序。翻译方案 (3.2) 要求按从左到右的顺序求值, 并允许语义动作包含任何程序片段。

我们将按照下面的方法之一来控制 SDD 中的副作用:

1. 支持那些不会对属性求值产生约束的附带副作用, 换句话说, 如果按照依赖图的任何拓扑顺序进行属性求值时都可以产生“正确的”翻译结果, 我们就允许副作用存在。这里的“正确”要视具体应用而定。
2. 对允许的求值顺序添加约束, 使得以任何允许的求值顺序都会产生相同的翻译结果, 这些约束可以被看作隐含加入到依赖图中的边

## 第 3.2 节 语法制导的翻译方案

语法制导翻译方案 (syntax-directed translation scheme, SDT) 是语法制导定义 (syntax-directed definition, SDD) 的一种补充。所有语法制导定义的应用都可以使用语法制导的翻译方案来实现。语法制导的翻译方案是在产生式右部中嵌入了程式片段 (称为语义动作, 它们可以出现在产生式体的任何地方) 的上下文无关文法 (CFG)

SDT 可以看作是 SDD 的具体实现方案。任何 SDT 都可以通过下面的方法实现: 首先建立一棵语法分析树, 然后按照从左到右的深度优先顺序来执行这些动作 (前序遍历)。不过, 通常情况下, SDT 是在语法分析中实现的, 而不会真的构造一棵语法分析树。那么下面就是如何使用 SDT 来实现两个重要的 SDD, i.e:

1. 基本文法可以使用 LR 技术分析, 且 SDD 是 S 属性的 (仅仅用综合属性的 SDD 称为 S 属性的 SDD)
2. 基本文法可以使用 LL 技术分析, 且 SDD 是 L 属性的 (在一个产生式所关联的各属性之间, 依赖图的边可以从左到右, 但不能从右到左 (因此称为 L 属性的, L 是 Left 的首字母))

语法制导的翻译方案和语法制导定义不同之处是他的语义动作 (在此不叫语义规则) 放在花括号内, 并且可以插入到产生式右部的任何地方, 即这种表示法动作与分析交错。若  $A \rightarrow \alpha\{\cdots\}\beta$ , 那么花括号中语义动作的执行要在  $\alpha$  的推导结束之后, 在  $\beta$  的推导开始之前。下面是有关继承属性的几个限制:

---

<sup>1</sup>一个没有副作用的 SDD 有时也称为属性文法 (attribute grammar)。一个属性文法的规则仅仅影响其他属性值和常量值来定义一个属性值。(龙书 P184, line 7)

1. 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
2. 一个动作不能引用该动作右边符号的综合属性
3. 左值的综合属性要放在右部的末端
4. 多个继承属性，要考虑次序，避免形成环

## 第 3.3 节 S 属性定义的自下而上计算

仅仅使用综合属性的语法制导定义称为 S 属性的 SDD，或 S-属性定义、S-SDD。

### 3.3.1 语法树

抽象语法树（简称语法树）是分析树的浓缩表示。在语法树中，算符（非终结符）和关键字（终结符）不是作为叶结点，而是作为分支结点。语法树上的内部结点都代表运算，其子节点都是他的运算对象。如同在分析树中那样，在语法树中也可以把属性附加到结点。

语法树的结点可以用有若干指针域的记录来实现。对于算符结点，一个域放运算符，作为结点对象的标记，其余两个域存放只想运算对象的指针。对于基本运算对象结点，一个放运算对象类型，一个放他的值。在真正用于翻译时，语法树的结点可能还有其他域来保存其他属性值，或者属性值的指针（比如说，对于产生式  $S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$ ，运算符 **if-then-else** 就有 4 个域（自己和三个运算对象）

### 3.3.2 S 属性的 SDD

每个属性都是综合属性。在依赖图中，总是通过子结点的属性值来计算父结点的属性值。

语法制导定义 SDD 是对 CFG 的推广 ① 将每个文法符号和一个语义属性集合相关联 ② 将每个产生式和一组语义规则相关联，用来计算该产生式中各文法符号的属性值

### 将一个 S 属性的 SDD 转换为 SDT

将每个语义动作都放在产生式的最后，并且在按照这个产生式体归约为产生式头的时候执行这个动作。所有动作都在产生式最右端的 SDT 称为“后缀翻译方案”。可以通过 ① 基于分析树的语法制导翻译方案 ② 语法分析栈来实现。

### 3.3.3 后缀 SDT 的语法分析栈实现

后缀 SDT 可以在 LR 语法分析的过程中实现，当归约发生时执行相应的语义动作。各个文法符号的属性值可以放到栈中的某个位置。S 属性定义的翻译器可以借助 LR 分析器的生成器来实现，LR 可以把文法符号的综合属性放入栈内。

如果所有属性都是综合属性，并且所有动作都位于产生式的末端，那么我们可以在把产生式体归约为产生式头的时候计算各个属性的值（why？这个时候产生式体中的所有符号都已经位于符号表中了——想一想综合属性的定义）。如果我们使用  $A \rightarrow XYZ$  这样的产生式进行归

## S-属性定义的SDT实现-1

中国科学技术大学  
University of Science and Technology of China

### □基于分析树的语法制导翻译方案

$L \rightarrow E \text{ n}$	{print ( $E.val$ )}	V1
$E \rightarrow E_1 + T$	{ $E.val = E_1.val + T.val$ }	V2
$E \rightarrow T$	{ $E.val = T.val$ }	V3
$T \rightarrow T_1 * F$	{ $T.val = T_1.val * F.val$ }	V4
$T \rightarrow F$	{ $T.val = F.val$ }	V5
$F \rightarrow (E)$	{ $F.val = E.val$ }	V6
$F \rightarrow \text{digit}$	{ $F.val = \text{digit.lexval}$ }	V7

- 语句  $8+5*2$  的分析树如右
- 深度优先可知动作执行顺序
  - $V7_1, V5_1, V3, V7_2, V5_2, V7_3, V4, V2$

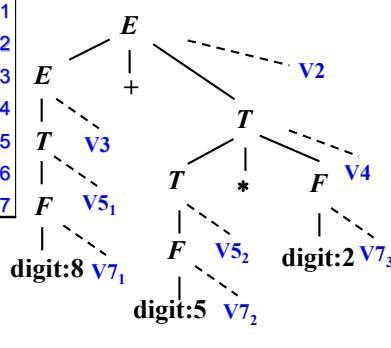


图 3.1: 基于分析树的语法制导翻译方案

约,那么此时 X、Y 和 Z 的所有属性都是可用的,并且都位于已知的位置上(栈顶)。在这个动作之后, A 和他的属性都位于栈的顶端,即现在存放 X 的记录的位置上。

这又引出了另一种语义动作的表示方法,即用栈中相对于栈顶的位置来表示各个属性。如下面的 SDT 语句:

$$A \rightarrow XYZ \{A.val = f(X.val, Y.val, Z.val)\}$$

在 state 和 val 栈上可以转变为如下操作:

$$\begin{cases} state[top - 2] = A \\ val[top - 2] = f(val[top], val[top - 1], val[top - 2]) \\ top = top - 2 \end{cases}$$

扩展 LR 的语法分析栈:

- 在分析栈中使用一个附加的域来存放综合属性值。若支持多个属性,那么可以在栈中存放指针
- 每一个栈元素包含状态、文法符号、综合属性三个域。也可以将分析栈看成三个平行的栈,分别是状态栈、文法符号栈、综合属性栈,分开看的理由是,入栈出栈并不完全同步(即有些时候会有某一项不变的情况)
- 语义动作将修改为对栈中文法符号属性的计算

栈操作代码:分为对照的两部分,产生式和代码段。和之前的代码段(对综合属性的赋值及运算)不同,这里由于将正在被分析的结点(们)的属性都入栈了(还维护了一个 top 指针),现在将这些属性用他们在栈中的位置来表达。注意在这个自底向上的分析中,每一行栈的排布是:产生式“右值”从右向左依次从栈顶向下排,在执行这一步的归约之后 pop 出去所有的右值,并将左值入栈。

总的来说，采用自底向上分析，例如 LR 分析，首先给出 S-属性定义，然后，把 S-属性定义变成可执行的代码段，放到产生式尾部，这就构成了翻译程序。（注：看一下紫书 P118-119 的表 4.5，前三栏分别为状态栈、文法符号栈、综合属性栈）。随着语法分析的进行，归约前调用相应的语义子程序，完成翻译的任务。

## 第 3.4 节 L 属性定义的自上而下计算

显然，S 属性定义属于 L 属性定义。

边分析边翻译的方式能否用于继承属性？

- 属性的计算次序一定受分析方法所限定的分析树结点建立次序的限制
- 分析树的结点是自左向右生成
- 如果属性信息是自左向右流动，那么就有可能在分析的同时完成属性计算

可以在语法分析过程中实现的 SDT 包括后缀 SDT 和后面的一种“实现了 L 属性定义的”SDT。不是所有的 SDT 都可以在语法分析过程中实现（详见龙书 P197 的例子）

### 3.4.1 产生式内部带有语义动作的 SDT

动作可以放置在产生式体中的任何位置上。当一个动作左边当所有符号都被处理过后，该动作立刻执行。因此，如果我们有一个产生式  $B \rightarrow X\{a\}Y$ ，那么当我们识别到 X（如果 X 是终结符）或者所有从 X 推导出当终结符号之后，动作 a 就会立即执行。更准确地讲，

- 如果语法分析过程是自底向上的，那么我们在 X 的此次出现位于语法分析栈的栈顶时，立刻执行动作 a。从栈操作的过程上来看，这时恰好刚刚执行完产生式头为 X 的产生式的归约，故此时根据本节的定义，执行动作 a 的条件已经完备
- 如果语法分析过程是自顶向下的，那么我们在试图展开 Y 的本次出现（如果 Y 是非终结符）或者在输入（输入符号表）中检测 Y（如果 Y 是终结符）之前执行语义动作 a。试想为何会将动作 a 放在这个位置？其实联想继承属性的求值方法不难看出，只有执行了这里的动作（如计算一个 X 的继承属性）才能在拆分 Y 的时候获得所有需要的值。想一下语法树和 M 点。

### 3.4.2 从 SDT 中消除左递归

前面语法分析的时候（2.2.2 节）有对消除左递归的描述。当文法是 SDT 的一部分时，我们还需要考虑如何处理其中的动作。

**情况 1：语句无赋值影响** 先考虑简单的情况，即只需要关心一个 SDT 中的动作的执行顺序的情况（如每个动作只打印一个字符串）。在这种情况下，可以应用下面这个原则完成这个转化：

- 当转换文法时，将动作当成终结符号处理

这个原则基于下面的思想：文法转换保持了由文法生成的符号串中终结符号的顺序（因为打印只有副作用而对各个符号对赋值没有任何影响）。因此，这些动作在任何从左到右对语法分析中都按照相同的顺序执行，不管这个分析是自顶向下还是自底向上的。

**情况 2：动作是计算属性的值** 好消息：如果这个 SDD 是属性的，那么我们总是可以通过将计算属性值的动作放在新产生式中的适当位置来构造一个 SDT（想一下前一个小节‘S 属性的自下而上计算’中的处理）。

下面是一个通用的解决方案，以解决只有单个递归产生式、单个非递归产生式并且该左递归非终结符号只有单个属性的情况（这么描述十分抽象，事实上可以看下面这个例子）。将这个方案推广到多个递归/非递归产生式的情况并不困难，但是写起来非常麻烦。假设这两个（S 属性的）产生式是：

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

基础文法会被改写成：

$$\begin{aligned} A &\rightarrow X R \\ R &\rightarrow Y R \mid \epsilon \end{aligned}$$

为了完成这个翻译，我们使用如下 SDT：（对于 R，使用一个继承属性  $R.i$  来累计从  $A.a$  的值开始不断应用  $g$  所得的结果

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

注意到这里还有一个综合属性  $R.s$ 。当 R 不再生成文法符号 Y 时才开始计算这个属性的值，这个时间点时以产生式  $R \rightarrow \epsilon$  的使用为标志的

## 第 3.5 节 L 属性定义的 SDT

很多翻译应用可以用 L 属性来解决，**下面的方法通过遍历语法分析树来完成翻译工作（重要！）**

1. 建立语法分析树并注释。这个方法对任何非循环定义的 SDD 都有效
2. 构造语法分析树，加入动作，并按照前序遍历顺序来执行这些动作。可以处理任何 L 属性定义，下面的第一节讨论了如何把一个 L 属性 SDD 转变为 SDT，还特别讨论了**如何根据这样的 SDD 的语义规则把语义动作嵌入到产生式中**
3. 使用一个递归下降的语法分析器。他为每个终结符都建立了一个函数。对非终结符 A 对函数以参数的方式接受 A 的继承属性，并返回 A 的综合属性
4. 使用一个递归下降的语法分析器，以边扫描边分析的方式生成代码

5. 与  $LL$  语法分析器结合，实现一个  $SDT$ 。属性的值存放在语法分析栈中，而各个规则从栈中的已知位置获取需要的属性值。
6. 与  $LR$  语法分析器结合，实现一个  $SDT$ 。这个方法会让人觉得惊讶，因为一个  $L$  属性 SDD 的  $SDT$  通常有一些动作位于产生式中间，而在一个  $LR$  语法分析过程中，我们只有在构造出一个产生式体的全部符号之后才能肯定确实可以使用这个产生式。然而，**如果基础文法是  $LL$  的，总是可以按照自底向上的方式来处理语法分析和翻译过程**

对于  $L$  属性的计算，考虑结合  $SDT$ ，在语法分析过程中进行翻译

1. 自顶向下计算
  - (a) 递归下降分析器
  - (b)  $LL$  分析器
2. 自底向上计算，考虑与  $LR$  分析器的结合
  - (a) 删除翻译方案中嵌入的动作
  - (b) 继承属性在分析栈中的计算

### 3.5.1 将 $L$ 属性的 SDD 转换为 $SDT$

假设基础文法以自顶向下的方法进行语法分析，因为如果不是这样，那么翻译过程常常无法和一个  $LL$  或  $LR$  语法分析器一起完成。对于任何文法，我们只需要将动作附加到一棵语法分析树中，并在对这棵树进行前序遍历时执行这些动作，便可以实现下面的技术。将一个  $L$  属性的 SDD 转换为一个  $SDT$  的规则如下：

1. 把计算一个产生式左部符号的综合属性的动作放置在这个产生式右部的最右端
2. 将计算某个非终结符号  $A$  的继承属性的动作插入到产生式右部中紧靠在  $A$  的本次出现之前的位置上（多个继承属性，要考虑次序，防止形成环）

### 3.5.2 在递归下降语法分析过程中进行翻译

在 2.2.1 节中提到，一个递归下降的语法分析器对每个非终结符号  $A$  都有一个函数  $A$ （伪代码描述见 2.4 节）。我们可以按照如下方法把这个语法分析器扩展为一个翻译器：

1. 为每个非终结符  $A$  构造一个函数
  - (a)  $A$  的每个继承属性对该函数的一个形参
  - (b) 函数的返回值是  $A$  的综合属性值
2. 在函数体中，要进行语法分析并处理属性
  - (a) 首先选择适当的  $A$  的产生式
  - (b) 用局部变量保存产生式中文法符号的属性
  - (c) 对产生式体中的终结符号，读入符号并获取其综合属性（由词法分析得到）
  - (d) 对产生式体中的非终结符，调用相应函数，记录返回值

### 3.5.3 边扫描边生成代码

### 3.5.4 L 属性的 SDD 和 LL 语法分析

需要重新看录课，正确性未知。在龙书 5.5.3, page 208

需要对分析栈进行扩充：增加动作记录 action、增加属性记录

假设一个 L 属性 SDD 是一个 LL 文法，并且已经按照前面所述的方法转换为一个 SDT，其语义动作被嵌入到各个产生式中。然后，就可以在 LL 语法分析过程中完成翻译过程，**其中的语法分析栈需要扩展，以存放语义动作和属性求值所需的某些数据项**。一般来说，这些数据项是属性值的复制

这里注意一个细节：SDD 叫语法制导定义，SDT 叫语法制导翻译方案，而现在我们做的是语法制导翻译这个过程

除了那些代表终结符和非终结符的记录之外，语法分析栈中还将保存动作记录和综合记录，其中动作记录表示即将被执行的动作，而综合记录保存非终结符号的综合属性值。使用下列两个原则来管理栈中的属性：

1. 非终结符 A 的继承属性放在表示这个非终结符号的栈记录中。对这些属性求值的代码通常使用紧靠在 A 的栈记录之上的动作记录来表示。实际上，从 L 属性的 SDD 到 SDT 的转换方法保证了动作记录将紧靠在 A 的上面
2. 非终结符 A 的综合属性放在一个单独的综合记录中，他在栈中紧靠在 A 的记录下面

### 3.5.5 L 属性的 SDD 的自底向上语法分析

考虑与 LR 分析器的结合

我们可以使用自底向上的方法来完成任何可以用自顶向下方式完成的翻译过程。更准确的说，给定一个以 LL 文法为基础属性的 L 属性 SDD，可以修改这个文法，并在 LR 语法分析过程中计算这个新文法之上的 SDD。这个“技巧”包括三个部分：

1. 以按照 3.5.1 节中的方法构造得到的 SDT 为起点。这样的 SDT 在各个非终结符之前放置语义动作来计算他的继承属性，并且在产生式后端放置一个动作来计算综合属性
2. 对每个内嵌的语义动作，向这个文法中引入一个标记非终结符来替换他。每个这样的位置都有一个不同的标记，并且对于任意一个标记 M 都有一个产生式  $M \rightarrow \epsilon$
3. 如果标记非终结符 M 在某个产生式  $A \rightarrow \alpha\{a\}\beta$  中替换了语义动作 a，对 a 进行修改得到  $a'$ ，并且将  $a'$  关联到  $M \rightarrow \epsilon$  上。这个动作  $a'$ 
  - (a) 将动作 a 需要的 A 或者  $\alpha$  中符号的任何属性作为 M 的继承属性进行复制
  - (b) 按照 a 中的方法计算各个属性，但是将计算得到的这些属性作为 M 的综合属性

**为什么标记能正确工作？** 标记是只能推导出  $\epsilon$  的非终结符，每个标记在所有产生式体中只出现一次。下面正式证明如果一个文法是 LL 的，那么标记非终结符号可以被插入到产生式体中的任何位置，并且结果文法是 LR 的。

事实上，如果文法是 LL 的，那么只需要看输入符号串  $w$  的 FIRST 符号（如果  $w$  为空则是 FOLLOW 符号），就可以确定  $w$  是否可以从  $A$  开始，经过一个以产生式  $A \rightarrow \alpha$  开头的推导序列得到。因此，如果我们用自底向上的方式对  $w$  进行语法分析，那么只要  $w$  的开头出现在输入中，就可以确定  $w$  的一个前缀必须被归约成为  $\alpha$ ，然后再归约到  $S$ 。特别是，如果在  $\alpha$  的任何位置插入标记，相应的 LR 状态将隐含的表明这个标记必定存在，并将在输入的正确位置上把  $\varepsilon$  归约为标记

# 第 4 章 中间代码生成

中间表示设计的选择随编译器不同而不同。中间表示可以是一种实际的语言，也可以是编译各阶段共享的内部数据结构。C 是一种编程语言，但它经常被当作一种中间形式，这是因为它灵活，能生成高效的机器代码，并且他的编译器到处可用。

本章按照龙书编排，因为我不太喜欢 PPT 和紫书的顺序 orz

## 第 4.1 节 中间语言

后缀表示 波兰式、逆波兰式……

图形表示（有向无环图，即 DAG）

### 4.1.1 三地址代码

三地址代码是语法树或者 DAG 的一种线性表示，其中新增加的临时名字对应图的内部结点。

静态单赋值形式，SSA

## 第 4.2 节 表达式的翻译<sup>1</sup>

本节介绍在翻译表达式和语句时出现的问题。在本节中，先考虑从表达式到三地址代码（4.1.1）的翻译。一个带有多个运算符的表达式（比如  $a + b * c$  将被翻译成为每条指令最多包含一个运算符的指令序列。类似的，一个数组引用  $A[i][j]$  将被扩展为一个计算该引用的地址的三地址指令序列。我们将在下一章考虑表达式的类型检查。

### 4.2.1 符号表中的名字

在介绍中间语言时，为了直观起见，让名字本身直接出现在三地址代码中，实际上应该把名字理解为他们在符号表中位置的指针。编译器在处理表达式、赋值语句等构造中的名字时，需要在符号表中查找它的定义，获得他的属性，然后在生成的三地址代码中使用它在符号表中位置的指针。下面一个小节中的 *top.get* 函数即是此作用（略去了检查是否返回值为 NIL 的步骤）

---

<sup>1</sup>龙书 6.4 节

### 4.2.2 表达式中的运算

本章中表达式均用符号 **E** 表示。

下面的语法制导定义使用 **S** 的属性 *code* 以及表达式 **E** 的属性 *addr* 和 *code*, 为一个赋值语句 **S** 生成三地址代码。属性 *S.code* 和 *E.code* 分别表示 **S** 和 **E** 对应的三地址代码。属性 *E.addr* 则表示存放 **E** 的值的地址。回忆一下前面对于三地址代码的描述 (4.1.1), 一个地址可以是变量名字、常量或者编译器产生的临时量。

表 4.1: 表达式的三地址代码

产生式	语义规则
$S \rightarrow \mathbf{id} = E ;$	$S.\text{code} = E.\text{code} \parallel \text{gen}(\text{top.get}(\mathbf{id}.lexeme) ' =' E.\text{addr})$
$E \rightarrow E_1 + E_2$	$E.\text{addr} = \mathbf{new Temp}()$ $E.\text{code} = E_1.\text{code} \parallel E_2.\text{code} \parallel \text{gen}(E.\text{addr}' =' E_1.\text{addr}' +' E_2.\text{addr})$
$  -E_1$	$E.\text{addr} = \mathbf{new Temp}()$ $E.\text{code} = E_1.\text{code} \parallel \text{gen}(E.\text{addr}' ='' \mathbf{minus}' E_1.\text{addr})$
$  (E_1)$	$E.\text{addr} = E_1.\text{addr}$ $E.\text{code} = E_1.\text{code}$
$  \mathbf{id}$	$E.\text{addr} = \text{top.get}(\mathbf{id}.lexeme)$ $E.\text{code} = ''$

第一个表项 (非终结符 **S**) 为一个完整的语法分析的语句, 而非终结符 **E** 则是此语句中的表达式。其中二元运算以加法为例。

令 *top* 表示当前的符号表。当函数 *top.get* 被应用于 **id** 的这个实例的字符串表示 **id.lexeme** 时, 它返回对应的符号表条目。

**new Temp()** 指令会产生一个新的临时变量 *t*, 而连续执行这个指令会产生一系列互不相同的临时变量  $t_1, t_2, \dots$

*gen* 用于表达式和字符串的组合。为方便起见, 使用记号  $\text{gen}(x' =' y' +' z)$  来表示三地址指令  $x = y + z$ 。当被传递给 *gen* 时, 变量 *x*、*y*、*z* 的位置上出现的表达式将首先被求值, 而像等号这样的引号内的字符串则按照字面值传递 (在 SDD 中, *gen* 构造一条指令并返回他, 而在 SDT 中, *gen* 构造出一条指令, 并增量地将它添加到指令流中去)

对 *addr* 属性的赋值可以看作是将这个非终结符的 *attributes* 填入符号表, 而对这个属性的调用可以看作是对符号表的查找。**S** 不具有这个 *addr* 的操作是因为不需要在其他 (对表达式 *code* 进行处理的) 地方用到 **S** 的值。

除了对左式为 **S** 的产生式的 SDD, 在书写时都包含两部分, 即对 *E.addr* 和对 *E.code* 的赋值。

考虑上面的语法制导定义的最后一个产生式  $E \rightarrow \mathbf{id}$ 。若表达式只是一个标识符, 比如说 *x*, 那么 *x* 本身就保存了这个表达式本身的值。这个产生式对应的语义规则把 *E.addr* 定义为指向该 **id** 的实例对应的符号表条目的指针。*E.code* 被设置为空串

当翻译产生式  $E \rightarrow E_1 + E_2$  时，首先将二者当 *code* 连接起来，然后再加上一条将  $E_1$  和  $E_2$  相加的指令（即 **gen** 语句）。新增加的这条指令将求和的结果放入一个为  $E$  生成的临时变量中，用  $E.\text{addr}$  表示。（为什么要这么处理呢？首先作为一个整体，非终结符  $E$  的任务就是计算  $E_1$  和  $E_2$  的和，所以一定需要添加那条 **gen** 语句。其次这个翻译过程可以看作是一个沿着语法分析树自底向上回溯的过程，生成的  $E$  的 *code* 里面应当包含底层的所有代码，这样才能在顶层也可以获取  $E_1$  和  $E_2$  的值。）

### 4.2.3 数组元素的寻址

如 OS 中所述，数组元素被存储在一块连续的空间中。对于  $k$  维数组，元素  $A[i_1][i_2] \cdots [i_k]$  的相对地址可用如下公式计算：

$$\text{base} + i_1 \times w_1 + i_2 \times w_2 + \cdots + i_k \times w_k$$

其中 **base** 时分配给数组  $A$  的内存块的相对地址，即  $A[0]$  的相对地址。而  $w_i$  是第  $i$  个维度的宽度。另一种表示方式是根据第  $j$  维上数组元素第  $j$  个数  $n_j$  和该数组第每个元素第宽度  $w = w_k$  来计算的。如下：

$$\text{base} + ((\cdots ((i_1 \times n_2 + i_2) \times n_3 + i_3) \cdots) \times n_k + i_k) \times w_k$$

## 第 4.3 节 布尔表达式和控制流语句

在编程语言中，布尔表达式有两个基本目的，第一是用于计算逻辑值，第二而且更经常的是作为条件表达式，用于控制流语句，如 **if-then**、**if-then-else** 和 **while-do** 语句。

**if-else** 语句、**while** 语句这类语句的翻译和对布尔表达式对翻译是结合在一起的。在程序设计语言中，布尔表达式经常用到：

1. 改变控制流。布尔表达式被用作语句中改变控制流的条件表达式。这些布尔表达式的值由程序到达的某个位置隐含地指出。例如，在 **if**( $E$ ) $S$  中，如果运行到语句  $S$ ，就意味着表达式  $E$  的取值为真。
2. 计算逻辑值。一个布尔表达式的值可以表达 *true* 或者 *false*。这样的布尔表达式也可以像算术表达式一样，使用带有逻辑运算符的三地址指令进行求值

### 4.3.1 布尔表达式

下面是本节所用的布尔表达式文法：(**relop** 是关系运算符，如  $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $==$ ,  $!=$  等)

$$B \rightarrow B \text{ or } B \mid B \text{ and } B \mid \text{not } B \mid (B) \mid E \text{ relop } E \mid \text{true} \mid \text{false}$$

实现布尔表达式有两种方法：将真和假数值化，这样布尔表达式的计算就类似于算术表达式的运算（完全计算）；另一种方法就是借助控制流，即用程序中的位置来表示布尔表达式的值，适用于短路计算的情况。

### 4.3.2 短路代码

在短路（跳转）代码中，布尔运算符 `&&`、`||` 和 `!` 被翻译成跳转指令。运算符本身不出现在代码中，布尔表达式的值是通过代码序列中的位置来表示的。

短路运算是指，当通过逻辑运算符的左操作数就已经可以判断此表达式的值时，代码就不会再进入右边的代码块了。此称为短路运算。在短路运算中，上面三个布尔运算符所起到的逻辑作用在中间代码块中由转移语句 `goto` 来实现。

例如语句

```
if(x < 100||x > 200&&x! = y)x = 0;
```

可以被翻译成如下所示的代码：

```
if x < 100 goto L2
ifFalse x > 100 goto L1
ifFalse x != y goto L1
L1: x = 0
L2:
```

在这个翻译中，如果程序的控制流到达 L2，就表示这个表达式为真；如果表达式为假，则程序控制流将跳过 L2 块（包含了赋值语句），直接转到 L1

## 第 4.4 节 控制流语句的翻译

将控制流语句翻译成其语义规则 (SDT) 分为两部分：对使用的各标签的定义；使用标签完成对产生式左值代码段的分段跳转式构造。这里对后者稍作解释：就像一个汇编文件一样，控制流语句会将代码进行分块处理，各个块之间会有跳转的逻辑联系。这些逻辑联系由继承属性 `bb.next` 或者强制跳转语句 `goto` 来定义，而强制跳转语句将被显式地写入到对 `S.code` 的构造当中

先来说两个龙书上面的简单控制流语句及其语义规则：

$$P \rightarrow S \text{ 和 } S \rightarrow \mathbf{assign}$$

前者的语义规则为

$$S.next = newlabel()$$

$$P.code = S.code \parallel label(S.next)$$

而后者的语义规则为

$$S.code = \mathbf{assign}.code$$

控制流语句（if-then、if-then-else、while-do、顺序执行）由下面的文法产生（B 是布尔表达式）：

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

这四种语句对应的代码栈如下图所示，这几个图所示的三地址代码的结构可以帮助理解 next 的跳转。需要注意的是， $S.next$  并没有被显式地画出来，而是作为（已知的）整个语句的出口，在后面的代码生成语法制导定义中被赋值给某些（语句块的）出口

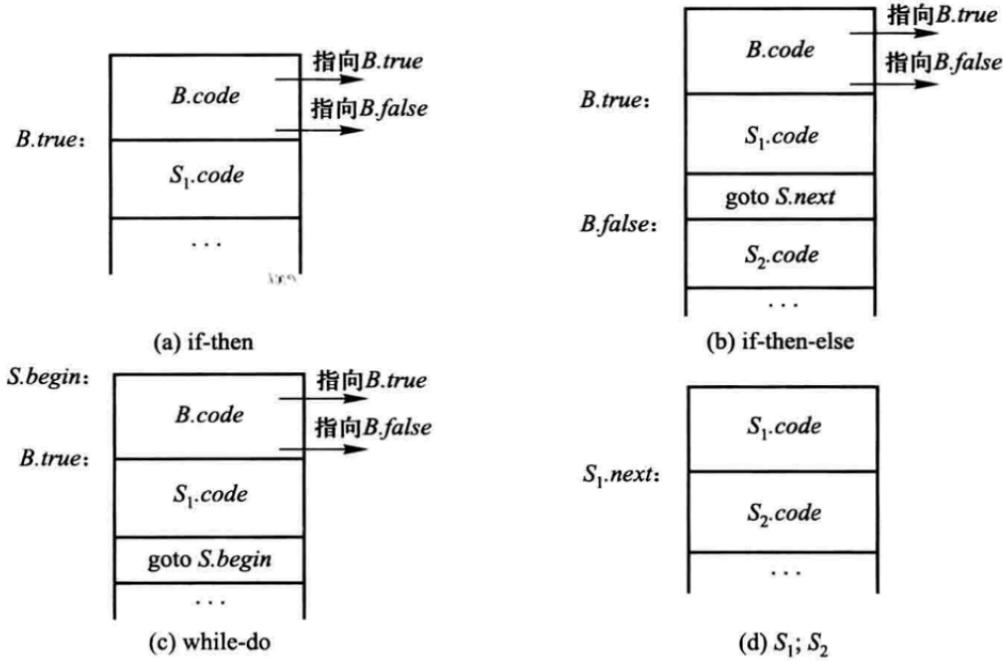


图 4.1: if-then, if-then-else, while-do 和顺序语句的代码（栈）

#### 4.4.1 if 语句中间代码生成的 SDD

姑且命名语句  $S \rightarrow \text{if } B \text{ then } S_1$  为产生式  $\gamma$ 。

这个“产生式”的每一个“非终结符”( $S, B, S_1$ )都是一段代码，即他们的综合属性  $S.code, etc.$ 。每一个代码段都要有一个或多个“出口”，也就是说在执行结束这一段代码之后程序应当去往何方。这些“出口”用“非终结符”的继承属性来表示。可以调用 `newlabel()` 来产生新标号，新标号是跳转到这个产生式内部的时候，用（实际上是另一个非终结符的 `.begin` 的位置的）一个内部入口，而不是（将整个产生式封装起来看，放在“左值”的 `.next` 属性中的）一个总的出口，来对这个 `.next` 属性赋值。即：

1. 标号指向内部的三地址代码时需要调用 `newlabel()`
2. 标号指向外部的三地址代码时从  $S$  继承

语义规则：

$$B.true = \text{newlabel}()$$

$$B.false = S.next$$

$$S_1.next = S.next$$

$$S.code = B.code \parallel \text{gen}(B.true, ':') \parallel S_1.code$$

#### 4.4.2 带有 else 的语句的中间代码生成 SDD

考虑带有 else 的语句  $S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$ 。

对布尔表达式 B，用两个标号  $B.\text{true}$  和  $B.\text{false}$  分别表示 B 为真和为假时控制流应该转向的标号，这两个属性由 B 的上下文决定。注意在  $S_1.\text{code}$  的后面有 'goto'， $S.\text{next}$ ，这条指令是需要的，因为  $S_1$  执行结束意味着 S 执行结束，并且当  $S_1$  是赋值语句的时候肯定会执行这条指令。

(我觉得这块紫书说的不如龙书清楚…下面是龙书上的表述) 由于布尔表达式中有一些向外跳转的指令，他们在 B 为真时跳转到  $S_1$  代码的第一条指令；B 为假时跳转到  $S_2$  的代码的第一条指令。然后，控制流从  $S_1$  或者  $S_2$  跳转到紧跟在 S 的代码之后的三地址指令——该指令的标号由继承属性  $S.\text{next}$  指定。在  $S_1$  的代码之后有一条  $\text{goto } S.\text{next}$  指令，使得控制流越过  $S_2$  的代码。 $S_2$  的代码之后不需要 goto 语句，因为  $S_2.\text{next}$  就是  $S.\text{next}$ 。语义规则如下：

```

 $B.\text{true} = \text{newlabel}()$ 
 $B.\text{false} = \text{newlabel}()$ 
 $S_1.\text{next} = S.\text{next}$ 
 $S_2.\text{next} = S.\text{next}$ 
 $S.\text{code} = B.\text{code} \parallel \text{label}(B.\text{true}) \parallel S_1.\text{code} \parallel \text{gen('goto' } S.\text{next}) \parallel \text{label}(B.\text{false}) \parallel S_2.\text{code}$ 

```

#### 4.4.3 while 语句的中间代码生成 SDD

while 语句从逻辑上像是纯 if 语句的改版。while 同样由  $B.\text{code}$  和  $S_1.\text{code}$  组成。由于循环的需要，在这个 while 语句的第一条指令上附加一个标号 begin。与 if-else 中的处理类似，在  $S_1$  的末尾需要一条 goto 语句来重新回到 begin (总之记住只要不是顺序执行的就要加 goto 去跳转。这块待议，为什么从  $B.\text{code}$  跳转到  $B.\text{false}$  就不需要加 goto 呢？)。语义规则如下：

```

 $\text{begin} = \text{newlabel}()$ 
 $B.\text{true} = \text{newlabel}()$ 
 $B.\text{false} = S.\text{next}$ 
 $S_1.\text{next} = \text{begin}$ 
 $S.\text{code} = \text{label(begin)} \parallel B.\text{code} \parallel \text{label}(B.\text{true}) \parallel S_1.\text{code} \parallel \text{gen('goto' begin)}$ 

```

#### 4.4.4 顺序结构中间代码生成 SDD

主要思想是“为每一语句  $S_1$  引入其后的下一条语句的标号”。语义规则如下：

```

 $S_1.\text{next} = \text{newlabel}()$ 
 $S_2.\text{next} = S.\text{next}$ 
 $S.\text{code} = S_1.\text{code} \parallel \text{label}(S_1.\text{next}) \parallel S_2.\text{code}$ 

```

## 第 4.5 节 布尔表达式的控制流翻译

针对布尔表达式的语义规则是对于上一节中语句的语义规则的一个补充, *i.e.* 上面的中间代码中的 B 部分即为这一节中要讨论的布尔表达式。一个布尔表达式 B 被翻译成一个一组三地址指令, 它将使用条件或无条件跳转指令来对 B 求值。这些跳转指令对目标是两个标号之一: 当 B 为真时是 *B.true*, 反之 *B.false*

**用控制流来实现计算** 原则有两个: ① 布尔运算符 and, or, not 不出现在翻译后的代码中 ② 用程序中的位置来表示值

### 4.5.1 或运算

对于产生式  $B \rightarrow B_1 \parallel B_2$ , 我们现在用短路计算来进行考虑。如果  $B_1$  为真, 那么我们立刻知道 B 本身也为真, 而不需要再考虑  $B_2$ , 因此  $B_1.true$  和  $B.true$  相同。如果  $B_1$  为假, 那么需要对  $B_2$  求值, 因此我们将  $B_1.false$  设置成  $B_2$  的代码的第一条指令的标号。这时相当于将表达式 B 的真假决定权交给了  $B_2$ , 于是  $B_2$  的真假出口分别等于 B 的真假出口。语义规则如下:

$$\begin{aligned} B_1.true &= B.true \\ B_1.false &= newlabel() \\ B_2.true &= B.true \\ B_2.false &= B.false \\ B.code &= B_1.code \parallel label(B_1.false) \parallel B_2.code \end{aligned}$$

### 4.5.2 且运算

产生式  $B \rightarrow B_1 \&\& B_2$  的翻译方法类似与  $B \rightarrow B_1 \parallel B_2$ 。如果  $B_1$  为假, 则 B 为假; 反之考虑  $B_2$  的真值。其语义规则如下:

$$\begin{aligned} B_1.true &= newlabel() \\ B_1.false &= B.false \\ B_2.true &= B.true \\ B_2.false &= B.false \\ B.code &= B_1.code \parallel label(B_1.true) \parallel B_2.code \end{aligned}$$

### 4.5.3 非运算

由于非运算仅有一个操作数, 故不需要为  $B \rightarrow !B_1$  产生新的代码, 只需要将 B 中的真假出口对换, 就可以得到  $B_1$  的真假出口。由于没有短路运算, 并不需要进行 B 内部的跳转, 也不

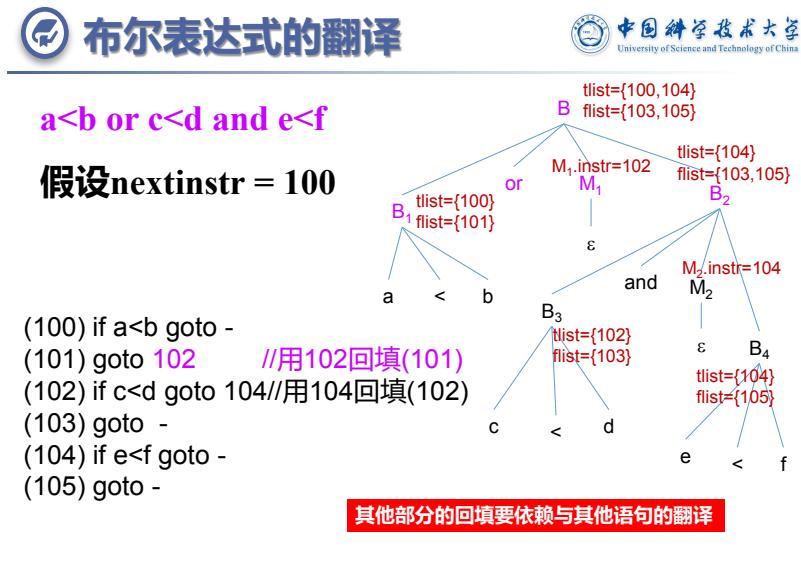
需要引入新的标记。其语义规则如下：

$$\begin{aligned}B_1.\text{true} &= B.\text{false} \\B_1.\text{false} &= B.\text{true} \\B.\text{code} &= B_1.\text{code}\end{aligned}$$

#### 4.5.4 关系运算 (relop)

产生式  $B \rightarrow E_1 \text{ relop } E_2$ , 其中 E 就是一个表达式, 具有属性  $addr$  和  $code$

### 第 4.6 节 回填



2020/11/23

Cheng @ Compiler Fall 2020, USTC

100

图 4.2: 标号回填示例

为布尔表达式和控制流语句生成目标代码时, 关键问题之一是将一个跳转指令和该指令的目标匹配起来。以表达式  $\text{if}(B)S$  为例, 在生成  $goto B.\text{false}$  时, 并不知道跳过 S 应该跳到哪里 (因为  $S.\text{code}$  还没有加进来呢)。在前面我们的处理方法是将标号作为继承属性传递到生成相关跳转指令的地方 (就是上面的  $B.\text{false}$ )。但是, 这样的做法要求再进行一遍处理, 将标号和具体地址绑定起来。这两趟分析分别是生成语法树, 和深度优先遍历树并计算属性值。

回填用于解决布尔表达式短路计算翻译中, 产生了转移目标不明确的条件或无条件代码的问题。解决方案是当生成跳转指令时, 暂时不指定目标地址, 当有关目标地址确定后, 再填回到翻译代码中。在具体实现时, 可以将有相同转移目标的转移代码的编号串起来形成链, 可以方便回填目标地址。该 list 变成了综合属性, 可以与 LR 结合 (注: 后面的翻译均是与 LR 结合的语法制导翻译方案)

对于各层布尔表达式  $B$ , 维护了两个综合属性:  $B.\text{truelist}, B.\text{falselist}$ 。 $\text{truelist}$  是一个包含跳转或条件跳转指令的列表, 向其中插入  $B$  为真时控制流应当转向的标号。在生成  $B$  的代码时, 跳转到真或假出口的跳转指令时不完整的, 标号字段尚未填写。这些不完整的跳转指令被保存在两个 list 中。

具体过程是, 首先在生成产生式左值 `code` 的过程 (就是前面说的那个 `code` 的式子, 添加指令和组合指令) 中对各指令 (`code`) 进行标号, 在这个过程中 `goto` 的标号暂时空着不填。然后在代码段的中间部分插入标记点。都写完之后回填标记点的标号。

# 第 5 章 类型检查

本节由龙书 6.5 节和紫书第五章构成。

静态的语义分析类型检查，利用逻辑规则分析运算分量的类型与运算符预期是否匹配。主要包括三个层次：① 形式化描述类型结构（类型表达式）② 判定两个类型相同的依据（类型等价：结构等价和名字等价）③ 定义一组逻辑规则检查语句或者表达式中是否存在逻辑错误（语法制导翻译方案实现、函数和算符的重载）

## 第 5.1 节 类型表达式

类型可以是语法的一部分，因此也是结构的。

基本类型是类型表达式：`integer`、`real`、`char`、`boolean`、`type_error`（出错类型，用于在类型检查中传递错误）、`void`（无类型，是语句的类型）。可以为类型表达式命名，类（class）名也是类型表达式。

将类型构造算子作用域类型表达式可以得到新的类型表达式（如 `array`，如果  $T$  是类型表达式， $N$  是一个整数，则  $\text{array}(N, T)$  是类型表达式），（又如指针类型构造算子 `pointer`，如果  $T$  是类型表达式，则  $\text{pointer}(T)$  是类型表达式），（再如笛卡尔乘积类型构造算子  $\times$ ，如果  $T_1$  和  $T_2$  是类型表达式，则  $T_1 \times T_2$  也是类型表达式。主要用于描述列表和元组，如表示函数的参数），（还有函数类型构造算子  $\rightarrow$ ，若  $T_1, T_2, \dots, T_n$  和  $R$  是类型表达式，则  $T_1 \times T_2 \times \dots \times T_n \rightarrow R$  也是），（以及记录类型构造算子 `record`。若有标识符  $N_1, N_2, \dots, N_n$  以及对应的类型表达式  $T_1, T_2, \dots, T_n$ ，则  $\text{record}((N_1 \times T_1) \times (N_2 \times T_2) \times \dots \times (N_n \times T_n))$  也是类型表达式

### 5.1.1 自动生成类型表达式

即构造类型表达式的 SDD。

首先需要为每个文法符号设置综合属性  $t$  和继承属性  $b$ 。其中  $t$  是该符号对应的类型表达式，而  $b$  的作用是将类型信息沿着语法树从左向右传递

### 5.1.2 构造类型表达式的 SDT

语法制导翻译方案就是将对属性值的动作（code）嵌入到表达式中，见（3.2）。不过在这个过程中会遇到一些问题，比如继承属性的计算就与 LR 分析不匹配，因此，如果使用 LR，就需要改造文法。与前面对 `goto` 语句的处理类似，这里也采取了标号方法

```

typedef struct {
    int    a;
    float b;
    char   c [10];
} example;

```

*example 的类型表达式：*

$$\begin{aligned}
&\text{record } ((a \times \text{int}) \times (b \times \text{float})) \times \\
&((c \times (\text{array } (10, \text{char}))) )
\end{aligned}$$

图 5.1: 记录类型构造算子

## 第 5.2 节 类型等价

**结构等价** 如下规则：① 两个类型表达式完全相同（当无类型名时）：类型表达式树一样，相同的类型构造符作用于相同的子表达式。② 当有类型名时，用他们所定义的类型表达式代换他们，所得的表达式完全相同（类型定义无环时）

**名等价** 把每一个类型名看成一个可区别的类型，两个类型表达式名字等价当且仅当：他们是相同的基本类型、不进行名字代换就能结构等价

C 语言对除记录（结构体）以外的所有类型使用结构等价，而记录类型用的是名字等价，以避免类型图中的环。下面一个例子：

## 第 5.3 节 类型检查

 例题

中国科学技术大学  
University of Science and Technology of China

在X86/Linux机器上，编译器报告最后一行有错误：

**incompatible types in return**

```
typedef int A1[10];           | A2 *fun1() {      允许 只要不是记录不同结构
typedef int A2[10];           |   return(&a);    体的，都是结构等价的
A1 a;                         | }
typedef struct {int i;}S1;     | S2 fun2() {      不允许 定义的是两个struct,
typedef struct {int i;}S2;     |   return(s);    不允许记录为结构等价
S1 s;                          | }
```

在C语言中，数组和结构体都是构造类型，为什么上面第2个函数有类型错误，而第1个函数却没有？

11/23/2020 Cheng @ Compiler Fall 2020, USTC 54

图 5.2: 结构体是名等价

# 第6章 与类型相关的中间代码生成

## 第6.1节 符号表

符号表的使用和修改贯穿了编译的全过程，它存储了各种信息，如变量名、函数名、对象、类、interface、类型信息、占用空间、作用域等。它用于编译过程中的分析与合成：①语义分析：如使用前声明检查、类型检查、确定作用域等 ②合成：如类型表达式构造、内存空间分配等

## 第6.2节 声明语句的翻译

声明语句翻译的要点有：

1. 分配存储单元：名字、类型、字宽、偏移
2. 作用域的管理：过程调用
3. 记录类型的管理
4. 不产生中间代码指令，但是要更新符号表

如下例： $T \rightarrow integer \mid real \mid array[num]of T_1 \uparrow T_1$ ，T 可以具有 type（变量所具有的类型）、width（该类型数据所占的字节数）、offset（变量的存储偏移地址）

## 第6.3节 作用域的管理

管理作用域（过程嵌套声明）：每个过程中声明的符号要置于该过程的符号表中；方便地找到子过程和父过程对应的符号。

符号表的特点及数据结构：

1. 各过程有各自的符号表：哈希表
2. 符号表之间有双向链：父  $\rightarrow$  子：过程中包含哪些子过程定义；子  $\rightarrow$  父：分析完子过程后继续分析父过程
3. 维护符号表栈 (tblptr) 和地址偏移量栈 (offset)：保存尚未完成的过程的符号表指针和相对地址