

# 编译原理笔记

上官凝

2020 年 11 月 26 日

# 目录

<b>1 词法分析</b>	<b>3</b>
1.1 词法记号及属性、描述和识别 . . . . .	3
1.1.1 词法记号和属性 . . . . .	3
1.1.2 串和语言 . . . . .	3
1.1.3 正则表达式 . . . . .	4
1.1.4 状态转换图 . . . . .	4
1.1.5 关于 FLEX 的正则表达式 . . . . .	5
1.2 有限自动机 . . . . .	5
1.2.1 NFA 和 DFA . . . . .	5
1.2.2 正则表达式到 NFA . . . . .	5
1.2.3 NFA 到 DFA . . . . .	6
1.2.4 DFA 的化简 . . . . .	6
<b>2 语法分析</b>	<b>7</b>
2.1 上下文无关文法 . . . . .	7
2.1.1 CFG 推导 . . . . .	7
2.1.2 文法的二义性 . . . . .	7
2.1.3 悬空 else 文法 . . . . .	8
2.1.4 左递归 . . . . .	8
2.2 自顶向下分析方法 . . . . .	8
2.2.1 递归下降分析方法 . . . . .	8
2.2.2 消除左递归 . . . . .	9
2.2.3 有左因子的文法 . . . . .	9
2.2.4 复杂的回溯代价太高 . . . . .	9
2.3 预测分析法 . . . . .	10
2.3.1 “第一个”集合和“跟随”集合 . . . . .	10
2.3.2 LL(1) 文法 . . . . .	10
2.4 递归下降的预测分析 . . . . .	11
2.4.1 预测分析表 . . . . .	11
2.5 自底向上分析方法 . . . . .	12

2.5.1 移进 (shift)-归约 (reduce) 分析技术 . . . . .	12
2.6 LR(k) 分析技术 . . . . .	13
2.7 SLR . . . . .	13
2.7.1 产生规则 . . . . .	13
2.7.2 有效项目 . . . . .	14
2.7.3 SLR 分析表 . . . . .	14
2.7.4 SLR(1) 文法 . . . . .	15
2.7.5 判定满足 SLR 文法输入串 . . . . .	15
2.8 规范的 LR 分析方法 . . . . .	15
2.8.1 构造 LR(1) 项目集规范族 . . . . .	15
2.8.2 构造规范的 LR 分析表 . . . . .	16
2.8.3 LALR . . . . .	16
<b>3 语法制导的翻译 . . . . .</b>	<b>17</b>
3.1 语法制导的定义 . . . . .	17
3.1.1 综合属性和继承属性 . . . . .	17
3.1.2 注释分析树和属性依赖图 . . . . .	18
3.2 S 属性定义的自下而上计算 . . . . .	18
3.2.1 语法树 . . . . .	19
3.2.2 S 属性的自下而上计算 . . . . .	19
3.3 L 属性定义的自上而下计算 . . . . .	19
3.3.1 翻译方案 . . . . .	19
<b>4 中间代码生成 . . . . .</b>	<b>21</b>
4.1 中间语言 . . . . .	21
4.2 布尔表达式和控制流语句 . . . . .	21
4.2.1 布尔表达式 . . . . .	21
4.2.2 控制流语句的翻译 . . . . .	22
4.2.3 if 语句中间代码生成的 SDD . . . . .	22
4.2.4 带有 else 的语句的中间代码生成 . . . . .	23

# 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）

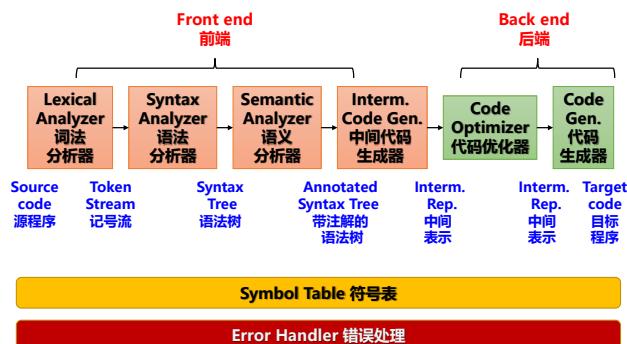


图 1.1: 编译器总览

### 1.1.3 正则表达式

约定：

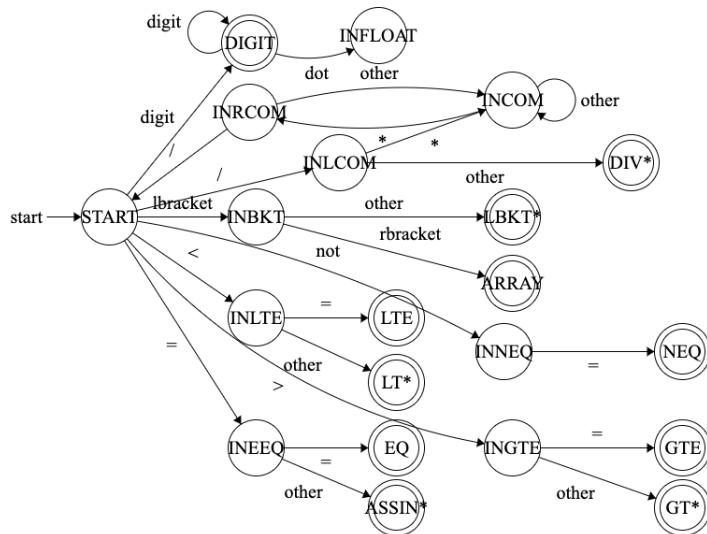
1. 闭包运算 ( $*$ ) 有最高的优先级并且是左结合的运算
2. 连接运算（两个正规式并列）的优先级次之且也是左结合的运算
3. 选择运算 ( $|$ ) 的优先级最低并且也是左结合的运算
4. 零个或一个实例 ( $r?$ ) 表示  $r | \epsilon$
5. 字符组 ( $[abc]$ ) 表示  $a | b | c$
6. 缩写字符组 ( $[a - z]$ ) 表示  $a | b | \dots | z$

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：

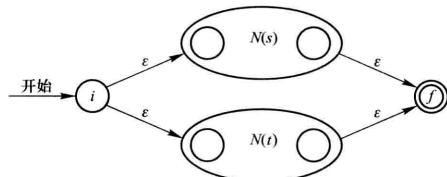


图 2.16 识别正规式  $s|t$  的 NFA

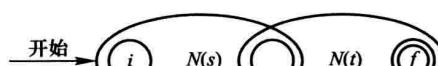
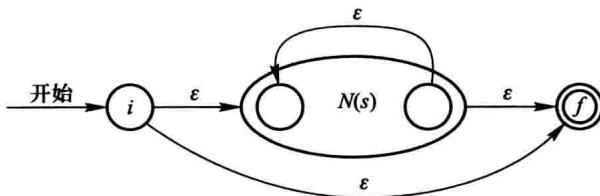


图 2.17 识别正规式  $st$  的 NFA

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：



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

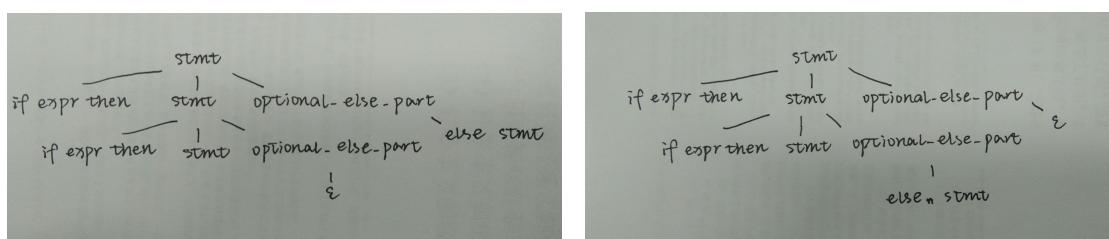
$$A \rightarrow \alpha A'$$

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

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

- 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)
- 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:

- 移进-归约冲突 (如何决策选择移进 (构造句柄) 还是归约)
- 进行归约时, 确定右句型中将要归约的子串 (识别句柄)
- 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  $\beta_2 \neq \epsilon$  则应当移进（继续读取才能规约为 A）；若  $\beta_2 = \epsilon$ ，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  $\epsilon$ ），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

## 第 3.2 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。

### 3.2.1 语法树

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）

### 3.2.2 S 属性的自下而上计算

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。

扩展 LR 的语法分析栈：

1. 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
2. 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）
3. 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）。

## 第 3.3 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。

### 3.3.1 翻译方案

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:

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, 考虑次序, 避免形成环

# 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。

## 第 4.1 节 中间语言

后缀表示

图形表示（有向无环图，即 DAG）

**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 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

### 4.2.1 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；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。

### 4.2.2 控制流语句的翻译

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

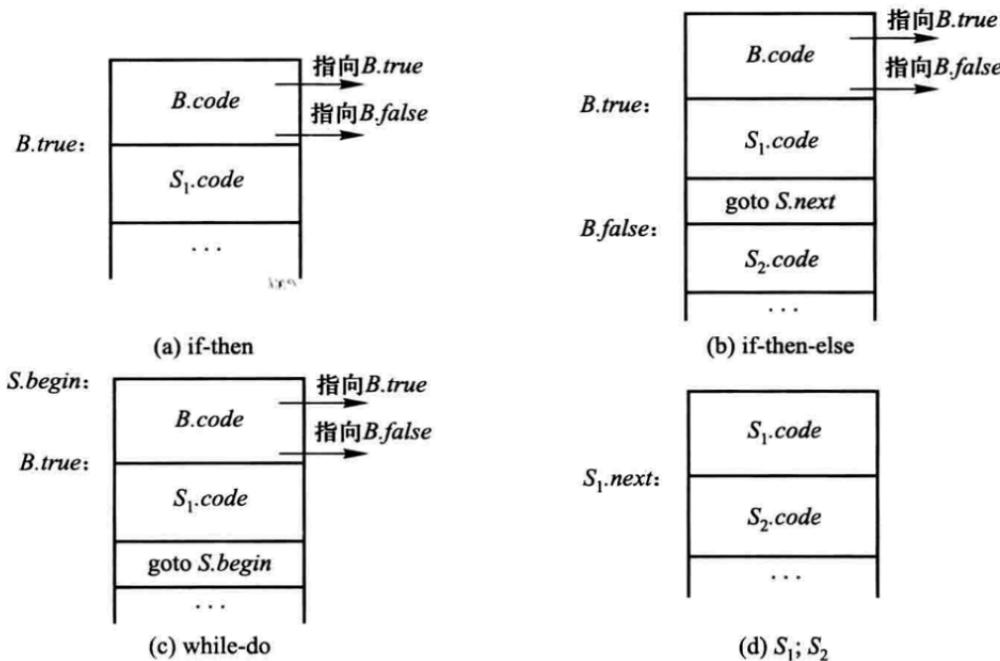


图 4.1: if-then, if-then-else, while-do 和顺序语句的代码（栈）

### 4.2.3 if 语句中间代码生成的 SDD

姑且命名语句  $S \rightarrow \mathbf{if} \ B \ \mathbf{then} \ S_1$  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：

$$\begin{aligned} B.\text{true} &= \text{newlabel}() \\ B.\text{false} &= S.\text{next} \\ S_1.\text{next} &= S.\text{next} \\ S.\text{code} &= B.\text{code} \parallel \text{gen}(B.\text{true}, ':') \parallel S_1.\text{code} \end{aligned}$$

#### 4.2.4 带有 `else` 的语句的中间代码生成

考虑带来 `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  $S_1.\text{code}$  的后面由 `goto S.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