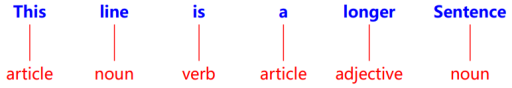
编译原理与技术

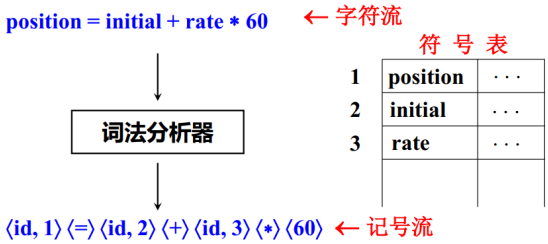
第一周 9.14

词法分析

- 人类在理解自然语言时，首先要识文断字



- 通常称为线性分析（linear analysis）
- 将程序字符流分解为记号（Token）序列
  - 形式：< token_name, attribute_value >
  - 记号名是同类词法单元共用的名称，而属性值是一个词法单元有别与同类中其他词法单元的特征值



1. 标识符position的记号是<**id**, 1>, 其中**id**是标识符的总称，1代表position在符号表中的条目

> 符号表的条目用来存放标识符的各种属性，如它的名字和类型

2. 赋值号 = 形成的记号是<**assign**>，因为该记号只有一个实例，因此不需要以属性值来区分实例

> 为了直观起见，可用< = >作为赋值号的记号名

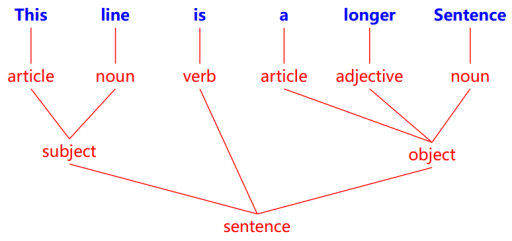
3. 其余标识符和运算符的记号如上图所示

> 这里出现了常数60，一般来说，程序中出现的常数也要放到符号表或单独的常数表中，形成记号<**number**，60在表中的条目>

4. 分隔单词的空格通常在词法分析时被删去

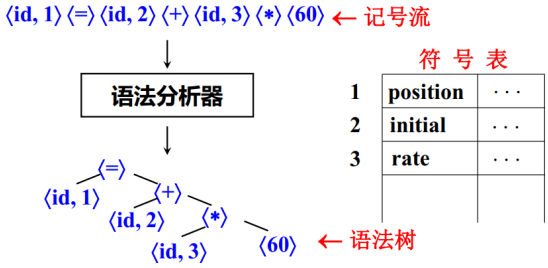
语法分析

- 人类在理解自然语言时，其次要理解句子结构



- 也称为层次分析（Hierachical analysis）

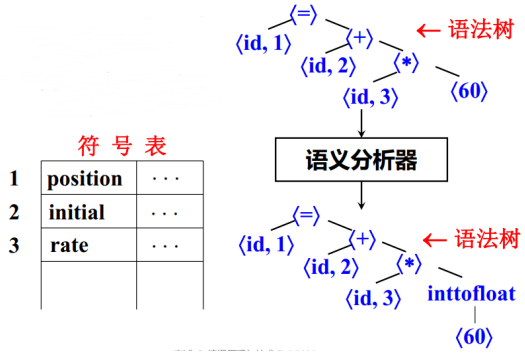
- 也称为解析（Parsing），它检查词法分析输出的记号流是否符合编程语言的语法规则，并在词法记号的基础上创建语法结构



- 语法树
  - 内部节点表示运算
  - 子节点表示运算的运算对象

语义分析

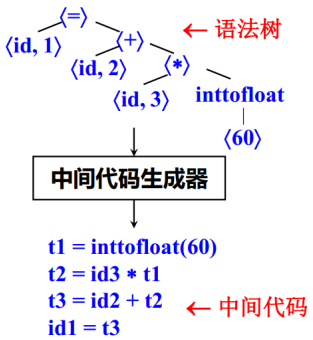
- 人类在理解自然语言时，最后要理解句子的含义
- 编译器会检查程序中的不一致
  - 如：类型检查（type checking）



上图分析后会发现，所有的变量都是实型，而60是整型。进行类型检查后发现*作用于实型变量rate和整数60，因此建立了一个额外的算符结点inttofloat，显式地把整数转变为实数。

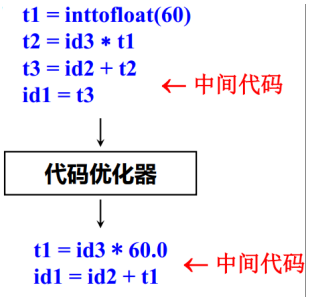
中间代码生成

- 是源语言与目标语言之间的桥梁



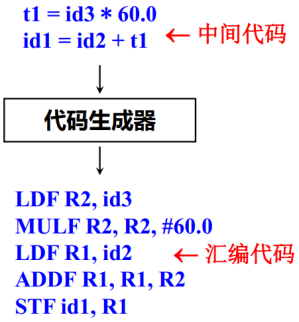
代码优化

- 机器无关的代码优化便于生成执行时间更快、更短或能耗更低的目标代码



代码生成

- 如果目标语言是机器代码，必须为变量选择寄存器或内存位置



> 每条指令的“F”告知指令处理浮点数

符号表管理

- 符号表
  - 为每个变量名字保存一个记录的数据结构
  - 记录的域是该名字的属性

阶段的分组

- 前端：词法分析、语法分析、语义分析、中间代码生成
- 后端：代码优化、代码生成
- 实际的编译器中，源于几个阶段的活动可以组合在一起，各阶段之间也无须显式构造

解释器

- 是不同于编译器的另一类语言处理器
- 不像编译器那样通过翻译生成目标程序，而是直接执行源程序所指定的运算
- 也需要对源程序词法分析、语法分析和语义分析
- 解释器的效率一般较低，解释执行需要找到一种适合解释器的中间语言以缩短反复分析源程序需要的时间

第一周 9.17

词法分析

- 程序示例：

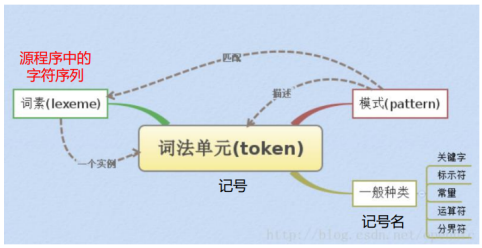
```
if(i == j)
    printf("equal!");
else
    num5 = 1;
```

- 程序是以字符串的形式传递给编译器的

> \tif (i == j)\n\t\tprintf(“equal!”);\n\telse\n\t\tnum5 = 1;

- 目的：将输入字符串识别为有意义的子串
  - 子串的种类（Name）
  - 可帮助解释和理解该子串的属性（Attribute）
  - 可描述具有相同特征的子串的模式（Pattern）

三个关键术语



- 词法记号
  - 简称记号，是由记号名和属性值构成的二元组
  - 属性值不是必需项
  - 记号名是代表一类同法单元的抽象名字
  - 经常直接用记号名来引用记号
- 模式
  - 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
  - 词法单元（ lexeme ），又称单词，是源程序中匹配一个记号模式的字符序列

词法单元(记号)、实例与模式

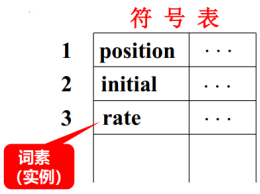
```
if (i == j) printf(“equal!”);
else num5 = 1;
```

记号名	实例（词素）	模式的非形式描述
if	if	字符i, f
else	else	字符e, l, s, e
relation	==, < , <= , ...	== 或 < 或 <= 或 ...
id	i, j, num5	由字母开头的字母数字串
number	1, 3.1, 10, 2.8 E12	任何数值常数
literal	“equal!”	引号“和” 之间任意不含引号本身的字符串

词法单元

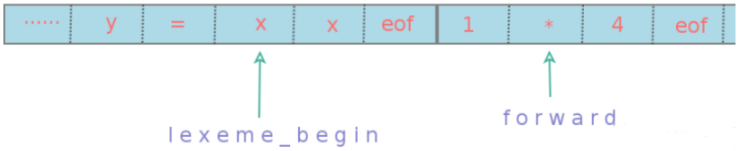
- 由一个记号名和一个可选的属性值组成
  - token := <token_name, attribute_value>
- 属性记录词法单元的附加属性，影响语法分析对该词法单元的翻译
  - 例：标识符id的属性会包括词法单元实例、类型、第一次出现的位置等

    - 保存在符号表（Symbol table）中，以便编译的各个阶段取用
    - 可以为空（optional）
- 示例：
    - position = initial + rate * 60的记号和属性值：
    - <id， 指向符号表中position条目的指针>
    - < assign _ op>
    - <id， 指向符号表中initial条目的指针 >
    - <add_op >
    - <id， 指向符号表中rate条目的指针 >
    - <mul_op >
    - <number， 整数值60>



Lookahead方法

- 词法分析
    - 从左到右读取输入串，每次识别出一个token实例
    - 可能需要“lookahead”来判断当前是否是一个token实例的结尾、下一个实例的开始



自动化的挑战

- **可否用空格分词?**
    - a>b, if(expression)
    - 不能，需要用扫描匹配或者回溯的方法
- **如何应对任意的词素序列?**
    - int a, int aaa, int aaaa, int aaaaaaa
    - 虽然程序可以任意，但是记号的类型数量可控，为每一种类型设计匹配模式
    - 利用模式，进行模式匹配
- **可否处理可能出现的歧义?**
    - i, f vs. if, = vs. ==
    - 最长匹配原则 + 关键字保留原则

词法分析器的自动生成

词法单元的描述： 正则式

串和语言

- 术语
    - 字母表： 符号的有限集合，例： ∑= { 0, 1}
    - 串：符号的有穷序列，例： 0110, ε
    - 语言： 字母表上的一个串集

{ε, 0, 00, 000, ...}, {ε}, ∅

- 句子： 属于语言的串

> 注意区别：ε，{ε}， ∅

- 串的运算
  - 连接（积）： xy， sε = εs = s
  - 指数（幂）： s^0为ε， s^i为$s^{i-1}s$（i > 0）
- 语言的运算
  - 并：L ∪ M = {s | s ∈L 或 s ∈ M }
  - 连接：LM = {st | s ∈ L 且 t ∈ M}
  - 幂：L^0是{ε}， L^i是$L^{i-1}L$
  - 闭包：L^* = L^0 ∪ L^1 ∪ L^2 ∪ ...
  - 正闭包：L^+ = L^1 ∪ L^2 ∪ ...

> 优先级：幂 > 连接 > 并

- 示例
  - L: { A, B, ..., Z, a, b, ..., z }, D: { 0, 1, ..., 9 }
  - L ∪ D, LD, L^6, L^*, L(L ∪ D)*, D^+

正则表达式（Regular Expr）

- ∑ = {a, b}
  - a | b {a, b}
  - (a | b) (a | b) {aa, ab, ba, bb}
  - aa | ab | ba | bb {aa, ab, ba, bb}
  - a^* 由字母a构成的所有串集
  - (a | b)* 由a和b构成的所有串集
- 复杂的例子
  - (00 | 11 | ((01 | 10) (00 | 11) * (01 | 10))) *
  - 句子： 01001101000010000010111001
- 正则式用来表示简单的语言

正则式	定义的语言	备注
ε	{ε}	
a	{a}	a ∈ ∑
(r)	L(r)	r是正则式
(r)|(s)	L(r)∪L(s)	r和s是正则式
(r)(s)	L(r)L(s)	r和s是正则式
(r)*	(L(r))*	r是正则式

> ((a) (b)*)|(c)可以写成ab*| c
>
> 优先级：闭包* > 连接 > 选择 |

正则定义

- bottom-up方法
  - 对于比较复杂的语言，为了构造简洁的正则式，可先构造简单的正则式，再将这些正则式组合起来，形成一个与该语言匹配的正则序列

    $d_1 \rightarrow r_1$

    $d_2 \rightarrow r_2$

    ...

    $d_n \rightarrow r_n$
  - 各个d_i的名字都不同， 是新符号， not in ∑
  - 每个r_i都是 ∑ ∪{d_1, d_2, ..., d_{i-1} }上的正则式

正则定义的例子

- C语言的标识符是字母、数字和下划线组成的串
  - letter_ → A | B | ... | Z | a | b | ... | z | _
  - digit → 0 | 1 | ... | 9
  - id → letter_ (letter_ |digit) *
- 无符号数集合，例1946,11.28,63E8,1.99E-6
  - digit → 0 | 1 | ... | 9 [0-9]
  - digits → digit digit*
  - optional_fraction → .digits|ε
  - optional_exponent → (E (+ | — | ε) digits) | ε
  - number→digits optional_fraction optional_exponent
  - — 简化表示
    - number → digit$^+$ (.digit$^+$)? (E[+-]? digit$^+$)?

> 注意区分：？ 和 *

- 考虑下面的正规定义：
  - **while** → while
  - **do** → do
  - **relop** → < | < = | = | < > | > | > =
  - **letter_** → [A-Za-z_]
  - **id** → letter_ (letter_ | digit)*
  - **number** → digit$^+$ (.digit$^+$)? (E[+-]? digit$^+$)?

  假定词法单元之间在必要时用空格（或制表符 、换行符）分开，词法分析器通过把剩余输人的前缀和下面的正规定义 **WS** 相比较来完成忽略词法单元之间的空白
  - **delim** → blank | tab | newline
  - **ws** → delim$^+$

问题：正则式是静态的定义，如何通过正则式动态识别输入串?

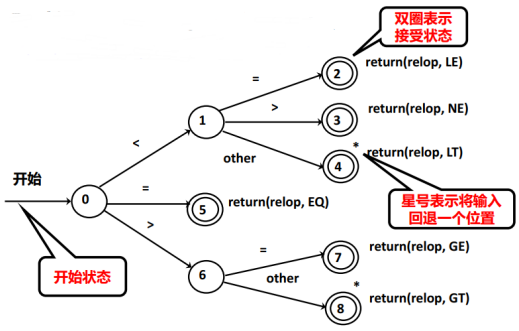
第二周 9.21

词法分析器的自动生成

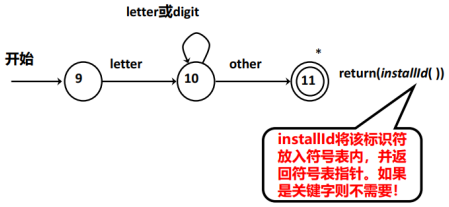
词法单元的识别：转换图

词法记号的识别：转换图

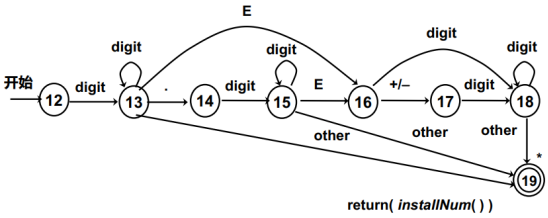
- 转换图(transition diagram)
  - relop → < | < = | = | < > | > | > =



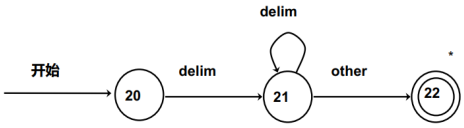
- 标识符和关键字的转换图
  - id → letter (letter | digit)*



- 无符号数的转换图
  - number → digit+ (.digit+)? (E[+-]? digit+)?

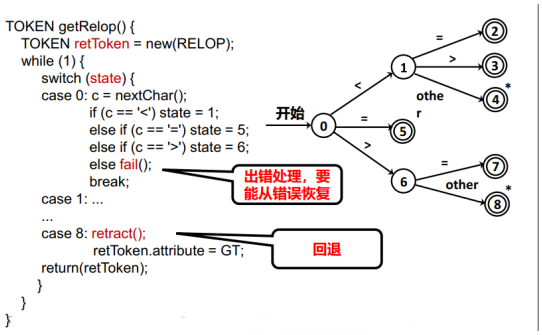


- 空白的转换图
  - delim → blank | tab | newline
  - ws → delim+



基于转换图的词法分析

- 例： relop的转换图的概要实现



词法分析中的冲突及解决

- R = Whitespace | Integer | Identifier|'+'
  - 识别 "foo+3"
    - "f" 匹配 R, 更精确地说是 Identifier
    - 但是 "fo"也匹配 R, "foo" 也匹配, 但 "foo+"不匹配
  - Maximal match 规则:
    - 如果 $x_1...x_i \in L(R)$ 并且 $x_1...x_k \in L(R)$
    - 选择匹配 R 的最长前缀
    - 最长匹配规则在实现时： lookahead, 不符合则回退
- R = Whitespace| 'new'| Integer| Identifier
  - 识别 "new foo"
    - "new" 匹配 R, 更精确地说是 'new'
    - 但是 "new"也匹配 Identifier
  - 优先 match 规则:
    - 如果 $x_1...x_i \in L(R_j)$ 并且 $x_1...x_i \in L(R_k)$
    - 选择先列出的模式 (j 如果 j < k)
    - 必须将 'new'列在 Identifier的前面

词法错误

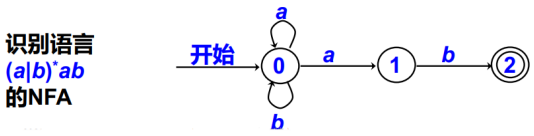
- 词法分析器对源程序采取非常局部的观点
  - 例：难以发现下面的错误

    fi (a == f (x)) ...
- 在实数是"数字串.数字串"格式下
  - 可以发现 123.x 中的错误
- 紧急方式的错误恢复
  - 删掉当前若干个字符，直至能读出正确的记号
  - 会给语法分析器带来混乱
- 错误修补
  - 进行增、删、替换和交换字符的尝试
  - 变换代价太高，不值得

有限自动机： NFA、 DFA

- **不确定的有限自动机（简称NFA）是一个数学模型，它包括:**
  - 有限的状态集合S
  - 输入符号集合∑
  - 转换函数move : $S \times (\sum \cup \{\varepsilon\}) \rightarrow P(S)$ (幂集)
  - 状态s_0是唯一的开始状态

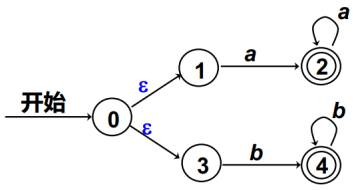
- F ⊆ S是接受状态集合



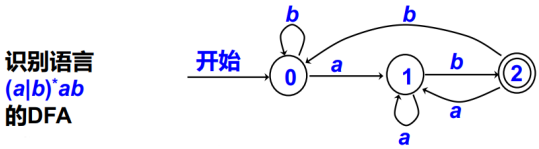
- NFA的转换表

状态	输入符号	
	a	b
0	{0, 1}	{0}
1	∅	{2}
2	∅	∅

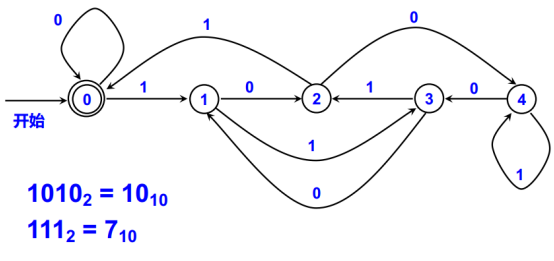
- 例 识别 $aa^*|bb^*$的NFA



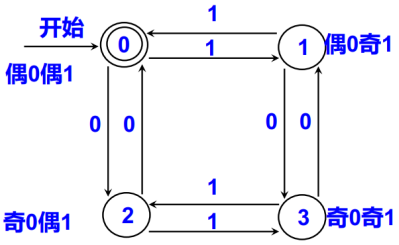
- 利用NFA识别token的问题
  - 转换函数move：$S \times (\sum \cup \{\varepsilon\}) \rightarrow P(S)$
  - 对于一个token，
    - 有可能要尝试很多不同的路径，
    - 大部分路径都是白费功夫
    - 尝试+回退的方式 => 效率很低
    - 考虑很多project，百万行代码+
  - 思考： 有没有一种确定的形式化描述，对于输入的一个符号，只有唯一的跳转？
- 确定的有限自动机（简称DFA)也是一个数学模型，包括：
  - 有限的状态集合S
  - 输入符号集合∑
  - 转换函数move : $S \times \sum \rightarrow S$，且可以是部分函数
  - 状态s_0是唯一的开始状态
  - F ⊆ S是接受状态集合



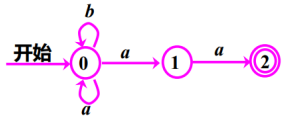
- 例 DFA，识别{0,1}上能被5整除的二进制数



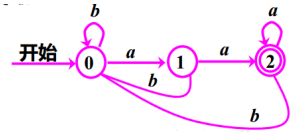
- 例DFA,接受 0和1的个数都是偶数的字符串



- NFA vs. DFA
  - NFA和DFA识别同一组语言（常规语言）
  - 主要区别：
    - 转换函数
      - $S \times (\sum \cup \{\varepsilon\}) \rightarrow P(S)$ NFA
      - $S \times \sum \rightarrow S$ DFA
    - DFA不接受ε作为输入
  - DFA执行速度更快
    - 没有多余的选择来考虑
  - 对于一个给定的词法判断，NFA比DFA更简洁

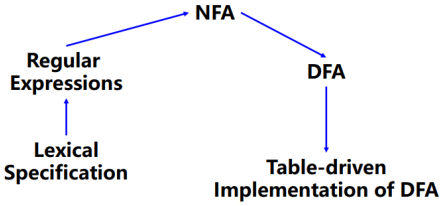


  - 而DFA与NFA的复杂度相比是指数级的



正则表达式→NFA

- 正则表达式 = Specification
- 有限自动机 = Implementation



- 二者之间的转换：
  - 用语法制导的算法，它用正则表达式的语法结构来指导有限自动机的构造过程

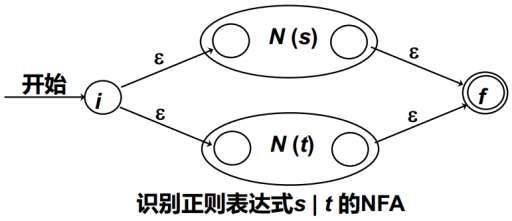
语法制导的构造算法

- 首先构造识别ε 和字母表中一个符号a的NFA
  - 重要特点：仅一个接受状态，它没有向外的转换

识别正则表达式ε的NFA　　识别正则表达式a的NFA

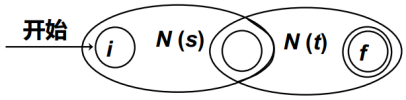
- 对于加括号的正则表达式(s)，其NFA可用s的NFA（用N(s)表示）代替

- 构造识别主算符为选择的正则表达式的NFA
  - 重要特点：仅一个接受状态，它没有向外的转换



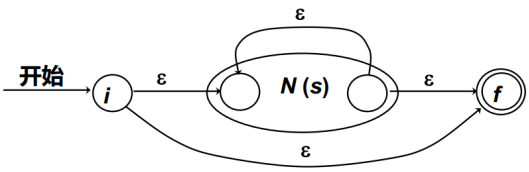
识别正则表达式$s \mid t$ 的NFA

- 构造识别主算符为连接的正则表达式的NFA
  - 重要特点：仅一个接受状态，它没有向外的转换



识别正则表达式st 的NFA

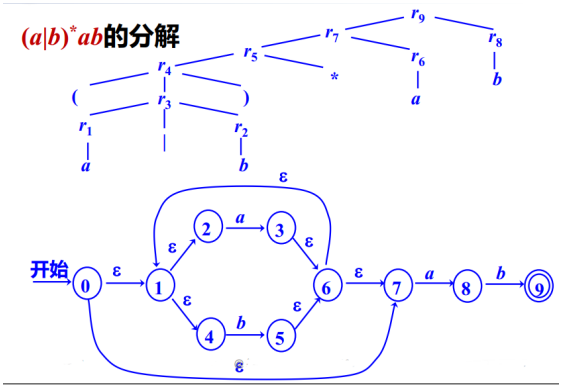
- 构造识别主算符为闭包的正则表达式的NFA
  - 重要特点：仅一个接受状态，它没有向外的转换



- 由本方法产生的NFA具有下列性质：
  - N(r)的状态数最多是r中符号和算符总数的两倍
  - N(r)只有一个接受状态，接受状态没有向外的转换
  - N(r)的每个状态有(1)一个其标号为∑中符号的指向其它状态的转换， 或者(2)最多两个指向其它状态的ε转换

NFA构造过程举例

- $(a \mid b)^*ab$的分解

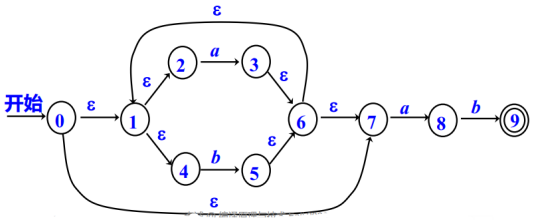


NFA → DFA

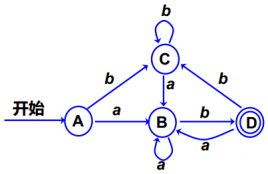
- 子集构造法
  - DFA的一个状态是NFA的一个状态集合
  - 读了输入$a_1\ a_2 \ldots a_n$后， NFA能到达的所有状态： $s_1, s_2, \ldots, s_k$， 则DFA到达状态 $\{s_1, s_2, \ldots, s_k\}$
  - ε-闭包（ε- closure） ： 状态s 的ε-闭包是 s经 ε转换所能到达的状态集合
  - NFA的初始状态的 ε-闭包对应于DFA的初始状态

- 针对每个DFA 状态 – NFA状态子集A， 求输入每个a_i 后能到达的NFA状态的ε-闭包并集（ ε- closure(move(A, a_i))）， 该集合对应于DFA中的一个已有状态，或者是一个要新加的DFA状态

- 例(a|b)*ab，NFA如下，把它变换为DFA



- 最后化简为下图：



> 子集构造法不一定得到最简DFA

DFA →化简的DFA

- A和B是可区别的状态
  - 从A出发，读过单字符b构成的串，到达非接受状态C，而从B出发，读过串b，到达接受状态D
- A和C是不可区别的状态
  - 无任何串可用来像上面这样区别它们

> 可区别的状态要分开对待

1. 按是否是接受状态来区分

   {A, B, C}, {D}

   move({A, B, C}, a) = {B}

   move({A, B, C}, b) = {C, D}

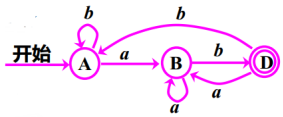
2. 继续分解

   {A, C}, {B}, {D}

   move({A, C}, a) = {B}

   move({A, C}, b) = {C}

3. 发现A和C是不可区分状态，合并AC，得到最简DFA



- 思考问题 ： 正则表达式（a|b）* 与（a* |b* ）*是否等价?
  - 提示：可利用其最简化DFA得
  - 答案：等价

总结

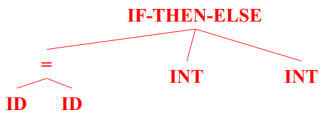
- 词法分析器的作用和接口，用高级语言编写词法分析器等内容
- 掌握下面涉及的一些概念，它们之间转换的技巧、方法或算法
  - 非形式描述的语言↔正则表达式
  - 正则表达式→ NFA
  - 非形式描述的语言↔ NFA
  - NFA → DFA
  - DFA →最简DFA
  - 非形式描述的语言 ↔DFA（或最简DFA）

第二周 9.24

语法分析

语法分析器简介

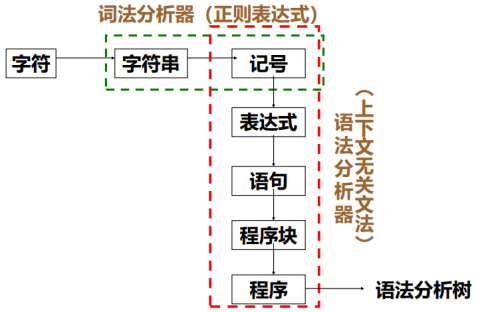
- 输入：从词法分析器中获得的记号序列
- 输出：程序的语法树（syntax or parse tree）
  - 语法树表示了源程序的层次化语法结构
  - 语法树是一种中间代码形式
- COOL 语言
  - if x = y then 1 else 2 fi
- 语法分析器的输入
  - IF ID = ID THEN INT ELSE INT FI
- 语法分析器的输出



- 不是所有的记号序列都是合法(valid)的
- 语法分析器需要区分合法和非法的记号序列
- 因此需要：
  - 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

正则表达式的局限

- 正则表达式的表达能力
  - 定义一些简单的语言，能表示给定结构的固定次数的重复或者没有指定次数的重复
    - 例： $a\ (ba)^5$, $a\ (ba)^*$
  - 不能用于描述配对或嵌套的结构
    - 例1：配对括号串的集合，如不能表达 $(^n)^n$, $n \geq 0$
    - 例2：{*wcw* | *w*是*a*和*b*的串}
  - 原因： 有限自动机无法记录访问同一状态的次数
- 词法分析器与语法分析器



上下文无关文法

- **上下文无关文法（Context-free Grammar，或CFG）是四元组（V_T, V_N, S, P）**
  - V_T: 终结符集合（基本符号，终结符↔记号名）
  - V_N: 非终结符集合（变量，非空有限集，$V_T \cap V_N = \emptyset$）
  - S : 开始符号，非终结符中的一个
  - P : 产生式集合
    - 产生式形式： $A \to \alpha$，$A \in V_N$，$\alpha \in (V_T \cup V_N)^*$
- 例 ({id, +, *, -, (,)}, {expr, op}, expr, P)
  - *expr* → *expr op expr*
  - *expr* → (*expr*)
  - *expr* → - *expr*
  - *expr* → id
  - *op* → +
  - *op* → *

CFG-简化表示

- 简化表示：引入选择运算符
  - *expr* → *expr op expr* | (*expr*) | - *expr* | id
  - *op* → + | *
- 简化表示
  - $E \to E\ A\ E \mid (E) \mid -E \mid \text{id}$
  - $A \to + \mid *$
- 请写出语言{$(^n)^n$, n ≥0 } 的CFG文法
  - $S \to (S) \mid \varepsilon$

正则表达式与CFG的区别

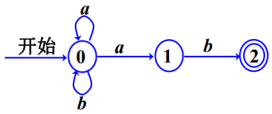
- 都能表示语言
- 能用正则表达式表示的语言都能用CFG表示
  - 正则表达式

    $(a|b)^*ab$
  - CFG文法

    $A_0 \to a\ A_0 \mid b\ A_0 \mid a\ A_1$

    $A_1 \to b\ A_2$

    $A_2 \to \varepsilon$



- NFA →上下文无关文法
  - 确定终结符集合
  - 为每个状态引入一个非终结符A_i
  - 如果状态i有一个a转换到状态j,引入产生式$A_i \rightarrow aA_j$,如果i是接受状态，则引入$A_i \rightarrow \varepsilon$
- 请为描述所有由0或1组成的回文字符串的语言设计CFG文法
  - $S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$
- 问题：如何判断一个CFG文法是否可以描述特定语言?

CFG-推导

- 推导 （Derivation）
  - 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
  - 把产生式看成重写规则，把符号串中的非终结符用其产生式右部的串来代替
- 例 E → E + E | E * E | (E) | - E | id
  - E → -E → -(E) → -(E + E) → -(id + E) → -(id + id)
- 记法：

$$S \Rightarrow^* \alpha: \text{0步或多步推导}$$
$$S \Rightarrow^+ w: \text{1步或多步推导}$$

最左推导和最右推导

- E → E + E | E * E | (E) | - E | id
- 最左推导（leftmost derivation）
  - 每步代换最左边的非终结符

$$E \Rightarrow_{lm} -E \Rightarrow_{lm} -(E) \Rightarrow_{lm} -(E+E)$$
$$\Rightarrow_{lm} -(\mathrm{id}+E) \Rightarrow_{lm} -(\mathrm{id}+\mathrm{id})$$

- 最右推导（rightmost or canonical derivation， 规范推导）
  - 每步代换最右边的非终结符

$$E \Rightarrow_{rm} -E \Rightarrow_{rm} -(E) \Rightarrow_{rm} -(E+E)$$
$$\Rightarrow_{rm} -(E+\mathrm{id}) \Rightarrow_{rm} -(\mathrm{id}+\mathrm{id})$$

CFG-简化表示

- 考虑如下文法：
  - *expr* → *term rest*
  - *rest* → + *term rest* | - *term rest* | ε
  - *term* → 0 | 1 | … | 9

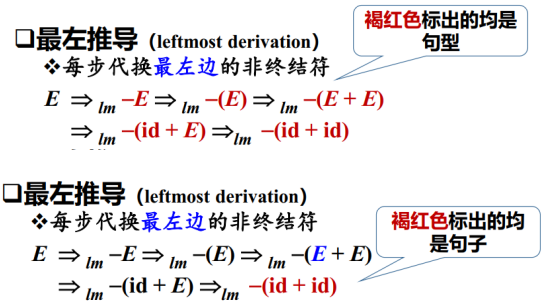
$$\begin{aligned} expr &\Rightarrow_{lm} \mathbf{term\ rest} \\ &\Rightarrow_{lm} \mathbf{1\ rest} \\ &\Rightarrow_{lm} \mathbf{1 + term\ rest} \\ &\Rightarrow_{lm} \mathbf{1 + 2\ rest} \\ &\Rightarrow_{lm} \mathbf{1 + 2 - term\ rest} \\ &\Rightarrow_{lm} \mathbf{1 + 2 - 3\ rest} \\ &\Rightarrow_{lm} \mathbf{1 + 2 - 3} \end{aligned}$$

- 上下文无关是什么意思?

- 上下文无关指的是在文法推导的每一步 $\alpha A\beta \Rightarrow \alpha\gamma\beta$符号串γ仅依据A 的产生式推导， 而无需依赖A的上下文α和β

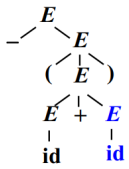
语言、文法、句型、句子

- 上下文无关语言
  - 上下文无关文法G产生的语言：从开始符号S 出发，经 →+推导所能到达的所有仅由终结符组成的串
  - 句型(sentential form)： S →*α ， S是开始符号，α 是由终结符和/或非终结符组成的串， 则α 是文法G的句型
  - 句子(sentence)： 仅由终结符组成的句型
- 等价的文法
  - 它们产生同样的语言



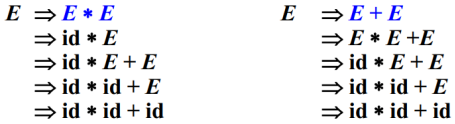
分析树(parse or syntax tree)

- 语法分析树是推导的图形表示形式
- 例 E → E + E | E * E | (E) | - E | id
  - -(id+id)最左推导的分析树

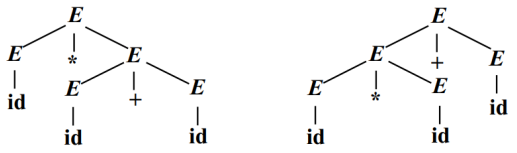


文法的二义性

- 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
- 例 E → E + E | E * E | (E) | - E | id
  - id * id + id有两个不同的最左推导



  - id * id + id有两棵不同的分析树



消除二义性

- 表达式产生二义性的原因

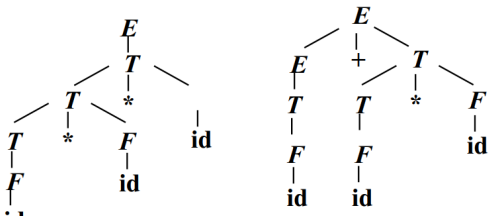
> +， *操作都是左结合的，并且在运算中有不同的优先级，但是在这个文法中没有得到体现

- 没有一般性的方法，但可通过定义运算优先级和结合律来消除二义性
- 用一种层次观点看待表达式
  - id * id * (id+id) + id * id + id
  - id * id * (id+id)

> E → E + E
>
> 从不同的E推导得到不同的树

- 新的非二义文法
  - E → E + T | T
  - T → T * F | F
  - F → id | (E)

> 根据算符不同的优先级，引入新的非终结符



id * id * id 和 id + id * id 的分析树

- 悬空else文法
  - stmt → if *expr* then *stmt*
    | if *expr* then *stmt* else *stmt*
    | other
- 句型： if *expr* then if *expr* then *stmt* else *stmt*
- 两个最左推导：

$$
\begin{aligned}
stmt &\Rightarrow \textbf{if } expr \textbf{ then } stmt \\
&\Rightarrow \textbf{if } expr \textbf{ then if } expr \textbf{ then } stmt \textbf{ else } stmt \\
stmt &\Rightarrow \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
&\Rightarrow \textbf{if } expr \textbf{ then if } expr \textbf{ then } stmt \textbf{ else } stmt
\end{aligned}
$$

- 无二义的文法
  - 每个else与最近的尚未匹配的then匹配

$$
\begin{aligned}
stmt &\rightarrow matched\_stmt \\
&\quad | \ unmatched\_stmt \\
matched\_stmt &\rightarrow \textbf{if } expr \textbf{ then } matched\_stmt \\
&\qquad \textbf{else } matched\_stmt \\
&\quad | \ \textbf{other} \\
unmatched\_stmt &\rightarrow \textbf{if } expr \textbf{ then } stmt \\
&\quad | \ \textbf{if } expr \textbf{ then } matched\_stmt \\
&\qquad \textbf{else } unmatched\_stmt
\end{aligned}
$$

语言与文法

- 上下文无关文法的优点
  - 文法给出了精确的，易于理解的语法说明
  - 自动产生高效的分析器
  - 可以给语言定义出层次结构
  - 以文法为基础的语言的实现便于语言的修改
- 上下文无关文法的缺点
  - 文法只能描述编程语言的大部分语法

分离词法分析器的理由

- 为什么要用正则表达式定义词法
  - 词法规则非常简单，不必用上下文无关文法
  - 对于词法记号，正则表达式描述简洁且易于理解
  - 从正则表达式构造出的词法分析器效率高
- 分离词法分析和语法分析的好处（软件工程视角）
  - 简化设计
  - 编译器的效率会改进
  - 编译器的可移植性加强
  - 便于编译器前端的模块划分

语法分析的主要方法

- 自顶向下（Top-down）
  - 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
  - 分析树的构造方法
    - 从根部开始
  - 即便是进行消除左递归、提取左公因子操作，仍然存在一些程序语言，他们对应的文法不是LL(1)
- 自底向上（Bottom-up）
  - 针对输入串，尝试根据产生式规则规约（reduce）到文法的开始符号
  - 分析树的构造方法
    - 从叶子开始
  - 比top-down分析方法更一般化

递归下降分析法 （Recursive Descent Parsing ）

- 考虑以下文法
  - $E \rightarrow T \mid T + E$
  - $T \rightarrow \text{int} \mid \text{int} * T \mid (E)$
- 输入串: (int_5)
- 从左到右扫描输入串
- 从开始非终结符E开始
- 按顺序尝试E的产生式
- 递归下降的预测分析

    - 为每一个非终结符写一个分析过程
    - 这些过程可能是递归的
- 例
    - *type* → *simple*

        | ↑ id

        | array [*simple*] of *type*
    - simple → integer

        | char

        | num dotdot num
- 一个辅助过程

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

void type( ) {
if ( (lookahead == integer) || (lookahead == char) ||
(lookahead == num) )
    simple( );
else if ( lookahead == '↑' ) {
    match('↑');
    match(id);
}
else if (lookahead == array) {
    match(array);
    match( '[');
    simple( );
    match( ']' );
    match(of );
    type( );
}
else
    error( );
}

void simple( ) {
if ( lookahead == integer)
    match(integer);
else if (lookahead == char)
    match(char);
else if (lookahead == num) {
    match(num);
    match(dotdot);
    match(num);
}
else
    error( );
}
```

第三周 9.28

递归下降分析法

- 递归下降的预测分析
  - 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
  - 为每一个非终结符写一个分析过程
    - 该过程可以调用其他非终结符的过程及match
    - 这些过程可能是递归的

模拟推导

- 考虑以下文法：

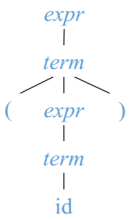
  expr → *term* | *term* + *expr*

  term → id | (*expr*)

- 输入串: (id)
- 分析过程：
  - 从左到右扫描输入串
  - 开始符号： *expr*
  - 按顺序尝试产生式

```
void match (terminal t) {
        if (lookahead==t)
            lookahead = nextToken( );
        else error( );
}
void expr() {
     term();
     if (lookahead ≠ ε) expr();
}
void term(){
    if (lookahead is id){
        match(lookahead);
    } else if (lookahead == '(') {
        match('(');
        expr();
        match(')');
    } else report("语法错误");
}
```

- 语法树



递归下降的问题

- 可能进入无限循环
- 考虑以下文法
  - S → Sa
- 该文法是左递归的(left-recursive)
- 自顶向下分析方法无法处理左递归

消除左递归

- 文法左递归

$$A \Rightarrow^{+} A\alpha$$

- 直接左递归
  - $A \rightarrow A\alpha \mid \beta$ ， 其中α， β 不以A开头
  - 串的特点 $\beta\alpha \ldots \alpha$
- 消除直接左递归
  - $A \rightarrow \beta A'$
  - $A' \rightarrow \alpha A' \mid \varepsilon$
- 例 算术表达文法

  $E \rightarrow E + T \mid T$ （ $T + T \ldots + T$ ）

  $T \rightarrow T * F \mid F$ （ $F * F \ldots * F$ ）

  $F \rightarrow \text{id} \mid (E)$
- 消除左递归后文法

  $E \rightarrow TE'$

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

  $T \rightarrow FT'$

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

  $F \rightarrow \text{id} \mid (E)$

消除间接左递归

- 非直接左递归

  $S \rightarrow Aa \mid b$

  $A \rightarrow Sd \mid \varepsilon$
- 先变换成直接左递归

  $S \rightarrow Aa \mid b$

  $A \rightarrow Aad \mid bd \mid \varepsilon$
- 再消除左递归

  $S \rightarrow Aa \mid b$

  $A \rightarrow bd\,A' \mid A'$

  $A' \rightarrow adA' \mid \varepsilon$

递归下降的问题2

- 有左因子的(left -factored)文法：
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
- 提左因子(left factoring)
  - 推后选择产生式的时机，以便获取更多信息

    $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ 等价于

    $A \rightarrow \alpha A'$

    $A' \rightarrow \beta_1 \mid \beta_2$

提左因子(left factoring)

- 例 悬空else的文法

  stmt → if *expr* then *stmt*

  | if *expr* then *stmt* else *stmt*

  | other

  提左因子

  stmt → if *expr* then *stmt optional_else_part*

  | other

  optional_else_part → else *stmt*

  | ε

> 算法仍然二义！！！

递归下降的问题3

- 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

预测分析法 (Predictive parsing)

- 与递归下降法相似，但
  - 不会对若干产生式进行尝试
  - 没有回溯
  - 通过向前看一些记号来预测需要用到的产生式
- 此方法接受LL(k)文法
  - L意为从左到右扫描输入串
  - L意为最左推导
  - k意为提前预测k个tokens
  - 一般使用LL(1)

LL(1)文法

- 对文法加什么样的限制可以保证没有回溯?
- 先定义两个和文法有关的函数
  - FIRST(α) = {a | α =>* a..., a ∈ V_T}

    特别是， α =>*</sup> ε时，规定ε ∈ FIRST(α)
  - FOLLOW(A) = {a | S =>* ...Aa..., a∈ V_T}

    如果非终结符A是某个句型的最右符号， 那么$属于FOLLOW($A$)

    $ 是输入串的结束符号

FIRST(X)

- 计算FIRST(X), $X \in V_T \cup V_N$
  - $X \in V_T$ ， FIRST(X) = {X}
  - $X \in V_N$ 且 $X \rightarrow$ ε

则将 ε加入到FIRST(*X*)

- $X \in V_N$ 且 $X \rightarrow Y_1\ Y_2 \ldots Y_k$
  - 如果 $a \in$ FIRST(Y_i)且ε在FIRST(Y_1), ...,FIRST(Y_{i-1})中，则将 a加入到FIRST(X)
  - 如果 ε在FIRST(Y_1), ..., FIRST(Y_k)中，则将ε加入到FIRST(*X*)

> FIRST集合只包括终结符和ε

表达式文法：无左递归的

- 例：$E \rightarrow TE'$

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

  $T \rightarrow FT'$

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

  $F \rightarrow$ id | (*E*)
- FIRST(*E*) = FIRST(*T*) = FIRST(*F*) = { (, id }
- FIRST(*E'*) = {+, ε}
- FRIST(*T'*) = {*, ε}

FOLLOW(A)

- 计算FOLLOW(*A*), $A \in V_N$
  - $加入到FOLLOW(*A*)，当A是开始符号
  - 如果$A \rightarrow \alpha B\beta$ ，则FIRST(β)-{ε}加入到FOLLOW(*B*)
  - 如果$A \rightarrow \alpha B$ 或$A \rightarrow \alpha B\beta$且$\varepsilon \in$ FIRST(β)，则FOLLOW(*A*)加入到FOLLOW(*B*)

表达式文法：无左递归的

- 例：$E \rightarrow TE'$

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

  $T \rightarrow FT'$

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

  $F \rightarrow$ id | (*E*)
- FOLLOW(*E*) = FOLLOW(*E'*) = {), $}
- FOLLOW(*T*) = FOLLOW (*T'*) = {+,), $}
- FOLLOW(*F*) = {+, *,), $}
- LL(1)文法的定义
  - 任何两个产生式$A \rightarrow \alpha \mid \beta$ 都满足下列条件：
    - FIRST(α) ∩ FIRST(β) = ∅
    - 若$\beta \Rightarrow^* \varepsilon$ ，那么FIRST(α) ∩ FOLLOW(*A*) = ∅
- 该条件存在的必要性
  - 容易理解
  - 每次通过输入词法单元记号和FIRST集合匹配产生式的时候，需要有唯一的选择
- 假设FIRST(α) ∩ FOLLOW(*A*) = {a}

  a ∈ FIRST(α): $A \Rightarrow^* a\alpha'$

  a ∈ FOLLOW(*A*): $B \Rightarrow^* \ldots Aa\ldots$

  由于$\beta \Rightarrow^* \varepsilon$ ，所以遇到a时，无法判断用哪一个产生式

- 可以用$A \rightarrow \alpha$ 来对A进行展开
- 亦可以用$A \rightarrow \beta$和$\beta \Rightarrow^* \varepsilon$最后把$A$消掉

- 例如, 考虑下面文法

  面临a...时，第2步推导不知用哪个产生式

  $S \rightarrow A\ B$

  $A \rightarrow a\ b \mid \varepsilon$

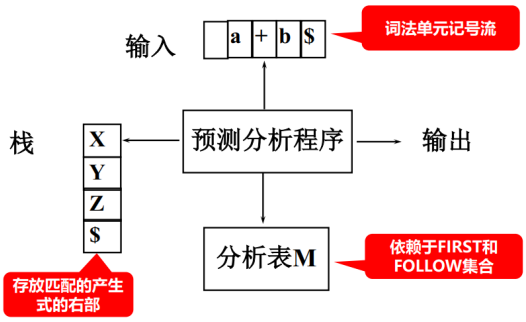
  $B \rightarrow a\ C$

  $C \rightarrow \ldots$

  $a \in FIRST(ab) \cap FOLLOW(A)$

- LL(1)文法有一些明显的性质
  - 没有公共左因子
  - 不是二义的
  - 不含左递归

非递归的预测分析



预测分析表M的构造

- 对文法的每个产生式$A \rightarrow \alpha$， 执行(1)和(2)
  - 1） 对FIRST(α)的每个终结符a， 把$A \rightarrow \alpha$ 加入$M[A, a]$
  - 2） 如果ε在FIRST(α)中， 对FOLLOW(A)的每个终结符b（包括\$），把$A \rightarrow \alpha$加入$M[A, b]$
- M中其它没有定义的条目都是error
- 行：非终结符；列：终结符 或\$ ； 单元：产生式

非终结符	输 入 符 号					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$id			$F \rightarrow (E)$		

第四周 10.10

预测分析举例

预测分析器接受输入id * id + id的前一部分动作

栈	输入	输出
$E	id * id + id$	
$E'T	id * id + id$	E → TE'
$E'T'F	id * id + id$	T → FT'
$E'T'id	id * id + id$	F → id
$E'T'	* id + id$	匹配id
$E'T'F*	* id + id$	T' → *FT'
$E'T'F	id + id$	匹配*
$E'T'id	id + id$	F → id
$E'T'	+ id$	匹配id
$E'	+ id$	T' →ε
$E'T+	+ id$	E' → +TE'
$E'T	id$	匹配+
$E'T'F	id$	T → FT'
$E'T'id	id$	F → id
$E'T'	$	匹配id
$E'	$	T' →ε
$	$	E' →ε

多重定义

例： *stmt* → **if** *expr* **then** *stmt e_part* | **other**
e_part → **else** *stmt*| ε　　*expr* → *b*

非终结符	输入符号			
	other	*b*	else	...
stmt	*stmt* → other			
e_part			*e_part*→ else *stmt* *e_part* → ε	
expr		*expr* → *b*		

多重定义条目意味着文法左递归或者是二义的

多重定义的消除

例：删去*e_part* → ε，这正好满足else和近的then配对
LL(1)文法：预测分析表无多重定义的条目

非终结符	输入符号			
	other	*b*	else	...
stmt	*stmt* → other			
e_part			*e_part*→ else *stmt* ~~*e_part* → ε~~	
expr		*expr* → *b*		

预测分析的错误恢复

- 编译器的错误
  - 词法错误，如标识符、关键字或算符的拼写错
  - 语法错误，如算术表达式的括号不配对
  - 语义错误，如算符作用于不相容的运算对象
  - 逻辑错误，如无穷的递归调用
- 非递归预测分析在什么场合下发现错误
  - 栈顶的终结符和下一个输入符号不匹配
  - 栈顶是非终结符A， 输入符号是a， 而M[A , a]是空白

自底向上分析方法

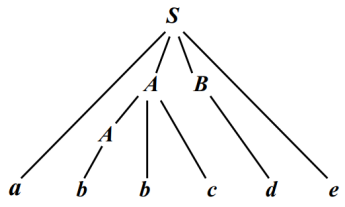
- 归约(右推导的逆过程)
- 句柄(可归约串), 可能不唯一
- 冲突：移进-归约、归约-归约

归约(Reduce)

- 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
- 最终，把输入串归约成文法的开始符号

例 $S \to aABe$
$A \to Abc \mid b$
$B \to d$
输入串: abbcde

ab	(读入ab)
aA	(归约)
aAbc	(再读入bc)
aA	(归约)
aAd	(再读入d)
aAB	(归约)
aABe	(再读入e)
S	(归约)



$$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$$

- 需要解决两个问题
  - 在读入串的过程中，如何识别可以归约的子串?
  - 在进行归约的时候，选择哪一个产生式?

句柄(Handles)

- 句型的句柄（可归约串）
  - 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

    S → aABe

    A → Abc | b

    B → d

    $$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$$
  - 句柄的右边仅含终结符
  - 如果文法二义，那么句柄可能不唯一

例 句柄不唯一

E → E + E | E * E | (E) | id

$$
\begin{aligned}
E &\Rightarrow_{rm} E * E \\
&\Rightarrow_{rm} E * E + E \\
&\Rightarrow_{rm} E * E + \mathrm{id}_3 \\
&\Rightarrow_{rm} E * \mathrm{id}_2 + \mathrm{id}_3 \\
&\Rightarrow_{rm} \mathrm{id}_1 * \mathrm{id}_2 + \mathrm{id}_3
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow_{rm} E + E \\
&\Rightarrow_{rm} E + \mathrm{id}_3 \\
&\Rightarrow_{rm} E * E + \mathrm{id}_3 \\
&\Rightarrow_{rm} E * \mathrm{id}_2 + \mathrm{id}_3 \\
&\Rightarrow_{rm} \mathrm{id}_1 * \mathrm{id}_2 + \mathrm{id}_3
\end{aligned}
$$

在句型E * E + id_3中，句柄不唯一

第五周 10.12

移进-归约分析技术

- 用栈实现移进-归约分析
  - 栈保存已扫描过的文法符号，缓冲区存放还未分析的其余符号
  - 移进(shift)：将下一个输入符号放到栈顶，以形成句柄
  - 归约(reduce)：将句柄替换为对应的产生式的左部非终结符
  - 接受(accept)：分析成功
  - 报错(error)：发现语法错误
- 先通过分析输入串id_1*id_2 + id_3时的动作序列来了解移进-归约分析的工作方式

栈	输入	动作
$	id_1*id_2 + $id_3$$	移进
$$id_1$	*id_2 + $id_3$$	按E→id归约
$E	*id_2 + $id_3$$	移进
$E*	id_2 + $id_3$$	移进
$E*$id_2$	+ $id_3$$	按E→id归约
$E*E	+ $id_3$$	移进
$E*E+	$id_3$$	移进
$E*E+$id_3$	$	按E→id归约
$E*E+E	$	按E→E+E归约
$E*E	$	按E→E*E归约
$E	$	接受

移进-归约分析中的冲突

- 要想很好地使用移进-归约方式，尚需解决一些问题
  - 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)
  - 进行归约时，如何确定选择哪一个产生式

移进-归约冲突

- 例

  stmt → if *expr* then *stmt*

  | if *expr* then *stmt* else *stmt*

  | other

- 如果移进-归约分析器处于格局(configuration)
  - 栈：... if expr then stmt　　——归约?
  - 输入：else ... $　　——移进?

归约-归约冲突

- 例

  stmt → id (parameter_list) | expr = expr
  parameter_list→parameter_list, parameter | parameter
  parameter → id
  expr → id (expr_list) | id
  expr_list → expr_list, expr | expr

- 由A(I, J)开始的语句
  - 栈：... id (id　　归约成expr还是parameter?
  - 输入：，id）...
- 需要修改文法中的第一个产生式，并利用栈中信息

  stmt → procid (parameter_list) | expr = expr
  parameter_list→parameter_list, parameter | parameter
  parameter → id
  expr → id (expr_list) | id
  expr_list → expr_list, expr | expr

- 由A(I, J)开始的语句(词法分析查符号表,区分第一个id)
  - 栈：... procid（ id　　归约成expr还是parameter?
  - 输入：，id）...

LR(k)分析技术

- LR分析器的简单模型
  - action, goto函数
- 简单的LR方法（简称SLR）
  - 活前缀，识别活前缀的DFA/NFA，SLR算法
- 规范的LR方法
- 向前看的LR方法（简称LALR）

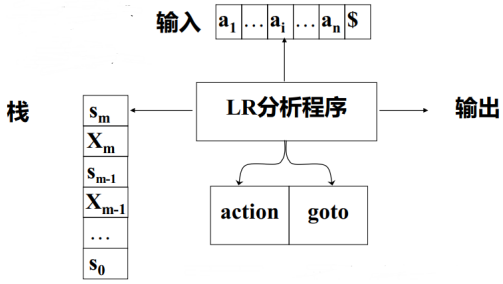
语法分析的主要方法

- 自顶向下（Top-down）
  - 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
  - LL(1)文法及非递归预测分析方法
  - left-to-right scan + leftmost derivation
- 自底向上（Bottom-up）

- 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
- LR(k)文法及其分析器
- left-to-right scan + rightmost derivation

LR分析器

- s_j: 总结了栈中该状态以下的信息
- X_i: 代表文法符号
- action[s_m, a_i]: 移进|归约|接受|出错
- goto[s_{m-r} , A]=s_j: 移进A和s_j (归约后使用)



LR语法分析器

- 关键在于构造LR分析表
  - 计算所有可能的状态
    - 每一个状态描述了语法分析过程中所处的位置
    - 可确定正在分析的产生式集合
    - 可确定句柄形成的中间步骤
  - 明确状态之前的跳转关系
  - 明确状态与输入之间对应的移进或者归约操作

LR语法分析器的格局

- LR语法分析的每一步都形成一个格局config

$$(s_0X_1s_1X_2s_2\ldots X_ms_m, \quad a_ia_{i+1}\ldots a_n\$)$$

栈的内容　　尚未处理的输入

- 代表最右句型$X_1X_2\ldots X_ma_ia_{i+1}\ldots a_n$
- $X_1X_2\ldots X_m$是最右句型的一个前缀
- 每一个前缀都对应一个状态，因此，找出所有可能在栈里出现的前缀，就可以确定所有的状态
- 状态之间的转换 <==> 前缀之间的转换
- 在栈顶为s，下一个字符为a的格局下，前缀为p
  - 何时移进？当p包含句柄的一部分且存在p'= pa
  - 何时归约？当p包含整个句柄时

LR分析: 基本概念

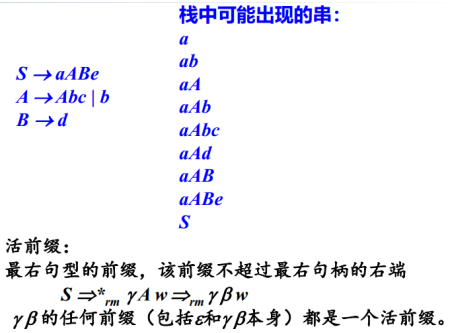
- 活前缀或可行前缀 (viable prefix):
  - 最右句型的前缀，该前缀不超过最右句柄的右端

$$S \Rightarrow^*_{rm} \gamma A w \Rightarrow_{rm} \gamma\beta w$$

  - γβ的任何前缀（包括ε和γβ本身）都是活前缀

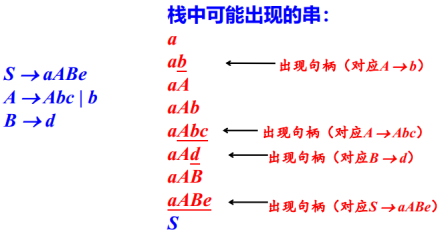
- 都出现在栈顶

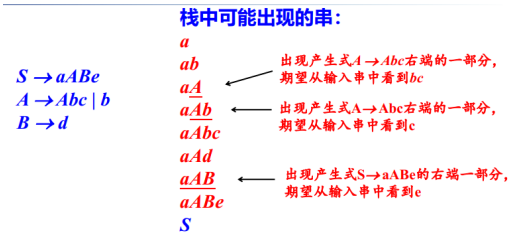
活前缀



活前缀与句柄的关系

- 活前缀已含有句柄， 表明产生式A→β的右部β已出现在栈顶
- 活前缀只含句柄的一部分符号如β_1表明A→$\beta_1\beta_2$的右部子串β_1已出现在栈顶，当前期待从输入串中看到β_2推出的符号



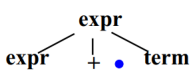


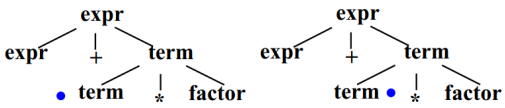
LR分析方法的特点

- 栈中的文法符号总是形成一个活前缀
- 分析表的转移函数本质上是识别活前缀的DFA
- 栈顶的状态符号包含确定句柄所需的一切信息
- 是已知的最一般的无回溯的移进-归约方法
- 能分析的文法类是预测分析法能分析的文法类的真超集
- 能及时发现语法错误
- 手工构造分析表的工作量太大

SLR分析表的构造

- SLR (Simple LR)
- LR(0)项目（简称项目）
  - 在右部的某个地方加点的产生式
  - 加点的目的是用来表示分析过程中的状态





> 项代表了一个可能的前缀
>
> 点的左边代表历史信息，点的右边代表展望信息。

- 例 A → XYZ对应有四个项目

  A → ·XYZ

  A → X·YZ

  A → XY·Z

  A → XYZ·

- 例 A → ε只有一个项目和它对应

  A → ·

- 从文法构造识别活前缀的DFA
- 从上述DFA构造分析表

构造识别活前缀的DFA

1. 拓（增）广文法 (augmented grammar)

   E' → E

   E → E+T|T

   T → T*F|F

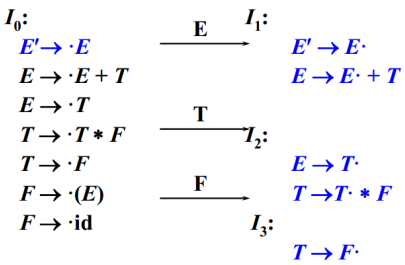
   F → (E) | id

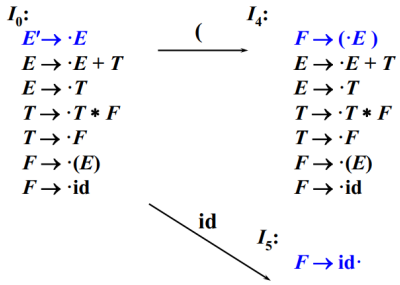
   当且仅当分析器使用E' → E归约时，宣告分析成功

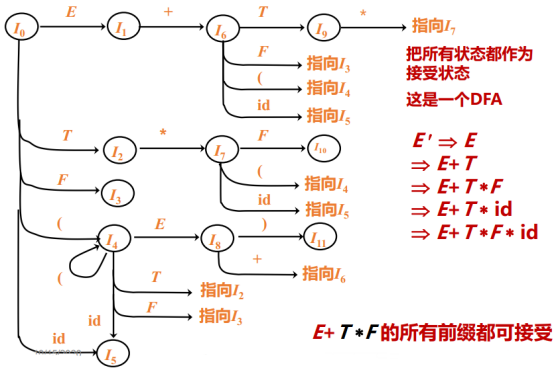
2. 构造LR(0)项目集规范族

- 项集族是若干可能前缀的集合，对应DFA的状态

> 求项目集的闭包closure(I)
>
> 闭包函数closure(I)
>
> 1. I的每个项目均加入closure(I)
> 2. 如果A→α·Bβ在 closure(I)中，且B→γ是产生式，那么如果项目B→ ·γ还不在closure(I)中的话，那么把它加入。







第五周 10.15

SLR分析表的构造

- 从文法构造识别活前缀的DFA
- 从上述DFA构造分析表

从DFA构造SLR分析表

- 状态 i 从 I_i 构造，它的 action 函数如下确定：
- 如果[A→α·aβ]在 I_i 中，并且goto(I_i , a) = I_j， 那么置action[i, a]为s_j
- 如果[A→α·]在I_i 中，那么对FOLLOW(A)中的所有a， 置action[i, a]为r_j,j是产生式 A→α的编号
- 如果[S'→S·]在I_i 中，那么置action[i, $]为接受acc
- 上面的a是终结符

如果出现动作冲突，那么该文法就不是SLR(1)文法

- 使用下面规则构造状态i的goto函数：
  - 对所有的非终结符A， 如果goto(I_i , A) = I_j , 那么goto[i, A] = j
- 分析器的初始状态是包含[S'→S·]的项目集对应的状态

不能由上面两步定义的条目都置为error

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

判定满足SLR文法输入串

- 依据上述SLR(1)分析表
- 参照slide 9 -25的分析方法
  - 文法符号栈
  - 输入缓冲区
  - 选择的行为
    - 移进、归约、接受、报错

SLR分析器——解释

- 例 I_2:

  (2)E → T·

  (3)T → T·* F
- 归约：因为FOLLOW(E) = {$, +,)},

  所以 action[2, $]=action[2, +]=action[2,)]=r2
- 移进：因为圆点在中间， 且点后面是终结符，

  所以， action[2, *] = s7

活前缀的概念-revisit

- LR(0)自动机刻画了可能出现在文法符号栈中的所有串；
- 栈中的内容一定是某个最右句型的前缀；
- 但是不是所有前缀都会出现在栈中

$$E \Rightarrow^*_{rm} F * \mathbf{id} \Rightarrow_{rm} (\mathrm{E}) * \mathbf{id}$$

- 栈中只能出现(, (E, (E)，而不会出现(E)*
- 因为看到*时， (E)是句柄，会被归约成为F
- 活前缀或可行前缀 (viable prefix)：
  - 最右句型的前缀，该前缀不超过最右句柄的右端

$$S \Rightarrow^*_{rm} \gamma A w \Rightarrow_{rm} \gamma \beta w$$

  - γ β的任何前缀（包括ε和γ β本身）都是活前缀
  - 都出现在栈顶

> LR(0) 自动机能够识别活前缀
>
> SLR(1)分析器正是基于这个事实

有效项目

- 如果 $S' \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta_1\beta_2 w$，那么就说项目A→$\beta_1 \cdot \beta_2$对活前缀$\alpha\beta_1$是有效的
  - 一个项目可能对好几个活前缀都是有效的

$$E' \to E \qquad E \to E + T \mid T$$
$$T \to T * F \mid F \qquad F \to (E) \mid \mathbf{id}$$

  - 项E → ·E+T 对 ε 和 (这两个活前缀都有效

$$E' \Rightarrow E \Rightarrow E+T \qquad (\alpha,\ \beta_1\text{都为空})$$
$$E' \Rightarrow E \Rightarrow (E) \Rightarrow (E+T) \qquad (\alpha\text{=“(”},\ \beta_1\text{为空})$$

- 该DFA读过ε 和(后到达不同的状态，那么项目E → ·E+T就出现在对应的不同项目集中
- 一个项目可能对好几个活前缀都是有效的
  - 如果$\beta_2 \neq \varepsilon$，应该移进
  - 如果$\beta_2 = \varepsilon$， 应该用产生式$A \rightarrow \beta_1$归约
- 一个项目可能对好几个活前缀都是有效的
- 一个活前缀可能有多个有效项目

一个活前缀γ的有效项目集就是，从这个DFA的初态出发，沿着标记为γ的路径到达的那个项目集（状态）

- 例 串E + T *是活前缀，读完它后， DFA处于状态I_7

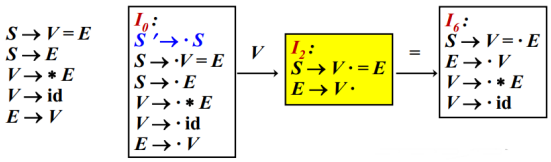
$$I_7: \quad T \rightarrow T * \cdot F,\ F \rightarrow \cdot(E),\ F \rightarrow \cdot \mathrm{id}$$

$$\begin{aligned}
E' &\Rightarrow E \\
&\Rightarrow E+T \\
&\Rightarrow E+T*F \\
&\Rightarrow E+T*\mathrm{id} \\
&\Rightarrow E+T*F*\mathrm{id}
\end{aligned}
\qquad
\begin{aligned}
E' &\Rightarrow E \\
&\Rightarrow E+T \\
&\Rightarrow E+T*F \\
&\Rightarrow E+T*(E) \\
&
\end{aligned}
\qquad
\begin{aligned}
E' &\Rightarrow E \\
&\Rightarrow E+T \\
&\Rightarrow E+T*F \\
&\Rightarrow E+T*\mathrm{id} \\
&
\end{aligned}$$

包含活前缀的最右推导，且I_7中所有的项目对该活前缀是有效的

规范的LR方法

SLR(1)文法的描述能力有限



项目S → V ·= E使得action[2, =] = s6

项目E → V ·使得action[2, =] = r5 因为Follow(E)={=, $}

产生移进-归约冲突，但该文法不是二义的

- 目标：在识别活前缀DFA的状态中， 增加信息，排除一些不正确的归约操作
- 方法：添加了前向搜索符
  - 一个项目A→α·β ，如果最终用这个产生式进行归约之后，期望看见的符号是a，则这个加点项的前向搜索符是a
  - 上述项目可以写成：A→α·β ， a
- 与SLR(1)分析的区别
  - 项目集的定义发生了改变： LR(0) => LR(1)
  - closure(I) 和GOTO函数需要修改
- LR(1)项目：

  [A→α·β ， a]
- 当项目由两个分量组成，第一分量为SLR中的项，第二分量为搜索符（向前看符号）
- LR(1)中的1代表了搜索符a的长度
- 使用注意事项：
  - 当β不为空时， a不起作用
  - 当β为空时，如果下一个输入符号是a，将按照A→α 进行归约

        - a的集合是FOLLOW(A)的子集
- LR(1)项目[A→α·β， a]对活前缀γ 有效：
    - 如果存在着推导 $S \Rightarrow^*_{rm} \delta Aw \Rightarrow_{rm} \delta\alpha\beta w$，其中：
        - γ = δα；
        - a是w的第一个符号， 或者w是ε且a是$

举例

- 例 S → BB

  B → bB | a
- 从最右推导 $S \Rightarrow^*_{rm} bbBba \Rightarrow_{rm} bbbBba$看出：
- 令A = B, α = b, β =B , δ = bb, γ =δα = bbb, w = ba

[$B \to b \cdot B, b$]对活前缀$\gamma = bbb$是有效的

步骤

- 构造LR(1)项目集规范族
    - 也就是构造识别活前缀的DFA
- 构造规范的LR分析表
    - 状态之间的转换关系

构造LR(1)项目集规范族

- 基础运算1：计算闭包CLOSURE(I)
    - I中的任何项目都属于CLOSURE(I)
    - 若有项目 [A→α·Bβ， a]在CLOSURE(I)中， 而B→γ是文法中的产生式， b是FIRST(βa)中的元素，则[B→·γ, b]也属于CLOSURE(I)

      > 保证在用B→γ 进行归约后，
      >
      > - 出现的输入字符b是句柄αBβ中B的后继符号
      > - 或者是 αBβ归约为A后可能出现的终结符
- 基础运算2：通过GOTO(I,X)算CLOSURE(J)
    - 将J置为空集
    - 若有项目 [A→α·Xβ, a]在I中， 那么将项目[A→αX·β, a]放入J中

      > 注意： GOTO(I,X)中的X可以是终结符或非终结符
- 具体算法
    - 初始项目集I_0：

      I_0= CLOSURE([S'→·S, $]) 将$作为向前的搜索符
    - 设C为最终返回的项目集族，初始为C={I_0}
    - 重复以下步骤
        - 对C中的任意项目集I，重复
            - 对每一个文法符号X(终结符或非终结符)
                - 如果GOTO(I,X) ≠ ∅ 且 GOTO(I,X) ∉ C，那么将GOTO(I,X)放入C
                - 注： 上述GOTO(I,X)是上述计算闭包的GOTO
        - 当C中项目集不再增加为止

构造规范的LR分析表

- 构造识别拓广文法G'活前缀的DFA
  - 基于LR(1)项目族来构造
- 状态i的action函数如下确定：
  - 如果[A →α ·aβ, b]在I_i中，且goto(I_i, a) = I_j， 那么置action[i, a]为s_j (此时，不看b)
  - 如果[A→α ·, a]在I_i中，且A ≠ S'， 那么置action[i,a]为r_j (此时，不再看FOLLOW(A))
  - 如果[S'→S·, $]在$I_i$中，那么置action[i, $] = acc

    > 如果上述构造出现了冲突，那么文法就不是LR(1)的
- 状态i的goto函数如下确定：
  - 如果goto(I_i, A) = I_j, 那么goto[i, A] = j
- 分析器的初始状态是包含[S'→·S, $]的项目集对应的状态

> 用上面规则未能定义的所有条目都置为error

第六周 10.19

非SLR(1)但是LR(1)文法

考虑文法：

> $S \rightarrow V = E$
> $S \rightarrow E$
> $V \rightarrow * E$
> $V \rightarrow$ id
> $E \rightarrow V$

计算初始闭包I_0：$S' \rightarrow \cdot S$，$

定义里： [A → α·B β, a]　　FIRST(βa)

这里：　 [S' → ε ·S ε, $]　　FIRST(ε$) = {$}

S → ·V = E, $
S → ·E, $

定义里： [A → α·B β, a]　　FIRST(βa)

这里：　 [S → ε ·V = E, $]　　FIRST(=E$) = {=}

定义里： [A → α·B β, a]　　FIRST(βa)

这里：　 [S → ε ·E ε, $]　　FIRST(ε$) = {$}

V → ·* E, =
V → ·id, =

E → ·V, $

V → ·* E, $
V → ·id, $

- 可通过合并搜索符简化

初始闭包I_0：

> $S' \rightarrow \cdot S$，$

> S → ·V = E, $
> S → ·E, $
>
> V → ·* E, =/$
> V → ·id, = /$
>
> E → ·V, $

每一个SLR(1)文法都是LR(1)的

LALR分析方法

- 研究LALR的原因
  - 规范LR分析表的状态数偏多
- LALR特点
  - LALR和SLR的分析表有同样多的状态，比规范LR分析表要小得多
  - LALR的能力介于SLR和规范LR之间
  - LALR的能力在很多情况下已经够用
- LALR分析表构造方法
- 通过合并规范LR(1)项目集来得到
- 合并识别 LR(1)文法的活前缀的DFA中的相同核心项目集(同心项目集，注意：不是项)
- 同心的LR(1)项目集
  - 核心：项目集中第一分量的集合
  - 略去搜索符后它们是相同的集合
  - 例： [B →·bB, $] 与 [B →·bB, b/a]

    > 蓝色：第一分量
    >
    > 红色：第二分量
- 构造LALR(1)分析表
  - 构造LR(1)项目集规范族$C = \{I_0, I_1, ..., I_n\}$
  - 构造LALR(1)项目集规范族$C' = \{J_0, J_1, ..., J_k\}$，其中任意项目集$J_i = I_n \cup I_m \cup ... \cup I_t$
    - I_n，I_m，... ，$I_t \in C$ 且具有共同的核心
  - 按构造规范LR(1)分析表的方式构造分析表

> 如没有语法分析动作冲突，那么给定文法就是LALR(1)文法

- 合并同心项目集可能会引起冲突
  - 同心集的合并不会引起新的移进-归约冲突

  项目集1　　项目集2

  [A→α ·, a]　　[B→β·aγ, b]

  [B→β·aγ, c]　　[A→α ·, d]

  ...　　...

  如果有移进归约冲突，则合并前就有冲突
- 同心集的合并有可能产生新的归约-归约冲突

$S' \to S$
$S \to aAd \mid bBd \mid aBe \mid bAe$
$A \to c$
$B \to c$

对ac有效的项目集

$$\boxed{\begin{array}{l} A \to c\cdot, d \\ B \to c\cdot, e \end{array}}$$

对bc有效的项目集

$$\boxed{\begin{array}{l} A \to c\cdot, e \\ B \to c\cdot, d \end{array}}$$

合并同心集后

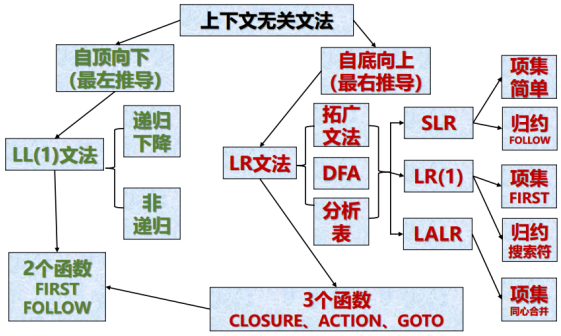
$$\boxed{\begin{array}{l} A \to c\cdot, d/e \\ B \to c\cdot, d/e \end{array}}$$

该文法是LR(1)的但不是LALR(1)的

LR分析法总结

		SLR	LALR	LR(1)
初始状态		$[S' \to \cdot S]$	$[S' \to \cdot S, \$]$	$[S' \to \cdot S, \$]$
项目集		LR(0) CLOSURE(I)	合并LR(1)项目集族的同心项目集	LR(1), CLOSURE(I) 搜索符考虑FISRT(βa)
动作	移进	$[A \to \alpha\cdot a\beta] \in I_i$ GOTO(I_i, a) = I_j ACTION[i, a] = sj	与LR(1)一致	$[A \to \alpha\cdot a\beta, b] \in I_i$ GOTO(I_i, a) = I_j ACTION[i, a] = sj
	归约	$[A \to \alpha\cdot] \in I_i, A \neq S'$ $a \in$ FOLLOW(A) ACTION[i, a] = rj	与LR(1)一致	$[A \to \alpha\cdot, a] \in I_i$ $A \neq S'$ ACTION[i, a] = rj
	接受	$[S' \to S\cdot] \in I_i$ ACTION[i, \$] = acc	与LR(1)一致	$[S' \to S\cdot, \$] \in I_i$ ACTION[i, \$] = acc
	出错	空白条目	与LR(1)一致	空白条目
GOTO		GOTO(I_i, A) = I_j GOTO[i, A] = j	与LR(1)一致	GOTO(I_i, A) = I_j GOTO[i, A] = j
状态量		少(几百)	与SLR一样	多(几千)

语法分析技术总结



LR和LL分析方法的比较

	LR(1)方法	LL(1)方法
建立分析树	自底而上	自顶而下
归约or推导	规范归约	最左推导
决定使用产生式的时机	看见产生式整个右部推出的串后(句柄)	看见产生式推出的第一个终结符后
对文法的限制	无	无左递归、无公共左因子
分析表	状态×文法符号，大	非终结符×终结符，小
分析栈	状态栈，信息更多	文法符号栈
确定句柄	根据栈顶状态和下一个符号便可以确定句柄和归约所用产生式	无句柄概念
语法错误	决不会将出错点后的符号移入分析栈	和LR一样，决不会读过出错点而不报错