# 词法分析

## NFA转DFA：子集法

## DFA化简

书：分割法

编程：（深度优先、）广度优先

## 正规文法转NFA：略

|  |  |
| --- | --- |
|  | 转换函数 |
|  |  |
|  |  |

## NFA转正规文法：略

|  |  |
| --- | --- |
|  | 产生式 |
|  |  |
| 可接受状态 |  |

## 正规式转正规文法

|  |  |
| --- | --- |
| 正规式产生式 | 替换为 |
|  |  |
|  |  |
|  |  |

## 正规文法转正规式

结点：非终结符、右侧没有非终结符的产生式。

边：右侧有非终结符的产生式。

|  |  |
| --- | --- |
| 产生式 | 正规式产生式 |
|  |  |
|  |  |
|  |  |

|  |
| --- |
| 用规则1消掉所有自环：n1个A→有A用规则3合并成1个A→有A。n2个A→没有A。用规则1，结果是n2个新的A→没有A。  对每个结点：  　　用规则2去掉这个结点  　　用规则1消掉所有自环 |

## 正规式转NFA：略

<https://en.wikipedia.org/wiki/Thompson%27s_construction>

## NFA转正规式

<https://cs.stackexchange.com/questions/2016/how-to-convert-finite-automata-to-regular-expressions>

|  |
| --- |
| 多个开始状态的改成1个开始状态，多个结束状态的改成1个结束状态。  对每个结点对(A,B)：  　　若A==B：自环{ε}  　　若A!=B：没边就是∅，有边要用“|”（∪）把重边合并  对每个结点A，A!=开始状态，A!=结束状态：  　　对每个结点对(B!=A,C!=A)：  　　　　边B→C |=（∪=） 边B→A·(边A→A)\*·边A→C  　　　　边B→B同理  　　　　边C→B同理  　　　　边C→C同理 |

# 语法分析

## LL（1）文法的判别

假设文法中不包含多余规则和有害规则。

①判断每个非终结符是否能推出ε

|  |
| --- |
| 对每个产生式A→α：  　　若α==ε：  　　　　A能推出空串，A入队  　　　　（所有A→…不再有用）删除所有A→…  每个vT入队  当队非空：  　　v=出队  　　若v是非终结符不能推出空串/v是终结符：  　　　　对每个产生式A→α：  　　　　　　若A→α右侧有v：  　　　　　　　　（A→α不可能推出空串）删除A→α  　　　　　　　　若A的（可能推出空串的）产生式数量降为0：  　　　　　　　　　　A不能推出空串，A入队  　　若v是非终结符能推出空串：  　　　　对每个产生式A→α：  　　　　　　若v==A：  　　　　　　　　（A→α不再有用）删除A→α  　　　　　　若v!=A：  　　　　　　　　删除α中的v  　　　　　　　　若α==ε：  　　　　　　　　　　A能推出空串，A入队  　　　　　　　　　　（所有A→…不再有用）删除所有A→… |

②计算每个产生式右部的FIRST集

根据定义计算、由关系图法计算

计算每个非终结符（、终结符）的FIRST集

|  |
| --- |
| 结点：非终结符、终结符  边：刚开始没有边。u→v表示v的开始符号u也要有。  对每个产生式：  　　若产生式是A→至少0个[能推出空串的非终结符] [不能推出空串的非终结符/终结符]…：  　　　　添加边：A→那几个非终结符  　　　　添加边：A→那个非终结符/终结符  　　若产生式是A→至少0个[能推出空串的非终结符]：  　　　　添加边：A→那几个非终结符  对每个非终结符A：  　　若第①步结果中A能推出空串：  　　　　ε加入A的FIRST集中 |

计算每个产生式右部的FIRST集

|  |
| --- |
| 若产生式右部是 至少0个[能推出空串的非终结符] [不能推出空串的非终结符/终结符]…：  　　那几个非终结符的(FIRST集-{ε})∪那个非终结符/终结符的FIRST集  若产生式右部是 至少0个[能推出空串的非终结符]：  　　那几个非终结符的FIRST集∪{ε} |

③计算每个非终结符的FOLLOW集

根据定义计算、由关系图法计算

|  |
| --- |
| 结点：非终结符的FOLLOW集、非终结符的FIRST集、终结符、#  边：刚开始没有边  添加边：FOLLOW(S)→#  对每个产生式：  　　若产生式是A→…B 至少0个[能推出空串的非终结符] [不能推出空串的非终结符/终结符]…：  　　　　添加边：FOLLOW(B)→FIRST(那几个非终结符)  　　　　添加边：FOLLOW(B)→FIRST(那个非终结符/终结符)  　　若产生式是A→…B 至少0个[能推出空串的非终结符]：  　　　　添加边：FOLLOW(B)→FIRST(那几个非终结符)  　　　　添加边：FOLLOW(B)→FOLLOW(A) |

④计算每个产生式的SELECT集：略

## 非LL（1）文法到LL（1）文法等价变换

提取左公共因子：略

消除直接左递归

消除间接左递归

假设文法中不含回路。

|  |
| --- |
| 结点：非终结符  边：产生式A→B…，就添加边A→B  当图中有圈A1→A2→…→An→A1，n>=2时：  　　产生式An→A1…代入A(n-1)→An…，得到临时产生式A(n-1)→A1…  　　…  　　产生式A2→A1…代入A1→A2…，得到临时产生式A1→A1…  　　将产生式An→A1…替换为临时产生式A1→A1…  　　消除直接左递归 |

## 简单优先分析法，构造优先关系矩阵

|  |
| --- |
| 仿照[2.1](#_LL（1）文法的判别)求出所有文法符号的FISRT集和LAST集。  对每个产生式：  　　对产生式右部每两个相邻符号v1、v2：  　　　　v1=v2  　　　　v1<v2的FIRST集中每个符号  　　　　v1的LAST集中每个符号>v2、v2的FIRST集中每个符号 |

## 简单优先分析法：略

## 算符优先分析法，构造算符优先关系表

|  |
| --- |
| 仿照[2.1](#_LL（1）文法的判别)求出所有非终结符的FISRTVT和LASTVT。  对每个产生式：  　　对产生式右部每两个相邻符号vT1vT2或vT1vNvT2：  　　　　vT1=vT2  对每个产生式：  　　对产生式右部每两个相邻符号vTvN：  　　　　vT<vN的FIRSTVT中每个终结符  对每个产生式：  　　对产生式右部每两个相邻符号vNvT：  　　　　vN的LASTVT中每个符号>vT |

## 算符优先分析算法：略

## LR（0）分析，构造识别活前缀的FA

### 通过计算可归前缀

仿照[2.](#_非LL（1）文法到LL（1）文法等价变换)2

消除自环：若自环是（A→A，α），则A的LC集|=（∪=）α\*

消除环

|  |
| --- |
| 结点：非终结符和它的LC集，一开始只有S'有{ε}  边：产生式A→αB…，就添加边（B→A，α）  当图中有圈A1→A2→…→An→A1，n>=2时：  　　产生式An→αnA1…代入A(n-1)→α(n-1)An…，得到临时产生式A(n-1)→α(n-1)αnA1…  　　…  　　产生式A2→α2…α(n-1)αnA1…代入A1→α1A2…，得到临时产生式A1→α1…α(n-1)αnA1…  　　将产生式An→αnA1…替换为临时产生式A1→α1…α(n-1)αnA1…  　　消除自环 |

从S'出发，深度优先（现在图中已经没有环了）（、广度优先）

### 通过列出所有项目：略

### 通过构造G'的LR（0）项目集规范族：略

# 语义分析

## 基于S-属性文法的语义计算：略

（自底向上，LR分析）

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 状态栈 | 符号栈 | 语义栈 |  | 状态栈 | 符号栈 | 语义栈 |  | 状态栈 | 符号栈 | 语义栈 |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | 的综合属性 |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  | 的综合属性 |  |  |  |  |
|  |  |  |  |  |  | 的综合属性 |  |  |  | 的综合属性 |
|  |  |  |  |  |  |  |  |  |  |  |

## 基于L-属性文法的语义计算

自顶向下深度优先从左至右遍历分析树：在一个结点上调用访问函数前要给继承属性（相当于输入），调用访问函数后得到综合属性（相当于输出）。

下推栈的表驱动预测分析，略。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 下推栈 | 语义栈 |  | 下推栈 | 语义栈 |  | 下推栈 | 语义栈 |  | 下推栈 | 语义栈 |
|  |  |  | 计算的继承属性 |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | 计算的继承属性 |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | （的属性） |  |  | 的属性 |  |  |  |
|  |  |  | 计算的继承属性 | （的属性） |  |  | 的属性 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | 计算的综合属性 | （的属性） |  | 计算的综合属性 | 的属性 |  |  |  |
|  |  |  |  |  |  |  |  |  |  | … |

## 基于S-翻译模式的语义计算

和S-属性文法相同。

## 基于L-翻译模式的语义计算

### 自顶向下

### 自底向上

<https://stackoverflow.com/questions/243383/why-cant-c-be-parsed-with-a-lr1-parser>