编译原理复习

Garone Lombard

2023 年 12 月 20 日

摘要

编译原理烤漆复习手册

目录

1 绪论

编译程序概念 用高级语言编制的程序，计算机不能立即执行，必须通过一个“翻译程序”加工，转化为与其等价的机器语言程序，机器才能执行。这种翻译程序，称之为**编译程序**

源程序 用汇编语言或高级语言编写的程序称为源程序

目标程序 用**目标语言**所表示的程序

目标语言 可以是介于源语言和机器语言之间的“中间语言”，可以是某种机器的机器语言也可以是某机器的汇编语言

翻译程序 将源程序转换为目标程序的程序称为翻译程序，指各种语言的翻译器，包括汇编程序和编译程序，是汇编程序、编译程序以及各种变换程序的总称

汇编 若源程序用汇编语言书写，经过翻译程序得到用机器语言表示的程序，这时的翻译程序就称之为汇编程序，这种翻译过程称为汇编

编译 若源程序是用高级语言书写，经加工后得到目标程序，这种翻译过程称“编译”(Compile)

编译程序 5 阶段都要做的事 符号表管理 + 错误处理

遍 对源程序（包括源程序中间形式）从头到尾扫描一次，并做有关的加工处理，生成新的源程序中间形式或目标程序，通常称之为一遍。

生成中间代码的目的 便于优化处理、便于编译程序的移植

扩充的 BNF 表示 就是比 BNF 多了一些符号，比如 [...] 表示可选项，{...} 表示重复 0 次或多次，(...) 表示 0 次或 1 次

2 文法和语言

文法定义 四元组

$$G = (V_T, V_N, P, S) \tag{1}$$

2.1 文法的分类

0 型文法 无限制文法，只要求产生式的左部存在一个非终结符即可。

$$\forall \alpha \to \beta \in P \qquad \alpha\text{中至少包含一个}V_N \tag{2}$$

1 型文法 上下文有关文法，在0 型文法的基础上 进一步要求产生式的左部长度小于等于右部长度

$$\begin{aligned} \forall \alpha \to \beta \in P \qquad & |\alpha| \le |\beta| \\ \text{产生式的一般形式} \quad & \alpha 1 A \alpha 2 \to \alpha 1 \beta \alpha 2 (\beta \neq \epsilon) \end{aligned} \tag{3}$$

2 型文法 上下文无关文法 (**可以描述大部分程序设计语言的文法构造**)，要求产生式的左部只能是一个非终结符

$$\forall \alpha \to \beta \in P \qquad \alpha \in V_N \tag{4}$$

3 型文法 正则文法，只有两种形式，左线性文法 or 右线性文法 (注意是要求某一文法的所有产生式均符合左/右线性文法，而不只是 P1 满足左线性，P2 满足右线性)。
正则文法和正则表达式是等价的，对于任意一个正则文法 G，都存在定义同一语言的正则表达式 r，反之亦然。

- 左线性文法：$A \to \omega B \quad or \quad A \to \omega$
- 右线性文法：$A \to B\omega \quad or \quad A \to \omega$

例如

$$S \to a|b|c|d$$
$$S \to aT|bT|cT|dT$$
$$T \to a|b|c|d|0|1|2|3|4|5$$
$$T \to aT|bT|cT|dT|0T|1T|2T|3T|4T|5T$$

2.2 (句型的) 短语

已知文法

$$E \to E + E$$
$$E \to E * E$$
$$E \to -E$$
$$E \to (E)$$
$$E \to idenfr$$

对于句型：$-(E + E)$，可构造如下分析树

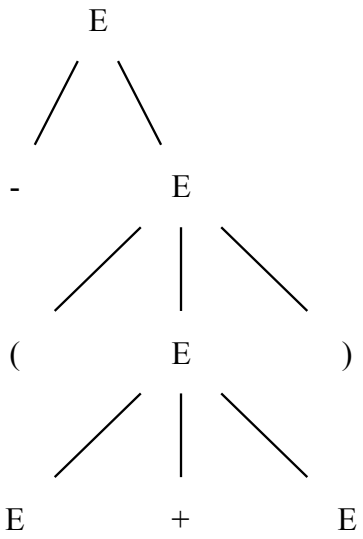


图 1: 分析树

- $E + E$ 是句型 $-(E + E)$ 相对于规则 $E \to E + E$ 的短语, 直接短语, 句柄 (子树层级为 1)
- $(E + E)$ 是句型 $-(E + E)$ 相对于规则 $E \to (E)$ 的短语
- $-(E + E)$ 是句型 $-(E + E)$ 相对于规则 $E \to -E$ 的短语

需要注意的是, 直接短语一定是某产生式的右部, 但某产生式的右部不一定是给定句型的直接短语

3 词法分析

3.1 有穷自动机

基本概念 省略...

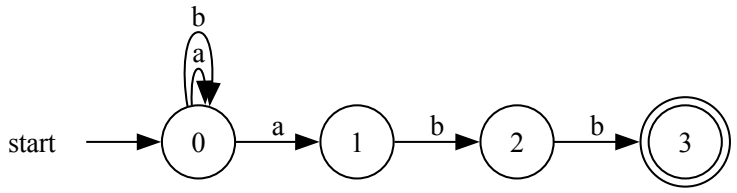


图 2: FA

最长前缀匹配原则 当输入串的多个前缀与一个或多个模式匹配时，总是选择最长的前缀匹配。也就是说在到达某个终态后，只要输入串上还有符号，FA 就会继续读入下一个符号，以寻求尽可能长度的匹配。

正则表达式和有穷自动机是等价的

3.1.1 确定的有穷自动机 (DFA)

定义 DFA 是一个五元组，$M = (S, \Sigma, \delta, s_0, F)$

- S：有穷状态集
- Σ：输入符号表
- δ：状态转移函数，$\forall s \in S, a \in \Sigma, \delta(s, a)$ 表示从状态 s 出发，沿着标记为 a 的边所能到达的状态 (**唯一**)
- s_0：初始状态
- F：终态**集**

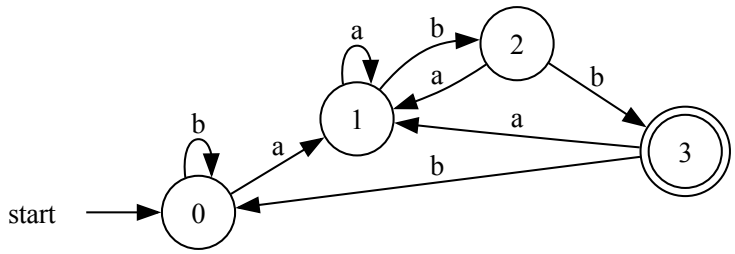


图 3: DFA

状态 \ 输入	a	b
0	1	0
1	1	2
2	1	3
3*	1	0

表 1: 转换表

DFA 的算法实现

- 输入：以文件结束符 eof 结尾的字符串 x，DFA M 的开始状态 s_0，接受状态集合 F，状态转换函数 $move(s, a)$
- 输出：M 接受则输出”yes”，拒绝则输出”no”
- 算法：

```
s=s0;
c=nextChar();
while(c!=eof){
    s=move(s,c);
    c=nextChar();
}
if(s in F) output("yes");
```

```
8 else output("no");
```

3.1.2 不确定的有穷自动机 (NFA)

定义 NFA 是一个五元组，$M = (S, \Sigma, \delta, s_0, F)$

- S：有穷状态集
- Σ：输入符号表
- δ：状态转移函数，$\forall s \in S, a \in \Sigma, \delta(s, a)$ 表示从状态 s 出发，沿着标记为 a 的边所能到达的状态**集合**
- s_0：初始状态
- F：终态**集**

NFA 和 DFA 的唯一区别就是状态转换函数的状态不唯一

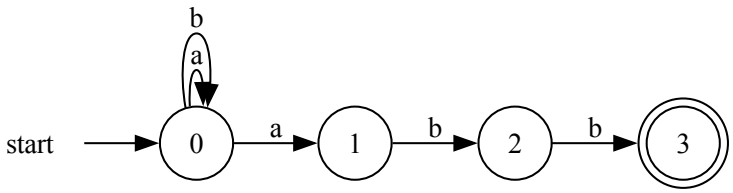


图 4: NFA

状态 \ 输入	a	b
0	{0,1}	{0}
1	ϕ	{2}
2	ϕ	{3}
3*	ϕ	ϕ

表 2: 转换表

DFA 和 NFA 具有等价性，即对于任意一个 NFA，都存在一个 DFA，使得两者能够识别相同的语言，反之亦然。

带有 ϵ 转换的 NFA ϵ-NFA，是一种特殊的 NFA，其状态转换函数 δ 中，$\delta(s, \epsilon)$ 表示从状态 s 出发，不读入任何输入符号，直接转移到下一个状态。

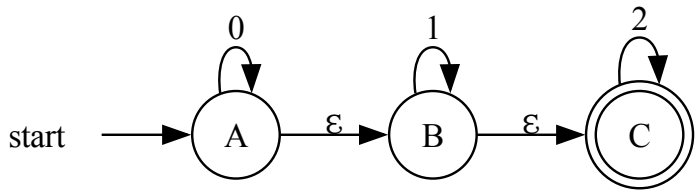


图 5: ϵ-NFA

可以证明，对于任意一个 ϵ-NFA，都存在一个 DFA，使得两者能够识别相同的语言，反之亦然。

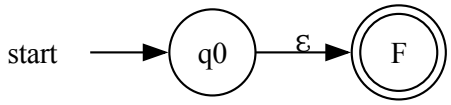
也就是说，*DFA*、*NFA*、*ϵ-NFA 都具有等价性*。

3.1.3 从正则表达式到 DFA 的转换

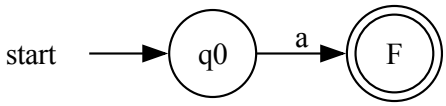
直接将正则表达式转换为 DFA 相当困难，所以一般采取 $RE- > NFA- > DFA$ 的形式

正则表达式到 NFA 的转换 对应关系如下

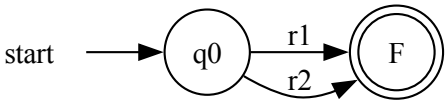
- ϵ 对应的 NFA



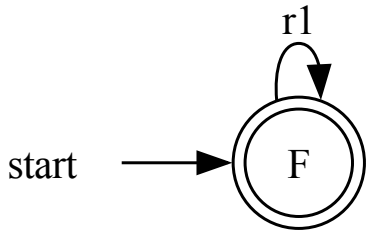
- 字母表 Σ 中符号 a 对应的 NFA



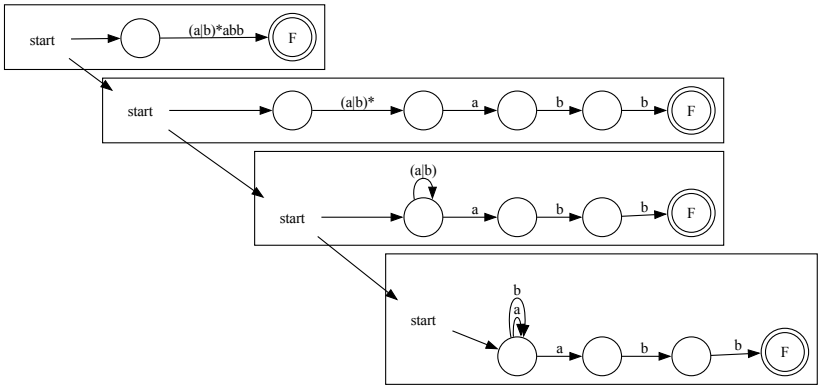
- $r = r_1 r_2$ 对应的 NFA



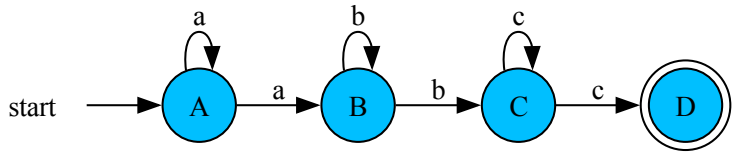
- $r = (r_1)*$ 对应的 NFA



- $r = (a|b)^*abb$ 对应的 NFA



NFA 到 DFA 的转换 如下所示

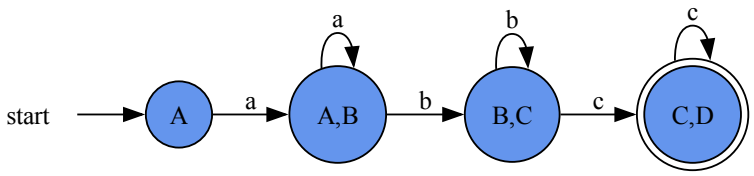


首先绘制状态转换表

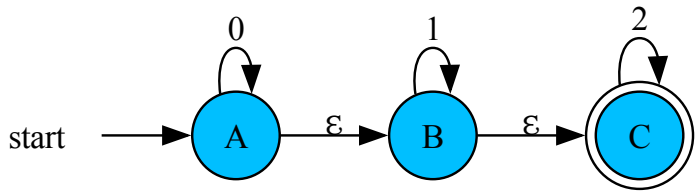
输入 / 状态	a	b	c
A	{A,B}	ϕ	ϕ
B	ϕ	{B,C}	ϕ
C	ϕ	ϕ	{C,D}
D*	ϕ	ϕ	ϕ

表 3: 转换表

与 NFA 等价的 DFA 的每一个状态都是一个由 NFA 状态构成的集合



ϵ-NFA 到 DFA 的转换

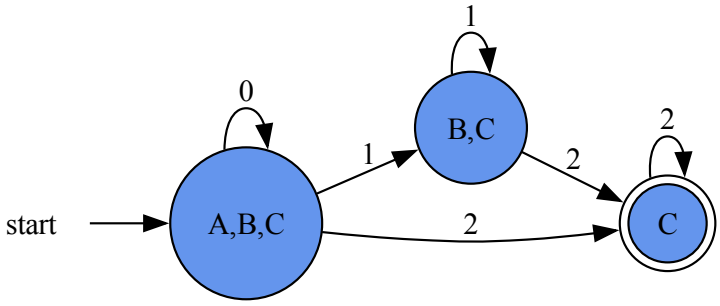


同样绘制状态转换表

输入 / 状态	0	1	2
A	{A,B,C}	{B,C}	{C}
B	ϕ	{B,C}	{C}
C*	ϕ	ϕ	{C}

表 4: 转换表

需要注意的是，由于初始即可达 A,B,C，所以初始状态应该是 A,B,C 而不是 A



子集构造法

- 输入：NFA N
- 输出：DFA D
- 算法：一开始，$\epsilon - closure(s_0)$ 是 Dstates 中唯一的状态，且未加标记；

```
while(在Dstates中有一个未标记状态T){
    tag(T);
    for(每个输入符号a){
        U=closure(move(T,a));
        if(U not in Dstates){
            add(Dstates,U);
        }
        Dtran[T,a]=U;
    }
}
```

操作	描述
$\epsilon - closure(s)$	能够从 NFA 的开始状态只通过 ϵ 转换直接到达的 NFA 状态集合
$\epsilon - closure(T)$	能从集合 T 中的某个 NFA 状态只通过 ϵ 转换直接到达的 NFA 状态集合
$move(T, a)$	能从集合 T 中的某个 NFA 状态通过标号为 a 的转换到达的 NFA 状态的集合

NFA 的化简例题 。。。。。。

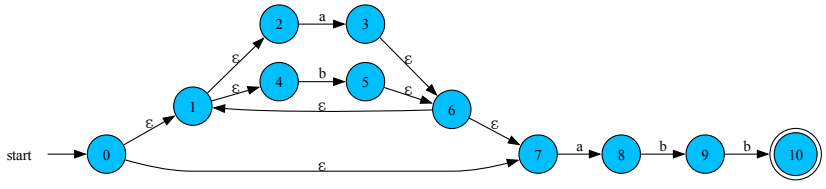


图 6: NFA

输入 / 状态	a	b
0	{1,2,3,4,6,7,8}	{1,2,4,5,6,7}
1	{1,2,3,4,6,7}	{1,2,4,5,6,7}
2	{1,2,3,4,6,7}	ϕ
3	{1,2,3,4,6,7,8}	{1,2,4,5,6,7}
4	ϕ	{1,2,4,5,6,7}
5	{1,2,3,4,6,7,8}	{1,2,4,5,6,7}
6	{1,2,3,4,6,7,8}	{1,2,4,5,6,7}
7	{8}	ϕ
8	ϕ	{9}
9	ϕ	{10}
10*	ϕ	ϕ

表 5: 转换表 1(无效)

$\epsilon - closure(T_0)$	{0,1,2,4,7}	
输入 / 状态	a	b
T0={0,1,2,4,7}	{1,2,3,4,6,7,8}	{1,2,4,5,6,7}
T1={1,2,3,4,6,7,8}	{1,2,3,4,6,7,8}	{1,2,4,5,6,7,9}
T2={1,2,4,5,6,7}	{1,2,3,4,6,7,8}	{1,2,4,5,6,7}
T3={1,2,4,5,6,7,9}	{1,2,3,4,6,7,8}	{1,2,4,5,6,7,10}
T4={1,2,4,5,6,7,10}	{1,2,3,4,6,7,8}	{1,2,4,5,6,7}

表 6: 转换表 2(正确)

输入 / 状态	a	b
T0={0,1,2,4,7}	T1	T2
T1={1,2,3,4,6,7,8}	T1	T3
T2={1,2,4,5,6,7}	T1	T2
T3={1,2,4,5,6,7,9}	T1	T4
T4*={1,2,4,5,6,7,10}	T1	T2

表 7: 转换表 2(正确)

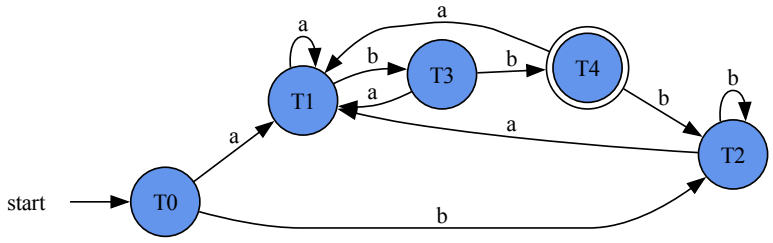


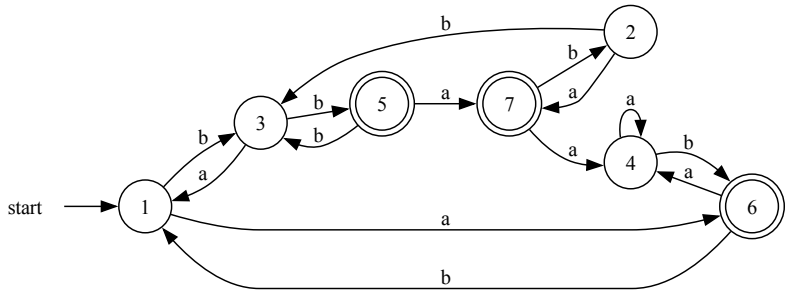
图 7: DFA

3.1.4 DFA 的最小化

概念 一个有穷自动机可以通过消除无用状态和合并等价状态来最小化

- 无用状态：从该自动机的开始状态出发，任何输入串都无法到达的状态 (从该状态出发，没有通路抵达终态)
- 等价状态：条件如下
  1. 一致性条件：状态 s 和 t 必须同时为可接受状态或不可接受状态
  2. 蔓延性条件：对于所有输入符号，状态 s 和 t 必须转换到等价态

分割法 把一个 DFA(不含无用态) 的状态分成一些不相交的子集，使得任何不同的两个子集的状态都是可区分的，且同一个子集中的任何状态都是等价的



第一步都是固定的,把状态分为终态和非终态两个集合 {1,2,3,4},{5,6,7}
接下来考察 {1,2,3,4} 是否可分

状态 \ 输入	a	b
1	6(E)	3(NE)
2	7(E)	3(NE)
3	1(NE)	5(E)
4	4(NE)	6(E)

因此可以将集合拆分为 {1,2},{3,4},{5,6,7}

状态 \ 输入	a	b
1	6(P3)	3(P2)
2	7(P3)	3(P2)

显然 {1,2} 不可拆分

状态 \ 输入	a	b
3	1(P1)	5(P3)
4	4(P2)	6(P3)

{3,4} 可拆分为 {3},{4}
此时集合为 {1,2},{3},{4},{5,6,7}

状态 \ 输入	a	b
5	7(P4)	3(P2)
6	4(P3)	1(P1)
7	4(P3)	2(P1)

{5,6,7} 可拆分为 {5},{6,7}
因此最终得到的集合为 {1,2},{3},{4},{5},{6,7}

状态 \ 输入	a	b
1	6(P5)	3(P2)
2	7(P5)	3(P2)
3	1(P1)	5(P4)
4	4(P3)	6(P5)
5	7(P5)	3(P2)
6	4(P3)	1(P1)
7	4(P3)	2(P1)

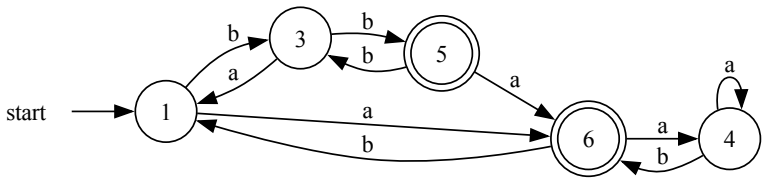


图 8: 最小化后的 DFA

4 语法分析

4.1 自顶向下语法分析

概念 从分析树的顶部向底部方向构造分析树，也就是从文法开始符号 S 从左向右推导句子 w 的过程

最左推导 总是选择每个句型的最左非终结符进行替换，其反过程称为最右规约

最右推导 总是选择每个句型的最右非终结符进行替换，其反过程称为最左规约

在自底向上的分析中，总是采用最左规约的方式，因此**把最左规约成为规范规约**，而**把最右推导称为规范推导**

最左推导和最右推导具备唯一性，因为对于每个句型而言，其最左/右终结符是唯一的

4.2 文法转换

左递归文法 如果一个文法中有一个非终结符 A 使得对某个串存在推导 $A \to^{+} Aa$，那么这个文法就是左递归文法，这会使递归下降分析器陷入无限循环

处理办法如下 (可消除直接左递归, 其实是将其转化为了右递归)

$$\begin{aligned} & A \to A\alpha|\beta \\ & A \to A\alpha \to A\alpha\alpha\alpha \to \beta\alpha\alpha\alpha\ldots \\ regex = {} & \beta\alpha^{*} \\ & A \to \beta A^{'} \\ & A^{'} \to \alpha A^{'}|\epsilon \end{aligned} \tag{5}$$

同理，对于左递归推导 $E \to E + T|T$，消除左递归可得一下等价文法

$$\begin{aligned} & E \to TE^{'} \\ & E^{'} \to +TE^{'}|\epsilon \end{aligned} \tag{6}$$

ϵ 产生式的使用时机 如果当前某终结符 A 与当前输入 a 不匹配时，若存在 $A \to \epsilon$，可以通过检查 a 是否可以出现在 A 的后面 (那不就是查看 A 的 FOLLOW 集吗?)，来决定是否可以使用产生式 $A \to \epsilon$

4.2.1 FIRST 集

概念 串首终结符，给定一个文法符号串 a，a 的 FIRST(a) 被定义为可以从 a 推导出的所有串首终结符的集合，如果 $a \Rightarrow^* \epsilon$，那么 ϵ 也在 FIRST(a) 中

4.2.2 FOLLOW 集

概念 可能在某个句型中紧跟在 A 后边的终结符 a 的集合

如果 A 是某个句型的最右符号，则将结束符 # 添加到 FOLLOW(A) 中

$$
\begin{aligned}
E &\to TE^{'} \\
E^{'} &\to +TE^{'}|\epsilon \\
T &\to FT^{'} \\
T^{'} &\to *FT^{'}|\epsilon \\
F &\to (E)|id
\end{aligned}
\tag{7}
$$

$$
\begin{aligned}
FIRST(E) &= \{(, id\} \\
FIRST(E^{'}) &= \{+, \epsilon\} \\
FIRST(T) &= \{(, id\} \\
FIRST(T^{'}) &= \{*, \epsilon\} \\
FIRST(F) &= \{(, id\}
\end{aligned}
\tag{8}
$$

$$
\begin{aligned}
FOLLOW(E) &= \{), \#\} \\
FOLLOW(E^{'}) &= \{), \#\} \\
FOLLOW(T) &= \{+,), \#\} \\
FOLLOW(T^{'}) &= \{+,), \#\} \\
FOLLOW(F) &= \{*, +,), \#\}
\end{aligned}
\tag{9}
$$

$$
\begin{aligned}
E &\to TE^{'} & SELECT(1) &= FIRST(T) = \{(, id\} \\
E^{'} &\to +TE^{'}|\epsilon & SELECT(2) &= \{+\} \\
E^{'} &\to \epsilon & SELECT(3) &= FOLLOW(E^{'}) = \{), \#\} \\
T &\to FT^{'} & SELECT(4) &= FIRST(F) = \{(, id\} \\
T^{'} &\to *FT^{'}|\epsilon & SELECT(5) &= \{*\} \\
T^{'} &\to \epsilon & SELECT(6) &= FOLLOW(T^{'}) = \{+,), \#\} \\
F &\to (E)|id & SELECT(7) &= \{(\} \\
F &\to id & SELECT(8) &= \{id\}
\end{aligned} \tag{10}
$$

4.2.3 SELECT 集

概念 产生式 $A \to \beta$ 的可选集是指可以选用该产生式进行推导时对应的输入符号的集合，记为 $SELECT(A \to \beta)$

- $SELECT(A \to \alpha\beta) = \{\alpha\}$
- $SELECT(A \to \epsilon) = FOLLOW(A)$

如果每个具有**相同左部**的各个产生式的可选集互不相交的话，就可以做出确定的分析

产生式 $A \to \alpha$ 的可选集 SELECT

- 如果 $\epsilon \notin FIRST(\alpha)$, 则 $SELECT(A \to \alpha) = FIRST(\alpha)$
- 如果 $\epsilon \in FIRST(\alpha)$, 则 $SELECT(A \to \alpha) = (FIRST(\alpha) - \{\epsilon\}) \cup FOLLOW(\alpha)$

4.2.4 LL(1) 文法

概念 文法 G 是 LL(1) 的，当且仅当 G 的任意两个具有相同左部的产生式 $A \to \alpha|\beta$ 满足下列条件

- 如果 α 和 β 均不能推导出 ϵ，则 $FIRST(\alpha) \cap FIRST(\beta) = \phi$
- α 和 β 至多只能有一个能推导出 ϵ(不能两个均能推导出 ϵ，那样的话 $SELECT(\alpha) \cap SELECT(\beta) = FOLLOW(A)$，也就是说 SELECT 集相交了)

1. 如果 $\alpha \Rightarrow^* \epsilon$, 则 $FIRST(\alpha) \cap FOLLOW(A) = \phi$
2. 如果 $\beta \Rightarrow^* \epsilon$, 则 $FIRST(\beta) \cap FOLLOW(A) = \phi$

总结一句话，就是 $SELECT$ 集别相交就行

4.3 自底向上语法分析

概念 自底向上的语法分析采用最左规约的方式

通用框架：**移入-归约分析**

$$\begin{aligned} E &\to E + E \\ E &\to E * E \\ E &\to (E) \\ E &\to id \end{aligned} \tag{11}$$

栈	剩余输入	动作
#	id+(id+id)#	
# id	+(id+id)#	移入
# E	+(id+id)#	归约:$E \to id$
# E+	(id+id)#	移入
# E+(	id+id)#	移入
# E+(id	+id)#	移入
# E+(E	+id)#	归约:$E \to id$
# E+(E+	id)#	移入
# E+(E+id	)#	移入
# E+(E+E	)#	归约:$E \to id$
# E+(E	)#	归约:$E \to E + E$
# E+(E)	#	移入
# E+E	#	归约:$E \to (E)$
# E	#	归约:$E \to E + E$

移入-归约分析中存在的问题 句柄的错误识别问题，需要使用 LR 分析法解决

$$\begin{aligned} < S > &\rightarrow var < IDS >:< T > \\ < IDS > &\rightarrow i \\ < IDS > &\rightarrow < IDS >, i \\ < T > &\rightarrow real|int \end{aligned} \tag{12}$$

栈	剩余输入	动作
#	var ia,ib:real#	
# var	ia,ib:real#	移入
# var ia	,ib:real#	移入
# var <IDS>	,ib:real#	归约:$< IDS > \rightarrow i$
# var <IDS>,	ib:real#	移入
# var <IDS>,ib	:real#	移入
# var <IDS>,<IDS>	:real#	归约:$< IDS > \rightarrow i$
# var <IDS>,<IDS>:	real#	移入
# var <IDS>,<IDS>:real	#	移入
# var <IDS>,<IDS>:<T>	#	归约:$T > \rightarrow real|int$
# var <IDS>,<IDS>:<T>	#	ERROR

4.3.1 LR 分析法

关键问题在于如何**正确的识别句柄**

$$S \rightarrow BBB \rightarrow aBB \rightarrow b \tag{13}$$

- sn：将符号 a、状态 n 压入栈
- rn：使用第 n 个产生式进行规约

状态	ACTION			GOTO	
	a	b	#	S	B
0	s3	s4		1	2
1			acpt		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

栈		剩余输入	动作
状态	符号		
0	#	bab#	
0 4	# b	ab#	
0	# B	ab#	
0 2	# B	ab#	
0 2 3	# Ba	b#	
0 2 3 4	# Bab	#	
0 2 3	# BaB	#	
0 2 3 6	# BaB	#	
0 2	# BB	#	
0 2 5	# BB	#	
0	# S	#	
0 1	# S	#	

```
// 输入符号串为w#,状态栈初始为0
// s代表栈顶状态
while(1){
    if(ACTION[s,a]==st){
        push_state(t);
        push_symbol(a);
    }
    else if(ACTION[s,a]=rt){// rt是归约A->b
        for(int i=0;i<|b|;i++){
```

```
10              pop_state();
11          }
12          push_state(GOTO(A,t));
13          push_symbol(A);
14          output(A->b);
15      }
16      else if(ACTION[s,a]=accept){
17          break;// 语法分析完成
18      }
19      else{
20          ERROR();
21      }
22  }
```

LR 分析的分析程序非常简单，关键是怎么构造给定文法的 LR 分析表

4.3.2 LR(0) 分析法

LR(0) 项目的概念 右部某位置标有原点的产生式称为相应文法的一个 LR(0) 项目

$$A \to \alpha_1 \cdot \alpha_2 \tag{14}$$

项目描述了句柄识别的状态

对于推导 $S \to bBB$

- $S \to \cdot bBB$ 点右侧是终结符，称为**移进项目**
- $S \to b \cdot BB$ 点右侧是非终结符，称为**待约项目**
- $S \to bB \cdot B$ 点右侧是非终结符，称为**待约项目**
- $S \to bBB \cdot$ 点位于产生式末尾，称为**归约项目**
- $A \to \epsilon$ 只对应一个项目 $A \to \cdot$

增广文法 如果 G 是一个以 S 为开始符号的文法，则 G 的增广文法 G' 就是在 G 中加上新开始符号 S' 和产生式 $S' \to S$ 而得到的文法

$$
\begin{aligned}
&E \to E + T \\
&E \to T \\
&T \to T * F \\
&T \to F \\
&F \to (E) \\
&F \to id
\end{aligned}
\tag{15}
$$

等价增广文法如下，引入这个新的开始产生式的目的是使得文法开始符号仅出现在一个产生式的左部，从而使得分析器只有一个接受状态

$$
\begin{aligned}
&E^{'} \to E \\
&E \to E + T \\
&E \to T \\
&T \to T * F \\
&T \to F \\
&F \to (E) \\
&F \to id
\end{aligned}
\tag{16}
$$

后继项目 同属于一个产生式的项目，但圆点的位置只相差一个符号，则称后者是前者的后继项目

比如 $S \to \alpha b \cdot \beta$ 就是 $S \to \alpha \cdot b\beta$ 的后继项目

可以把所有等价的项目组成一个项目集，称为项目集闭包，每个项目集闭包对应着自动机的一个状态

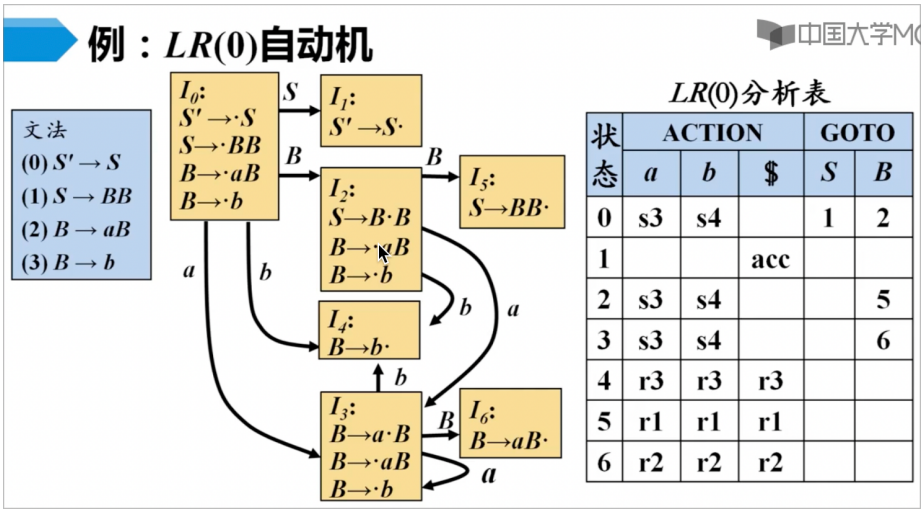


图 9: LR(0) 项目集

如果 LR(0) 分析表中没有语法分析动作冲突，那么给定的文法就称为 LR(0) 文法

4.3.3 SLR 分析法

解决了部分语法分析动作冲突的问题，但是无法解决移入归约冲突 (移入符 α 和 $FOLLOW(X)$ 的冲突问题)

因为 SLR 分析只是简单的考察了下一个输入符号 b 是否属于与归约项目 $A \to \alpha$ 相关联的 $FOLLOW(A)$，但 $b \in FOLLOW(A)$ 只是归约 a 的一个必要条件，而非充分条件

对于产生式 $A \to \alpha$ 的归约，在不同的使用位置，A 会要求不同的后继符号，也就是说对于不同位置，A 的后继符号集合应该是 $FOLLOW(A)$ 的子集

4.3.4 LR(1) 分析法

规范 LR(1) 项目：将一般形式为 $[A \to \alpha \cdot \beta, a]$ 的项称为 $LR(1)$ 项，其中 $A \to \alpha\beta$ 是一个产生式，a 是一个终结符。它表示在当前状态下，A 后面要求紧跟的终结符，称为该项的展望符

4.3.5 算符优先语法分析

基本思路 只考虑算符之间的优先关系，也就是只考虑终结符之间的优先关系

$$\begin{aligned} G[E]:\ & E \to T|E+T|E-T \\ & T \to F|T*F|T/F \\ & F \to (E)|i \end{aligned} \tag{17}$$

设有文法 G，如果 G 中没有形如 $A \to ...BC...$ 的产生式，其中 B,C 为**非终结符**，则称 G 为*算符文法*

设 G 是一个不含 ϵ 产生式的算符文法，a 和 b 是任意两个终结符，A,B,C 是非终结符，算法优先关系 $\doteq$ <·> 定义如下

1. $a \doteq b$ 当且仅当 G 中含有形如 $A \to ...ab...$ 或 $A \to ...aBb...$ 的产生式
2. $a \lessdot b$ 当且仅当 G 中含有形如 $A \to ...aB...$ 的产生式且 $B \Rightarrow^+ b...$ 或 $B \Rightarrow^+ Cb...$，也就是说 $a \lessdot FIRSTVT(B)$
3. $a \gtrdot b$ 当且仅当 G 中含有形如 $A \to ...Bb...$ 的产生式且 $B \Rightarrow^+ ...a$ 或 $B \Rightarrow^+ ...aC$，也就是说 $LASTVT(B) \gtrdot b$

设 G 是一个不含 ϵ 产生式的算符文法，如果任一终结符对 (a,b) 之间*至多*有 $\doteq$ <·> 三种关系中的一种成立，则称 G 是一个**算符优先文法**

需要注意的是，终结符之间的优先关系是有序的，也就是说 (a,b) 和 (b,a) 是两个终结符对，允许存在 $a \lessdot b, b \doteq a$ 同时存在

根据定义判断算符优先文法的优先级简直是神经质的，那么如何方便的做到这一点呢?

方法 ：求解 FIRSTVT 集和 LASTVT 集

- $FIRSTVT(B) = \{b|B \Rightarrow^+ b...\ or\ B \Rightarrow^+ Cb...\}$
- $LASTVT(B) = \{a|B \Rightarrow^+ ...a\ or\ B \Rightarrow^+ ...aC\}$

构造规则如下

- FIRSTVT

1. 若有 $T \to a...$ 或 $T \to Ra...$，则 $a \in FIRSTVT(T)$
2. 若有 $a \in FIRSTVT(R)$ 且有产生式 $T \to R...$,则 $a \in FIRSTVT(T)$
3. 迭代

- LASTVT

1. 若有 $T \to ...a$ 或 $T \to ...aR$，则 $a \in LASTVT(T)$
2. 若有 $a \in LASTVT(R)$ 且有产生式 $T \to ...R$,则 $a \in LASTVT(T)$
3. 迭代

举例文法

$$\begin{aligned} &E \to E+T|T \\ &T \to T*F|F \\ &F \to (E)|i \end{aligned} \tag{18}$$

F	+	*	(	)	i
E	1	1	1		1
T		1	1		1
F			1		1

L	+	*	(	)	i
E	1	1		1	1
T		1		1	1
F				1	1

算符优先分析表	+	*	(	)	i
+	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$
*	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$
)	$\gtrdot$	$\gtrdot$		$\gtrdot$	
i	$\gtrdot$	$\gtrdot$		$\gtrdot$	

$$
\begin{aligned}
G[S]:\ & S \to S;G|G \\
& G \to G(T)|H \\
& H \to a|(S) \\
& T \to T+S|S
\end{aligned}
\tag{19}
$$

给出上述符号串 **a;(a+a)** 的算符优先分析过程。

F	;	(	)	a	+
S	1	1		1	
G		1		1	
H		1		1	
T	1	1		1	1

L	;	(	)	a	+
S	1		1	1	
G			1	1	
H			1	1	
T	1		1	1	1

算符优先表	;	(	)	a	+
;	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	$\lessdot$
)	$\gtrdot$	$\gtrdot$	$\gtrdot$	err	$\gtrdot$
a	$\gtrdot$	$\gtrdot$	$\gtrdot$	err	$\gtrdot$
+	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$

栈	当前输入符	输入串剩余部分	下一步动作
#	a	;(a+a)#	移入
#a	;	(a+a)#	归约
#T	;	(a+a)#	移入
#T;	(	a+a)#	移入
#T;(	a	+a)#	移入
#T;(a	+	a)#	归约
#T;(T	+	a)#	移入
#T;(T+	a	)#	移入
#T;(T+a	)	#	归约
#T;(T+T	)	#	归约
#T;(T	)	#	移入
#T;(T)	#		归约
#T;T	#		归约
#T	#		移入
#T#			结束

5 语法制导翻译

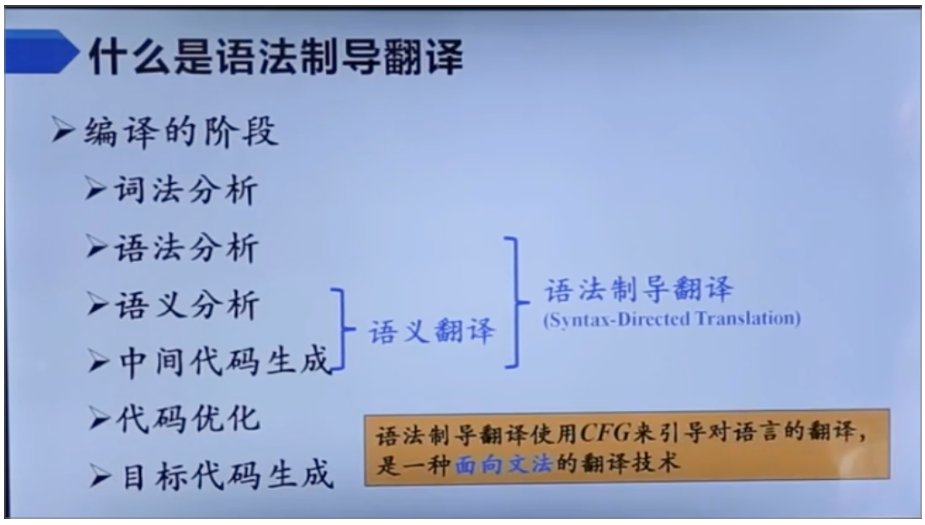


图 10: 语法制导翻译

如何表示语义信息 给 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 (比如说变量类型、值、存放地址等等)

如何计算语义属性 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 (语法规则) 相关联的语义规则来计算分析树中各节点的语义属性值

5.1 语法制导定义 SDD

SDD 是对 CFG 的拓展，它将每个文法符号和一个语义属性集合相关联，将每个产生式和一组语义规则相关联，这些规则用于计算该产生式中各文法符号的属性值

如果 X 是一个文法符号，a 是 X 的一个属性，那么 $X.a$ 表示 X 的属性 a 在某个标号为 X 的分析树节点上的值

文法符号的属性种类

- 综合属性：在分析树节点 N 上的非终结符 A 的综合属性只能通过 N 的子节点本身的属性值来定义
- 继承属性：在分析树节点 N 上的非终结符 A 的继承属性只能通过 N 的父节点、N 的兄弟节点、N 本身的属性值来定义

$$\begin{aligned} E &\to E_1 + T \\ E.val &= E_1.val + T.val \\ D &\to TL \\ L.inh &= T.type \end{aligned} \tag{20}$$

5.2 语法制导翻译 SDT

概念 SDT 是在产生式右部嵌入了程序片段的 CFG，这些程序片段称为语义动作。按照惯例，语义动作写在产生式右部的花括号中

$$D \to T\{L.inh = T.type\}L \tag{21}$$

这里的含义是，当分析出终结符 T 的时候，就可以把 T 的属性值赋给 L 的 inh 属性

6 中间代码生成

6.1 活动记录

一个典型的活动记录可以分为三部分：

1. 局部数据区：存放模块中定义的各个**局部变量**
2. 参数区：存放*隐式参数*和*显式参数 (形参数据区)*
   - prev abp: 存放调用模块记录基地址
   - ret addr: 返回地址
   - ret value: 函数返回值
3. display 区：**存放各外层模块活动记录的基地址**