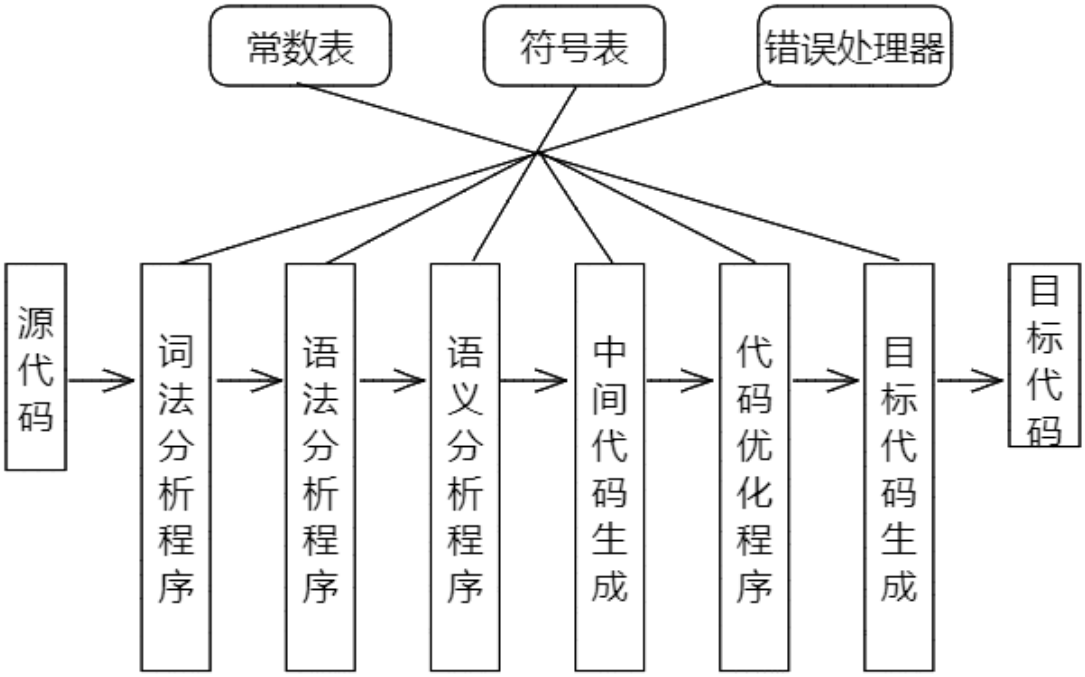
编译器逻辑阶段的划分

源代码->词法分析->语法分析->语义分析->中间代码生成->代码优化->目标代码生成->目标代码



正规表达式的定义和作用和正规集的定义

正规表达式：是用特定的运算符及运算对象按规则构造的表达式
正规表达式的作用：用有限的表达式去解决无限个字符串匹配的问题
正规集：每个正规表达式匹配（或代表、或表示）一个字符串的集合
设有字母表为Σ，辅助字母表Σ'={ϕ, ε, | , . , * , ( ) }，正规表达式和它所表示的正规集(字符串的集合)的递归定义如下：

ε 和 ϕ 是 Σ 上的正规式，它们所表示的正规集分别为{ε } 和{ };

若 a∈Σ，则 a 是 Σ 上的正规式，它所表示的正规集为{a};

若 r 和 s 是 Σ 上的正规式，它们所表示的正规集分别为 L(r)和 L(s)，则：

r|s 是正规式，表示的正规集为 L(r|s)=L(r)∪L(s) ；

rs 是正规式，表示的正规集为 L(rs)=L(r)L(s) ；

r*是正规式，表示的正规集为(L(r))*。

(r)是正规式，表示的正规集为 L(r);

有限次使用上述步骤 3 而定义的表达式是 Σ 上的正规式，由这些正规式所表示的符号串集合是 Σ 上的正规集。

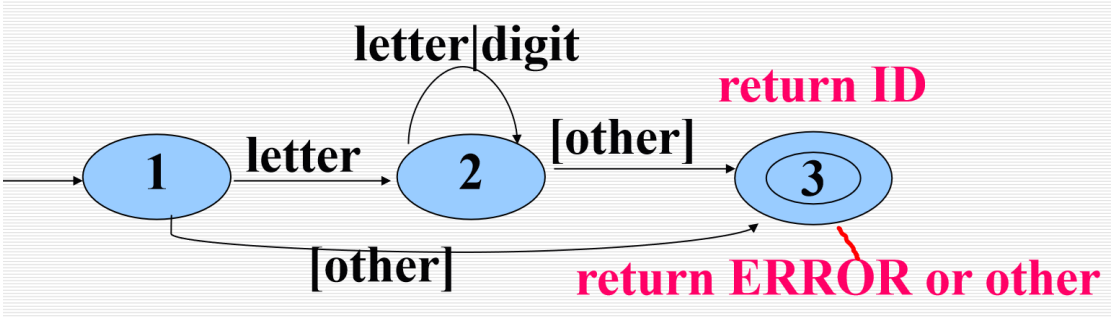
确定性有穷自动机的定义

DFA 的组成是一个五元组：

$$DFA = \{\Sigma, S, T, S_0, A\}$$

Σ：有限个输入符号组成的字母表
S：有限个状态的集合
$T: S \times \Sigma \to S$：转换函数
$s_0 \in S$：初始状态
$A \subseteq S$：终止状态

用代码实现有穷自动机




```
t = input()  # 输入字符串
t = list(t)

# 不写完整了
LETTER = ['a', 'b', 'c', 'd']
DIGIT = ['1', '2', '3', '4']
status = 1

for i in t:
    if status == 1:
        if i in LETTER:
            status = 2
        else:
            status = 3
            print('ERROR')
    elif status == 2:
        if i in LETTER or i in DIGIT:
            status = 2
        else:
            status = 3
            print('ID')
```

递归下降语法分析程序撰写

G[S]：S→aAcBe　　A→b　　B→d

```
void S(void)
{
    match(a);
    A();
    match(c);
    B();
    match(e);
}

void A(void)
{
    match(b);
}

void B(void)
{
    match(d);
}
```

S→a|^|(T)
T→SA
A→,SA|ε

```
void S(void)
{
    if token=a then
        match(a);
    if token=^ then
        match(^);
    if token=( then
    {
        match(();
        T();
        match());
    }
}
```

```
void T(void)
{
    S();
    A();
}
```

```
void A(void)
{
    if token=, then
    {
        match(,);
        S();
        A();
    }
}
```

ε不用识别

上下无关文法的定义，其在自动化编译中的作用

上下文无关文法 G 是一个四元组：$G = (V_T, V_N, P, S)$

V_T：终结符集合

V_N：非终结符集合

P：文法产生式$A \to a$集合，$A \in V_N, a \in (V_T \cup V_N)^*$

S：开始符号，$S \in V_N$

上下文无关文法可以帮助我们可以构造有效的分析算法来检验一个给定字串是否是由某个上下文无关文法产生的。并且很多程序设计语言都是通过上下文无关文法来定义的。

如何绘制语法树

通过文法的最左推导或者最右推导绘制，如果两种推导方式画出的树不同，则说明该文法有二义性。

文法G[E]:

E → E+E

E → E*E

E → (E)

E → i

字符串：i+i*i

最左推导:

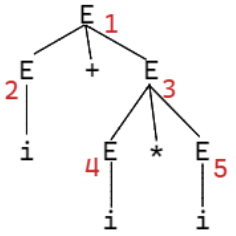
E → E+E

⇒ i+E

⇒ i+E*E

⇒ i+i*E

⇒ i+i*i



最右推导:

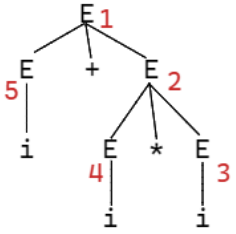
E → E+E

⇒ E+E*E

⇒ E+E*i

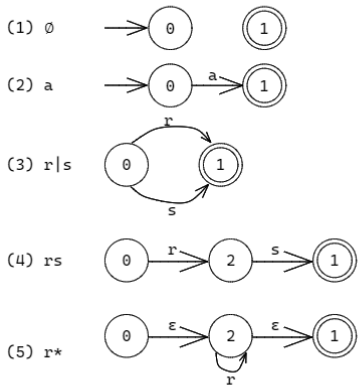
⇒ E+i*i

⇒ i+i*i

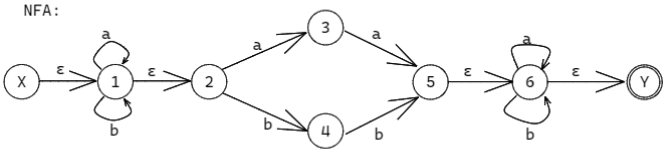


正规表达式转 NFA，再转 DFA

正规表达式(a|b)*(aa|bb)(a|b)*



NFA:



DFA:

如果初始状态的ε闭包包括终止状态，则该闭包也是终止状态

ε-closure({X,1,2}, a)

ε-closure(X)	a	b
{X,1,2}	{1,2,3}	{1,2,4}
{1,2,3}	{1,2,3,5,6,Y}	{1,2,4}
{1,2,4}	{1,2,3}	{1,2,4,5,6,Y}
{1,2,3,5,6,Y}	{1,2,3,5,6,Y}	{1,2,4,6,Y}
{1,2,4,5,6,Y}	{1,2,3,6,Y}	{1,2,4,5,6,Y}
{1,2,4,6,Y}	{1,2,3,6,Y}	{1,2,4,5,6,Y}
{1,2,3,6,Y}	{1,2,3,5,6,Y}	{1,2,4,6,Y}

如果新状态集包含终结态，则新状态集是终结态

First 集求法

设$\boldsymbol{G}=\{\boldsymbol{V_T},\boldsymbol{V_N},\boldsymbol{S},\boldsymbol{P}\}$是上下文无关文法

1 若$X \in V_T$，则$First(X)=\{X\}$

2 若$X \in V_N$，且有产生式$X \to a \ldots$，$a \in V_T$，则$a \in First(X)$

3 若$X \to \varepsilon$，则$\varepsilon \in First(X)$

4 若$X \to Y \ldots$，$Y \in V_N$，则$(First(Y)-\{\varepsilon\}) \in First(X)$

4.1 若$X \to Y_1 Y_2 \ldots Y_n$，$Y_i \in V_N$，且对$\forall 1 \le i \le m < n$，有$\varepsilon \in First(Y_i)$，则$\cup_{i=1}^{m}(First(Y_i)-\{\varepsilon\}) \in First(X)$

4.2 若$X \to Y_1Y_2 \dots Y_n$，$Y_i \in V_N$，且对$\forall 1 \le i \le n$，有$\varepsilon \in First(Y_i)$，则$\cup_{i=1}^{m}(First(Y_i)) \in First(X)$

Follow 集求法

设$G = \{V_T, V_N, S, P\}$是上下文无关文法

1 将{#}加入$Follow(S)$

2 若$X \to \mu A\beta$，则把$First(\beta) - \{\varepsilon\}$加入$Follow(A)$中；若有$\beta \to \varepsilon$，则把$Follow(X)$加入$Follow(A)$中

验证文法是否为 LL(1)

设$G = \{V_T, V_N, S, P\}$是上下文无关文法

定义：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：

1 若α, β均不能推导出ε，则$First(\alpha) \cap First(\beta) = \emptyset$

2 α和β至多有一个能推导出ε

3 若$\alpha \to \varepsilon$，则$First(\alpha) \cap Follow(A) = \emptyset$

算法：

1 消除左递归

2 计算 First 集和 Follow 集

3 按照定义验证

构造 LL(1)预测分析表 M

设$G = \{V_T, V_N, S, P\}$是上下文无关文法

表的竖行是文法的非终结符

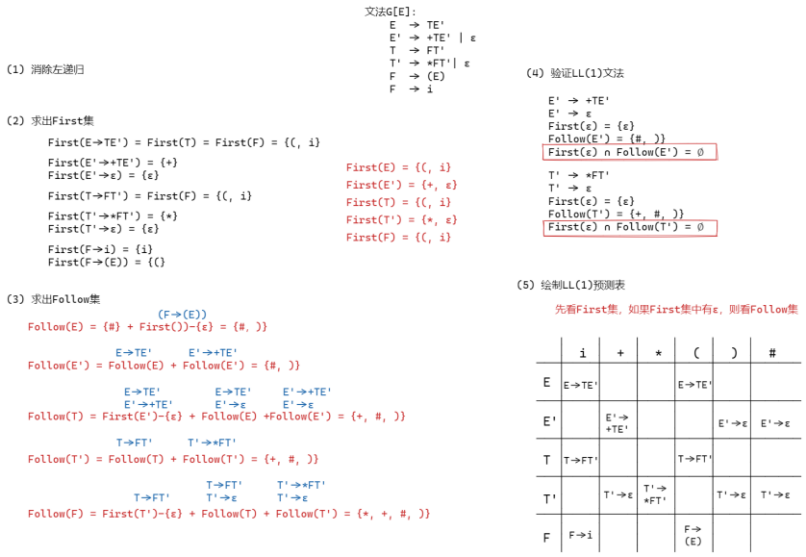
表的横行是文法的终结符和#

对于 G 中的$A \to \alpha$：

1 对每个$a \in First(\alpha)$，把$A \to \alpha$加入$M[A, a]$

2 若$\varepsilon \in First(\alpha)$，对任何$b \in Follow(A)$，把$A \to \varepsilon$加入$M[A, b]$

LL(1)文法完整分析过程



	i	+	*	(	)	#
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→i			F→(E)		

(6) LL(1)分析

输入串: i+i*i

分析栈	输入符号	动作
#E	i+i*i#	E→TE'
#E'T	i+i*i#	T→FT'
#E'T'F	i+i*i#	F→i
#E'T'i	i+i*i#	匹配 i
#E'T'	+i*i#	T'→ε
#E'	+i*i#	E'→+TE'
#E'T+	+i*i#	匹配 +
#E'T	i*i#	T→FT'
#E'T'F	i*i#	F→i
#E'T'i	i*i#	匹配 i
#E'T'	*i#	T'→*FT'
#E'T'F*	*i#	匹配 *
#E'T'F	i#	F→i
#E'T'i	i#	匹配 i
#E'T'	#	T'→ε
#E'	#	E'→ε
#	#	acc

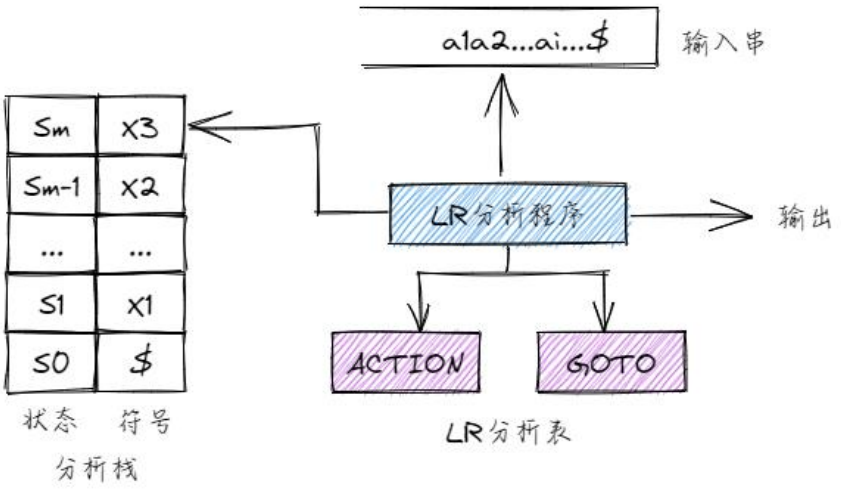
画出文法的 LR(0)状态机

先构造一个拓广初始状态$S' \rightarrow \cdot S$

若点·后面跟着一个非终结符，则把非终结符的产生递归加入状态中

将点·向后移动一个字符，代表移进该字符，跳入新状态，直到移进#，代表字符串被接受

LR 算法的流程图



LR(0)文法分析过程

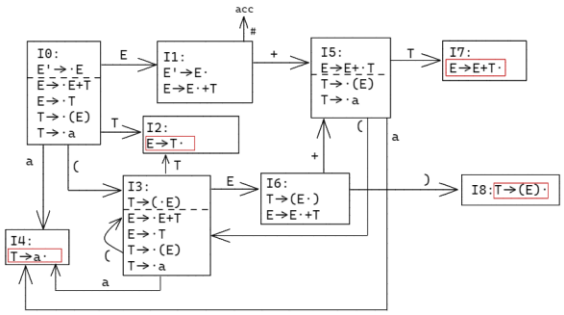
（下面的例子其实是 SLR（1）文法，构建分析表错了，但是分析步骤的思路是对的，就不改了）

文法G[E]:
E → E+T | T
T → (E) | a

(1) 拓广文法

由已知可得G[E']:
(0) E' → E
(1) E → E + T
(2) E → T
(3) T → (E)
(4) T → a

(2) 构建项目集DFA图



(3) 构建分析表

表的竖行是DFA的状态（项目集）
表的横行的ACTION是终结符和#，GOTO是非终结符

点·在最后面，代表项目结束
将该状态对应的拓广文法的
编号全部加入ACTION中

I0输入一个E，进入I1

状态	ACTION					GOTO	
	a	+	(	)	#	E	T
0	S4		S3			1	2
1		S5			acc		
2	r2	r2	r2	r2	r2		
3	S4		S3			6	2
4	r4	r4	r4	r4	r4		
5	S4		S3				7
6		S5		S8			
7	r1	r1	r1	r1	r1		
8	r3	r3	r3	r3	r3		

(4) LR(0)分析

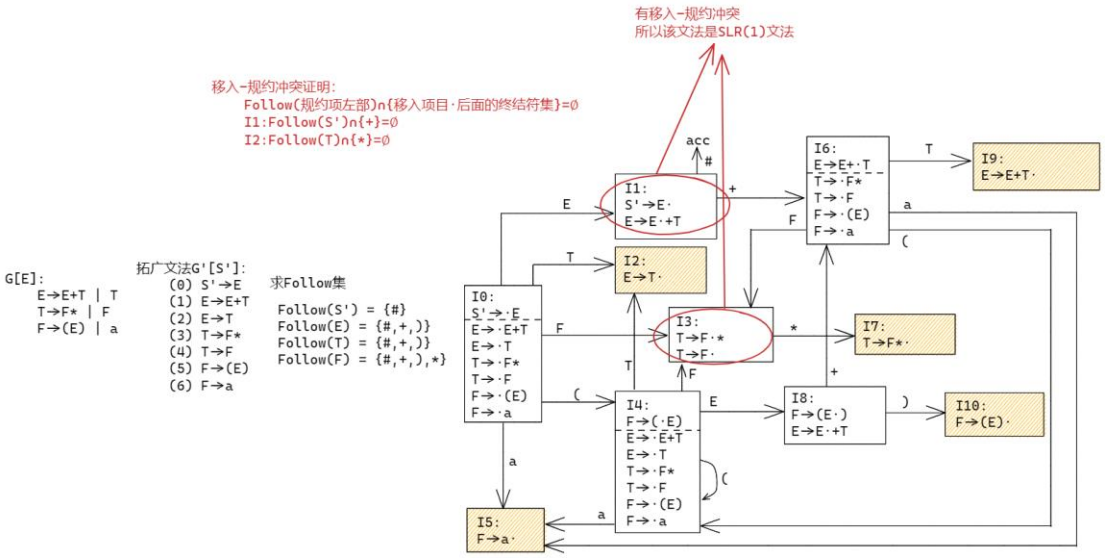
输入串：(a)

状态	分析栈	输入	动作	
0	#	(a)#	S3移入	← M[0,(]
03	#(	a)#	S4移入	
034	#(a	)#	r4规约	T→a
03	#(	T)#	goto 2	
032	#(T	)#	r2规约	E→T
03	#(	E)#	goto 6	
036	#(E	)#	S8移入	
0368	#(E)	#	r3规约	T→(E)
0	#	T#	goto 2	
02	#T	#	r2规约	E→T
0	#	E#	goto 1	
01	#E	#	acc	

弹出n个字符
状态就回退n步

SLR(1)文法分析过程

步骤同 LR(0)分析过程，但是多求一步 Follow 集



构建SLR(1)分析表

状态	Action						Goto		
	a	+	*	(	)	#	E	T	F
0	S5			S4			1	2	3
1		S6				acc			
2		r2			r2	r2			
3			S7						
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7		r3			r3	r3			
8		S6			S10				
9		r1			r1	r1			
10		r5	r5		r5	r5			

点·在最后面，表示项目结束
将该状态对应的拓广文法的编号
全部加入ACTION的
Follow(文法左项)中

LR(1)文法分析过程

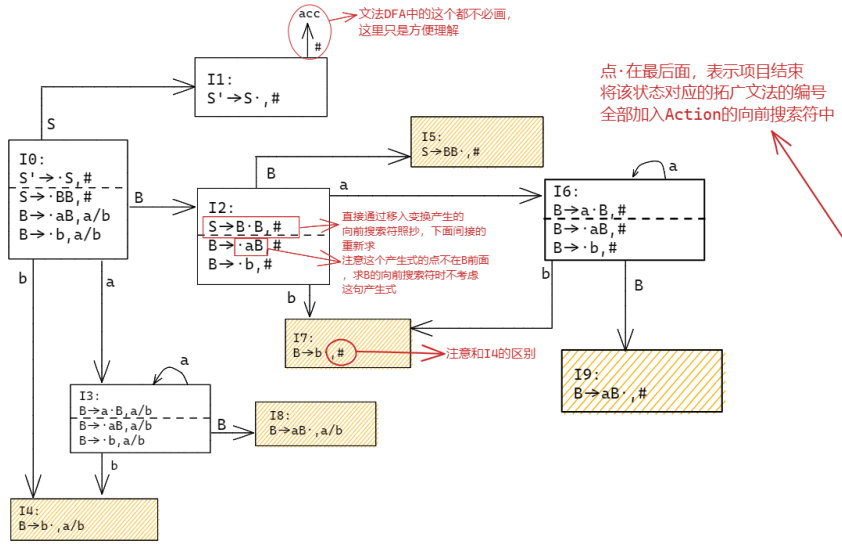
G[S]:
S→BB
B→aB | b

拓广文法G'[S']:
(0)S'→S
(1)S→BB
(2)B→aB
(3)B→b

构造带向前搜索符的DFA

(1) 文法开始符S'的Follow集为S'的向前搜索符
(2) 先看前面再看后面:
例如: A→α·Bβ, 前向搜索符(A)
B→ ...
要求前向搜索符(B)，则看前面的
产生式X的·后面有B，再看这个X的B后面的β。
如果β不能推出ε，则将first(β)加入前向搜索符(B);
如果β能推出ε，则将前向搜索符(A)也加入前向搜索符(B).

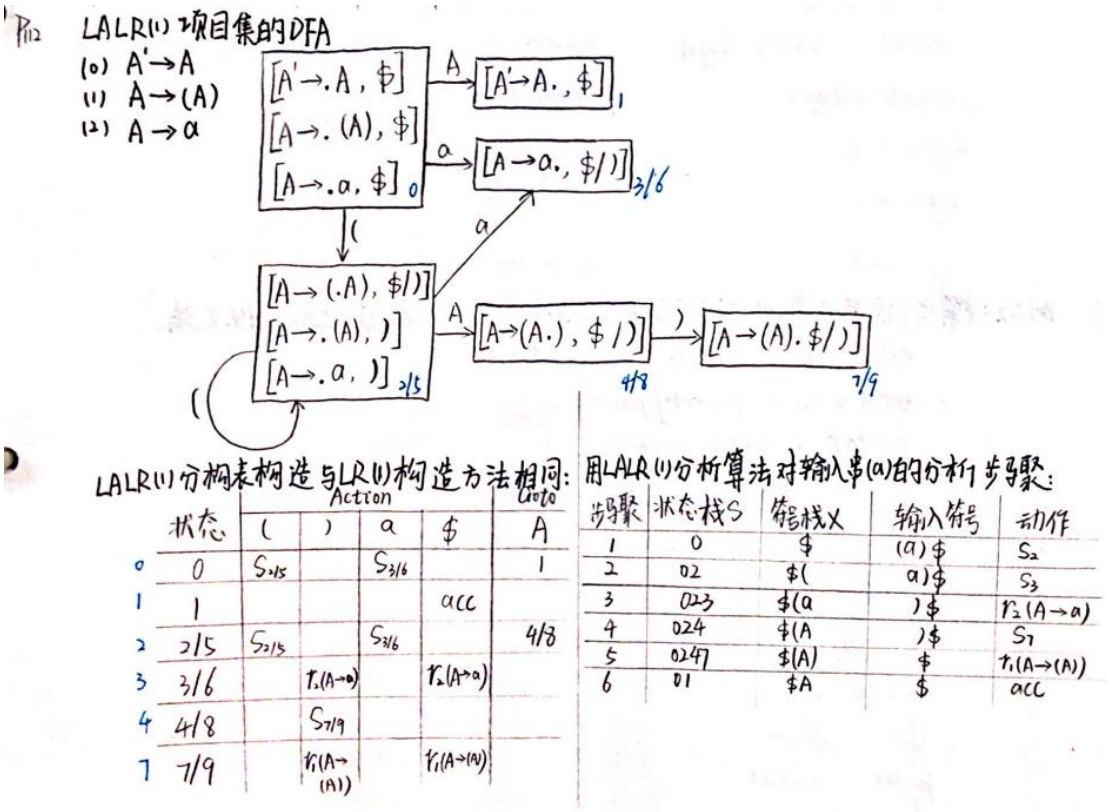
以I0为例:
S'→·S,#
S→·BB,#
B→·aB,a/b
B→·b,a/b
First(B)



状态	Action			Goto	
	a	b	#	S	B
0	S3	S4		1	2
1			acc		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

LALR(1)分析过程

在 LR(1)的基础上合并同心集。同心集：DFA 项目集中 LR(0)完全相同的项目集。



	状态	(	)	a	$	Goto A
0	0	S2/5		S3/6		1
1	1				acc	
2	2/5	S2/5		S3/6		4/8
3	3/6		r2(A→a)		r2(A→a)	
4	4/8		S7/9			
7	7/9		r1(A→(A))		r1(A→(A))	

步骤	状态栈S	符号栈X	输入符号	动作
1	0	$	(a)$	S_2
2	02	$(	a)$	S_3
3	023	$(a	)$	$r_2(A→a)$
4	024	$(A	)$	S_7
5	0247	$(A)	$	$r_1(A→(A))$
6	01	$A	$	acc

判断 LR 状态机是否存在冲突

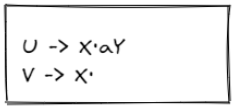
1 归约项目：$A \to \alpha \cdot$，点在产生式的最后面

2 接受项目：$S \to \alpha \cdot$，开始文法 S 对应的归约项目

3 移入项目：$A \to \alpha \cdot x\beta$，点后面跟着终结符$x$，下一步就是移入$x$

4 待约项目：$A \to \alpha \cdot X\beta$，点后面跟着非终极符

移入-归约冲突：若产生式的右部是另一产生式的前缀

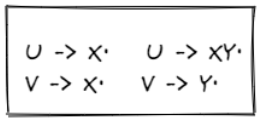


冲突产生条件：

1 两个产生式出现在自动机中的同一状态

2 $Follow(V) \cap \{a\} = \emptyset$

归约-归约冲突：若不同产生式有相同右部或产生式的右部是另一产生式的后缀



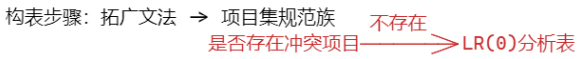
冲突产生条件：

1 两个产生式出现在自动机中的同一状态

2 $Follow(U) \cap Follow(V) = \emptyset$

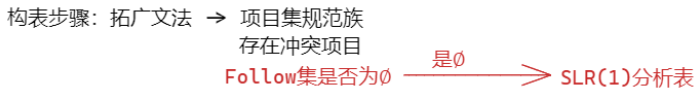
区分 LR(0)、SLR(1)、 LR(1)、 LALR(1)

LR(0)文法：不存在冲突项目（移入-归约、归约-归约）

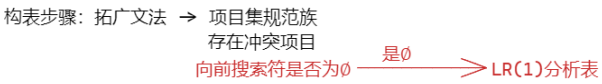


任何一个 LR(0)文法一定是一个 SLR(0)文法

SLR(1)文法：存在冲突项目（移入-归约）



LR(1)文法：存在冲突项目（移入-归约、归约-归约）



任何一个 SLR(1)文法一定是一个 LALR(1)文法

LALR(1)文法：合并同心集后无归约-归约冲突

什么是符号表？符号表有哪些重要作用？

符号表是用来记录编译过程中的各种信息的表格

符号表的作用表现为：

1 登记编译过程输入和输出信息；

2 在语义分析过程中用于语义检查和中间代码生成；

3 作为目标代码生成阶段地址分配的依据。

静态语法分析的任务

计算各类语法成分发语义信息（属性信息）

类型检查、上下文有关问题的检查、唯一性检查、控制流检查、Break 和 continue 语句是否在循环结构中、参数类型和数量是否匹配、数组下标是否越界、数组下标是否为整数

属性文法的定义

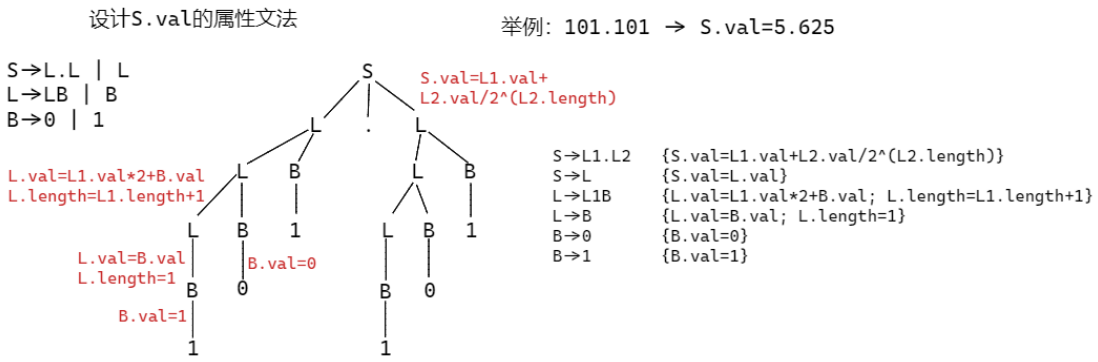
属性文法是个三元组：$A=(G,V,F)$

G：上下文无关语言

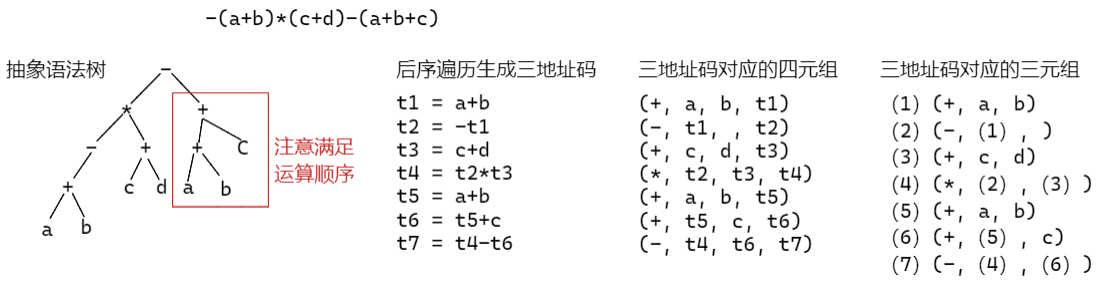
V：属性的有限集合

F：有关属性的语义规则的有限集合

求文法的属性文法



简单赋值语录的翻译



x[i]:=y的三元组：

(0) ([]=,x,i)

(1) (assign, (0) , y)

x:=y[i]的三元组：

(0) (=[], y, i)

(1) (assign,x, (0))

划分基本块并画出流程图

1. 求出四元式程序中各个基本块的入口语句：

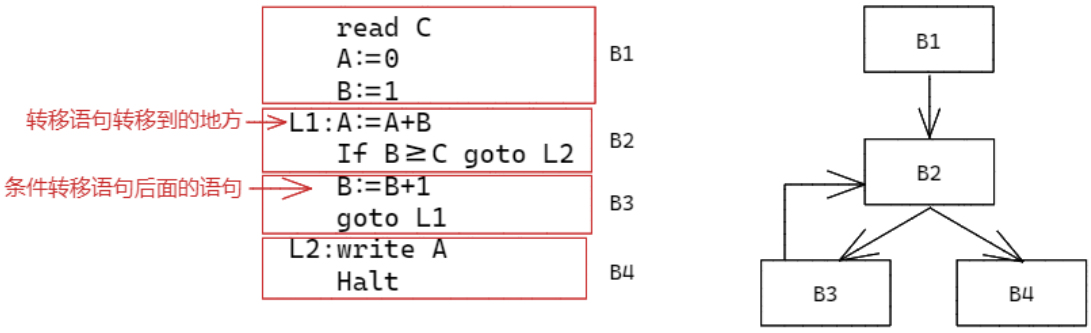
（1）程序第一个语句，或

（2）能由条件转移语句或无条件转移语句转移到的语句，或

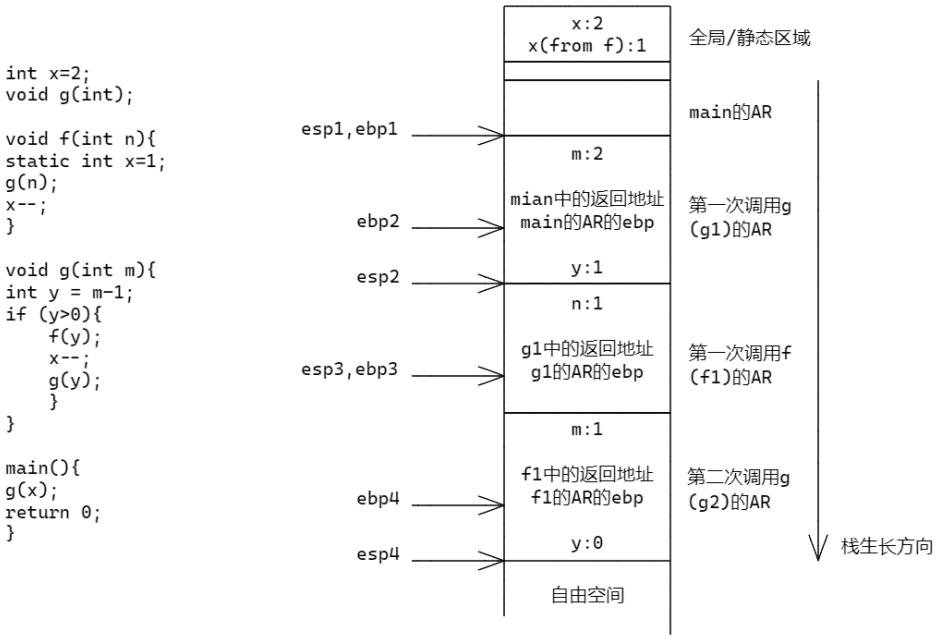
（3）紧跟在条件转移语句后面的语句

2. 对以上求出的每个入口语句，确定其所属的基本块。它是由该入口语句到下一入口语句（不包括该入口语句）、或到一转移语句（包括该转移语句）、或一停语句（包括该停语句）之间的语句序列组成的。

3. 凡未被纳入某一基本块中的语句，可以从程序中删除



程序执行时的基于栈的运行时环境



fp 指针的作用

fp 是控制链指针，即存储当前活动记录的控制链（即一个地址），作用如下：

1. 通过该指针可以访问主调函数的活动记录。即允许当前的被调函数执行完毕时，用它来恢复主调函数的活动记录。

2. 通过该指针可以访问当前执行函数的实参和局部变量。

构建基本块的 DAG

1. 准备操作数的结点

如果$NODE(B)$无定义，则构造一标记为 B 的叶结点，并定义$NODE(B)$为这个结点：

如果当前四元式是 0 型（$A := n$），则记$NODE(A)$的值为n，转 **4**；

如果当前四元式是 1 型（$A := op\ B$），则转 **2(1)**；

如果当前四元式是 2 型（$A := B\ op\ C$），则

(1) 如果$NODE(C)$无定义，则构造一标记为C的叶结点并定义这个结点

(2) 转 **2(2)**；

2. 合并已知量

(1) 如果$NODE(B)$是标记为常数的叶结点，则转 **2(3)**；否则转 **3(1)**；

(2) 如果$NODE(B)$和$NODE(C)$都是标记为常数的叶结点，则转 **2(4)**；否则转 **3(2)**；

(3) 执行$op\ B$（即合并已知量）。令得到的新常数为P。如果$NODE(B)$是处理当前四元式时新构造出来的结点，则删除它。如果$NODE(P)$无定义，则构造一用P作为标记的叶结点n，置$NODE(P) = n$，转 **4**；

(4)执行$B\ op\ C$（即合并已知量）。令得到的新常数为P。如果$NODE(B)$或$NODE(C)$是处理当前四元式时新构造出来的结点，则删除它。如果$NODE(P)$无定义，则构造一用P作标记的叶结点n，置$NODE(P) = n$，转 **4**；

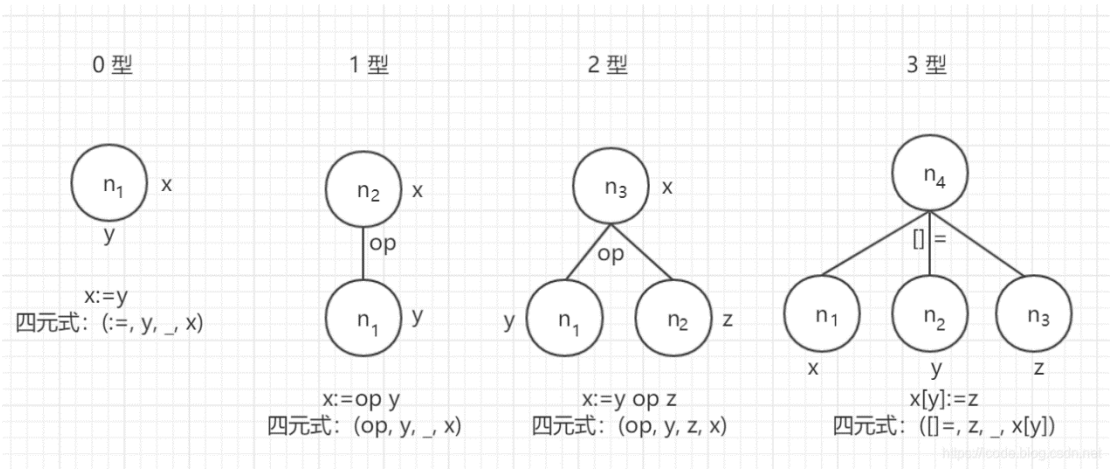
3. 删除公共子表达式

（1）检查 DAG 中是否已有一结点，其唯一后继为$NODE(B)$且标记为op（即公共子表达式）。如果没有，则构造该结点n，否则，把已有的结点作为它的结点并设该结点为n。转 **4**；

（2）检查 DAG 中是否已有一结点，其左后继为$NODE(B)$，右后继为$NODE(C)$，且标记为op（即公共子表达式）。如果没有，则构造该结点n，否则，把已有的结点作为它的结点并设该结点为n。转 **4**；

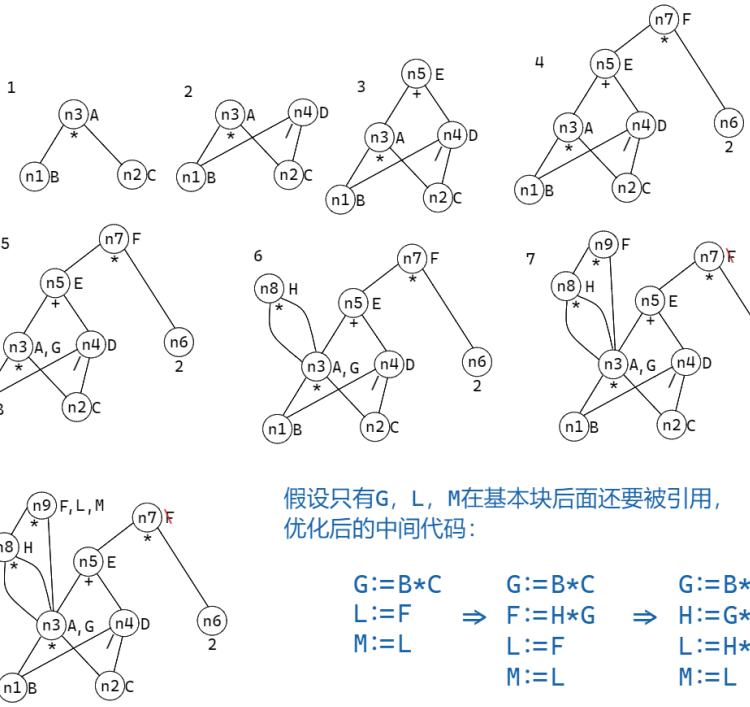
4. 删除无用赋值

如果$NODE(A)$无定义，则把A附加在结点n上并令$NODE(A) = n$；否则，先把A从$NODE(A)$结点上的附加标识符集中删除（注意，如果$NODE(A)$是叶结点，则其A标记不删除）。把A附加到新结点n上并置$NODE(A) = n$。转处理下一个四元式。



B1:

A:=B*C
D:=B/C
E:=A+D
F:=2*E
G:=B*C
H:=G*G
F:=H*G
L:=F
M:=L



假设只有G，L，M在基本块后面还要被引用，优化后的中间代码：

G:=B*C
L:=F
M:=L

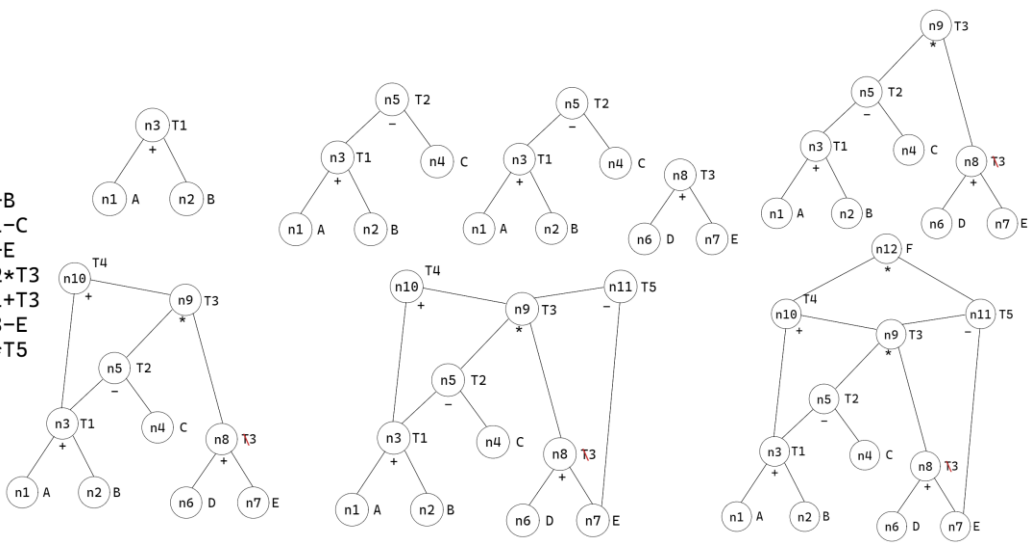
⇒

G:=B*C
F:=H*G
L:=F
M:=L

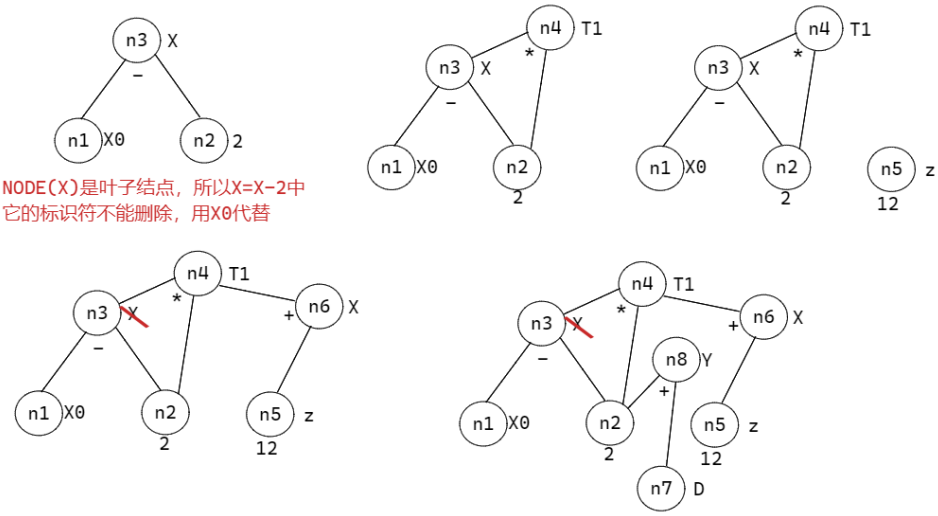
⇒

G:=B*C
H:=G*G
L:=H*G
M:=L

T1:=A+B
T2:=T1-C
T3:=D+E
T3:=T2*T3
T4:=T1+T3
T5:=T3-E
F:=T4*T5



X=X-2
T1=X*2
Z=9+3
X=Z+T1
Y=D+2



NODE(X)是叶子结点，所以X=X-2中它的标识符不能删除，用X0代替