第五章 自下而上的语法分析

- 规范归约
- 算符优先分析
- LR分析法

赵银亮

Shift-Reduce Parser Example

Stack

Expr → *Expr Op Expr*
Expr → (*Expr*)
Expr → - *Expr*
Expr → num
Op → +
Op → -
Op → *

Input String

num	*	(	num	+	num	)

5.1 自下而上分析基本问题

- 推导的逆过程：归约
  - 给定Token串（输入串），逐步归约，最后归约成文法开始符号，表示分析成功。

Shift-Reduce Parser Example

Expr → *Expr Op Expr*
Expr → (*Expr*)
Expr → - *Expr*
Expr → num
Op → +
Op → -
Op → *

num	*	(	num	+	num	)

自下而上的分析过程

语法分析栈　　输入串　　动作

#	⇦	*InputString#*

移进(Shift)

归约(Reduce)

语法分析栈　　输入串

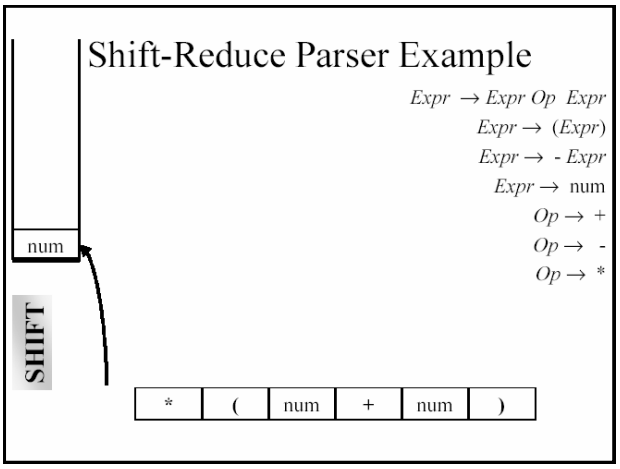
S	⇦	#	接受(Accept)
#			

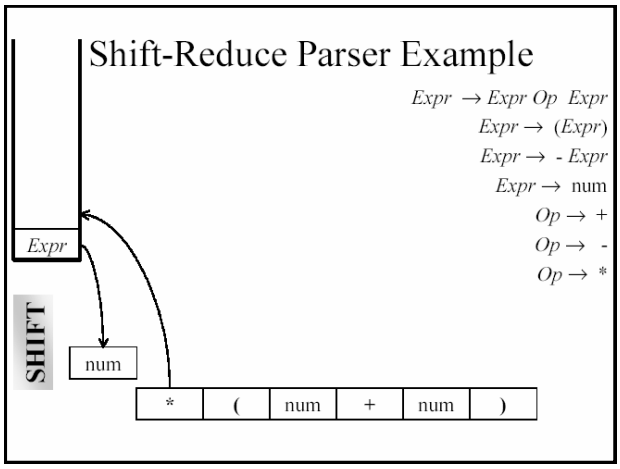
Shift-Reduce Parser Example

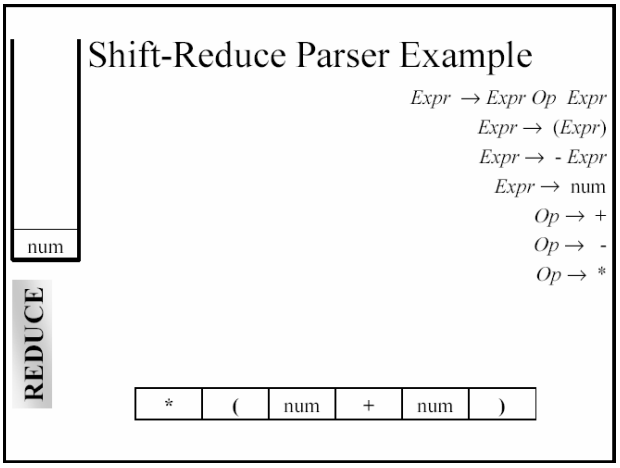
Expr → *Expr Op Expr*
Expr → (*Expr*)
Expr → - *Expr*
Expr → num
Op → +
Op → -
Op → *

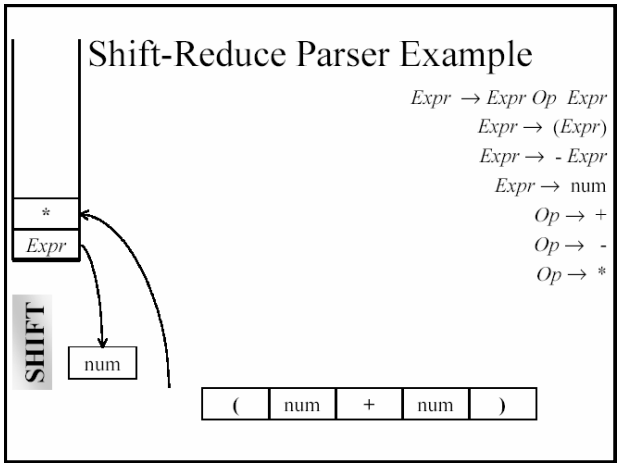
SHIFT

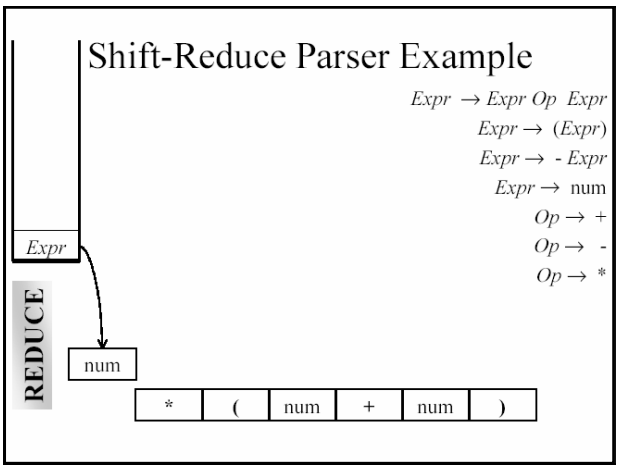
num	*	(	num	+	num	)

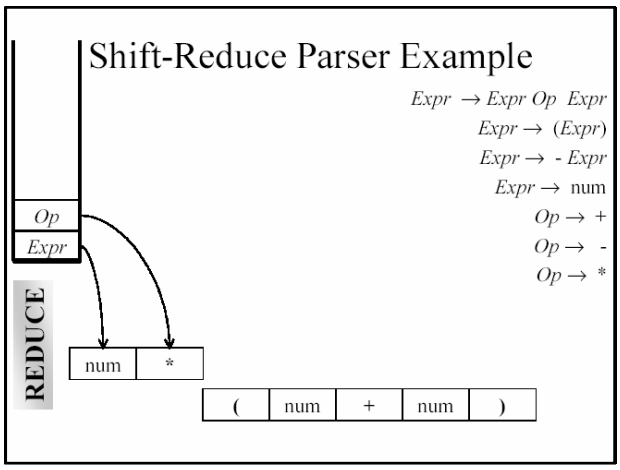

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
num
SHIFT
* (num + num)

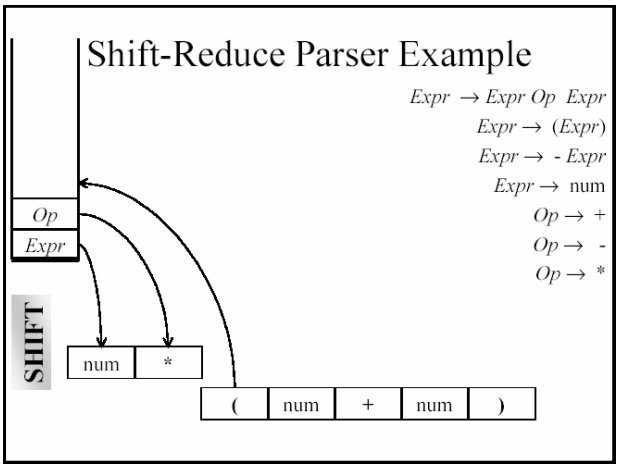

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
Expr
num
SHIFT
* (num + num)

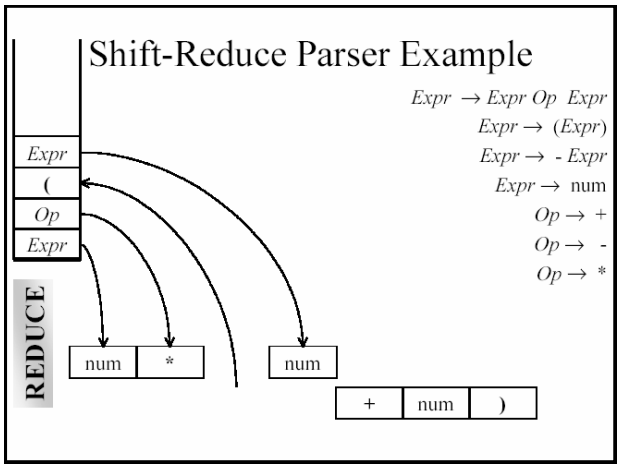

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
num
REDUCE
* (num + num)

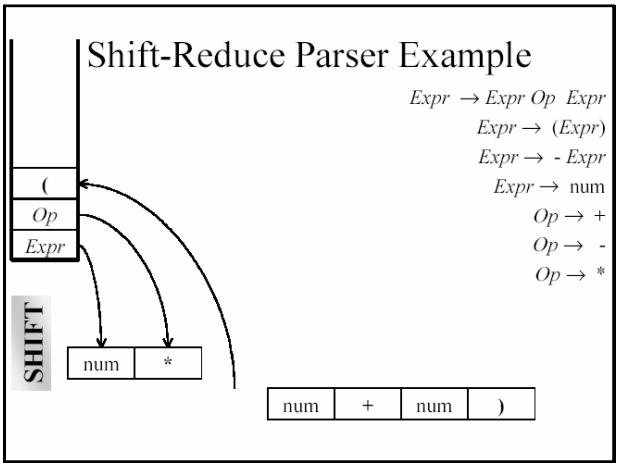

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
*
Expr
num
SHIFT
(num + num)

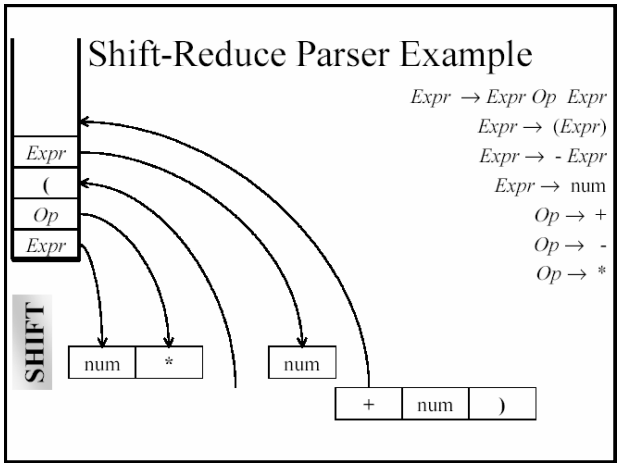

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
Expr
num
REDUCE
* (num + num)

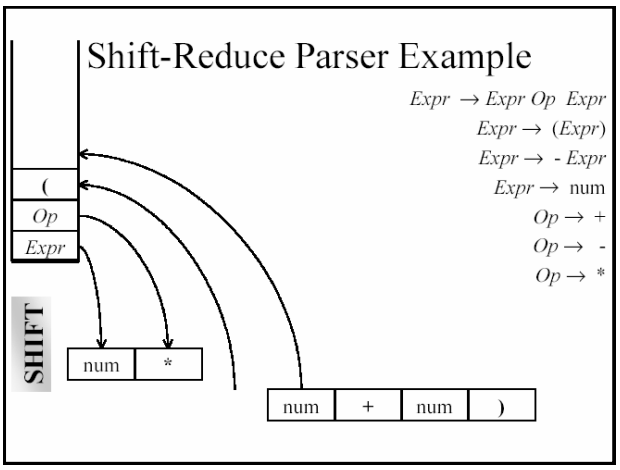

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
Op
Expr
num *
REDUCE
(num + num)

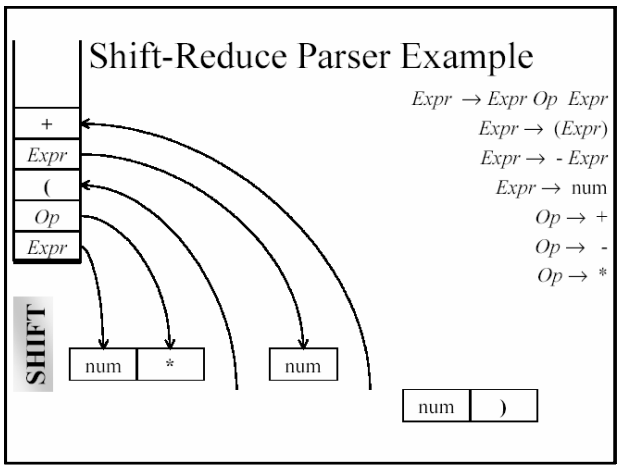

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
Op
Expr
SHIFT
num
*
(num + num)

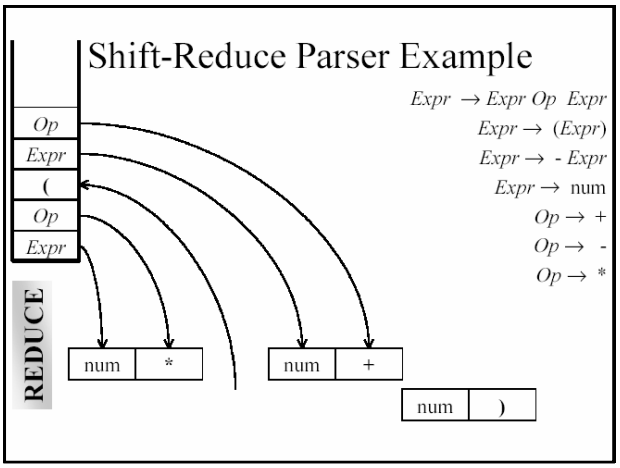

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
Expr
(
Op
Expr
REDUCE
num
*
num
+ num)

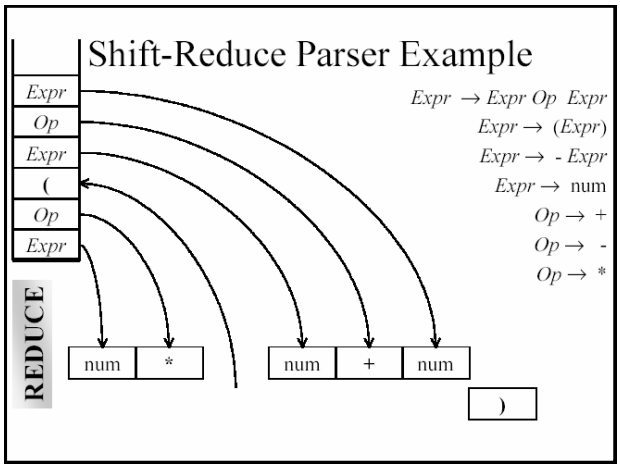

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
(
Op
Expr
SHIFT
num
*
num + num)

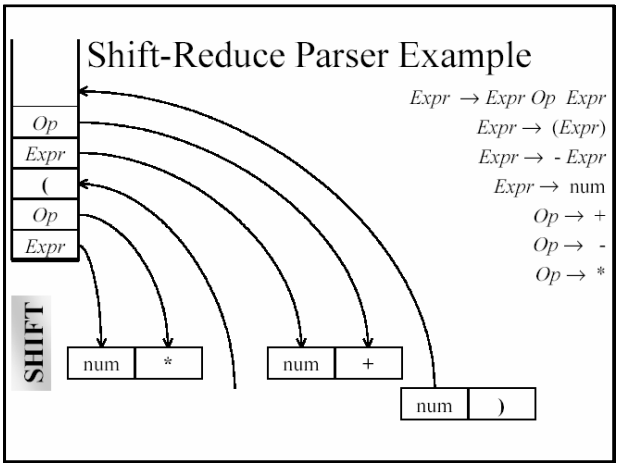

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
Expr
(
Op
Expr
SHIFT
num
*
num
+ num)

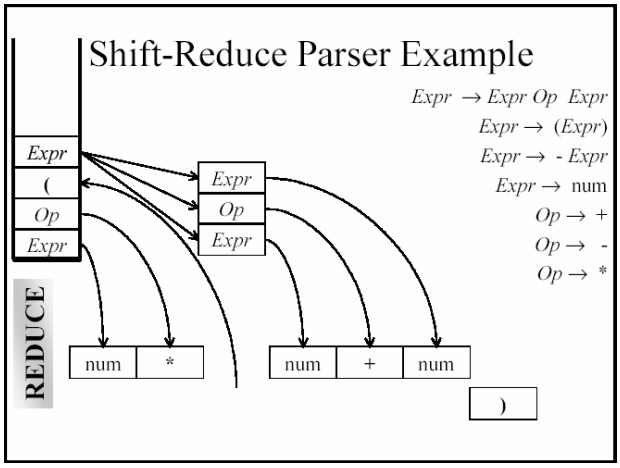

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
(
Op
Expr
SHIFT
num
*
num + num)

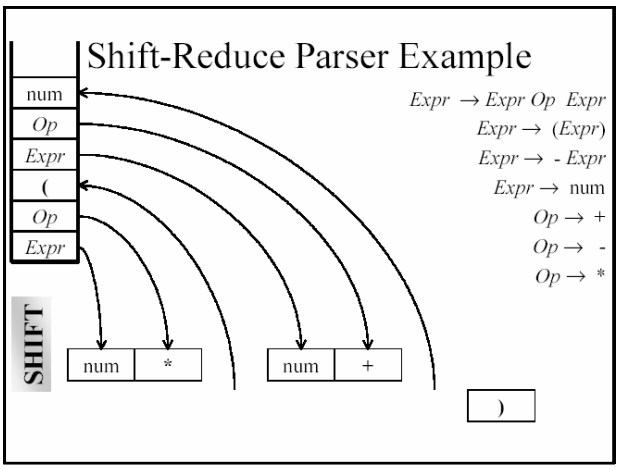

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
+
Expr
(
Op
Expr
SHIFT
num
*
num
num)

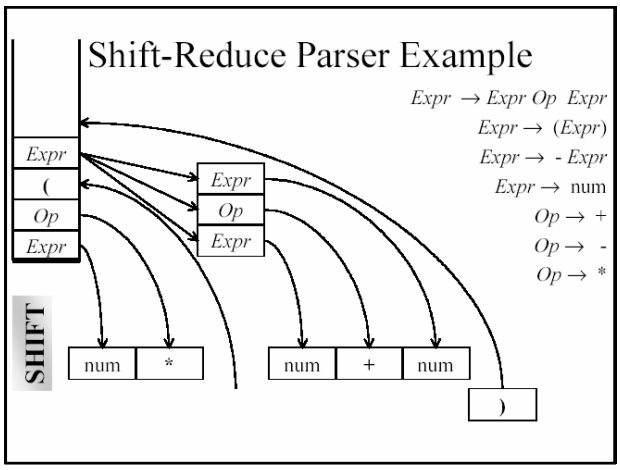

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
REDUCE
Op
Expr
(
Op
Expr
num
*
num
+
num
)

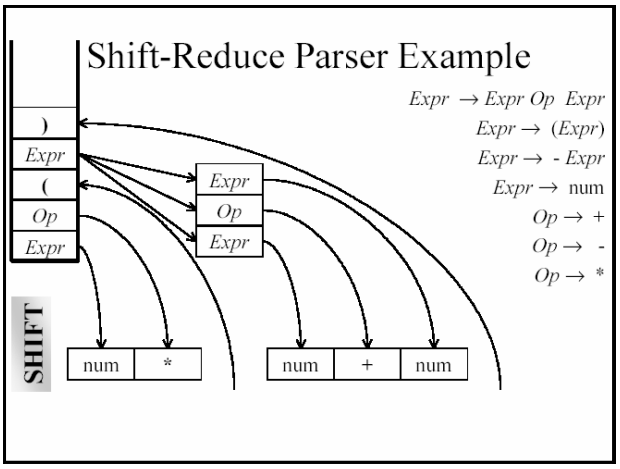

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
REDUCE
Expr
Op
Expr
(
Op
Expr
num
*
num
+
num
)

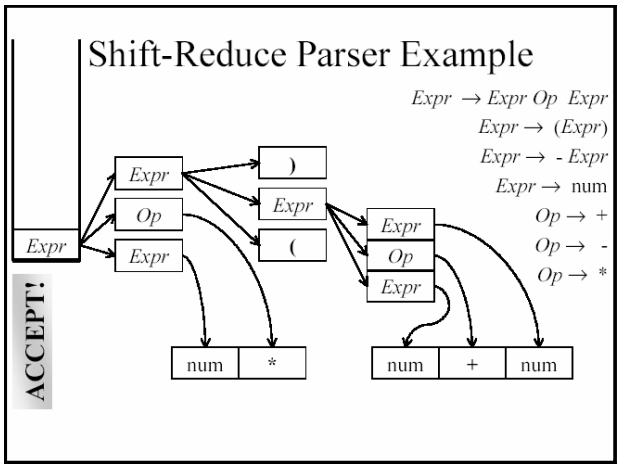

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
SHIFT
Op
Expr
(
Op
Expr
num
*
num
+
num
)

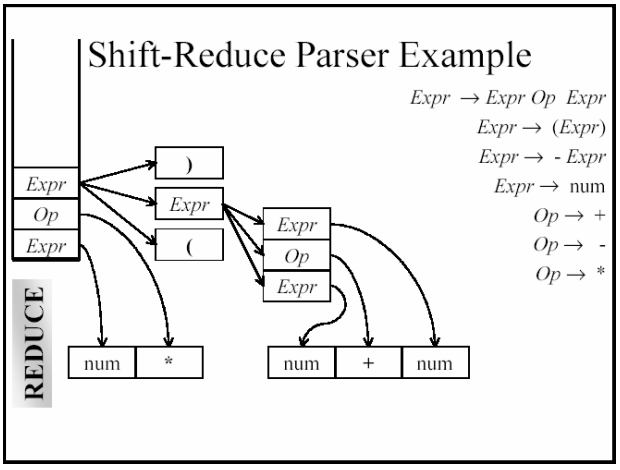

Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
REDUCE
Expr
(
Op
Expr
Expr
Op
Expr
num
*
num
+
num
)


Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
SHIFT
num
Op
Expr
(
Op
Expr
num
*
num
+
)


Shift-Reduce Parser Example
Expr → Expr Op Expr
Expr → (Expr)
Expr → - Expr
Expr → num
Op → +
Op → -
Op → *
SHIFT
Expr
(
Op
Expr
Expr
Op
Expr
num
*
num
+
num
)

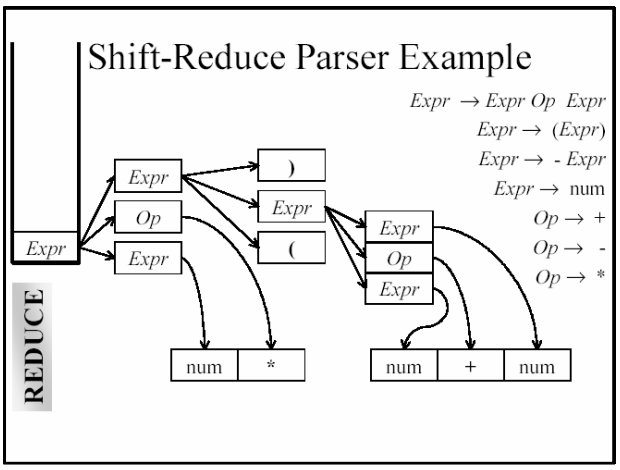






观察

- 输入串中的当前Token被移进栈，即Push到栈顶，当前Token指向下一个
- 如果栈顶部分的符号串刚好跟某个非终结符A的一个候选式一样，则这部分全部出栈，然后将A压入栈。这个过程即是归约。归约所产生的动作为输出该产生式或者构造语法树
- 分析的初始状态、接受状态、出错状态
- 分析过程中可能出现冲突：
  - Top of the stack may match RHS of multiple productions
  - But that may not be the right match
  - May need to shift an input and later find a different reduction



以上分析过程的另一种表示

文法G

Expr → Expr Op Expr

Expr → (Expr)

Expr → - Expr

Expr →num

Op → +

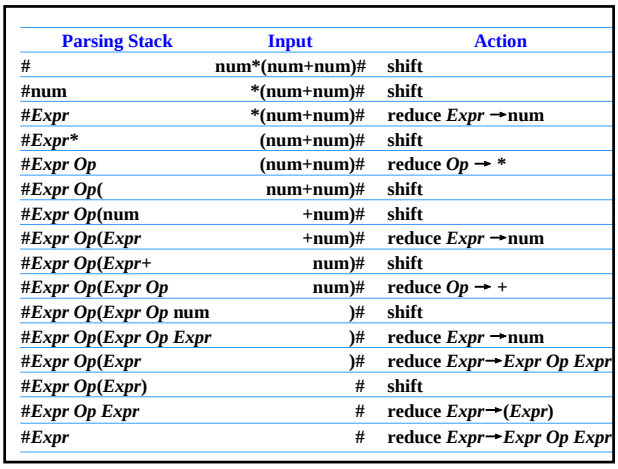
Op → -

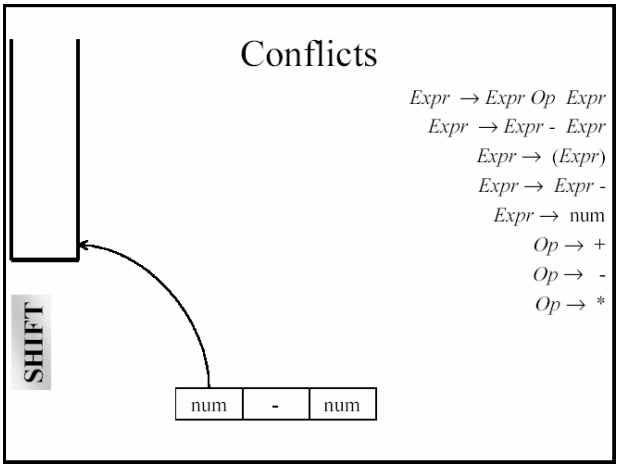
Op → *

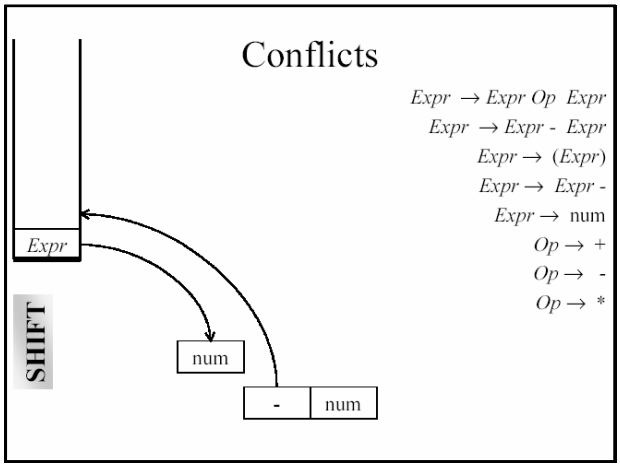
InputString

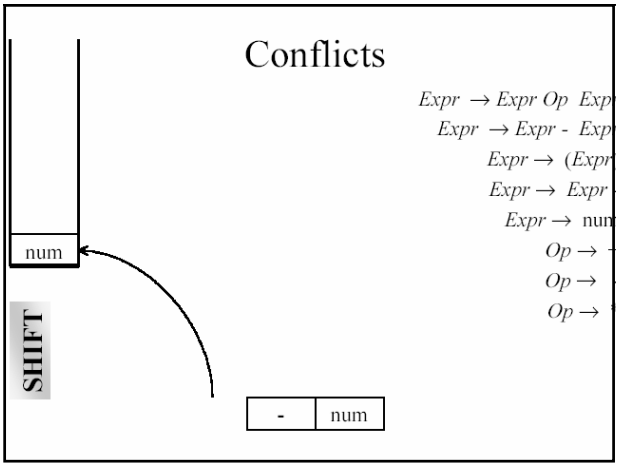
num * (num + num)

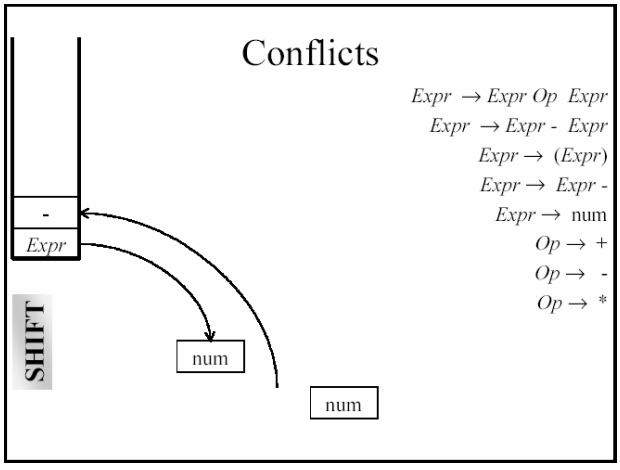
Parsing Stack	Input	Action
#	num*(num+num)#	shift
#num	*(num+num)#	shift
#*Expr*	*(num+num)#	reduce *Expr* →num
#*Expr**	(num+num)#	shift
#*Expr Op*	(num+num)#	reduce *Op* → *
#*Expr Op*(	num+num)#	shift
#*Expr Op*(num	+num)#	shift
#*Expr Op*(*Expr*	+num)#	reduce *Expr* →num
#*Expr Op*(*Expr*+	num)#	shift
#*Expr Op*(*Expr Op*	num)#	reduce *Op* → +
#*Expr Op*(*Expr Op* num	)#	shift
#*Expr Op*(*Expr Op Expr*	)#	reduce *Expr* →num
#*Expr Op*(*Expr*	)#	reduce *Expr*→*Expr Op Expr*
#*Expr Op*(*Expr*)	#	shift
#*Expr Op Expr*	#	reduce *Expr*→(*Expr*)
#*Expr*	#	reduce *Expr*→*Expr Op Expr*

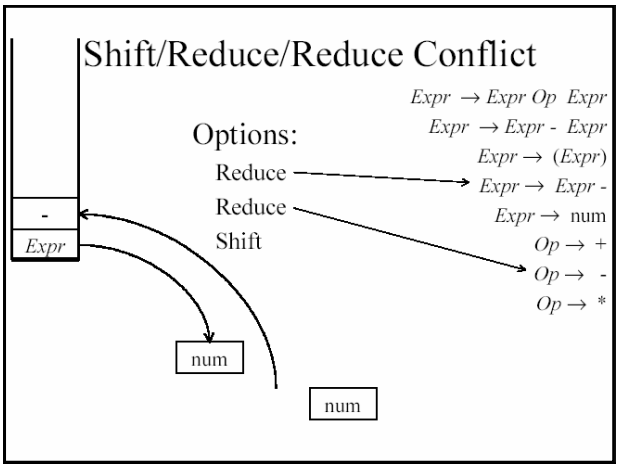


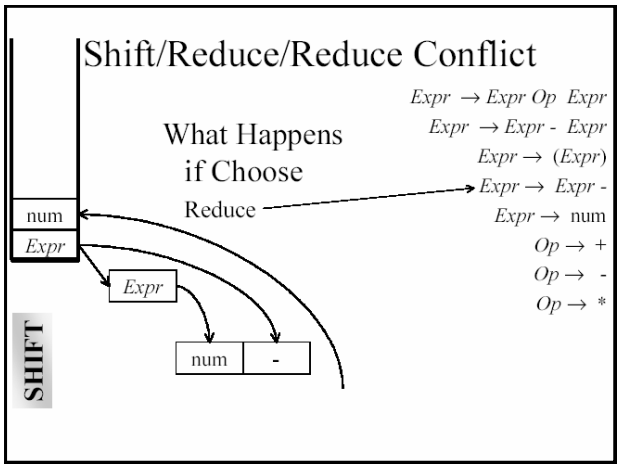


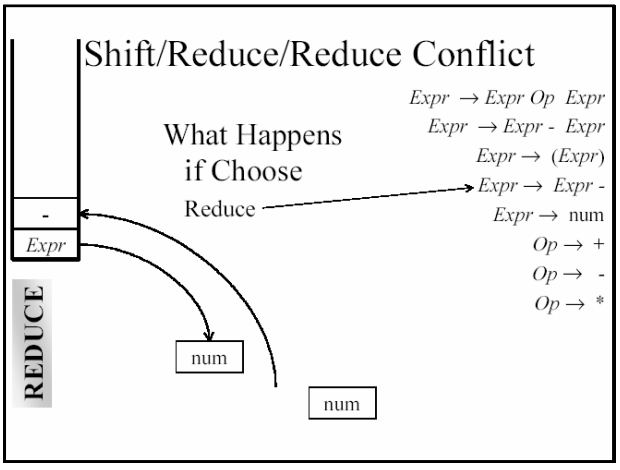


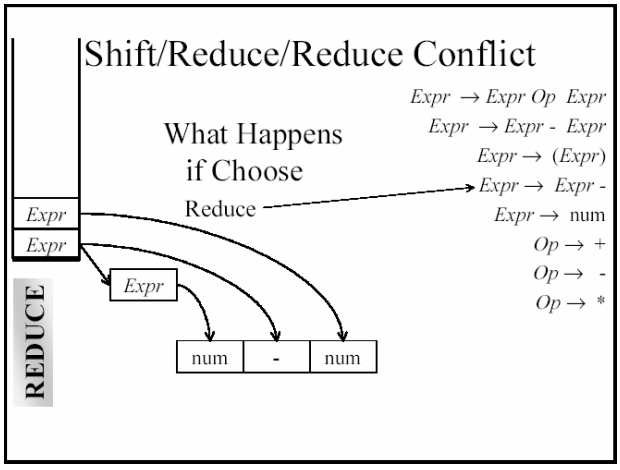


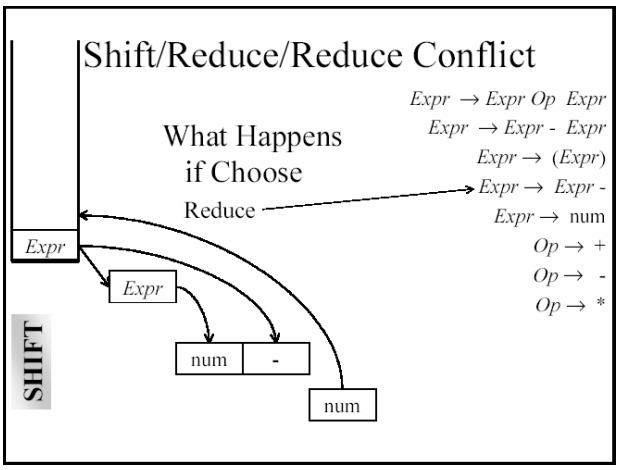


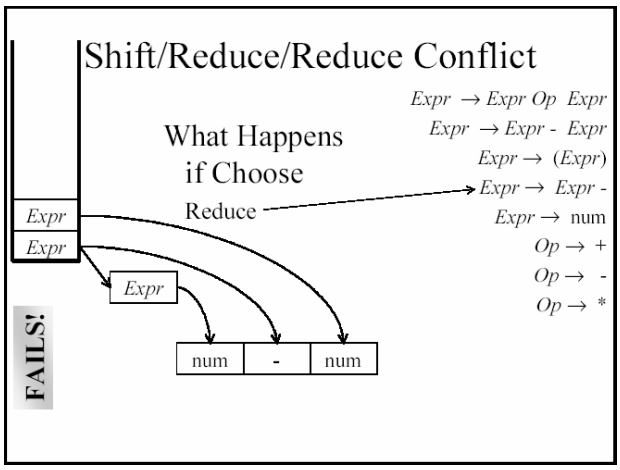

Shift/Reduce/Reduce Conflict
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *
Options:
Reduce
Reduce
Shift
-
Expr
num
num

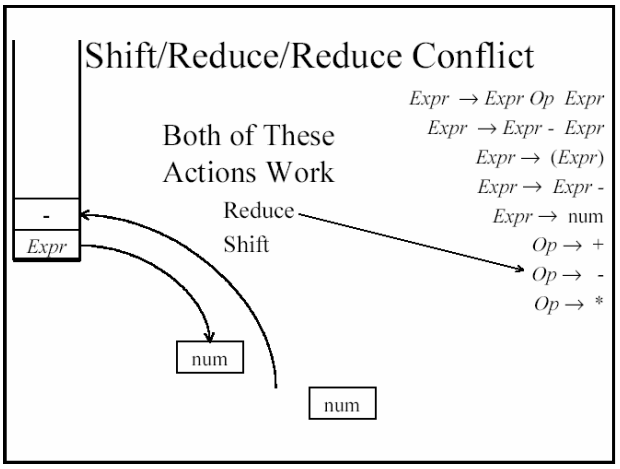

Shift/Reduce/Reduce Conflict
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *
What Happens if Choose
Reduce
num
Expr
Expr
num
-
SHIFT

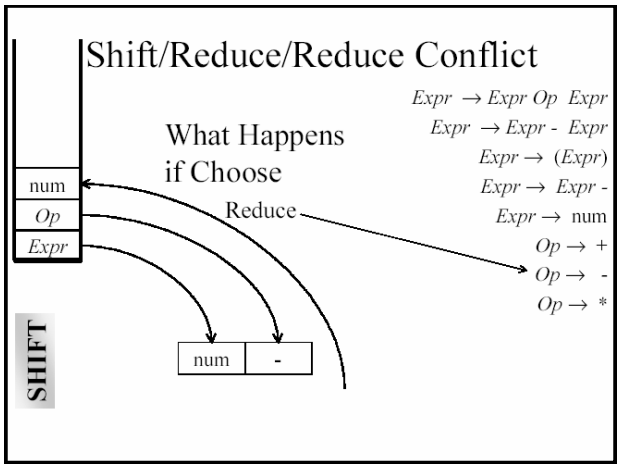

Shift/Reduce/Reduce Conflict
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *
What Happens if Choose
Reduce
-
Expr
num
num
REDUCE

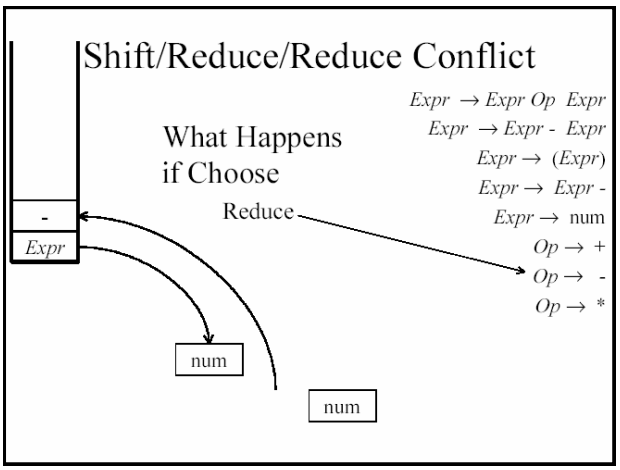

Shift/Reduce/Reduce Conflict
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *
What Happens if Choose
Reduce
Expr
Expr
Expr
num
-
num
REDUCE

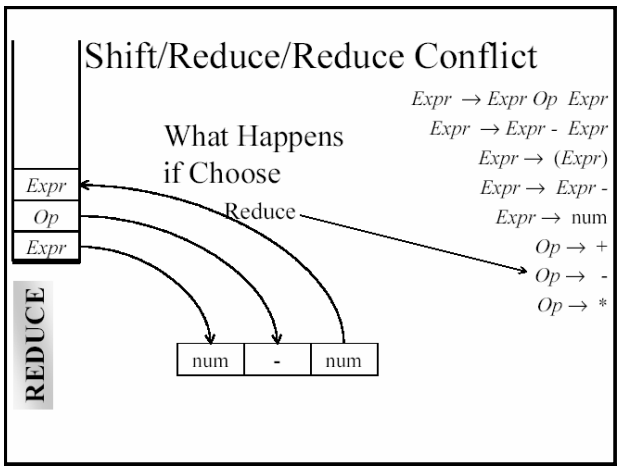

Shift/Reduce/Reduce Conflict
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *
What Happens if Choose
Reduce
Expr
Expr
num
-
num
SHIFT

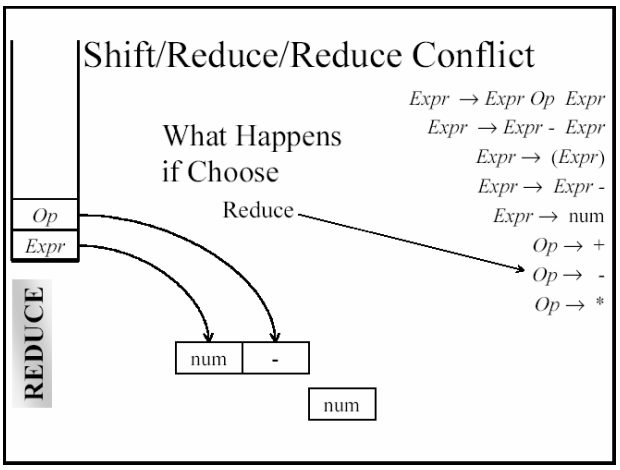

Shift/Reduce/Reduce Conflict
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *
What Happens if Choose
Reduce
Expr
Expr
Expr
num
-
num
FAILS!

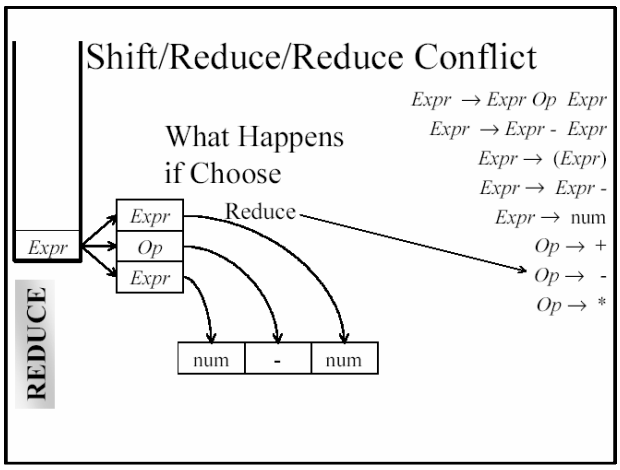

Shift/Reduce/Reduce Conflict
Both of These
Actions Work
Reduce
Shift
-
Expr
num
num
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *

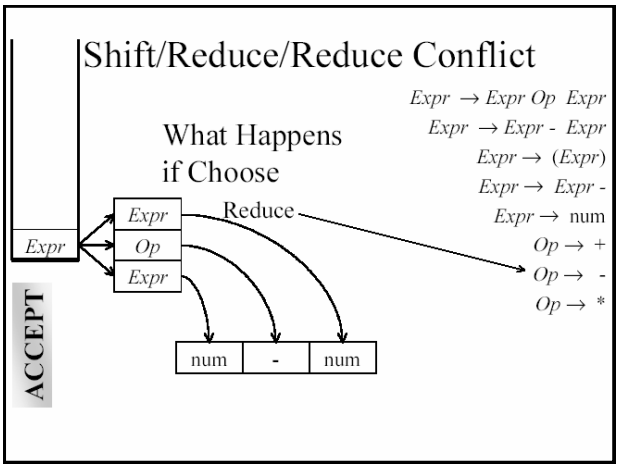

Shift/Reduce/Reduce Conflict
What Happens
if Choose
Reduce
num
Op
Expr
SHIFT
num
-
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *

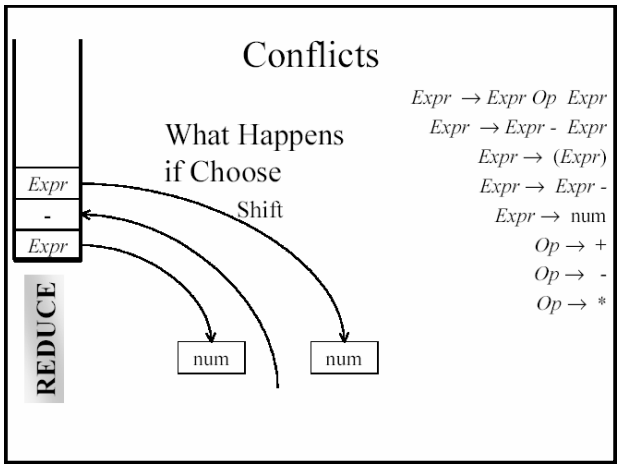

Shift/Reduce/Reduce Conflict
What Happens
if Choose
Reduce
-
Expr
num
num
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *

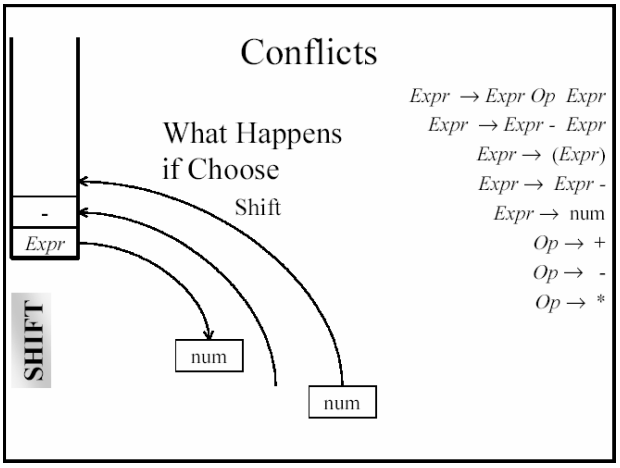

Shift/Reduce/Reduce Conflict
What Happens
if Choose
Reduce
Expr
Op
Expr
REDUCE
num
-
num
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *

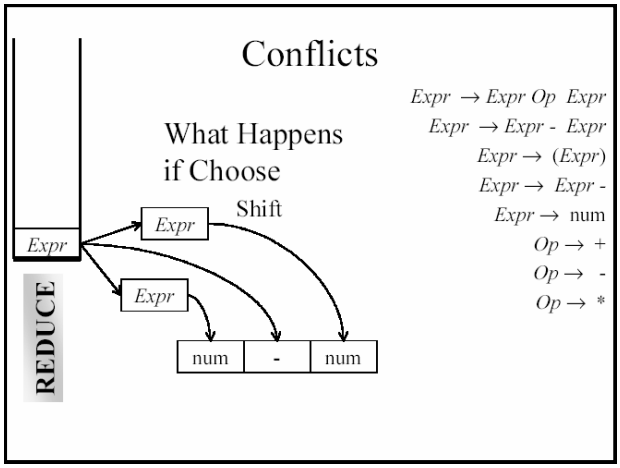

Shift/Reduce/Reduce Conflict
What Happens
if Choose
Reduce
Op
Expr
REDUCE
num
-
num
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *

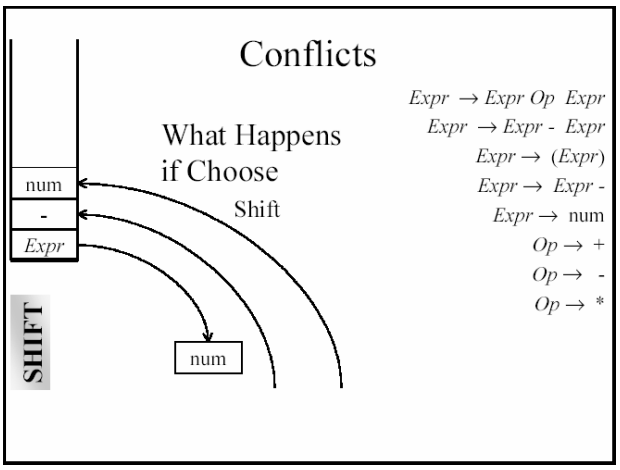

Shift/Reduce/Reduce Conflict
What Happens
if Choose
Reduce
Expr
Expr
Op
Expr
REDUCE
num
-
num
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *

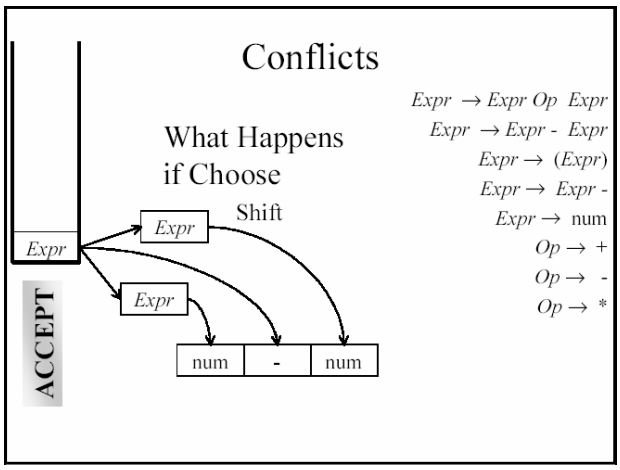

Shift/Reduce/Reduce Conflict
What Happens if Choose
Reduce
Expr
Op
Expr
num
-
num
ACCEPT
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *

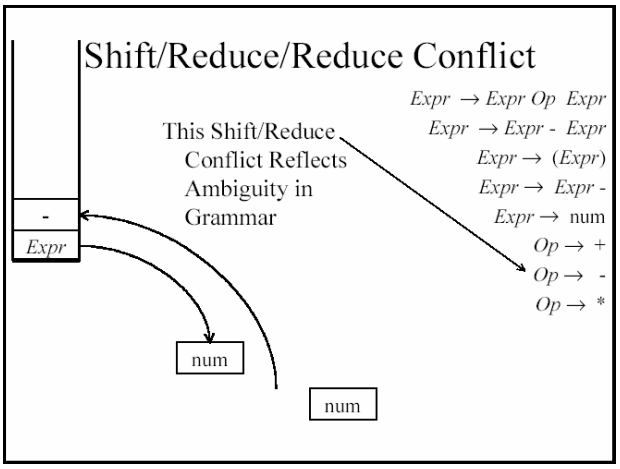

Conflicts
What Happens if Choose
Shift
Expr
-
Expr
num
num
REDUCE
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *


Conflicts
What Happens if Choose
Shift
-
Expr
num
num
SHIFT
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *


Conflicts
What Happens if Choose
Shift
Expr
Expr
Expr
num
-
num
REDUCE
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *

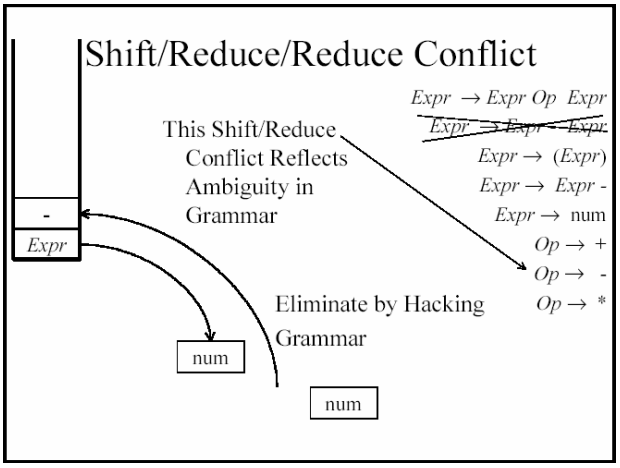

Conflicts
What Happens if Choose
Shift
num
-
Expr
num
SHIFT
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *


Conflicts
What Happens if Choose
Shift
Expr
Expr
Expr
num
-
num
ACCEPT
Expr → Expr Op Expr
Expr → Expr - Expr
Expr → (Expr)
Expr → Expr -
Expr → num
Op → +
Op → -
Op → *



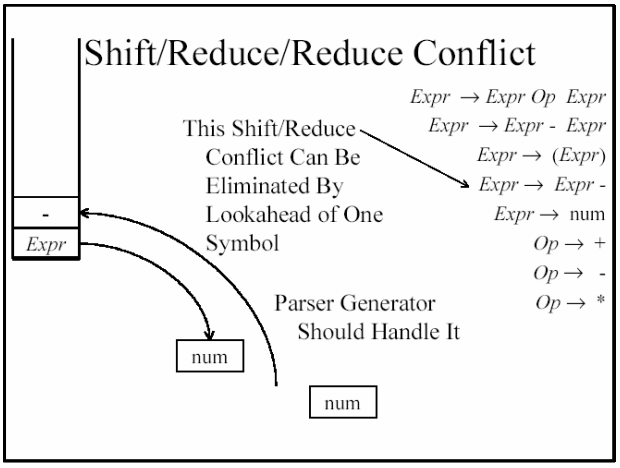
观 察

- 当前栈顶部分与多个产生式右部匹配；
- 移进归约冲突反映了文法有二义性；
- 通过消除文法二义性可以避免某些移进归约冲突；
- 通过向前查看一个Token可以避免某些移进归约冲突；



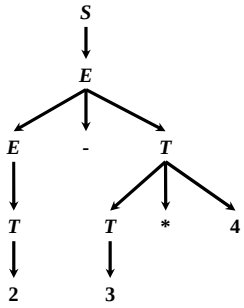
5.2 规范归约

- **短语：**给定文法G, S是G的开始符号，设$\alpha\ \beta\ \delta$是G的一个句型，那么，如果有S =>* α A δ 且 A =>+ β，则称β是句型$\alpha\ \beta\ \delta$相对于非终结符A的短语。
- **直接短语：**如果有 A=> β，则称β是句型$\alpha\ \beta\ \delta$相对于规则A→ β的直接短语



例：

S→E
E→E+T|E–T|T
T→T*i|T/i|i



句型：2-3*4

2是句型2-3*4相对于T和E的短语，是相对于T→i的直接短语

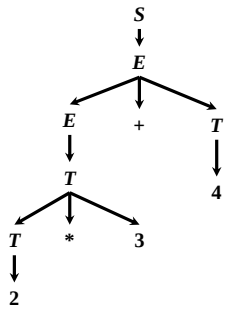
3是句型2-3*4相对于T的短语，是相对于T→i的直接短语

3*4是句型2-3*4相对于T的短语，不是直接短语

2-3*4是句型2-3*4相对于E和S的短语，不是直接短语

例：

S→E
E→E+T|E–T|T
T→T*i|T/i|i



句型：2*3+4

2是句型2*3+4相对于T的短语，是相对于T→i的直接短语

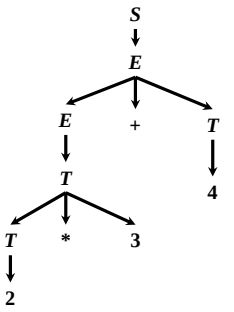
4是句型2*3+4相对于T的短语，是相对于T→i的直接短语

2*3是句型2*3+4相对于T和E的短语,不是直接短语

2-3*4是句型2*3+4相对于E和S的短语，不是直接短语

例：

S→E
E→E+T|E–T|T
T→T*i|T/i|i



句型：2*3+4

2是句型2*3+4相对于T的短语，是相对于T→i的直接短语，是句柄

4是句型2*3+4相对于T的短语，是相对于T→i的直接短语

2*3是句型2*3+4相对于T和E的短语,不是直接短语

2-3*4是句型2*3+4相对于E和S的短语，不是直接短语

规范归约（续）

- 句柄：一个句型的最左直接短语称为该句型的句柄。

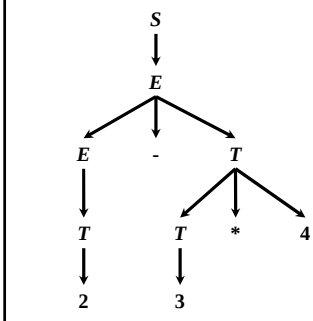
规范归约（续）

- 最右推导：从文法开始符号开始，每一次直接推导中总是将当前句型中最右边的那个非终结符应用它的产生式规则变成下一个句型，也称为规范推导
- 规范句型：由规范推导所得的句型称为规范句型；
- 规范归约：是关于给定句型的一个最右推导的逆过程，也称最左归约

假定α是文法G的一个句子，S是G的开始符号，称序列
$\alpha_n, \alpha_{n-1}, ..., \alpha_1$
是α的一个规范规约，如果此序列满足：
(1)$\alpha_n = \alpha, \alpha_0 = S$;
(2)对任意$0<i \le n$,α_{i-1}是对α_i将其句柄替换为相应产生式左部而得

例：

S→E
E→E+T|E–T|T
T→T*i|T/i|i



句型：2-3*4

2是句型2-3*4相对于T和E的短语，是相对于T→i的直接短语，是句柄

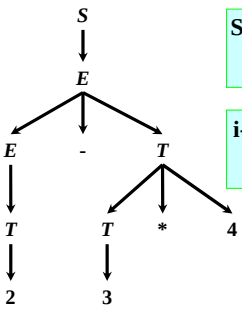
3是句型2-3*4相对于T的短语，是相对于T→i的直接短语

3*4是句型2-3*4相对于T的短语，不是直接短语

2-3*4是句型2-3*4相对于E和S的短语，不是直接短语

例：

S→E
E→E+T|E–T|T
T→T*i|T/i|i



句子：2-3*4

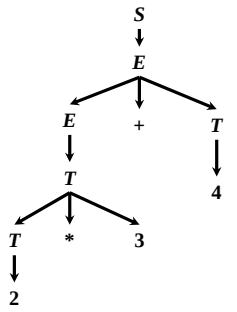
S=>E=>E-T=>E-T*i
=>E-i*i=>T-i*i=>i-i*i

i-i*i<=T-i*i<=E-i*i<=
E-T*i<=E-T<=E<=S

例：

S→E
E→E+T|E–T|T
T→T*i|T/i|i

句子：2*3+4

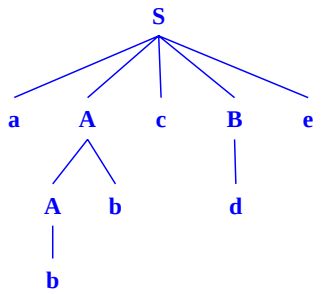


S=>E=>E+T=>E+i
=>T+i=>T*i+i=>i*i+i

i*i+i<=T*i+i<=T+i <=E+i
<=E+T<=E<=S

规范归约：

abbcde <=aAbcde <=aAcde <=aAcBe <=S



语法分析树

文法G2:
S->aAcBe A->b
A->Ab B->d

例5.1

文法G2:
S->aAcBe A->b
A->Ab B->d

输入串：abbcde

最右推导：

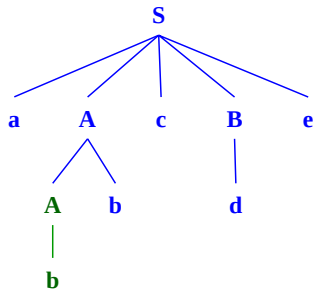
S => aAcBe => aAcde => aAbcde => abbcde

最左归约：

abbcde <= aAbcde <= aAcde <= aAcBe <= S

规范归约：

abbcde <=aAbcde <=aAcde <=aAcBe <=S

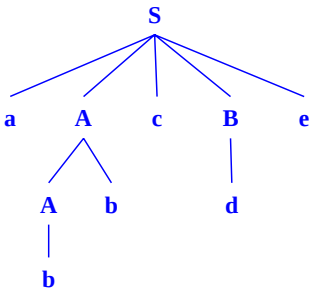


语法分析树

文法G2:
S->aAcBe A->b
A->Ab B->d

规范归约：

abbcde <=aAbcde <=aAcde <=aAcBe <=S

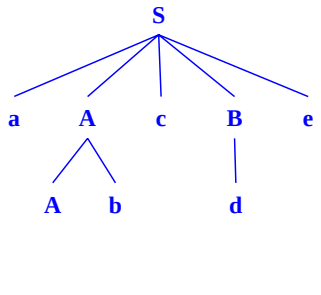


语法分析树

文法G2:
S->aAcBe A->b
A->Ab B->d

规范归约：

abbcde <=aAbcde <=aAcde <=aAcBe <=S

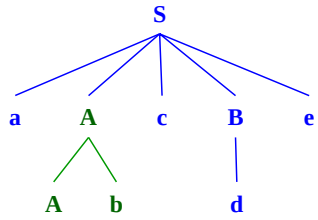


语法分析树

文法G2:
S->aAcBe A->b
A->Ab B->d

规范归约：

abbcde <=aAbcde <=aAcde <=aAcBe <=S

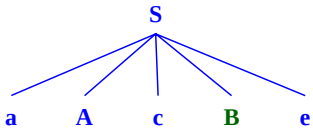


语法分析树

文法G2:
S->aAcBe A->b
A->Ab B->d

规范归约：

abbcde <=aAbcde <=aAcde <=aAcBe <=S

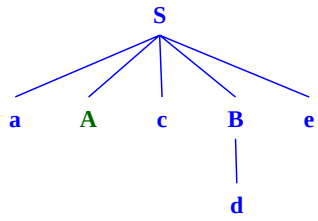


语法分析树

文法G2:
S->aAcBe A->b
A->Ab B->d

规范归约：

abbcde <=aAbcde <=aAcde <=aAcBe <=S

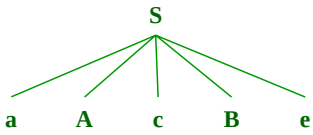


语法分析树

文法G2:
S->aAcBe A->b
A->Ab B->d

规范归约：

abbcde <=aAbcde <=aAcde <=aAcBe <=S

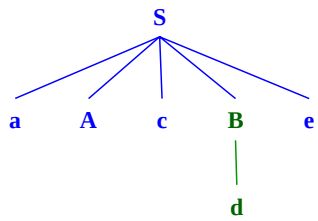


语法分析树

文法G2:
S->aAcBe A->b
A->Ab B->d

规范归约：

abbcde <=aAbcde <=aAcde <=aAcBe <=S



语法分析树

文法G2:
S->aAcBe A->b
A->Ab B->d

规范归约：

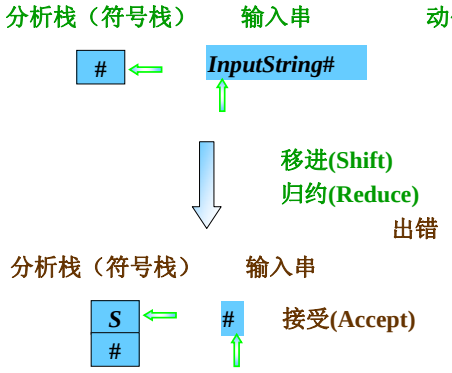
abbcde <=aAbcde <=aAcde <=aAcBe <=S

S

语法分析树

文法G2:
S->aAcBe A->b
A->Ab B->d

5.2.3 符号栈的使用与语法树的表示



5.3.1 算符文法和算符优先文法

- 算符文法：
  - 如果一个文法的任何产生式右部都不含两个相继（并列）的非终结符，即不含有如下形式的产生式右部：

    …QR…

  则我们称该文法为算符文法。

- 对于不含 ε 产生式的算符文法G，如果其中任意一对终结符(a, b)至多只满足下述三种情况之一：
  - $a \doteq b$
  - $a \lessdot b$
  - $a \gtrdot b$

则称G是一个算符优先文法，其中，

- $a \doteq b$ 当且仅当G中有P→…ab…或P→…aQb…
- $a \lessdot b$ 当且仅当G中有P→…aR…且$R \Rightarrow^{+} b$…或$R \Rightarrow^{+} Qb$…
- $a \gtrdot b$ 当且仅当G中有P→…Rb…且$R \Rightarrow^{+}$…a或$R \Rightarrow^{+}$…aQ

5.3 算符优先分析

- 算符优先文法
- 构造优先关系表
- 算符优先分析算法
- 优先函数

例、优先关系表

E→E+T|T　T→T*F|F
F→P^F|P　P→(E)|i

	+	*	^	i	(	)	#
+	⋗	⋖	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋖	⋖	⋖	⋗	⋗
^	⋗	⋗	⋖	⋖	⋖	⋗	⋗
i	⋗	⋗	⋗			⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	≐	
)	⋗	⋗	⋗			⋗	⋗
#	⋖	⋖	⋖	⋖	⋖		≐

$a \doteq b$ 当且仅当G中有P→…ab…或P→…aQb…
$a \lessdot b$ 当且仅当G中有P→…aR…且$R \Rightarrow^{+} b$…或$R \Rightarrow^{+} Qb$…
$a \gtrdot b$ 当且仅当G中有P→…Rb…且$R \Rightarrow^{+}$…a或$R \Rightarrow^{+}$…aQ

5.3.2 构造优先关系表

a ≐ b　当且仅当G中有P→…ab…或P→…aQb…
a ⋖ b　当且仅当G中有P→…aR…且R⇒+b…或R⇒+Qb…
a ⋗ b　当且仅当G中有P→…Rb…且R⇒+…a或R⇒+…aQ

- 对每个非终结符R求出所有的b，其中，
  - R⇒+b…或R⇒+Qb…
  - R⇒+…a或R⇒+…aQ
- FIRSTVT(R), LASTVT(R)

构造集合LASTVT(P)的算法

- P,Q是文法G的非终结符，a是终结符，重复直到不变：
  - 若G有产生式P→…a或P→…aQ，则a∈LASTVT(P)
  - 若a∈LASTVT(Q),且有P→…Q，则a∈LASTVT(P)

```
设置数组F和栈S; F[P,a]初始化为FALSE,P∈V_N,a ∈V_T
Foreach 产生式P→…a或P→…aQ
  If (!F[P,a]){F[P,a]:=TRUE; push(S,<P,a>)}
While (!empty(S)){
  <Q,a>:=top(S); pop(S);
  Foreach 产生式P→…Q
    If (!F[P,a]){F[P,a]:=TRUE; push(S,<P,a>)} }
```

LASTVT(P)={a| F[P,a]:=TRUE}

构造非终结符的FIRSTVT集和LASTVT集

- 构造优先关系表时，要找出所有终结符对之间的优先关系，而对于≐可以直接检查所有的产生式规则即可得出；而对于⋖和⋗需要在直接检查产生式规则基础上并借助FIRSTVT集和LASTVT集完成。

FIRSTVT(P)={a | P⇒+a…或P⇒+Qa… , a∈ V_T 而Q∈V_N }
LASTVT(P)={a | P⇒+…a 或P⇒+…aQ , a∈ V_T 而Q∈V_N }

a ≐ b　当且仅当G中有P→…ab…或P→…aQb…
a ⋖ b　当且仅当G中有P→…aR…且b ∈FIRSTVT(R)
a ⋗ b　当且仅当G中有P→…Rb…且a ∈LASTVT(R)

构造优先表M

a ≐ b　当且仅当G中有P→…ab…或P→…aQb…
a ⋖ b　当且仅当G中有P→…aR…且b ∈FIRSTVT(R)
a ⋗ b　当且仅当G中有P→…Rb…且a ∈LASTVT(R)

```
Foreach 产生式P→X1X2…Xn
  For (i:=1; i<n; i++){
  If (Xi,Xi+1 ∈ VT) M[Xi , Xi+1] := ≐;
  If (i≦n-2&&Xi,Xi+2 ∈ VT && Xi+1 ∈ VN)M[Xi , Xi+2]:=≐;
  If (Xi ∈ VT && Xi+1 ∈ VN)
      foreach a∈FIRSTVT (Xi+1) M[Xi , a] := ⋖;
  If (Xi ∈ VN && Xi+1 ∈ VT)
      foreach a∈LASTVT (Xi) M[a, Xi+1] := ⋗;
  }
```

构造集合FIRSTVT(P)的算法

- P,Q是文法G的非终结符，a是终结符，重复直到不变：
  - 若G有产生式P→a…或P→Qa…，则a∈FIRSTVT(P)
  - 若a∈FIRSTVT(Q),且有P→Q…，则a∈FIRSTVT(P)

```
设置数组F和栈S;  F[P,a]初始化为FALSE,P∈V_N,a ∈V_T
Foreach 产生式P→a…或P→Qa…
  If (!F[P,a]){F[P,a]:=TRUE; push(S,<P,a>)}
While (!empty(S)){
  <Q,a>:=top(S); pop(S);
  Foreach 产生式P→Q…
    If (!F[P,a]){F[P,a]:=TRUE; push(S,<P,a>)} }
```

FIRSTVT(P)={a| F[P,a]:=TRUE}

优先表中对#的处理

- S是文法G的开始符号，
  - 给G中添加一个产生式
    S'→#S#
  - 显然：
    M[#,#]:= ≐;
    foreach a∈FIRSTVT(S) M[#, a]:= ⋖;
    foreach b∈ LASTVT(S) M[b, #]:= ⋗;

例

E→E+T|T　T→T*F|F　S→#E#

F→P^F|P　P→(E)|i

	+	*	^	i	(	)	#
+	⋗	⋖	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋖	⋖	⋖	⋗	⋗
^	⋗	⋗	⋖	⋖	⋖	⋗	⋗
i	⋗	⋗	⋗			⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	≐	
)	⋗	⋗	⋗			⋗	⋗
#	⋖	⋖	⋖	⋖	⋖		≐

a ≐ b　当且仅当G中有P→…ab…或P→…aQb…

a ⋖ b　当且仅当G中有P→…aR…且b ∈FIRSTVT(R)

a ⋗ b　当且仅当G中有P→…Rb…且a ∈LASTVT(R)

算符优先分析算法

```
k:=1;   S[k]:='#';
do{
   把下一个输入符号读进a中;
   if (S[k] ∈ VT) j:=k else j:=k-1;
   while (S[j] ⋗ a){
        do{    Q:=S[j];
               if(S[j-1] ∈ VT) j:=j-1 else j:=j-2
        }while(S[j] ⋗ Q || S[j] ≐ Q)
   把S[j+1]…S[k]归约为某个N;
   k:=j+1;   S[k]:=N;
   }
   if (S[j] ⋖ a || S[j] ≐ a){
        k:=k+1; S[k]:=a
   }else error()
} while(a!='#' )
```

5.3.3 算符优先分析算法

- 最左素短语
  - 文法G,开始符号S，对于S =>* αβδ若S =>* αAδ且 A =>+ β,则称β是句型αβδ的一个短语。
  - 所谓素短语是指这样一个短语,它至少含有一个终结符,并且,除自身之外,不再含任何更小的素短语。(只针对算符文法)
  - 句型中最左边的那个素短语叫最左素短语。

5.4 优先函数

- $\theta \in V_T$，定义自然数函数 f(θ) 和g(θ)，使得：
  - 若 $\theta_1 \lessdot \theta_2$，则f($\theta_1$)<g($\theta_2$)
  - 若 $\theta_1 \doteq \theta_2$，则f(θ_1)=g(θ_2)
  - 若 $\theta_1 \gtrdot \theta_2$，则f($\theta_1$)>g($\theta_2$)

那么，f 称为入栈优先函数，g 称为比较优先函数。

- 使用优先函数替代优先关系表的优点：
  - 优：便于比较运算；节省存储空间。

定理

一个算符优先文法G的任何句型

$\#N_1a_1N_2a_2\ldots N_na_nN_{n+1}\#$

其中$a_i \in V_T$,i=1..n; $N_i = \varepsilon$ 或者$N_i \in V_N$,i=1..n+1

的最左素短语是满足以下条件的最左子串

$N_ja_j\ldots N_ia_iN_{i+1}$

其中

$a_{j-1} \lessdot a_j$

$a_j \doteq a_{j+1}, \ldots, a_{i-1} \doteq a_i$

$a_i \gtrdot a_{i+1}$

i_1*(i_2+i_3)#

⋖ i_1 ⋗ * ⋖(i_2 ⋗ + ⋖ i_3 ⋗) ⋗

优先函数的问题

- 正确性问题
  - if $\theta_1 \lessdot \theta_2$，then f($\theta_1$) < g($\theta_2$)；反之不然
- 存在性问题
  - 并非总能把表映射到优先函数。

	a	b
a	≐	⋗
b	≐	≐

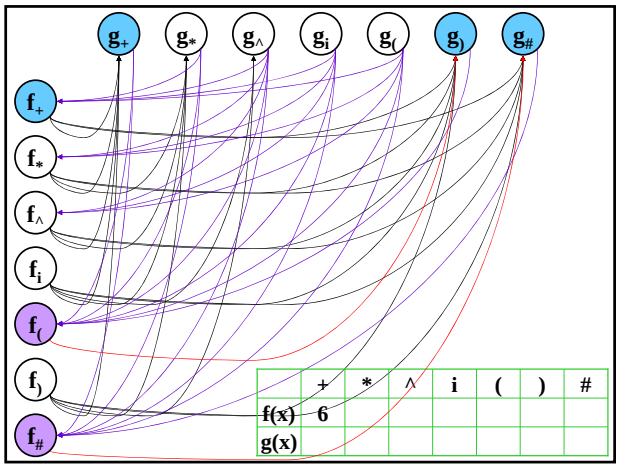
⇒

	g(a)	g(b)
f(a)	=	>
f(b)	=	=

- 唯一性问题
  - 若f和g是算符优先文法G的优先函数，那么对于任意自然数C，f+C和g+C也是G的优先函数

构造优先函数的方法

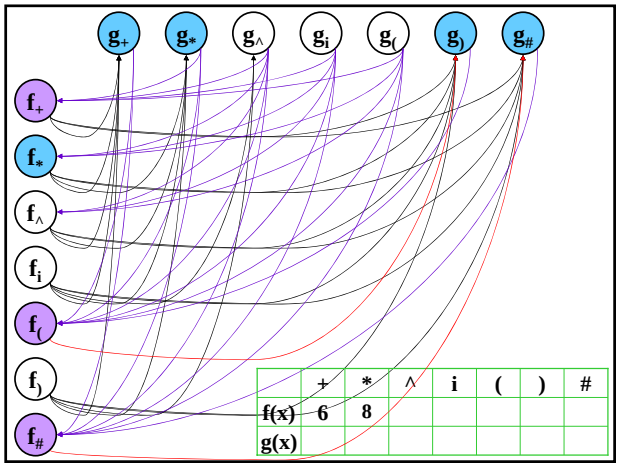
- 已知优先关系表，构造一个有向图H=(V,E)，
  - foreach ($a \in V_T$){ 置$f_a \in V$;置$g_a \in V$} 结点为终结符a对应的符号f_a和g_a;
  - if ($a \doteq b$)置(f_a, g_b) $\in E$ 画结点f_a到结点g_b一条弧;
  - if ($a \lessdot \doteq b$)置(g_b, f_a) $\in E$ 画结点g_b到结点f_a一条弧;
- 定义优先函数f和g,
  - 令f(a)为从f_a出发所能达到的结点个数+1;
  - 令g(a)为从g_a出发所能达到的结点个数+1;
- 检查f和g有无矛盾，若有则不存在优先函数否则成功

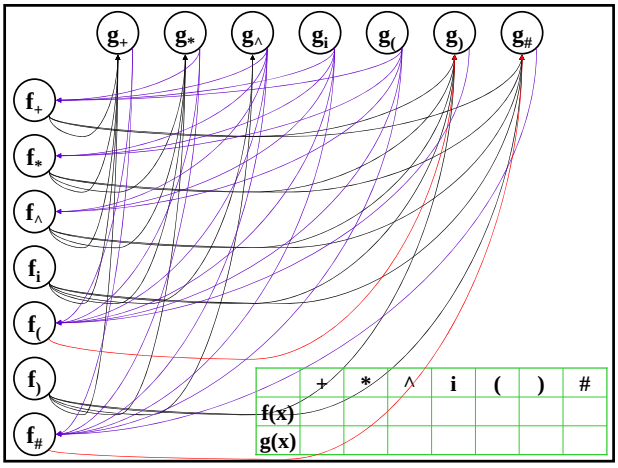


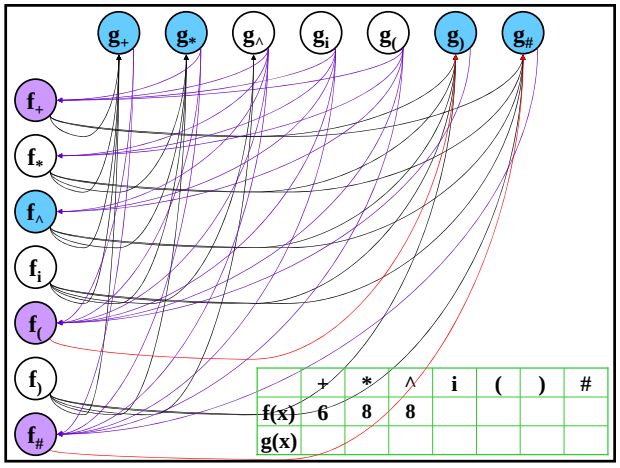
例. 优先关系表

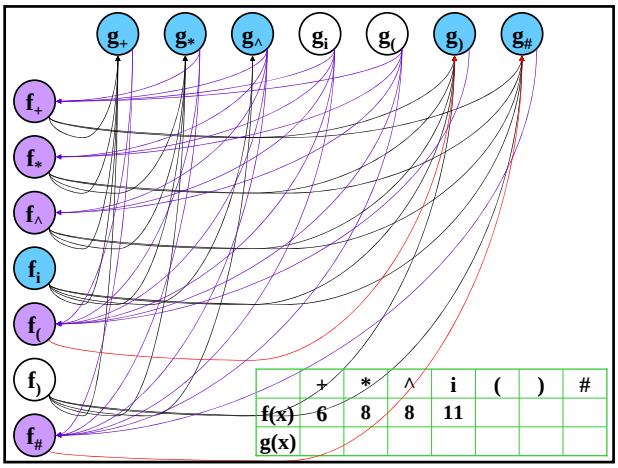
E→E+T|T　T→T*F|F
F→P^F|P　P→(E)|i

	+	*	^	i	(	)	#
+	⋗	⋖	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋖	⋖	⋖	⋗	⋗
^	⋗	⋗	⋖	⋖	⋖	⋗	⋗
i	⋗	⋗	⋗			⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	≐	
)	⋗	⋗	⋗			⋗	⋗
#	⋖	⋖	⋖	⋖	⋖		≐

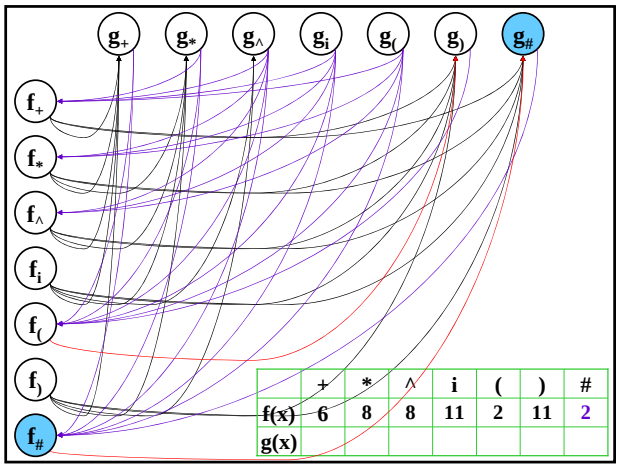




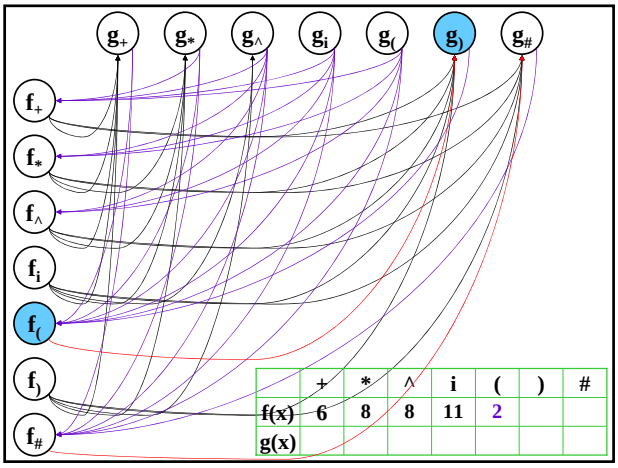




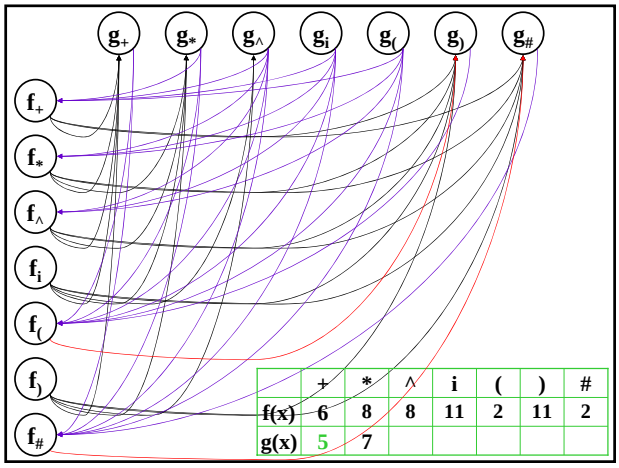
	+	*	^	i	(	)	#
f(x)	6	8	8	11			
g(x)							



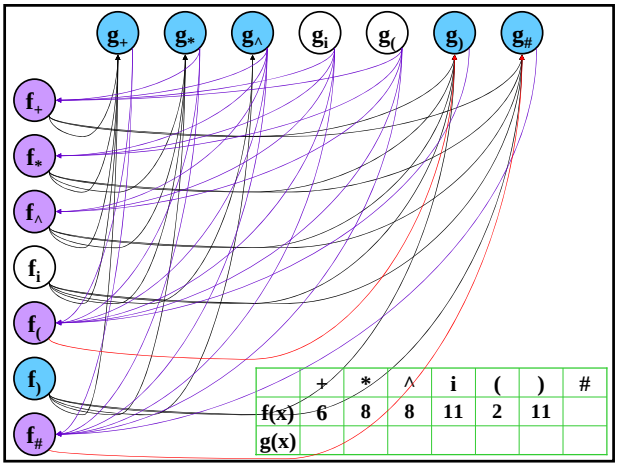
	+	*	^	i	(	)	#
f(x)	6	8	8	11	2	11	2
g(x)							



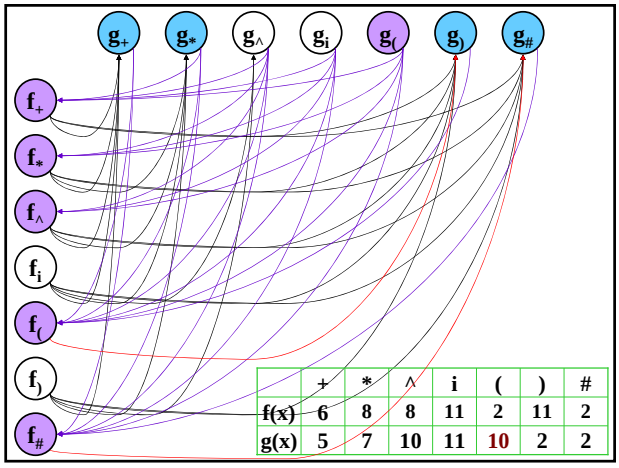
	+	*	^	i	(	)	#
f(x)	6	8	8	11	2		
g(x)							



	+	*	^	i	(	)	#
f(x)	6	8	8	11	2	11	2
g(x)	5	7					



	+	*	^	i	(	)	#
f(x)	6	8	8	11	2	11	
g(x)							



	+	*	^	i	(	)	#
f(x)	6	8	8	11	2	11	2
g(x)	5	7	10	11	10	2	2

结果验证

x	+	*	^	i	(	)	#
f(x)	6	8	8	11	2	11	2
g(x)	5	7	10	11	10	2	2

if $f(\theta_1) < g(\theta_2)$ then $\theta_1 \lessdot \theta_2$

	+	*	^	i	(	)	#
+	⩾	⋖	⋖	⋖	⋖	⩾	⩾
*	⩾	⩾	⋖	⋖	⋖	⩾	⩾
^	⩾	⩾	⋖	⋖	⋖	⩾	⩾
i	⩾	⩾	⩾			⩾	⩾
(	⋖	⋖	⋖	⋖	⋖	≐	
)	⩾	⩾	⩾			⩾	⩾
#	⋖	⋖	⋖	⋖	⋖		≐

错误处理（续）

- 错误：在归约最左素短语时无对应的产生式规则
- 方法：寻找接近的候选式，并根据情况报错
  - 若缺少终结符a，则“a为非法符号”
  - 若多出终结符a ，则“希望出现a ”

优先函数构造方法的正确性

- 若优先函数存在，那么：
  - 若$a \doteq b$ 则$f(a) = g(b)$
  - 若$a \lessdot b$则$f(a) < g(b)$
  - 若$a \gtrdot b$则$f(a) > g(b)$

错误处理（续）

- 错误：在归约最左素短语时无对应的产生式规则
- 可归约串：$N_1b_1N_2b_2N_3\cdots N_kb_kN_{k+1}$,其中$b_1 \doteq b_2 \cdots \doteq b_k$
  - 初始符号集: $\{a \in V_T \mid a \lessdot b_1\}$
  - 结尾符号集: $\{a \in V_T \mid b_k \gtrdot a\}$
- 方法：
  - 推断非终结符情况
  - 推断初始、结尾符号集情况

算符优先分析中的错误处理

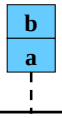
- 出错的情况：
  - 在归约最左素短语时无对应的产生式规则；
  - 栈顶终结符与当前符号间无优先关系；

举例

- E+E
  - 缺E
- i
  - 两边不能够出现非终结符
- (E)
  - 缺E
  - 两边不能够出现非终结符

- 栈顶终结符与当前符号间无优先关系
  - 若a ⋖≐ c 则将b从栈顶移去
  - 若b ⋖≐ d 则将c删除
  - 找出b ⋖≐ e ⋖≐ c则把e插入输入串前端

分析栈



输入串

cd…#

LR分析的思想(续)

Stack	Input	Action
#	(())#	shift
#(	())#	shift
#((	))#	shift
#(()	)#	reduce $X \rightarrow ()$
#(X	)#	shift
#(X)	#	reduce $X \rightarrow (X)$
#X	#	reduce $S \rightarrow X$
#S	#	accept

$S \rightarrow X$
$X \rightarrow (X)$
$X \rightarrow ()$

如何确定句柄？
“历史”+
“展望”+当前符号

规范句型
活前缀

用DFA来实现

活前缀：规范句型的一个前缀(任意首部)
该前缀是不含句柄之后的任何符号。

5.4 LR分析法

- LR分析的思想
- 构造DFA控制shift和reduce动作
- 由DFA构造分析表

5.3.2 LR(0)项目集

- 需要知道对于给定的产生式我们到目前为止已经扫描过了多少？

$X \rightarrow (X)$

在这儿？ 在这儿？ 在这儿？ 在这儿？

由此生成4个项目
$X \rightarrow \bullet (X)$
$X \rightarrow (\bullet X)$
$X \rightarrow (X \bullet)$
$X \rightarrow (X) \bullet$

文法：
$S \rightarrow X$
$X \rightarrow (X)$
$X \rightarrow ( )$

5.4.1 LR分析的思想

状态栈 符号栈 输入串 动作

s_0 # $a_1a_2\ldots a_n$#

- 动作
  - 将符号和状态分别进栈；
  - 根据给定的产生式归约；
  - 接受；
- 根据当前符号和当前状态决定动作
- 每个动作指明了下一个状态是什么

$s_0 s_1 \ldots s_m$ $\#X_1X_2\ldots X_m$ $a_ia_{i+1}\ldots a_m$#

$S \rightarrow X\#$
$X \rightarrow (X)$
$X \rightarrow ( )$

$S \rightarrow \bullet X\#$
$S \rightarrow X\bullet \#$
$X \rightarrow \bullet (X)$
$X \rightarrow (\bullet X)$
$X \rightarrow (X \bullet)$
$X \rightarrow (X) \bullet$
$X \rightarrow \bullet ( )$
$X \rightarrow (\bullet)$
$X \rightarrow ( ) \bullet$

文法 G 的项目

文法 G

S'→E

E→aA|bB

A→cA|d

B→cB|d

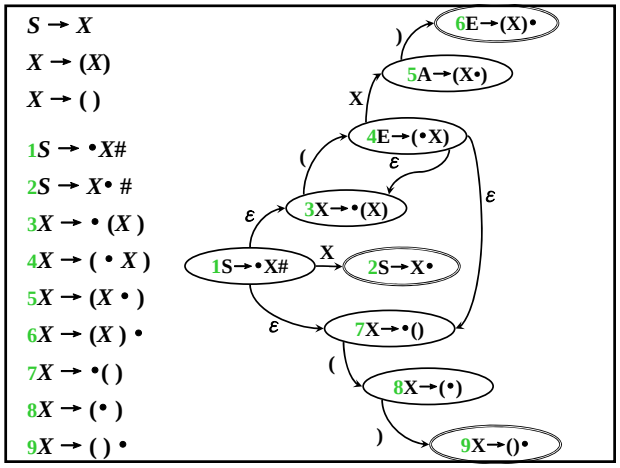
① S'→•E
② S'→E•
③ E→•aA
④ E→a•A
⑤ E→aA•
⑥ E→•bB
⑦ E→b•B
⑧ E→bB•
⑨ A→•cA
⑩ A→c•A
11 A→cA•
12 A→•d
13 A→d•
14 B→•cB
15 B→c•B
16 B→cB•
17 B→•d
18 B→d•

构造识别活前缀的NFA

- 每个项目构成一个状态；项目S'→•S为NFA的唯一初态；
- 从每个状态$X\to X_1\cdots X_{i-1}\bullet X_i\cdots X_n$ 到状态 $X\to X_1\cdots X_i\bullet X_{i+1}\cdots X_n$画一条标记为$X_i$的弧；
  - 另外，若X_i为非终结符，那么到所有形如 $X_i\to\bullet\gamma$ 的项目所在的状态 画 ε 弧

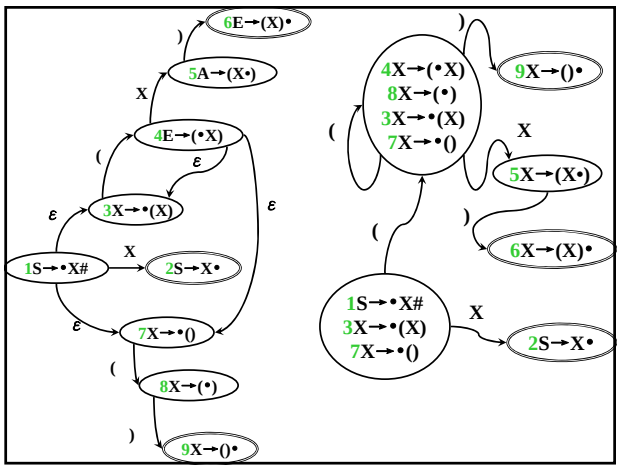
项目中的关键点

- 状态对应于项目集
- 若状态包含项目 A → α • c β
  - 分析程序期望最终采用产生式A→α c β进行归约
  - 分析程序已经扫描了α
  - 期望输入流能够包含c, 然后 β
- 若状态包含项目 A → α •
  - 分析程序已经扫描了α
  - 将使用A → α进行归约
- 若状态包含项目 S → α•#并且输入串为空
  - 分析程序接受该输入

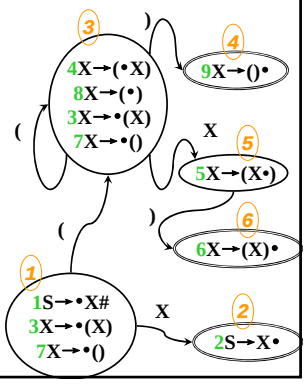


项目与行为的相互关联

- 如果当前状态包含项目 A → α • c β并且当前输入符号是c
  - 分析程序将 c 移到栈顶
  - 下一状态会包含A → α c • β
- 如果当前状态包含项目 A → α •
  - 分析程序使用A → α进行归约
- 如果当前状态包含项目 S → α •#并且输入串为空
  - 分析程序接受该输入



		Action
1	#	(()# shift
13	#(	()# shift
133	#((	)# shift
1334	#(()	)# reduce $X \rightarrow ()$
135	#(X	)# shift
1356	#(X)	# reduce $X \rightarrow (X)$
12	#X	# reduce $S \rightarrow X$
		# accept



例2 构造文法 G 的 LR(0)分析表

文法 G

S'→E

E→aA|bB

A→cA|d

B→cB|d

① S'→•E
② S'→E•
③ E→•aA
④ E→a•A
⑤ E→aA•
⑥ E→•bB
⑦ E→b•B
⑧ E→bB•
⑨ A→•cA
⑩ A→c•A
⑪ A→cA•
⑫ A→•d
⑬ A→d•
⑭ B→•cB
⑮ B→c•B
⑯ B→cB•
⑰ B→•d
⑱ B→d•

5.3.3 LR(0)分析表的构造

$DFA:(S,\Sigma,\delta,s_0,F),\Sigma=V_T\cup V_N\cup\{\#\}$

$if\ (m\xrightarrow{a\in V_T}n)M[m,ACTION(a)]:=sn;$

$if\ (m\xrightarrow{X\in V_N}n)M[m,GOTO(X)]:=n;$

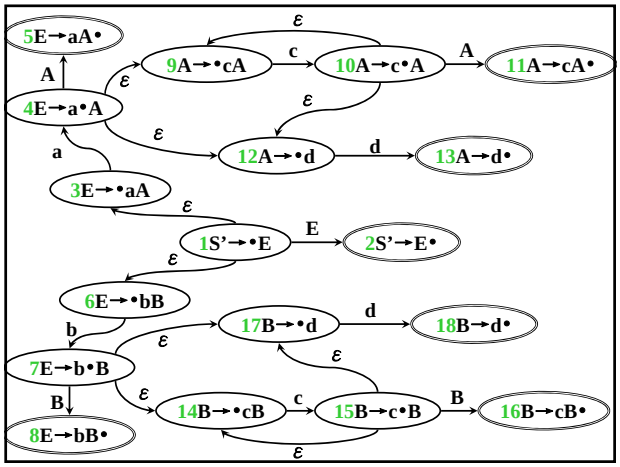
$if\ (\{A\rightarrow\alpha\square\}\subseteq itemset(m))\forall a\in V_T,M[m,ACTION(a)]:=rk;$

$m,n\in S;$

k为规则$A\rightarrow\alpha$的编号；

sn为shift to n缩写；

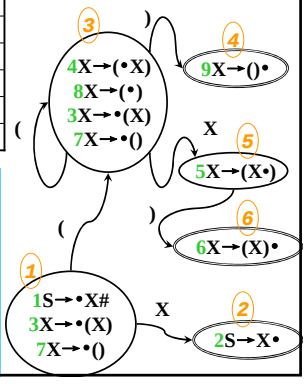
rk为reduce to k缩写



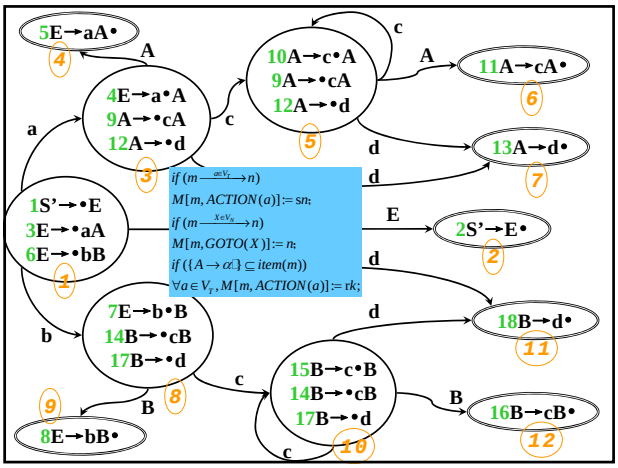
State	ACTION (	)	#	GOTO X
1	s3			2
2			acc	
3	s3	s4		5
4	r3	r3	r3	
5		s6		
6	r2	r2	r2	

$if\ (m\xrightarrow{a\in V_T}n)$
$M[m,ACTION(a)]:=sn;$
$if\ (m\xrightarrow{X\in V_N}n)$
$M[m,GOTO(X)]:=n;$
$if\ (\{A\rightarrow\alpha\square\}\subseteq itemset(m))$
$\forall a\in V_T,M[m,ACTION(a)]:=rk;$

例1.构造分析表



	a	b	c	d	E	A	B
{1,3,6}	{4,9,12}	{7,14,17}			{2}		
{4,9,12}			{9,10,12}	{13}		{5}	
{7,14,17}			{14,15,17}	{18}			{8}
{2}							
{9,10,12}			{9,10,12}	{13}		{11}	
{13}							
{5}							
{14,15,17}			{14,15,17}	{18}			{16}
{18}							
{8}							
{11}							
{16}							



作 业

- P133
- 1, 2, 3, *5(1), *5(2)

上机题B

- 编写程序输入一个算符优先文法，自动构造出优先关系表（并输出），该程序还能对输入的句子进行语法分析，依序输出分析过程中归约的最左素短语（本题与上一章题目任选一个完成）。

状态	ACTION(动作)					GOTO(转换)		
	a	b	c	d	#	E	A	B
1	s3	s8				2		
2					acc			
3			s5	s7			4	
4	r5	r5	r5	r5	r5			
5			s5	s7			6	
6	r11	r11	r11	r11	r11			
7	r13	r13	r13	r13	r13			
8			s10	s11				9
9	r8	r8	r8	r8	r8			
10			s10	s11				12
11	r18	r18	r18	r18	r18			
12	r16	r16	r16	r16	r16			

项目集的闭包

- 闭包寻找同一“状态”中的所有项目

Fixed Point Algorithm for Closure**(I)**

- Every item in **I** is also an item in Closure**(I)**
- If A→ α • B β is in Closure**(I)** and B→ • γ is an item, then add B→ • γ to Closure**(I)**
- Repeat until no more new items can be added to Closure**(I)**

LR分析法概貌

- LR(k)
  - L-从左往右扫描输入串
  - R-最右推到的逆过程
  - K-向前查看k个符号
- LR(0)
- SLR（简单LR表构造法）
- 规范LR
- LALR（ 向前LR表构造法）

Example of Closure

- Closure($\{X \rightarrow (\cdot X)\}$)

$$\left\{\begin{array}{l} X \rightarrow \quad (\cdot X) \\ X \rightarrow \cdot (X) \\ X \rightarrow \cdot (\ ) \end{array}\right\}$$

- Items

$S \rightarrow \cdot X\$$
$S \rightarrow X \cdot \$$
$X \rightarrow \cdot (X)$
$X \rightarrow (\cdot X)$
$X \rightarrow (X \cdot)$
$X \rightarrow (X) \cdot$
$X \rightarrow \cdot (\ )$
$X \rightarrow (\cdot)$
$X \rightarrow (\quad) \cdot$

Another Example

- closure({$S \rightarrow \cdot X\$$})

$$\left\{\begin{array}{l} S \rightarrow \cdot X\$ \\ X \rightarrow \cdot (X) \\ X \rightarrow \cdot ( ) \end{array}\right\}$$

- Items

$S \rightarrow \cdot X\$$
$S \rightarrow X \cdot \$$
$X \rightarrow \cdot (X)$
$X \rightarrow (\cdot X)$
$X \rightarrow (X \cdot)$
$X \rightarrow (X) \cdot$
$X \rightarrow \cdot ( )$
$X \rightarrow (\cdot)$
$X \rightarrow ( ) \cdot$

Another Example of Goto

- Goto ({$X \rightarrow \cdot(X)$}, ()

$$\left\{\begin{array}{l} X \rightarrow (\cdot X) \\ X \rightarrow \cdot (X) \\ X \rightarrow \cdot ( ) \end{array}\right\}$$

- Items

$S \rightarrow \cdot X\$$
$S \rightarrow X \cdot \$$
$X \rightarrow \cdot (X)$
$X \rightarrow (\cdot X)$
$X \rightarrow (X \cdot)$
$X \rightarrow (X) \cdot$
$X \rightarrow \cdot ( )$
$X \rightarrow (\cdot)$
$X \rightarrow ( ) \cdot$

项目集的Goto()

- 在当前状态，通过消耗掉一个文法符号，Goto找到一个新的状态

算法 *Goto*(I, X)，其中I为项目集，X为文法符号

- Goto(I, X) = Closure({A→ α X• β | A→ α •X $\beta \in$ I })
- goto 是一个新的集合，通过把点移过X得到

构造DFA的状态

- 从项目 $S \rightarrow \cdot\ \beta$ #开始
- 将Closure({ *Goal* → • ***S*** $}) 作为第一个状态
- 取一个状态 **I**
  - 对**I** 中的每个项目 A→ α • X β
    - 求出 Goto**(I, X)**
    - 如果 Goto**(I, X)还不是状态，则置为状态**
    - 从状态**I**到状态Goto**(I, X)**添加一条弧X
- 重复直到没有新的添加为止

Example of Goto

- Goto ({$X \rightarrow (\cdot X)$}, X)

$$\left\{X \rightarrow (X \cdot\ )\right\}$$

- Items

$S \rightarrow \cdot X\$$
$S \rightarrow X \cdot \$$
$X \rightarrow \cdot (X)$
$X \rightarrow (\cdot X)$
$X \rightarrow (X \cdot)$
$X \rightarrow (X) \cdot$
$X \rightarrow \cdot ( )$
$X \rightarrow (\cdot)$
$X \rightarrow ( ) \cdot$