第八章 自下而上的语法分析
2024

概述

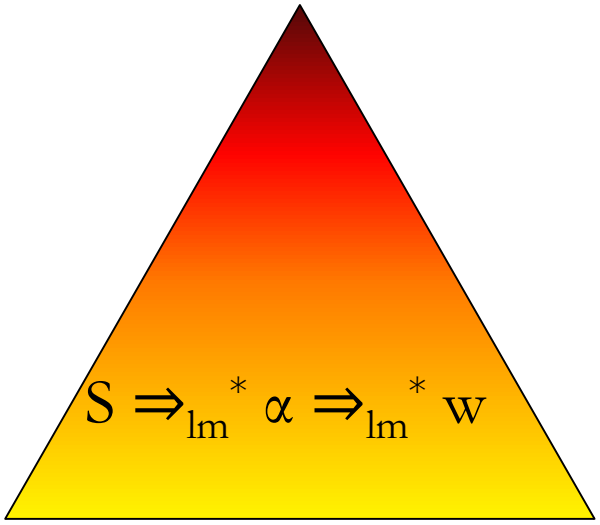
- 一个上下文无关语言的每一个句子与它自己的一颗语法树构成的集合是这个语言的语法分析。
- 自上而下语法分析是给出句子的最左推导
  - 等价于先根遍历语法树
  - LL(1)分析框架
- 自下而上语法分析是为句子找到最左归约
  - 等价于对语法树的句柄剪枝过程
  - SLR(1)分析框架

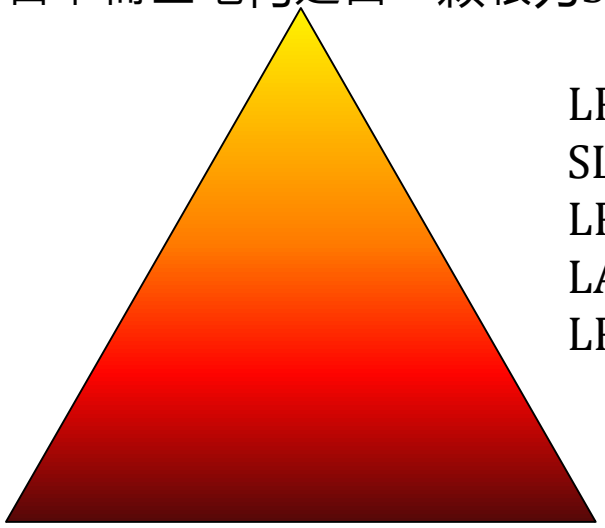
语法分析方案

- 自上而下的分析对于文法要求很高，这又成为该方法的局限：过度修剪文法带来副作用；有大批文法不能转换为LL(1)文法；LL(k)向前查看更多的输入符号代价高。
- 自下而上的分析适用更广泛的文法，优点是不需要像LL分析中那么大量的文法修剪即可进行LR分析。也能容忍一些歧义性存在，如运算符优先级这种常见情形。

寻找到一个最左归约 $w \Leftarrow_{lm}^{*} \beta \Leftarrow_{lm}^{*} S$；

以w为产物，自下而上地构建出一颗根为S的语法树。





LR(0)
SLR(1)
LR(1)
LALR
LR(k)

从归约的非确定性说起

- **定义8.1** 归约 $w \Leftarrow^* \alpha \Leftarrow^* \ldots$ 是非确定性的，如果存在如下两种情形之一：

- （情形1）归约产生的符号串中含有多个可归约串，形式地，$\alpha=\delta_i\gamma_i\eta_i, \delta_i, \eta_i\in(\mathbb{V}\cup T)^*, \gamma_i\in ran(\mathcal{P}), 1\leq i\leq n, n>1$；

- （情形2）多个变元的候选式与一个可归约串相同，即 $\alpha=\delta\gamma\eta, \delta,\eta\in(\mathbb{V}\cup T)^*, (A_i, \gamma)\in\mathcal{P}, 1\leq i\leq m, m>1$。

- 最左归约对情形1有了限定，限定为最左可归约串。

最左归约的非确定性

- **定义8.3**非确定性的最左归约形如，
- $w \Leftarrow_{lm}^{*} \rho x \Leftarrow_{lm}^{*} \ldots$, s.t. $\rho=\delta\gamma\beta \wedge \gamma \in ran(\mathcal{P}) \rightarrow \beta=\varepsilon, x \in \tau(w)$

- 其中，存在：
- （情形1）ρ有多个后缀都是可归约串，形式地，
  $|\tau(\rho) \cap ran(\mathcal{P})|>1$。

- （情形2）有共享候选式是ρ的可归约串，形式地，
  $\gamma \in (\tau(\rho) \cap ran(\mathcal{P}))$, $(A_i, \gamma) \in \mathcal{P}$, $i=1,\ldots,n$, $n>1$。

可是基于栈的过程

移进-归约（shift-reduce）分析框架

- 语法分析任务：寻找到一个最左归约或否定之
  - $w \Leftarrow_{lm}^{*} \beta \Leftarrow_{lm}^{*} S$
- 利用栈来完成任务（物理空、逻辑空）
  - $\varepsilon \cdot w \Leftarrow_{lm}^{*} \rho \cdot ay \Leftarrow_{lm}^{*} \alpha\gamma \cdot ay \Leftarrow_{lm} \alpha A \cdot ay \Leftarrow_{lm}^{*} S \cdot \varepsilon$
  - $Z_0 \cdot w\# \Leftarrow_{lm}^{*} Z_0\rho \cdot ay\# \Leftarrow_{lm}^{*} Z_0\alpha\gamma \cdot ay\# \Leftarrow_{lm} Z_0\alpha A \cdot ay\# \Leftarrow_{lm}^{*} Z_0 S \cdot \#$
- 形成移进-归约（shift-reduce）分析框架：
  - 初始化栈为Z_0，剩余串为w#，形成句型$Z_0 \cdot w\#$。
  - 过程中对于当前句型 $Z_0\rho \cdot ay\#$，若有$\gamma \in \tau(\rho)$且$(A, \gamma) \in \mathcal{P}$，则$\rho = \alpha\gamma$被更新为$\alpha A$并形成直接归约$\alpha\gamma \cdot ay \Leftarrow_{lm} \alpha A \cdot ay$（归约），否则$\rho$被更新为$\rho a$形成0步归约$\rho \cdot ay \Leftarrow_{lm}^{*} \rho a \cdot y$（移进）
  - 过程进行到句型为$Z_0 S \cdot \#$时得出接受之结论。
  - 过程中若出现其它情形，则拒绝并可报出语法错误。

进一步确定化

- 规范归约⇨
  - $Z_0w\# \Rightarrow^* Z_0\alpha\gamma\cdot y\# \Rightarrow Z_0\alpha A\cdot y\# \Rightarrow^* Z_0\beta\cdot az\# \Rightarrow Z_0\beta a\cdot z\# \Rightarrow^* Z_0S\cdot\#$
    其中：γ为A的候选式；β的所有后缀都不是可归约串。
- 规范归约模拟器PDA N
- LR分析框架
- LR分析框架的程序实现
  - LR(0)分析表
  - SLR(1)分析表
  - ~~LR(1)分析表~~

5.1 自下而上分析基本问题

- 自下而上分析：
  - 从输入串开始连续归约，每个归约步都是将当前符号串中某个候选式的出现替换成对应变元。如此连续归约，直到归约成文法开始符号，表示分析成功。
  - 我们限定归约策略为一种，为最左归约（试图找到确定性的最左归约）。

- 具体化为“移进-归约”分析法：
  - 用一个符号栈，把输入符号一个一个地移进到栈里，当栈顶形成某个产生式的一个候选式时，即把栈顶的这一部分（称为可归约串）替换为该产生式左部变元。
  - 分析过程中每一步直接归约都与语法树构建一一对应。

Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

Stack

Input String

num	*	(	num	+	num	)

Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

num	*	(	num	+	num	)

Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow (Expr)$

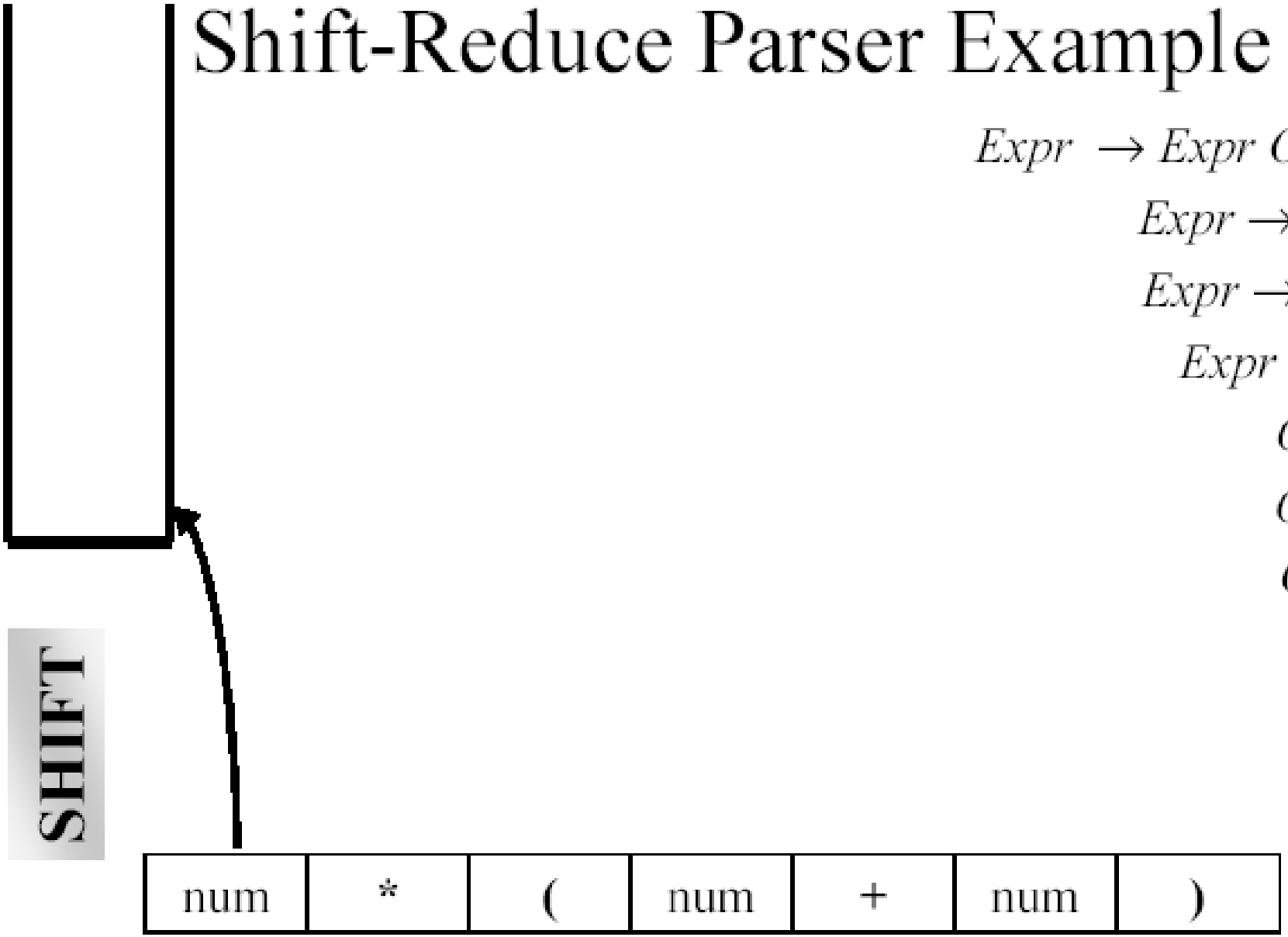
$Expr \rightarrow -\ Expr$

$Expr \rightarrow \text{num}$

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$



Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

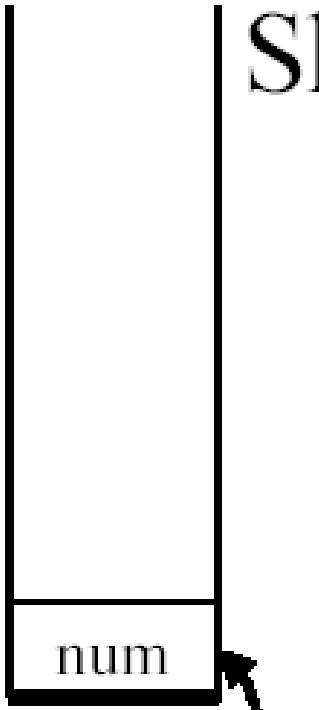
$Expr \rightarrow\ -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow\ +$

$Op \rightarrow\ -$

$Op \rightarrow\ *$



SHIFT

*	(	num	+	num	)

Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

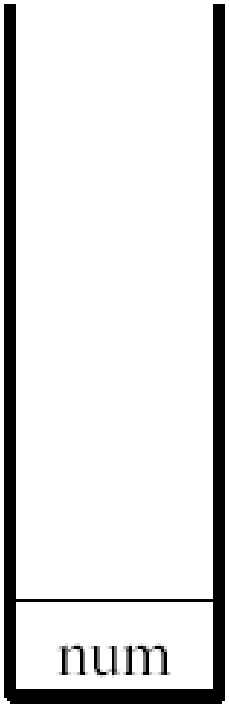
$Expr \rightarrow\ -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow\ +$

$Op \rightarrow\ \ -$

$Op \rightarrow\ *$



REDUCE

*	(	num	+	num	)

Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow (Expr)$

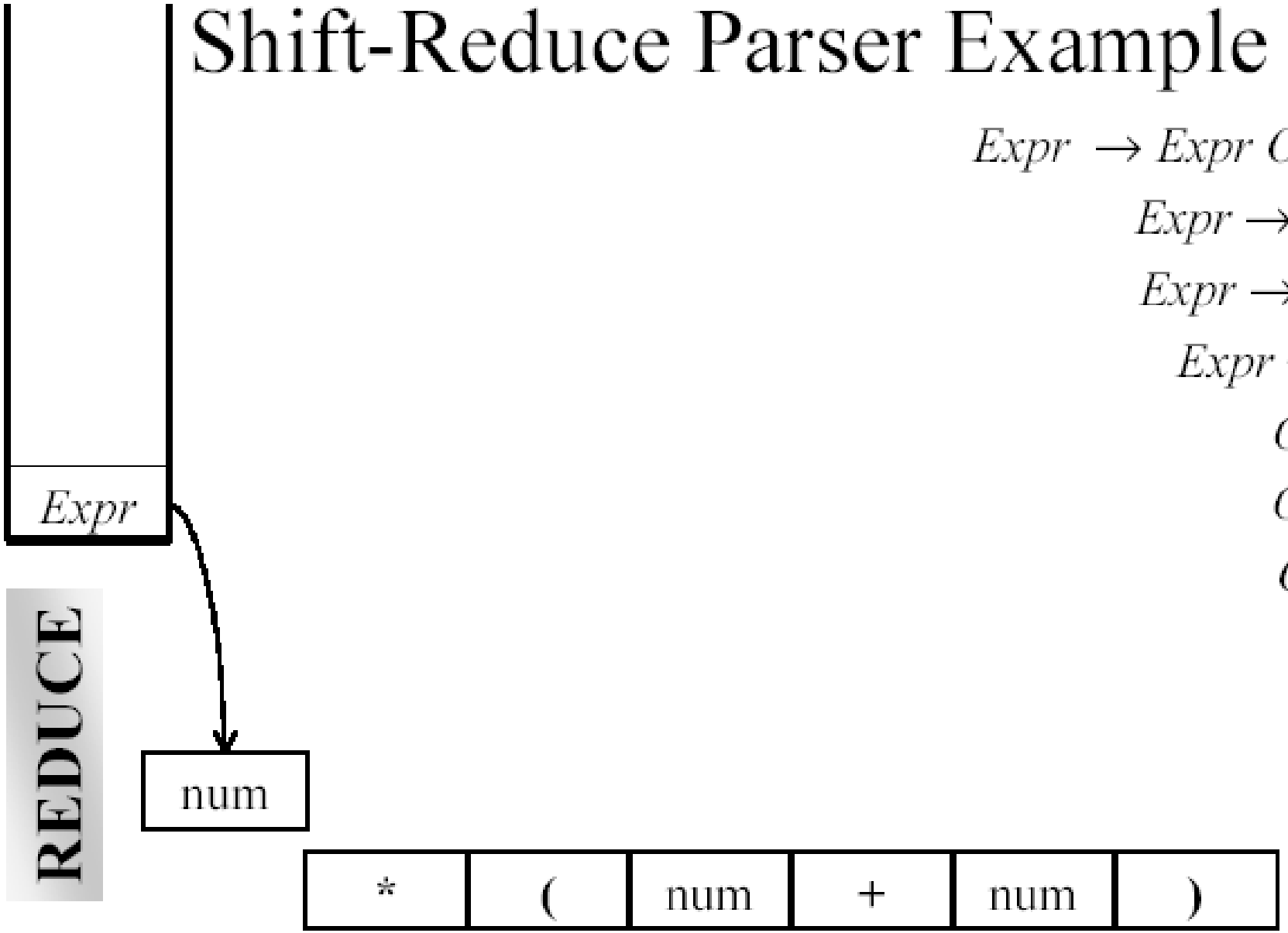
$Expr \rightarrow -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$



Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow (Expr)$

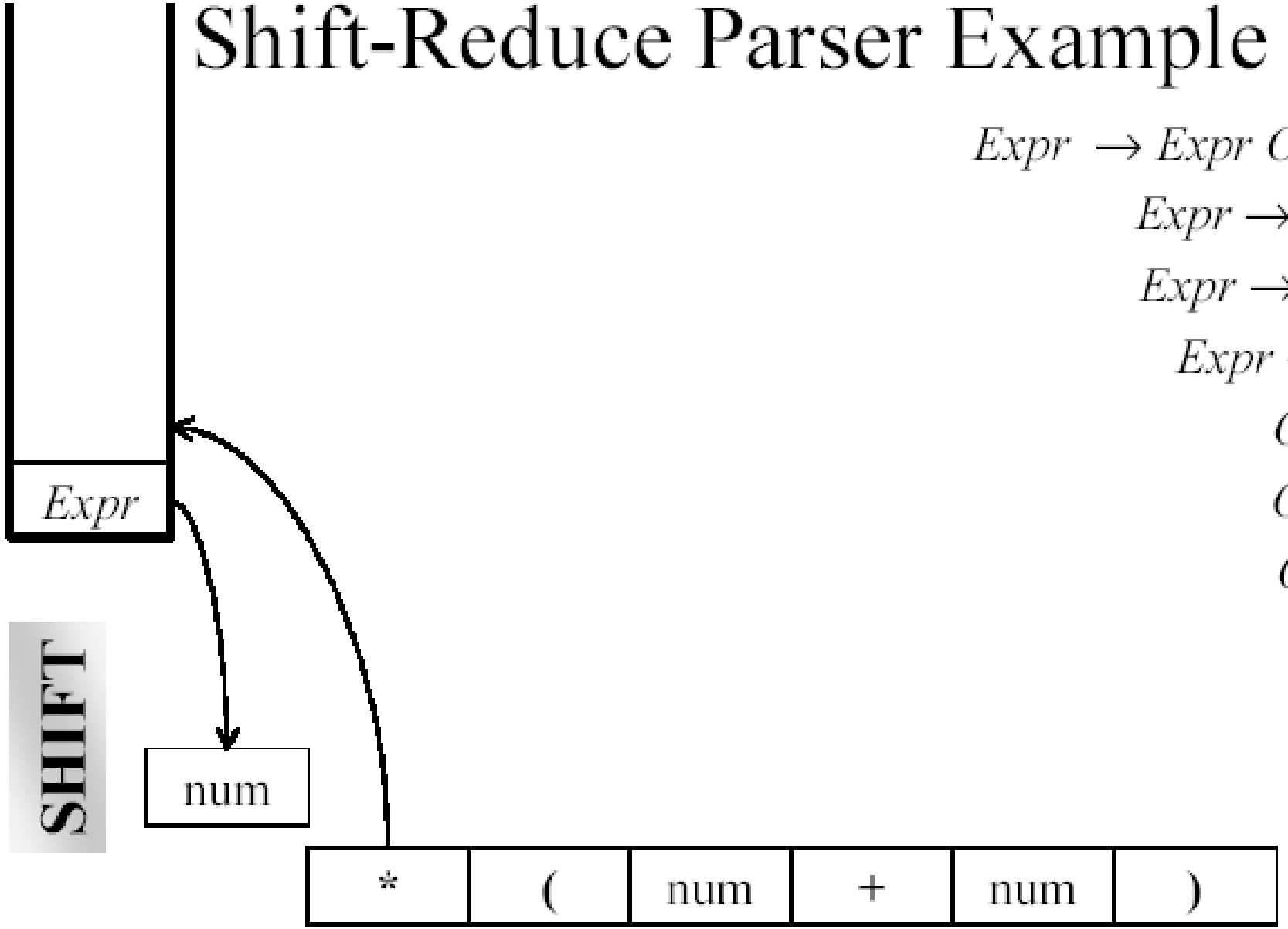
$Expr \rightarrow -\ Expr$

$Expr \rightarrow \text{num}$

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$



Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow (Expr)$

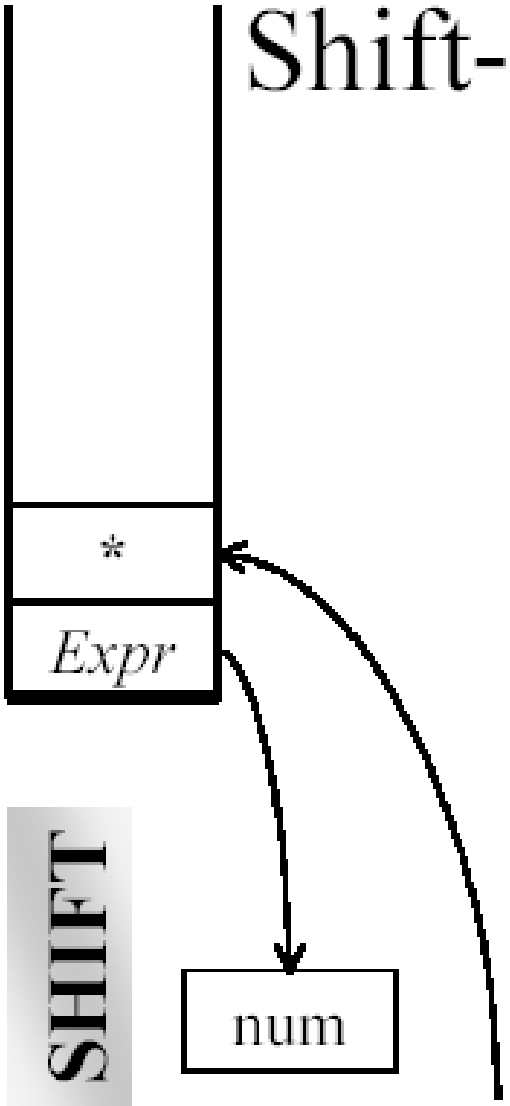
$Expr \rightarrow -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$



(	num	+	num	)

Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

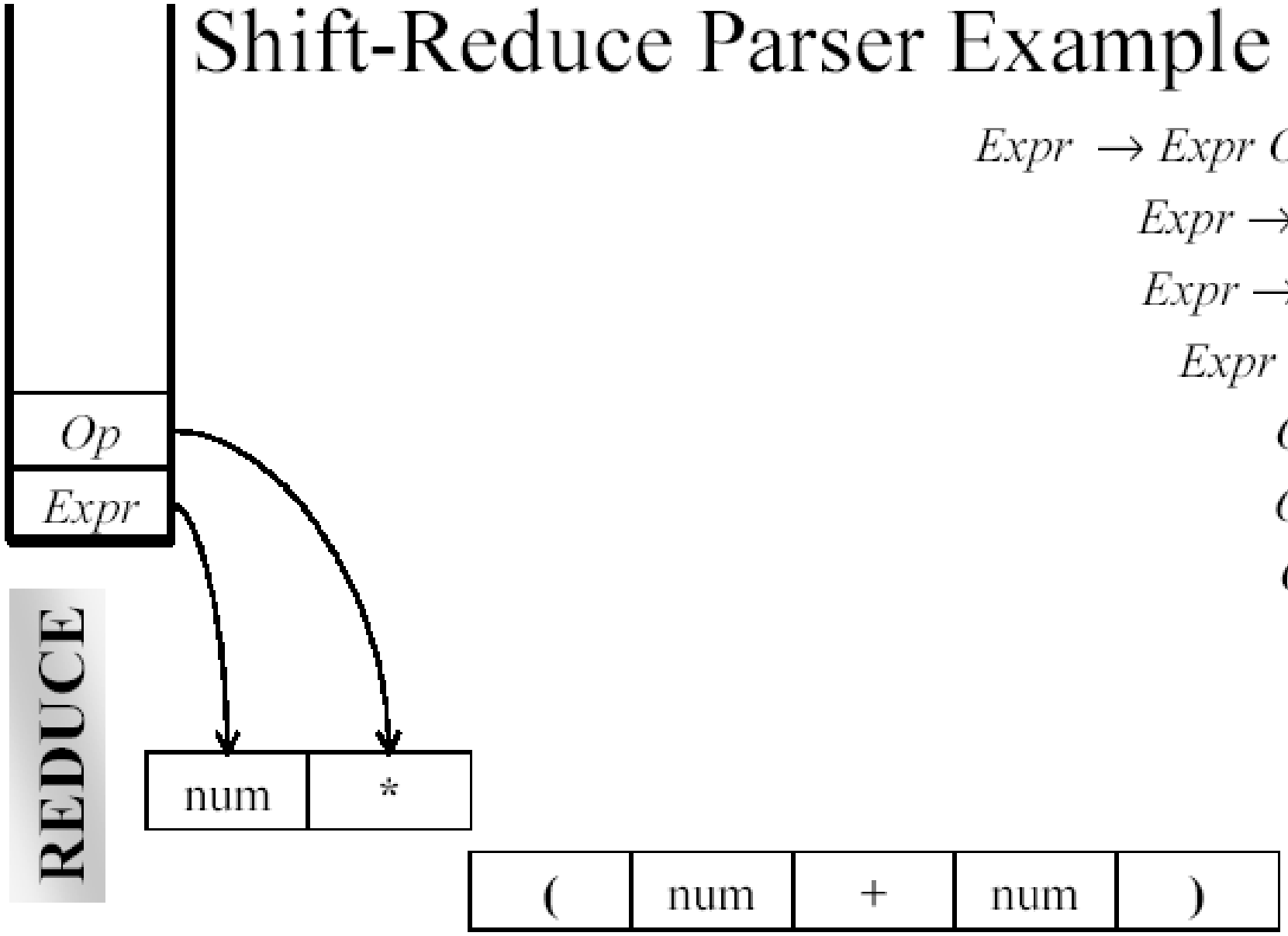
$Expr \rightarrow\ -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow\ +$

$Op \rightarrow\ -$

$Op \rightarrow\ *$



Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow (Expr)$

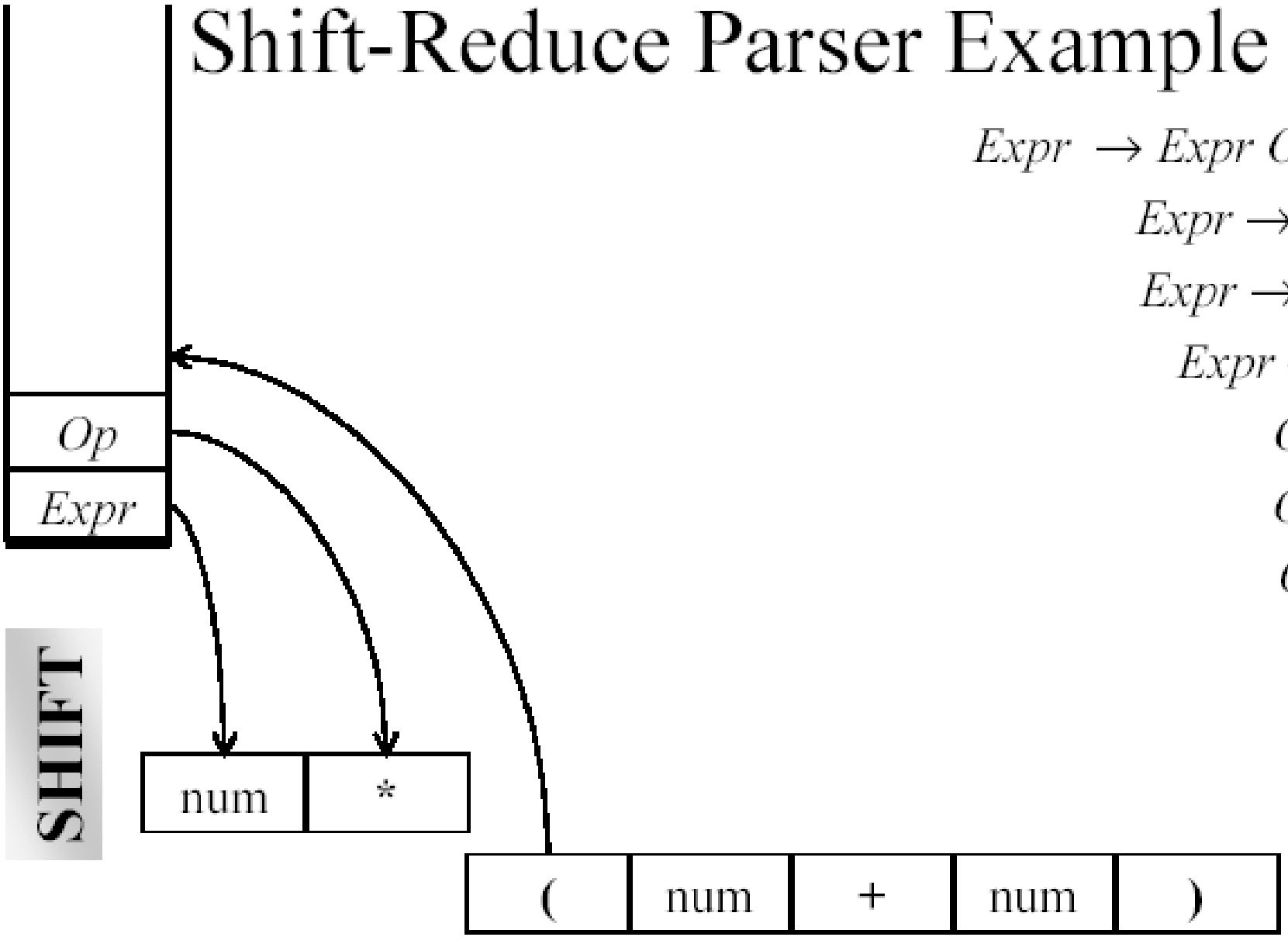
$Expr \rightarrow -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$



Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

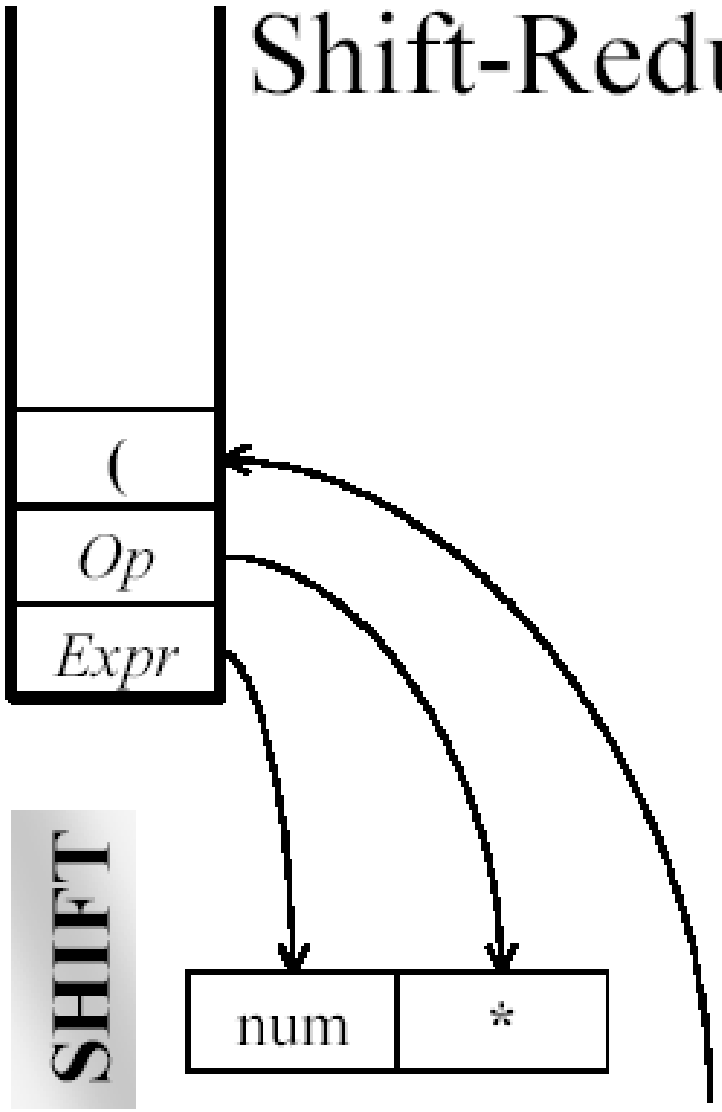
$Expr \rightarrow\ -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow\ +$

$Op \rightarrow\ -$

$Op \rightarrow\ *$



num	+	num	)

Shift-Reduce Parser Example

Expr → *Expr Op Expr*

Expr → (*Expr*)

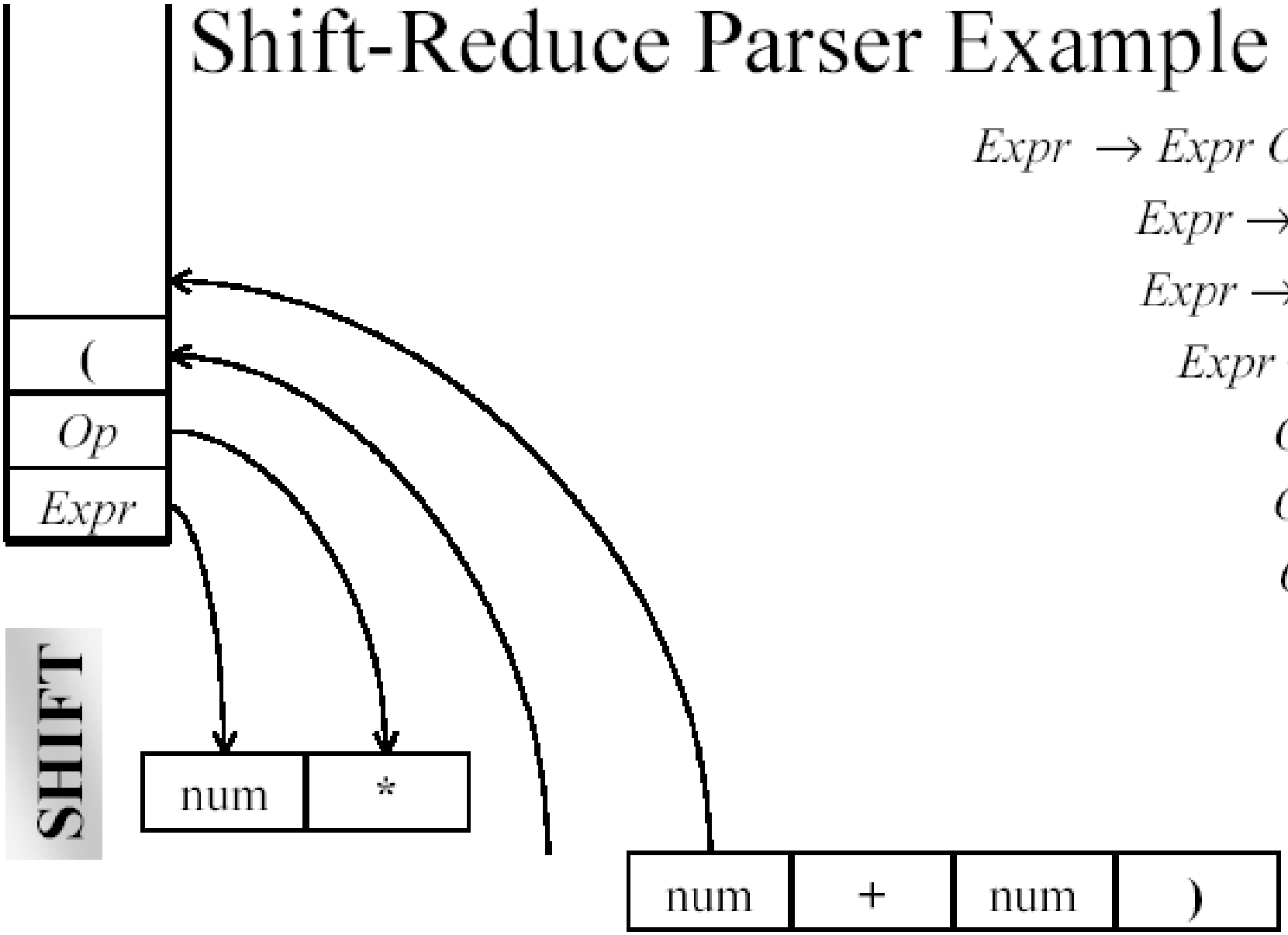
Expr → - *Expr*

Expr → num

Op → +

Op → -

Op → *



Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow (Expr)$

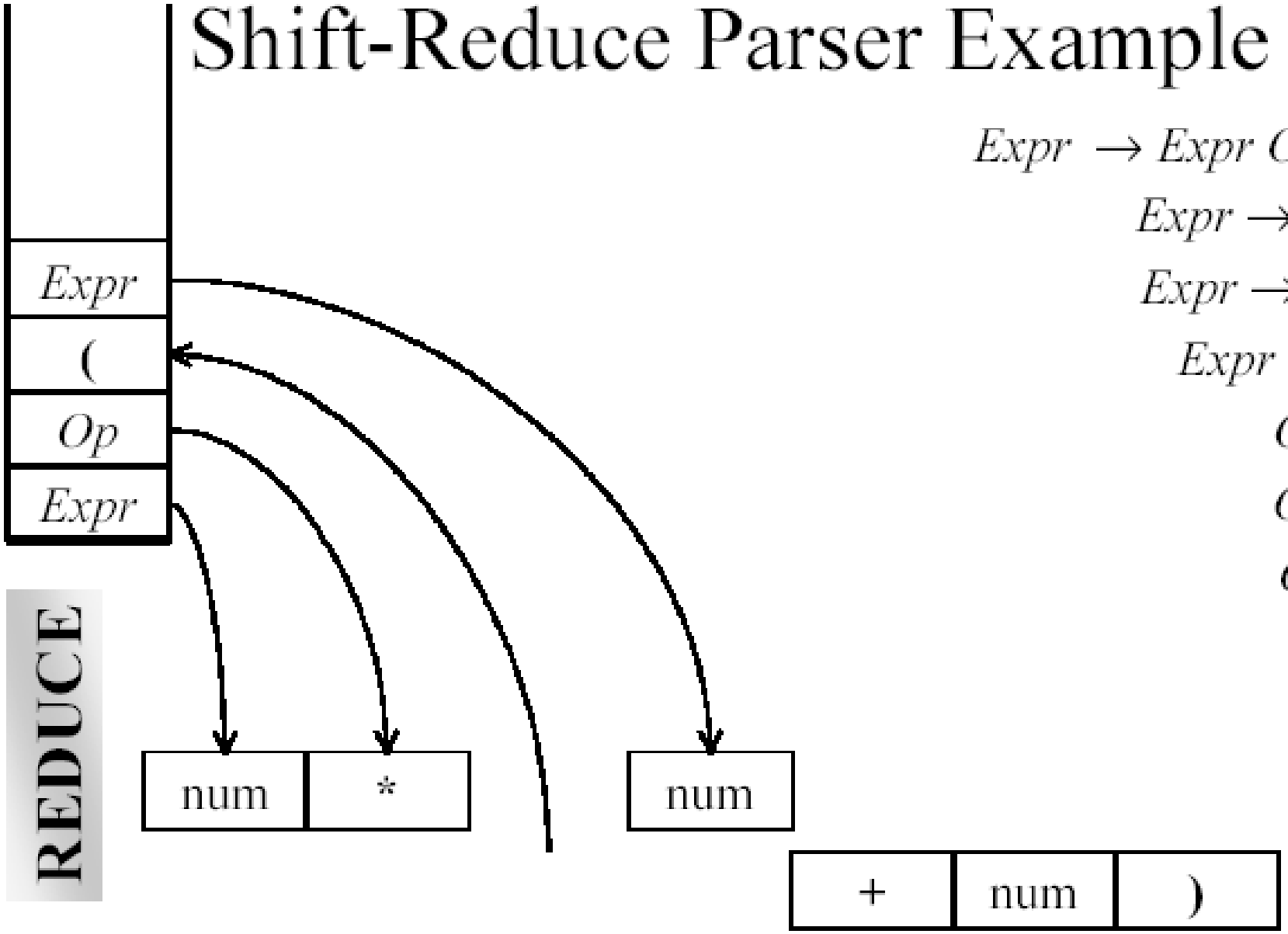
$Expr \rightarrow -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$



Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

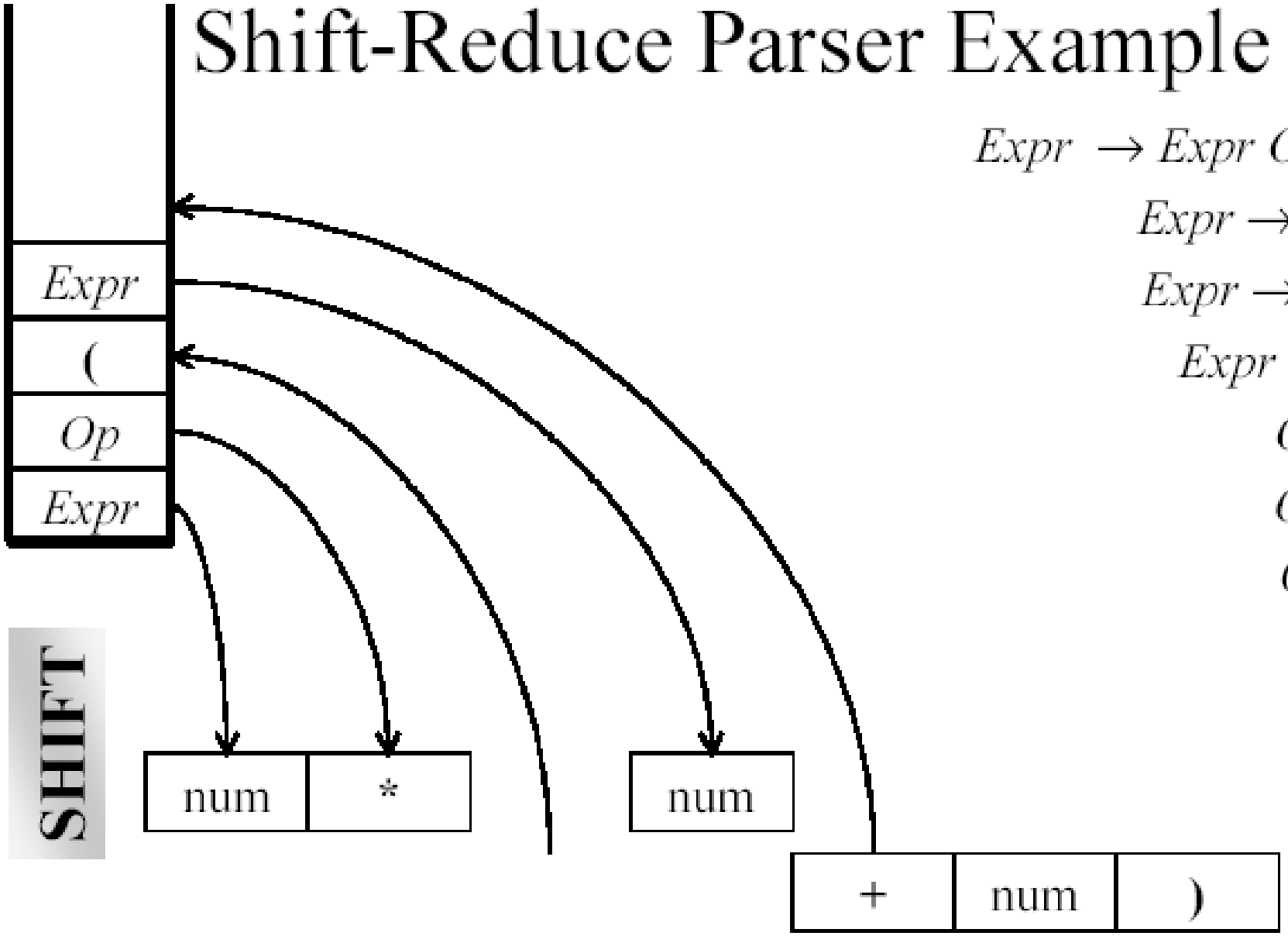
$Expr \rightarrow\ -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow\ +$

$Op \rightarrow\ -$

$Op \rightarrow\ *$



Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

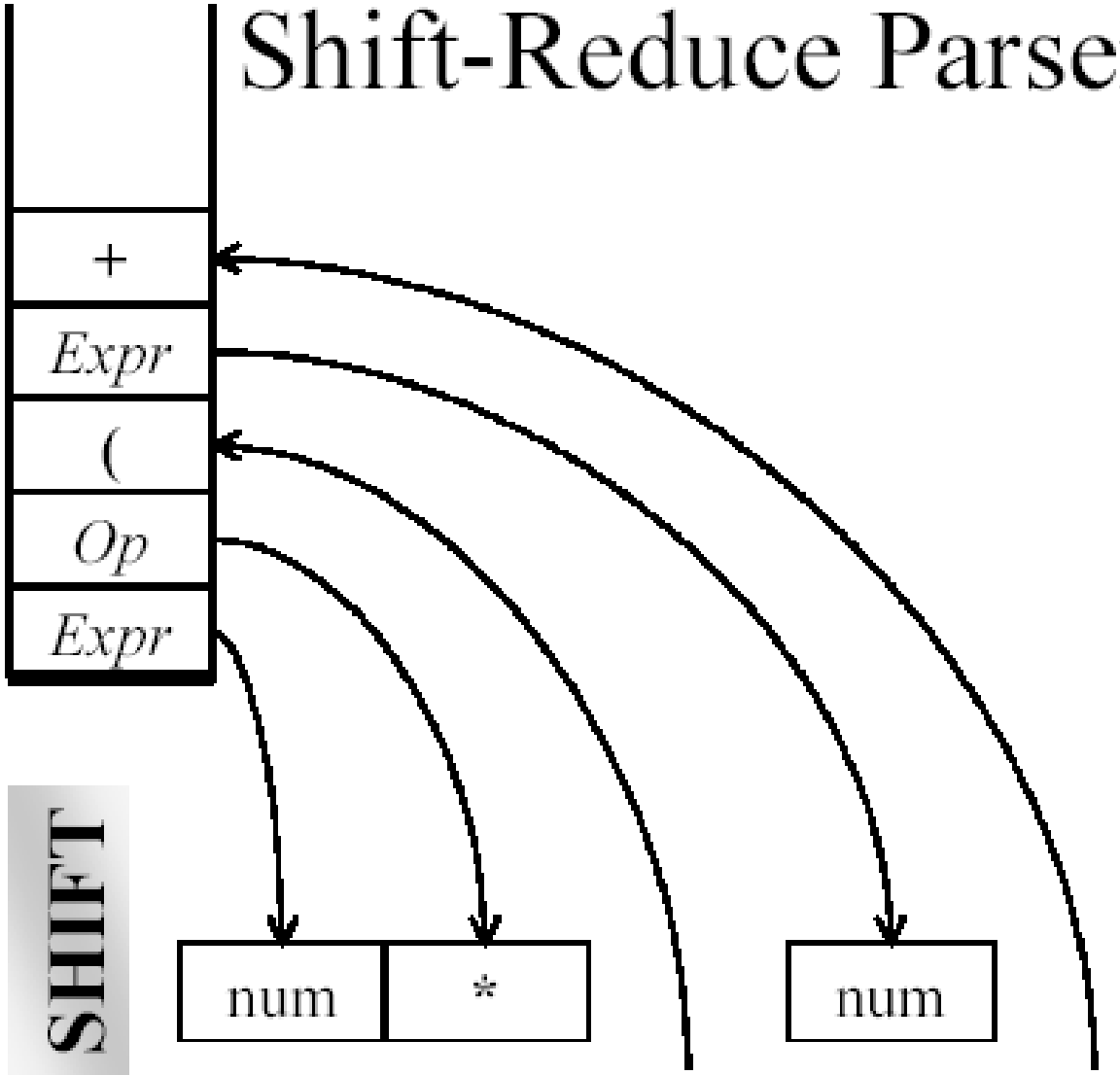
$Expr \rightarrow\ -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow\ +$

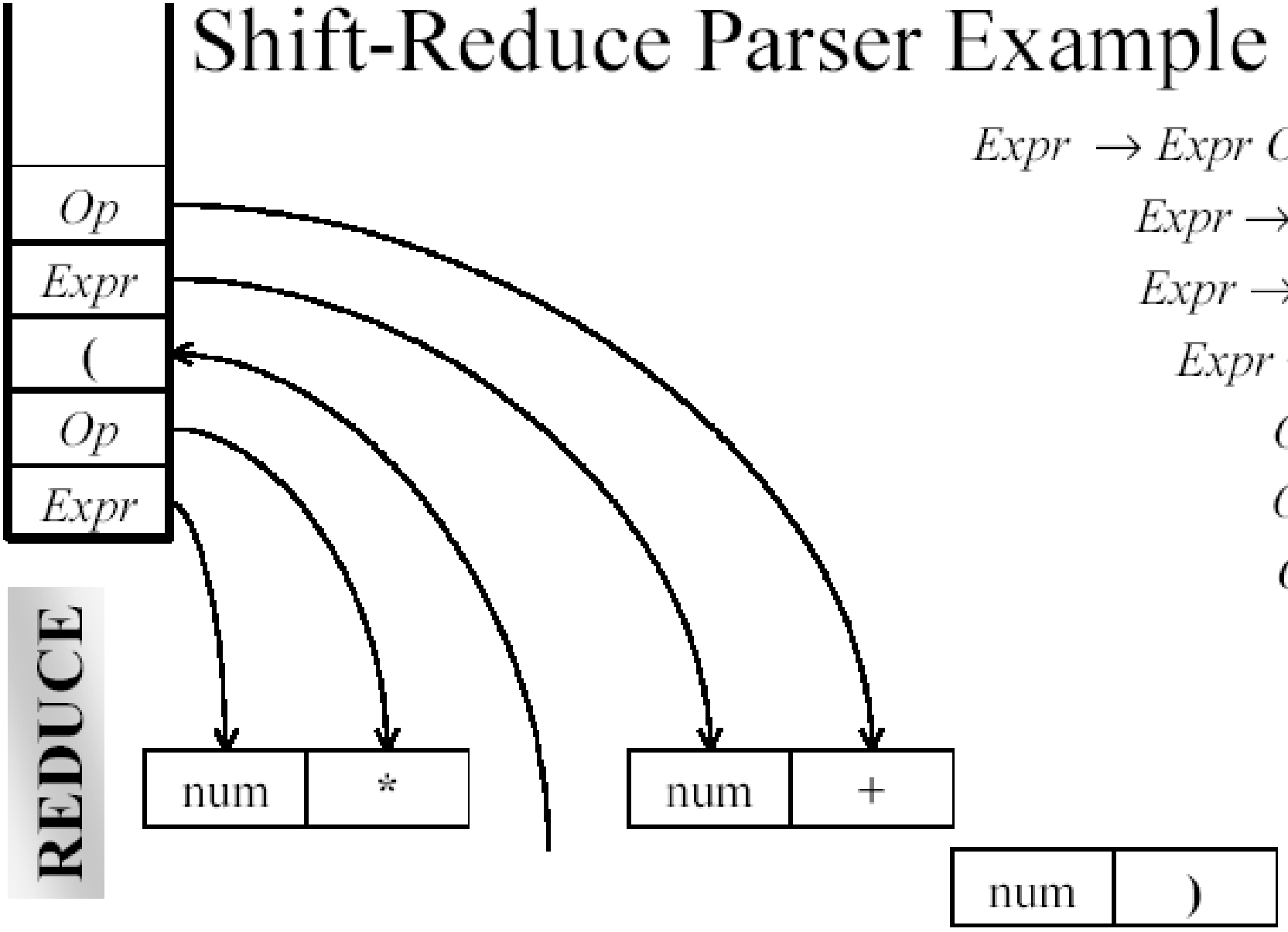
$Op \rightarrow\ -$

$Op \rightarrow\ *$



num	)

Shift-Reduce Parser Example



$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

$Expr \rightarrow\ -\ Expr$

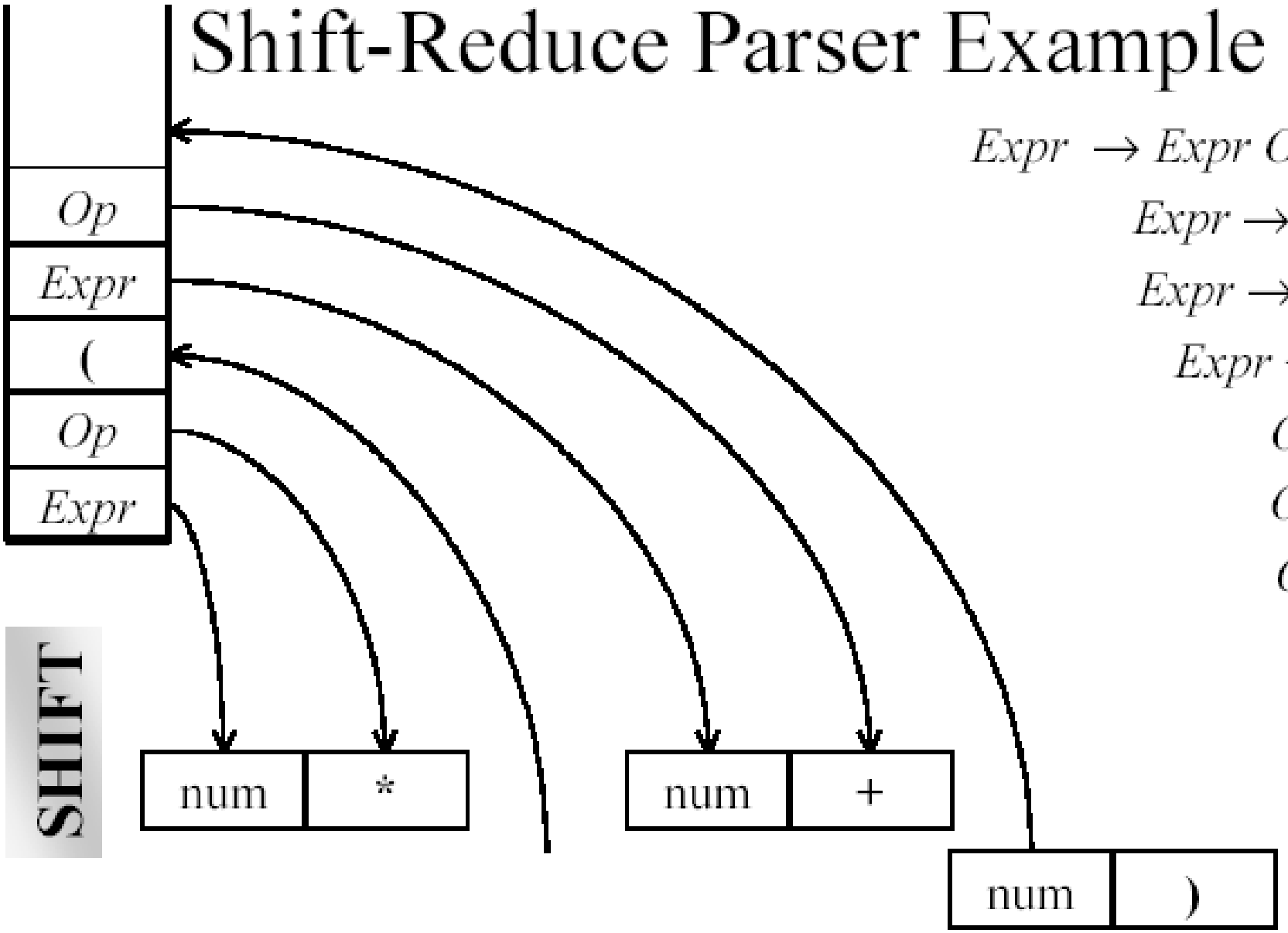
$Expr \rightarrow$ num

$Op \rightarrow\ +$

$Op \rightarrow\ -$

$Op \rightarrow\ *$

Shift-Reduce Parser Example



$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

$Expr \rightarrow\ -\ Expr$

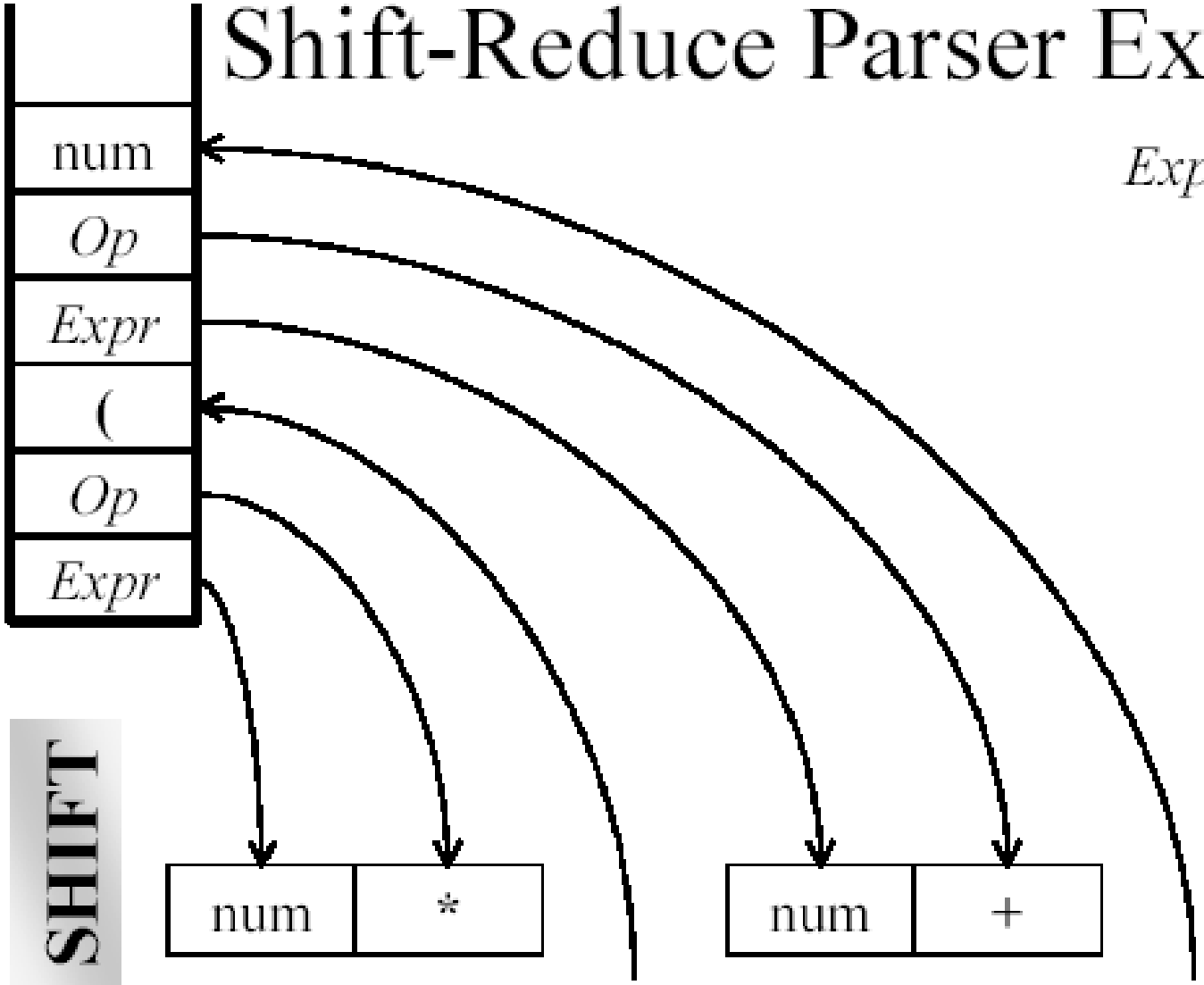
$Expr \rightarrow$ num

$Op \rightarrow\ +$

$Op \rightarrow\ -$

$Op \rightarrow\ *$

Shift-Reduce Parser Example



$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

$Expr \rightarrow\ -\ Expr$

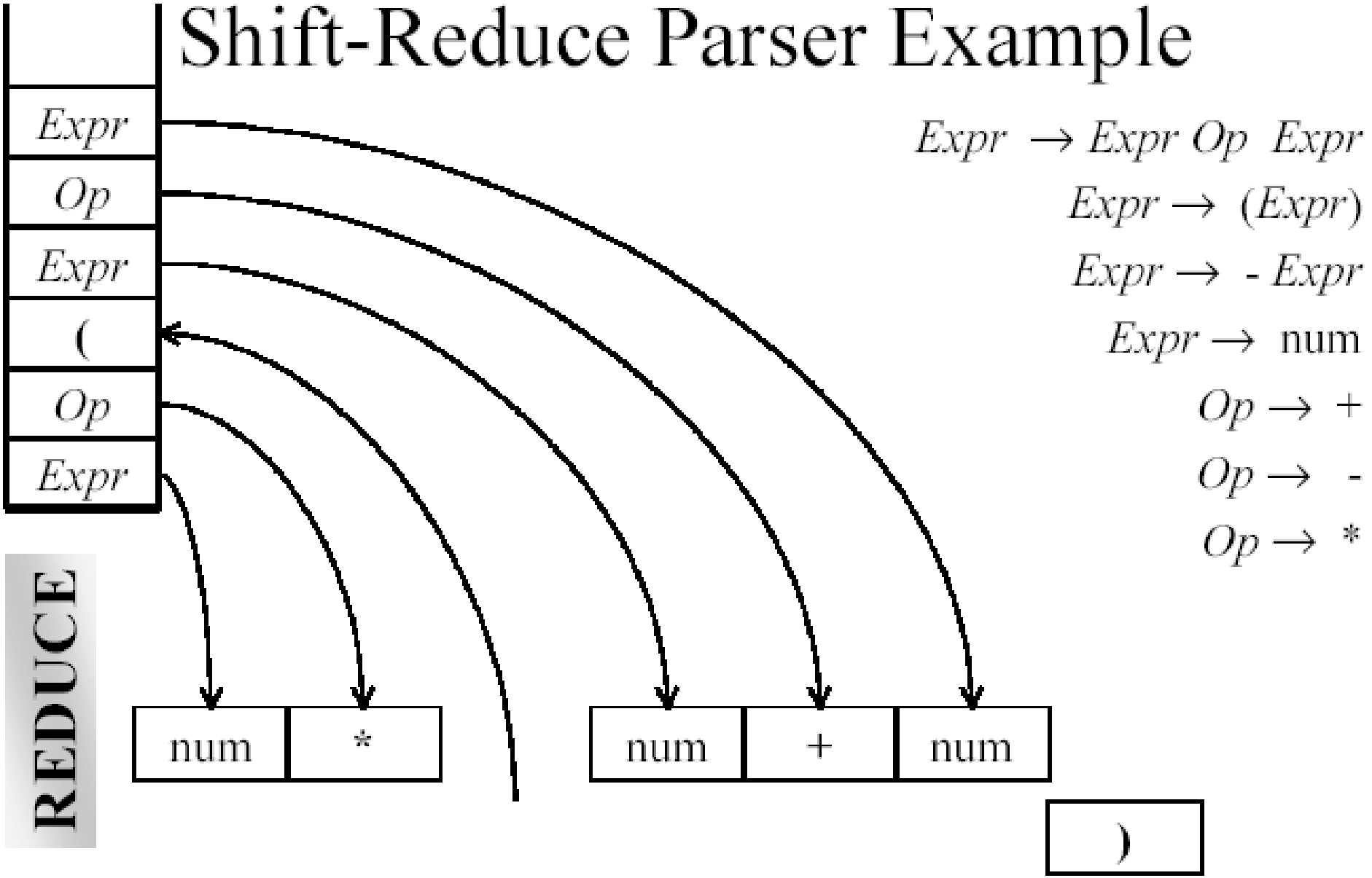
$Expr \rightarrow$ num

$Op \rightarrow\ +$

$Op \rightarrow\ -$

$Op \rightarrow\ *$

Shift-Reduce Parser Example



Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

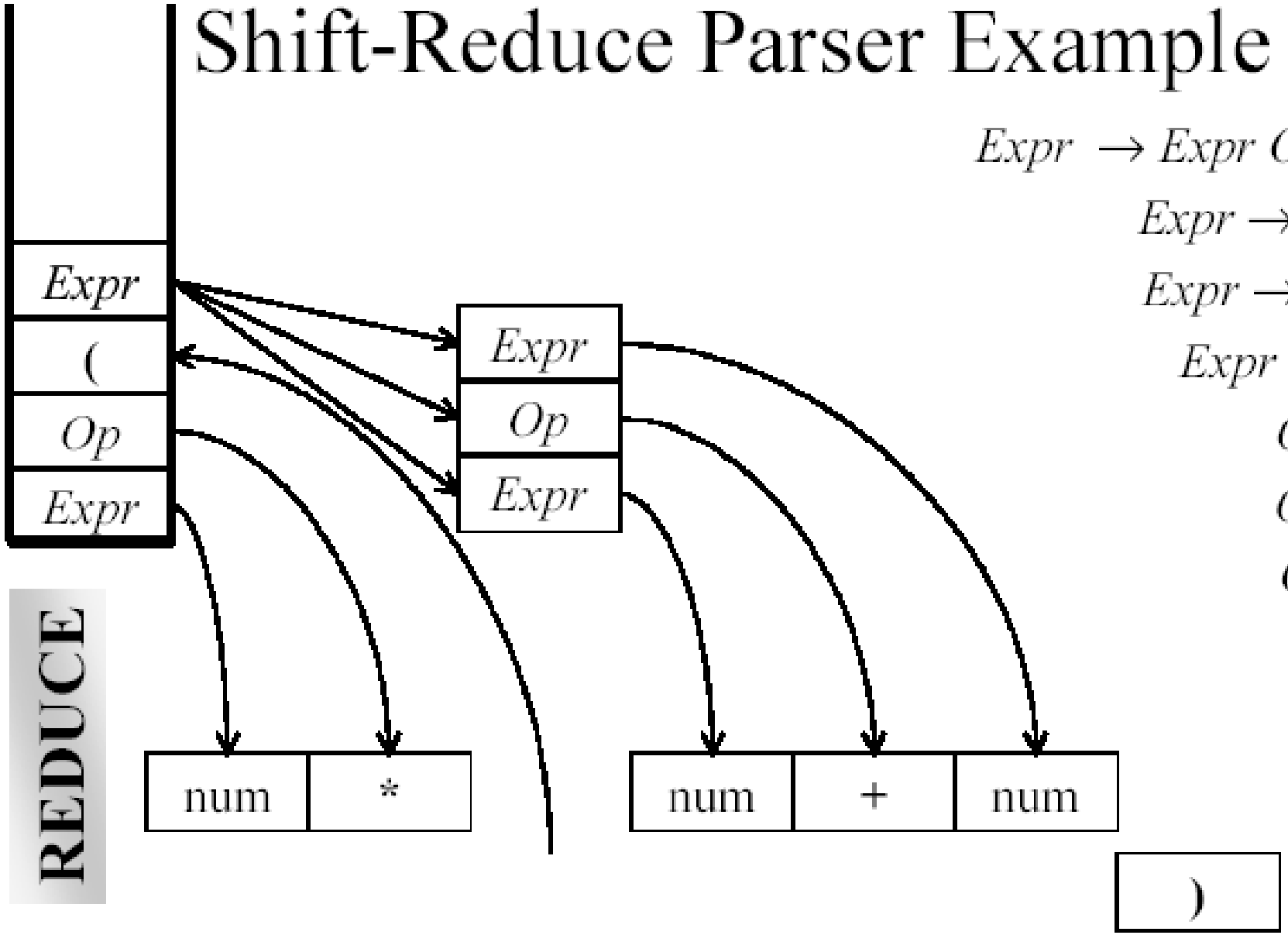
$Expr \rightarrow\ -\ Expr$

$Expr \rightarrow$ num

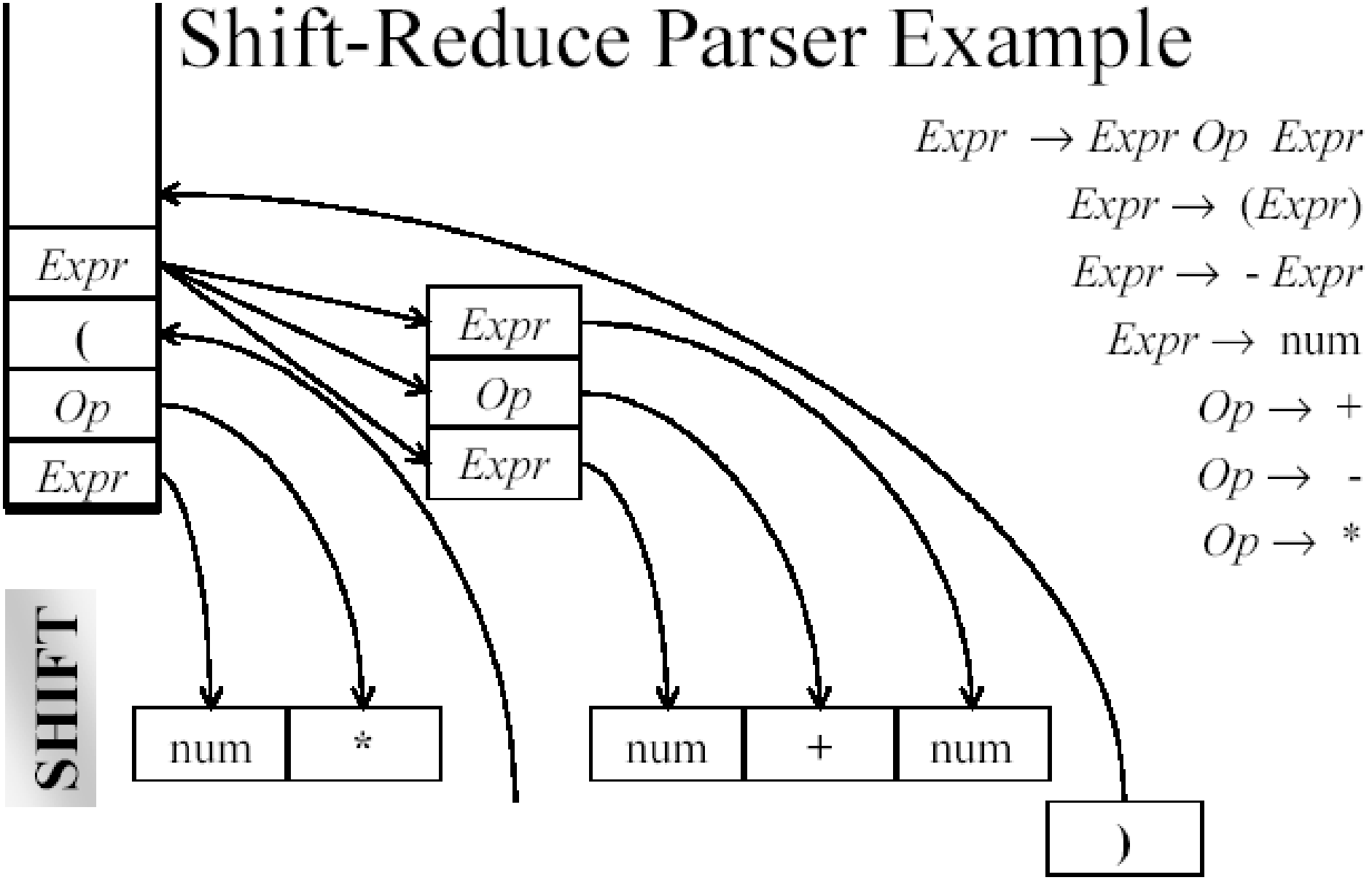
$Op \rightarrow\ +$

$Op \rightarrow\ -$

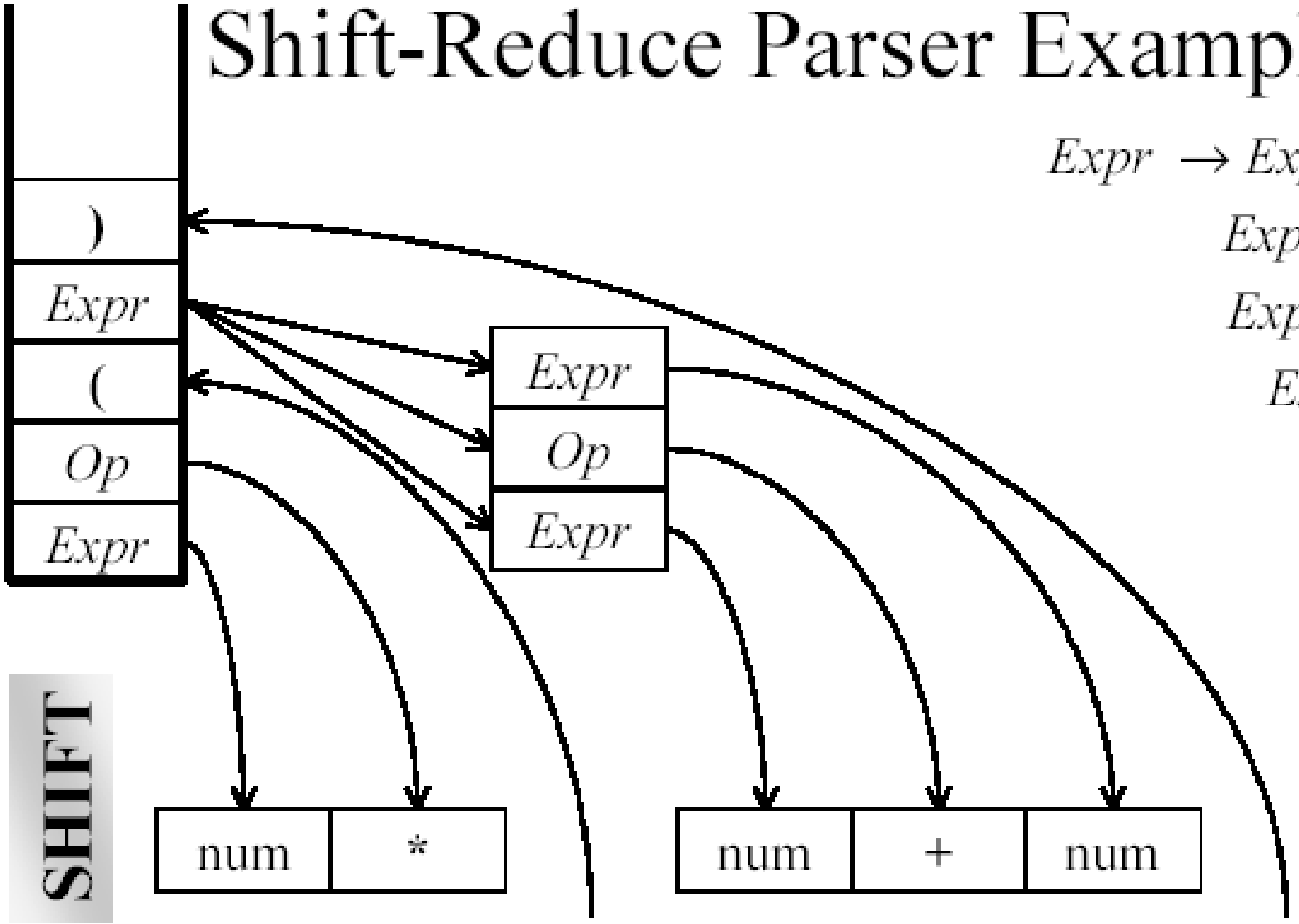
$Op \rightarrow\ *$



Shift-Reduce Parser Example



Shift-Reduce Parser Example



$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

$Expr \rightarrow\ -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow\ +$

$Op \rightarrow\ -$

$Op \rightarrow\ *$

Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow (Expr)$

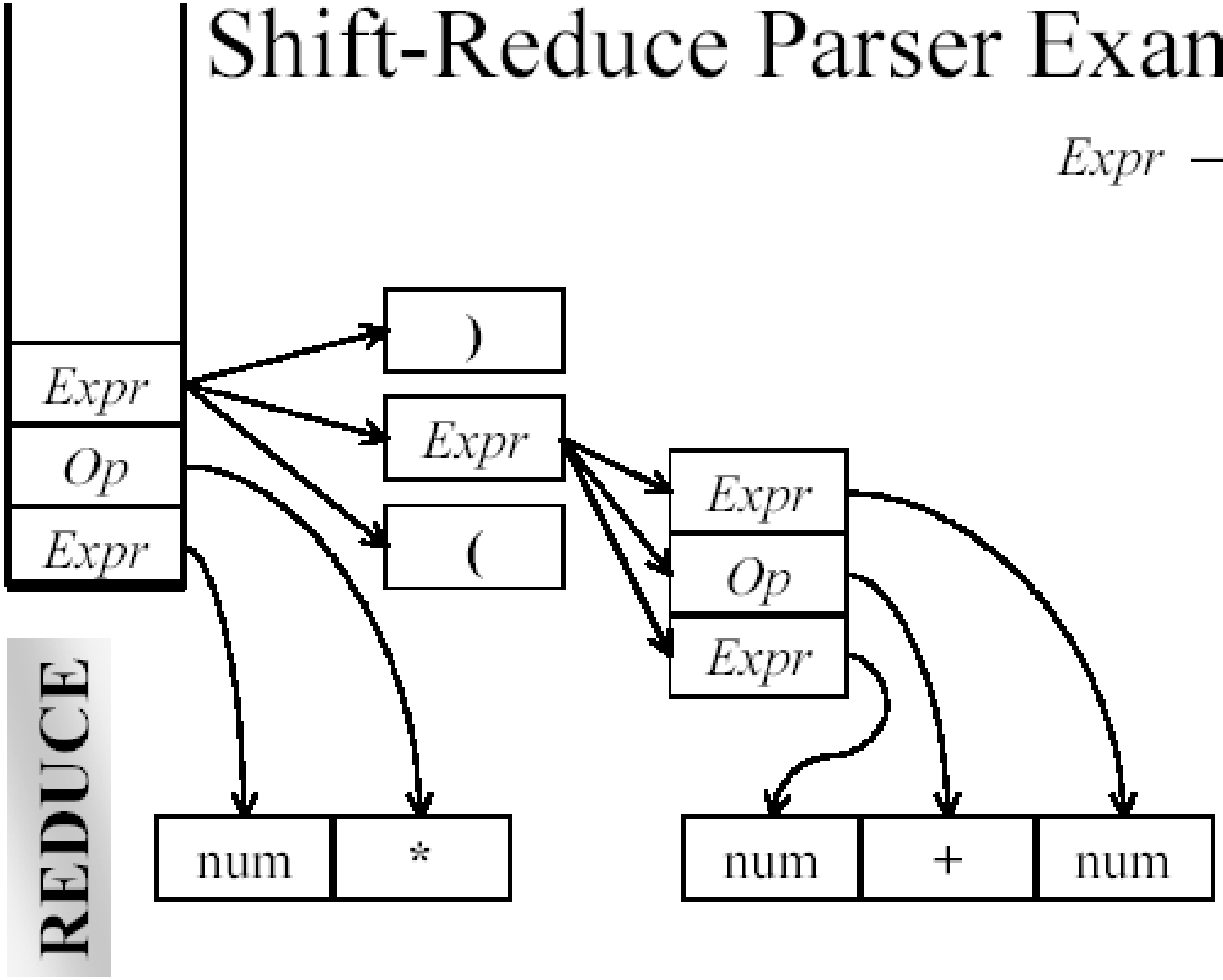
$Expr \rightarrow -Expr$

$Expr \rightarrow num$

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$



Shift-Reduce Parser Example

$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

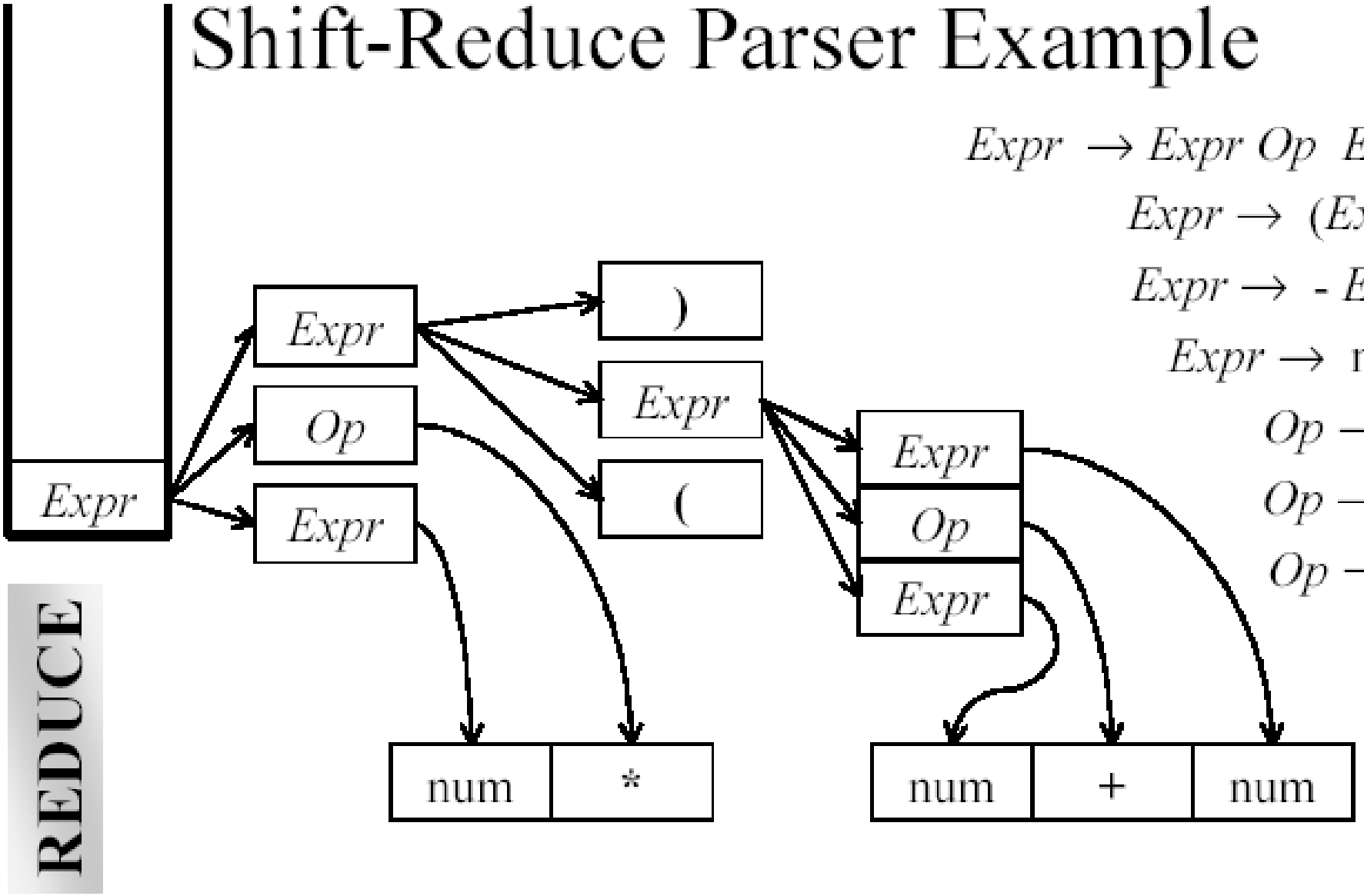
$Expr \rightarrow\ -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow\ +$

$Op \rightarrow\ -$

$Op \rightarrow\ *$



Shift-Reduce Parser Example

Expr → *Expr Op Expr*

Expr → (*Expr*)

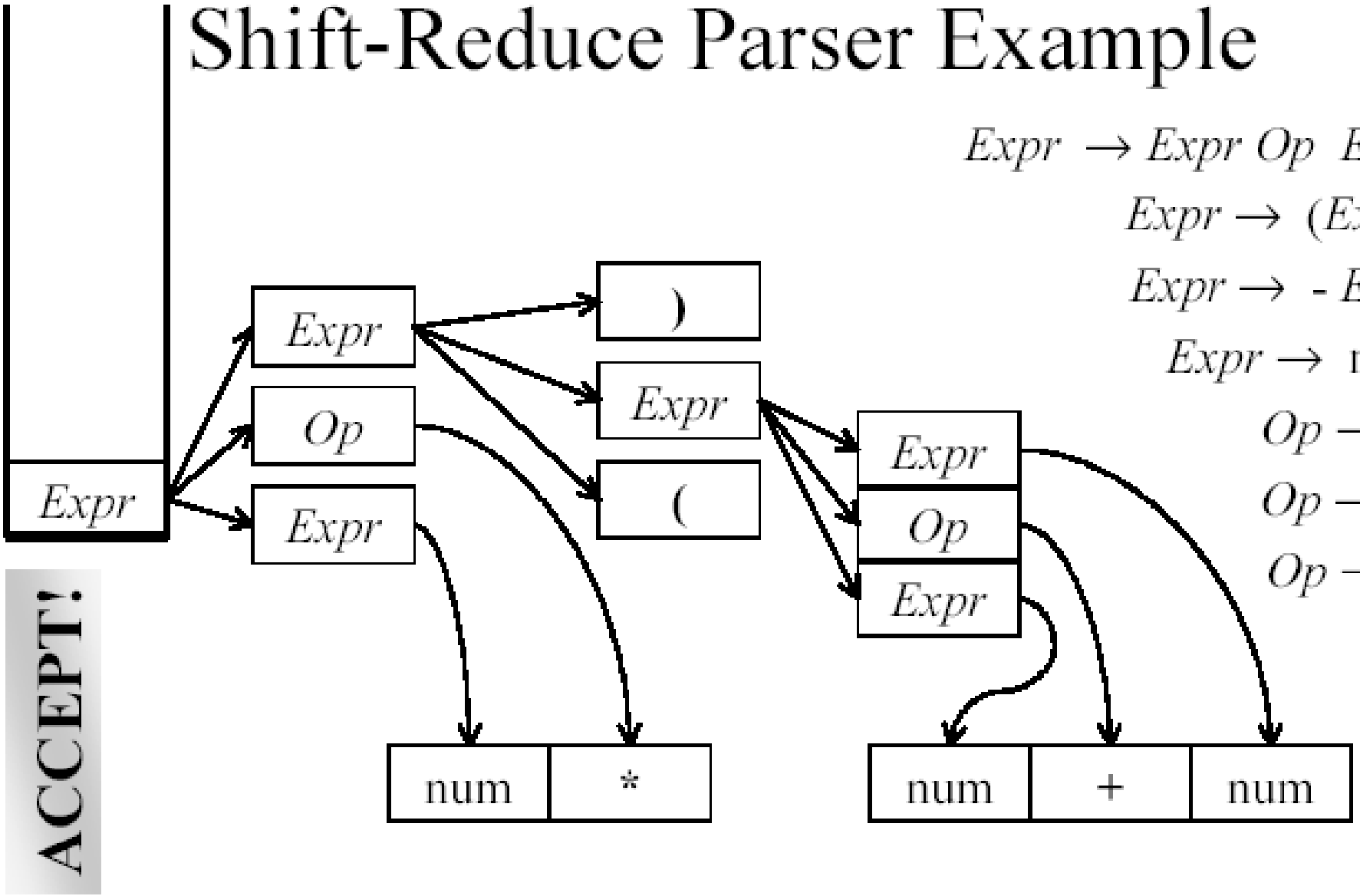
Expr → - *Expr*

Expr → num

Op → +

Op → -

Op → *



观 察

- 一个不断移进-归约的过程，从输入串开始，到开始符号结束或者出错结束。
- 如果能成功的话，在过程的每一步都得到一个句型。
- 在过程的每一步，总是做出如下的一个选择：
  - 输入串中的当前输入符号被移进栈，即压入到栈顶，当前输入符号更改为下一个。
  - 如果栈顶部分的符号串刚好跟某个变元A的某个候选式一样，则这部分全部出栈，然后将A压入栈。这个过程即是归约。将来可以给归约附加一些行为或动作如构造语法树、进行语义分析等。
- 注意到移进-归约分析过程对应于最左归约。

Parsing Stack	Input String	Action
Z_0	num*(num+num)#	shift
Z_0num	*(num+num)#	shift
Z_0*Expr*	*(num+num)#	reduce *Expr* →num
Z_0*Expr**	(num+num)#	shift
Z_0*Expr Op*	(num+num)#	reduce *Op* → *
Z_0*Expr Op*(	num+num)#	shift
Z_0*Expr Op*(num	+num)#	shift
Z_0*Expr Op*(*Expr*	+num)#	reduce *Expr* →num
Z_0*Expr Op*(*Expr*+	num)#	shift
Z_0*Expr Op*(*Expr Op*	num)#	reduce *Op* → +
Z_0*Expr Op*(*Expr Op* num	)#	shift
Z_0*Expr Op*(*Expr Op Expr*	)#	reduce *Expr* →num
Z_0*Expr Op*(*Expr*	)#	red. *Expr*→*Expr Op Expr*
Z_0*Expr Op*(*Expr*)	#	shift
Z_0*Expr Op Expr*	#	reduce *Expr*→(*Expr*)
Z_0*Expr*	#	red. *Expr*→*Expr Op Expr*

栈的变化

- 初始时栈顶为Z_0;
- 移进当前输入符号至栈顶（压入该符号）;
- 当栈顶出现“可归约串”则弹出并将该串的归约变元压入;
- 分析成功时栈顶为开始符号，下面为#;
- 分析不成功时以报错结束。

- 可能的问题:
  - 怎么知道可归约串出现了？如何确定它的归约变元?
  - 如果栈顶有可归约串，是归约还是不归约?
  - 如果栈顶没有可归约串，就移进还是断言为失败?
  - 栈上若有可归约串，它们一定都是在栈顶吗?

分析过程中的冲突

- 知道“可归约串”出现、确定出以它为候选式的变元、移进还是失败，会面临多选的情形。
- 对于确定性的过程而言，多选步骤即就是冲突。

- 宏观来看，确定性的自下而上分析过程面临两种冲突：
  - 栈顶部分可能与多个候选式匹配；
  - 归约-归约冲突

  - 也可能即使有匹配的候选式也不能选，而继续移进直到随后找到其它候选式匹配。
  - 移进-归约冲突

Conflicts

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow$ ($Expr$)

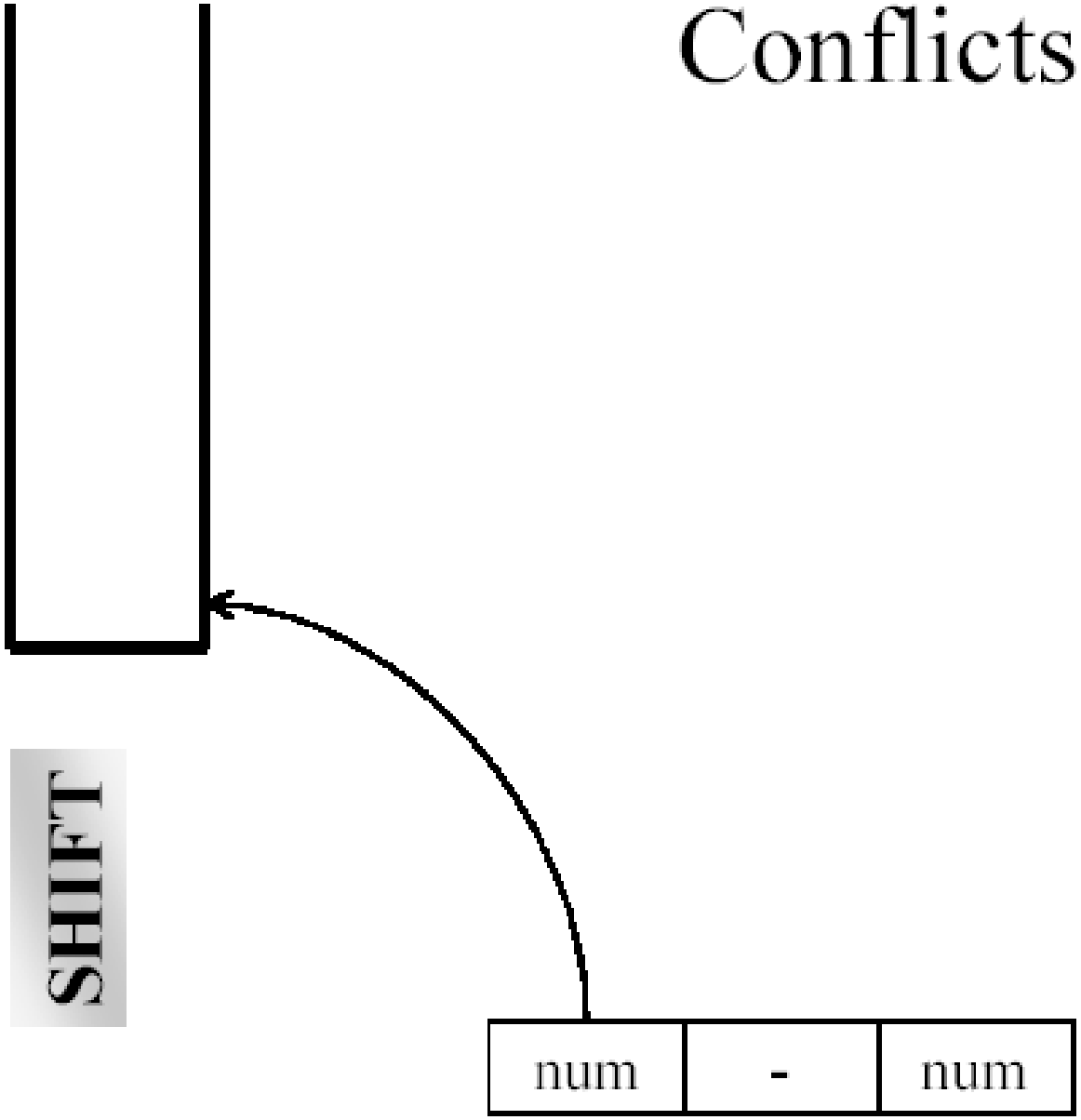
$Expr \rightarrow Expr$ -

$Expr \rightarrow$ num

$Op \rightarrow$ +

$Op \rightarrow$ -

$Op \rightarrow$ *



Conflicts

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow (Expr)$

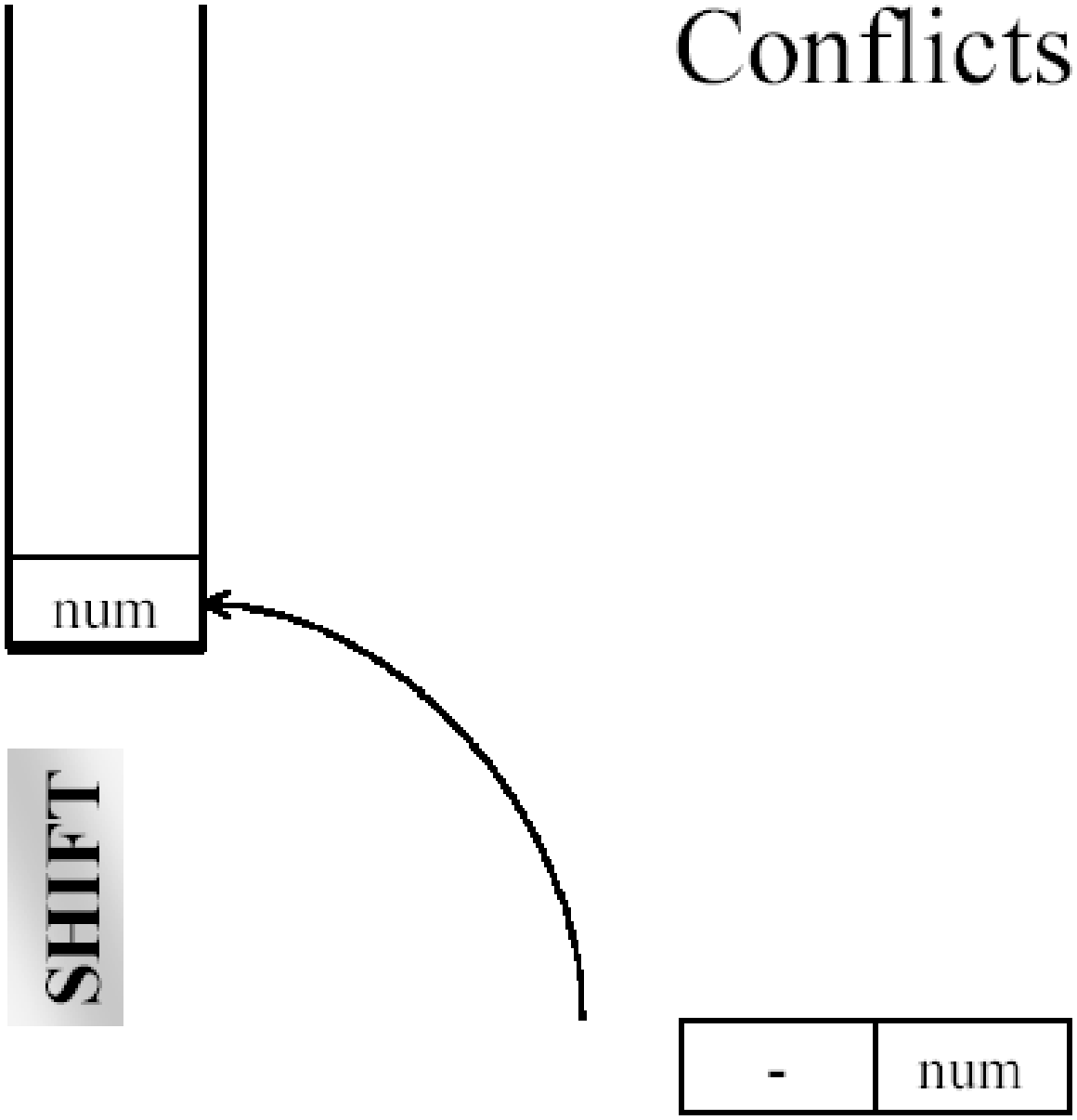
$Expr \rightarrow Expr$ -

$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow$ -

$Op \rightarrow *$



Conflicts

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow$ ($Expr$)

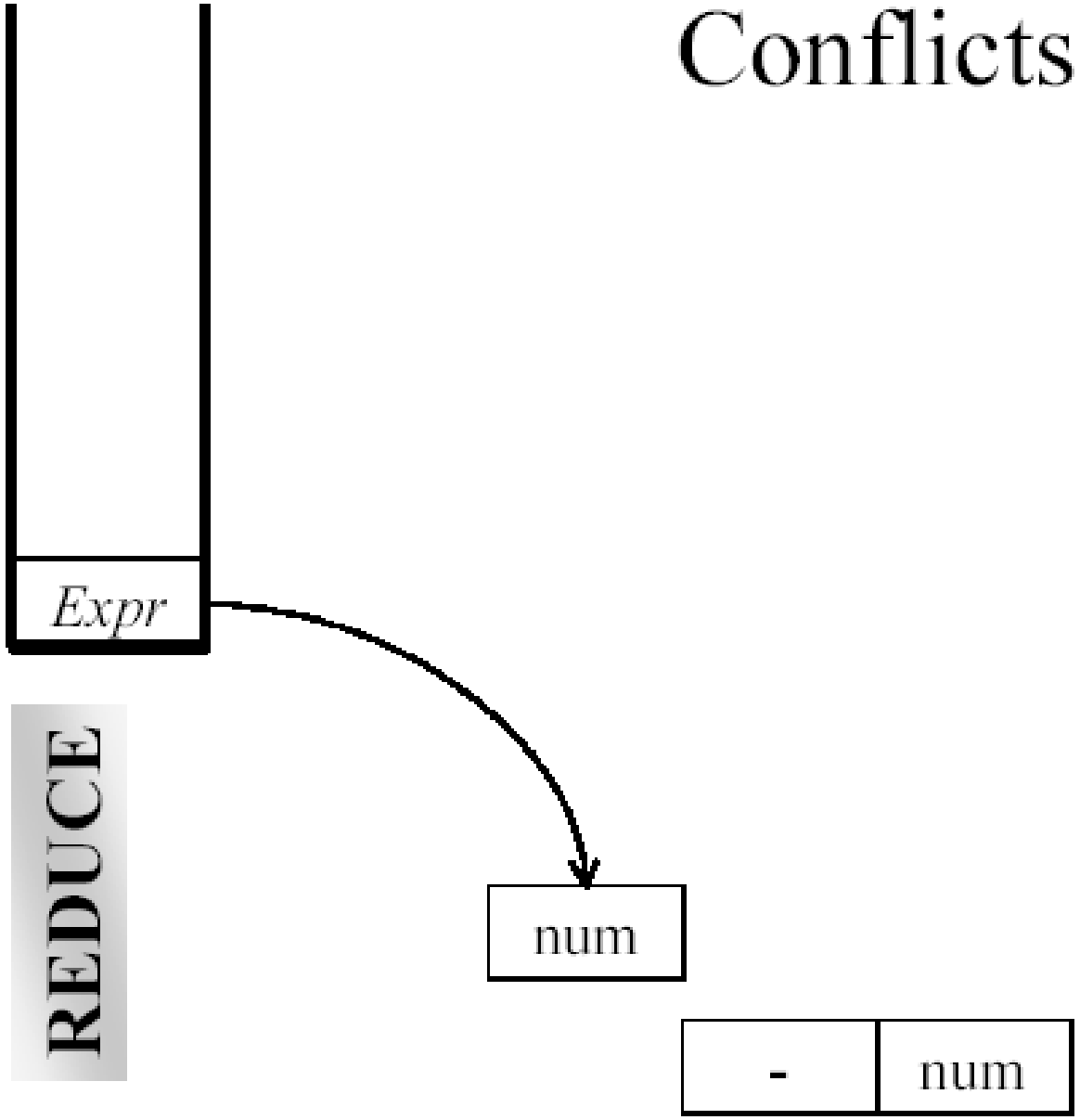
$Expr \rightarrow Expr$ -

$Expr \rightarrow$ num

$Op \rightarrow$ +

$Op \rightarrow$ -

$Op \rightarrow$ *



Conflicts

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow$ ($Expr$)

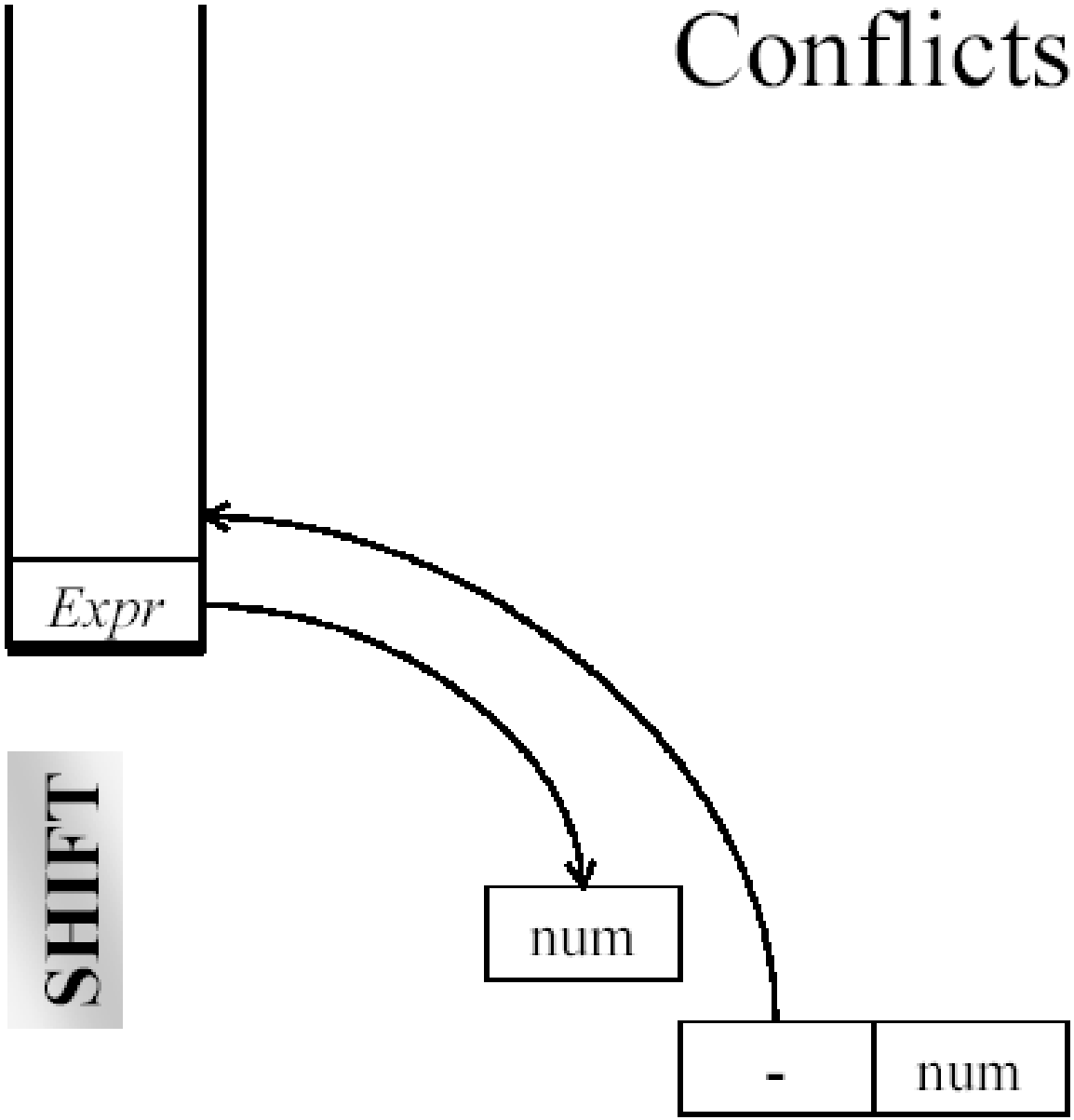
$Expr \rightarrow Expr$ -

$Expr \rightarrow$ num

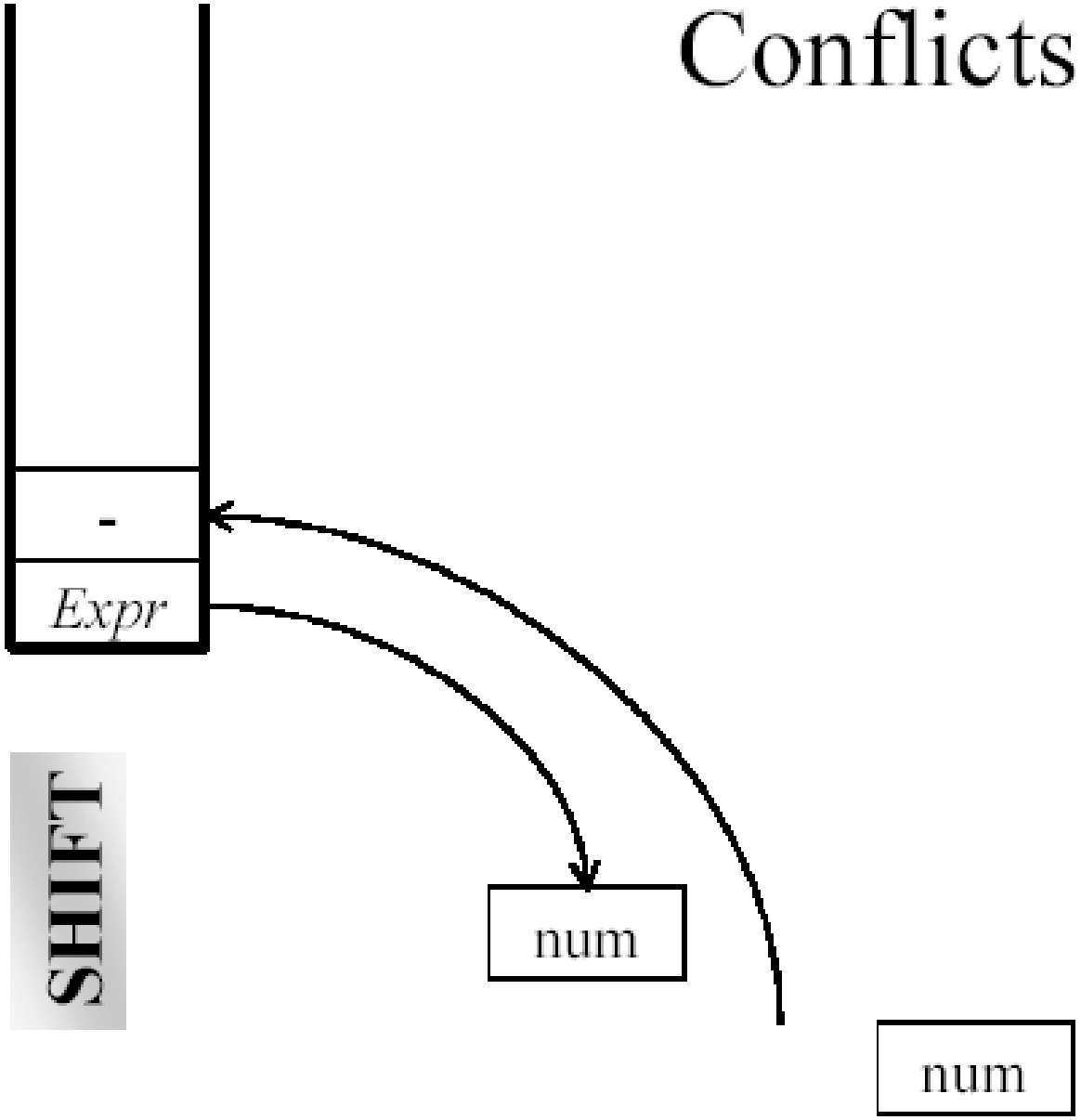
$Op \rightarrow$ +

$Op \rightarrow$ -

$Op \rightarrow$ *



Conflicts



$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow Expr\ -\ \ Expr$

$Expr \rightarrow\ (Expr)$

$Expr \rightarrow\ \ Expr\ -$

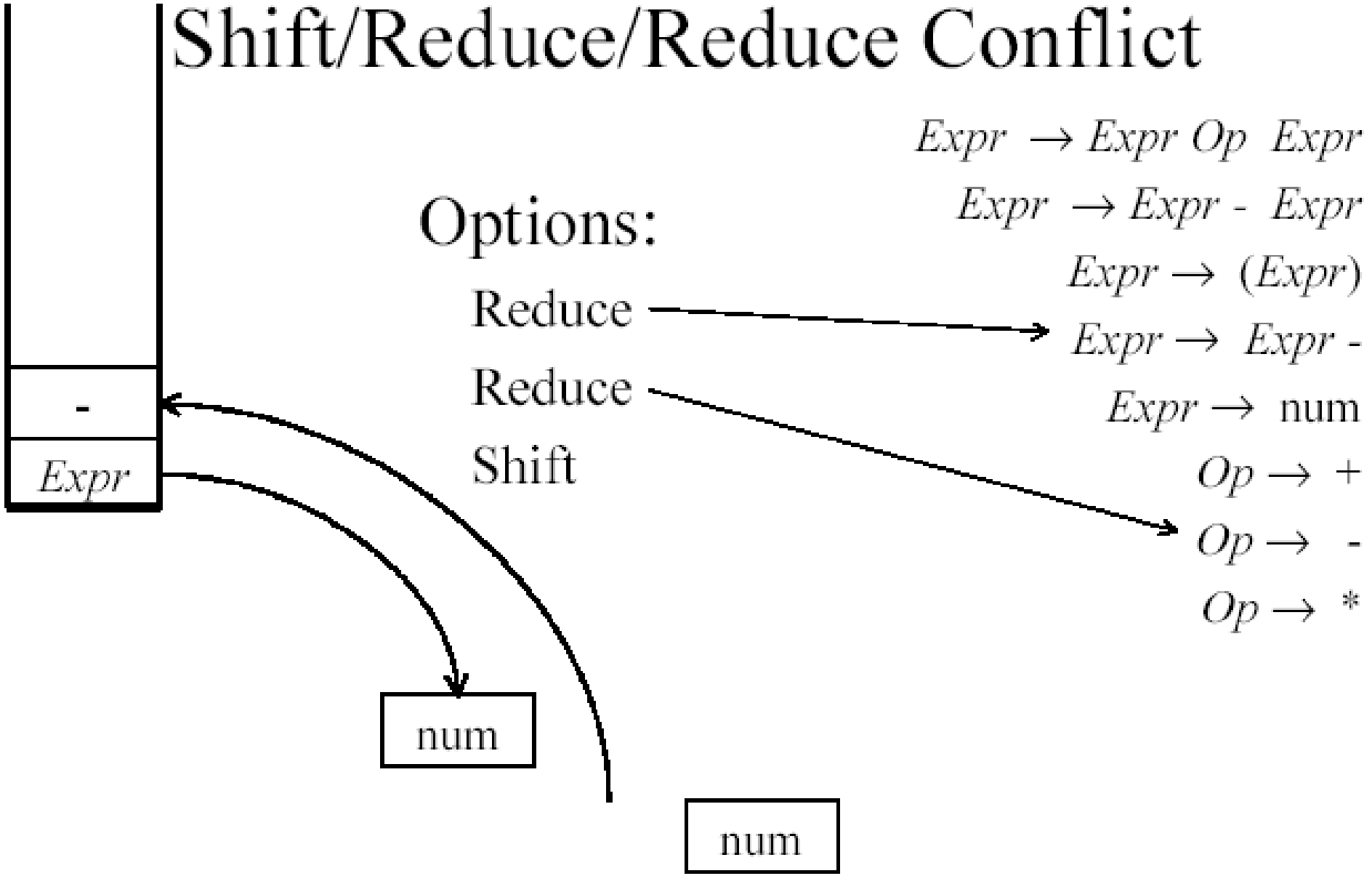
$Expr \rightarrow$ num

$Op \rightarrow\ +$

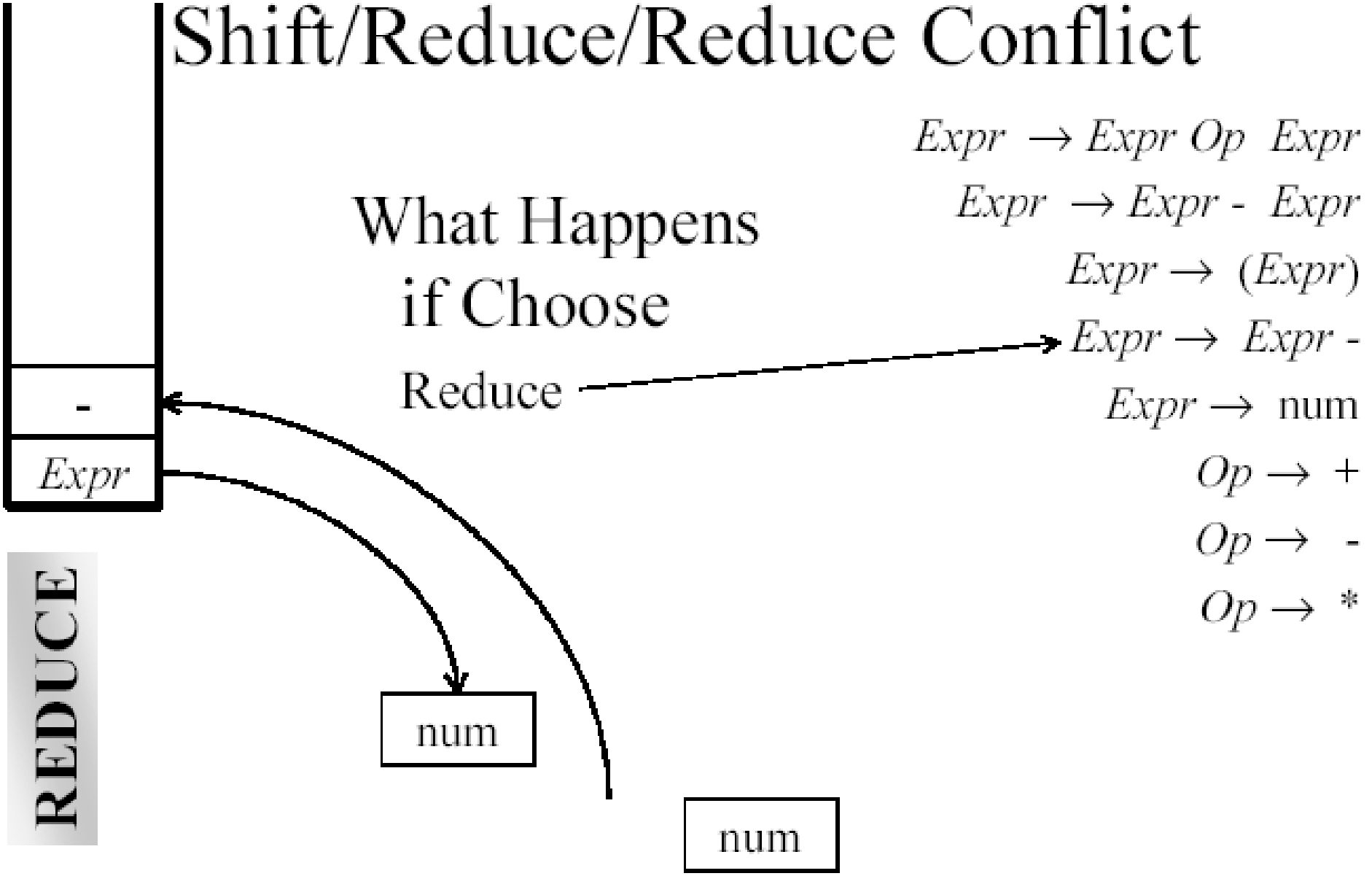
$Op \rightarrow\ \ -$

$Op \rightarrow\ *$

Shift/Reduce/Reduce Conflict



Shift/Reduce/Reduce Conflict



Shift/Reduce/Reduce Conflict

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr\ -\ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow Expr\ -$

$Expr \rightarrow num$

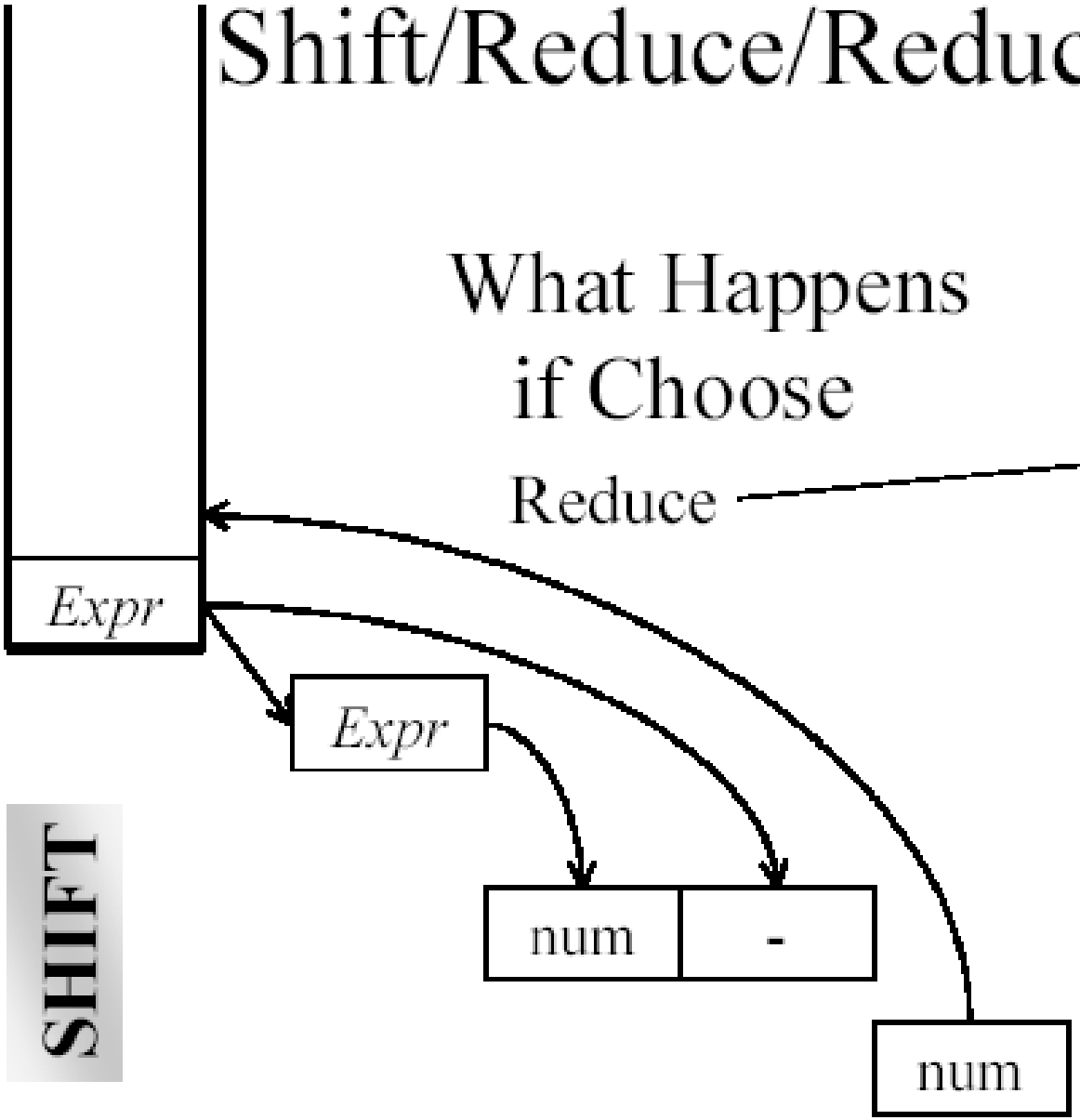
$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

What Happens if Choose

Reduce



Shift/Reduce/Reduce Conflict

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr\ -\ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow Expr\ -$

$Expr \rightarrow num$

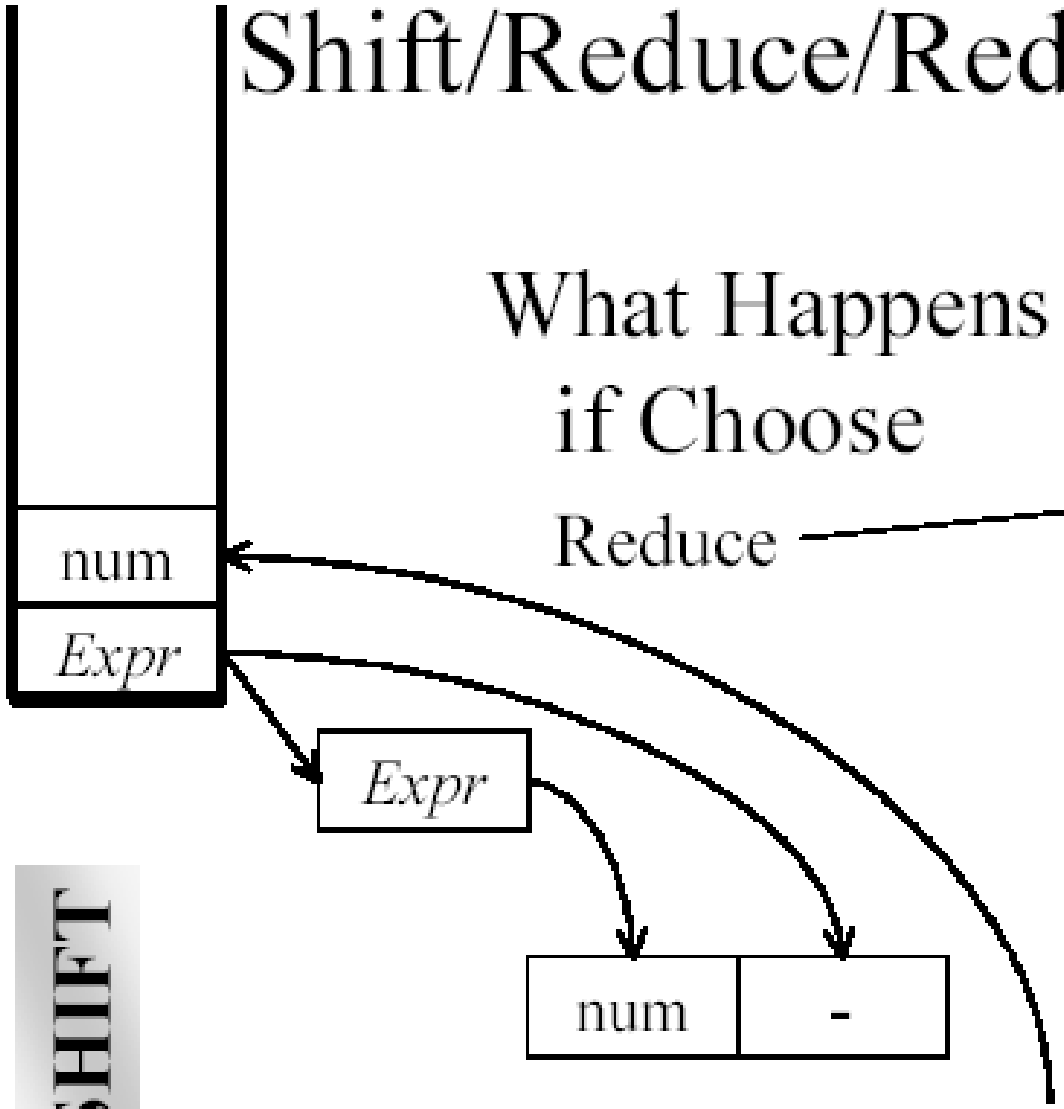
$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

What Happens if Choose

Reduce



Shift/Reduce/Reduce Conflict

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow$ ($Expr$)

$Expr \rightarrow Expr$ -

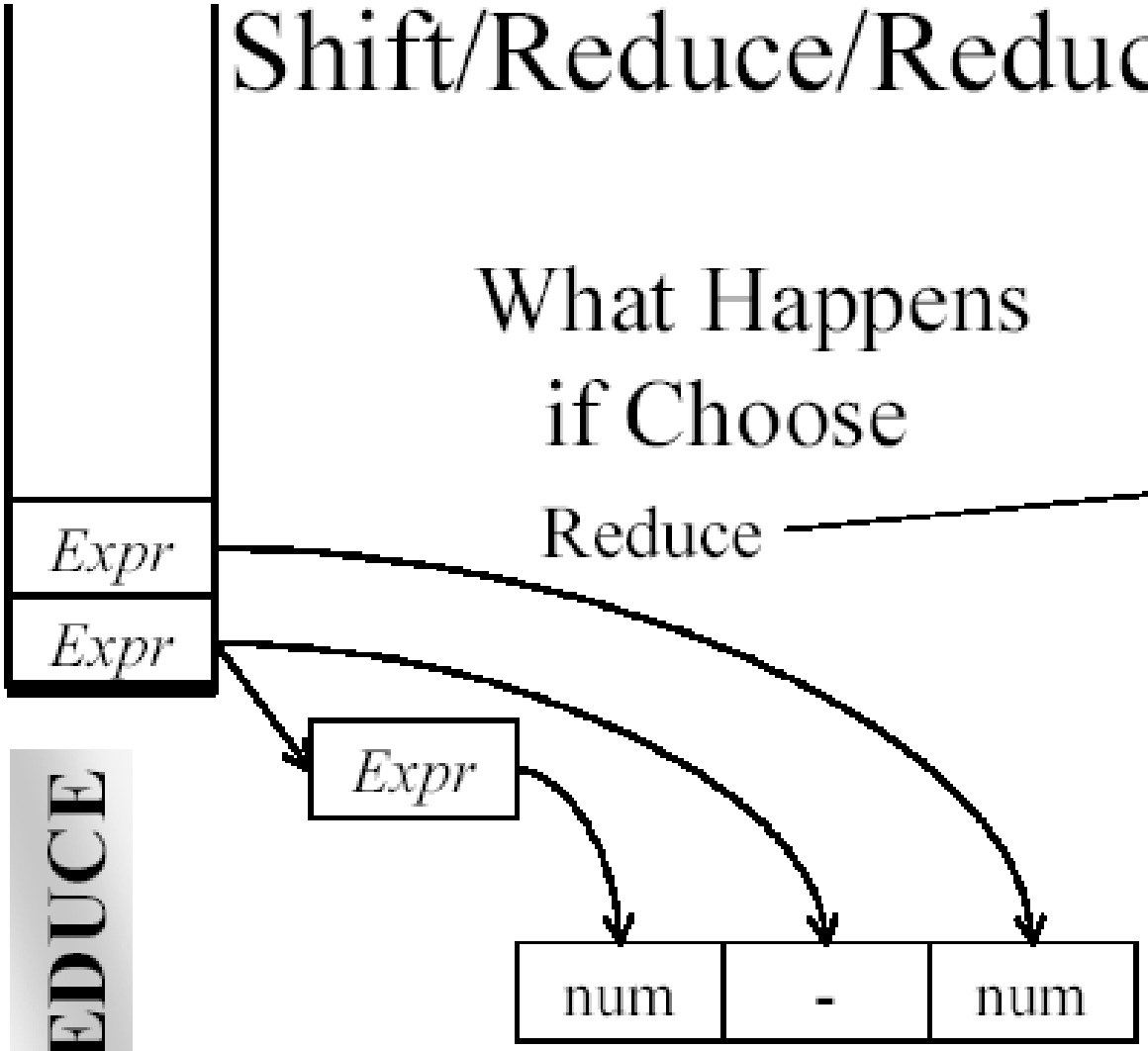
$Expr \rightarrow$ num

$Op \rightarrow$ +

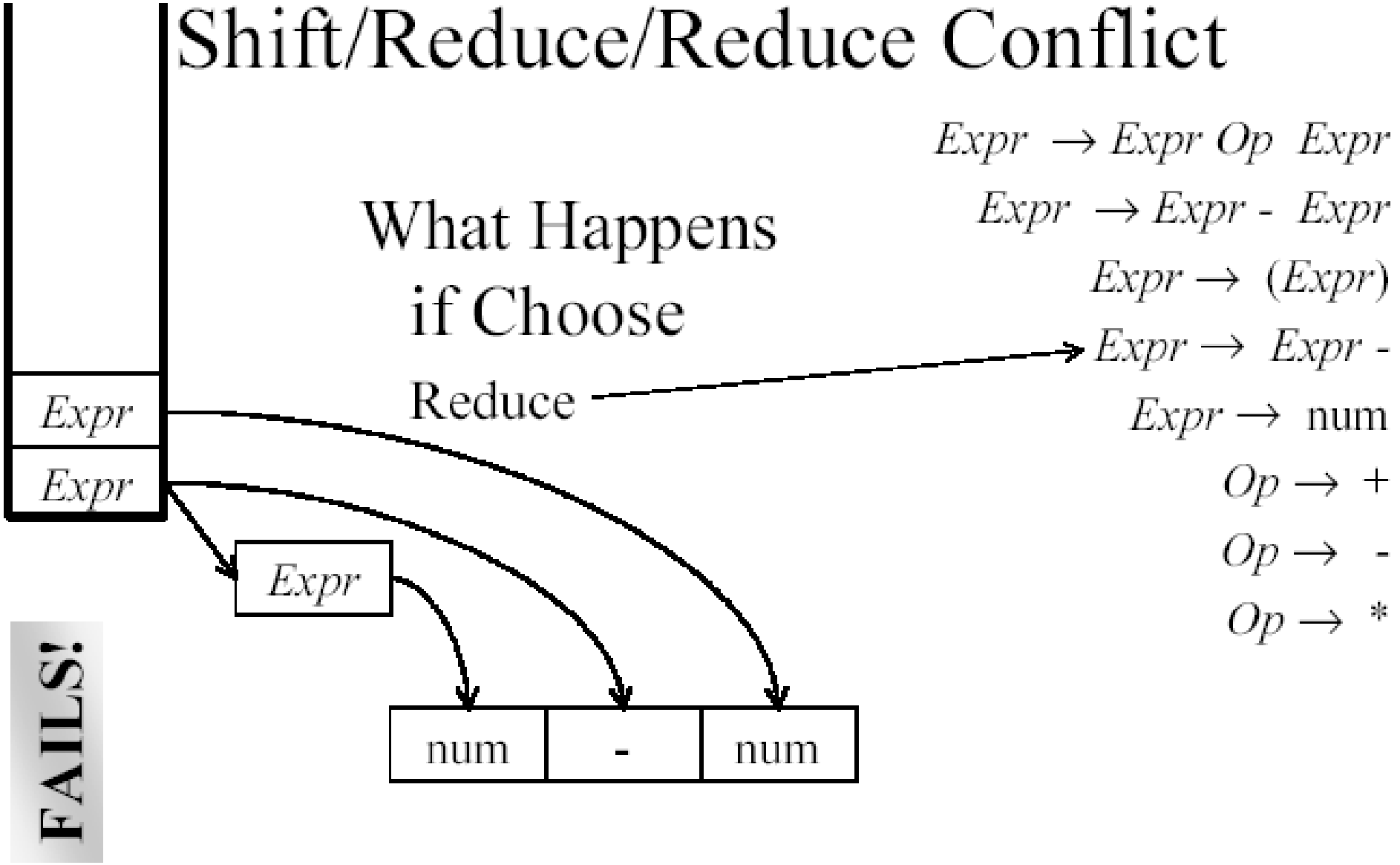
$Op \rightarrow$ -

$Op \rightarrow$ *

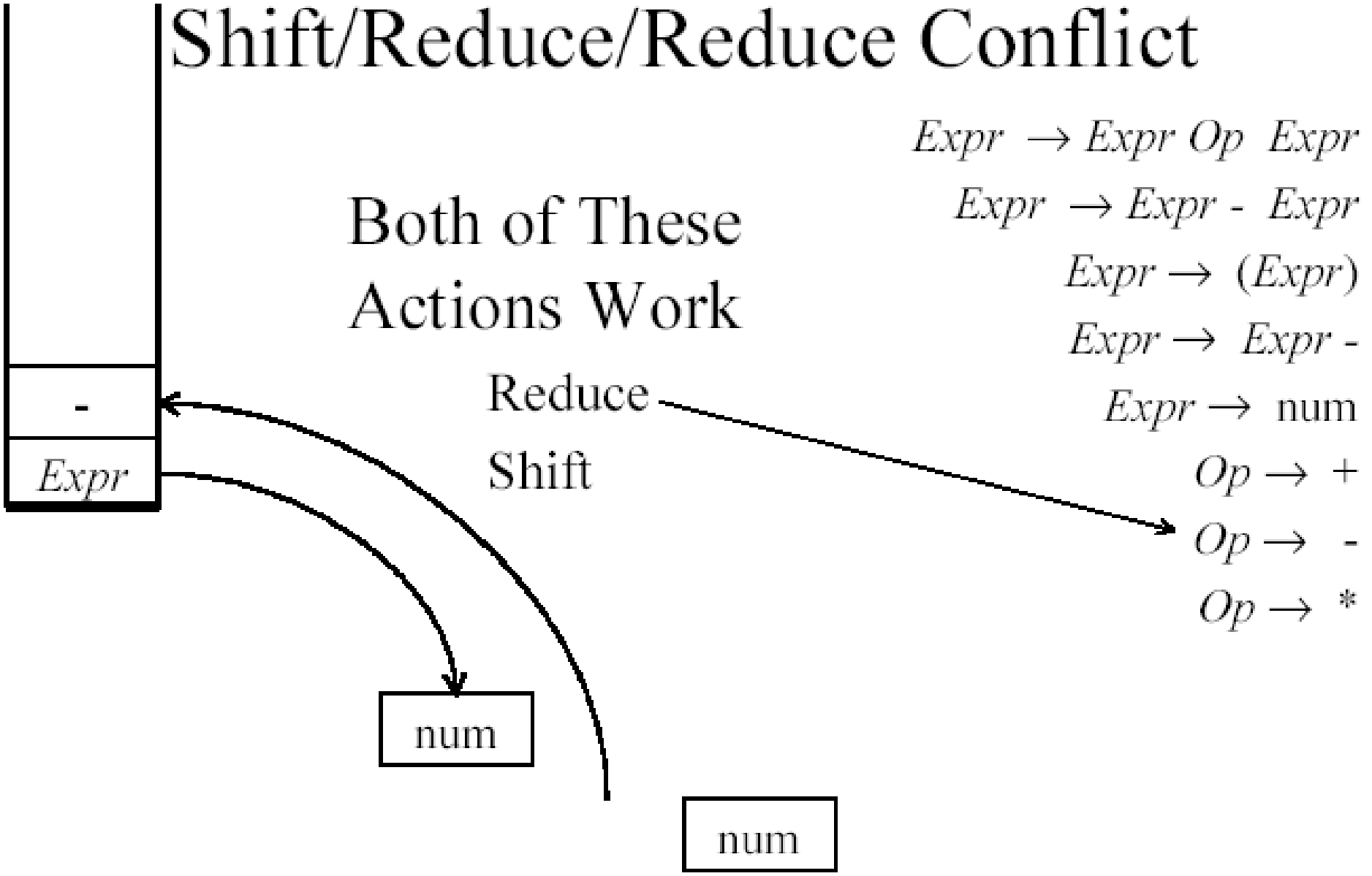
What Happens if Choose Reduce



Shift/Reduce/Reduce Conflict



Shift/Reduce/Reduce Conflict



Shift/Reduce/Reduce Conflict

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow$ ($Expr$)

$Expr \rightarrow Expr$ -

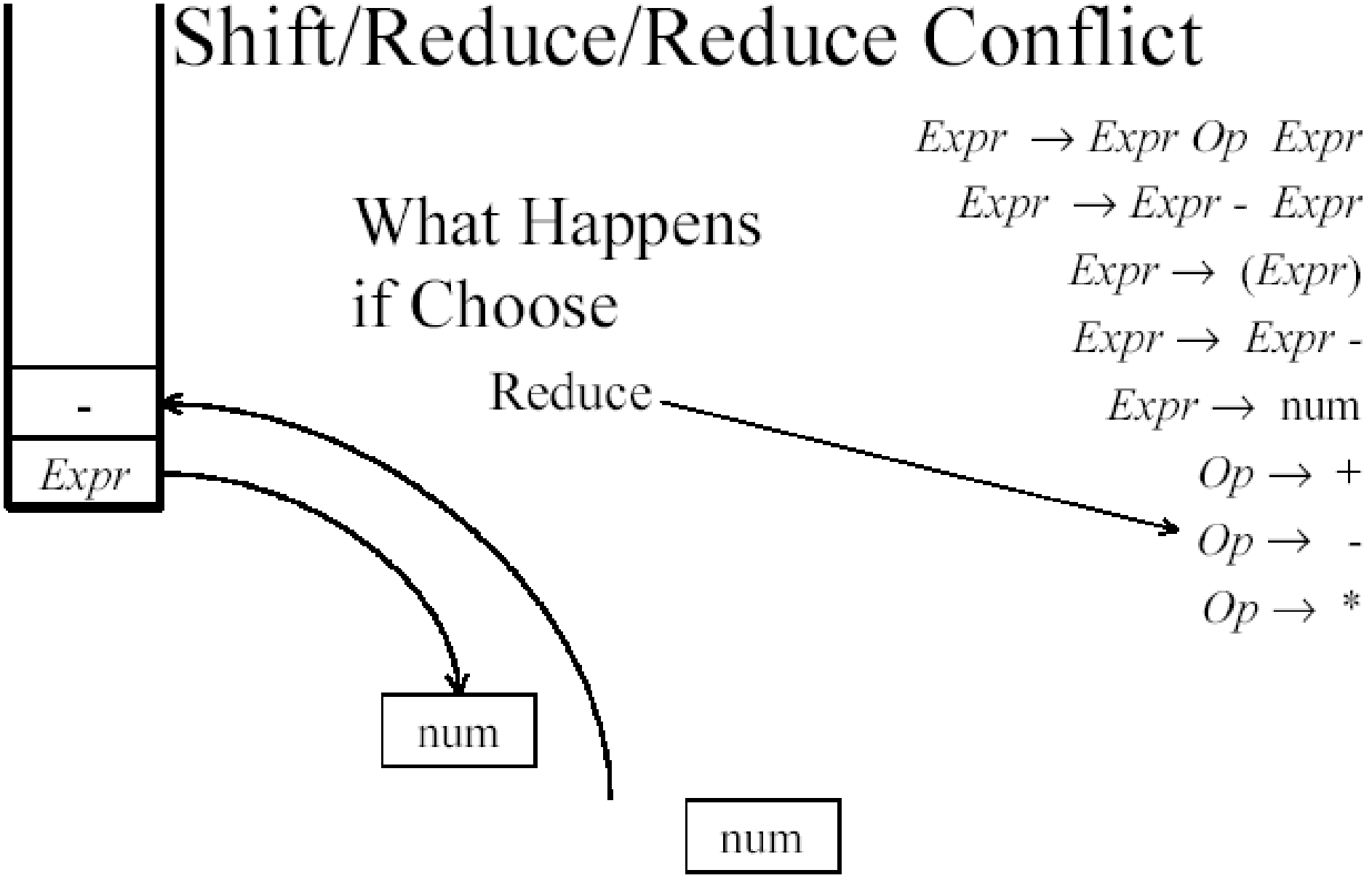
$Expr \rightarrow$ num

$Op \rightarrow$ +

$Op \rightarrow$ -

$Op \rightarrow$ *

What Happens if Choose



Shift/Reduce/Reduce Conflict

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr\ -\ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow Expr\ -$

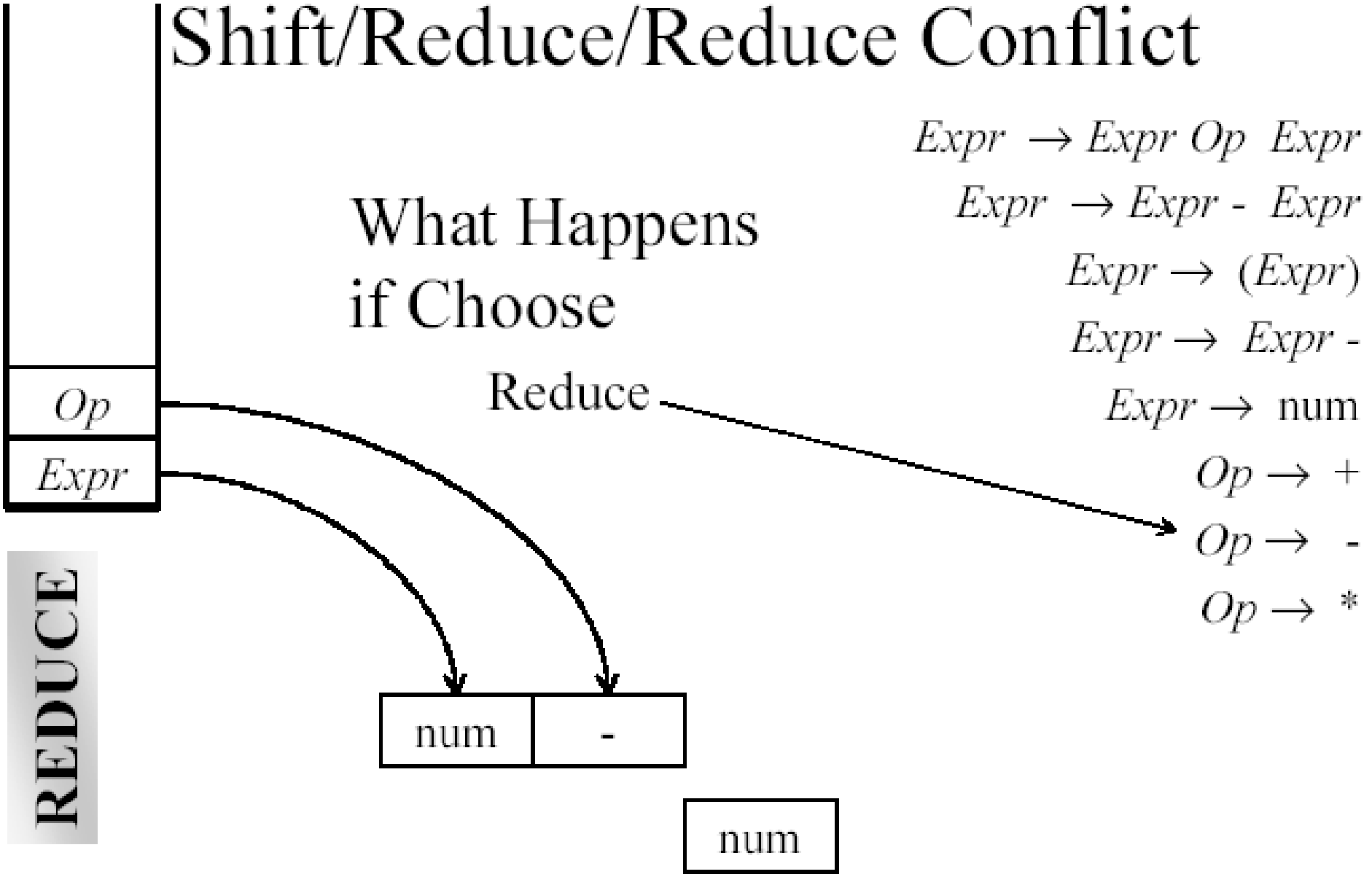
$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

What Happens if Choose



Shift/Reduce/Reduce Conflict

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow$ ($Expr$)

$Expr \rightarrow Expr$ -

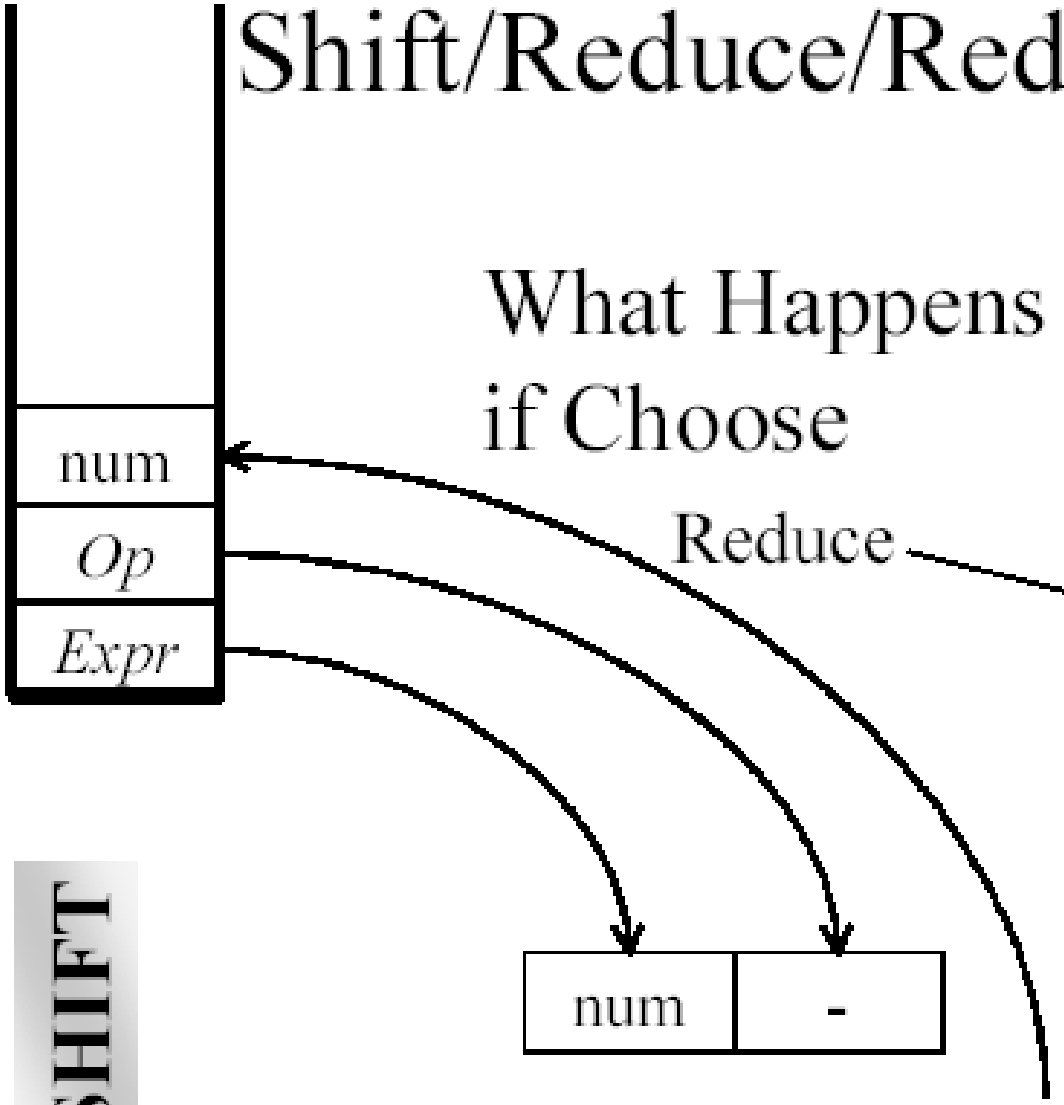
$Expr \rightarrow$ num

$Op \rightarrow$ +

$Op \rightarrow$ -

$Op \rightarrow$ *

What Happens if Choose



Shift/Reduce/Reduce Conflict

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr\ -\ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow Expr\ -$

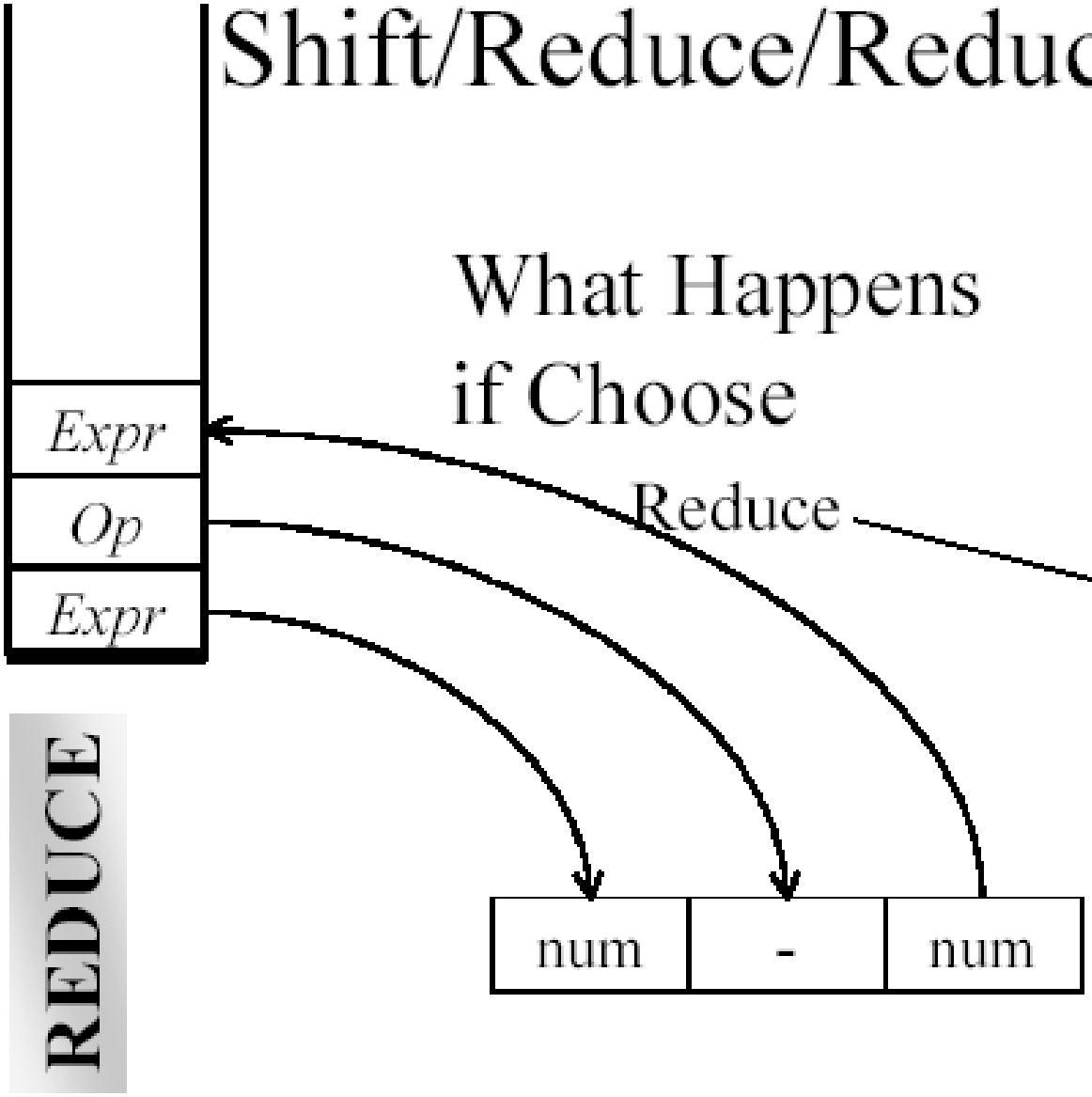
$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

What Happens if Choose Reduce



REDUCE

Shift/Reduce/Reduce Conflict

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr\ -\ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow Expr\ -$

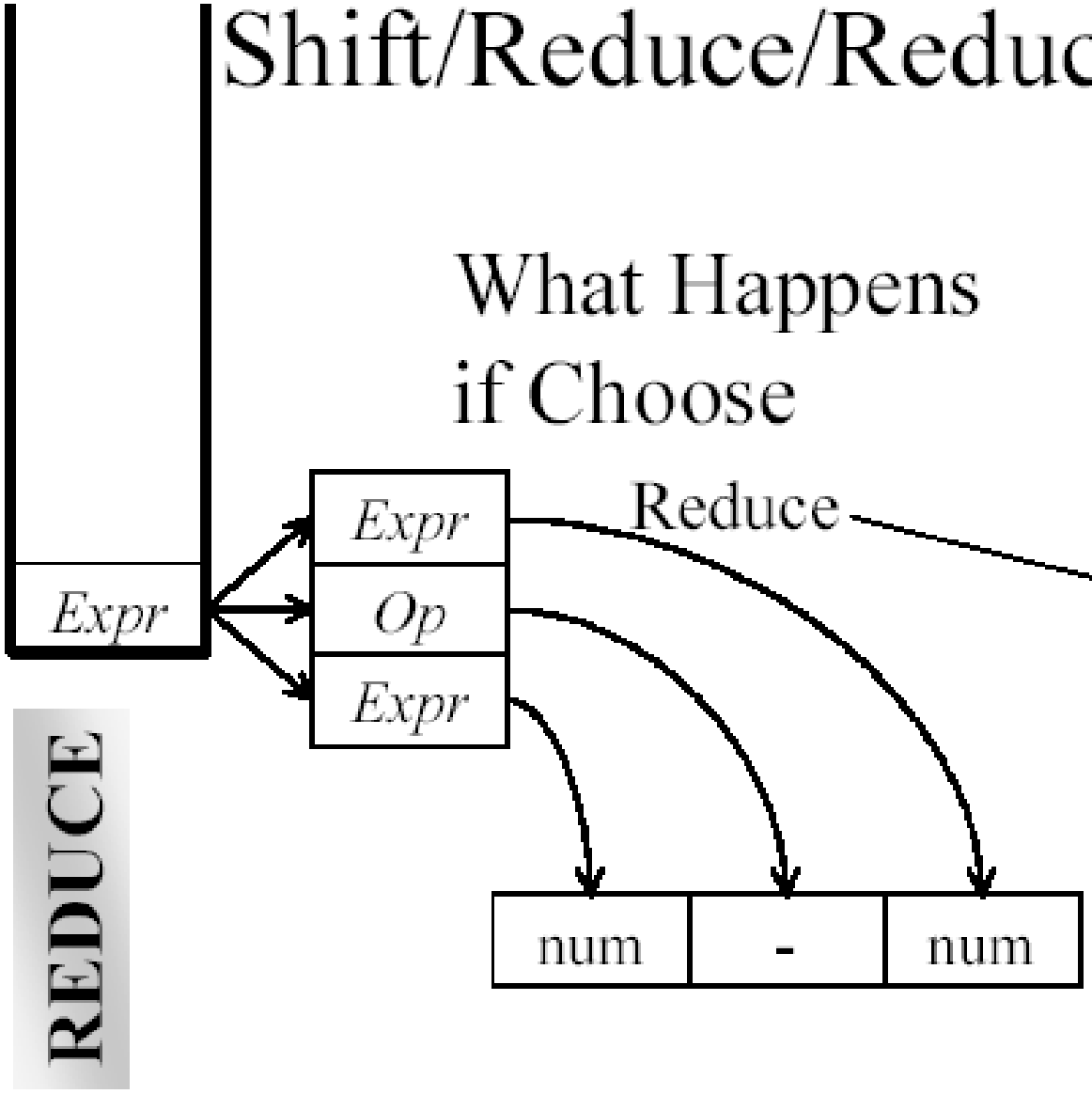
$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

What Happens if Choose



Shift/Reduce/Reduce Conflict

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr\ -\ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow Expr\ -$

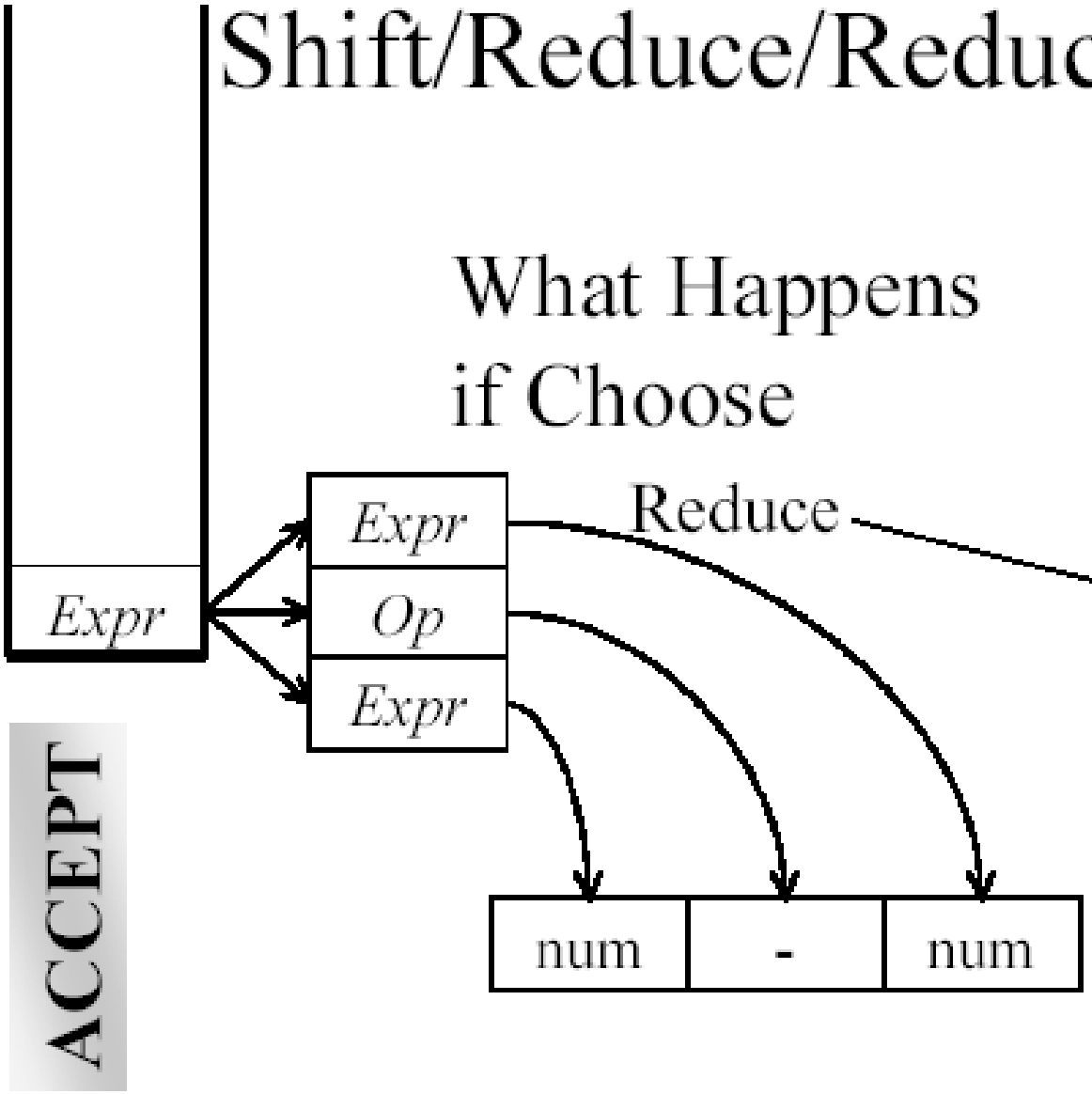
$Expr \rightarrow$ num

$Op \rightarrow +$

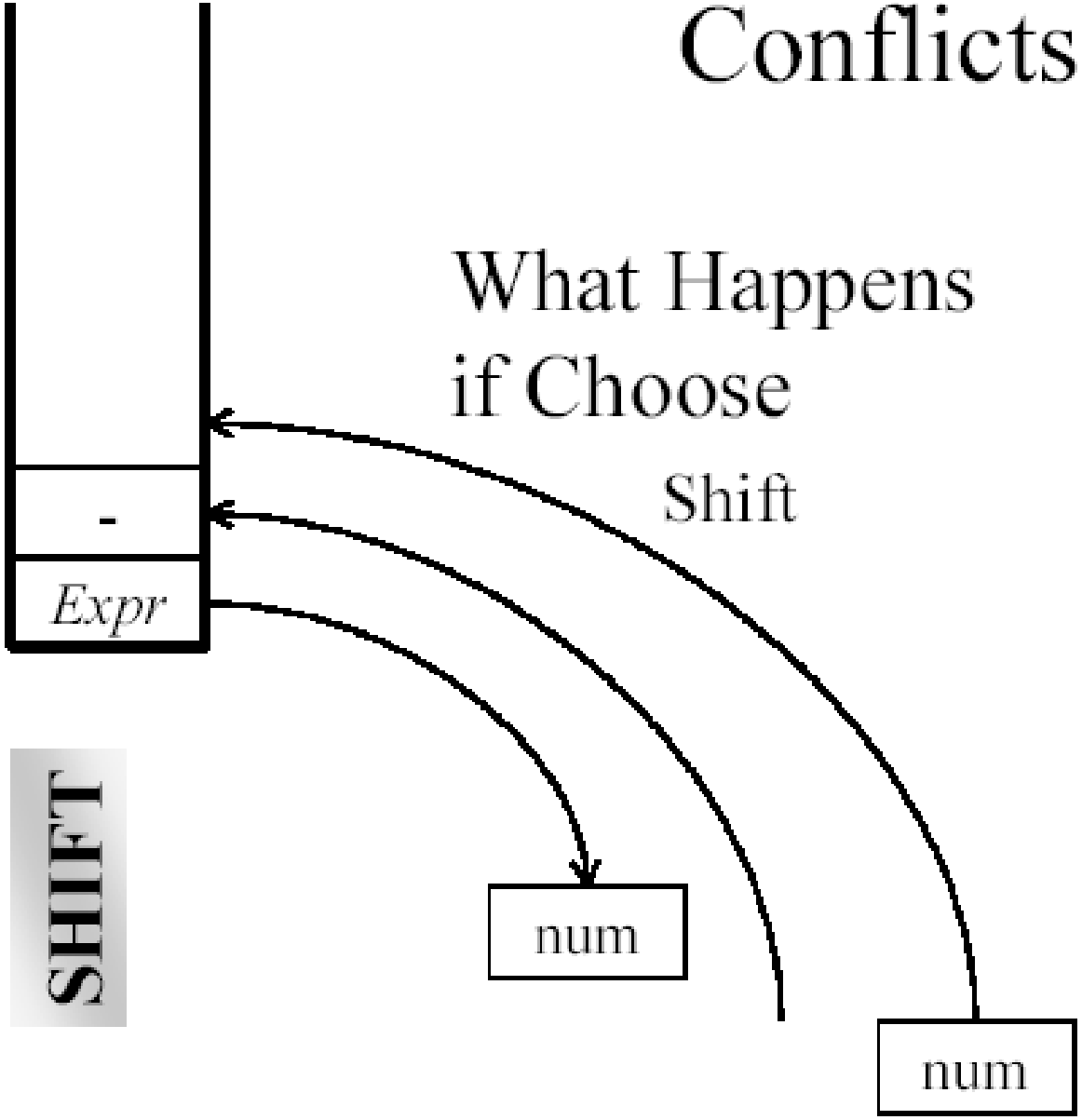
$Op \rightarrow -$

$Op \rightarrow *$

What Happens if Choose Reduce



Conflicts



$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow Expr\ -\ \ Expr$

$Expr \rightarrow\ (Expr)$

$Expr \rightarrow\ \ Expr\ -$

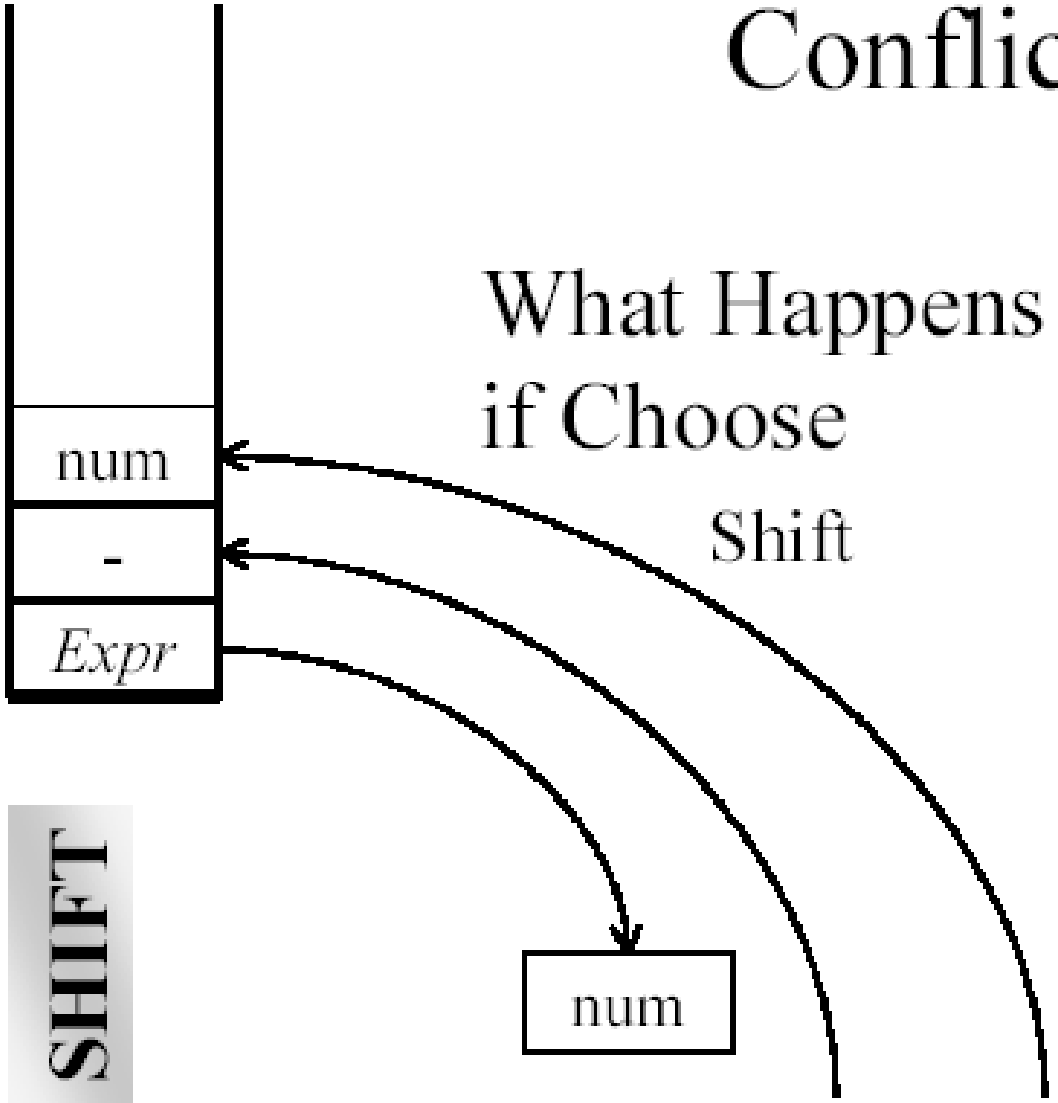
$Expr \rightarrow$ num

$Op \rightarrow\ +$

$Op \rightarrow\ \ -$

$Op \rightarrow\ *$

Conflicts



$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow$ ($Expr$)

$Expr \rightarrow Expr$ -

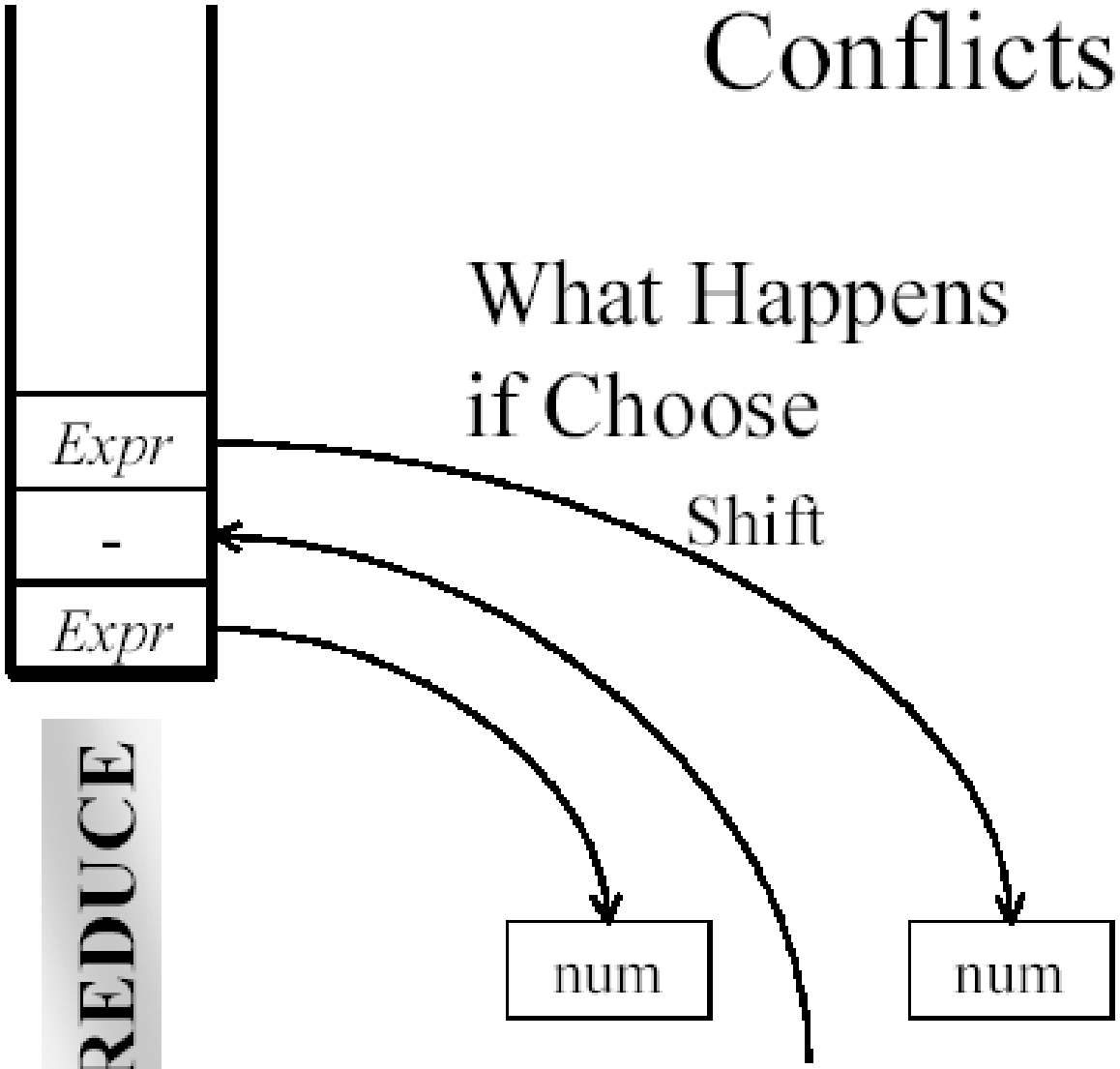
$Expr \rightarrow$ num

$Op \rightarrow$ +

$Op \rightarrow$ -

$Op \rightarrow$ *

Conflicts



$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow Expr$ -

$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow$ -

$Op \rightarrow *$

Conflicts

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow$ ($Expr$)

$Expr \rightarrow Expr$ -

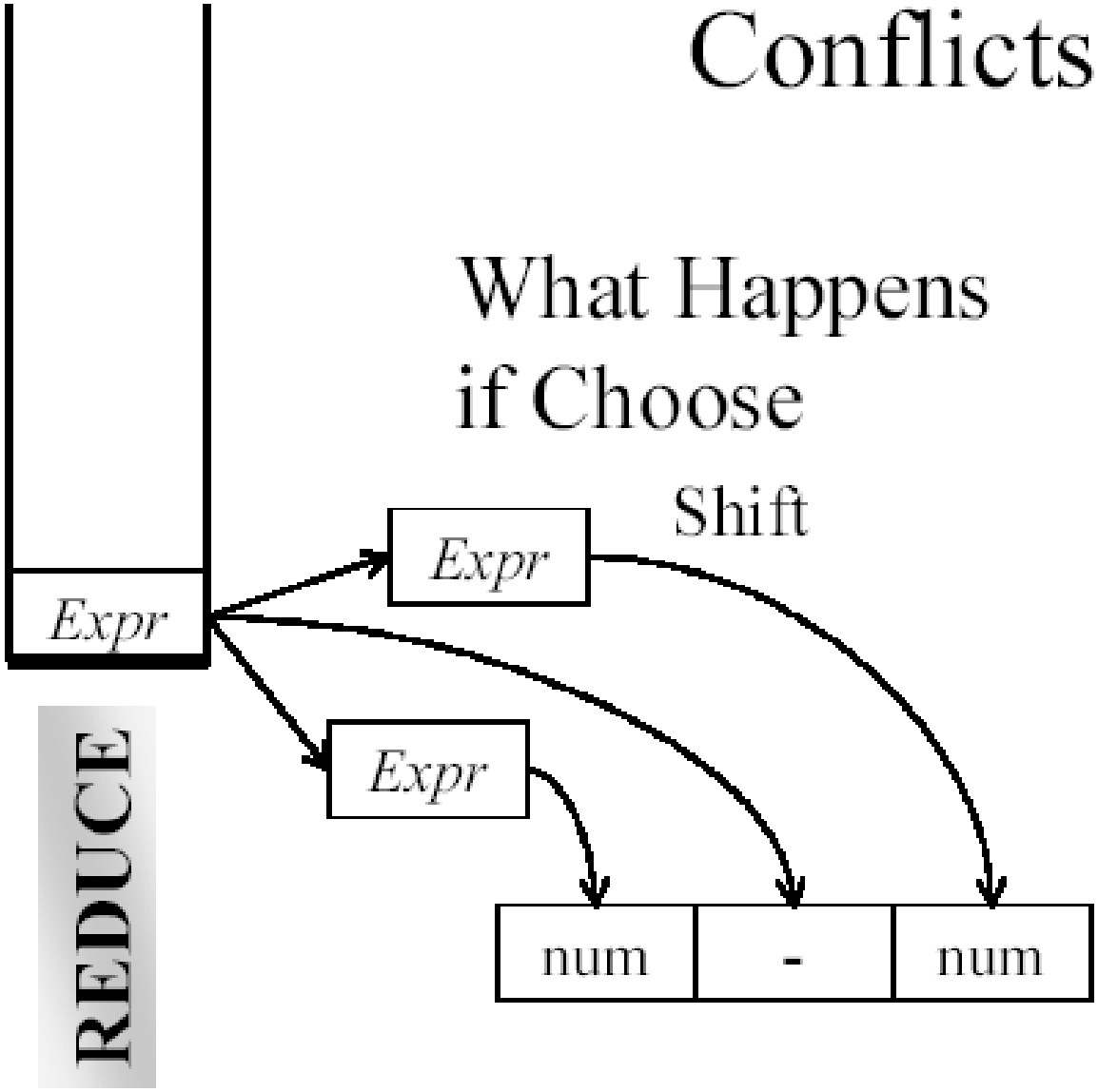
$Expr \rightarrow$ num

$Op \rightarrow$ +

$Op \rightarrow$ -

$Op \rightarrow$ *

What Happens if Choose



Conflicts

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow$ ($Expr$)

$Expr \rightarrow Expr$ -

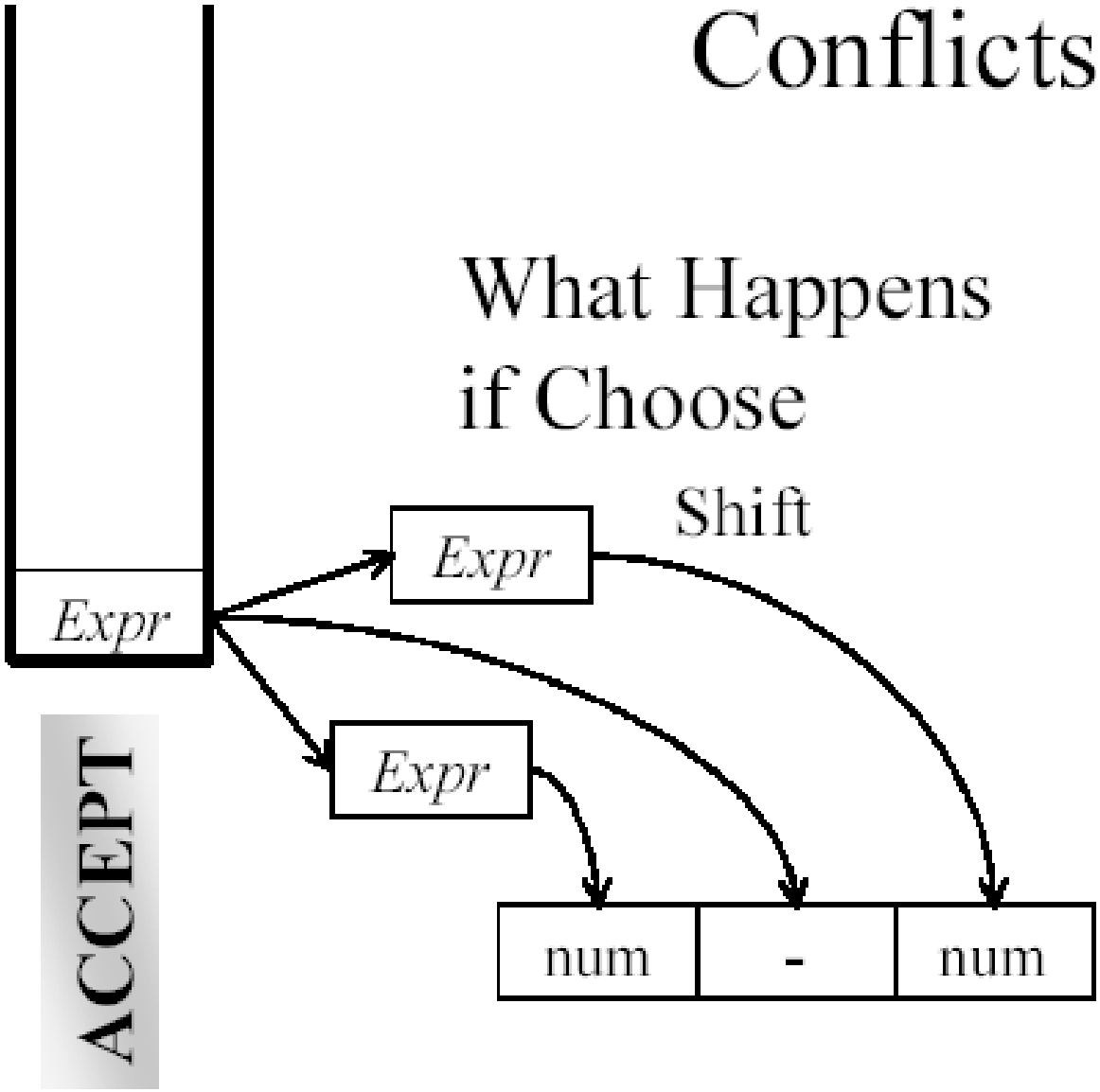
$Expr \rightarrow$ num

$Op \rightarrow$ +

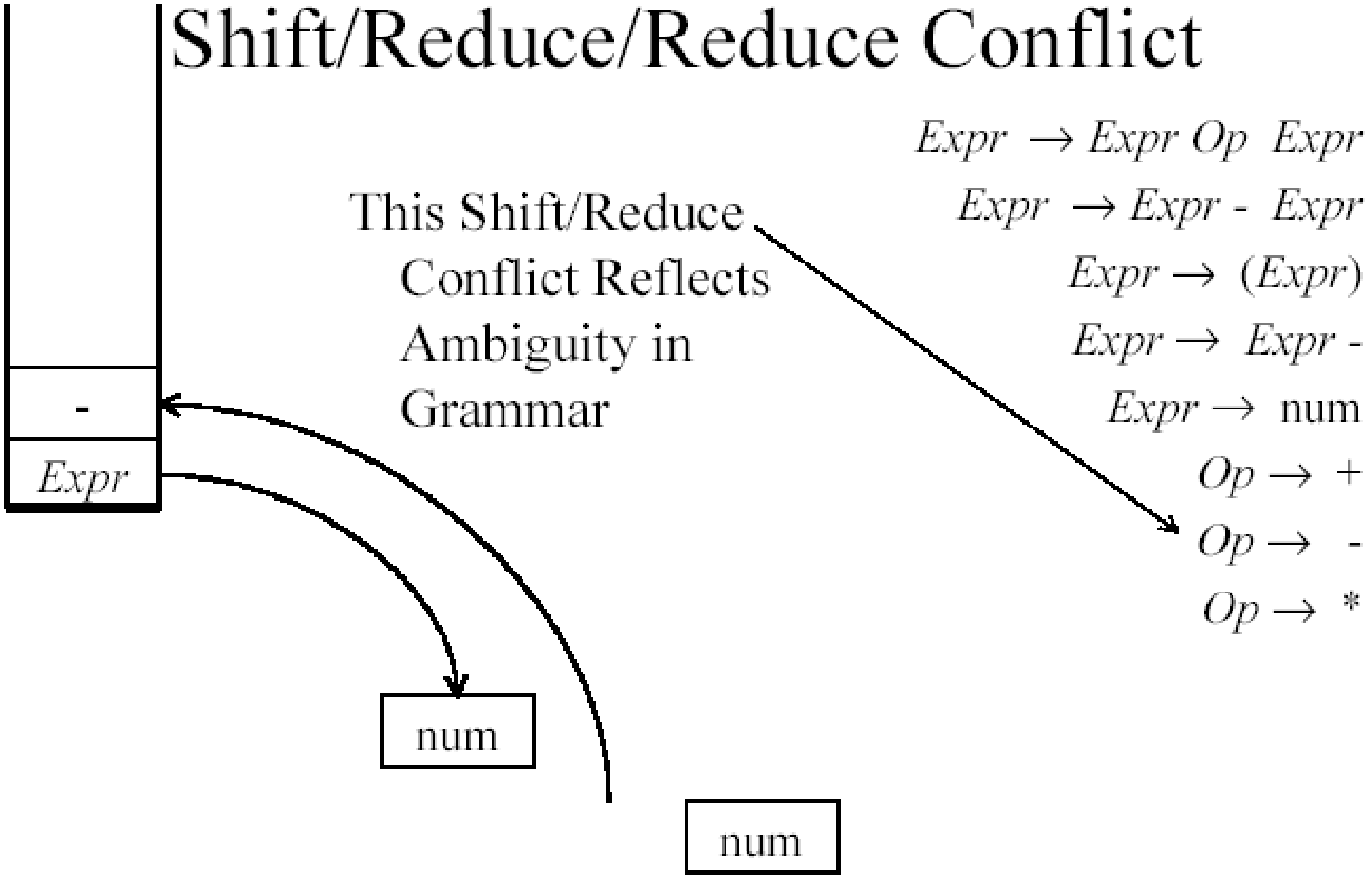
$Op \rightarrow$ -

$Op \rightarrow$ *

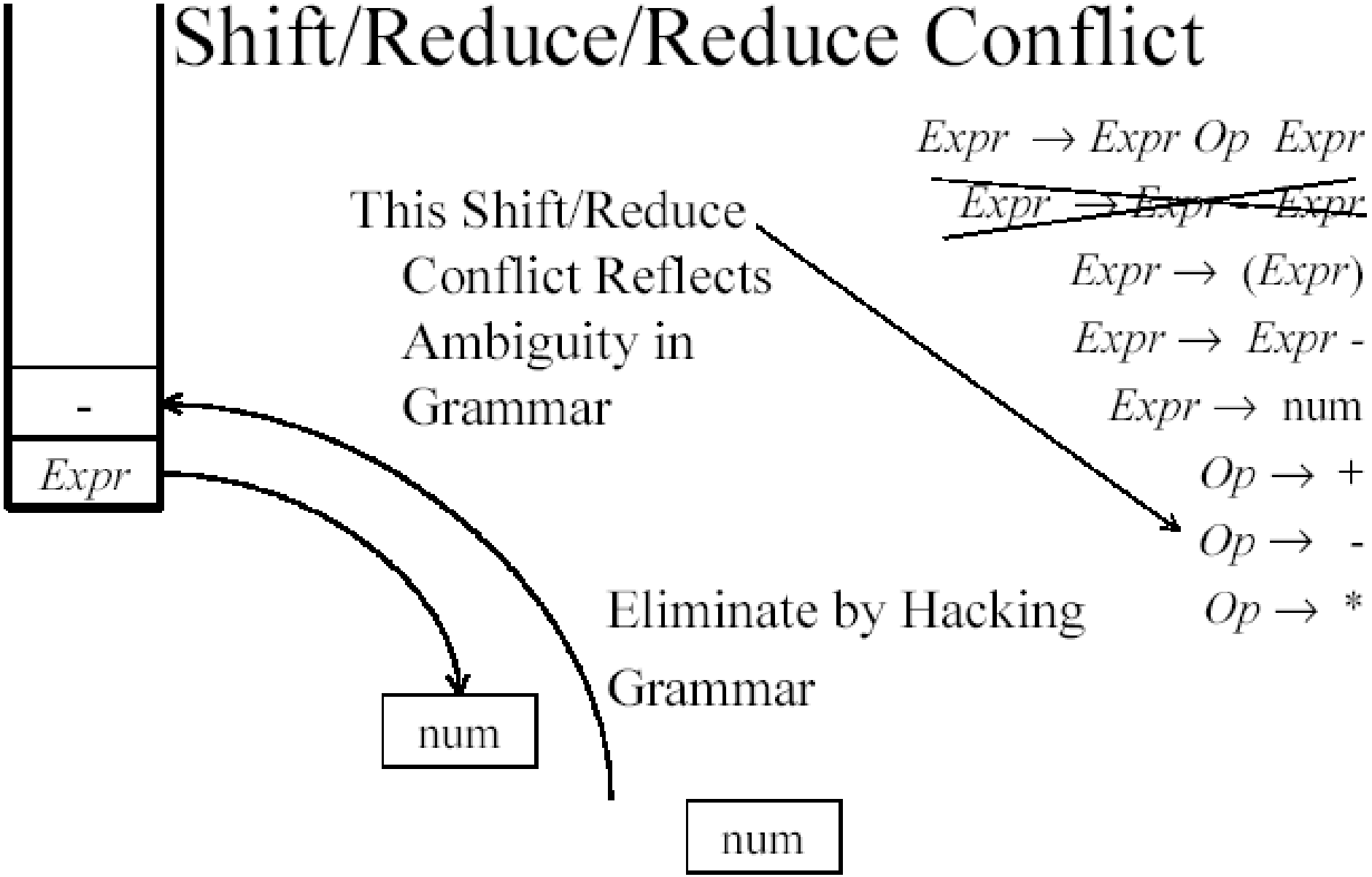
What Happens if Choose Shift



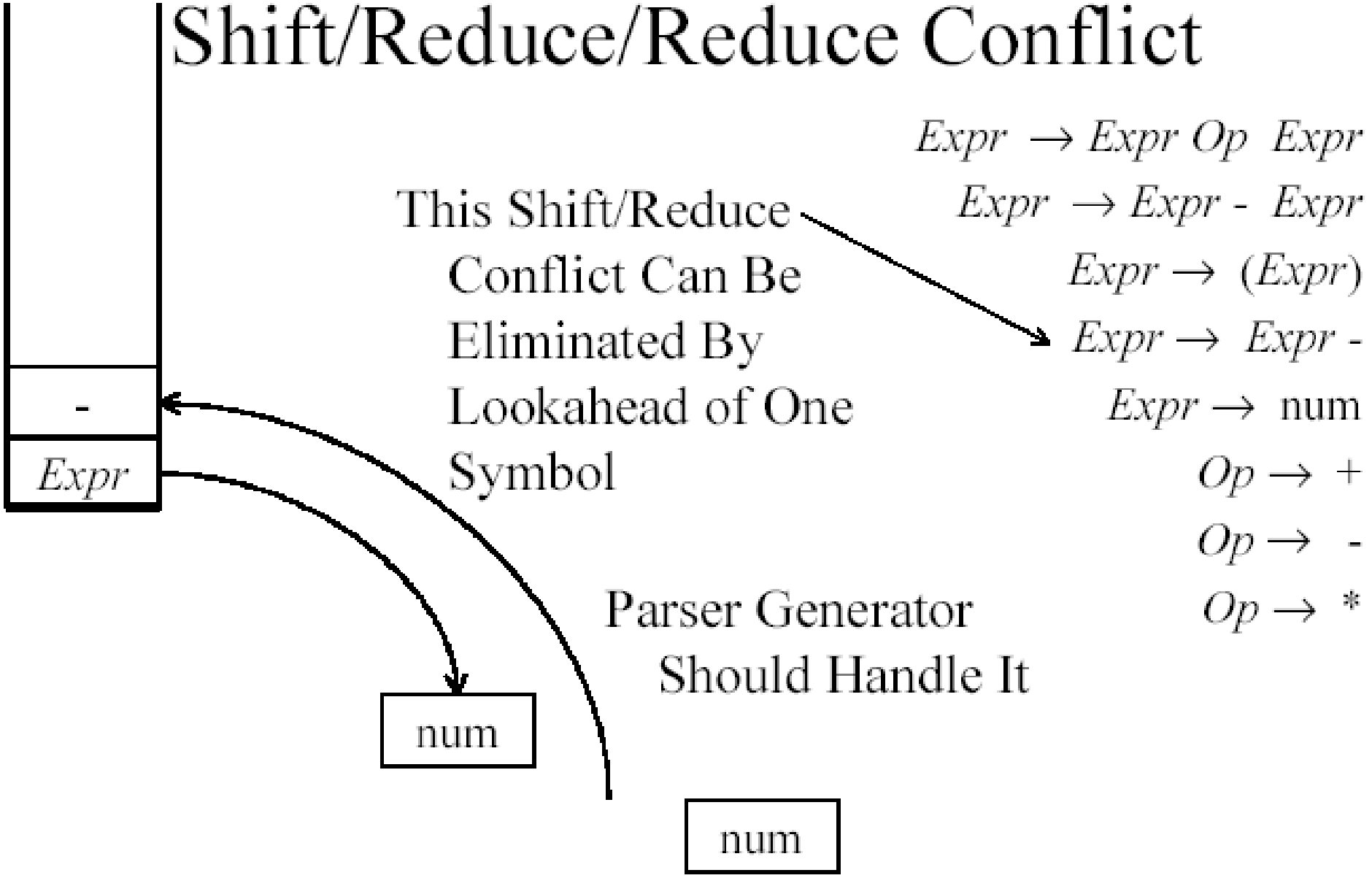
Shift/Reduce/Reduce Conflict



Shift/Reduce/Reduce Conflict



Shift/Reduce/Reduce Conflict



观 察

- “归约-归约”冲突：当前栈顶部分与多个产生式右部匹配：
  - 一个可归约串匹配多个变元的候选式，或者，
  - 多个可归约串匹配多个候选式。
- “移进-归约”冲突：当可归约串出现时可以归约也可以继续移进。

- 造成冲突的原因是什么呢？
  - 能在文法上找原因。
  - 文法有歧义性才造成此类冲突？
  - 与当前输入符号有关？
  - 与当前输入符号及其随后的k-1个符号有关？
  - 通过消除文法歧义性、向前查看k个符号，几乎避免了冲突的发生？

构建确定化最左归约的思路

- 最右推导的逆过程是确定化的最左归约，被称为规范归约。
- 规范归约与最左归约的异同之处？

$E \to E\ O\ E$

$E \to (E)$

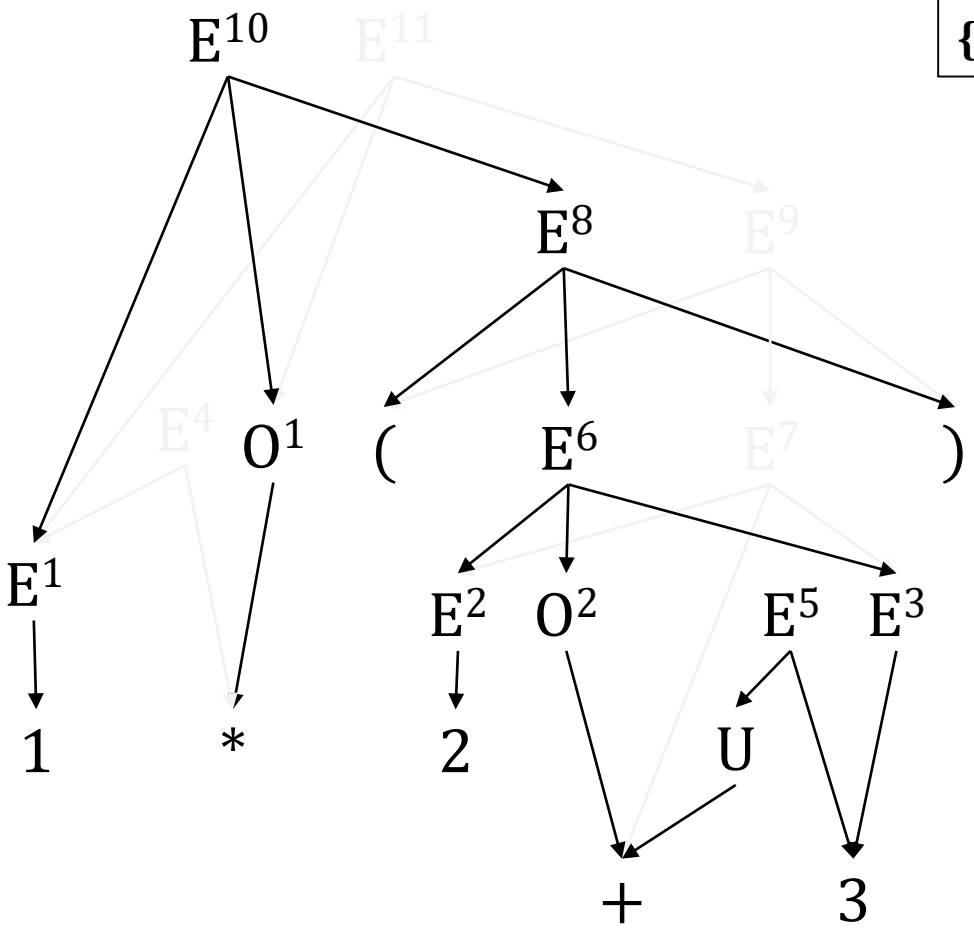
$E \to i$

$O \to +$

$O \to *$

$U \to +$

$E \to U\ i$



$\{E^1, O^1, E^2, E^5\}$
$\{E^{10}\}$

8.2.3 规范归约

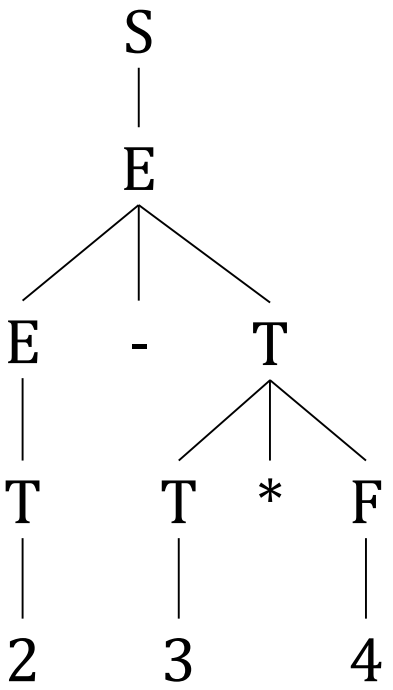
- 首先引入几个术语：对于CFG (V, T, P, S)，
  - 短语：对于句型αAβ若有$A \Rightarrow^{+} \delta$则称δ 是句型αδβ的短语，是相对于变元A的短语。
  - 直接短语：对于句型αAβ若有$A \Rightarrow \gamma$ 则称γ是句型αγβ的直接短语，是相对于A→γ的直接短语。
  - 句柄：句型中的最左直接短语被称为该句型的句柄。
  - 每个句型都有唯一一个句柄。

- 若有$S \Rightarrow^{*} \beta \Rightarrow^{*} w$那么β被称为句型。若这样的w不存在β 还是不是句型？在规范归约过程不存在该问题。

例：短语、直接短语、句柄

句子：2-3*4

S→E
E→E+T|E–T|T
T→T*i|T/i|i



2是句子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是句子2-3*4相对于T的短语，是相对于T→i的直接短语

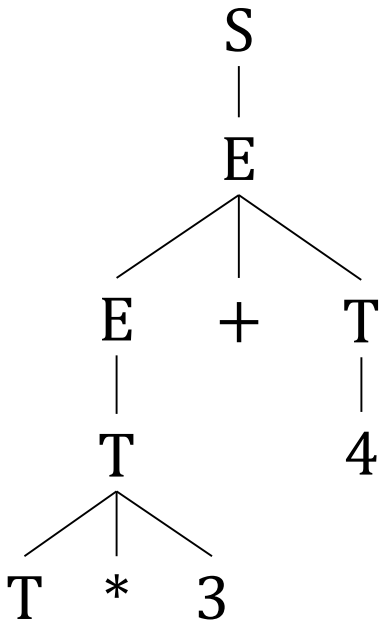
3*4是句子2-3*4相对于T的短语，不是直接短语

2-3*4是句子2-3*4相对于E和S的短语，不是直接短语

例：短语、直接短语、句柄

句型：T*3+4

S→E
E→E+T|E–T|T
T→T*i|T/i|i



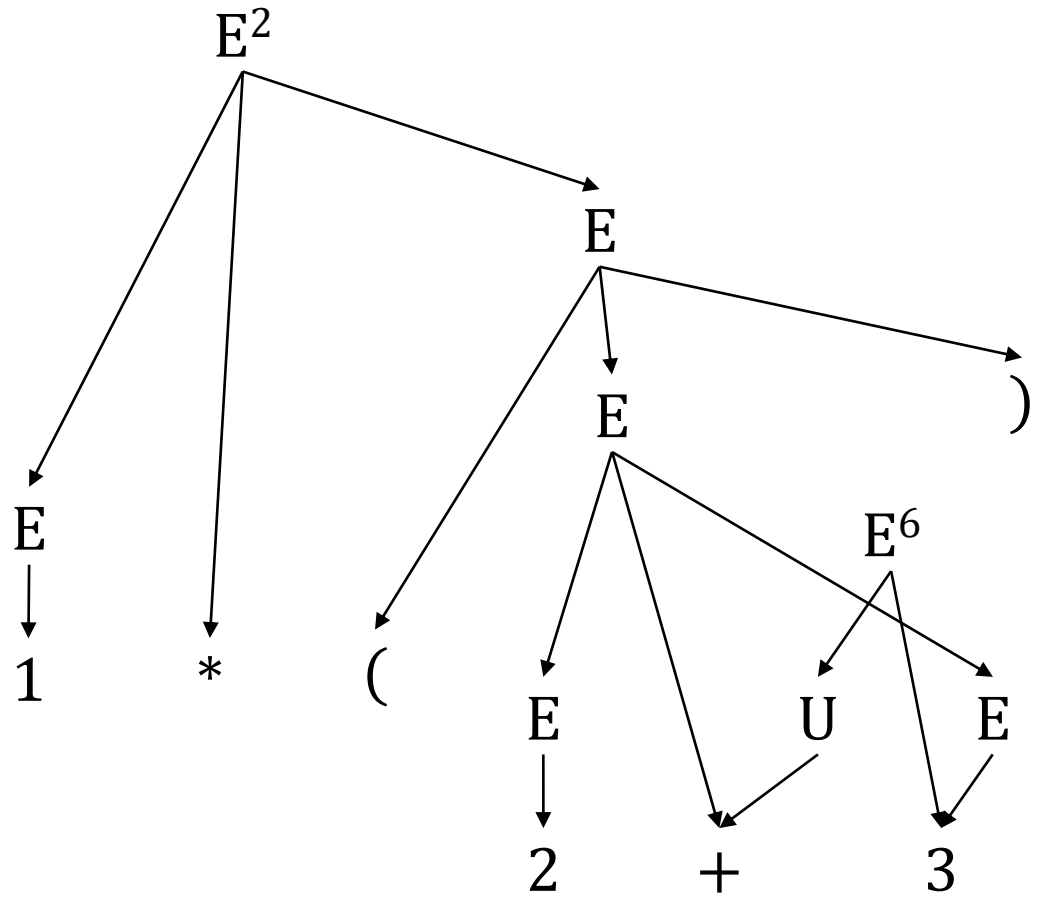
T*3是句型T*3+4相对于T和E的短语，是相对于T→T*i的直接短语，是句柄。

4是句型T*3+4相对于T的短语，是相对于T→i的直接短语。

T*3+4是句型T*3+4相对于E和S的短语,不是直接短语

思考：规范句型中句柄之后的符号都是终结符。

最左归约语法树构建的不确定性示例



术语

- CFG (V, T, P, S)的几个术语：
- 规范归约是一种最左归约，是最右推导的逆过程，记为$\Mapsto$
- 最右推导又被称为规范推导
- 规范句型：从句子的规范推导得出的句型。

- 对于$\alpha_0 = w$， $\alpha_n = S$，称$\alpha_0,...,\alpha_n$是一个规范归约序列，$n \geq 1$，如果满足下面任意一条：
  - $\alpha_{i+1} \Rightarrow_{rm} \alpha_i$，$i=0,...,n-1$
  - α_i的句柄γ_i被替换为A_{i+1}得到α_{i+1}，$(A_{i+1},\gamma_i) \in P$，$i=0,...,n-1$

- 用规范归约算符$\Mapsto$表示为$\alpha_0 \Mapsto^* \alpha_1 \Mapsto^* ... \Mapsto^* \alpha_n$
- $\Mapsto$既指归约一步（句型有变），也指移进一步（句型没变）

规范归约（句柄剪枝）示例

归约次序与计算次序没有相关性，目的是生成代码。

$P \to \check{D}\ \check{S}$

$\check{D} \to \varepsilon \mid D\ ;\ \check{D}$

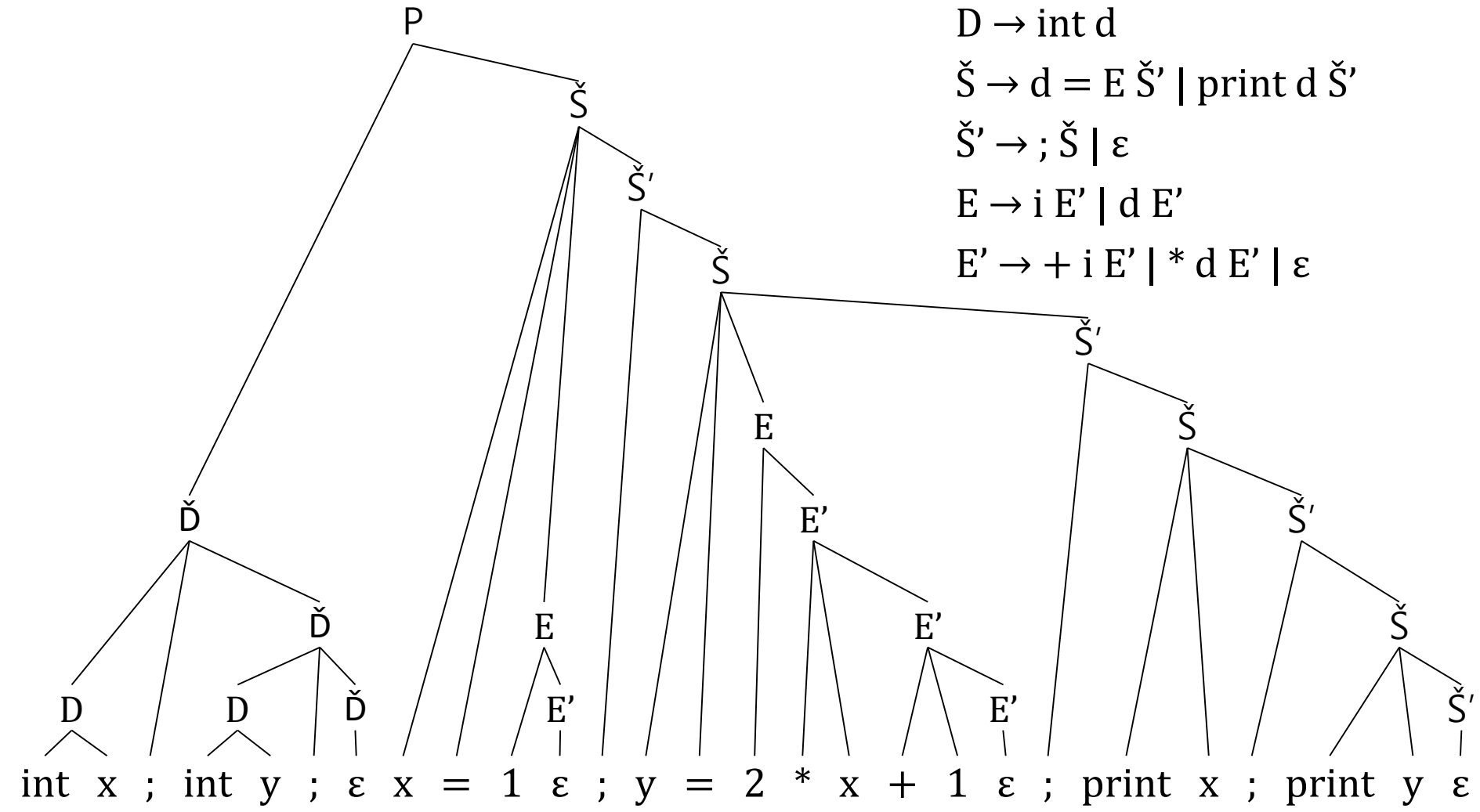
$D \to \text{int d}$

$\check{S} \to \text{d} = E\ \check{S}' \mid \text{print d}\ \check{S}'$

$\check{S}' \to ;\ \check{S} \mid \varepsilon$

$E \to \text{i}\ E' \mid \text{d}\ E'$

$E' \to + \text{i}\ E' \mid * \text{d}\ E' \mid \varepsilon$



对规范归约过程的观察

$Z_0w\# \Rightarrow^* Z_0\alpha\gamma\cdot y\# \Rightarrow Z_0\alpha A\cdot y\# \Rightarrow^* Z_0\beta\cdot az\# \Rightarrow Z_0\beta a\cdot z\# \Rightarrow^* Z_0S\cdot\#$

其中：γ为A的候选式；β的所有后缀都不是可归约串。

•int x; int y; x=1; y=2*x+1; print x; print y ⇨

int x•; int y; x=1;…

D;int y•;x=1;…

D;D;•x=1;…

D;D;Ď•x=1;…

D;Ď•x=1;…

Ď•x=1;…

Ďx=1•; y=2*x+1;…

Ďx=1E'•; y=2*x+1;…

Ďx=E•; y=2*x+1;…

Ďx=E; y=2*x+1•;…

Ďx=E; y=2*x+1E'•;…

Ďx=E; y=2*xE'•;…

Ďx=E;y=2E'•;…

Ďx=E;y=E•;print x;print y

Ďx=E;y=E;print x;print y•

Ďx=E;y=E;print x;print yŠ'•

Ďx=E;y=E;print x;Š•

Ďx=E;y=E;print xŠ'•

Ďx=E;y=E;Š•

Ďx=E;y=EŠ'•

Ďx=E;Š•

Ďx=EŠ'• ⇨ ĎŠ• ⇨ P•

P→ĎŠ

Ď→ε

Ď→D;Ď

D→int d

Š→d=EŠ'

Š→print dŠ'

Š'→ε

Š'→;Š

E→iE'

E→dE'

E'→+ iE'

E'→* dE'

E'→ε

用移进归约框架模拟规范归约

- 活前缀：规范句型的前缀，它不包括句柄之后的符号。
- 最大活前缀：规范句型的一个前缀，它的后缀为句柄。
- 最大活前缀的任何前缀都是活前缀。
- 规范归约中栈内容的变化规律
  - 栈内容永远都是活前缀
  - 当栈内容为最大活前缀时，归约，即实现$\Mapsto$
  - 当栈内容不是最大活前缀时，移进，即实现$\Mapsto$
  - 特别地，若文法含有ε候选式，任何活前缀都是最大的。

提炼“移进-归约”过程p172

- 将规范句型$\rho\cdot x$的活前缀ρ作为栈内容，将x作为剩余串。
- 初始化$\rho^0=\varepsilon$，$x^0=w$。
- 设第i步栈内容为ρ^i，剩余串为x^i。
- 如果ρ^i是最大活前缀，即规范句型$\rho^i\cdot x^i$的句柄为$\gamma\in\tau(\rho)\cap ran(\mathcal{P})$，设$(A,\gamma)\in\mathcal{P}$，那么做归约（reduce）动作，即置栈内容为$\rho=\pi(\rho,|\rho|-|\gamma|)A$，当前句型仍然是$\rho x$，其中$\rho$已被修改，$x$没变。
- 如果ρ^i不是最大活前缀，那么做移进（shift）动作，即置$\rho=\rho\pi(x,1)$，句型变化为$\rho\pi(x,1)\cdot\tau(x,|x|-1)$。
- 如果$\rho^i=S$且$x^i=\varepsilon$，分析成功（accept动作）。
- 如果$\rho^i\neq S$且$x^i=\varepsilon$，分析不成功。

例

栈	w	动作
#	(())#	移进
#(	())#	移进
#((	))#	移进
#(()	)#	归约 X→()
#(X	)#	移进
#(X)	#	归约 X→(X)
#X	#	归约 S→X
#S	#	accept

1S→X
2X→(X)
3X→()

Z_0•(())#　　//初始化
⊨ Z_0(•())#　　//移进
⊨ Z_0((•))#　　//移进
⊨ Z_0(()•)#　　//移进
⊨ Z_0(X•)#　　//归约
⊨ Z_0(X)•#　　//移进
⊨ Z_0X•#　　//归约
⊨ Z_0S•#　　//归约；成功

观察：活前缀、句柄

例：存在冲突

- 若ρ≠δγΛ(A,γ)∈$\mathcal{P}$则移进；
- 若有ρ=δγΛ(A,γ)∈$\mathcal{P}$则归约。

S→En
S→n
S→ε
E→n+

#•n+n#
⇨ #•n+n#　　//移进归约冲突；选移进
⇨ #n•+n#　　//移进归约冲突、归约归约冲突；选移进
⇨ #n+•n#　　//移进归约冲突；选归约E→ n+
⇨ #E•n#　　//移进归约冲突；选移进
⇨ #En•#　　//归约归约冲突；选归约S→En
⇨ #S•#　　//分析成功

8.3构建规范规约模拟器

- 在讨论了栈的变化规律基础上，基于PDA模型，考虑用NFA控制栈的动作。称为模拟规范归约的模拟器记为itemPDA
- 文法项目与文法项目集
- 增广文法
- 活前缀的可用项目
- 活前缀的有效项目
- 构建itemDFA

文法项目集

- **定义8.7** 文法项目集和文法项目。文法G的项目集 itemset(G)={(A, α·β) |(A, αβ)∈$\mathcal{P}$}，其中(A, α·β)被称为文法项目，直观地表示为A→α·β，是由产生式规则A→αβ产生的项目。
- 另一种定义：产生式A→X_1...X_n的对应项目是A→•X_1...X_n，A→X_1•...X_n，...，A→X_1...X_n•，特别地，产生式A→ε的对应项目是A→•。
- 可以看出，由产生式A→γ为文法项目集贡献|γ|+1个项目。

8.3构建规范规约模拟器

- 在讨论了栈的变化规律基础上，基于PDA模型，考虑用NFA控制栈的动作。

•(())	//初始化
⊨(•())	//移进
⊨((•))	//移进
⊨(()•)	//移进
⊨(X•)	//归约
⊨(X)•	//移进
⊨X•	//归约
⊨S•	//成功

历史+未来

S→X
X→(X)
X→()

S→ •X
S→ X•
X→ •(X)
X→ (•X)
X→ (X•)
X→ (X)•
X→ •()
X→ (•)
X→ ()•

分析过程中栈内容变化规律采用状态来表达：

- 设置两类状态，移进状态和归约状态
- 在移进类状态下，栈不是最大活前缀。是否允许移进当前输入符号？
  - 在当前状态里指明当前输入符号为哪些终结符时移进，否则出错。
  - 在 [A→α•cβ]状态，如果tok=c那么移进
- 在归约类状态下，栈为最大活前缀。用哪个产生式归约？
  - 设句柄为γ，在当前状态里指明用A→γ归约。
  - 在 [A→γ•]状态，表示栈顶句柄γ归约为A。

分析过程中栈内容变化规律采用状态来表达：

- 观察上述移进归约过程中活前缀变化规律：
- 在[A→γ•]状态，栈为最大活前缀。注意到之前经过了一系列关于活前缀的变化过程，先后经历了这些状态：依次经过这些状态[A→π(γ, k)•τ(γ, |γ|-k)]，k=0,...,|γ|
- 设 $\gamma=X_1X_2...X_n$，X_i为文法符号，即单个变元或终结符，即就是[A→•γ] ,..., [A→X_1•X_2...X_n],..., [A→X_1X_2•...X_n],..., [A→γ•]
- 在[A→γ•]状态进行归约意味着完成了一个愿望，即归约为变元A。那么对于输入串w的语法分析就是消耗完w达成归约为S这个愿望。
- 那么，用A→γ归约这个愿望是在什么时候许下的呢？显然是在[A→•γ] 状态下许下的。
- 用A→γ归约（或简要地，归约出A）称为分析过程的一个目标，当分析过程进行到状态[A→γ•]时该目标就达成了。

分析过程中栈内容变化规律

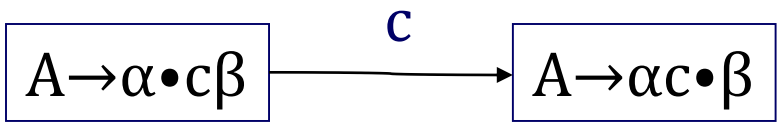
- 用$A\to\gamma$归约（或简要地，归约出A）称为分析过程的一个目标，为了达成这个目标，就有从状态$[A\to\bullet\gamma]$到状态$[A\to\gamma\bullet]$的状态转移路径，依次经过这些状态$[A\to\pi(\gamma,k)\bullet\tau(\gamma,|\gamma|-k)]$，$k=0,\ldots,|\gamma|$，这些中间状态都是处在实现这个目标的过程中，已经完成了一些（称为历史），即$\pi(\gamma,k)$，还剩下一些（称为未来），即$\tau(\gamma,|\gamma|-k)$。
- 一般地表示为$A\to\gamma^{past}\bullet\gamma^{future}$，该状态表示达成$A\to\gamma$归约目标需要一个过程，正进行到这个阶段，已经完成归约出γ^{past}这部分了（注意它是当前活前缀的后缀）。还需要继续努力，以完成归约出γ^{future}这部分（这些属于未来的事情）。
- 那么眼下又建立了一个子目标，就是⇨出点右符号，即$\pi(\gamma^{future},1)$。如果是变元，那么这个子目标就是归约出该变元，否则是终结符，那么与当前输入符号匹配并压入栈，也用⇨表示。
- 归约为A这个目标的实现，可能消耗输入串x。

基于文法穷尽所有状态及状态转移关系

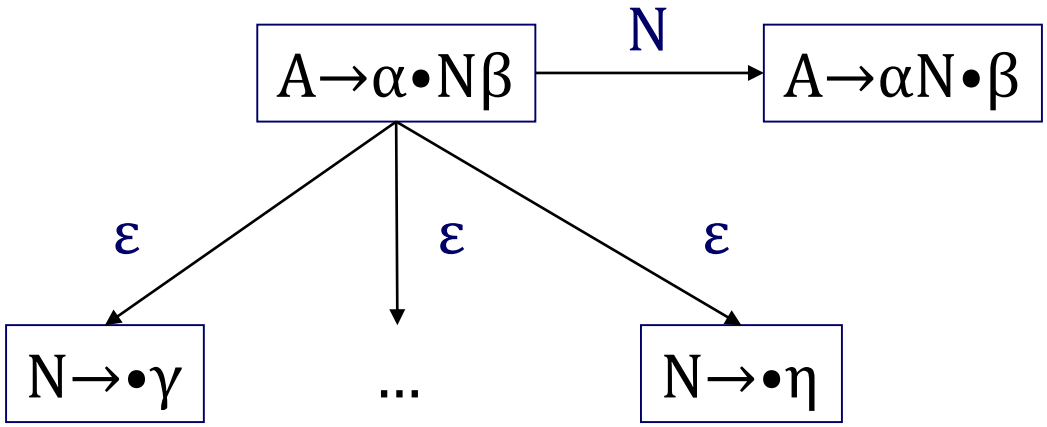
- 使用NFA实现穷举，称为itemNFA
- 已知CFG(V,T,$\mathcal{P}$,S)它的增广文法的项目集为itemset(G')。该集合就是itemNFA的状态集合， VUT是字母表，[S'→•S]是初始状态， {[S'→S•]}为接受状态集合。
- 转移函数：

itemNFA的转移函数

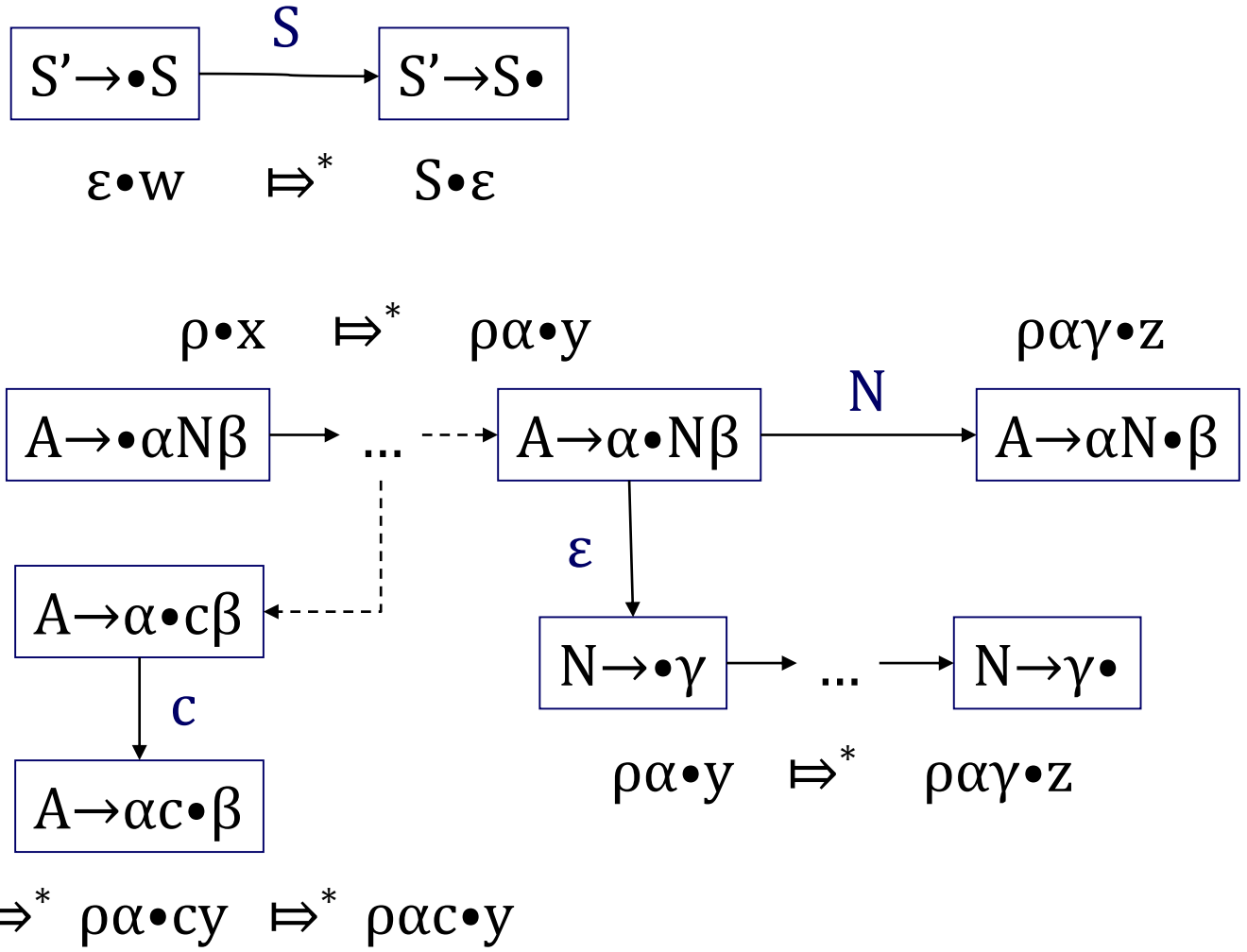
- $\upsilon([A\to\alpha\bullet c\beta], c)=\{[A\to\alpha c\bullet\beta]\}$ 对应于规范归约
  $\ldots\alpha\bullet c\ldots \Mapsto \ldots\alpha c\bullet\ldots$



- $\upsilon([A\to\alpha\bullet N\beta], \varepsilon)= \bigcup(N,\gamma)\in\mathcal{P}\cdot\{[N\to\bullet\gamma]\}$
- $\upsilon([A\to\alpha\bullet N\beta], N)=\{[A\to\alpha N\bullet\beta]\}$
- 对应于规范归约
  $\delta\alpha\bullet xy \Mapsto \delta\alpha N\bullet y$
  $x \Mapsto^{*} N$



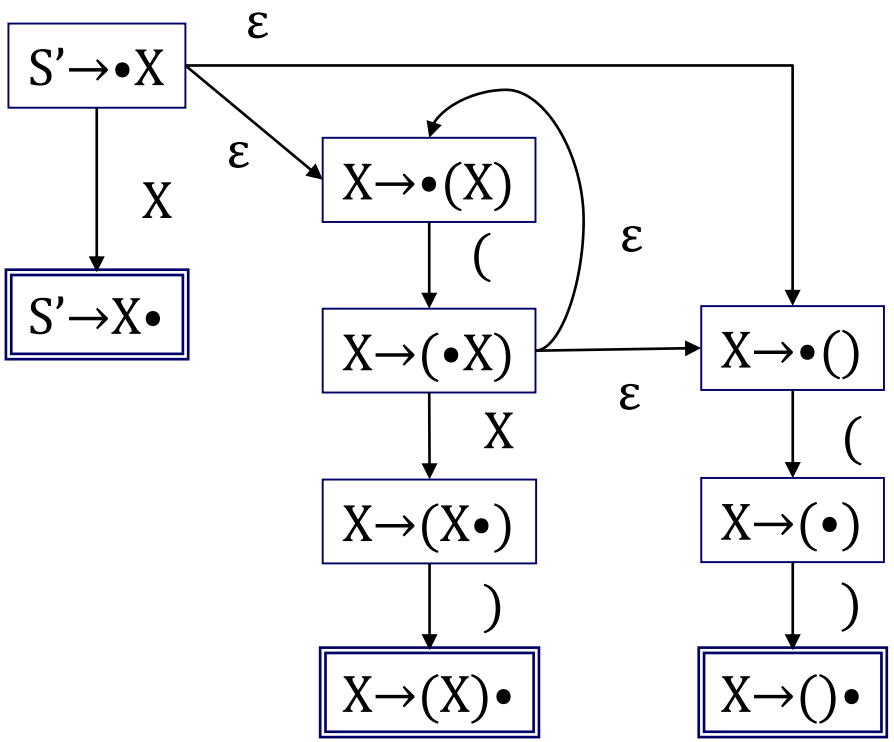
itemNFA示例


S
S'→•S
S'→S•
ε•w ⇨* S•ε
ρ•x ⇨* ρα•y
ραγ•z
A→•αNβ
...
A→α•Nβ
N
A→αN•β
ε
A→α•cβ
N→•γ
...
N→γ•
c
A→αc•β
ρα•y ⇨* ραγ•z
ρ•x ⇨* ρα•cy ⇨* ραc•y

移进归约分析过程的状态描述

- 已知增广文法 S'→X X→(X) X→() 试构建itemNFA。

ε•(())　　//初始化
⇨ (•())　　//移进
⇨ ((•))　　//移进
⇨ (()•)　　//移进
⇨ (X•)　　//归约
⇨ (X)•　　//移进
⇨ X•　　//归约成功



许愿与达愿（树立与实现目标）

Z_0•(())#　　//初始化

⇨ Z_0(•())#　　//移进

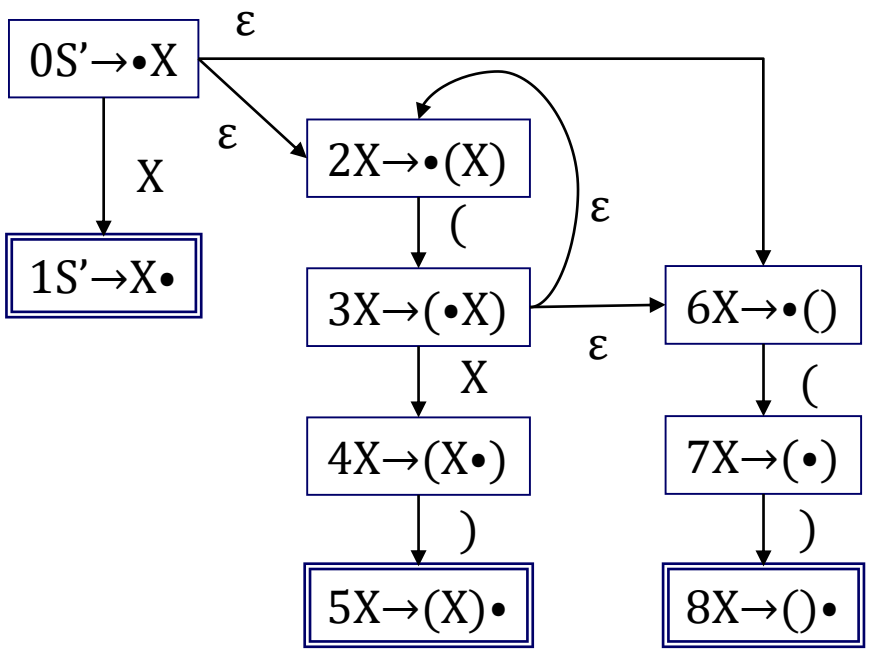
⇨ Z_0((•))#　　//移进

⇨ Z_0(()•)#　　//移进

⇨ Z_0(X•)#　　//归约

⇨ Z_0(X)•#　　//移进

⇨ Z_0X•#　　//归约成功



{0,2,6}三愿望; (; {3,2,6,7}后二愿望; (; {3,2,6,7}后二愿望;); {8}; 归约X→();

{0,2,6}三愿望; (; {3,2,6,7}达成第三愿望; X; {4};); {5}; 归约X→(X);

{0,2,6}达成第二愿望; X; {1}; 归约S'→X同时达成第一愿望同时分析成功。

文法项目集和术语

- G的项目集itemset(G)：
- S → •X　　初始项目、待约项目
- S → X•　　接受项目、归约项目、完全项目
- X → •(X)　　初始项目、移进项目
- X → (•X)　　待约项目
- X → (X•)　　移进项目
- X → (X)•　　归约项目、完全项目
- X → •()　　初始项目、移进项目
- X → (•)　　移进项目
- X → ()•　　归约项目、完全项目

CFG G:　S →X　X→ (X)　X→()

项目与活前缀的关联

- 为了明确表达ρ是规范句型ρx的活前缀，把句型写为ρ•x
- 项目A→α•β被称为是活前缀ρ=δα的可用项目
- 可用项目的意义在于表示这个活前缀ρ 是在状态[A→α•β]下可能存在的，并表明是从许愿用A→ αβ归约，到达此愿过程中的一步，即使该愿望后来没有达成。这个状态是从状态[A→•γ]到状态[A→•γ]的转移路径上一个状态结点，伴随这类转移路径的或许会有ρ•x ⇨* δα•zy ⇨* δαβ•y ⇨ δA•y。
- 如果活前缀ρ的可用项目A→α•β若还满足

  $$w \Rightarrow^* \rho\bullet x \Rightarrow^* \delta\alpha\bullet zy \Rightarrow^* \delta\alpha\beta\bullet y \Rightarrow \delta A\bullet y \Rightarrow^* \varepsilon\bullet S,$$

  那么A→α•β被称为活前缀ρ的有效项目

有效项目

Z_0•(())#　　0, 2

⇨ Z_0(•())#　　3, 6

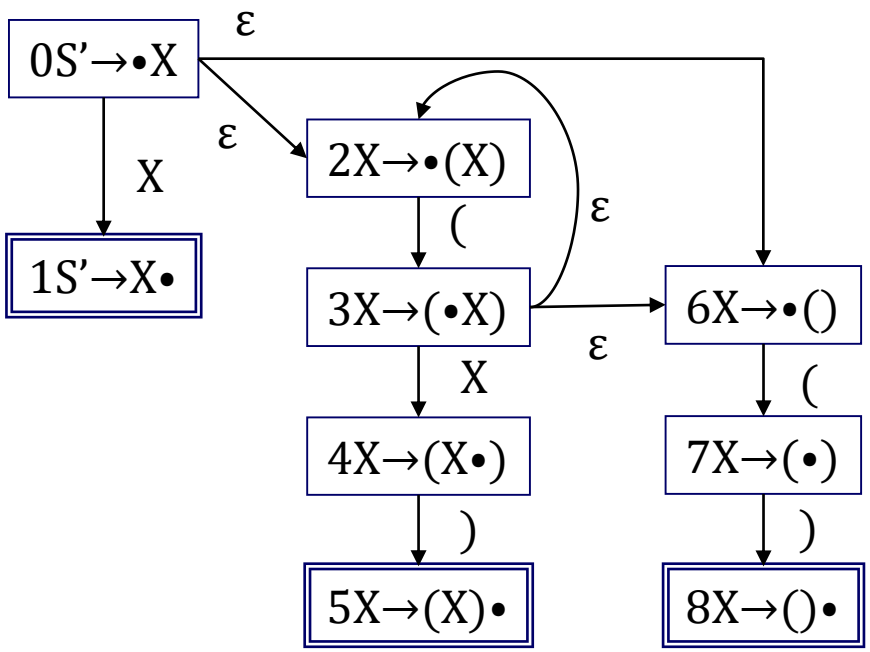
⇨ Z_0((•))#　　7

⇨ Z_0(()•)#　　8

⇨ Z_0(X•)#　　4

⇨ Z_0(X)•#　　5

⇨ Z_0X•#　　1



{0,2,6}三愿望; (; {3,2,6,7}后二愿望; (; {3,2,6,7}后二愿望;); {8}; 归约X→();

{0,2,6}三愿望; (; {3,2,6,7}达成第三愿望6; X; {4};); {5}; 归约X→(X);

{0,2,6}达成第二愿望2; X; {1}; 归约S'→X同时达成第一愿望1同时分析成功。

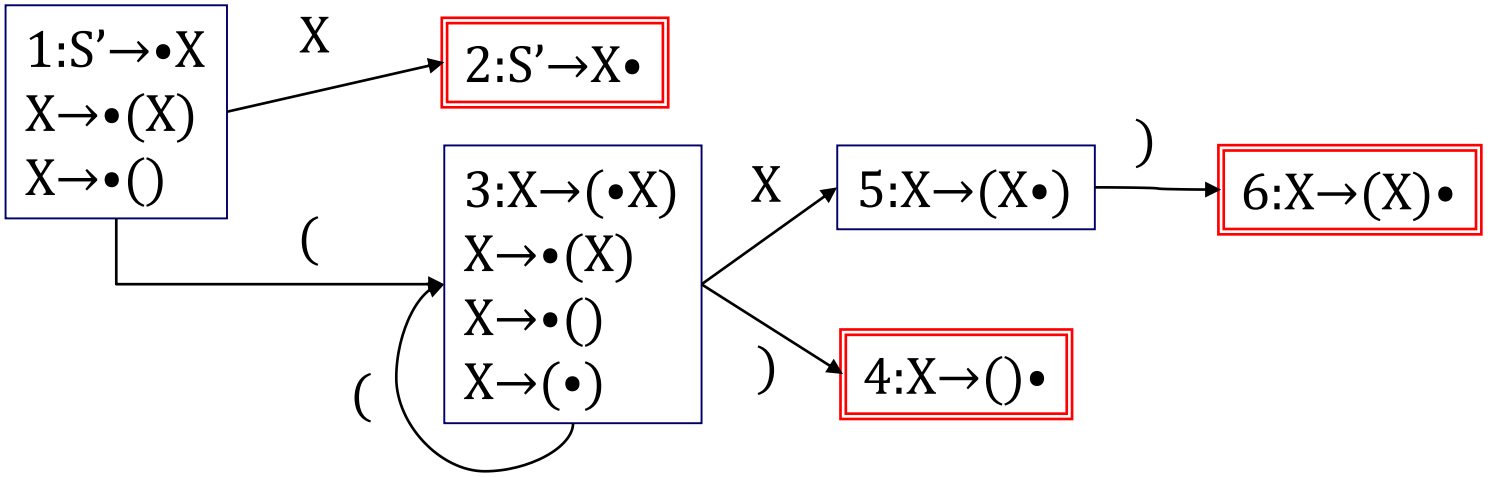
识别活前缀

- 一个有效项目对应多个活前缀。
- 在itemNFA中，每个状态是一个项目，以它为有效项目的所有活前缀只有在这个状态下才存在在栈上。
- 在itemNFA的所有状态下存在的活前缀集合就是所有的可能的活前缀。每个状态下存在的活前缀集合就是itemNFA所能识别的全部活前缀。
- w规范归约为S过程中依次出现的活前缀，就会对应于itemNFA一条状态转移路径，反过来这条路径就指明了当前状态下当前归约步的活前缀应该是什么。
- 将itemNFA确定化为itemDFA就得到了模拟规范归约的itemPDA。

确定的分析过程

[X→γ•]时，双栈同步弹出|γ|个符号，符号栈压入X，状态找顶s替换为υ(s, X)。

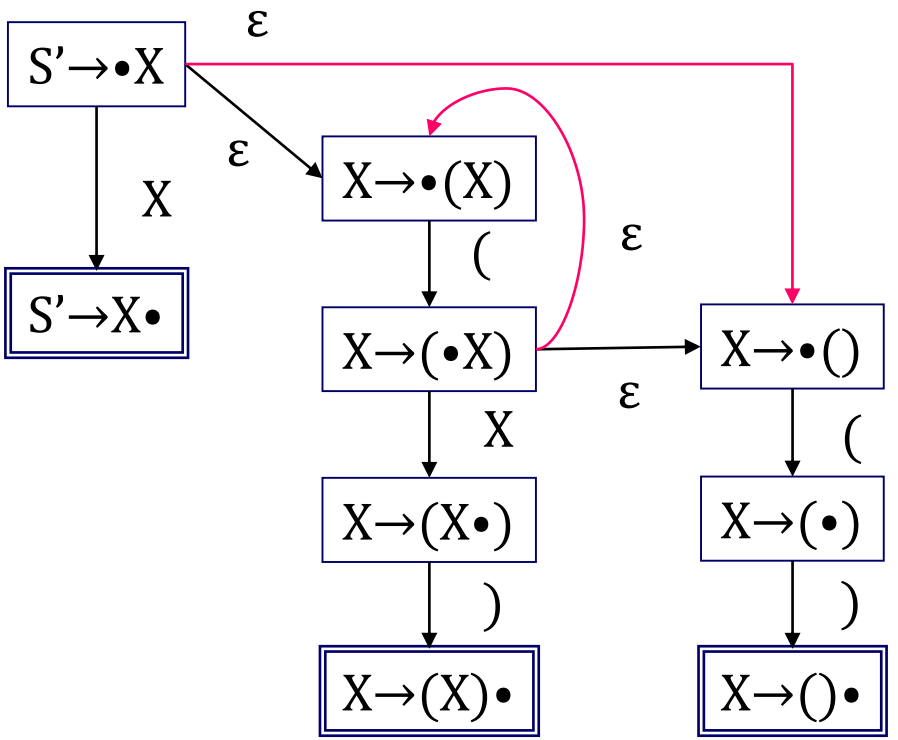
状态栈	符号栈	剩余串	动作
1	Z_0	(())#	shift 移进
13	Z_0(	())#	shift 移进
133	Z_0((	))#	shift 移进
1334	Z_0(()	)#	reduce X→()
135	Z_0(X	)#	shift 移进
1356	Z_0(X)	#	reduce X→(X)
12	Z_0X	#	reduce S→X；接受

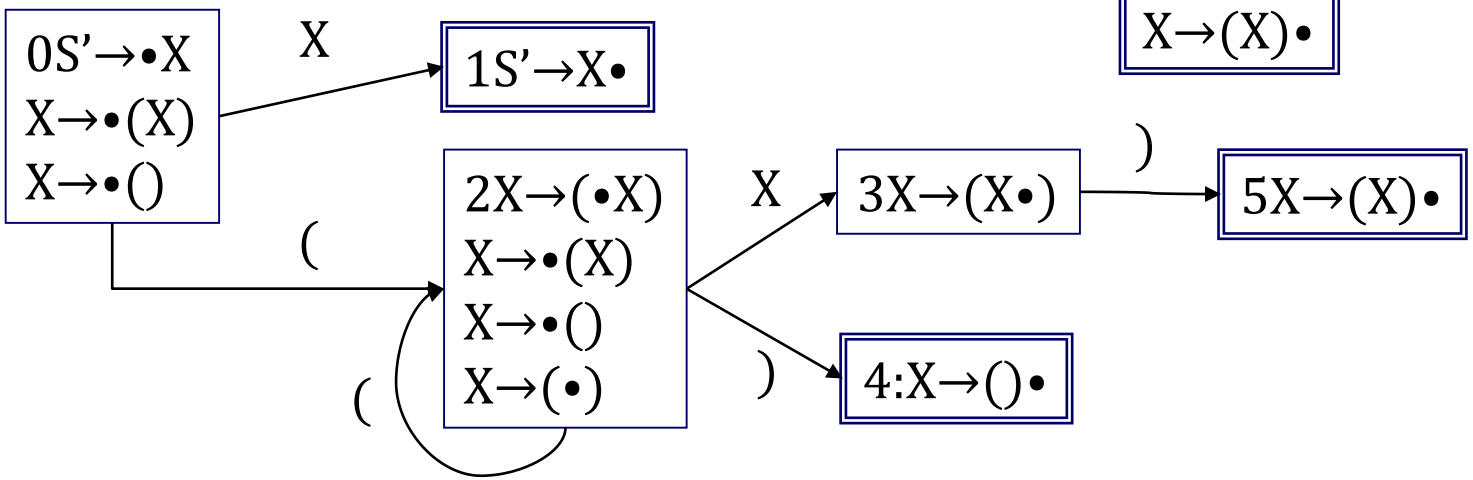


构造识别活前缀的DFA

①构建itemNFA并转itemDFA；

②直接构建itemDFA；





CFG: S'→X X→(X) X→()

有效项目集与规范簇

- 一个活前缀的所有有效项目构成的集合被称为识别该活前缀的有效项目集，简称项目集。
- 注意：分析活前缀与它的有效项目集之间这种对应关系是从文法入手建立识别活前缀方法的关键之处。
- 文法G的所有的有效项目集构成的集合称为该文法的LR(0)项目集规范簇，简称规范簇或LR(0)规范簇。
- 考察L(G)的每个元素（句子）的规范推导，收集到所有活前缀；再基于Item(G)文法确定每个活前缀的有效项目集，这些项目集构成的集合就是该文法的LR(0)规范簇。
- 任意一个活前缀都有唯一一个有效项目集；
- 规范归约过程中，活前缀α增长为活前缀α'，其中|α'|=|α|+1，是因为α的有效项目集到α'的有效项目集有状态转移关系；
- 识别活前缀的NFA和DFA给出了有效项目集间的转移关系。
- 活前缀的DFA的状态集合就是LR(0)项目集规范簇。

直接构造识别活前缀的DFA

- 给定CFG G=(𝕍,T,$\mathcal{P}$,S)，构造识别活前缀的DFA(Q, Σ, υ, q_0, F)。
- itemDFA 的状态是项目集合
- 初始状态q_0=ω([S'→•S])
  - 项目集q的ε-闭包ω(q)：
  - 如果[A→α•Nβ]∈ω(q)，那么[N→•γ]∈ω(q)， 其中(A,αNβ),(N,γ)∈$\mathcal{P}$。
  - 直观地，itemNFA中， [A→α•Nβ]到[N→•γ]有ε-转移。
- 对于q∈Q，令p=υ(q,X)，X∈𝕍∪T，那么，
  - p= ω({[A→αX•η] | [A→α•Xη]∈q})，
  - p∈Q，
  - 若p仅含完全项目，那么p∈F。

构造itemDFA算法

输入：CFG G=(𝕍,T,$\mathcal{P}$,S)

输出：识别活前缀的DFA(Q, V∪T, υ, q_0, F)。

Q=φ; F=φ; υ=φ;

将q_0=ω([S'→•S])加入Q;

while (Q中存在未标记元素) {

令q是Q中一个未标记元素，并标记q;

for (X∈𝕍∪T) {p=φ;

for (s∈q && s==[A→α•Xη]) 将[A→αX•η]加入p;

if(p!=φ){ p= ω(p); 将p加入Q;

将((q,X),p)加入υ; }　　//即，令υ(q,X)=p

若p含完全项目，那么将p加入F; }

}

基于itemDFA的LR(0)分析算法

- 输入：识别活前缀的DFA(Q,𝕍∪T,υ,q_0,F)，CFG(𝕍,T,$\mathcal{P}$,S)。
- 输出：分析成功与否。

```
scan();                    //读入当前符号赋给全局变量tok
bi-push(q0,#);             //双栈同步压入初始状态q0和符号#
while(1){bi-top(q,a);      //将双栈栈顶符号分别赋给q和a
  if ((p=υ(q,tok))∉F)      //是移进状态
    bi-push(p,tok);  //把p压入状态栈、tok压入符号栈
  else if((p=υ(q,tok))∈F){      //是句柄识别状态
    令p==[A→γ•]，(A,γ)∈P
    bi-pop(|γ|);       //双栈都退掉|γ|个符号
    bi-push(p,A);
    if(A==S) break;          //已满足接受条件
  }else error(); }      //其它情况都属于错误
```

构造LR(0)分析表

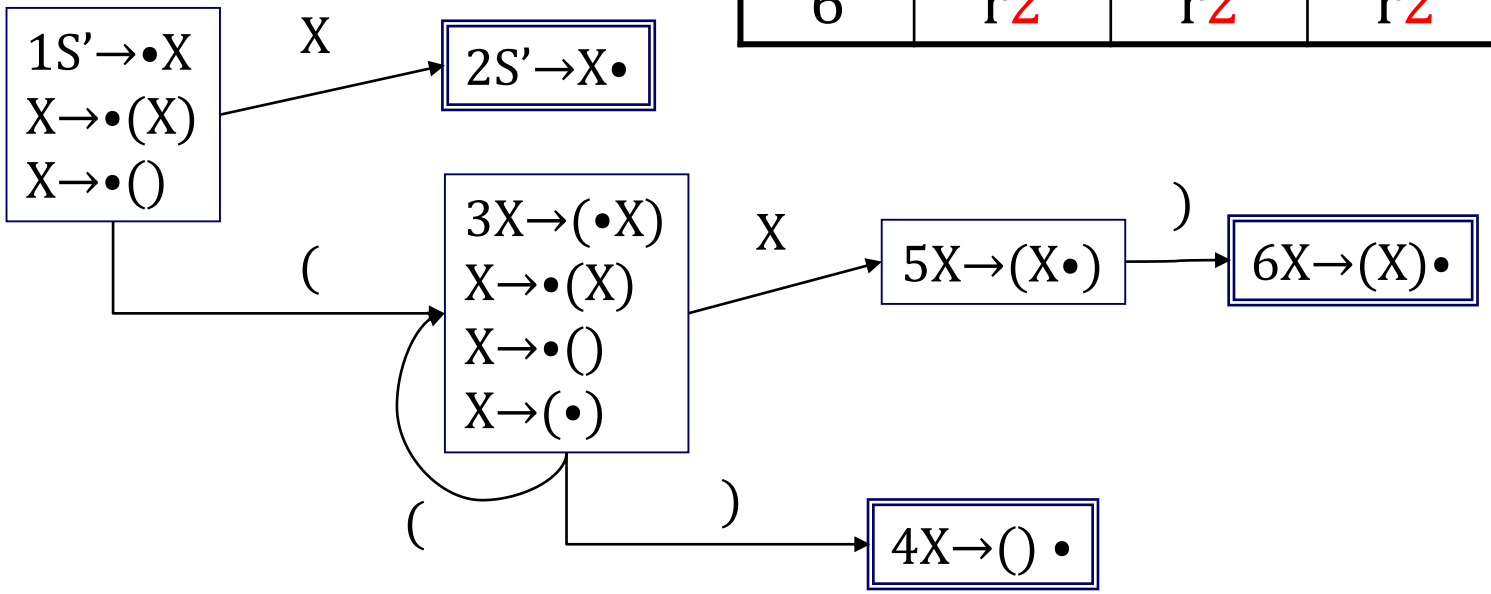
- 根据itemDFA构造LR(0)分析表：
- ACTION[m,a]=?
  - sn：若有从m射出标记为a的弧射入n；
  - acc：m含接受项目；
  - rk：m含产生式k的完全项目（非接受项目）。
- GOTO[m,A]=?
  - n：若有从m射出标记为A的弧射入n。

- LR(0)分析表是itemPDA的转移函数，确定性地控制栈的移进-归约过程。

例：LR(0)分析表

- 给定CFG；
- 先求出DFA；
- 再填ACTION表；
- 填GOTO表；
- 完成。

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



1S→X
2X→(X)
3X→()

基于分析表的分析过程

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

ACTION[*m*,*a*]:

- s*n*: bi-push(*n*,*a*)
- r*n*: 按产生式*n*A→α归约:
  bi-pop(|α|);bi-top(m',X]
  bi-push((GOTO[m',A],A)。
- acc: 分析成功。

状态栈	符号栈	剩余串	动作
1	#	(())#	shift
13	#(	())#	shift
133	#((	))#	shift
1334	#(()	)#	reduce X→()
135	#(X	)#	shift
1356	#(X)	#	reduce X→(X)
12	#X	#	reduce S→X
		#	accept

1S→X
2X→(X)
3X→()

基于分析表的LR(0)分析算法

- 输入：分析表，w，双栈空栈。输出：分析成功与否。

```
bi-stack(1,#); //双栈初始化分别为初始状态1和符号#
scan();        //读入当前符号赋给全局变量tok
while(1){
   bi-top(m,c);          //将双栈栈顶符号分别赋给m和c
   if (ACTION[m,tok]==sn)bi-push(n,tok);
   else if(ACTION[m,tok]==rk){
         bi-pop(body-size(k)); //弹出产生式k的体长那么多
         A=head(k);            //产生式的头，即左部变元
         bi-top(m',c);   bi-push(GOTO[m',A],A);
    }else if(ACTION[m,tok]==acc)break;   //接受
    else error();          //其它情况都属于错误
}
```

8.4 SLR(1)分析法

- LR(0)分析要求基于LR(0)文法进行，
- 满足LR(0)文法的条件，即LR(0)规范簇中的项目集要求：
  - 如果含有一个完全项目，就不能再含有任何别的项目。
  - 如果含有移进项目，这些移进项目的点后终结符必须各不相同。
- 基于LR(0)文法的分析算法有确定的流程。

- SLR(1)分析的思想
- 查看当前输入符号来消解LR(0)规范簇中的冲突：
  - 移进-归约冲突；
  - 归约-归约冲突。

冲突例子回顾

- 若ρ≠δγΛ(A,γ)∈$\mathcal{P}$则移进；
- 若有ρ=δγΛ(A,γ)∈$\mathcal{P}$则归约。

S→En
S→n
S→ε
E→n+

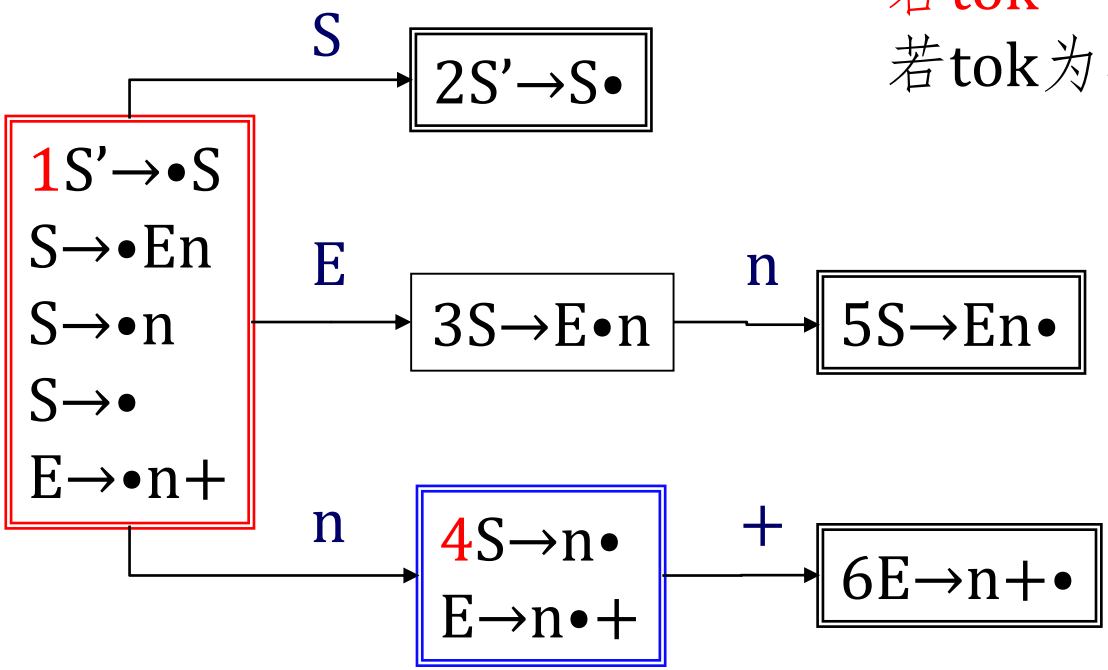
#•n+n#
⇨ #•n+n#　　//移进归约冲突；选移进
⇨ #n•+n#　　//移进归约冲突、归约归约冲突；选移进
⇨ #n+•n#　　//移进归约冲突；选归约E→n+
⇨ #E•n#　　//移进归约冲突；选移进
⇨ #En•#　　//归约归约冲突；选归约S→En
⇨ #S•#　　//分析成功

冲突消解思路

S'→S

S→En|n|ε

E→n+



状态1移进归约冲突。

冲突消解思路：

若tok∈FOLLOW(S)则归约；

若tok==n则移进(推迟消解)；

若tok为其它则出错。

状态4移进归约冲突。

消解：

若tok==#则归约；

若tok==+则移进；

若tok为其它则出错。

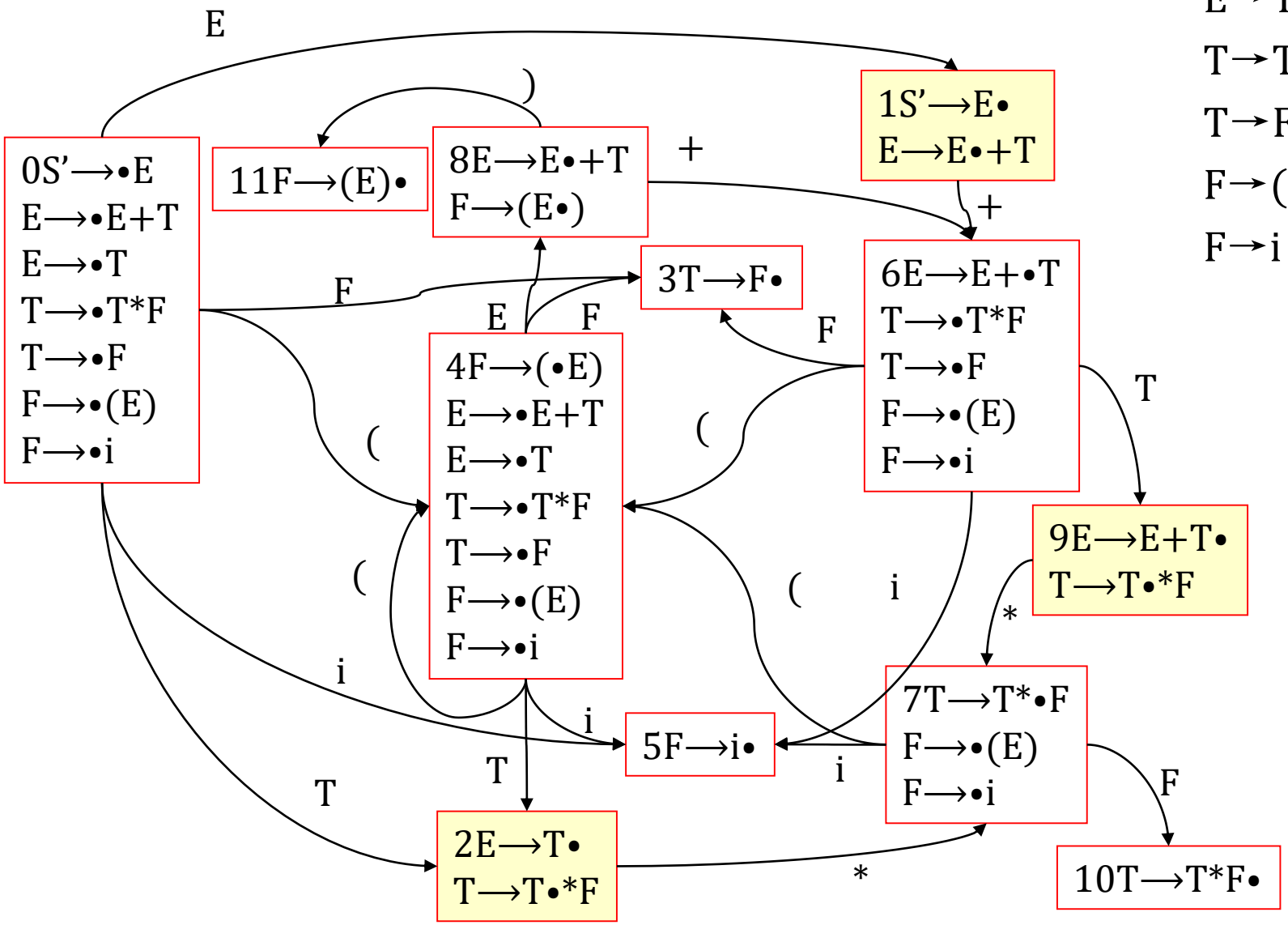
这种情况属于SLR(1)消解方法，所以该文法是SLR(1)文法。

SLR(1)冲突消解规则

- DFA的当前状态包含 [A→γ•]
  - 如果当前输入符号a∈FOLLOW(A)，那么用A→γ归约。
- DFA的当前状态包含{X→α•cβ, A→α•}且c∉FOLLOW(A)，
  - 若a=c则移进；
  - 若a∈FOLLOW(A)则用A→α归约；
- DFA当前状态包含{X→α•cβ, A→α•, B→α•}且c∉FOLLOW(A)，c∉FOLLOW(B)，FOLLOW(A)∩FOLLOW(B)=φ，
  - 若a=c则移进；
  - 若a∈FOLLOW(A)则用A→α归约；
  - 若a∈FOLLOW(B)则用B→α归约；
- 若DFA的状态是其它有冲突的情形，则消解失败并报告错误。

例：构造SLR(1)分析表

S→E
E→E+T
E→T
T→T*F
T→F
F→(E)
F→i



例：冲突消解与填表

0: S→E
1: E→E+T
2: E→T
3: T→T*F
4: T→F
5: F→(E)
6: F→i

- 规范集0没有冲突，直接填表
- ACTION[0,(]=s5；ACTION[0,i]=s4
- GOTO[0,E]=1；GOTO[0,T]=2；GOTO[0,F]=3
- 规范集1移进-归约冲突
- +∉FOLLOW(S)，ACTION[1,+]=s6，
- FOLLOW(S)={#}，ACTION[1,#]=acc
- 规范集2移进-归约冲突
- FOLLOW(E)={#,+,)}，
  ACITON[2,#]=ACTION[2,+]=ACTION[2,)]=r2；
- ACTION[2,*]=s7
- 规范集3无冲突
- FOLLOW(T)={+,*,),#}，
  ACTION[3,+]=ACTION[3,*]=ACTION[3,)]=ACTION[3,#]=r4

例：冲突消解与填表

0: S→E
1: E→E+T
2: E→T
3: T→T*F
4: T→F
5: F→(E)
6: F→i

- 项目集4没有冲突，直接填表
- ACTION[4,(]=s4；ACTION[4,i]=s5
- GOTO[4,E]=8；GOTO[4,T]=2；GOTO[4,F]=3
- 项目集5无冲突
- FOLLOW(F)={+,*,),#}，
  ACTION[5,+]=ACTION[5,*]=ACTION[5,)]=ACTION[5,#]=r6
- 项目集6无冲突
- ACTION[6,(]=s4；ACTION[6,i]=s5
- GOTO[4,T]=9；GOTO[4,F]=3
- 项目集7无冲突
- ACTION[7,(]=s4；ACTION[7,i]=s5
- GOTO[4,F]=10

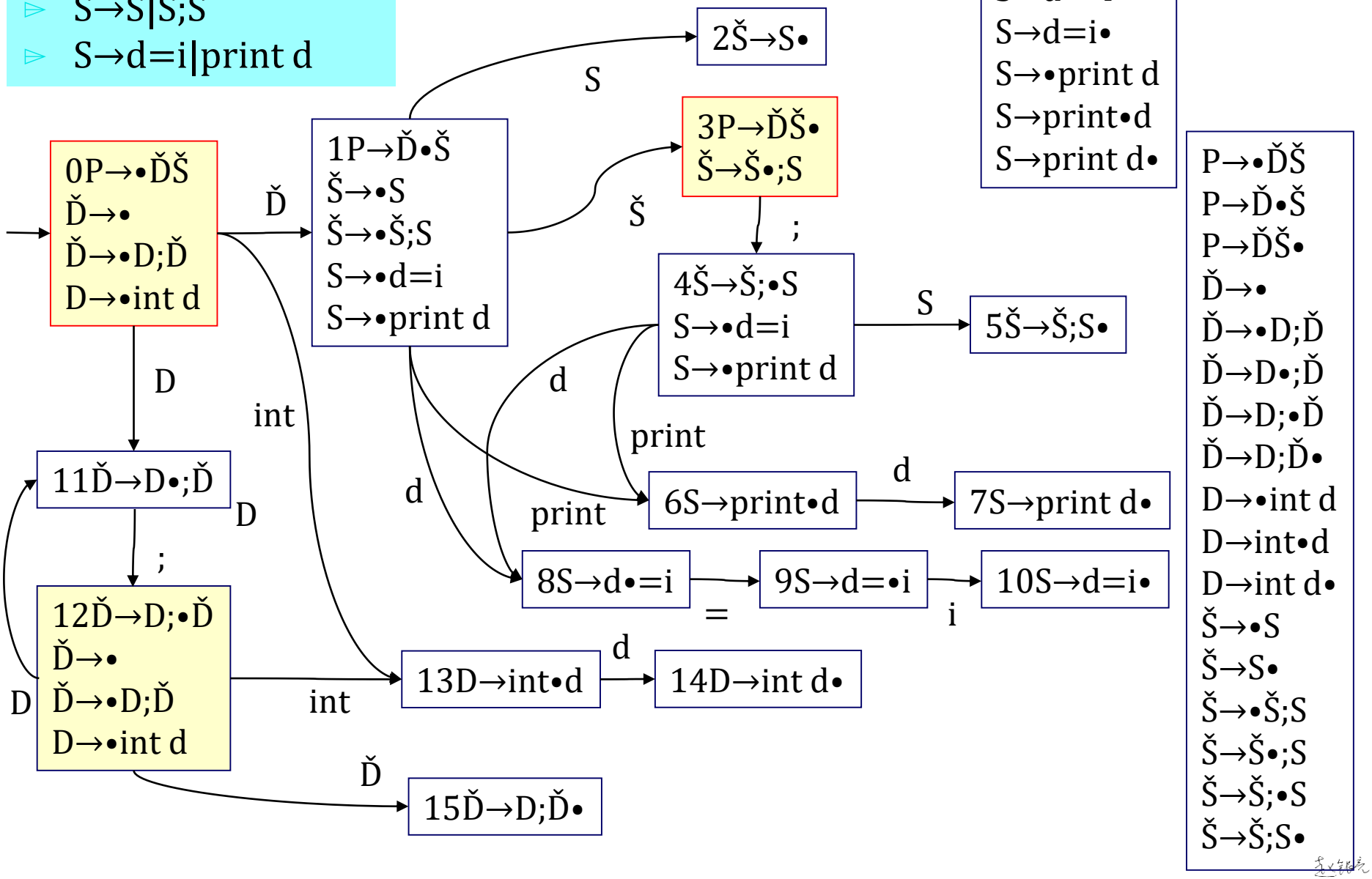
例：冲突消解与填表

0: S→E
1: E→E+T
2: E→T
3: T→T*F
4: T→F
5: F→(E)
6: F→i

- 项目集8没有冲突，直接填表
- ACTION[8,)]=s11；ACTION[4,+]=s6
- GOTO[4,E]=8；GOTO[4,T]=2；GOTO[4,F]=3
- 项目集9移进-归约冲突
- FOLLOW(E)={+,),#}， ACTION[9,*]=s7
- ACTION[9,+]=ACTION[9,)]=ACTION[9,#]=r1
- 项目集10无冲突
- FOLLOW(T)={+,*,),#}，ACTION[10,+]=ACTION[10,*]=ACTION[10,)]=ACTION[10,#]=r3
- 项目集11无冲突
- FOLLOW(T)={+,*,),#}，ACTION[11,+]=ACTION[11,*]=ACTION[11,)]=ACTION[11,#]=r5

例：构造DFA

- P→ĎŠ
- Ď→ε|D;Ď
- D→int d
- Š→S|Š;S
- S→d=i|print d



例：SLR(1)分析表

	;	int	d	=	i	pri	#	P	Ď	Š	D	S
0		s13	r2			r2			1		11	
1			s8			s6				3		2
2	r5						r5					
3	s4						acc					
4			s8			s6						5
5	r6						r6					
6			s7									
7	r8						r8					
8				s9								
9					s10							
10	r7						r7					
11	s12											
12		s13	r2			r2			15		11	
13			s14									
14	r4											
15			r3			r3						

1:P→ĎŠ
2:Ď→ε
3:Ď→D;Ď
4:D→int d
5:Š→S
6:Š→Š;S
7:S→d=i
8:S→print d

DFA当前状态包含
{X→α•cβ, A→α•, B→α•}且
c∉FOLLOW(A)，
c∉FOLLOW(B)，
FOLLOW(A)∩FOLLOW(B)=φ，
那么，
若a=c则移进；
若a∈FOLLOW(A)则用A→α归约；
若a∈FOLLOW(B)则用B→α归约。

0:P→•ĎŠ
Ď→•
Ď→•D;Ď
D→•int d

3:P→ĎŠ•
Š→Š•;S

12:Ď→D;•Ď
Ď→•
Ď→•D;Ď
D→•int d

FIRST:
P {d,int,print}
Ď {ε,int}
Š {d,print}
D {int}
S {d,print}

FOLLOW:
P {#}
Ď {d,print}
Š {#,;}
D {;}
S {#,;}

SLR(1)消解简单冲突的通用规则

- 假定LR(0)规范簇的某个规范集中含有m个移进项目和n个归约项目：
  - $X_i \rightarrow \alpha \bullet c_i \beta_i$, $i=1,...,m$,
  - $A_i \rightarrow \alpha \bullet$, $i=1,...,n$。
- 该规范集满足：
  - $card(\{c_i \mid i=1,...,m\})=m$； （可延迟：<=m）
  - $\{c_i \mid i=1,...,m\}$，$FOLLOW(A_1)$，…，$FOLLOW(A_n)$两两相交都为空集。
- 当前状态为该规范集、当前输入符号为a时，分析过程为：
  - 若$a \in \{c_i \mid i=1,...,m\}$那么移进；否则，
  - 若$a \in FOLLOW(A_k)$，$k=1,...,$或n，则用产生式$A_k \rightarrow \alpha$进行归约；否则，
  - 其它情况报错。

如何在SLR(1)中容忍其它冲突

- 定理：任何歧义性文法都不是LR文法。
- 证明略。

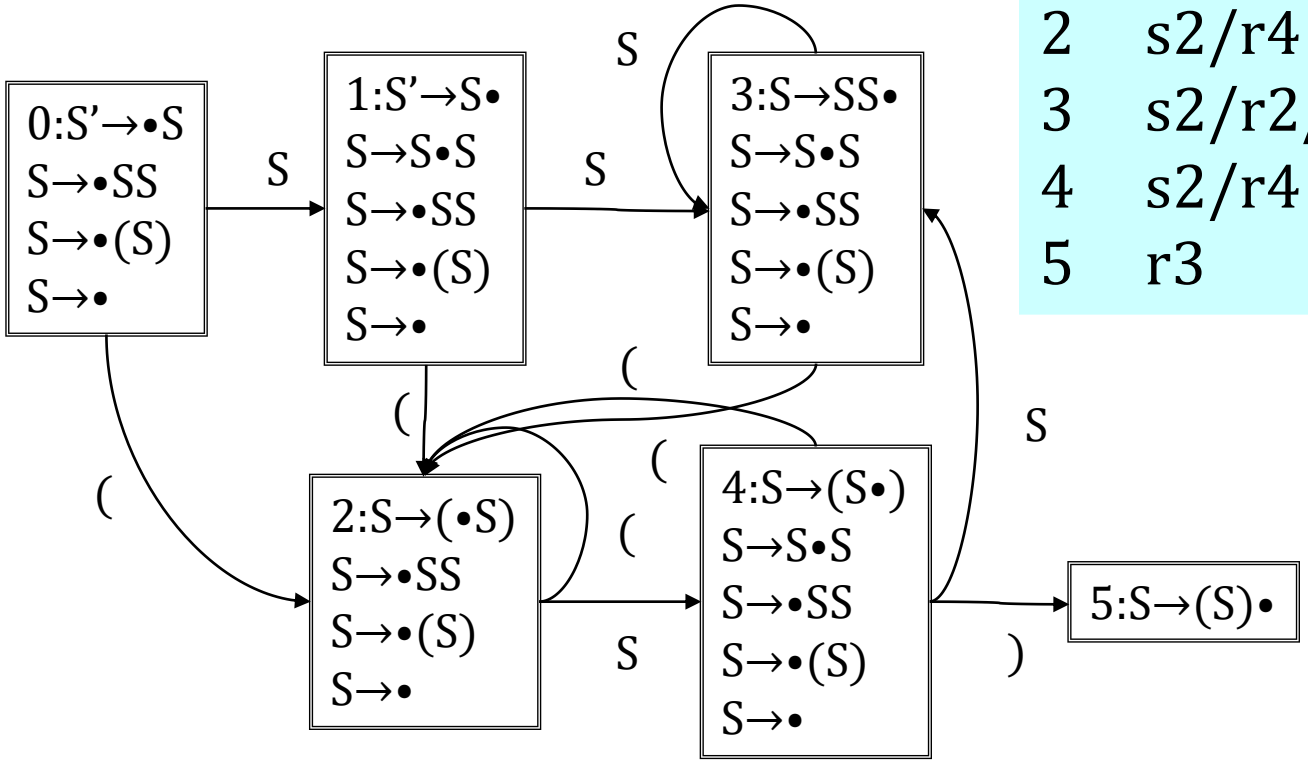
- 歧义性文法带来许多好处：
  - 规范簇更小；
  - 表示更简单；
  - 分析过程更简短等。

- 那么，考虑容忍歧义性的途径：
  - 让更多的冲突消解规则起作用；
  - 通过修剪文法推迟归约来回避风头。

SLR(1)中其它冲突

- S'→S
- S→SS|(S)|ε

	(	)	#	S
0	s2/r4	r4	r4	1
1	s2/r1/r4	r4	acc	3
2	s2/r4	r4	r4	4
3	s2/r2/r4	r2/r4	r2	3
4	s2/r4	s5/r4	r4	3
5	r3	r3	r3	



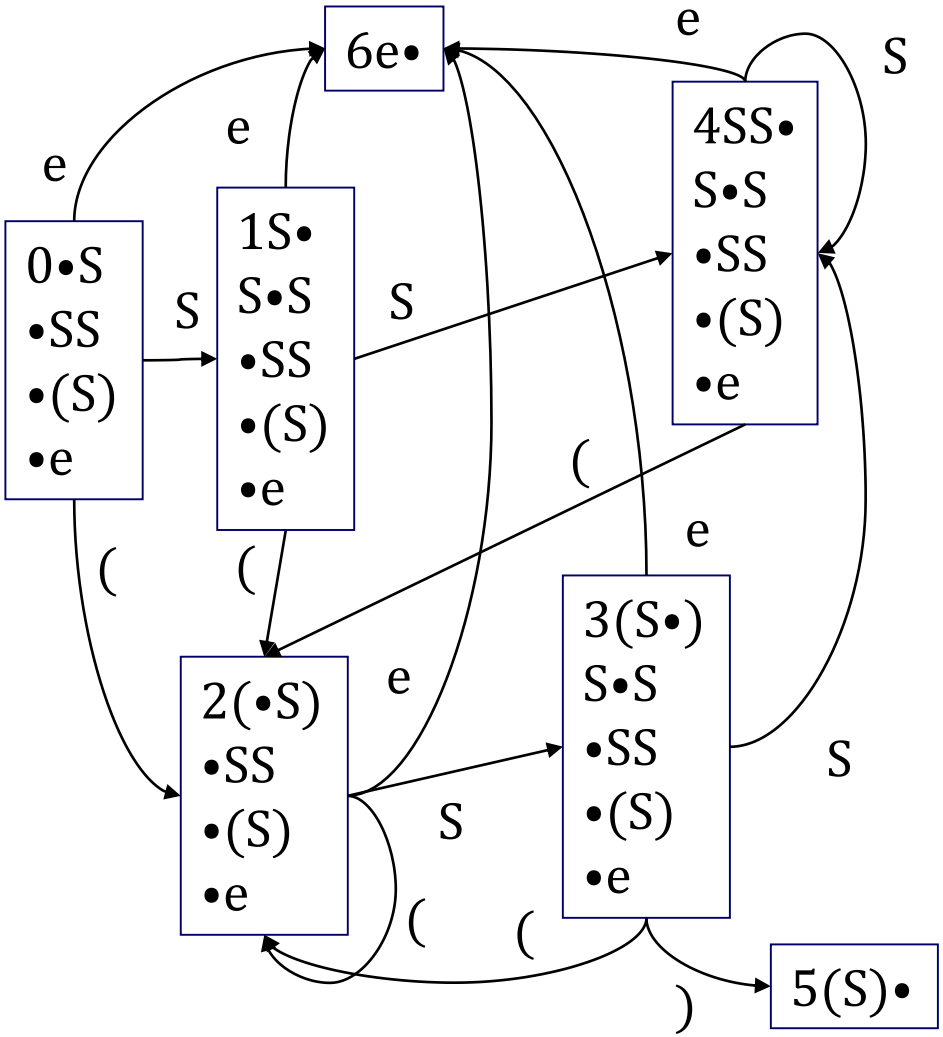
S'	{(,ε}	{#}
S	{(,ε}	{(,),#}

1:S'→S
2:S→SS
3:S→(S)
4:S→ ε

文法$G_{(e)}$

S'→S
S→SS
S→(S)
S→e

	e	(	)	#	S
0	s6	s2			1
1	s6	s2		acc	4
2	s6	s2			3
3	s6	s2	s5		4
4	s6	s2/r2	r2	r2	4
5		r3	r3	r3	
6	r4	r4	r4	r4	



是文法歧义性导致的冲突

1	#	()()#	s4
14	#(	)()#	r4
145	#(S	)()#	s6
1456	#(S)	()#	r3
12	#S	()#	s4
124	#S(	)()#	r4
1245	#S(S	)()#	s6
12456	#S(S)	()#	r3
123	#SS	()#	r2
12	#S	()#	s4
124	#S(	)#	r4
1245	#S(S	)#	s6
12456	#S(S)	#	r3
123	#SS	#	r2
12	#S	#	acc

	(	)	#	S	S'
1	s4/r4	r4	r4	2	
2	s4/r4	r4	acc	3	
3	s4/r2	r2	r2	3	
4	s4/r4	r4	r4	5	
5	s4/r4	s6	r4	3	
6	r3	r3	r3		

1:S'→S
2:S→SS
3:S→(S)
4:S→ε

为什么S→SS在程序设计语言中少见?

- 靠LR(1)解决，或，
- 文法中避免同一符号并列出现

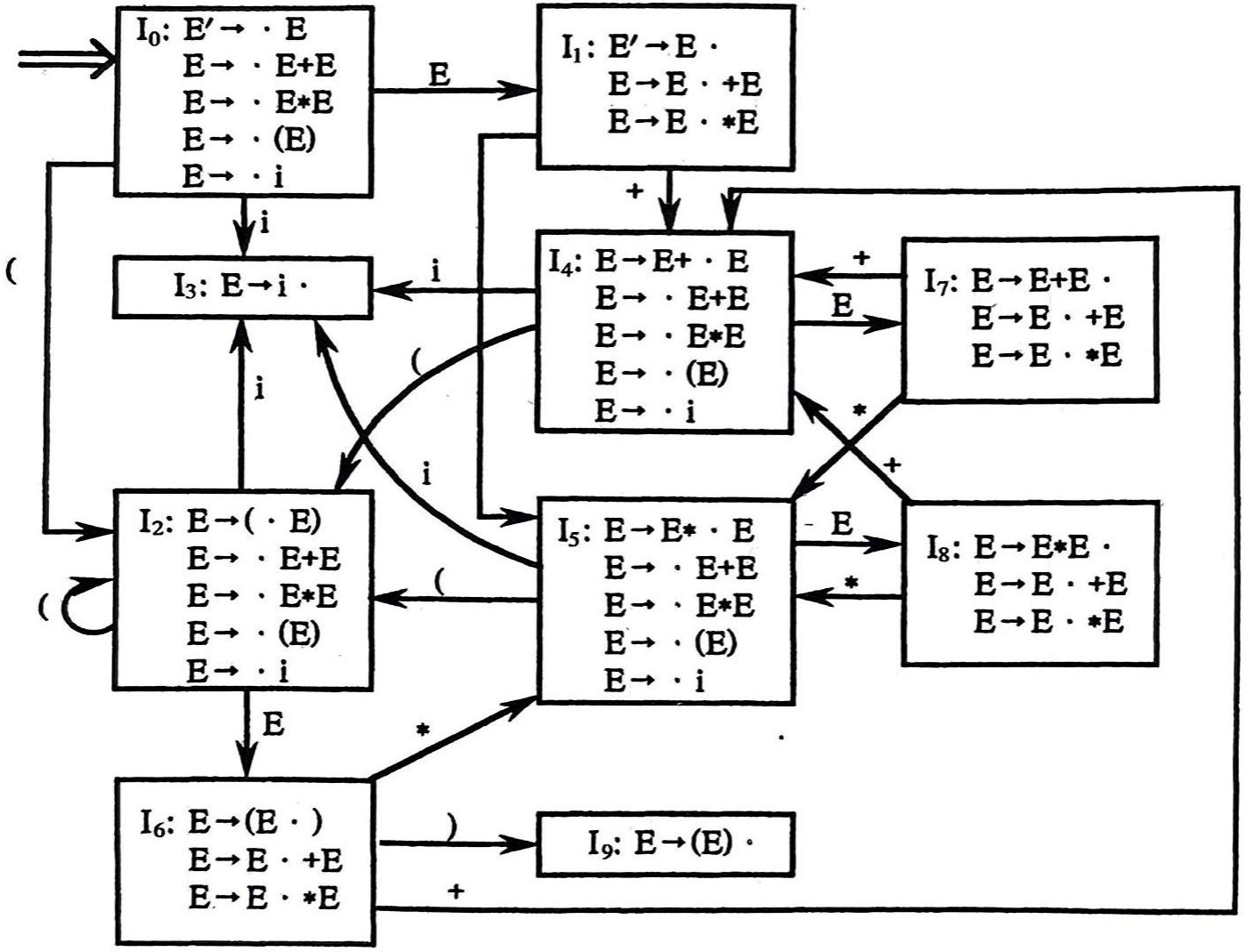
1	#	(()))#	s4
14	#(	()))#	s4
144	#((	))#	s4
1444	#(((	))#	r4
14445	#(((S	))#	s4
144456	#(((S)	)#	r3
1445	#((S	)#	s6
12456	#((S)	)#	r3
123	#(S	)#	r2

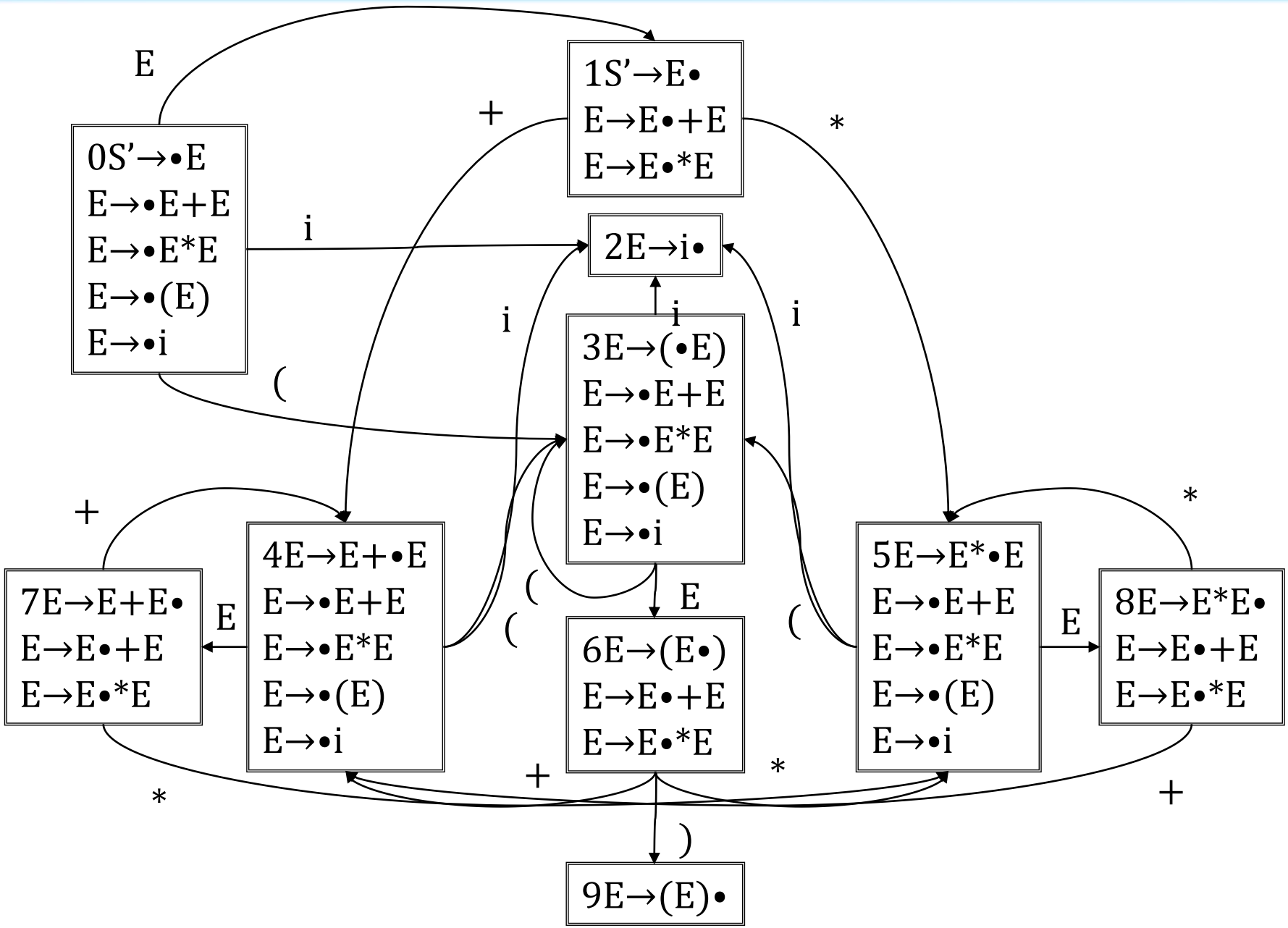
	(	)	#	S	S'
1	s4/r4	r4	r4	2	
2	s4/r4	r4	acc	3	
3	s4/r2	r2	r2	3	
4	s4/r4	r4	r4	5	
5	s4/r4	s6	r4	3	
6	r3	r3	r3		

1:S'→S
2:S→SS
3:S→(S)
4:S→ε

SLR(1)中歧义性引起的冲突

- 利用优先次序规则消解冲突




0S'→•E
E→•E+E
E→•E*E
E→•(E)
E→•i
1S'→E•
E→E•+E
E→E•*E
2E→i•
3E→(•E)
E→•E+E
E→•E*E
E→•(E)
E→•i
4E→E+•E
E→•E+E
E→•E*E
E→•(E)
E→•i
5E→E*•E
E→•E+E
E→•E*E
E→•(E)
E→•i
6E→(E•)
E→E•+E
E→E•*E
7E→E+E•
E→E•+E
E→E•*E
8E→E*E•
E→E•+E
E→E•*E
9E→(E)•
E
+
*
i
(
)

利用优先次序规则来消解冲突

- 状态I_1移进-归约冲突。 用SLR(1)规则来消解。
- E'→E•　　E→E•+E | E•*E
- FOLLOW(E')={#}， tok为#时归约，为+或*时移进。

- 状态I_7移进-归约冲突。 用优先次序规则来消解。
- E→E+E• | E•+E | E•*E
- FOLLOW(E)={#,+,*,)}，tok为+、)、#时都归约，为*时移进
- 等价于这个状态中没有第二个项目。第二个项目不起作用
- 状态I_8移进-归约冲突。 用优先次序规则来消解。
- E→E*E• | E•+E | E•*E
- tok∈FOLLOW(E)时归约。
- 等价于这个状态中没有第二、三个项目。

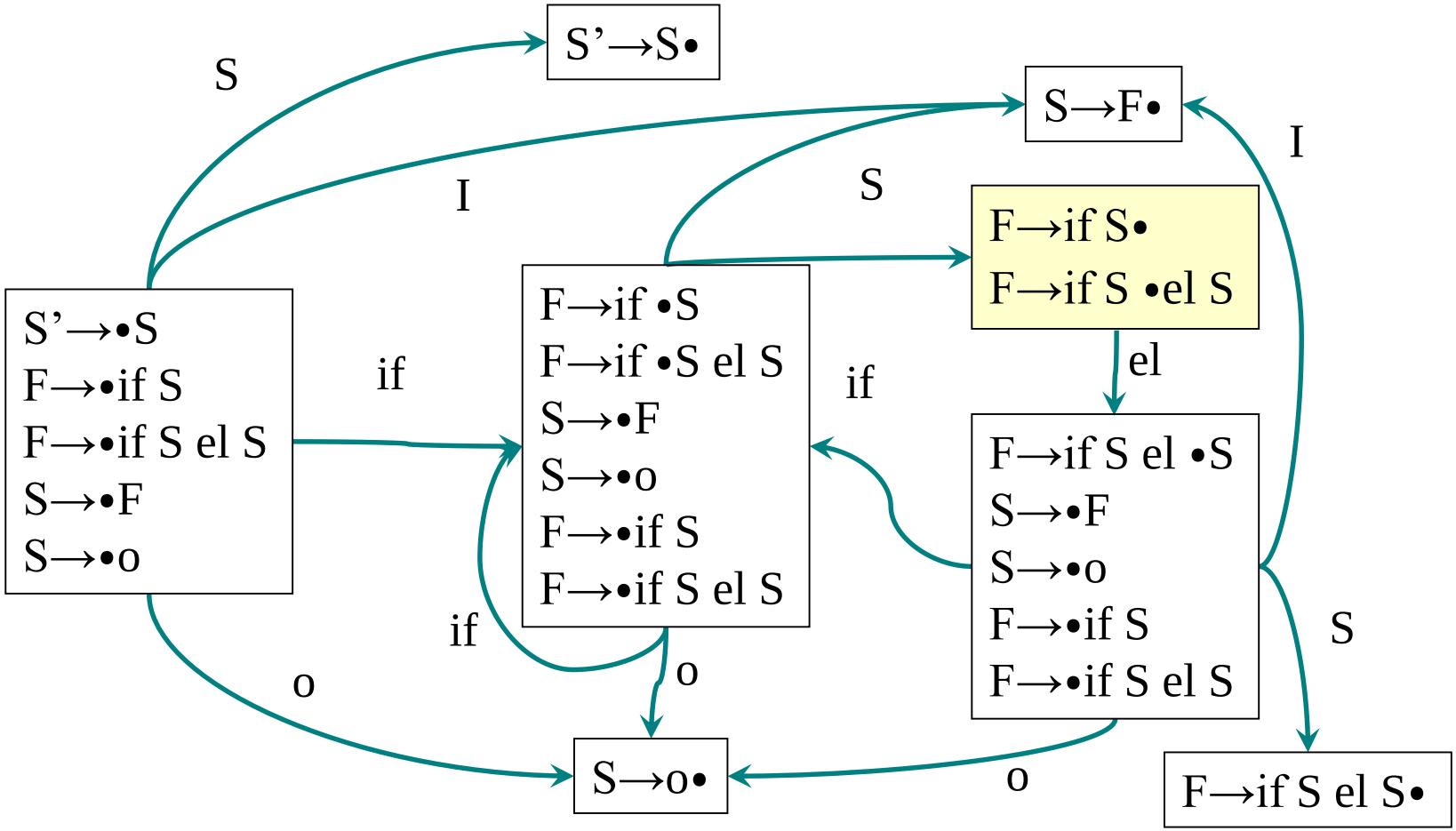
悬挂else歧义性文法

- S → F | o
- F → if S | if S el S
- FOLLOW(F)={el,#}

- Š → Š ; S | S
- S → F | o
- F → if S | if S el S
- FOLLOW(F)={el,#,;}

用最近匹配原则来消解悬挂ELSE冲突

- FOLLOW(F)={el,#}
- 在该状态时，若tok为el则移进，为#则归约



其他容忍冲突方法：修剪文法推迟归约

- 遇到移进-归约冲突不能消解时，修剪文法推迟归约，从而使得移进项目和归约项目不同时在一个项目集里。
- 如：$A\rightarrow\alpha N\beta \mid \alpha\gamma\delta\ N\rightarrow\gamma \mid \xi$
- 活前缀$\alpha\gamma$的有效项目集含有：$N\rightarrow\gamma\bullet$　　和 $A\rightarrow\alpha\gamma\bullet\delta$
- 那么，当$FIRST(\delta)\cap FOLLOW(N)=C\neq\varphi$时，可能存在移进-归约冲突，具体地，对于$tok\in C$可以移进也可以按$N\rightarrow\gamma$归约，如果$\delta$的首符集含有c的话。

- 修剪文法以消除此类冲突，
- $A\rightarrow\alpha\gamma\beta \mid \alpha\xi\beta \mid \alpha\gamma\delta$
- 此时，针对于活前缀$\alpha\gamma$的冲突不存在了。事实上推后到归约项目$A\rightarrow\alpha\gamma\beta\bullet$和$A\rightarrow\alpha\gamma\delta\bullet$了

SLR(1)局限性

- SLR(1)+冲突消解可处理几乎所有实用语言结构。
- 局限：
  - itemset={X →α•cβ, Y →α•bγ, A →α•, B →α•}
  - 无法满足{c,b}∩FOLLOW(A)∩FOLLOW(B)=φ

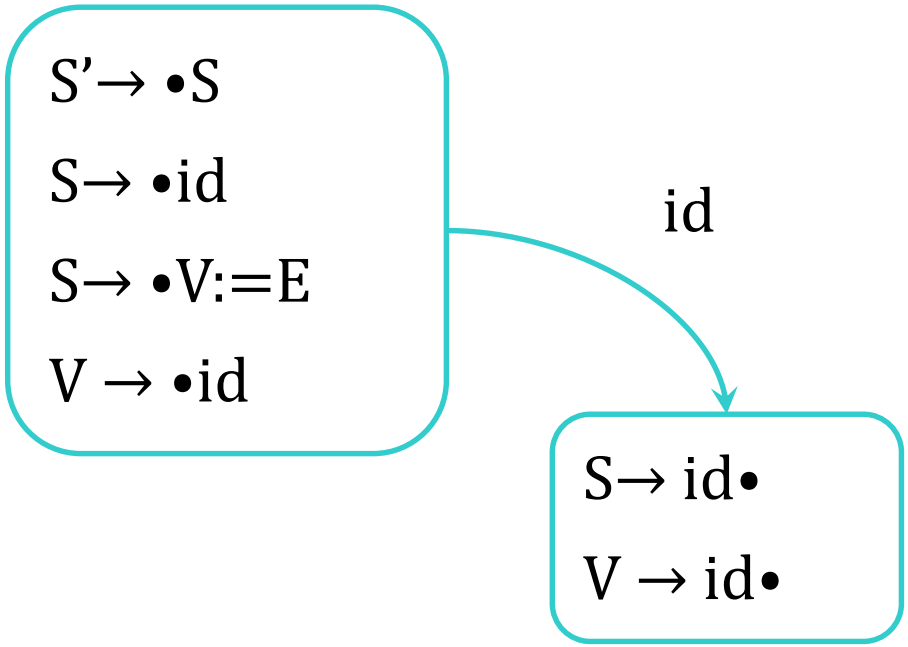
S → id|V:=E

V → id

E → V|n

Follow(S)={#}

Follow(V)={:=,#}



小结

- 规范句型，规范推导，规范归约，句柄，短语，直接短语；
- 活前缀，最大活前缀，有效项目，增广文法，LR(0)规范簇，规范集；
- 识别活前缀DFA,itemDFA；itemPDA, itemNFA,
- 移进-归约冲突，归约-归约冲突；冲突消解规则；
- LR(0)分析表，SLR(1)分析表；
- SLR(1)简单冲突冲突消解；
- SLR(1)其它三种冲突消解；
- 自下而上分析的算法描述。
- 作业 P187：习题8.1-8.5

思考

- LR(0)文法允许：
- ε-产生式
- 左递归变元
- 候选式有公共前缀的变元
- 单位产生式
- 无用符号
- 多候选式开始符号
- ε-产生式的项目是什么？既是完全项目又是初始项目
- 必须是增广文法

选作题：写出SLR(1)分析器

- P → Ď Š
- Ď → ε | Ď D ;
- D → T d | T d[i] | T d (Ă) {Ď Š}
- T → int | void
- Š → S | Š ; S
- S → d = E | if (B) S else S | while (B) S | return E | {Š} | d(Ě)
- Ă → ε | Ă A ;
- A → T d | T d[] | T d()
- Ě → ε | Ě E ,
- E → i | d | d[Ě] | d(Ě) | E op E
- B → E rop E | E

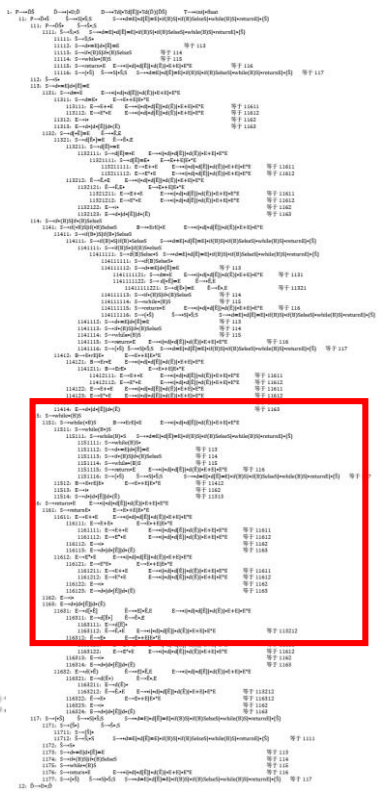
构造分析器示例

①写出识别活前缀的DFA;

②确定有哪些冲突并给出冲突消解规则;

③写出分析表及分析器代码,并测试完善。

```
11414: E→d•|d•[Ě]|d•(Ě)                                          等于 1163
115: S→while•(B)S
    1151: S→while(•B)S      B→•ErE|•E      E→•i|•d|•d[Ě]|•d(Ě)|•E+E|•E*E
        11511: S→while(B•)S
            115111: S→while(B)•S      S→•d=E|•d[Ě]=E|•if(B)S|•if(B)SelseS|•while(B)S|•returnE|•{Š}
                1151111: S→while(B)S•
                1151112: S→d•=E|d•[Ě]=E                          等于 113
                1151113: S→if•(B)S|if•(B)SelseS                  等于 114
                1151114: S→while•(B)S                            等于 115
                1151115: S→return•E      E→•i|•d|•d[Ě]|•d(Ě)|•E+E|•E*E      等于 116
                1151116: S→{•Š}      Š→•S|•Š;S      S→•d=E|•d[Ě]=E|•if(B)S|•if(B)SelseS|•while(B)S|•returnE|•{Š}      等于 117
        11512: B→E•rE|E•      E→E•+E|E•*E                        等于 11412
        11513: E→i•                                              等于 1162
        11514: S→d•|d•[Ě]|d•(Ě)                                  等于 11313
116: S→return•E      E→•i|•d|•d[Ě]|•d(Ě)|•E+E|•E*E
    1161: S→returnE•      E→E•+E|E•*E
        11611: E→E+•E      E→•i|•d|•d[Ě]|•d(Ě)|•E+E|•E*E
            116111: E→E+E•      E→E•+E|E•*E
                1161111: E→E+•E      E→•i|•d|•d[Ě]|•d(Ě)|•E+E|•E*E      等于 11611
                1161112: E→E*•E      E→•i|•d|•d[Ě]|•d(Ě)|•E+E|•E*E      等于 11612
            116112: E→i•                                         等于 1162
            116113: E→d•|d•[Ě]|d•(Ě)                             等于 1163
        11612: E→E*•E      E→•i|•d|•d[Ě]|•d(Ě)|•E+E|•E*E
            116121: E→E*E•      E→E•+E|E•*E
                1161211: E→E+•E      E→•i|•d|•d[Ě]|•d(Ě)|•E+E|•E*E      等于 11611
                1161212: E→E*•E      E→•i|•d|•d[Ě]|•d(Ě)|•E+E|•E*E      等于 11612
            116122: E→i•                                         等于 1162
            116123: E→d•|d•[Ě]|d•(Ě)                             等于 1163
    1162: E→i•
    1163: E→d•|d•[Ě]|d•(Ě)
        11631: E→d[•Ě]      Ě→•E|•Ě,E      E→•i|•d|•d[Ě]|•d(Ě)|•E+E|•E*E
            116311: E→d[Ě•]      Ě→Ě•,E
                1163111: E→d[Ě]•
                1163112: Ě→Ě,•E      E→•i|•d|•d[Ě]|•d(Ě)|•E+E|•E*E      等于 1132
```



关于SLR(1)的结论

- 能消解的冲突都已经有了消解规则，并已经体现在分析表中
  - SLR(1)默认的冲突及消解；
  - 优先级冲突；悬挂else冲突；运算次序冲突；
  - 文法中没有变元并列的候选式。
- SLR(1)分析表已经消解了所有能消解的冲突。
- 仍然存在不为SLR(1)能消解的冲突，但已经充分了对于PL而言？

基于分析表的SLR(1)分析算法

▻ 输入：分析表，w，双栈空栈。输出：分析成功与否。

```
bi-stack(1,#);
scan();
while(1){
   bi-top(m,c);
   if (ACTION[m,tok]==sn)bi-push(n,tok);
   else if(ACTION[m,tok]==rk){
          bi-pop(body-size(k));
          A=head(k);
          bi-top(m',c); bi-push(GOTO[m',A],A);
    }else if(ACTION[m,tok]==acc)break;
    else error();
}//与LR(0)的没有区别
```