MIT 6.1100
Introduction to Shift-Reduce Parsing

Martin Rinard
Massachusetts Institute of Technology

Orientation

- Specify Syntax Using Context-Free Grammar
  - Nonterminals
  - Terminals
  - Productions
- Given a grammar, Parser Generator produces a parser
  - Starts with input string
  - Produces parse tree

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow -\ Expr$

$Expr \rightarrow \mathrm{num}$

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

Today's Lecture

- How generated parser works
- How parser generator produces parser
- Central mechanism
  - Pushdown automaton, which implements
  - Shift-reduce parser

Pushdown Automata

- Consists of
  - Pushdown stack (can have terminals and nonterminals)
  - Finite state automaton control
- Can do one of three actions (based on state and input):
  - Shift:
    - Shift current input symbol from input onto stack
  - Reduce:
    - If symbols on top of stack match right hand side of some grammar production NT $\rightarrow \beta$
    - Pop symbols (β) off of the stack
    - Push left hand side nonterminal (NT) onto stack
  - Accept the input string

Shift-Reduce Parser Example

(1) *Expr* $\rightarrow$ *Expr Op Expr*
(2) *Expr* $\rightarrow$ (*Expr*)
(3) *Expr* $\rightarrow$ - *Expr*
(4) *Expr* $\rightarrow$ num
(5) *Op* $\rightarrow$ +
(6) *Op* $\rightarrow$ -
(7) *Op* $\rightarrow$ *

Stack

Input String

num	*	(	num	+	num	)

Shift-Reduce Parser Example

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow (Expr)$

(3) $Expr \rightarrow -\ Expr$

(4) $Expr \rightarrow$ num

(5) $Op \rightarrow +$

(6) $Op \rightarrow -$

(7) $Op \rightarrow *$

num	*	(	num	+	num	)

Shift-Reduce Parser Example

(1) *Expr* → *Expr Op Expr*

(2) *Expr* → (*Expr*)

(3) *Expr* → - *Expr*

(4) *Expr* → num

(5) *Op* → +

(6) *Op* → -

(7) *Op* → *

SHIFT

num	*	(	num	+	num	)

Shift-Reduce Parser Example

(1) *Expr* $\rightarrow$ *Expr Op Expr*

(2) *Expr* $\rightarrow$ (*Expr*)

(3) *Expr* $\rightarrow$ - *Expr*

(4) *Expr* $\rightarrow$ num

(5) *Op* $\rightarrow$ +

(6) *Op* $\rightarrow$ -

(7) *Op* $\rightarrow$ *

num

SHIFT

*	(	num	+	num	)

Shift-Reduce Parser Example

(1) *Expr* $\rightarrow$ *Expr Op Expr*

(2) *Expr* $\rightarrow$ (*Expr*)

(3) *Expr* $\rightarrow$ - *Expr*

(4) $\boxed{Expr \rightarrow \text{num}}$

(5) *Op* $\rightarrow$ +

(6) *Op* $\rightarrow$ -

(7) *Op* $\rightarrow$ *

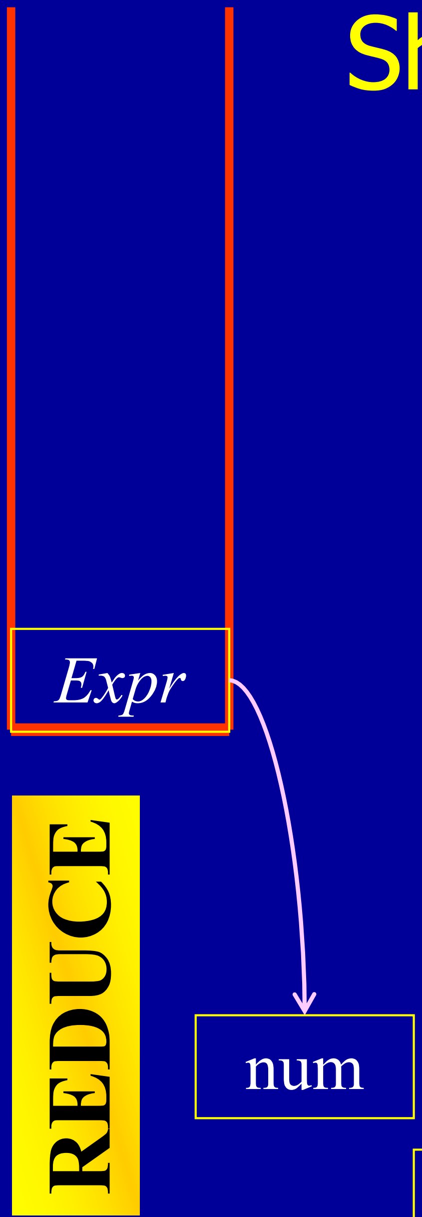
num

REDUCE

*	(	num	+	num	)

Shift-Reduce Parser Example

(1) *Expr* → *Expr Op Expr*
(2) *Expr* → (*Expr*)
(3) *Expr* → - *Expr*
(4) $\boxed{\textit{Expr} \rightarrow \text{num}}$
(5) *Op* → +
(6) *Op* → -
(7) *Op* → *



*	(	num	+	num	)

Shift-Reduce Parser Example

(1) *Expr* $\rightarrow$ *Expr Op Expr*
(2) *Expr* $\rightarrow$ (*Expr*)
(3) *Expr* $\rightarrow$ - *Expr*
(4) *Expr* $\rightarrow$ num
(5) *Op* $\rightarrow$ +
(6) *Op* $\rightarrow$ -
(7) *Op* $\rightarrow$ *

Expr

SHIFT

num

*	(	num	+	num	)

Shift-Reduce Parser Example

(1) *Expr* → *Expr Op Expr*

(2) *Expr* → (*Expr*)

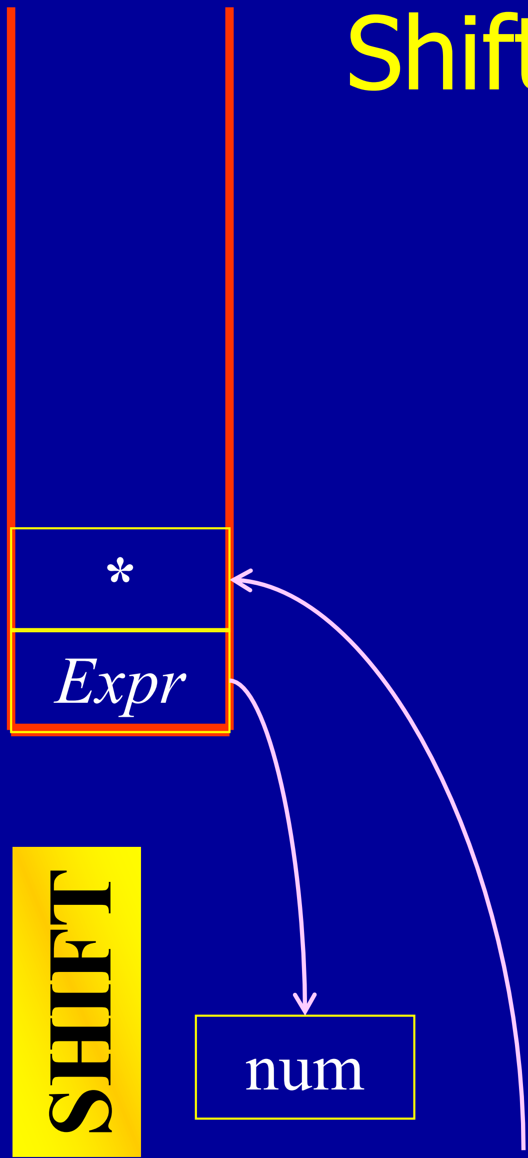
(3) *Expr* → - *Expr*

(4) *Expr* → num

(5) *Op* → +

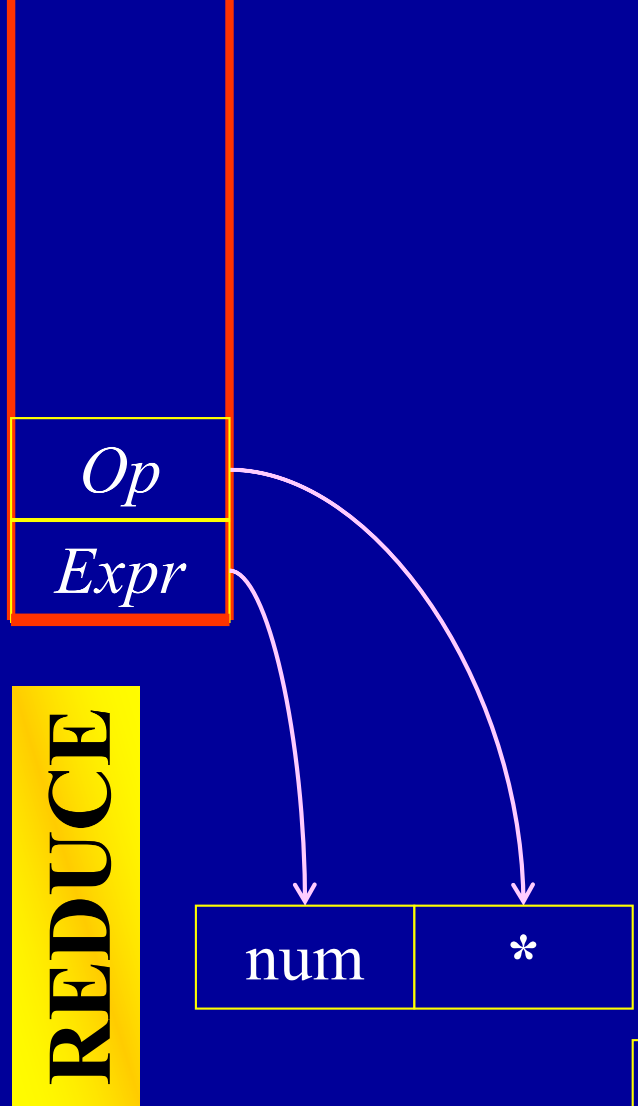
(6) *Op* → -

(7) *Op* → *



(	num	+	num	)

Shift-Reduce Parser Example



(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow (Expr)$

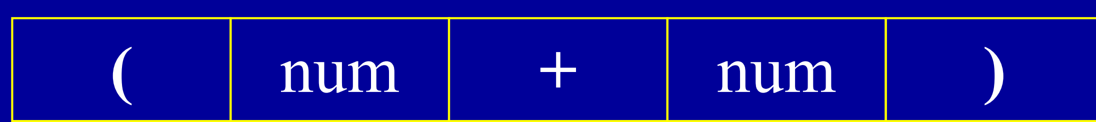
(3) $Expr \rightarrow -\ Expr$

(4) $Expr \rightarrow \text{num}$

(5) $Op \rightarrow +$

(6) $Op \rightarrow -$

(7) $\boxed{Op \rightarrow *}$



Shift-Reduce Parser Example

(1) *Expr* $\rightarrow$ *Expr Op Expr*

(2) *Expr* $\rightarrow$ (*Expr*)

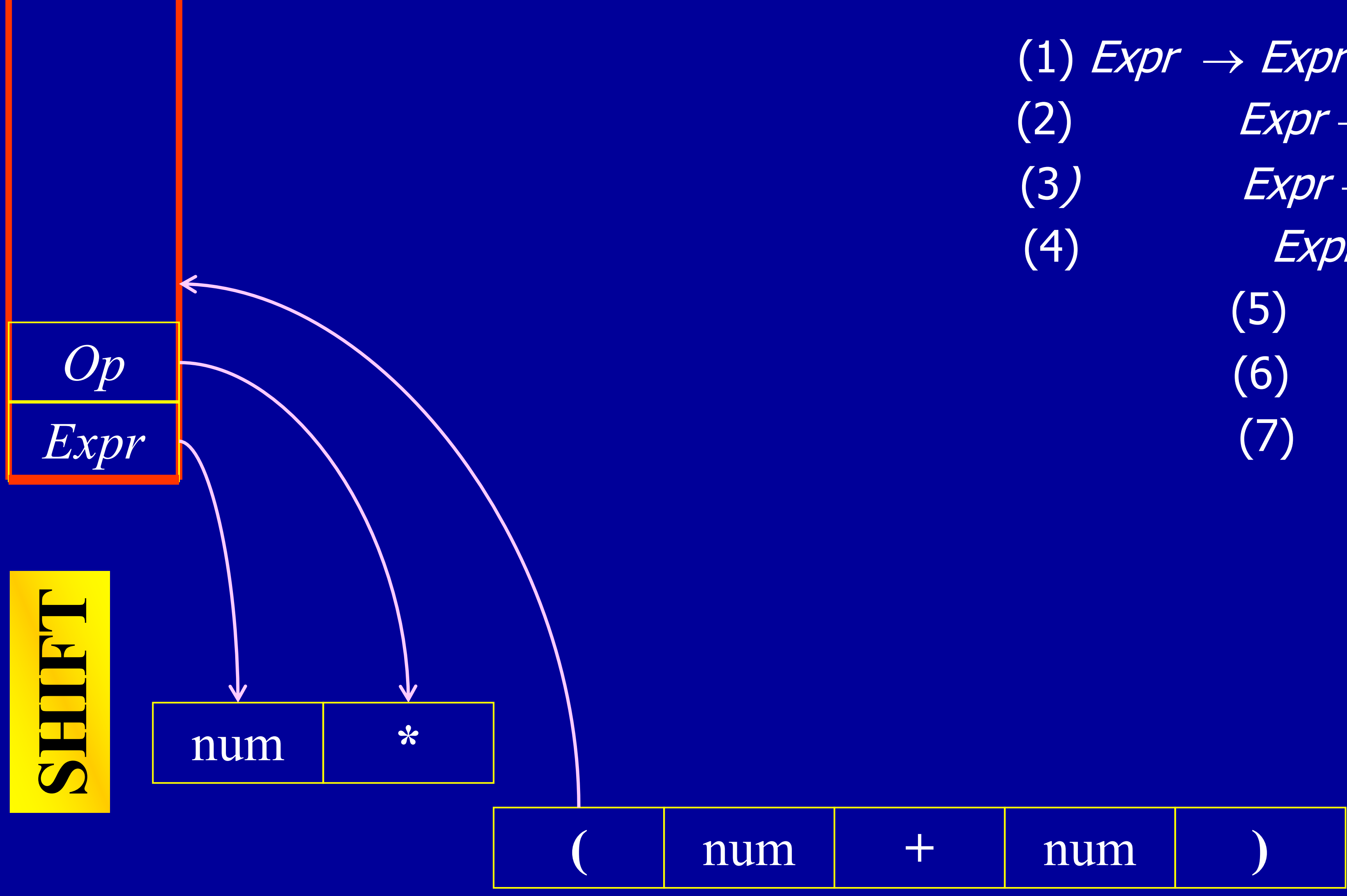
(3) *Expr* $\rightarrow$ - *Expr*

(4) *Expr* $\rightarrow$ num

(5) *Op* $\rightarrow$ +

(6) *Op* $\rightarrow$ -

(7) *Op* $\rightarrow$ *



Shift-Reduce Parser Example

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow (Expr)$

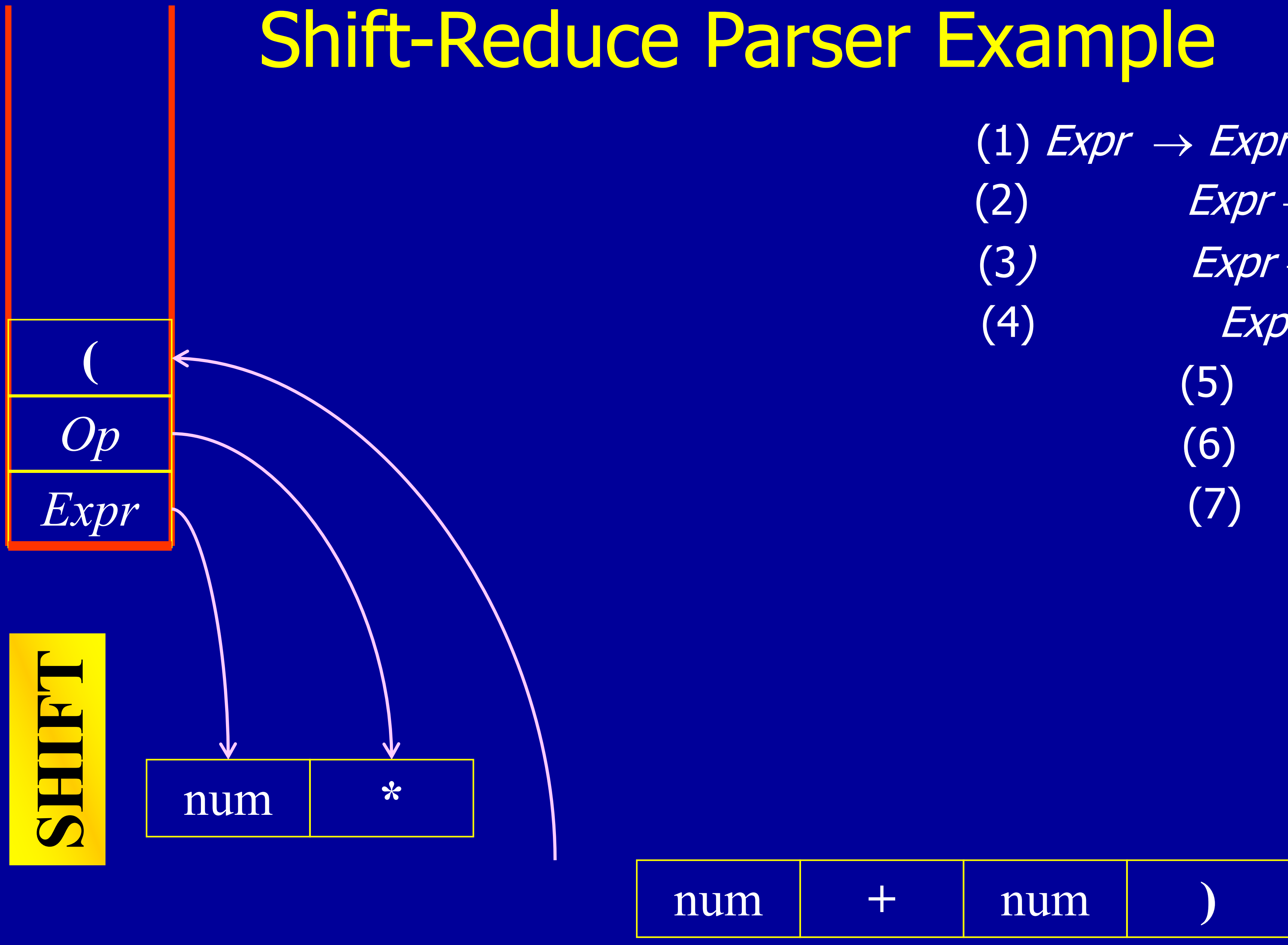
(3) $Expr \rightarrow -\ Expr$

(4) $Expr \rightarrow$ num

(5) $Op \rightarrow +$

(6) $Op \rightarrow -$

(7) $Op \rightarrow *$



Shift-Reduce Parser Example

(1) *Expr* → *Expr Op Expr*

(2) *Expr* → (*Expr*)

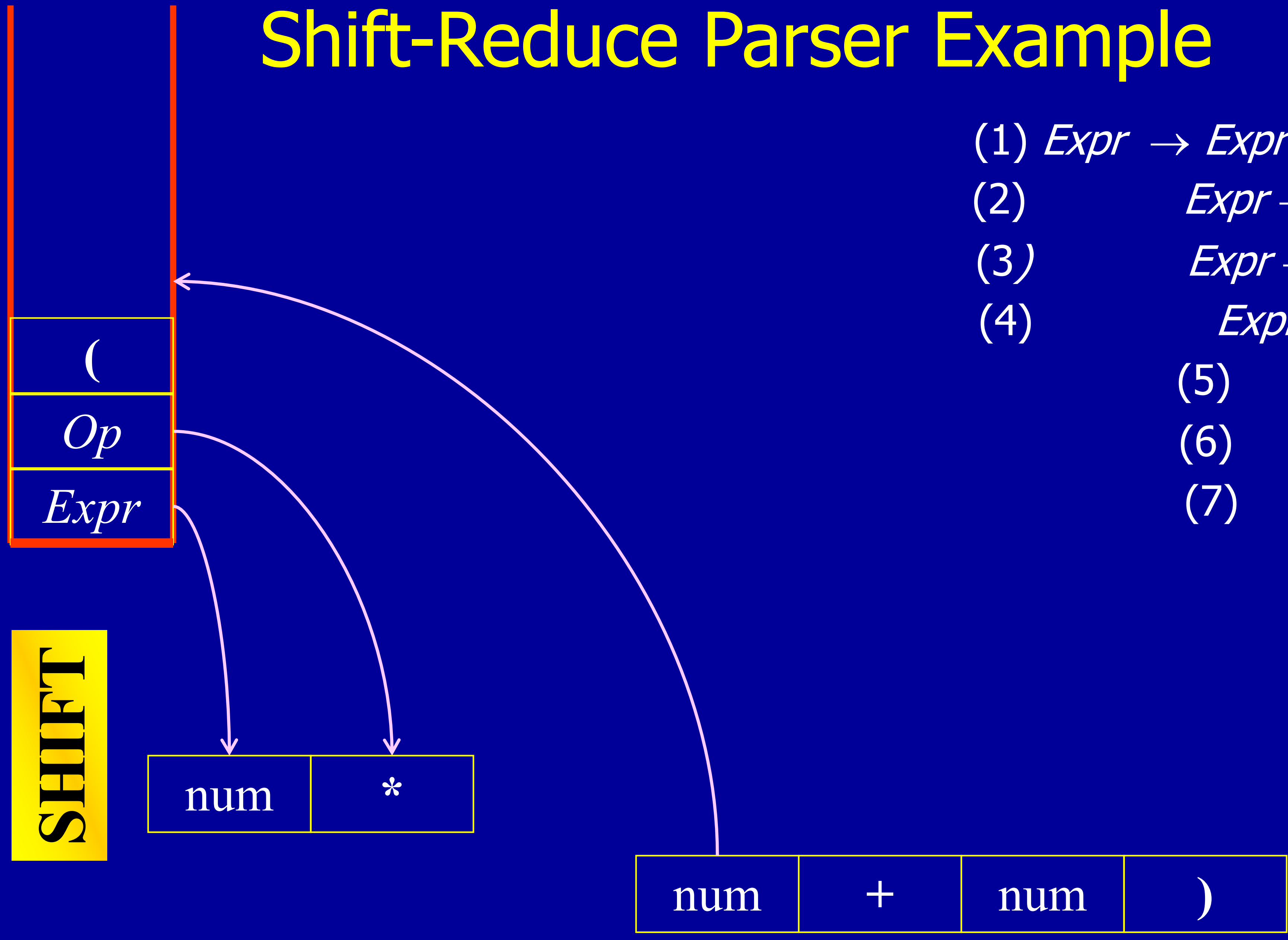
(3) *Expr* → - *Expr*

(4) *Expr* → num

(5) *Op* → +

(6) *Op* → -

(7) *Op* → *



Shift-Reduce Parser Example

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow (Expr)$

(3) $Expr \rightarrow -\ Expr$

(4) $Expr \rightarrow$ num

(5) $Op \rightarrow +$

(6) $Op \rightarrow -$

(7) $Op \rightarrow *$

num
(
Op
Expr

SHIFT

num	*

+	num	)

Shift-Reduce Parser Example

(1) *Expr* → *Expr Op Expr*

(2) *Expr* → (*Expr*)

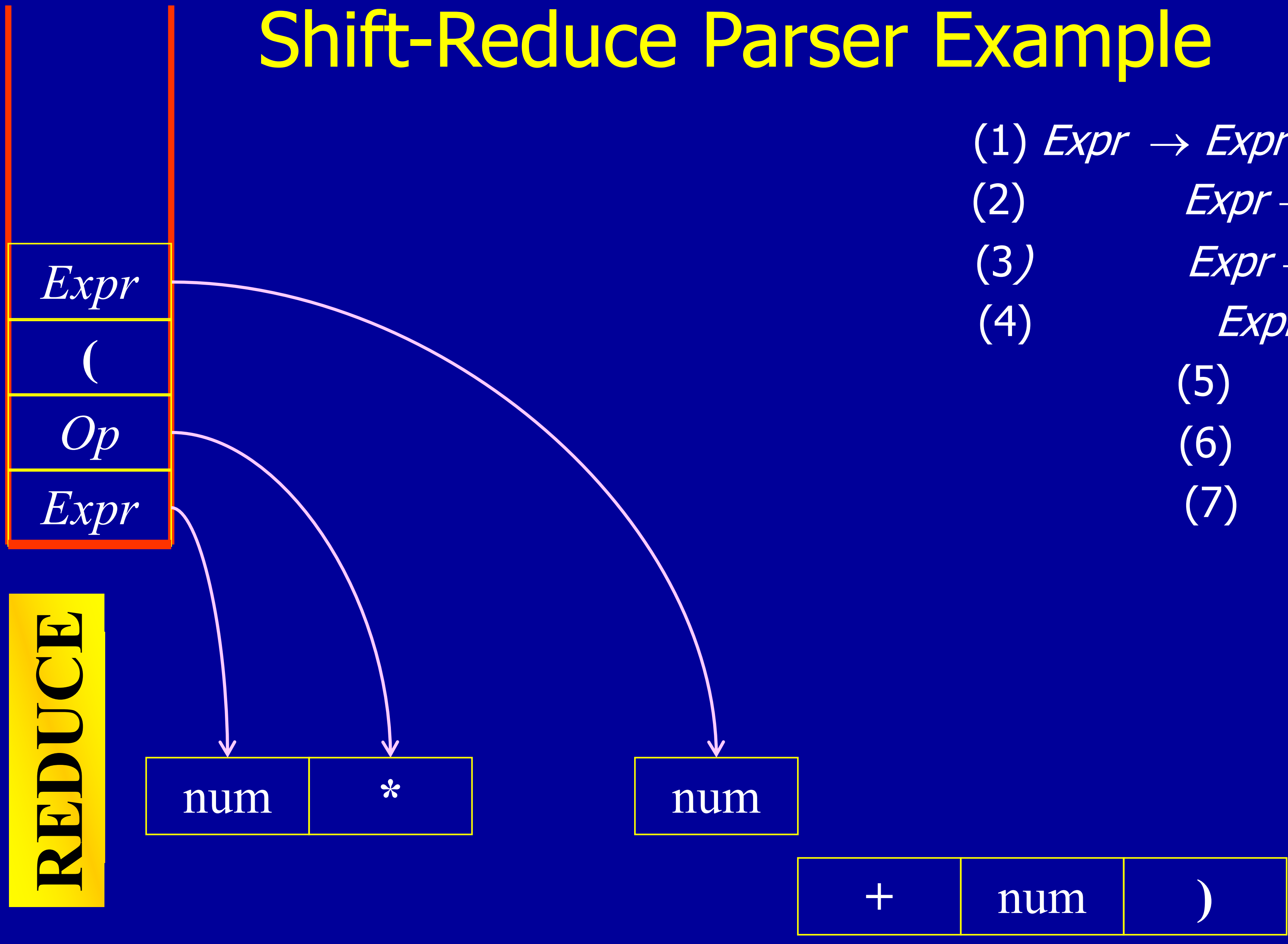
(3) *Expr* → - *Expr*

(4) *Expr* → num

(5) *Op* → +

(6) *Op* → -

(7) *Op* → *



Shift-Reduce Parser Example

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow (\ Expr\ )$

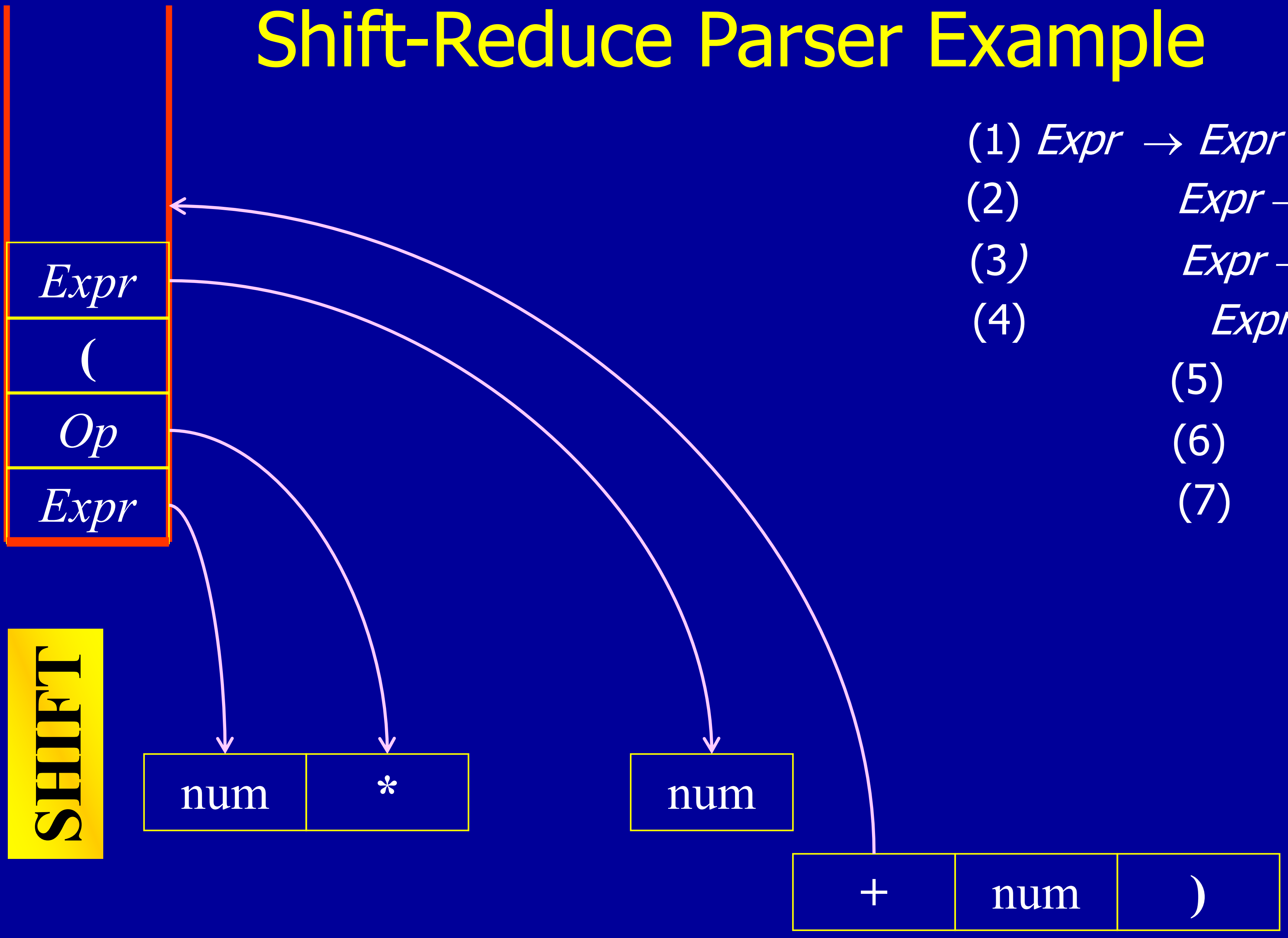
(3) $Expr \rightarrow -\ Expr$

(4) $Expr \rightarrow \text{num}$

(5) $Op \rightarrow +$

(6) $Op \rightarrow -$

(7) $Op \rightarrow *$



Shift-Reduce Parser Example

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow (\ Expr\ )$

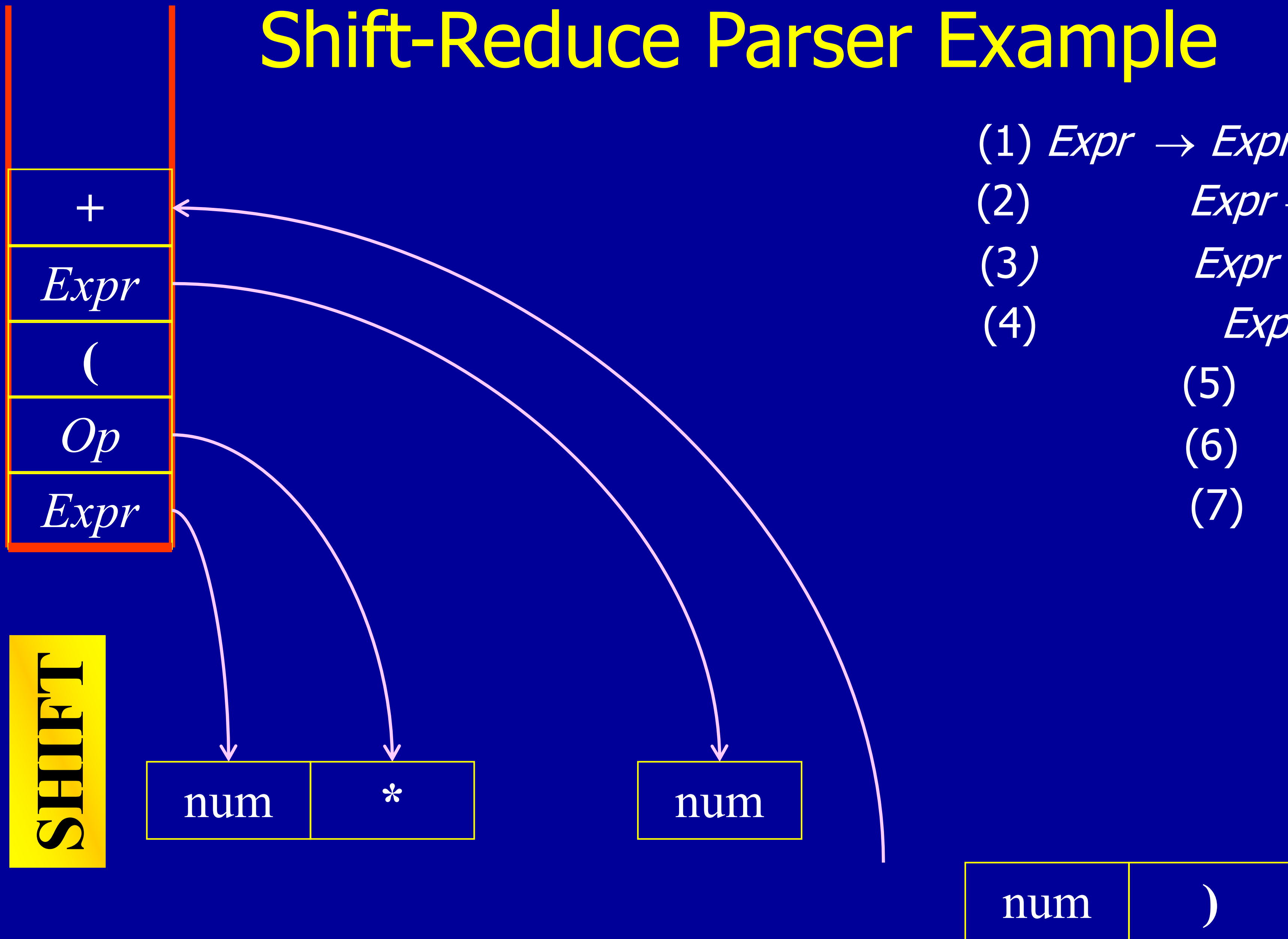
(3) $Expr \rightarrow -\ Expr$

(4) $Expr \rightarrow$ num

(5) $Op \rightarrow +$

(6) $Op \rightarrow -$

(7) $Op \rightarrow *$



Shift-Reduce Parser Example

(1) *Expr* $\rightarrow$ *Expr Op Expr*

(2) *Expr* $\rightarrow$ (*Expr*)

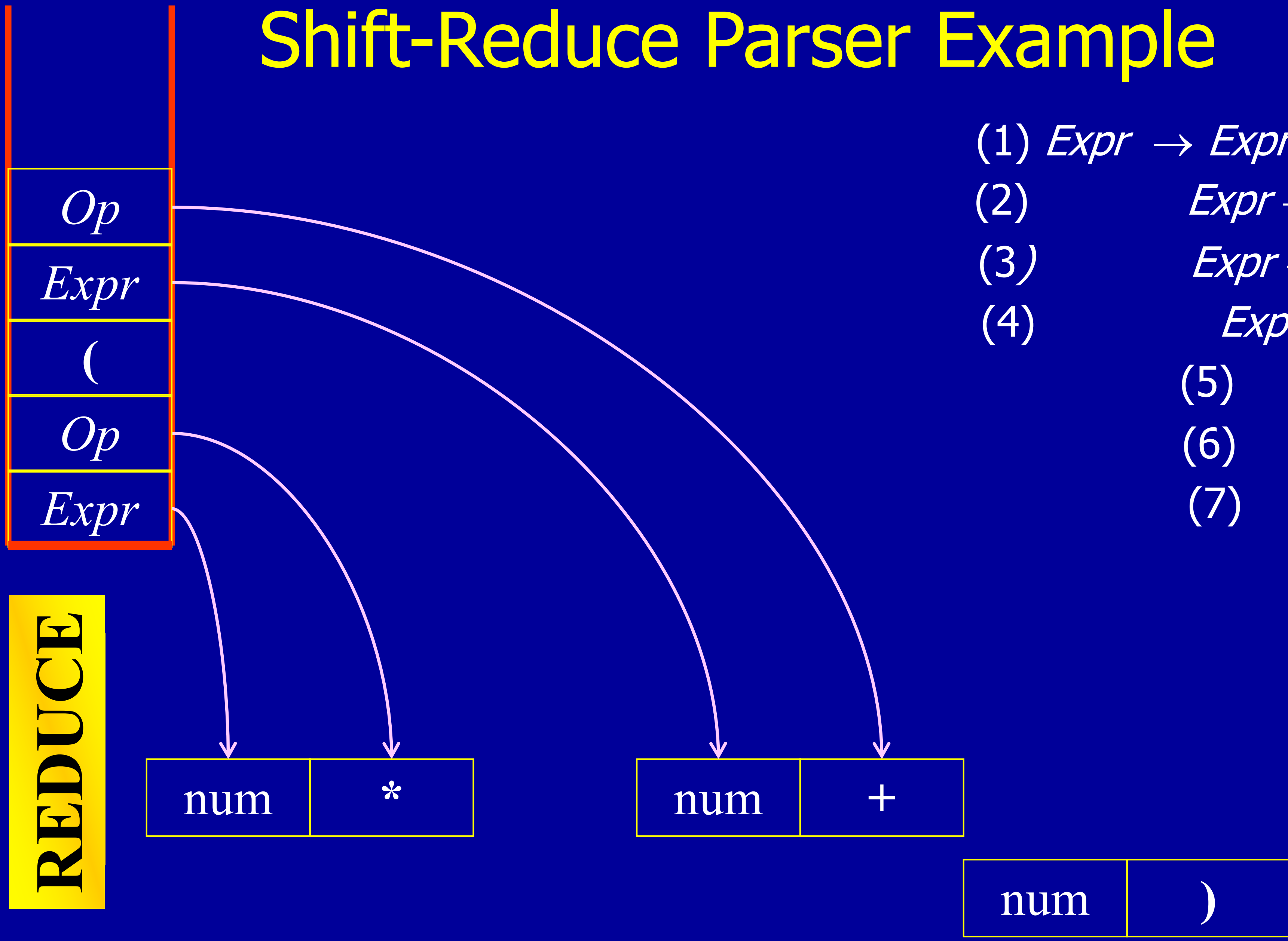
(3) *Expr* $\rightarrow$ - *Expr*

(4) *Expr* $\rightarrow$ num

(5) *Op* $\rightarrow$ +

(6) *Op* $\rightarrow$ -

(7) *Op* $\rightarrow$ *



Shift-Reduce Parser Example

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow$ ($Expr$)

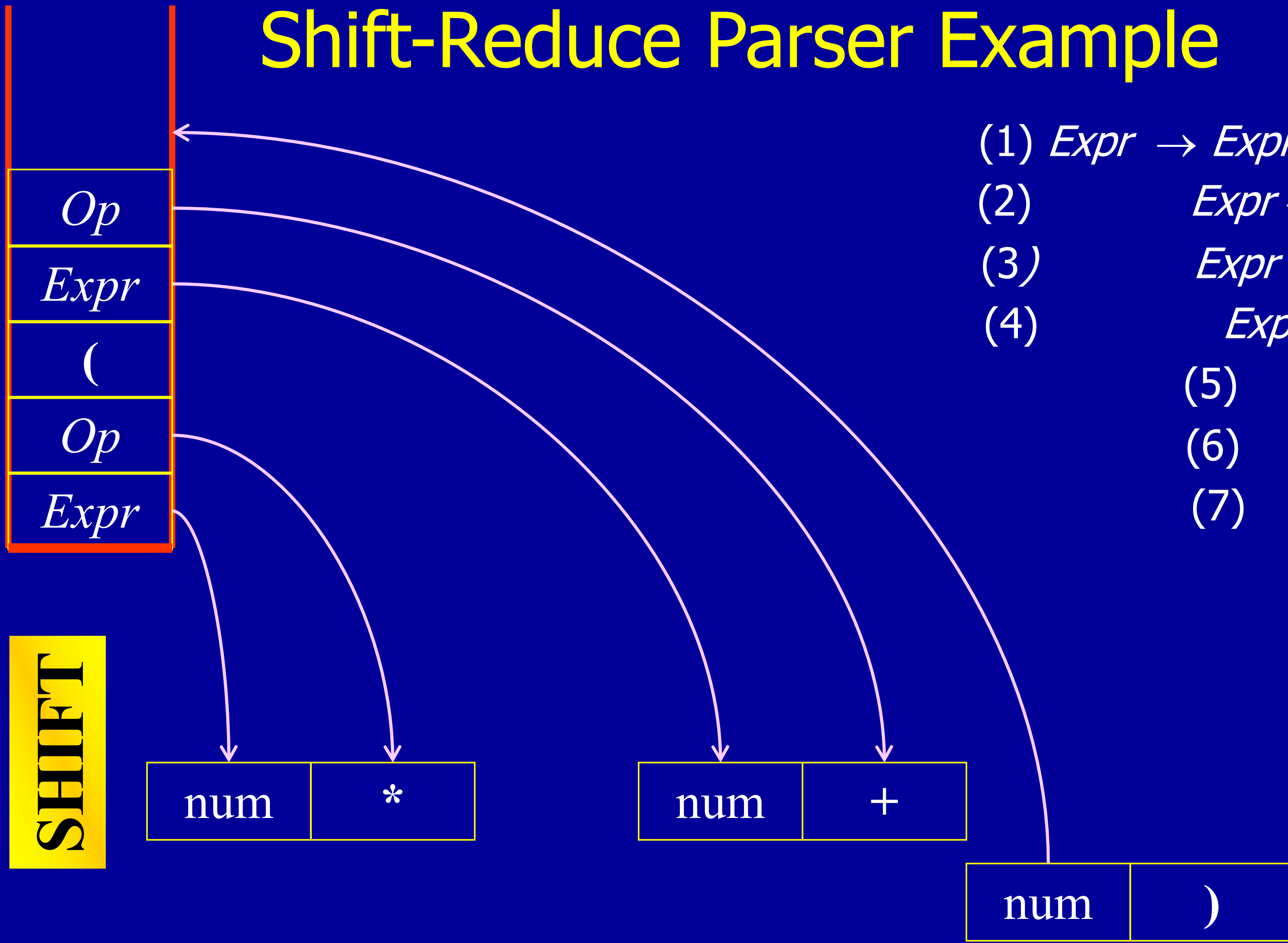
(3) $Expr \rightarrow$ - $Expr$

(4) $Expr \rightarrow$ num

(5) $Op \rightarrow$ +

(6) $Op \rightarrow$ -

(7) $Op \rightarrow$ *



Shift-Reduce Parser Example

(1) *Expr* → *Expr Op Expr*

(2) *Expr* → (*Expr*)

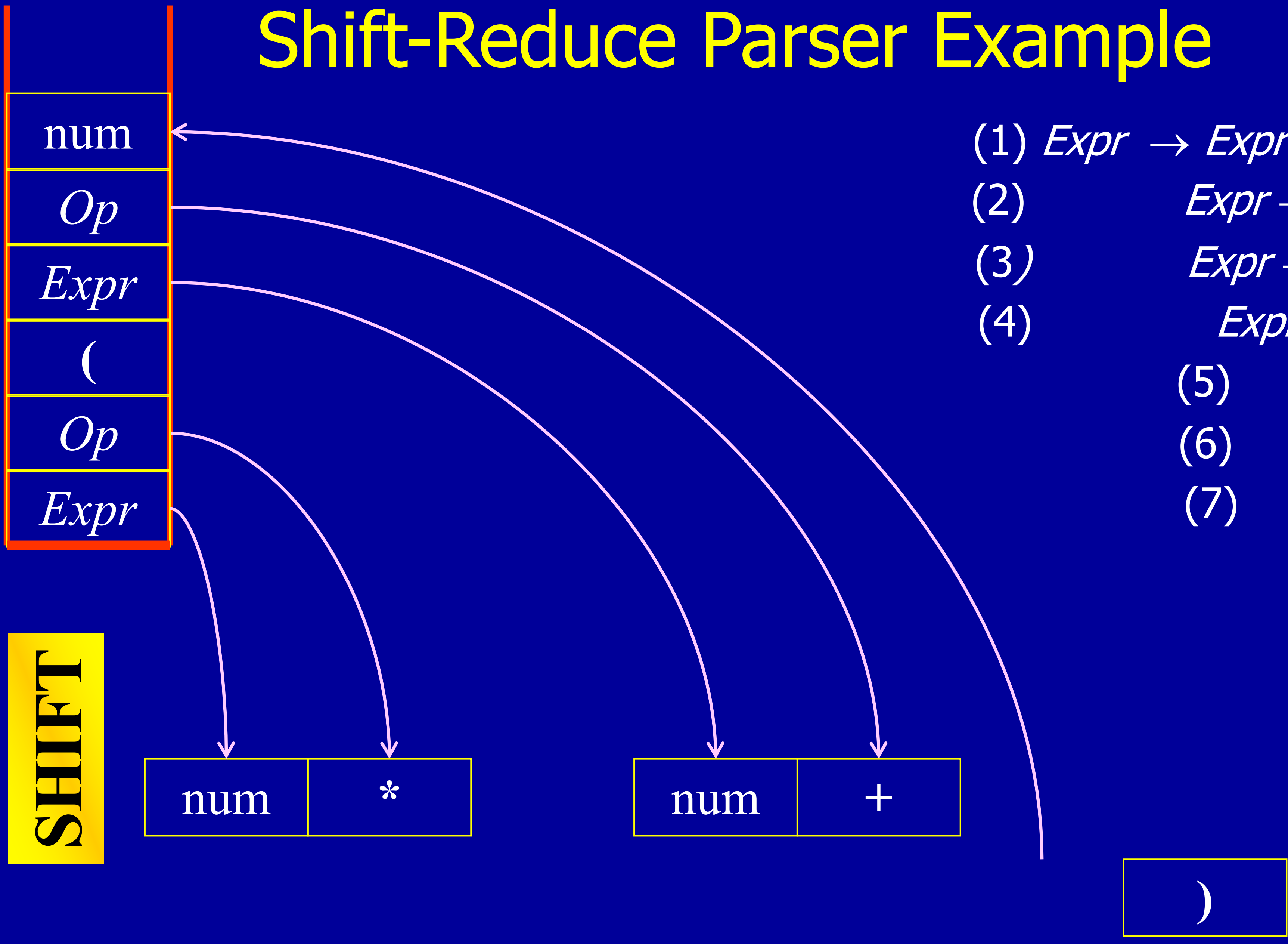
(3) *Expr* → - *Expr*

(4) *Expr* → num

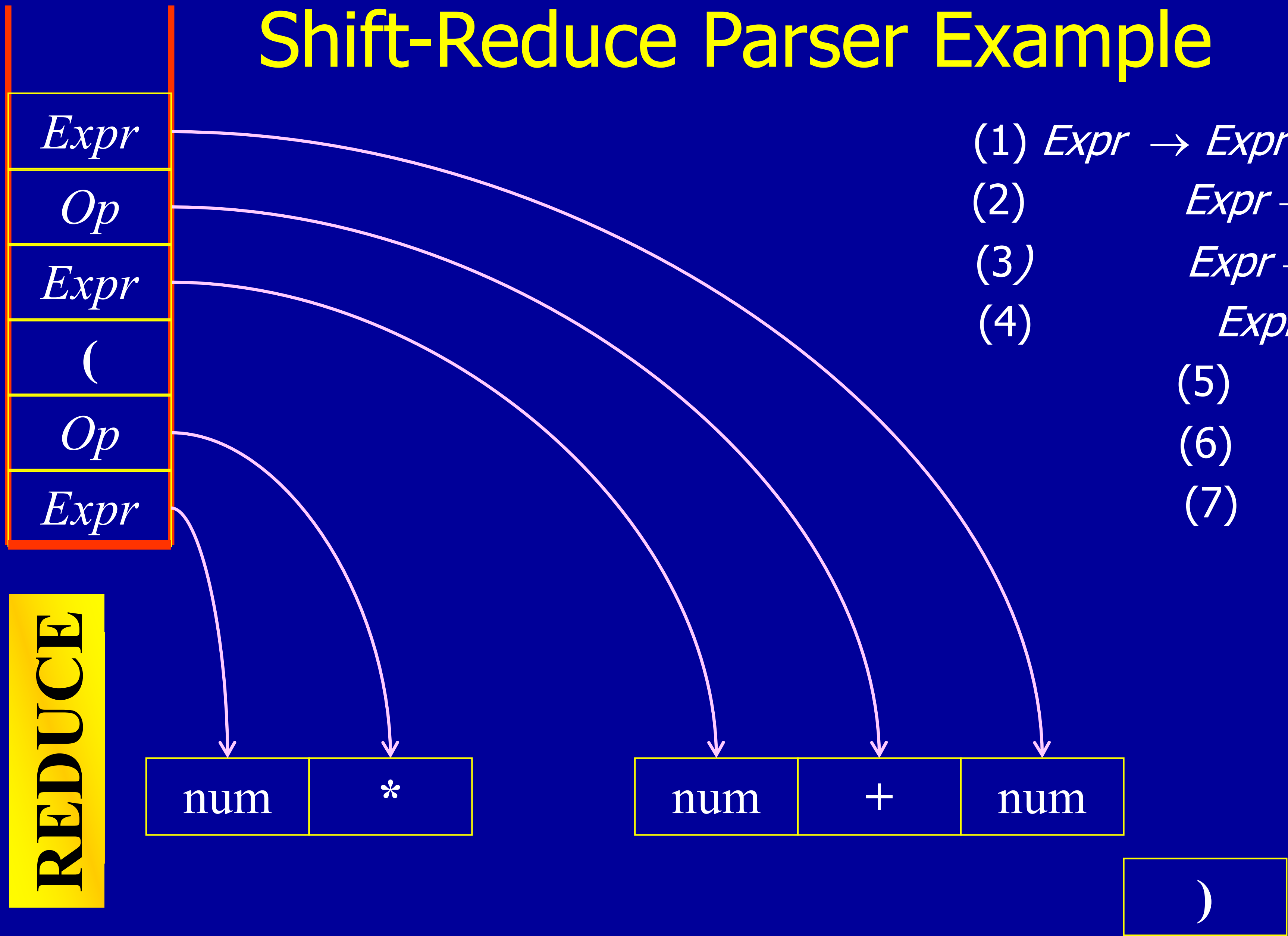
(5) *Op* → +

(6) *Op* → -

(7) *Op* → *



Shift-Reduce Parser Example



(1) $Expr \rightarrow Expr\ Op\ \ Expr$

(2) $Expr \rightarrow$ ($Expr$)

(3) $Expr \rightarrow$ - $Expr$

(4) $Expr \rightarrow$ num

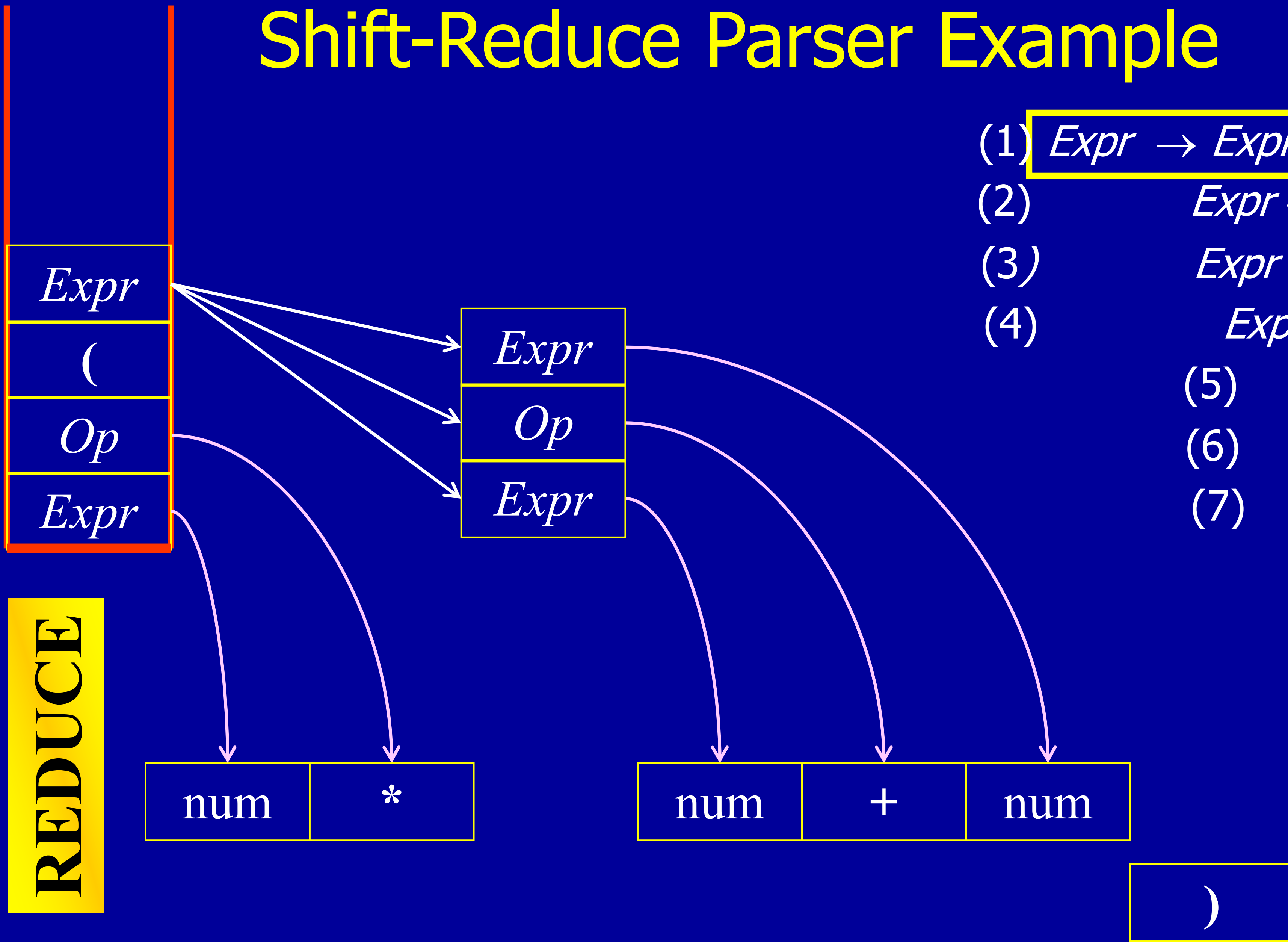
(5) $Op \rightarrow$ +

(6) $Op \rightarrow$ -

(7) $Op \rightarrow$ *

Shift-Reduce Parser Example

(1) $\boxed{Expr \rightarrow Expr\ Op\ \ Expr}$
(2) $Expr \rightarrow (Expr)$
(3) $Expr \rightarrow -\ Expr$
(4) $Expr \rightarrow$ num
(5) $Op \rightarrow +$
(6) $Op \rightarrow -$
(7) $Op \rightarrow *$



Shift-Reduce Parser Example

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow (Expr)$

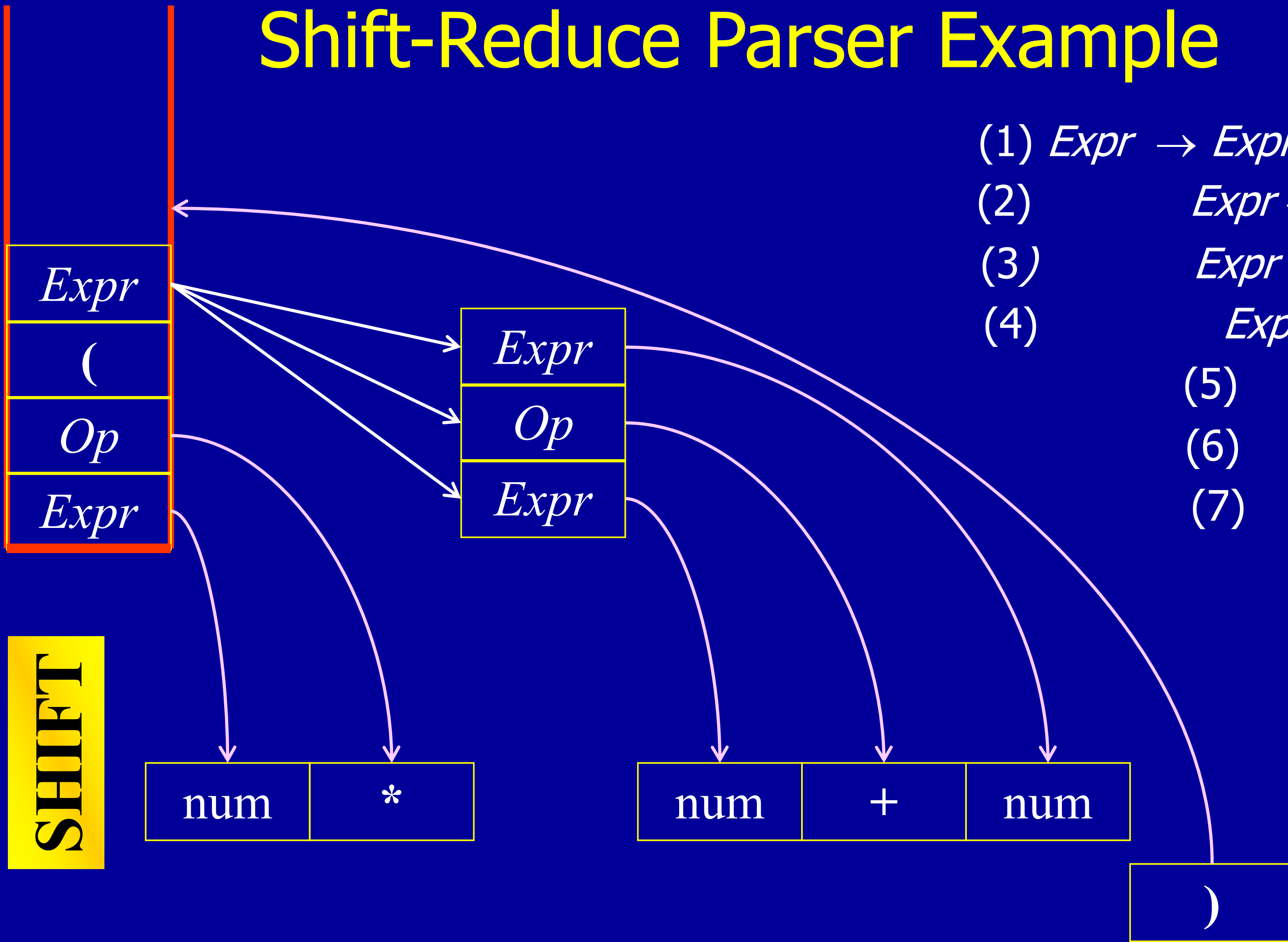
(3) $Expr \rightarrow -\ Expr$

(4) $Expr \rightarrow \text{num}$

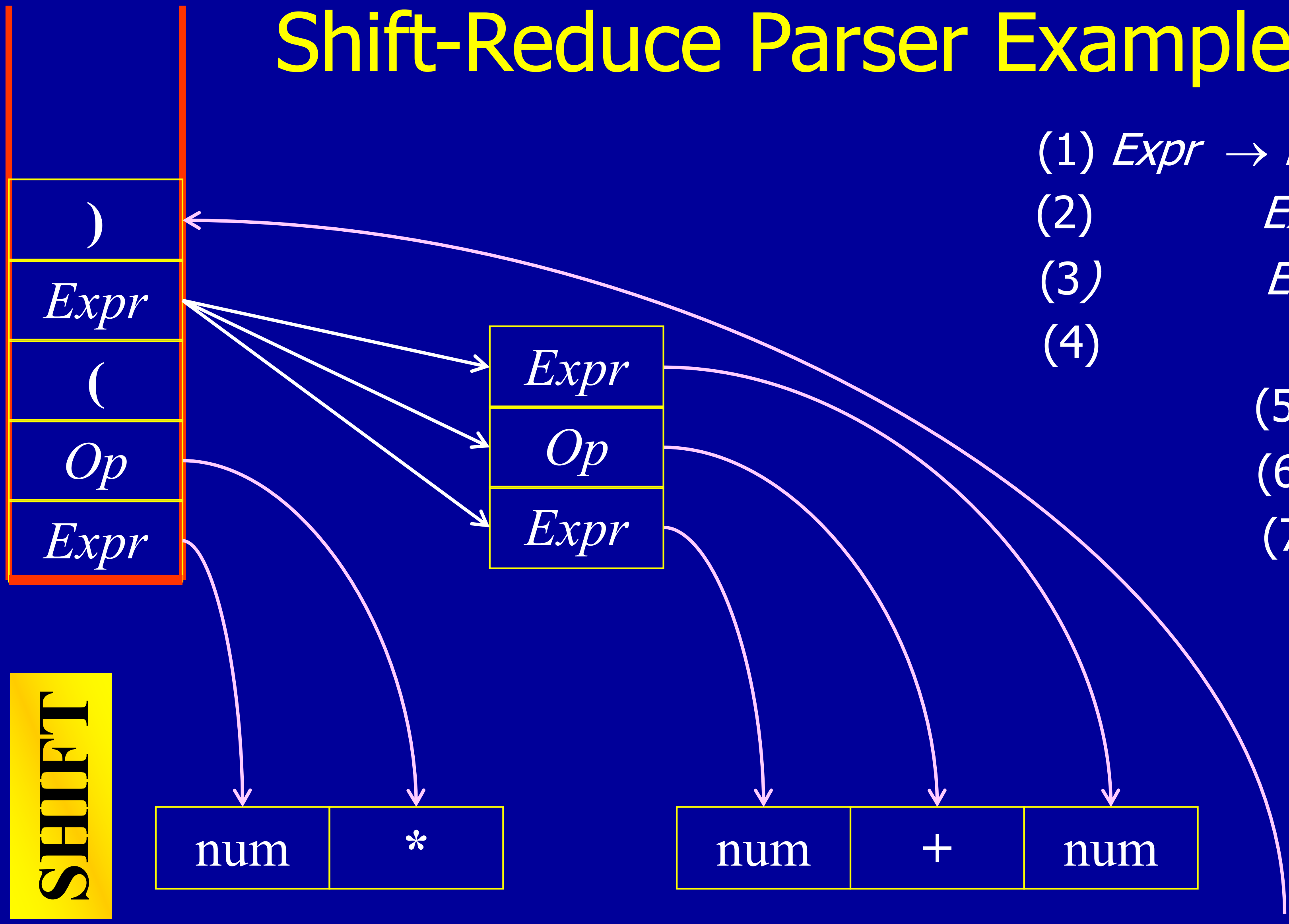
(5) $Op \rightarrow +$

(6) $Op \rightarrow -$

(7) $Op \rightarrow *$



Shift-Reduce Parser Example



(1) $Expr \rightarrow Expr\ Op\ \ Expr$

(2) $Expr \rightarrow (\,Expr\,)$

(3) $Expr \rightarrow -\ Expr$

(4) $Expr \rightarrow$ num

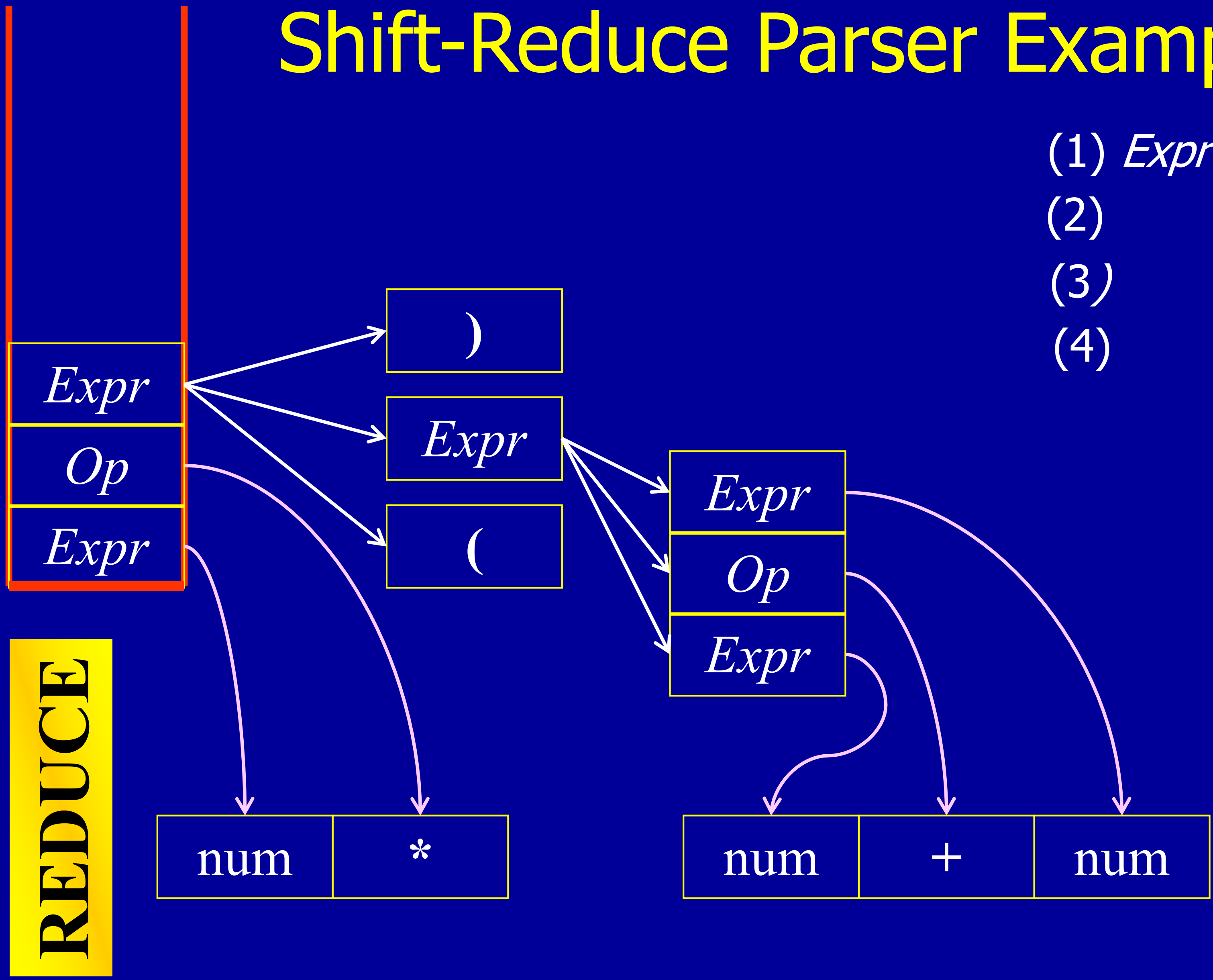
(5) $Op \rightarrow +$

(6) $Op \rightarrow -$

(7) $Op \rightarrow *$

Shift-Reduce Parser Example

(1) *Expr* → *Expr Op Expr*
(2) $\boxed{Expr \rightarrow (Expr)}$
(3) *Expr* → - *Expr*
(4) *Expr* → num
(5) *Op* → +
(6) *Op* → -
(7) *Op* → *



Shift-Reduce Parser Example

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow (Expr)$

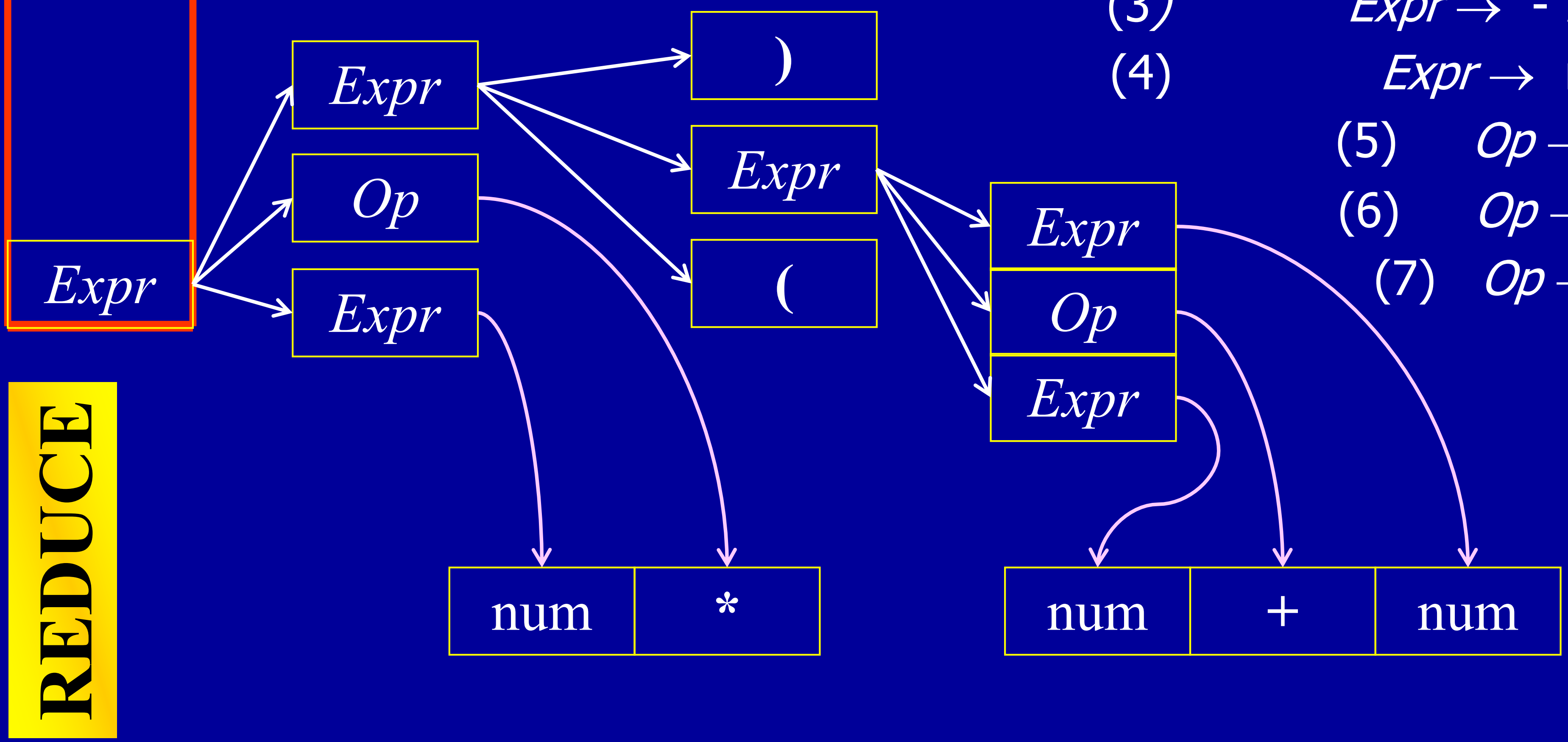
(3) $Expr \rightarrow -\ Expr$

(4) $Expr \rightarrow$ num

(5) $Op \rightarrow +$

(6) $Op \rightarrow -$

(7) $Op \rightarrow *$



Shift-Reduce Parser Example

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow (Expr)$

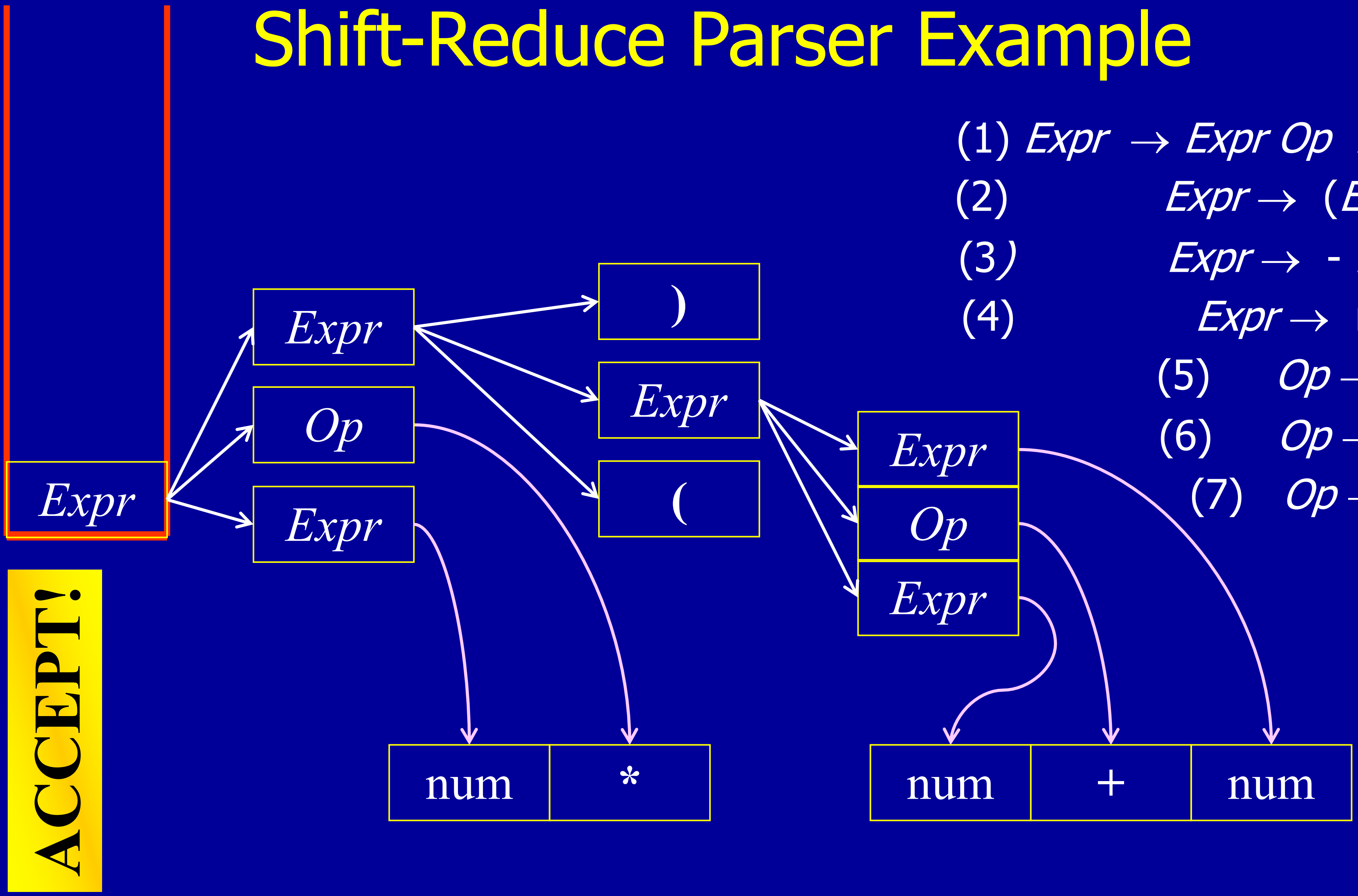
(3) $Expr \rightarrow -\ Expr$

(4) $Expr \rightarrow$ num

(5) $Op \rightarrow +$

(6) $Op \rightarrow -$

(7) $Op \rightarrow *$



Basic Idea

- Goal: reconstruct parse tree for input string
- Read input from left to right
- Build tree in a bottom-up fashion
- Use stack to hold pending sequences of terminals and nonterminals

Potential Conflicts

- Reduce/Reduce Conflict
  - Top of the stack may match RHS of multiple productions
  - Which production to use in the reduction?
- Shift/Reduce Conflict
  - Stack may match RHS of production
  - But that may not be the right match
  - May need to shift an input and later find a different reduction

Conflicts

- Original Grammar

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

- New Grammar

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr - Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow Expr -$

$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

Conflicts

(1) $Expr \rightarrow Expr\ Op\ Expr$
(2) $Expr \rightarrow Expr$ - $Expr$
(3) $Expr \rightarrow (Expr)$
(4) $Expr \rightarrow Expr$ -
(5) $Expr \rightarrow$ num
(6) $Op \rightarrow$ +
(7) $Op \rightarrow$ -
(8) $Op \rightarrow$ *

num	-	num

Conflicts

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow Expr$ - $Expr$

(3) $Expr \rightarrow$ ($Expr$)

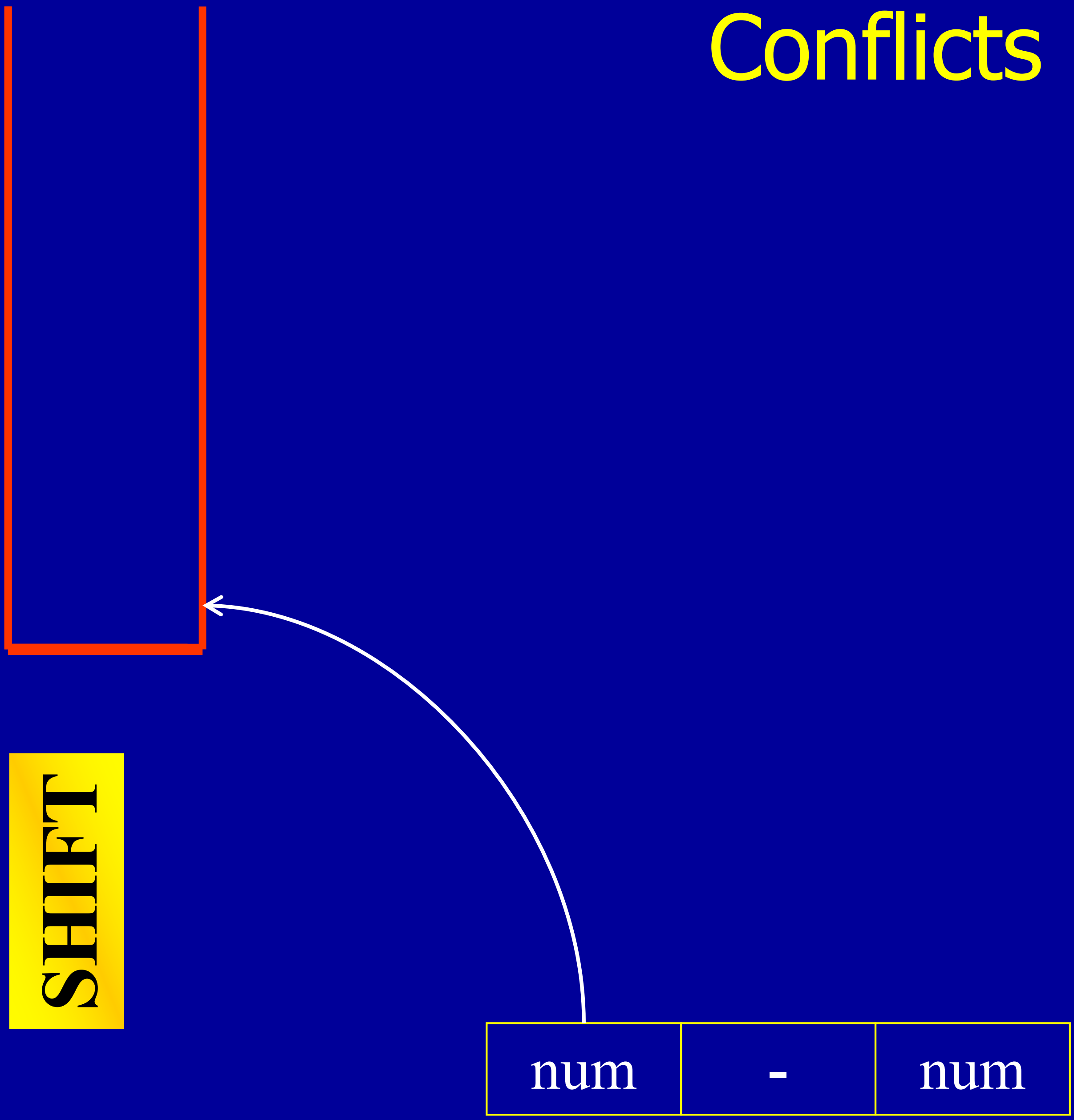
(4) $Expr \rightarrow Expr$ -

(5) $Expr \rightarrow$ num

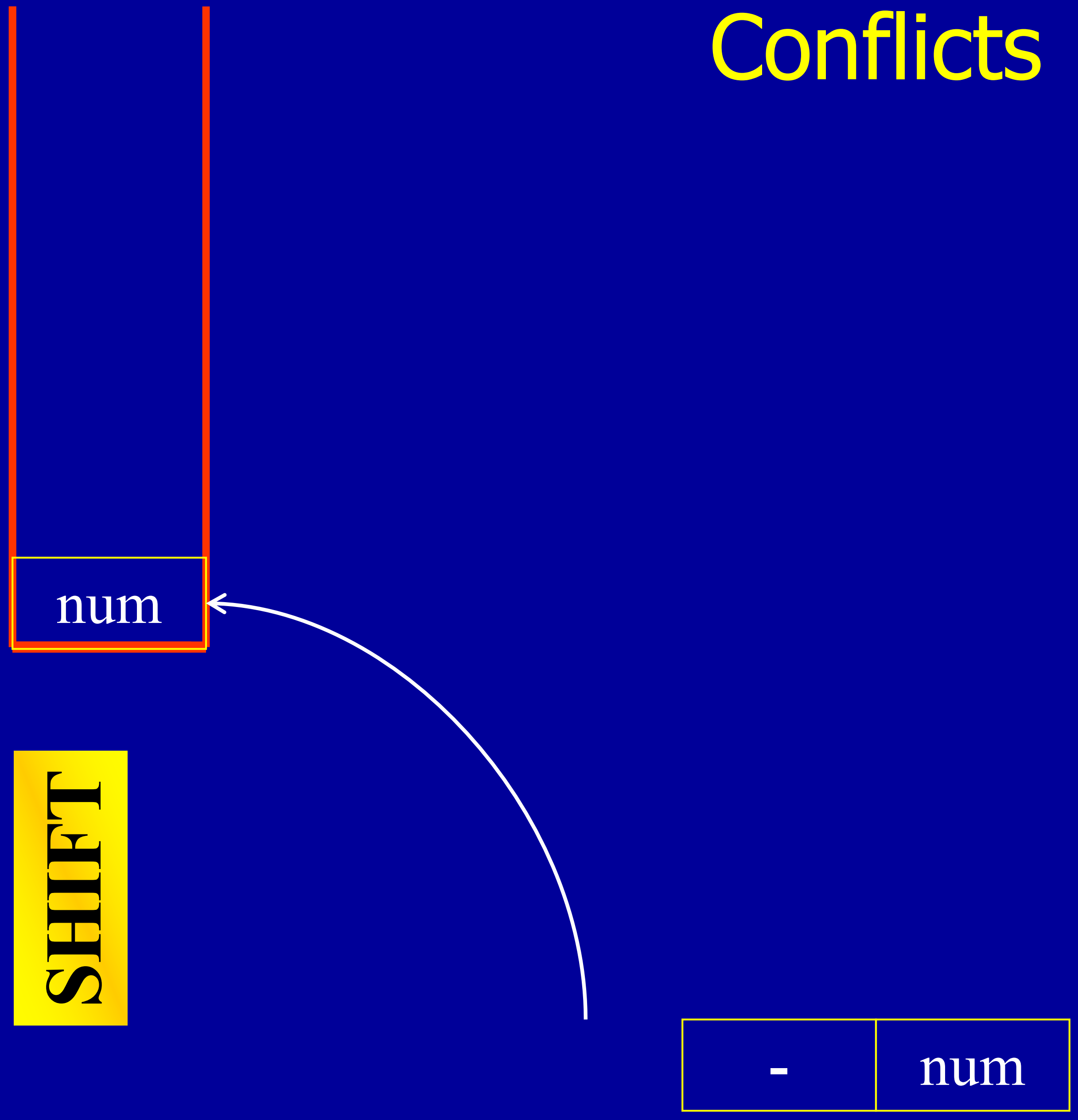
(6) $Op \rightarrow$ +

(7) $Op \rightarrow$ -

(8) $Op \rightarrow$ *



Conflicts



(1) *Expr* → *Expr Op Expr*

(2) *Expr* → *Expr* - *Expr*

(3) *Expr* → (*Expr*)

(4) *Expr* → *Expr* -

(5) *Expr* → num

(6) *Op* → +

(7) *Op* → -

(8) *Op* → *

Conflicts

(1) $Expr \rightarrow Expr\ Op\ \ Expr$

(2) $Expr \rightarrow Expr$ - $Expr$

(3) $Expr \rightarrow$ ($Expr$)

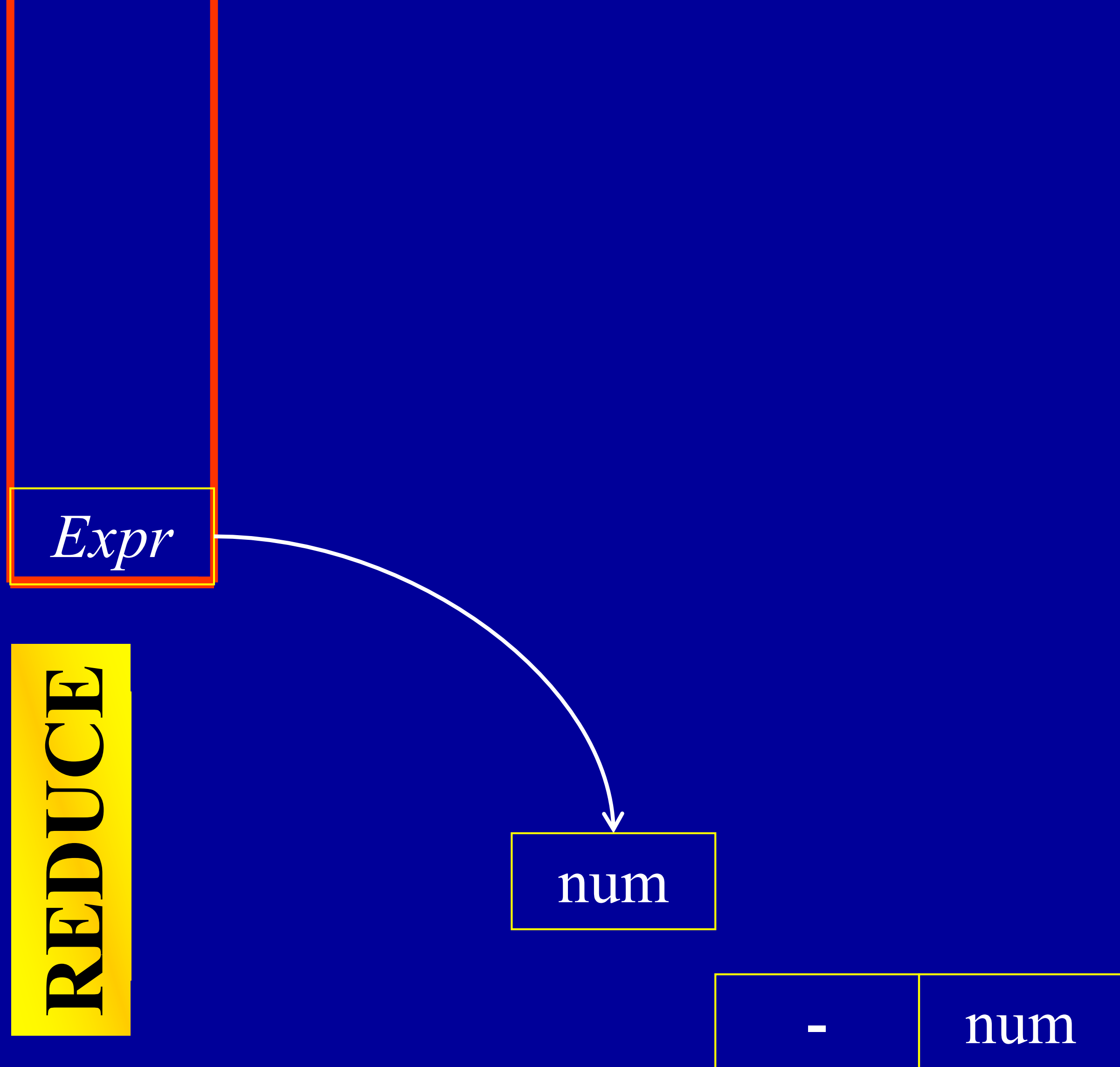
(4) $Expr \rightarrow Expr$ -

(5) $Expr \rightarrow$ num

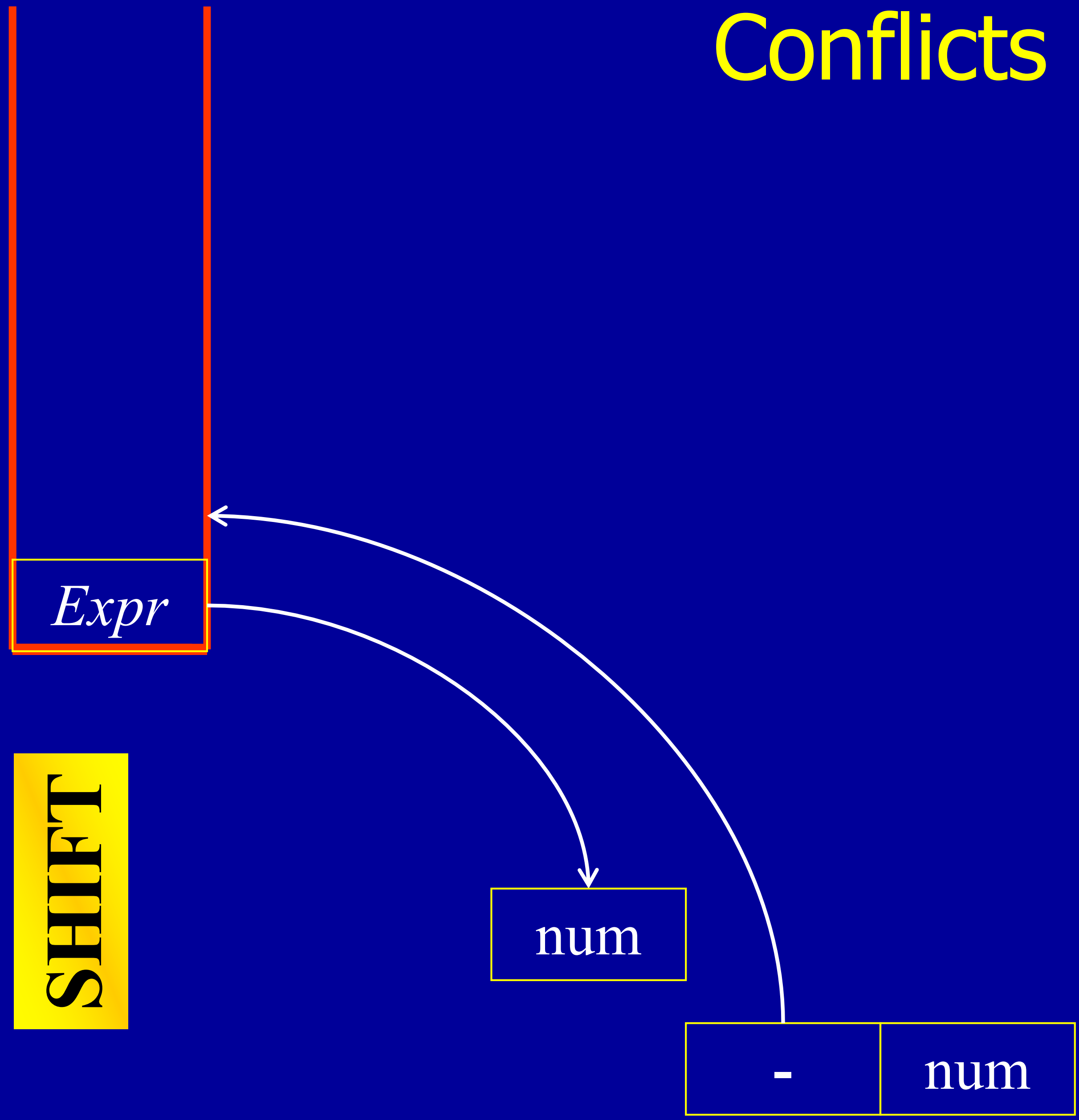
(6) $Op \rightarrow$ +

(7) $Op \rightarrow$ -

(8) $Op \rightarrow$ *



Conflicts



(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow Expr\ \text{-}\ Expr$

(3) $Expr \rightarrow (Expr)$

(4) $Expr \rightarrow Expr\ \text{-}$

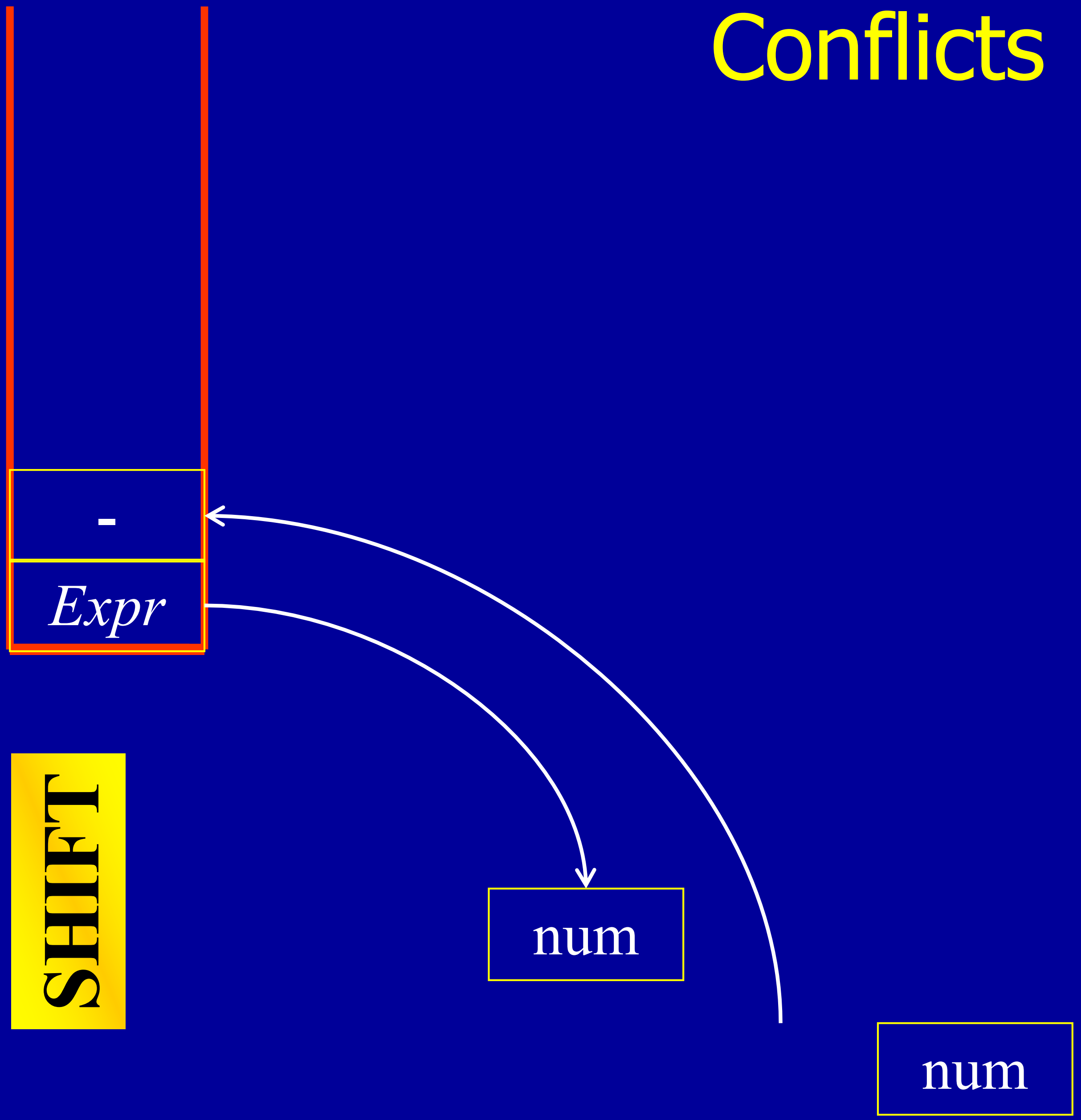
(5) $Expr \rightarrow \text{num}$

(6) $Op \rightarrow +$

(7) $Op \rightarrow \text{-}$

(8) $Op \rightarrow *$

Conflicts



(1) *Expr* → *Expr Op Expr*

(2) *Expr* → *Expr* - *Expr*

(3) *Expr* → (*Expr*)

(4) *Expr* → *Expr* -

(5) *Expr* → num

(6) *Op* → +

(7) *Op* → -

(8) *Op* → *

Shift/Reduce/Reduce Conflict

(1) $Expr \rightarrow Expr\ Op\ \ Expr$

(2) $Expr \rightarrow Expr$ - $Expr$

(3) $Expr \rightarrow$ ($Expr$)

(4) $Expr \rightarrow Expr$ -

(5) $Expr \rightarrow$ num

(6) $Op \rightarrow$ +

(7) $Op \rightarrow$ -

(8) $Op \rightarrow$ *

Options:

Reduce

Reduce

Shift

-

Expr

num

num

Shift/Reduce/Reduce Conflict

What Happens if Choose

Reduce

(1) *Expr* $\rightarrow$ *Expr Op Expr*

(2) *Expr* $\rightarrow$ *Expr* - *Expr*

(3) *Expr* $\rightarrow$ (*Expr*)

(4) $\boxed{Expr \rightarrow Expr\ -}$

(5) *Expr* $\rightarrow$ num

(6) *Op* $\rightarrow$ +

(7) *Op* $\rightarrow$ -

(8) *Op* $\rightarrow$ *

-

Expr

REDUCE

num

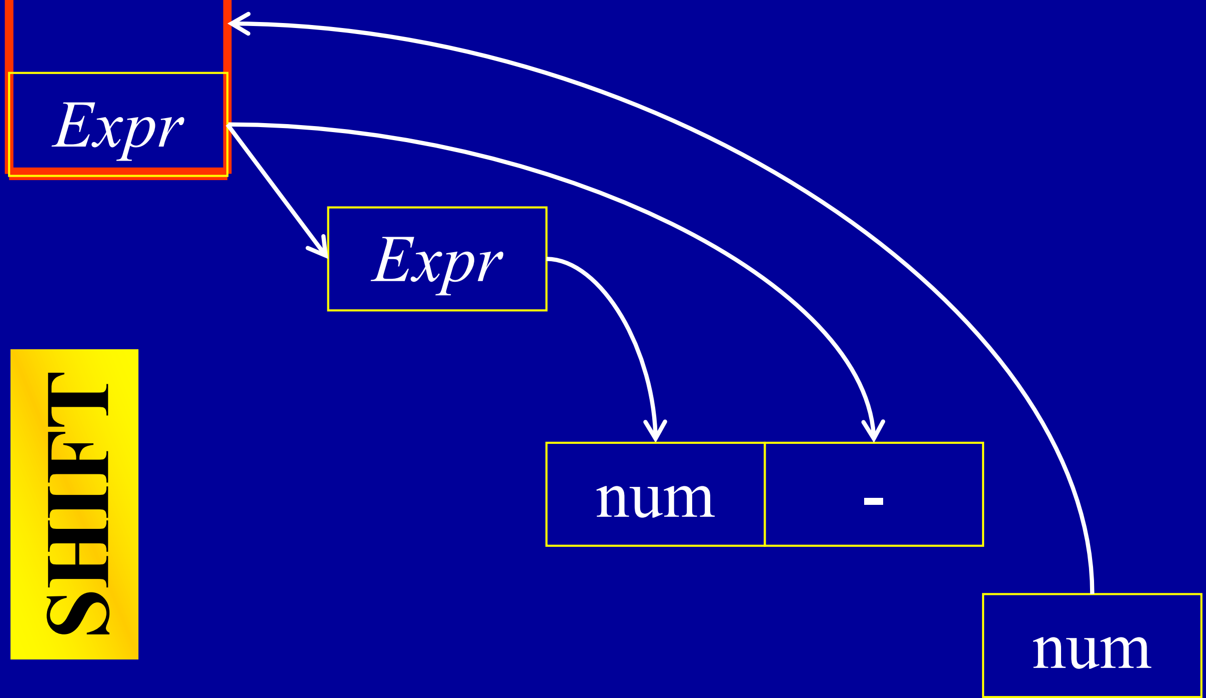
num

Shift/Reduce/Reduce Conflict

(1) *Expr* → *Expr Op Expr*
(2) *Expr* → *Expr - Expr*
(3) *Expr* → (*Expr*)
(4) *Expr* → *Expr -*
(5) *Expr* → num
(6) *Op* → +
(7) *Op* → -
(8) *Op* → *

What Happens if Choose

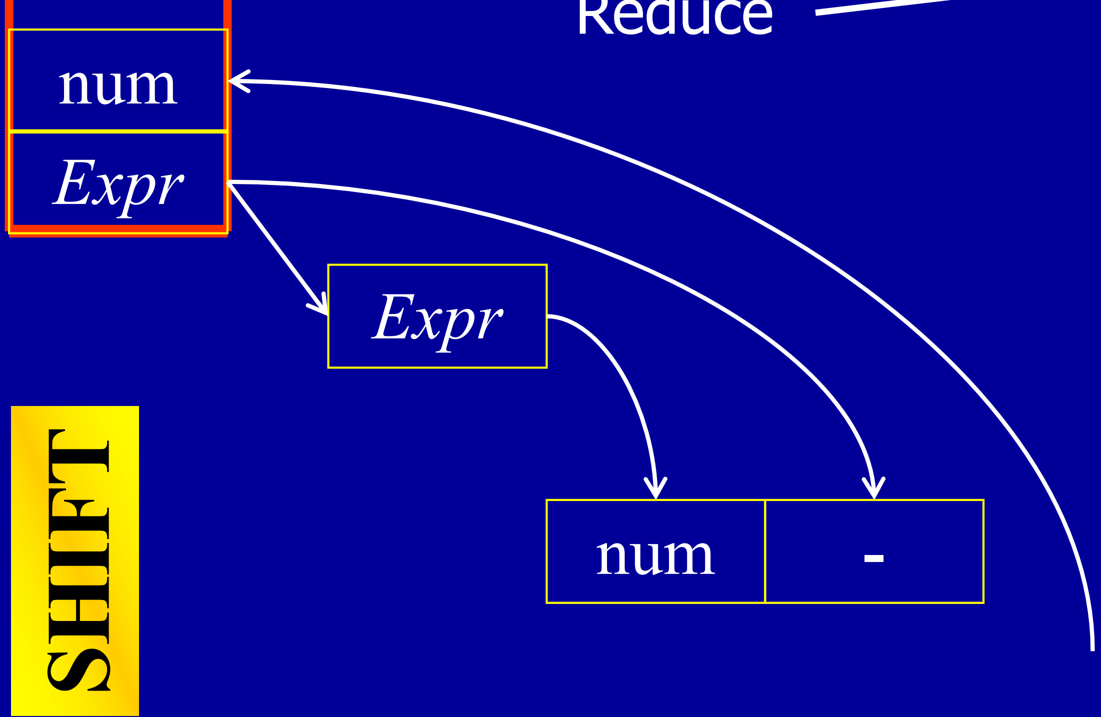
Reduce



Shift/Reduce/Reduce Conflict

What Happens if Choose Reduce

(1) $Expr \rightarrow Expr\ Op\ Expr$
(2) $Expr \rightarrow Expr - Expr$
(3) $Expr \rightarrow (Expr)$
(4) $Expr \rightarrow Expr -$
(5) $Expr \rightarrow$ num
(6) $Op \rightarrow +$
(7) $Op \rightarrow -$
(8) $Op \rightarrow *$

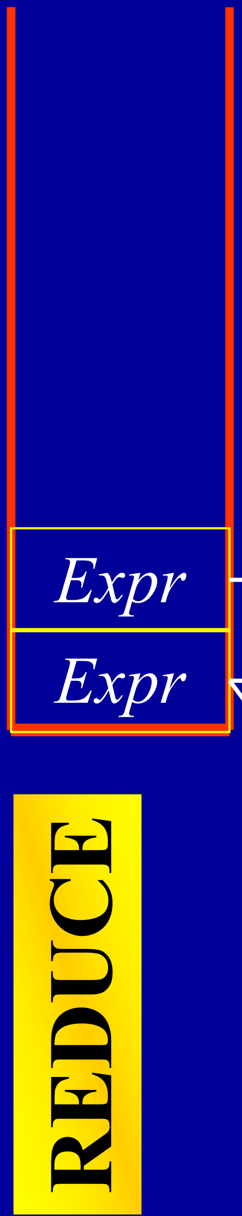


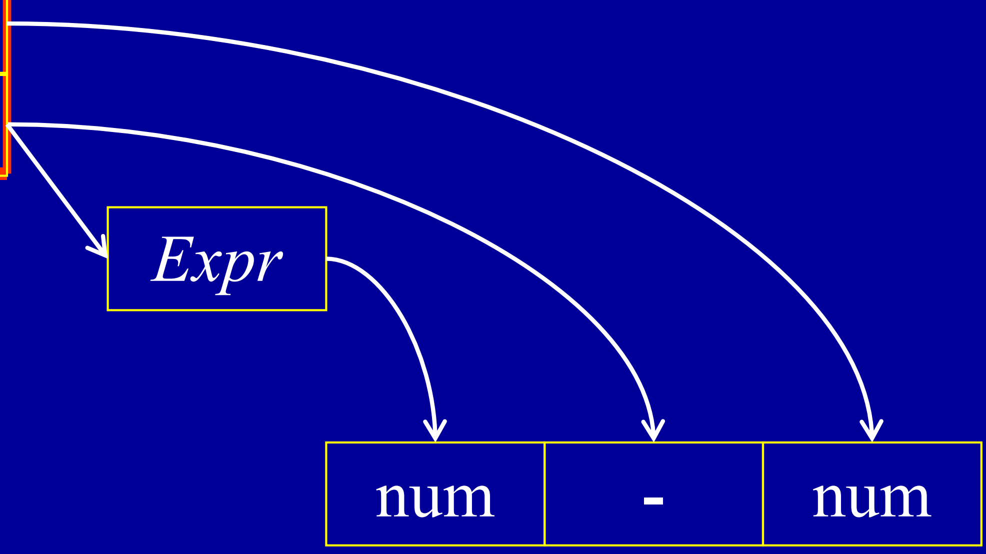
Shift/Reduce/Reduce Conflict

(1) $Expr \rightarrow Expr\ Op\ Expr$
(2) $Expr \rightarrow Expr$ - $Expr$
(3) $Expr \rightarrow (Expr)$
(4) $Expr \rightarrow Expr$ -
(5) $Expr \rightarrow$ num
(6) $Op \rightarrow$ +
(7) $Op \rightarrow$ -
(8) $Op \rightarrow$ *

What Happens if Choose

Reduce





Shift/Reduce/Reduce Conflict

What Happens if Choose

Reduce

(1) *Expr* → *Expr Op Expr*

(2) *Expr* → *Expr - Expr*

(3) *Expr* → (*Expr*)

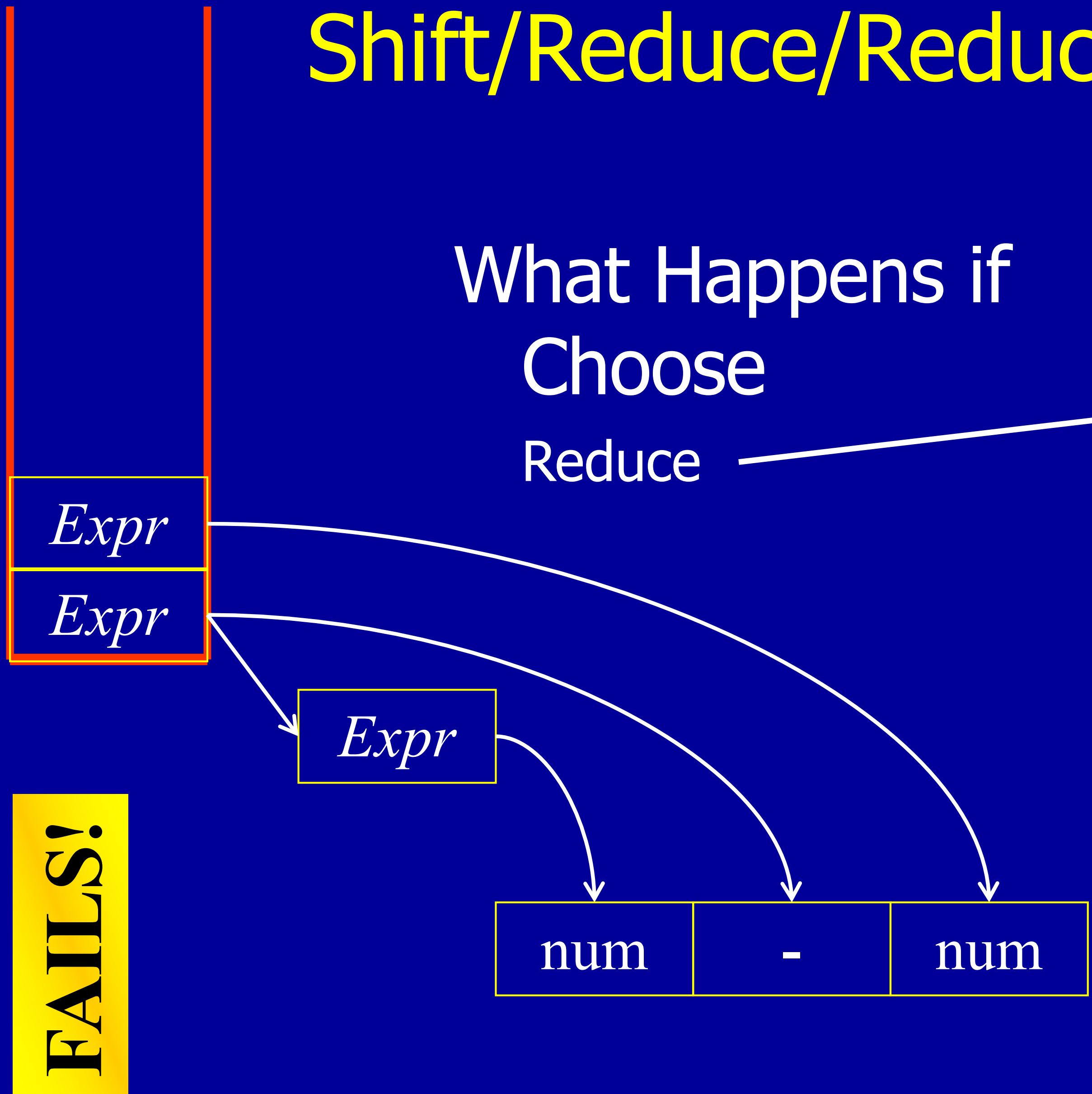
(4) *Expr* → *Expr -*

(5) *Expr* → num

(6) *Op* → +

(7) *Op* → -

(8) *Op* → *



Shift/Reduce/Reduce Conflict

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow Expr\ \text{-}\ Expr$

(3) $Expr \rightarrow (Expr)$

(4) $Expr \rightarrow Expr\ \text{-}$

(5) $Expr \rightarrow$ num

(6) $Op \rightarrow +$

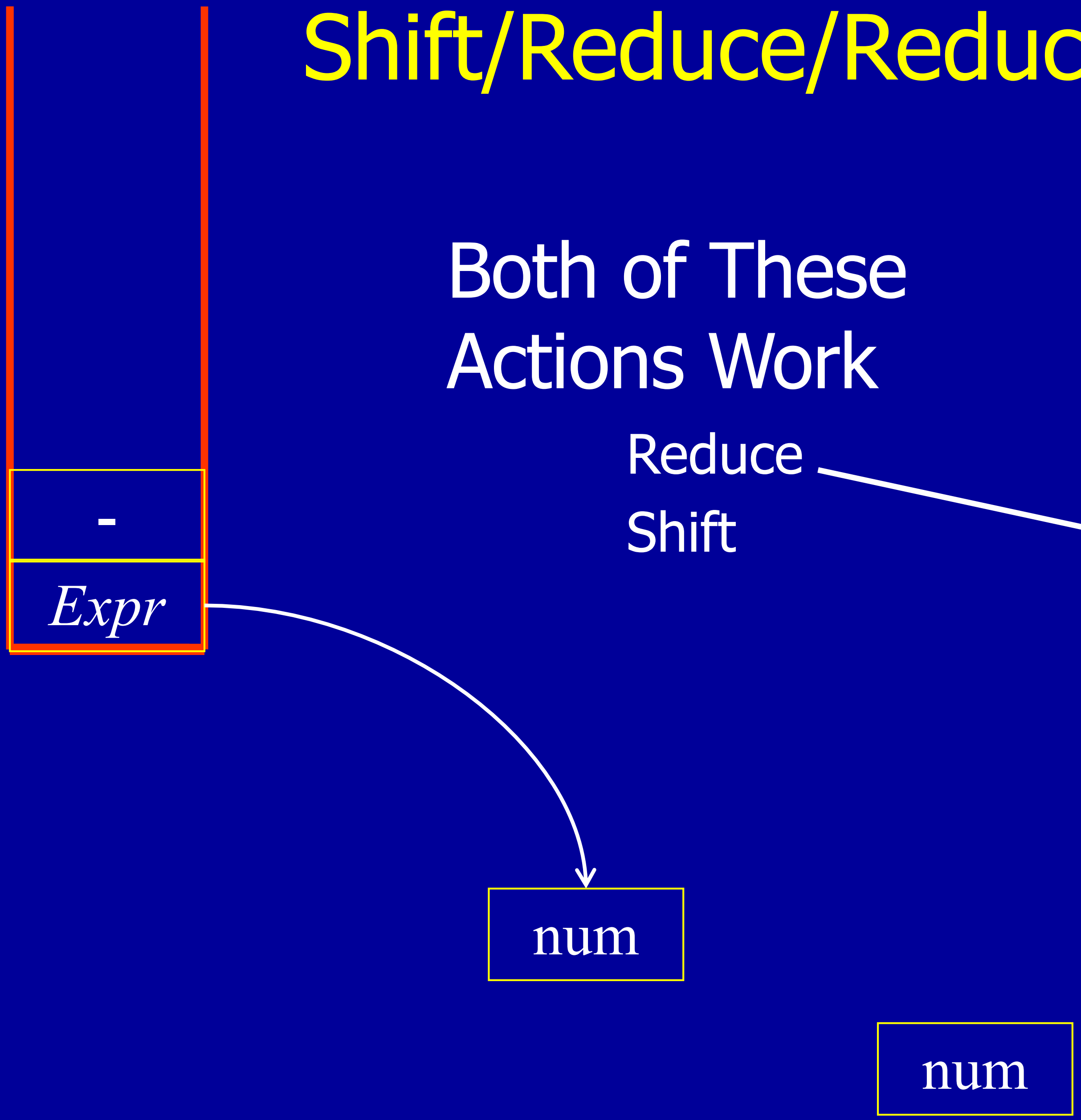
(7) $Op \rightarrow$ -

(8) $Op \rightarrow *$

Both of These Actions Work

Reduce

Shift



Shift/Reduce/Reduce Conflict

(1) $Expr \rightarrow Expr\ Op\ Expr$
(2) $Expr \rightarrow Expr$ - $Expr$
(3) $Expr \rightarrow$ ($Expr$)
(4) $Expr \rightarrow Expr$ -
(5) $Expr \rightarrow$ num
(6) $Op \rightarrow$ +
(7) $Op \rightarrow$ -
(8) $Op \rightarrow$ *

What Happens if Choose

Reduce

-

Expr

num

num

Shift/Reduce/Reduce Conflict

(1) *Expr* → *Expr Op Expr*
(2) *Expr* → *Expr - Expr*
(3) *Expr* → (*Expr*)
(4) *Expr* → *Expr -*
(5) *Expr* → num
(6) *Op* → +
(7) *Op* → -
(8) *Op* → *

What Happens if Choose

Reduce

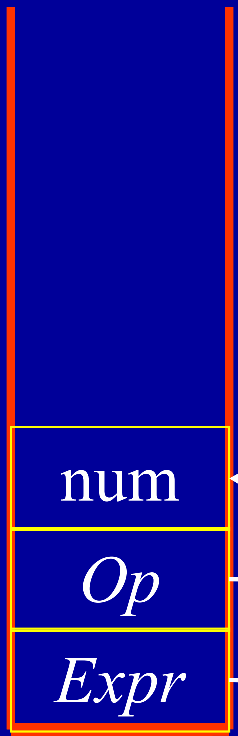
Op
Expr

REDUCE

num	-

num

Shift/Reduce/Reduce Conflict



What Happens if Choose

Reduce

(1) *Expr* → *Expr Op Expr*

(2) *Expr* → *Expr* - *Expr*

(3) *Expr* → (*Expr*)

(4) *Expr* → *Expr* -

(5) *Expr* → num

(6) *Op* → +

(7) *Op* → -

(8) *Op* → *

SHIFT

Shift/Reduce/Reduce Conflict

(1) $Expr \rightarrow Expr\ Op\ \ Expr$

(2) $Expr \rightarrow Expr\ -\ \ Expr$

(3) $Expr \rightarrow (Expr)$

(4) $Expr \rightarrow Expr\ -$

(5) $Expr \rightarrow$ num

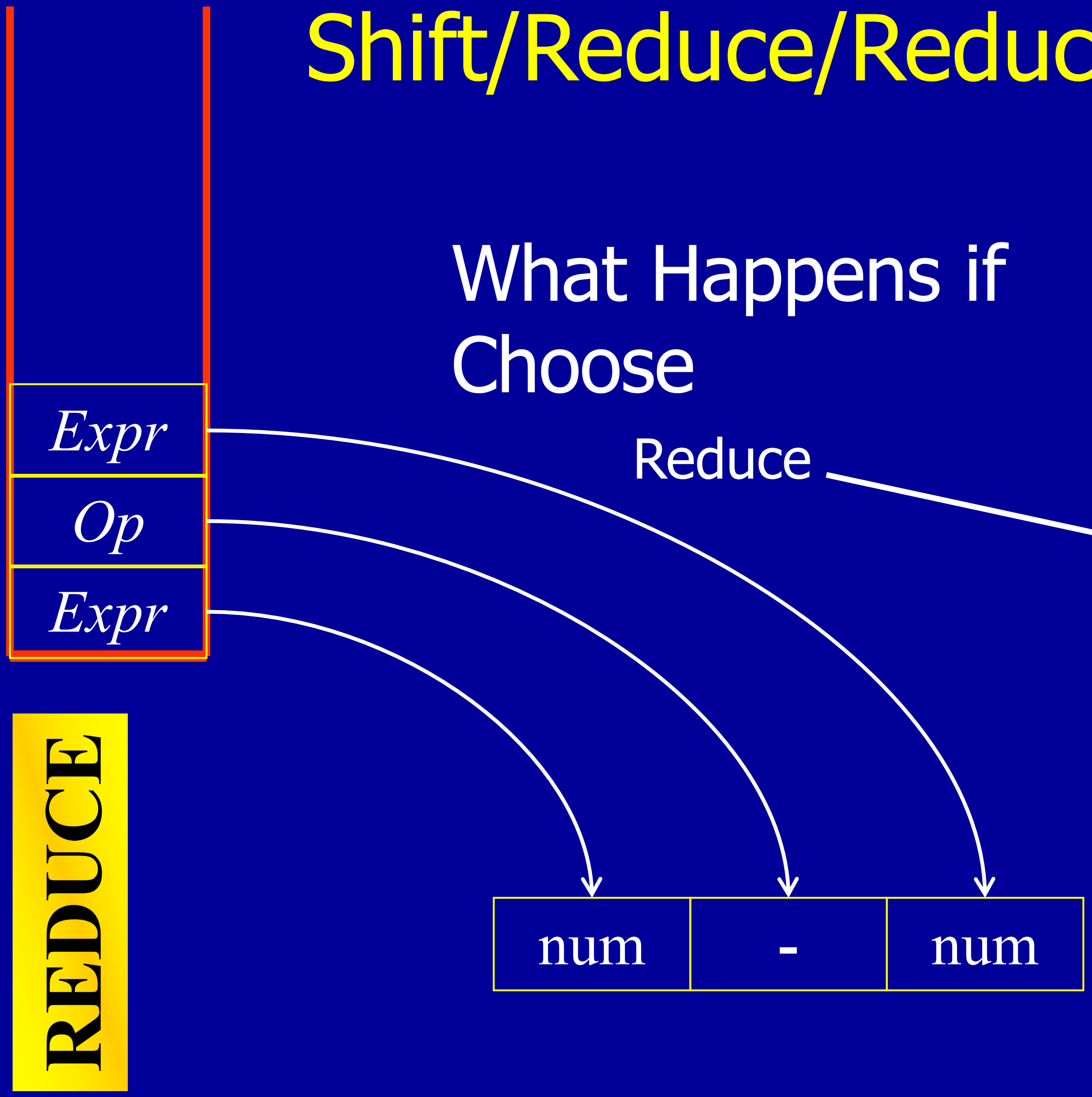
(6) $Op \rightarrow +$

(7) $Op \rightarrow -$

(8) $Op \rightarrow *$

What Happens if Choose

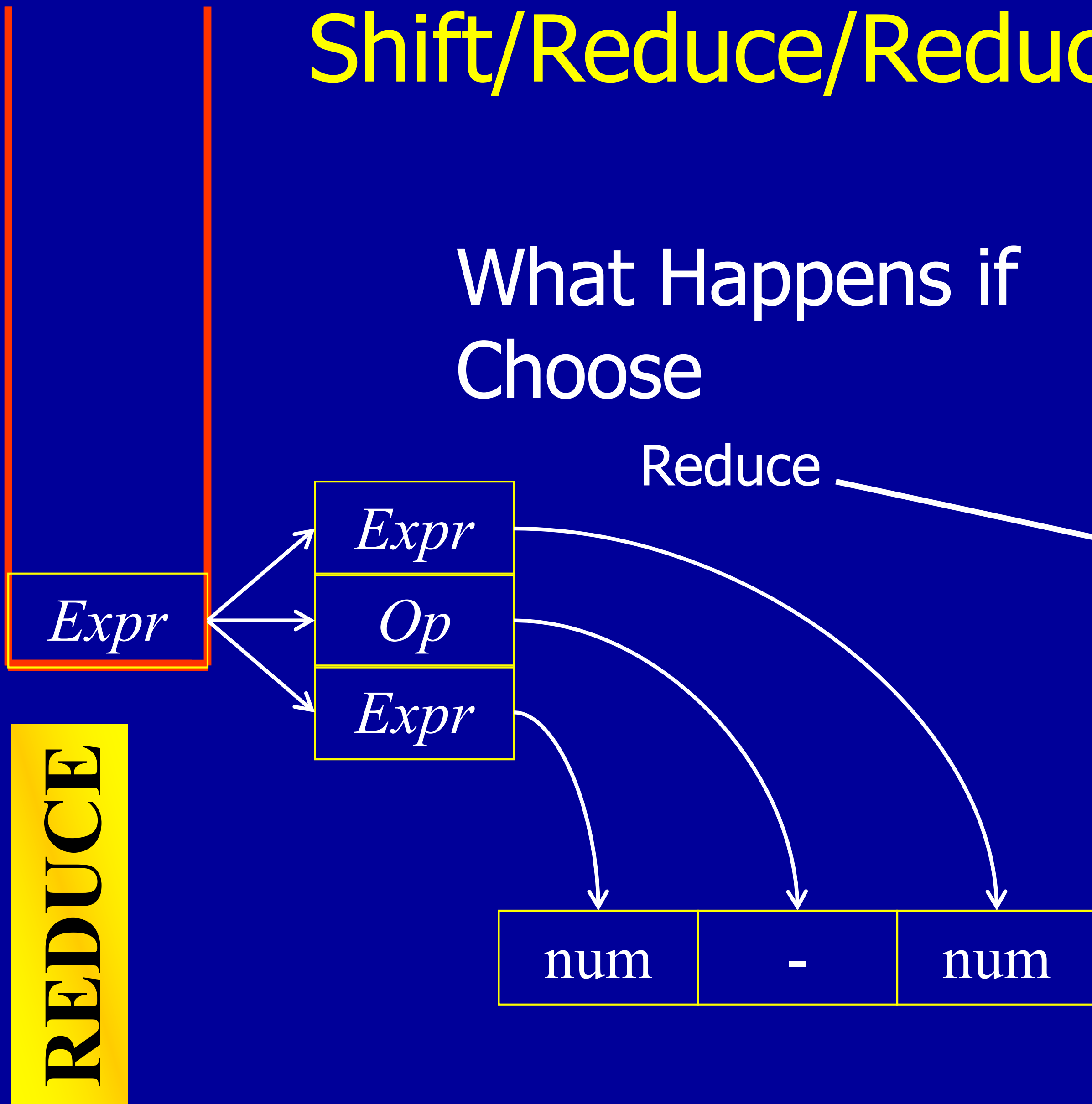
Reduce



Shift/Reduce/Reduce Conflict

What Happens if Choose Reduce

(1) $Expr \rightarrow Expr\ Op\ Expr$
(2) $Expr \rightarrow Expr$ - $Expr$
(3) $Expr \rightarrow (Expr)$
(4) $Expr \rightarrow Expr$ -
(5) $Expr \rightarrow$ num
(6) $Op \rightarrow$ +
(7) $Op \rightarrow$ -
(8) $Op \rightarrow$ *



Shift/Reduce/Reduce Conflict

What Happens if Choose

Reduce

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow Expr$ - $Expr$

(3) $Expr \rightarrow (Expr)$

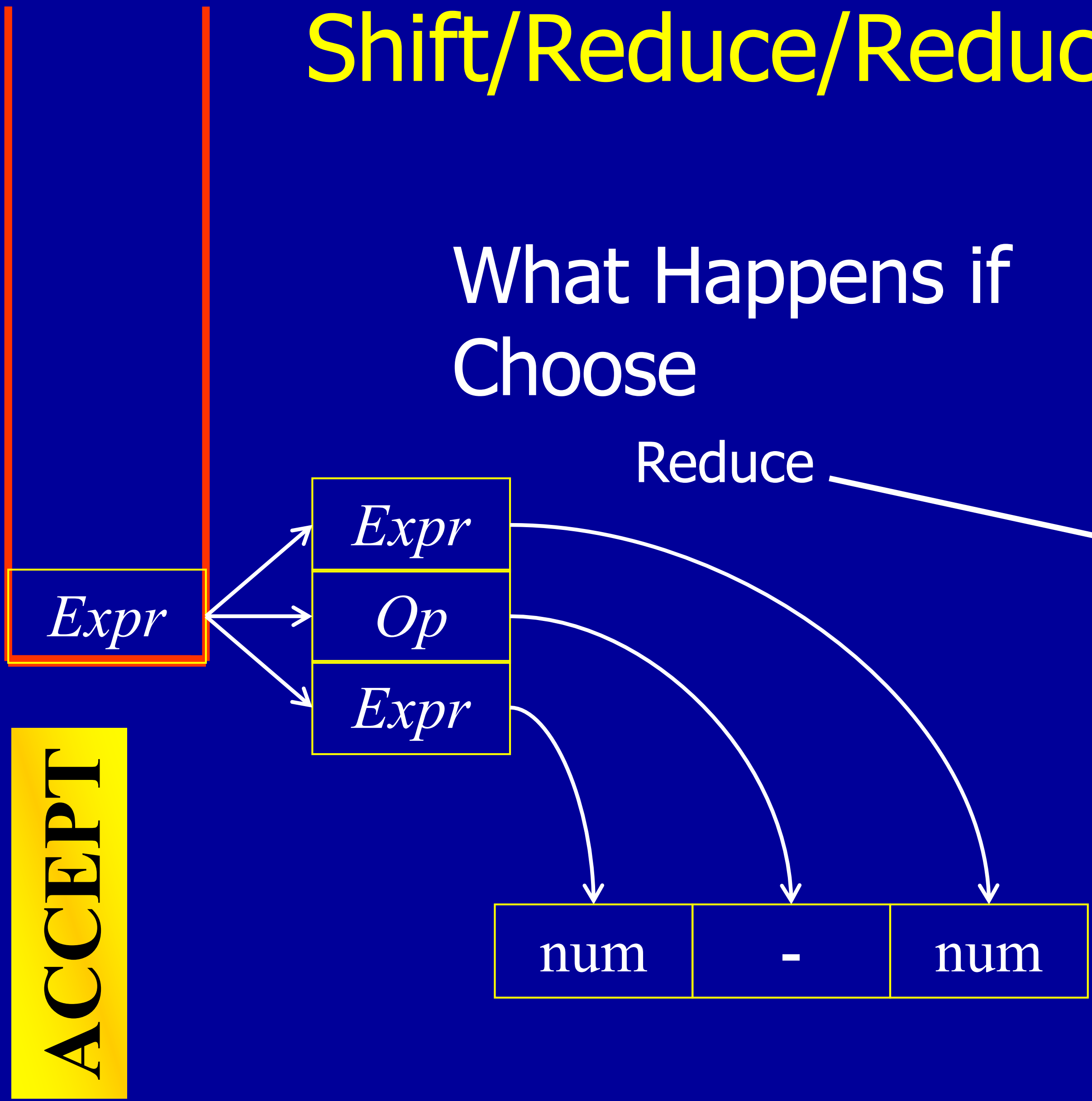
(4) $Expr \rightarrow Expr$ -

(5) $Expr \rightarrow$ num

(6) $Op \rightarrow$ +

(7) $Op \rightarrow$ -

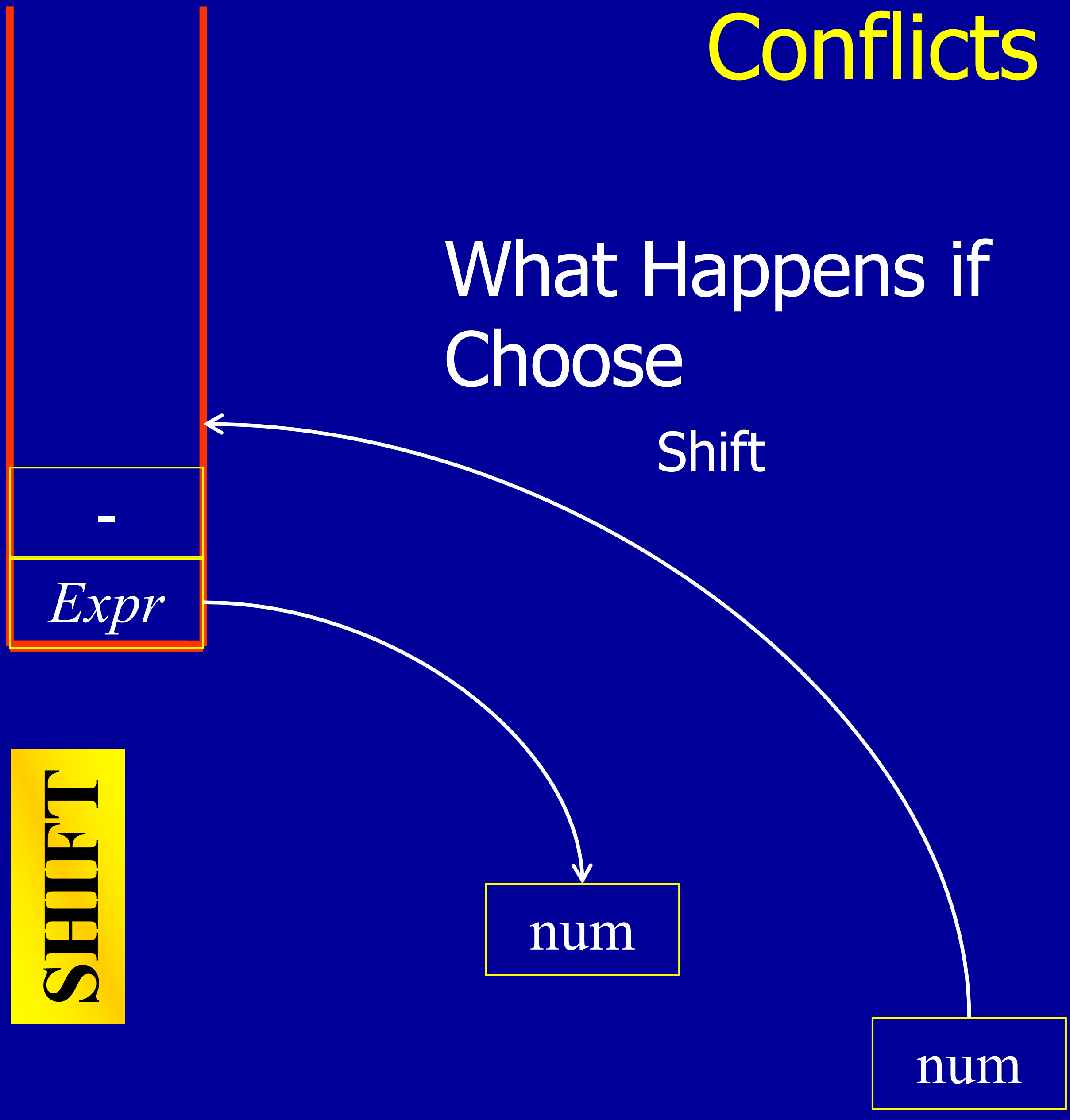
(8) $Op \rightarrow$ *



Conflicts

What Happens if Choose

Shift



(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow Expr\ \text{-}\ Expr$

(3) $Expr \rightarrow (Expr)$

(4) $Expr \rightarrow Expr\ \text{-}$

(5) $Expr \rightarrow \text{num}$

(6) $Op \rightarrow +$

(7) $Op \rightarrow \text{-}$

(8) $Op \rightarrow *$

Conflicts

What Happens if Choose
Shift

(1) *Expr* → *Expr Op Expr*
(2) *Expr* → *Expr* - *Expr*
(3) *Expr* → (*Expr*)
(4) *Expr* → *Expr* -
(5) *Expr* → num
(6) *Op* → +
(7) *Op* → -
(8) *Op* → *

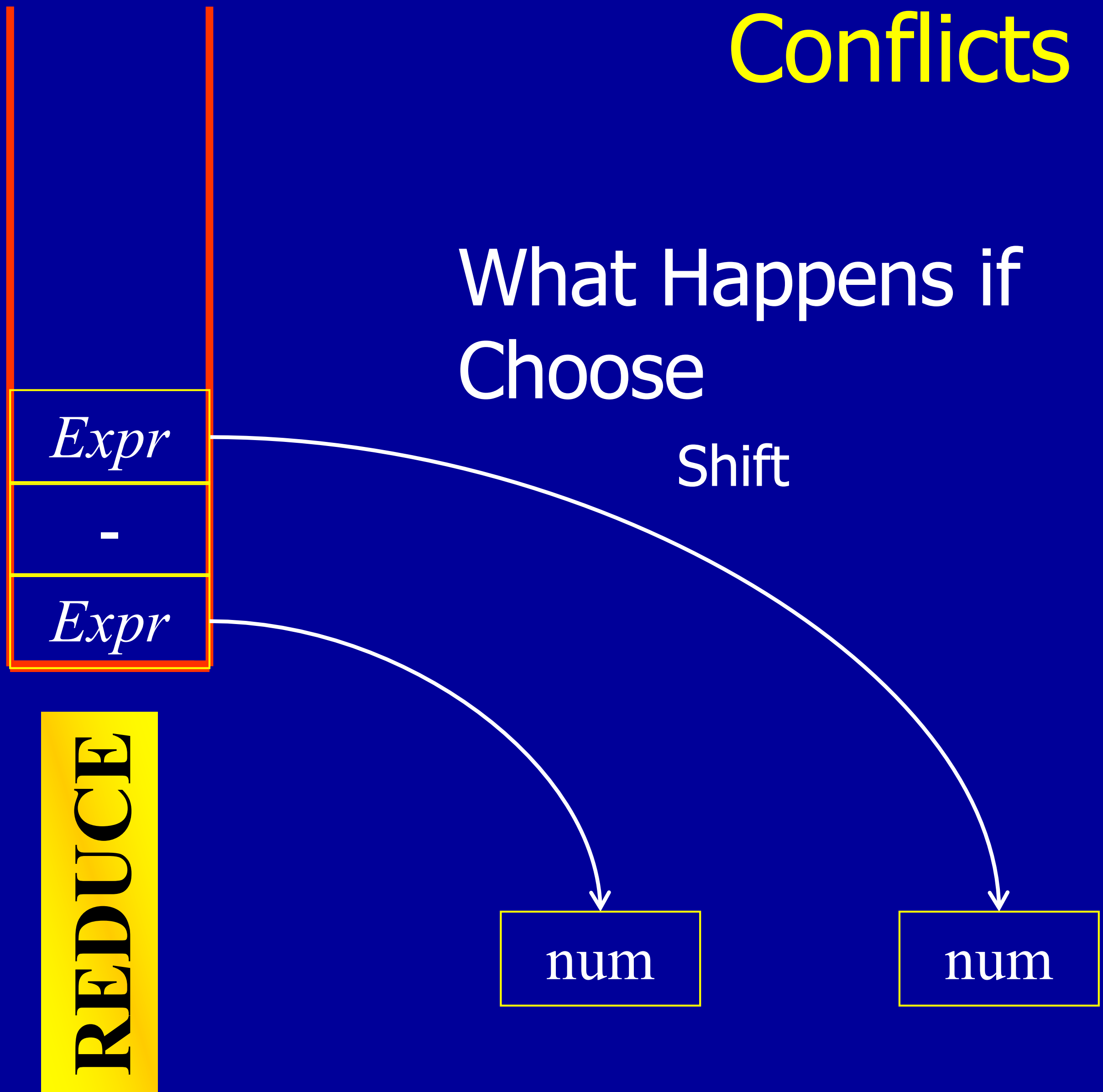
num
-
Expr

SHIFT

num

Conflicts

What Happens if Choose
Shift



(1) $Expr \rightarrow Expr\ Op\ \ Expr$

(2) $Expr \rightarrow Expr$ - $Expr$

(3) $Expr \rightarrow$ ($Expr$)

(4) $Expr \rightarrow Expr$ -

(5) $Expr \rightarrow$ num

(6) $Op \rightarrow$ +

(7) $Op \rightarrow$ -

(8) $Op \rightarrow$ *

Conflicts

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow Expr$ - $Expr$

(3) $Expr \rightarrow (Expr)$

(4) $Expr \rightarrow Expr$ -

(5) $Expr \rightarrow$ num

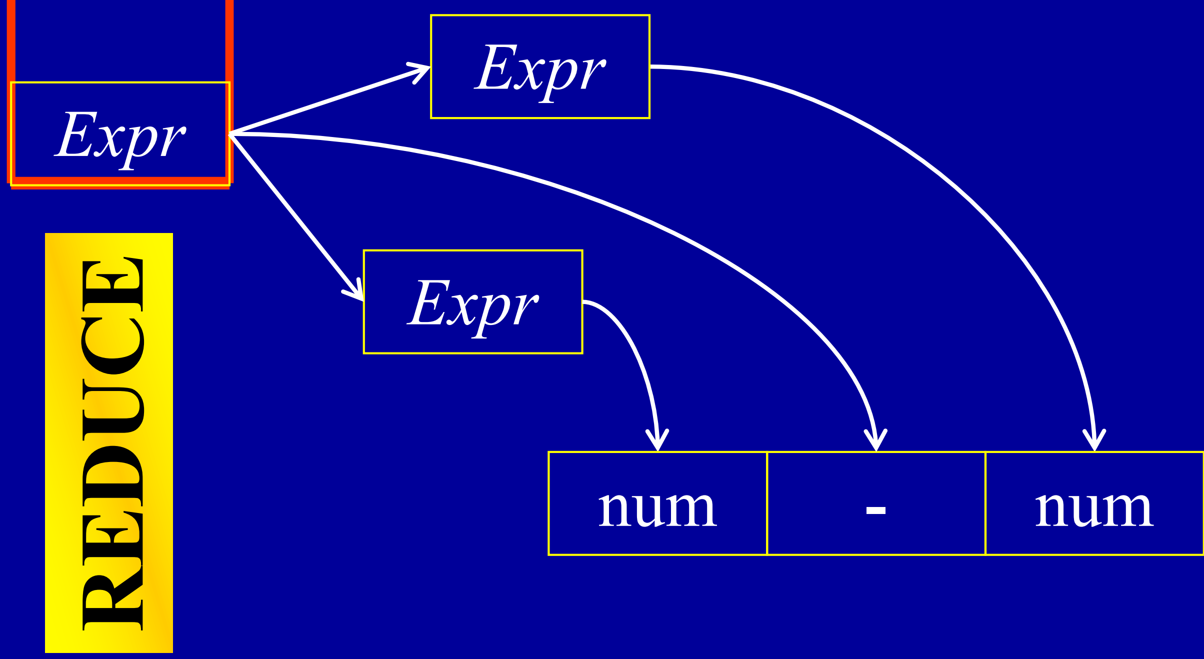
(6) $Op \rightarrow$ +

(7) $Op \rightarrow$ -

(8) $Op \rightarrow$ *

What Happens if Choose

Shift



Conflicts

What Happens if Choose

Shift

(1) *Expr* $\rightarrow$ *Expr Op Expr*

(2) *Expr* $\rightarrow$ *Expr - Expr*

(3) *Expr* $\rightarrow$ (*Expr*)

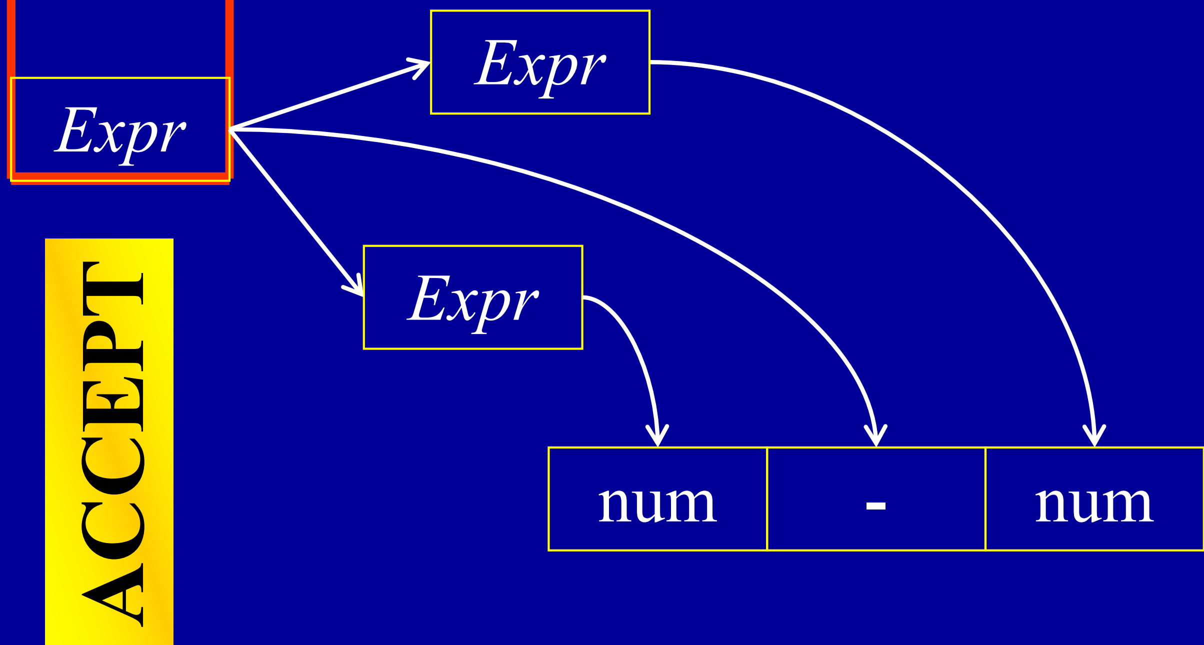
(4) *Expr* $\rightarrow$ *Expr -*

(5) *Expr* $\rightarrow$ num

(6) *Op* $\rightarrow$ +

(7) *Op* $\rightarrow$ -

(8) *Op* $\rightarrow$ *



Shift/Reduce/Reduce Conflict

This Shift/Reduce Conflict Reflects Ambiguity in Grammar

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow Expr$ - $Expr$

(3) $Expr \rightarrow (Expr)$

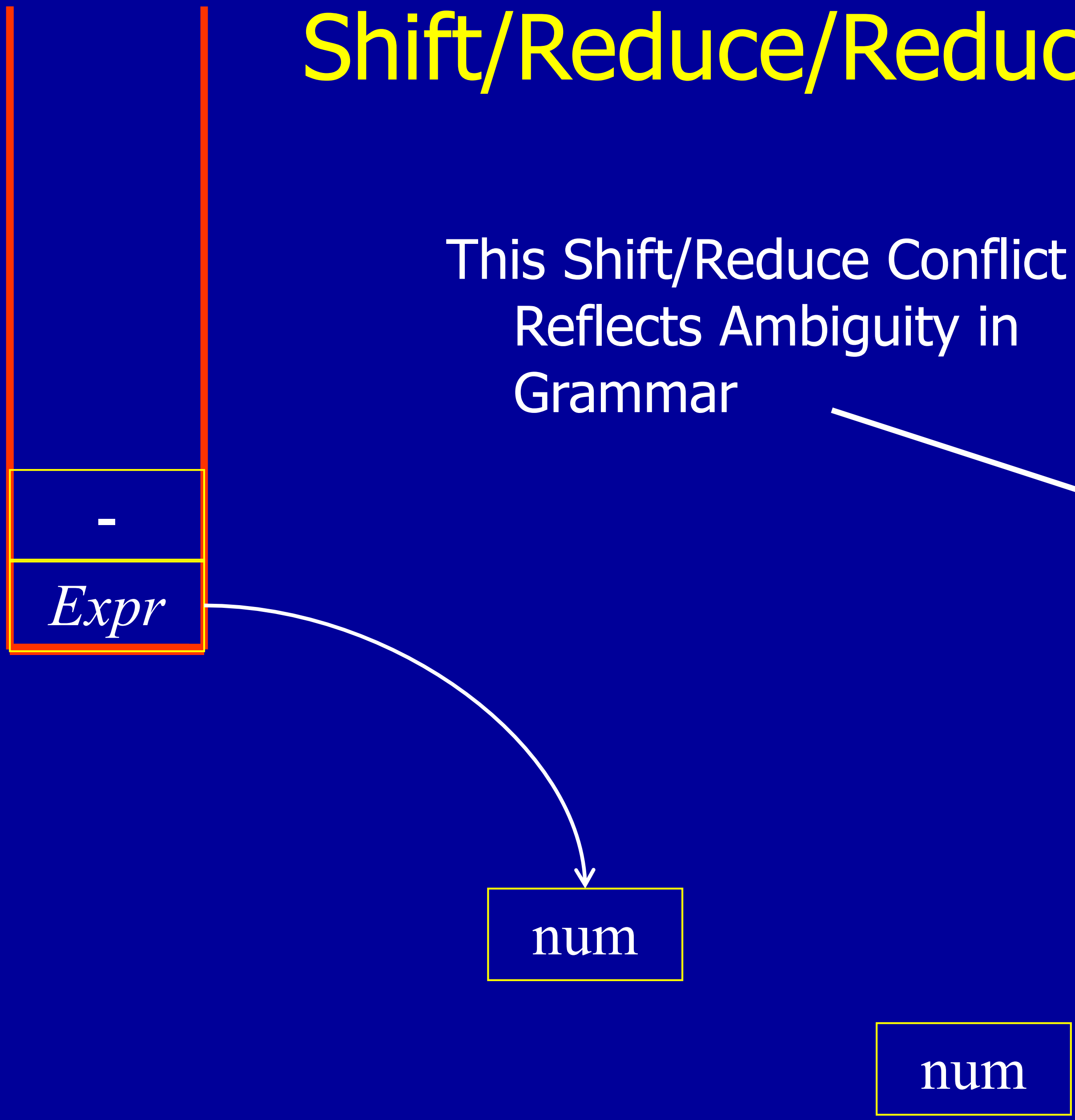
(4) $Expr \rightarrow Expr$ -

(5) $Expr \rightarrow$ num

(6) $Op \rightarrow +$

(7) $Op \rightarrow$ -

(8) $Op \rightarrow *$



Shift/Reduce/Reduce Conflict

(1) $Expr \rightarrow Expr\ Op\ Expr$

~~(2) $Expr \rightarrow Expr$ - $Expr$~~

(3) $Expr \rightarrow$ ($Expr$)

(4) $Expr \rightarrow Expr$ -

(5) $Expr \rightarrow$ num

(6) $Op \rightarrow$ +

(7) $Op \rightarrow$ -

(8) $Op \rightarrow$ *

This Shift/Reduce Conflict Reflects Ambiguity in Grammar

-

$Expr$

num

Eliminate by Hacking Grammar

num

Shift/Reduce/Reduce Conflict

-

Expr

This Shift/Reduce Conflict Can Be Eliminated By Lookahead of One Symbol

(1) $Expr \rightarrow Expr\ Op\ Expr$

(2) $Expr \rightarrow Expr$ - $Expr$

(3) $Expr \rightarrow (Expr)$

(4) $Expr \rightarrow Expr$ -

(5) $Expr \rightarrow$ num

(6) $Op \rightarrow +$

(7) $Op \rightarrow$ -

(8) $Op \rightarrow *$

Parser Generator Should Handle It

num

num

Constructing a Parser

- We will construct version with no lookahead
- Key Decisions
  - Shift or Reduce
  - Which Production to Reduce
- Basic Idea
  - Build a DFA to control shift and reduce actions
  - In effect, convert grammar to pushdown automaton
  - Encode finite state control in parse table

Parser State

- Input Token Sequence ($ for end of input)
- Current State from Finite State Automaton
- Two Stacks
  - State Stack (implements finite state automaton)
  - Symbol Stack (terminals from input and nonterminals from reductions)

Integrating Finite State Control

- Actions
  - Push Symbols and States Onto Stacks
  - Reduce According to a Given Production
  - Accept
- Selected action is a function of
  - Current input symbol
  - Current state of finite state control
- Each action specifies next state
- Implement control using parse table

Parse Tables

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Implements finite state control
- At each step, look up
  - Table[top of state stack] [input symbol]
- Then carry out the action

Parse Table Example

	ACTION			Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack | Symbol Stack | Input | Grammar

$$S \rightarrow X\$ \quad (1)$$

$$X \rightarrow (X) \quad (2)$$

$$X \rightarrow (\ ) \quad (3)$$

(())

s0

X

Parser Tables

		ACTION		Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Shift to s*n*
  - Push input token into the symbol stack
  - Push s*n* into state stack
  - Advance to next input symbol

Parser Tables

State	ACTION (	)	 $	Goto X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Reduce (*n*)
  - Pop both stacks as many times as the number of symbols on the RHS of rule *n*
  - Push LHS of rule *n* into symbol stack

Parser Tables

		ACTION		Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Reduce (*n*) (continued)
  - Look up
  - Table[top of the state stack][top of symbol stack]
  - Push that state (in goto part of table) onto state stack

Parser Tables

	ACTION			Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Accept
  - Stop parsing and report success

Parse Table In Action

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack　Symbol Stack　Input　Grammar

(())$

$$S \rightarrow X\$ \quad (1)$$
$$X \rightarrow (X) \quad (2)$$
$$X \rightarrow (\ ) \quad (3)$$

s0

Parse Table In Action

	ACTION			Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack Symbol Stack Input Grammar

(())$

$$S \rightarrow X\$ \quad (1)$$

$$X \rightarrow (X) \quad (2)$$

$$X \rightarrow (\;\;) \quad (3)$$

s0

Parse Table In Action

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack

s2
s0

Symbol Stack

(

Input

())$

Grammar

$$S \to X\$ \quad (1)$$

$$X \to (X) \quad (2)$$

$$X \to (\ ) \quad (3)$$

Parse Table In Action

	ACTION			Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack	Symbol Stack	Input
s2	(	())$
s0		

Grammar

$$S \rightarrow X\$ \quad (1)$$
$$X \rightarrow (X) \quad (2)$$
$$X \rightarrow (\ ) \quad (3)$$

Parse Table In Action

		ACTION		Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack

s2
s2
s0

Symbol Stack

(
(

Input

))$

Grammar

$S \rightarrow X\$ \quad (1)$

$X \rightarrow (X) \quad (2)$

$X \rightarrow (\ ) \quad (3)$

Parse Table In Action

	ACTION			Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack	Symbol Stack	Input	Grammar
s2 s2 s0	((	))$	$S \rightarrow X\$$ (1) $X \rightarrow (X)$ (2) $X \rightarrow (\ )$ (3)

Parse Table In Action

	ACTION			Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack

s5
s2
s2
s0

Symbol Stack

)
(
(

Input

)$

Grammar

$$S \rightarrow X\$ \quad (1)$$
$$X \rightarrow (X) \quad (2)$$
$$X \rightarrow (\ ) \quad (3)$$

Parse Table In Action

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack

s5
s2
s2
s0

Symbol Stack

)
(
(

Input

)$

Grammar

$S \rightarrow X\$ \quad (1)$

$X \rightarrow (X) \quad (2)$

$\boxed{X \rightarrow (\ ) \quad (3)}$

Step One: Pop Stacks

		ACTION		Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack

s5
s2
s2
s0

Symbol Stack

)
(
(

Input

)$

Grammar

$S \rightarrow X\$ \quad (1)$

$X \rightarrow (X) \quad (2)$

$\boxed{X \rightarrow (\ ) \quad (3)}$

Step One: Pop Stacks

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack	Symbol Stack	Input	Grammar
s2	(	)$	$S \rightarrow X\$ \quad (1)$
s0			$X \rightarrow (X) \quad (2)$
			$\boxed{X \rightarrow (\ ) \quad (3)}$

Step Two: Push Nonterminal

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack Symbol Stack Input Grammar

)$

$S \rightarrow X\$ \quad (1)$

$X \rightarrow (X) \quad (2)$

$\boxed{X \rightarrow (\ ) \quad (3)}$

State Stack
s2
s0

Symbol Stack
(

Step Two: Push Nonterminal

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack

s2
s0

Symbol Stack

X
(

Input

)$

Grammar

$S \rightarrow X\$ \quad (1)$

$X \rightarrow (X) \quad (2)$

$\boxed{X \rightarrow (\ ) \quad (3)}$

Step Three: Use Goto, Push New State

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack

s2
s0

Symbol Stack

X
(

Input

)$

Grammar

$$S \rightarrow X\$ \quad (1)$$
$$X \rightarrow (X) \quad (2)$$
$$X \rightarrow (\ ) \quad (3)$$

Step Three: Use Goto, Push New State

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack

s3
s2
s0

Symbol Stack

X
(

Input

)$

Grammar

$S \rightarrow X\$ \quad (1)$

$X \rightarrow (X) \quad (2)$

$X \rightarrow (\;) \quad (3)$

Parse Table In Action

	ACTION			Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack

s3
s2
s0

Symbol Stack

X
(

Input

)$

Grammar

$$S \rightarrow X\$ \quad (1)$$
$$X \rightarrow (X) \quad (2)$$
$$X \rightarrow (\ ) \quad (3)$$

Parse Table In Action

	ACTION			Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack

s4
s3
s2
s0

Symbol Stack

)
X
(

Input

$

Grammar

$$S \to X\$ \quad (1)$$
$$X \to (X) \quad (2)$$
$$X \to (\ ) \quad (3)$$

Parse Table In Action

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack

s4
s3
s2
s0

Symbol Stack

)
X
(

Input

$

Grammar

$S \rightarrow X\$ \quad (1)$

$\boxed{X \rightarrow (X) \quad (2)}$

$X \rightarrow (\ ) \quad (3)$

Step One: Pop Stacks

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack

s4
s3
s2
s0

Symbol Stack

)
X
(

Input

$

Grammar

$S \rightarrow X\$ \quad (1)$

$\boxed{X \rightarrow (X) \quad (2)}$

$X \rightarrow (\ ) \quad (3)$

Step One: Pop Stacks

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack | Symbol Stack | Input | Grammar

$

$S \rightarrow X\$ \quad (1)$

$\boxed{X \rightarrow (X) \quad (2)}$

$X \rightarrow (\ ) \quad (3)$

s0

Step Two: Push Nonterminal

		ACTION		Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack   Symbol Stack   Input   Grammar

$

$$S \rightarrow X\$ \quad (1)$$

$$\boxed{X \rightarrow (X) \quad (2)}$$

$$X \rightarrow (\ ) \quad (3)$$

s0

Step Two: Push Nonterminal

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack　Symbol Stack　Input　Grammar

$

$S \rightarrow X\$ \quad (1)$

$\boxed{X \rightarrow (X) \quad (2)}$

$X \rightarrow (\ ) \quad (3)$

s0

X

Step Three: Use Goto, Push New State

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack	Symbol Stack	Input	Grammar
		$	$S \to X\$ \quad (1)$
			$\boxed{X \to (X) \quad (2)}$
s0	X		$X \to (\ ) \quad (3)$

Step Three: Use Goto, Push New State

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack	Symbol Stack	Input	Grammar
		$	$S \rightarrow X\$$ (1)
s1			$\boxed{X \rightarrow (X) \quad (2)}$
s0	X		$X \rightarrow (\ )$ (3)

Accept the String!

	ACTION			Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State Stack	Symbol Stack	Input
s1		$
s0	S	

Grammar

$$S \rightarrow X\$ \quad (1)$$
$$X \rightarrow (X) \quad (2)$$
$$X \rightarrow (\ ) \quad (3)$$

Key Concepts

- Pushdown automaton for parsing
  - Stack, Finite state control
  - Parse actions: shift, reduce, accept
- Parse table for controlling parser actions
  - Indexed by parser state and input symbol
  - Entries specify action and next state
  - Use state stack to help control
- Parse tree construction
  - Reads input from left to right
  - Bottom-up construction of parse tree

MIT 6.1100
Parse Table Construction

Martin Rinard

Massachusetts Institute of Technology

Parse Tables (Review)

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Implements finite state control
- At each step, look up
  - Table[top of state stack] [input symbol]
- Then carry out the action

Parse Tables (Review)

		ACTION		Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Shift to s*n*
  - Push input token into the symbol stack
  - Push s*n* into state stack
  - Advance to next input symbol

Parse Tables (Review)

	ACTION			Goto
State	**(**	**)**	**$**	**X**
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Reduce (*n*)
  - Pop both stacks as many times as the number of symbols on the RHS of rule *n*
  - Push LHS of rule *n* into symbol stack

Parser Generators and Parse Tables

- Parser generator (YACC, CUP)
  - Given a grammar
  - Produces a (shift-reduce) parser for that grammar
- Process grammar to synthesize a DFA
  - Contains states that the parser can be in
  - State transitions for terminals and non-terminals
- Use DFA to create an parse table
- Use parse table to generate code for parser

Example

- The grammar

$$S \rightarrow X\,\$ \qquad (1)$$

$$X \rightarrow (X) \qquad (2)$$

$$X \rightarrow (\ \ ) \qquad (3)$$

DFA States Based on Items

- We need to capture how much of a given production we have scanned so far

$$X \rightarrow (\quad X \quad)$$

Are we here? Or here? Or here? Or here?

Items

- We need to capture how much of a given production we have scanned so far

$$X \rightarrow (\quad X \quad)$$

- Production Generates 4 items
  - $X \rightarrow \bullet\ (X)$
  - $X \rightarrow (\ \bullet X)$
  - $X \rightarrow (X \bullet)$
  - $X \rightarrow (X)\ \bullet$

Example of Items

- The grammar

  $S \rightarrow X\ \$$

  $X \rightarrow (X)$

  $X \rightarrow (\ )$

- Items

  $S \rightarrow \bullet X \$$

  $S \rightarrow X \bullet \$$

  $X \rightarrow \bullet (X)$

  $X \rightarrow (\bullet X)$

  $X \rightarrow (X \bullet)$

  $X \rightarrow (X) \bullet$

  $X \rightarrow \bullet (\ )$

  $X \rightarrow (\bullet)$

  $X \rightarrow (\ ) \bullet$

Notation

- If write production as $A \rightarrow \alpha\ c\ \beta$
  - α is sequence of grammar symbols, can be terminals and nonterminals in sequence
  - c is terminal
  - β is sequence of grammar symbols, can be terminals and nonterminals in sequence
- If write production as $A \rightarrow \alpha \bullet B\ \beta$
  - α, β as above
  - B is a single grammar symbol, either terminal or nonterminal

Key idea behind items

- States correspond to sets of items
- If the state contains the item $A \to \alpha \bullet c\ \beta$
  - Parser is expecting to eventually reduce using the production $A \to \alpha\ c\ \beta$
  - Parser has already parsed an α
  - It expects the input may contain c, then β
- If the state contains the item $A \to \alpha \bullet$
  - Parser has already parsed an α
  - Will reduce using $A \to \alpha$
- If the state contains the item $S \to \alpha \bullet \$$ and the input buffer is empty
  - Parser accepts input

Correlating Items and Actions

- If the current state contains the item $A \rightarrow \alpha \bullet c\ \beta$ and the current symbol in the input buffer is c
  - Parser shifts c onto stack
  - Next state will contain $A \rightarrow \alpha\ c \bullet \beta$
- If the current state contains the item $A \rightarrow \alpha \bullet$
  - Parser reduces using $A \rightarrow \alpha$
- If the current state contains the item $S \rightarrow \alpha \bullet \$$ and the input buffer is empty
  - Parser accepts input

Closure() of a set of items

- Closure finds all the items in the same "state"
- Fixed Point Algorithm for Closure(I)
  - Every item in I is also an item in Closure(I)
  - If $A \rightarrow \alpha \bullet B \beta$ is in Closure(I) and $B \rightarrow \bullet \gamma$ is an item, then add $B \rightarrow \bullet \gamma$ to Closure(I)
  - Repeat until no more new items can be added to Closure(I)

Example of Closure

- Closure($\{X \to (\bullet X)\}$)

$$\left\{ \begin{array}{l} X \to (\bullet X) \\ X \to \bullet (X) \\ X \to \bullet (\,) \end{array} \right\}$$

- Items

$S \to \bullet X\$$

$S \to X \bullet \$$

$X \to \bullet (X)$

$X \to (\bullet X)$

$X \to (X \bullet)$

$X \to (X) \bullet$

$X \to \bullet (\,)$

$X \to (\bullet)$

$X \to (\,) \bullet$

Another Example

- Closure($\{S \rightarrow \bullet X\$\}$)

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

- Items

$S \rightarrow \bullet X\$$

$S \rightarrow X \bullet \$$

$X \rightarrow \bullet (X)$

$X \rightarrow (\bullet X)$

$X \rightarrow (X \bullet)$

$X \rightarrow (X) \bullet$

$X \rightarrow \bullet (\ )$

$X \rightarrow (\bullet)$

$X \rightarrow (\ ) \bullet$

Goto() of a set of items

- Goto finds the new state after consuming a grammar symbol while at the current state

- Algorithm for Goto(I, X)
  where I is a set of items
  and X is a grammar symbol

$$Goto(I, X) = Closure(\{ A \rightarrow \alpha X \bullet \beta \mid A \rightarrow \alpha \bullet X \beta \text{ in } I \})$$

- goto is the new set obtained by "moving the dot" over X

Example of Goto

- Goto $(\{X \to (\bullet X)\}, X)$

$$\left\{ X \to (X \bullet \ ) \qquad\qquad\qquad\qquad \right\}$$

- Items

$S \to \bullet X\$$

$S \to X \bullet \$$

$X \to \bullet (X)$

$X \to (\bullet X)$

$X \to (X \bullet)$

$X \to (X) \bullet$

$X \to \bullet (\ )$

$X \to (\bullet)$

$X \to (\ ) \bullet$

Another Example of Goto

- Goto $(\{X \to \bullet(X)\}, ()$

$$\left\{\begin{array}{l} X \to (\bullet X) \\ X \to \bullet(X) \\ X \to \bullet() \end{array}\right\}$$

- Items

$S \to \bullet X\$$

$S \to X \bullet \$$

$X \to \bullet(X)$

$X \to (\bullet X)$

$X \to (X \bullet)$

$X \to (X) \bullet$

$X \to \bullet()$

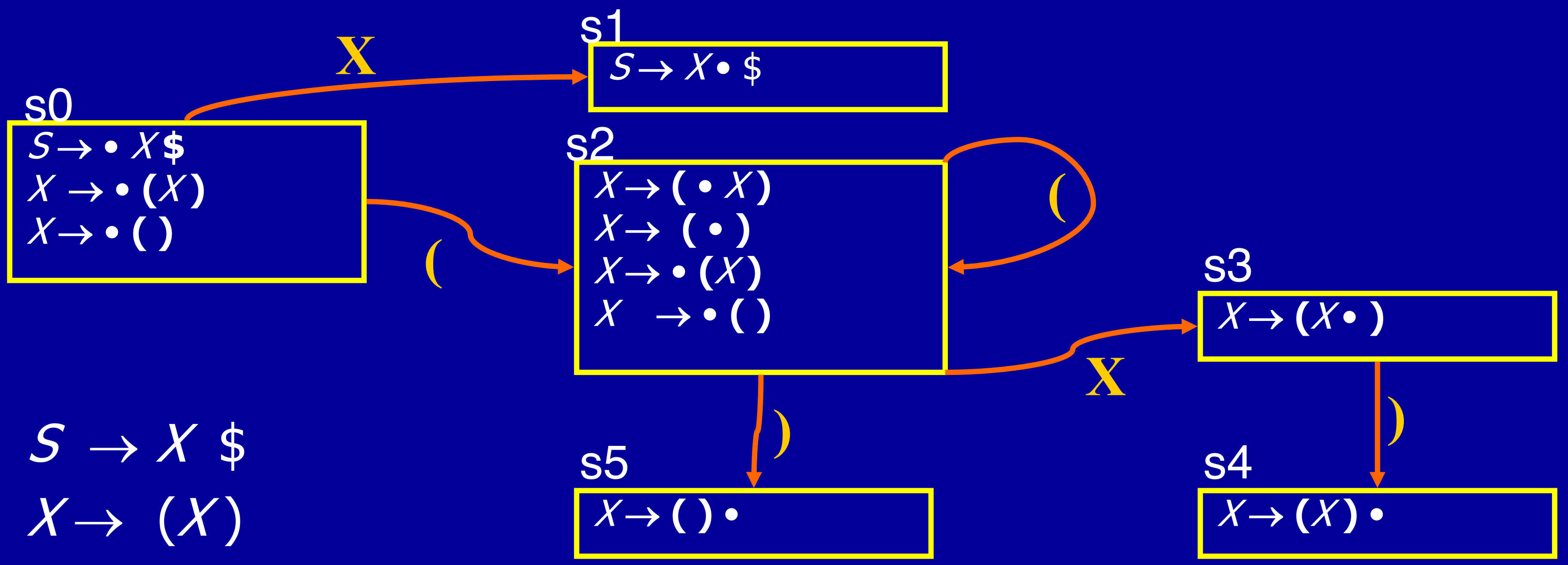
$X \to (\bullet)$

$X \to () \bullet$

Building the DFA states

- Start with the item $S \rightarrow \bullet \beta \$$
- Create the first state to be Closure($\{ S \rightarrow \bullet \beta \$ \}$)
- Pick a state I
  - for each item $A \rightarrow \alpha \bullet X \beta$ in I
    - If there exists an edge X from state I to state J, then add Goto(I,X) to J
    - Otherwise make a new state J, add edge X from state I to state J, and add Goto(I,X) to J
- Repeat until no more additions possible

DFA Example



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

Constructing A Parse Engine

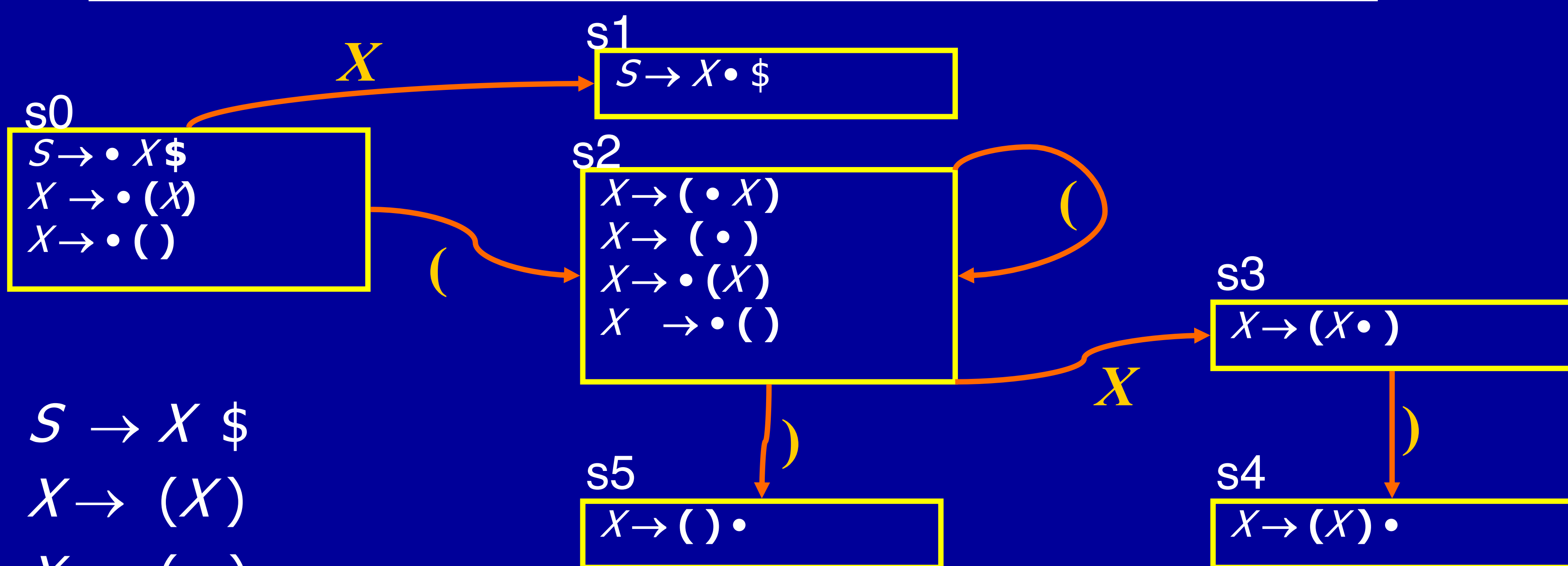
- Build a DFA - DONE
- Construct a parse table using the DFA

Creating the parse tables

- For each state
  - Transition to another state using a terminal symbol is a shift to that state (*shift to sn*)
  - Transition to another state using a non-terminal is a goto to that state (*goto sn*)
  - If there is an item $A \rightarrow \alpha \bullet$ in the state do a reduction with that production for all terminals (*reduce k*)

Building Parse Table Example

State	ACTION (	ACTION)	ACTION $	Goto X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	



$S \rightarrow X\ \$$

$X \rightarrow (X)$

$X \rightarrow (\ )$

Potential Problem

- No lookahead
- Vulnerable to unnecessary conflicts
  - Shift/Reduce Conflicts (may reduce too soon in some cases)
  - Reduce/Reduce Conflicts
- Solution: Lookahead
  - Only for reductions - reduce only when next symbol can occur after nonterminal from production
  - Systematic lookahead, split states based on next symbol, action is always a function of next symbol
  - Can generalize to look ahead multiple symbols

Reduction-Only Lookahead Parsing

- If a state contains $A \rightarrow \beta \bullet$
- Reduce by $A \rightarrow \beta$ only if next input symbol can follow A in some derivation
- Example Grammar

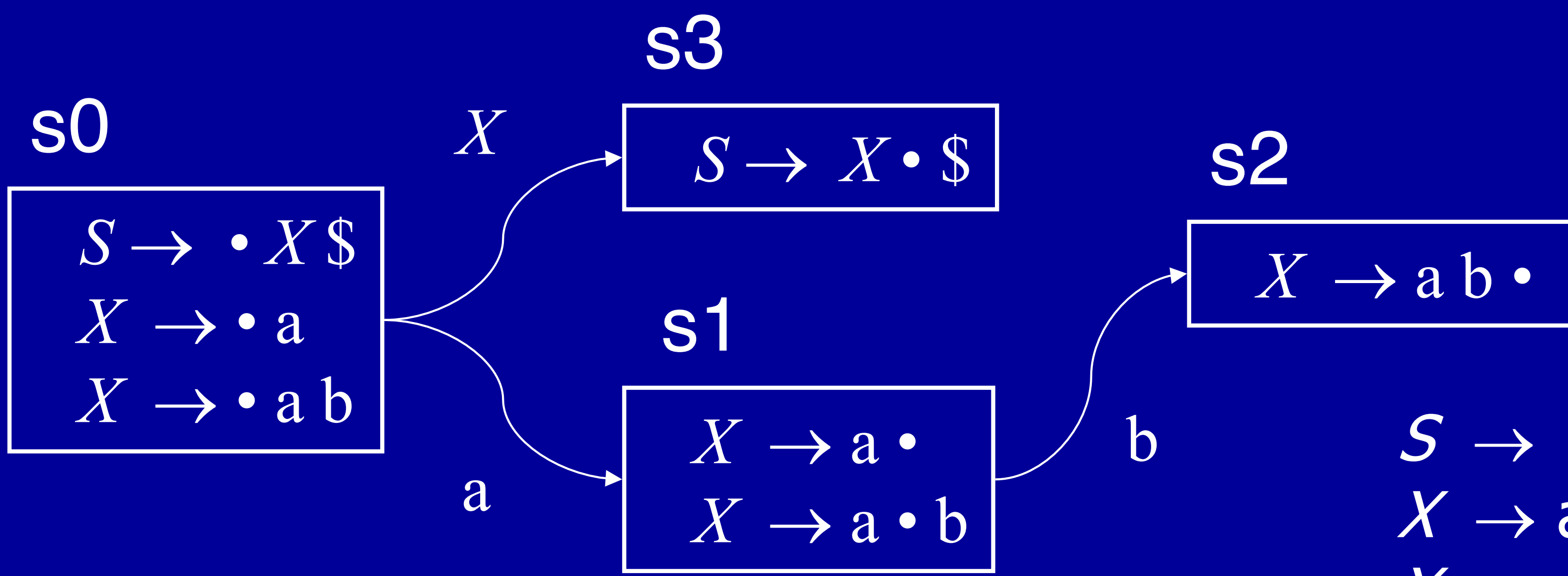
$$S \rightarrow X\$$$

$$X \rightarrow a$$

$$X \rightarrow a\ b$$

Parser Without Lookahead

	ACTION			Goto
State	**a**	**b**	**$**	**X**
s0	shift to s1	error	error	goto s3
s1	reduce(2)	S/R Conflict	reduce(2)	
s2	reduce(3)	reduce(3)	reduce(3)	
s3	error	error	accept	



$S \to X\,\$$
$X \to a$
$X \to a\, b$

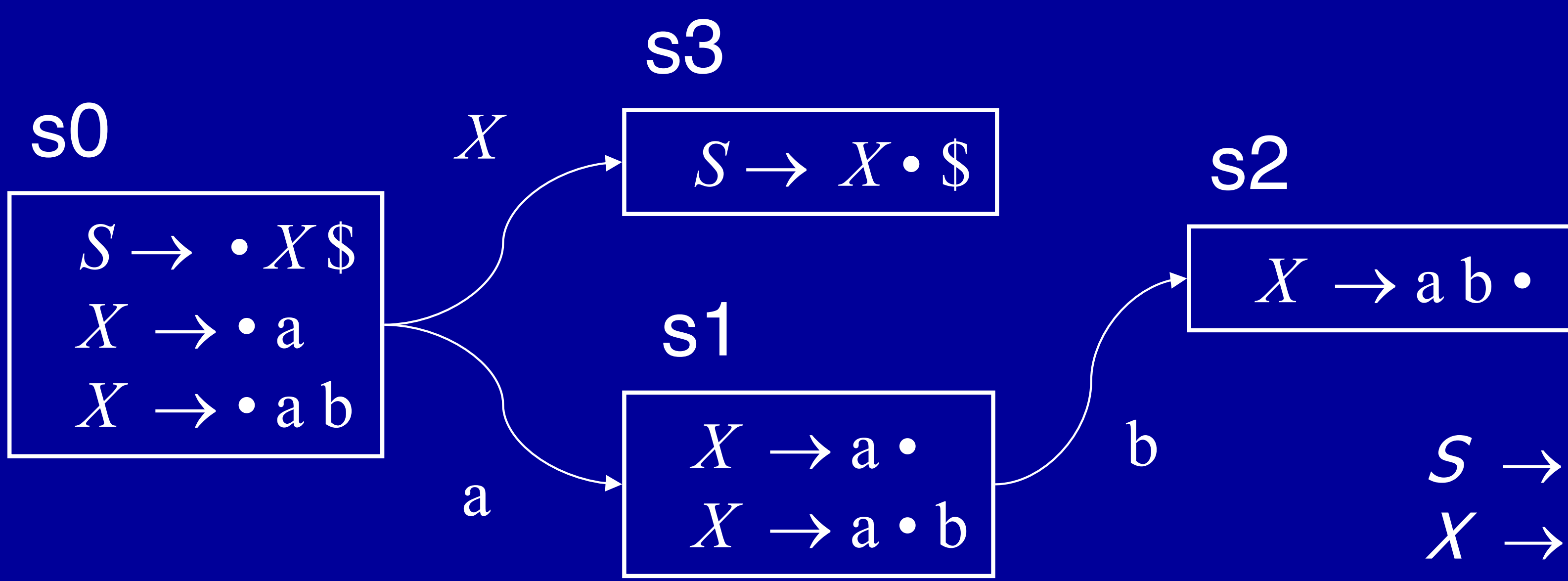
Creating parse tables with reduction-only lookahead

- For each state
  - Transition to another state using a terminal symbol is a shift to that state (*shift to sn*) (same as before)
  - Transition to another state using a non-terminal is a goto that state (*goto sn*) (same as before)
  - If there is an item $X \rightarrow \alpha \bullet$ in the state
    do a reduction with that production whenever the current input symbol T may follow X in some derivation (more precise than before)
- Eliminates useless reduce actions

New Parse Table

b never follows X in any derivation
resolve shift/reduce conflict to shift

		ACTION		Goto
State	**a**	**b**	**$**	**X**
s0	shift to s1	error	error	goto s3
s1	reduce(2)	shift to s2	reduce(2)	
s2	reduce(3)	reduce(3)	reduce(3)	
s3	error	error	accept	



$$S \to X\$$$
$$X \to a$$
$$X \to a\ b$$

More General Lookahead

- Items contain potential lookahead information, resulting in more states in finite state control
- Item of the form $[A \rightarrow \alpha \bullet \beta \quad T]$ says
  - The parser has parsed an α
  - If it parses a β and the next symbol is T
  - Then parser should reduce by $A \rightarrow \alpha\ \beta$
- In addition to current parser state, all parser actions are function of lookahead symbols

Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

  - **L** - parse from left to right
  - **R** - parse from right to left

Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

  - **L** - leftmost derivation
  - **R** - rightmost derivation

Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

  - Number of lookahead characters

□□ (□)

Terminology

- Many different parsing techniques
  - Each can handle some set of CFGs
  - Categorization of techniques

  - Examples: LL(0), LR(1)
  - This lecture
    - LR(0) parser
    - SLR parser – LR(0) parser augmented with follow information

$$\boxed{\mathbf{L}}\boxed{\mathbf{R}}\,(\boxed{\mathbf{k}})$$

Summary

- Parser generators – given a grammar, produce a parser
- Standard technique
  - Automatically build a pushdown automaton
  - Obtain a shift-reduce parser
    - Finite state control plus push down stack
    - Table driven implementation
- Conflicts: Shift/Reduce, Reduce/Reduce
- Use of lookahead to eliminate conflicts
  - SLR parsing (eliminates useless reduce actions)
  - LR(k) parsing (lookahead throughout parser)

Follow() sets in SLR Parsing

For each non-terminal *A*, Follow(*A*) is the set of terminals that can come after *A* in some derivation

Constraints for Follow()

- $\$ \in$ Follow(S), where S is the start symbol
- If $A \to \alpha B \beta$ is a production then First(β) $\subseteq$ Follow(B)
- If $A \to \alpha B$ is a production then Follow(A) $\subseteq$ Follow(B)
- If $A \to \alpha B \beta$ is a production and β derives ε then Follow(A) $\subseteq$ Follow(B)

Algorithm for Follow

for all nonterminals NT

  Follow(NT) = {}

Follow(S) = { \$ }

while Follow sets keep changing

  for all productions $A \rightarrow \alpha B \beta$

    Follow(B) = Follow(B) $\cup$ First(β)

    if (β derives ε) Follow(B) = Follow(B) $\cup$ Follow(A)

  for all productions $A \rightarrow \alpha B$

    Follow(B) = Follow(B) $\cup$ Follow(A)

Augmenting Example with Follow

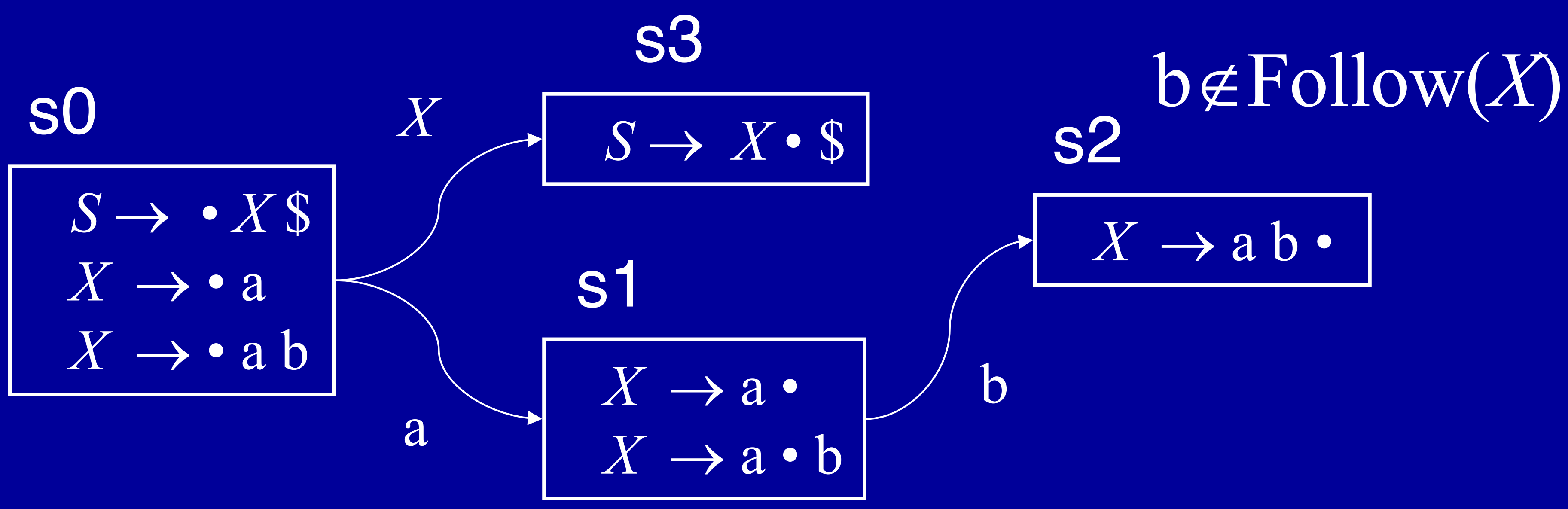
- Example Grammar for Follow

$$S \rightarrow X\$$$
$$X \rightarrow \mathrm{a}$$
$$X \rightarrow \mathrm{a\ b}$$

$$\mathrm{Follow}(S) = \{ \$ \}$$
$$\mathrm{Follow}(X) = \{ \$ \}$$

SLR Eliminates Shift/Reduce Conflict

		ACTION		Goto
State	**a**	**b**	**$**	**X**
s0	shift to s1	error	error	goto s3
s1	reduce(2)	shift to s2	reduce(2)	
s2	reduce(3)	reduce(3)	reduce(3)	
s3	error	error	accept	



Basic Idea Behind LR(1)

- Split states in LR(0) DFA based on lookahead
- Reduce based on item and lookahead

LR(1) Items

- Items will keep info on
  - production
  - right-hand-side position (the dot)
  - look ahead symbol
- LR(1) item is of the form [$A \to \alpha \bullet \beta$ T]
  - $A \to \alpha\ \beta$ is a production
  - The dot in $A \to \alpha \bullet \beta$ denotes the position
  - T is a terminal or the end marker ($)

Meaning of LR(1) Items

- Item $[A \rightarrow \alpha \bullet \beta \quad T]$ means
  - The parser has parsed an α
  - If it parses a β and the next symbol is T
  - Then parser should reduce by $A \rightarrow \alpha\ \beta$

- The grammar

  $S \rightarrow X\$$

  $X \rightarrow (X)$

  $X \rightarrow \varepsilon$

- Terminal symbols
  - '(' ')'
- End of input symbol
  - '$'

LR(1) Items

$[S \rightarrow \bullet X\$ \quad)]$

$[S \rightarrow \bullet X\$ \quad ( ]$

$[S \rightarrow \bullet X\$ \quad \$]$

$[S \rightarrow X \bullet \$ \quad)]$

$[S \rightarrow X \bullet \$ \quad ( ]$

$[S \rightarrow X \bullet \$ \quad \$]$

$[X \rightarrow \bullet (X) \quad)]$

$[X \rightarrow \bullet (X) \quad ( ]$

$[X \rightarrow \bullet (X) \quad \$]$

$[X \rightarrow (\bullet X) \quad)]$

$[X \rightarrow (\bullet X) \quad ( ]$

$[X \rightarrow (\bullet X) \quad \$]$

$[X \rightarrow (X \bullet \quad)]$

$[X \rightarrow (X \bullet) \quad ( ]$

$[X \rightarrow (X \bullet) \quad \$]$

$[X \rightarrow (X) \bullet \quad)]$

$[X \rightarrow (X) \bullet \quad ( ]$

$[X \rightarrow (X) \bullet \quad \$]$

$[X \rightarrow \bullet \quad)]$

$[X \rightarrow \bullet \quad ( ]$

$[X \rightarrow \bullet \quad \$]$

Creating a LR(1) Parser Engine

- Need to define Closure() and Goto() functions for LR(1) items
- Need to provide an algorithm to create the DFA
- Need to provide an algorithm to create the parse table

Closure algorithm

Closure(I)

repeat

for all items $[A \rightarrow \alpha \bullet X \beta \quad c]$ in I

for any production $X \rightarrow \gamma$

for any $d \in First(\beta c)$

$I = I \cup \{ [X \rightarrow \bullet \gamma \quad d] \}$

until I does not change

Goto algorithm

Goto(I, X)

J = { }

for any item [A $\rightarrow \alpha \bullet X \beta$ c] in I

J = J $\cup$ {[A $\rightarrow \alpha X \bullet \beta$ c]}

return Closure(J)

Building the LR(1) DFA

- Start with the item $[\langle S' \rangle \rightarrow \bullet \langle S \rangle \ \$ \quad I]$
  - I irrelevant because we will never shift \$
- Find the closure of the item and make an state
- Pick a state I
  - for each item $[A \rightarrow \alpha \bullet X \beta \quad c]$ in I
    - find Goto(I, X)
    - if Goto(I, X) is not already a state, make one
    - Add an edge X from state I to Goto(I, X) state
- Repeat until no more additions possible

Creating the parse tables

- For each LR(1) DFA state
  - Transition to another state using a terminal symbol is a shift to that state (*shift to sn*)
  - Transition to another state using a non-terminal symbol is a goto that state (*goto sn*)
  - If there is an item $[A \rightarrow \alpha \bullet\ a]$ in the state, action for input symbol a is a reduction via the production $A \rightarrow \alpha$ (*reduce k*)

LALR(1) Parser

- Motivation
  - LR(1) parse engine has a large number of states
  - Simple method to eliminate states
- If two LR(1) states are identical except for the look ahead symbol of the items
  Then Merge the states
- Result is LALR(1) DFA
- Typically has many fewer states than LR(1)
- May also have more reduce/reduce conflicts