© Manuel Cargaleiro

Syntactic Analysis II

Masters in Informatics and Computing Engineering (MIEIC), 3rd Year

João M. P. Cardoso
Dep. de Engenharia Informática, Faculdade de Engenharia (FEUP), Universidade do Porto, Porto, Portugal
Email: jmpc@fe.up.pt

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

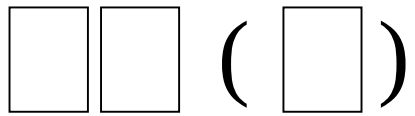
DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Previously we have seen

- *Top-Down Parser*
- Use of *Lookahead* to avoid *Backtracking*
- Parser as a set of procedures mutually recursive

Terminology

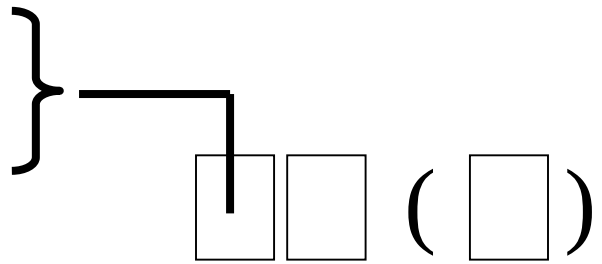
- There are many techniques for syntactic analysis
  - Each one can deal with some set of CFGs
  - Categorization of techniques



Terminology

- There are many techniques for syntactic analysis
  - Each one can deal with some set of CFGs
  - Categorization of techniques

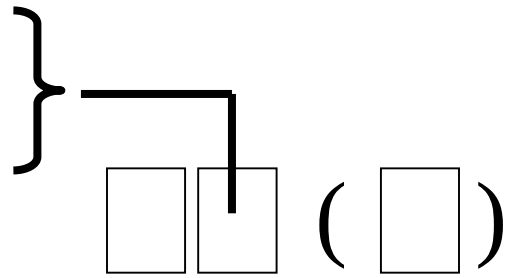
  - **L** – analysis from left to right
  - **R** - analysis from right to left



Terminology

- There are many techniques for syntactic analysis
  - Each one can deal with some set of CFGs
  - Categorization of techniques

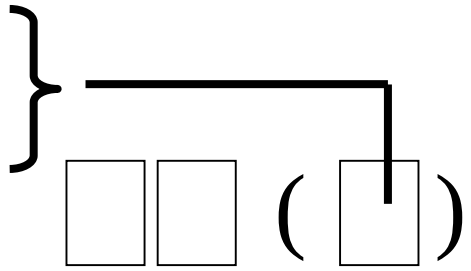
  - **L** – leftmost derivation
  - **R** – rightmost derivation



Terminology

- There are many techniques for syntactic analysis
  - Each one can deal with some set of CFGs
  - Categorization of techniques

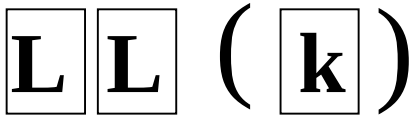
  - Lookahead value



Terminology

- There are many techniques for syntactic analysis
  - Each one can deal with some set of CFGs
  - Categorization of techniques

  - Examples: LL(1), LR(0)
  - Up to now we have seen: LL(k)
  - In the following classes
    - LR(k) analysis



Terminology

- LL(k)
  - *Top-down*, predictive
  - Leftmost derivation from top to bottom
- LR(k)
  - *Bottom-up*, *shift-reduce*
  - Rightmost derivation from bottom to top

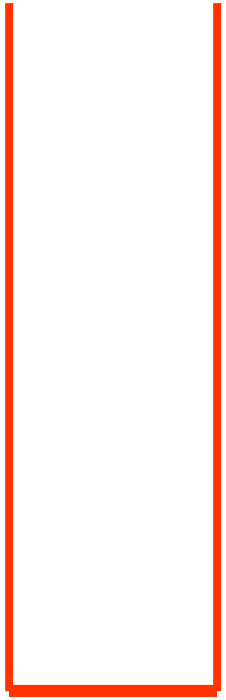
Bottom-up Syntactic Analyzer

- Central mechanism
  - Push-down automaton which implements
  - Shift-reduce parser

Push-Down Automaton

- Consists of
  - Pushdown stack (it can contain terminal and non-terminal symbols)
  - Control by a finite automaton
- It can do one of three actions:
  - Shift:
    - Shift current symbol in the input to the stack
  - Reduce:
    - If the symbol(s) in the top of the stack match(es) the RHS of any of the grammar productions
      - Pop of that symbol(s) from the stack
      - Push of the non-terminal of the LHS to the stack
  - Accepts the input as belonging to the language

Example: Shift-Reduce Parser



Stack

Expr → *Expr Op Expr*

Expr → (*Expr*)

Expr → - *Expr*

Expr → num

Op → +

Op → -

Op → *

Input string

num	*	(	num	+	num	)

Exemplo: Parser Shift-Reduce

Expr → *Expr Op Expr*

Expr → (*Expr*)

Expr → - *Expr*

Expr → num

Op → +

Op → -

Op → *

num	*	(	num	+	num	)

Example: Shift-Reduce Parser

Expr → *Expr Op Expr*

Expr → (*Expr*)

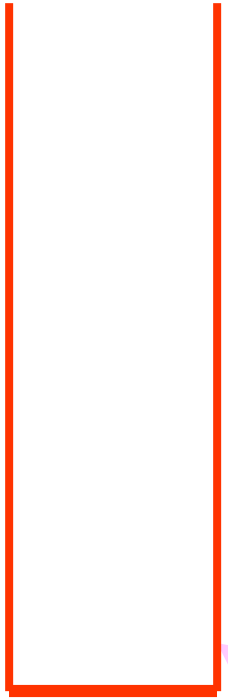
Expr → - *Expr*

Expr → num

Op → +

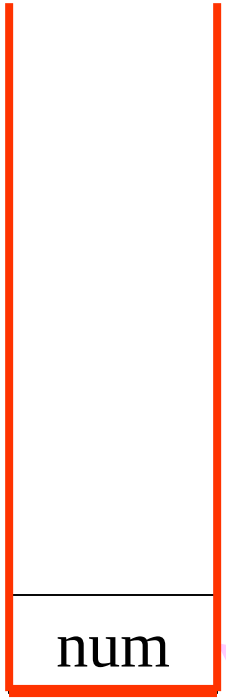
Op → -

Op → *



num	*	(	num	+	num	)

Example: Shift-Reduce Parser



SHIFT

Expr → *Expr Op Expr*

Expr → (*Expr*)

Expr → - *Expr*

Expr → num

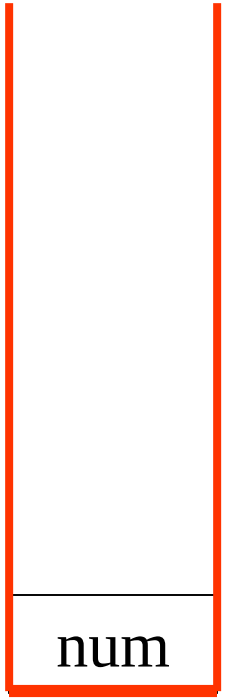
Op → +

Op → -

Op → *

*	(	num	+	num	)

Example: Shift-Reduce Parser



REDUCE

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow -\ Expr$

$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

*	(	num	+	num	)

Example: Shift-Reduce Parser

Expr → *Expr Op Expr*

Expr → (*Expr*)

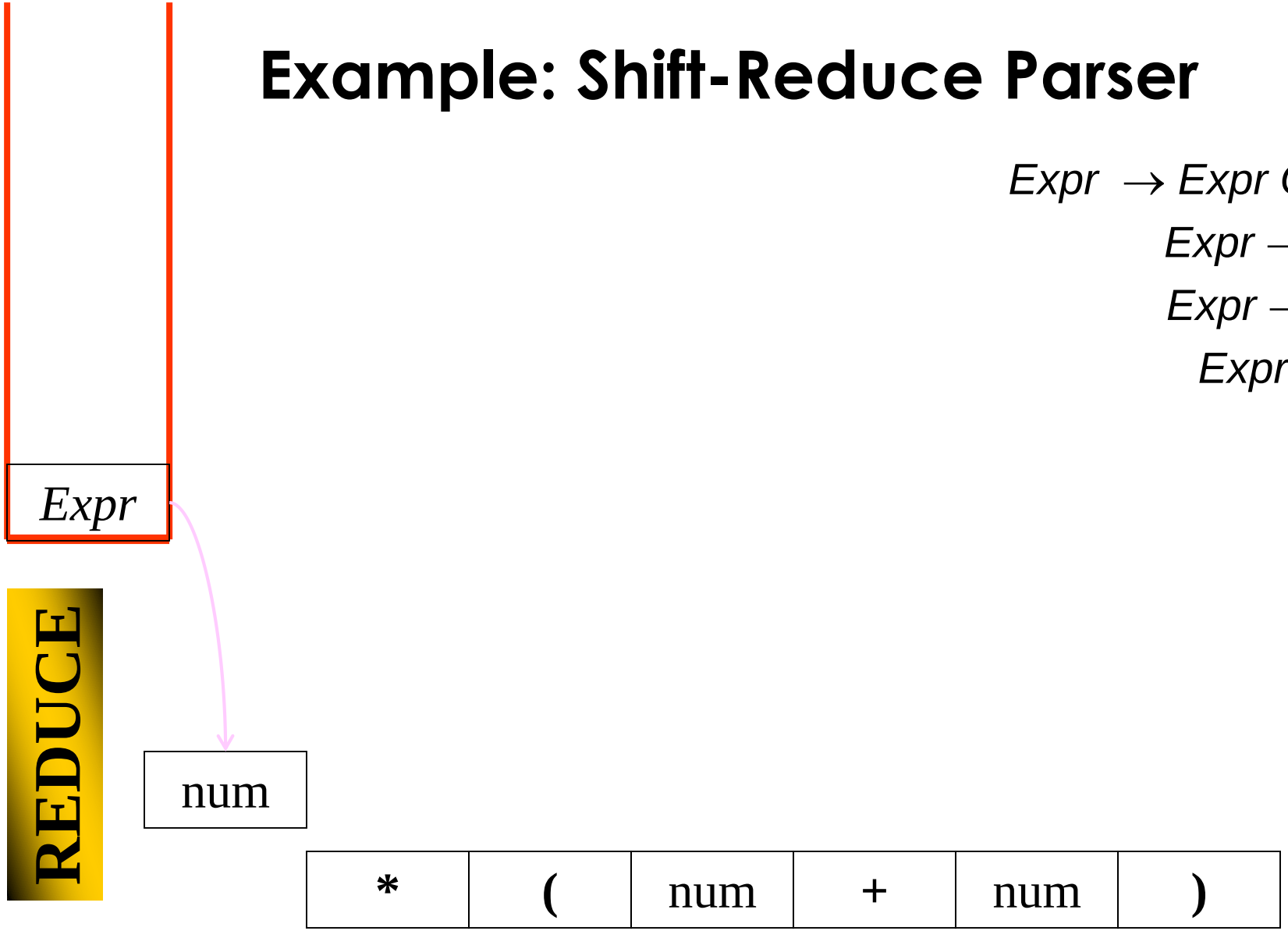
Expr → - *Expr*

Expr → num

Op → +

Op → -

Op → *



Example: Shift-Reduce Parser

Expr → *Expr Op Expr*

Expr → (*Expr*)

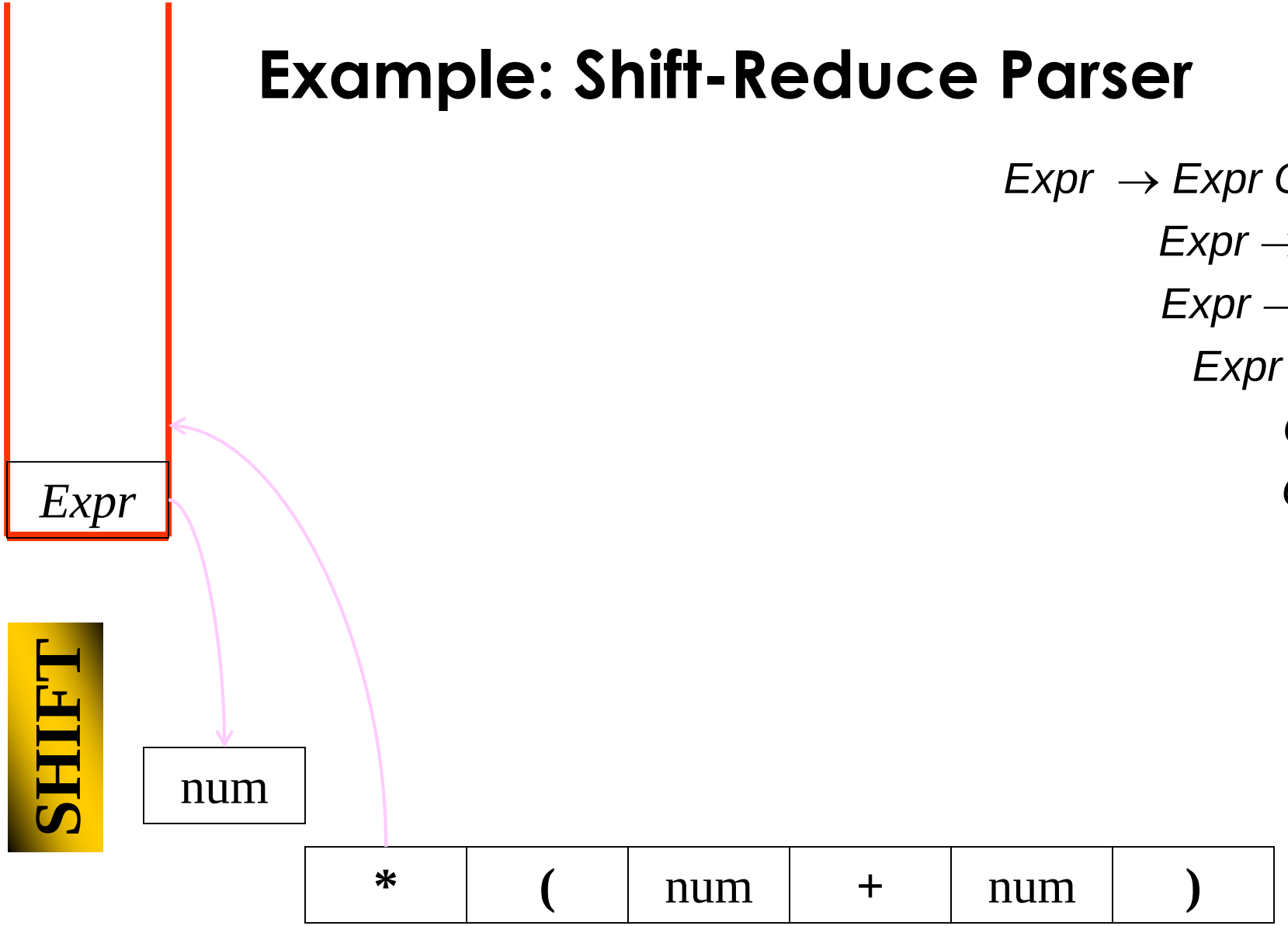
Expr → - *Expr*

Expr → num

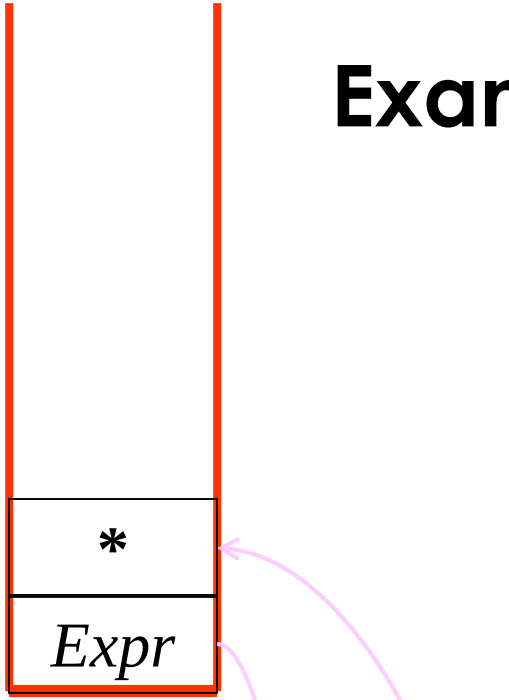
Op → +

Op → -

Op → *



Example: Shift-Reduce Parser



SHIFT

num

Expr → *Expr Op Expr*

Expr → (*Expr*)

Expr → - *Expr*

Expr → num

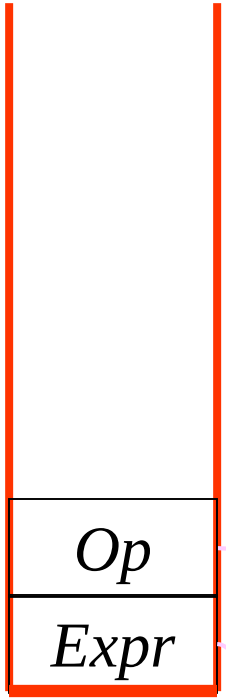
Op → +

Op → -

Op → *

(	num	+	num	)

Example: Shift-Reduce Parser



REDUCE

num	*

(	num	+	num	)

$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

$Expr \rightarrow\ -\ Expr$

$Expr \rightarrow\ \text{num}$

$Op \rightarrow\ +$

$Op \rightarrow\ -$

$Op \rightarrow\ *$

Example: Shift-Reduce Parser

Expr → *Expr Op Expr*

Expr → (*Expr*)

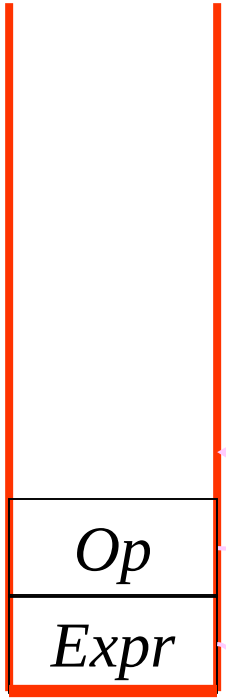
Expr → - *Expr*

Expr → num

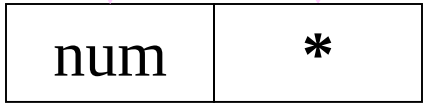
Op → +

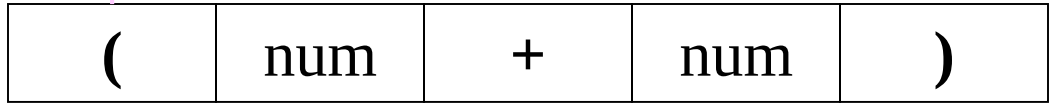
Op → -

Op → *

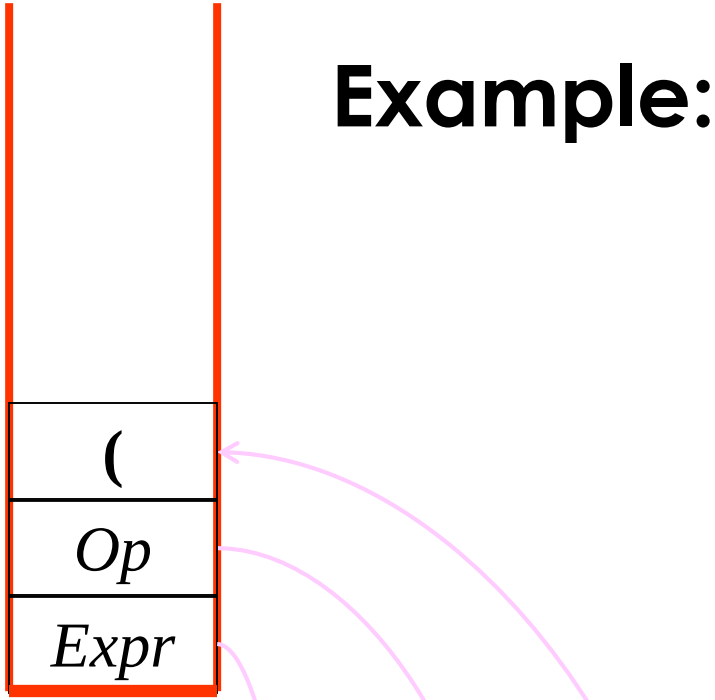


SHIFT





Example: Shift-Reduce Parser



SHIFT

num	*

num	+	num	)

Expr → *Expr Op Expr*

Expr → (*Expr*)

Expr → - *Expr*

Expr → num

Op → +

Op → -

Op → *

Example: Shift-Reduce Parser

Expr → *Expr Op Expr*

Expr → (*Expr*)

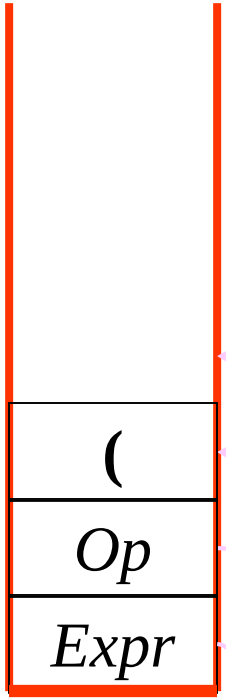
Expr → - *Expr*

Expr → num

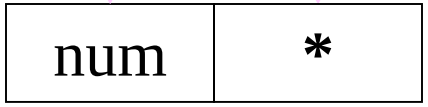
Op → +

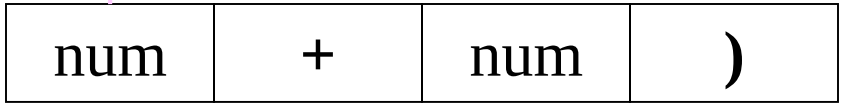
Op → -

Op → *

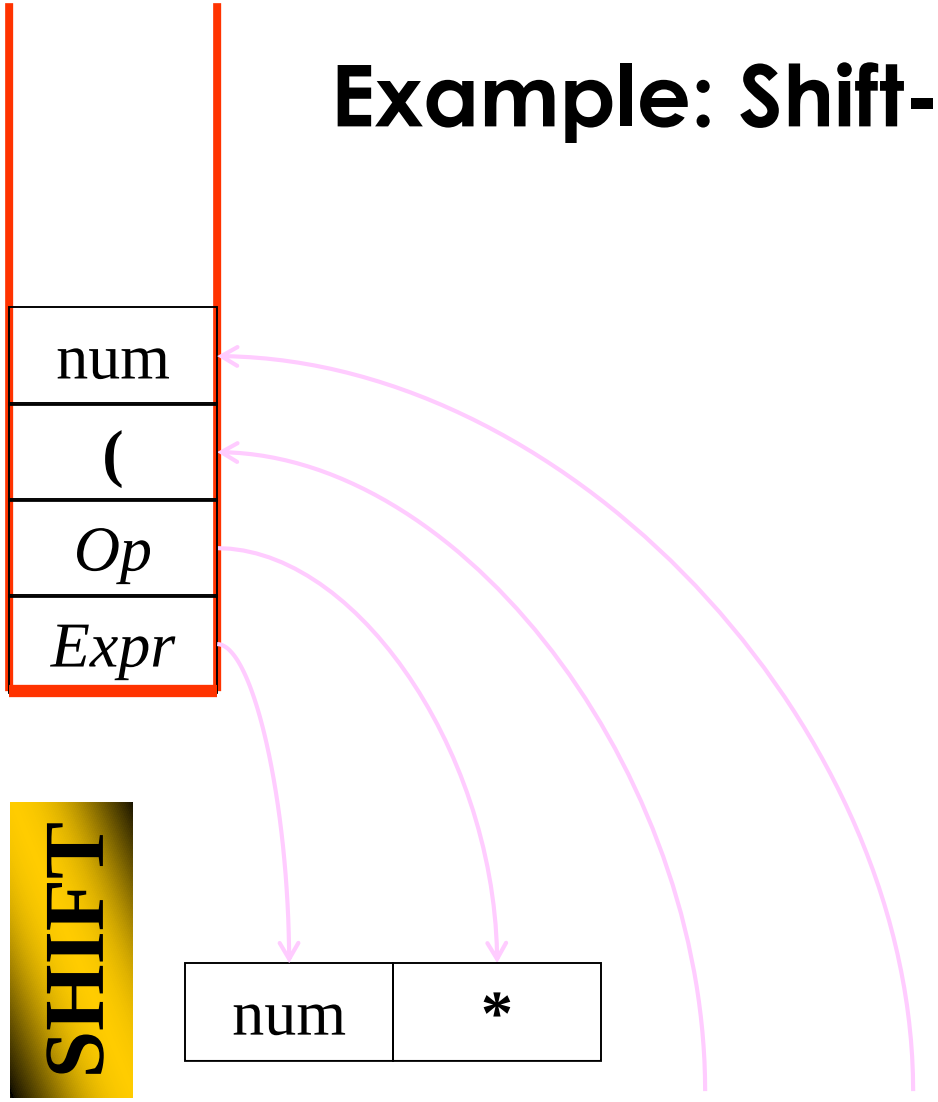


SHIFT





Example: Shift-Reduce Parser



Expr → *Expr Op Expr*

Expr → (*Expr*)

Expr → - *Expr*

Expr → num

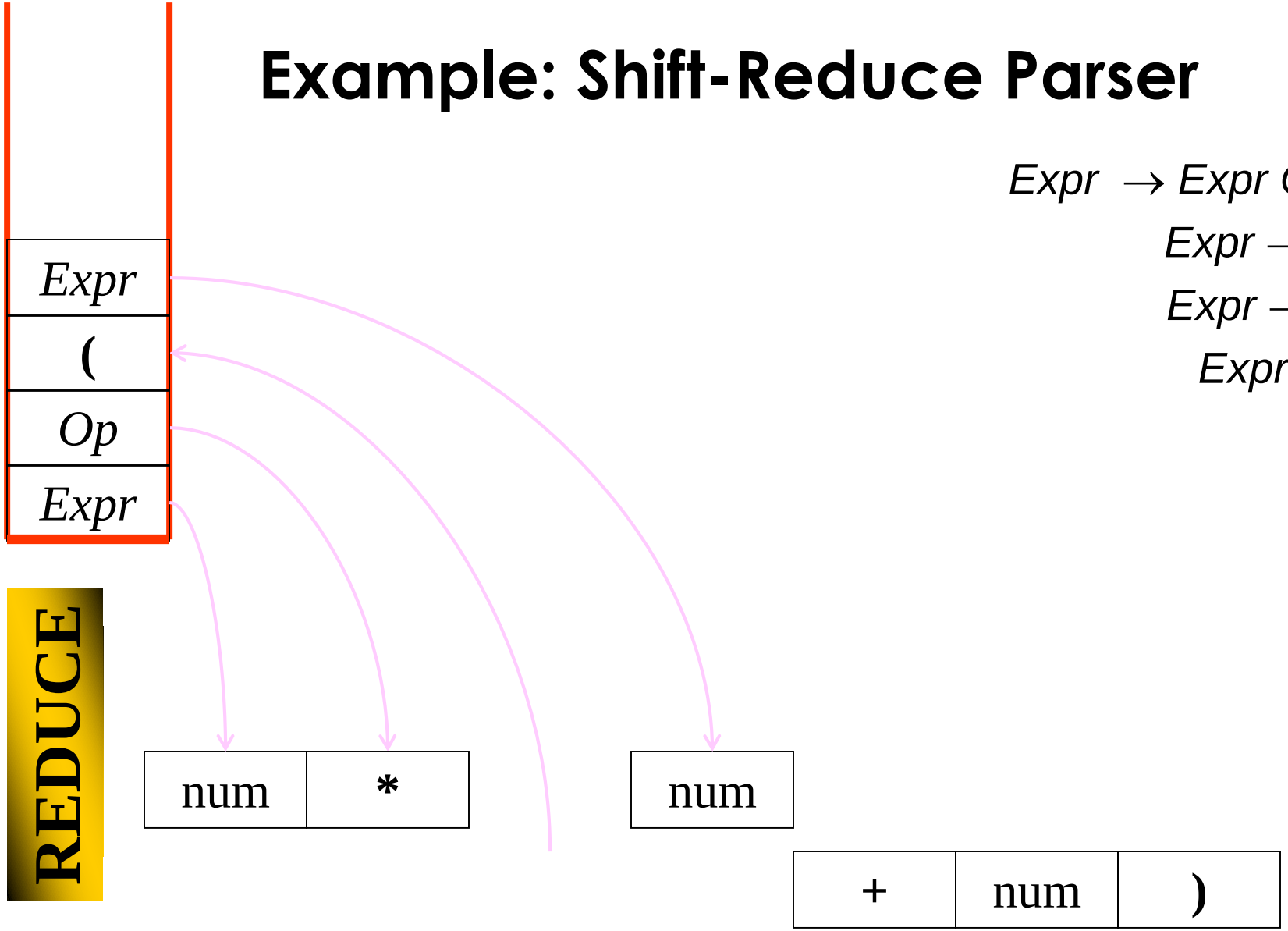
Op → +

Op → -

Op → *

+	num	)

Example: Shift-Reduce Parser



Expr → *Expr Op Expr*

Expr → (*Expr*)

Expr → - *Expr*

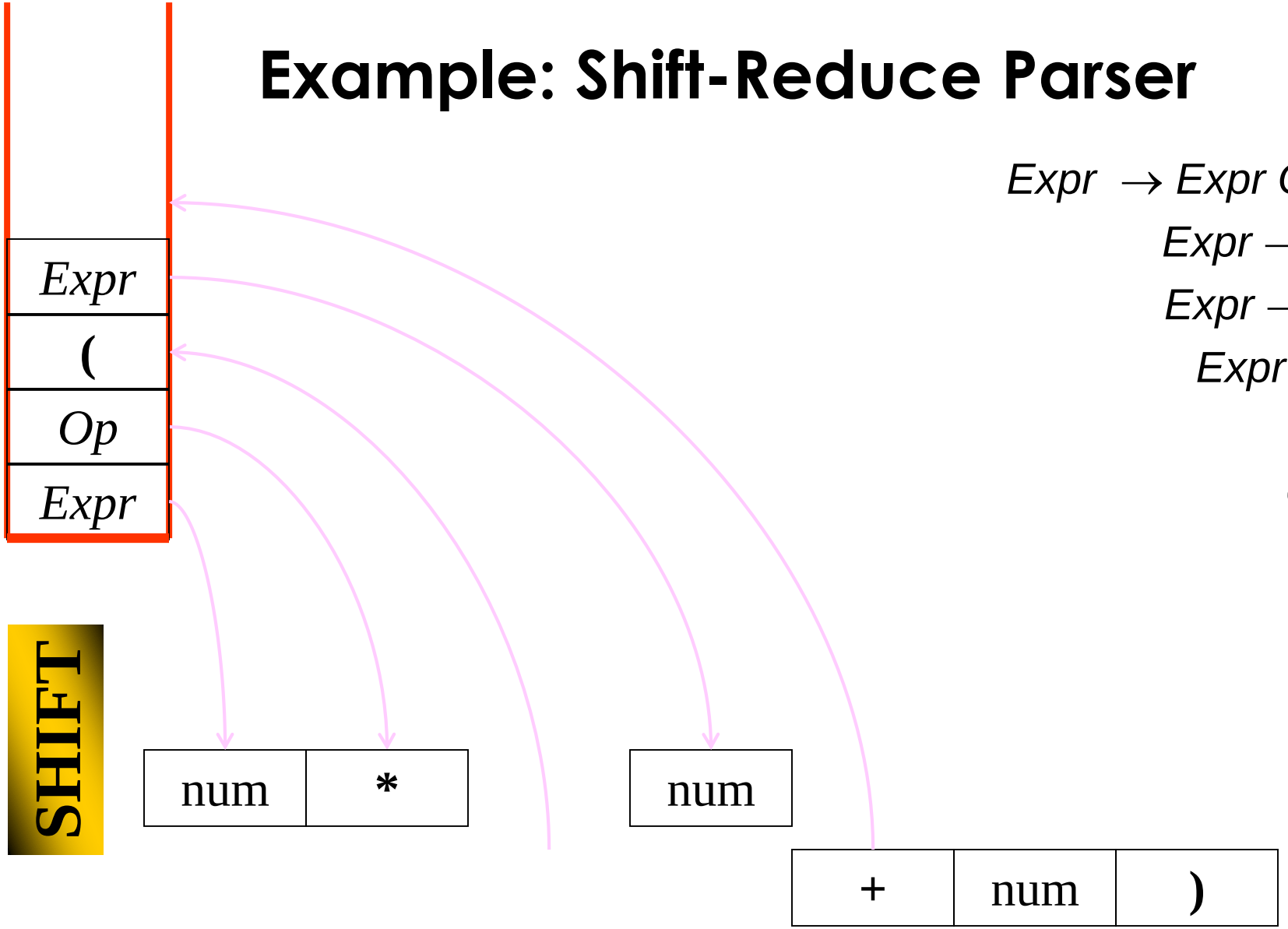
Expr → num

Op → +

Op → -

Op → *

Example: Shift-Reduce Parser



Expr → *Expr Op Expr*

Expr → (*Expr*)

Expr → - *Expr*

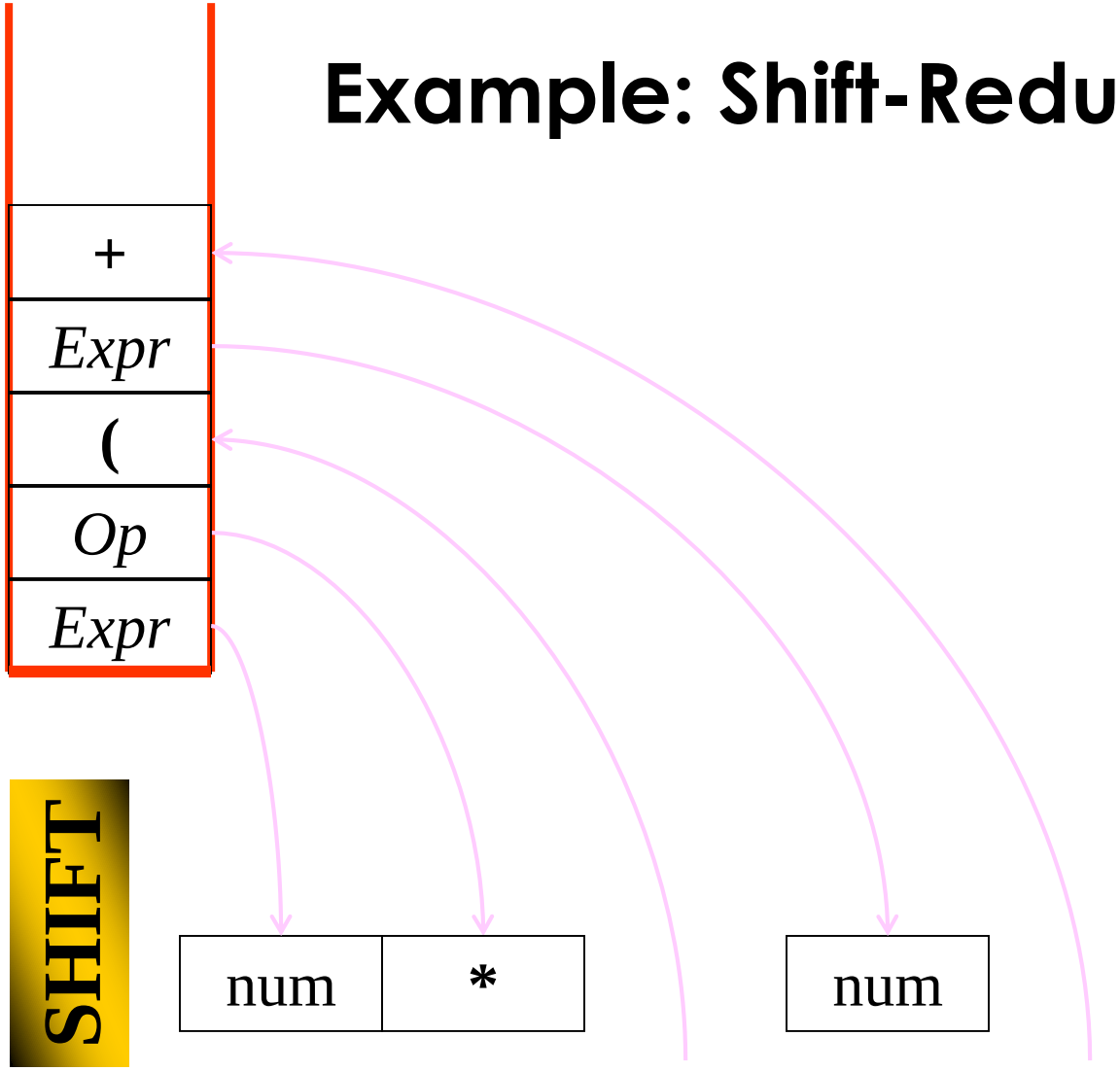
Expr → num

Op → +

Op → -

Op → *

Example: Shift-Reduce Parser



$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow$ - $Expr$

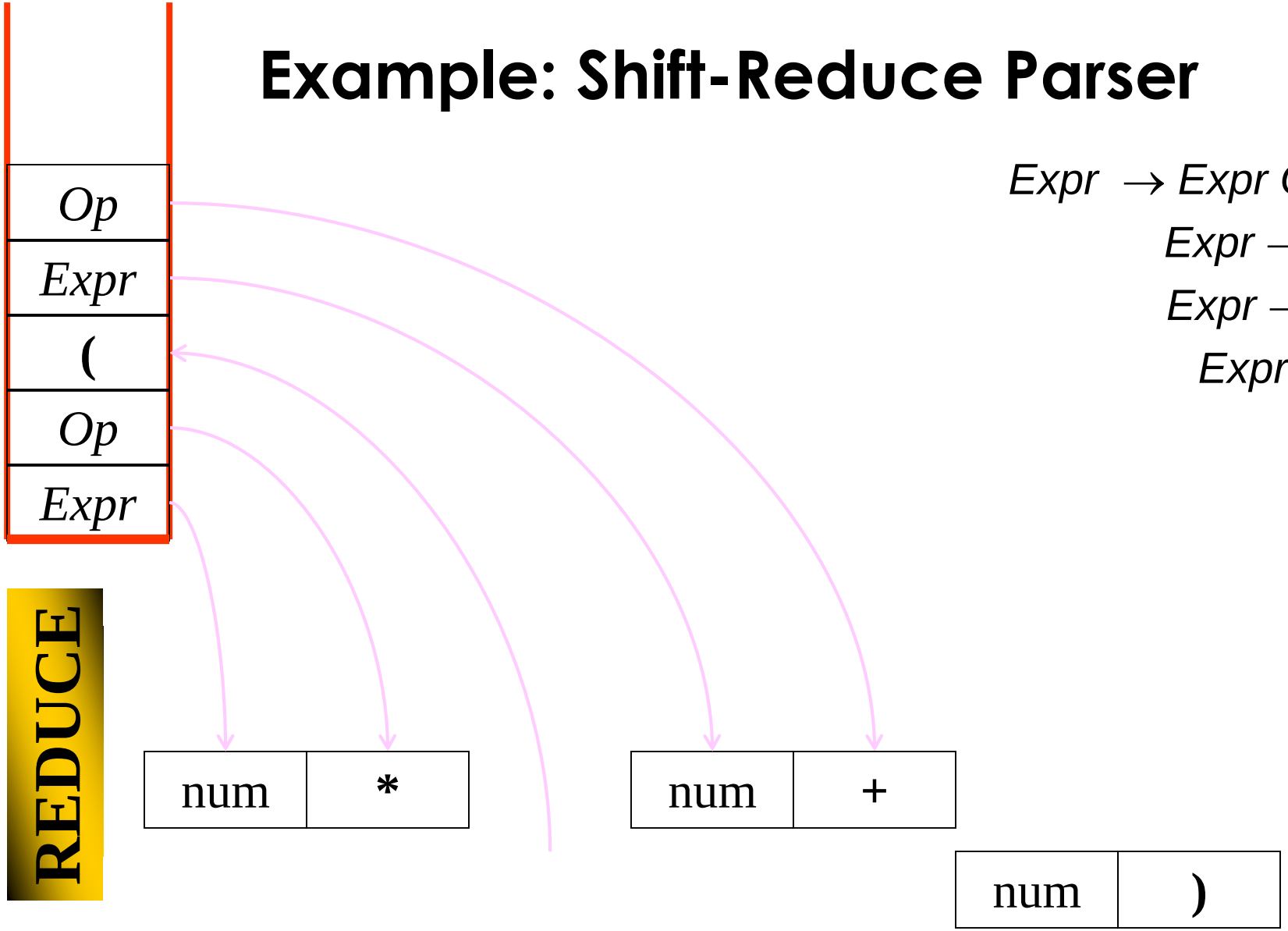
$Expr \rightarrow$ num

$Op \rightarrow$ +

$Op \rightarrow$ -

$Op \rightarrow$ *

Example: Shift-Reduce Parser



Expr → *Expr Op Expr*

Expr → (*Expr*)

Expr → - *Expr*

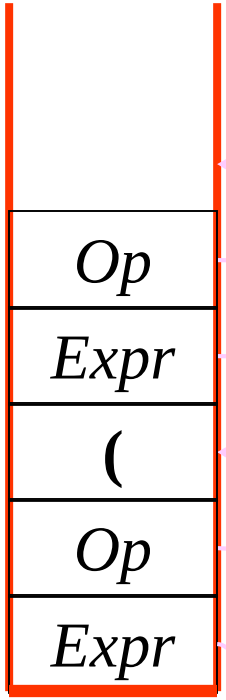
Expr → num

Op → +

Op → -

Op → *

Example: Shift-Reduce Parser



Expr → *Expr Op Expr*

Expr → (*Expr*)

Expr → - *Expr*

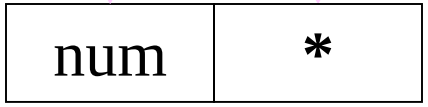
Expr → num

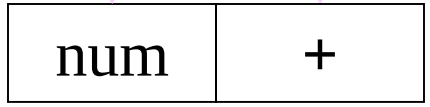
Op → +

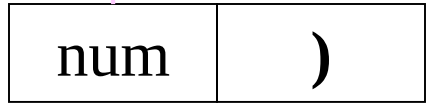
Op → -

Op → *

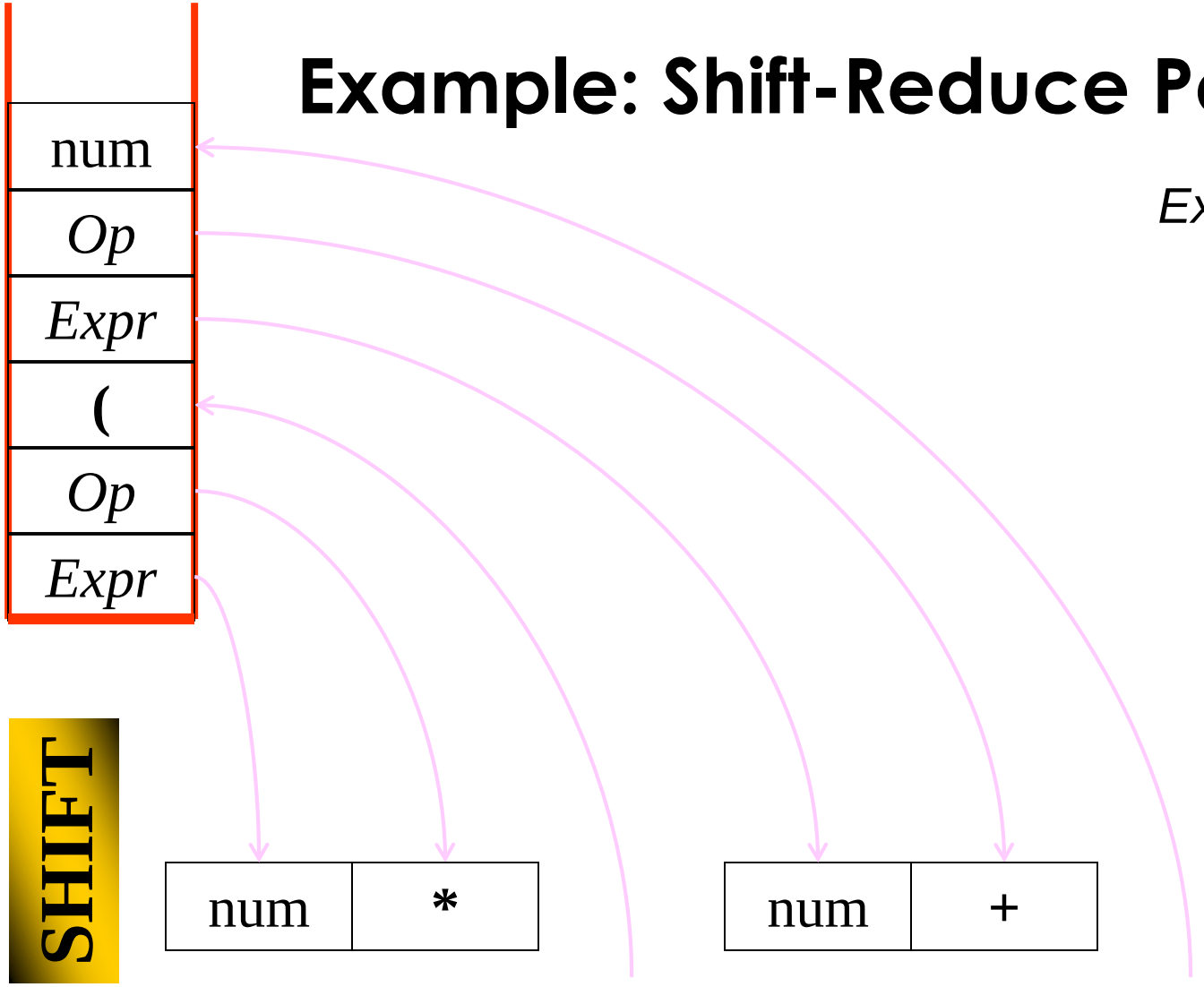
SHIFT







Example: Shift-Reduce Parser



Expr → *Expr Op Expr*

Expr → (*Expr*)

Expr → - *Expr*

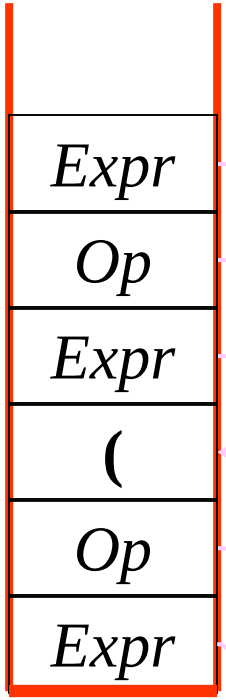
Expr → num

Op → +

Op → -

Op → *

Example: Shift-Reduce Parser



REDUCE

num	*

num	+	num

)

$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow$ - $Expr$

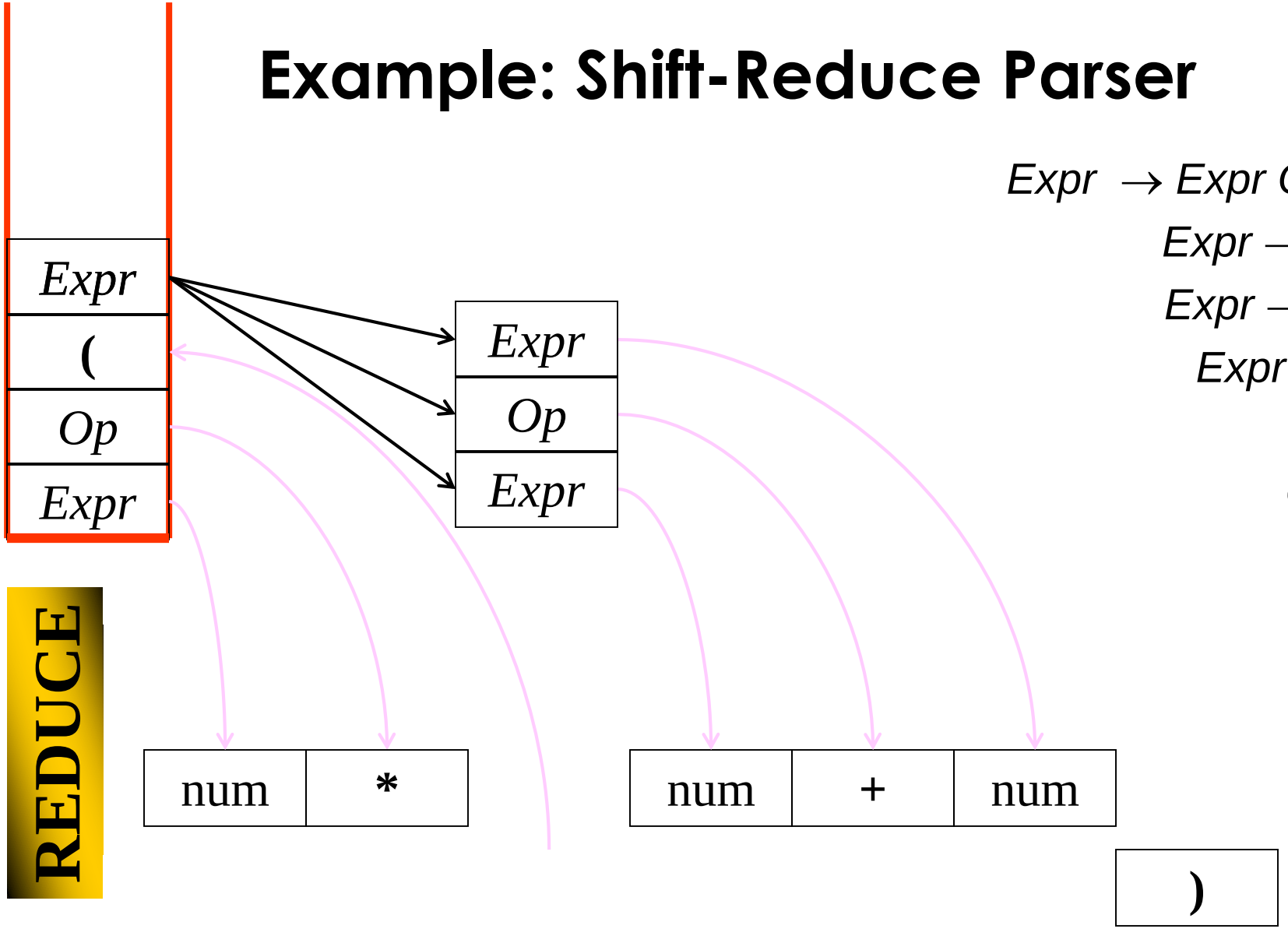
$Expr \rightarrow$ num

$Op \rightarrow$ +

$Op \rightarrow$ -

$Op \rightarrow$ *

Example: Shift-Reduce Parser



$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow -\ Expr$

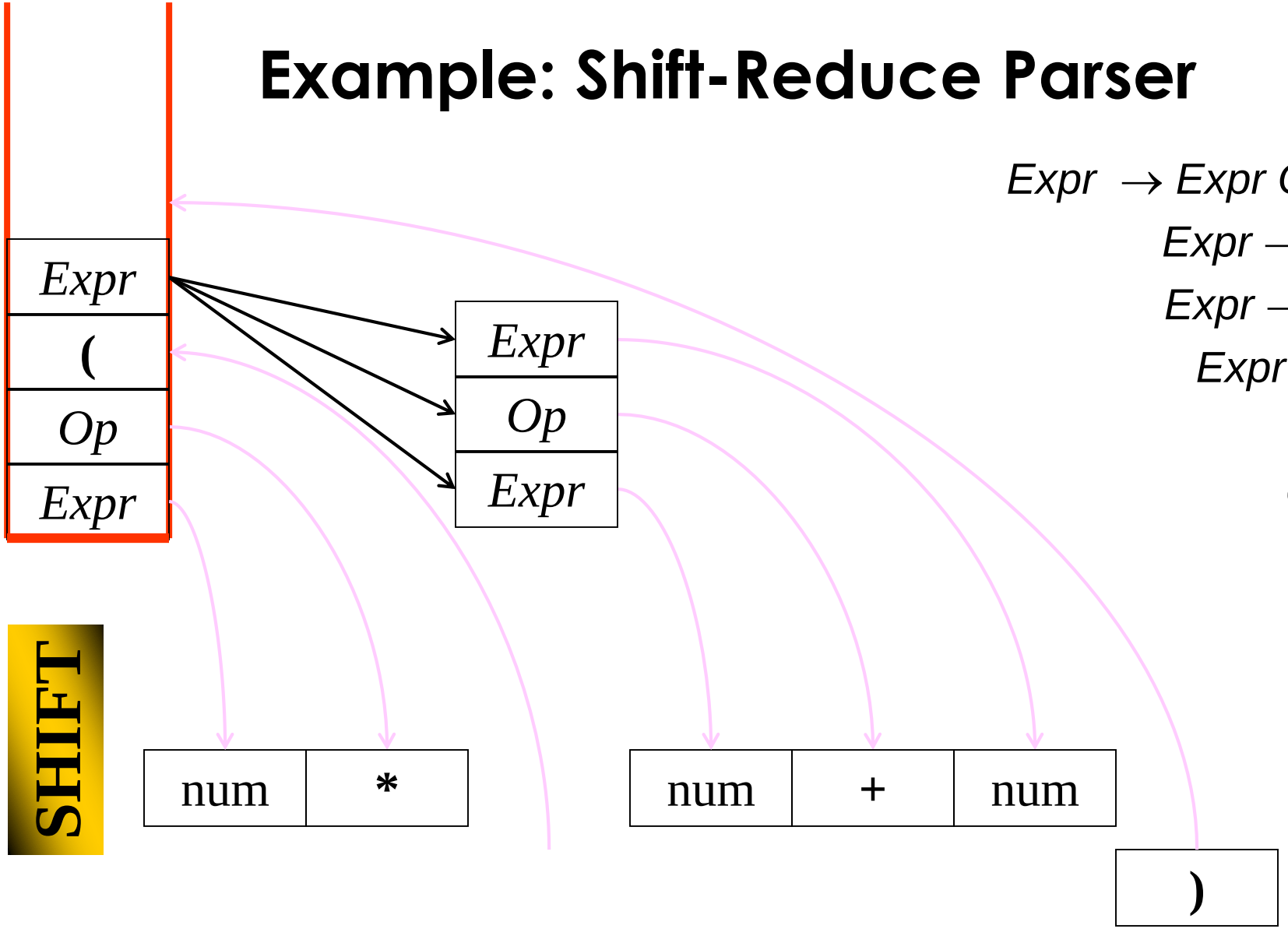
$Expr \rightarrow num$

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

Example: Shift-Reduce Parser



Expr → *Expr Op Expr*

Expr → (*Expr*)

Expr → - *Expr*

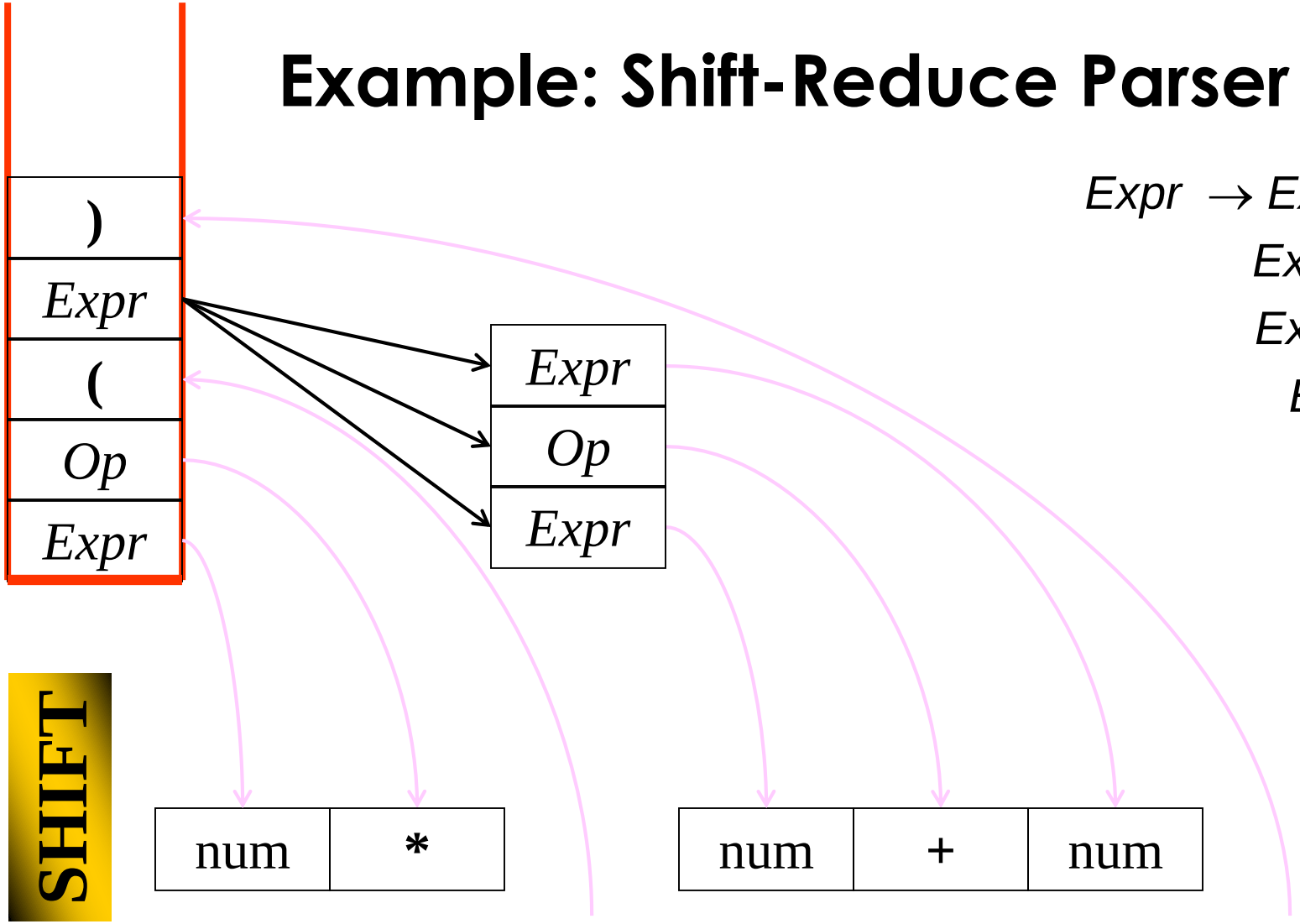
Expr → num

Op → +

Op → -

Op → *

Example: Shift-Reduce Parser



SHIFT

$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow\ (Expr)$

$Expr \rightarrow\ -\ Expr$

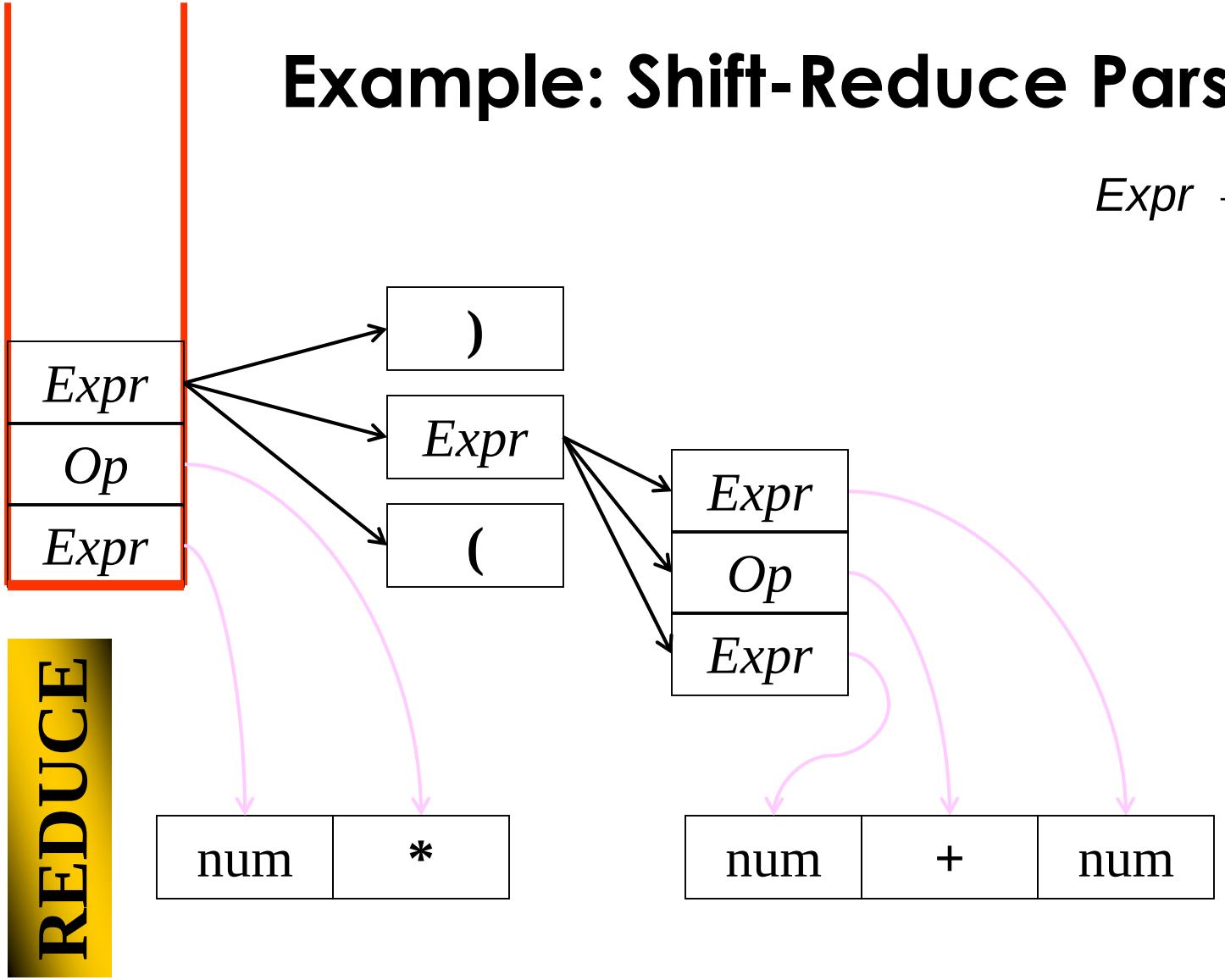
$Expr \rightarrow$ num

$Op \rightarrow\ +$

$Op \rightarrow\ -$

$Op \rightarrow\ *$

Example: Shift-Reduce Parser



Expr → *Expr Op Expr*

Expr → (*Expr*)

Expr → - *Expr*

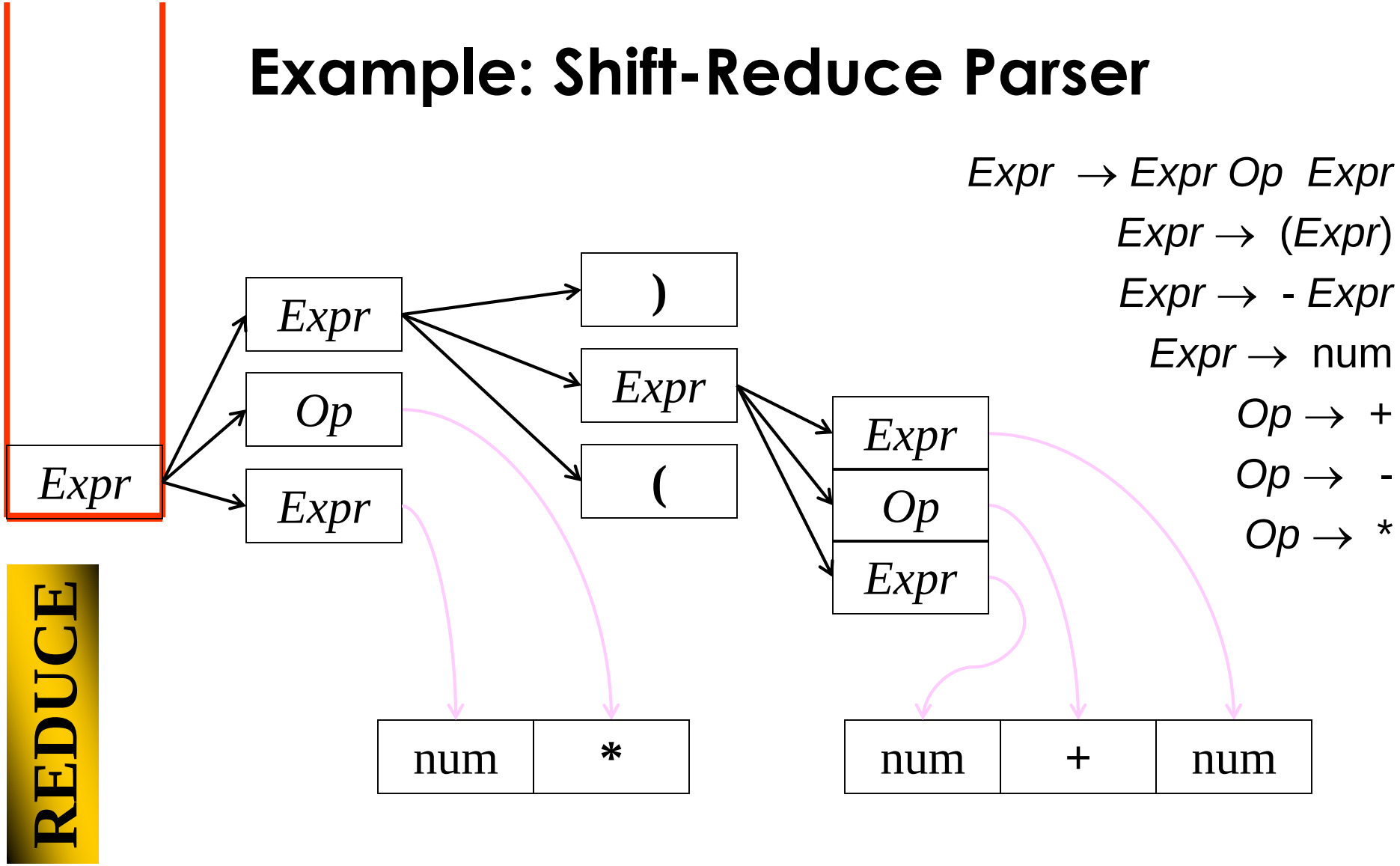
Expr → num

Op → +

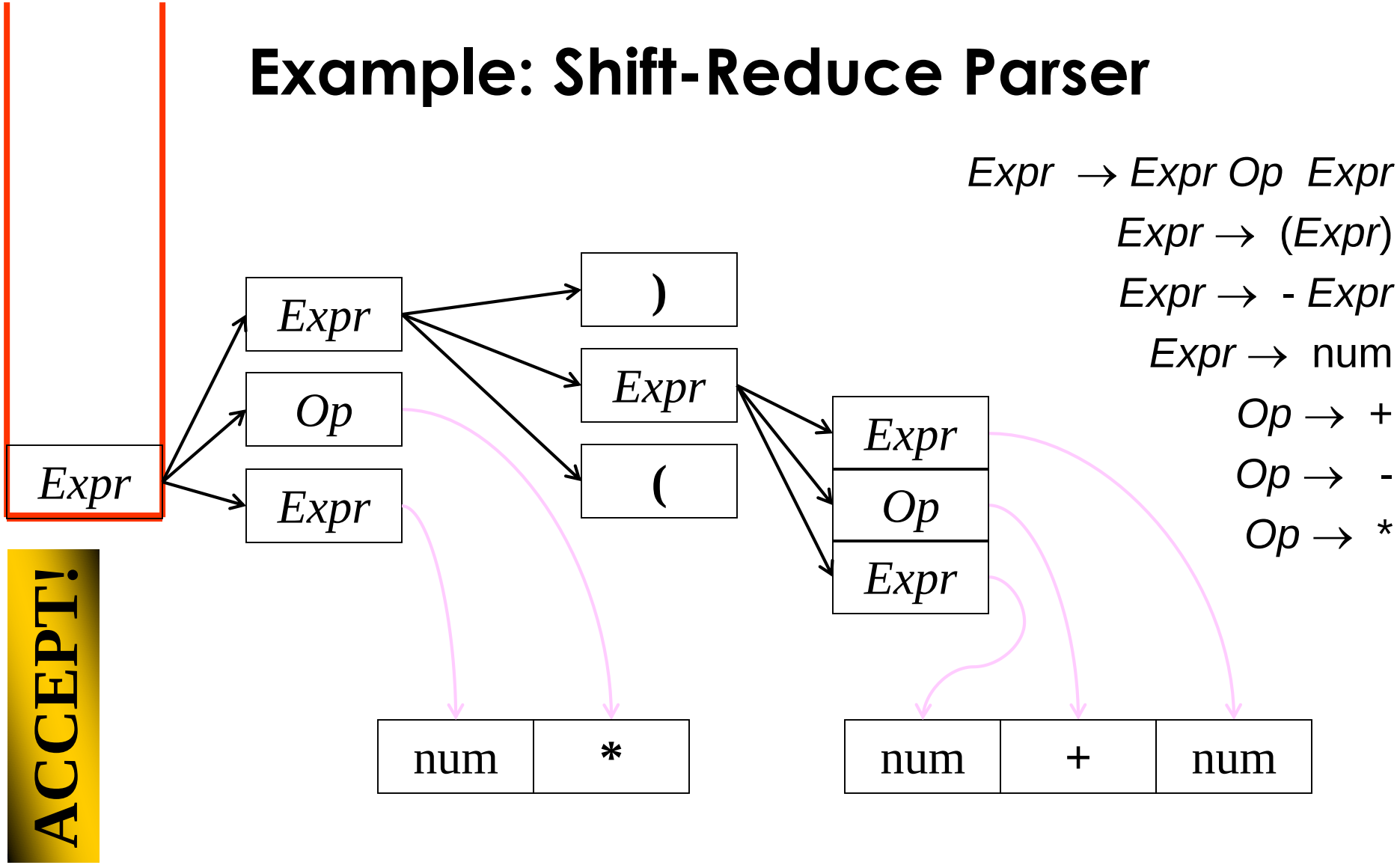
Op → -

Op → *

Example: Shift-Reduce Parser



Example: Shift-Reduce Parser



Conflicts that may occur

- Reduce/Reduce conflict
  - The top of the stack matches with RHS of multiple productions
  - What is the production to use in the reduction?
- Shift/Reduce conflict
  - The top of the stack matches the RHS of the production
  - But this might not be the perfect match
  - It can be necessary to shift the input and find latter on another reduction

Conflicts

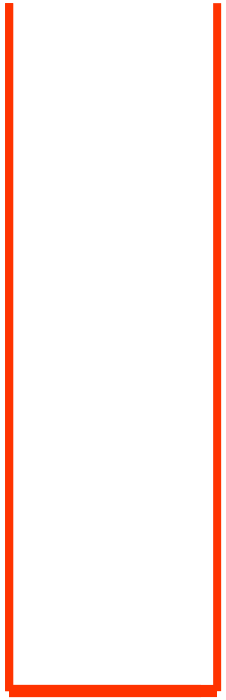
- Original grammar

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

- New grammar

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

Conflicts



Expr → *Expr Op Expr*

Expr → *Expr* - *Expr*

Expr → (*Expr*)

Expr → *Expr* -

Expr → num

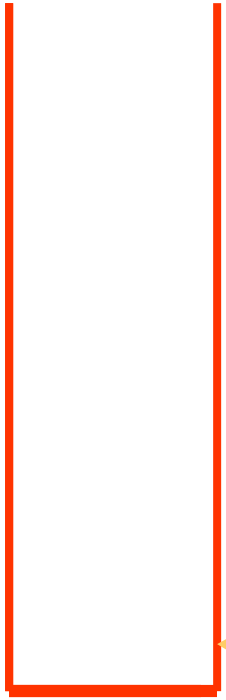
Op → +

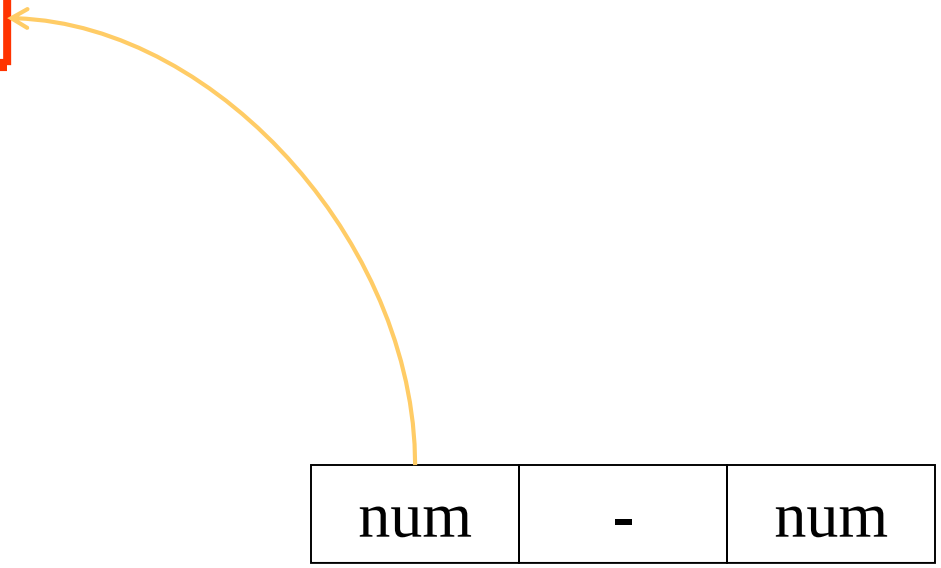
Op → -

Op → *

num	-	num

Conflicts





Expr → *Expr Op Expr*

Expr → *Expr* - *Expr*

Expr → (*Expr*)

Expr → *Expr* -

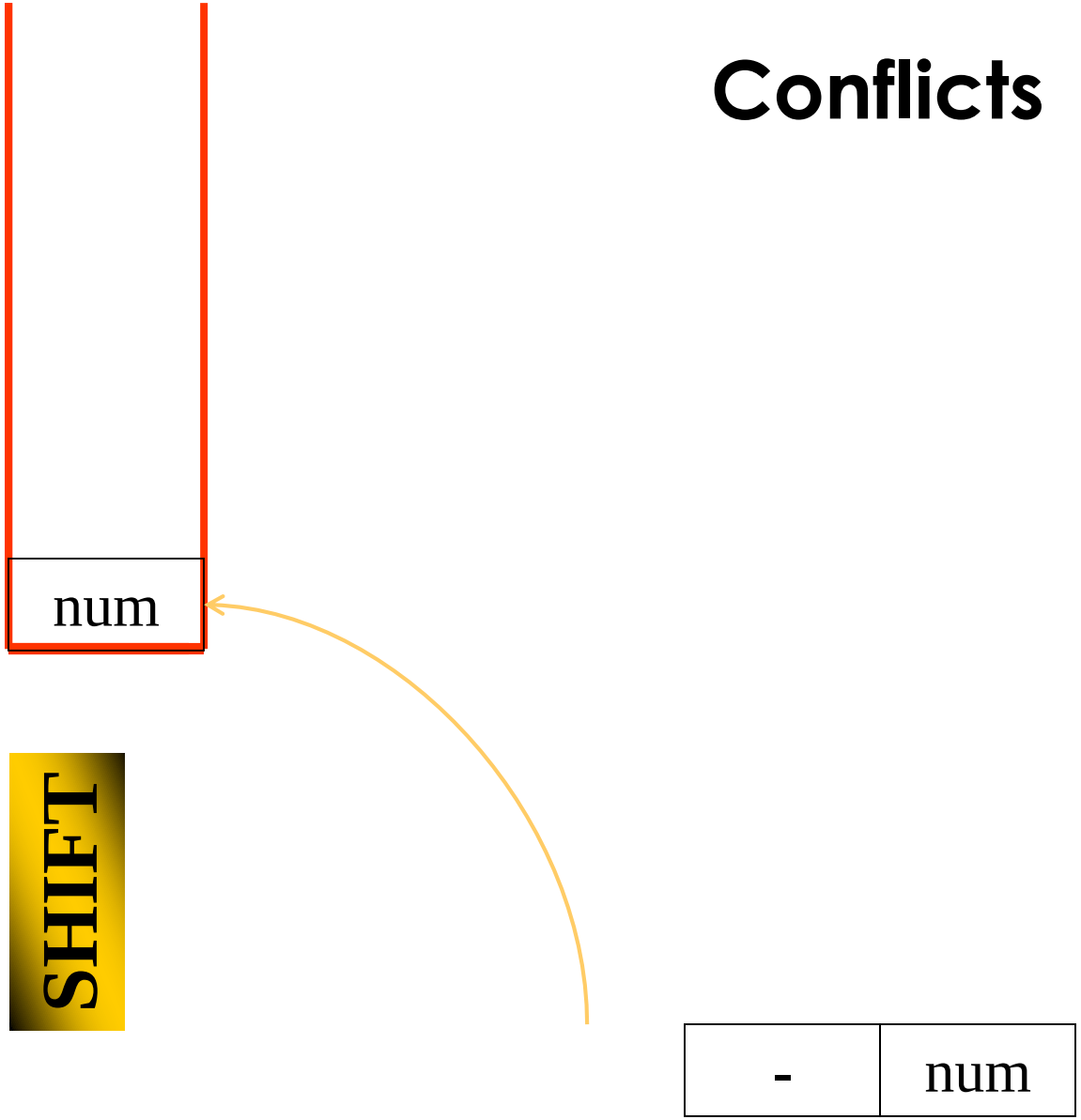
Expr → num

Op → +

Op → -

Op → *

Conflicts



Expr → *Expr Op Expr*

Expr → *Expr* - *Expr*

Expr → (*Expr*)

Expr → *Expr* -

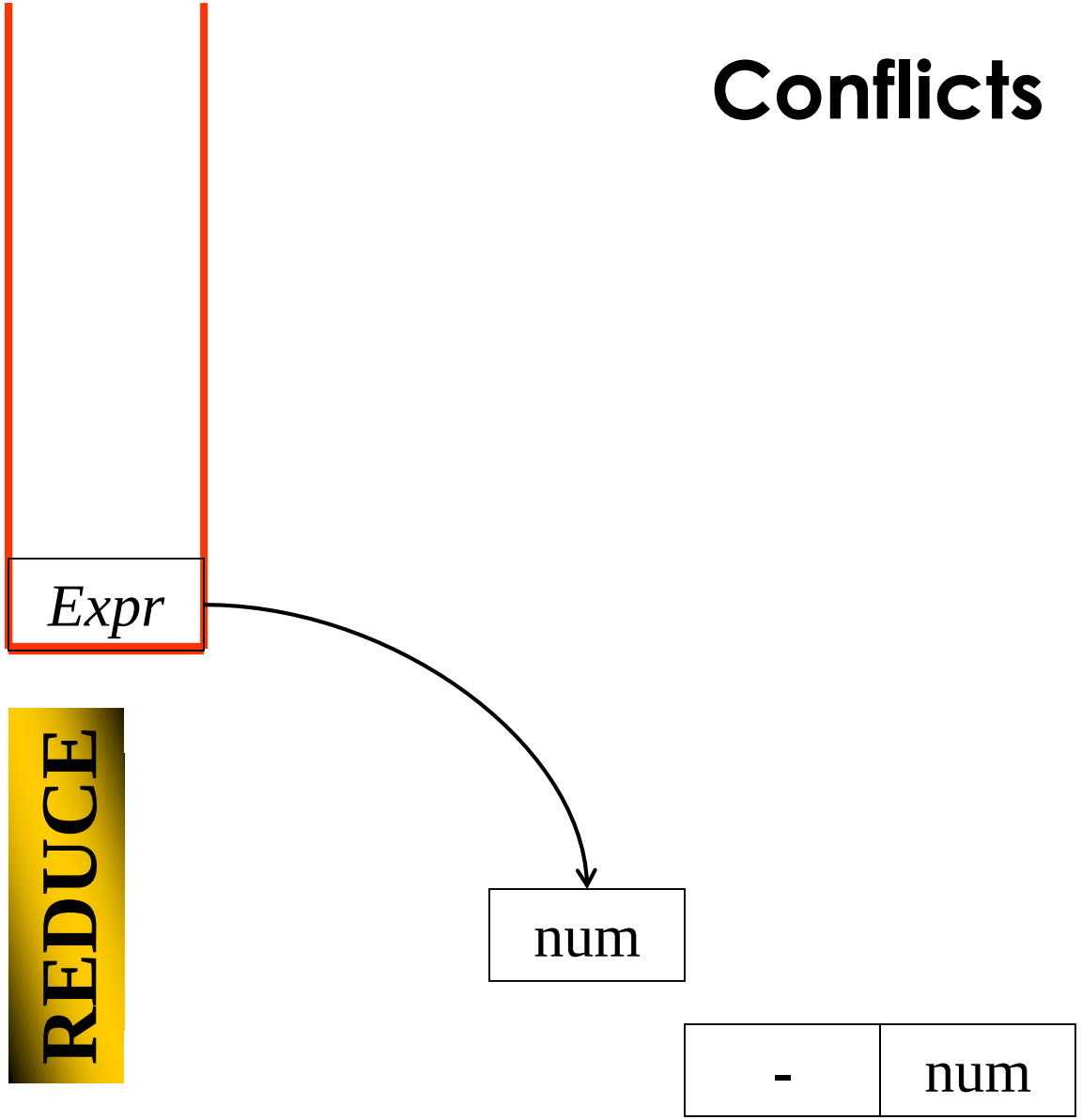
Expr → num

Op → +

Op → -

Op → *

Conflicts



Expr → *Expr Op Expr*

Expr → *Expr* - *Expr*

Expr → (*Expr*)

Expr → *Expr* -

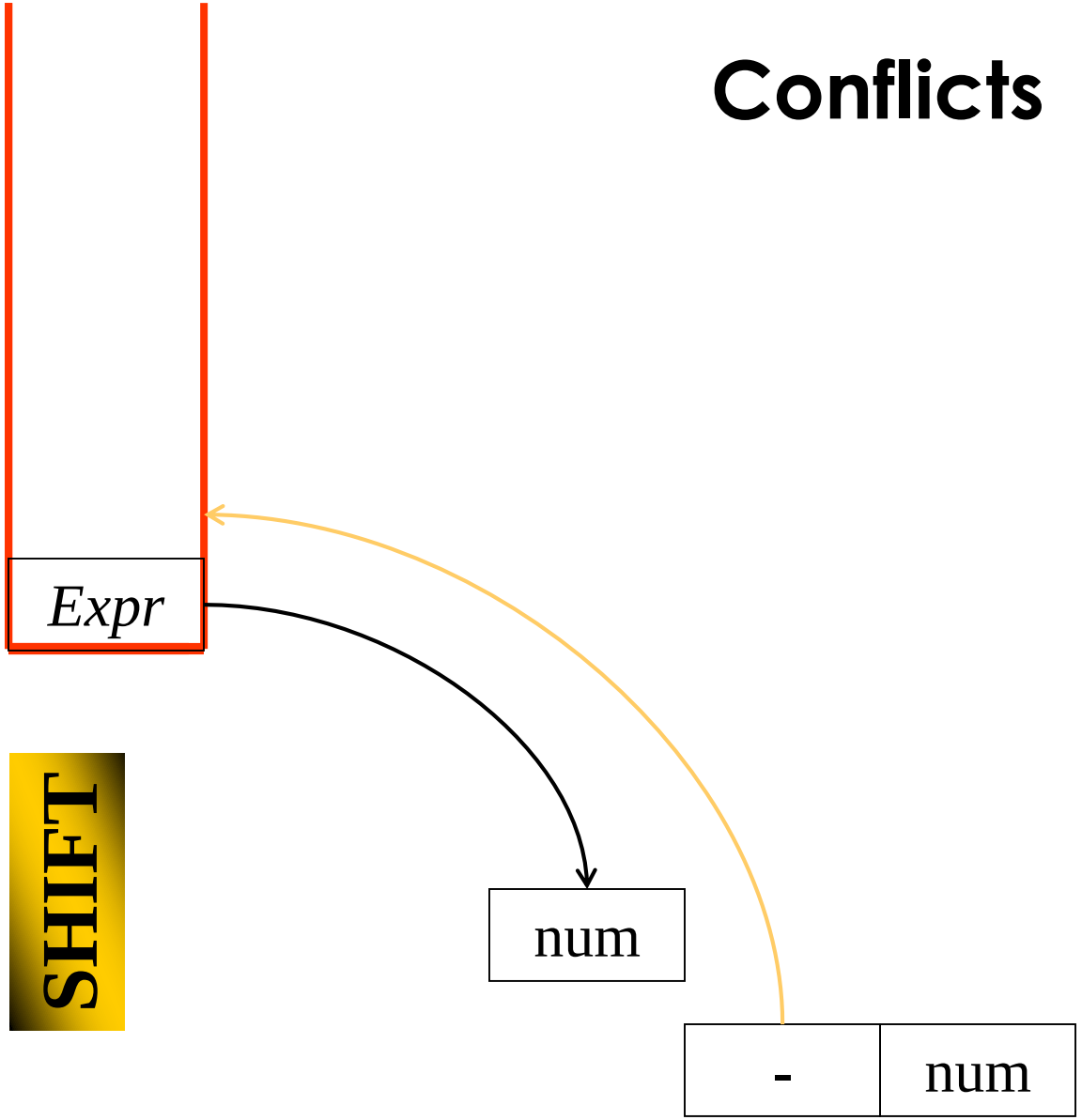
Expr → num

Op → +

Op → -

Op → *

Conflicts



Expr → *Expr Op Expr*

Expr → *Expr* - *Expr*

Expr → (*Expr*)

Expr → *Expr* -

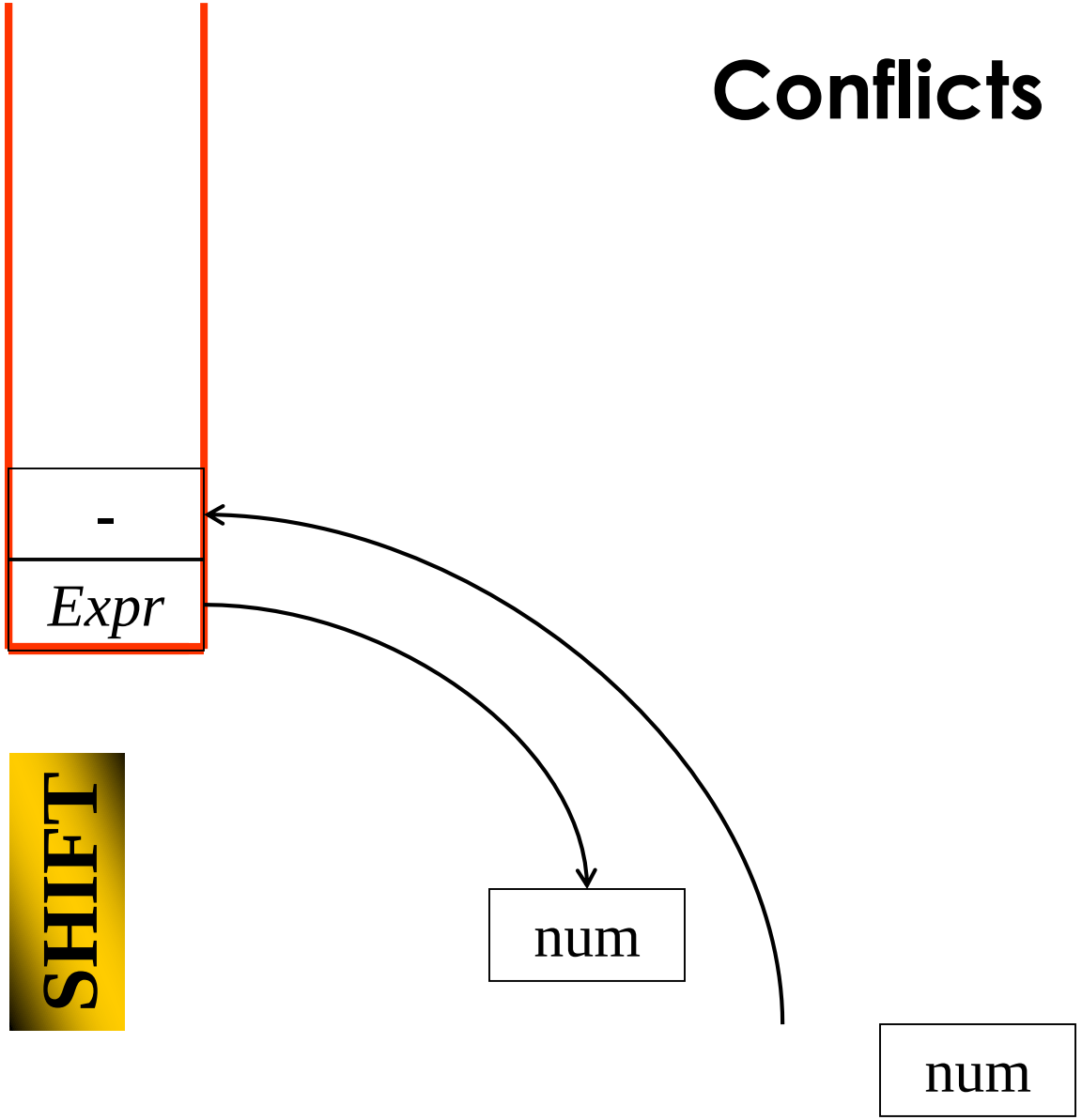
Expr → num

Op → +

Op → -

Op → *

Conflicts



$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow Expr\ -\ \ Expr$

$Expr \rightarrow\ (Expr)$

$Expr \rightarrow\ \ Expr\ -$

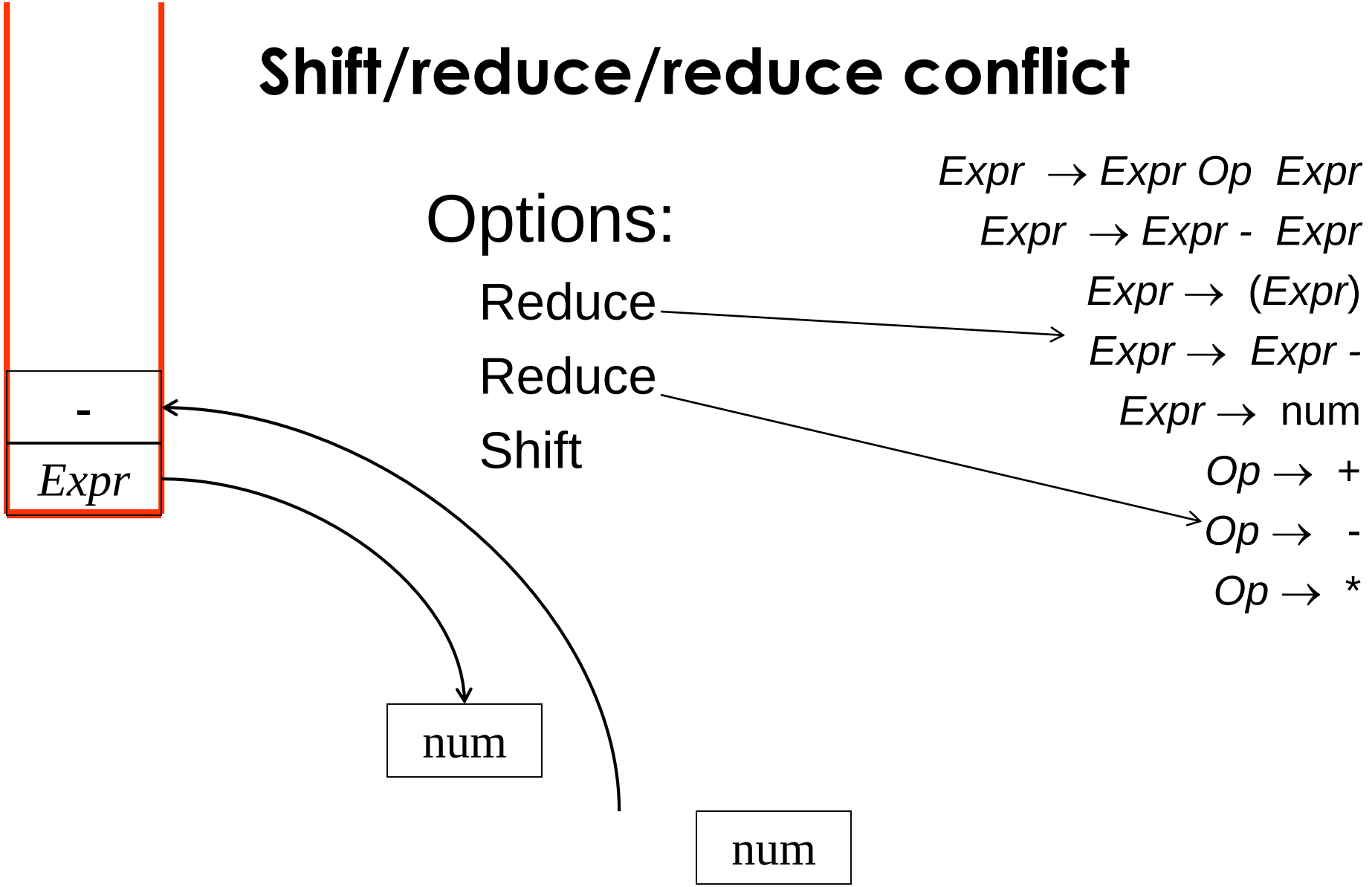
$Expr \rightarrow\ \ \text{num}$

$Op \rightarrow\ \ +$

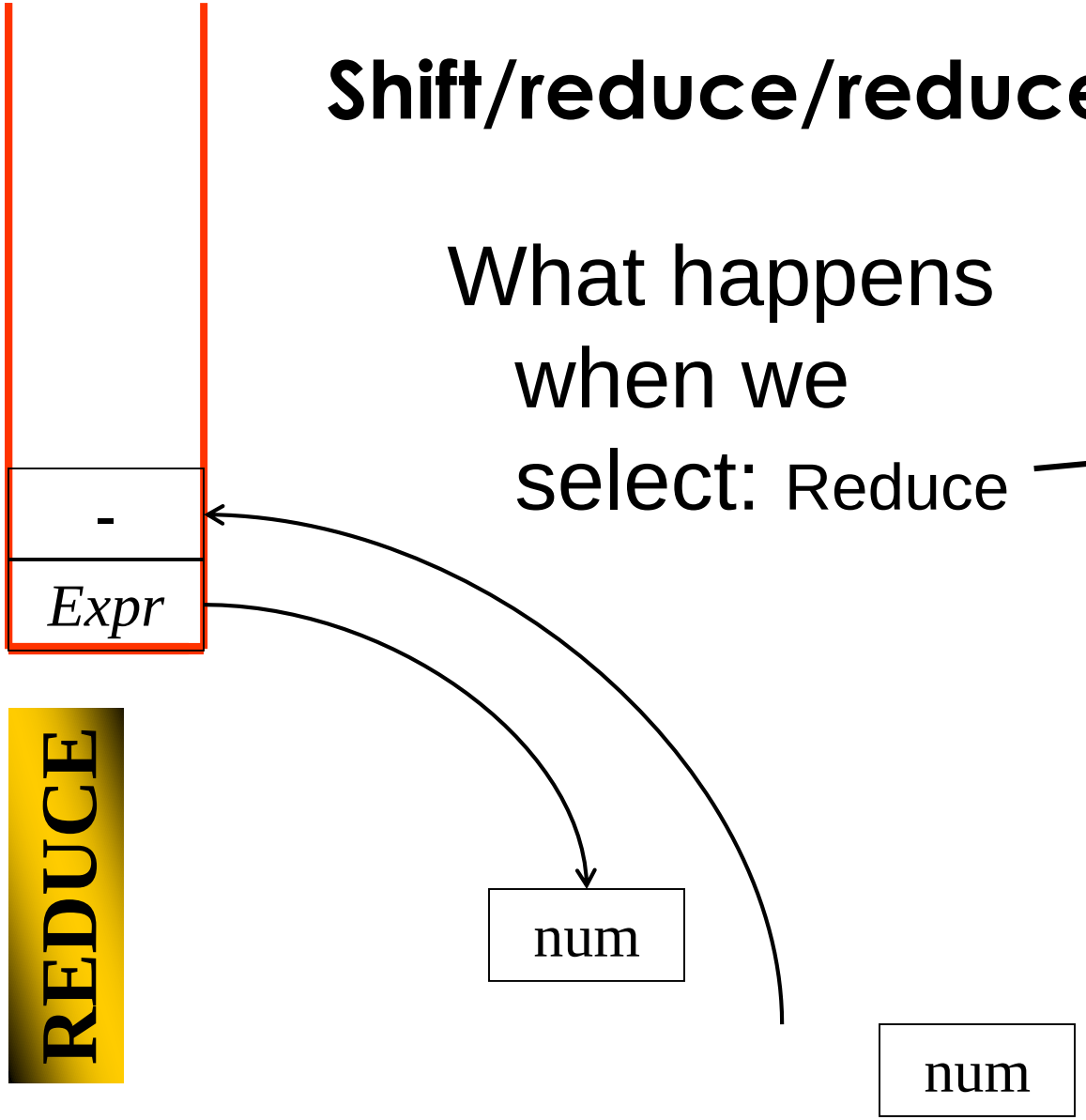
$Op \rightarrow\ \ -$

$Op \rightarrow\ \ *$

Shift/reduce/reduce conflict



Shift/reduce/reduce conflict



What happens when we select: Reduce

Expr → *Expr Op Expr*

Expr → *Expr* - *Expr*

Expr → (*Expr*)

Expr → *Expr* -

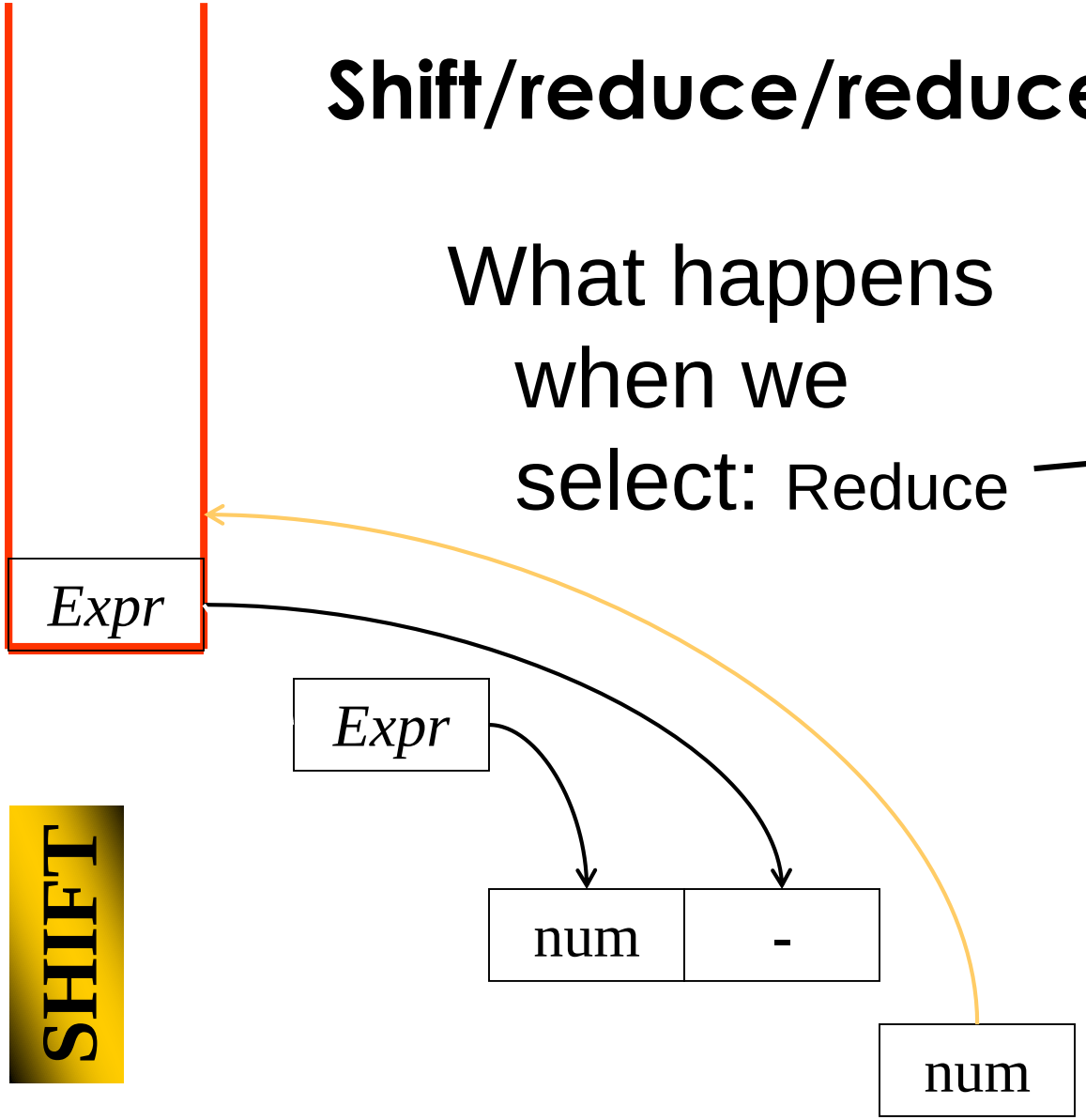
Expr → num

Op → +

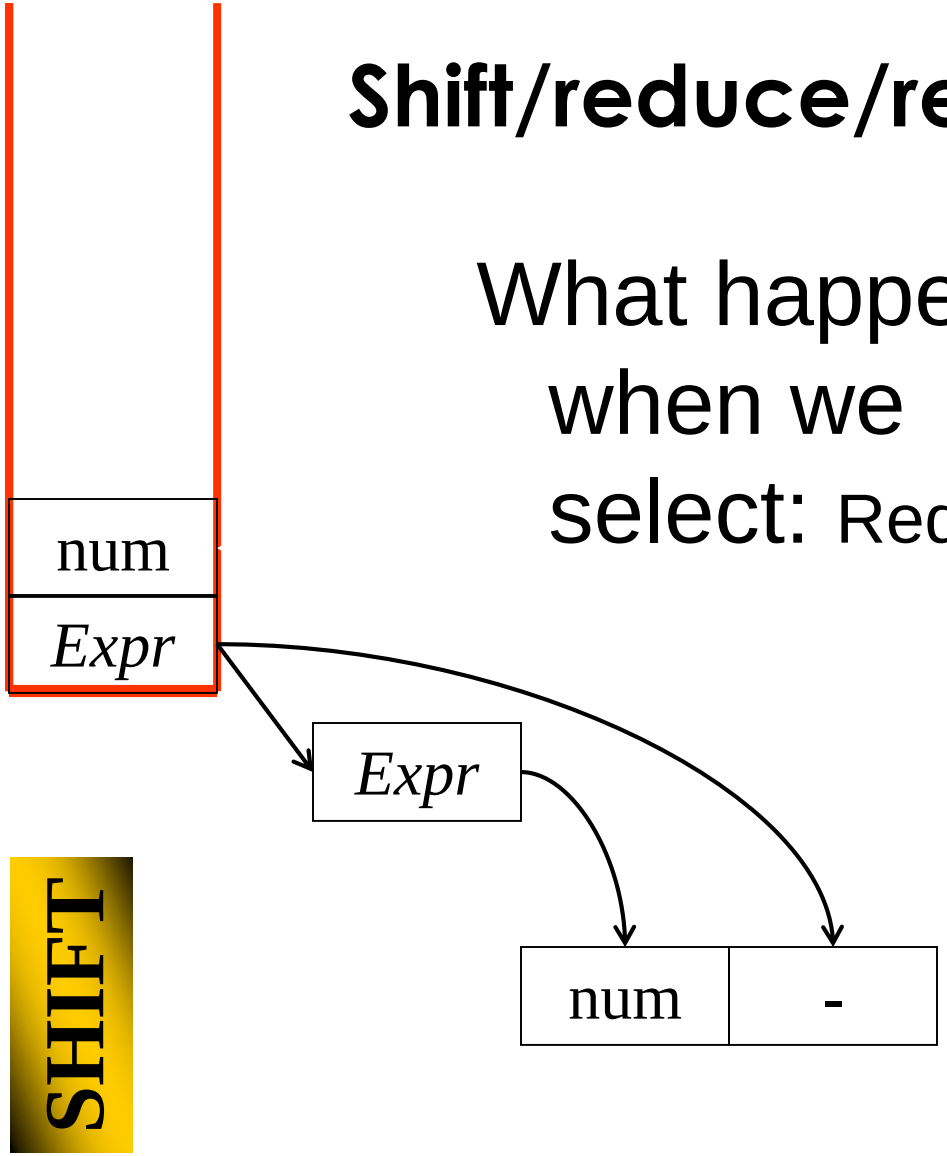
Op → -

Op → *

Shift/reduce/reduce conflict



Shift/reduce/reduce conflict



What happens when we select: Reduce

Expr → *Expr Op Expr*

Expr → *Expr* - *Expr*

Expr → (*Expr*)

Expr → *Expr* -

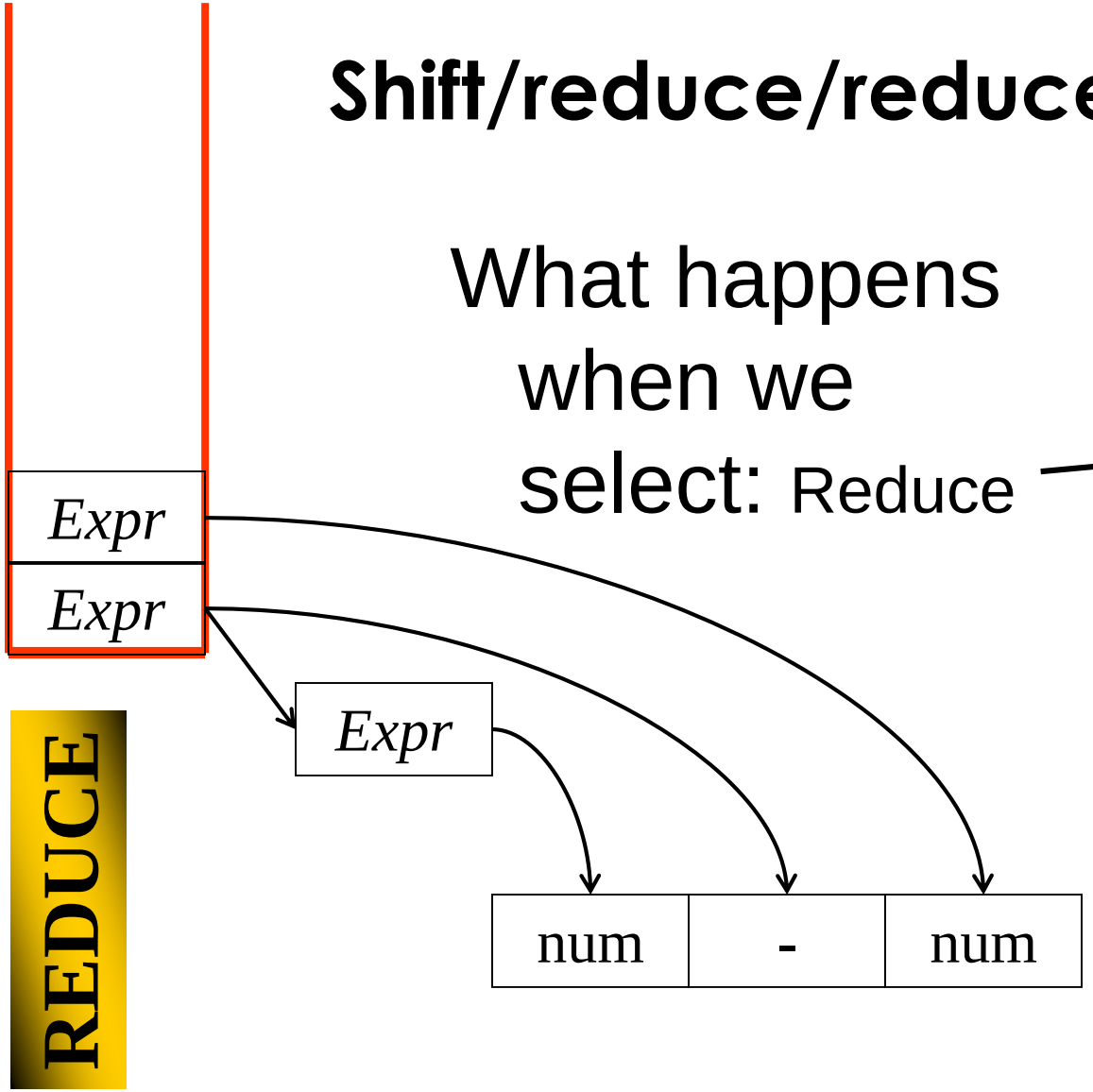
Expr → num

Op → +

Op → -

Op → *

Shift/reduce/reduce conflict



What happens when we select: Reduce

Expr → *Expr Op Expr*

Expr → *Expr* - *Expr*

Expr → (*Expr*)

Expr → *Expr* -

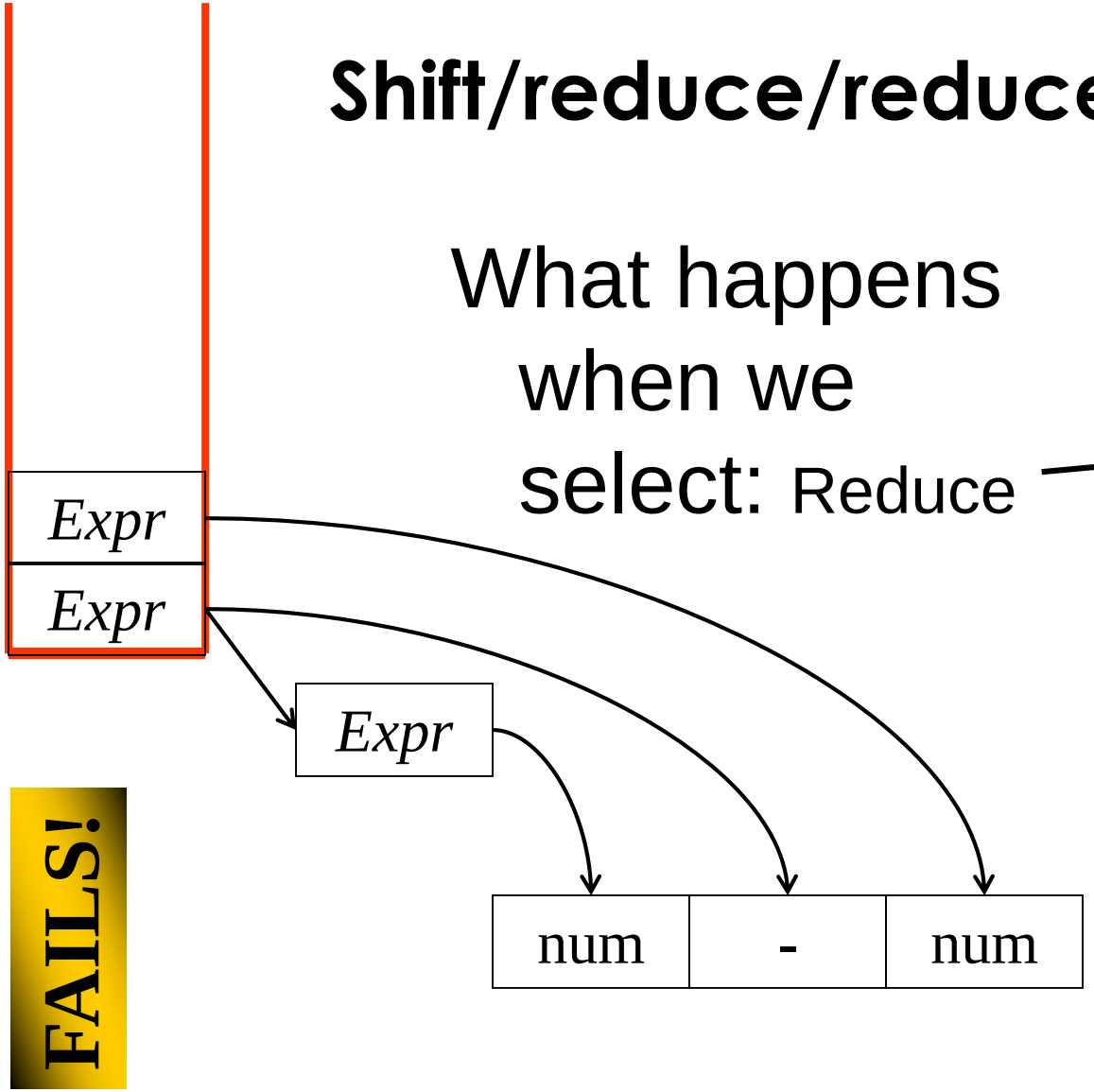
Expr → num

Op → +

Op → -

Op → *

Shift/reduce/reduce conflict



What happens when we select: Reduce

Expr → *Expr Op Expr*

Expr → *Expr* - *Expr*

Expr → (*Expr*)

Expr → *Expr* -

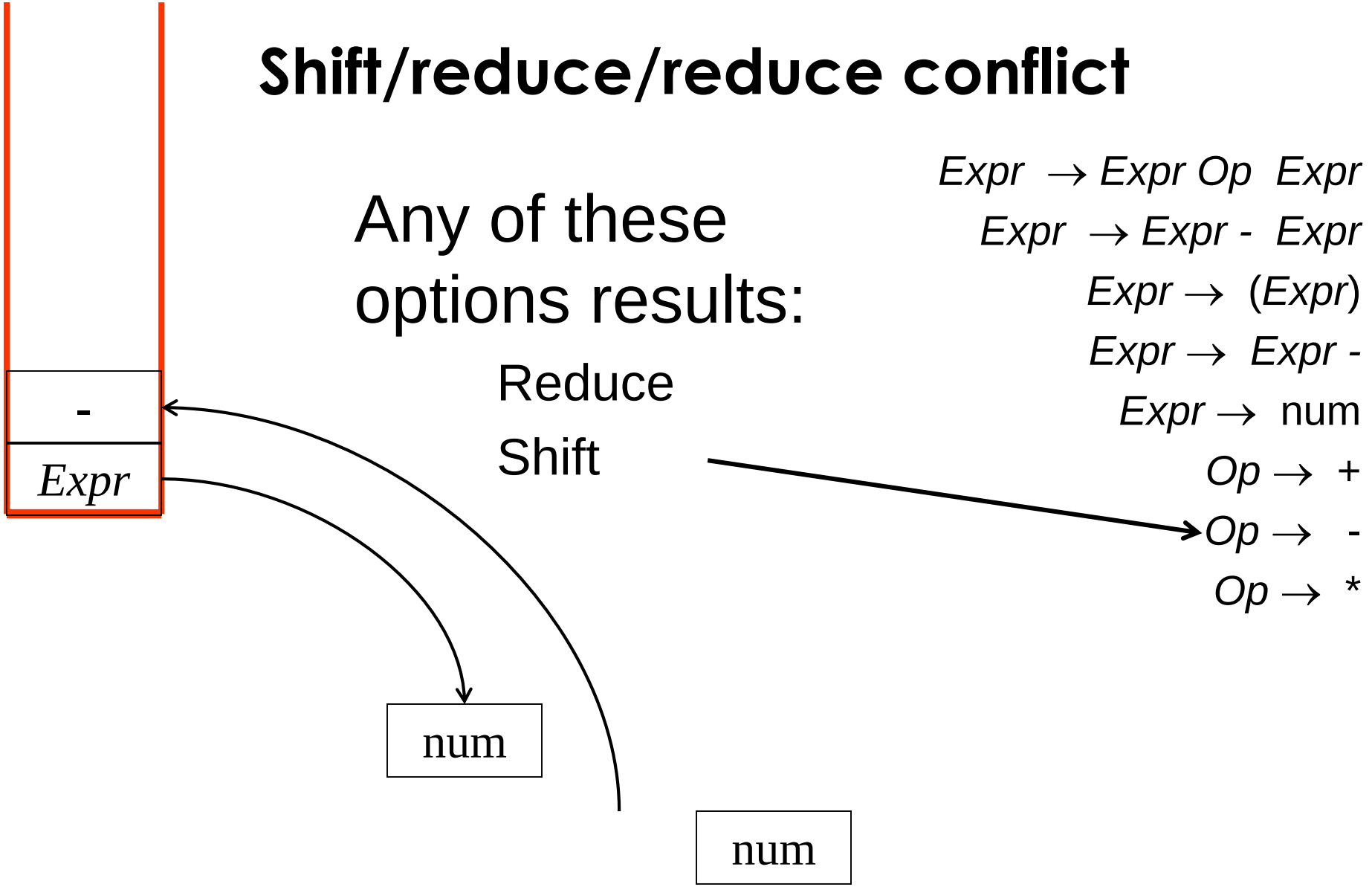
Expr → num

Op → +

Op → -

Op → *

Shift/reduce/reduce conflict



Any of these options results:

Reduce

Shift

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr\ -\ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow Expr\ -$

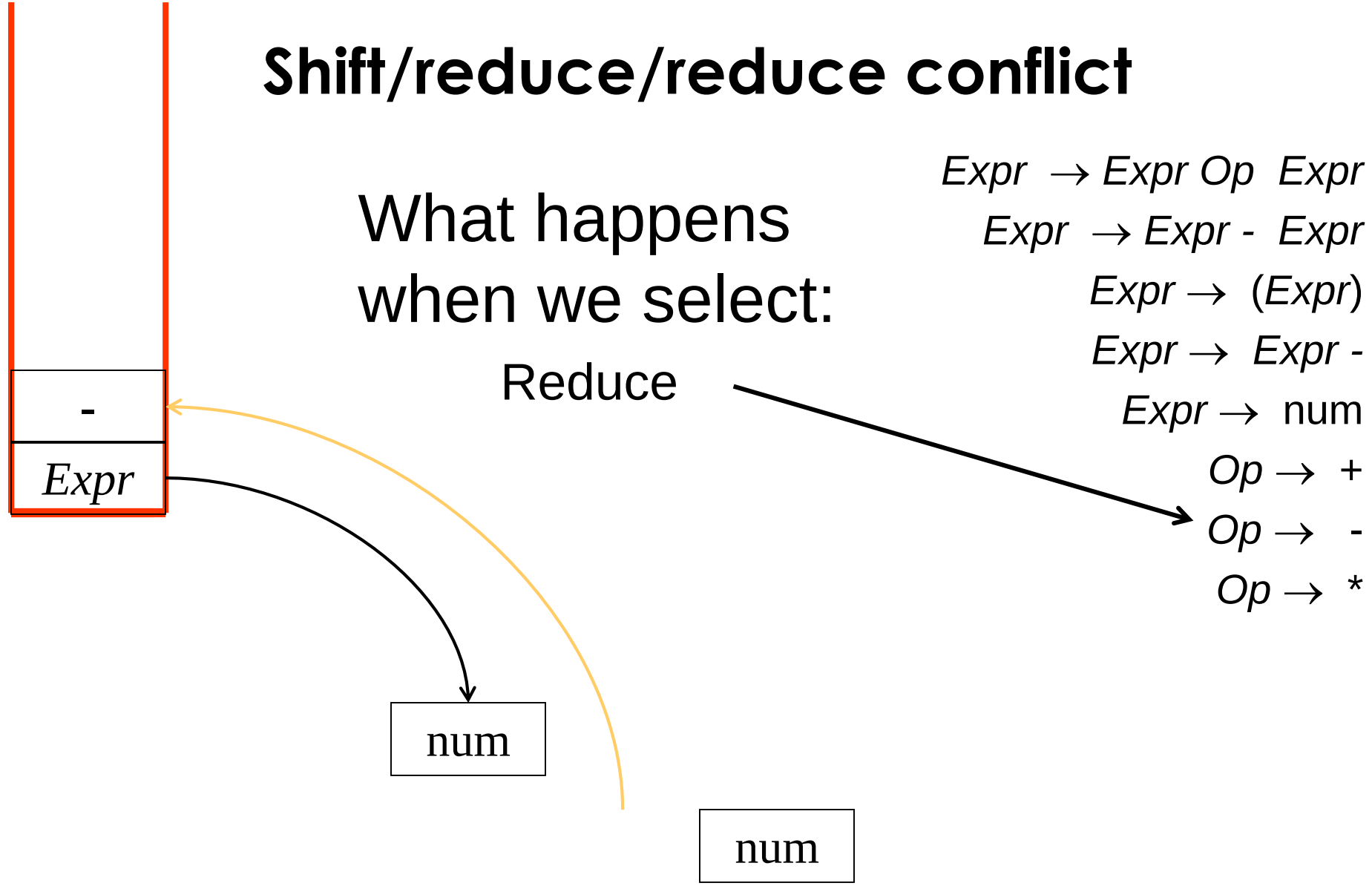
$Expr \rightarrow$ num

$Op \rightarrow +$

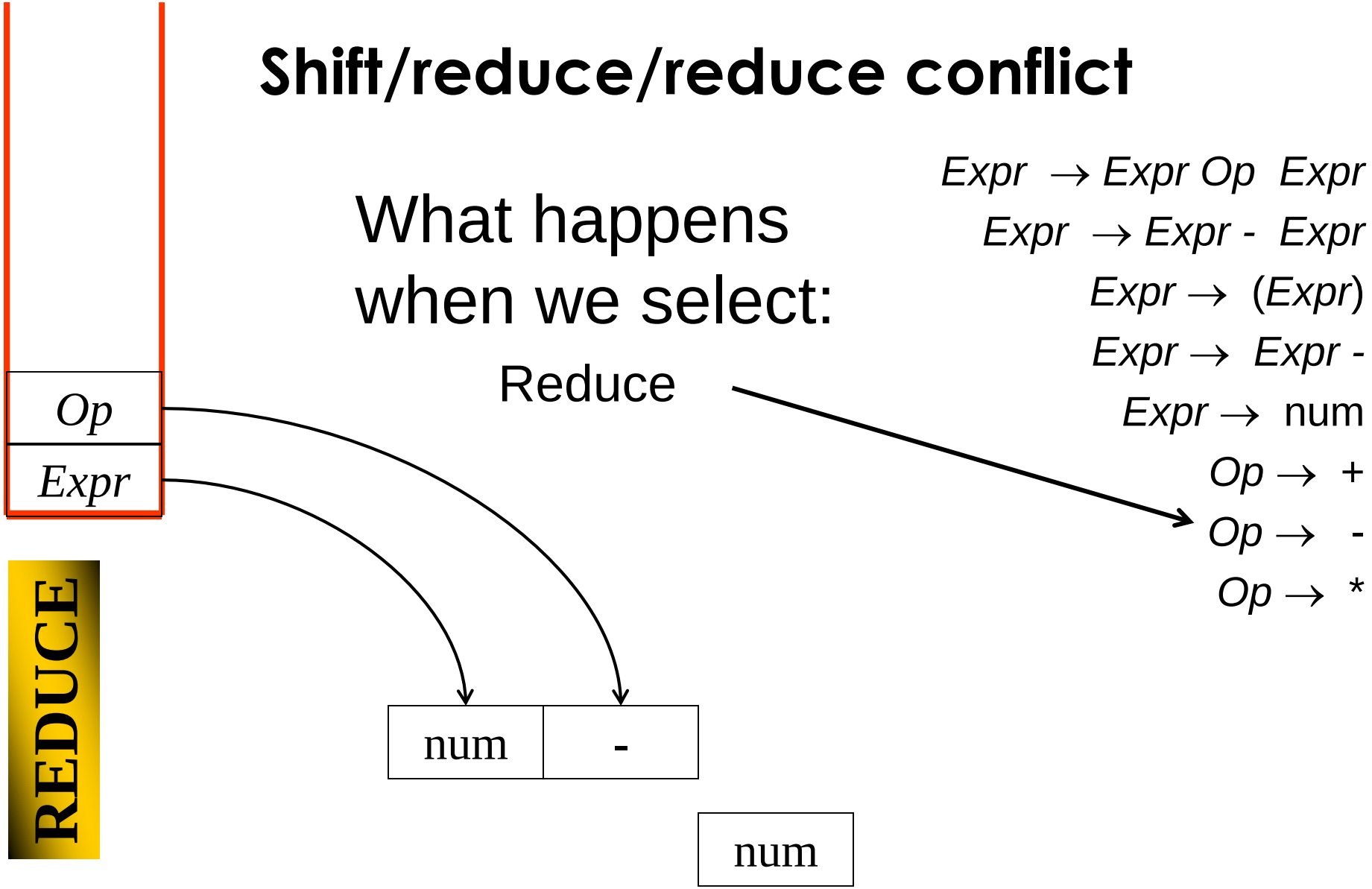
$Op \rightarrow -$

$Op \rightarrow *$

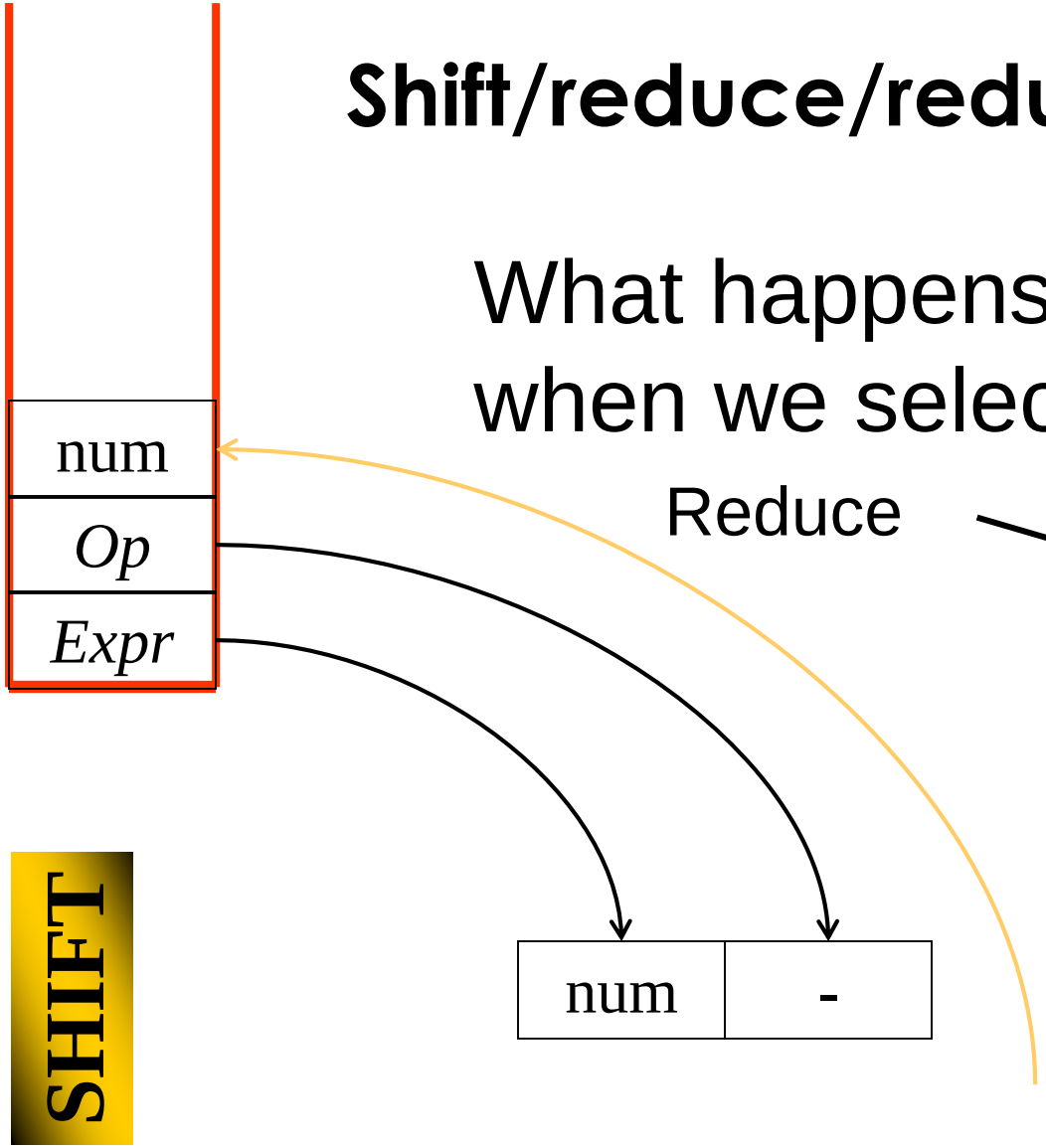
Shift/reduce/reduce conflict



Shift/reduce/reduce conflict



Shift/reduce/reduce conflict



What happens when we select:

Reduce

Expr → *Expr Op Expr*

Expr → *Expr* - *Expr*

Expr → (*Expr*)

Expr → *Expr* -

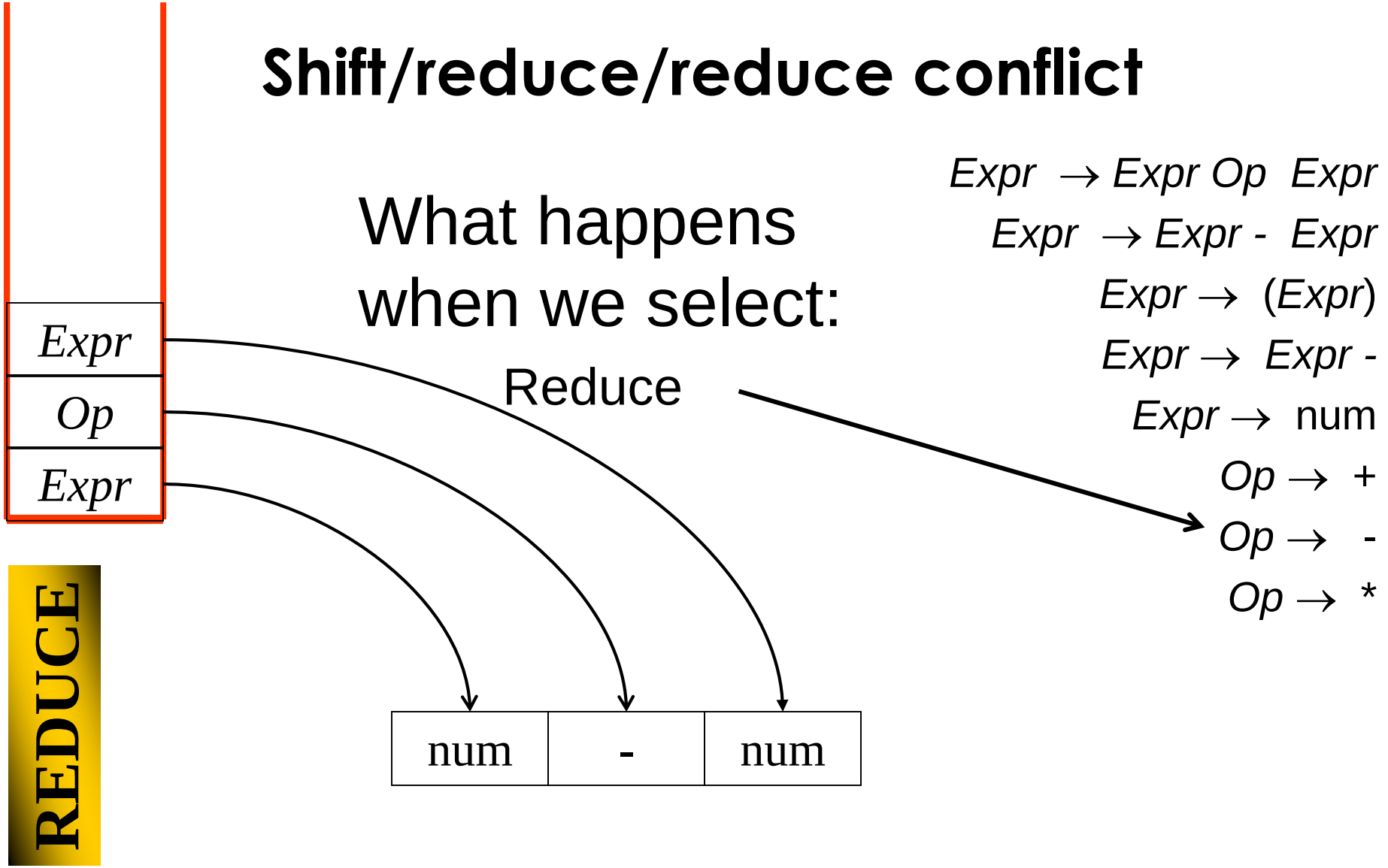
Expr → num

Op → +

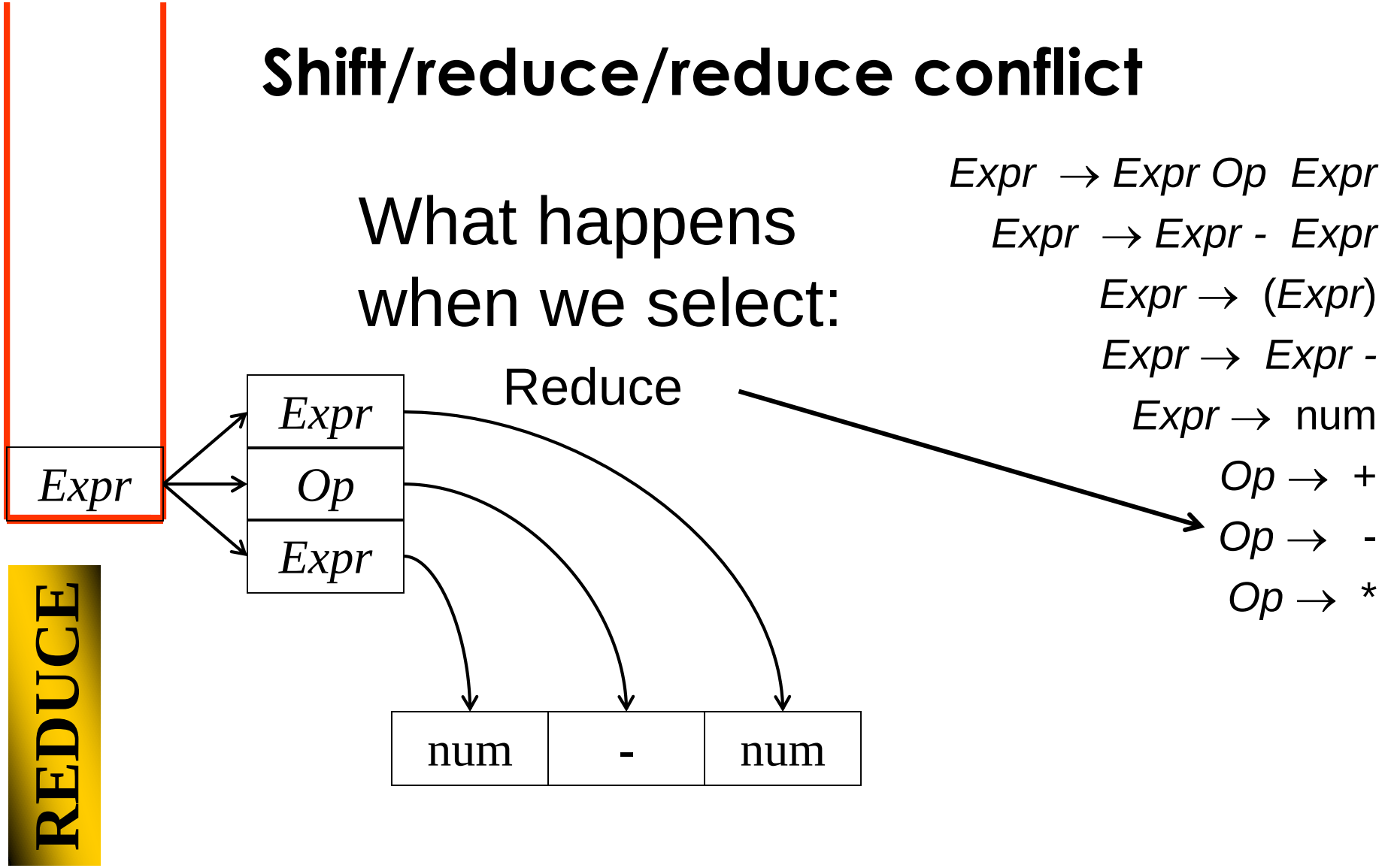
Op → -

Op → *

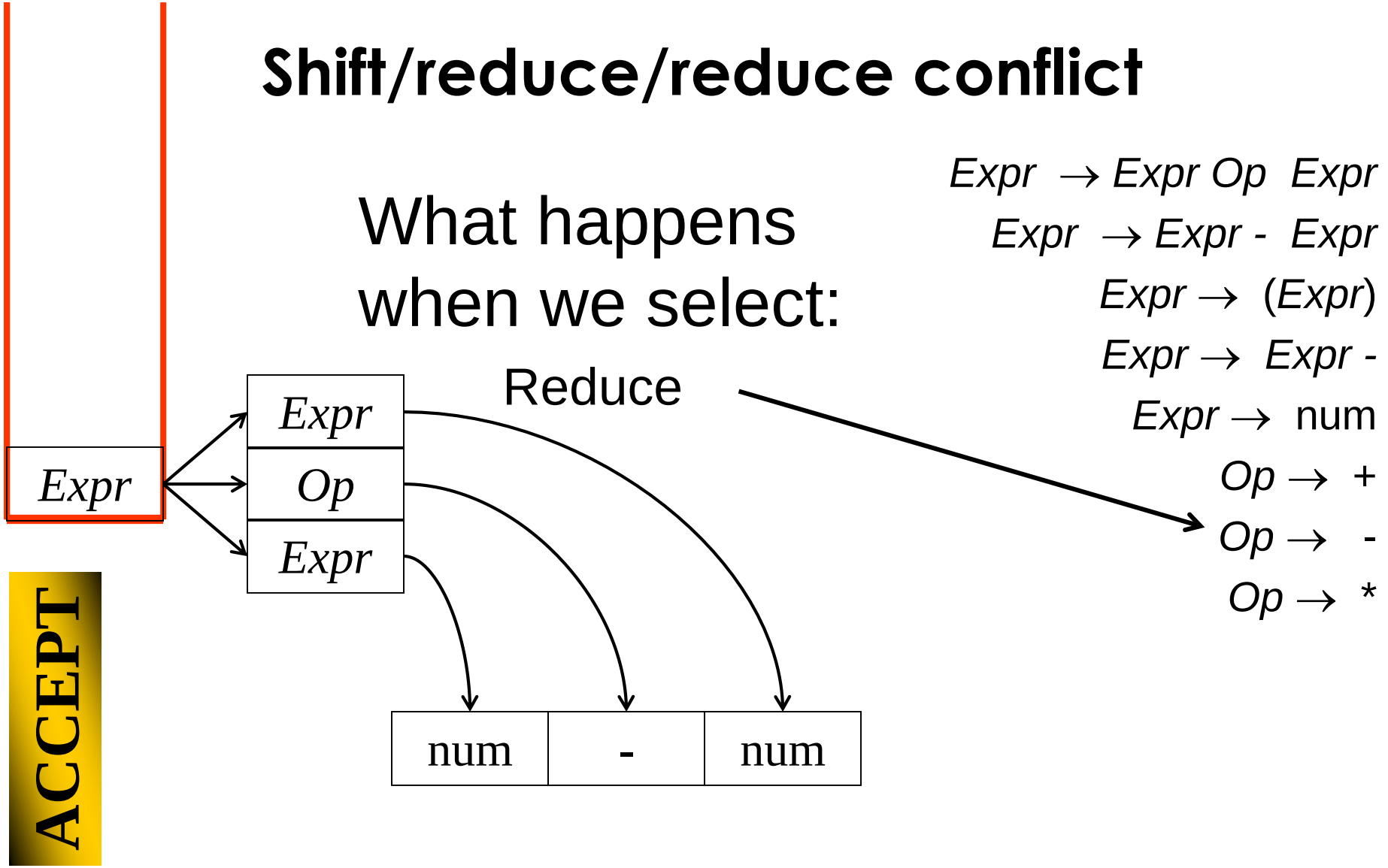
Shift/reduce/reduce conflict



Shift/reduce/reduce conflict



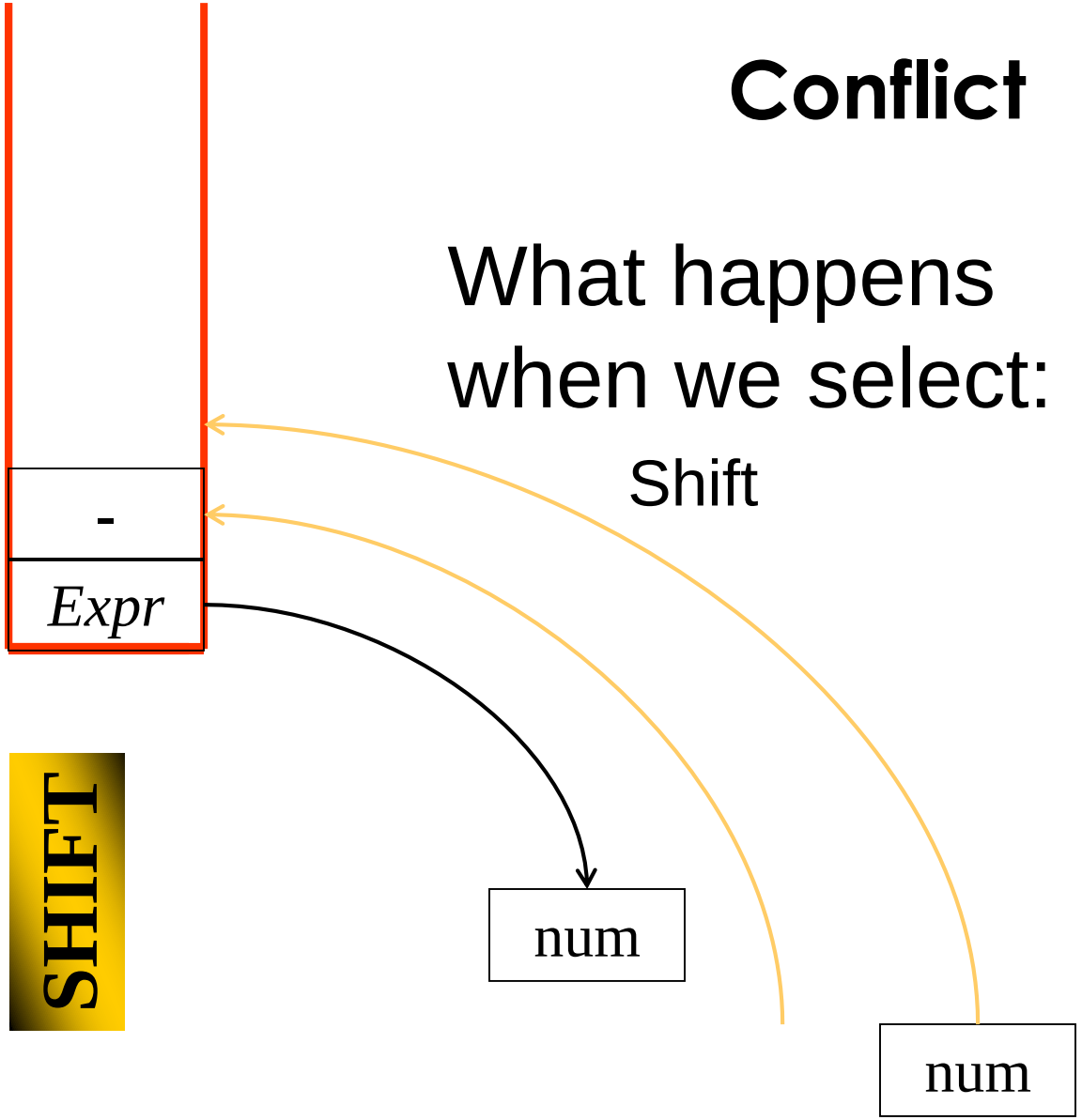
Shift/reduce/reduce conflict



Conflict

What happens when we select:

Shift



$Expr \rightarrow Expr\ Op\ \ Expr$

$Expr \rightarrow Expr\ -\ \ Expr$

$Expr \rightarrow (Expr)$

$Expr \rightarrow Expr\ -$

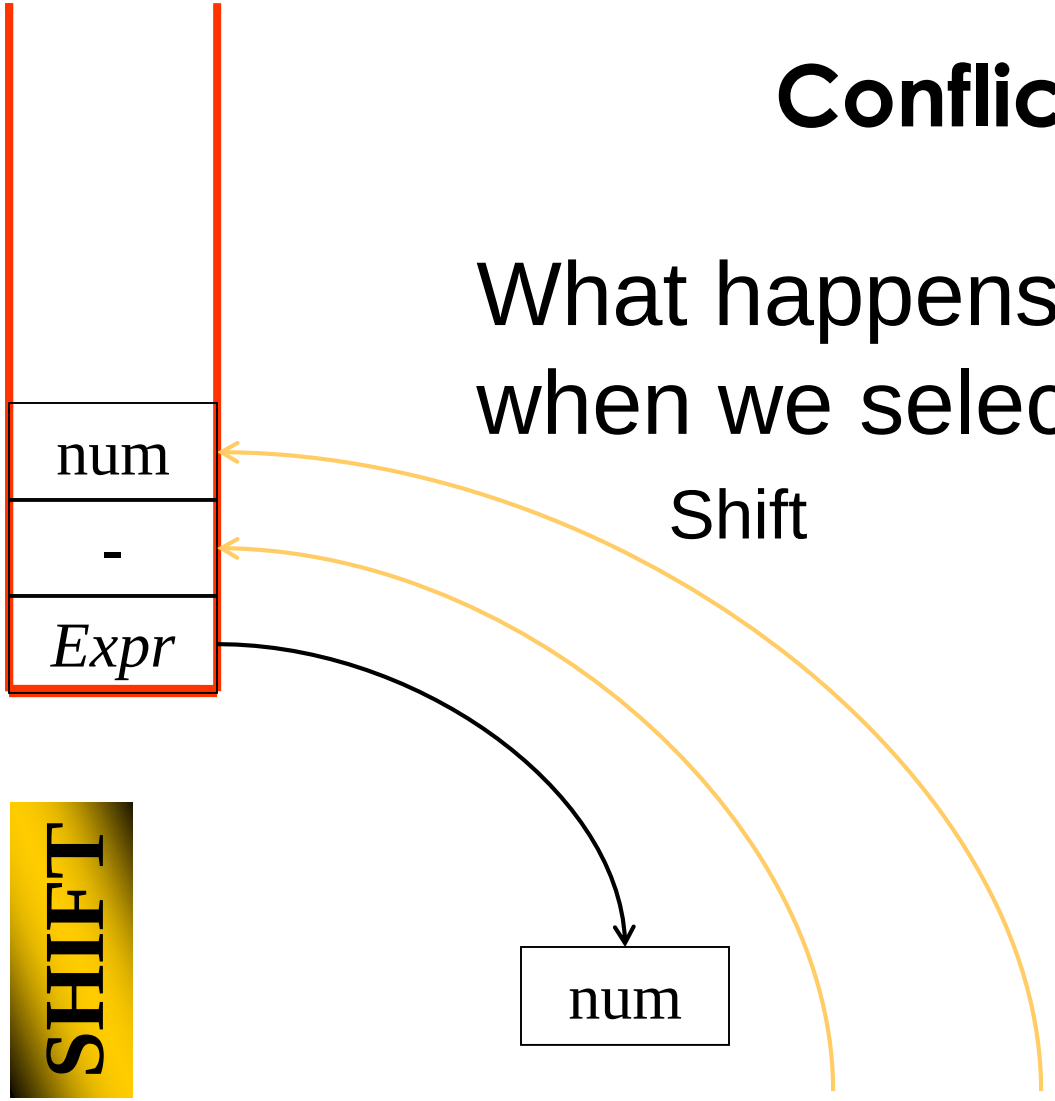
$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

Conflict



What happens when we select:

Shift

Expr → *Expr Op Expr*

Expr → *Expr* - *Expr*

Expr → (*Expr*)

Expr → *Expr* -

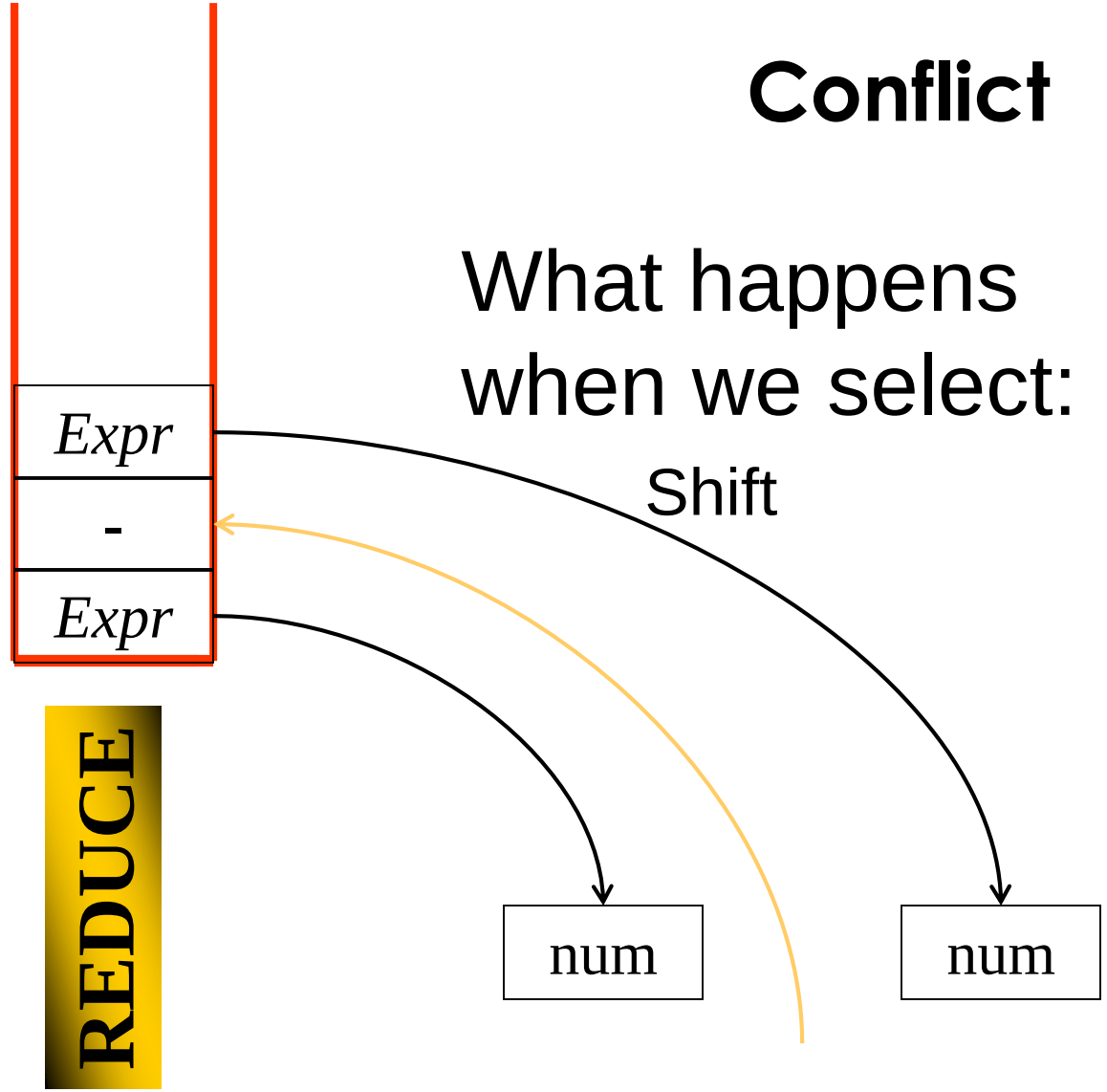
Expr → num

Op → +

Op → -

Op → *

Conflict



What happens when we select:

Shift

$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow$ ($Expr$)

$Expr \rightarrow Expr$ -

$Expr \rightarrow$ num

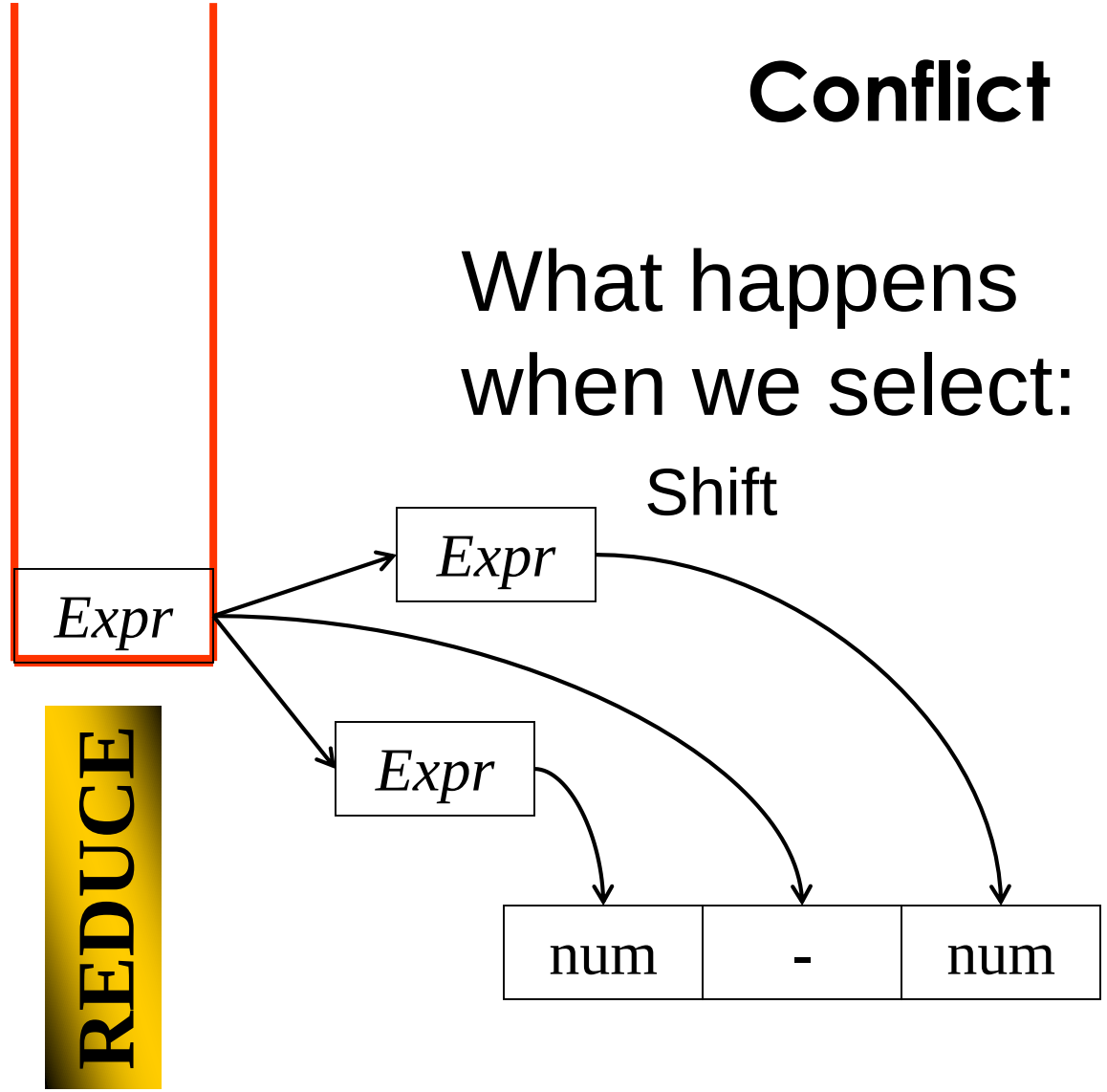
$Op \rightarrow$ +

$Op \rightarrow$ -

$Op \rightarrow$ *

Conflict

What happens when we select:



$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow$ ($Expr$)

$Expr \rightarrow Expr$ -

$Expr \rightarrow$ num

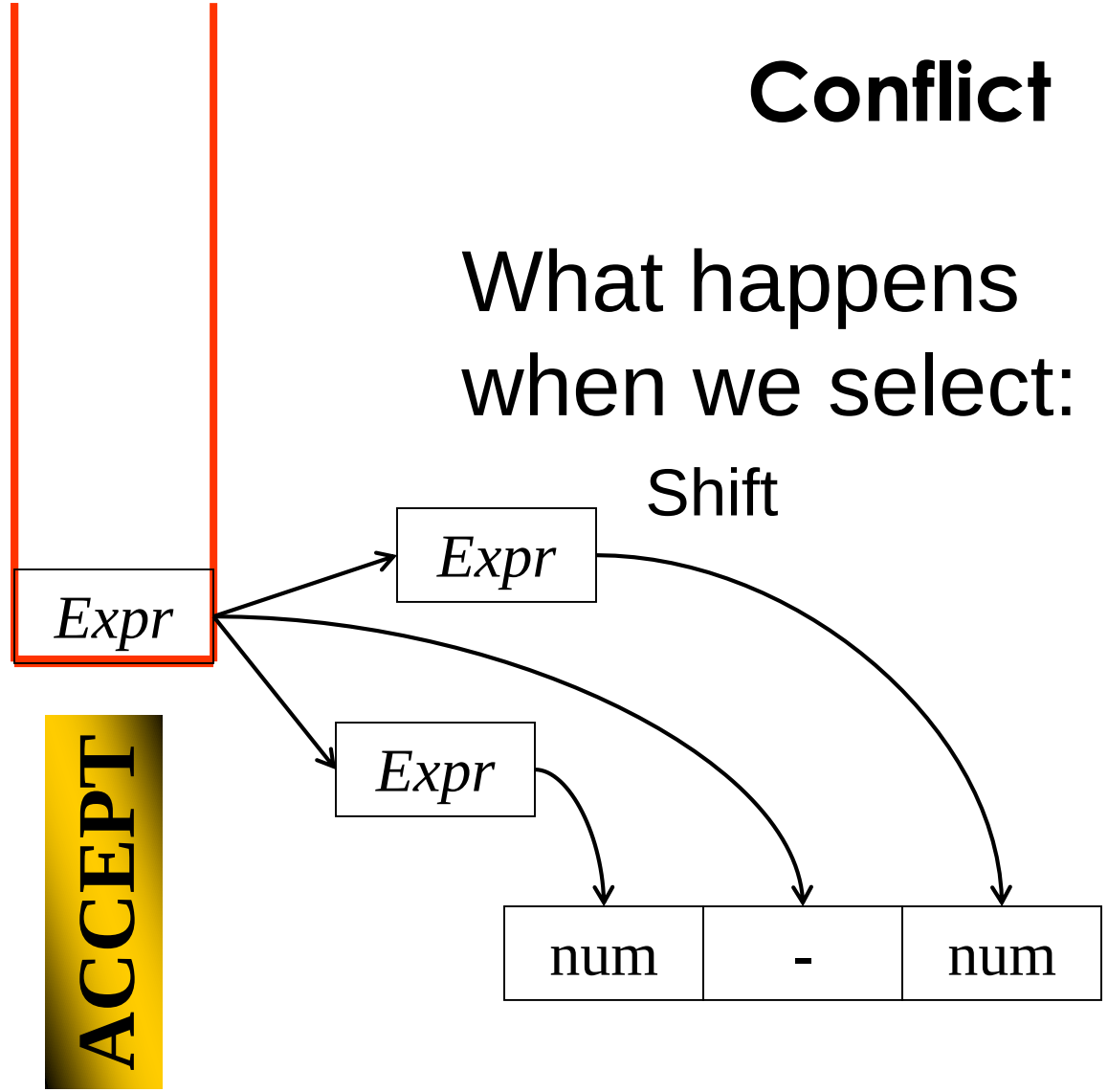
$Op \rightarrow$ +

$Op \rightarrow$ -

$Op \rightarrow$ *

Conflict

What happens when we select:



$Expr \rightarrow Expr\ Op\ Expr$

$Expr \rightarrow Expr$ - $Expr$

$Expr \rightarrow$ (*Expr*)

$Expr \rightarrow Expr$ -

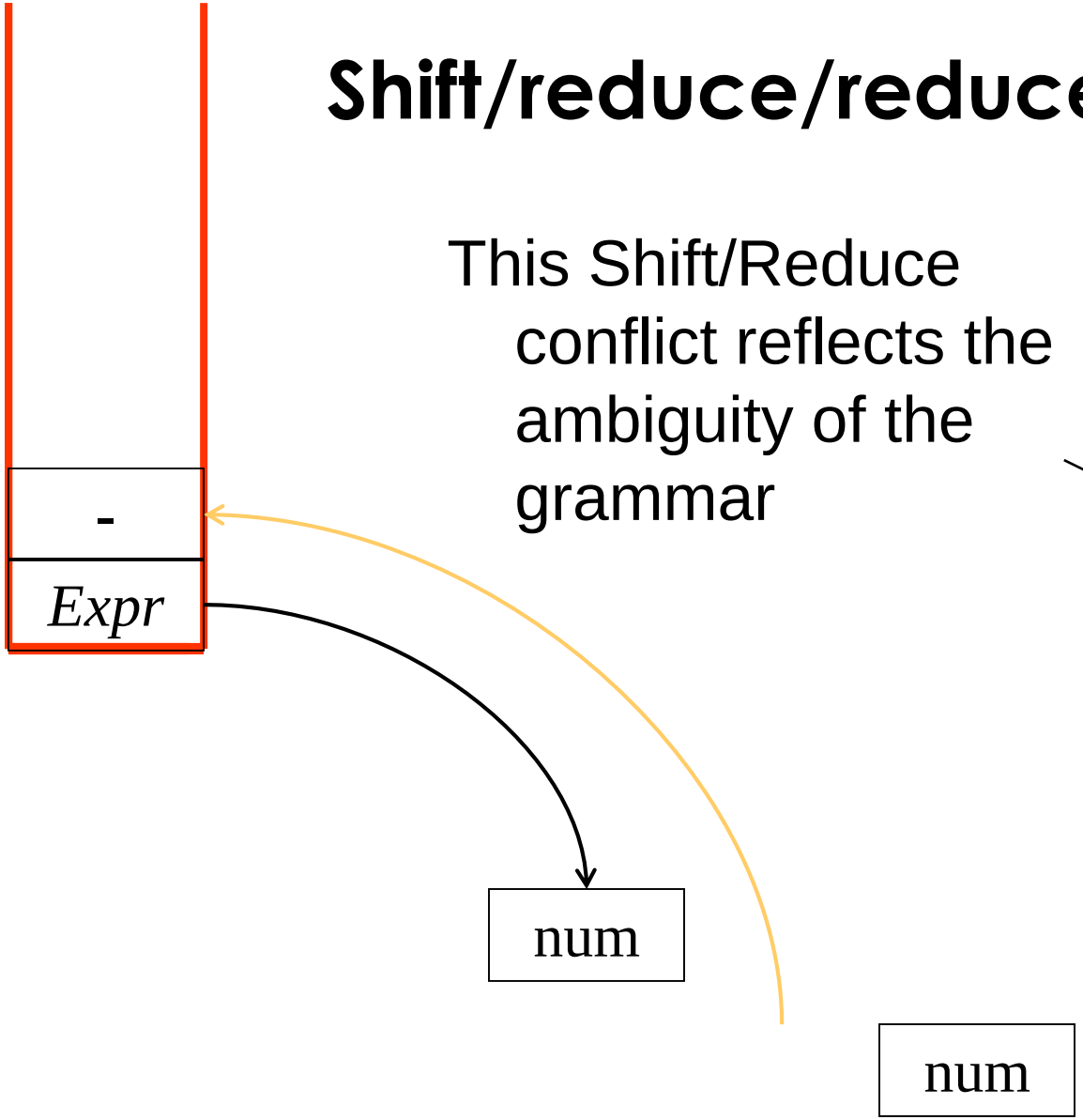
$Expr \rightarrow$ num

$Op \rightarrow$ +

$Op \rightarrow$ -

$Op \rightarrow$ *

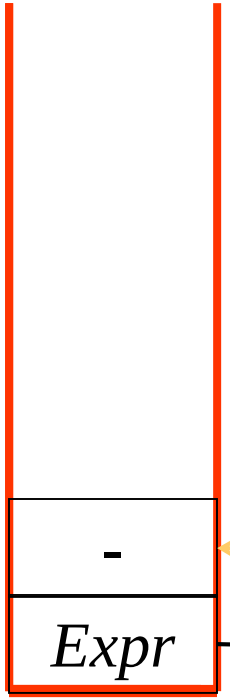
Shift/reduce/reduce conflict



This Shift/Reduce conflict reflects the ambiguity of the grammar

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

Shift/reduce/reduce conflict



This Shift/Reduce conflict reflects the ambiguity of the grammar

$Expr \rightarrow Expr\ Op\ Expr$

~~$Expr \rightarrow Expr - Expr$~~

$Expr \rightarrow (Expr)$

$Expr \rightarrow Expr\ -$

$Expr \rightarrow$ num

$Op \rightarrow +$

$Op \rightarrow -$

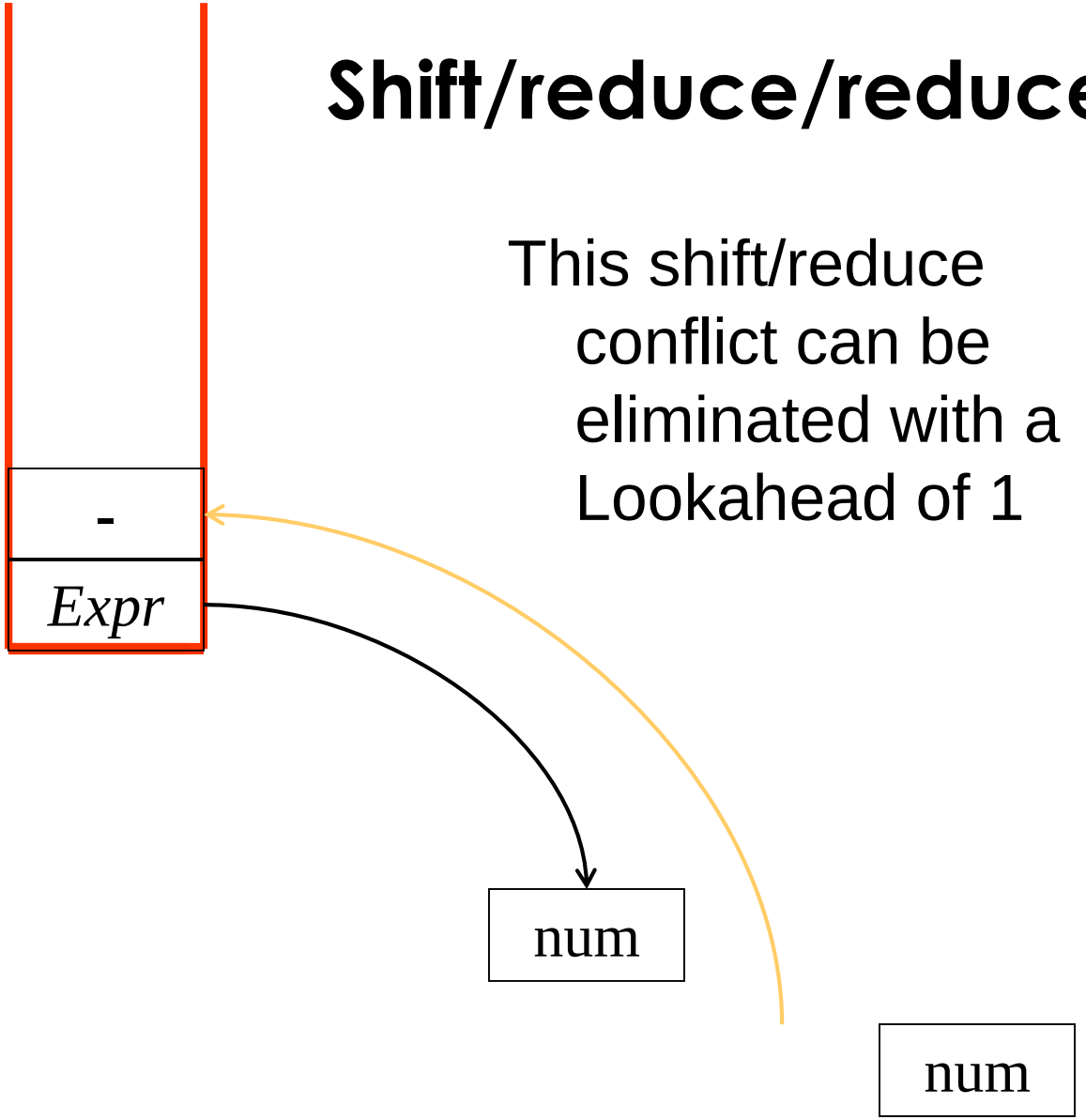
$Op \rightarrow *$

Eliminate changing the grammar

num

num

Shift/reduce/reduce conflict



This shift/reduce conflict can be eliminated with a Lookahead of 1

Expr → *Expr Op Expr*

Expr → *Expr* - *Expr*

Expr → (*Expr*)

→ *Expr* → *Expr* -

Expr → num

Op → +

Op → -

Op → *

Building an LR Parser

- Let's build it without *lookahead, i.e., an LR(0)*
- Key decisions
  - *Shift* or *Reduce*
  - What is the production to use?
- Basic idea
  - Build a DFA to control *shift* and *reduce* actions
  - The same as to convert the grammar to a *pushdown automaton*
  - Codify finite state control in a parse table

States of the LR Parser

- Sequence of tokens in the input ($ as the endmark to signal the end of the input)
- Current state of the finite automaton
- Two stacks
  - Stack of states (implements finite automaton)
  - Stack of symbols (input terminals and non-terminals of the reductions)

Controlling the states

- Actions
  - Put symbols and states in stack
  - Reduce according to a given production
  - Accept
  - Error
- Selected function is a function of
  - Current input symbol
  - Current state
- Each action specifies the next state
- Implement the control using the parser table

Parser table

		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)	

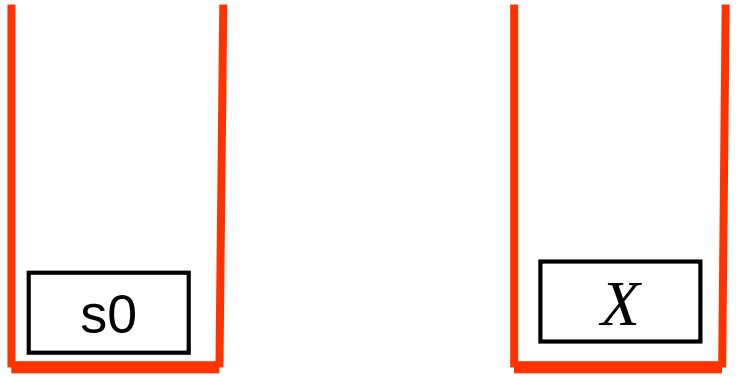
- Implements control of the finite states
- In each state see:
  - Table[top of the state stack][input symbol]
- Next, perform action

Example of parser table

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

	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



(())

Parser table

	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 *sn*
  - Put input token in the stack of the symbols
  - Put *sn* in the stack of states
  - Move to the next input symbol

Parser table

	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*)
  - Remove items of the two stacks as many times as the number of symbols in the RHS of production n
  - Put LHS of production n in the stack of symbols
  - See
    - Table[top of stack of states][top of stack of symbols]
  - Put that state (in the goto part of the table) in the stack os states

Parser table

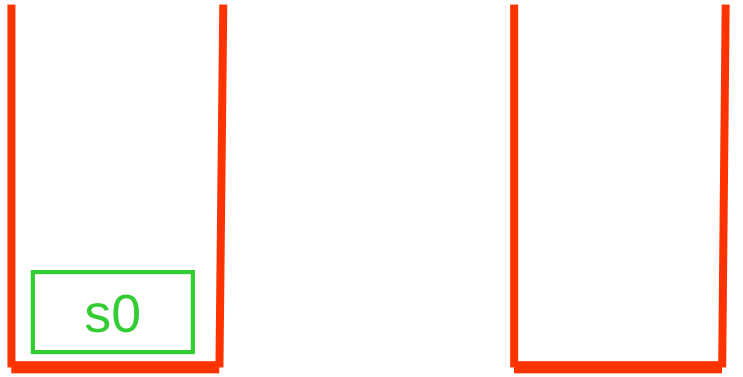
		ACTION		Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

- Accept
  - Stop analysis and report success!
- Error
  - Stop analysis and report error

Parser 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



(())$

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

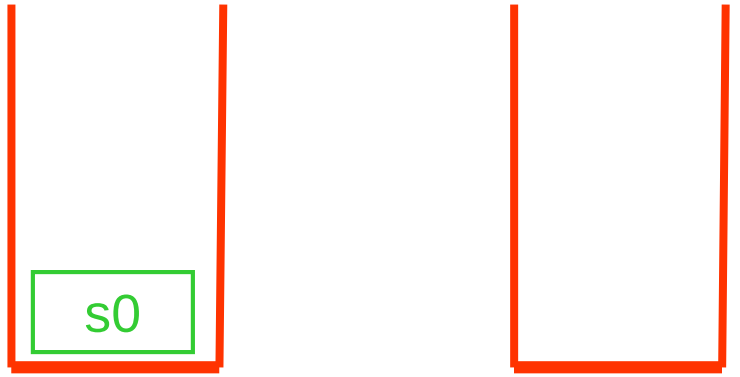
$X \rightarrow (X)$ (2)

$X \rightarrow (\ )$ (3)

Parser 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



(())$

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

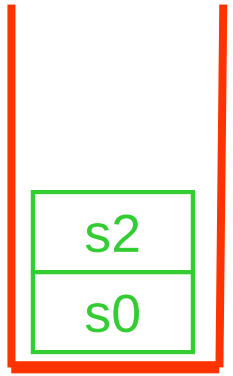
$X \rightarrow (X)\ \ (2)$

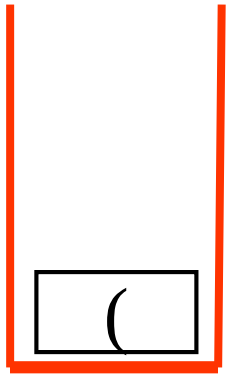
$X \rightarrow (\ )\ \ (3)$

Parser 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





())$

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

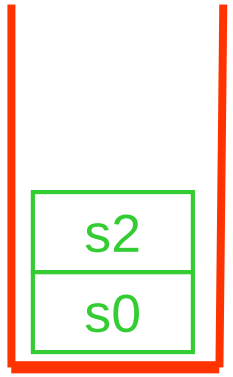
$X \rightarrow (X)$ (2)

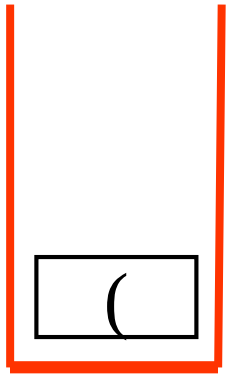
$X \rightarrow (\ )$ (3)

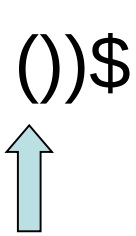
Parser table in action

		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







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

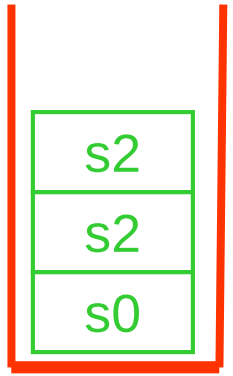
$X \rightarrow (X)$ (2)

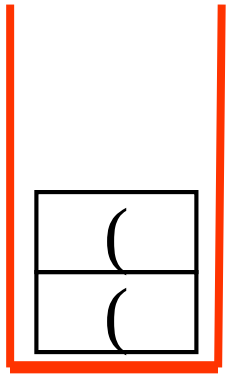
$X \rightarrow (\ )$ (3)

Parser 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





))$

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

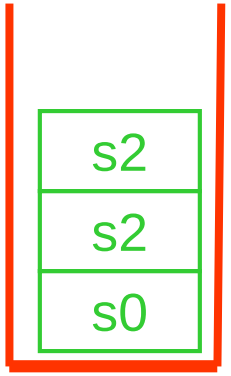
$X \rightarrow (X)$ (2)

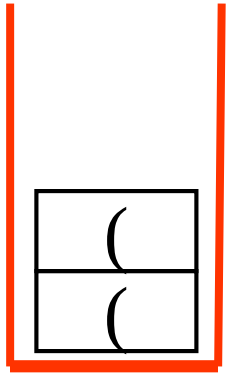
$X \rightarrow (\ )$ (3)

Parser 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





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

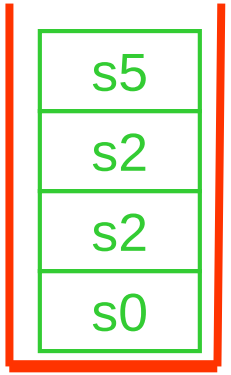
$X \rightarrow (X)$ (2)

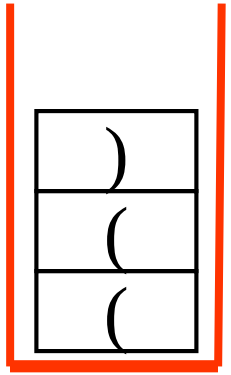
$X \rightarrow (\ )$ (3)

Parser 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





)$

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

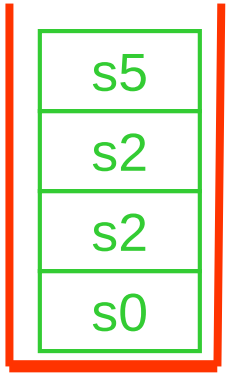
$X \rightarrow (X)$ (2)

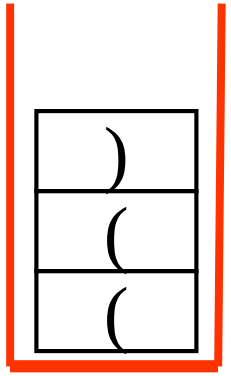
$X \rightarrow (\ )$ (3)

Parser table in action

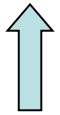
		ACTION		Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State stack Symbol stack Input Grammar





)$



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

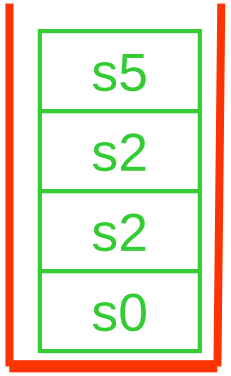
$X \rightarrow (X)\ \ (2)$

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

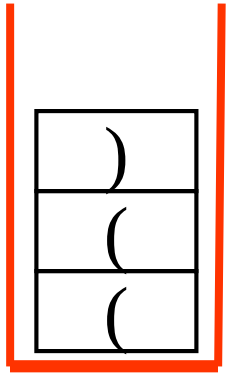
Step 1: pop stacks

		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

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

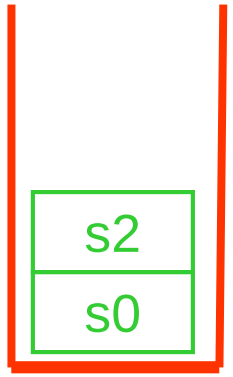
$X \rightarrow (X)\ \ (2)$

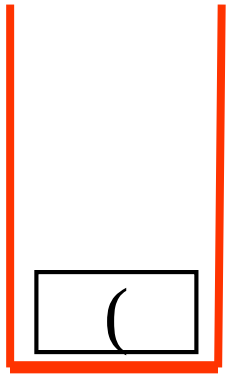
$X \rightarrow (\ )\ (3)$

Step 1: 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





)$

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

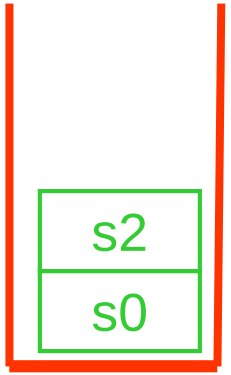
$X \rightarrow (X)$ (2)

$X \rightarrow (\ )$ (3)

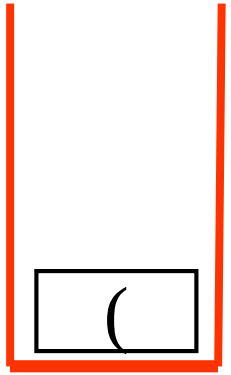
Step 2: push non-terminal

		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

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

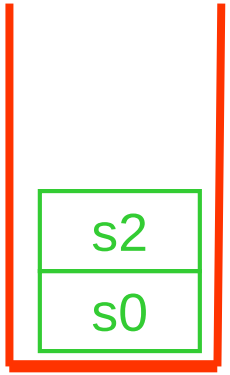
$X \rightarrow (X)\ \ (2)$

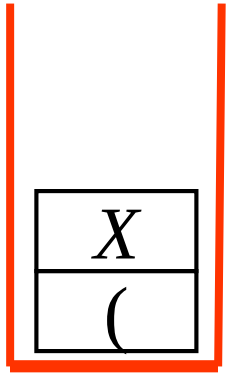
$X \rightarrow (\ )\ \ (3)$

Step 2: push non-terminal

		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





)$

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

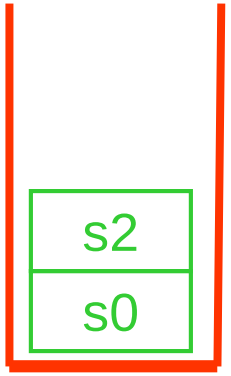
$X \rightarrow (X)\ \ (2)$

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

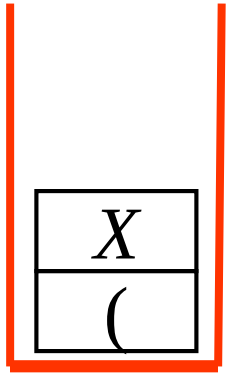
Step 3: use Goto, push new state

	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)

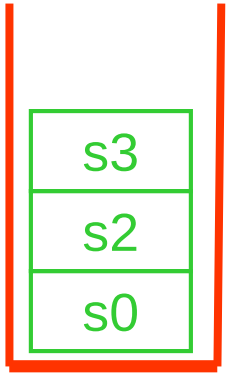
$X \rightarrow (X)$ (2)

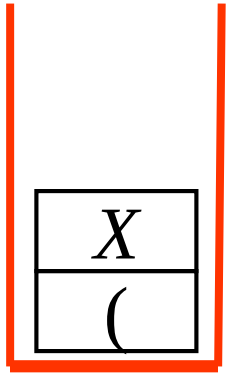
$X \rightarrow (\ )$ (3)

Step 3: 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)

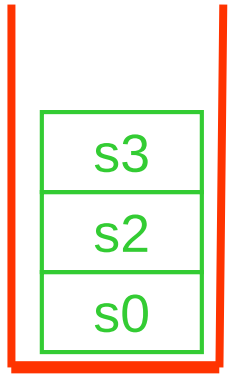
$X \rightarrow (X)$ (2)

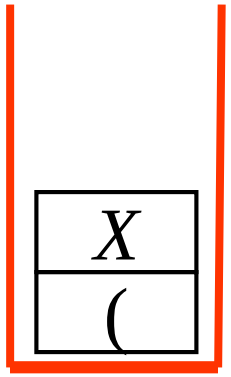
$X \rightarrow (\ )$ (3)

Parser table in action

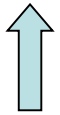
		ACTION		Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State stack Symbol stack Input Grammar





)$



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

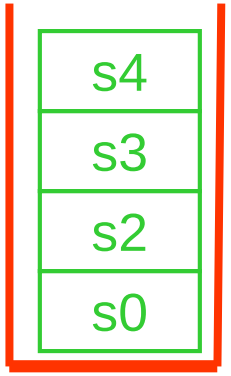
$X \rightarrow (X)$ (2)

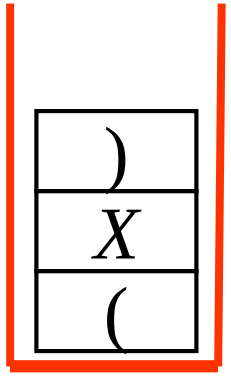
$X \rightarrow (\ )$ (3)

Parser 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





$

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

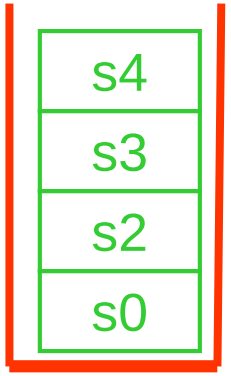
$X \rightarrow (X)$ (2)

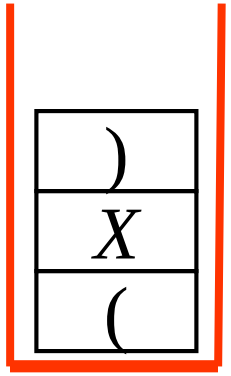
$X \rightarrow (\ )$ (3)

Parser 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





$

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

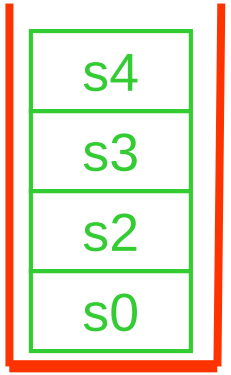
$X \rightarrow (X)\ \ (2)$

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

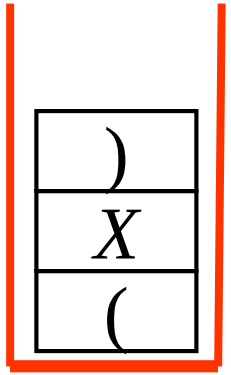
Step 1: pop stacks

		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

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

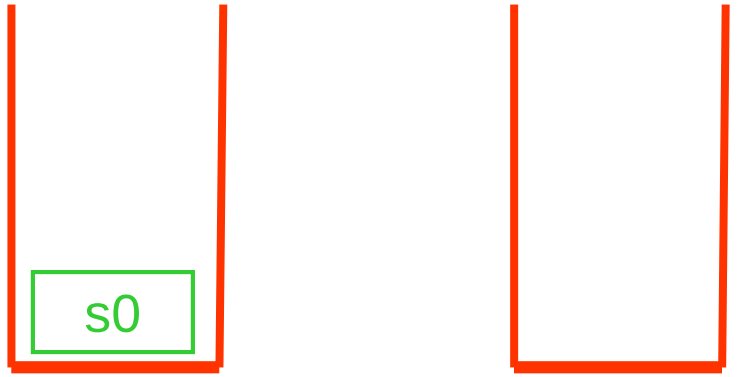
$X \rightarrow (X)\ \ (2)$

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

Step 1: 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



$

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

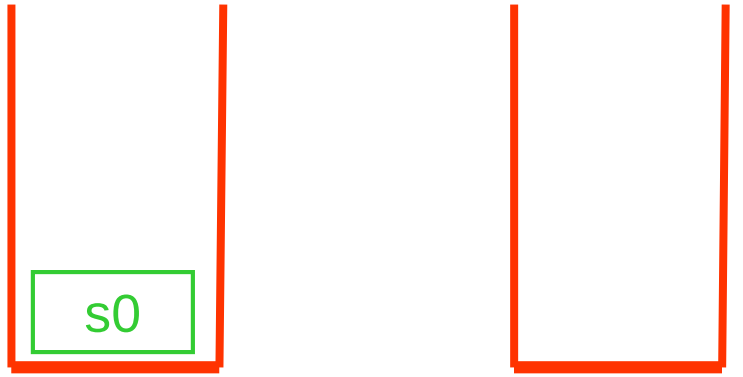
$X \rightarrow (X)\ \ (2)$

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

Step 2: push non-terminal

		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



$

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

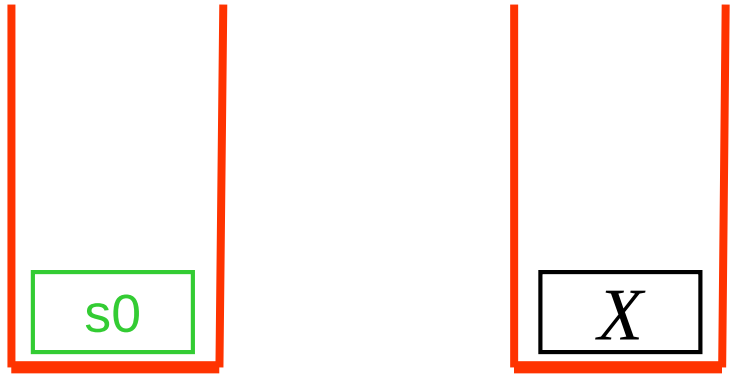
$X \rightarrow (X)\ \ (2)$

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

Step 2: push non-terminal

		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



$

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

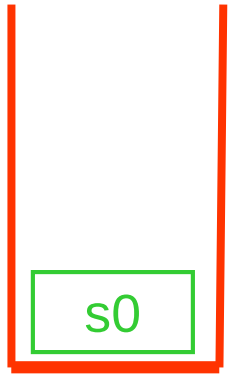
$X \rightarrow (X)\ \ (2)$

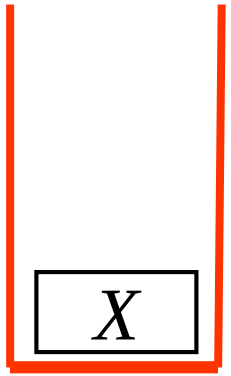
$X \rightarrow (\ )\ \ (3)$

Step 3: 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





$

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

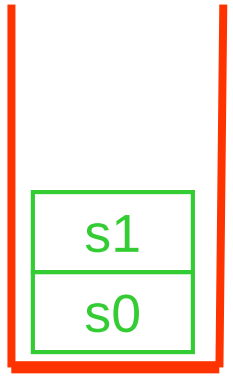
$X \rightarrow (X)\ \ (2)$

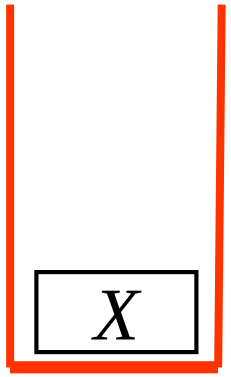
$X \rightarrow (\ )\ \ (3)$

Step 3: 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





$

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

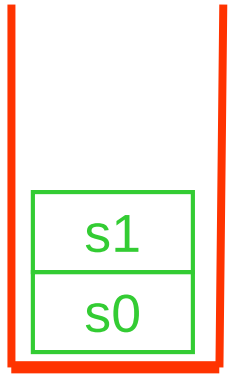
$X \rightarrow (X)\ \ (2)$

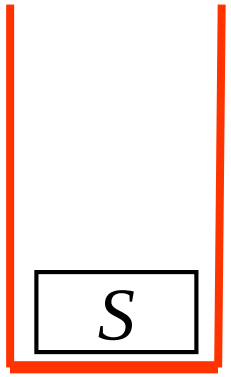
$X \rightarrow (\ )\ \ (3)$

Accept the String!

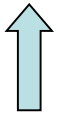
		ACTION		Goto
State	(	)	$	X
s0	shift to s2	error	error	goto s1
s1	error	error	accept	
s2	shift to s2	shift to s5	error	goto s3
s3	error	shift to s4	error	
s4	reduce (2)	reduce (2)	reduce (2)	
s5	reduce (3)	reduce (3)	reduce (3)	

State stack Symbol stack Input Grammar





$



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

$X \rightarrow (X)$ (2)

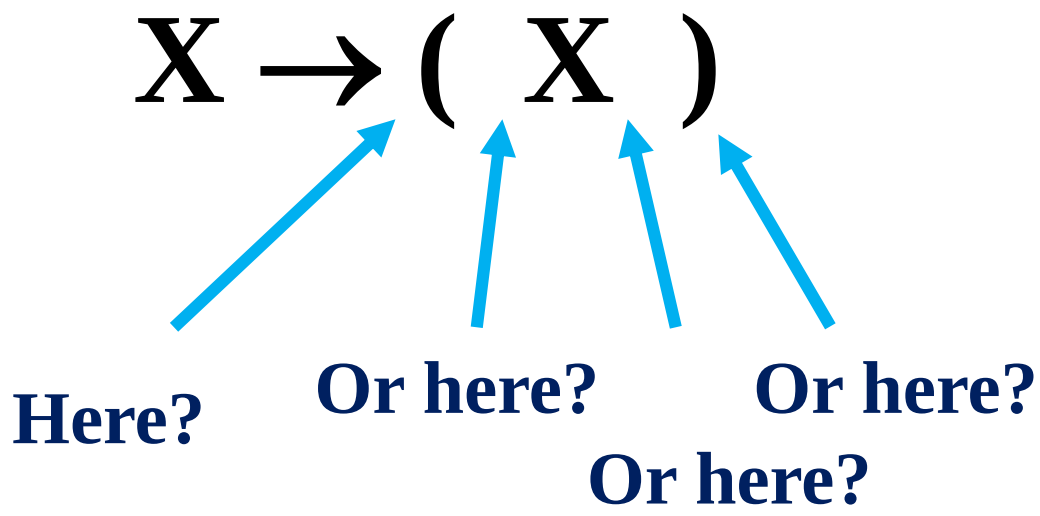
$X \rightarrow (\ )$ (3)

Building the LR Parser

- Synthesize a DFA
  - Represent all the possible states where the parser might be
  - Transitions from states with terminals and non-terminals
- Use the DFA to generate the parser table

Example

- DFA states based on items
  - We have to represent what has been already traversed in a production



- Grammar

S → X $ (1)

X → (X) (2)

X → ( ) (3)

Example

- DFA states based on items
  - We have to represent what has been already traversed in a production

$$\mathbf{X \rightarrow (\ X \ )}$$

- Grammar

  S → X $ (1)

  X → (X) (2)

  X → ( ) (3)

Production X → (X) generates 4 items:

X → • (X)

X → (• X)

X → (X •)

X → (X) •

Example

- DFA states based on items
  - We have to represent what has been already traversed in a production

- Grammar

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

$X \rightarrow (X)$ (2)

$X \rightarrow (\ )$ (3)

Items for all the productions of the grammar:

$S \rightarrow \bullet X\ \$$

$S \rightarrow X \bullet \$$

$X \rightarrow \bullet (X)$

$X \rightarrow (\bullet X)$

$X \rightarrow (X \bullet)$

$X \rightarrow (X) \bullet$

$X \rightarrow \bullet (\ )$

$X \rightarrow (\bullet\ )$

$X \rightarrow (\ \ ) \bullet$

Idea behind the items

- States correspond to sets of items
- If a state contains an item: $A \rightarrow \alpha \bullet c\ \beta$
  - The parser expects an eventual reduction using the production: $A \rightarrow \alpha\ c\ \beta$
  - The *parser* has already analyzed α
  - It expects that the input may contain c followed by β
- If a state contains an item: $A \rightarrow \alpha \bullet$
  - The *parser* has already analyzed α
  - It will reduce using: $A \rightarrow \alpha$
- If a state contains an item : $S \rightarrow \alpha \bullet \$$ and the input is empty
  - The parser accepts the input

Relation between items and actions

- If the current state contains the item: $A \rightarrow \alpha \bullet c \beta$ and the current symbol in the input is c
  - The *parser* shifts c to the symbol stack
  - The next state will contain $A \rightarrow \alpha c \bullet \beta$
- If the current state contains the item: $A \rightarrow \alpha \bullet$
  - It will reduce using: $A \rightarrow \alpha$
- If the current state contains the item: $S \rightarrow \alpha \bullet \$$ and the input is empty
  - The *parser* accepts the input

Closure() of a set of items

- Closure finds all the items of the same state
- Fixed-point algorithm for the Closure**(I)**
  - Each 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 there are none items to be add to Closure**(I)**

Examples of Closure()

- Closure({X→ (• X)})

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

- Closure({S → • X $})

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

- Items

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

Goto() of a set of items

- Goto finds the new state after a symbol of the grammar has been consumed in the current state
- Algorithm for *Goto*(I, X)
  - In which I is a set of items
  - and X is a terminal or non-terminal symbol of the grammar

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

  - Gives the new set obtained by the movement of the dot over X

Examples of Goto()

- Goto({X → (• X)}, X)

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

- Goto ({X → •(X)}, ()

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

- Items

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

Build the states of the DFA

- Start with item: $S \rightarrow \bullet\ \beta\ \$$
  - If does not exist add the first production with endmark $
  - Create the first state as being Closure({ Goal $\rightarrow \bullet$ **S** $})
- Select a state **I**
  - For each item $A \rightarrow \alpha \bullet X\ \beta$ in **I**
    - Determine Goto**(I, X)**
    - If Goto**(I, X)** is not in state, create a new state
    - Add an edge X from state **I** to the state Goto**(I, X)**
- Repeat until there are none possible modifications

Build the parser

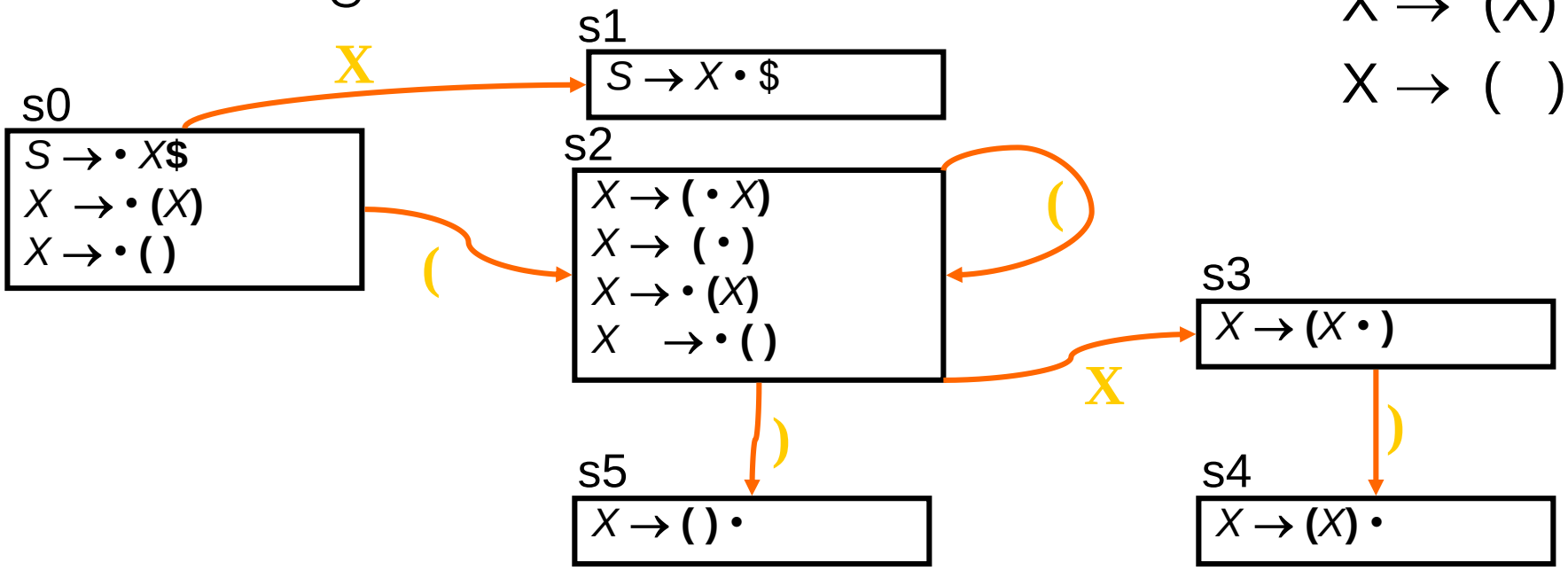
- Grammar

  S → X $ (1)

  X → (X) (2)

  X → ( ) (3)

- Build the DFA
  - DFA for the grammar:



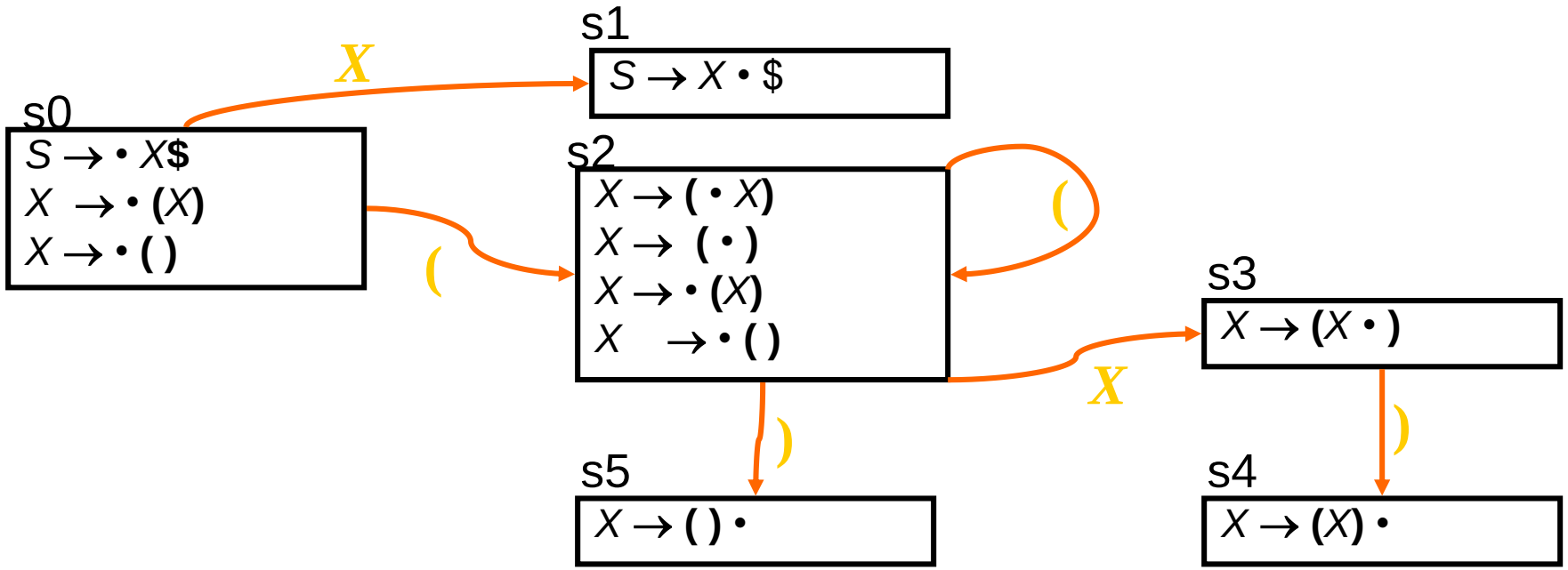
- Build the table of the parser using the DFA

Building the parser table

- For each state
  - Transition using a terminal symbol is a shift to the destination state (*shift to sn*)
  - Transition using a non-terminal state is a goto to the destination state (*goto sn*)
  - If there exists an item $\mathbf{A \rightarrow \alpha \bullet}$ in the state do a reduction with that production for all the terminals (*reduce k*)
  - If there exists an item $\mathbf{S \rightarrow X \bullet \$}$ in the state then place accept state for terminal $

Building the parser table

	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)	



Problems that can occur

- None *lookahead*
  - Vulnerability to unnecessary conflicts
    - *Shift/Reduce* conflicts (it can reduce too soon in some cases)
    - *Reduce/Reduce* conflicts
- Solution: *Lookahead*
  - Only for reductions – reduce only when the next symbol can occur after the non-terminal in the production
  - *Systematic lookahead: division of states based on next* symbol, action is always a function of the next symbol
  - It can be generalized to look ahead multiple symbols

Parser with *Lookahead* only in reductions

- **It is named as Simple LR: SLR(1) or simply SLR**
- If a state contains: $A \rightarrow \beta \bullet$
- Reduce $A \rightarrow \beta$ only if the next input symbol follow A in some derivation
- Grammar example:

$S \rightarrow X \$$ (1)

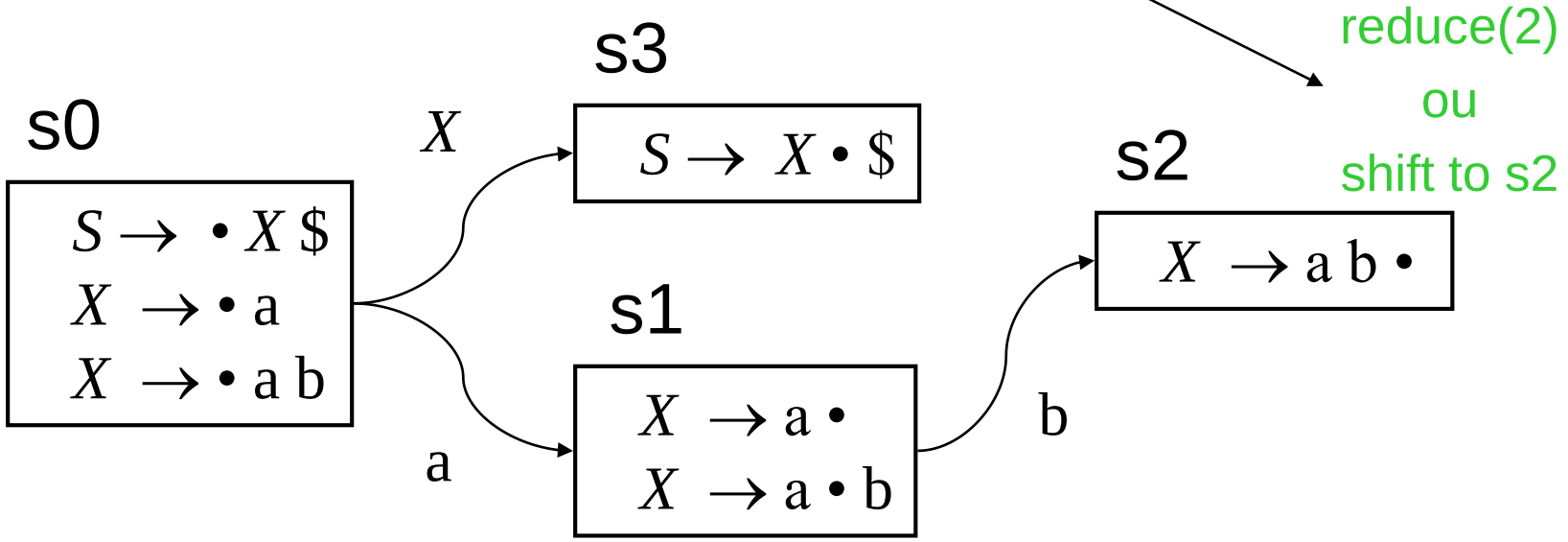
$X \rightarrow a$ (2)

$X \rightarrow a\ b$ (3)

$S \rightarrow X\ \$$ (1)
$X \rightarrow a$ (2)
$X \rightarrow a\ b$ (3)

Parser without *Lookahead*: LR(0)

		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	



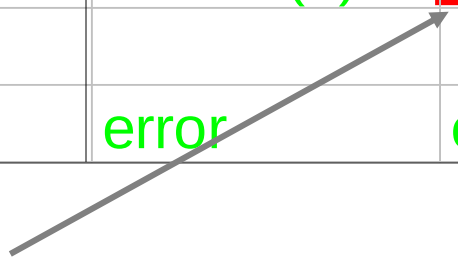
Parser table with only *lookahead* in reductions

- For each state
  - Transition with a terminal symbol is a shift to the destination state (*shift to sn*) (as before)
  - Transition with a non-terminal is a goto to the destination state (*goto sn*) (as before)
  - If there exists an item: $X \rightarrow \alpha$ in the state, do a reduce with that production as long as the input current symbol (T) can follow X in any derivation
- Eliminate non-useful reduce actions

$S \rightarrow X\ \$$ (1)
$X \rightarrow a$ (2)
$X \rightarrow a\ b$ (3)

New parser table

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



- *Reduce(2) reduction with production: X* $\rightarrow$ a
- As Follow(X) = {$}

  b never follows X in the derivations:

  resolve conflict *shift/reduce* with *shift*

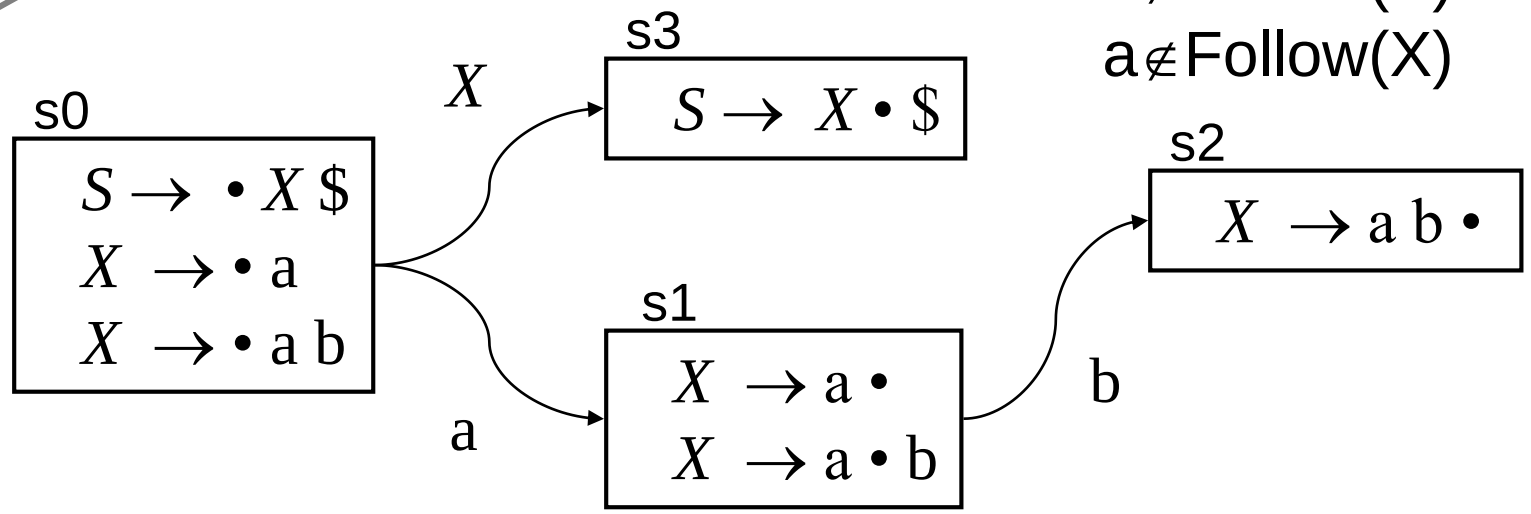
$S \rightarrow X\ \$$ (1)
$X \rightarrow a$ (2)
$X \rightarrow a\ b$ (3)

New parser table

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

b never follows X in the derivations: resolve conflict *shift/reduce* with *shift*

$b \notin Follow(X)$
$a \notin Follow(X)$



More generic lookahead

- Items contain info about potential *lookahead*, resulting in more states
  - Item of the form: [A $\rightarrow \alpha \bullet X \beta$ **c**] represent
    - Next symbol in the input is **c**
    - *Parser* already consumed α, expects to analyze X β, and then reduce with: A $\rightarrow \alpha \bullet X \beta$
- In addition to the current state in the parser table, all the parser actions are function of the *lookahead* symbol

Summary

- Parser generators – given a grammar generates a *parser*
- Technique of ascending syntactic analysis
  - Build automatically a *pushdown automata*
  - Obtains a *shift-reduce parser*
    - Finite state control + stack
    - Implementation based on table
- Conflicts: *Shift/Reduce*, *Reduce/Reduce*
- Use of *lookahead* to eliminate conflicts
  - Parser SLR(1) (eliminates non-useful reduction actions)
  - Parser LR(k) (use of generic *lookahead*)

Basic idea for LR(1)

- Split LR(0) states
  - DFA based on *lookahead*
- Reduce action is based on the item and in the *lookahead*

Items LR(1)

- Items maintain Information regarding:
  - production
  - *right-hand-side* position (the dot)
  - *lookahead symbol*
- Item LR(1) is of form [A $\rightarrow \alpha \bullet \beta$ T]
  - A $\rightarrow \alpha \beta$ is a production
  - The dot in A $\rightarrow \alpha \bullet \beta$ denotes the position
  - T is the terminal symbol or the endmark ($)
- Item [A $\rightarrow \alpha \bullet \beta$ T] implies
  - the *parser* has already analyzed α
  - If it analyzes β and the next symbol is T then the parser must reduce with A $\rightarrow \alpha \beta$

LR(1) items: example

- Grammar

  $S \rightarrow X\ \$$

  $X \rightarrow (X)$

  $X \rightarrow \varepsilon$

- Terminal symbols
  - ‘(‘ ‘)’
- Endmark
  - ‘$’

- Items LR(1)

$[S \rightarrow \bullet X\ \$,)]$
$[S \rightarrow \bullet X\ \$, (]$
$[S \rightarrow \bullet X\ \$, \$]$
$[S \rightarrow X \bullet \$,)]$
$[S \rightarrow X \bullet \$, (]$
$[S \rightarrow X \bullet \$, \$]$
$[X \rightarrow \bullet (X),)]$
$[X \rightarrow \bullet (X), (]$
$[X \rightarrow \bullet (X), \$]$
$[X \rightarrow (\bullet X),)]$
$[X \rightarrow (\bullet X), (]$
$[X \rightarrow (\bullet X), \$]$
$[X \rightarrow (X \bullet,)]$
$[X \rightarrow (X \bullet, (]$
$[X \rightarrow (X \bullet, \$]$
$[X \rightarrow (X) \bullet,)]$
$[X \rightarrow (X) \bullet, (]$
$[X \rightarrow (X) \bullet, \$]$
$[X \rightarrow \bullet,)]$
$[X \rightarrow \bullet, (]$
$[X \rightarrow \bullet, \$]$

Building the LR(1) parser

- It is necessary to define Closure() and Goto() functions for LR(1) items
- It is necessary an algorithm to build the DFA
- It is necessary an algorithm to build the parser table

Closure for LR(1)

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
return I

Goto for LR(1)

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 item: [Start $\rightarrow$ • S $?]
  - ? Is irrelevant as we will never shift $
- Determine the closure of the item and form the state
- Select one state I
  - for each item [A$\rightarrow \alpha$ • X β 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 nothing is changed

Building the LR(1) parser table

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

LR(1): example

- LR(1) parser table for the previous grammar?
- Steps to process: ()$

  - Grammar
    S → X **$**
    X → **(**X**)**
    X → **ε**
  - Terminal symbols
    - **‘(‘ ‘)’**
  - Endmark symbol
    - ‘$’

Look-Ahead LR(1) Analyzer or LALR(1)

- Motivation
  - Parser LR(1) has a high number of states
  - Simple method to eliminate states
- If two states LR(1) are identical except in the lookahead symbol of their items then merge the two states
- Result is a DFA LALR(1)
- Typically it has much less states than LR(1)
- It can have more reduce/reduce conflicts

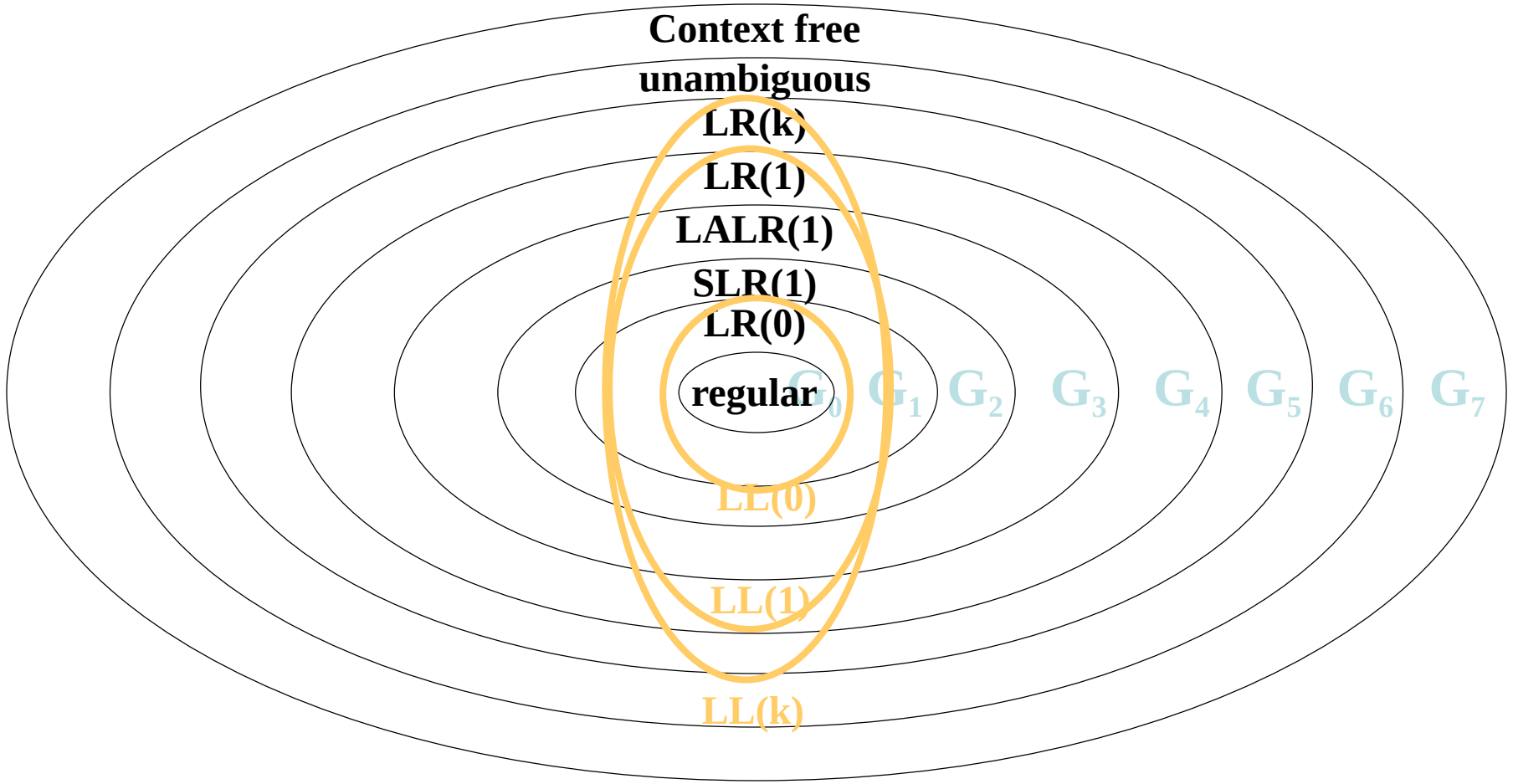
Classification of grammars

- Given a grammar, determine if it can have as syntactic analyzer:
  - LL(0), LL(1), LL(2), ..., LL(k)?
  - LR(0), LR(1), LR(2), ..., LR(k)?
  - SLR(1)?

Classification of grammars

- A grammar is said to be:
  - **LR(0)** if there exists a parser table LR(0) without conflicts (*reduce/reduce*, *shift/reduce*)
  - **SLR(1)** if there exists a parser table SLR(1) without conflicts (*reduce/reduce*, *shift/reduce*)
  - **LR(k)** if there exists a parser table LR(k) without conflicts (*reduce/reduce*, *shift/reduce*)
  - **LL(k)** if it can be analyzed by a top-down predictive parser with lookahead=k (if there exists a parser table LL(k) without conflicts)
  - …

Classification of grammars



Parser Generators

- Generating C code, http://dinosaur.compilertools.net/
  - Lex & Yacc
  - flex e bison
- Generating Java code:
  - JLex e CUP

  http://www.cs.princeton.edu/~appel/modern/java/JLex/
  http://www.cs.princeton.edu/~appel/modern/java/CUP/
  - SableCC, http://sablecc.org/
  - JavaCC (version X includes C++ generation): https://javacc.org/
- ANTLR Parser Generator (Java, C#, C++, and Python):
  - http://www.antlr.org/
- List with additional parser generators
  - http://catalog.compilertools.net/lexparse.html
  - http://catalog.compilertools.net/java.html