COMPILER DESIGN

HANDLED BY
DIVYA B
DEPT. OF CSE
VJEC, CHEMPERI

BOTTOM UP PARSING

- A bottom-up parser starts with the string of terminals itself and builds from the leaves upward, working backwards to the start symbol by applying the productions in reverse.
- A bottom-up parser searches for substrings of the working string that match the right side of some production.
- When it finds such a substring, it reduces it, i.e., substitutes the left side non-terminal for the matching right side.
- The goal is to reduce all the way up to the start symbol and report a successful parse.

RIGHTMOST DERIVATION

Consider the grammar

E → E+E

E → E*E

E → (E)

E→ id

And the input string is $id_1 + id_2 * id_3$

The rightmost derivation is:

$E \rightarrow \underline{E + E}$

$\rightarrow E + \underline{E * E}$

$\rightarrow E + E * \underline{id_3}$

$\rightarrow E + \underline{id_2} * id_3$

$\rightarrow \underline{Id_1} + id_2 * id_3$

HANDLE PRUNING

- The string of symbols to be replaced at each stage is called a **handle**.
- Reduction of a handle represents one step along the reverse of a rightmost derivation.
- The process of obtaining rightmost derivation in reverse order is called "**handle pruning**".

- Consider the input string as abbcde and the production as
  S→aABe
  A→Abc | b
  B→d

Right sentential form	Handle	Reducing production
abbcde	b	A→b
aAbcde	Abc	A→Abc
aAde	d	B→d
aABe	aABe	S→aABe
S		

- Consider the input string as id*id and the production as
  $E \to E+T \mid T$
  $T \to T*F \mid F$
  $F \to (E) \mid id$

Right sentential form	Handle	Reducing production
id*id	id	$F \to id$
F*id	F	$T \to F$
T*id	id	$F \to id$
T*F	T*F	$T \to T*F$
T	T	$E \to T$
E		

SHIFT REDUCE PARSING

- In Shift-Reduce parsing a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- The handle always appears at the top of the stack just before it is identified as the handle.
- We use $ to mark the bottom of the stack and also the right end of the input.

SHIFT REDUCE PARSING

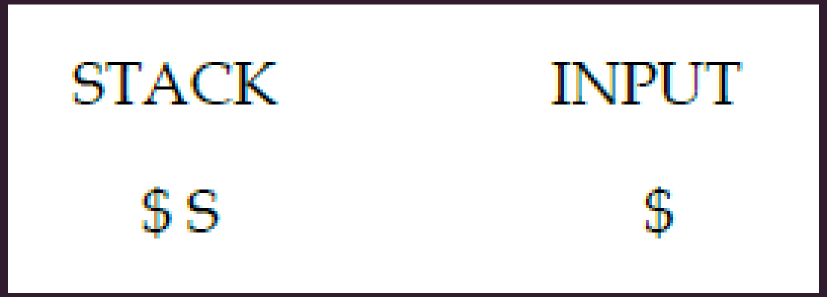
- Initially the stack is empty, and the string w is on the input.

STACK	INPUT
\$	w \$

- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack.
- It then reduces β to the head of the appropriate production.

SHIFT REDUCE PARSING

- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.



STACK	INPUT
$ S	$

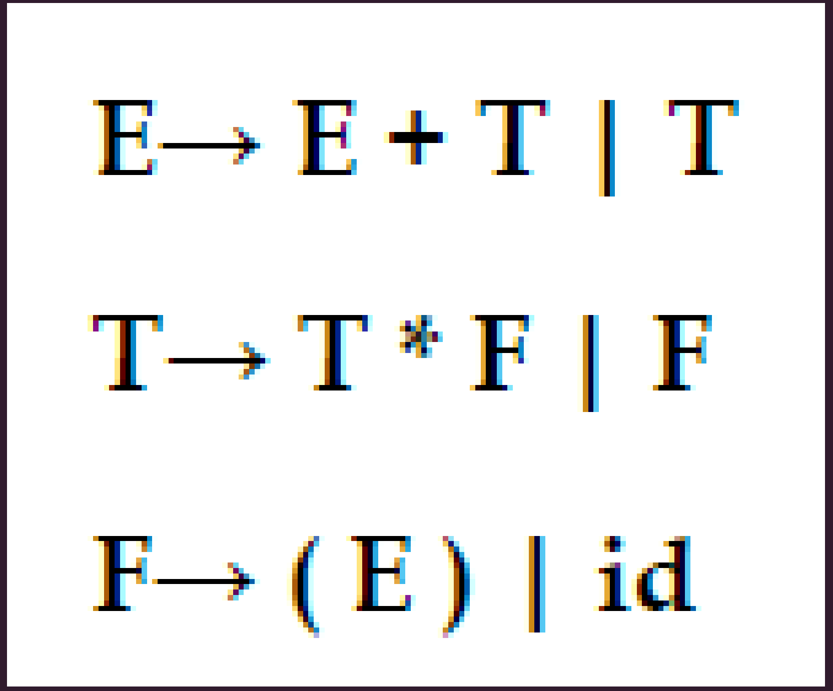
- Upon entering this configuration, the parser halts and announces successful completion of parsing.

SHIFT REDUCE PARSING

- There are four possible actions that the parser can make
  - **Shift**
    - The next input symbol is shifted onto the top of the stack.
  - **Reduce**
    - The parser knows the right end of the string to be reduced must be at the top of the stack. It must then locate the left end of the string within the stack and decide with what nonterminal to replace the string.
  - **Accept**
    - Announce successful completion of parsing.
  - **Error**
    - Discover a syntax error has occurred and calls an error recovery routine.

SHIFT REDUCE PARSING

- Consider the input string id_1*id_2 according to the expression grammar



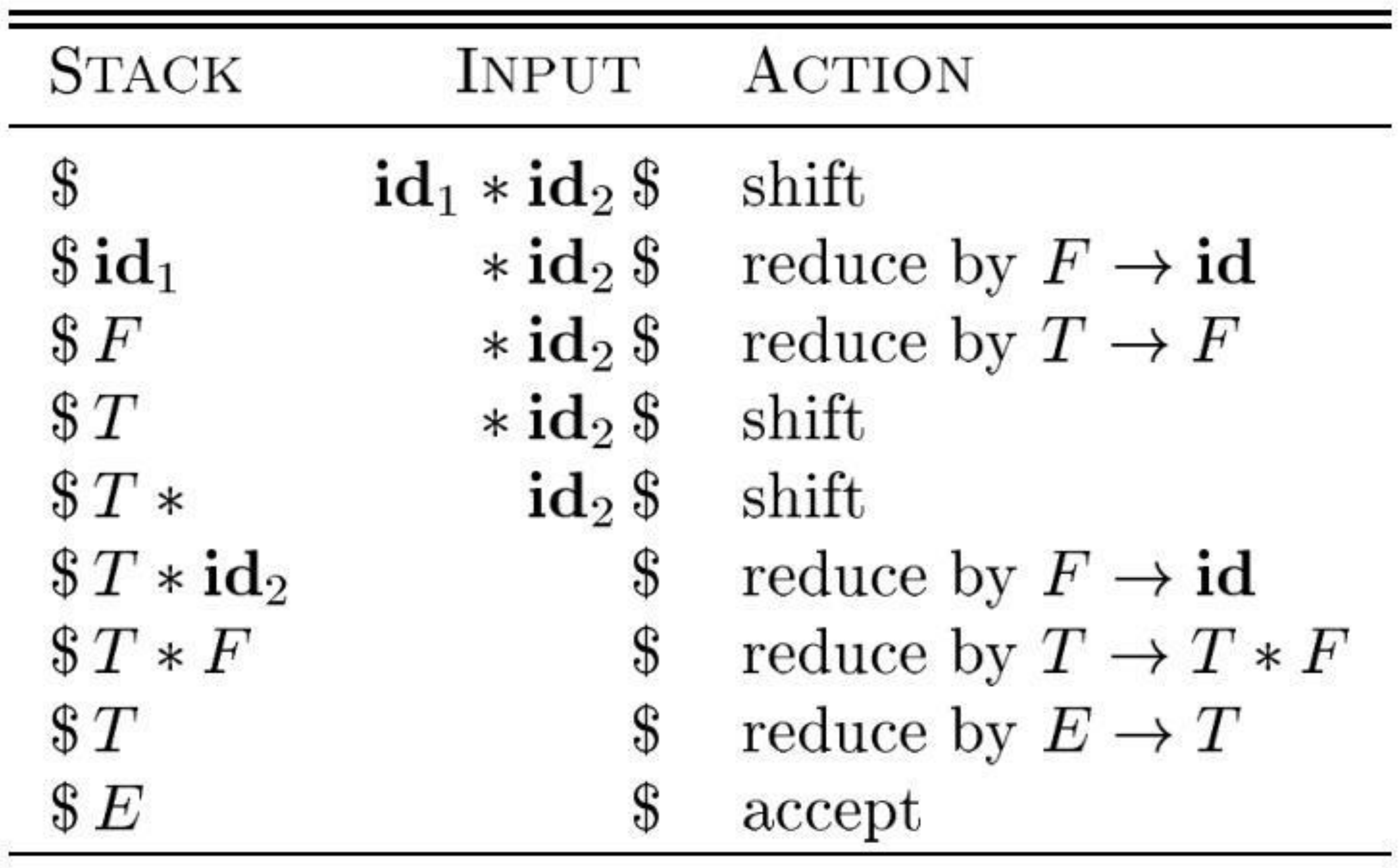
$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Divys-Compiler Design PPT

SHIFT REDUCE PARSING



STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2\,\$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2\,\$$	reduce by $F \rightarrow \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2\,\$$	reduce by $T \rightarrow F$
$\$\,T$	$* \mathbf{id}_2\,\$$	shift
$\$\,T *$	$\mathbf{id}_2\,\$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$\,T$	$\$$	reduce by $E \rightarrow T$
$\$\,E$	$\$$	accept

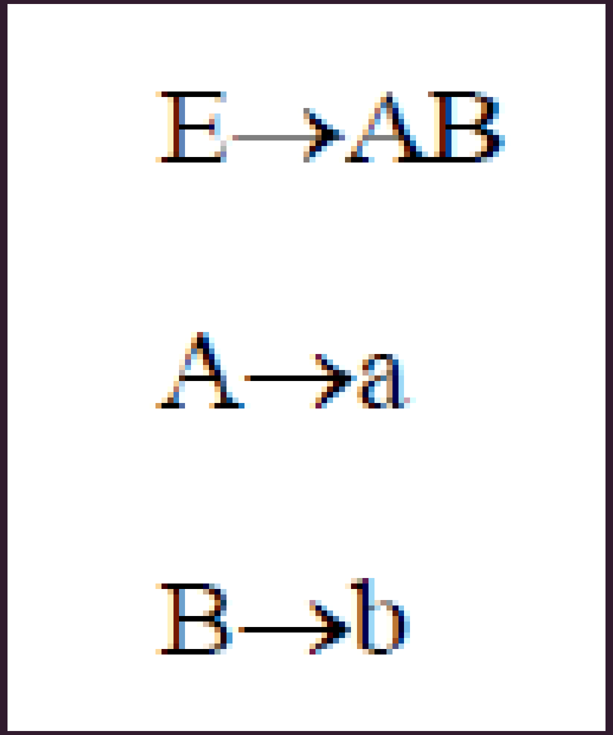
SHIFT REDUCE PARSING

- **Conflicts during shift reduce parsing**
  - There are context-free grammars for which shift-reduce parsing cannot be used.
  - Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack and also the next k input symbols, cannot decide whether to shift or to reduce (**a shift/reduce conflict**), or cannot decide which of several reductions to make (**a reduce/reduce conflict**).

OPERATOR PRECEDENCE PARSING

- Bottom-up parsers for a large class of context-free grammars can be easily developed using operator grammars.
- A Grammar G is Operator Grammar if it has the following properties
  - Production should not contain ϵ on its right side.
  - There should not be two adjacent non-terminals at the right side of production.

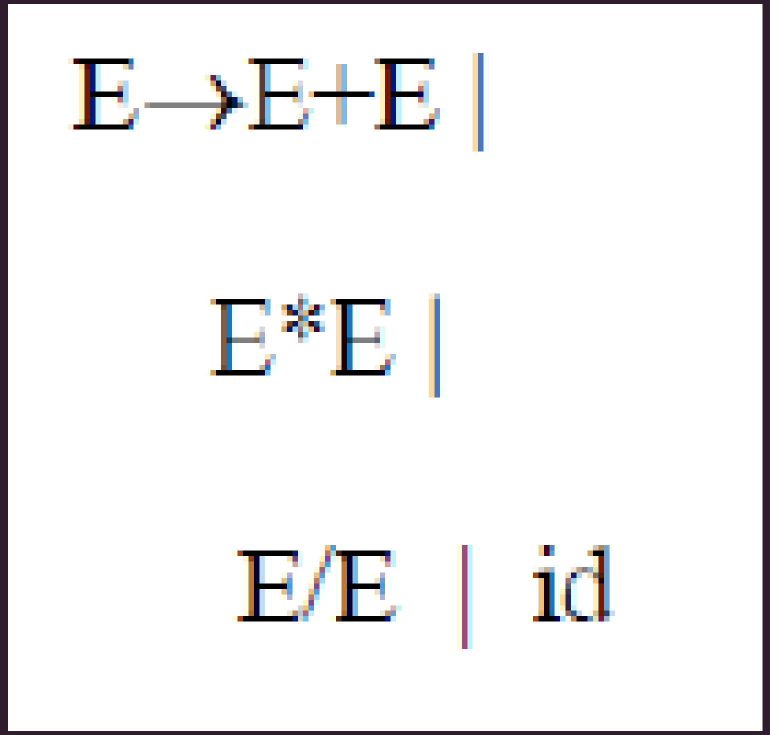
OPERATOR PRECEDENCE PARSING


$E \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$

Not operator grammar


$E \rightarrow E+E \mid$

$E*E \mid$

$E/E \mid id$

Operator grammar

OPERATOR PRECEDENCE PARSING

- In operator precedence parsing, three precedence relations are defined between pairs of terminals.

Relation	Meaning
$a <\cdot b$	a yields precedence to b (b has higher precedence than a)
$a =\cdot b$	a has same precedence as b
$a \cdot>$ b	a takes precedence over b (b has lower precedence than a)

OPERATOR PRECEDENCE PARSING

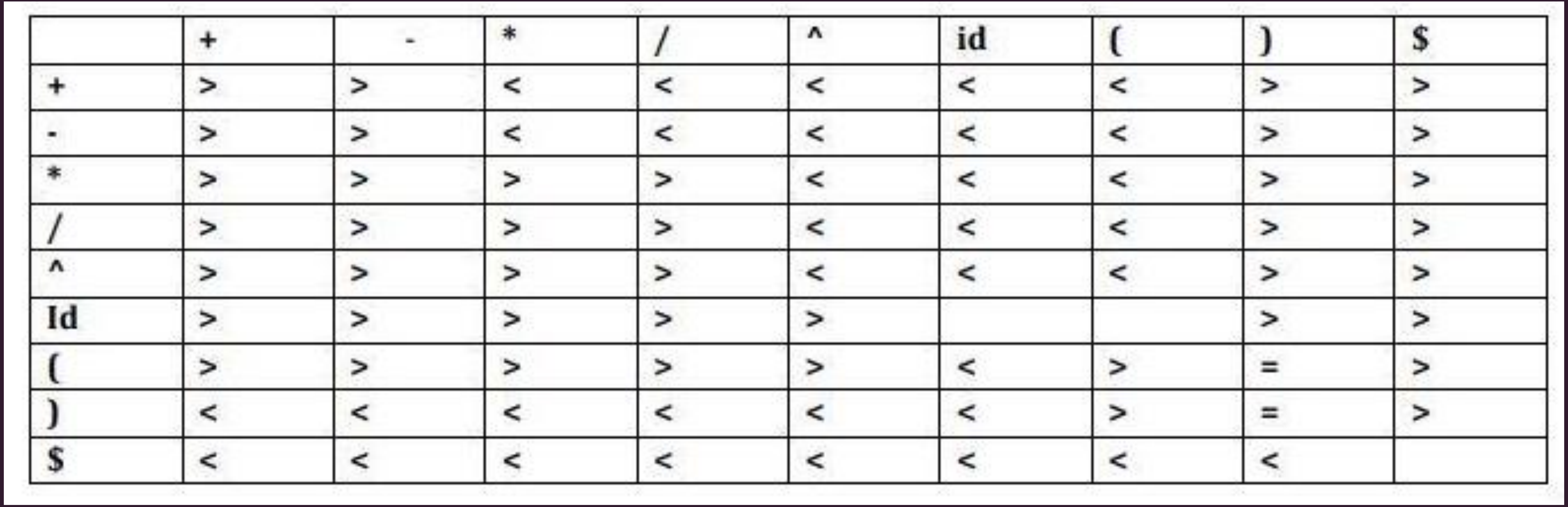
- Consider the input string id + id * id and the grammar as
  E⟶ E+E | E-E | E*E | E/E | (E) | -E | id
  The corresponding precedence relations is

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*****	<·	·>	·>	·>
$	<·	<·	<·	A

id, a, b , c - high
$ - low
+ ·> +
* ·> *
* ·> +
id ≠ id – leave as blank
$ and $ - Accept

String after precedence relation inserted is $ <. id .> + <. id .> * <. id .> $

Stack	Relation	Input	Action
$	<·	id+id*id$	Shift
$id	·>	+id*id$	Reduce by E→id
$E	<·	+id*id$	Shift
$E+	<·	id*id$	Shift
$E+id	·>	*id$	Reduce by E→id
$E+E	<·	*id$	Shift
$E+E*	<·	id$	Shift
$E+E*id	·>	$	Reduce by E→id
$E+E*E	·>	$	Reduce by E→E*E
$E+E	·>	$	Reduce by E→E+E
$E		$	Accept



	+	-	*	/	^	id	(	)	$
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
^	>	>	>	>	<	<	<	>	>
Id	>	>	>	>	>			>	>
(	>	>	>	>	>	<	>	=	>
)	<	<	<	<	<	<	>	=	>
$	<	<	<	<	<	<	<	<	