LR(0) and SLR parse table construction

Wim Bohm and Michelle Strout

CS, CSU

Parse table for List grammar

0: S' → S$ 1: S→(L)   2: S→x

3: L→S   4: L→L , S

	(	)	x	,	$	S	L
0	s2		s1			g3	
1	r2	r2	r2	r2	r2		
2	s2		s1			g6	g4
3					a		
4		s5		s7			
5	r1	r1	r1	r1	r1		
6	r3	r3	r3	r3	r3		
7	s2		s1			g8	
8	r4	r4	r4	r4	r4		

parse (x,(x))$

stack	*input*	*action*
0	*(x,(x))$*	*s2*
0(2	*x,(x))$*	*s1*
0(2x1	*,(x))$*	*r2: S→x*
0(2S6	*,(x))$*	*r3: L→S*
0(2L4	*,(x))$*	*s7*
0(2L4,7	*(x))$*	*s2*
0(2L4,7(2	*x))$*	*s1*
0(2L4,7(2x1	*))$*	*r2: S→x*
0(2L4,7(2S6	*))$*	*r3: L→S*
0(2L4,7(2L4	*))$*	*s5*
0(2L4,7(2L4)5	*)$*	*r1: S →(L)*
0(2L4,7S8	*)$*	*r4: L→L,S*
0(2L4	*)$*	*s5*
0(2L4)5	*$*	*r1:S→(L)*
03S	*$*	*a*

LR(0) table construction

Example grammar for Nested Lists:

0: S' → S$ 1: S→(L) 2: S→x 3: L→S 4: L→L , S

We start with an empty stack and with a complete S$ sentence on input

We indicate this as follows: S' →. S$

this (a rule with a dot in it) is called an item,

it indicates what is in the stack (left of .)

and what is to be expected on input (right of .)

The input can start with anything S can start with, eg an x or a (

We indicate this as follows:	**S' →.S$**
(we are making a DFA	**S→.x**
through another sub-set closure	**S→.(S)**
remember the NFA → DFA)	

We call this a state: state 0, the start state with an empty prefix (V[ε])

Shift, reduce, goto actions in LR(0) table construction

S' → .S$
S → .x
S → .(L)

0:V[ε]

S → x.

1:V[x]

goto action:

Also in state 0 we could have reduced to S

S' → S.$

3:V[S]

shift action:

In state 0 we can shift an x or shift a (

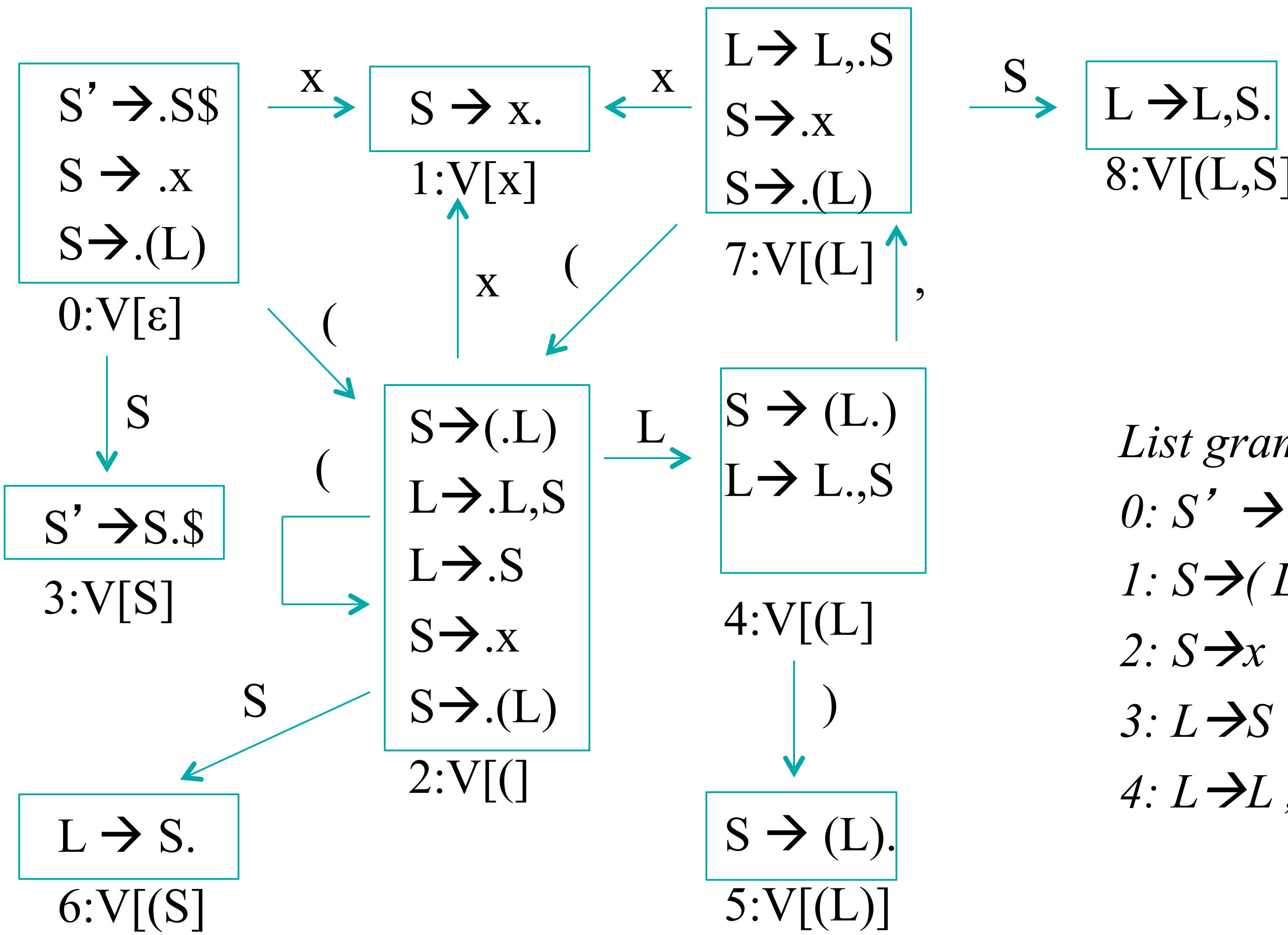
S → (.L)
L → .L,S
L → .S
S → .x
S → .(L)

2:V[(]

In state 1 we are at the end of an item. This will give rise to a **reduce action**

Transitions: the shifts and gotos explicitly connect the states. The reduces implicitly move to another state by popping the rhs off the stack, after which a goto with the lhs will produce a new next state

LR(0) states and transitions



List grammar

0: S' → S$

1: S → (L)

2: S → x

3: L → S

4: L → L , S

LR(0) Closure, Goto, State Diagram, Reduce

Closure(I): ***// state I***

repeat

for any item $A \rightarrow \alpha . X\beta$

for any $X \rightarrow \gamma$

I += $X \rightarrow . \gamma$

until I does not change

Goto(I,X): ***// state I , symbol X***

if X==$ return {} // no gotos for $

J = {}   // new state

for any item $A \rightarrow \alpha . X\beta$ in I

J += $A \rightarrow \alpha X . \beta$

return Closure(J) // close it

State Diagram construction

T = Closure({ $S' \rightarrow .S\$$ });   // states

E = {}   // edges (gotos and shifts)

repeat until no change in E or T

for each state I in T

for each item $A \rightarrow \alpha . X\beta$ in I

J = Goto(I,X);

T +=J ;

E += (X: (I,J)) // the edge (I,J) labeled X

Reduce(T):

R = {}

for each state I in T

for each item $A \rightarrow \alpha .$

R += (I, $A \rightarrow \alpha$)

Applying the Algorithm to the Nested Lists Example

$S' \rightarrow .S\$$

0:V[ε]

List grammar

0: $S' \rightarrow S\$$

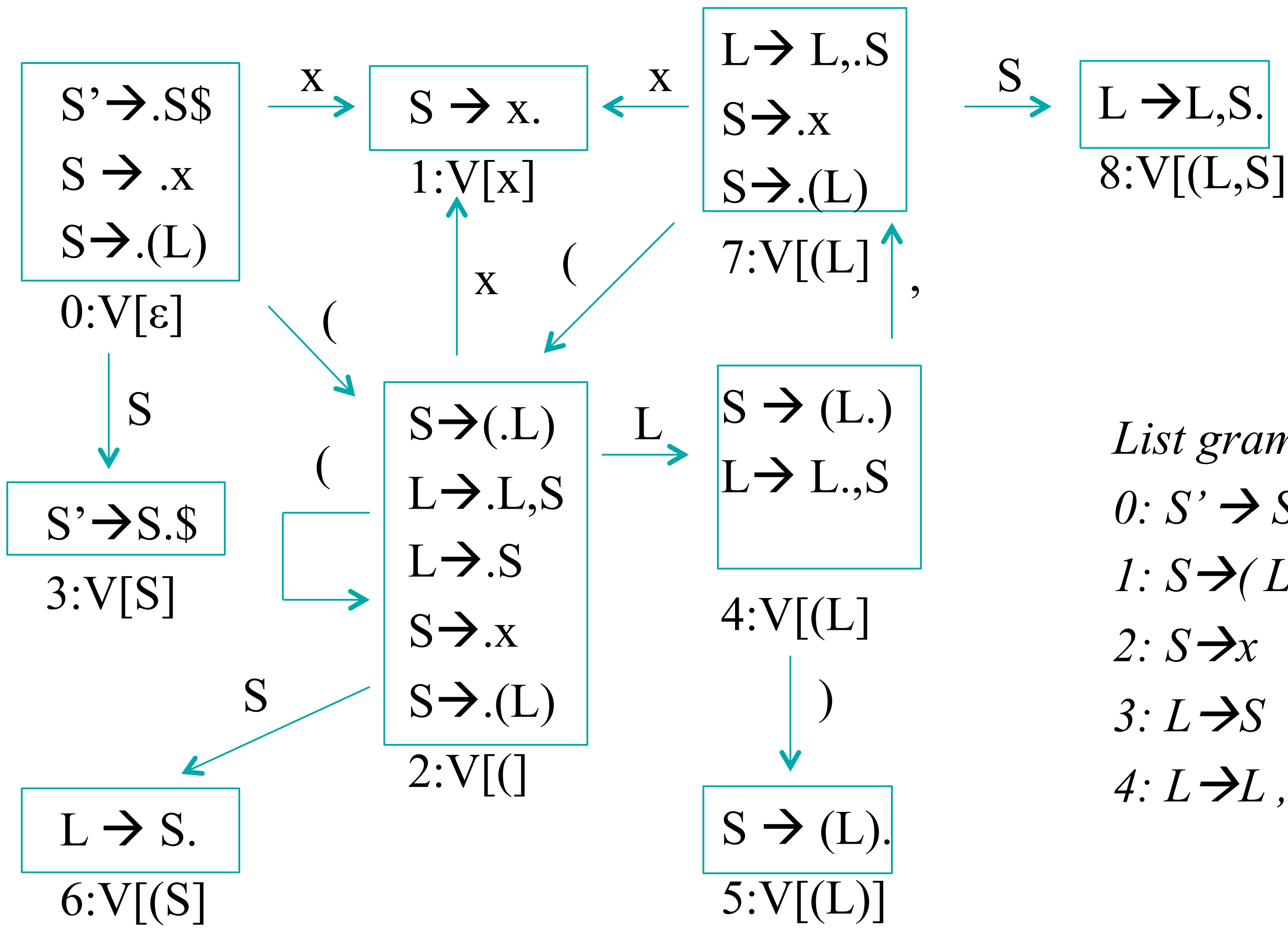
1: $S \rightarrow (L)$

2: $S \rightarrow x$

3: $L \rightarrow S$

4: $L \rightarrow L , S$

LR(0) states and transitions



List grammar

0: S’ → S$

1: S→(L)

2: S→x

3: L→S

4: L→L , S

LR(0) parse table construction

Parse table

rows: states

columns: terminals (for shift and reduce actions)

non-terminals (for goto actions)

For each edge (X: (I, J))

if X is terminal, put shift J at (I, X)

if X is non-terminal, put goto J at (I, X)

if I contains $S' \rightarrow S \;.\; \$$, put accept at (I, \$)

if I contains $A \rightarrow \alpha \;.$ where $A \rightarrow \alpha \;.$ has grammar rule number n

for each terminal x, put reduce reduce n at (I, x)

Parse table for List grammar

0: S' → S$ *1: S → (L)*   *2: S → x*

3: L → S   *4: L → L , S*

	(	)	x	,	$	S	L
0	**s2**		**s1**			**g3**	
1	**r2**	**r2**	**r2**	**r2**	**r2**		
2	**s2**		**s1**			**g6**	**g4**
3						**a**	
4		**s5**		**s7**			
5	**r1**	**r1**	**r1**	**r1**	**r1**		
6	**r3**	**r3**	**r3**	**r3**	**r3**		
7	**s2**		**s1**			**g8**	
8	**r4**	**r4**	**r4**	**r4**	**r4**		

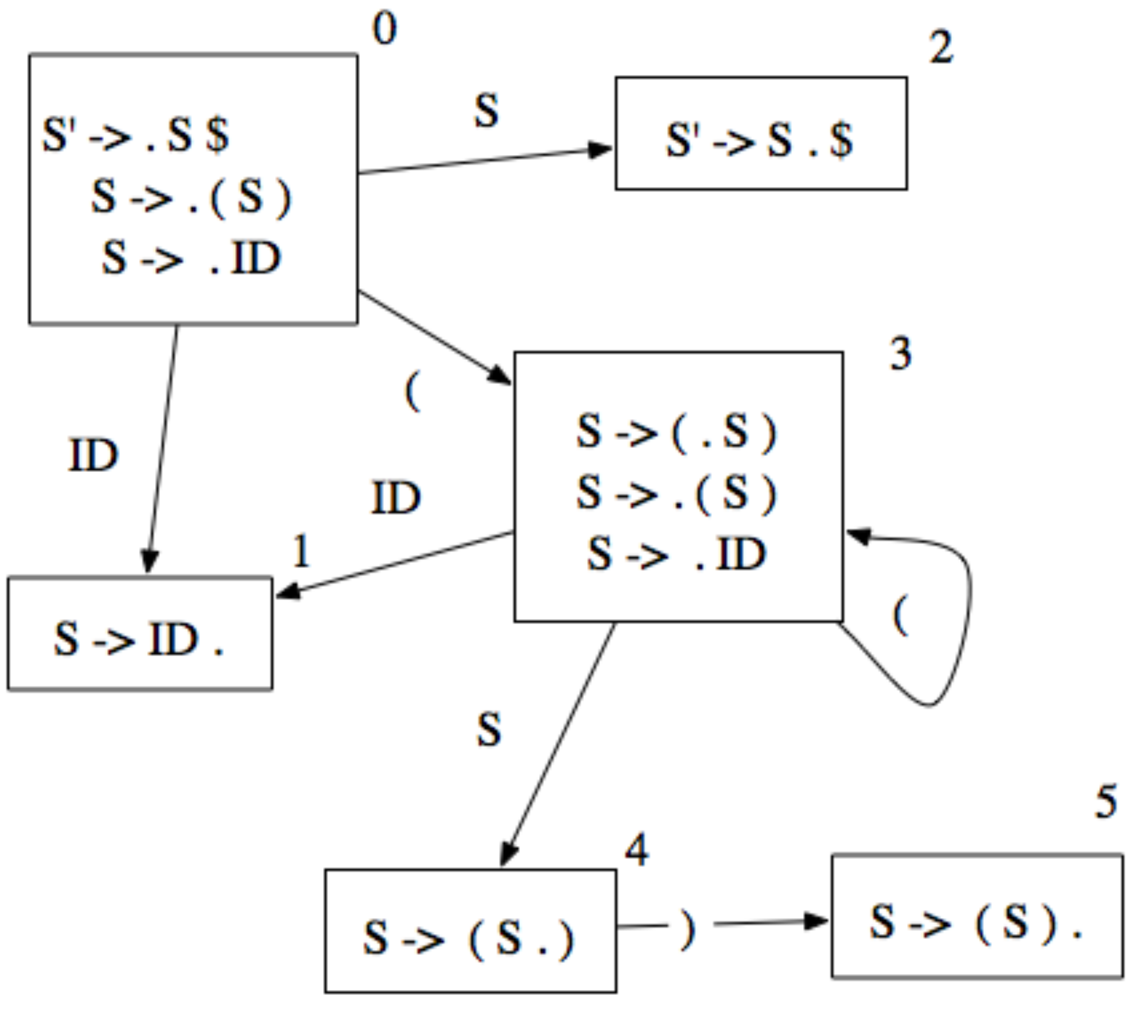
parse (x,(x))$

stack	*input*	*action*
0	*(x,(x))$*	*s2*
0(2	*x,(x))$*	*s1*
0(2x1	*,(x))$*	*r2: S→x*
0(2S6	*,(x))$*	*r3: L→S*
0(2L4	*,(x))$*	*s7*
0(2L4,7	*(x))$*	*s2*
0(2L4,7(2	*x))$*	*s1*
0(2L4,7(2x1	*))$*	*r2: S→x*
0(2L4,7(2S6	*))$*	*r3: L→S*
0(2L4,7(2L4	*))$*	*s5*
0(2L4,7(2L4)5	*)$*	*r1: S →(L)*
0(2L4,7S8	*)$*	*r4: L→L,S*
0(2L4	*)$*	*s5*
0(2L4)5	*$*	*r1:S→(L)*
03S	*$*	*a*

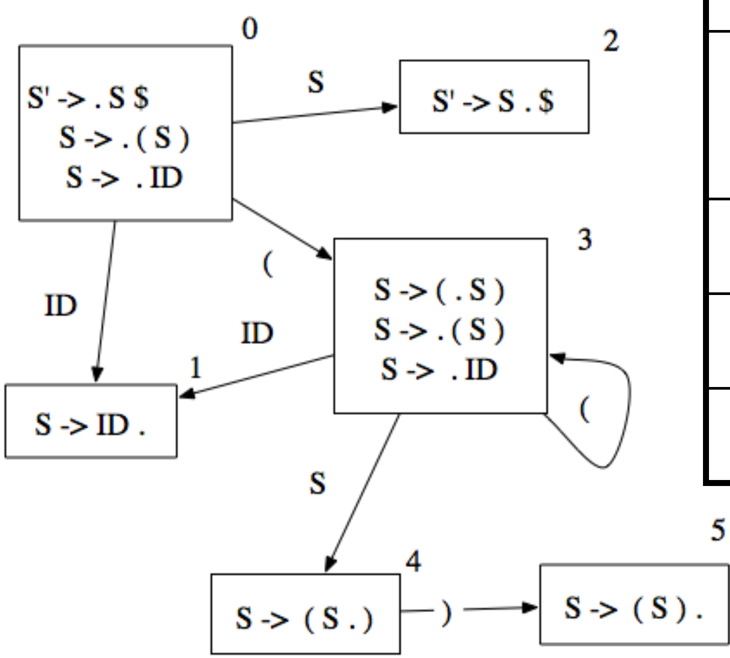
Building the LR Parse Table for LR(0), nested parens example

```
[0] S -> ( S )
[1] S’  -> S EOF
[2] S -> ID
```

LR(0) states for nested parens example



Building the Table from the State Diagram



	Action				Goto
State	(	)	$	ID	S
0	s3			s1	2
1	r2	r2	r2	r2	
2			accept		
3	s3			s1	4
4		s5			
5	r0	r0	r0	r0	

Suggested Exercise: Building the LR Parse Table for LR(0)

```
(0) S' -> E $
(1)E -> E || B
(2)E -> B
(3)B -> t
(4)B -> f
```

Problem with LR(0): shift reduce conflict

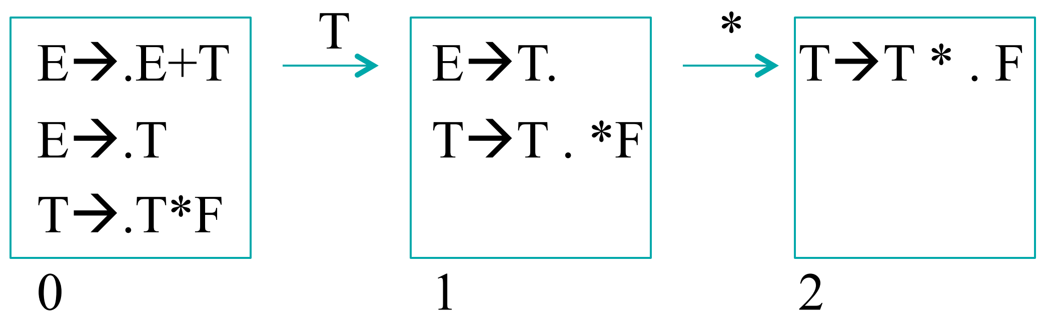
If there is an item A→α . item in I , we reduce for all terminals

This can cause CONFLICTS:

E→ E+T

E → T

T → T*F



In state 1:

we reduce (E→T.) AND we shift (T→T . *F)

What should we do?

LR(0) shift reduce conflict

We can resolve the conflict by looking at a right most derivation:

E→T→T*F→T*id→F*id→id*id

Stack	input	action
	id*id$	S
id	*id$	R: F→id
F	*id$	R: F→T
T	*id$	S
T*	id$	S
T*id	$	R: F→id
T*F	$	R: T→T*F
T	$	R:E→
E	$	accept

We should shift. WHY?

SLR parsing

SLR parsing is LR(0) parsing, but with a different reduce rule:

For each edge (X: (I, J))

if X is terminal, put shift J at (I, X)

if I contains A→α . where A→α . has rule number n

for each terminal x in Follow(A), put reduce reduce n at (I, x)

Build an SLR parser for our expression grammar

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

Need to build the transition diagram and follow sets

to decide where to put the reduce actions in the SLR table

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

SLR parse table (reduces only for follows)

State	id	+	*	(	)	$	E	T	F
0	s5			s4			g1	g2	g3
1		s6				a			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			g8	g2	g3
5		r6	r6		r6	r6			
6	s5			s4				g9	g3
7	s5			s4					g10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Stack	*input*	*action*
0	*a*(b+c)$*	*s5*
0a5	**(b+c)$*	*r6: F→id*
0F3	**(b+c)$*	*r4: T→F*
0T2	**(b+c)$*	*s7*
*0T2*7*	*(b+c)$*	*s4*
*0T2*7(4*	*b+c)$*	*s5*
*0T3*7(4b5*	*+c)$*	*r6: F→id*
*0T3*7(4F3*	*+c)$*	*r4: T→F*
*0T3*7(4T2*	*+c)$*	*r2: E→T*
*0T3*7(4E8*	*+c)$*	*s6*
*0T3*7(4E8+6*	*c)$*	*s5*
*0T3*7(4E8+6c5*	*)$*	*r6: F→id*
*0T3*7(4E8+6F3*	*)$*	*r4: T→F*
*0T3*7(4E8+6T9*	*)$*	*r1: E→E+T*
*0T3*7(4E8*	*)$*	*S11*
*0T3*7(4E8)11*	*$*	*r5: F→(E)*
*0T3*7F10*	*$*	*r3: T→T*F*
0T2	*$*	*r2: E→T*
0E1	*$*	*a*

E→E+T | T   T→T*F | F   F→ (E) | id   S→E$ input: a*(b+c)$

Stack	*input*	*action*
	a(b+c)$*	*S*
a	**(b+c)$*	*R: F→id*
F	**(b+c)$*	*R: T→F*
T	**(b+c)$*	*S*
*T**	*(b+c)$*	*S*
T(*	*b+c)$*	*S*
T(b*	*+c)$*	*R: F→id*
T(F*	*+c)$*	*R: T→F*
T(T*	*+c)$*	*R: E→T*
T(E*	*+c)$*	*S*

Stack	*input*	*action*
T(E+*	*c)$*	*S*
T(E+c*	*)$*	*R: F→id*
T(E+F*	*)S*	*R: T→F*
T(E+T*	*)$*	*R: E→E+T*
T(E*	*)$*	*S*
T(E)*	*$*	*R: F→(E)*
*T*F*	*$*	*R: T→T*F*
T	*$*	*R: E→T*
E	*$*	*accept*

*S→E$→T$→T*F$→T*(E)$→T*(E+T)$→T*(E+F)$→ T*(E+id)$→T*(T+id)$→*
T(F+id)$→T*(id+id)$→ F*(id+id)$→id*(id+id)$*

Rightmost derivation in reverse