More Parsing Algorithms

Eelco Visser

TU Delft

CS4200 | Compiler Construction | October 15, 2020

This Lecture

Predictive parsing

- predictive/recursive descent parsing
- LL parsing
- LL(k) grammars

Generalized LR Parsing

- LR parsing with shift/reduce conflicts

Predictive (Recursive Descent) Parsing

A Theory of Language: Formal Languages

Vocabulary Σ

- finite, nonempty set of elements (words, letters)
- alphabet

String over Σ

- finite sequence of elements chosen from Σ
- word, sentence, utterance

Formal language λ

- set of strings over a vocabulary Σ
- $\lambda \subseteq \Sigma^*$

A Theory of Languages: Formal Grammars

Formal grammar G = (N, Σ, P, S)

- nonterminal symbols N
- terminal symbols Σ
- production rules P ⊆ (N∪Σ)* N (N∪Σ)* × (N∪Σ)*
- start symbol S∈N

Grammar classes

- type-0, unrestricted
- type-1, context-sensitive: (a A c, a b c)
- type-2, context-free: P ⊆ N × (N ∪ Σ)*
- type-3, regular: (A, x) or (A, xB)

A Theory of Languages: Formal Languages

Formal grammar

- $G = (N, \Sigma, P, S)$

Derivation relation

- $\Rightarrow_G \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- $\alpha\ \beta\ \gamma \Rightarrow_G \alpha\ \beta'\ \gamma \iff \exists(\beta, \beta') \in P$

Formal language

- $L(G) \subseteq \Sigma^*$
- $L(G) = \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$

Classes of formal languages

Predictive Parsing: Recursive Descent

```
Exp = “while” Exp “do” Exp
```

```
public void parseExp() {
    consume(WHILE);
    parseExp();
    consume(DO);
    parseExp();
}
```

Predictive Parsing: Lookahead

```
Exp = “while” Exp “do” Exp
Exp = “if” Exp “then” Exp “else” Exp
```

```
public void parseExp() {

   switch current() {
      case WHILE: consume(WHILE); parseExp(); ...; break;
      case IF   : consume(IF); parseExp(); ...; break;
      default   : error();
   }
}
```

Predictive Parsing: Parse Table

Rows

- nonterminal symbols N
- symbol to parse

Columns

- terminal symbols Σ^k
- look ahead k

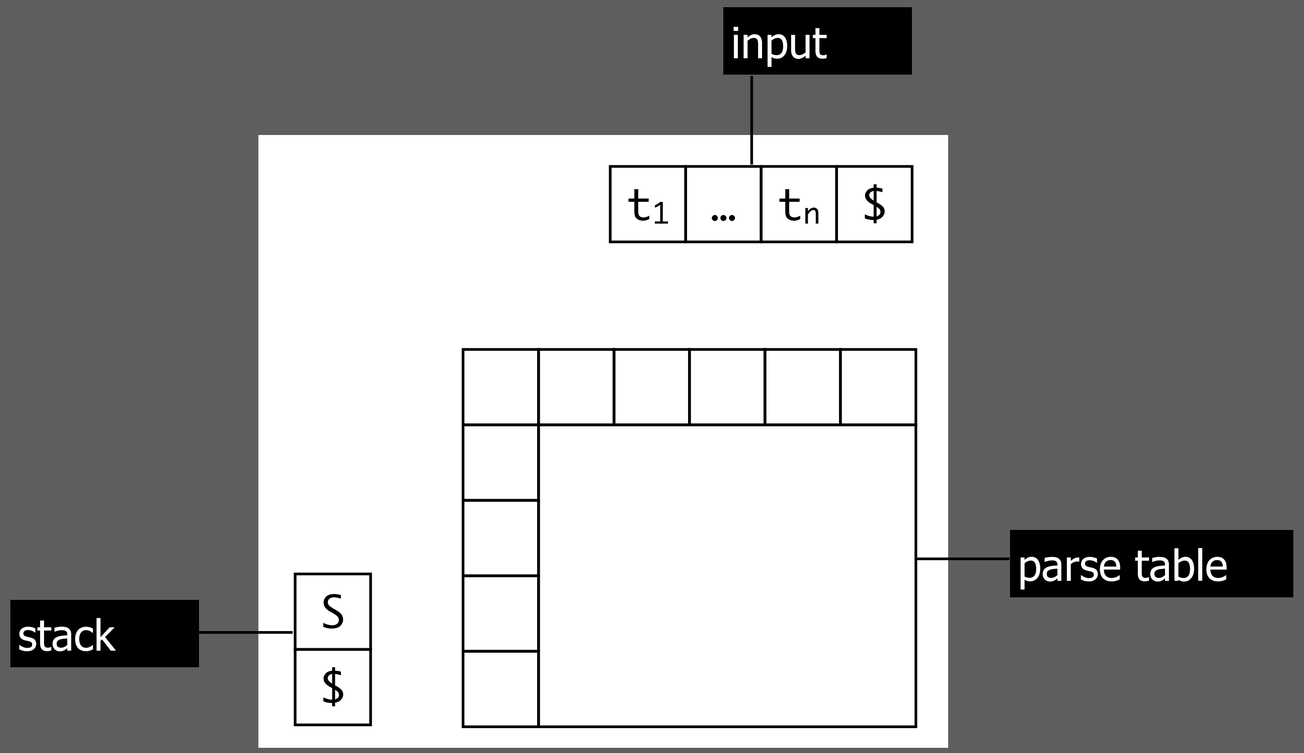
Entries

- production rules P
- possible conflicts

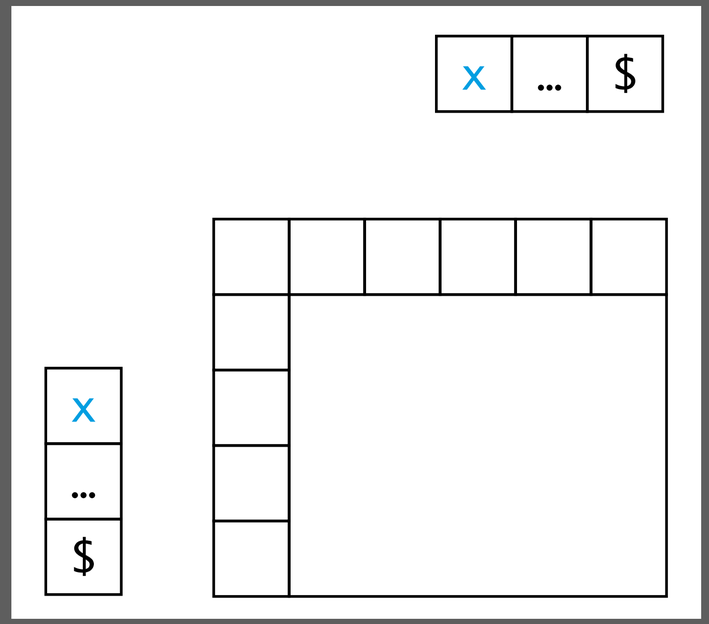
	T_1	T_2	T_3	...
N_1	$N_1 \rightarrow ...$		$N_1 \rightarrow ...$	
N_2		$N_2 \rightarrow ...$		
N_3		$N_3 \rightarrow ...$	$N_3 \rightarrow ...$	
N_4	$N_4 \rightarrow ...$			
N_5		$N_5 \rightarrow ...$		
N_6	$N_6 \rightarrow ...$	$N_6 \rightarrow ...$		
N_7			$N_7 \rightarrow ...$	
N_8	$N_8 \rightarrow ...$	$N_8 \rightarrow ...$	$N_8 \rightarrow ...$	
...				

With N on the stack and T in the input, predict P

Predictive Parsing: Automaton

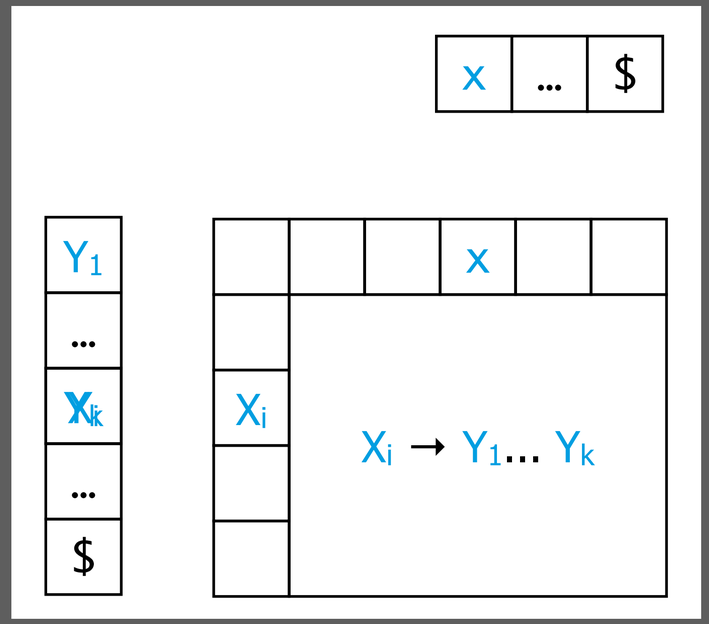


Predictive Parsing: Automaton



Recognize predicted symbol

Predictive Parsing: Automaton



Predict production

LL Parse Tables

Predictive Parsing: Filling the Table

entry $X = \alpha \in P$ at row X and column T

$T \in$ FIRST(α) — letters that α can start with

nullable(α) $\wedge$ $T \in$ FOLLOW(X) — letters that can follow X

$w \Rightarrow_G^* \varepsilon$

Predictive Parsing: Nullable

nullable(X)

- (X, ε) ∈ P ⟹ nullable(X)
- (X0, X1…Xk)∈P ∧ nullable(X1) ∧ … ∧ nullable(Xk)
  ⟹ nullable(X0)

nullable(α)

- nullable(ε)
- nullable(X1 … Xk) = nullable(X1) ∧ … ∧ nullable(Xk)

Predictive Parsing: First Set

FIRST(X)

- $X \in \Sigma$: FIRST(X) = {X}
- $(X0, X1 \ldots Xi \ldots Xk) \in P \wedge \text{nullable}(X1 \ldots Xi) \Rightarrow$

  FIRST(X0) ⊇ FIRST(Xi+1)

FIRST(w)

- FIRST(ε) = {}
- $\neg$nullable(X) ⇒ FIRST(Xw) = FIRST(X)
- nullable(X) ⇒ FIRST(Xw) = FIRST(X) ∪ FIRST(w)

Predictive Parsing: Follow Set

FOLLOW(X)

- (X0, X1 … Xi … Xk)∈P ∧ nullable(Xi+1 … Xk)
  ⇒ FOLLOW(Xi) ⊇ FOLLOW(X0)

- (X0, X1 … Xi … Xk)∈P
  ⇒ FOLLOW(Xi) ⊇ FIRST(Xi+1 … Xk)

Example

```
p0: Start = Exp EOF
p1: Exp   = Term Exp'
p2: Exp'  = "+" Term Exp'
p3: Exp'  =
p4: Term  = Fact Term'
p5: Term' = "*" Fact Term'
p6: Term' =
p7: Fact  = Num
p8: Fact  = "(" Exp ")"
```

	nullable	FIRST	FOLLOW
Start			
Exp			
Exp'			
Term			
Term'			
Fact			

Example: Nullable

```
p0: Start = Exp EOF
p1: Exp   = Term Exp'
p2: Exp'  = "+" Term Exp'
p3: Exp'  =
p4: Term  = Fact Term'
p5: Term' = "*" Fact Term'
p6: Term' =
p7: Fact  = Num
p8: Fact  = "(" Exp ")"
```

	nullable	FIRST	FOLLOW
Start			
Exp			
Exp'			
Term			
Term'			
Fact			

$(X, \varepsilon) \in P \Rightarrow \text{nullable}(X)$

$(X_0, X_1 \ldots X_k) \in P \wedge$

$\text{nullable}(X_1) \wedge \ldots \wedge \text{nullable}(X_k) \Rightarrow \text{nullable}(X_0)$

Example: Nullable

```
p0: Start = Exp EOF
p1: Exp   = Term Exp'
p2: Exp'  = "+" Term Exp'
p3: Exp'  =
p4: Term  = Fact Term'
p5: Term' = "*" Fact Term'
p6: Term' =
p7: Fact  = Num
p8: Fact  = "(" Exp ")"
```

	nullable	FIRST	FOLLOW
Start	no		
Exp	no		
Exp'	yes		
Term	no		
Term'	yes		
Fact	no		

$(X, \varepsilon) \in P \Rightarrow \text{nullable}(X)$

$(X_0, X_1 \ldots X_k) \in P \wedge$

$\text{nullable}(X_1) \wedge \ldots \wedge \text{nullable}(X_k) \Rightarrow \text{nullable}(X_0)$

Example: FIRST

```
p0: Start = Exp EOF
p1: Exp   = Term Exp'
p2: Exp'  = "+" Term Exp'
p3: Exp'  =
p4: Term  = Fact Term'
p5: Term' = "*" Fact Term'
p6: Term' =
p7: Fact  = Num
p8: Fact  = "(" Exp ")"
```

	nullable	FIRST	FOLLOW
Start	no		
Exp	no		
Exp'	yes		
Term	no		
Term'	yes		
Fact	no		

$(X_0, X_1 \dots X_i \dots X_k) \in P \wedge$

$\text{nullable}(X_1 \dots X_i) \Rightarrow \text{FIRST}(X_0) \supseteq \text{FIRST}(X_{i+1})$

Example: FIRST

```
p0: Start = Exp EOF
p1: Exp   = Term Exp'
p2: Exp'  = "+" Term Exp'
p3: Exp'  =
p4: Term  = Fact Term'
p5: Term' = "*" Fact Term'
p6: Term' =
p7: Fact  = Num
p8: Fact  = "(" Exp ")"
```

	nullable	FIRST	FOLLOW
Start	no	Num (	
Exp	no	Num (	
Exp'	yes	+	
Term	no	Num (	
Term'	yes	*	
Fact	no	Num (	

$(X_0, X_1 \dots X_i \dots X_k) \in P \wedge$

$\text{nullable}(X_1 \dots X_i) \Rightarrow \text{FIRST}(X_0) \supseteq \text{FIRST}(X_{i+1})$

Example: FOLLOW

```
p0: Start = Exp EOF
p1: Exp   = Term Exp'
p2: Exp'  = "+" Term Exp'
p3: Exp'  =
p4: Term  = Fact Term'
p5: Term' = "*" Fact Term'
p6: Term' =
p7: Fact  = Num
p8: Fact  = "(" Exp ")"
```

	nullable	FIRST	FOLLOW
Start	no	Num (	
Exp	no	Num (	
Exp'	yes	+	
Term	no	Num (	
Term'	yes	*	
Fact	no	Num (	

$(X_0, X_1 \dots X_i \dots X_k) \in P \wedge$

$\text{nullable}(X_{i+1} \dots X_k) \Rightarrow \text{FOLLOW}(X_i) \supseteq \text{FOLLOW}(X_0)$

$(X_0, X_1 \dots X_i \dots X_k) \in P \Rightarrow \text{FOLLOW}(X_i) \supseteq \text{FIRST}(X_{i+1} \dots X_k)$

Example: FOLLOW

```
p0: Start = Exp EOF
p1: Exp   = Term Exp'
p2: Exp'  = "+" Term Exp'
p3: Exp'  =
p4: Term  = Fact Term'
p5: Term' = "*" Fact Term'
p6: Term' =
p7: Fact  = Num
p8: Fact  = "(" Exp ")"
```

	nullable	FIRST	FOLLOW
Start	no	Num (	
Exp	no	Num (	) EOF
Exp'	yes	+	) EOF
Term	no	Num (	+) EOF
Term'	yes	*	+) EOF
Fact	no	Num (	* +) EOF

$(X_0, X_1 \dots X_i \dots X_k) \in P \wedge$

$\text{nullable}(X_{i+1} \dots X_k) \Rightarrow \text{FOLLOW}(X_i) \supseteq \text{FOLLOW}(X_0)$

$(X_0, X_1 \dots X_i \dots X_k) \in P \Rightarrow \text{FOLLOW}(X_i) \supseteq \text{FIRST}(X_{i+1} \dots X_k)$

Example: LL Parse Table

```
p0: Start = Exp EOF
p1: Exp   = Term Exp'
p2: Exp'  = "+" Term Exp'
p3: Exp'  =
p4: Term  = Fact Term'
p5: Term' = "*" Fact Term'
p6: Term' =
p7: Fact  = Num
p8: Fact  = "(" Exp ")"
```

	+	*	Num	(	)	EOF
Start						
Exp						
Exp'						
Term						
Term'						
Fact						

entry $(X, w) \in P$ at row X and column T

$T \in \text{FIRST}(w)$

$\text{nullable}(w) \wedge T \in \text{FOLLOW}(X)$

Example: LL Parse Table

```
p0: Start = Exp EOF
p1: Exp   = Term Exp'
p2: Exp'  = "+" Term Exp'
p3: Exp'  =
p4: Term  = Fact Term'
p5: Term' = "*" Fact Term'
p6: Term' =
p7: Fact  = Num
p8: Fact  = "(" Exp ")"
```

	+	*	Num	(	)	EOF
Start			p0	p0		
Exp			p1	p1		
Exp'	p2				p3	p3
Term			p4	p4		
Term'	p6	p5			p6	p6
Fact			p7	p8		

entry $(X, w) \in P$ at row X and column T

$T \in \text{FIRST}(w)$

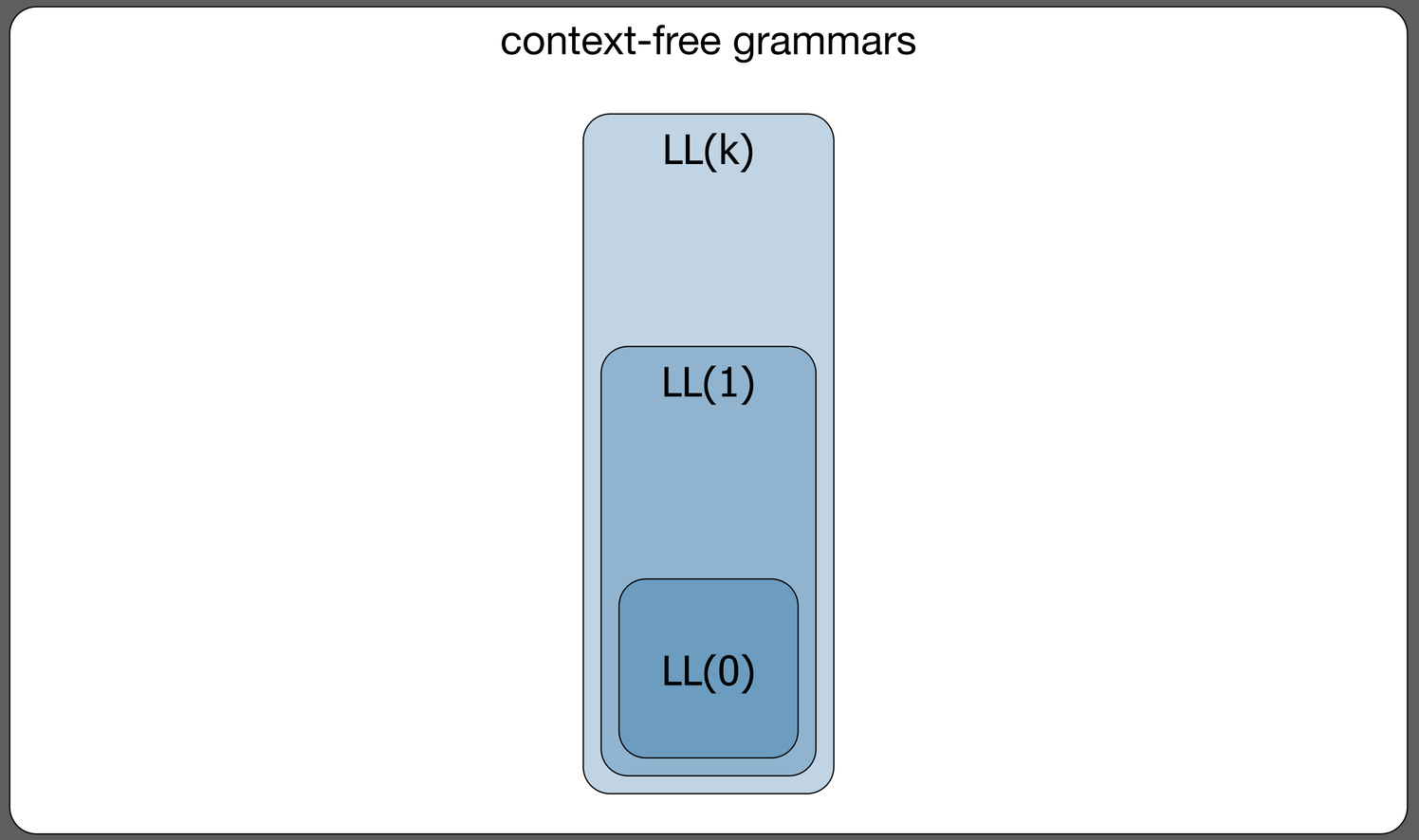
$\text{nullable}(w) \wedge T \in \text{FOLLOW}(X)$

Example: Parsing

```
p0: Start = Exp EOF
p1: Exp   = Term Exp'
p2: Exp'  = "+" Term Exp'
p3: Exp'  =
p4: Term  = Fact Term'
p5: Term' = "*" Fact Term'
p6: Term' =
p7: Fact  = Num
p8: Fact  = "(" Exp ")"
```

	+	*	Num	(	)	EOF
Start			p0	p0		
Exp			p1	p1		
Exp'	p2				p3	p3
Term			p4	p4		
Term'	p6	p5			p6	p6
Fact			p7	p8		

Grammar Classes



Given the next n tokens, predict next production

Predictive Parsing: Encoding Precedence

```
Exp = Num
Exp = “(” Exp “)”
Exp = Exp “*” Exp
Exp = Exp “+” Exp
```

```
Fact = Num
Fact = “(” Exp “)”
Term = Term “*” Fact
Term = Fact
Exp  = Exp “+” Term
Exp  = Term
```

Predictive Parsing: Eliminating Left Recursion

```
Term = Term “*” Fact
Term = Fact
Exp  = Exp “+” Term
Exp  = Term
```

```
Term’ = “*” Fact Term’
Term’ =
Term  = Fact Term’
Exp’  = “+” Term Exp’
Exp’  =
```

Predictive Parsing: Left Factoring

```
Exp = “if” Exp “then” Exp “else” Exp
Exp = “if” Exp “then” Exp
```

```
Exp  = “if” Exp “then” Exp Else
Else = “else” Exp
Else =
```

Summary

Summary

How can we parse context-free languages effectively?

- predictive parsing algorithms

Which grammar classes are supported by these algorithms?

- LL(k) grammars, LL(k) languages

How can we generate compiler tools from that?

- implement automaton
- generate parse tables

What are other techniques for implementing top-down parsers?

- Parser Combinators
- PEGs
- ALL(*)

Literature

Formal languages

- Noam Chomsky: Three models for the description of language. 1956
- J. E. Hopcroft, R. Motwani, J. D. Ullman: Introduction to Automata Theory, Languages, and Computation. 2006

Syntactic analysis

- Andrew W. Appel, Jens Palsberg: Modern Compiler Implementation in Java, 2nd edition. 2002
- Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Monica S. Lam: Compilers: Principles, Techniques, and Tools, 2nd edition. 2006

Literature

ALL(*)

- Terence John Parr, Sam Harwell, Kathleen Fisher. Adaptive LL(*) parsing: the power of dynamic analysis. In OOPSLA 2014.

Parsing Expression Grammars

- Bryan Ford. Parsing Expression Grammars: a recognition-based syntactic foundation. In POPL 2004.

Parser Combinators

- Graham Hutton. Higher-Order Functions for Parsing. Journal of Functional Programming, 1992.
- A. Moors, F. Piessens, Martin Odersky. Parser combinators in Scala. Technical Report Department of Computer Science, K.U. Leuven, February 2008.

Generalized LR Parsing

Generalized Parsing

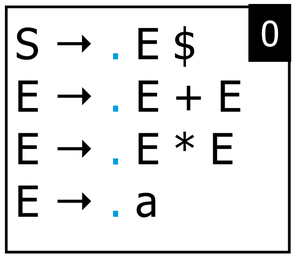
- Parse all interpretations of the input => handle ambiguous grammars
- Parsers split whenever finding an ambiguous interpretation and act in (pseudo) parallel
- Multiple parsers can join whenever they finish parsing an ambiguous fragment of the input
- Some parsers may "die", if the ambiguity was caused by a lack of lookahead

Generalized LR

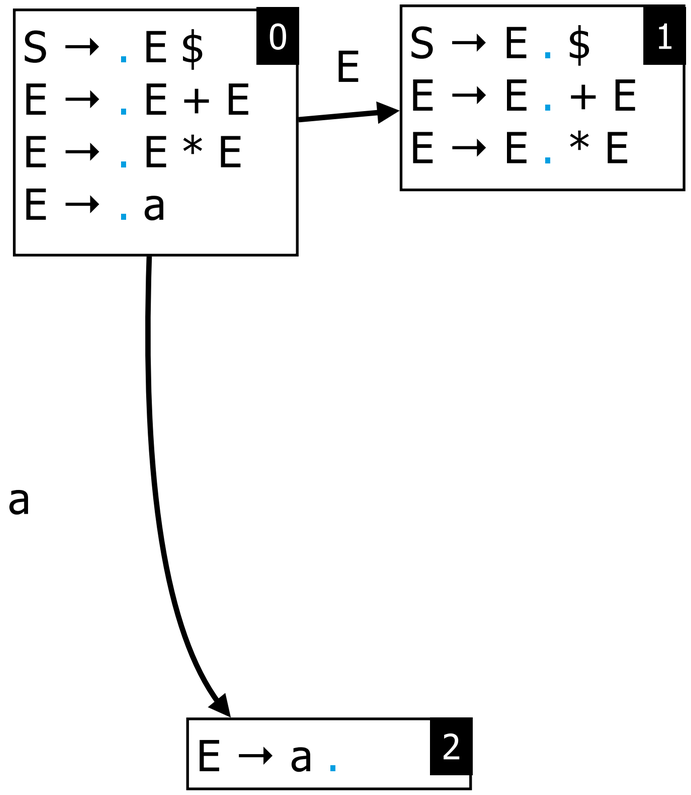
- Multiple parsers are synchronized on shift actions
- Each parser has its own stack, and as they share states, the overall structure becomes a graph (GSS)
- If two parsers have the same state on top of their stack, they are joined into a single parser
- Reduce actions affect all possible paths from the top of the stacks

```
S = E $
E = E + E
E = E * E
E = a
```

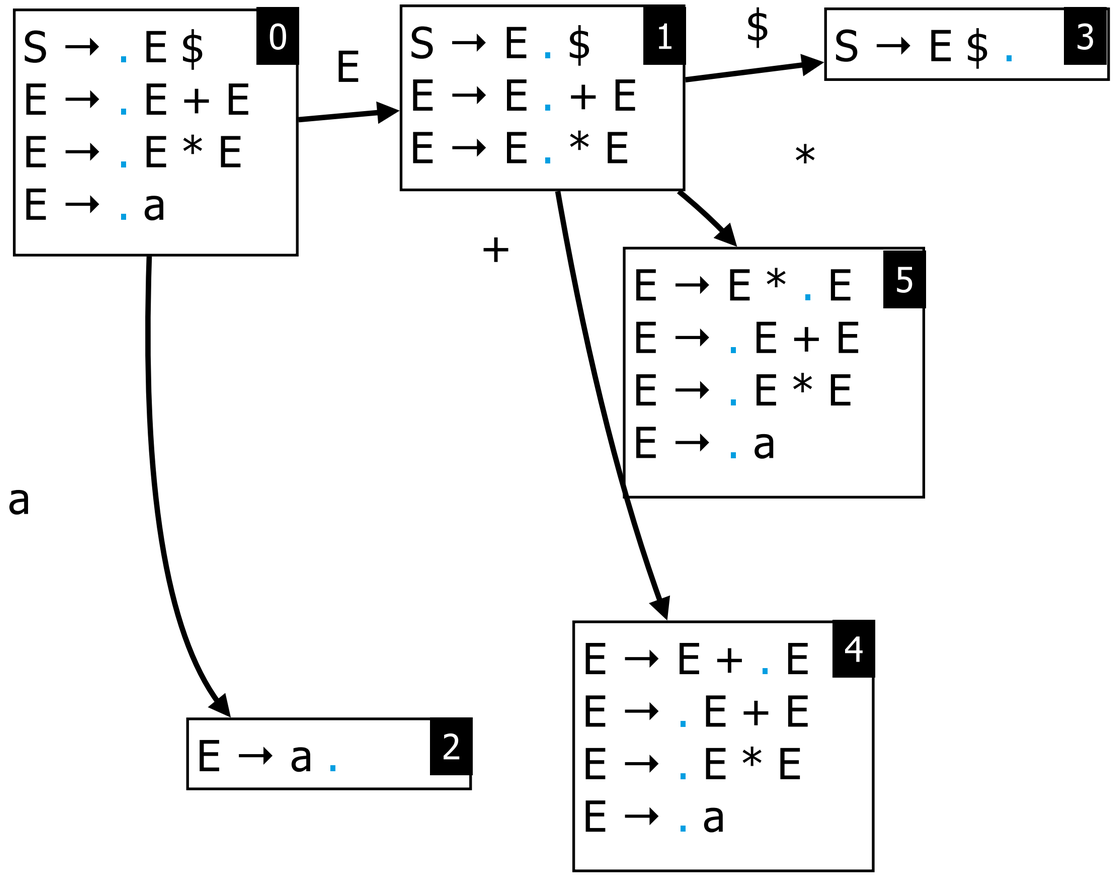
```
S = E $
E = E + E
E = E * E
E = a
```



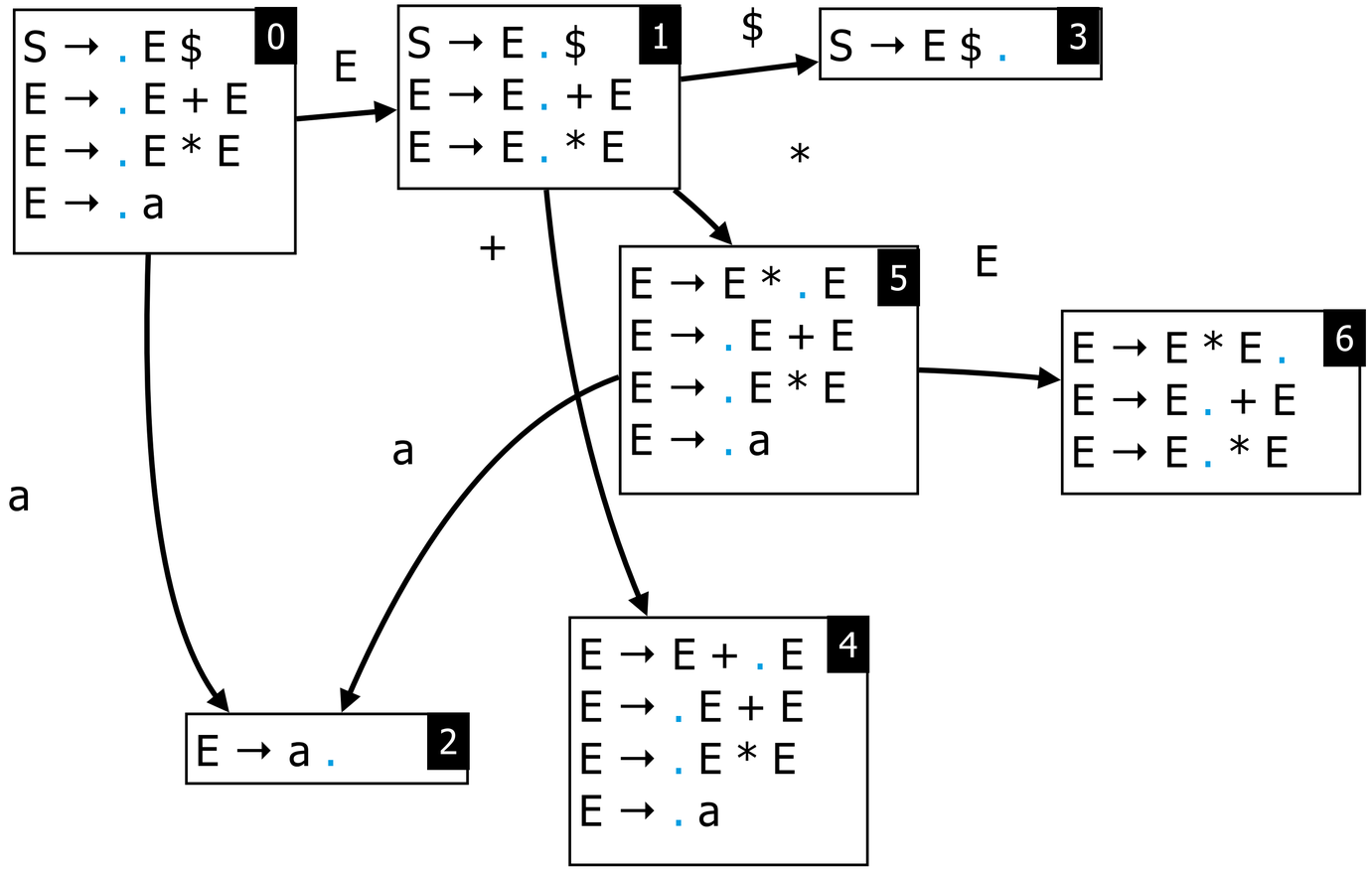
S = E $
E = E + E
E = E * E
E = a



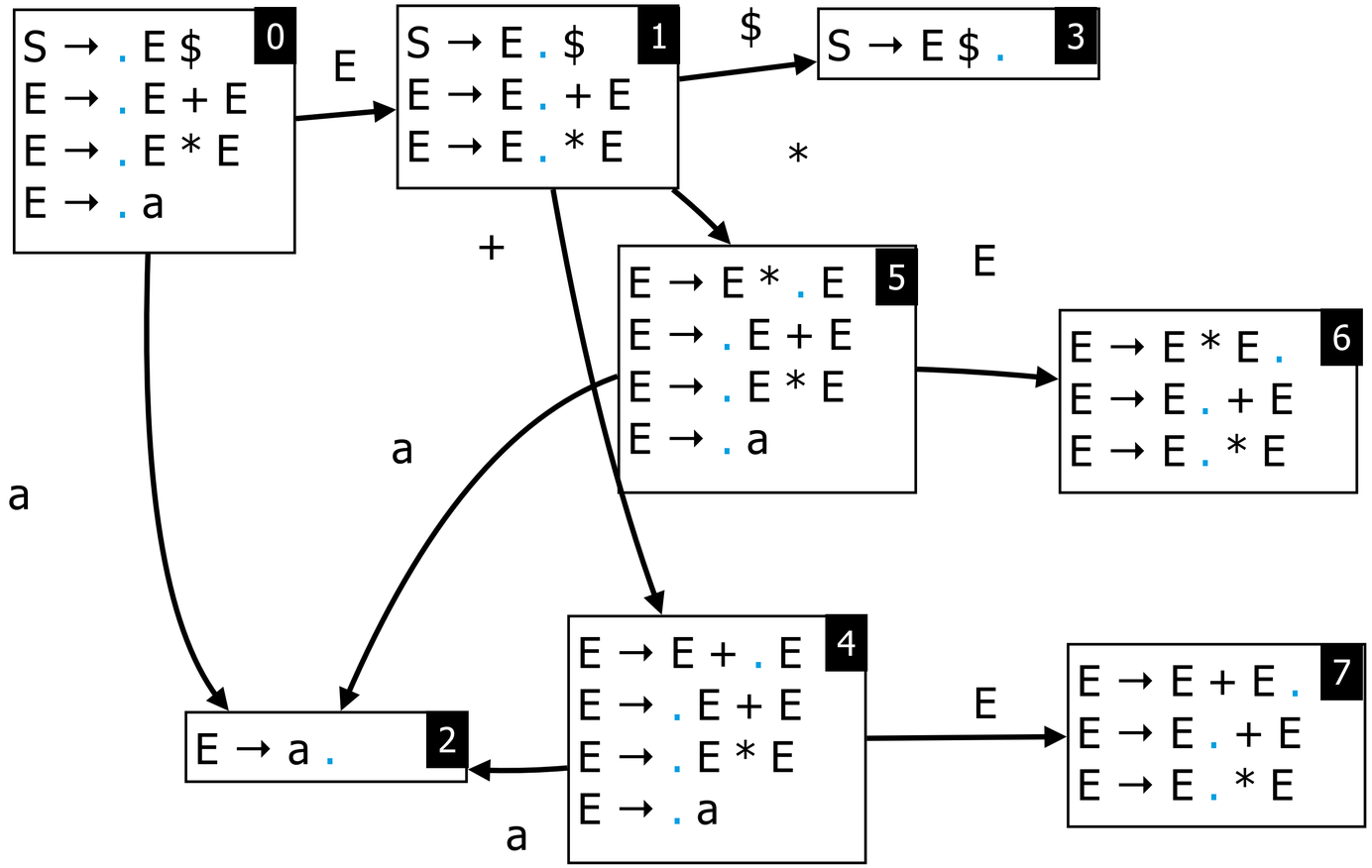
S = E $
E = E + E
E = E * E
E = a


S → . E $
E → . E + E
E → . E * E
E → . a
0
E
S → E . $
E → E . + E
E → E . * E
1
$
S → E $.
3
*
+
E → E * . E
E → . E + E
E → . E * E
E → . a
5
a
E → a .
2
E → E + . E
E → . E + E
E → . E * E
E → . a
4

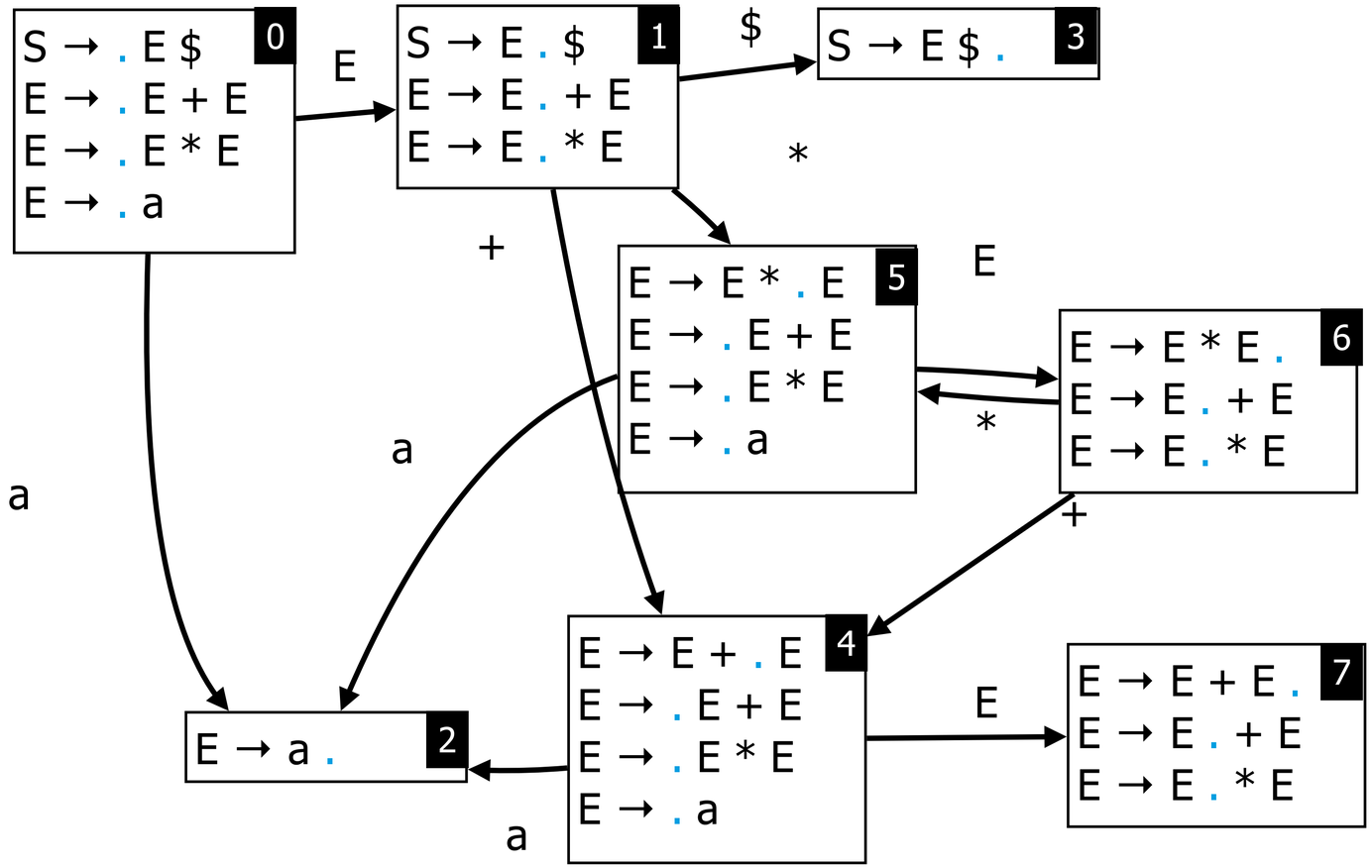
S = E $
E = E + E
E = E * E
E = a


S → . E $
E → . E + E
E → . E * E
E → . a
0
E
S → E . $
E → E . + E
E → E . * E
1
$
S → E $.
3
*
+
E → E * . E
E → . E + E
E → . E * E
E → . a
5
E
E → E * E .
E → E . + E
E → E . * E
6
a
a
E → a .
2
E → E + . E
E → . E + E
E → . E * E
E → . a
4

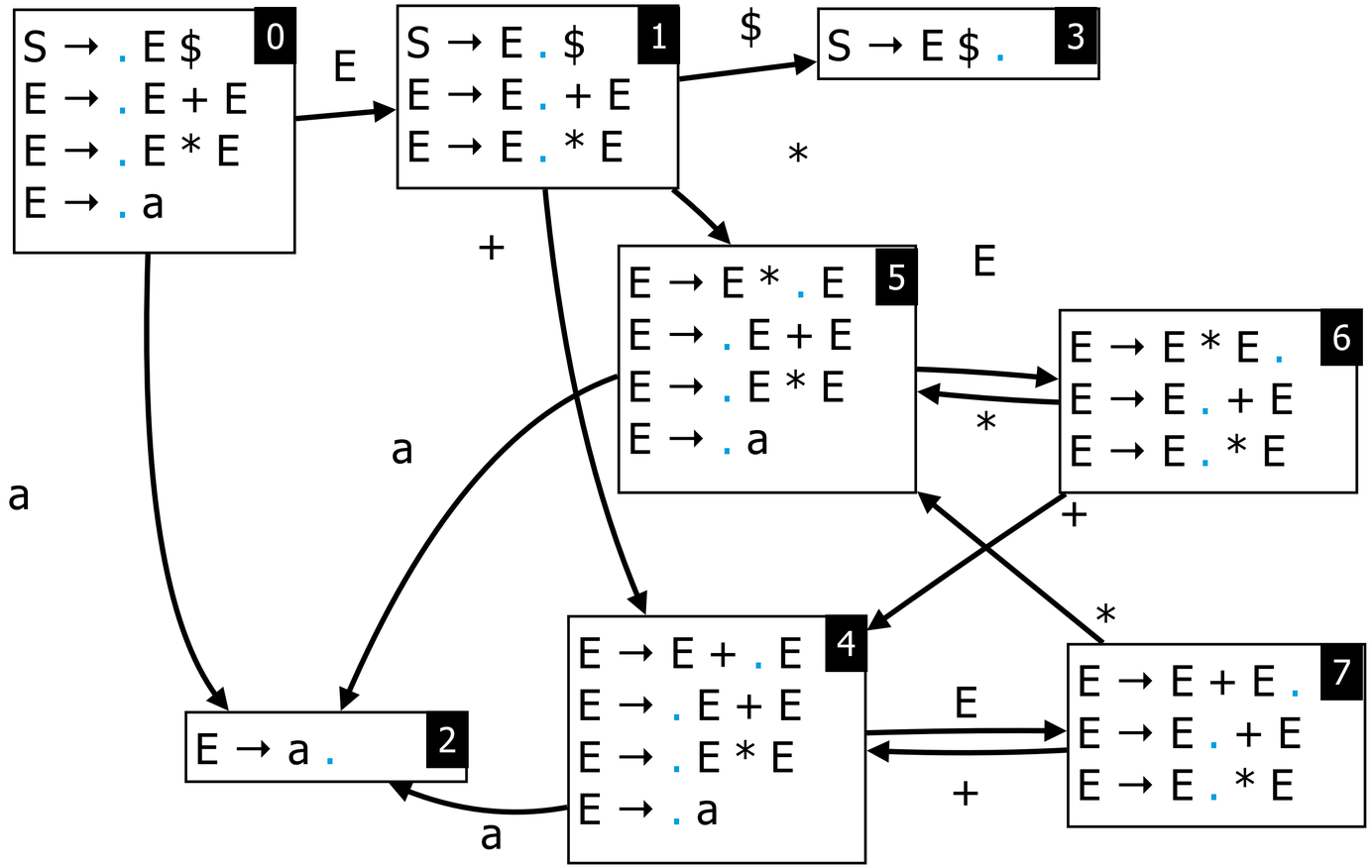
S = E $
E = E + E
E = E * E
E = a


S → . E $
E → . E + E
E → . E * E
E → . a
0
E
S → E . $
E → E . + E
E → E . * E
1
$
S → E $.
3
*
+
E → E * . E
E → . E + E
E → . E * E
E → . a
5
E
E → E * E .
E → E . + E
E → E . * E
6
a
a
E → a .
2
E → E + . E
E → . E + E
E → . E * E
E → . a
4
a
E
E → E + E .
E → E . + E
E → E . * E
7

S = E $
E = E + E
E = E * E
E = a


S → . E $
E → . E + E
E → . E * E
E → . a
0
E
S → E . $
E → E . + E
E → E . * E
1
$
S → E $.
3
*
+
E → E * . E
E → . E + E
E → . E * E
E → . a
5
E
E → E * E .
E → E . + E
E → E . * E
6
*
a
a
+
E → E + . E
E → . E + E
E → . E * E
E → . a
4
E → a .
2
a
E
E → E + E .
E → E . + E
E → E . * E
7

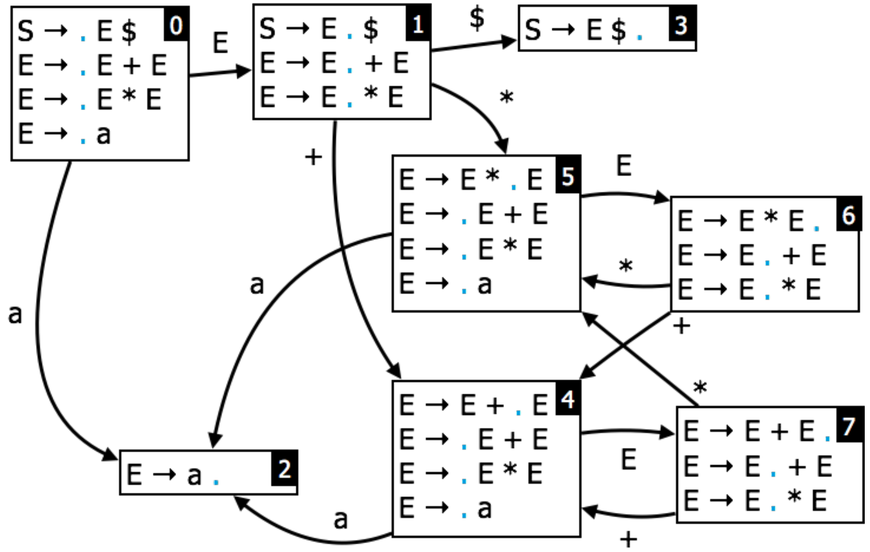
S = E $
E = E + E
E = E * E
E = a


S → . E $
E → . E + E
E → . E * E
E → . a
0
E
S → E . $
E → E . + E
E → E . * E
1
$
S → E $.
3
*
+
E → E * . E
E → . E + E
E → . E * E
E → . a
5
E
E → E * E .
E → E . + E
E → E . * E
6
*
a
a
+
*
E → E + . E
E → . E + E
E → . E * E
E → . a
4
E → a .
2
E
E → E + E .
E → E . + E
E → E . * E
7
+
a

SLR Table

Nonter	Nullable	First	Follow
S			
E			

State	Action				Goto	
	a	+	*	$	S	E
0						
1						
2						
3						
4						
5						
6						
7						

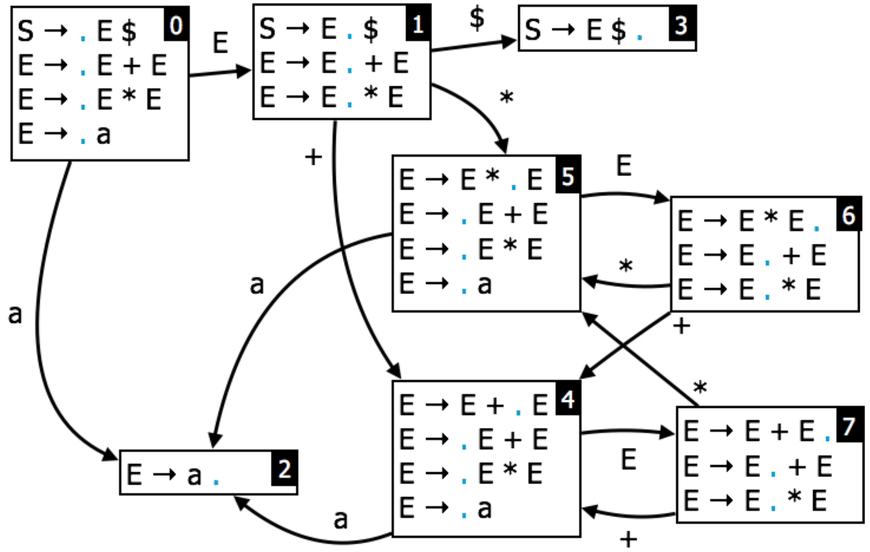



```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

SLR Table

Nonter	Nullable	First	Follow
S	no	a	-
E	no	a	+, *, $

State	Action				Goto	
	a	+	*	$	S	E
0						
1						
2						
3						
4						
5						
6						
7						

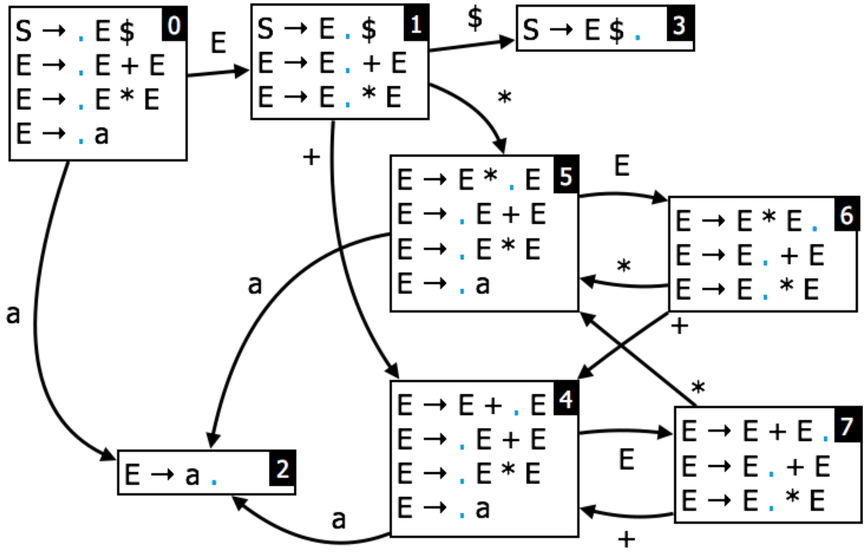



```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

SLR Table

Nonter	Nullable	First	Follow
S	no	a	-
E	no	a	+, *, $

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		



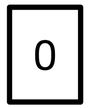

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

Parsing

input: a + a * a $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

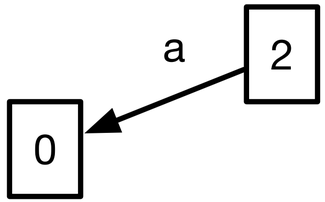


Parsing

input: + a * a $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		



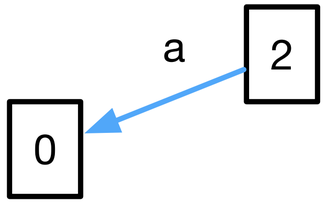
synchronize on shifts

Parsing

input: + a * a $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

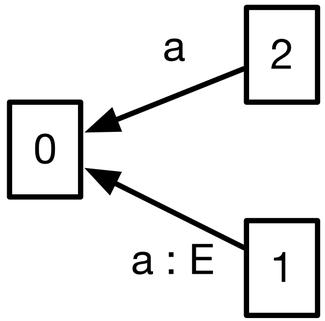


Parsing

input: + a * a $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

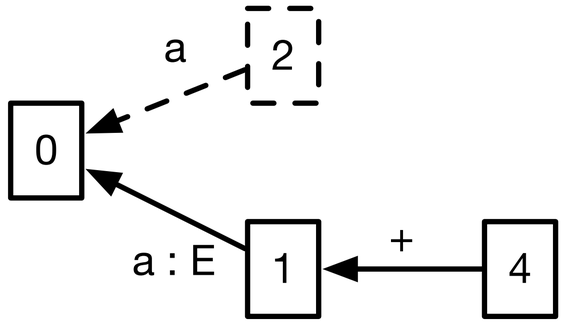


Parsing

input: a * a $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		



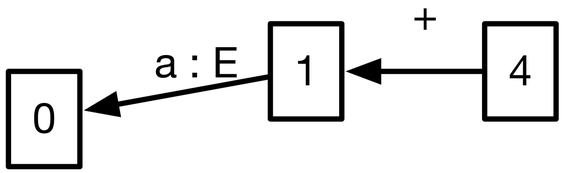
synchronize

Parsing

input: a * a $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		



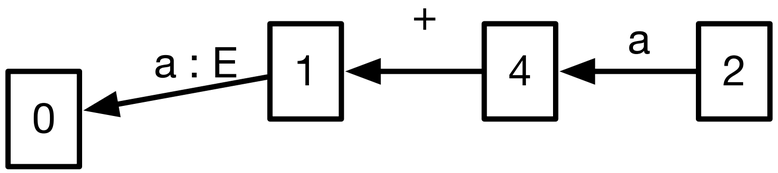
Parsing

input: * a $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

synchronize

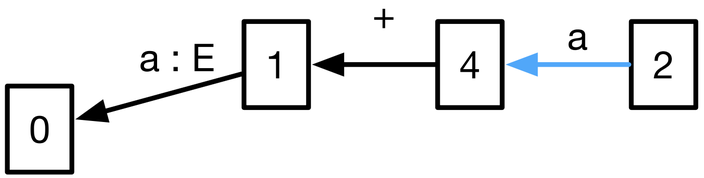


Parsing

input: * a $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

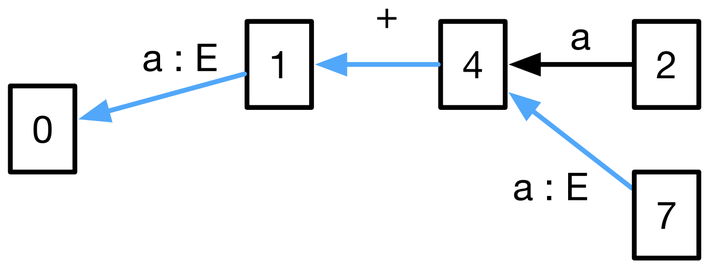


Parsing

input: * a $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

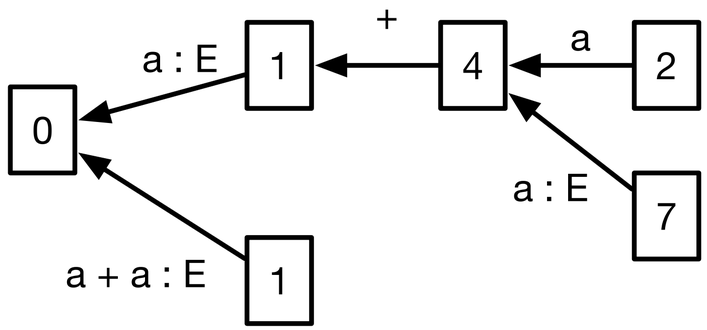


Parsing

input: * a $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		



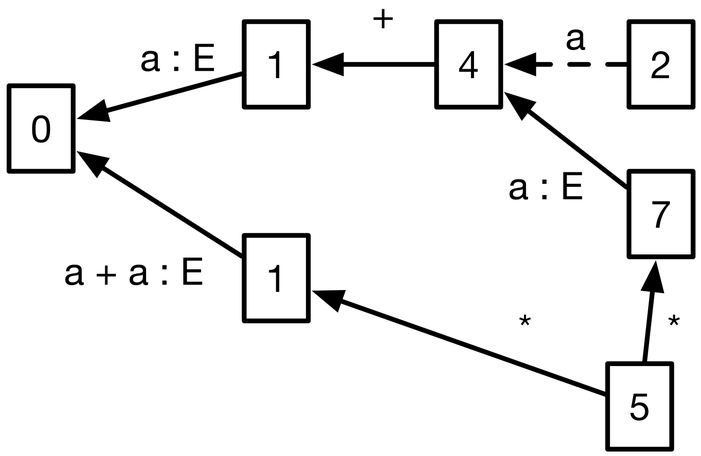
Parsing

input: a $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

synchronize

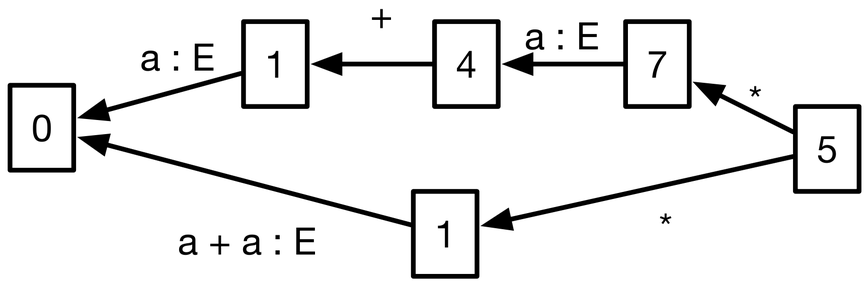


Parsing

input: a $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		



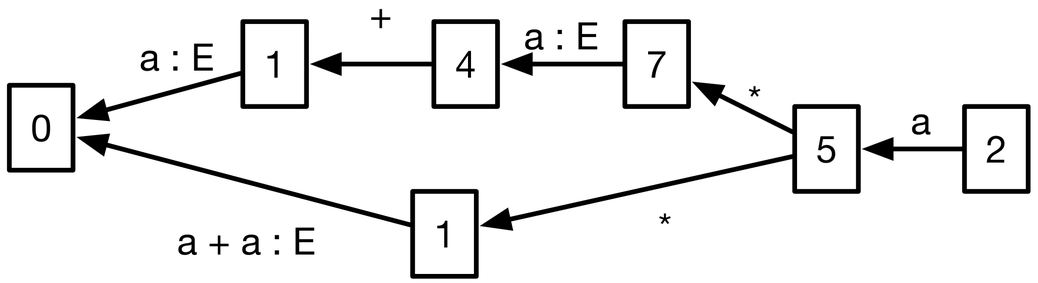
Parsing

input: $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

synchronize

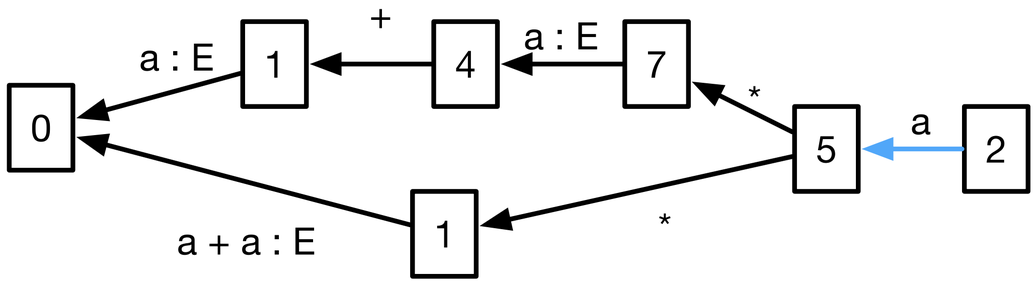


Parsing

input: $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

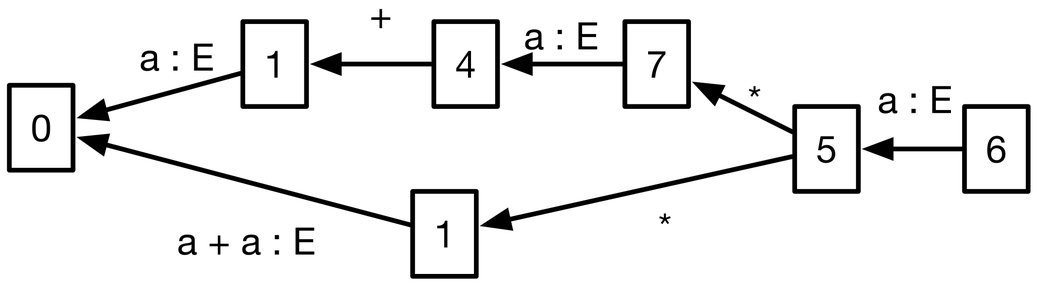


Parsing

input: $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

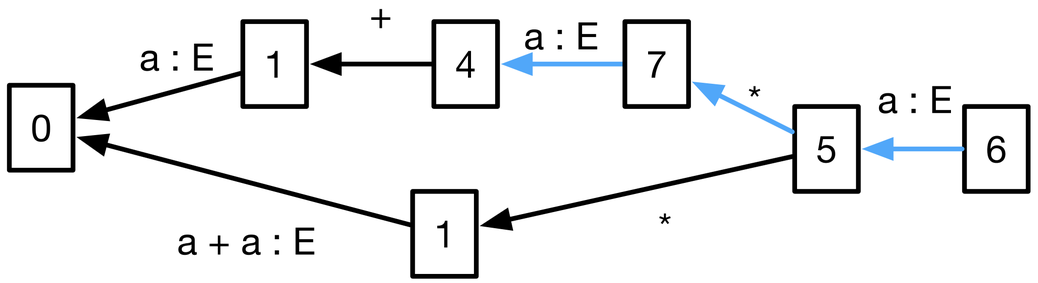


Parsing

input: $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

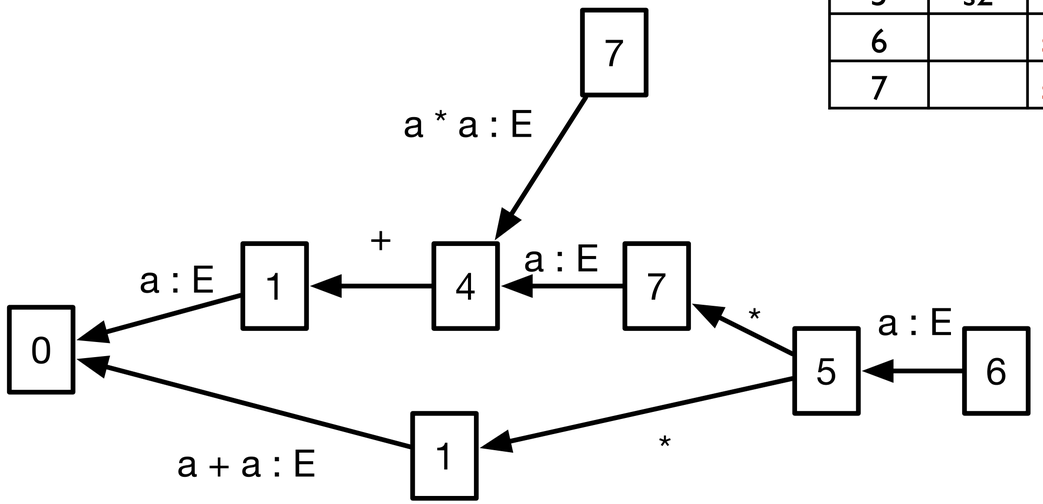


Parsing

input: $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

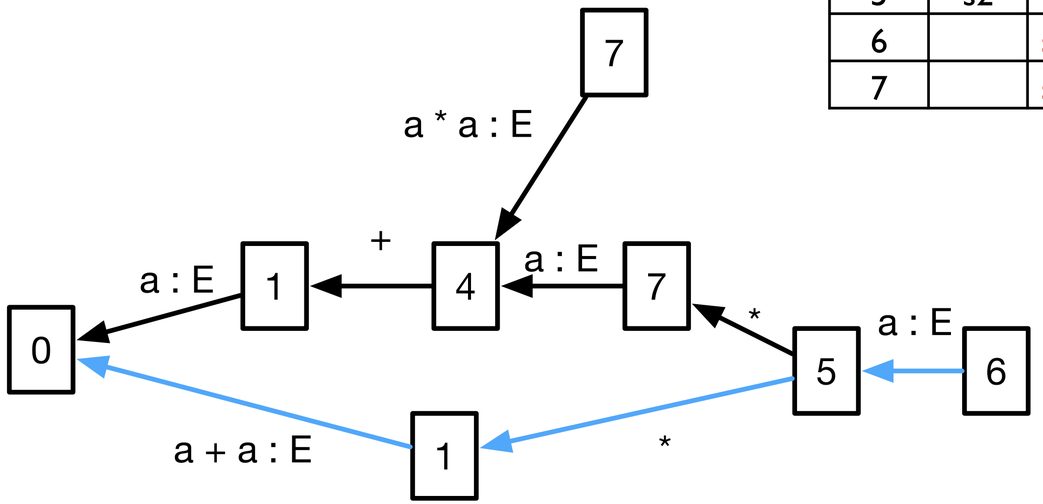


Parsing

input: $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

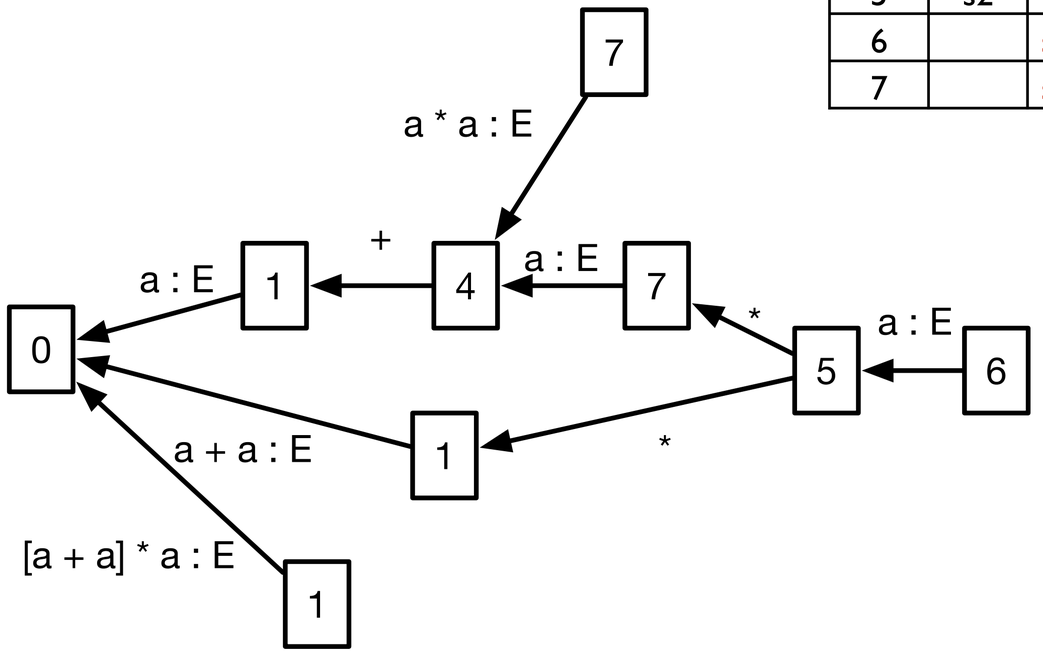


Parsing

input: $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

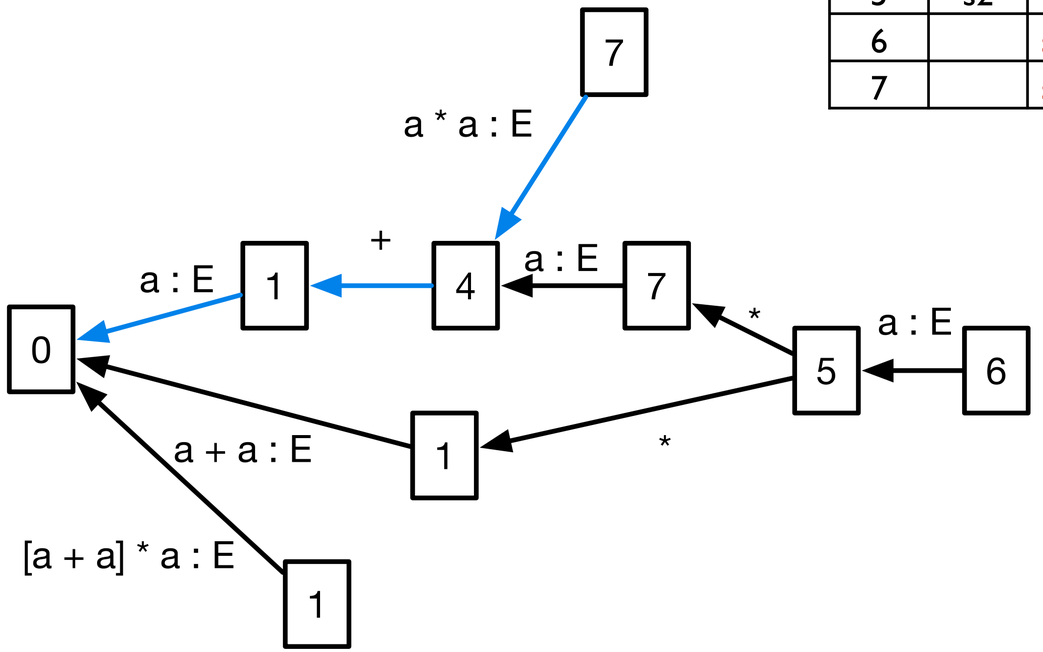


Parsing

input: $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

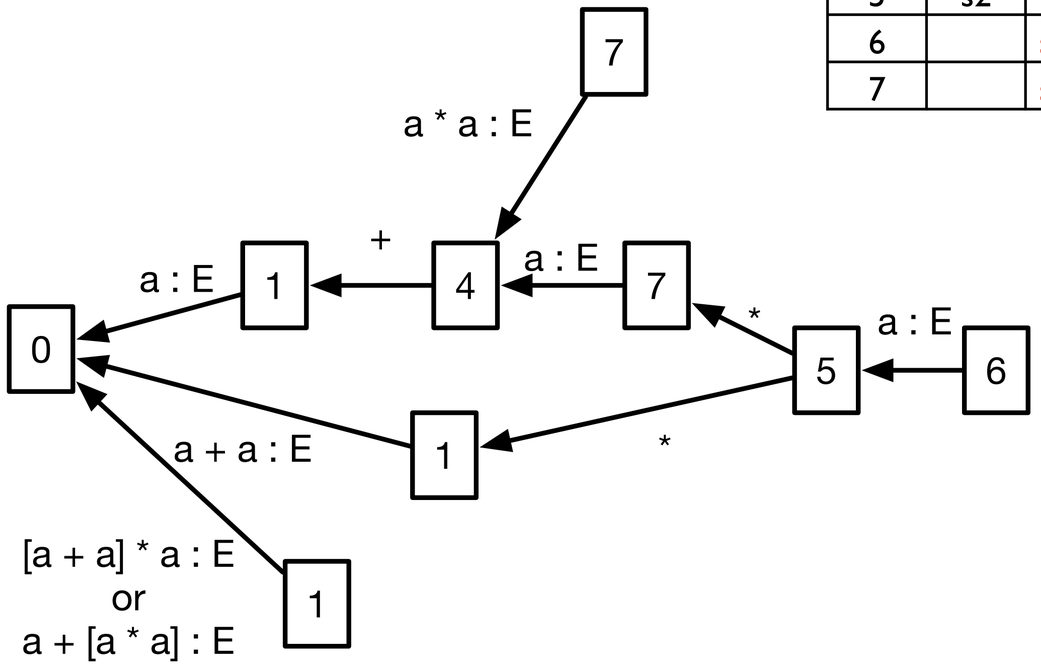


Parsing

input: $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		



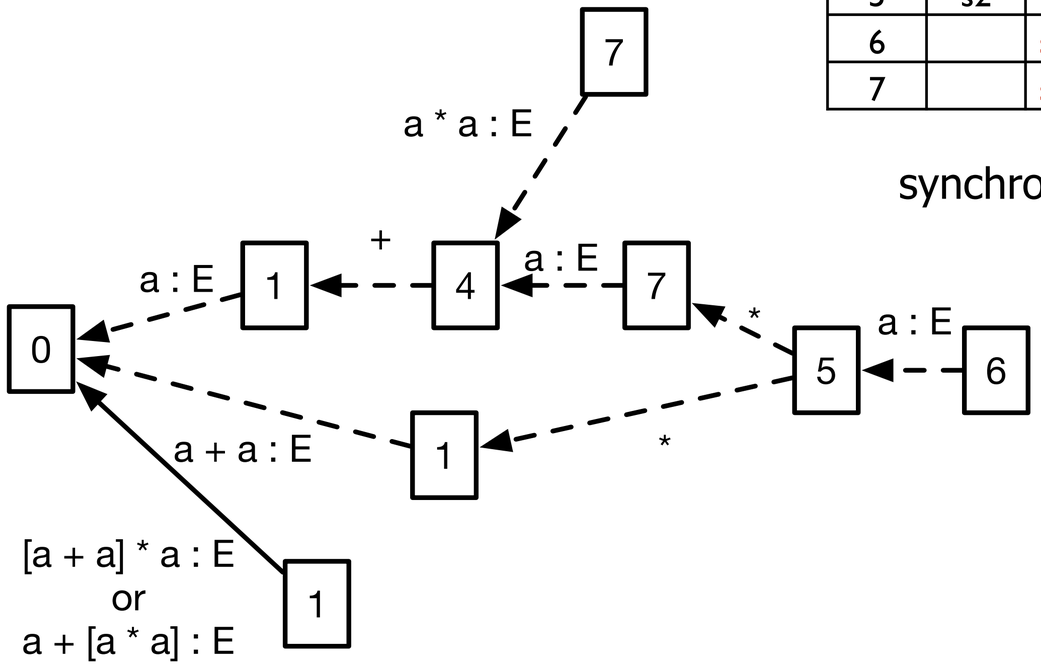
Parsing

input: $

```
(0) S = E $
(1) E = E + E
(2) E = E * E
(3) E = a
```

State	Action				Goto	
	a	+	*	$	S	E
0	s2					1
1		s4	s5	acc		
2		r3	r3	r3		
3	acc	acc	acc	acc		
4	s2					7
5	s2					6
6		s4/r2	s5/r2	r2		
7		s4/r1	s5/r1	r1		

synchronize



accept with trees on the link to initial state