More Parsing Algorithms

Eelco Visser

TU Delft

CS4200 | Compiler Construction | October 15, 2020

This Lecture

This Lecture

Predictive parsing

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

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

A Theory of Language: Formal Languages

Vocabulary Σ

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

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

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

A Theory of Languages: Formal Grammars

Formal grammar $G = (N, \Sigma, P, S)$

- nonterminal symbols N
- terminal symbols Σ
- production rules $P \subseteq (N \cup \Sigma)^* \, N \, (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- start symbol $S \in N$

A Theory of Languages: Formal Grammars

Formal grammar $G = (N, \Sigma, P, S)$

- nonterminal symbols N
- terminal symbols Σ
- production rules $P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$
- start symbol $S \in N$

Grammar classes

- type-0, unrestricted
- type-1, context-sensitive: `(a A c, a b c)`
- type-2, context-free: $P \subseteq N \times (N \cup \Sigma)^*$
- type-3, regular: `(A, x)` or `(A, xB)`

A Theory of Languages: Formal Languages

A Theory of Languages: Formal Languages

Formal grammar

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

A Theory of Languages: Formal Languages

Formal grammar

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

Derivation relation

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

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

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

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

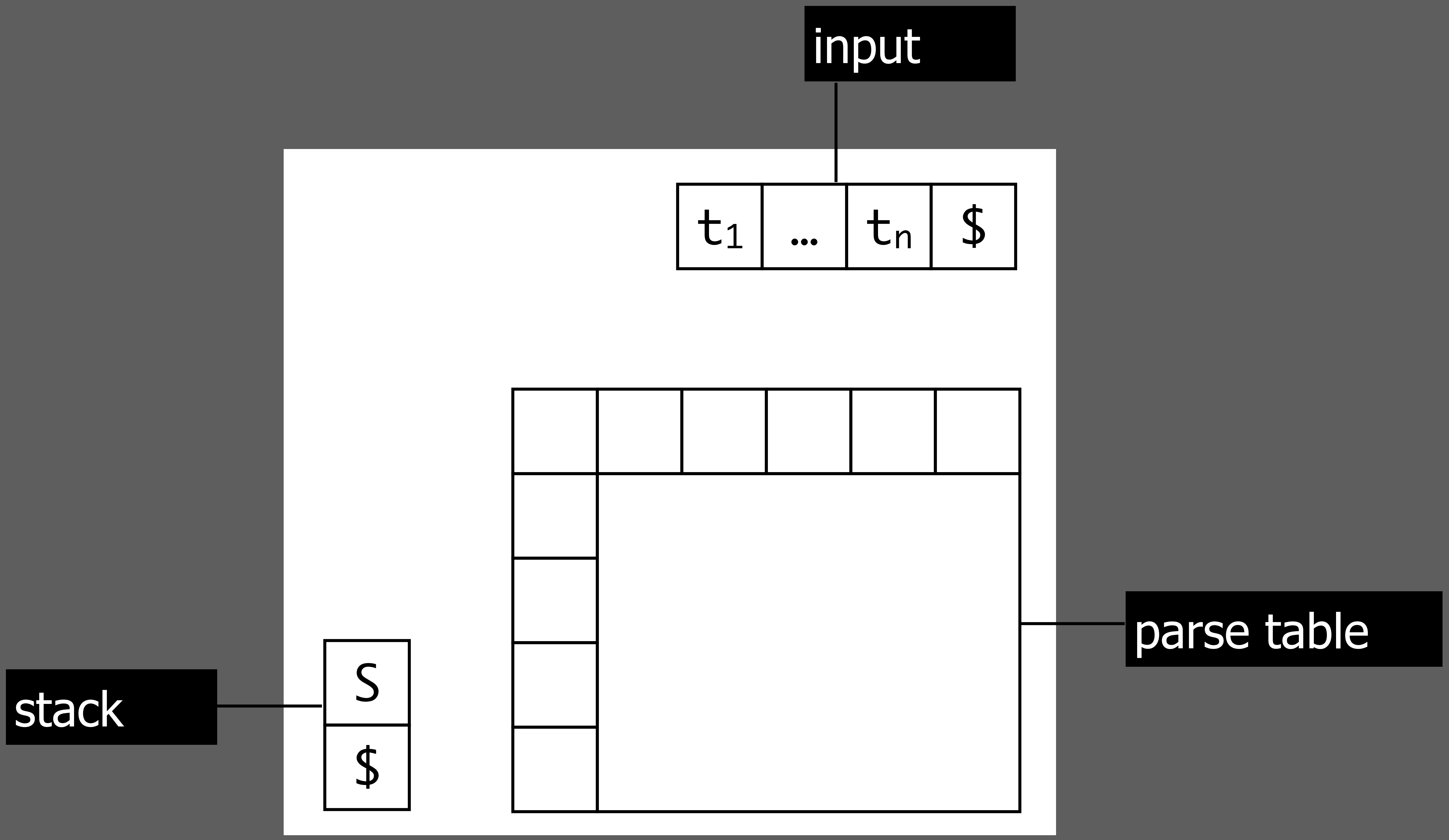
Entries

- production rules P
- possible conflicts

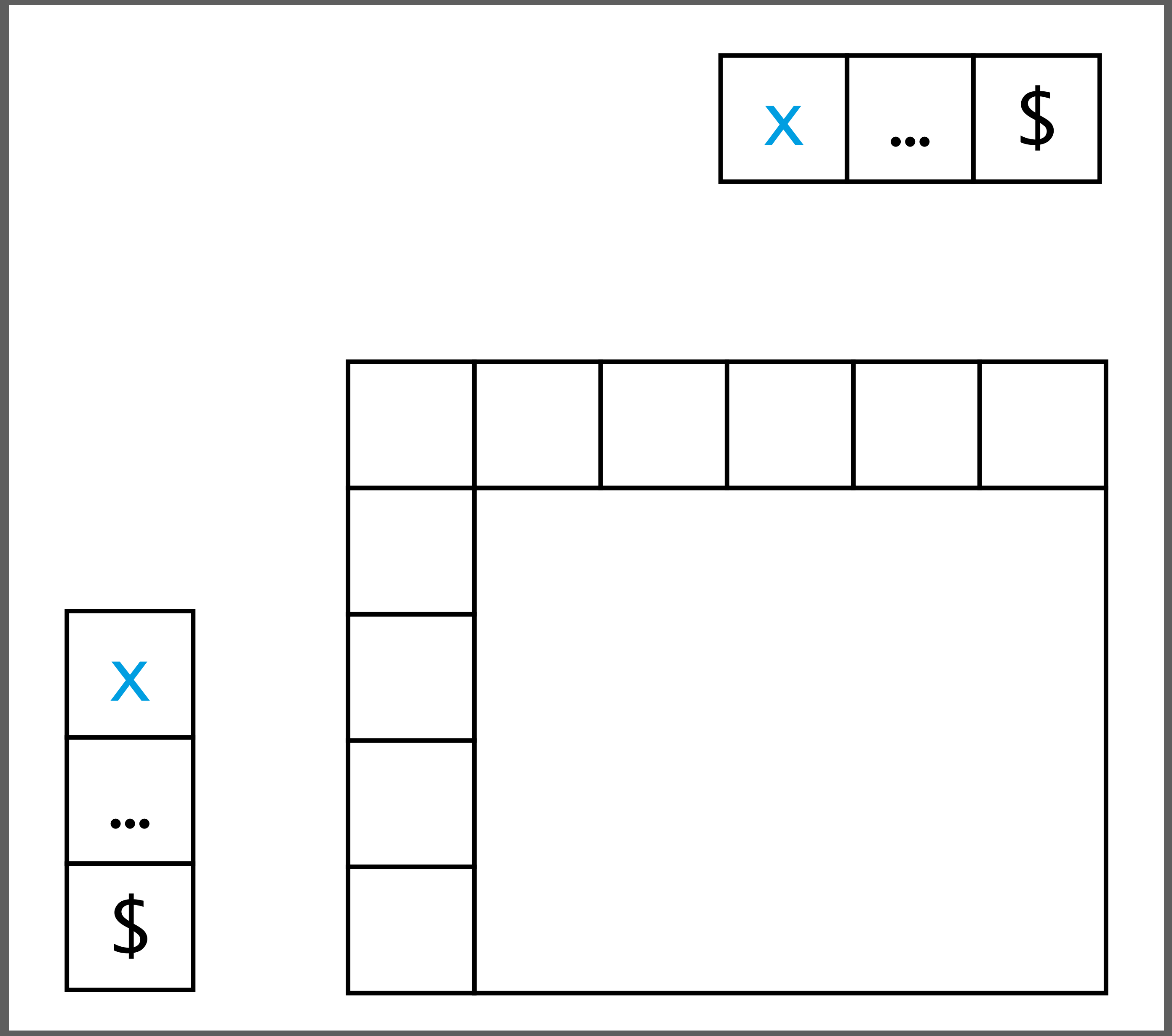
	T_1	T_2	T_3	…
N_1	N_1 →…		N_1 →…	
N_2		N_2 →…		
N_3		N_3 →…	N_3 →…	
N_4	N_4 →…			
N_5		N_5 →…		
N_6	N_6 →…	N_6 →…		
N_7			N_7 →…	
N_8	N_8 →…	N_8 →…	N_8 →…	
…				

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

Predictive Parsing: Automaton

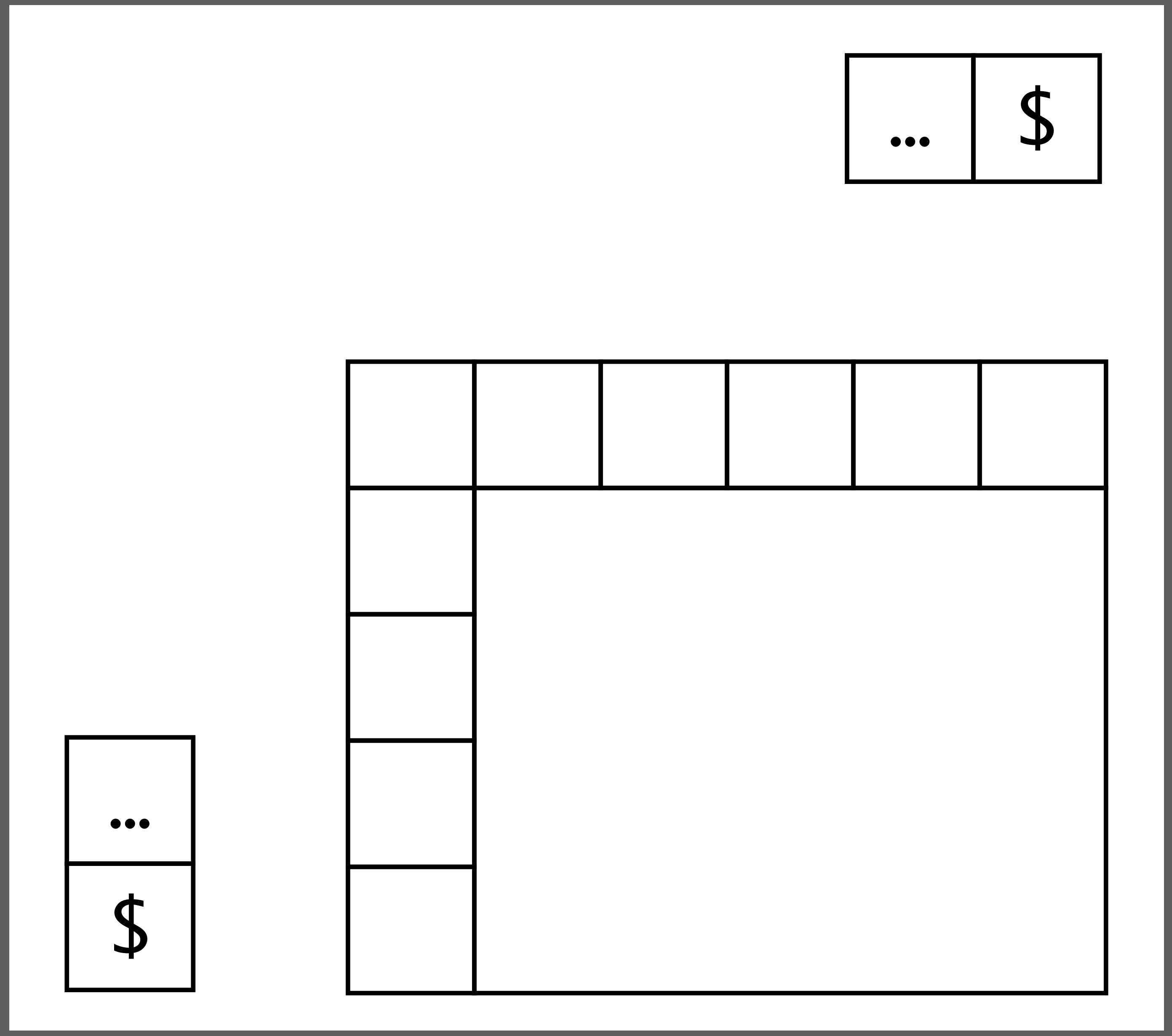


Predictive Parsing: Automaton



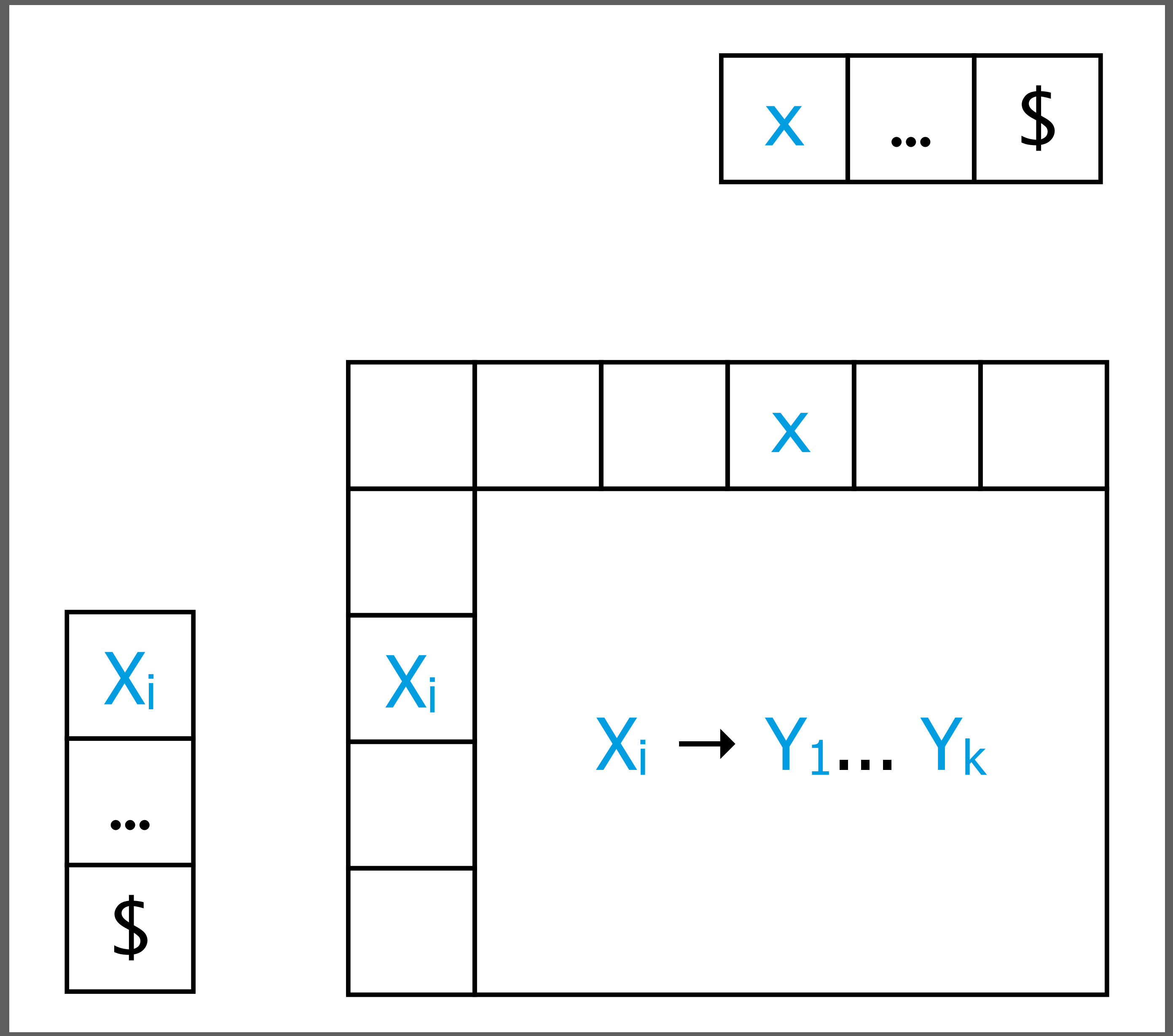
Recognize predicted symbol

Predictive Parsing: Automaton



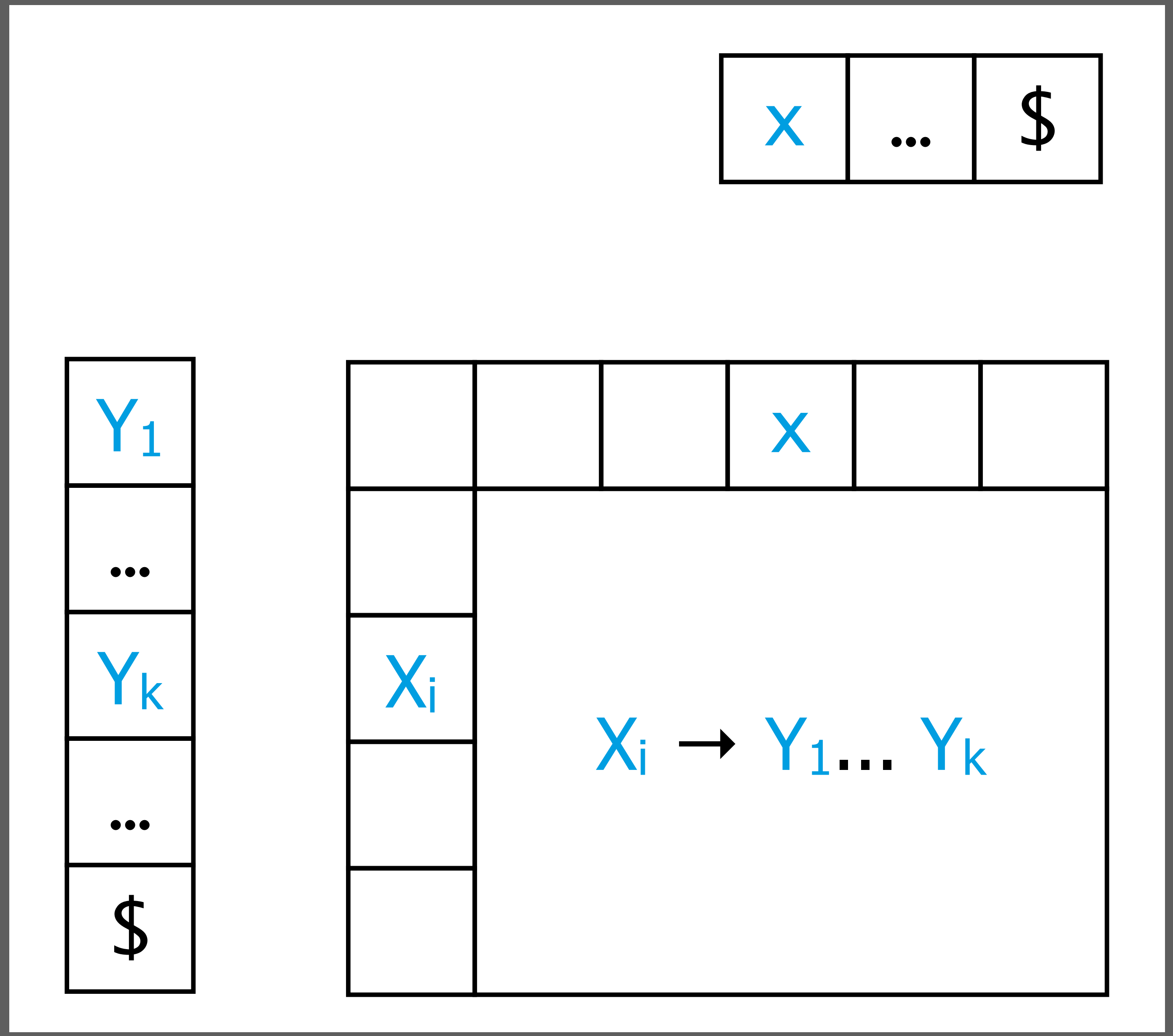
Recognize predicted symbol

Predictive Parsing: Automaton



Predict production

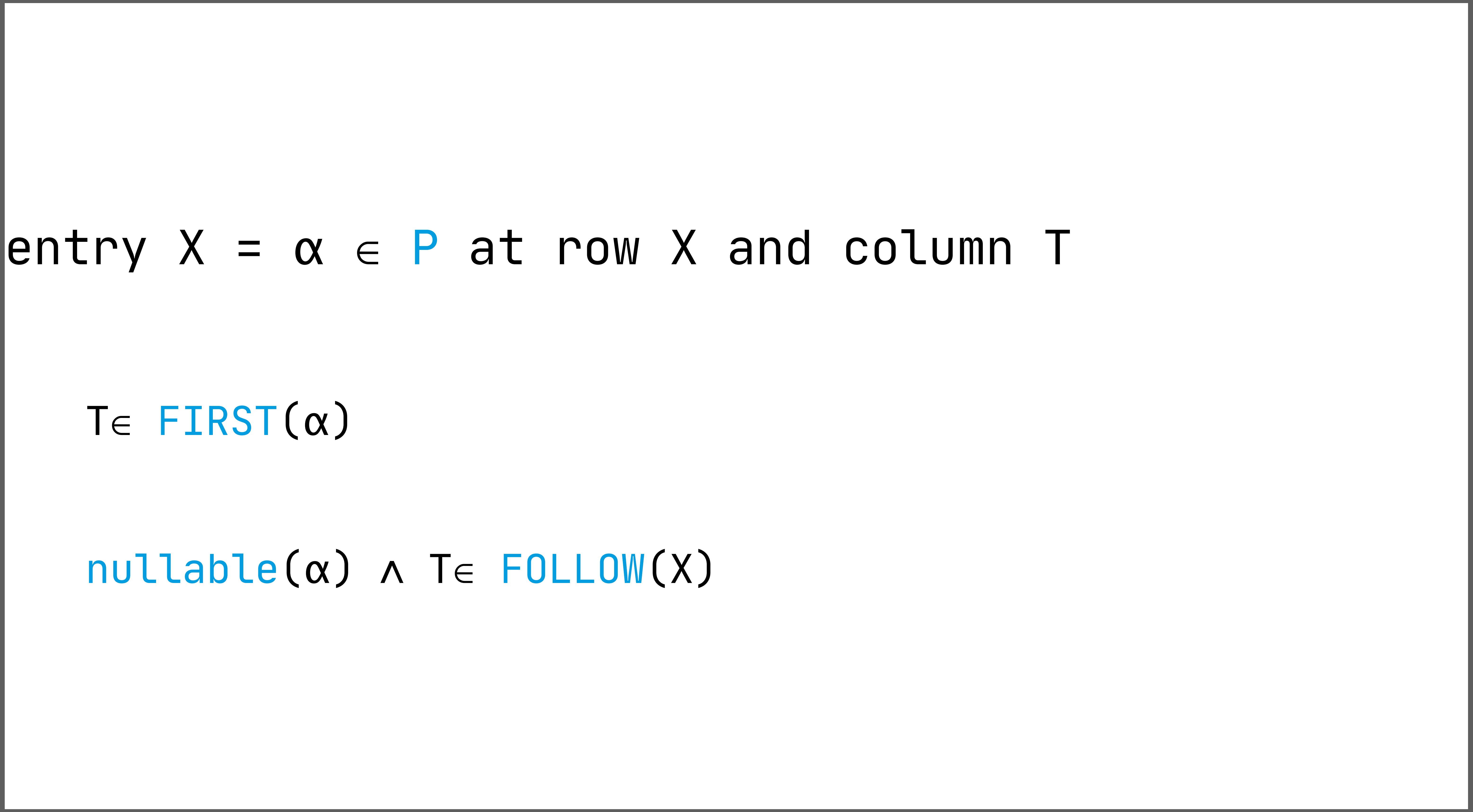
Predictive Parsing: Automaton



Predict production

LL Parse Tables

Predictive Parsing: Filling the Table



```
entry X = α ∈ P at row X and column T

    T∈ FIRST(α)

    nullable(α) ∧ T∈ FOLLOW(X)
```

Predictive Parsing: Filling the Table

entry X = α ∈ P at row X and column T

T∈ $\boxed{\text{FIRST}(\alpha)}$ — letters that ɑ can start with

nullable(α) ∧ T∈ FOLLOW(X)

Predictive Parsing: Filling the Table

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

$T \in$ FIRST(α)

nullable(α) $\wedge$ $T \in$ FOLLOW(X)

$w \Rightarrow_G^* \varepsilon$

Predictive Parsing: Filling the Table

entry X = α ∈ P at row X and column T

T∈ FIRST(α)

nullable(α) ∧ T∈ $\boxed{\mathtt{FOLLOW(X)}}$

letters that can follow X

Predictive Parsing: Nullable

nullable(X)

- $(X, \varepsilon) \in P \Rightarrow \mathrm{nullable}(X)$
- $(X0, X1 \ldots Xk) \in P \wedge \mathrm{nullable}(X1) \wedge \ldots \wedge \mathrm{nullable}(Xk)$
  $\Rightarrow \mathrm{nullable}(X0)$

nullable(α)

- $\mathrm{nullable}(\varepsilon)$
- $\mathrm{nullable}(X1 \ldots Xk) = \mathrm{nullable}(X1) \wedge \ldots \wedge \mathrm{nullable}(Xk)$

Predictive Parsing: First Set

FIRST(X)

- $X \in \Sigma$: $\mathrm{FIRST}(X) = \{X\}$
- $(X0,\ X1 \ldots Xi \ldots Xk) \in P \land \mathrm{nullable}(X1 \ldots Xi) \Rightarrow$ $\mathrm{FIRST}(X0) \supseteq \mathrm{FIRST}(Xi{+}1)$

FIRST(w)

- $\mathrm{FIRST}(\varepsilon) = \{\}$
- $\neg\mathrm{nullable}(X) \Rightarrow \mathrm{FIRST}(Xw) = \mathrm{FIRST}(X)$
- $\mathrm{nullable}(X) \Rightarrow \mathrm{FIRST}(Xw) = \mathrm{FIRST}(X) \cup \mathrm{FIRST}(w)$

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			

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

$(X_0, X_1 \dots X_k) \in P \land$

$\text{nullable}(X_1) \land \dots \land \text{nullable}(X_k) \Rightarrow \text{nullable}(X_0)$

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 “)”
```

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

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

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

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

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		

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

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

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 (	

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

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

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 (	

$(X_0, X_1 \ldots X_i \ldots X_k) \in P \land$

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

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

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

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

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

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

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)∈P at row X and column T

T∈ FIRST(w)

nullable(w) ∧ T∈ 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)∈P at row X and column T

T∈ FIRST(w)

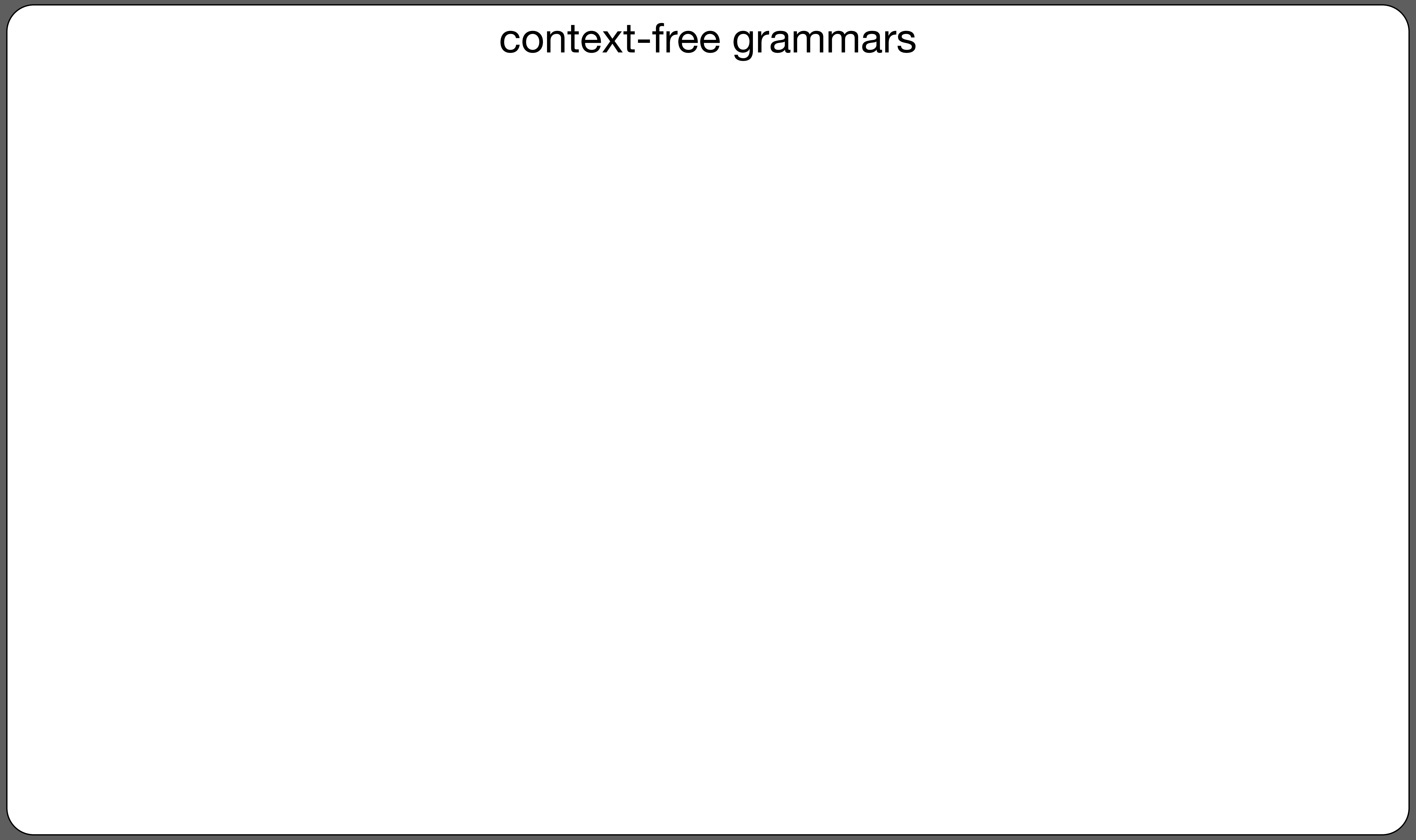
nullable(w) ∧ T∈ FOLLOW(X)

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		

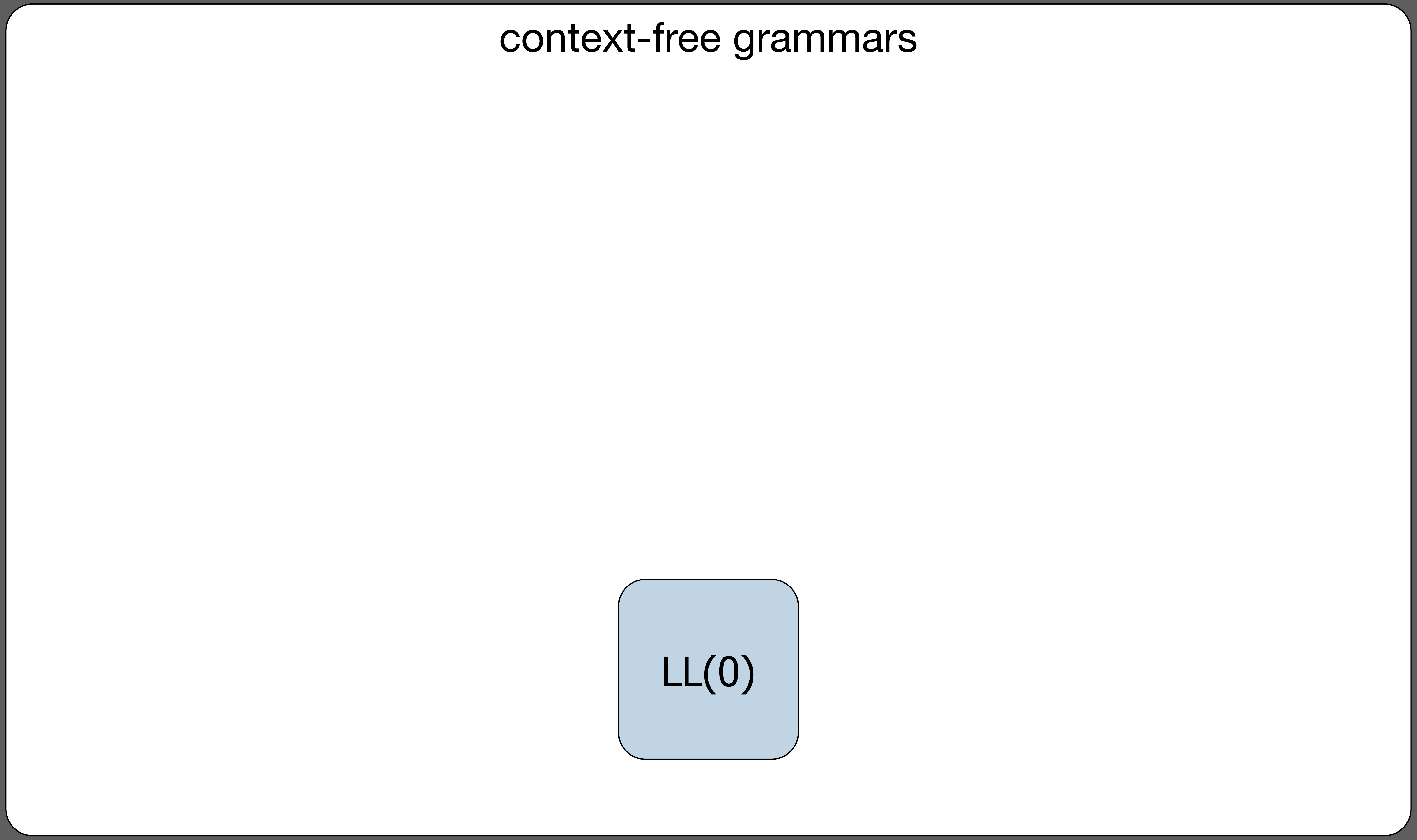
Grammar Classes



context-free grammars

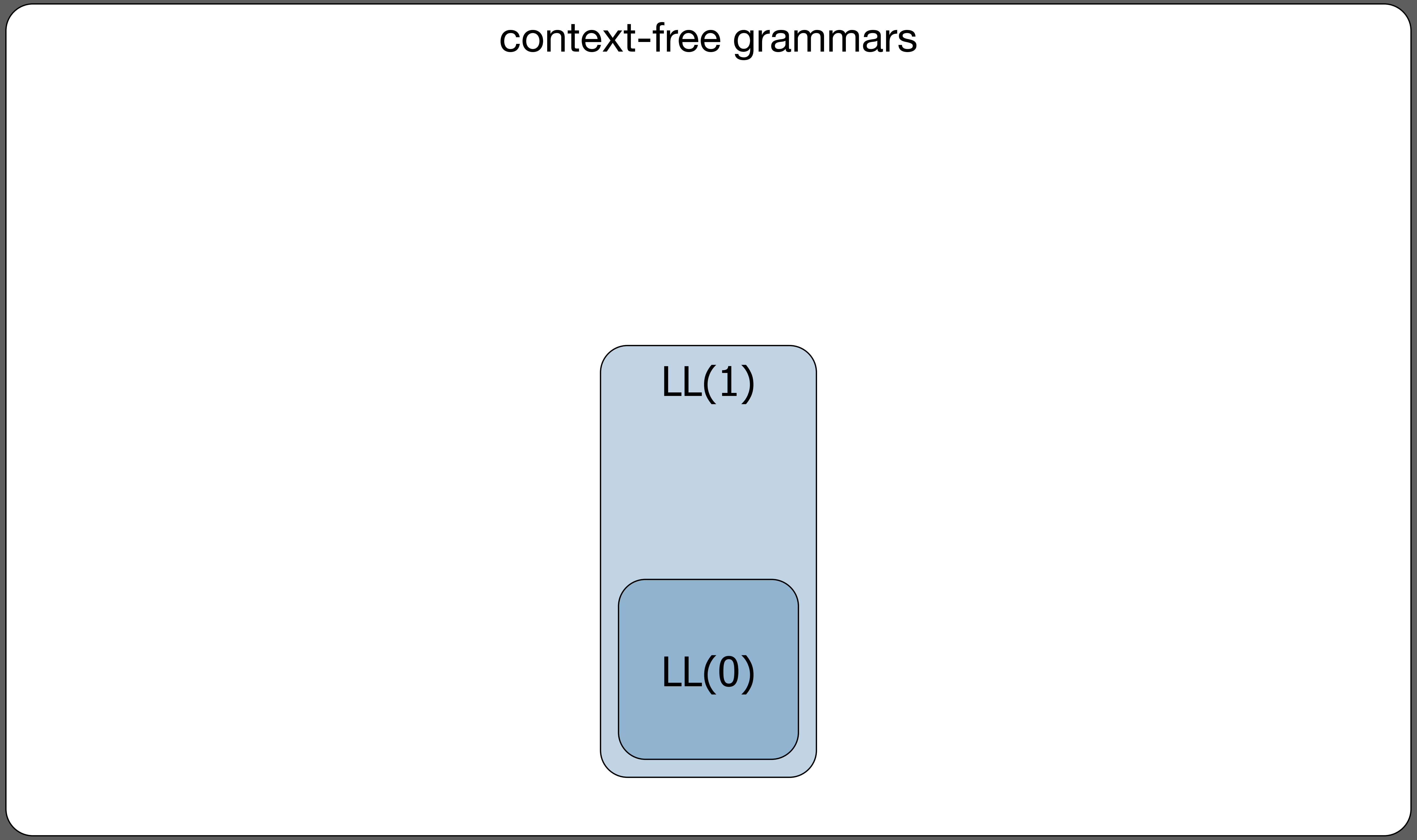
Given the next n tokens, predict next production

Grammar Classes



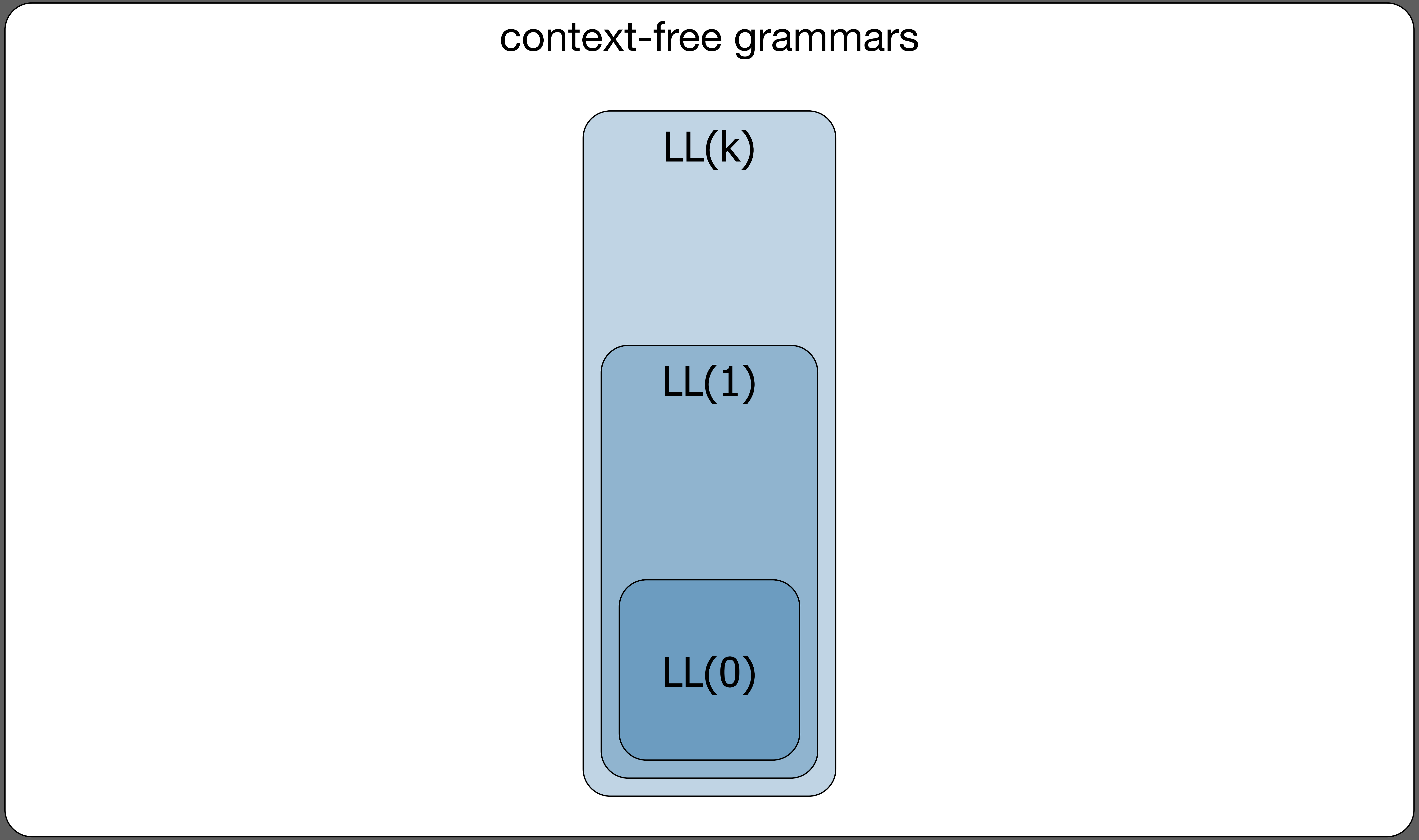
Given the next n tokens, predict next production

Grammar Classes



Given the next n tokens, predict next production

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

Summary

How can we parse context-free languages effectively?

- predictive parsing algorithms

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

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

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

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

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

Literature

ALL(*)

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

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.

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

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

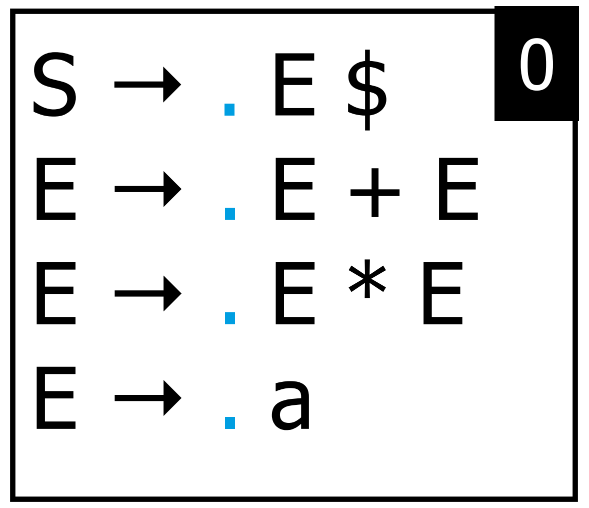
Generalized LR

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
```

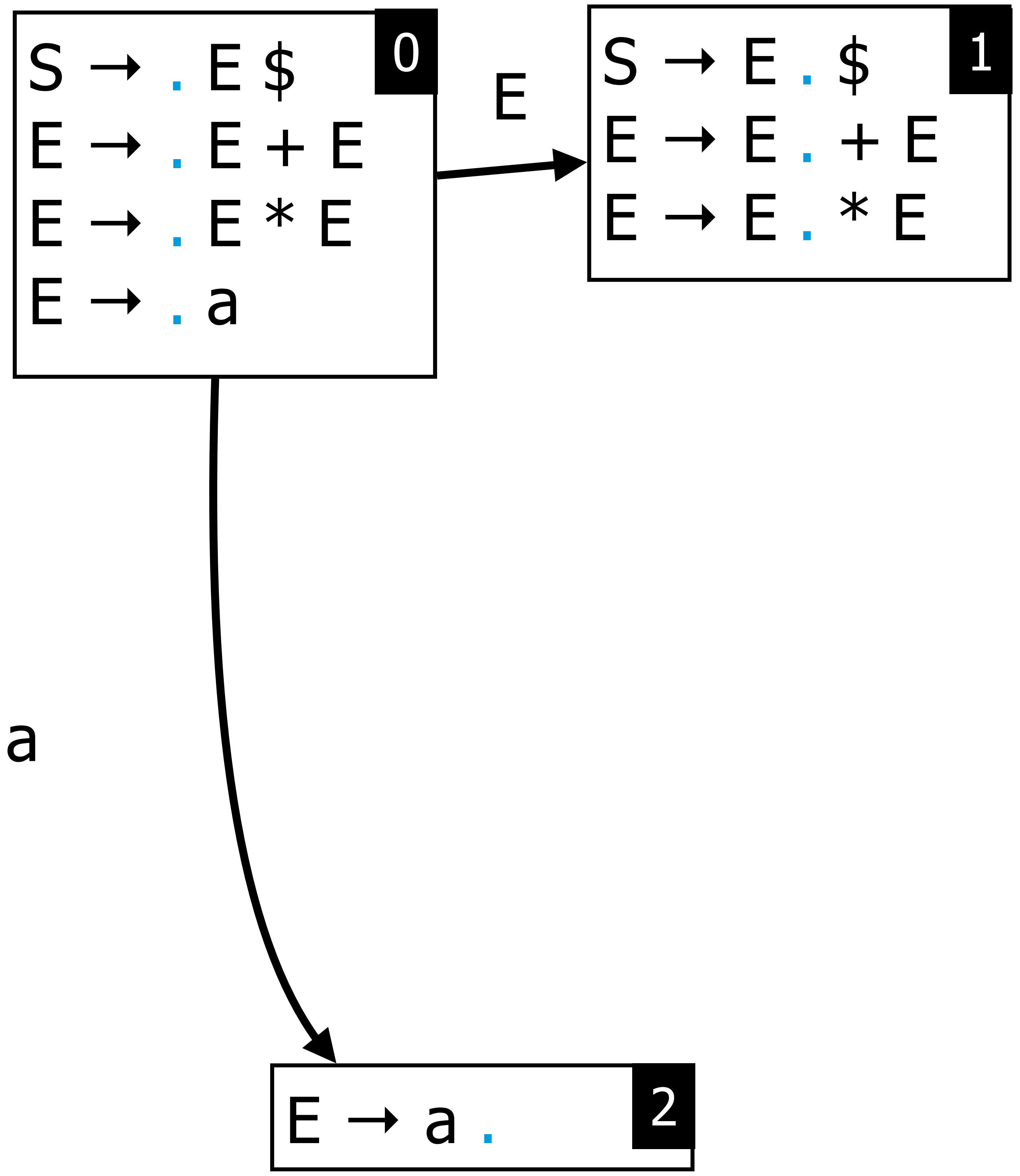
S = E $
E = E + E
E = E * E
E = a



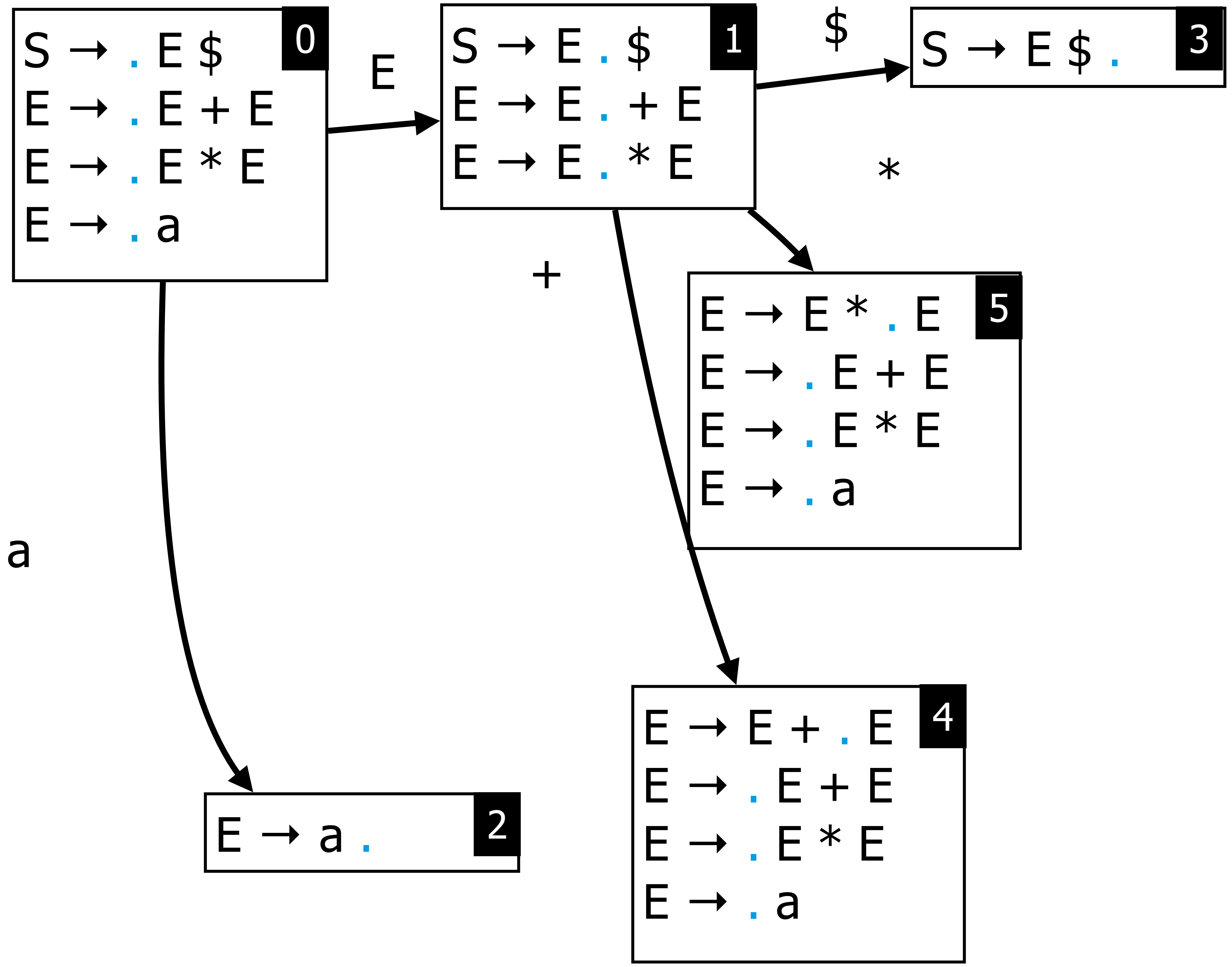
0

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
S → . E $
E → . E + E
E → . E * E
E → . a

E

1
S → E . $
E → E . + E
E → E . * E

$

3
S → E $.

*

+

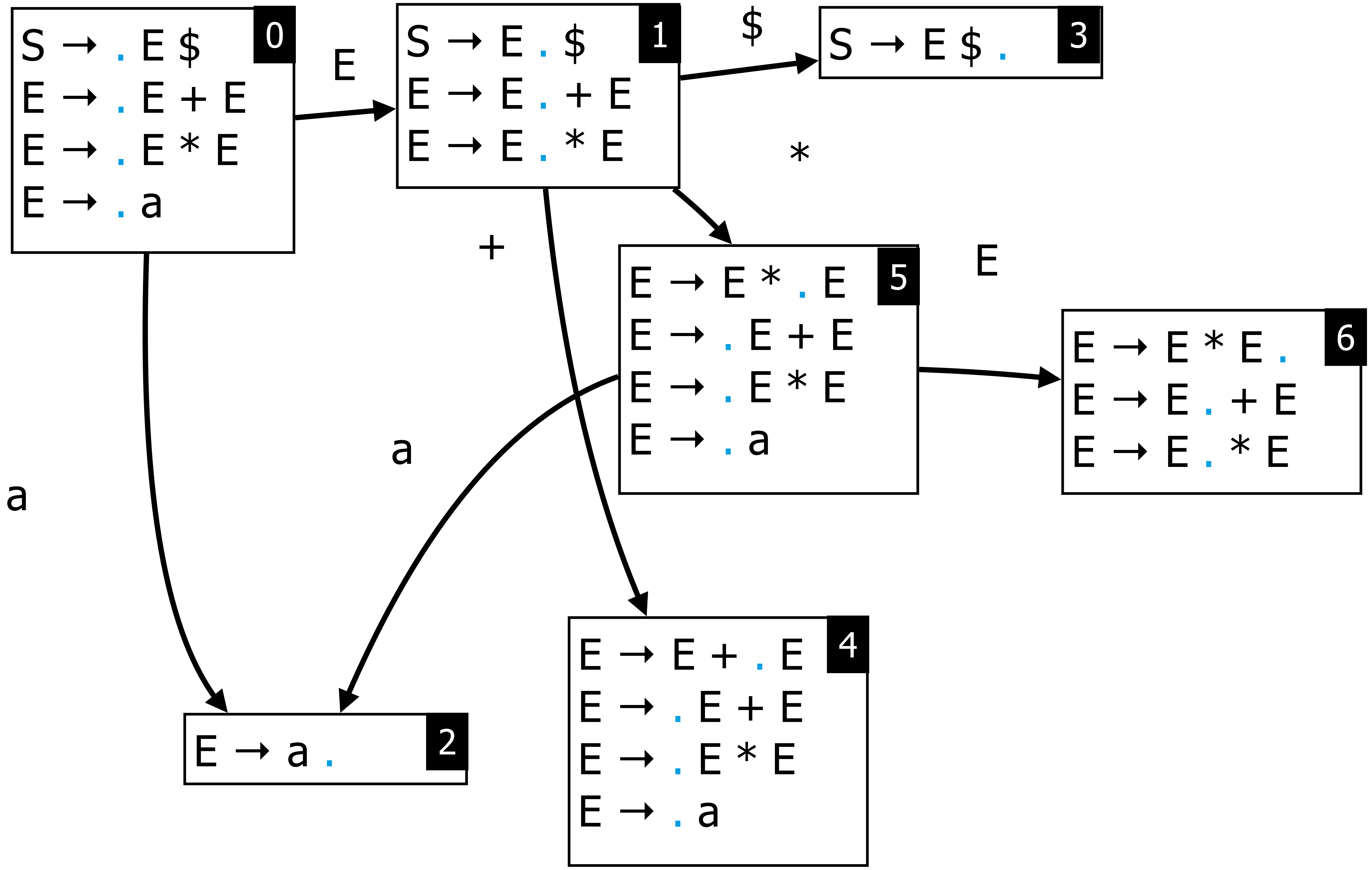
5
E → E * . E
E → . E + E
E → . E * E
E → . a

a

2
E → a .

4
E → E + . E
E → . E + E
E → . E * E
E → . a

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



0
S → . E $
E → . E + E
E → . E * E
E → . a

1
S → E . $
E → E . + E
E → E . * E

3
S → E $.

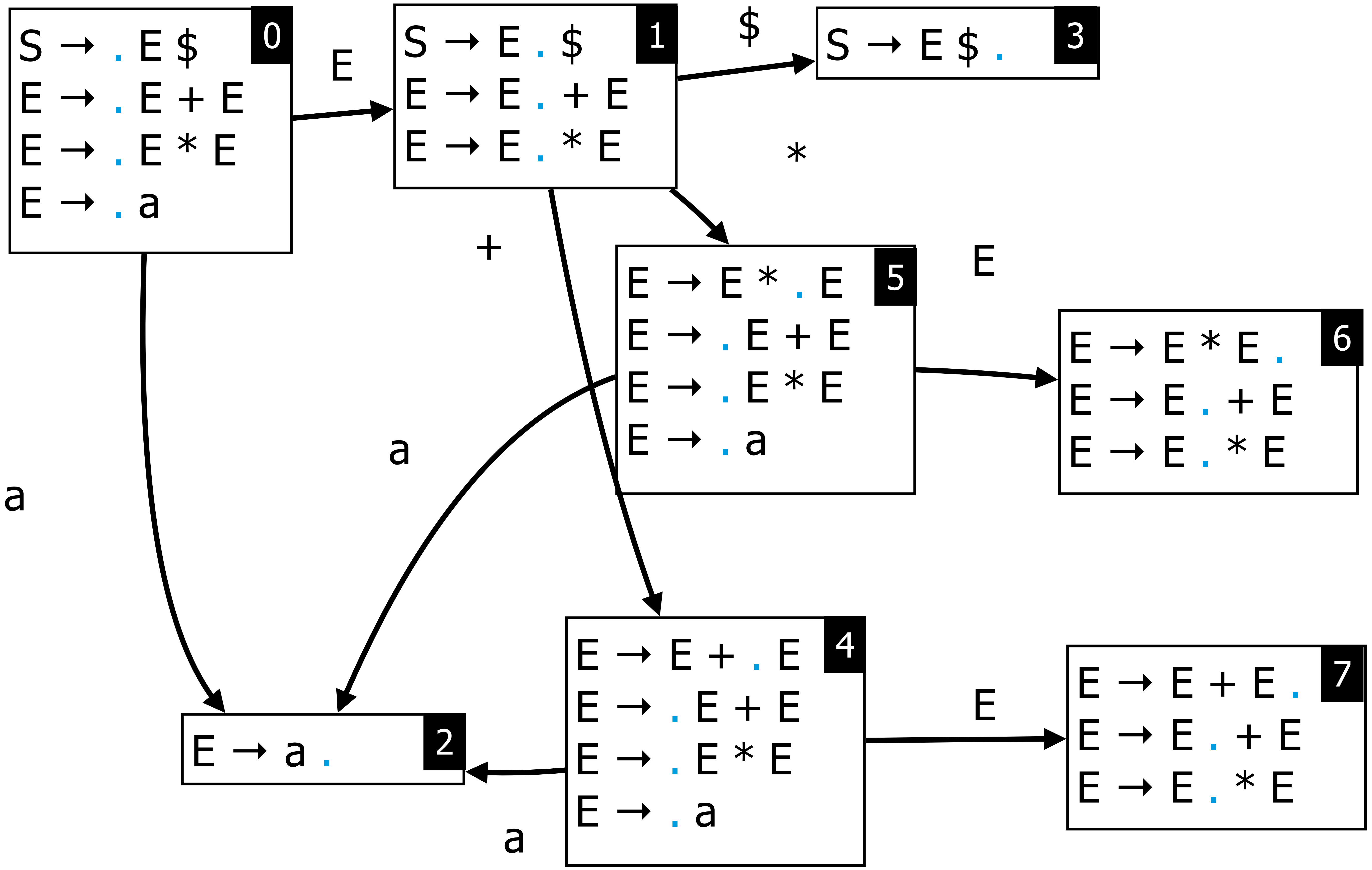
5
E → E * . E
E → . E + E
E → . E * E
E → . a

6
E → E * E .
E → E . + E
E → E . * E

2
E → a .

4
E → E + . E
E → . E + E
E → . E * E
E → . a

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



0
S → . E $
E → . E + E
E → . E * E
E → . a

1
S → E . $
E → E . + E
E → E . * E

2
E → a .

3
S → E $.

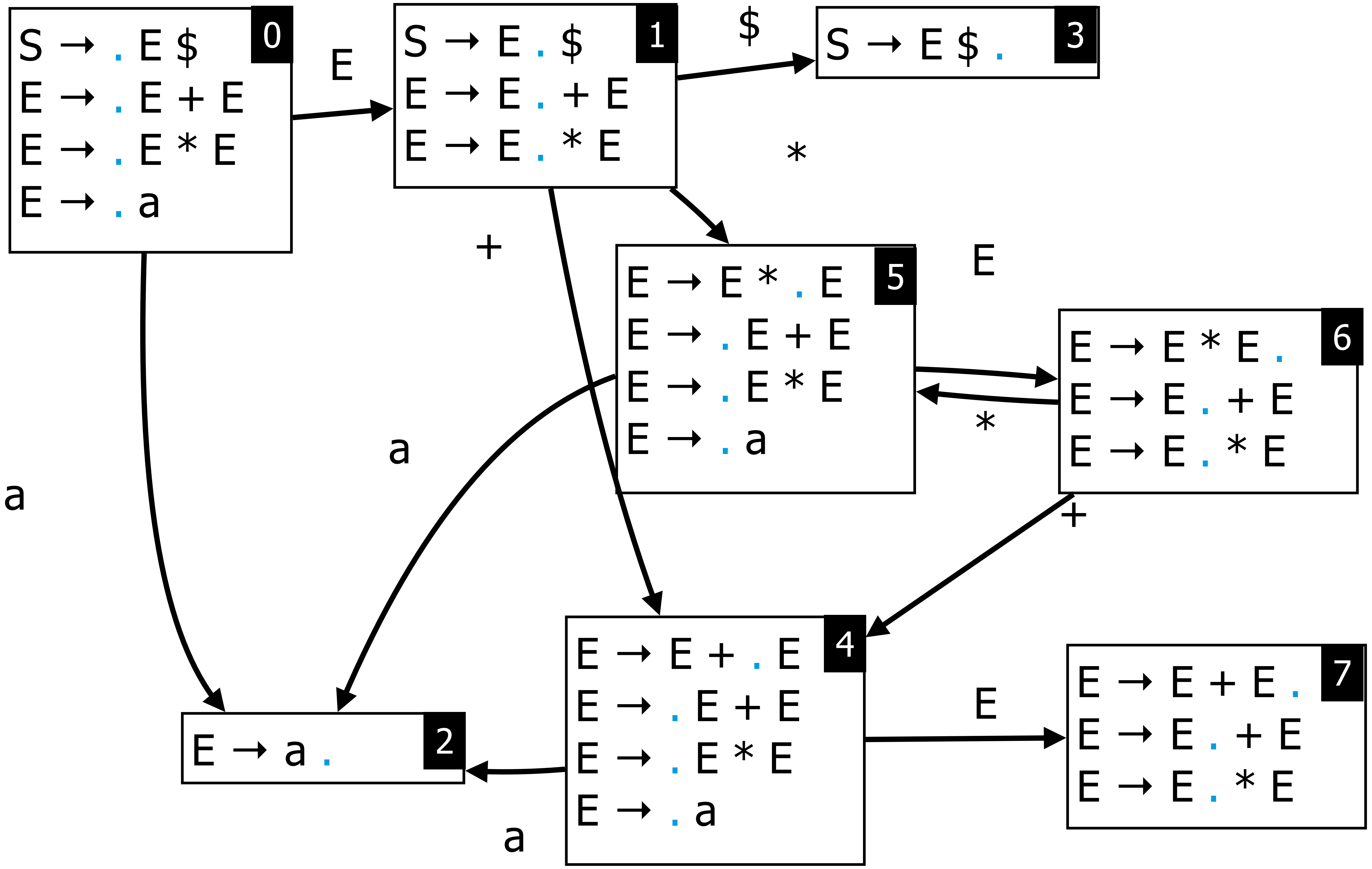
4
E → E + . E
E → . E + E
E → . E * E
E → . a

5
E → E * . E
E → . E + E
E → . E * E
E → . a

6
E → E * E .
E → E . + E
E → E . * E

7
E → E + E .
E → E . + E
E → E . * E

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



0
S → . E $
E → . E + E
E → . E * E
E → . a

1
S → E . $
E → E . + E
E → E . * E

3
S → E $.

5
E → E * . E
E → . E + E
E → . E * E
E → . a

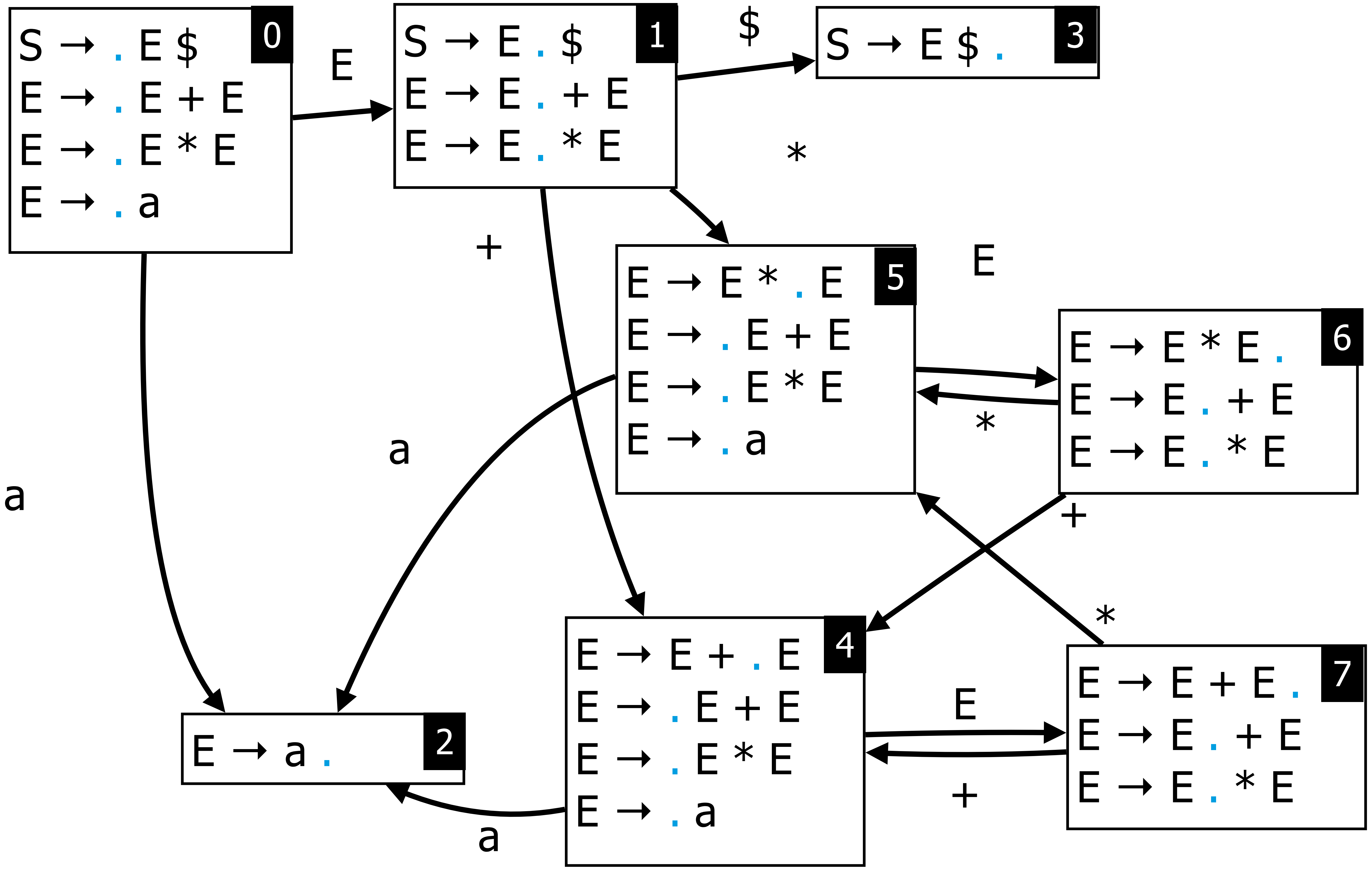
6
E → E * E .
E → E . + E
E → E . * E

2
E → a .

4
E → E + . E
E → . E + E
E → . E * E
E → . a

7
E → E + E .
E → E . + E
E → E . * E

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



0
S → . E $
E → . E + E
E → . E * E
E → . a

1
S → E . $
E → E . + E
E → E . * E

3
S → E $.

5
E → E * . E
E → . E + E
E → . E * E
E → . a

6
E → E * E .
E → E . + E
E → E . * E

2
E → a .

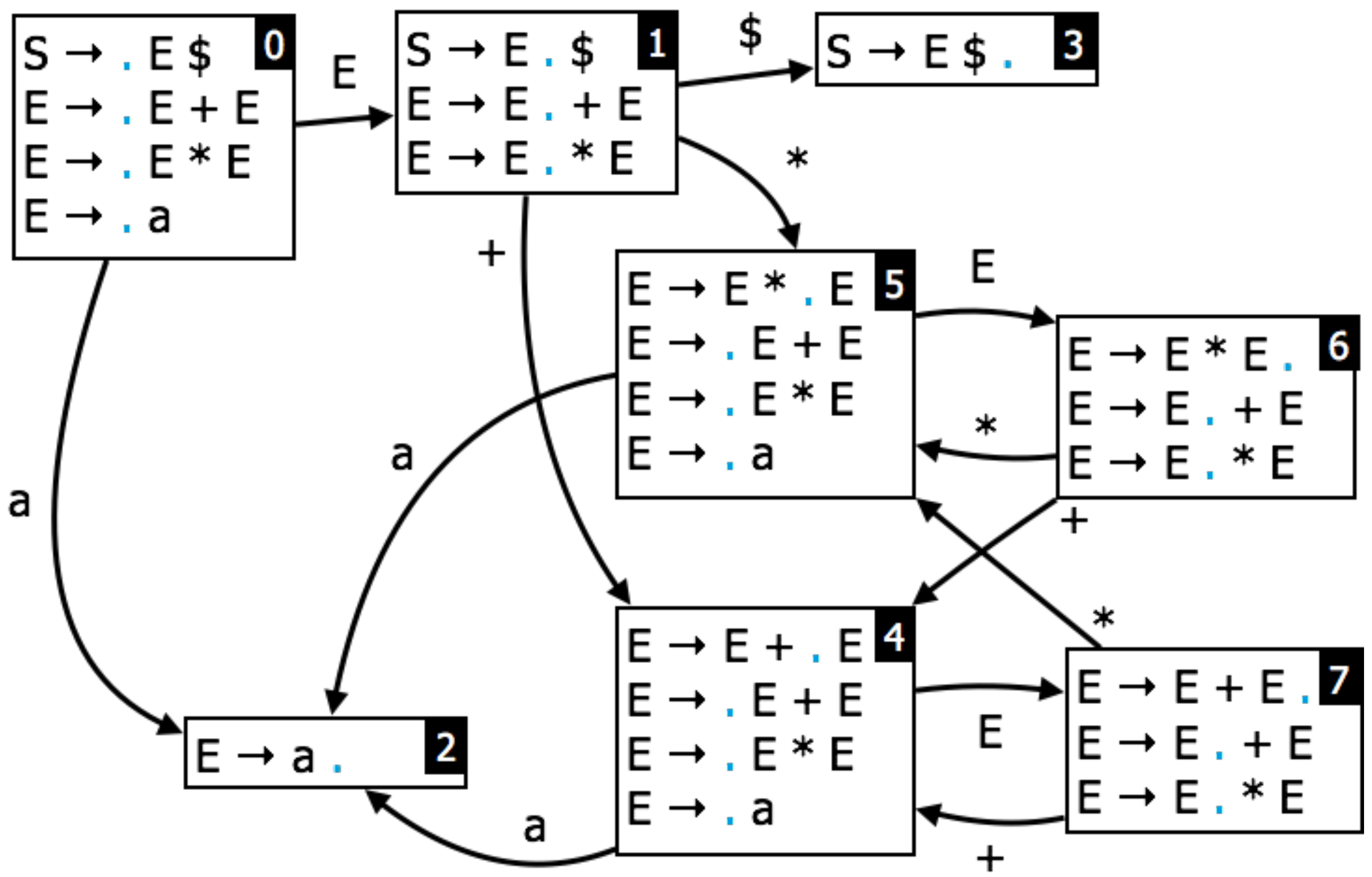
4
E → E + . E
E → . E + E
E → . E * E
E → . a

7
E → E + E .
E → E . + E
E → E . * E

SLR Table

Nonter	Nullable	First	Follow
S			
E			

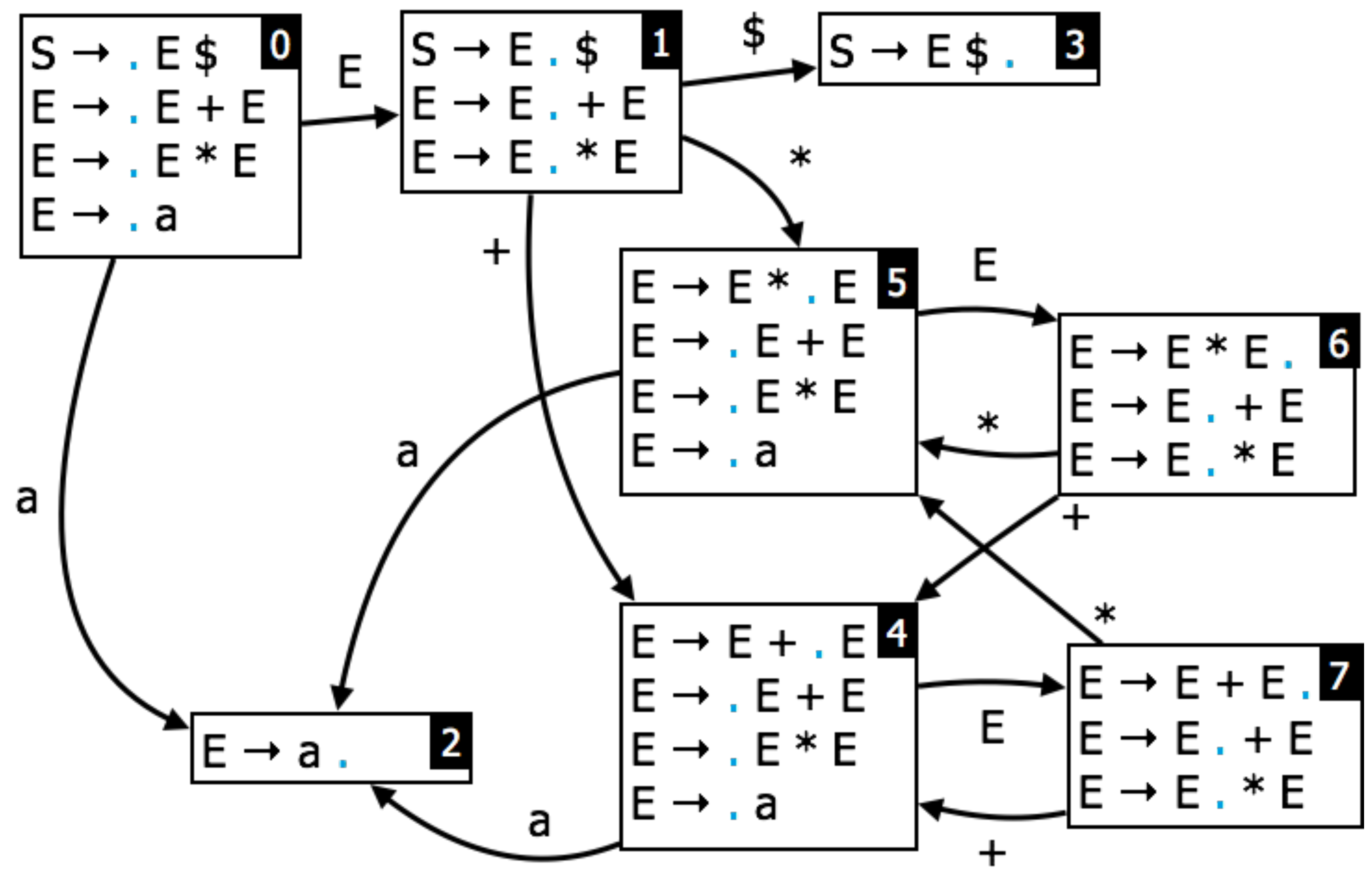
State	Action				Goto	
	a	+	*	$	S	E
0						
1						
2						
3						
4						
5						
6						
7						



```
(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	+, *, $


S → . E $ 0
E → . E + E
E → . E * E
E → . a

S → E . $ 1
E → E . + E
E → E . * E

S → E $. 3

E → a . 2

E → E + . E 4
E → . E + E
E → . E * E
E → . a

E → E * . E 5
E → . E + E
E → . E * E
E → . a

E → E * E . 6
E → E . + E
E → E . * E

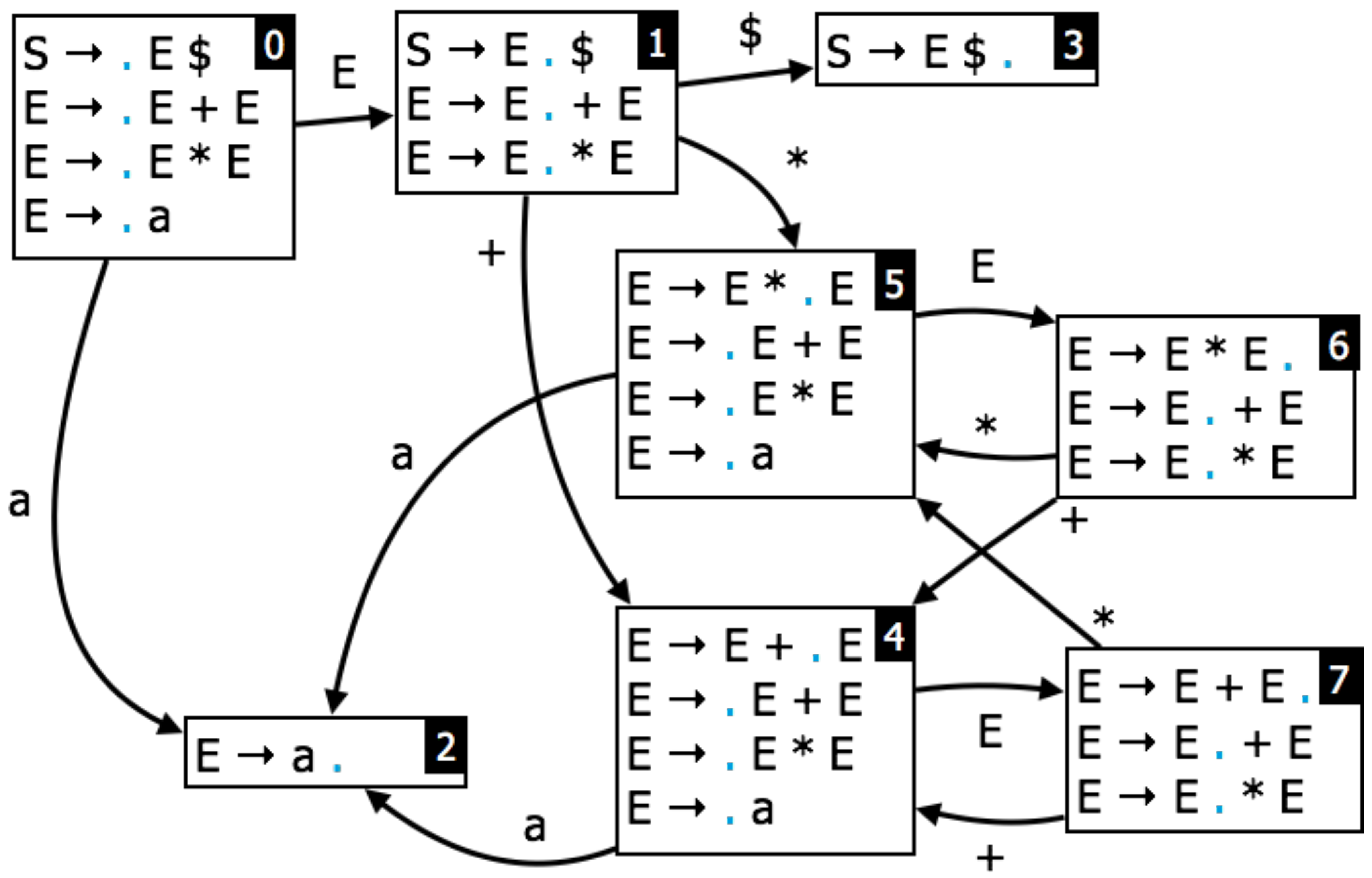
E → E + E . 7
E → E . + E
E → E . * E

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	+, *, $



S → . E $ 0
E → . E + E
E → . E * E
E → . a

S → E . $ 1
E → E . + E
E → E . * E

S → E $. 3

E → E * . E 5
E → . E + E
E → . E * E
E → . a

E → E * E . 6
E → E . + E
E → E . * E

E → E + . E 4
E → . E + E
E → . E * E
E → . a

E → E + E . 7
E → E . + E
E → E . * E

E → a . 2

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		

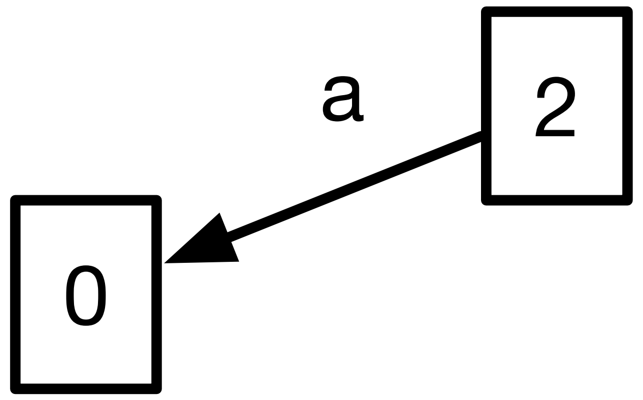
0

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		



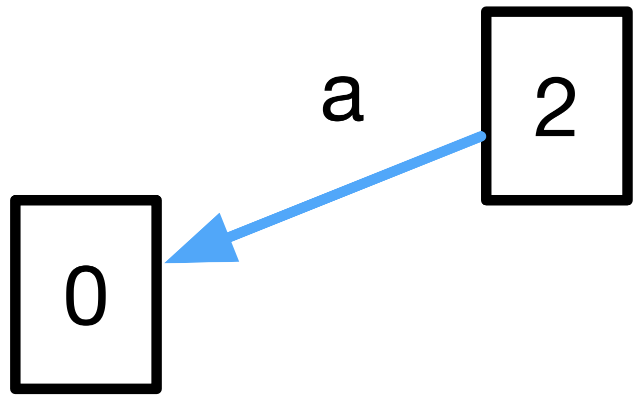
synchronize on shifts

Parsing

input: + a * a $

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

<table>
<tr><th rowspan="2">State</th><th colspan="4">Action</th><th colspan="2">Goto</th></tr>
<tr><th>a</th><th>+</th><th>*</th><th>$</th><th>S</th><th>E</th></tr>
<tr><td>0</td><td>s2</td><td></td><td></td><td></td><td></td><td>1</td></tr>
<tr><td>1</td><td></td><td>s4</td><td>s5</td><td>acc</td><td></td><td></td></tr>
<tr><td>2</td><td></td><td>r3</td><td>r3</td><td>r3</td><td></td><td></td></tr>
<tr><td>3</td><td>acc</td><td>acc</td><td>acc</td><td>acc</td><td></td><td></td></tr>
<tr><td>4</td><td>s2</td><td></td><td></td><td></td><td></td><td>7</td></tr>
<tr><td>5</td><td>s2</td><td></td><td></td><td></td><td></td><td>6</td></tr>
<tr><td>6</td><td></td><td>s4/r2</td><td>s5/r2</td><td>r2</td><td></td><td></td></tr>
<tr><td>7</td><td></td><td>s4/r1</td><td>s5/r1</td><td>r1</td><td></td><td></td></tr>
</table>

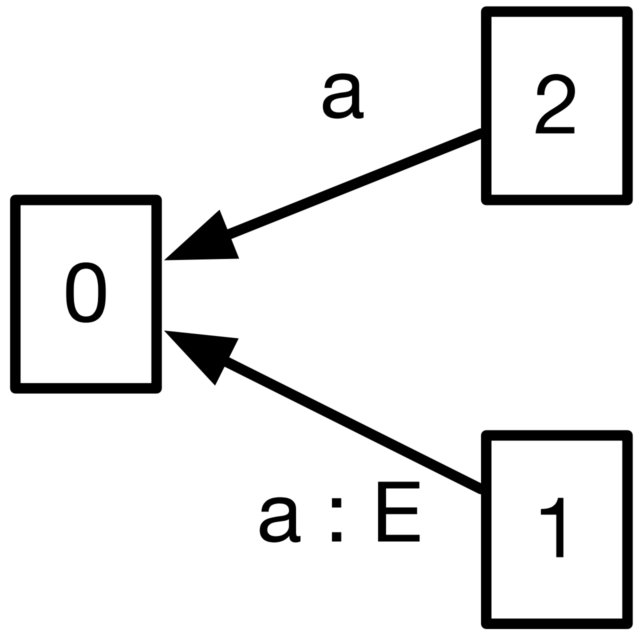


Parsing

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		

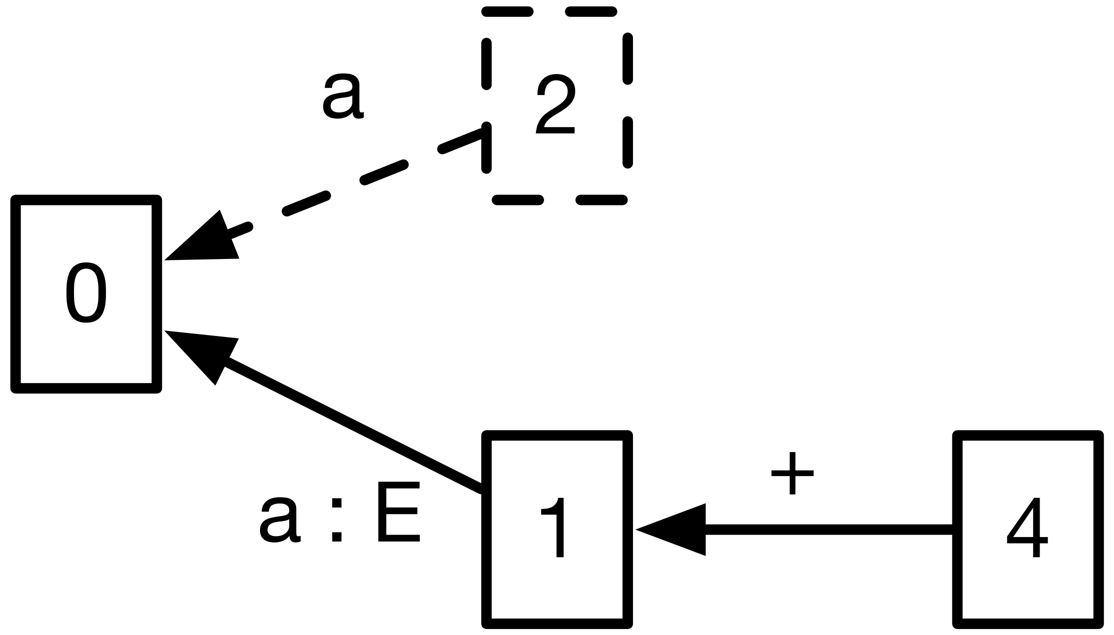


Parsing

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		



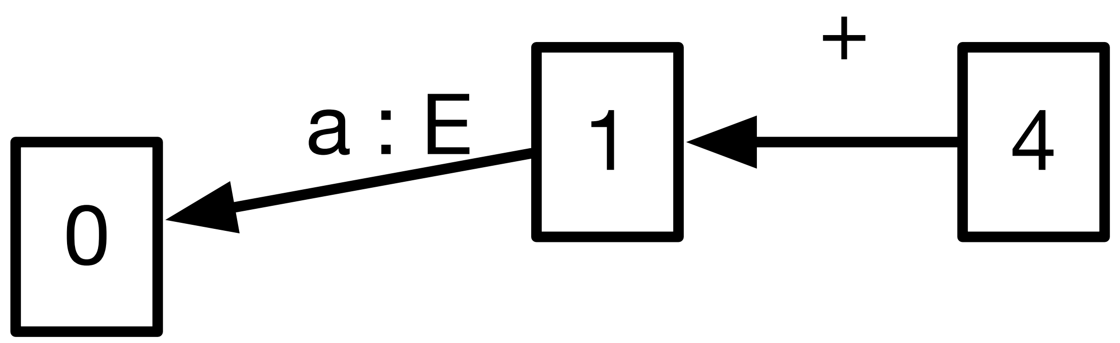
synchronize

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		



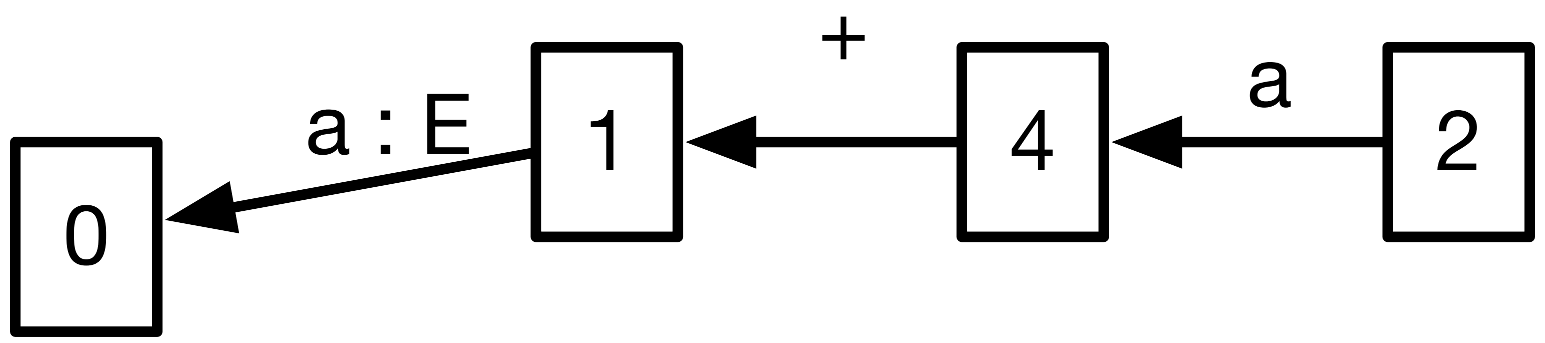
Parsing

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

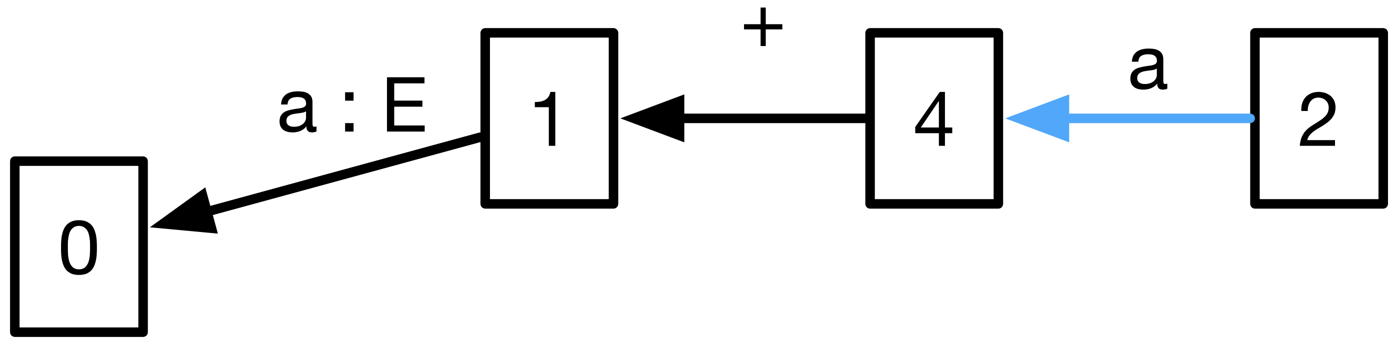


Parsing

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		

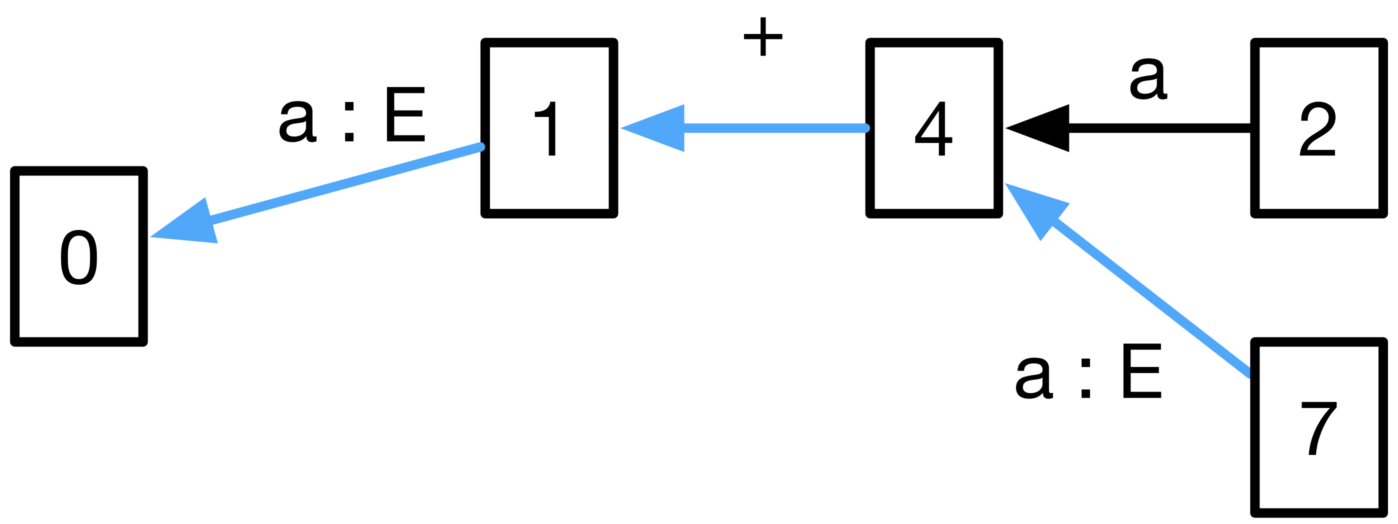


Parsing

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		

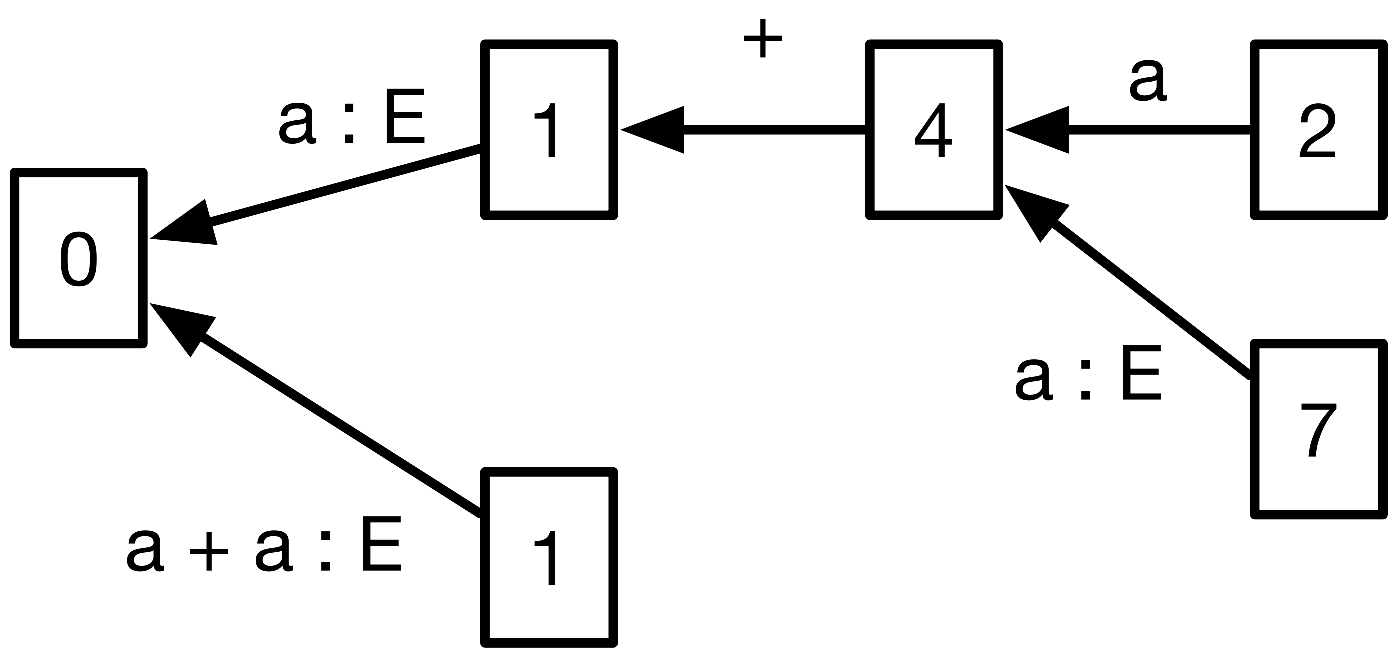


Parsing

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		



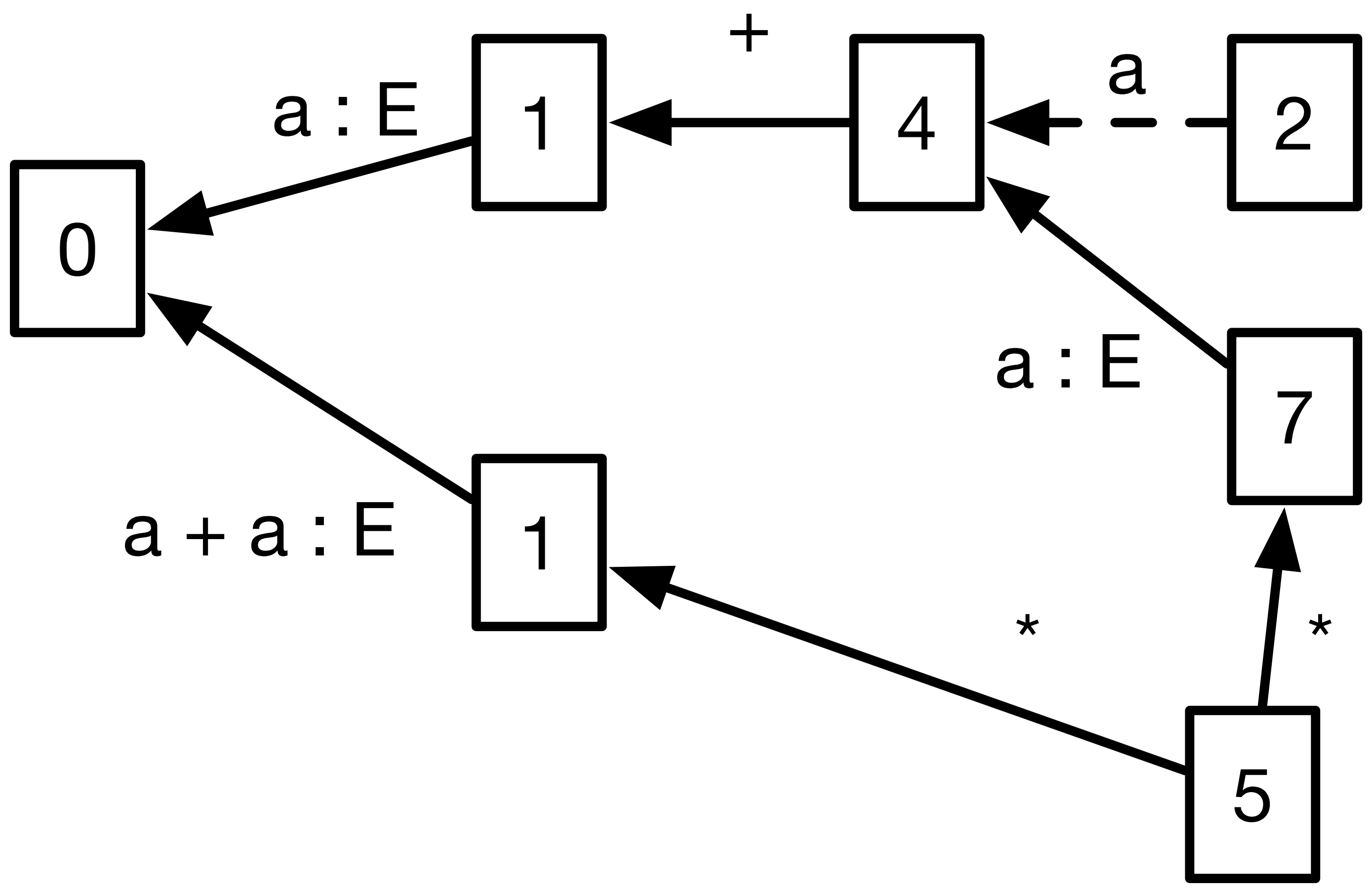
Parsing

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

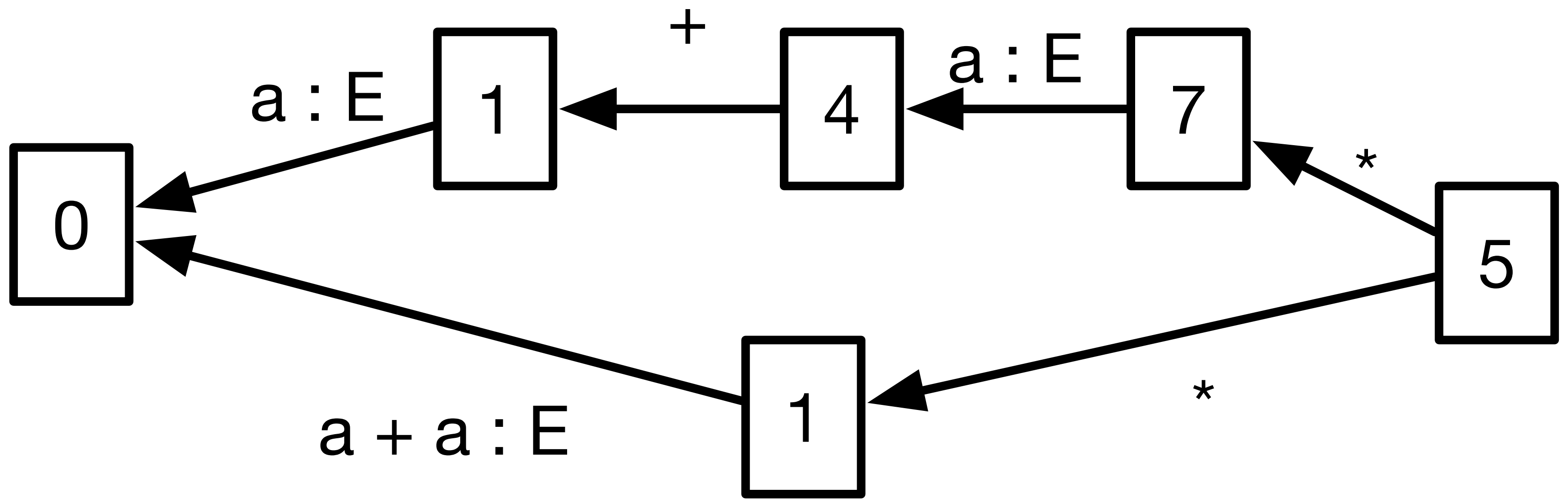


Parsing

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		



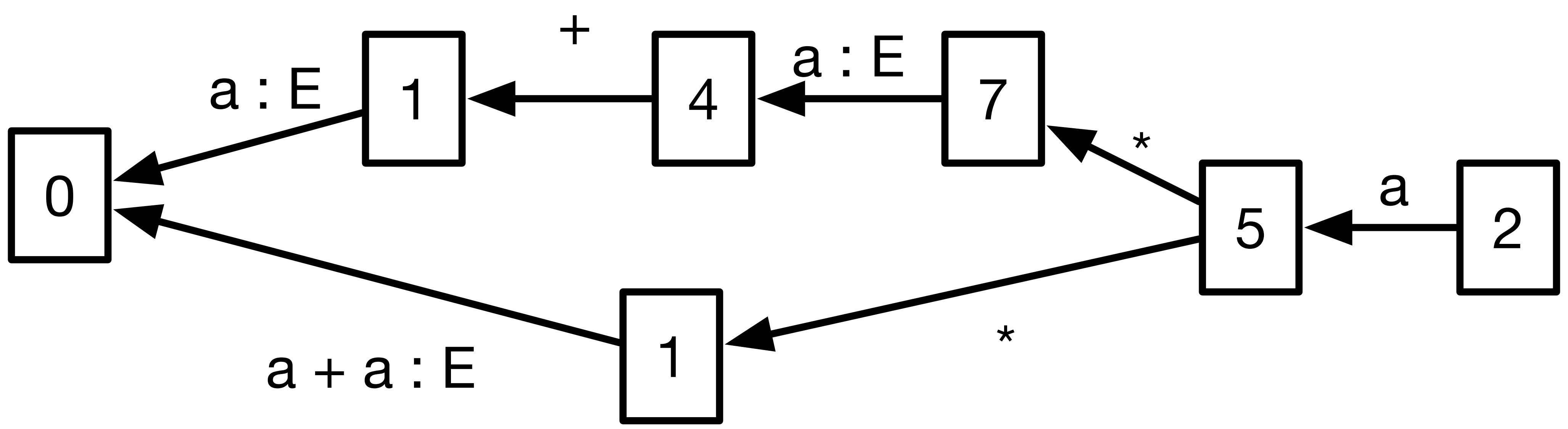
Parsing

input: $

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

State	Action				Goto	
	a	+	*	$	S	E
0	s2					I
I		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/rI	s5/rI	rI		

synchronize

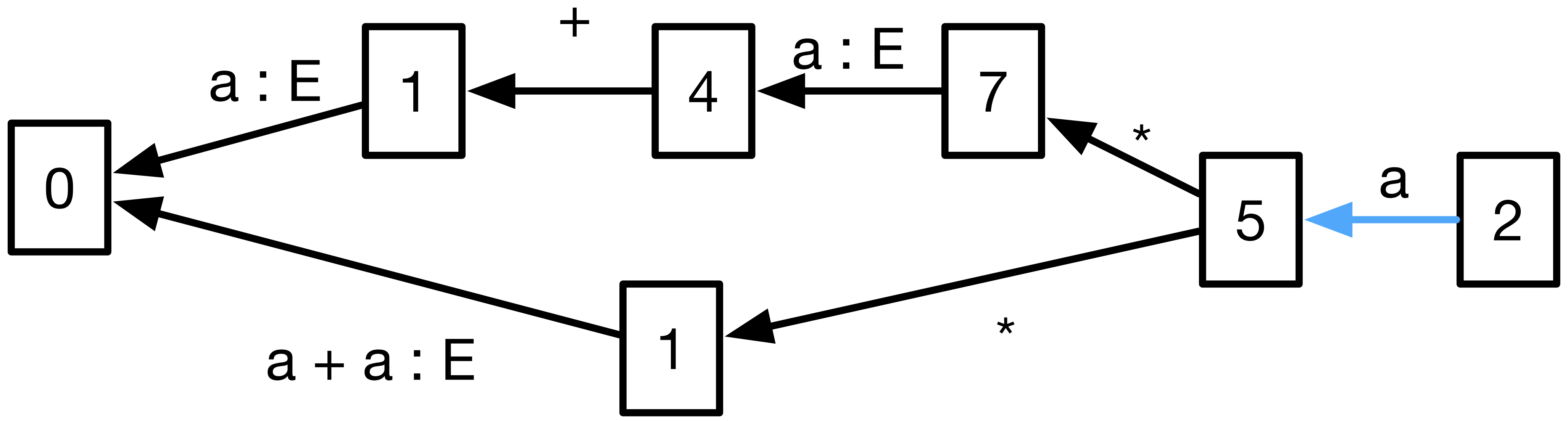


Parsing

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		

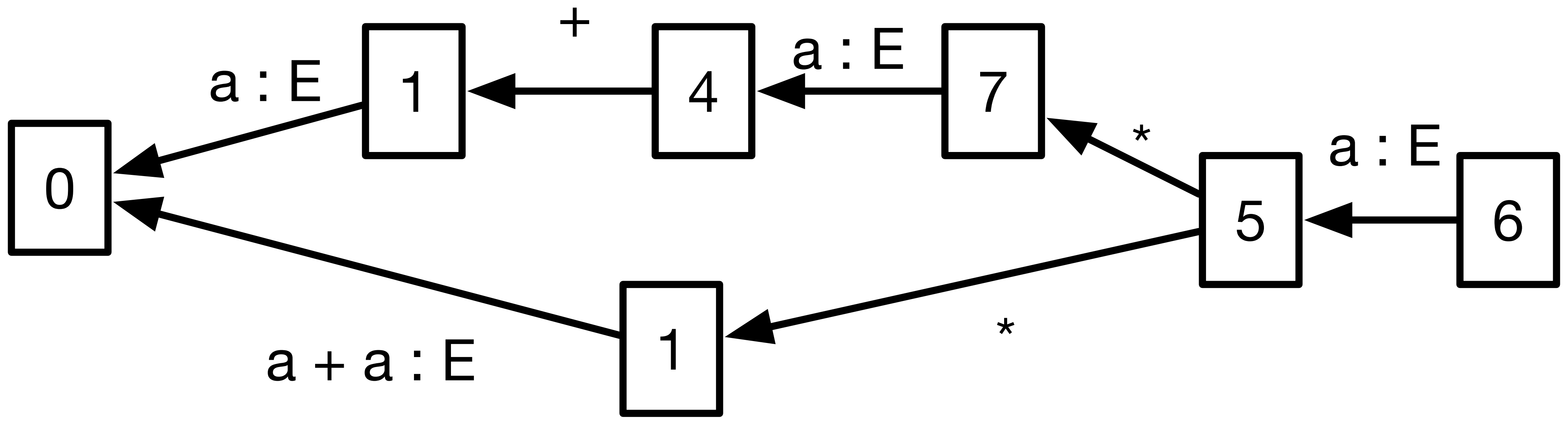


Parsing

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		

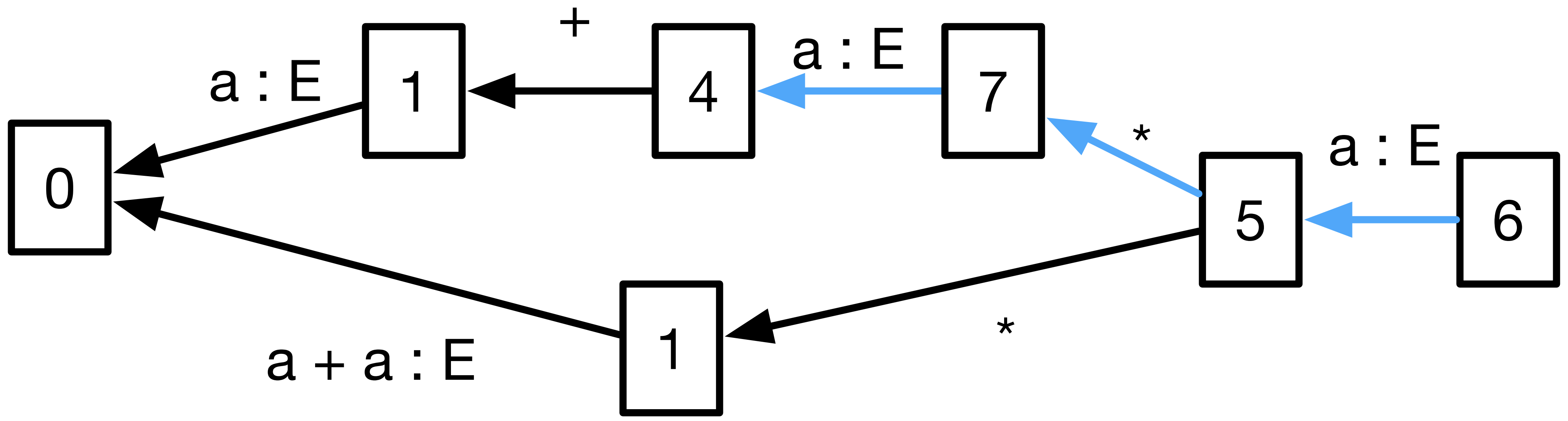


Parsing

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		

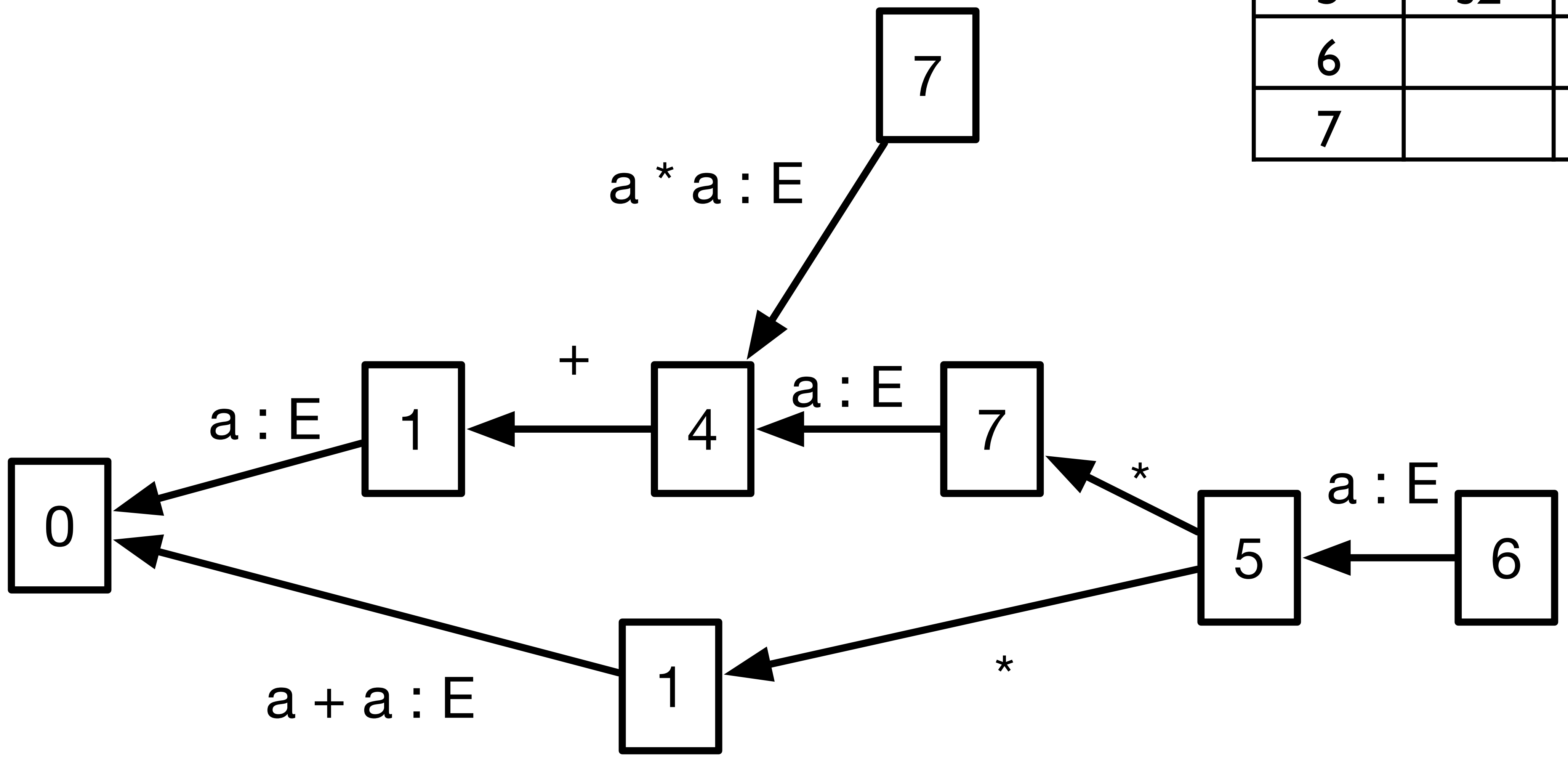


Parsing

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		

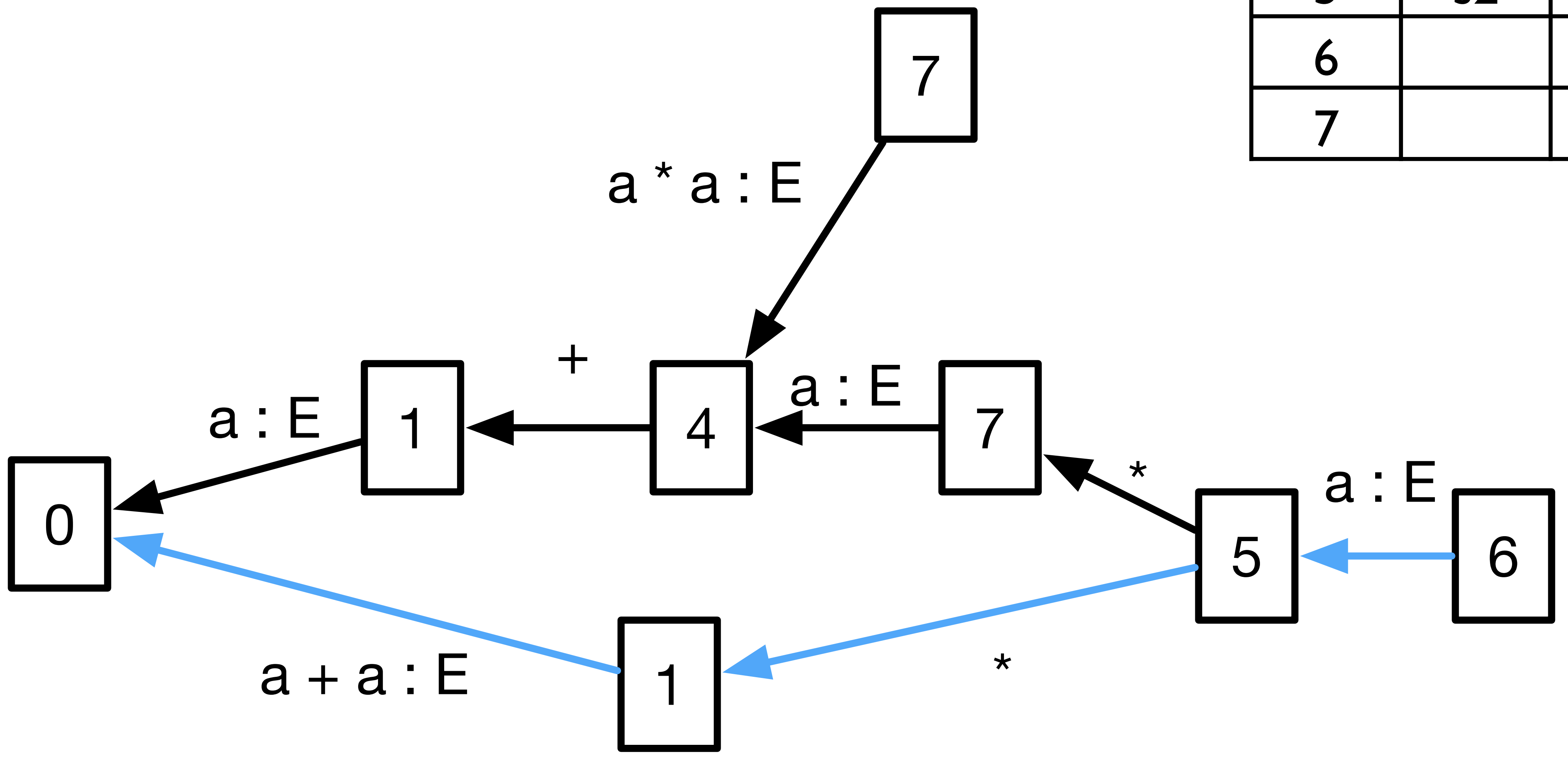


Parsing

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		

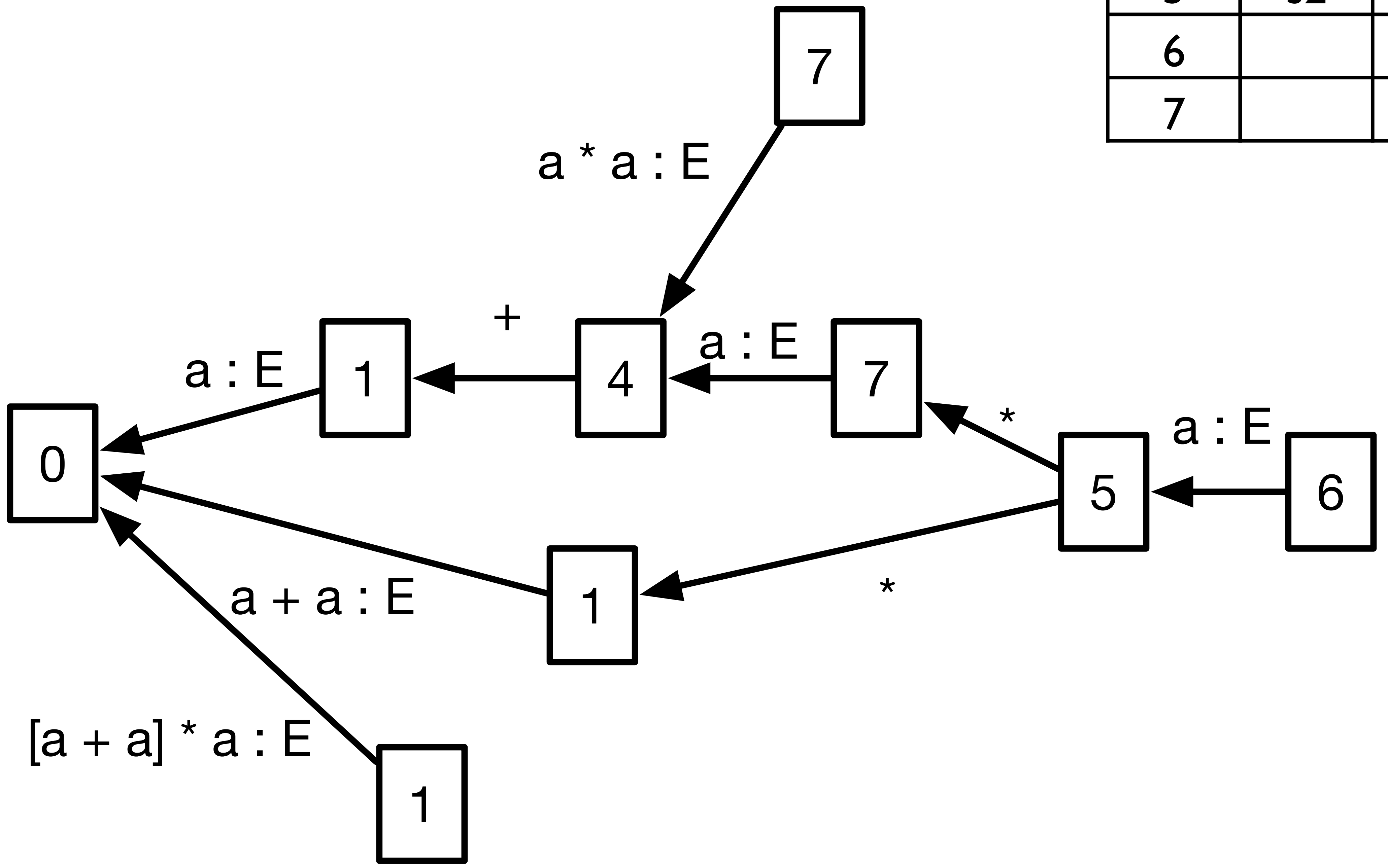


Parsing

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		

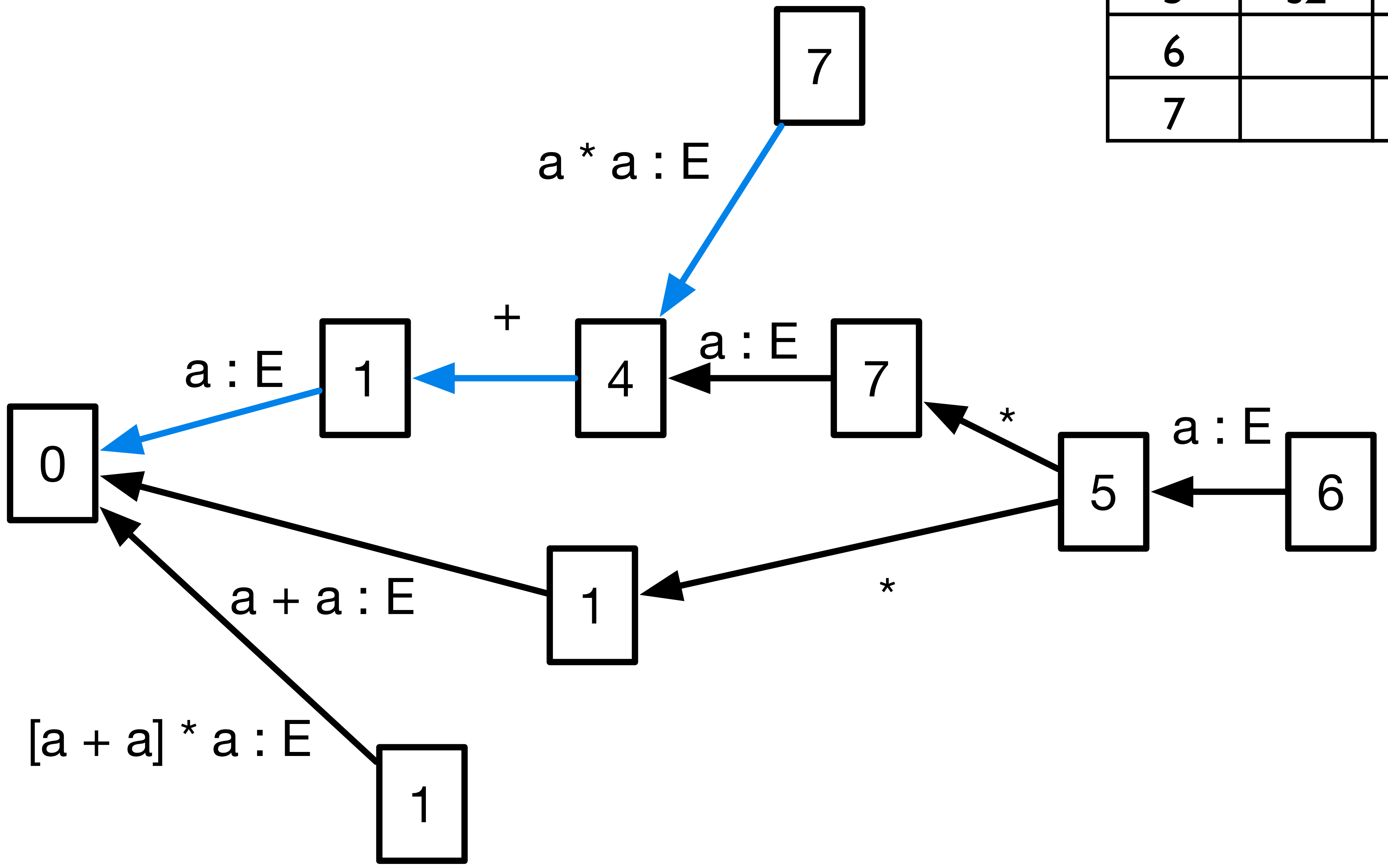


Parsing

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		

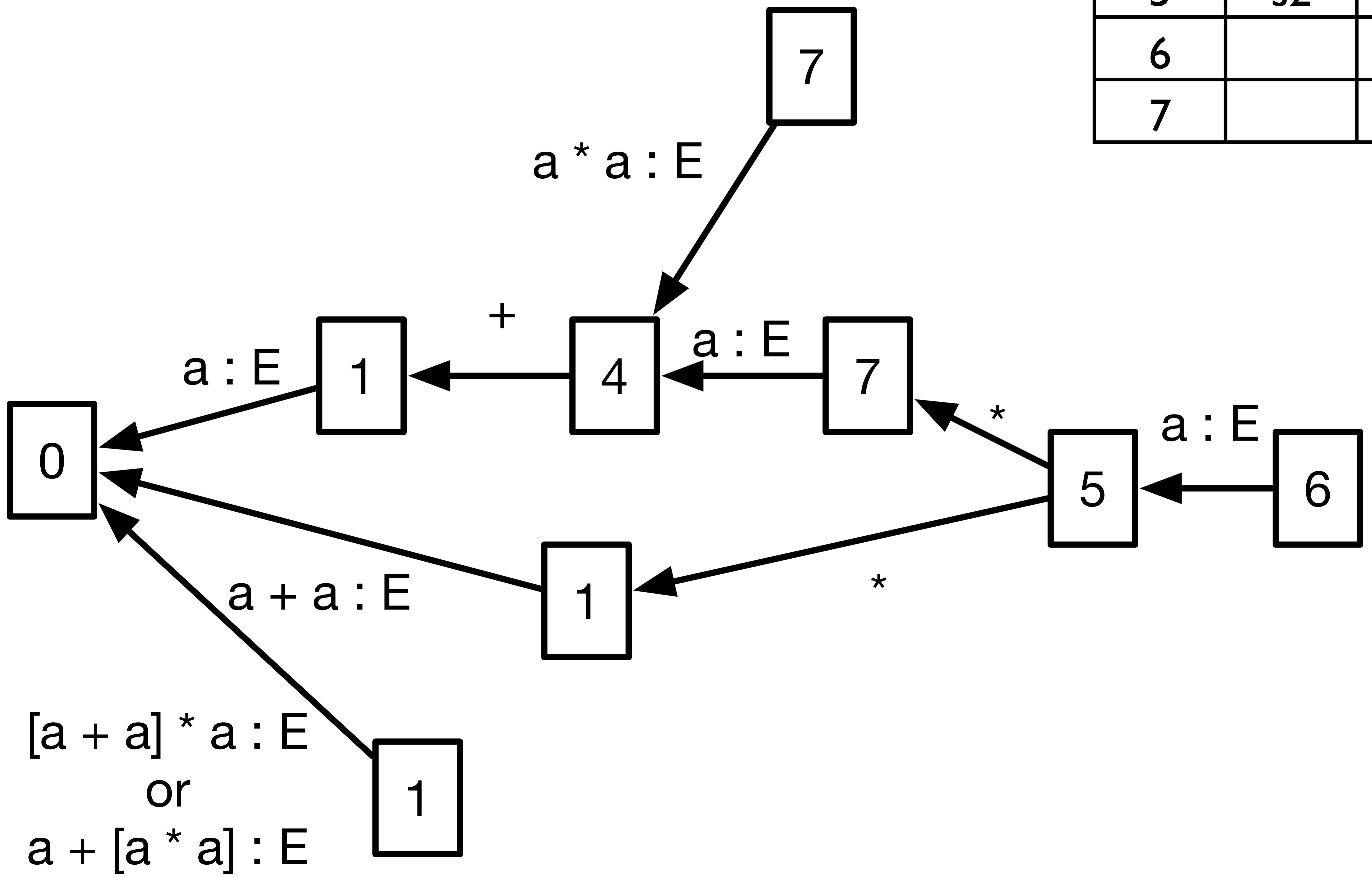


Parsing

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		

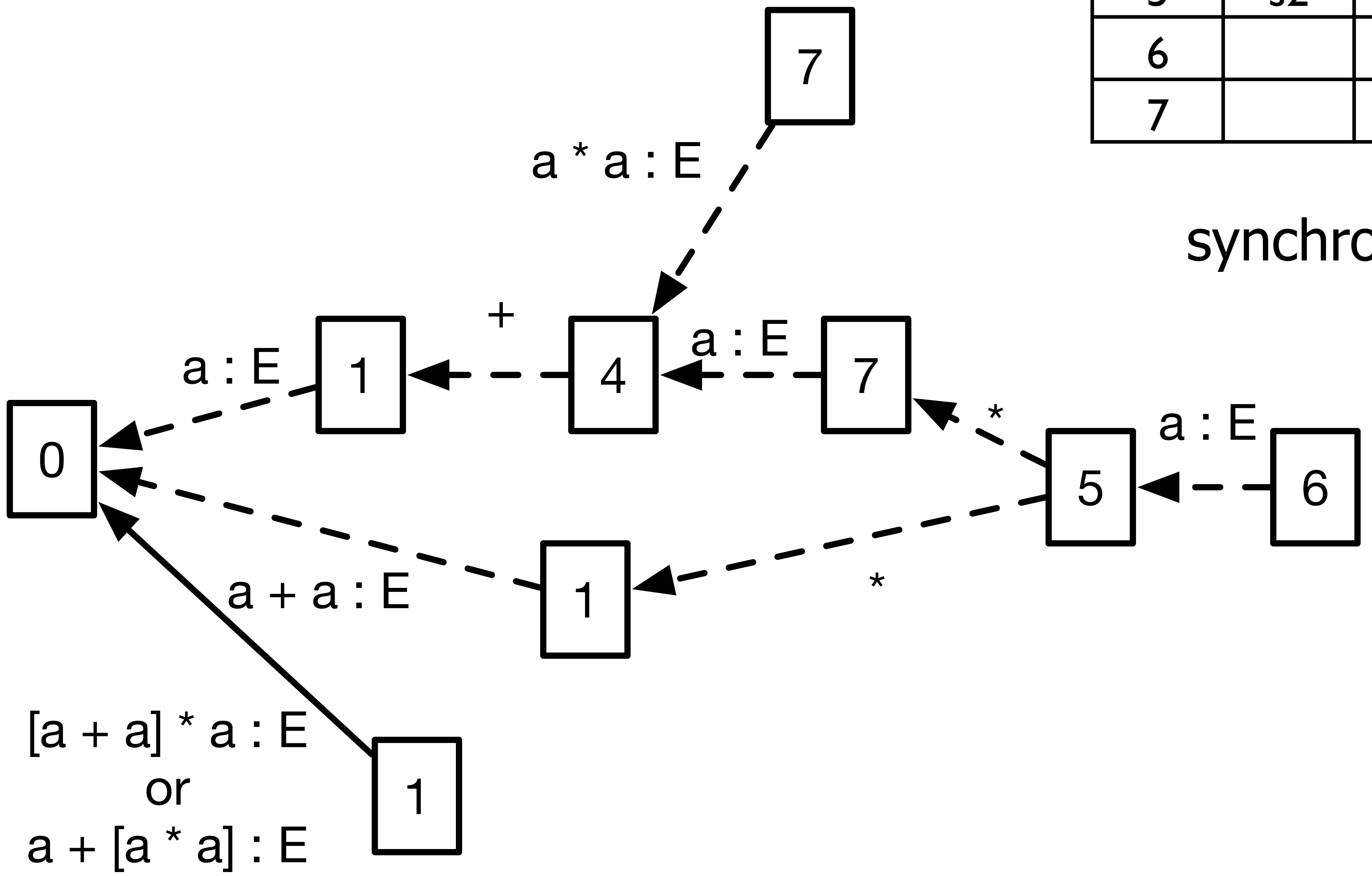


Parsing

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