Declare Your Language

Chapter 14: Parsing

Luís Eduardo de Souza Amorim

IN4303 Compiler Construction
TU Delft
December 2017

Syntax Definition
Parser Gen
Parse Table
Source Code
Parse
Type Check
Optimize
CodeGen
Machine Code

Declarative syntax definition

Derive parser from syntax definition

This Lecture

~~Lexical syntax~~

- ~~defining the syntax of tokens / terminals including layout~~
- ~~making lexical syntax explicit~~

~~Formatting specification~~

- ~~how to map (abstract syntax) trees to text~~

~~Syntactic completion~~

- ~~proposing valid syntactic completions in an editor~~

Parsing

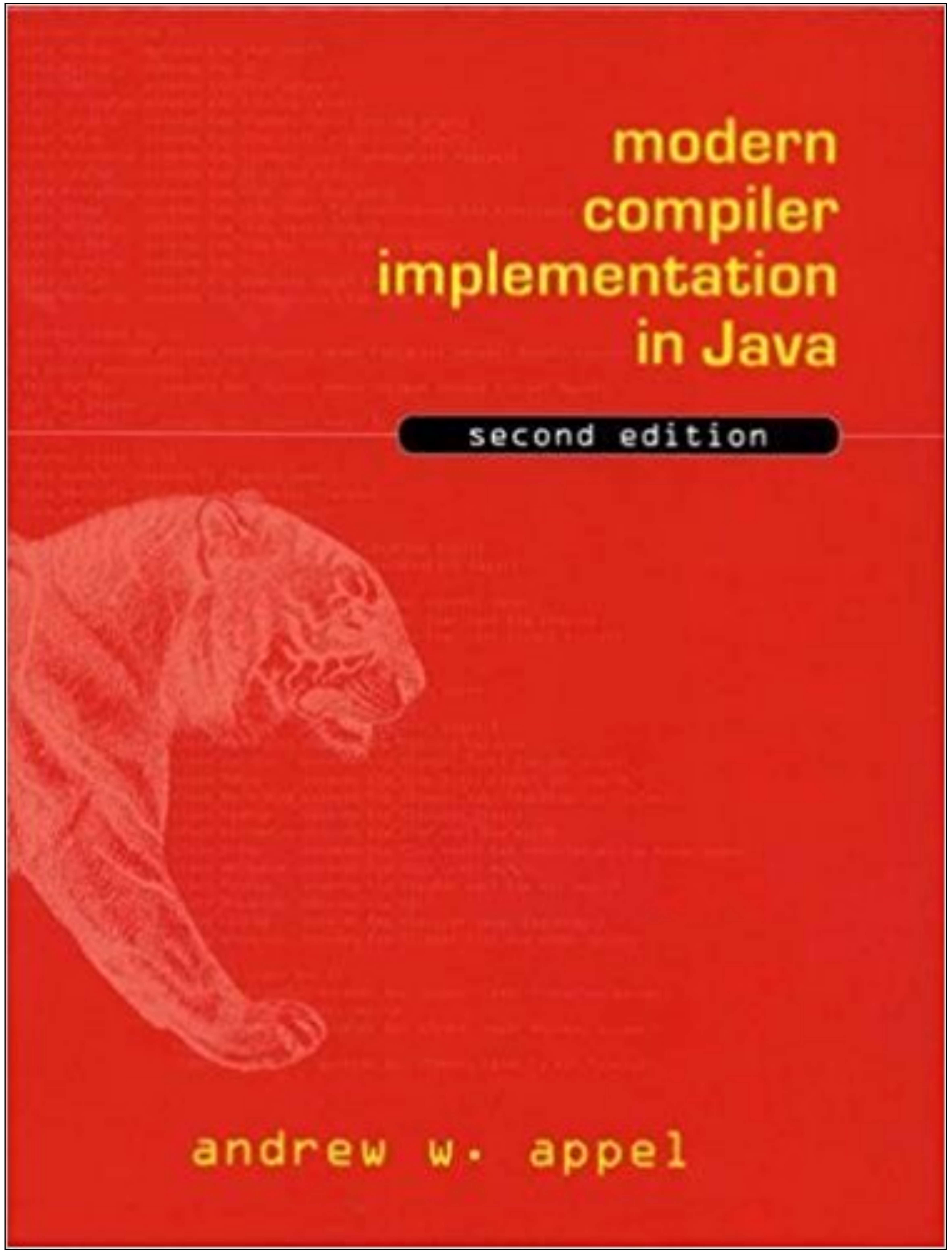
- interpreting a syntax definition to map text to trees

~~Declarative Disambiguation~~

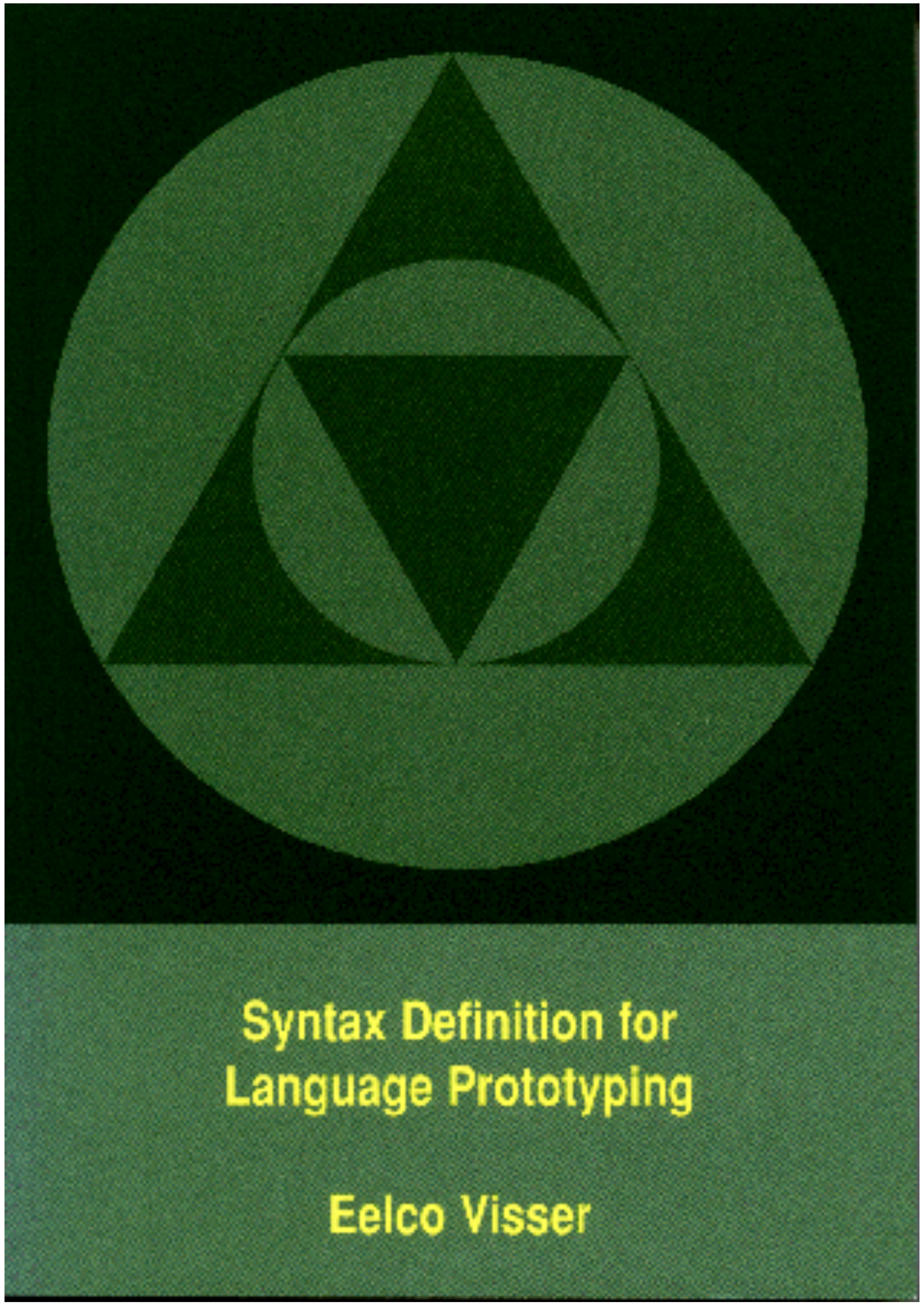
- ~~solving conflicts when parsing expression languages~~

Reading Material

LL/LR Parsing



SGLR Parsing


Syntax Definition for Language Prototyping

Eelco Visser

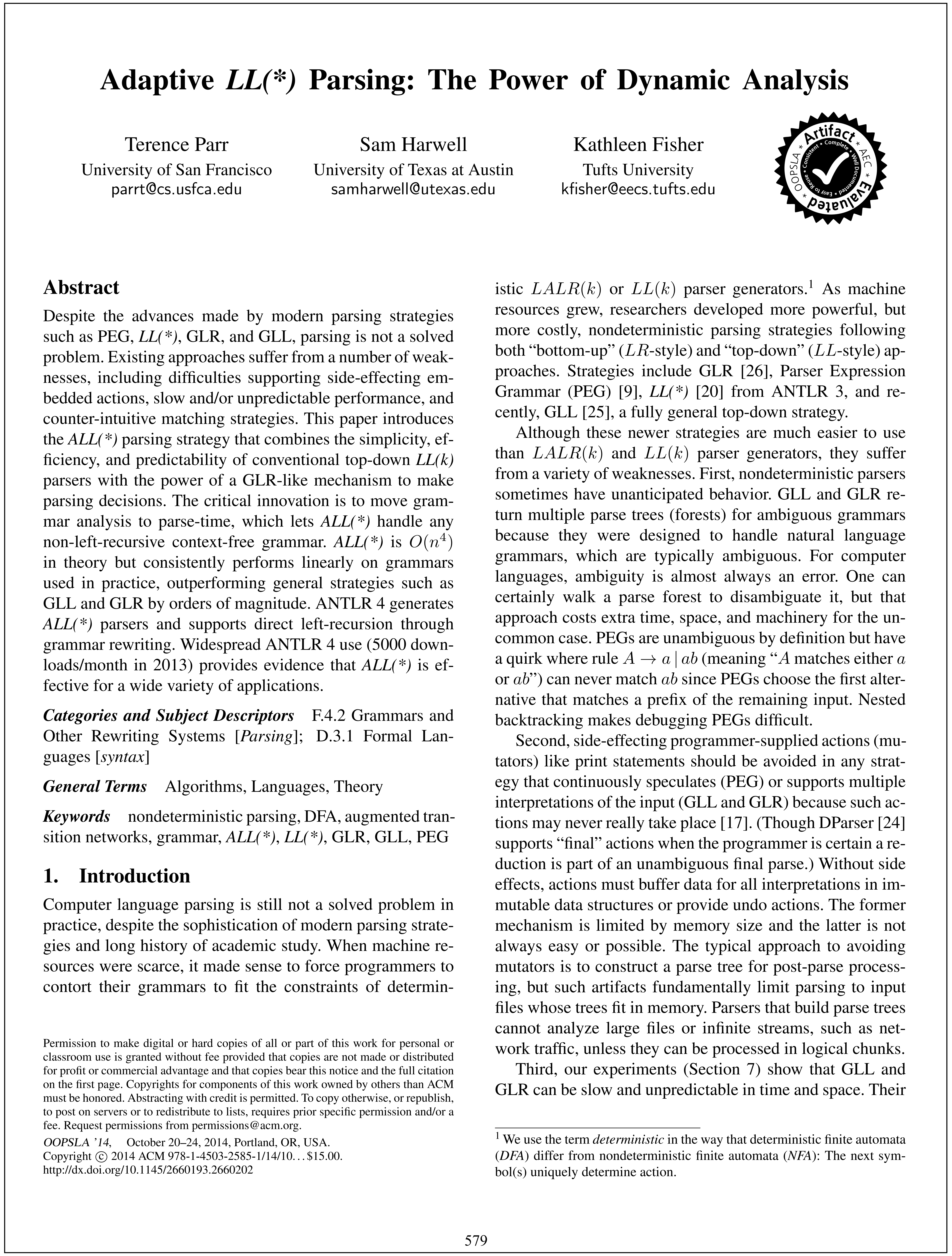
Eelco Visser, PhD thesis, University of Amsterdam.

https://eelcovisser.org/wiki/thesis

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

OOPSLA 2014

https://doi.org/10.1145/2714064.2660202



Adaptive *LL(*)* Parsing: The Power of Dynamic Analysis

Terence Parr
University of San Francisco
parrt@cs.usfca.edu

Sam Harwell
University of Texas at Austin
samharwell@utexas.edu

Kathleen Fisher
Tufts University
kfisher@eecs.tufts.edu

Abstract

Despite the advances made by modern parsing strategies such as PEG, *LL(*)*, GLR, and GLL, parsing is not a solved problem. Existing approaches suffer from a number of weaknesses, including difficulties supporting side-effecting embedded actions, slow and/or unpredictable performance, and counter-intuitive matching strategies. This paper introduces the *ALL(*)* parsing strategy that combines the simplicity, efficiency, and predictability of conventional top-down *LL(k)* parsers with the power of a GLR-like mechanism to make parsing decisions. The critical innovation is to move grammar analysis to parse-time, which lets *ALL(*)* handle any non-left-recursive context-free grammar. *ALL(*)* is $O(n^4)$ in theory but consistently performs linearly on grammars used in practice, outperforming general strategies such as GLL and GLR by orders of magnitude. ANTLR 4 generates *ALL(*)* parsers and supports direct left-recursion through grammar rewriting. Widespread ANTLR 4 use (5000 downloads/month in 2013) provides evidence that *ALL(*)* is effective for a wide variety of applications.

Categories and Subject Descriptors F.4.2 Grammars and Other Rewriting Systems [*Parsing*]; D.3.1 Formal Languages [*syntax*]

General Terms Algorithms, Languages, Theory

Keywords nondeterministic parsing, DFA, augmented transition networks, grammar, *ALL(*)*, *LL(*)*, GLR, GLL, PEG

1. Introduction

Computer language parsing is still not a solved problem in practice, despite the sophistication of modern parsing strategies and long history of academic study. When machine resources were scarce, it made sense to force programmers to contort their grammars to fit the constraints of deterministic $LALR(k)$ or $LL(k)$ parser generators.[1] As machine resources grew, researchers developed more powerful, but more costly, nondeterministic parsing strategies following both "bottom-up" (LR-style) and "top-down" (LL-style) approaches. Strategies include GLR [26], Parser Expression Grammar (PEG) [9], *LL(*)* [20] from ANTLR 3, and recently, GLL [25], a fully general top-down strategy.

Although these newer strategies are much easier to use than $LALR(k)$ and $LL(k)$ parser generators, they suffer from a variety of weaknesses. First, nondeterministic parsers sometimes have unanticipated behavior. GLL and GLR return multiple parse trees (forests) for ambiguous grammars because they were designed to handle natural language grammars, which are typically ambiguous. For computer languages, ambiguity is almost always an error. One can certainly walk a parse forest to disambiguate it, but that approach costs extra time, space, and machinery for the uncommon case. PEGs are unambiguous by definition but have a quirk where rule $A \rightarrow a \mid ab$ (meaning "A matches either a or ab") can never match ab since PEGs choose the first alternative that matches a prefix of the remaining input. Nested backtracking makes debugging PEGs difficult.

Second, side-effecting programmer-supplied actions (mutators) like print statements should be avoided in any strategy that continuously speculates (PEG) or supports multiple interpretations of the input (GLL and GLR) because such actions may never really take place [17]. (Though DParser [24] supports "final" actions when the programmer is certain a reduction is part of an unambiguous final parse.) Without side effects, actions must buffer data for all interpretations in immutable data structures or provide undo actions. The former mechanism is limited by memory size and the latter is not always easy or possible. The typical approach to avoiding mutators is to construct a parse tree for post-parse processing, but such artifacts fundamentally limit parsing to input files whose trees fit in memory. Parsers that build parse trees cannot analyze large files or infinite streams, such as network traffic, unless they can be processed in logical chunks.

Third, our experiments (Section 7) show that GLL and GLR can be slow and unpredictable in time and space. Their

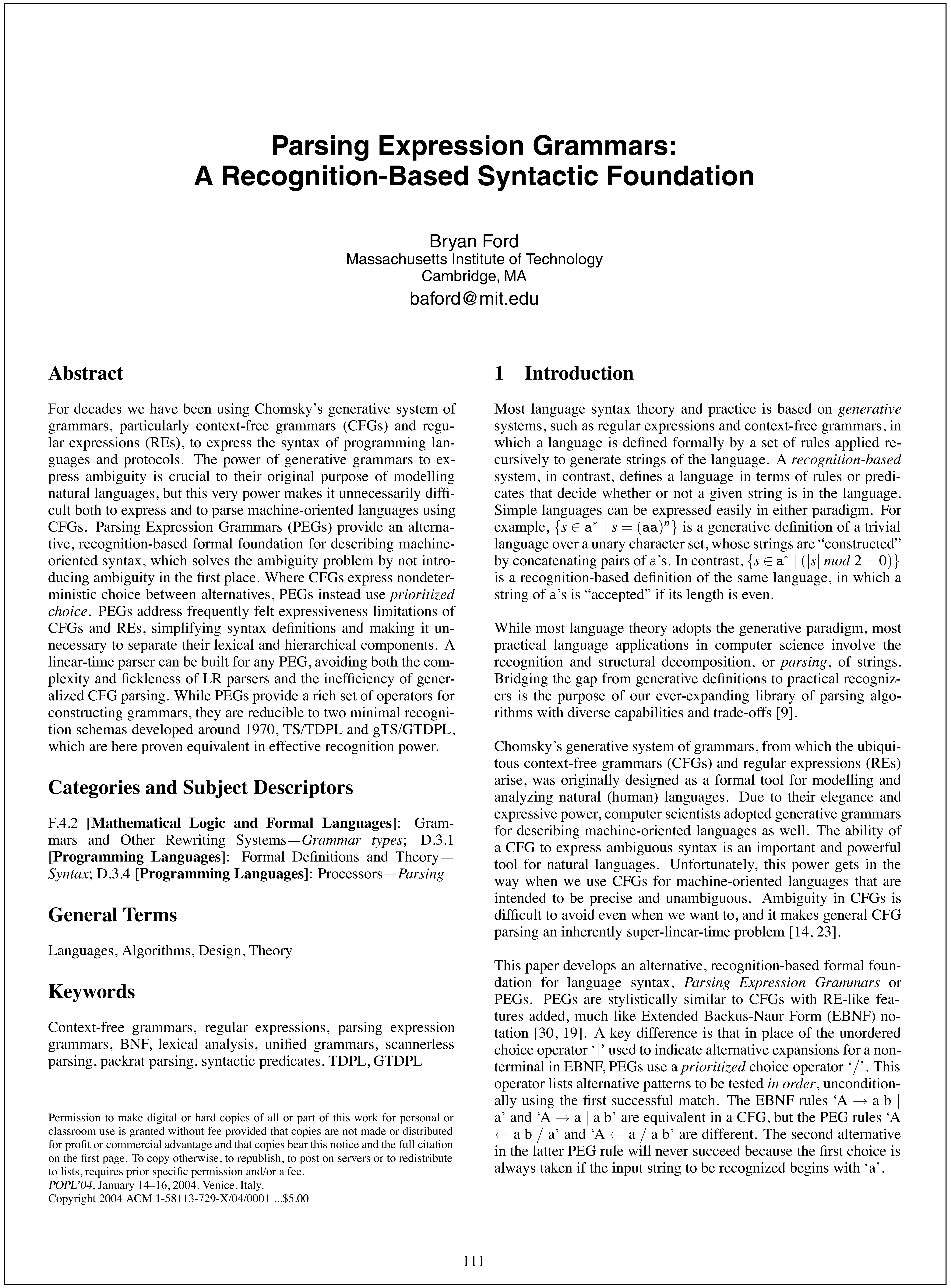
OOPSLA '14, October 20–24, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-2585-1/14/10…$15.00.
http://dx.doi.org/10.1145/2660193.2660202

[1] We use the term *deterministic* in the way that deterministic finite automata (*DFA*) differ from nondeterministic finite automata (*NFA*): The next symbol(s) uniquely determine action.

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

POPL 2004

https://doi.org/10.1145/982962.964011



Parsing Expression Grammars: A Recognition-Based Syntactic Foundation

Bryan Ford
Massachusetts Institute of Technology
Cambridge, MA
baford@mit.edu

Abstract

For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), to express the syntax of programming languages and protocols. The power of generative grammars to express ambiguity is crucial to their original purpose of modelling natural languages, but this very power makes it unnecessarily difficult both to express and to parse machine-oriented languages using CFGs. Parsing Expression Grammars (PEGs) provide an alternative, recognition-based formal foundation for describing machine-oriented syntax, which solves the ambiguity problem by not introducing ambiguity in the first place. Where CFGs express nondeterministic choice between alternatives, PEGs instead use *prioritized choice*. PEGs address frequently felt expressiveness limitations of CFGs and REs, simplifying syntax definitions and making it unnecessary to separate their lexical and hierarchical components. A linear-time parser can be built for any PEG, avoiding both the complexity and fickleness of LR parsers and the inefficiency of generalized CFG parsing. While PEGs provide a rich set of operators for constructing grammars, they are reducible to two minimal recognition schemas developed around 1970, TS/TDPL and gTS/GTDPL, which are here proven equivalent in effective recognition power.

Categories and Subject Descriptors

F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—*Grammar types*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax*; D.3.4 [**Programming Languages**]: Processors—*Parsing*

General Terms

Languages, Algorithms, Design, Theory

Keywords

Context-free grammars, regular expressions, parsing expression grammars, BNF, lexical analysis, unified grammars, scannerless parsing, packrat parsing, syntactic predicates, TDPL, GTDPL

1 Introduction

Most language syntax theory and practice is based on *generative* systems, such as regular expressions and context-free grammars, in which a language is defined formally by a set of rules applied recursively to generate strings of the language. A *recognition-based* system, in contrast, defines a language in terms of rules or predicates that decide whether or not a given string is in the language. Simple languages can be expressed easily in either paradigm. For example, $\{s \in \mathtt{a}^* \mid s = (\mathtt{aa})^n\}$ is a generative definition of a trivial language over a unary character set, whose strings are "constructed" by concatenating pairs of a's. In contrast, $\{s \in \mathtt{a}^* \mid (|s| \; mod \; 2 = 0)\}$ is a recognition-based definition of the same language, in which a string of a's is "accepted" if its length is even.

While most language theory adopts the generative paradigm, most practical language applications in computer science involve the recognition and structural decomposition, or *parsing*, of strings. Bridging the gap from generative definitions to practical recognizers is the purpose of our ever-expanding library of parsing algorithms with diverse capabilities and trade-offs [9].

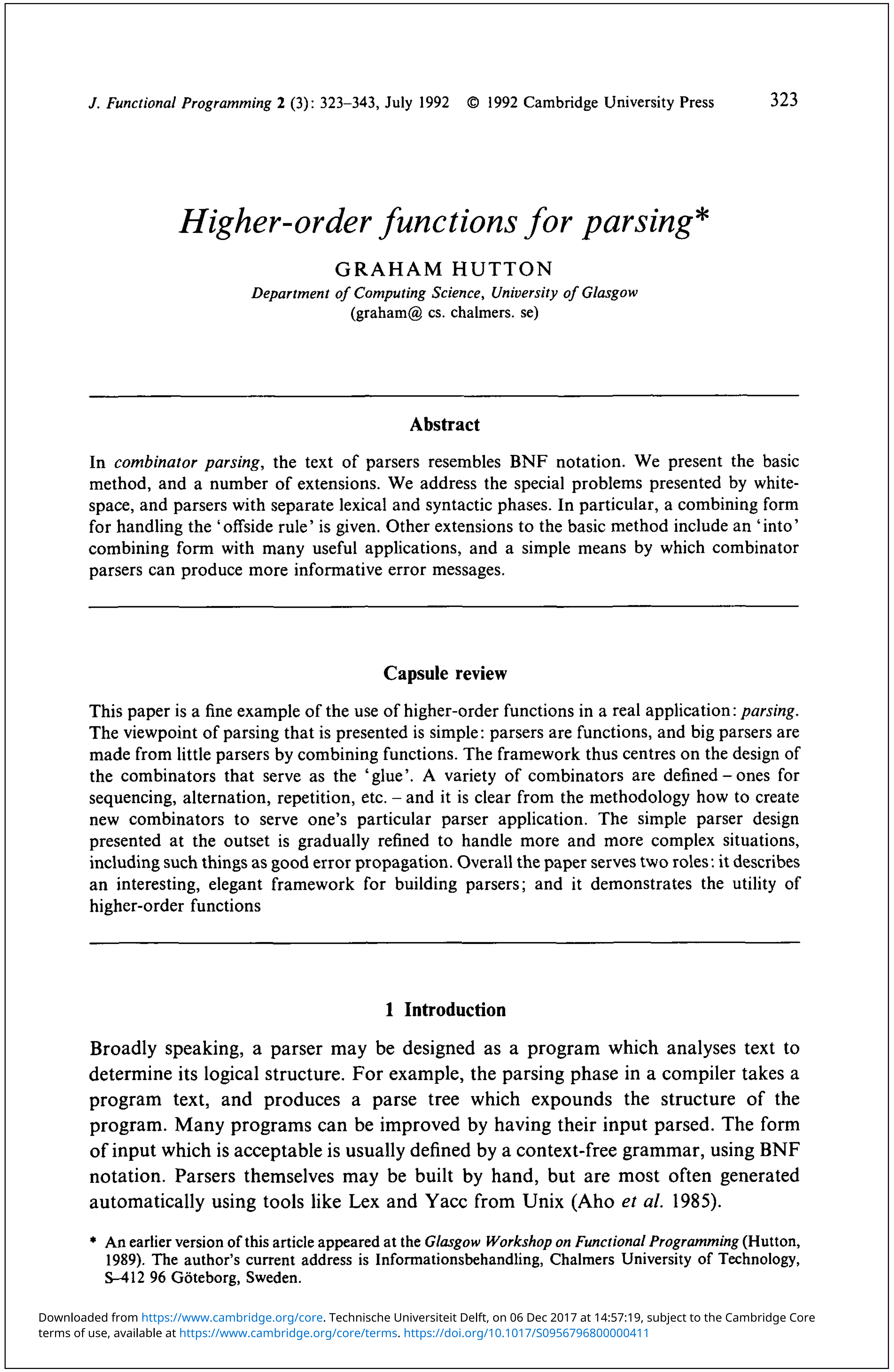
Chomsky's generative system of grammars, from which the ubiquitous context-free grammars (CFGs) and regular expressions (REs) arise, was originally designed as a formal tool for modelling and analyzing natural (human) languages. Due to their elegance and expressive power, computer scientists adopted generative grammars for describing machine-oriented languages as well. The ability of a CFG to express ambiguous syntax is an important and powerful tool for natural languages. Unfortunately, this power gets in the way when we use CFGs for machine-oriented languages that are intended to be precise and unambiguous. Ambiguity in CFGs is difficult to avoid even when we want to, and it makes general CFG parsing an inherently super-linear-time problem [14, 23].

This paper develops an alternative, recognition-based formal foundation for language syntax, *Parsing Expression Grammars* or PEGs. PEGs are stylistically similar to CFGs with RE-like features added, much like Extended Backus-Naur Form (EBNF) notation [30, 19]. A key difference is that in place of the unordered choice operator '|' used to indicate alternative expansions for a nonterminal in EBNF, PEGs use a *prioritized* choice operator '/'. This operator lists alternative patterns to be tested *in order*, unconditionally using the first successful match. The EBNF rules '$\mathrm{A} \rightarrow \mathrm{a\ b} \mid \mathrm{a}$' and '$\mathrm{A} \rightarrow \mathrm{a} \mid \mathrm{a\ b}$' are equivalent in a CFG, but the PEG rules '$\mathrm{A} \leftarrow \mathrm{a\ b} \,/\, \mathrm{a}$' and '$\mathrm{A} \leftarrow \mathrm{a} \,/\, \mathrm{a\ b}$' are different. The second alternative in the latter PEG rule will never succeed because the first choice is always taken if the input string to be recognized begins with 'a'.

Graham Hutton. Higher-Order Functions for Parsing.

Journal of Functional Programming

https://doi.org/10.1017/S0956796800000411



J. Functional Programming **2** (3): 323–343, July 1992 © 1992 Cambridge University Press 323

*Higher-order functions for parsing**

GRAHAM HUTTON
Department of Computing Science, University of Glasgow
(graham@ cs. chalmers. se)

Abstract

In *combinator parsing*, the text of parsers resembles BNF notation. We present the basic method, and a number of extensions. We address the special problems presented by white-space, and parsers with separate lexical and syntactic phases. In particular, a combining form for handling the 'offside rule' is given. Other extensions to the basic method include an 'into' combining form with many useful applications, and a simple means by which combinator parsers can produce more informative error messages.

Capsule review

This paper is a fine example of the use of higher-order functions in a real application: *parsing*. The viewpoint of parsing that is presented is simple: parsers are functions, and big parsers are made from little parsers by combining functions. The framework thus centres on the design of the combinators that serve as the 'glue'. A variety of combinators are defined – ones for sequencing, alternation, repetition, etc. – and it is clear from the methodology how to create new combinators to serve one's particular parser application. The simple parser design presented at the outset is gradually refined to handle more and more complex situations, including such things as good error propagation. Overall the paper serves two roles: it describes an interesting, elegant framework for building parsers; and it demonstrates the utility of higher-order functions

1 Introduction

Broadly speaking, a parser may be designed as a program which analyses text to determine its logical structure. For example, the parsing phase in a compiler takes a program text, and produces a parse tree which expounds the structure of the program. Many programs can be improved by having their input parsed. The form of input which is acceptable is usually defined by a context-free grammar, using BNF notation. Parsers themselves may be built by hand, but are most often generated automatically using tools like Lex and Yacc from Unix (Aho *et al.* 1985).

* An earlier version of this article appeared at the *Glasgow Workshop on Functional Programming* (Hutton, 1989). The author's current address is Informationsbehandling, Chalmers University of Technology, S-412 96 Göteborg, Sweden.

LR Parse Tables

Parse Table

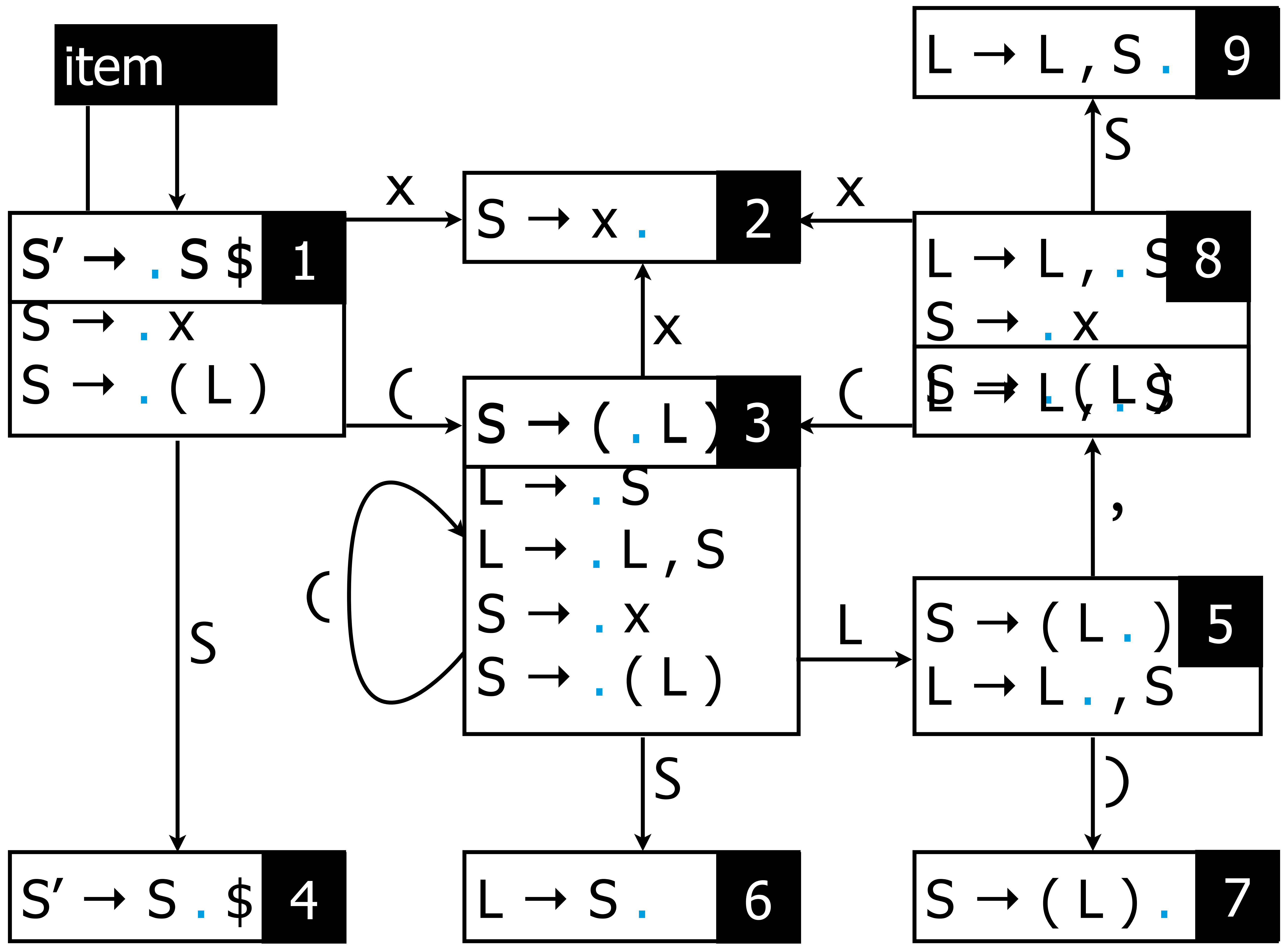
rows

- states of a DFA

- columns
  - topmost stack symbol
  - Σ, N
- entries
  - reduce, rule number
  - shift, goto state
  - goto state
  - accept state

	T_1	…	N_1	…
1	s 3			
2			g 5	
3	r 1			
4	r 2	a		
5				
6			g 1	
7	s 1			
8				
…				

Items, Closure & Goto



S → x
S → (L)
L → S
L → L , S

Final Table

(	x	,	x	)	$

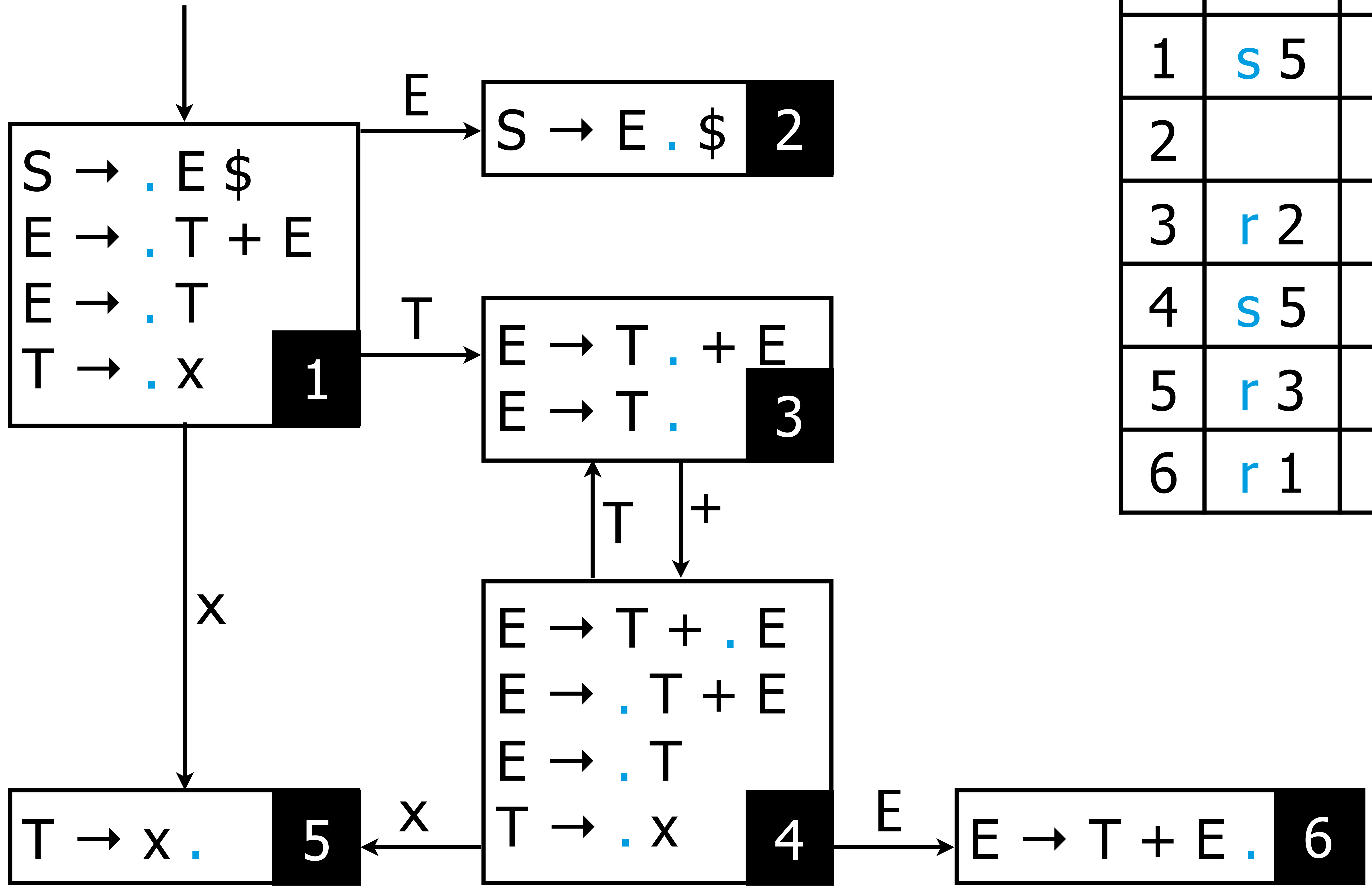
	(	)	x	,	$	S	L
1	s 3		s 2			g 4	
2	r 1	r 1	r 1	r 1	r 1		
3	s 3		s 2			g 6	g 5
4					a		
5		s 7		s 8			
6	r 3	r 3	r 3	r 3	r 3		
7	r 2	r 2	r 2	r 2	r 2		
8	s 3		s 2			g 9	
9	r 4	r 4	r 4	r 4	r 4		

2 9
7 8
5 6
3 4
1

S → x
S → (L)
L → S
L → L , S

Conflict Resolution

Shift-Reduce Conflicts



	x	+	$	E	T
1	s 5			g 2	g 3
2			a		
3	r 2	?	r 2		
4	s 5			g 6	g 3
5	r 3	r 3	r 3		
6	r 1	r 1	r 1		

E → T + E
E → T
T → x

Nullable, First and Follow Sets

$T \in \boxed{\text{FIRST}(w)}$ letters that w can start with

$\boxed{\text{nullable}(w)} \land T \in \boxed{\text{FOLLOW}(X)}$ letters that can follow X

$w \Rightarrow_G^* \varepsilon$

Nullable Set

nullable(X)

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

$(X_0, X_1 \dots X_k) \in P \wedge \text{nullable}(X_1) \wedge \dots \wedge \text{nullable}(X_k) \Rightarrow \text{nullable}(X_0)$

nullable(w)

$\text{nullable}(\varepsilon)$

$\text{nullable}(X_1 \dots X_k) = \text{nullable}(X_1) \wedge \dots \wedge \text{nullable}(X_k)$

First Set

$\mathrm{FIRST}(X)$

$X \in \Sigma : \mathrm{FIRST}(X) = \{X\}$

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

$\mathrm{FIRST}(w)$

$\mathrm{FIRST}(\varepsilon) = \{\}$

$\neg\mathrm{nullable}(X) \Rightarrow \mathrm{FIRST}(Xw) = \mathrm{FIRST}(X)$

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

Follow Set

FOLLOW(X)

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

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

Example

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

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

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

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

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

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

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

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

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		

Example

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

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

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		

Example

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

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

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 (	

Example

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

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

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

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 (	

Example

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

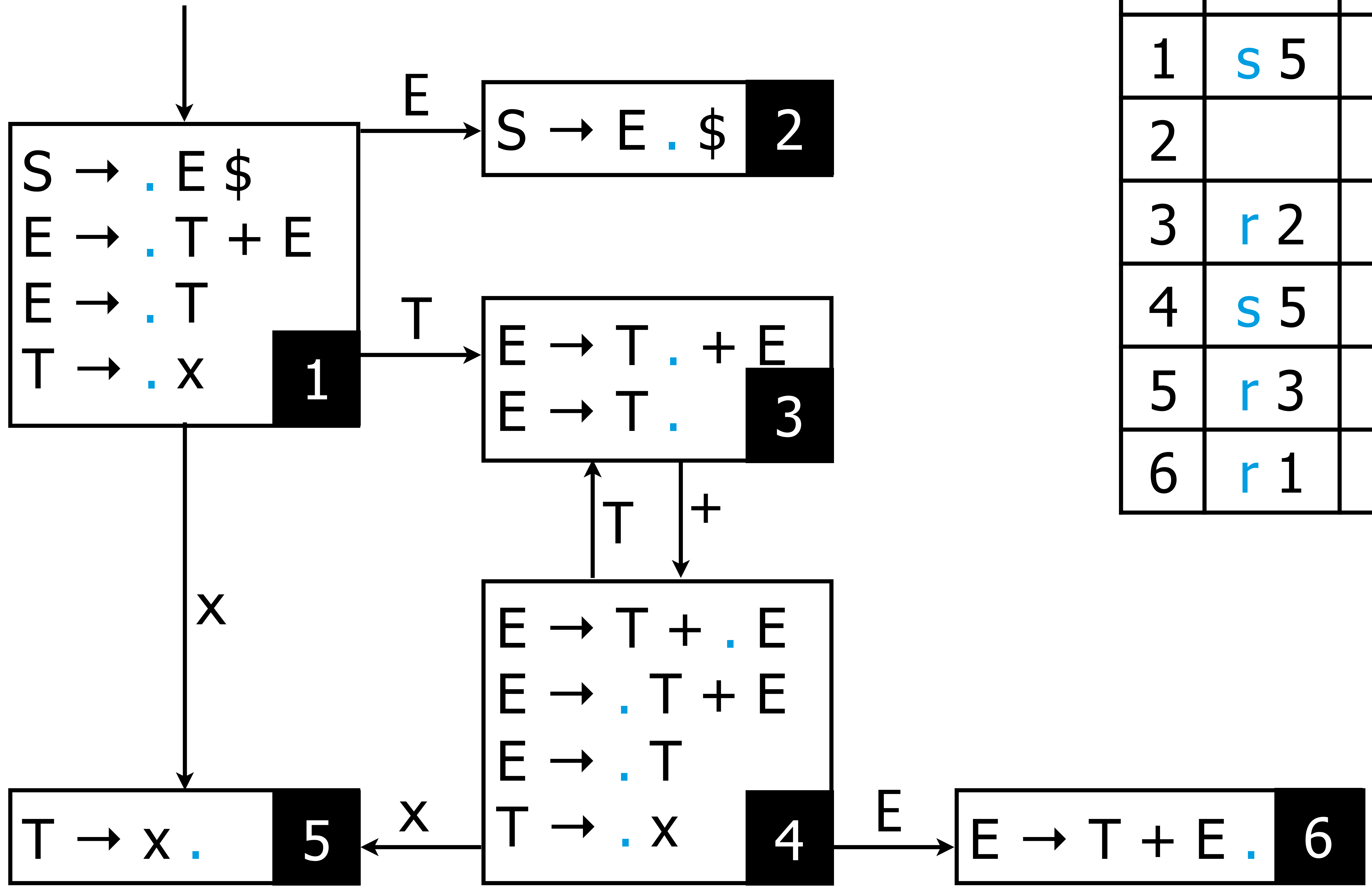
$\text{nullable}(X_{i+1} \ldots X_k) \Rightarrow \text{FOLLOW}(X_i) \supseteq \text{FOLLOW}(X_0)$

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

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

Shift-Reduce Conflicts



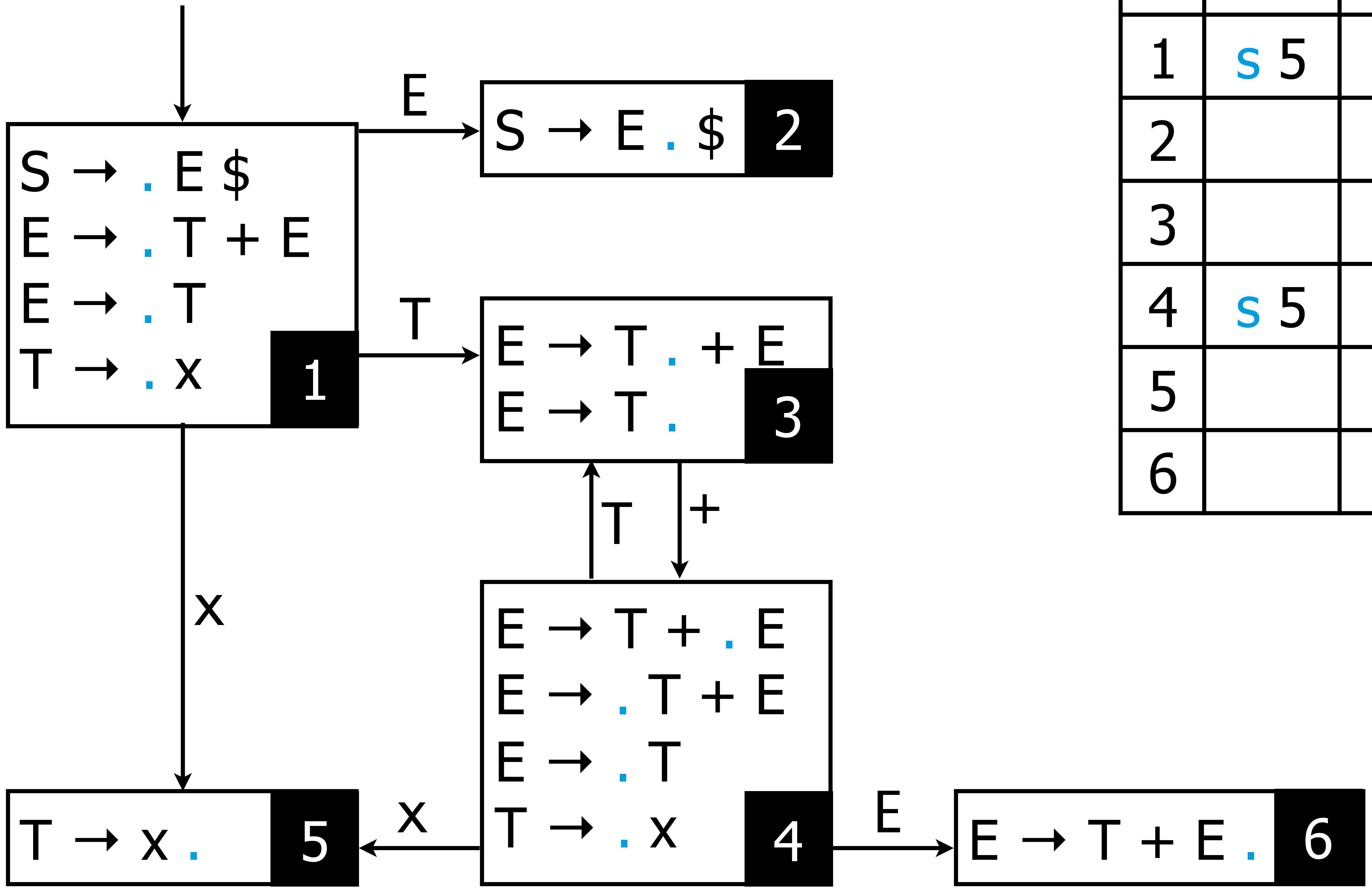
	x	+	$	E	T
1	s 5			g 2	g 3
2			a		
3	r 2	?	r 2		
4	s 5			g 6	g 3
5	r 3	r 3	r 3		
6	r 1	r 1	r 1		

E → T + E
E → T
T → x

SLR Parse Tables

Reduce a production S → …

on symbols k ∈ Σ, k ∈ Follow(S)

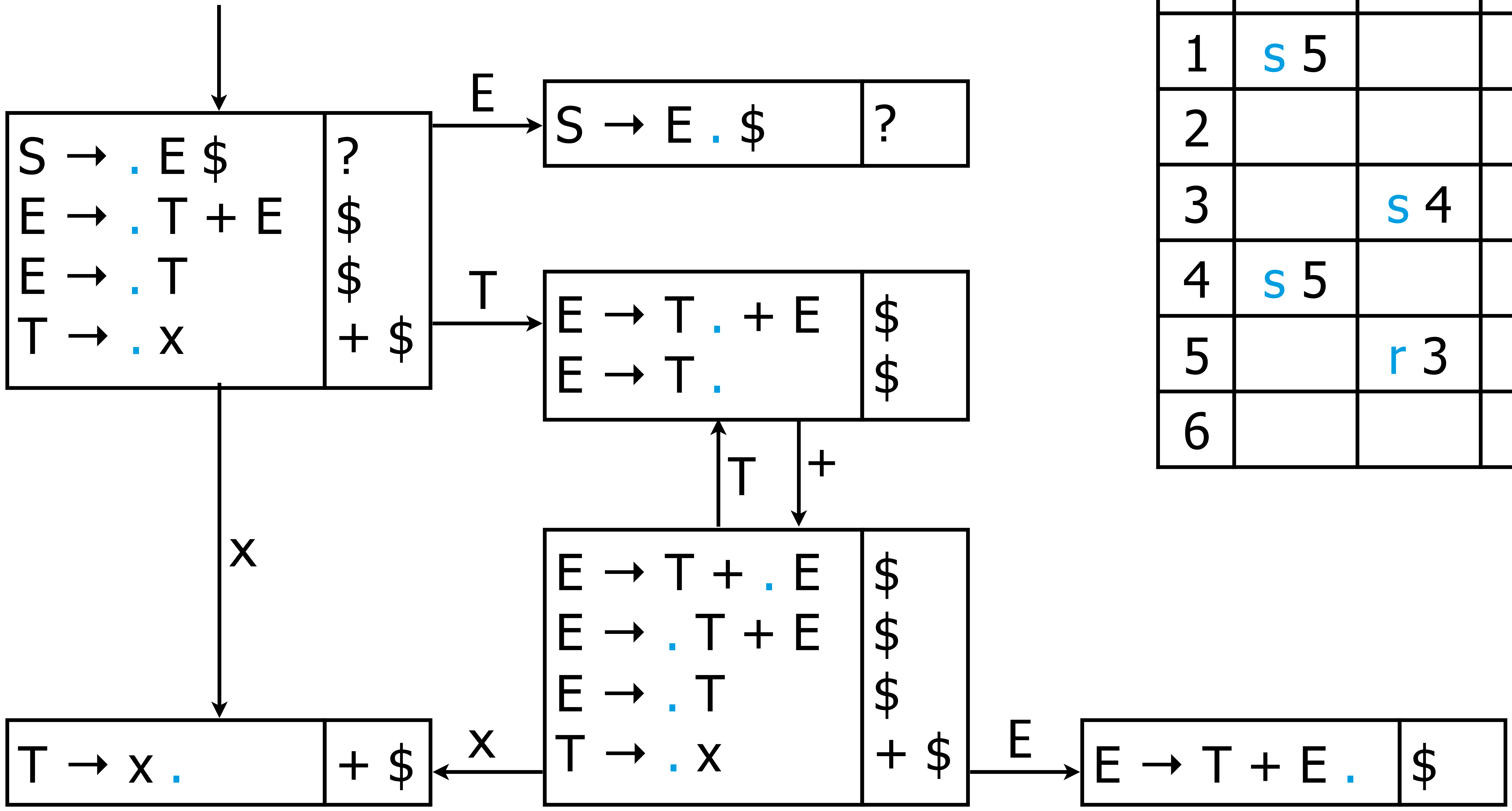


	x	+	$	E	T
1	s 5			g 2	g 3
2			a		
3		s 4	r 2		
4	s 5			g 6	g 3
5		r 3	r 3		
6			r 1		

E → T + E
E → T
T → x

LR(1) Parse Tables

- for every item A → α . X β, z
- for every rule X → γ
- for every w ∈ First(βz)
- add item X → . γ, w



	x	+	$	E	T
1	s 5			g 2	g 3
2			a		
3		s 4	r 2		
4	s 5			g 6	g 3
5		r 3	r 3		
6			r 1		

E → T + E
E → T
T → x

State-space reduction

unify states

- with same items
- and same outgoing transitions
- but different look-ahead sets

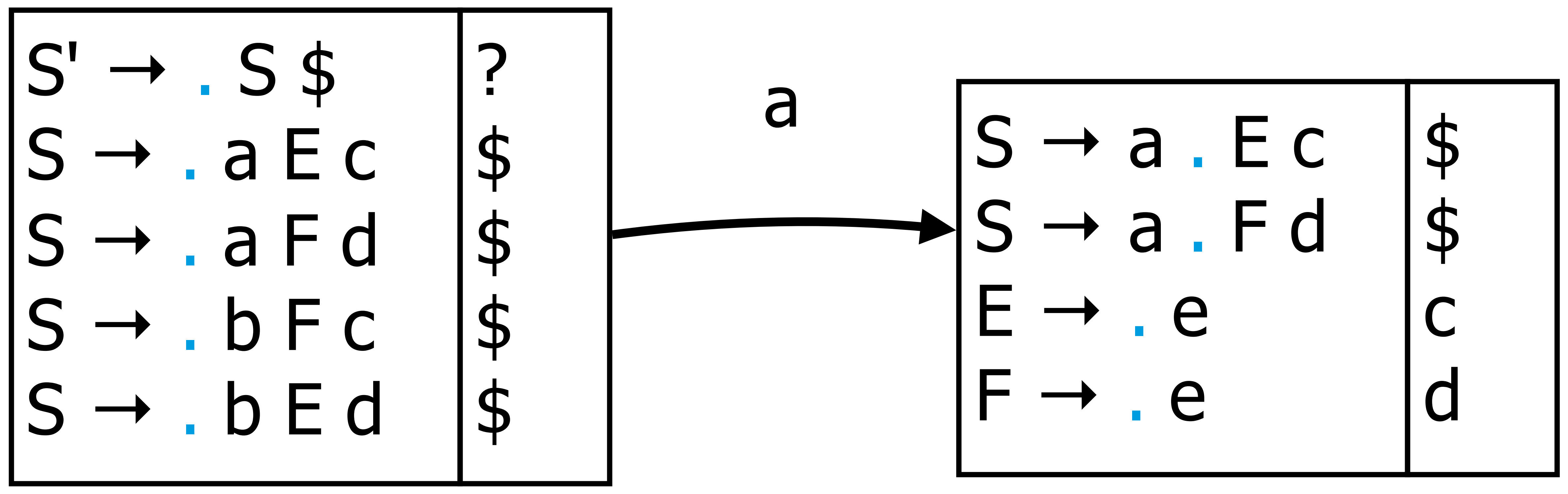
might introduce new conflicts

State-space reduction

S' → . S $	?
S → . a E c	$
S → . a F d	$
S → . b F c	$
S → . b E d	$

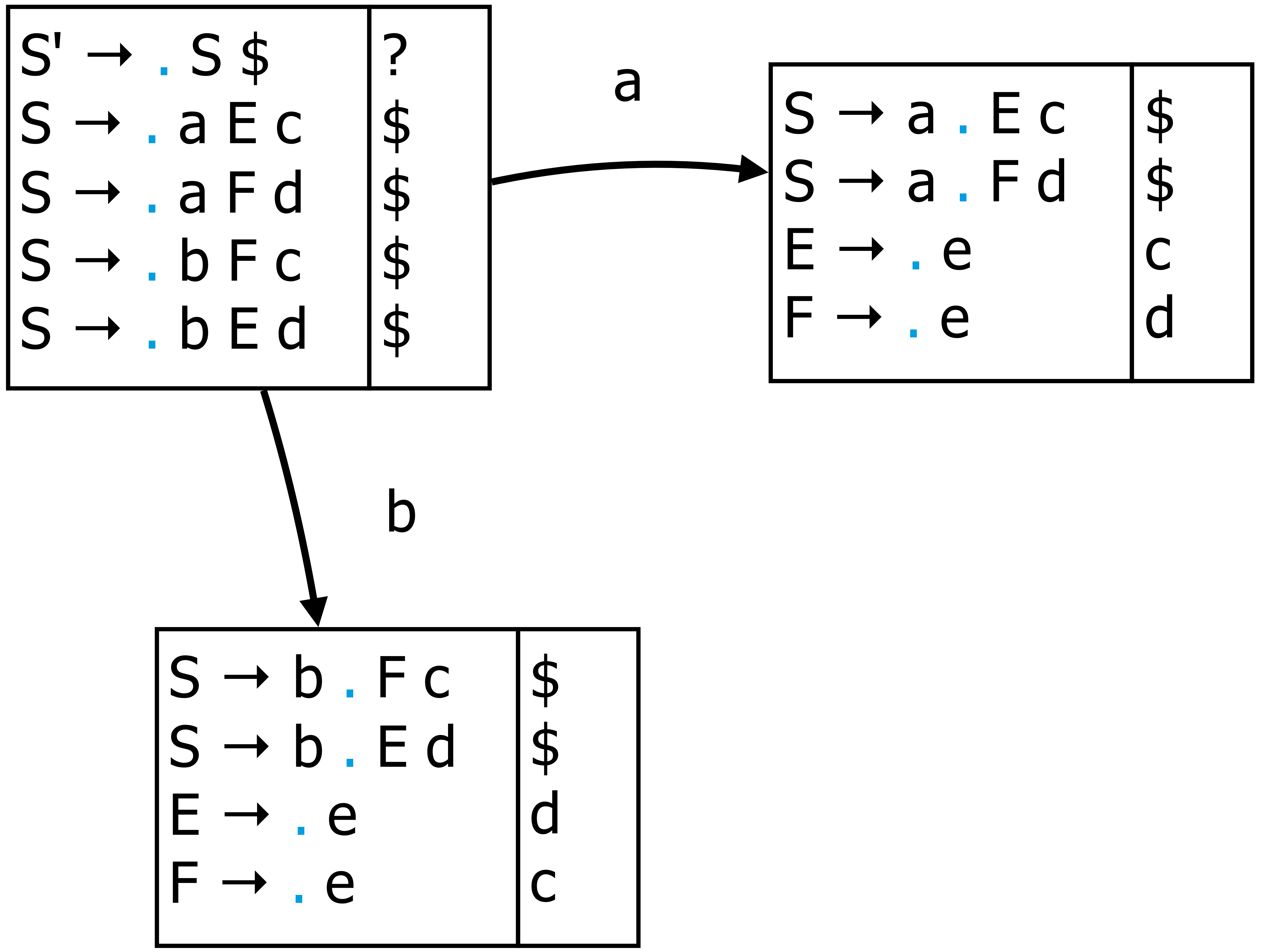
S’ → S $
S → a E c
S → a F d
S → b F c
S → b E d
E → e
F → e

State-space reduction



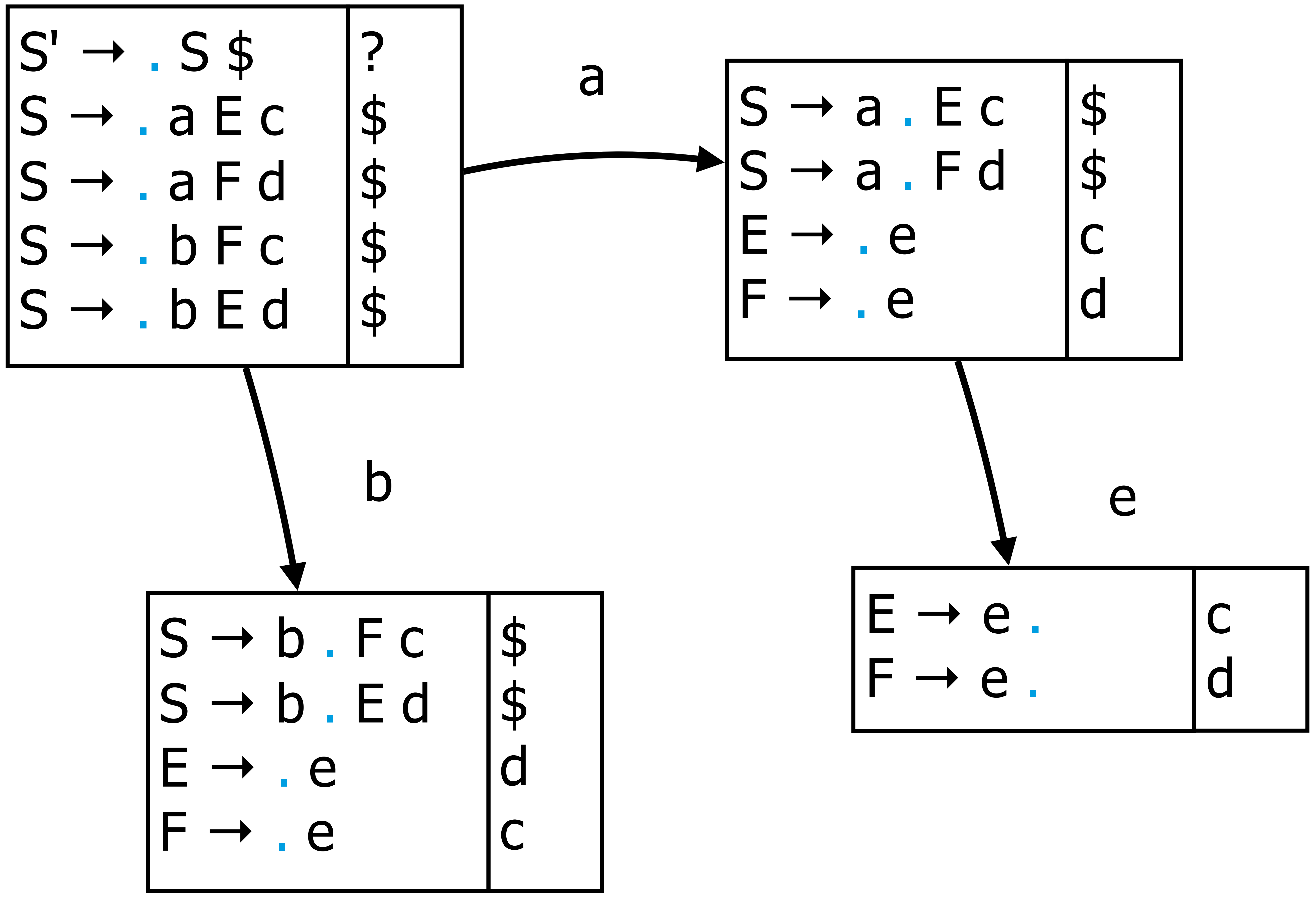
S′ → S $
S → a E c
S → a F d
S → b F c
S → b E d
E → e
F → e

State-space reduction



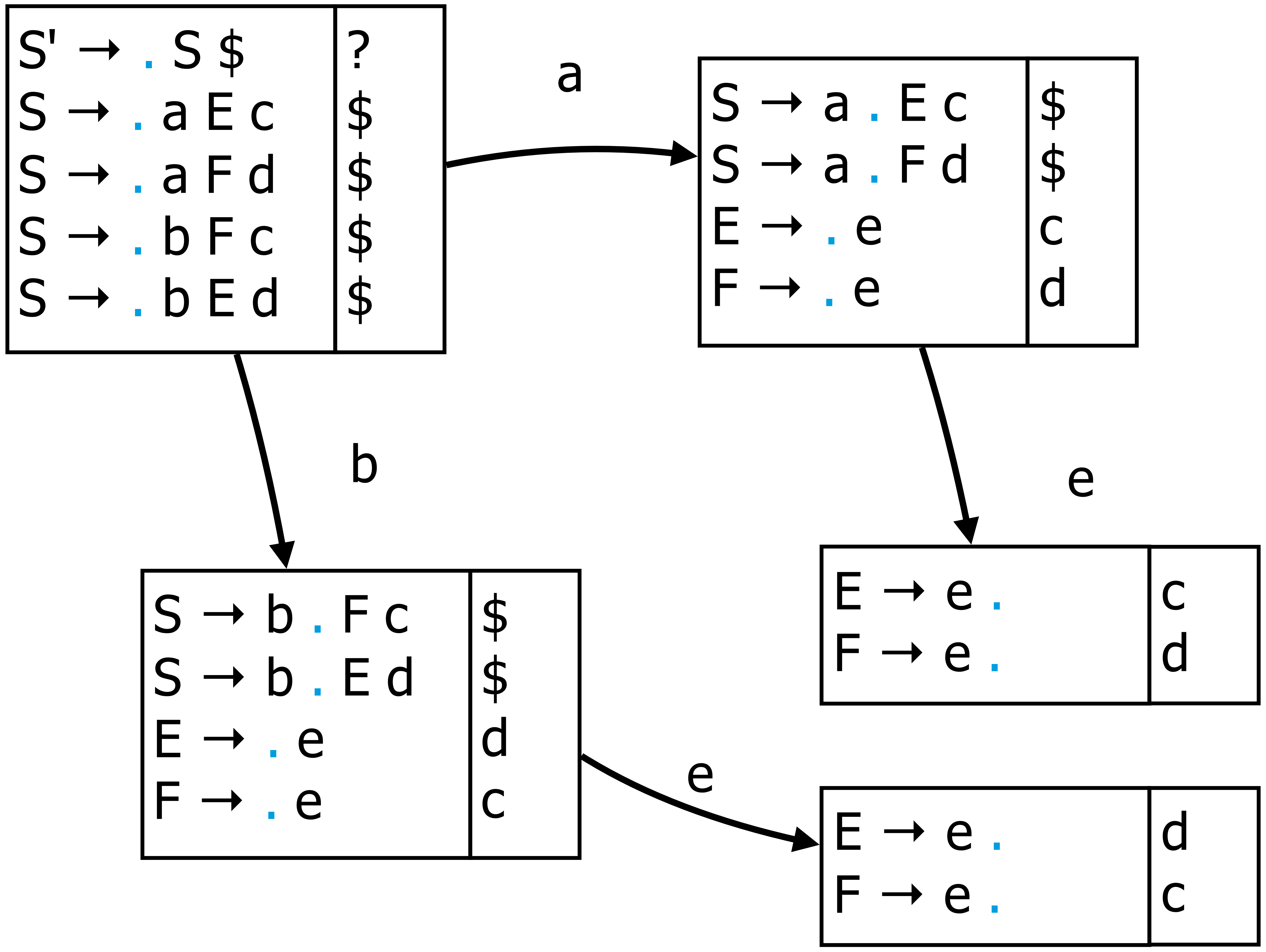
S′ → S $
S → a E c
S → a F d
S → b F c
S → b E d
E → e
F → e

State-space reduction



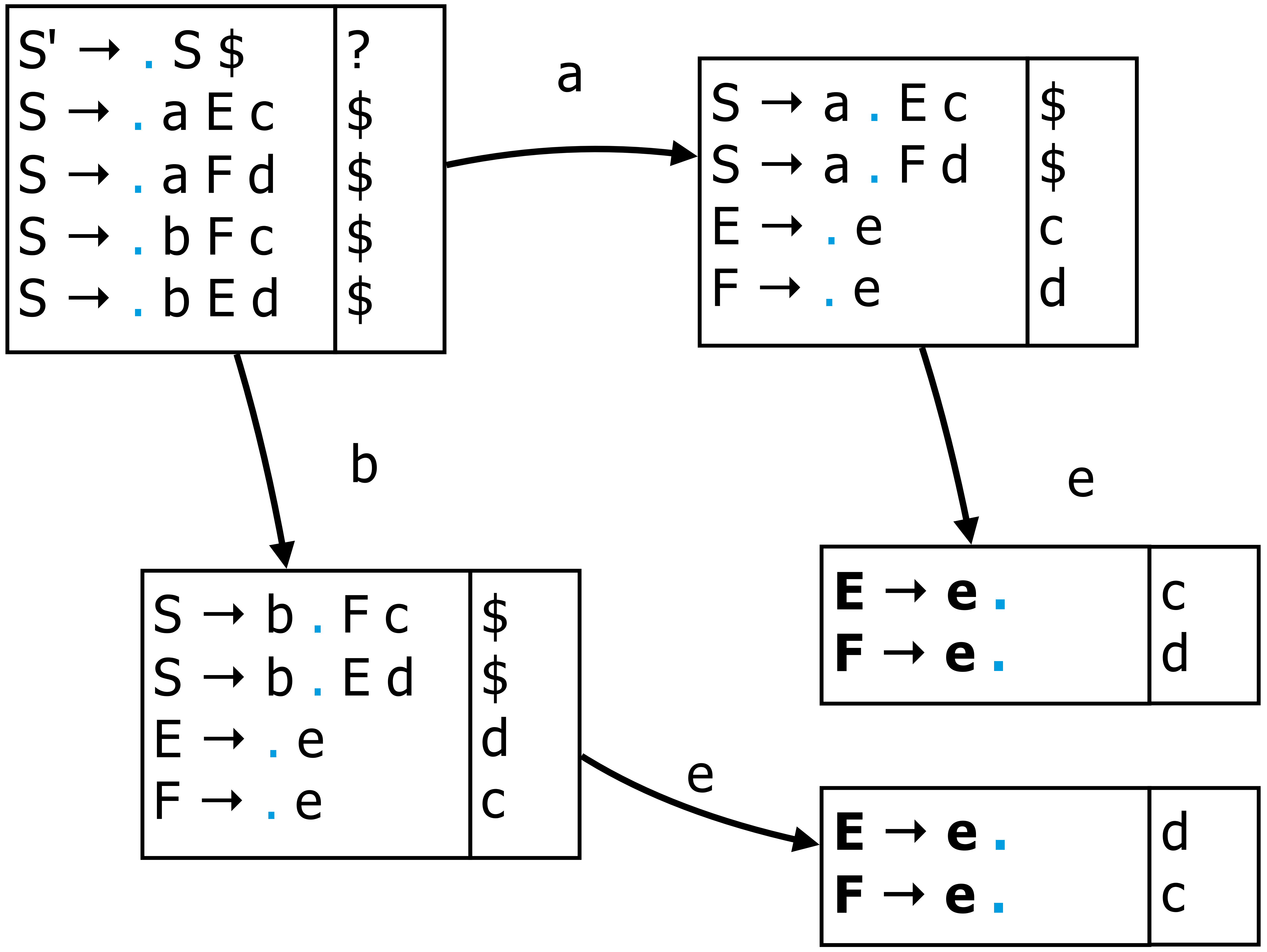
S’ → S $
S → a E c
S → a F d
S → b F c
S → b E d
E → e
F → e

State-space reduction



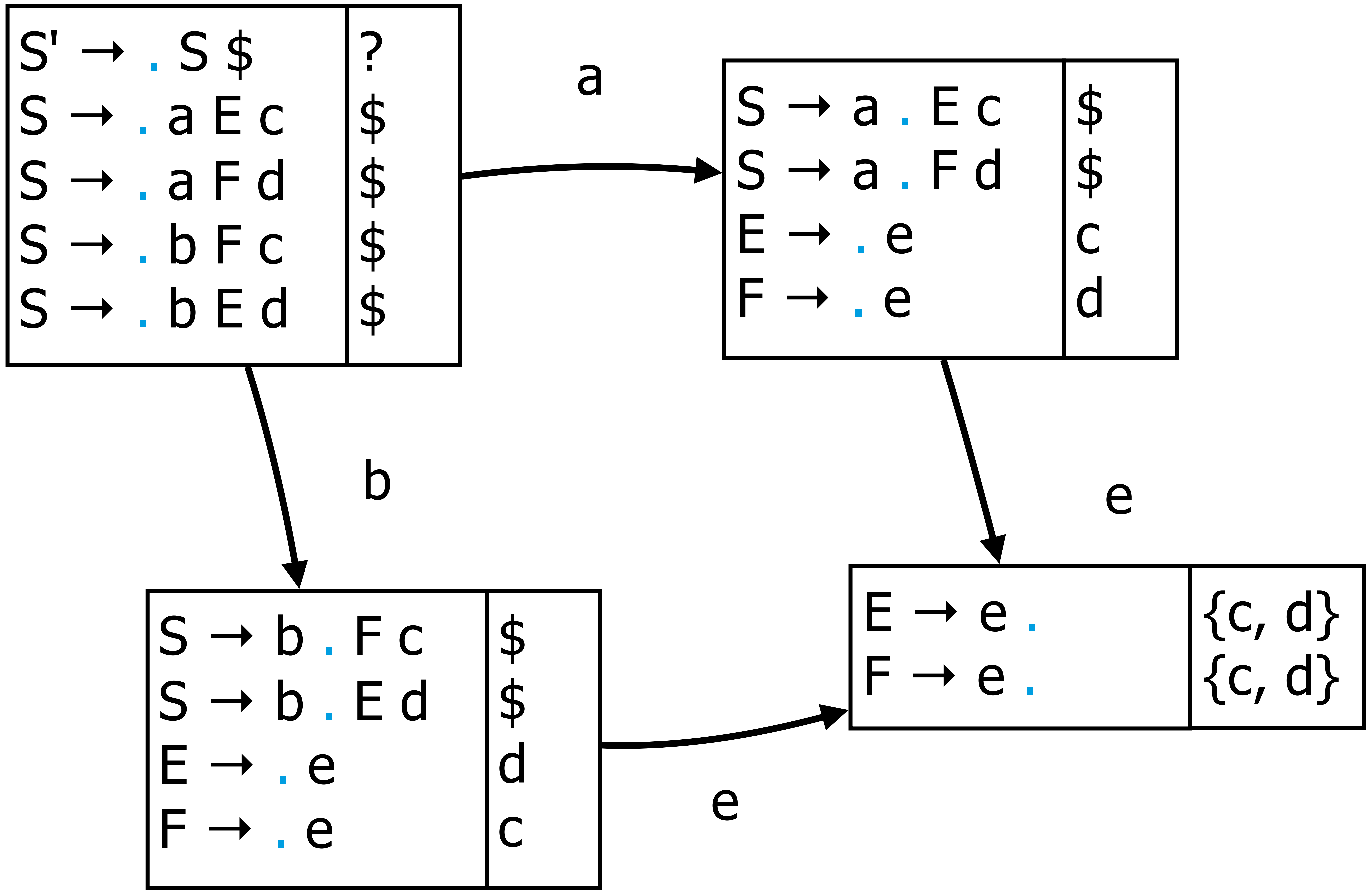
S′ → S $
S → a E c
S → a F d
S → b F c
S → b E d
E → e
F → e

State-space reduction



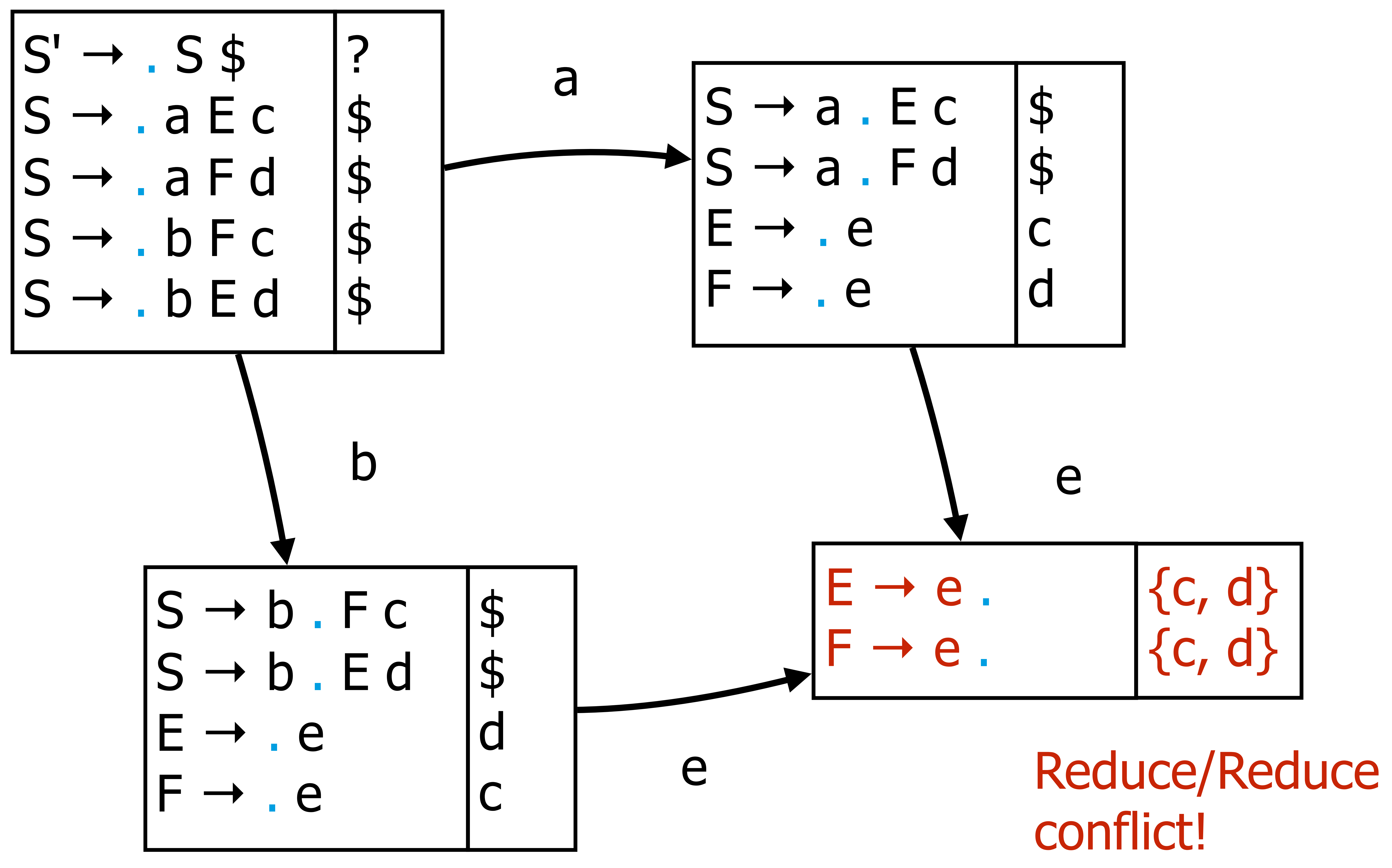
S' → S $
S → a E c
S → a F d
S → b F c
S → b E d
E → e
F → e

State-space reduction



S’ → S $
S → a E c
S → a F d
S → b F c
S → b E d
E → e
F → e

State-space reduction



S′ → S $
S → a E c
S → a F d
S → b F c
S → b E d
E → e
F → e

Generalized-LR Parsing

Generalized Parsing

- Parse all interpretations of the input, therefore it can 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.

SLR Table

S → E $
E → E + E
E → E * E
E → a

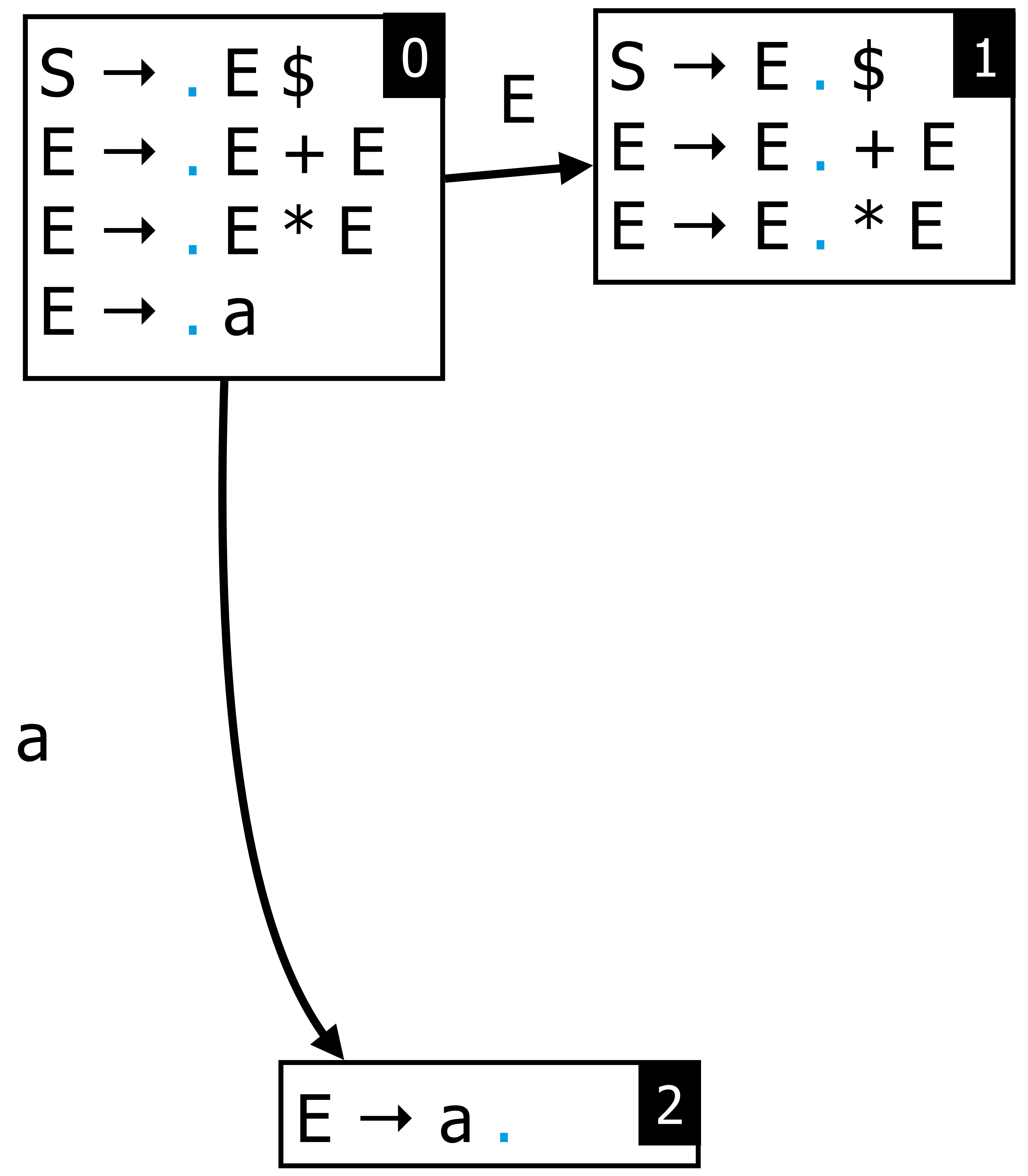
SLR Table

0

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

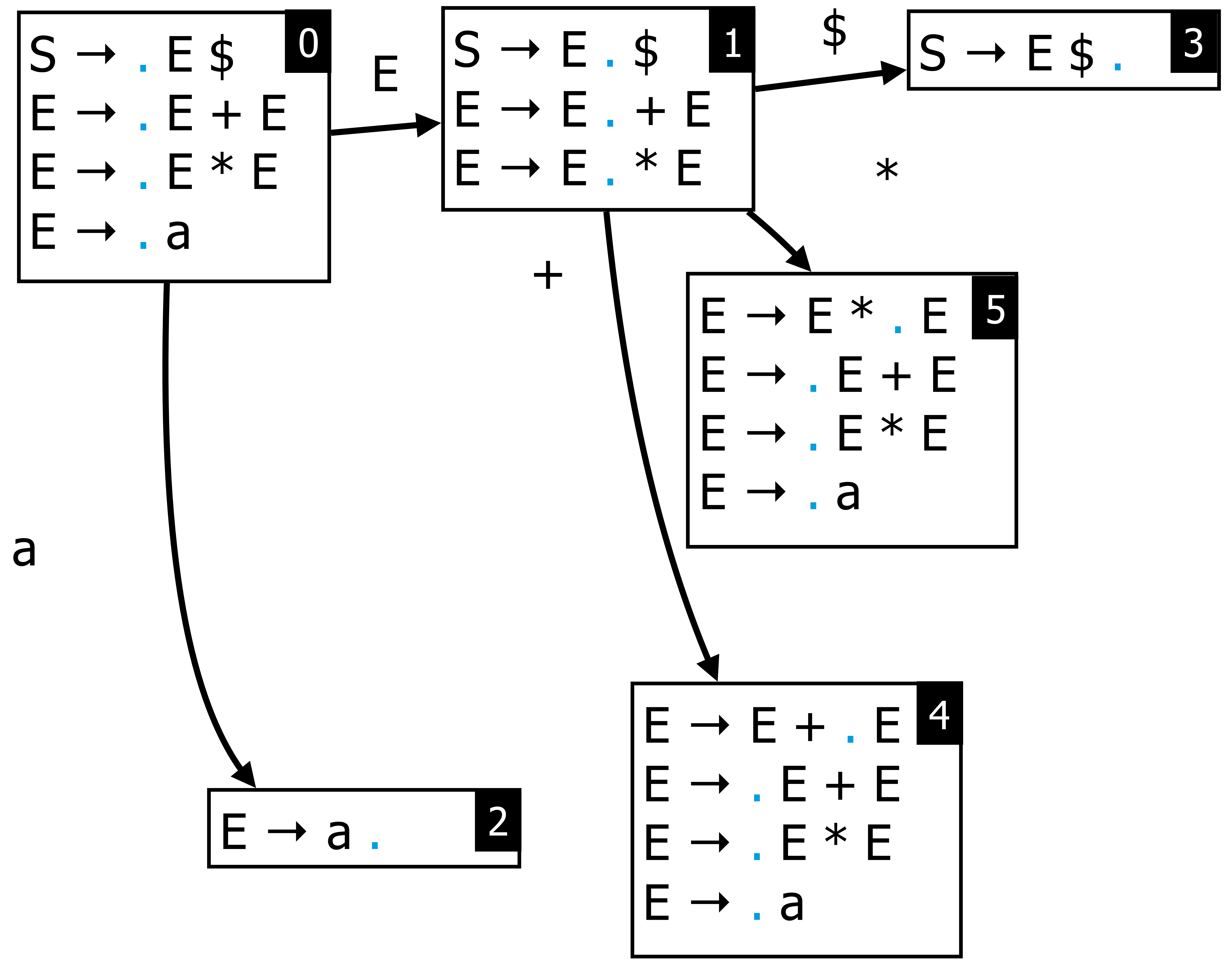
S → E $
E → E + E
E → E * E
E → a

SLR Table



S → E $
E → E + E
E → E * E
E → a

SLR Table



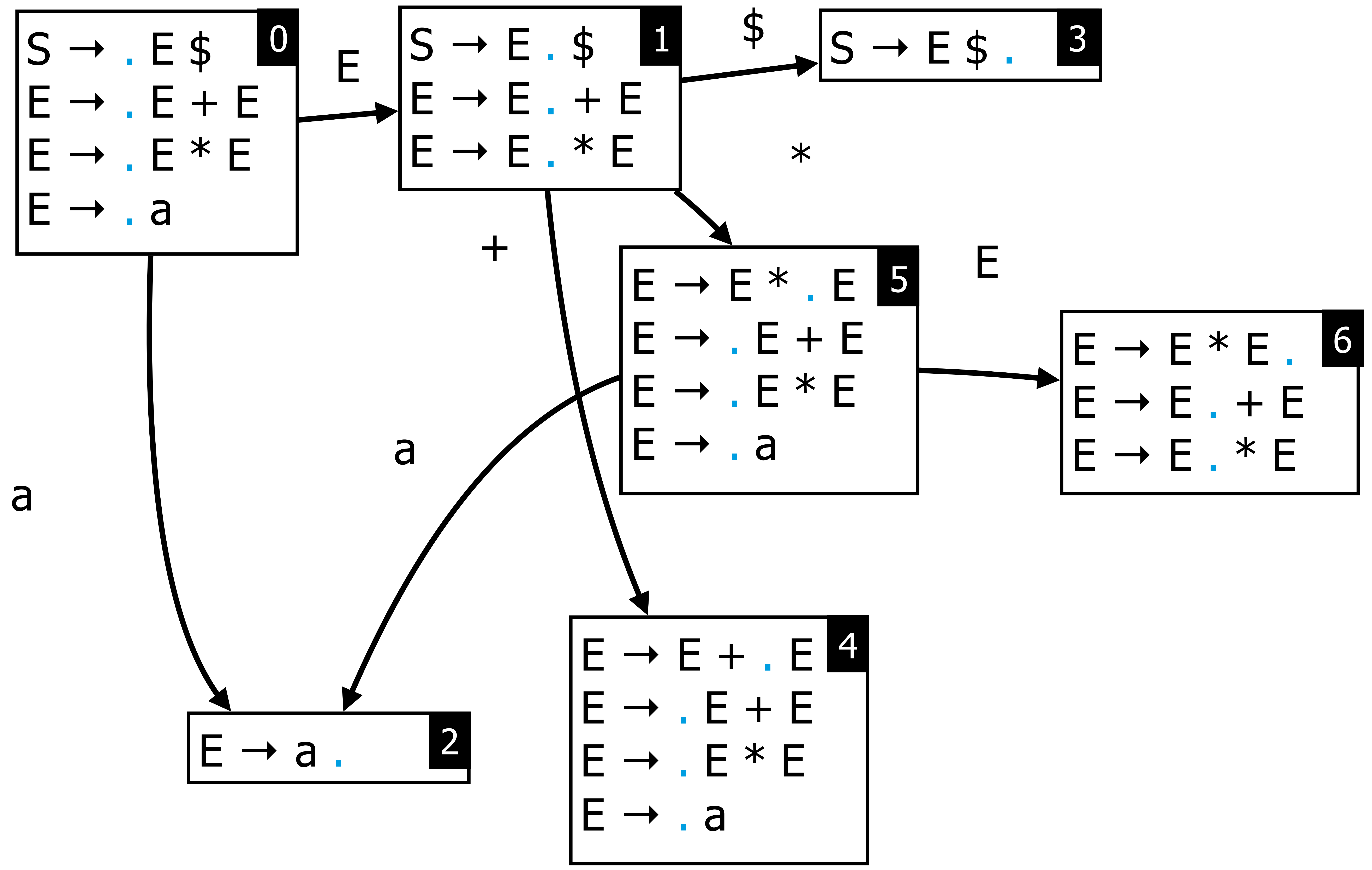
S → E $

E → E + E

E → E * E

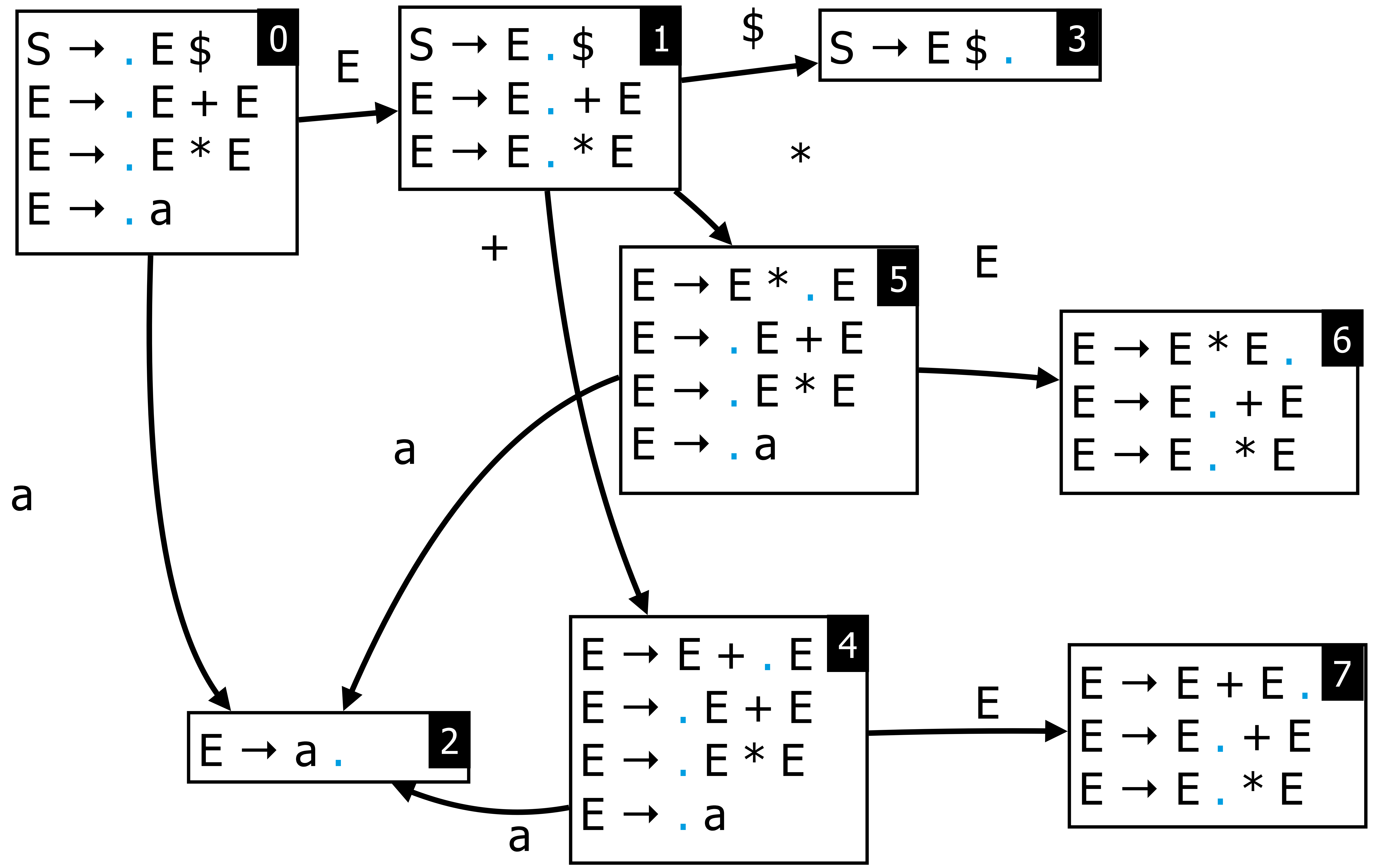
E → a

SLR Table



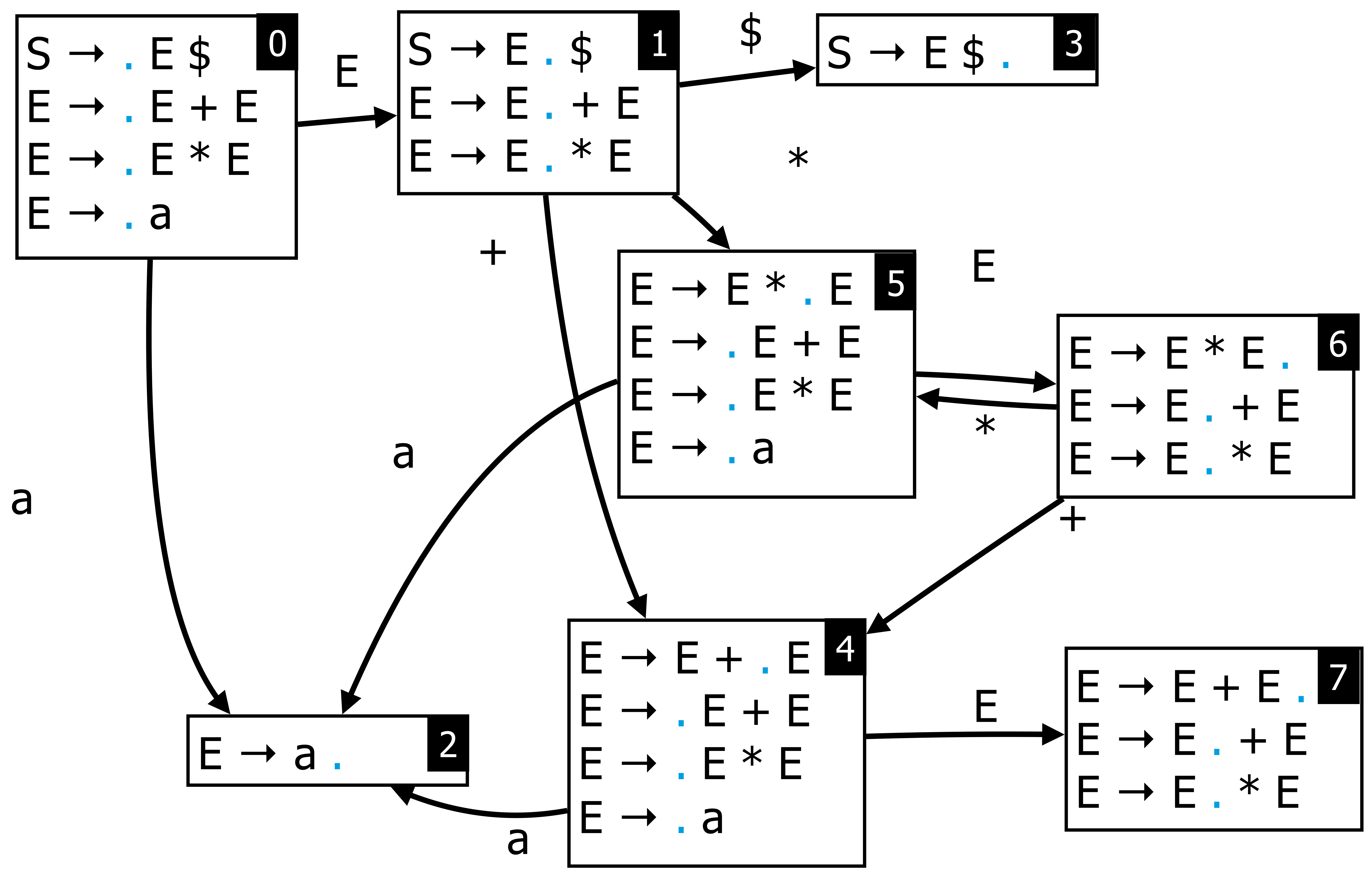
S → E $
E → E + E
E → E * E
E → a

SLR Table



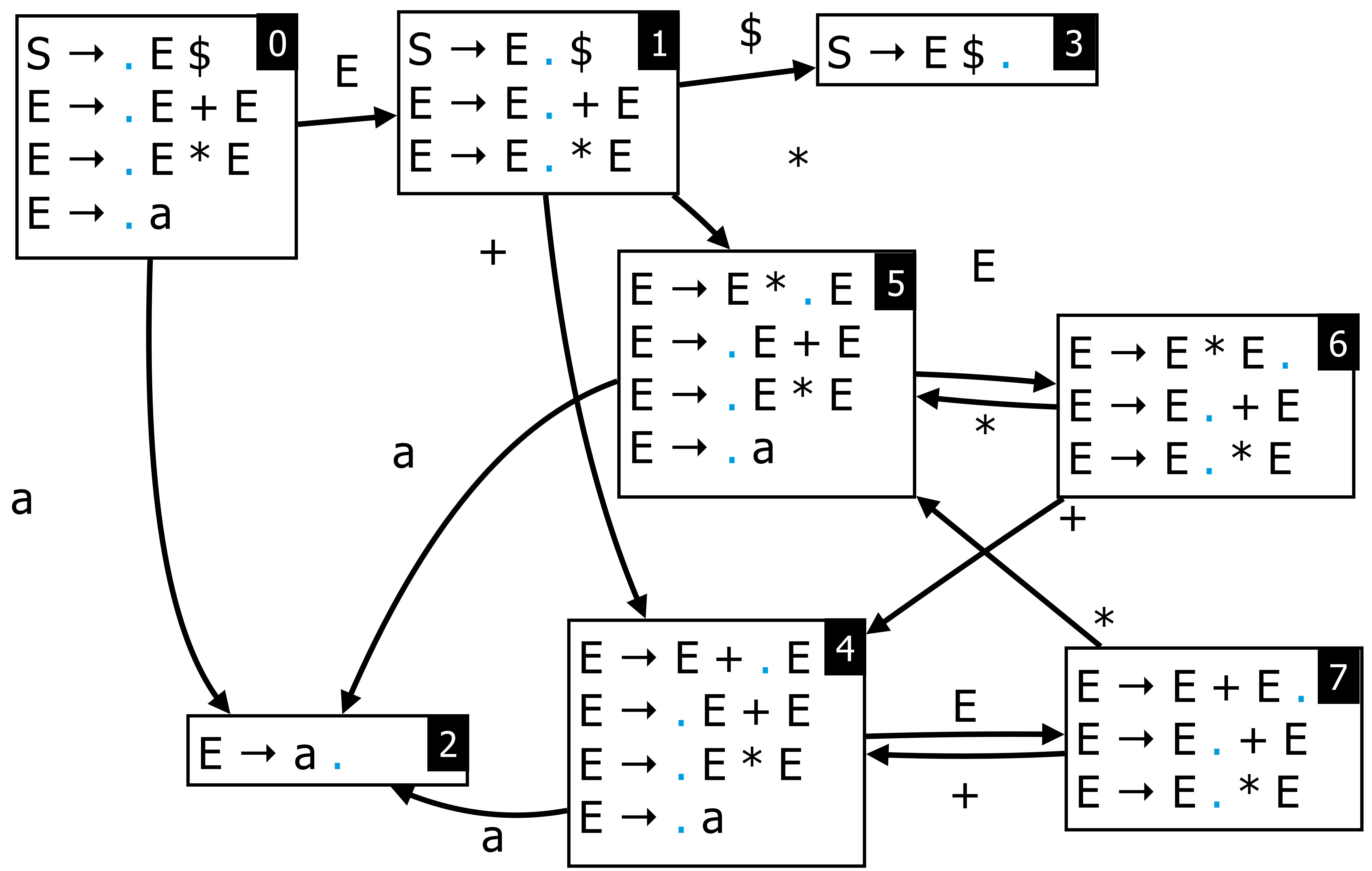
S → E $
E → E + E
E → E * E
E → a

SLR Table



S → E $
E → E + E
E → E * E
E → a

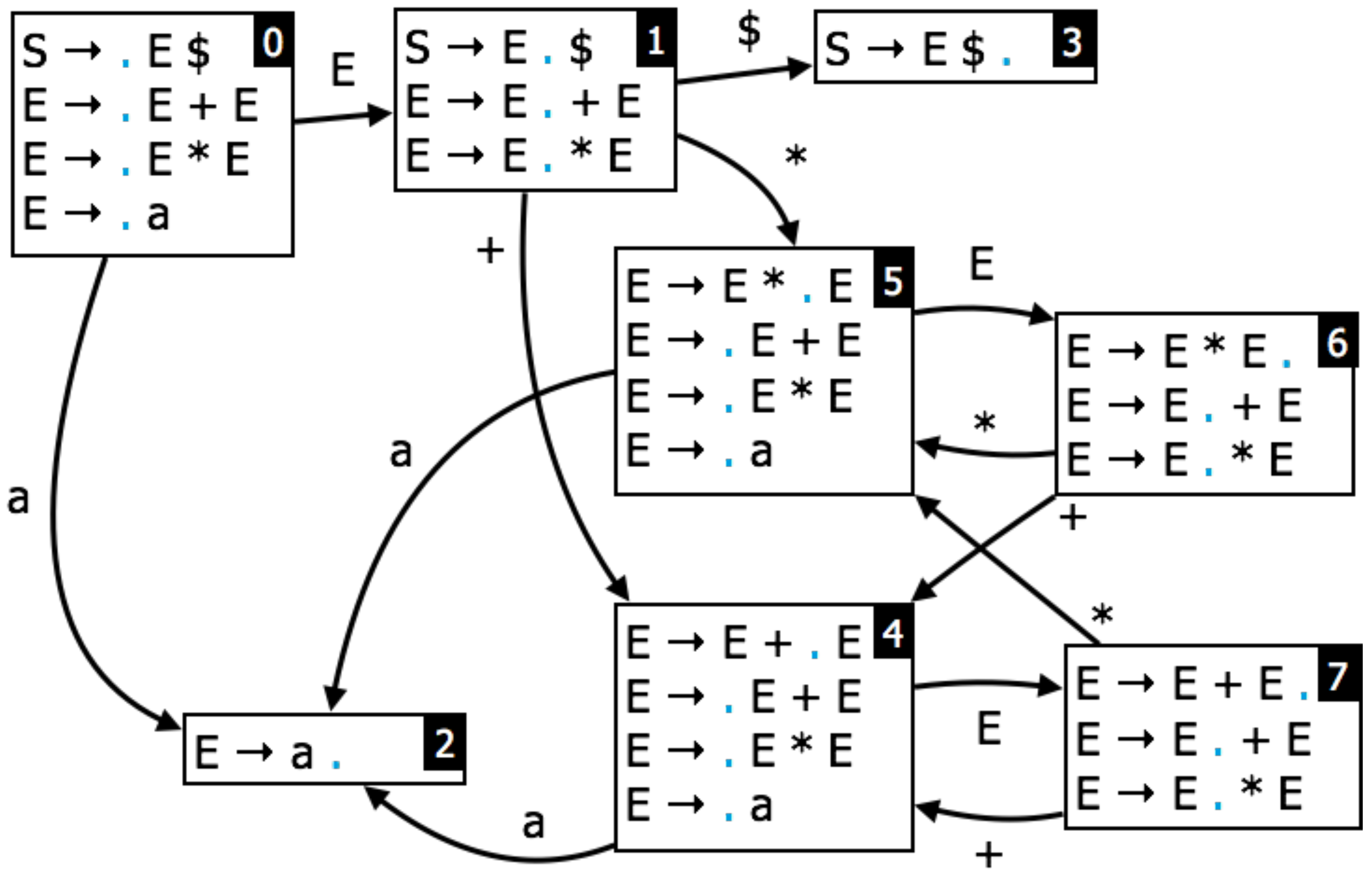
SLR Table



S → E $
E → E + E
E → E * E
E → a

SLR Table

Nonter	Nullable	First	Follow
S			
E			

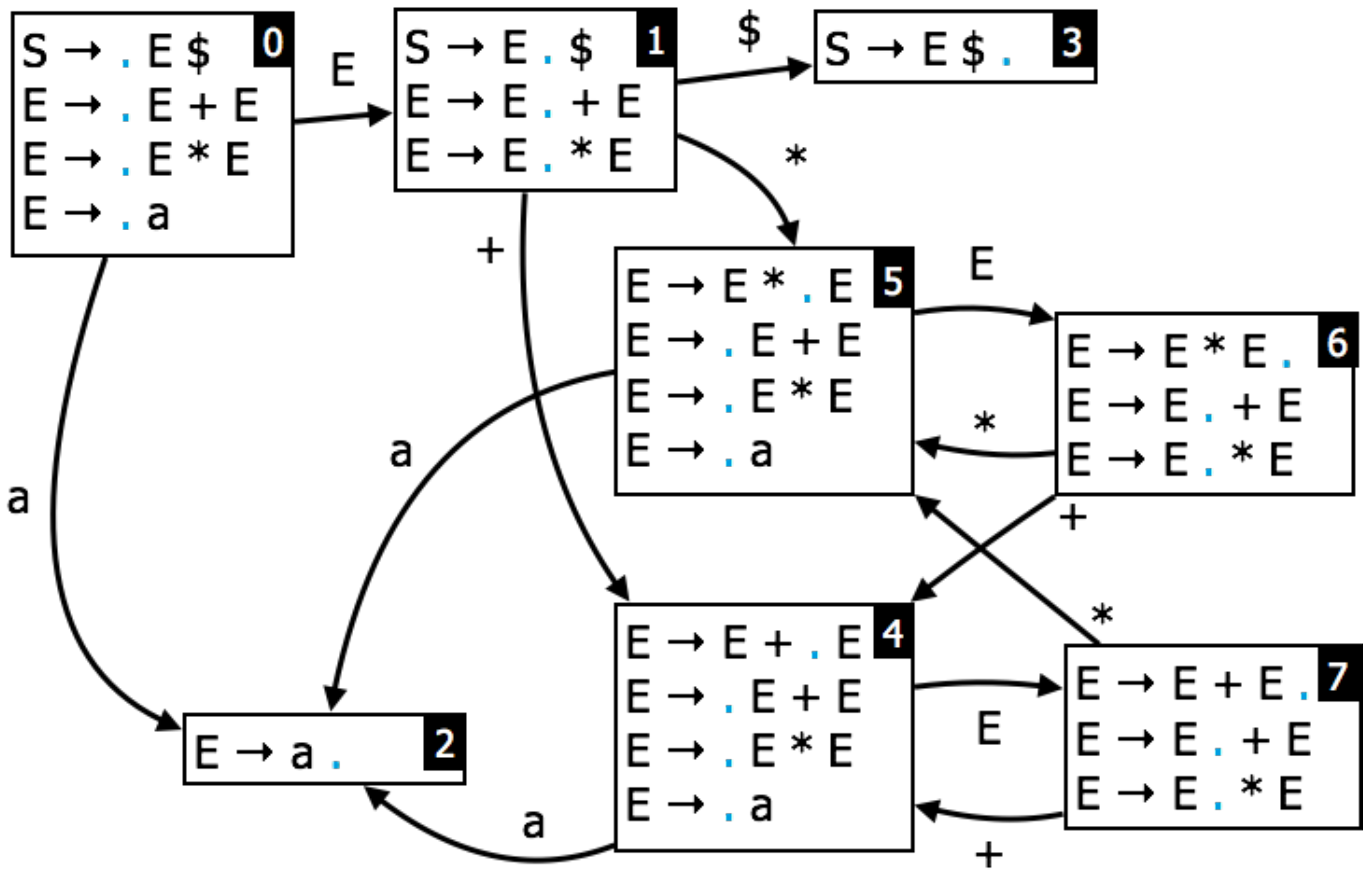


State	Action				Goto	
	a	+	*	$	S	E
0						
I						
2						
3						
4						
5						
6						
7						

(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	+, *, $

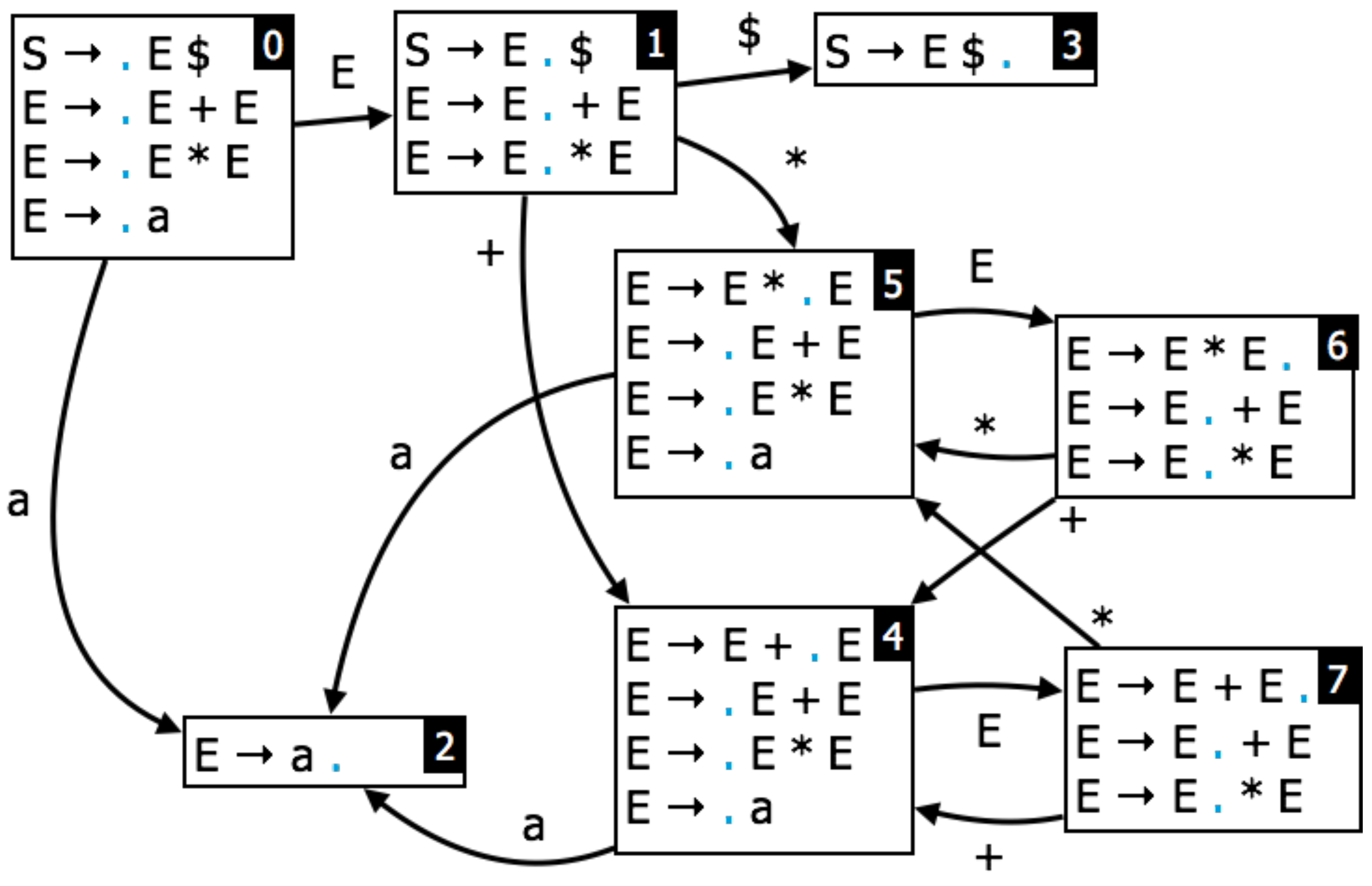


State	Action				Goto	
	a	+	*	$	S	E
0						
1						
2						
3						
4						
5						
6						
7						

(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		

$\boxed{0}$

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		

$$\begin{tikzcd} \boxed{0} & \boxed{2} \arrow[l, "a"'] \end{tikzcd}$$

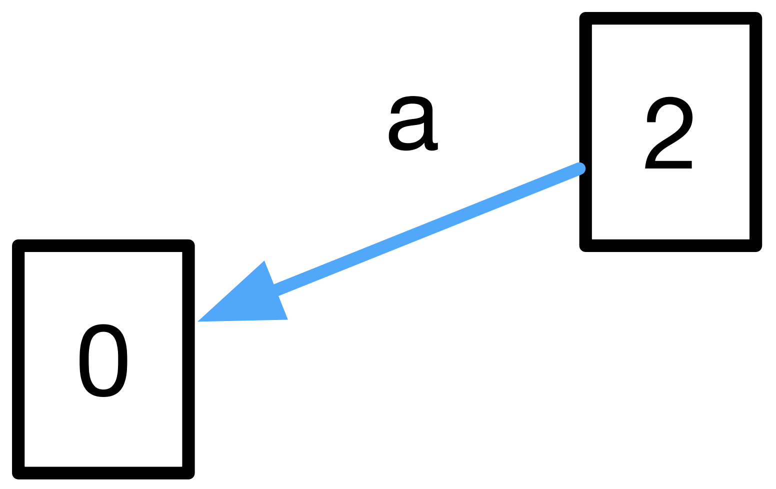
synchronize on shifts

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		

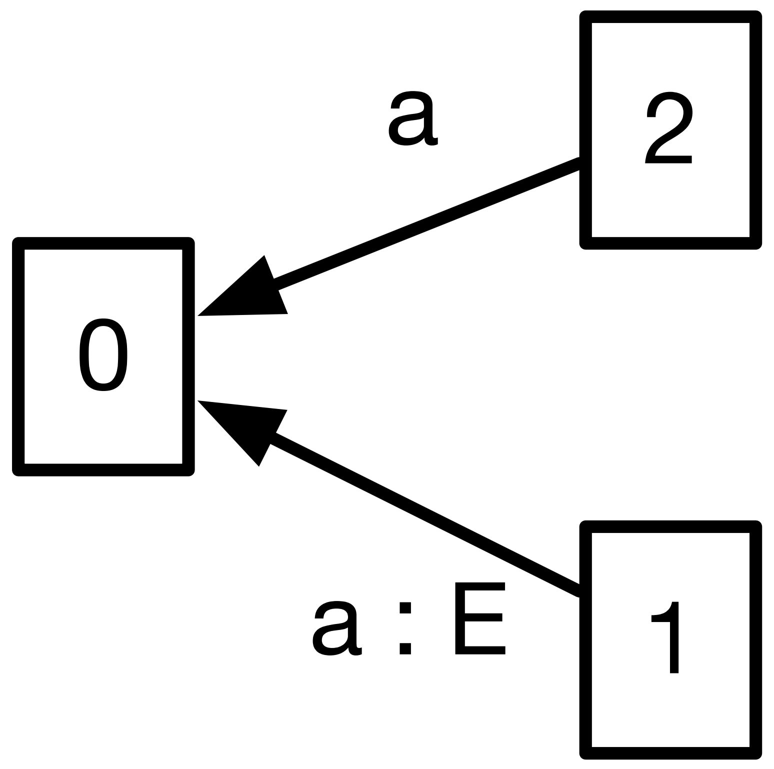


Parsing

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		

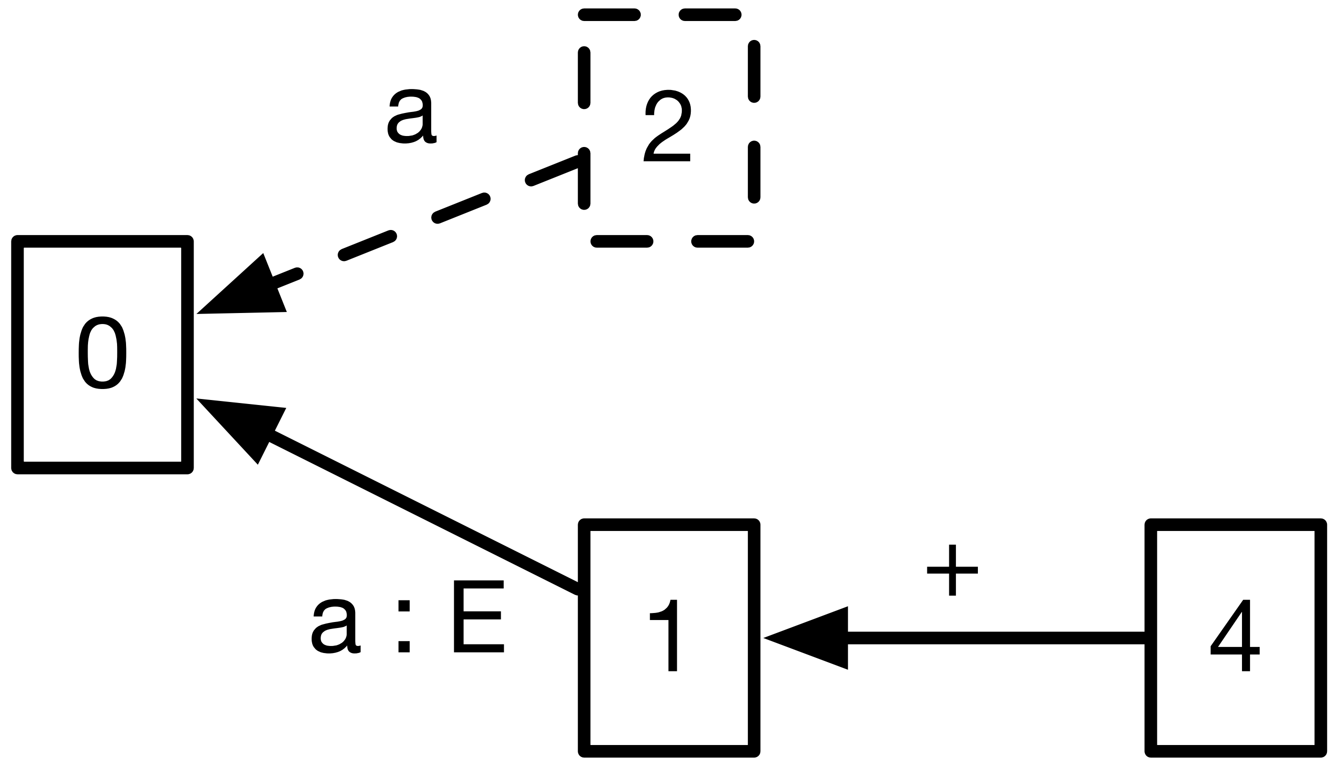


Parsing

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		



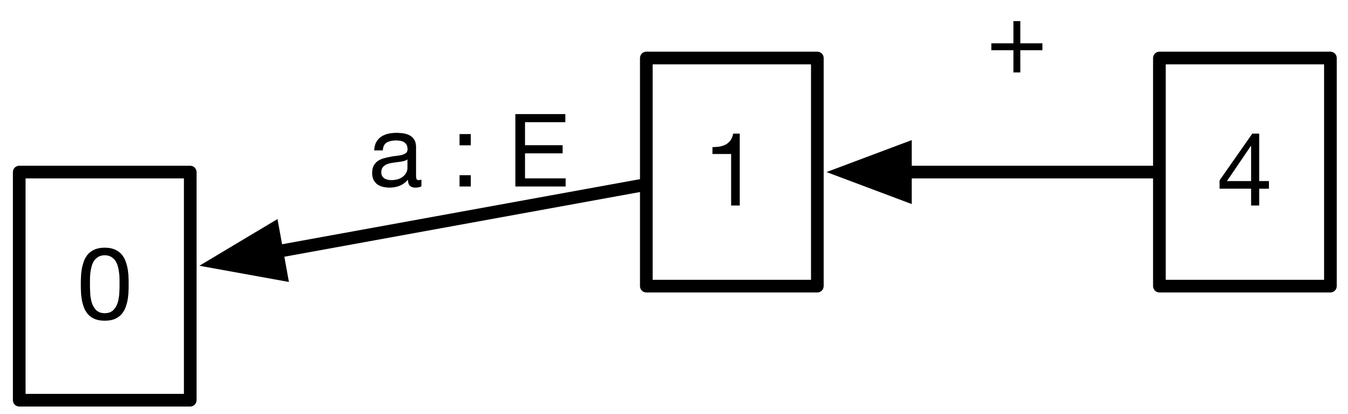
synchronize

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		



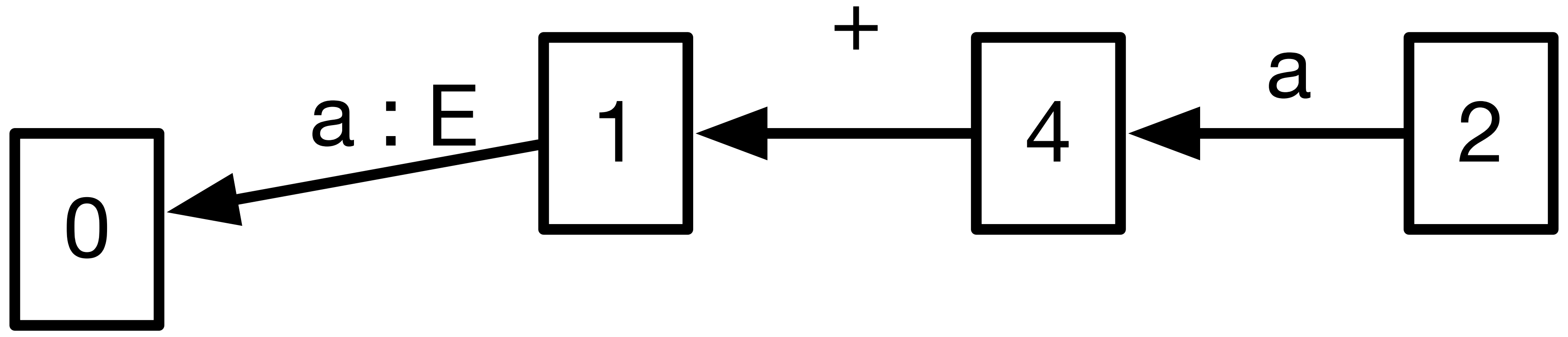
Parsing

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

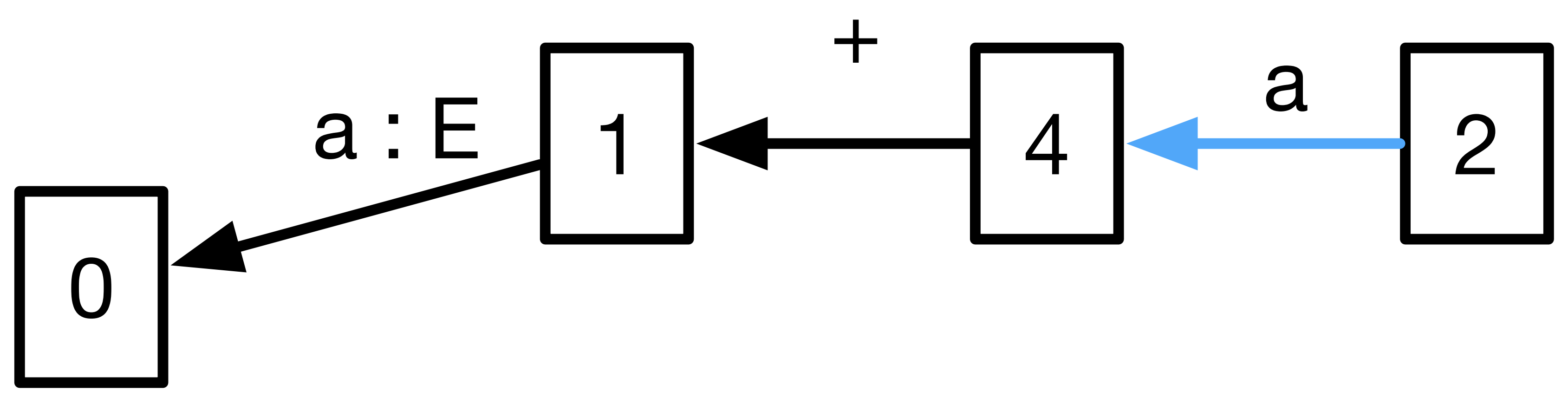

$$\begin{tikzcd} \boxed{0} & \boxed{1} \arrow[l, "a : E"'] & \boxed{4} \arrow[l, "+"'] & \boxed{2} \arrow[l, "a"'] \end{tikzcd}$$

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		

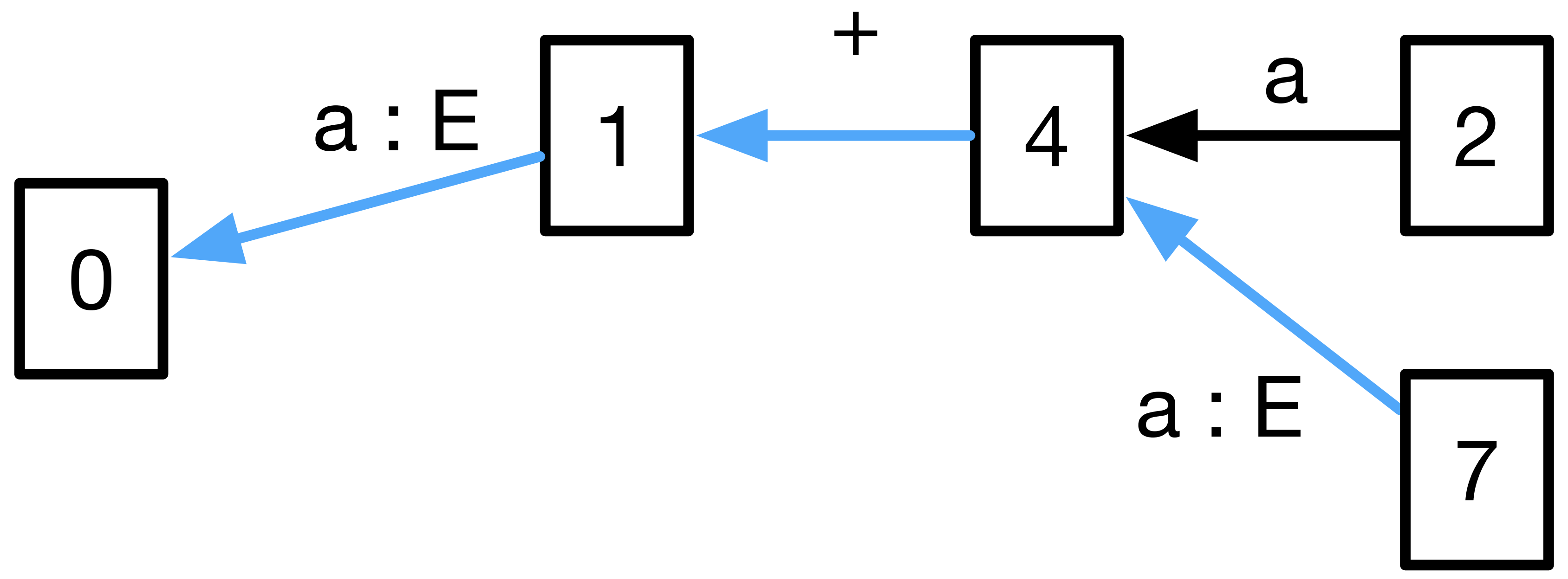


Parsing

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		

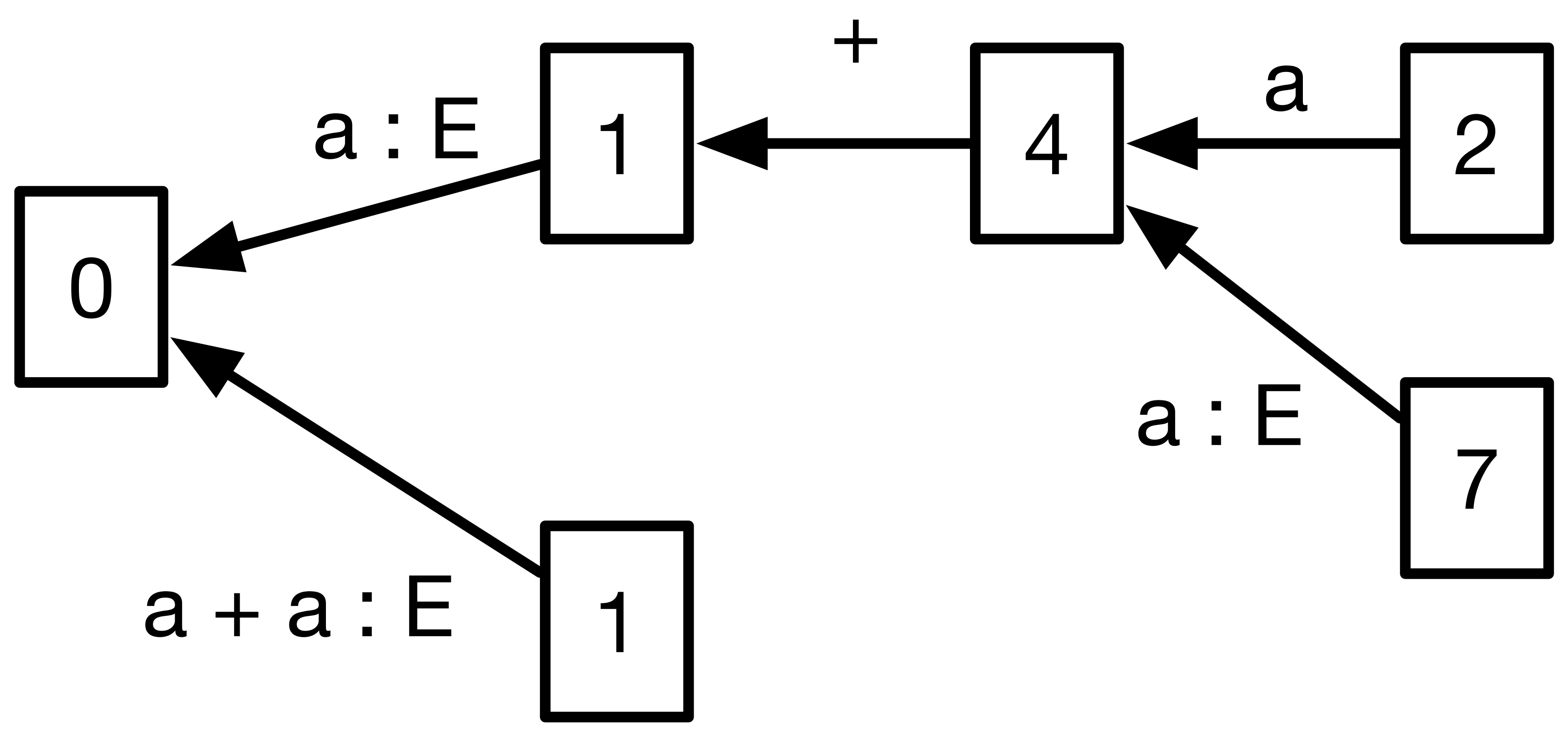


Parsing

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		



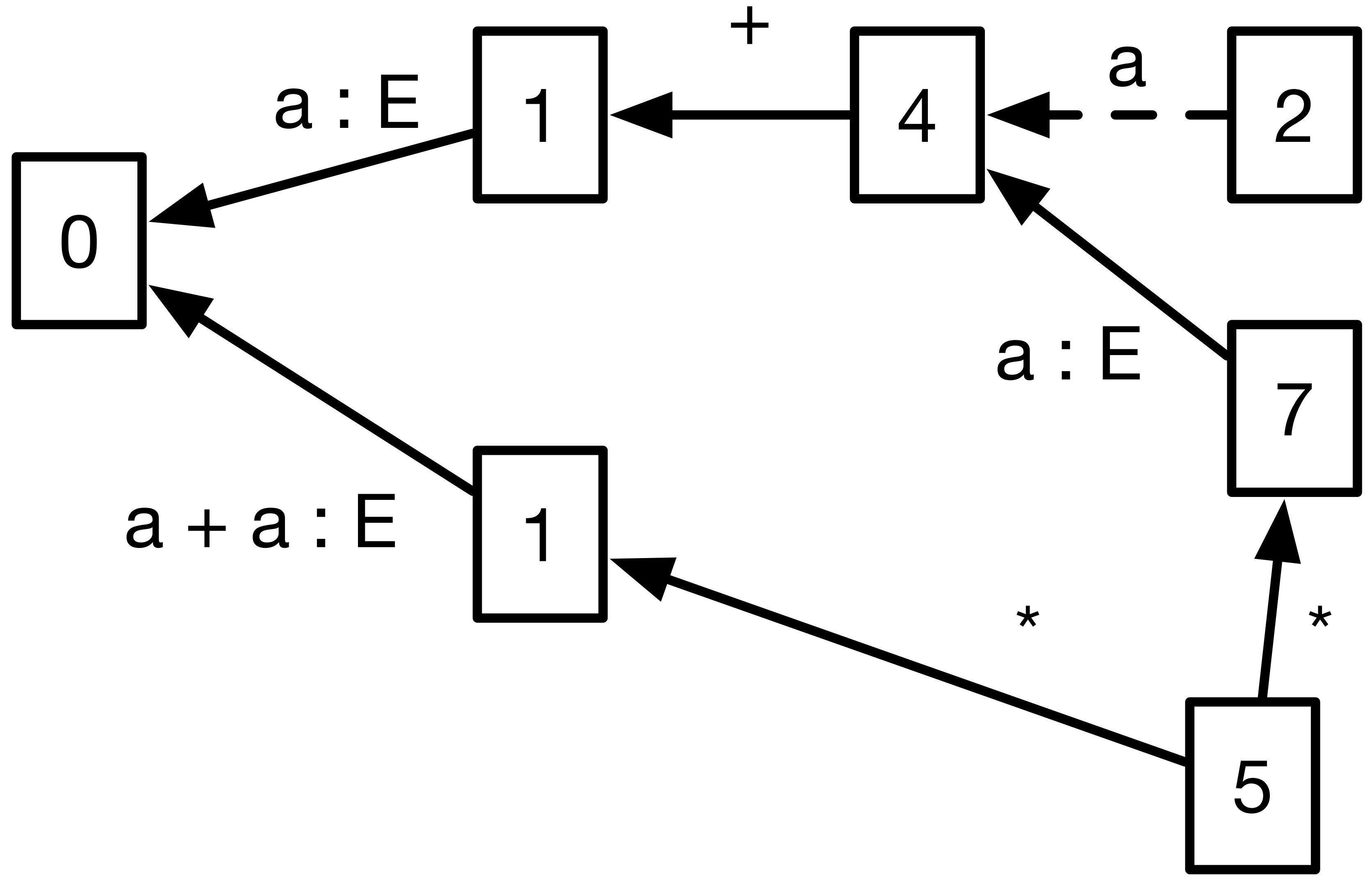
Parsing

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

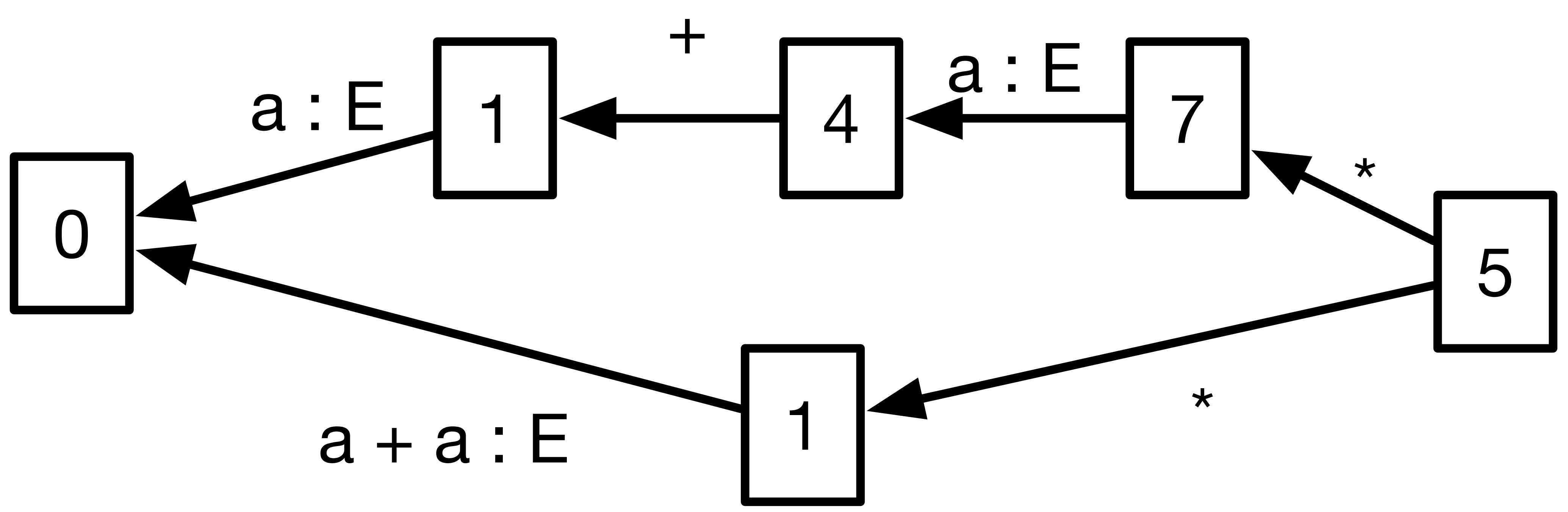


Parsing

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		



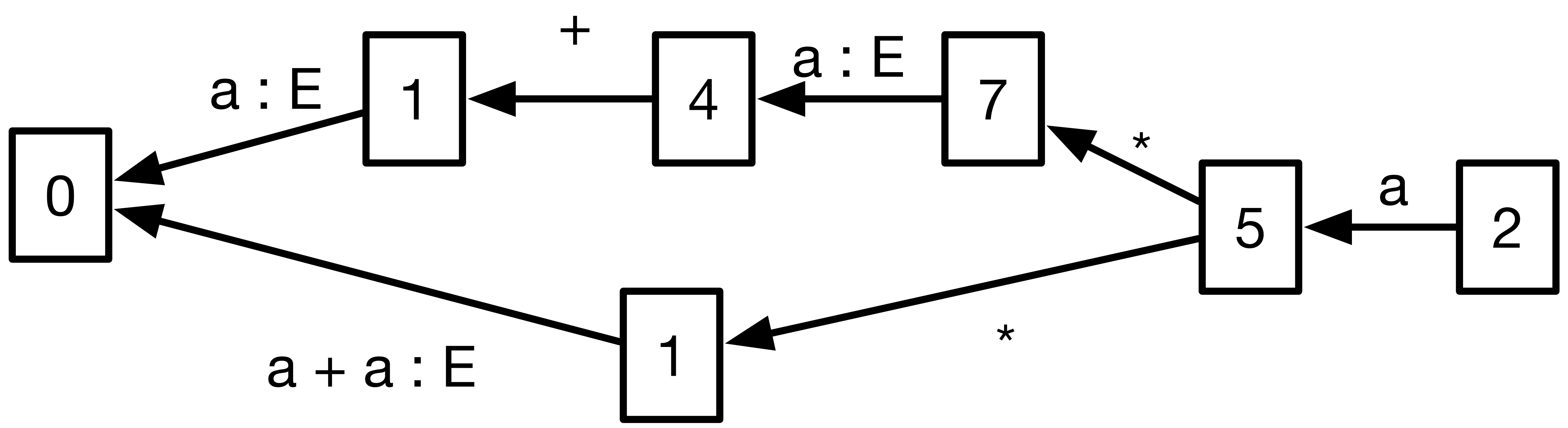
Parsing

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

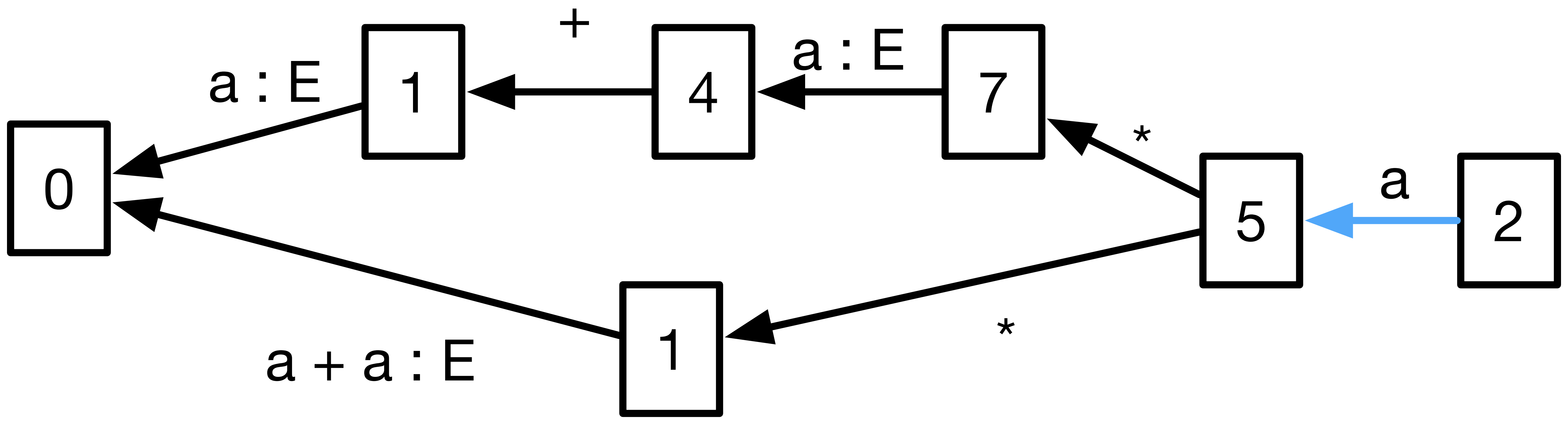


Parsing

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		

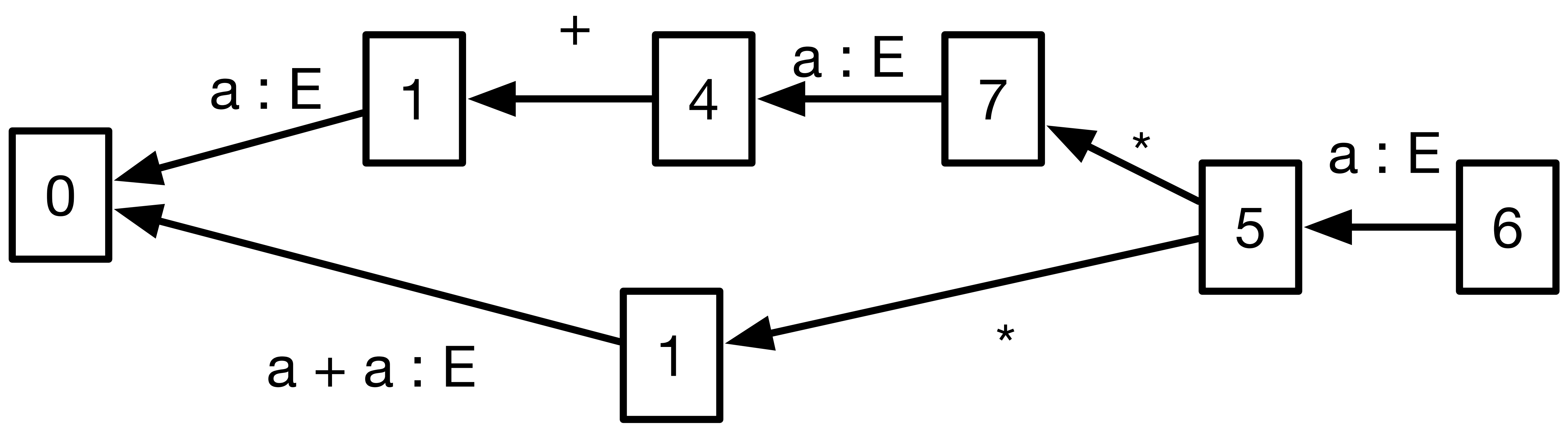


Parsing

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		

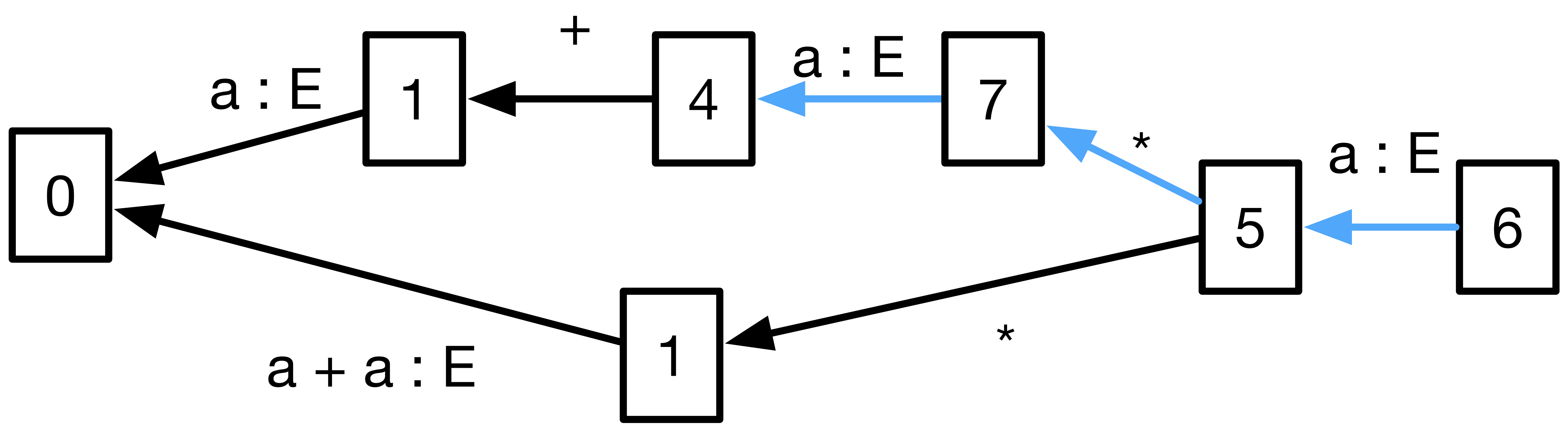


Parsing

input: $

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

<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>

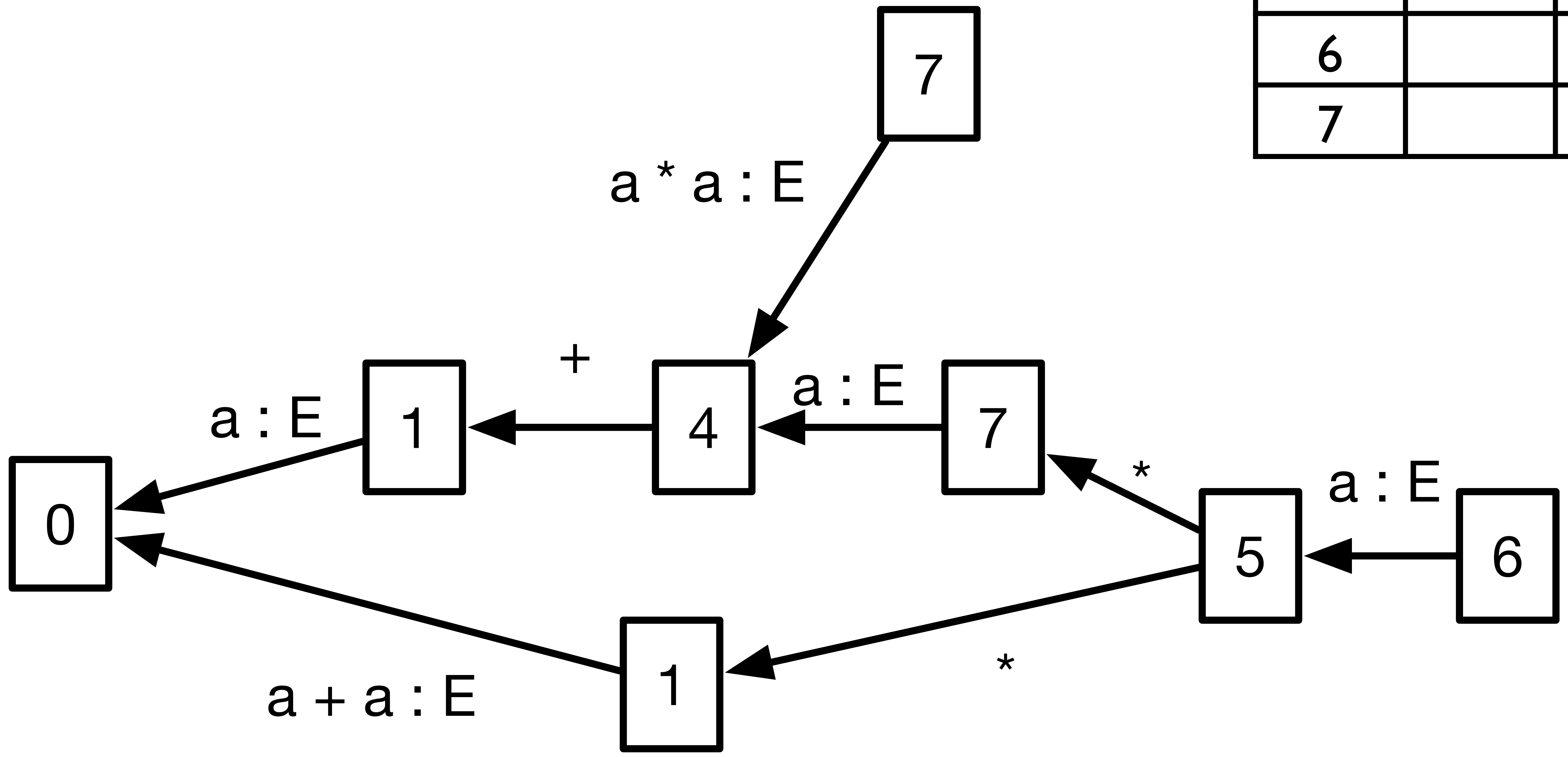


Parsing

input: $

(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		

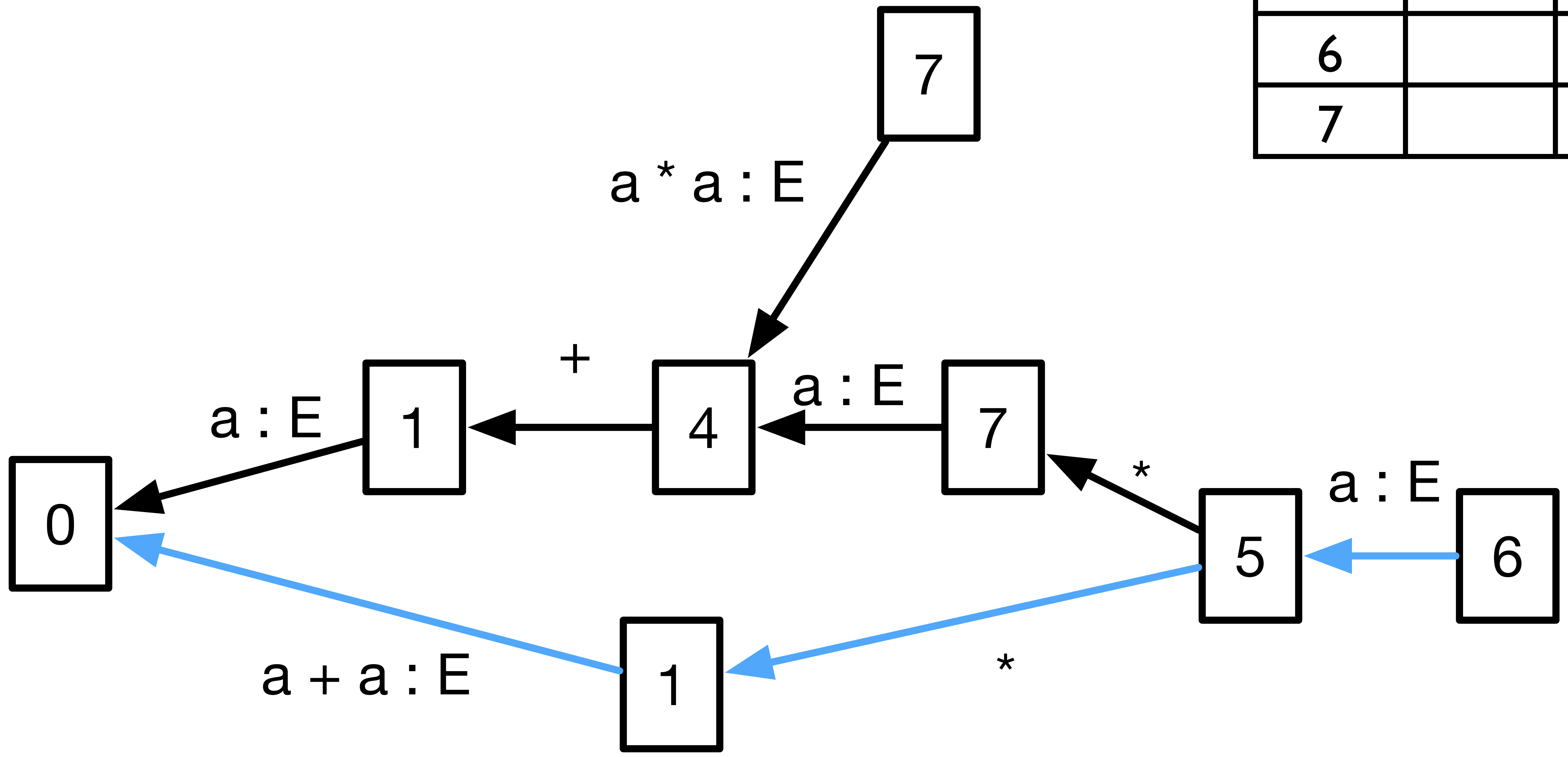


Parsing

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		

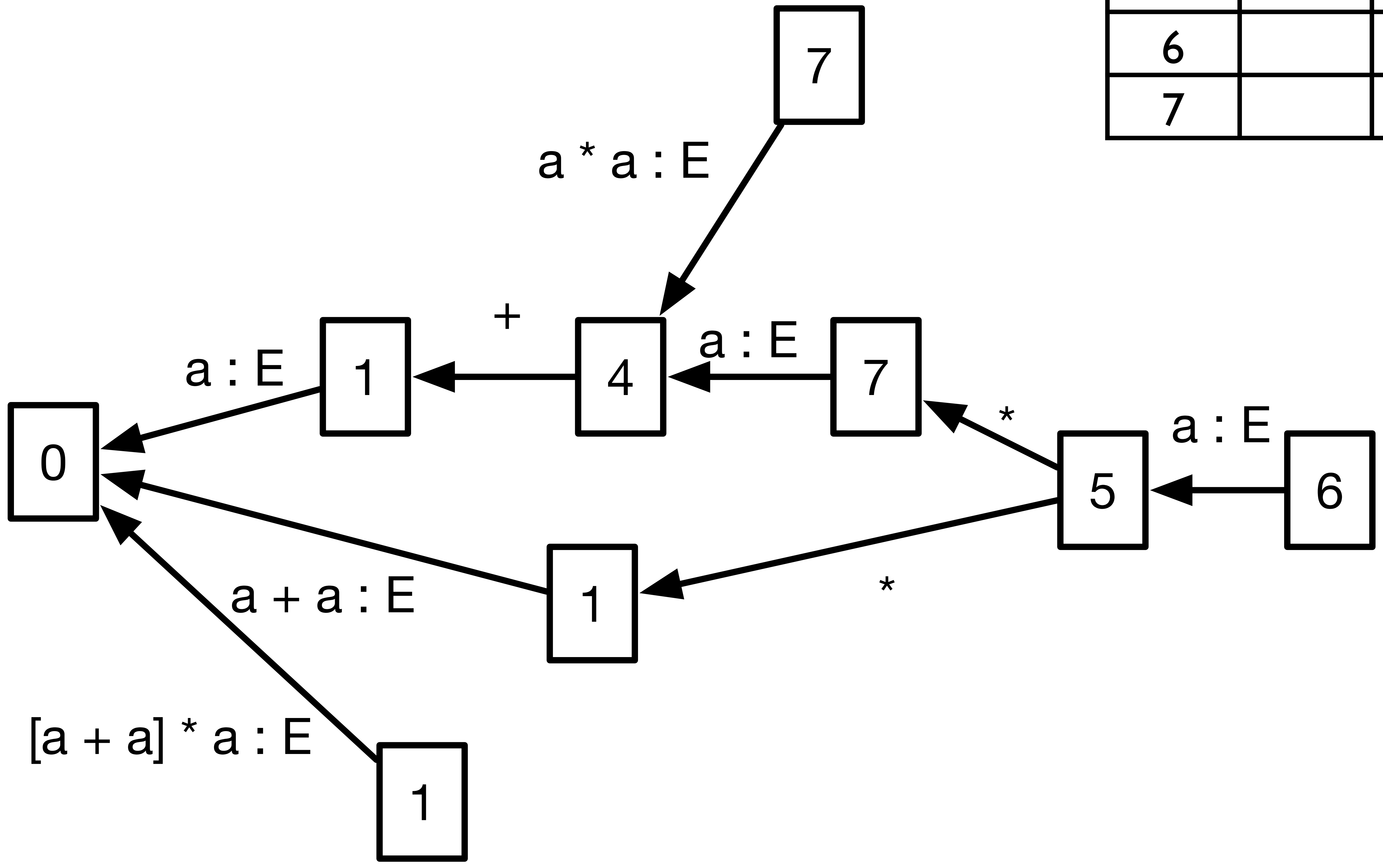


Parsing

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		

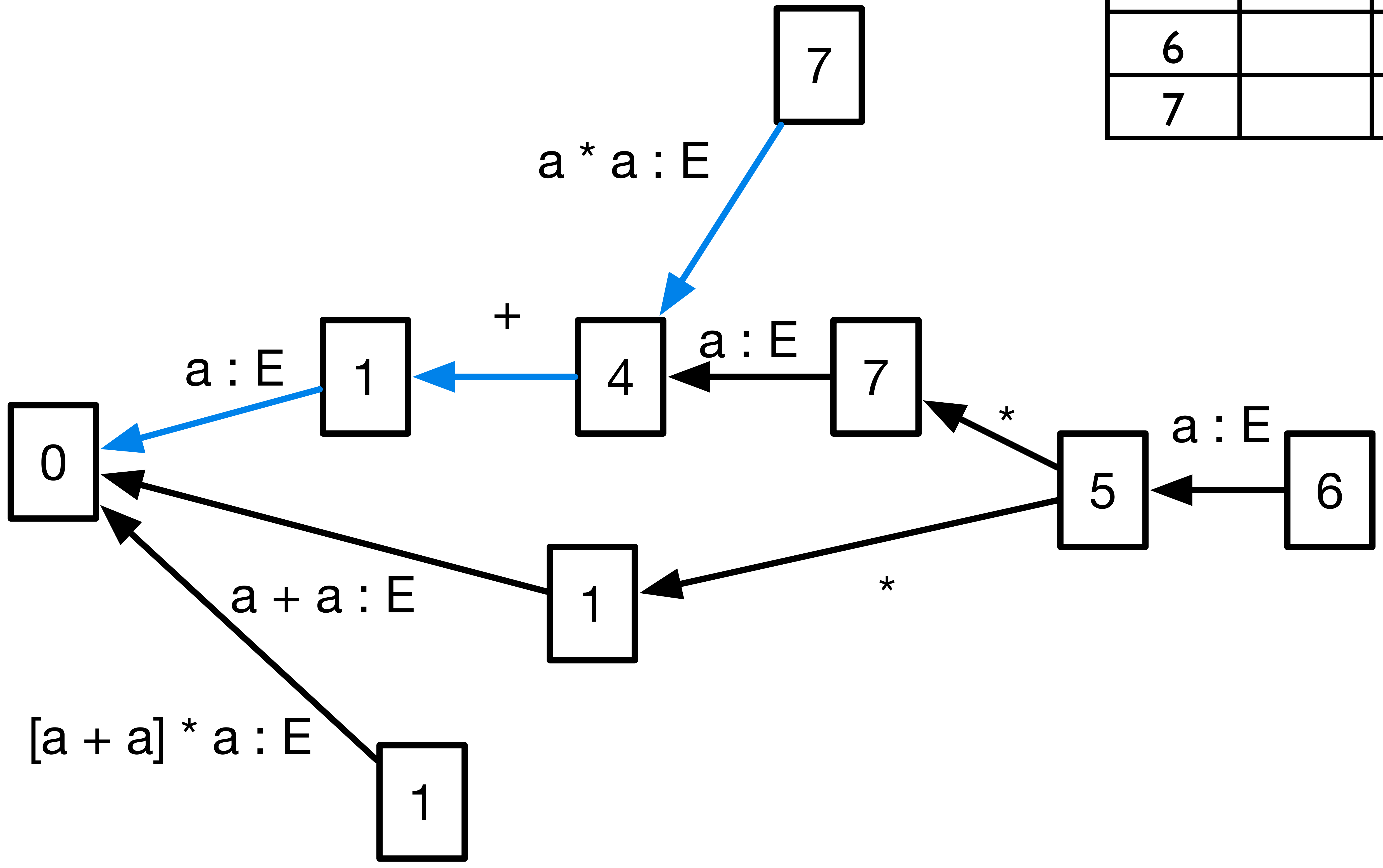


Parsing

input: $

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

<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>

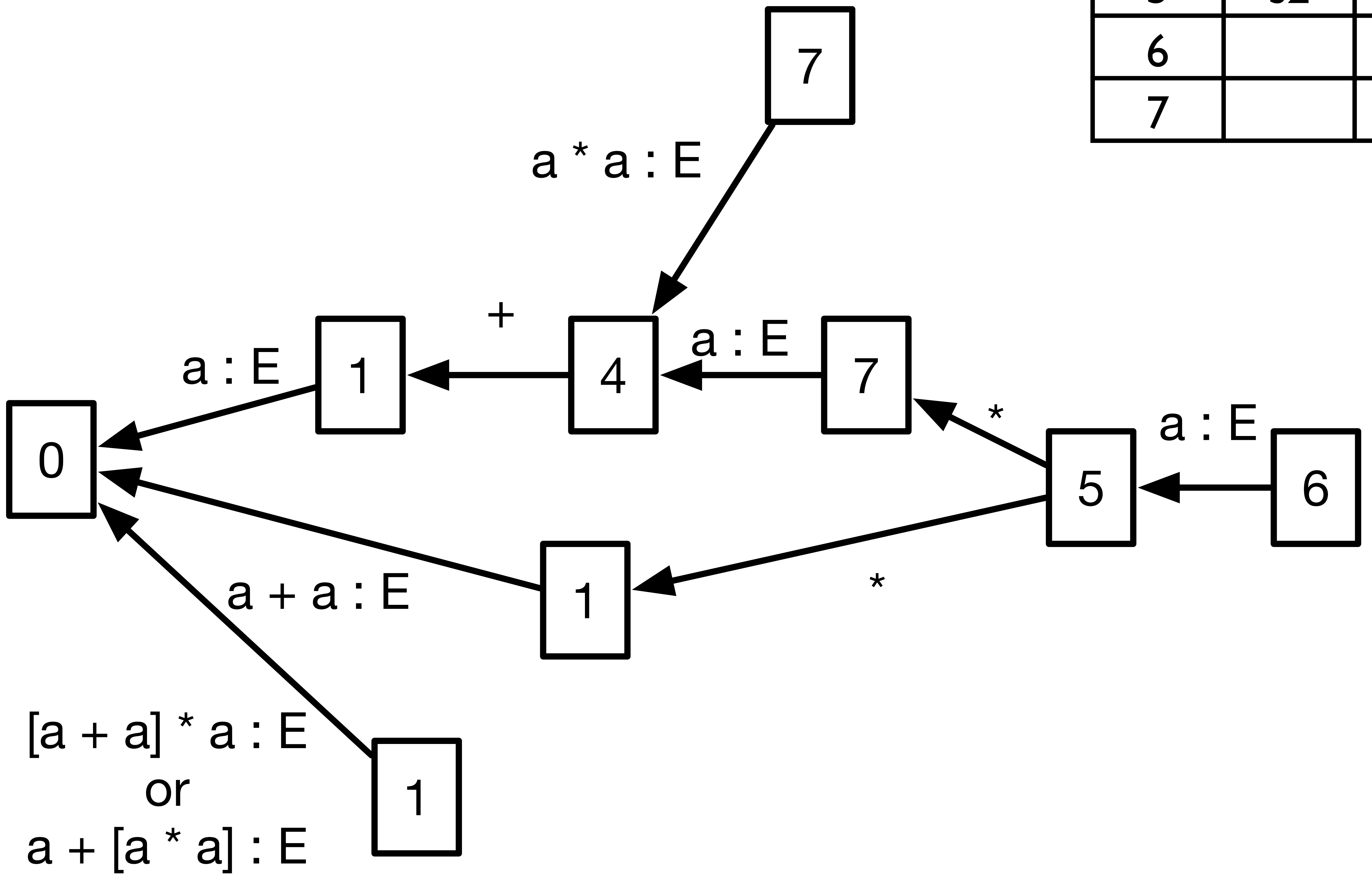


Parsing

input: $

(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		

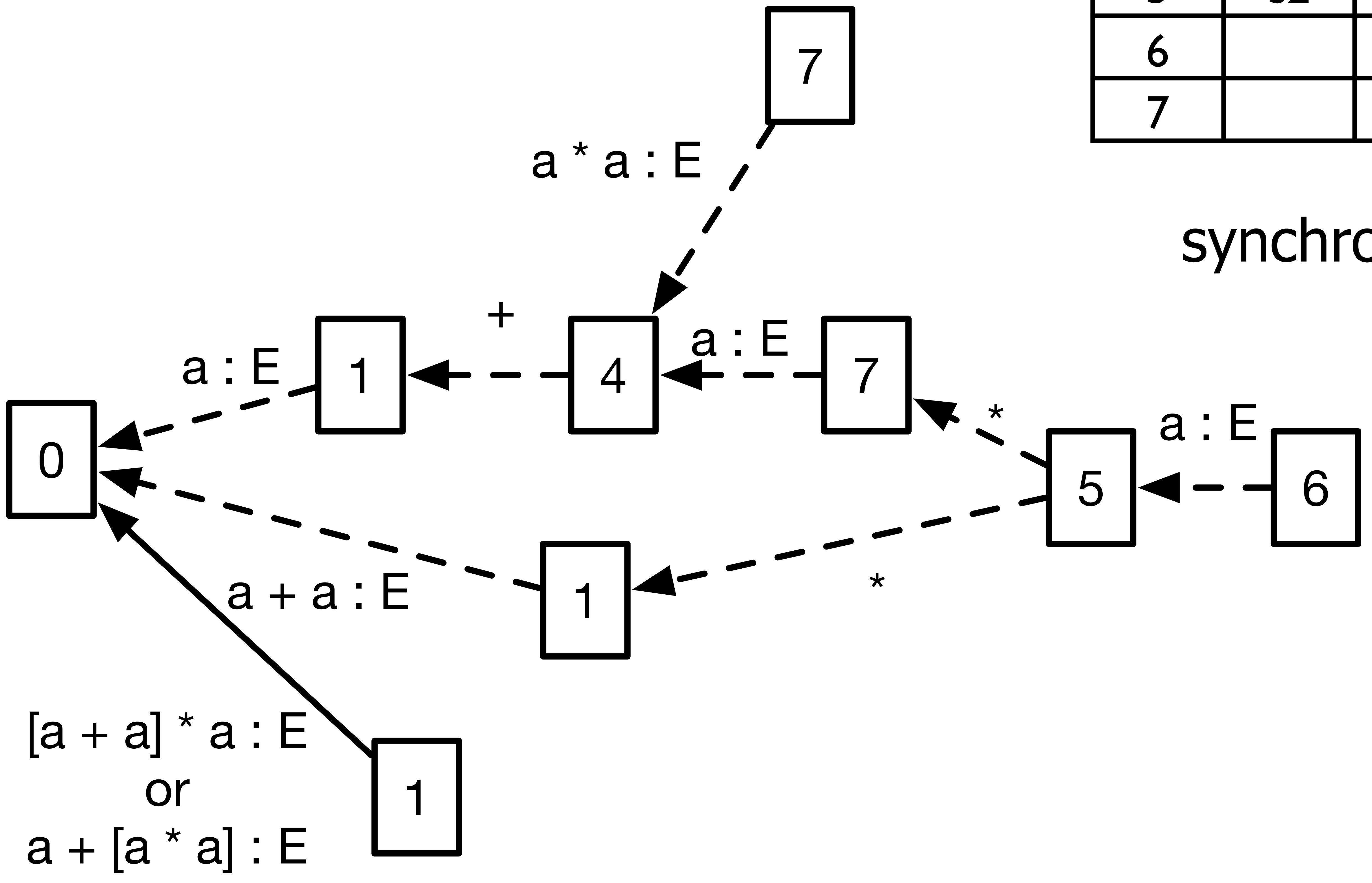


Parsing

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

Other Parsing Techniques

LL Parsing (Recursive Descent)

Exp → “while” Exp “do” Exp

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

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();
   }
}
```

Parser Combinators

- Parsers are modelled as functions, and larger parsers are built from smaller ones using higher-order functions.

- Can be implemented in any lazy functional language with higher-order style type system (e.g. Haskell, Scala).

- Parser combinators enable a recursive descent parsing strategy that facilitates modular piecewise construction and testing.

- Primitive parsers: **success**, **fail** and **match**

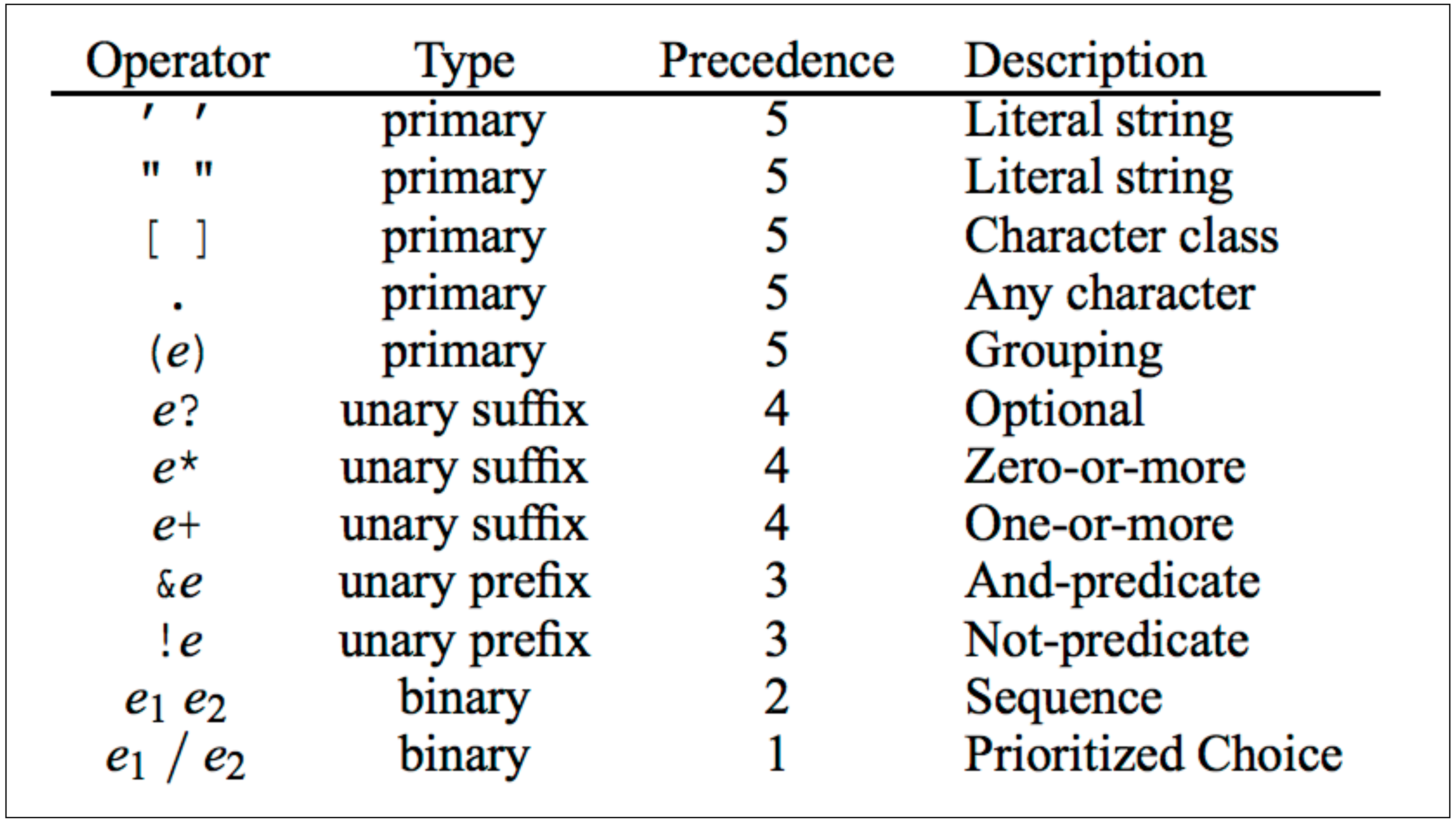
- Parser Combinators: **alternation** and **sequencing**

Parsing Expression Grammars

Rules are of the form:

A ← **e**

where A is a non-terminal
and **e** is a parsing expression.



Operator	Type	Precedence	Description
' '	primary	5	Literal string
" "	primary	5	Literal string
[ ]	primary	5	Character class
.	primary	5	Any character
(e)	primary	5	Grouping
e?	unary suffix	4	Optional
e*	unary suffix	4	Zero-or-more
e+	unary suffix	4	One-or-more
&e	unary prefix	3	And-predicate
!e	unary prefix	3	Not-predicate
$e_1\ e_2$	binary	2	Sequence
e_1 / e_2	binary	1	Prioritized Choice

Parsing Expression Grammars

- Prioritized choice '/'.

S ← **a b** / **a** ≠ S ← **a** / **a b**

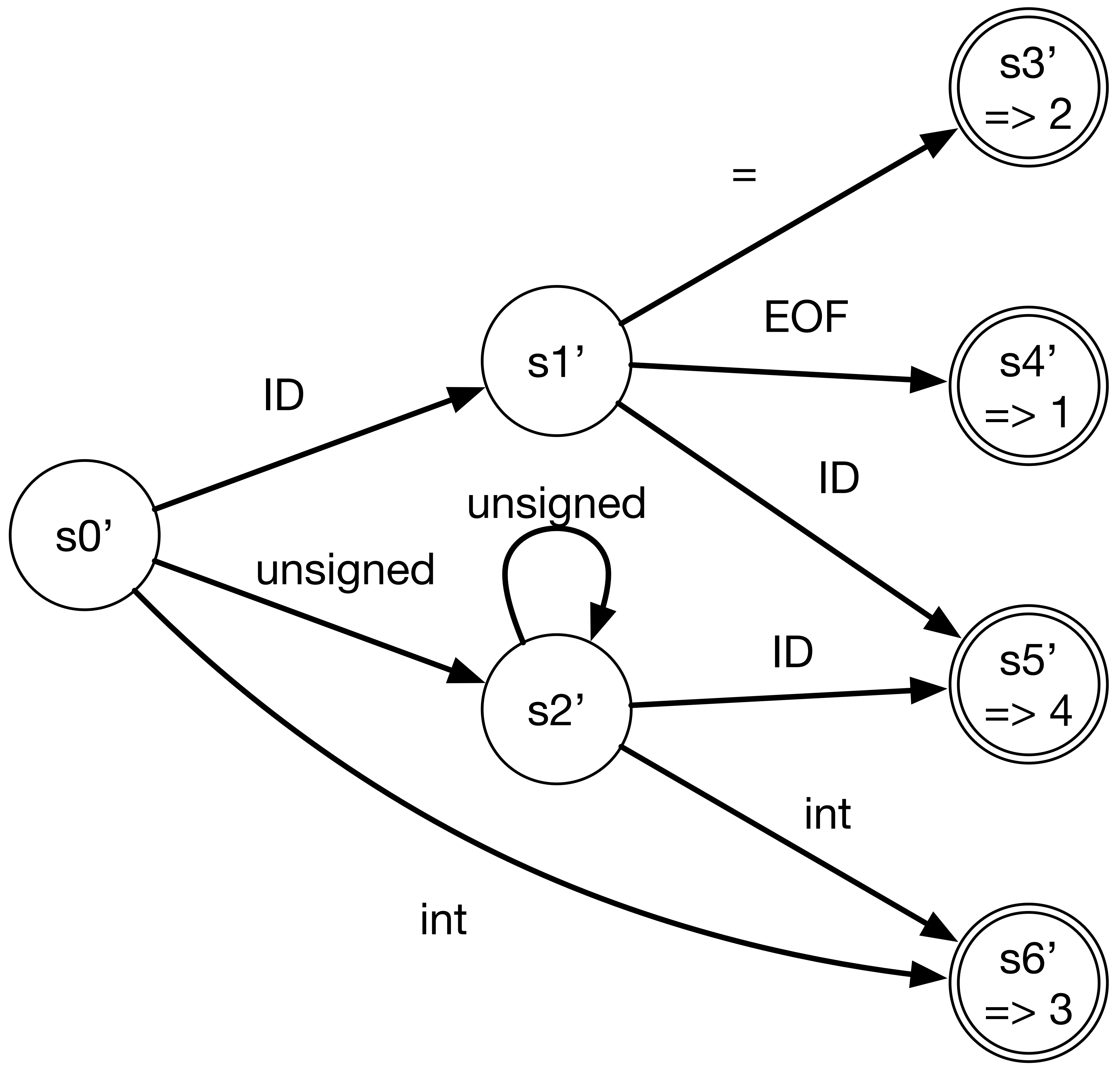
- Backtracks when parsing an alternative fails.
- Uses memoization of intermediate results to parse in linear time, called packrat parsing.
- Does not allow left-recursion.

LL(*)

- Parses LL-regular grammars: for any given non-terminal, parsers can use the entire remaining of the input to differentiate the alternative productions.
- Statically builds a DFA from the grammar productions to recognize lookahead.
- When in conflict, chooses the first alternative and backtracks when DFA lookahead recognition fails.

```
S : ID                  (1)
  | ID = E              (2)
  | unsigned* int ID    (3)
  | unsigned* ID ID     (4)
```

terminals: ID, **=**, **int**, **unsigned**



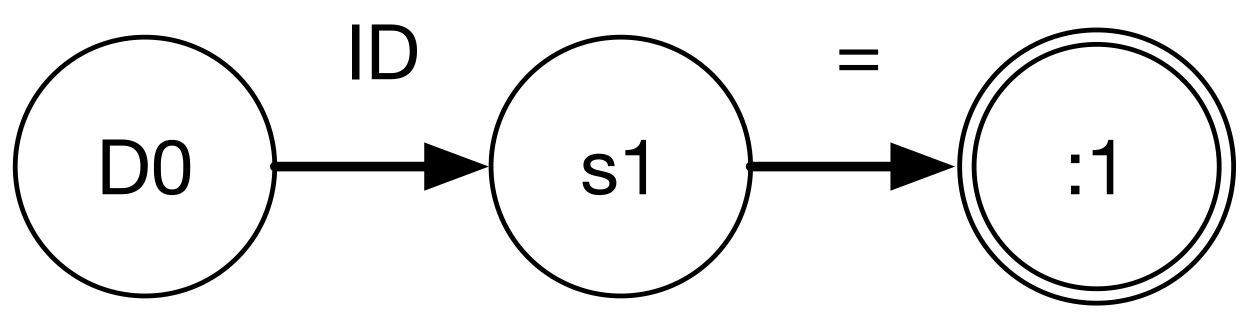
ALL(*)

Dynamically builds the lookahead DFA.

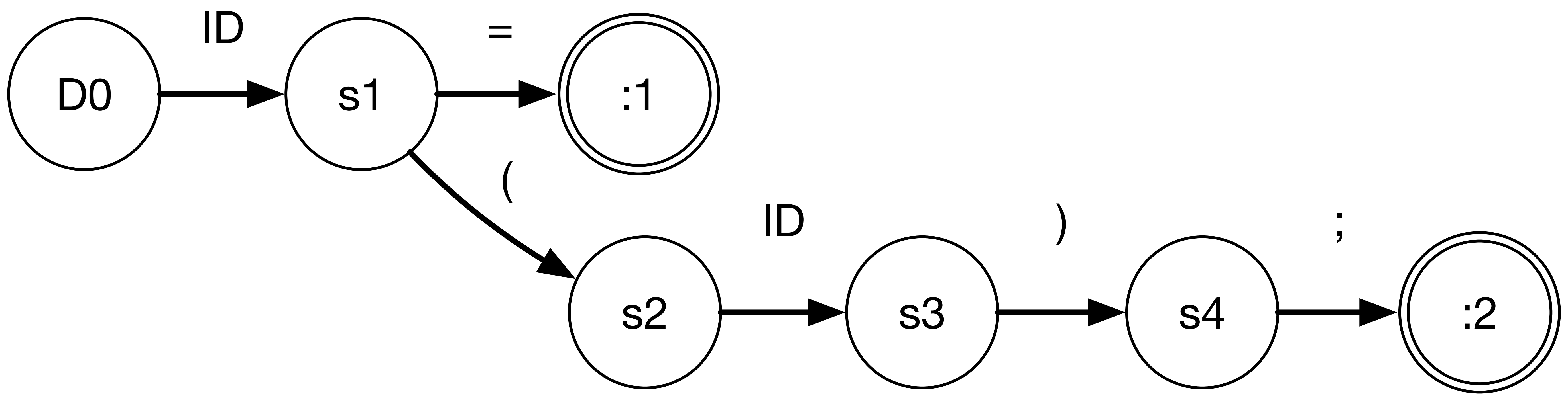
```
S : E = E ;
  | E ;

E : E * E
  | E + E
  | E ( E )
  | ID
```

After parsing:

x = y;



f(x);



Exam Question

TU Delft

Question 12: LR parsing

Let G_2 be a formal grammar with nonterminal symbols S, T and E, terminal symbols $\mathbf{x}$, $+$ and \$, start symbol S, and the following production rules:

$$\begin{array}{lcl} S & \rightarrow & E\ \$ \\ E & \rightarrow & T + E \\ E & \rightarrow & T \\ T & \rightarrow & \mathbf{x} \end{array}$$

(a) Explain the role of the terminal symbol \$.

(b) Construct a LR(0) parse table for the grammar.

(c) What kind of conflict does the resulting parse table contain?

(d) Explain two strategies to resolve this conflict.

(e) Assume the conflict is resolved in favour of a shift. Use the parse table to recognise the sentence $\mathbf{x} + \mathbf{x}$. Show the stack and the remaining input after each step.