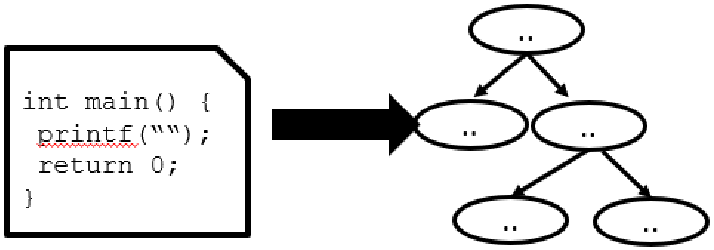
CSE110A: Compilers



```
int main() {
 printf("");
 return 0;
}
```

Topic:

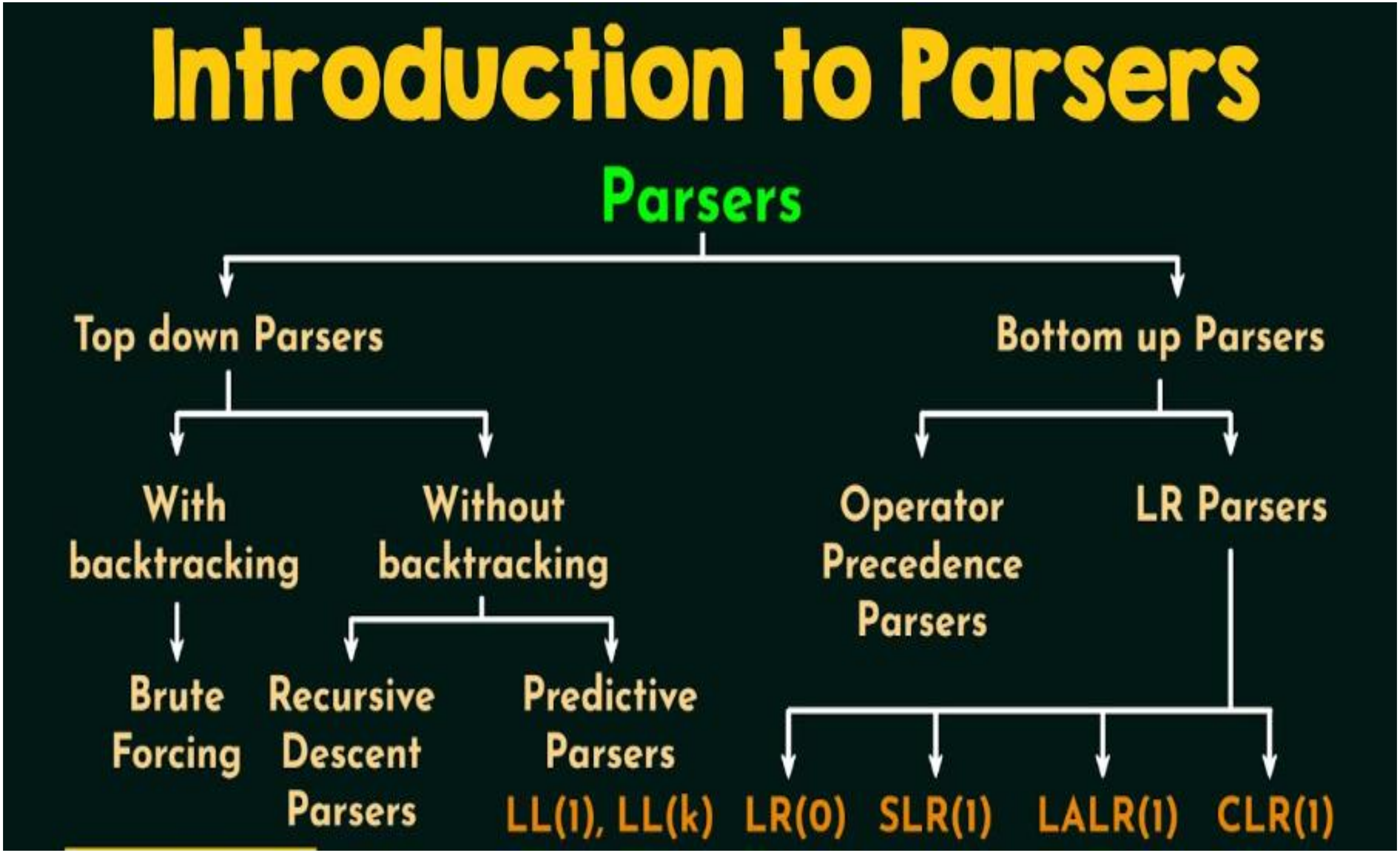
Bottom Up Parsing

Lecturer: Mrcelo Siero

Accreditation:

Most slides taken directly from Stanfords: CS143 Lecture 8 slide design by: Prof. Alex Aiken with some modifications by Marcelo Siero

Source: https://web.stanford.edu/class/cs143/lectures/lecture08.pdf



Introduction to Parsers

Parsers

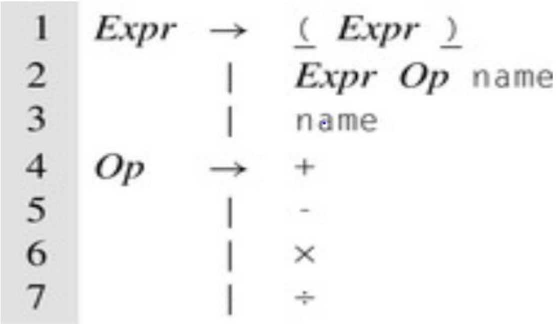
- Top down Parsers
  - With backtracking
    - Brute Forcing
  - Without backtracking
    - Recursive Descent Parsers
    - Predictive Parsers
      - LL(1), LL(k)
- Bottom up Parsers
  - Operator Precedence Parsers
  - LR Parsers
    - LR(0)
    - SLR(1)
    - LALR(1)
    - CLR(1)

Source: https://www.youtube.com/watch?v=OIKL6wFjFOo

Some Definitions

Context Free Grammar: Formally, a context-free grammar *G* is a quadruple (*T*, *NT*, *S*, *P*) where: T is a set of terminals, NT is a set of non-terminals, S is a Start symbol, and P a set of Productions all for language L(G).

Ambiguity: A grammar *G* is *ambiguous* if some sentence in *L*(*G*) has more than one rightmost (or leftmost) derivation.

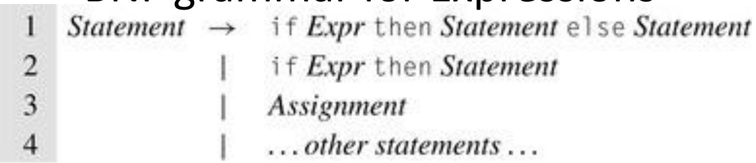


1	*Expr*	→	(*Expr*)
2		|	*Expr* *Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

BNF grammar for Expressions

Sentential Form: a string of symbols that occurs as one step in a valid derivation

Derivation: a sequence of rewriting steps that begins with the grammar's start symbol and ends with a sentence in the language

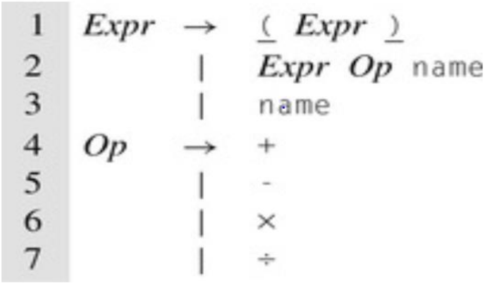


1	*Statement*	→	if *Expr* then *Statement* else *Statement*
2		|	if *Expr* then *Statement*
3		|	*Assignment*
4		|	. . . *other statements* . . .

Classical ambiguous grammar from Algol 60

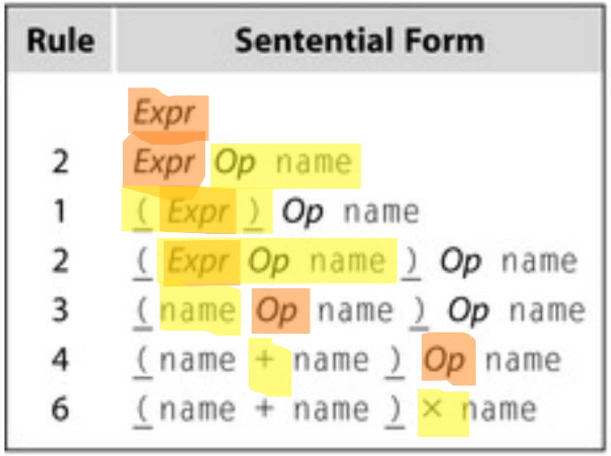
Sentence: a string of symbols that can be derived from the rules of a grammar, i.e. a sentential form without non-terminals.

Leftmost and Rightmost Derivations



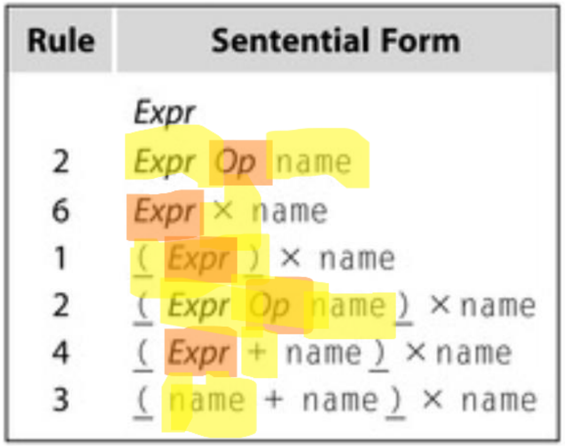
1	*Expr*	→	(*Expr*)
2		|	*Expr* *Op* name
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Note that leftmost derivations tend to be right associative
Rightmost derivations left associative.



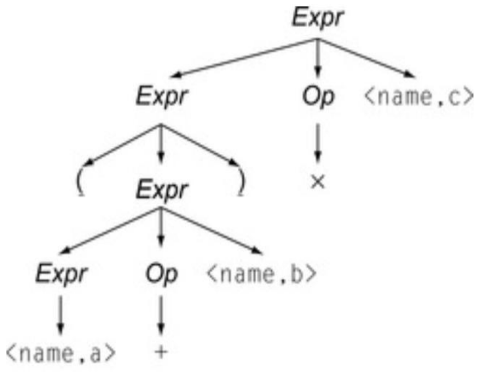
Rule	Sentential Form
	Expr
2	*Expr* *Op* name
1	(*Expr*) *Op* name
2	(*Expr* *Op* name) *Op* name
3	(name *Op* name) *Op* name
4	(name + name) *Op* name
6	(name + name) × name

Leftmost Derivation of (a + b) x c



Rule	Sentential Form
	Expr
2	*Expr* *Op* name
6	*Expr* × name
1	(*Expr*) × name
2	(*Expr* *Op* name) × name
4	(*Expr* + name) × name
3	(name + name) × name

Rightmost Derivation of (a + b) × c



Corresponding Parse Tree

Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing

- Bottom-up is a very popular method

- Concepts: Sate Table Algorithms for Advanced
- Course in compilers.

Bottom Up Parsing (Pseudo Code from EAC text book)

```
push $;
push start_state: $s_0$;
word = NextWord();

while (true)do:
  state = top_of_stack
  if (Action[state.word] = “reduce A::=b”):
     pop 2 * |b| symbols;
     state = top_of_stack
     push A;
     push Goto[state, A];
  elif (Action[state.word] = “shift $s_i$”):
     push word;
     push $s_i$;
     word = NextWord();
  elif (Action[state.word] = “accept”):
     break
  else:
     Fail();
report_success
```

```
   S         : Expr
1. Expr      : Expr + Term
2.             | Expr - Term
3.             | Term
4. Term      : Term * Factor
5.             | term / Factor
6.             | Factor
7. Factor    : ( Expr )
8.             | num
9.             | name
```

An Introductory Example

- Bottom-up parsers don’t need left-factored grammars

- Revert to the “natural” grammar for our example:

  $E \rightarrow T + E \mid T$

  $T \rightarrow \text{int} * T \mid \text{int} \mid (E)$

- Consider the string: int * int + int

E → T + E | T
T → int * T | int | (E)

The Idea

Bottom-up parsing reduces a string to the start symbol by inverting productions:

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	

E → T + E | T
T → int * T | int | (E)

Observation

- Read the productions in reverse (bottom to top)
- This is a reverse rightmost derivation!

int * int + int	T → int
int * T + int	T → int * T
T + int	T → int
T + T	E → T
T + E	E → T + E
E	

Important Fact #1

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation
In reverse, i.e. from a Sentence to Start symbol.

A Bottom-up Parse

E → T + E | T
T → int * T | int | (E)

int * int + int

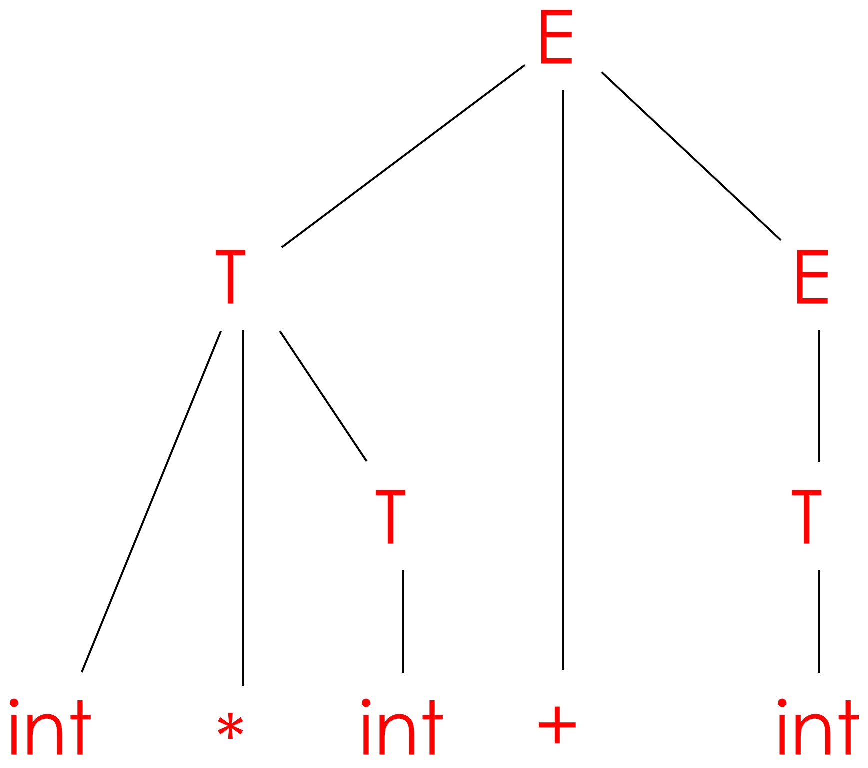
int * T + int

T + int

T + T

T + E

E



A Bottom-up Parse in Detail (1)

E → T + E | T
T → int * T | int | (E)

int * int + int

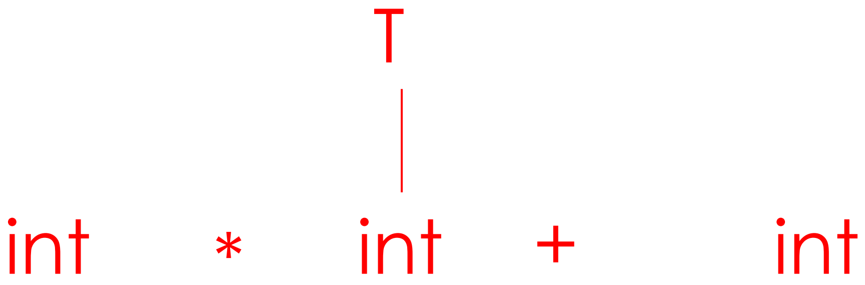
int * int + int

A Bottom-up Parse in Detail (2)

E → T + E | T
T → int * T | int | (E)

int * int + int

int * T + int



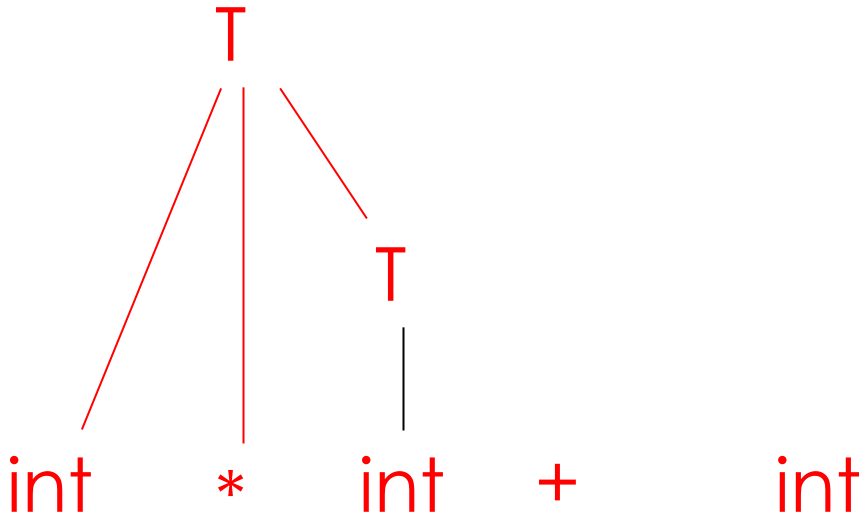
A Bottom-up Parse in Detail (3)

E → T + E | T
T → int * T | int | (E)

int * int + int

int * T + int

T + int



A Bottom-up Parse in Detail (4)

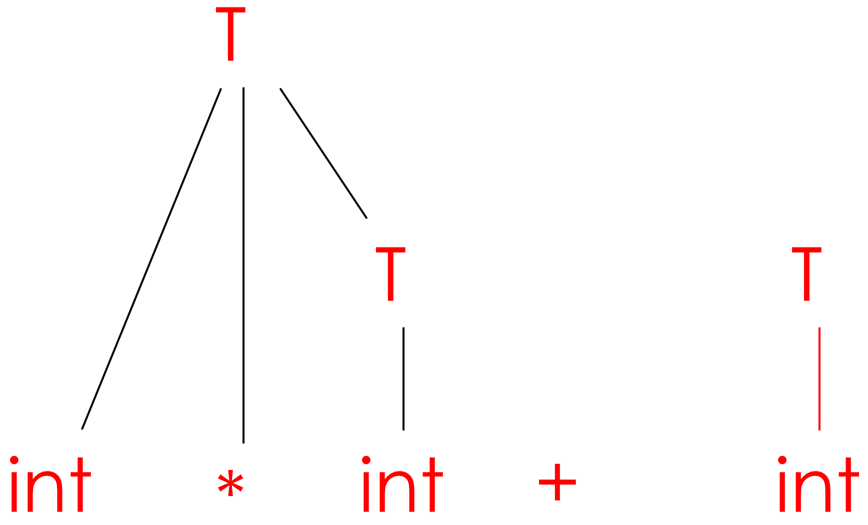
E → T + E | T
T → int * T | int | (E)

int * int + int

int * T + int

T + int

T + T



A Bottom-up Parse in Detail (5)

E → T + E | T
T → int * T | int | (E)

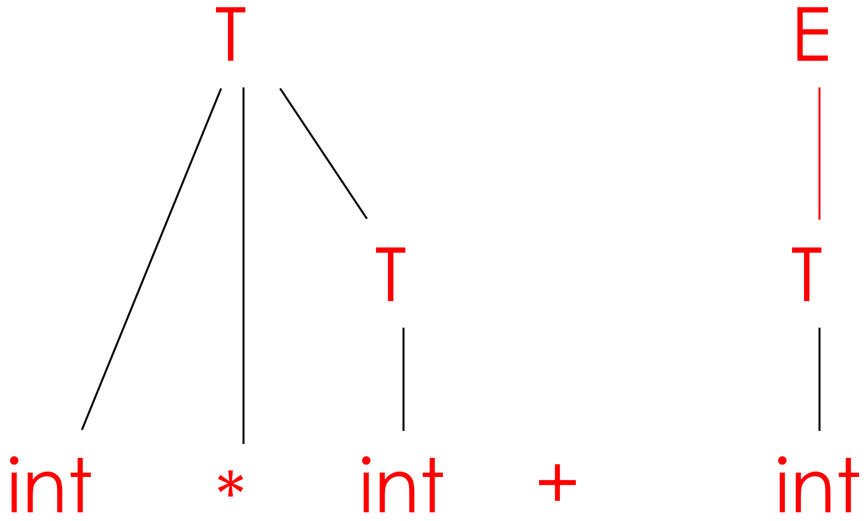
int * int + int

int * T + int

T + int

T + T

T + E



A Bottom-up Parse in Detail (6)

E → T + E | T
T → int * T | int | (E)

int * int + int

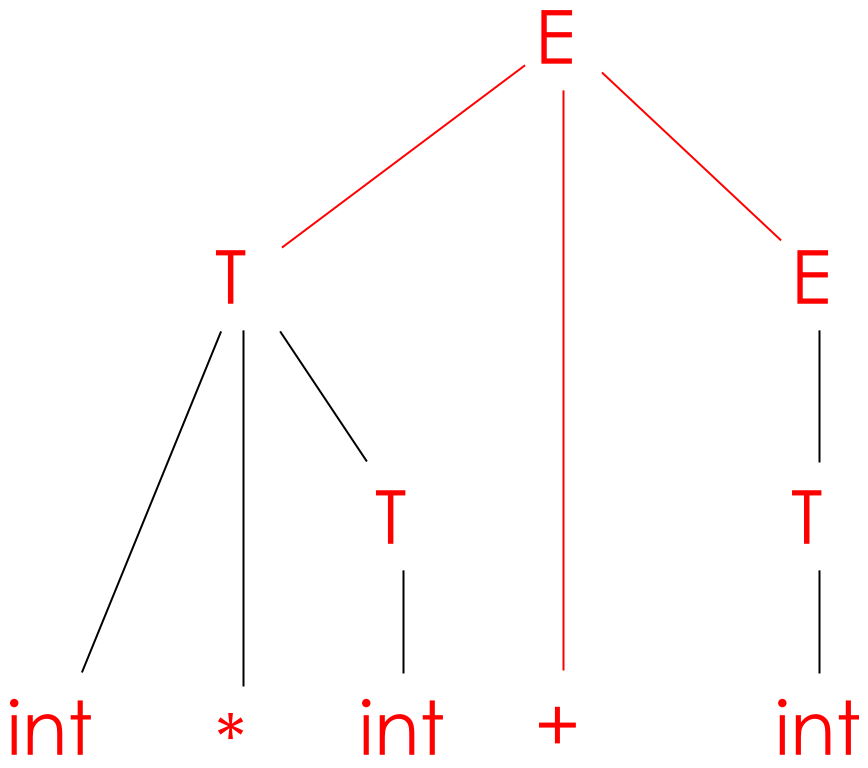
int * T + int

T + int

T + T

T + E

E



Where Do Reductions Happen?

Important Fact #1 has an interesting consequence:

- Let $\alpha\beta\omega$ be a step of a bottom-up parse
- Assume the next reduction is by $X \rightarrow \beta$
- Then ω is a string of terminals

Why? Because $\alpha X\omega \rightarrow \alpha\beta\omega$ is a step in a right-most derivation

Notation

- Idea: Split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - Left substring of sentential form has both terminals and non-terminals

- The dividing point is marked by a |
  - The | is not part of the string

- Initially, all input is unexamined $|x_1x_2 \ldots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift (Push onto a Stack)

Reduce (Apply a production rule)

Shift

- Shift: Move | one place to the right
  - – Shifts a terminal to the left string

ABC|xyz ⇒ ABCx|yz

Reduce

- Apply an inverse production at the right end of the left string
  - If $A \rightarrow xy$ is a production, then

$$\mathrm{Cbxy|ijk} \Rightarrow \mathrm{CbA|ijk}$$

The Example with Reductions Only

E → T + E | T
T → int * T | int | (E)

<table>
<tr><td>int * int | + int</td><td>reduce T → int</td></tr>
<tr><td>int * T | + int</td><td>reduce T → int * T</td></tr>
<tr><td>T + int |</td><td>reduce T → int</td></tr>
<tr><td>T + T |</td><td>reduce E → T</td></tr>
<tr><td>T + E |</td><td>reduce E → T + E</td></tr>
<tr><td>E |</td><td></td></tr>
</table>

The Example with Shift-Reduce Parsing

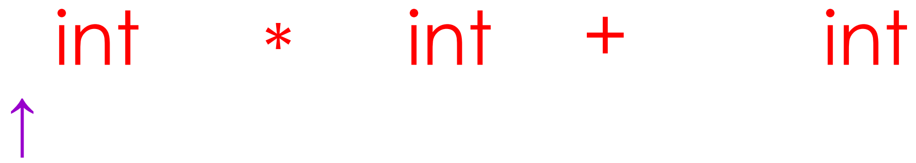
E → T + E | T
T → int * T | int | (E)

<table>
<tr><th></th><th></th></tr>
<tr><td>| int * int + int</td><td>shift</td></tr>
<tr><td>int | * int + int</td><td>shift</td></tr>
<tr><td>int * | int + int</td><td>shift</td></tr>
<tr><td>int * int | + int</td><td>reduce T → int</td></tr>
<tr><td>int * T | + int</td><td>reduce T → int * T</td></tr>
<tr><td>T | + int</td><td>shift</td></tr>
<tr><td>T + | int</td><td>shift</td></tr>
<tr><td>T + int |</td><td>reduce T → int</td></tr>
<tr><td>T + T |</td><td>reduce E → T</td></tr>
<tr><td>T + E |</td><td>reduce E → T + E</td></tr>
<tr><td>E |</td><td></td></tr>
</table>

A Shift-Reduce Parse in Detail (1)

E → T + E | T
T → int * T | int | (E)

| int * int + int



int * int + int

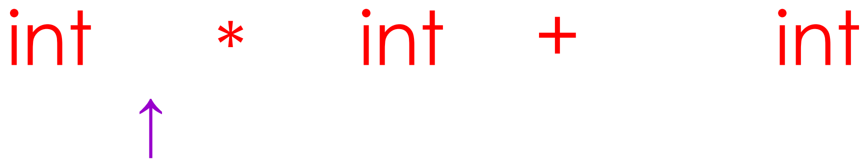
A Shift-Reduce Parse in Detail (2)

$E \rightarrow T + E \mid T$

$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$

| int * int + int

int | * int + int



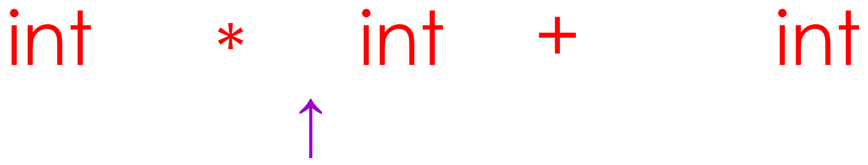
A Shift-Reduce Parse in Detail (3)

E → T + E | T
T → int * T | int | (E)

| int * int + int

int | * int + int

int * | int + int



A Shift-Reduce Parse in Detail (4)

E → T + E | T
T → int * T | int | (E)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * int + int

↑

A Shift-Reduce Parse in Detail (5)

E → T + E | T
T → int * T | int | (E)

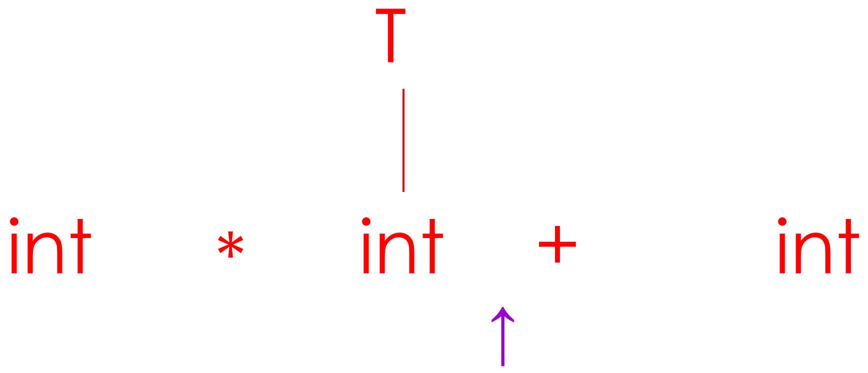
| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int



A Shift-Reduce Parse in Detail (6)

$E \rightarrow T + E \mid T$
$T \rightarrow int * T \mid int \mid (E)$

| int * int + int

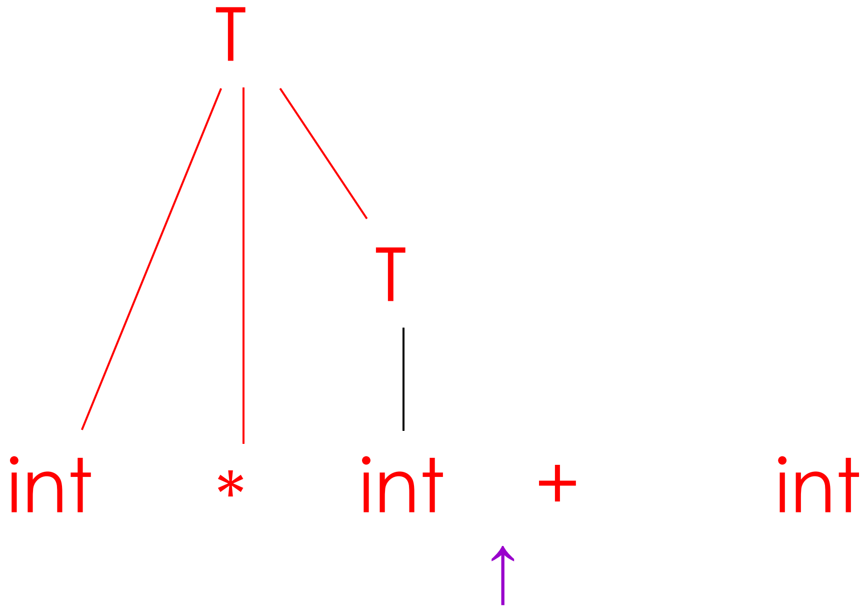
int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int



A Shift-Reduce Parse in Detail (7)

E → T + E | T
T → int * T | int | (E)

| int * int + int

int | * int + int

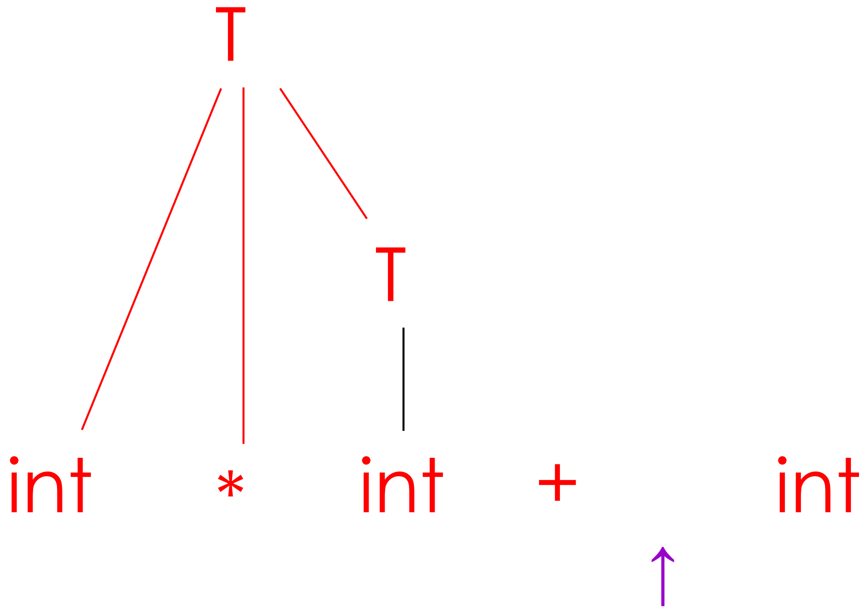
int * | int + int

int * int | + int

int * T | + int

T | + int

T + | int



A Shift-Reduce Parse in Detail (8)

E → T + E | T
T → int * T | int | (E)

| int * int + int

int | * int + int

int * | int + int

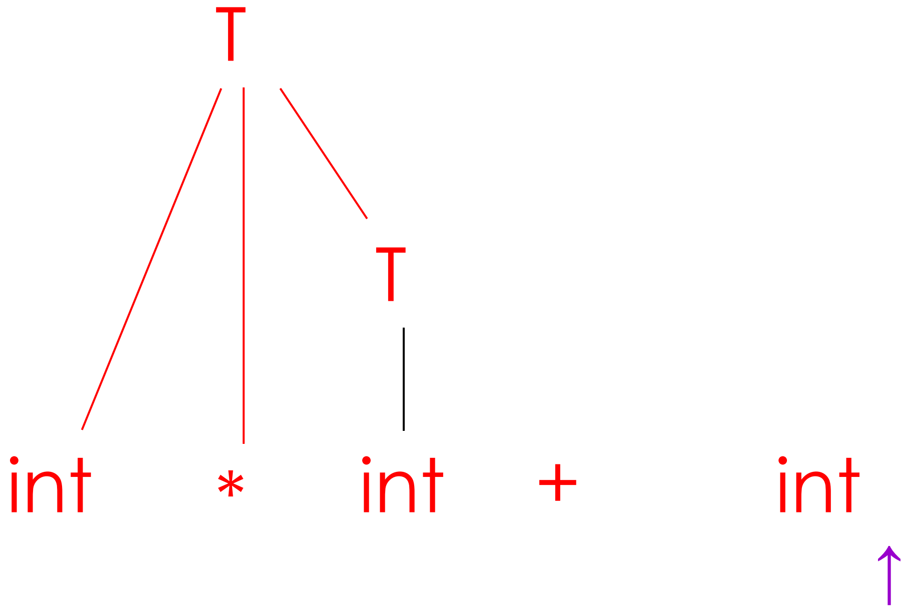
int * int | + int

int * T | + int

T | + int

T + | int

T + int |



A Shift-Reduce Parse in Detail (9)

E → T + E | T
T → int * T | int | (E)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

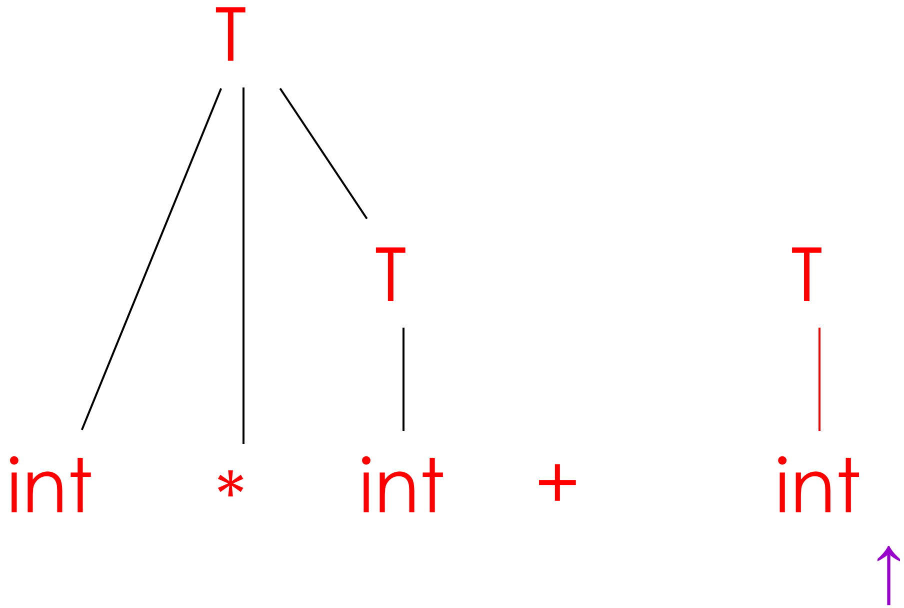
int * T | + int

T | + int

T + | int

T + int |

T + T |



A Shift-Reduce Parse in Detail (10)

E → T + E | T
T → int * T | int | (E)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

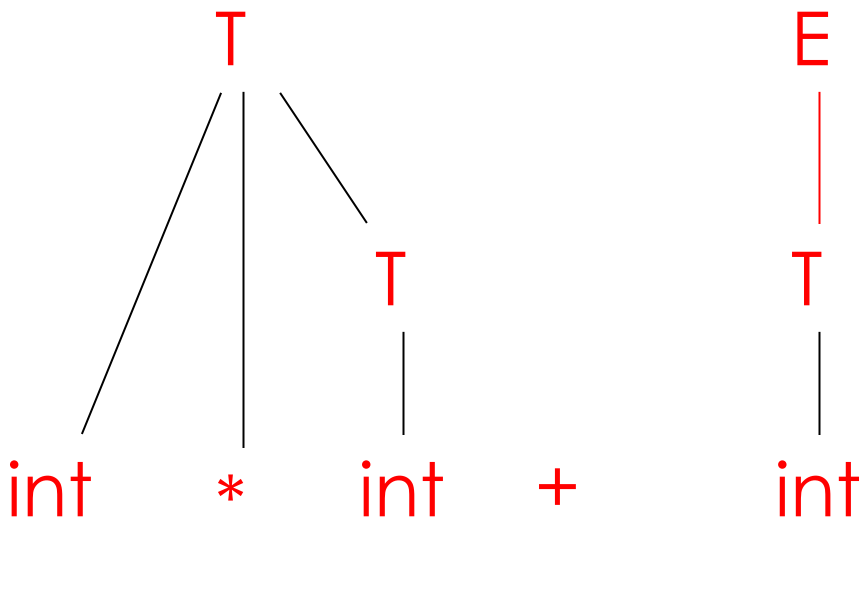
T | + int

T + | int

T + int |

T + T |

T + E |



A Shift-Reduce Parse in Detail (11)

E → T + E | T

T → int * T | int | (E)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int

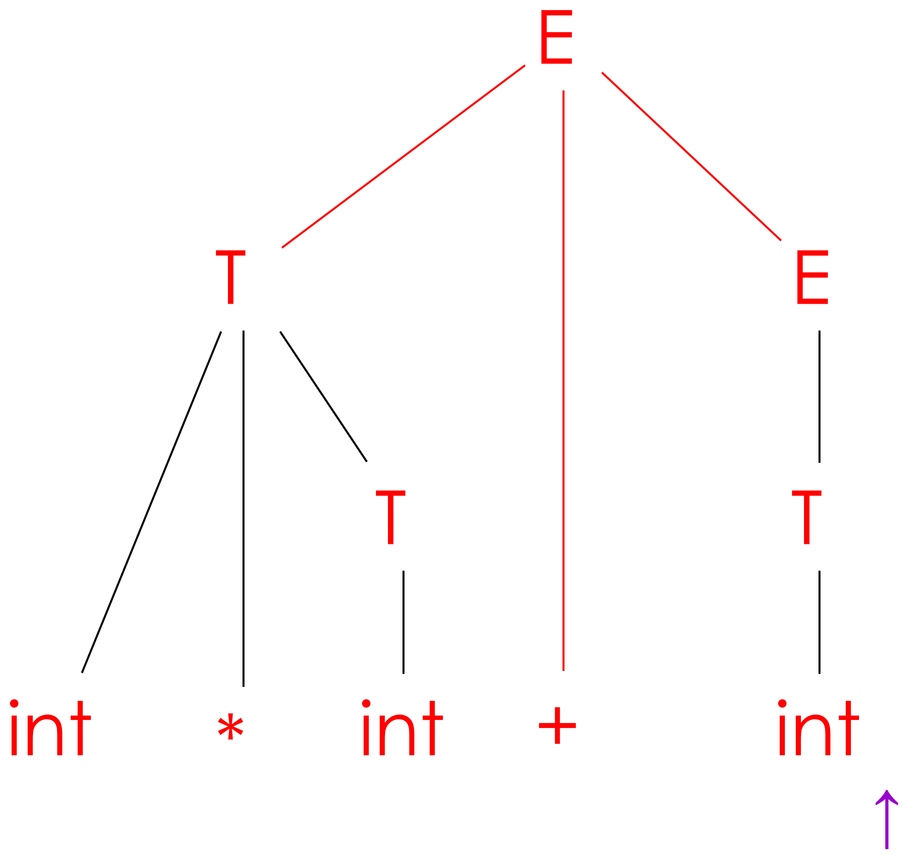
T + | int

T + int |

T + T |

T + E |

E |



The Stack

- Left string can be implemented by a stack
  - Top of the stack is the |

- **Shift pushes a terminal on the stack**

- **Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)**

Conflicts

- In a given state, more than one action (shift or reduce) may lead to a valid parse

- If it is legal to shift or reduce, there is a shift-reduce conflict

- If it is legal to reduce by two different productions, there is a reduce-reduce conflict

- When using a parser generator you may well see conflicts, that will have to be disambiguated.

Key Issue

- How do we decide when to shift or reduce?

- Example grammar:

  E → T + E | T
  T → int * T | int | (E)

- Consider step int | * int + int
  - We could reduce by T → int giving T | * int + int
  - A fatal mistake!
    - No way to reduce to the start symbol E

Definition: Handles

- Intuition: Want to reduce only if the result can still be reduced to the start symbol

- Assume a rightmost derivation

$$S \rightarrow^* \alpha X \omega \rightarrow \alpha \beta \omega$$

- Then $X \rightarrow \beta$ in the position after α is a handle of $\alpha\beta\omega$

- Can and must reduce at handles.

Handles (Cont.)

- Handles formalize the intuition
  - **A handle is a string that can be reduced and also allows further reductions back to the start symbol (using a particular production at a specific spot)**

- We only want to reduce at handles

- Note: We have said what a handle is, not how to find handles

Important Fact #2

Important Fact #2 about bottom-up parsing:

In shift-reduce parsing, handles appear only at the top of the stack, never inside

Why?

- Informal induction on # of reduce moves:

- True initially, stack is empty

- Immediately after reducing a handle
  - right-most non-terminal on top of the stack
  - next handle must be to right of right-most non-terminal, because this is a right-most derivation
  - Sequence of shift moves reaches next handle

Summary of Handles

- In shift-reduce parsing, handles always appear at the top of the stack

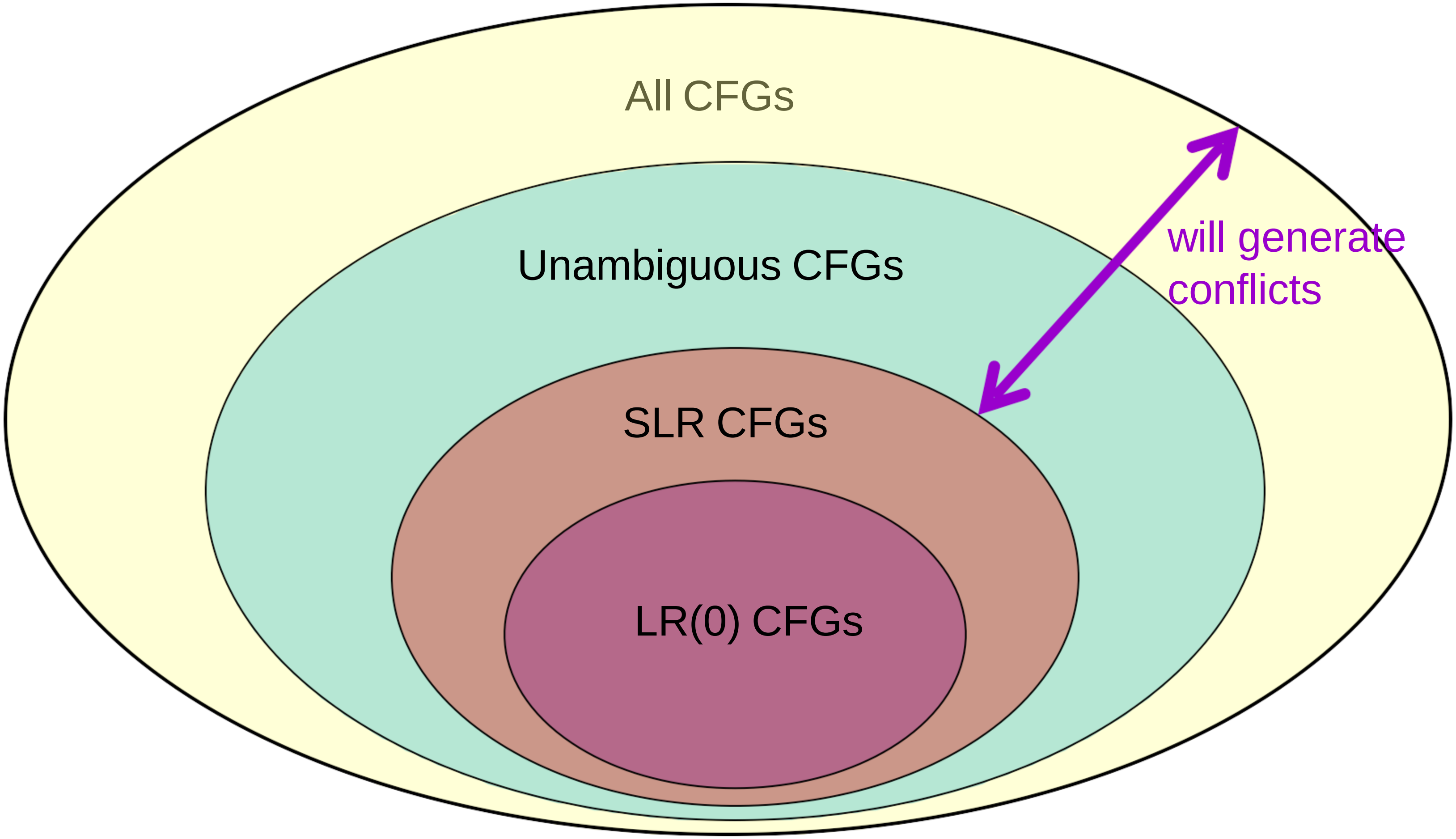
- **Parsing decisions** are made by scanning from **left to right**, but we only reduce **rightmost** valid production (handle). parser need not "look to the left" beyond what's on the stack.

- Bottom-up parsing algorithms are based on recognizing handles

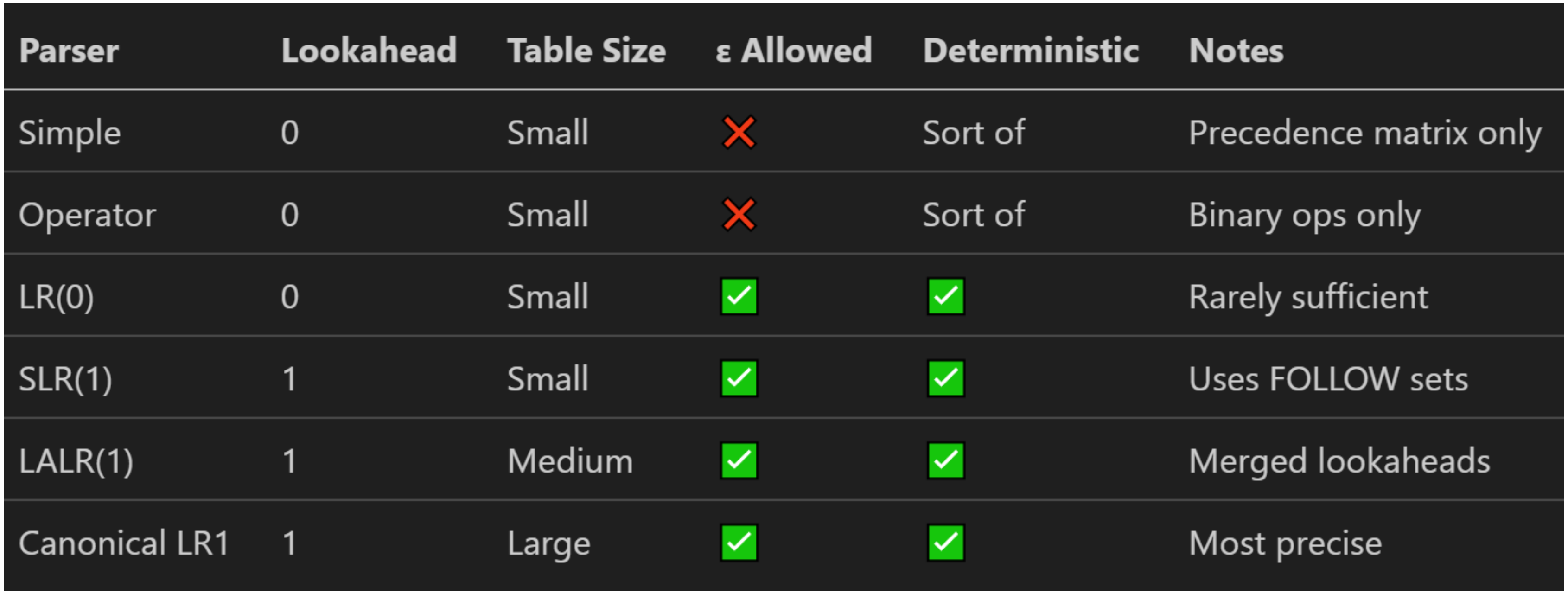
Recognizing Handles

- There are grammars with no known efficient algorithms to recognize handles
- Some CFGs, use proofs to guarantee either finding the handle, or determining its ambiguous conflict. These are known as: SLR(1), LALR(1), Canonical LR(1),
  - For the heuristics we use here, these are the SLR grammars
  - Other heuristics work for other grammars

Grammars


All CFGs
Unambiguous CFGs
SLR CFGs
LR(0) CFGs
will generate
conflicts

Categories of Bottom Up Grammars



Parser	Lookahead	Table Size	ε Allowed	Deterministic	Notes
Simple	0	Small	❌	Sort of	Precedence matrix only
Operator	0	Small	❌	Sort of	Binary ops only
LR(0)	0	Small	✅	✅	Rarely sufficient
SLR(1)	1	Small	✅	✅	Uses FOLLOW sets
LALR(1)	1	Medium	✅	✅	Merged lookaheads
Canonical LR1	1	Large	✅	✅	Most precise

Viable Prefixes

- It is not obvious how to detect handles

- At each step the parser sees only the stack, not the entire input; start with that . . .

α is a viable prefix if there is an ω such that
$\alpha|\omega$ is a state of a shift-reduce parser

Huh?

- What does this mean? A few things:
  - A viable prefix does not extend past the right end of the handle
  - It's a viable prefix because it is a prefix of the handle
  - As long as a parser has viable prefixes on the stack no parsing error has been detected

Important Fact #3

Important Fact #3 about bottom-up parsing:

Considering any (Simple) SLR(1) the set of viable prefixes is a regular language

Important Fact #3 (Cont.)

- Important Fact #3 is non-obvious
- We show how to compute automata that accept viable prefixes

Items

- An item is a production with a “.” somewhere on the rhs, denoting a focus point

- The items for T → (E) are

  T → .(E)
  T → (.E)
  T → (E.)
  T → (E).

Items (Cont.)

- The only item for $X \to \varepsilon$ is $X \to .$

- Items are often called “LR(0) items”

Intuition

- The problem in recognizing viable prefixes is that the stack has only bits and pieces of the rhs of productions
  - If it had a complete rhs, we could reduce

- These bits and pieces are always prefixes of rhs of productions

$E \to T + E \mid T$
$T \to int * T \mid int \mid (E)$

Example

Consider the input (int)

- Then (E |) is a state of a shift-reduce parse
- (E is a prefix of the rhs of $T \to (E)$
  - Will be reduced after the next shift
- Item $T \to (E.)$ says that so far we have seen (E of this production and hope to see)

Generalization

- The stack may have many prefixes of rhs's $\text{Prefix}_1\ \text{Prefix}_2 \ldots \text{Prefix}_{n-1}\ \text{Prefix}_n$

- Let Prefix_i be a prefix of rhs of $X_i \rightarrow \alpha_i$
  - Prefix_i will eventually reduce to X_i
  - The missing part of Prefix_{i-1} of α_{i-1} starts with X_i
  - i.e. there is a $X_{i-1} \rightarrow \text{Prefix}_{i-1}\ X_i\ \beta$ for some β

- Recursively, $\text{Prefix}_{k+1} \ldots \text{Prefix}_n$ eventually reduces to the missing part of α_k

E → T + E | T
T → int * T | int | (E)

An Example

Consider the string (int * int):

(int * | int) is a state of a shift-reduce parse

From top of the stack:

"ε" is a prefix of the rhs of E → T

"(" is a prefix of the rhs of T → (E)

"ε" is a prefix of the rhs of E → T

"int *" is a prefix of the rhs of T → int * T

E → T + E | T
T → int * T | int | (E)

An Example (Cont.)

The stack of items

T → int * .T

E → .T

T → (.E)

Says

We’ve seen int * of T → int * T

We’ve seen ε of E → T

We’ve seen (of T → (E)

Recognizing Viable Prefixes

Idea: To recognize viable prefixes, we must

- Recognize a sequence of partial rhs’s of productions, where
- Each sequence can eventually reduce to part of the missing suffix of its predecessor

An NFA Recognizing Viable Prefixes

1. Add a new start production $S' \rightarrow S$ to G

2. The NFA states are the items of G
   - (Including the new start production)

3. For item $E \rightarrow \alpha.X\beta$ add transition

$$E \rightarrow \alpha.X\beta \quad \rightarrow^{X} \quad E \rightarrow \alpha X.\beta$$

4. For item $E \rightarrow \alpha.X\beta$ and production $X \rightarrow \gamma$ add

$$E \rightarrow \alpha.X\beta \quad \rightarrow^{\varepsilon} \quad X \rightarrow .\gamma$$

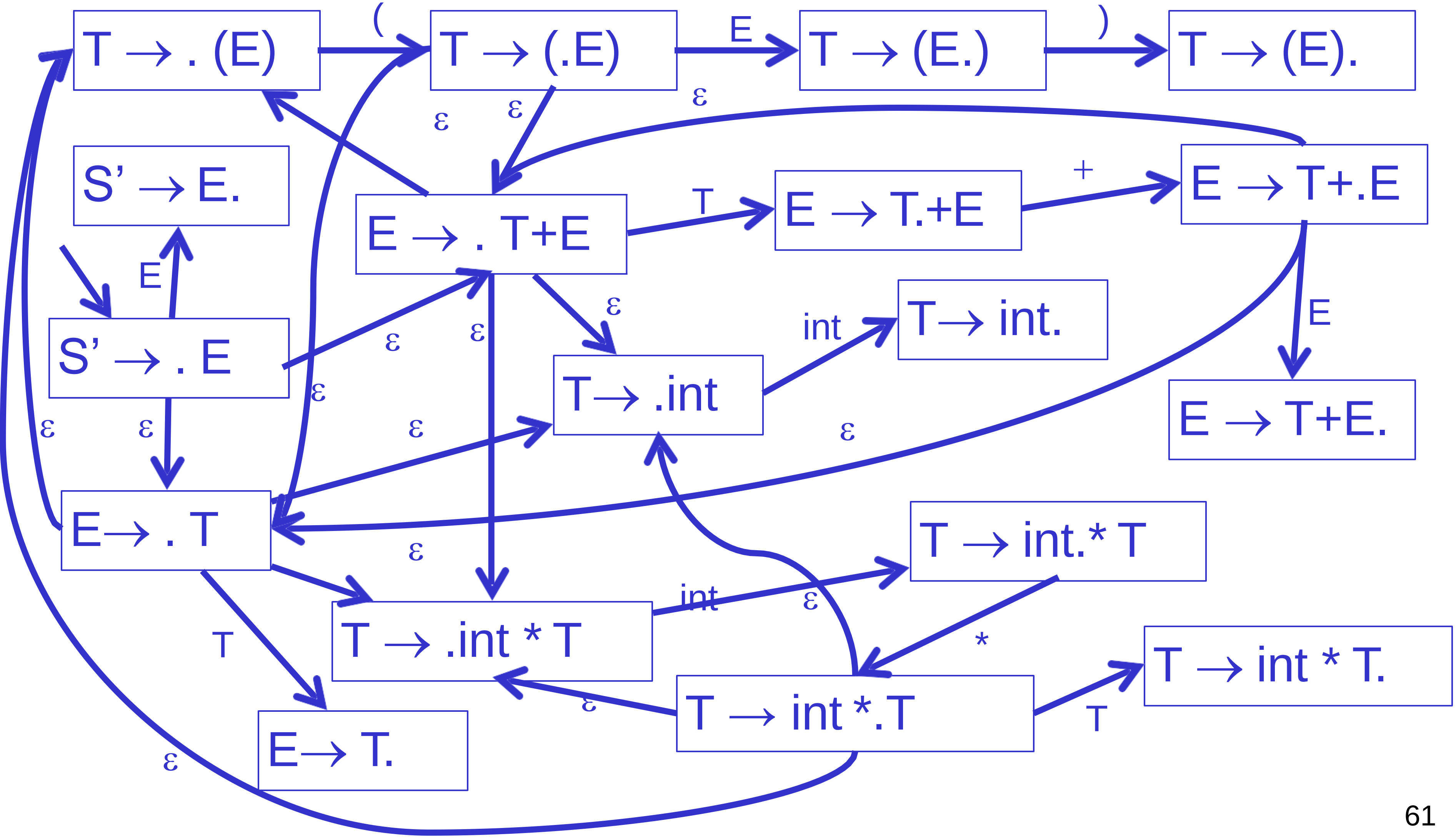
An NFA Recognizing Viable Prefixes (Cont.)

5. Every state is an accepting state

6. Start state is $S' \rightarrow .S$

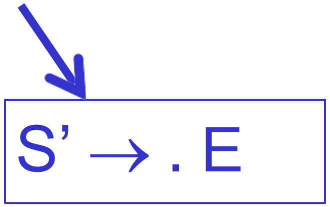
E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



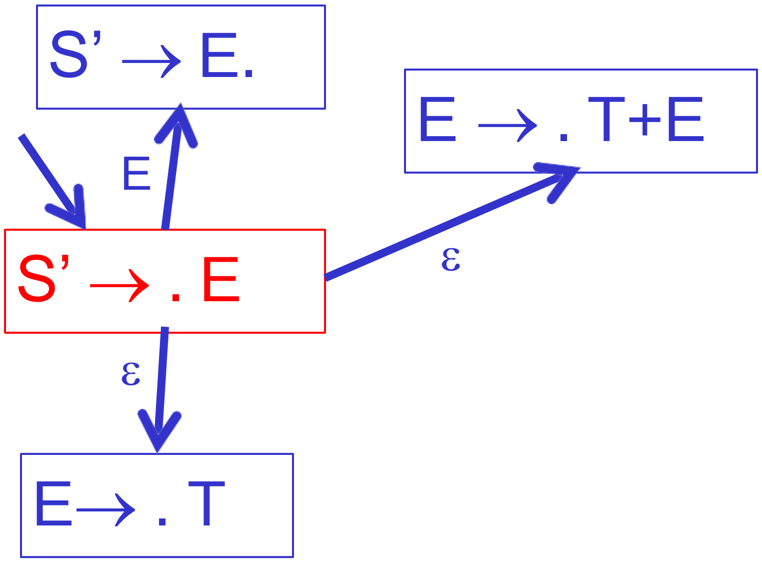
E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



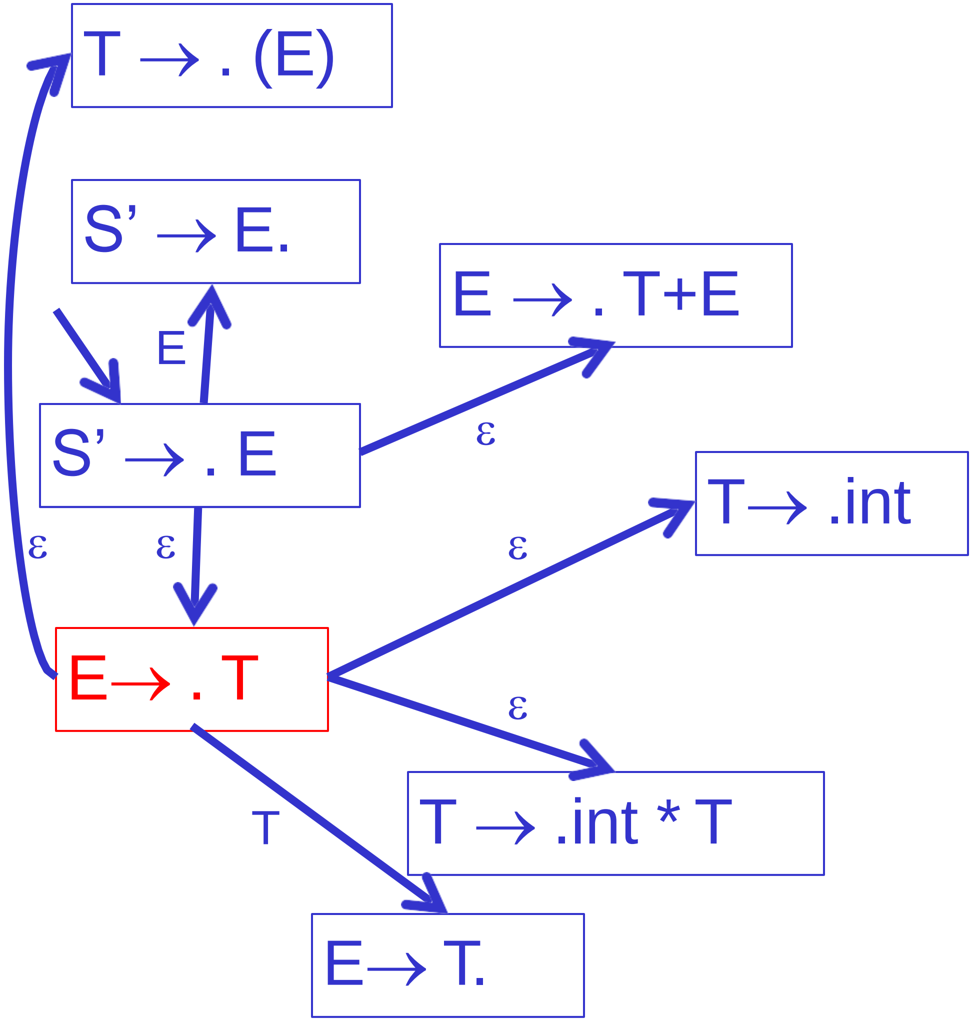
E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



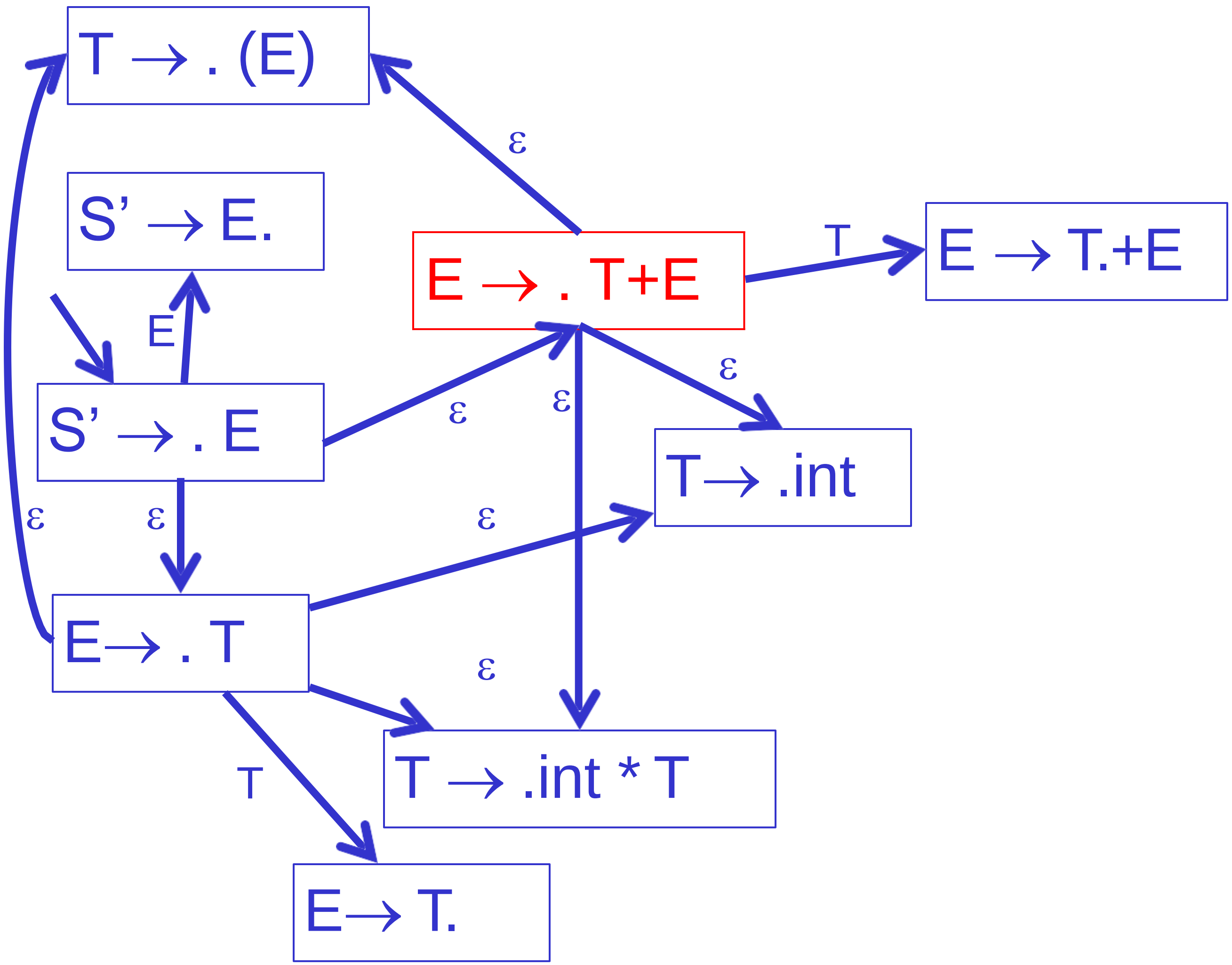
E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



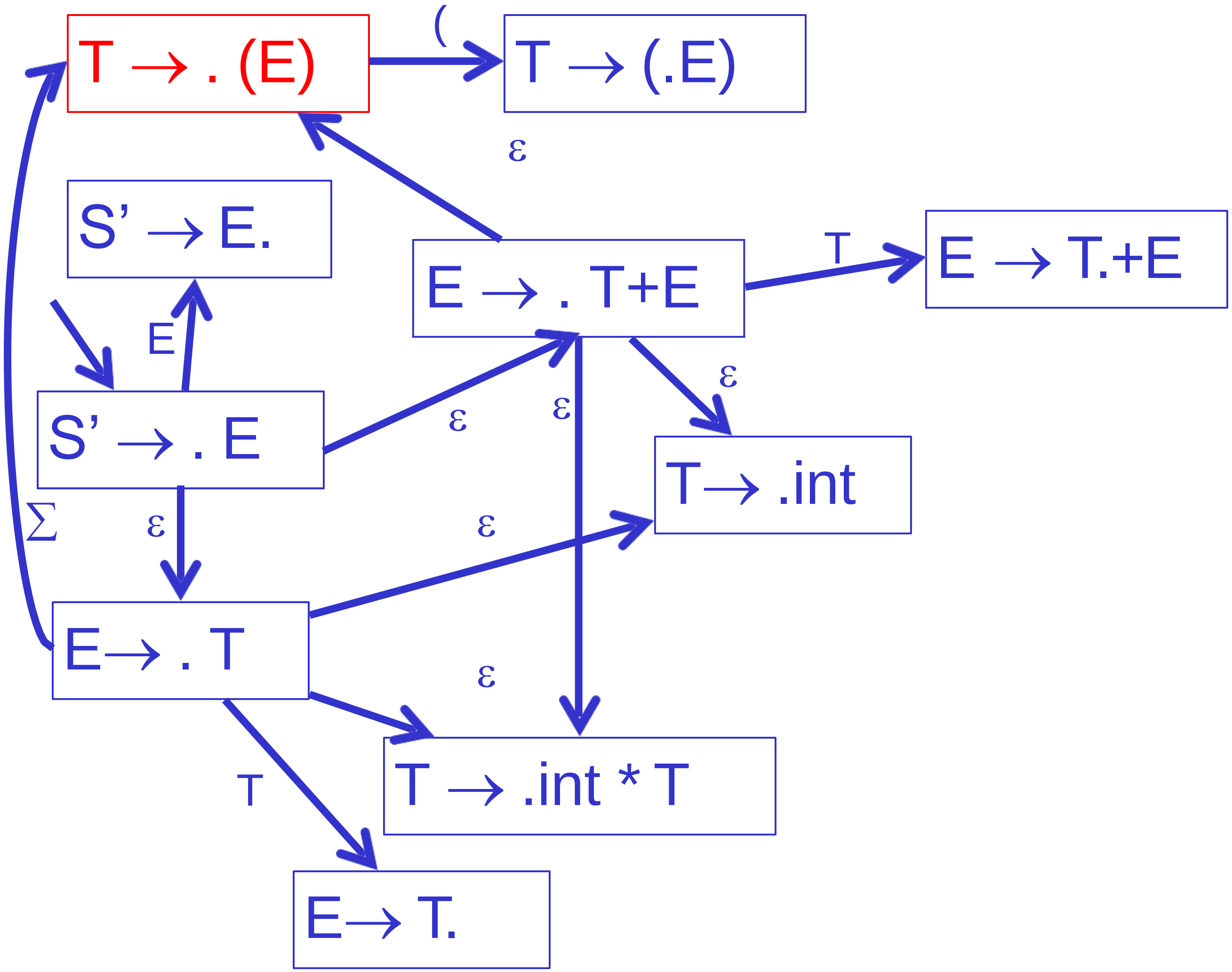
E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



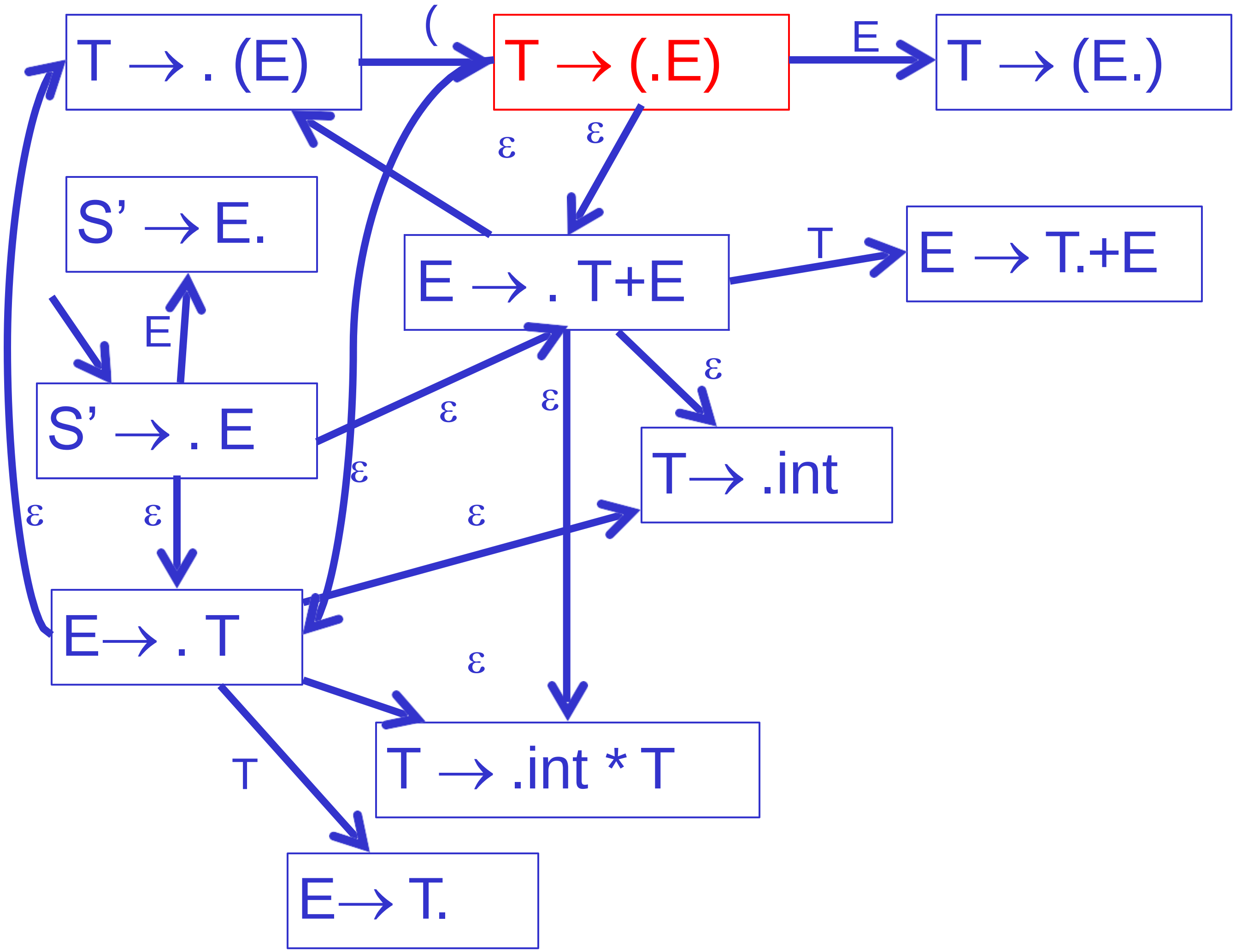
E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



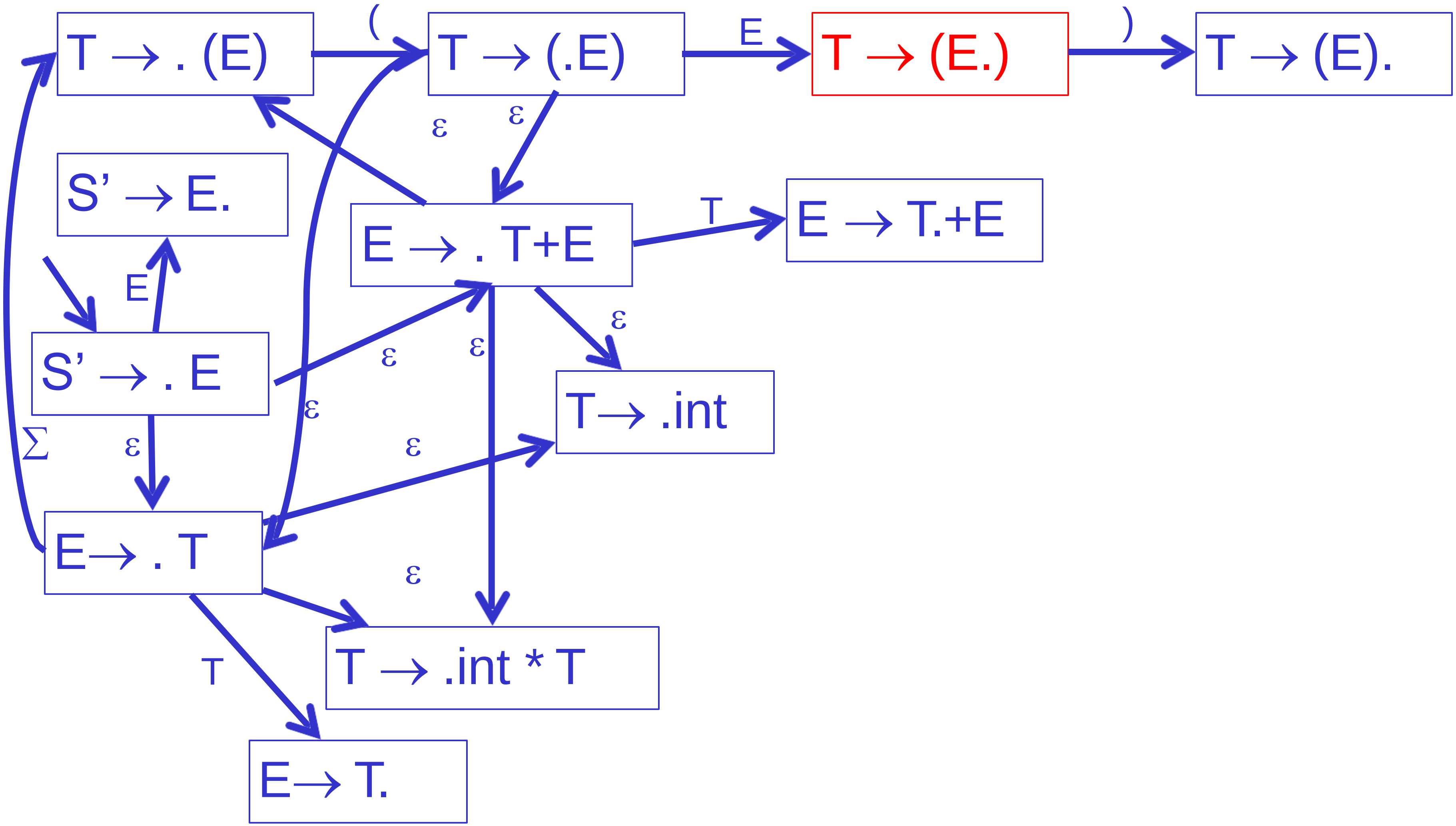
E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



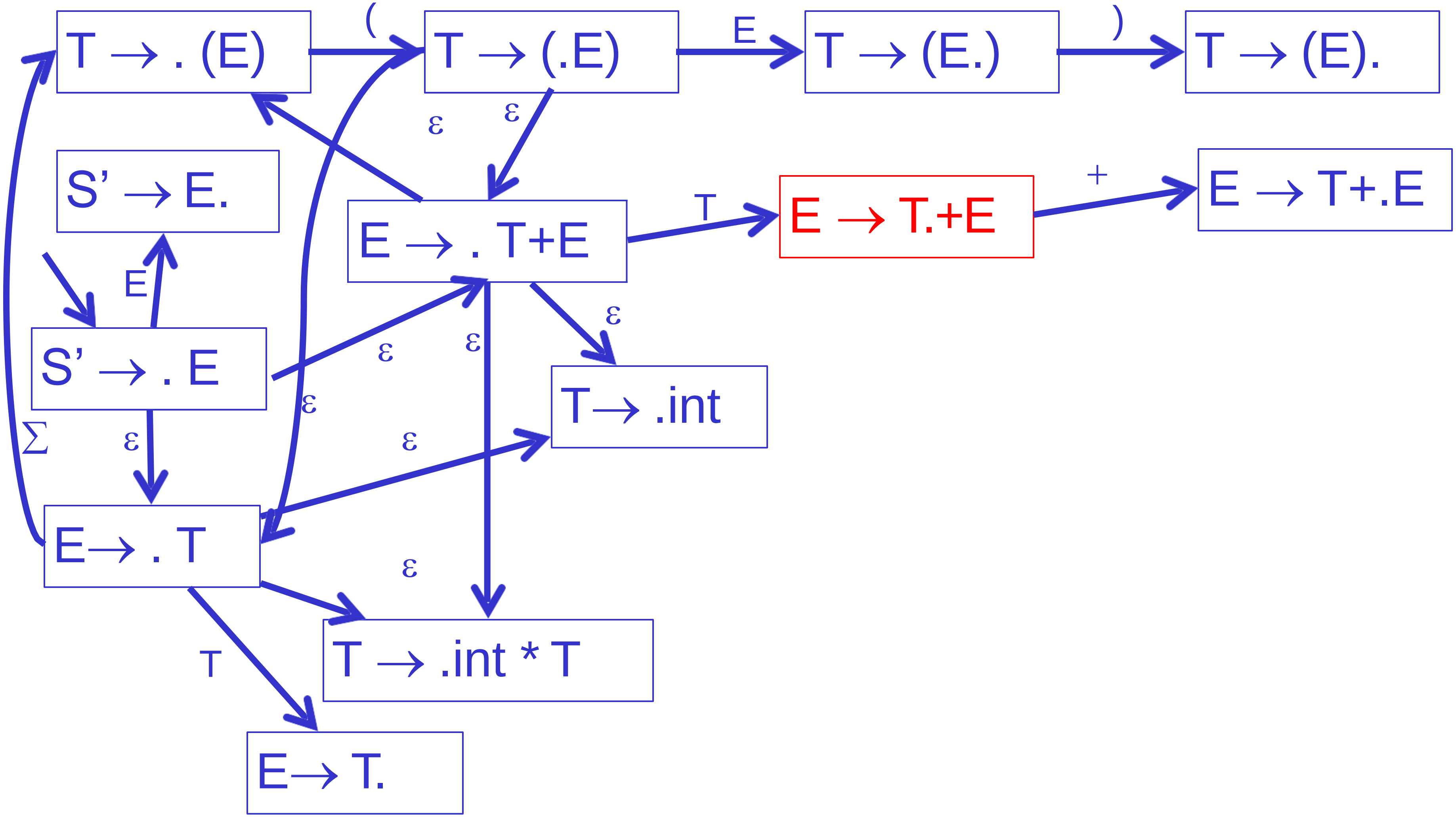
E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



T → . (E)

T → (.E)

T → (E.)

T → (E).

S’ → E.

E → . T+E

E → T.+E

E → T+.E

S’ → . E

T→ .int

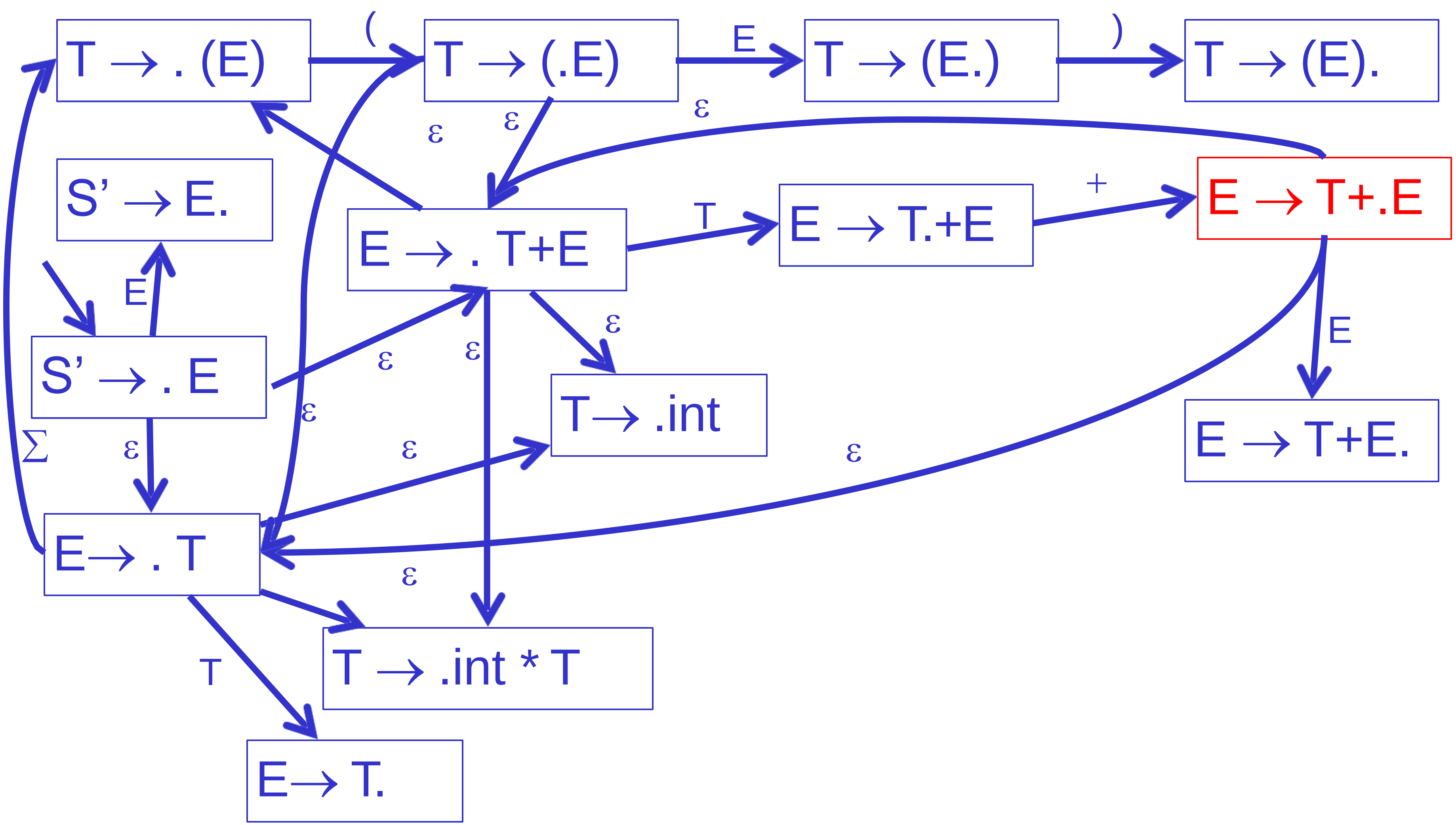
E→ . T

T → .int * T

E→ T.

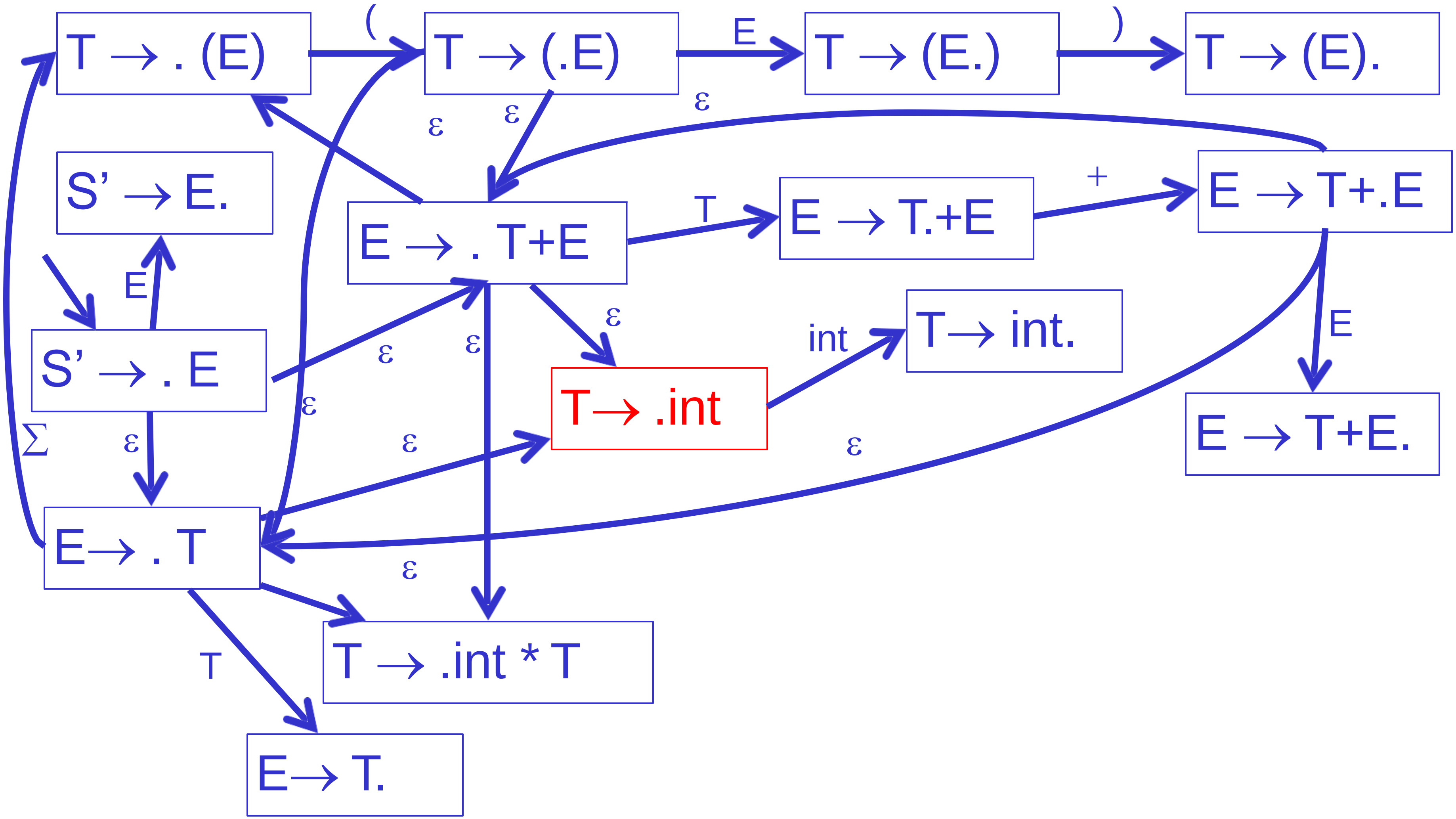
E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



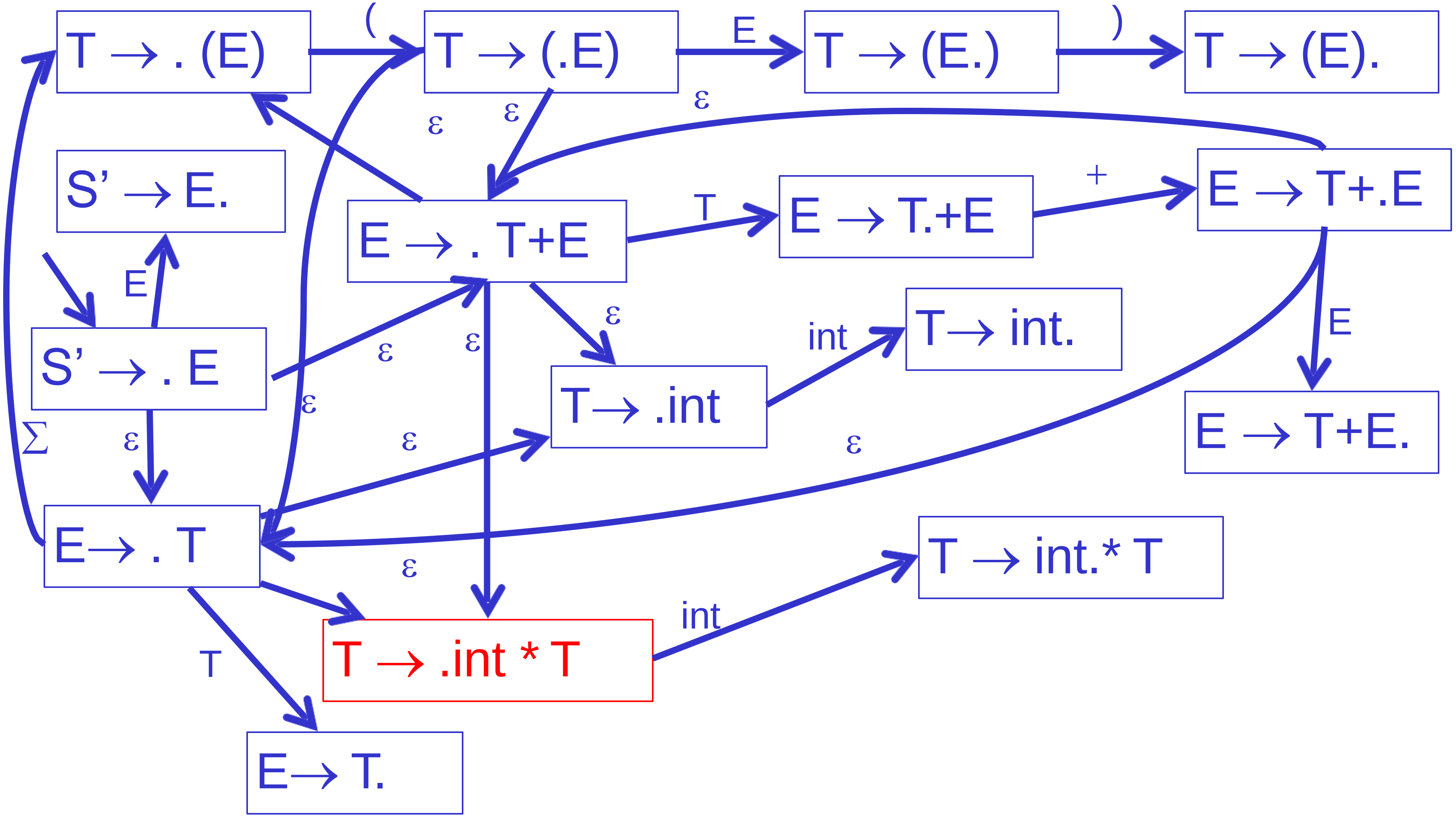
E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



T → . (E)

T → (.E)

T → (E.)

T → (E).

S’ → E.

E → . T+E

E → T.+E

E → T+.E

T→ int.

S’ → . E

T→ .int

E → T+E.

E→ . T

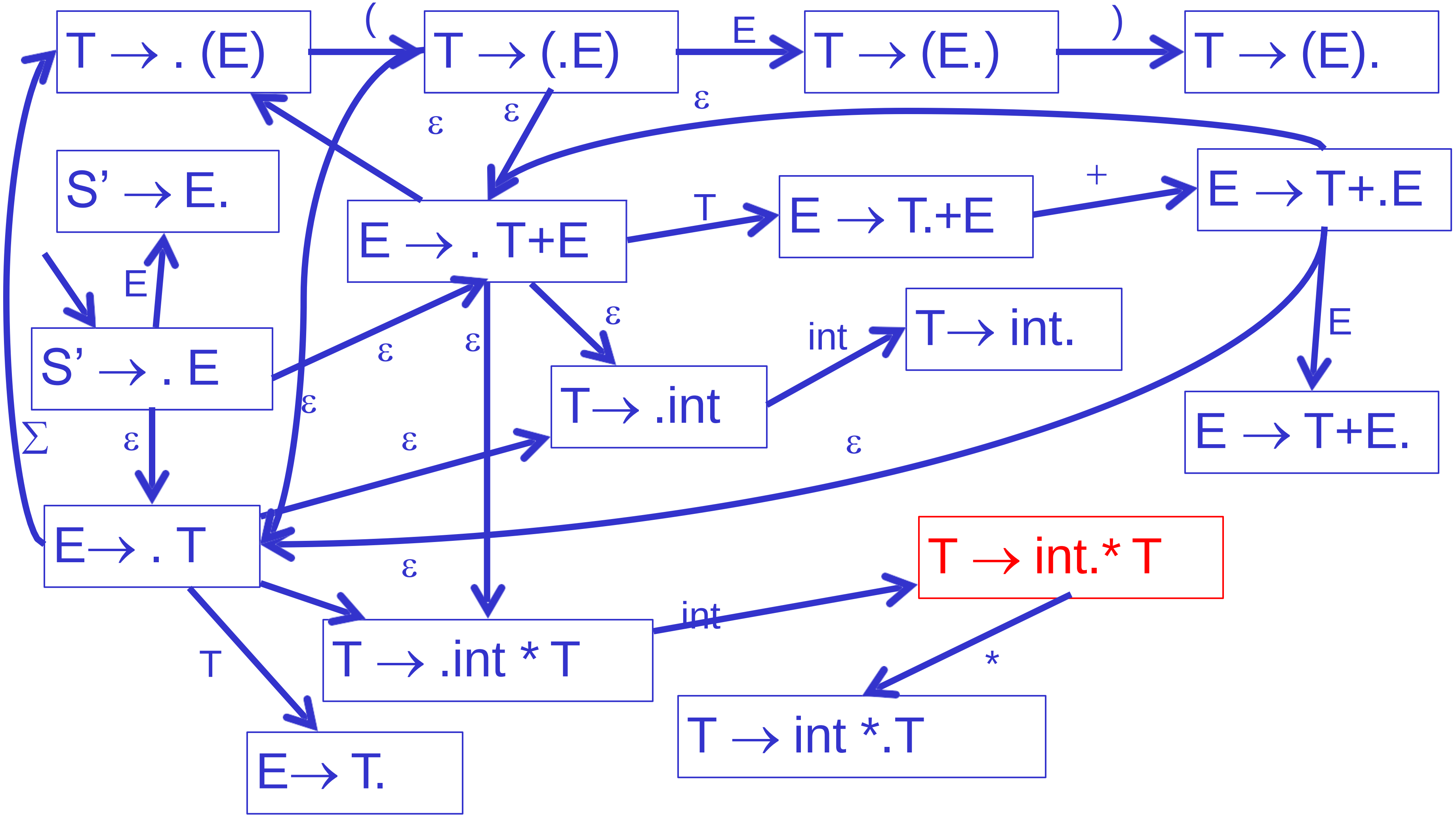
T → int.* T

T → .int * T

E→ T.

E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



T → . (E)

T → (.E)

T → (E.)

T → (E).

S’ → E.

E → . T+E

E → T.+E

E → T+.E

T→ int.

S’ → . E

T→ .int

E → T+E.

E→ . T

T → int.* T

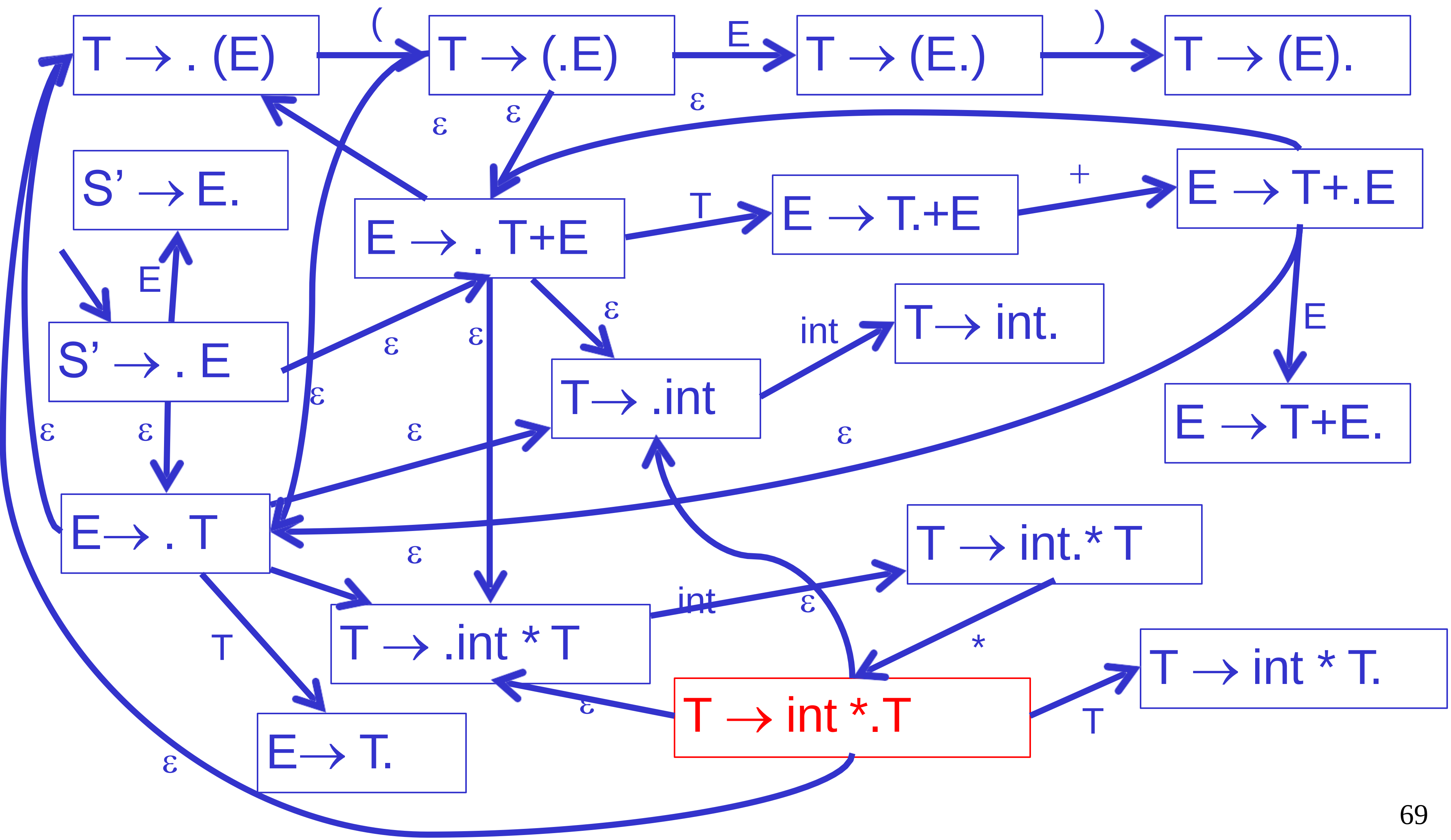
T → .int * T

T → int *.T

E→ T.

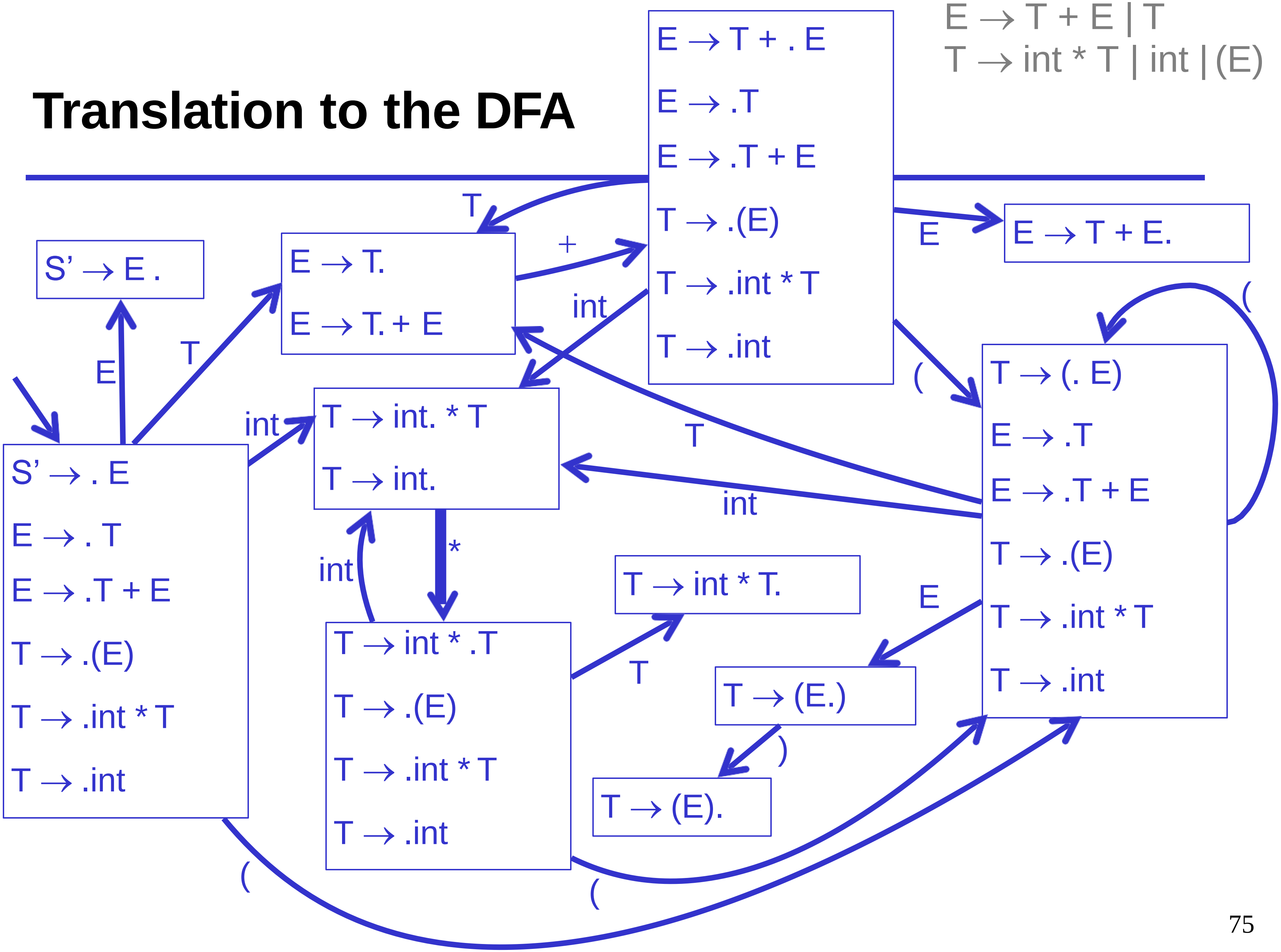
E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



Translation to the DFA

E → T + E | T
T → int * T | int | (E)



S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

S' → E .

E → T.
E → T. + E

E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

E → T + E.

T → int. * T
T → int.

T → int * .T
T → .(E)
T → .int * T
T → .int

T → int * T.

T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

T → (E.)

T → (E).

Lingo

The states of the DFA are

“canonical collections of items”

or

“canonical collections of LR(0) items”

The Dragon book gives another way of constructing LR(0) items

Valid Items

Item $X \to \beta.\gamma$ is valid for a viable prefix $\alpha\beta$ if

$$S' \to^* \alpha X \omega \to \alpha\beta\gamma\omega$$

by a right-most derivation

After parsing $\alpha\beta$, the valid items are the possible tops of the stack of items

Items Valid for a Prefix

An item I is valid for a viable prefix α if the DFA recognizing viable prefixes terminates on input α in a state s containing I

The items in s describe what the top of the item stack might be after reading input α

Valid Items Example

- An item is often valid for many prefixes

- Example: The item T → (.E) is valid for prefixes

(

((

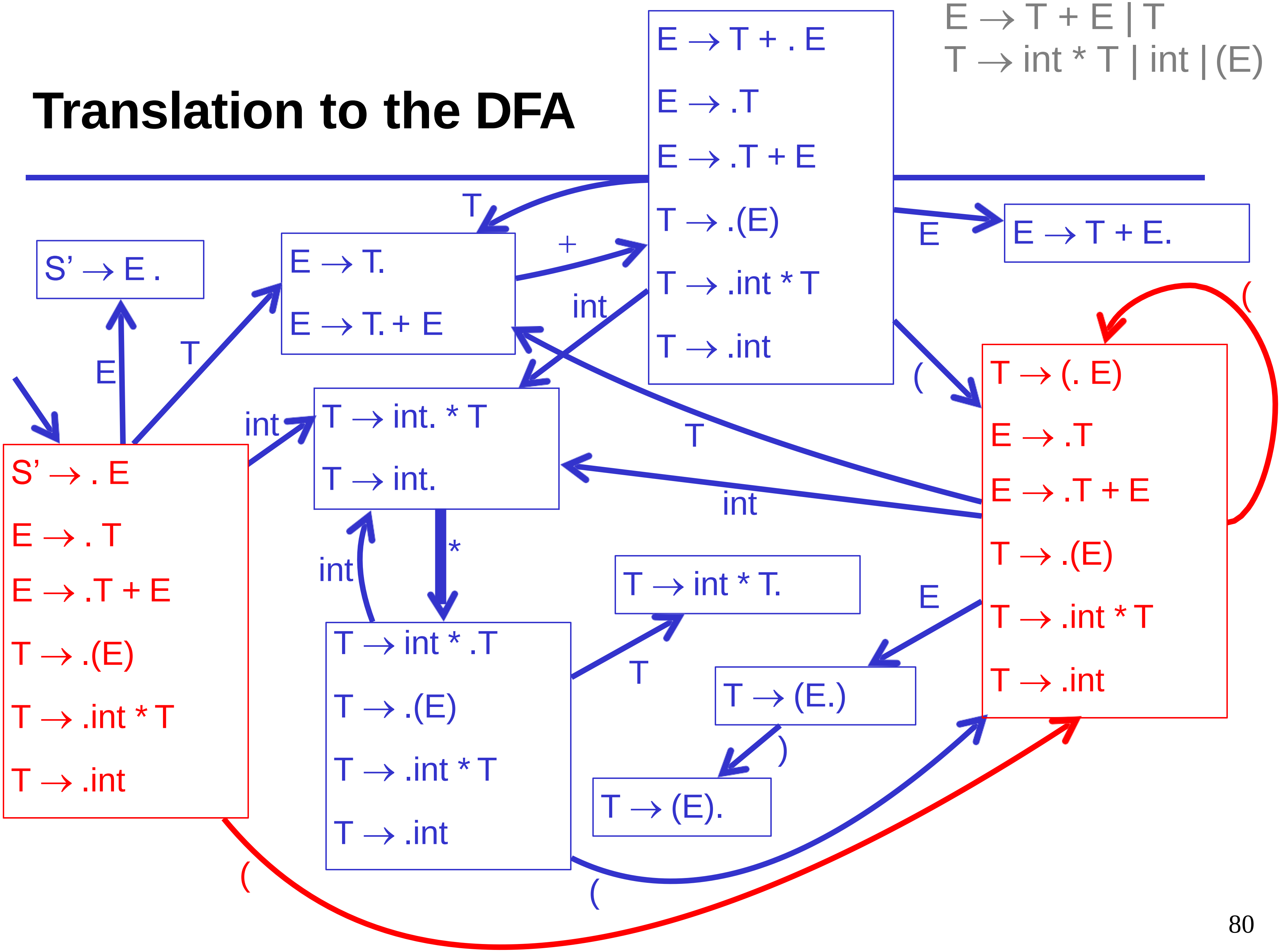
(((

((((

. . .

Translation to the DFA

E → T + E | T
T → int * T | int | (E)



S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

S' → E .

E → T.
E → T. + E

E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

E → T + E.

T → int. * T
T → int.

T → int * .T
T → .(E)
T → .int * T
T → .int

T → int * T.

T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

T → (E.)

T → (E).

LR(0) Parsing

- Idea: Assume
  - stack contains α
  - next input is t
  - DFA on input α terminates in state s
- Reduce by $X \to \beta$ if
  - s contains item $X \to \beta.$
- Shift if
  - s contains item $X \to \beta.t\omega$
  - equivalent to saying s has a transition labeled t

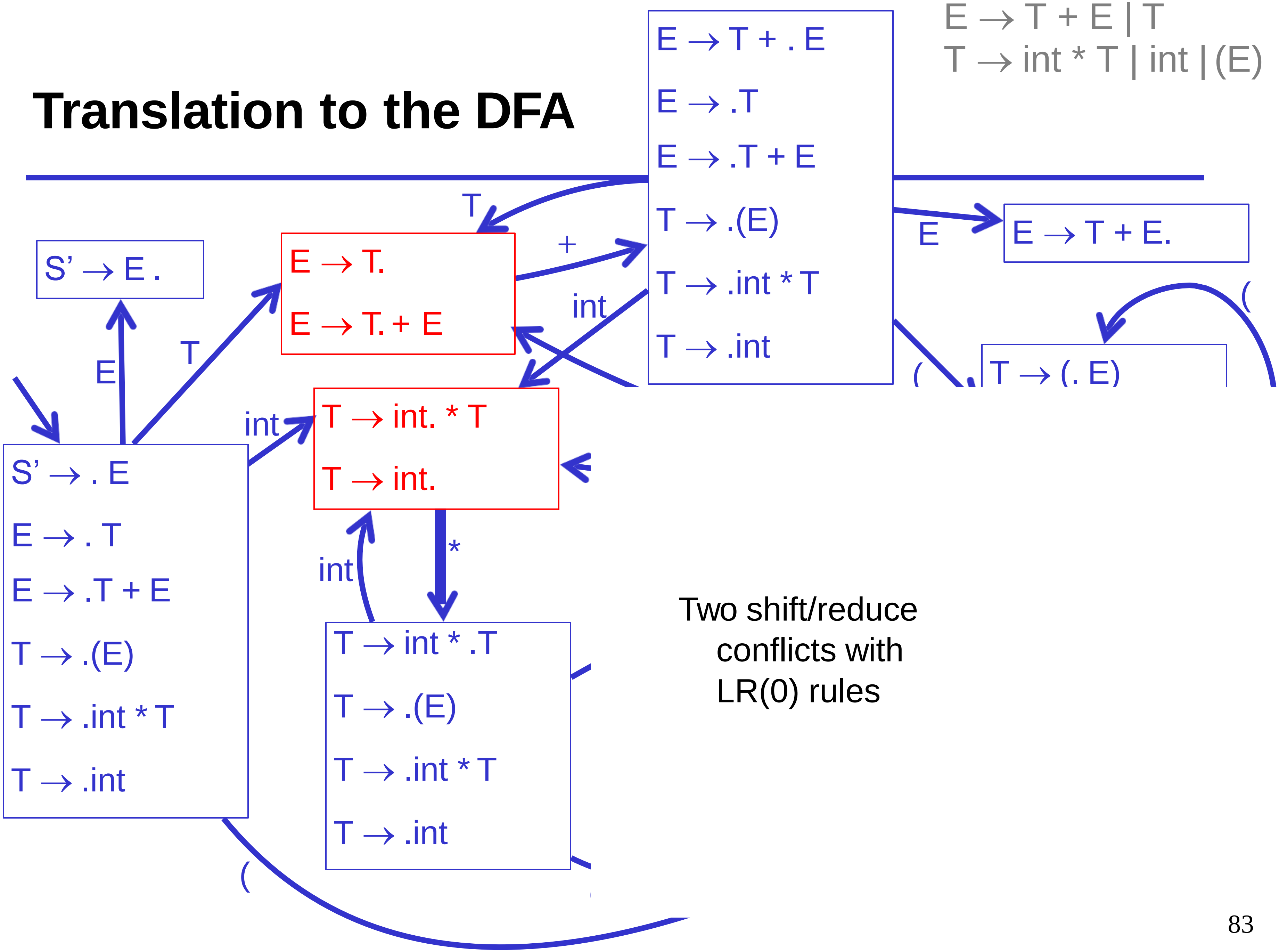
LR(0) Conflicts

- LR(0) has a reduce/reduce conflict if:
  - Any state has two reduce items:
  - $X \rightarrow \beta.$ and $Y \rightarrow \omega.$

- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item:
  - $X \rightarrow \beta.$ and $Y \rightarrow \omega.t\delta$

Translation to the DFA

E → T + E | T
T → int * T | int | (E)



S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

S' → E .

E → T.
E → T. + E

T → int. * T
T → int.

T → int * .T
T → .(E)
T → .int * T
T → .int

E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

E → T + E.

T → (. E)

Two shift/reduce conflicts with LR(0) rules

SLR

- LR = “Left-to-right scan”
- SLR = “Simple LR”

- SLR improves on LR(0) shift/reduce heuristics
  - Fewer states have conflicts

SLR Parsing

- Idea: Assume
  - stack contains α
  - next input is t
  - DFA on input α terminates in state s
- Reduce by $X \to \beta$ if
  - s contains item $X \to \beta.$
  - $t \in \text{Follow}(X)$
- Shift if
  - s contains item $X \to \beta.t\omega$

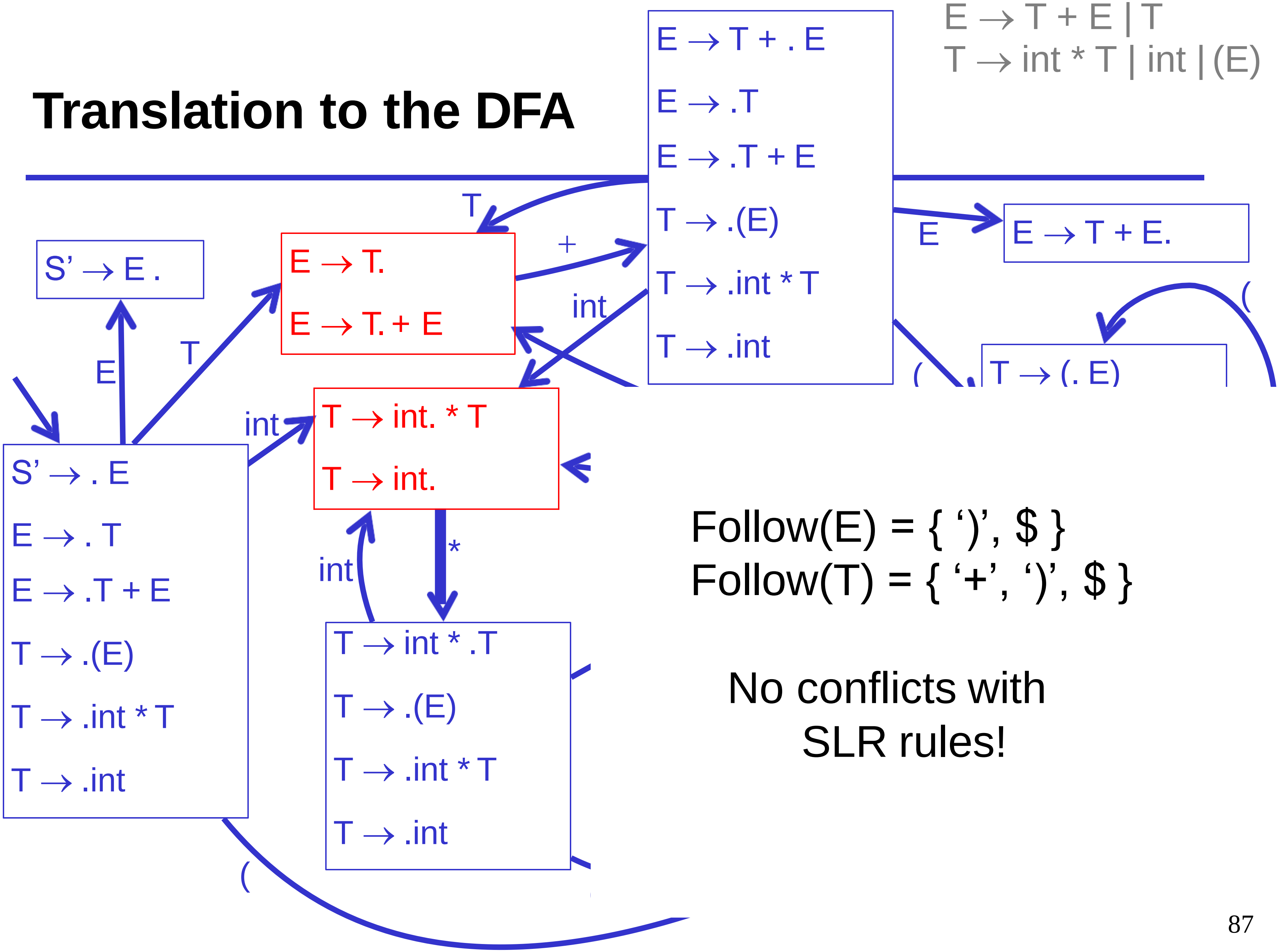
SLR Parsing (Cont.)

- If there are conflicts under these rules, the grammar is not SLR

- The rules amount to a heuristic for detecting handles
  - The SLR grammars are those where the heuristics detect exactly the handles

Translation to the DFA

E → T + E | T
T → int * T | int | (E)



S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

S' → E .

E → T.
E → T. + E

T → int. * T
T → int.

T → int * .T
T → .(E)
T → .int * T
T → .int

E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

E → T + E.

T → (. E)

Follow(E) = { ‘)’, $ }
Follow(T) = { ‘+’, ‘)’, $ }

No conflicts with SLR rules!

Precedence Declarations Digression

- Lots of grammars aren’t SLR
  - including all ambiguous grammars

- We can parse more grammars by using precedence declarations
  - Instructions for resolving conflicts

Precedence Declarations (Cont.)

- Consider our favorite ambiguous grammar:
  - E → E + E | E * E | (E) | int
- The DFA for this grammar contains a state with the following items:
  - E → E * E .   E → E . + E
  - shift/reduce conflict!
- Declaring "* has higher precedence than +" resolves this conflict in favor of reducing

Precedence Declarations (Cont.)

- The term “precedence declaration” is misleading
- These declarations do not define precedence; they define conflict resolutions
  - Not quite the same thing!

Unoptimized SLR Parsing Algorithm

1. Let M be DFA for viable prefixes of G
2. Let $|x_1 \ldots x_n\$$ be initial configuration
3. Repeat until configuration is $S|\$$
   - Let $\alpha|\omega$ be current configuration
   - Run M on current stack α
   - If M rejects α, report parsing error
     - Stack α is not a viable prefix
   - If M accepts α with items I, let t be next input
     - Reduce if $X \rightarrow \beta. \in I$ and $t \in \text{Follow}(X)$
     - Otherwise, shift if $X \rightarrow \beta.\, t\, \gamma \in I$
     - Report parsing error if neither applies

Notes

- If there is a conflict in the last step, grammar is not SLR(k)

- k is the amount of lookahead
  - In practice k = 1

- Will skip using extra start state S’ in following example to save space on slides

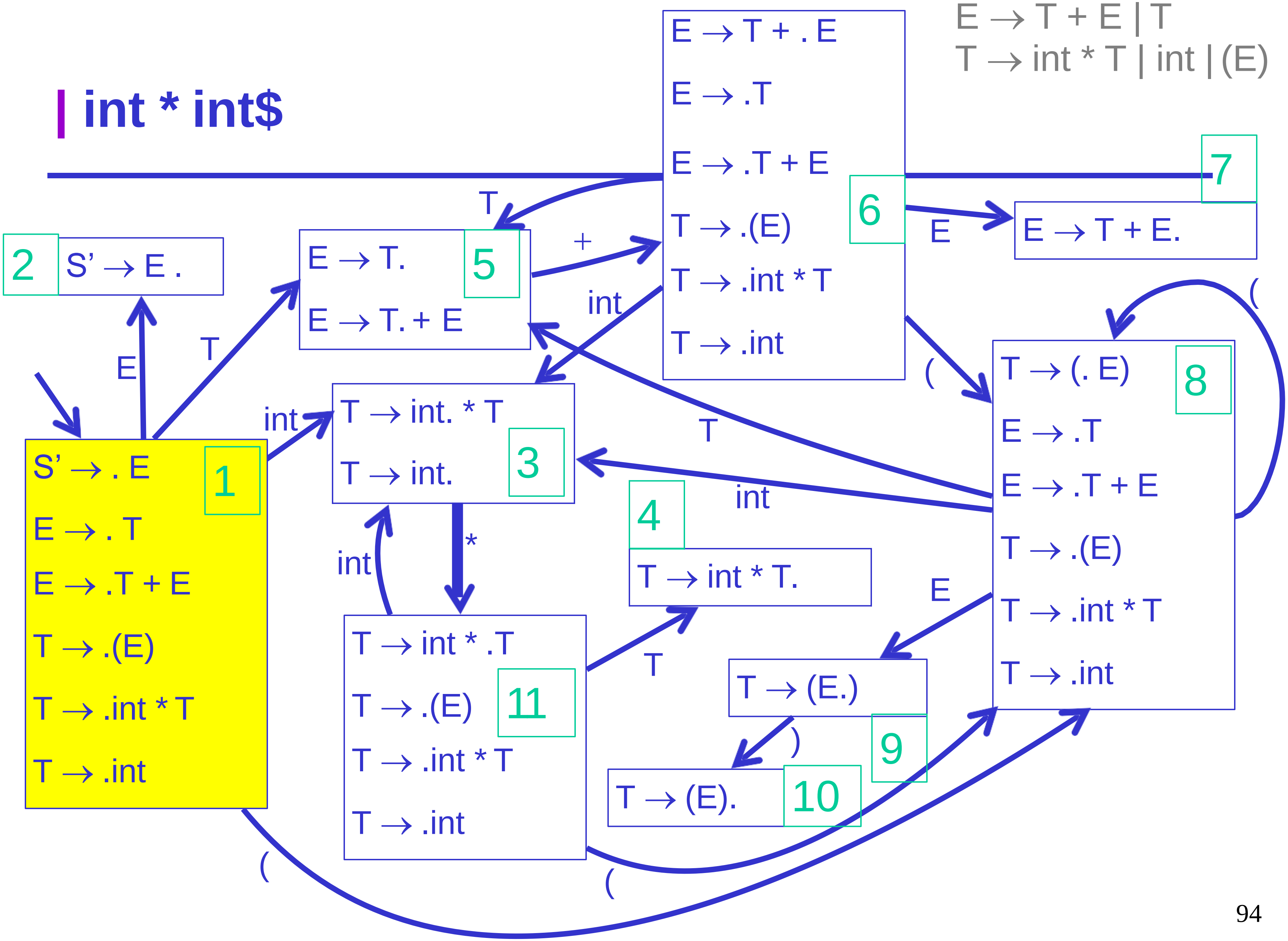
E → T + E | T
T → int * T | int | (E)

SLR Example

Configuration	DFA Halt State	Action
| int * int$	1	shift

E → T + E | T
T → int * T | int | (E)

| int * int$



1
S’ → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S’ → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

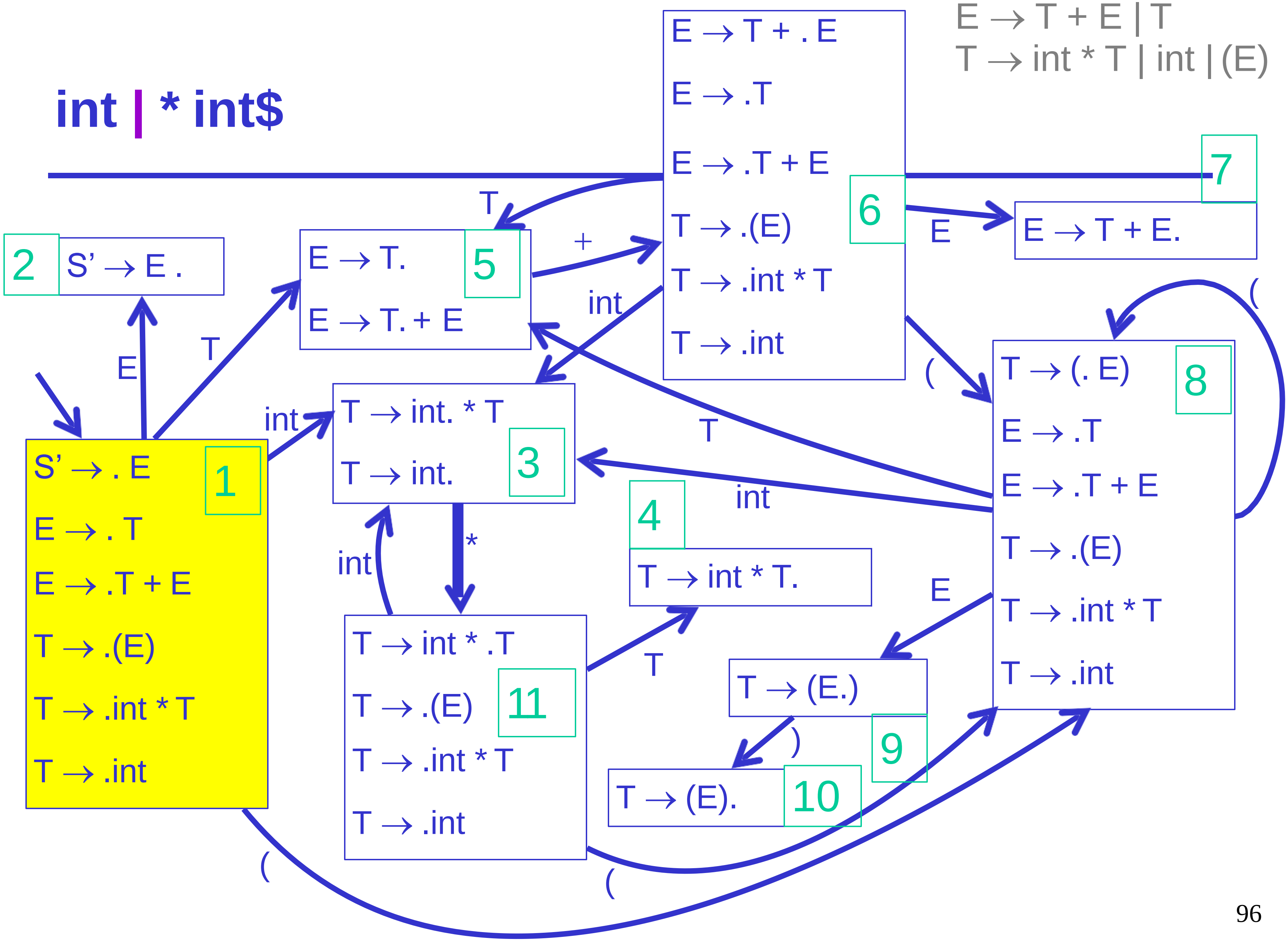
E → T + E | T
T → int * T | int | (E)

SLR Example

Configuration	DFA Halt State	Action
| int * int$	1	shift
int | * int$	3 * not in Follow(T)	shift

E → T + E | T
T → int * T | int | (E)

int | * int$



1
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S' → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

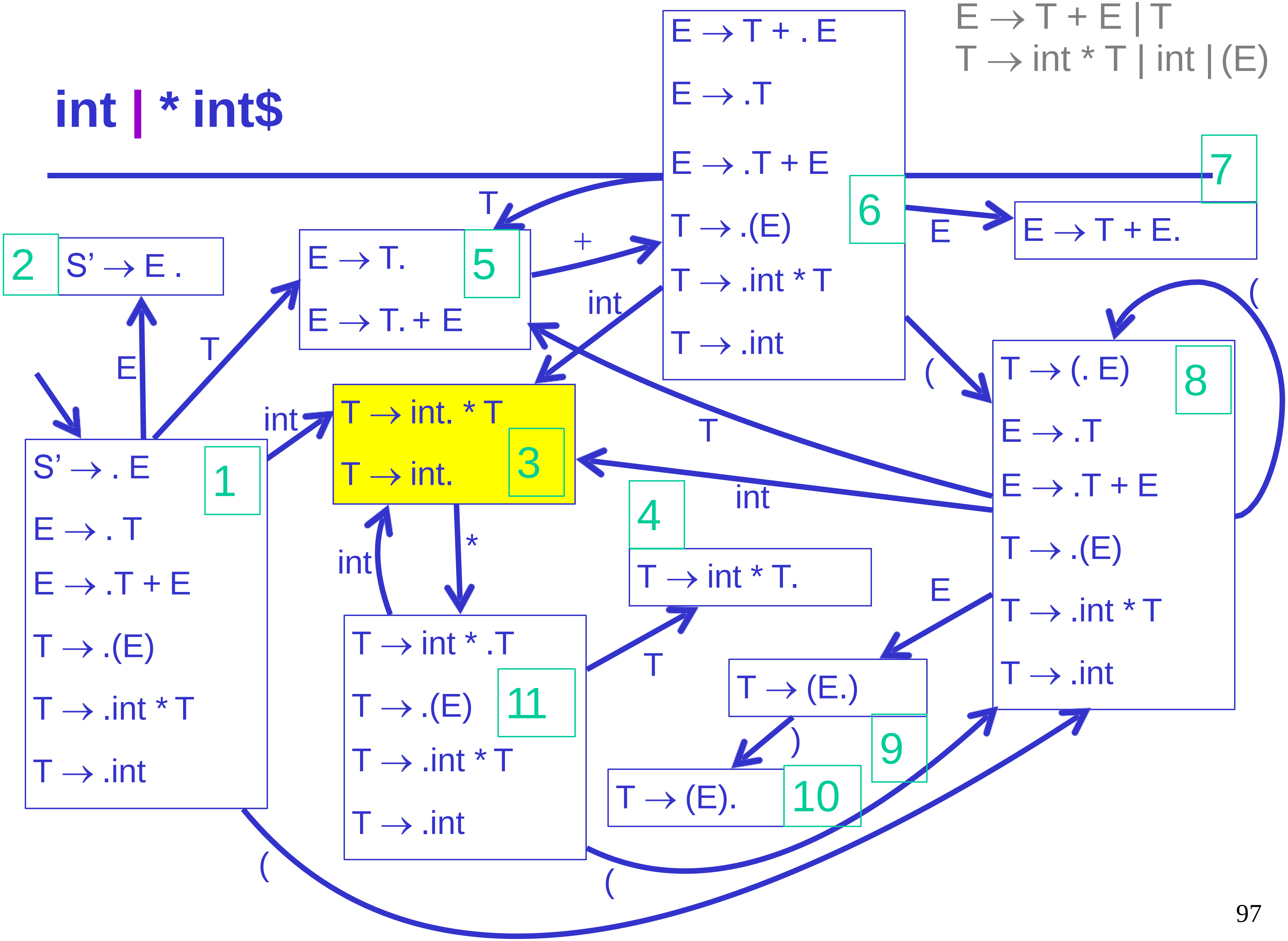
9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

E → T + E | T
T → int * T | int | (E)

int | * int$



1
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S' → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

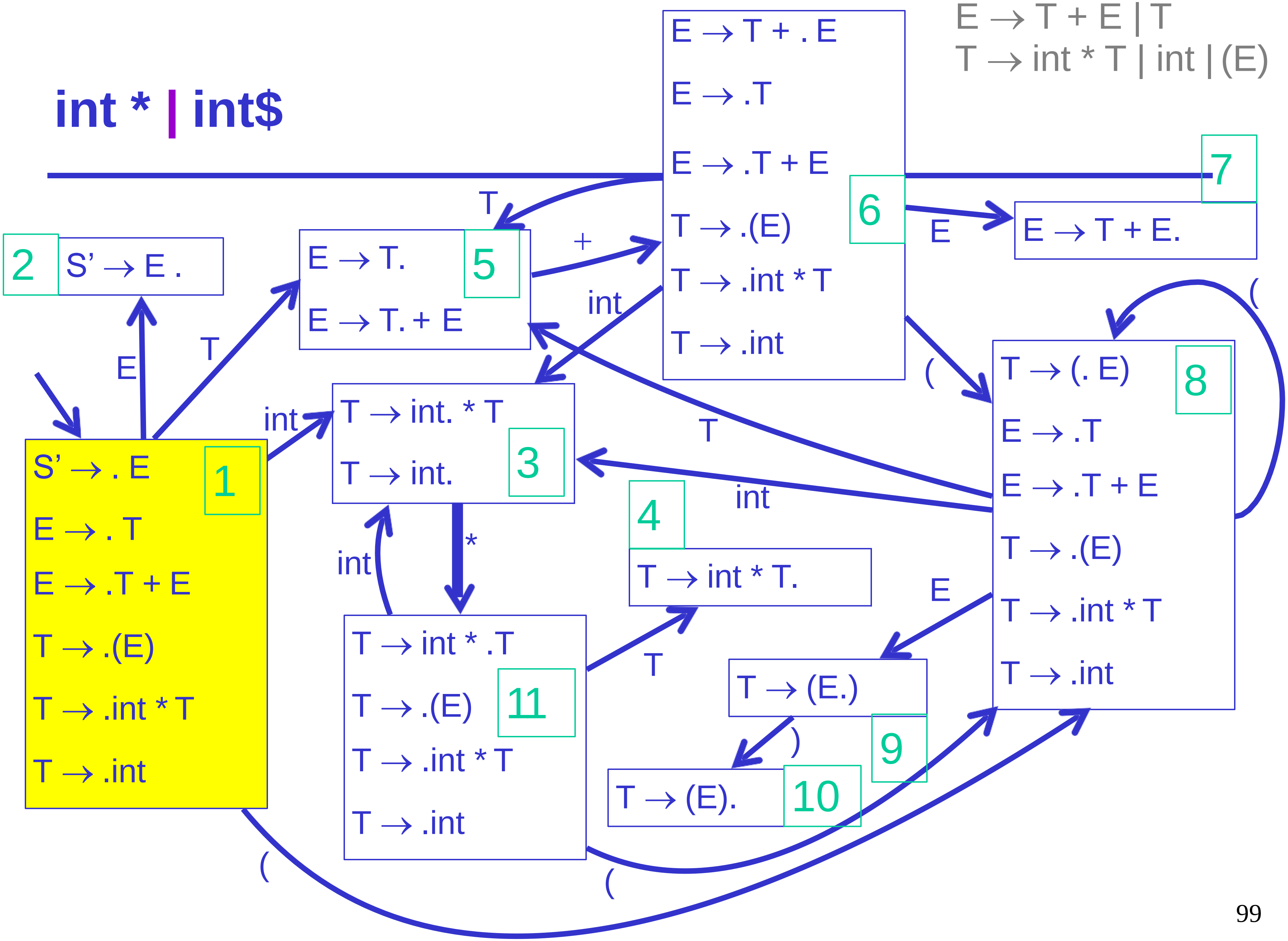
E → T + E | T
T → int * T | int | (E)

SLR Example

Configuration	DFA Halt State		Action
| int * int$	1		shift
int | * int$	3	* not in Follow(T)	shift
int * | int$	11		shift

E → T + E | T
T → int * T | int | (E)

int * | int$



1
S’ → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S’ → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

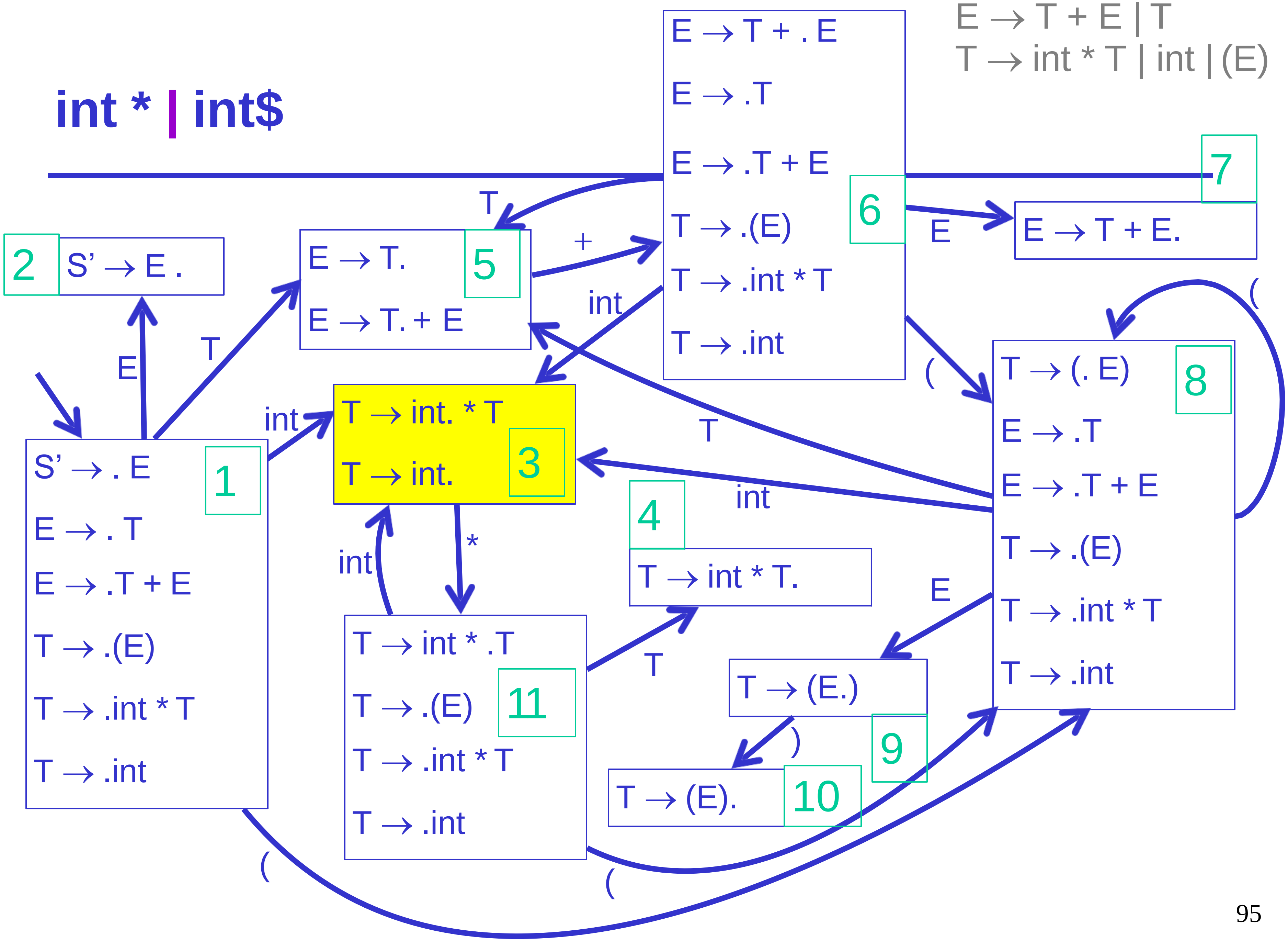
9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

E → T + E | T
T → int * T | int | (E)

int * | int$



1
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S' → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

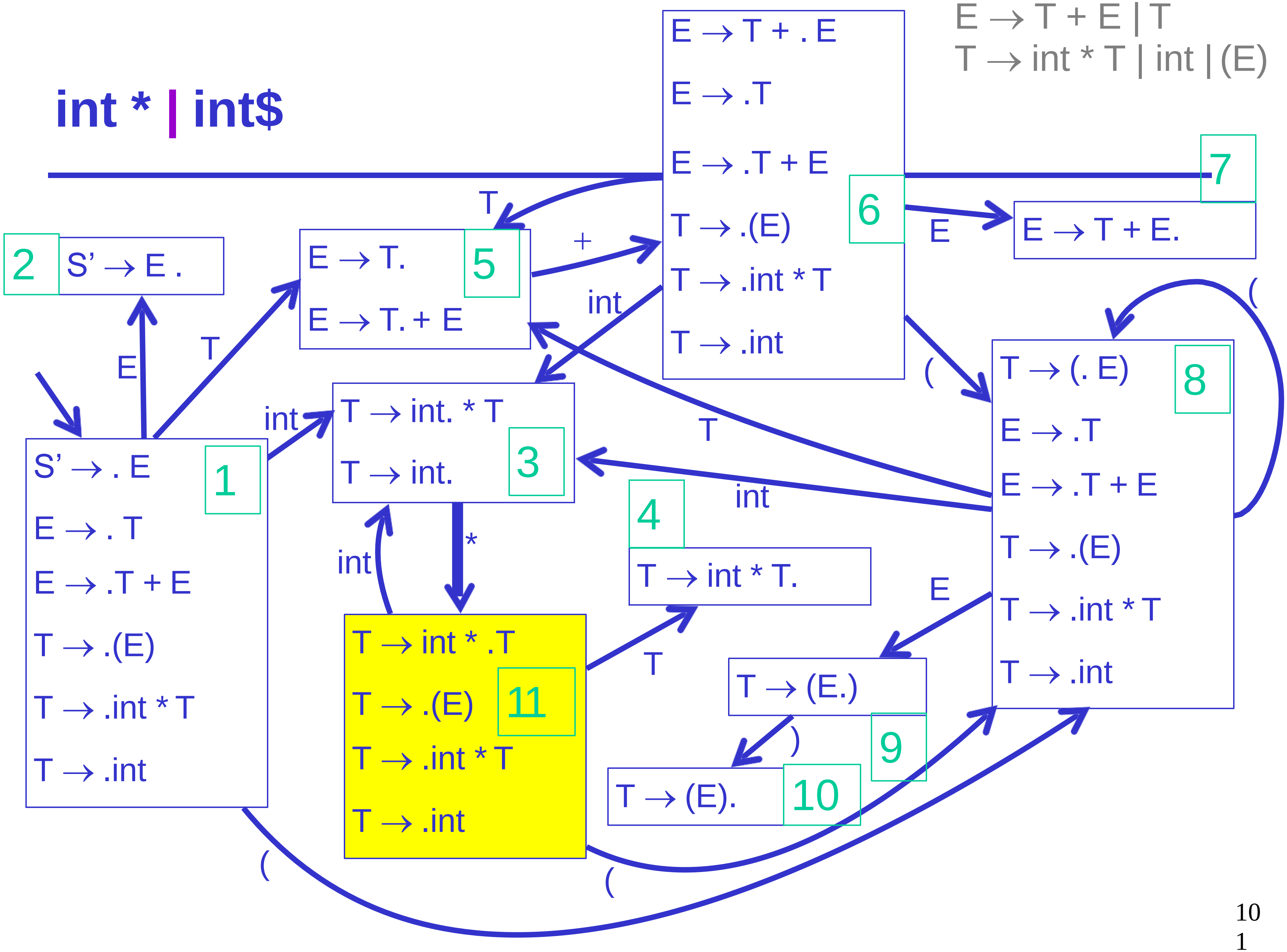
9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

E → T + E | T
T → int * T | int | (E)

int * | int$



1
S’ → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S’ → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

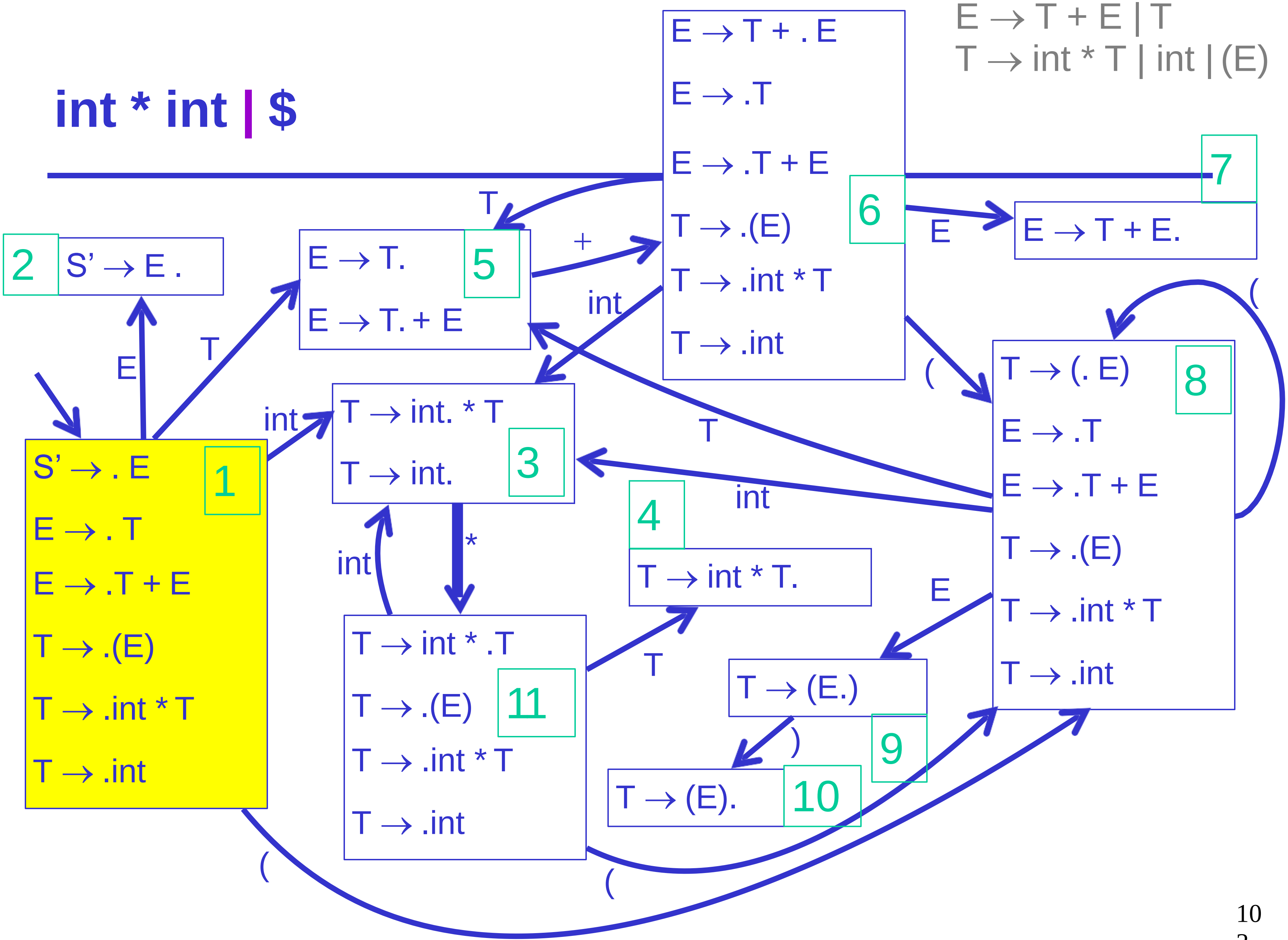
E → T + E | T
T → int * T | int | (E)

SLR Example

Configuration	DFA Halt State	Action
| int * int$	1	shift
int | * int$	3 * not in Follow(T)	shift
int * | int$	11	shift
int * int |$	3 $ ∈ Follow(T)	reduce T→int

E → T + E | T
T → int * T | int | (E)

int * int | $



1
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S' → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

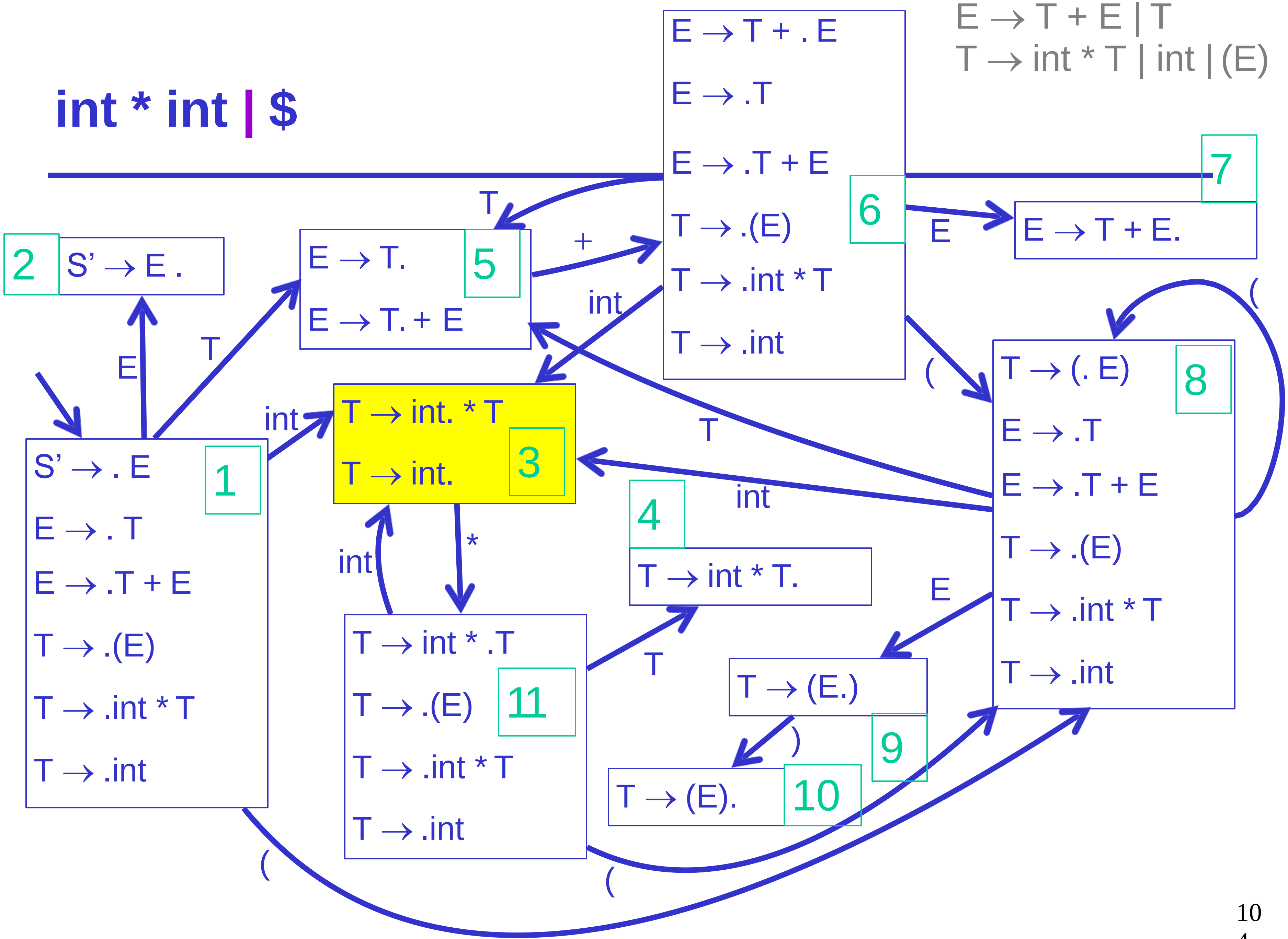
9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

E → T + E | T
T → int * T | int | (E)

int * int | $



1
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S' → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

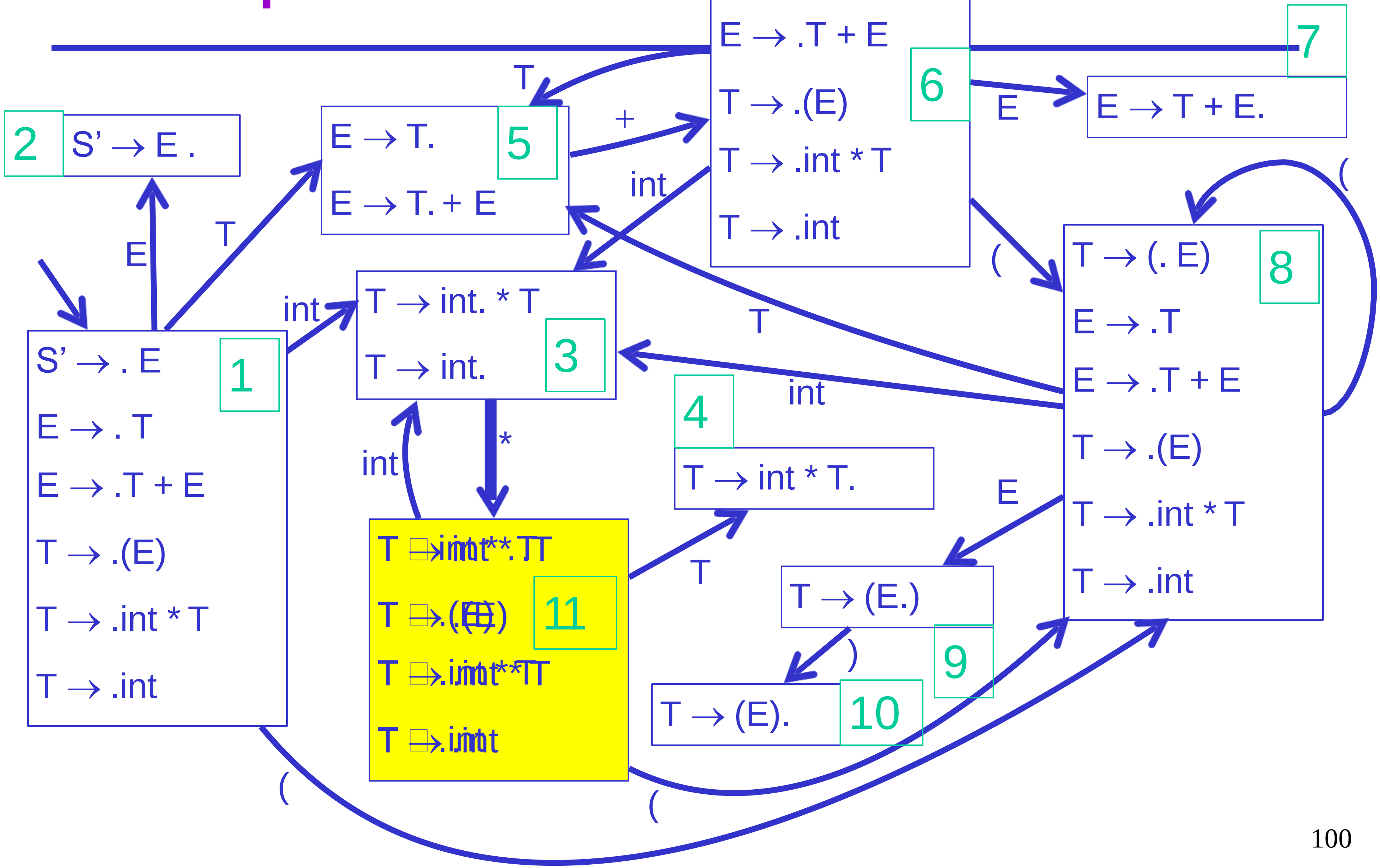
9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

int * int | $

E → T + E | T
T → int * T | int | (E)



1
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S' → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

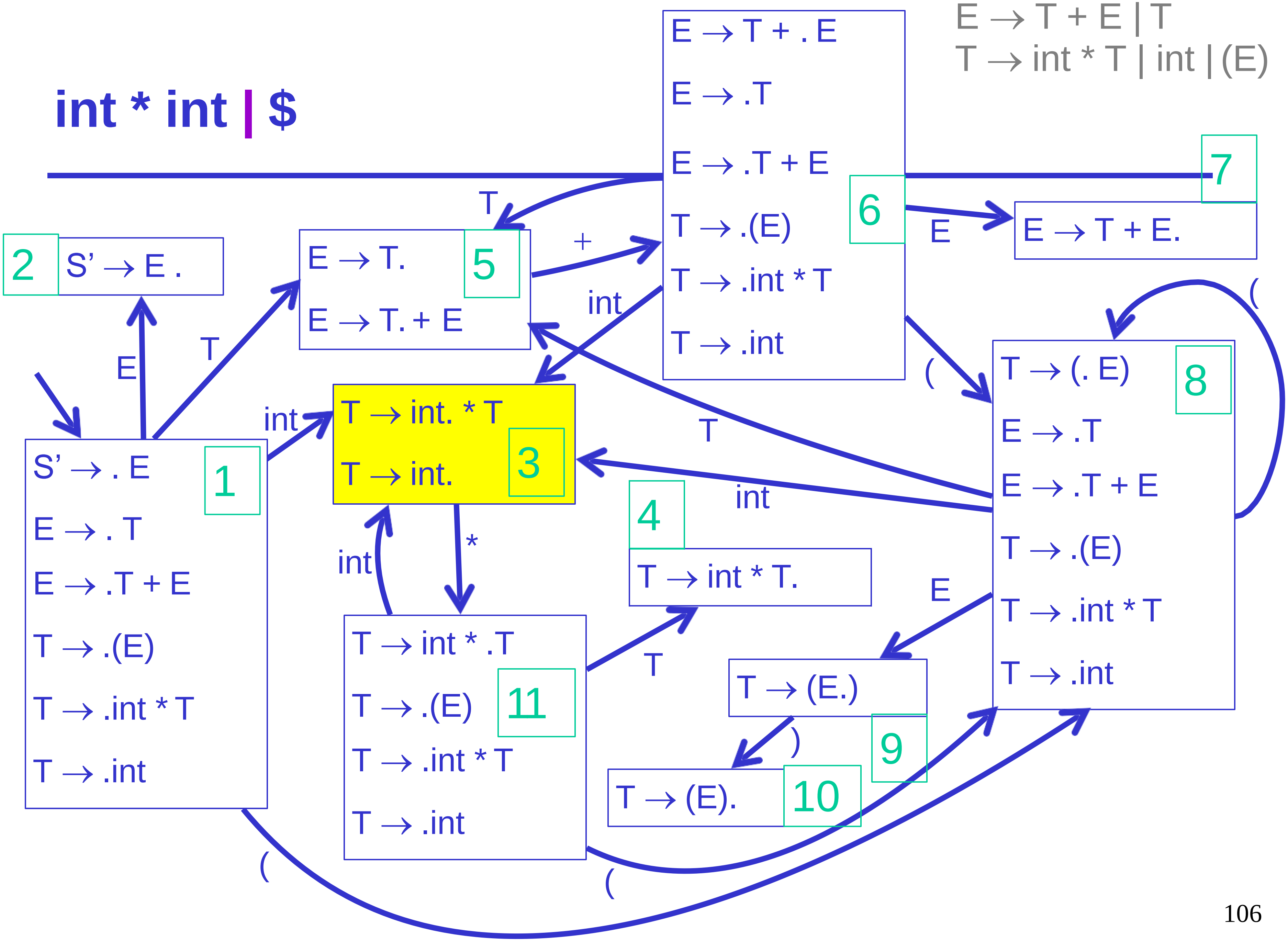
9
T → (E.)

10
T → (E).

11
T □→int**.TT
T □→.(E)
T □→.int**TT
T □→.int

E → T + E | T
T → int * T | int | (E)

int * int | $



1
S’ → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S’ → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

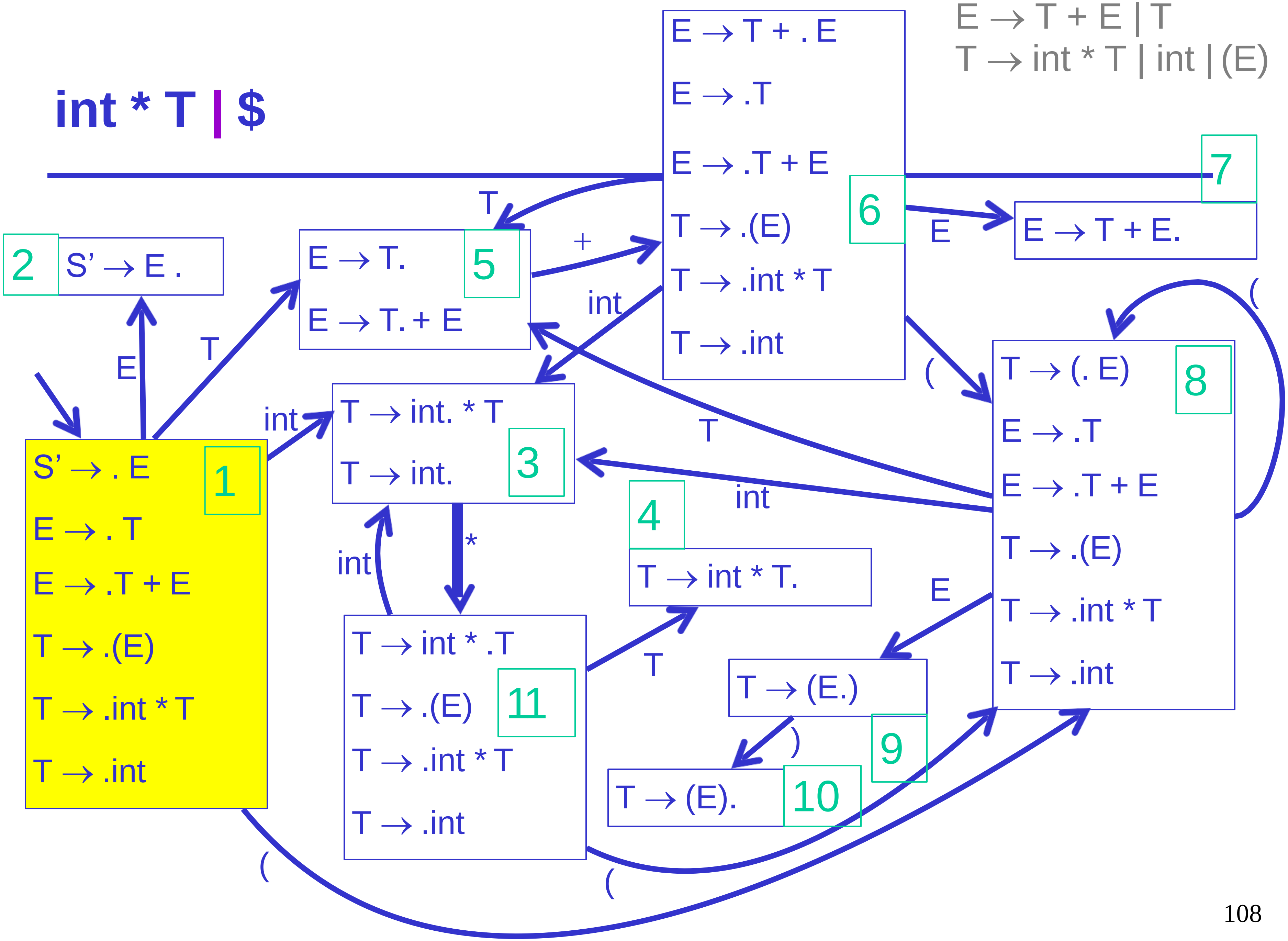
E → T + E | T
T → int * T | int | (E)

SLR Example

<table>
<tr><th>Configuration</th><th>DFA Halt State</th><th>Action</th></tr>
<tr><td>| int * int$</td><td>1</td><td>shift</td></tr>
<tr><td>int | * int$</td><td>3   * not in Follow(T)</td><td>shift</td></tr>
<tr><td>int * | int$</td><td>11</td><td>shift</td></tr>
<tr><td>int * int |$</td><td>3 $ ∈ Follow(T)</td><td>reduce T→int</td></tr>
<tr><td>int * T |$</td><td>4 $ ∈ Follow(T)</td><td>reduce T→int*T</td></tr>
</table>

E → T + E | T
T → int * T | int | (E)

int * T | $



1
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S' → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

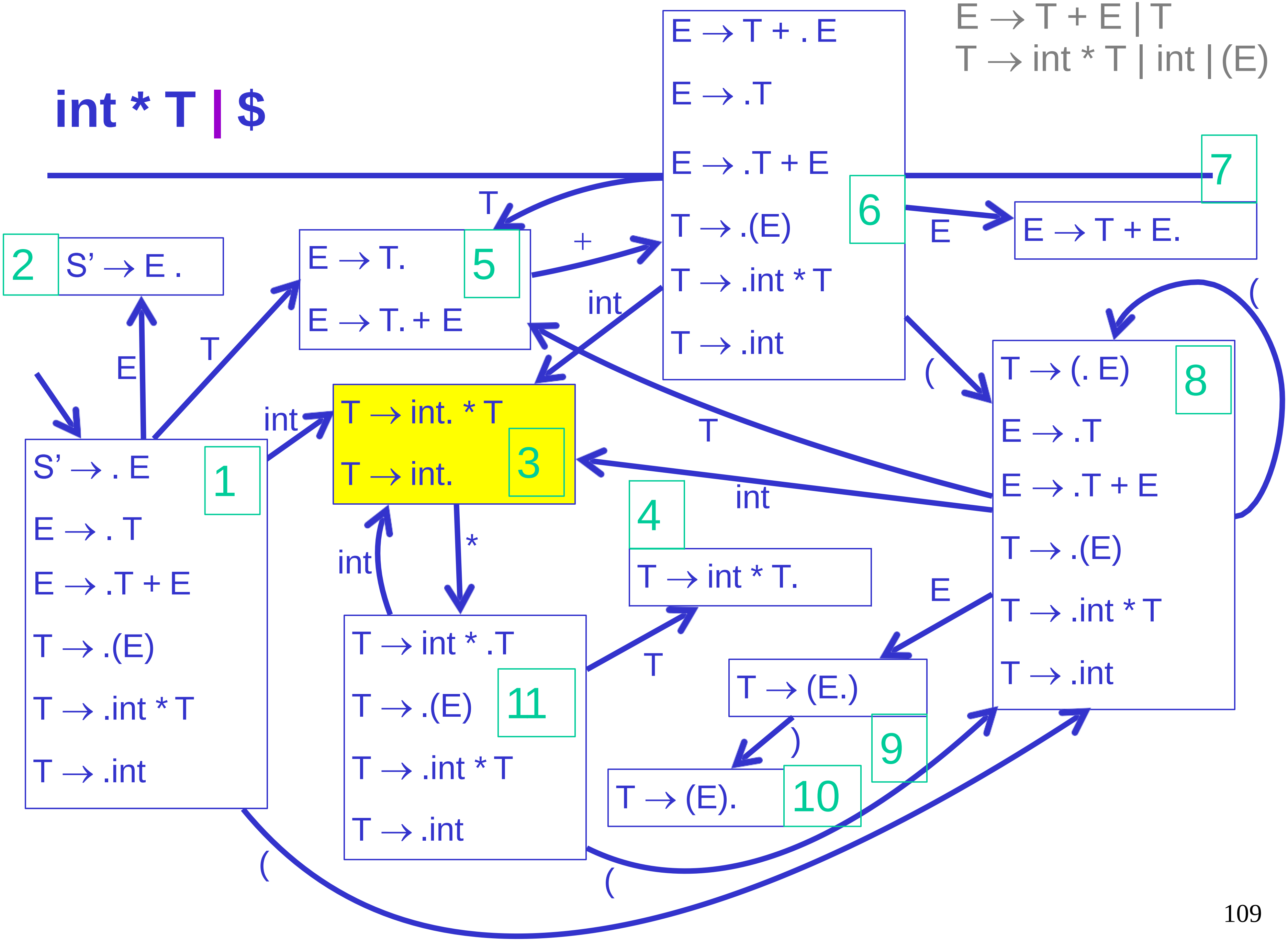
9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

E → T + E | T
T → int * T | int | (E)

int * T | $



1
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S' → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

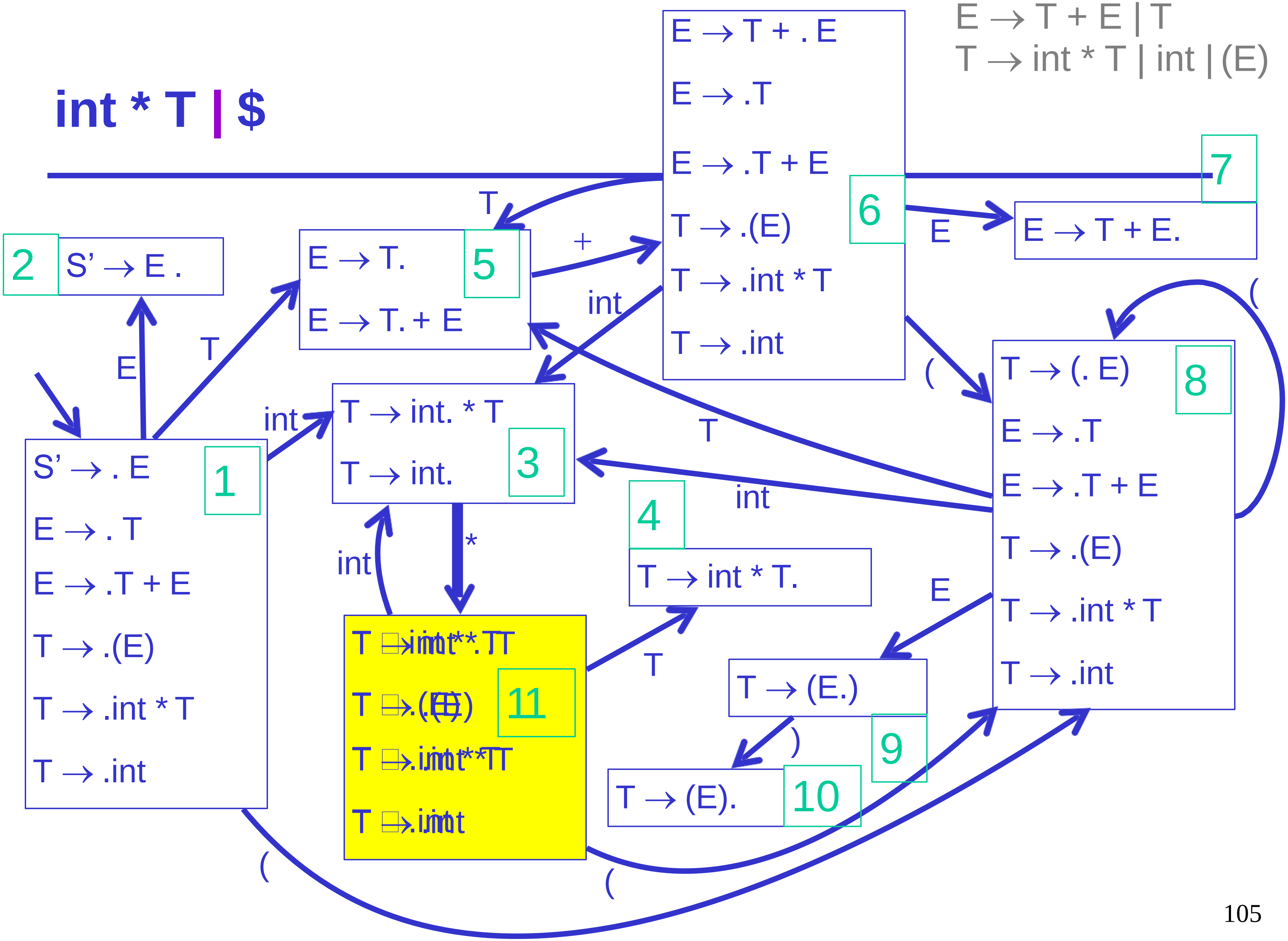
9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

E → T + E | T
T → int * T | int | (E)

int * T | $



1
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S' → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

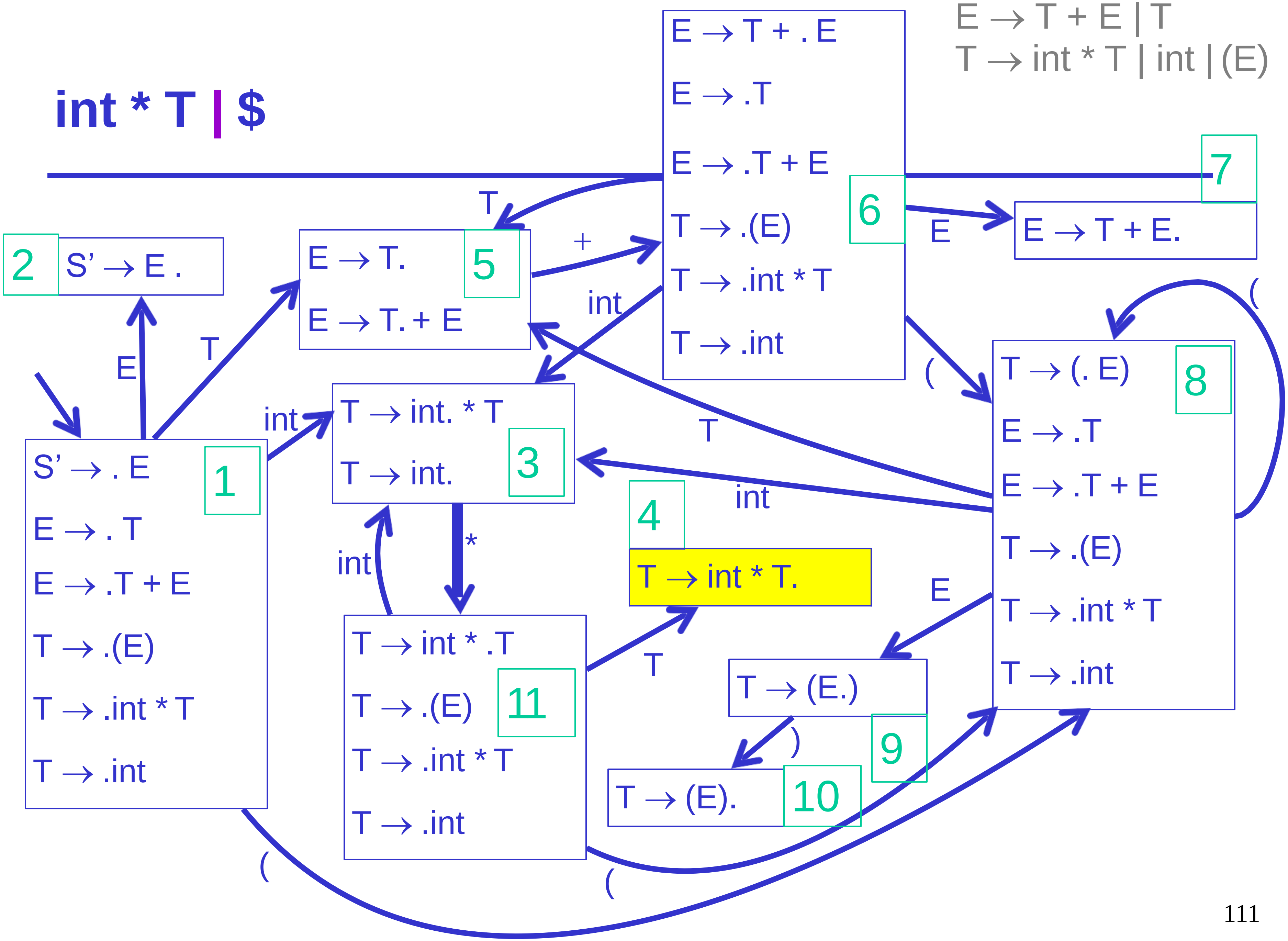
9
T → (E.)

10
T → (E).

11
T □→int * . T
T □→.(E)
T □→.int * T
T □→.int

E → T + E | T
T → int * T | int | (E)

int * T | $



1
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S' → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

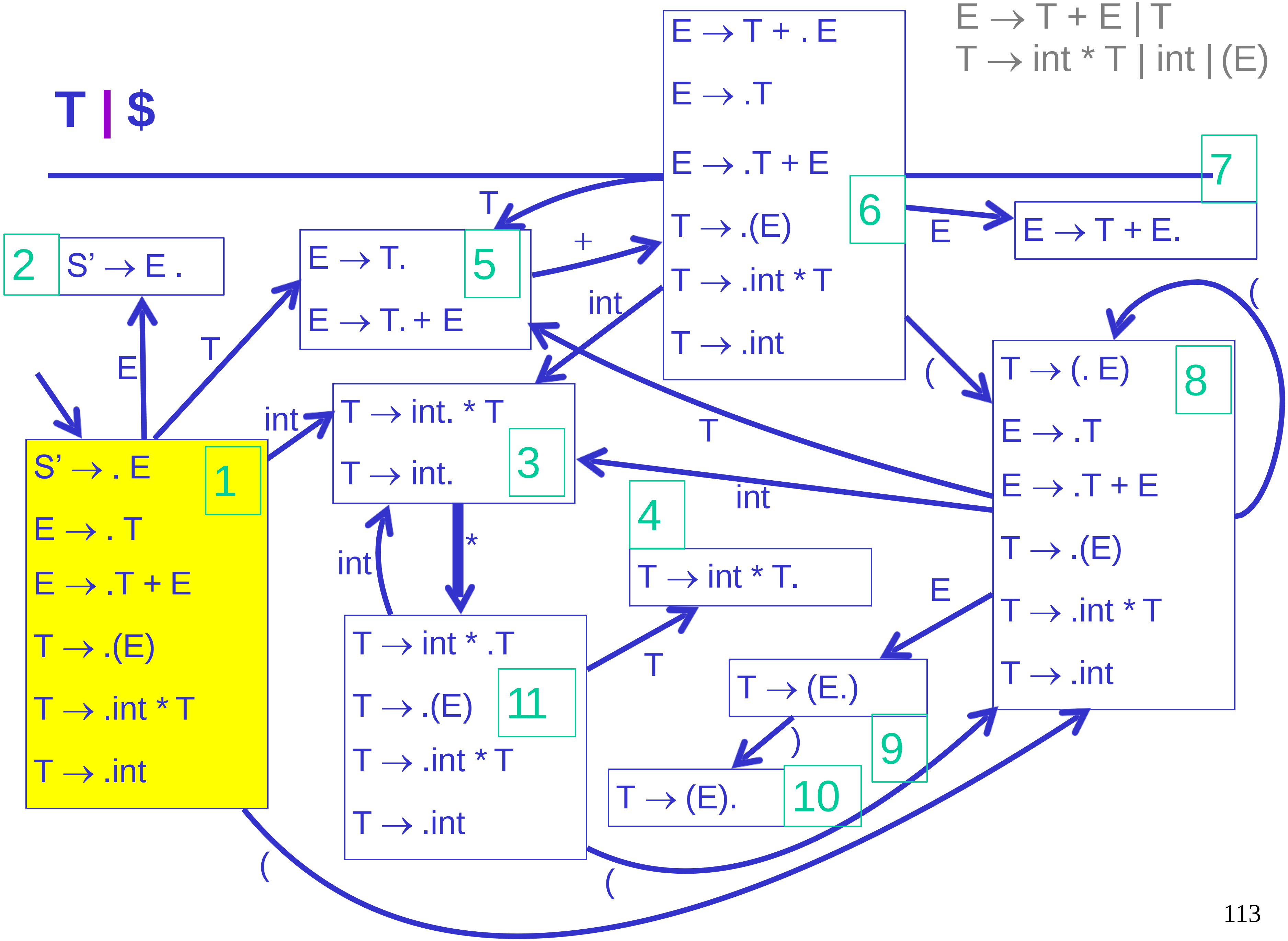
E → T + E | T
T → int * T | int | (E)

SLR Example

<table>
<tr><th>Configuration</th><th>DFA Halt State</th><th>Action</th></tr>
<tr><td>| int * int$</td><td>1</td><td>shift</td></tr>
<tr><td>int | * int$</td><td>3 * not in Follow(T)</td><td>shift</td></tr>
<tr><td>int * | int$</td><td>11</td><td>shift</td></tr>
<tr><td>int * int |$</td><td>3 $ ∈ Follow(T)</td><td>reduce T→int</td></tr>
<tr><td>int * T |$</td><td>4 $ ∈ Follow(T)</td><td>reduce T→int*T</td></tr>
<tr><td>T |$</td><td>5 $ ∈ Follow(T)</td><td>reduce E→T</td></tr>
</table>

E → T + E | T
T → int * T | int | (E)

T | $



1
S’ → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S’ → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

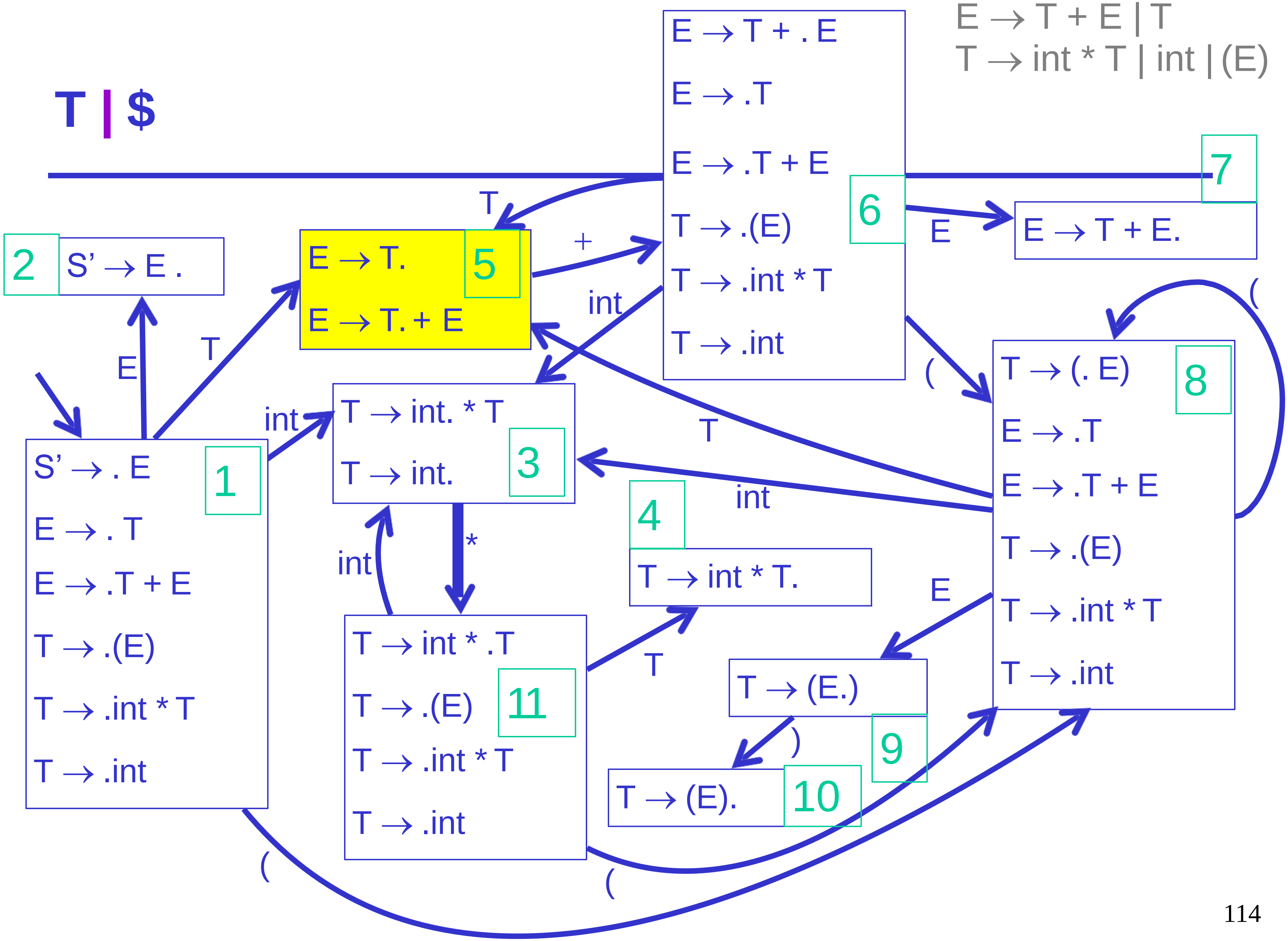
9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

E → T + E | T
T → int * T | int | (E)

T | $



1
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int

2
S' → E .

3
T → int. * T
T → int.

4
T → int * T.

5
E → T.
E → T. + E

6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

7
E → T + E.

8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int

9
T → (E.)

10
T → (E).

11
T → int * .T
T → .(E)
T → .int * T
T → .int

E → T + E | T
T → int * T | int | (E)

SLR Example

Configuration	DFA Halt State	Action
| int * int$	1	shift
int | * int$	3 * not in Follow(T)	shift
int * | int$	11	shift
int * int |$	3 $ $\in$ Follow(T)	reduce T→int
int * T |$	4 $ $\in$ Follow(T)	reduce T→int*T
T |$	5 $ $\in$ Follow(T)	reduce E→T
E |$		accept

An Improvement

- Rerunning the automaton at each step is wasteful
  - Most of the work is repeated

- Remember the state of the automaton on each prefix of the stack

- Change stack to contain pairs

symbol, DFA state □

An Improvement (Cont.)

- For a stack

  symbol$_1$, state$_1$ □. . . symbol$_n$, state$_n$ □

  state$_n$ is the final state of the DFA on symbol$_1$…symbol$_n$

- Detail: The bottom of the stack is dummy, start□ where
  - dummy is a dummy symbol
  - start is the start state of the DFA

Goto (DFA) Table

- Define goto[i,A] = j if $\text{state}_i \rightarrow^{A} \text{state}_j$

- goto is just the transition function of the DFA
  - One of two parsing tables

Refined Parser Moves

- Shift x
  - Push a, x□on the stack
  - a is current input
  - x is a DFA state
- Reduce X $\to \alpha$
  - As before
- Accept
- Error

Action Table

For each state s_i and terminal t

- If s_i has item $X \rightarrow \alpha.t\beta$ and $goto[i,t] = k$ then $action[i,t] = shift\ k$
- If s_i has item $X \rightarrow \alpha.$ and $t \in Follow(X)$ and $X \neq S'$ then $action[i,t] = reduce\ X \rightarrow \alpha$
- If s_i has item $S' \rightarrow S.$ then $action[i,\$] = accept$
- Otherwise, $action[i,t] = error$

SLR Parsing Algorithm

```
Let input = w$ be initial input
Let j = 0
Let DFA state 1 be the one with item $S' \to .S$
Let stack =  dummy, 1 □  //  symbol, state □
    repeat
        case action[top_state(stack), input[j]] of
            shift k:  push  input[j++], k
            □reduce $X \to \alpha$:
                pop $|\alpha|$ pairs,
                push  X, goto[top_state(stack),
            X]□accept: halt normally
            error: halt and report error
```

Notes on SLR Parsing Algorithm

- Note that the algorithm uses only the DFA states and the input
  - The stack symbols are never used!

- However, we still need the symbols for semantic actions

More Notes

- Some common constructs are not SLR(1)

- LR(1) is more powerful
  - Build lookahead into the items
  - An LR(1) item is a pair: (LR(0) item, x lookahead)
  - [T→ . int * T, $] means
    - After seeing T→ int * T reduce if lookahead is $
  - More accurate than just using follow sets
  - See Dragon Book