

Bottom-Up Parsing II

CS143

Lecture 8

Instructor: Fredrik Kjolstad

Slide design by Prof. Alex Aiken, with modifications

Review: Shift-Reduce Parsing

Bottom-up parsing uses two actions:

Shift

$ABC \mid xyz \Rightarrow ABCx \mid yz$

Reduce

$Cbxy \mid ijk \Rightarrow CbA \mid ijk$

Recall: 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
 - pushes a non-terminal on the stack
 - production lhs

Key Issue

- How do we decide when to shift or reduce?
- Example grammar:
$$E \rightarrow T + E \mid T$$
$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$
- Consider step $\text{int} \mid * \text{int} + \text{int}$
 - We could reduce by $T \rightarrow \text{int}$ giving $T \mid * \text{int} + \text{int}$
 - 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

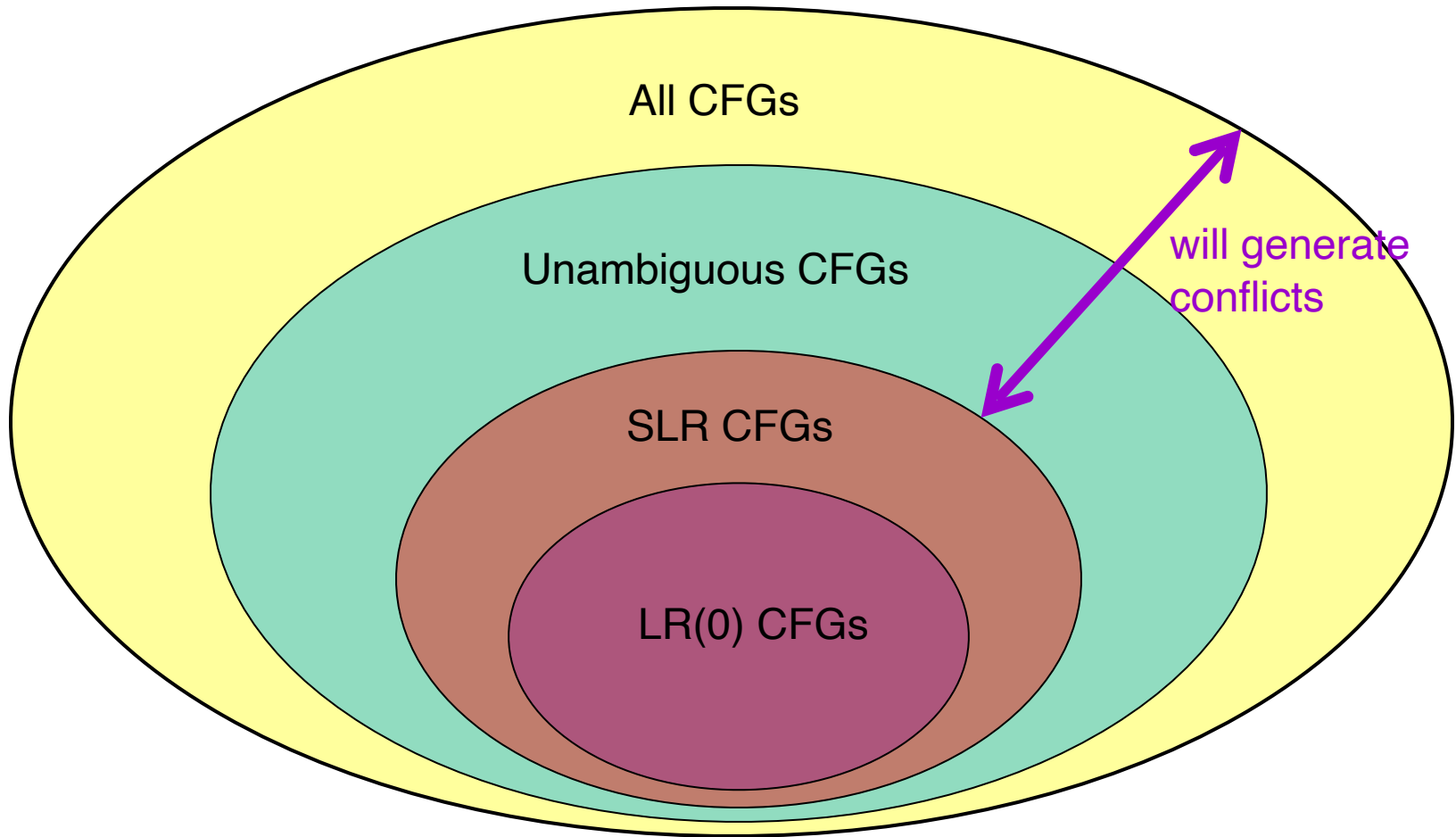
Summary of Handles

- In shift-reduce parsing, handles always appear at the top of the stack
- Handles are never to the left of the rightmost non-terminal
 - Therefore, shift-reduce moves are sufficient; the **|** need never move left
- Bottom-up parsing algorithms are based on recognizing handles

Recognizing Handles

- There are no known efficient algorithms to recognize handles
- Solution: use heuristics to guess which stacks are handles
- On some CFGs, the heuristics always guess correctly
 - For the heuristics we use here, these are the SLR grammars
 - Other heuristics work for other grammars

Grammars



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:

For any grammar, 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 $T \rightarrow (E)$ are
 - $T \rightarrow \cdot(E)$
 - $T \rightarrow (\cdot E)$
 - $T \rightarrow (E \cdot)$
 - $T \rightarrow (E) \cdot$

Items (Cont.)

- The only item for $X \rightarrow \varepsilon$ is $X \rightarrow \cdot$.
- 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

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$

Example

Consider the input (int)

- Then (E |) is a state of a shift-reduce parse
- (E is a prefix of the rhs of $T \rightarrow (E)$
 - Will be reduced after the next shift
- Item $T \rightarrow (E.)$ 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
 $\text{Prefix}_1 \text{Prefix}_2 \dots \text{Prefix}_{n-1} \text{Prefix}_n$
- 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, $\text{Prefix}_{k+1} \dots \text{Prefix}_n$ eventually reduces to the missing part of α_k

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

An Example

Consider the string $(\text{int} * \text{int})$:

$(\text{int} * \mid \text{int})$ is a state of a shift-reduce parse

From top of the stack:

“ $\text{int} *$ ” is a prefix of the rhs of $T \rightarrow \text{int} * T$

“ ϵ ” is a prefix of the rhs of $E \rightarrow T$

“(” is a prefix of the rhs of $T \rightarrow (E)$

$$E \rightarrow T + E \mid T$$

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

An Example (Cont.)

The stack of items

$$T \rightarrow \text{int} * .T$$

$$E \rightarrow .T$$

$$T \rightarrow (.E)$$

Says

We've seen $\text{int} *$ of $T \rightarrow \text{int} * T$

We've seen ϵ of $E \rightarrow T$

We've seen $($ of $T \rightarrow (E)$

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

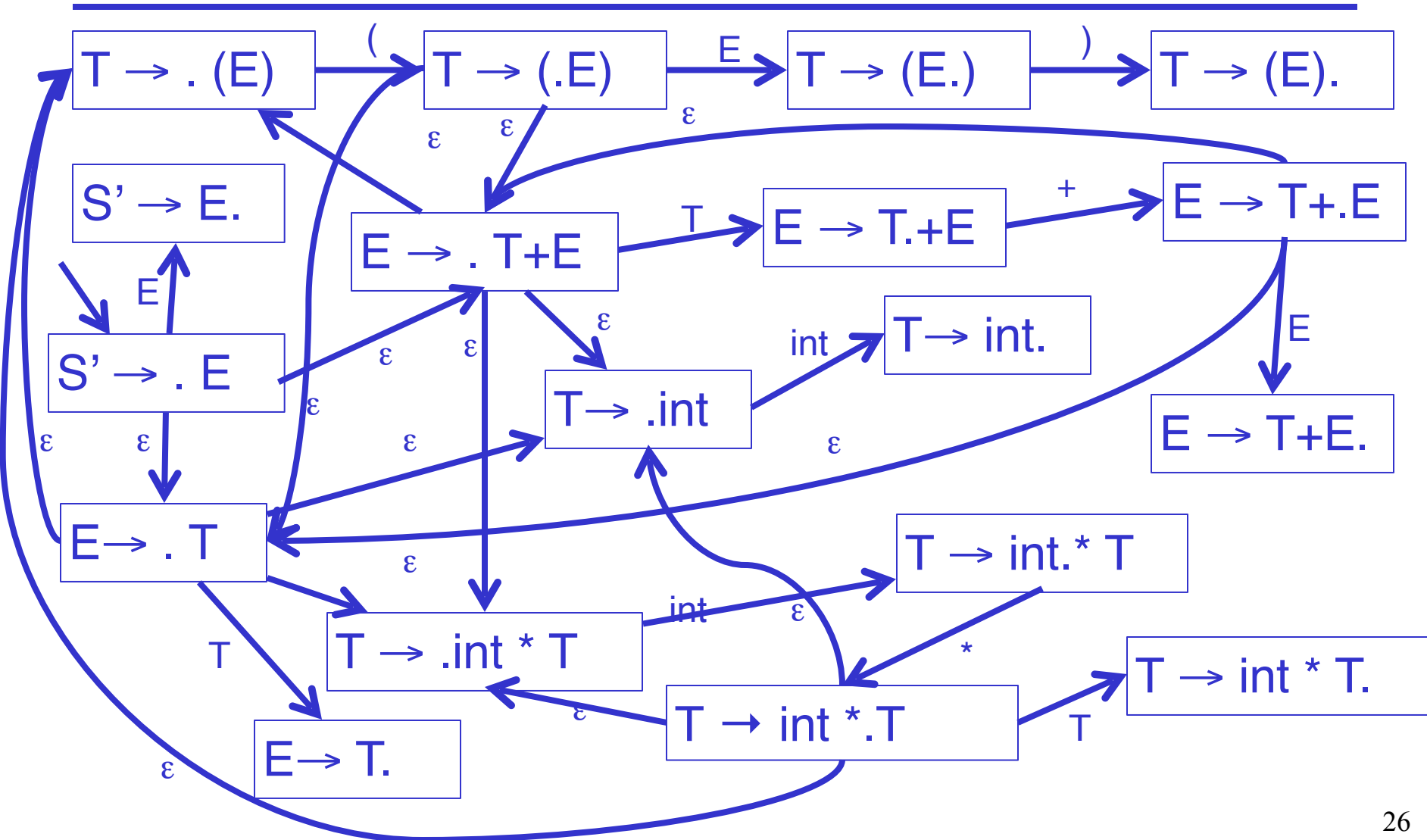
1. Add a new start production $S' \rightarrow S$ to G
2. The NFA states are the items of G
 - (Including the new start production)
3. For item $E \rightarrow \alpha.X\beta$ add transition
$$E \rightarrow \alpha.X\beta \xrightarrow{X} E \rightarrow \alpha X.\beta$$
4. For item $E \rightarrow \alpha.X\beta$ and production $X \rightarrow \gamma$ add
$$E \rightarrow \alpha.X\beta \xrightarrow{\epsilon} X \rightarrow .\gamma$$

An NFA Recognizing Viable Prefixes (Cont.)

- 5. Every state is an accepting state
- 6. Start state is $S' \rightarrow .S$

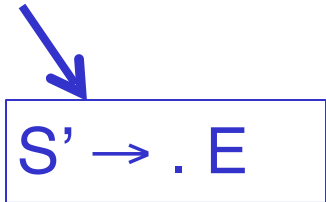
$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$

NFA for Viable Prefixes



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

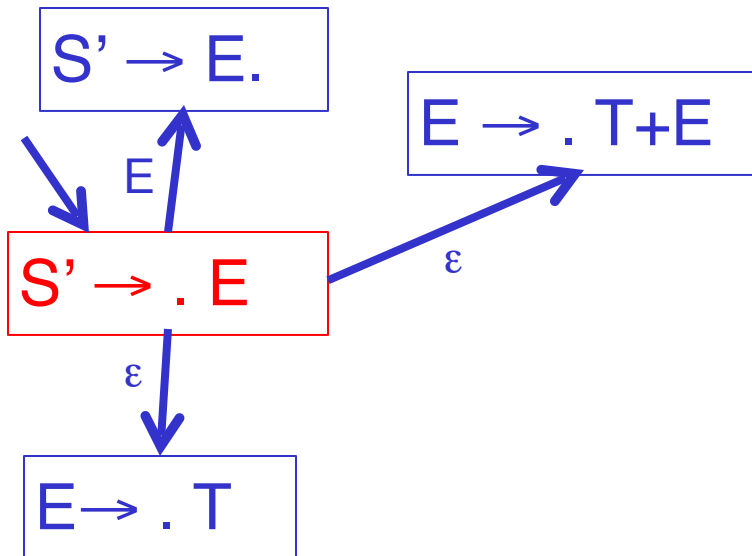
NFA for Viable Prefixes



$S' \rightarrow . E$

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

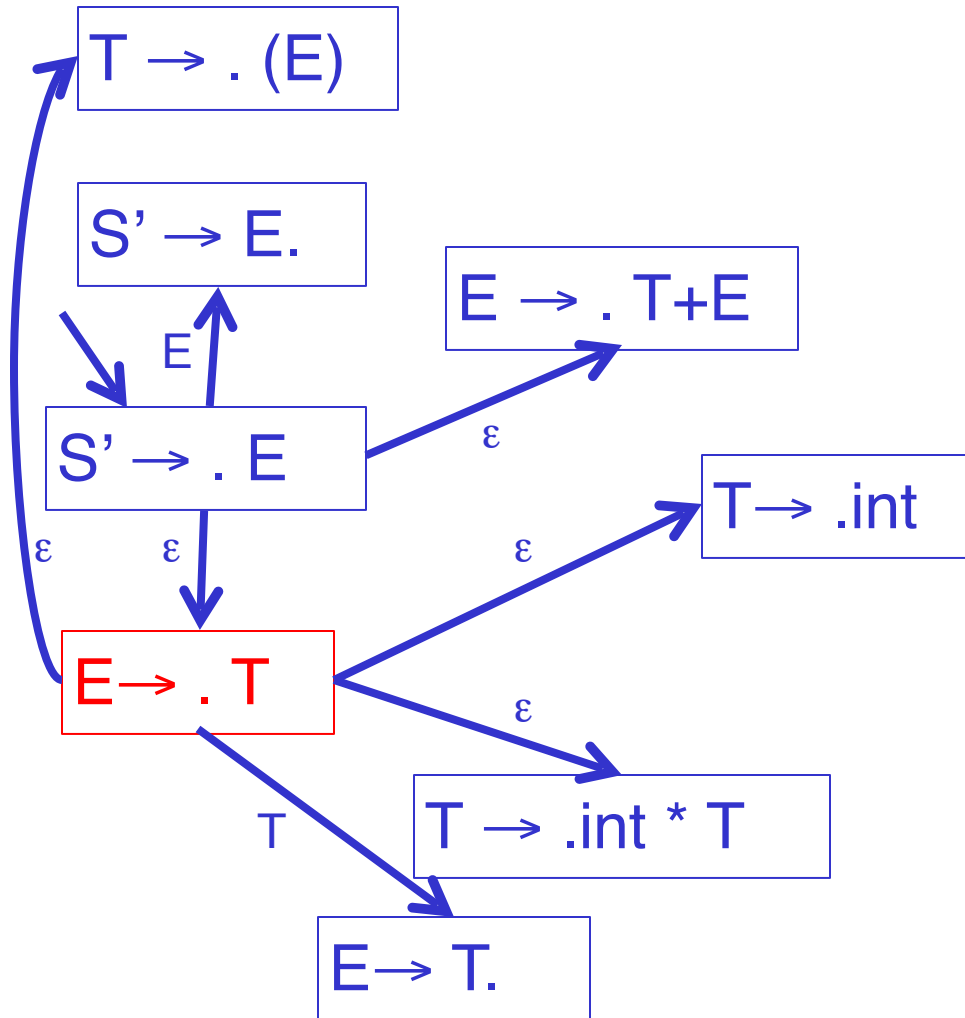
NFA for Viable Prefixes



$$E \rightarrow T + E \mid T$$

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

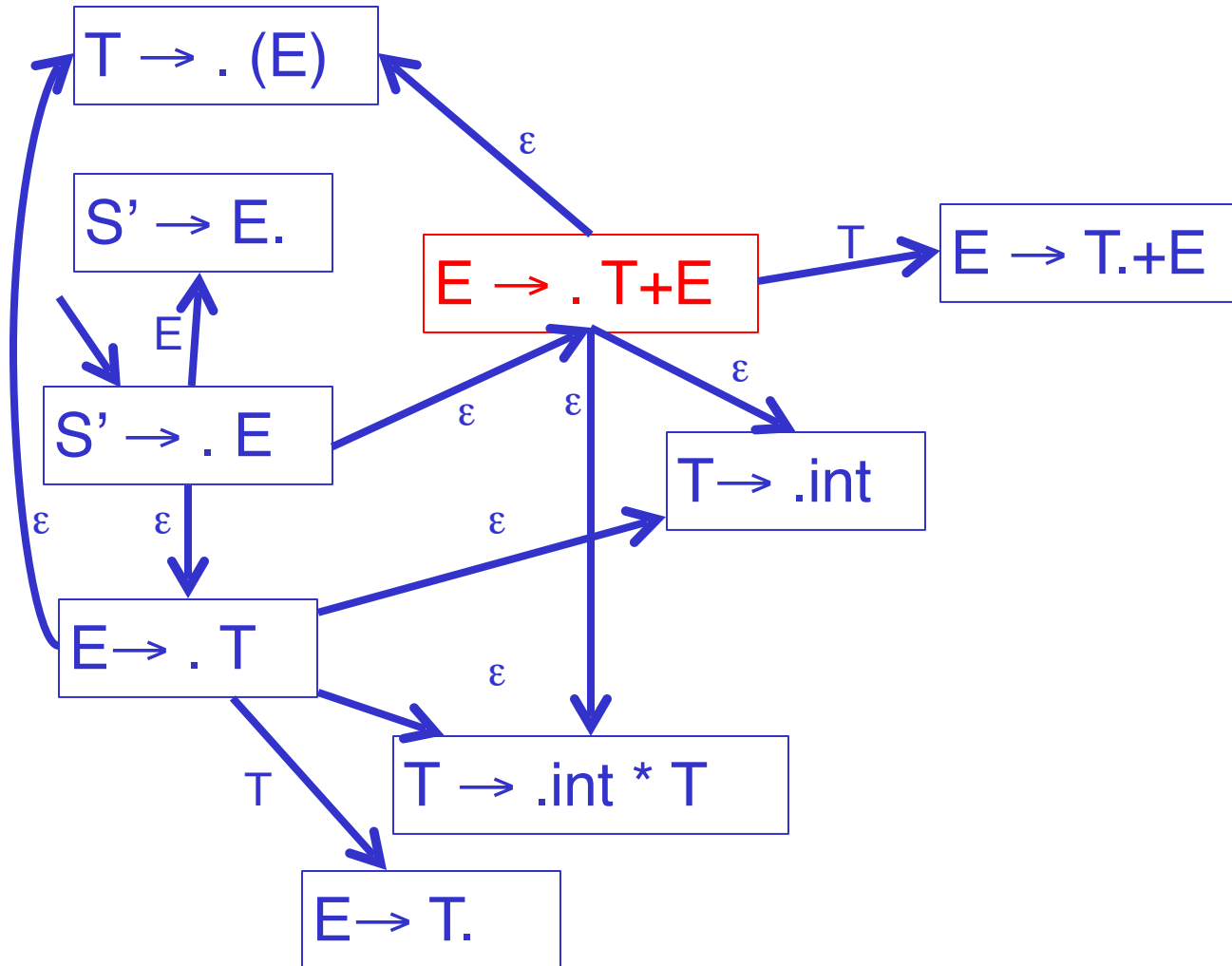
NFA for Viable Prefixes



$$E \rightarrow T + E \mid T$$

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

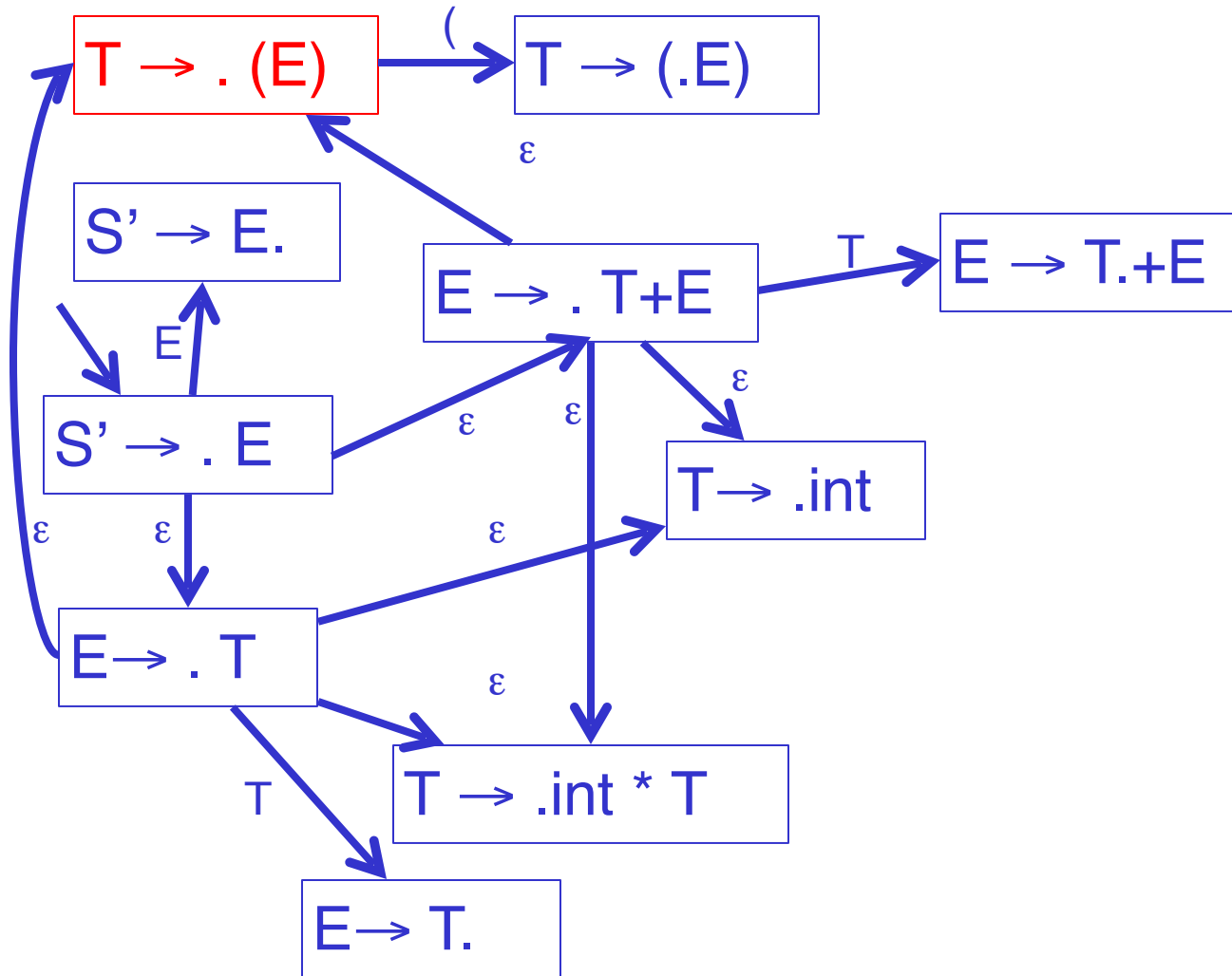
NFA for Viable Prefixes



$$E \rightarrow T + E \mid T$$

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

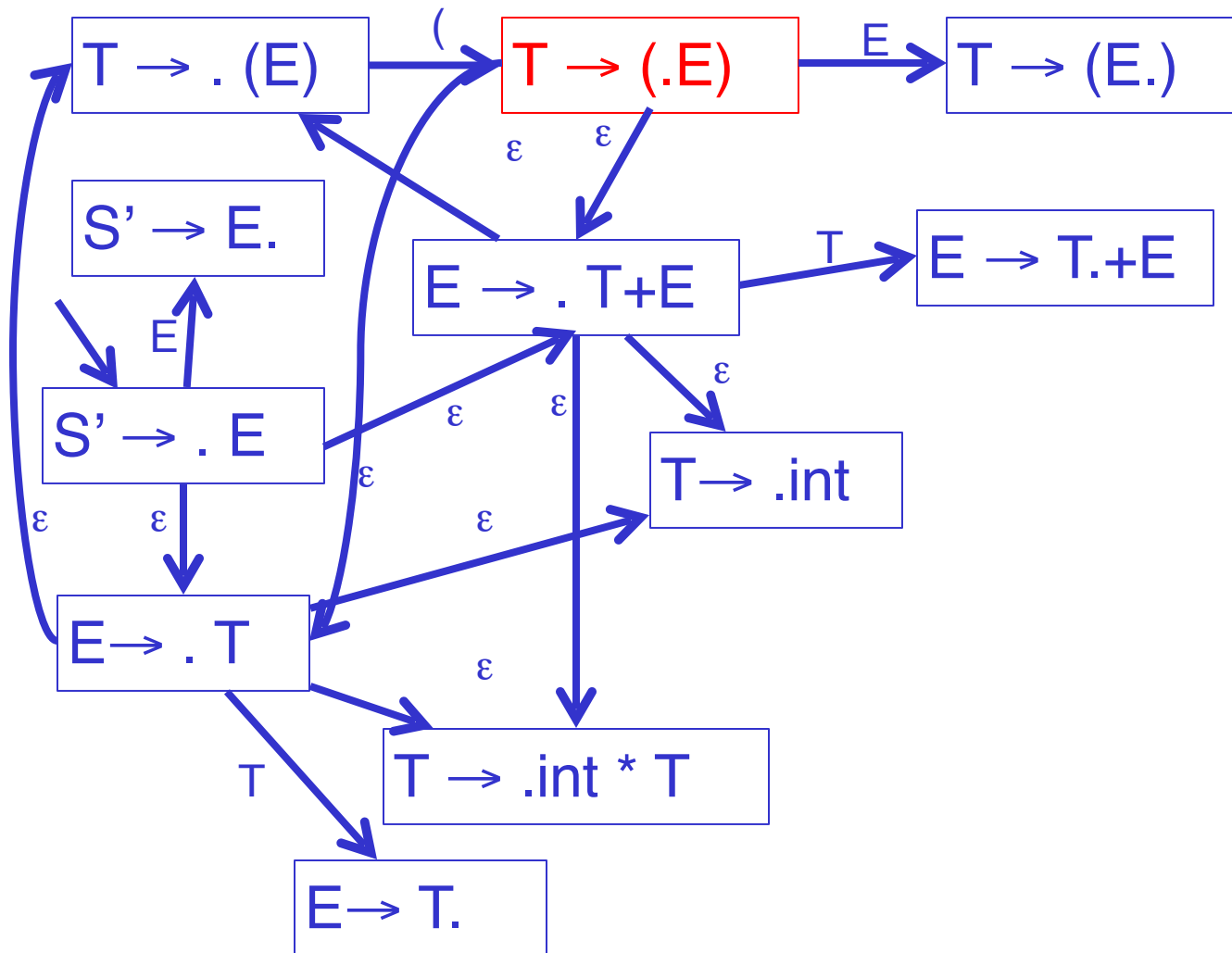
NFA for Viable Prefixes



$$E \rightarrow T + E \mid T$$

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

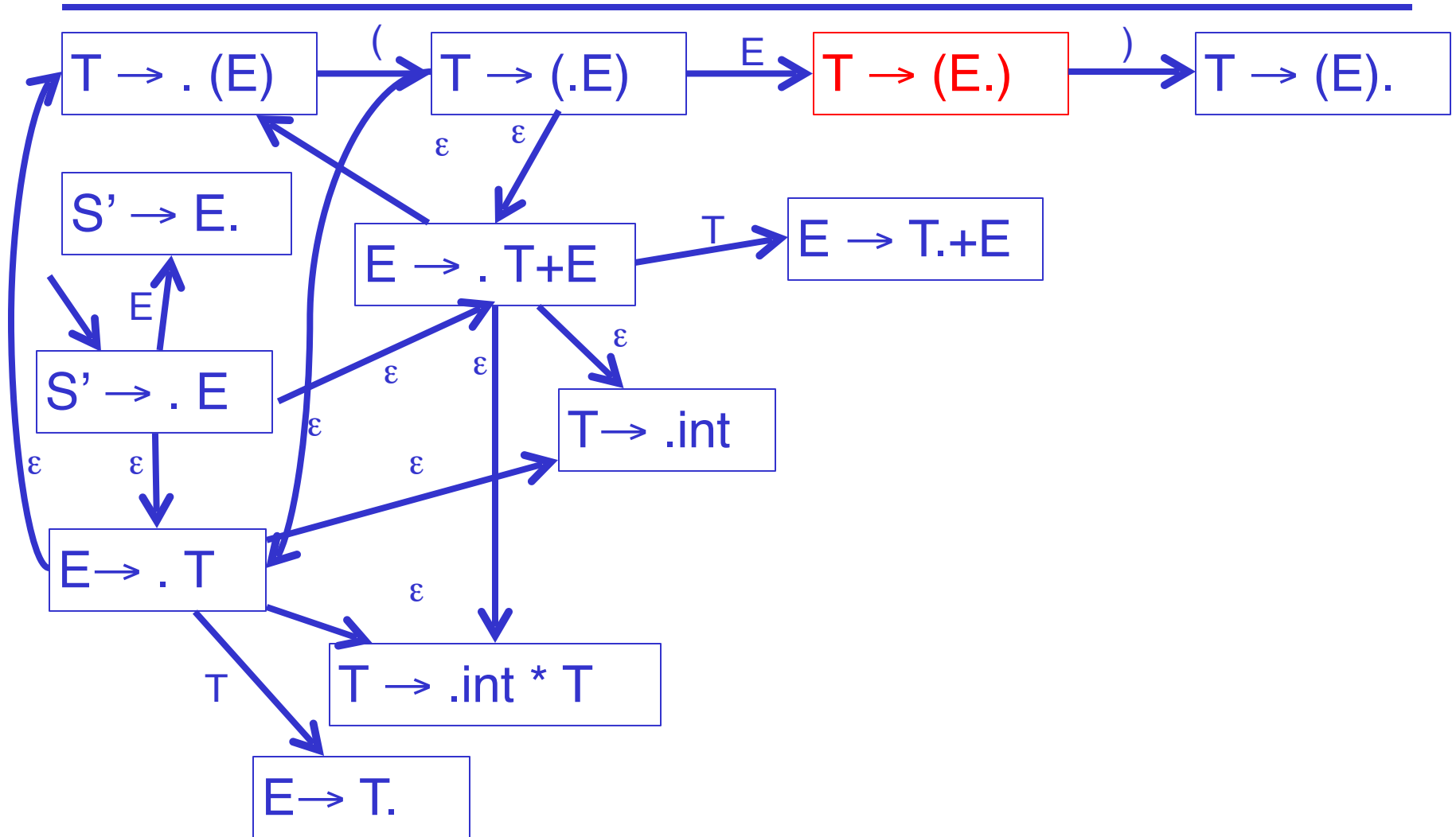
NFA for Viable Prefixes



$$E \rightarrow T + E \mid T$$

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

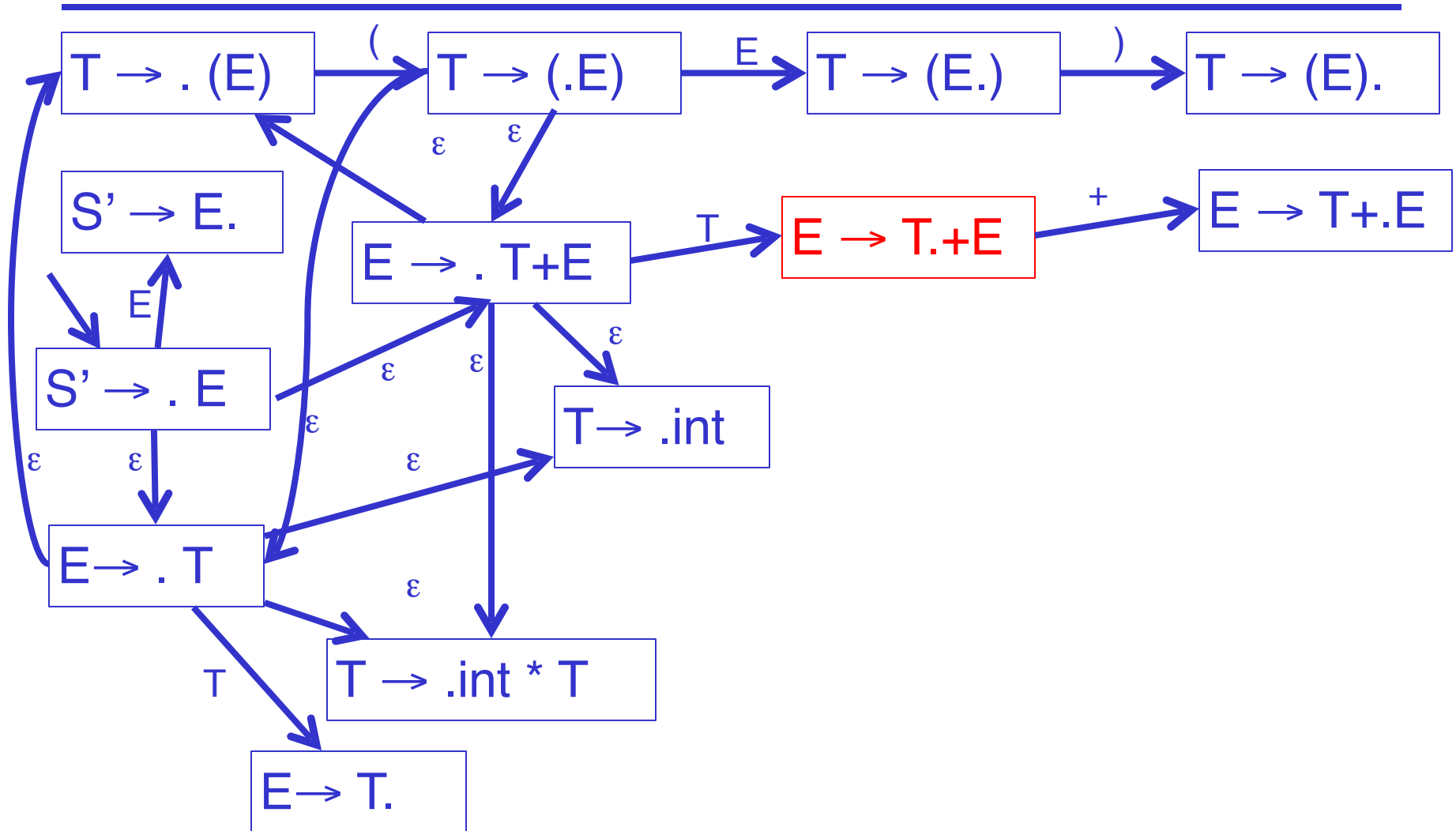
NFA for Viable Prefixes



$$E \rightarrow T + E \mid T$$

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

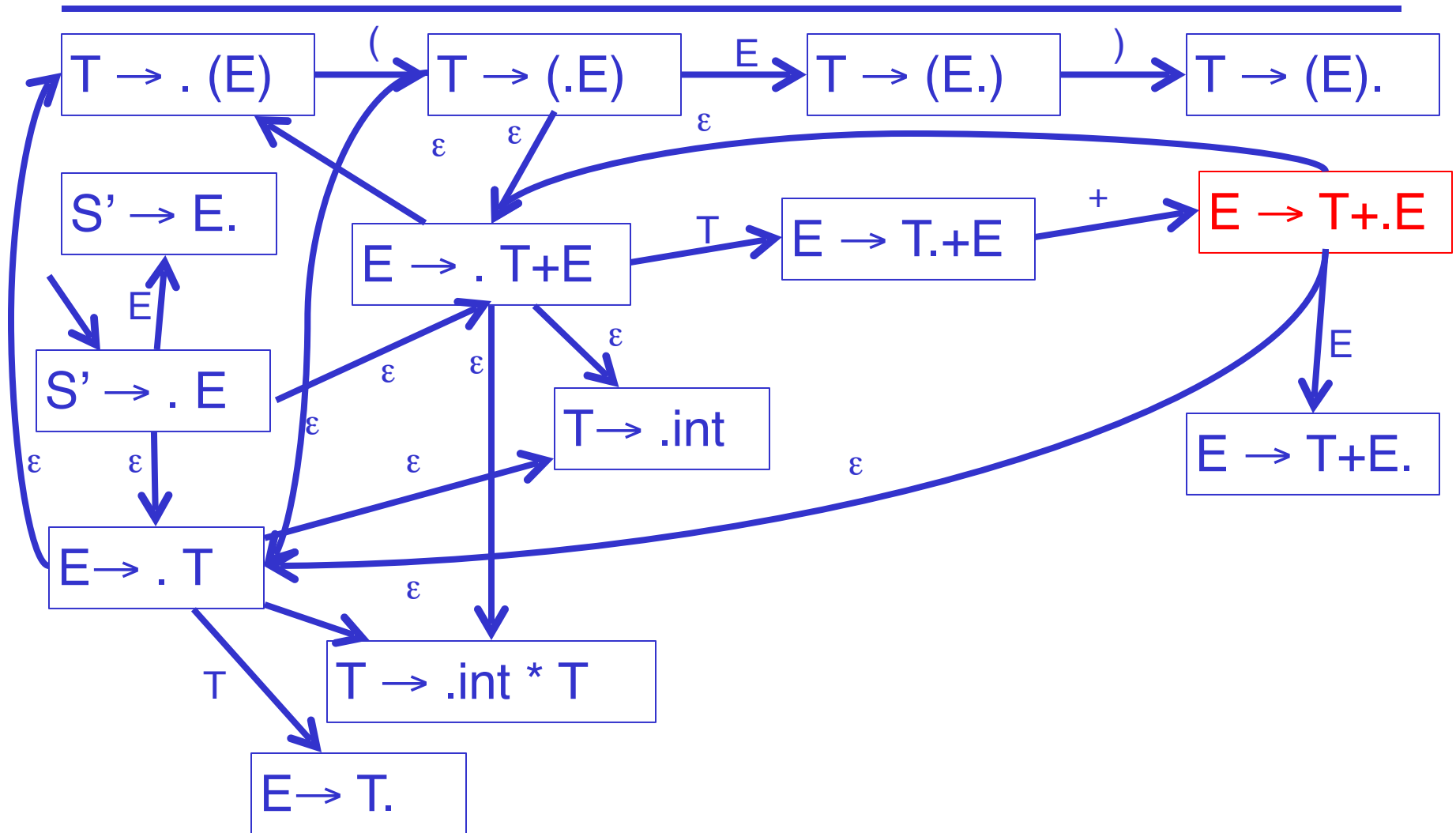
NFA for Viable Prefixes



$$E \rightarrow T + E \mid T$$

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

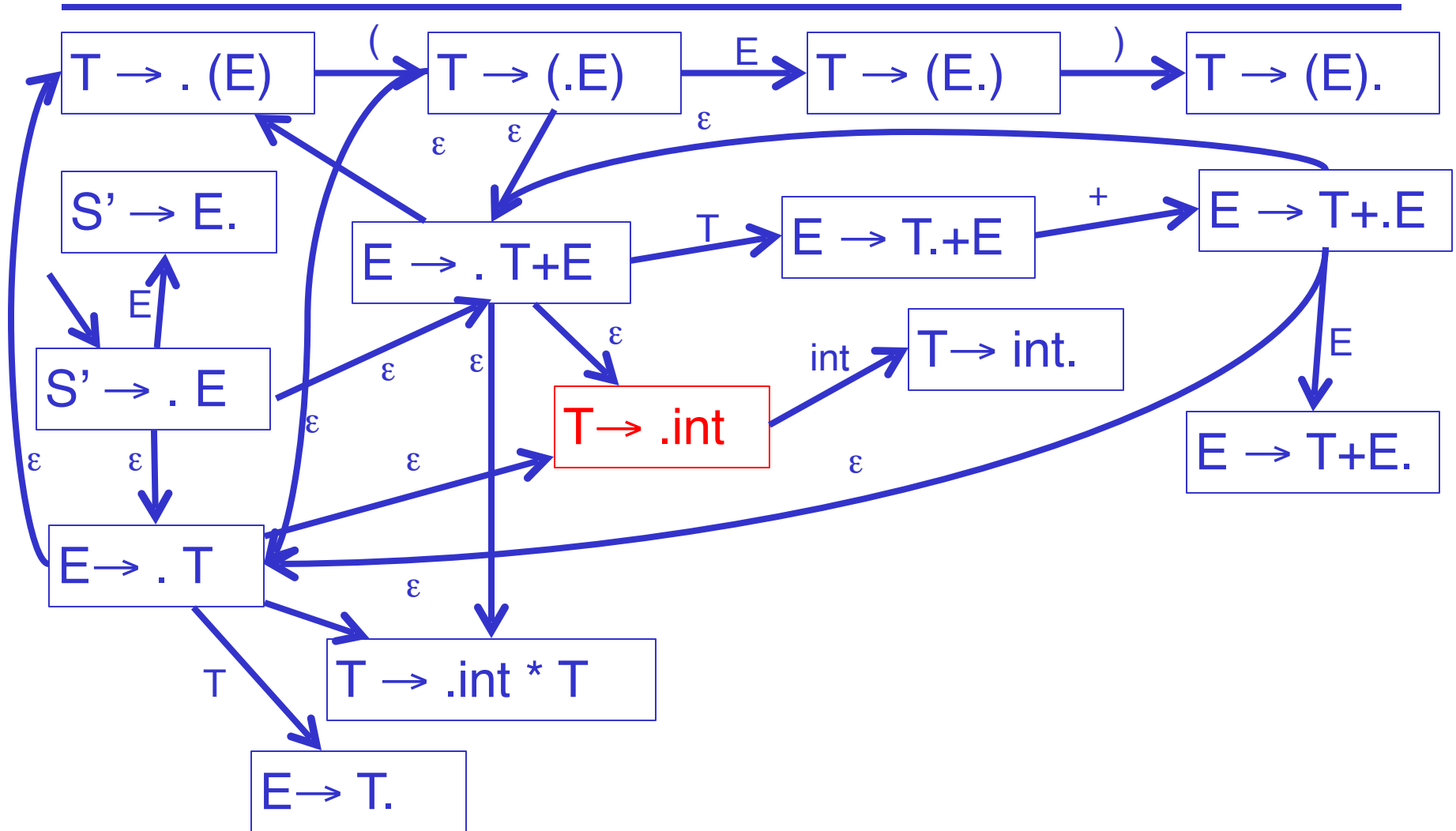
NFA for Viable Prefixes



$$E \rightarrow T + E \mid T$$

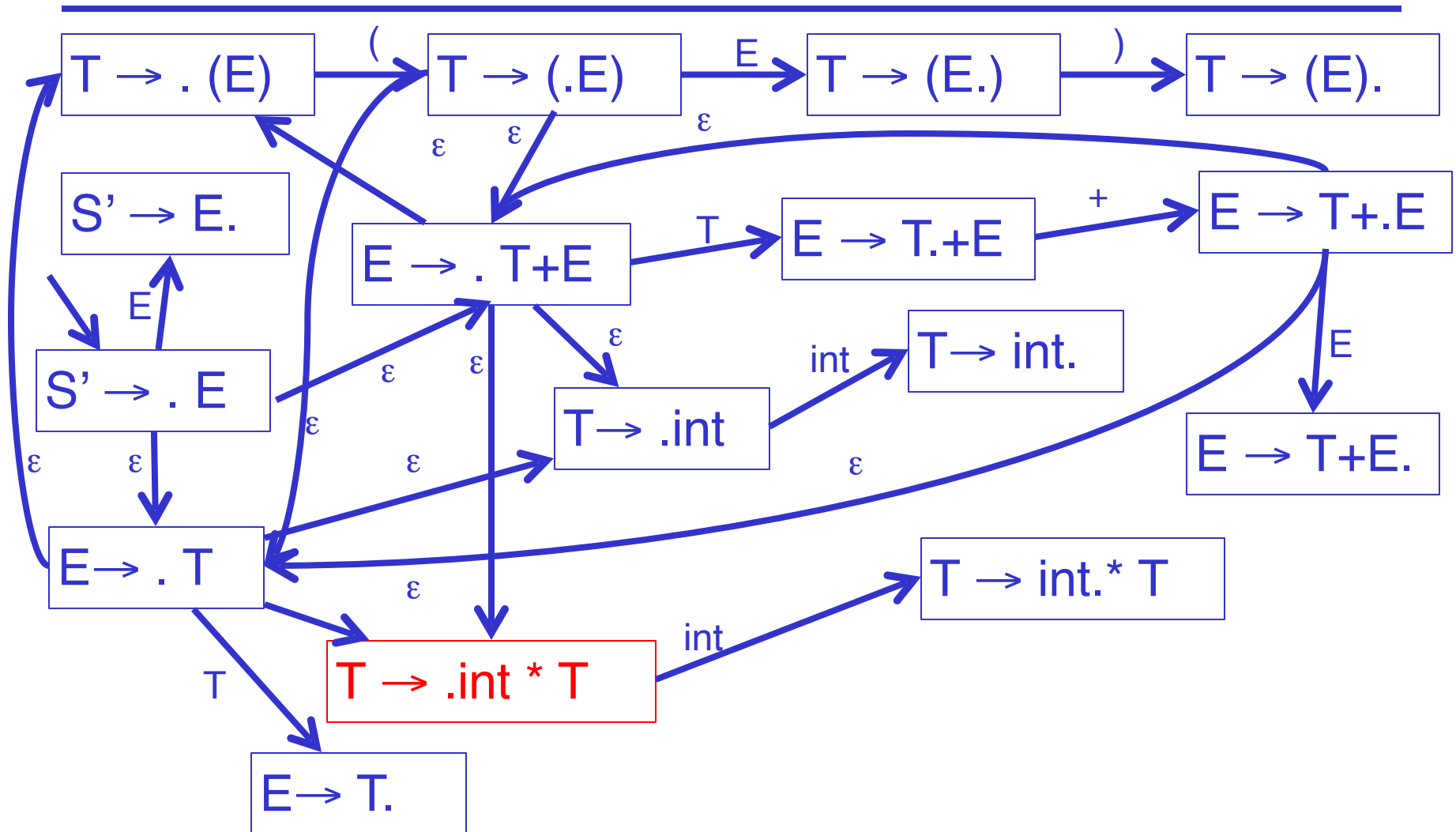
$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

NFA for Viable Prefixes



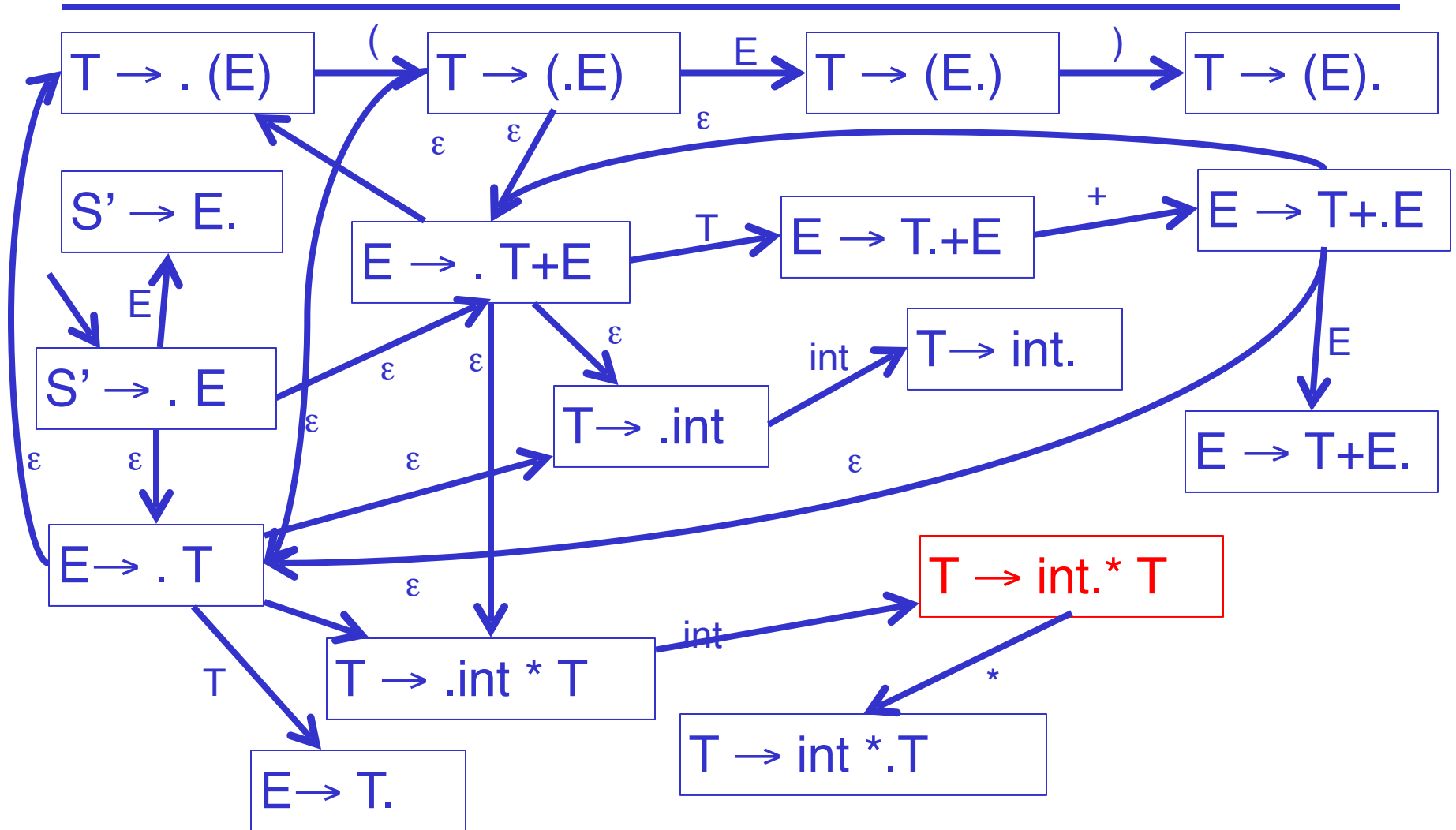
$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$

NFA for Viable Prefixes



$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$

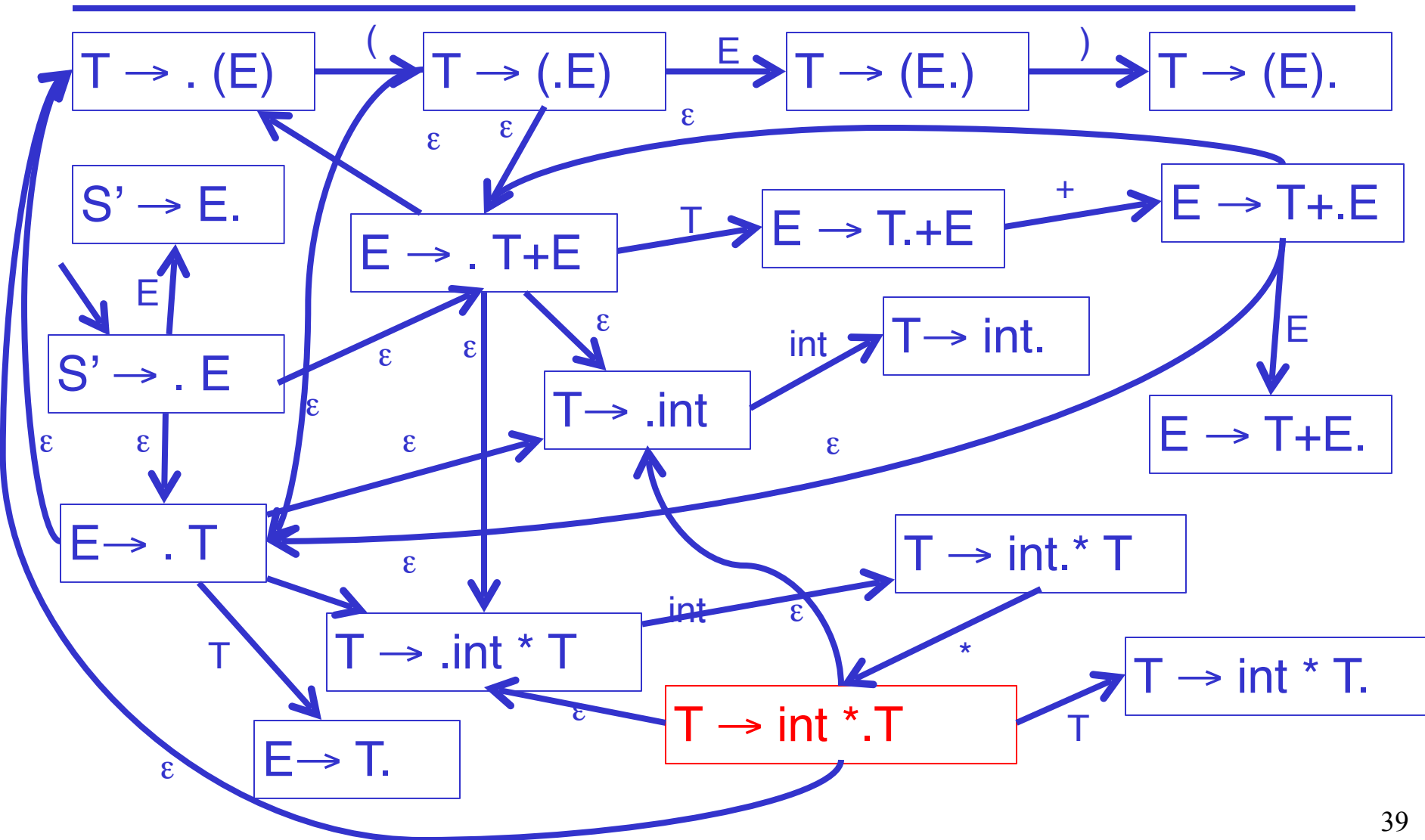
NFA for Viable Prefixes



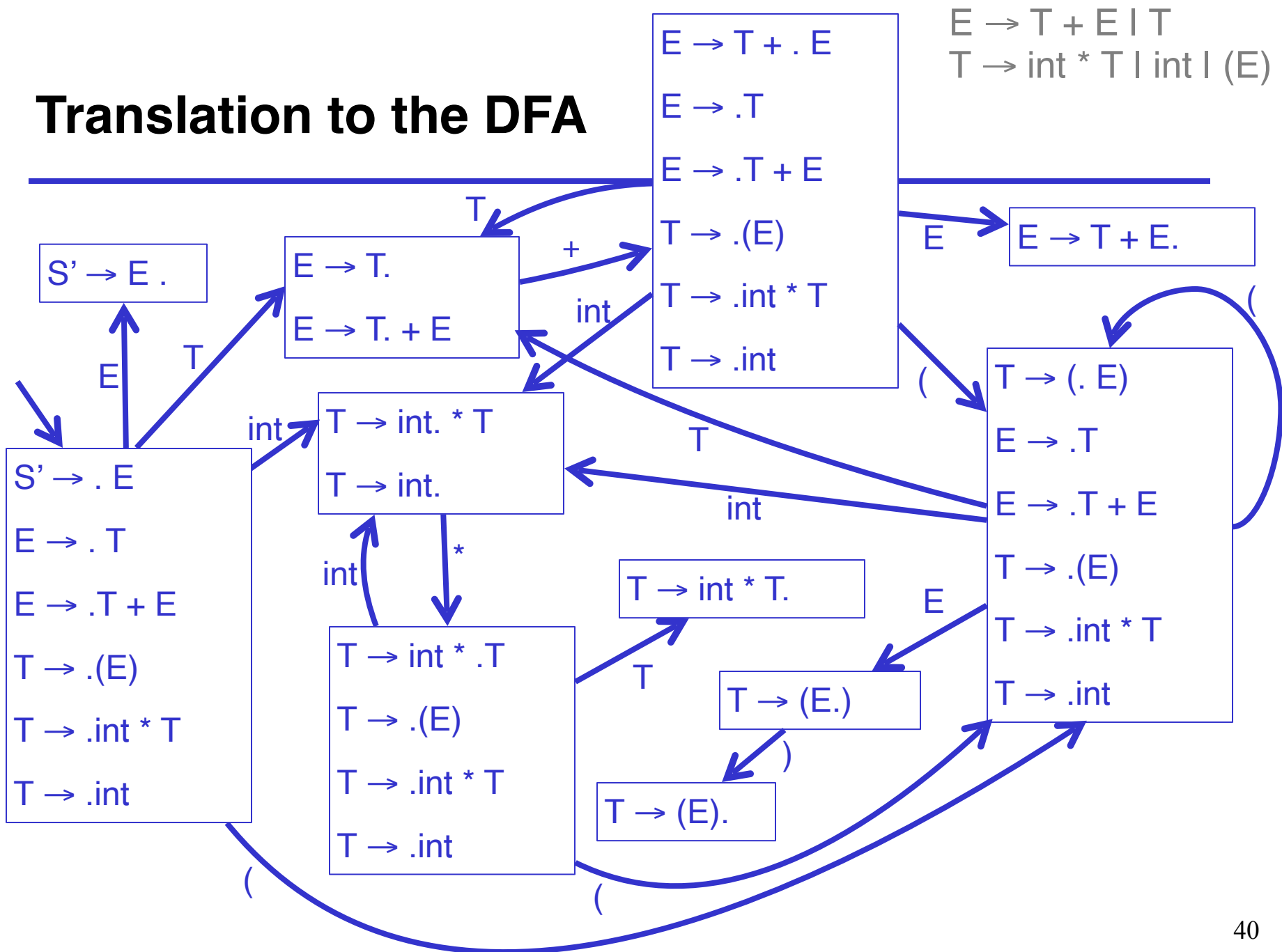
$$E \rightarrow T + E \mid T$$

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

NFA for Viable Prefixes



Translation to the DFA



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

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

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

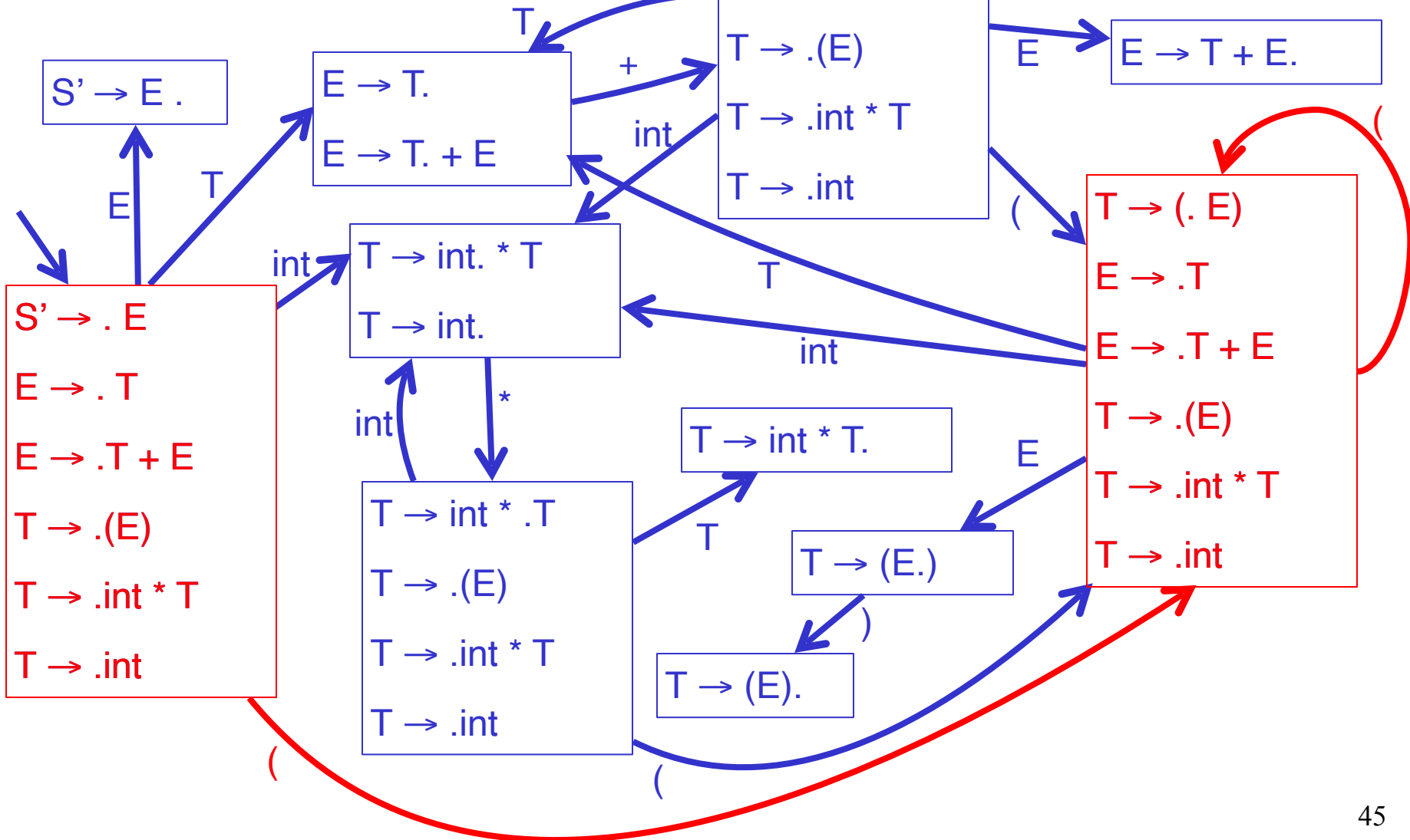
Valid Items Example

- An item is often valid for many prefixes
- Example: The item $T \rightarrow (.E)$ is valid for prefixes

(
(
(
(
...

Translation to the DFA

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



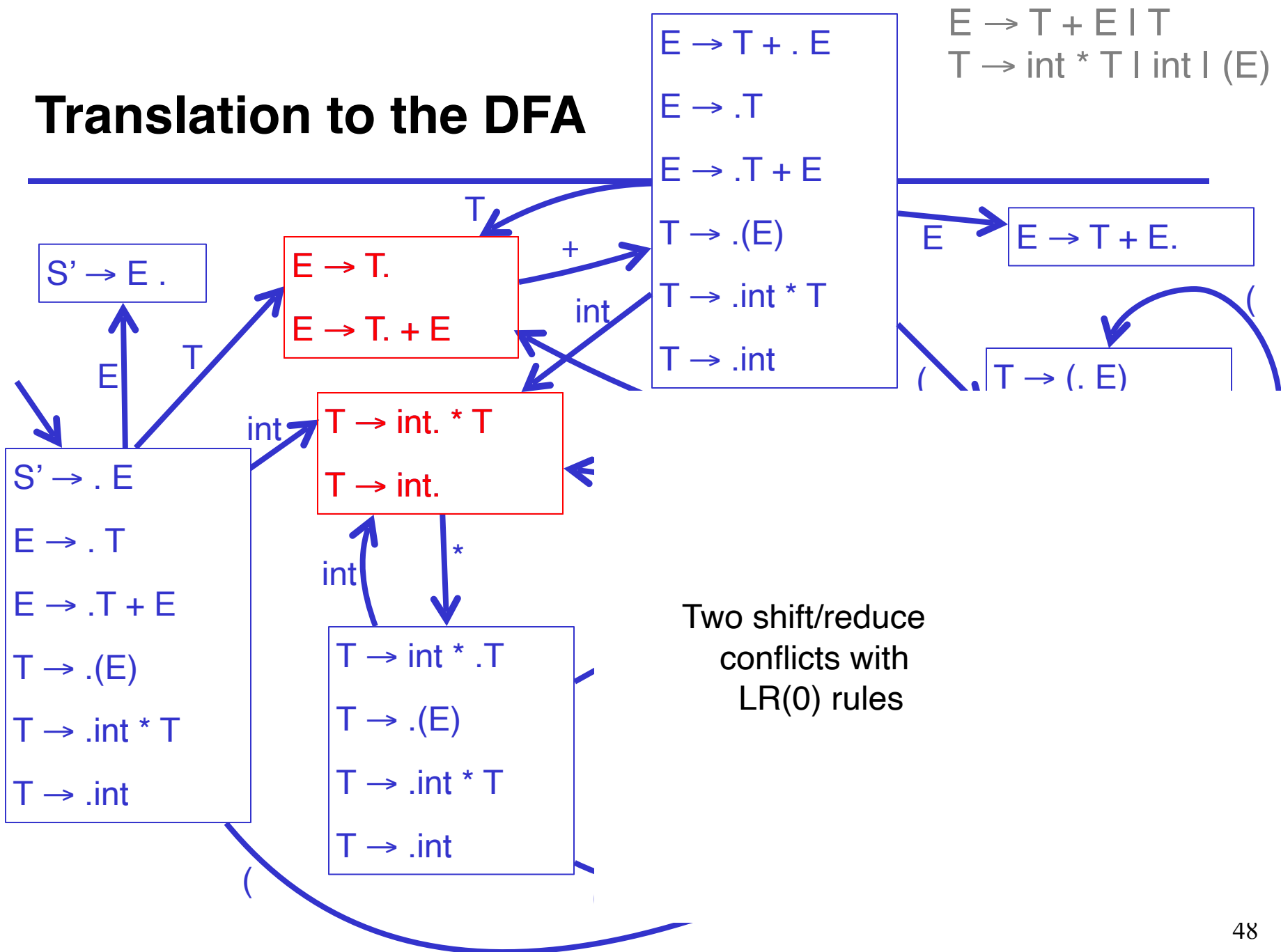
LR(0) Parsing

- Idea: Assume
 - stack contains α
 - next input is t
 - DFA on input α terminates in state s
- Reduce by $X \rightarrow \beta$ if
 - s contains item $X \rightarrow \beta$.
- Shift if
 - s contains item $X \rightarrow \beta.t\omega$
 - 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



SLR

- LR = “Left-to-right scan”
- SLR = “Simple LR”
- SLR improves on LR(0) shift/reduce heuristics
 - Fewer states have conflicts

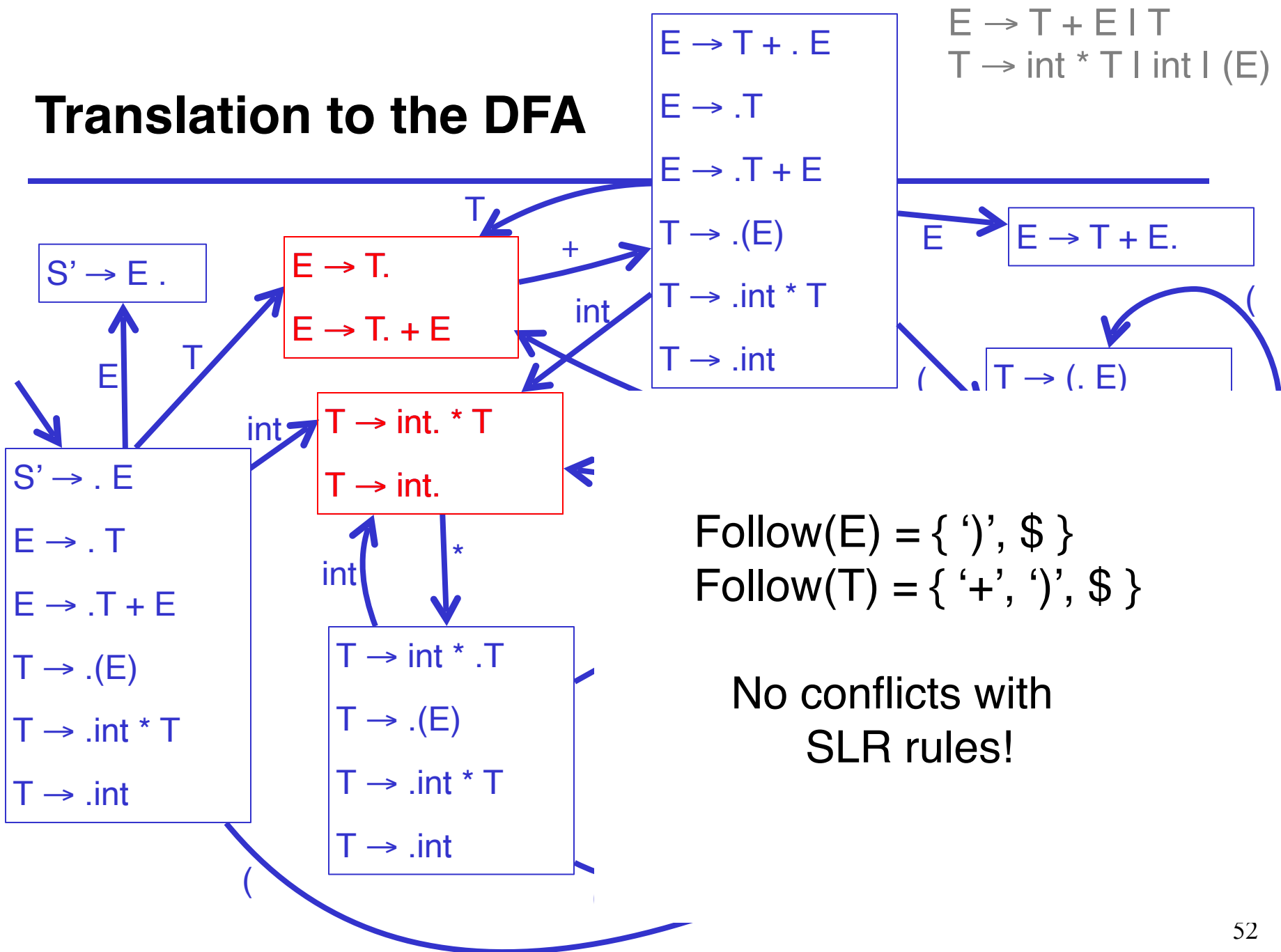
SLR Parsing

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

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



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:
 - $E \rightarrow E + E \mid E * E \mid (E) \mid \text{int}$
- The DFA for this grammar contains a state with the following items:
 - $E \rightarrow E * E \cdot$ $E \rightarrow E \cdot + E$
 - 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!

Naïve SLR Parsing Algorithm

1. Let M be DFA for viable prefixes of G
2. Let $|x_1 \dots x_n \$$ be initial configuration
3. Repeat until configuration is $SI\$$
 - Let $\alpha|w$ 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

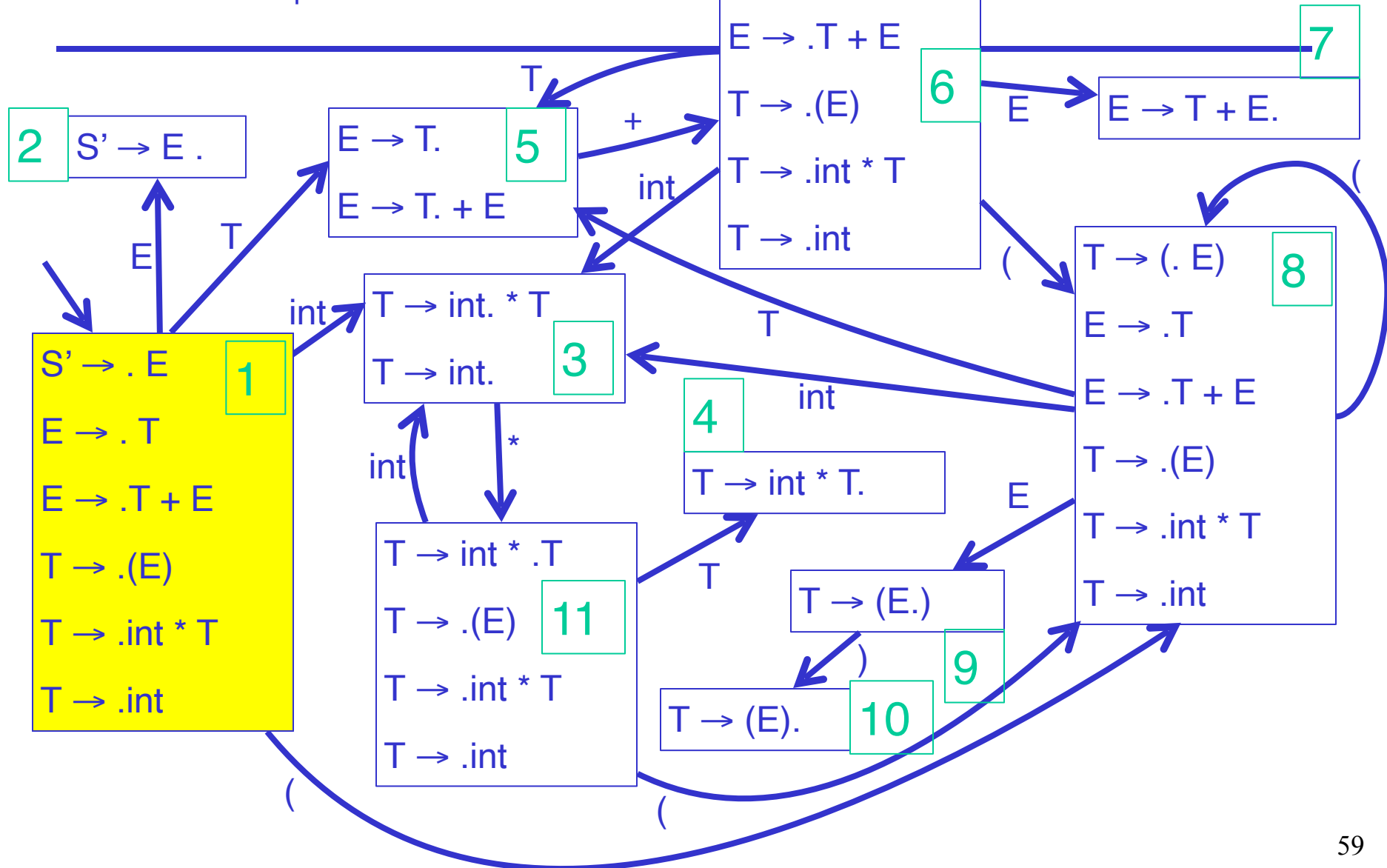
$$E \rightarrow T + E \mid T$$
$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

SLR Example

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

I int * int\$

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



$$E \rightarrow T + E \mid T$$

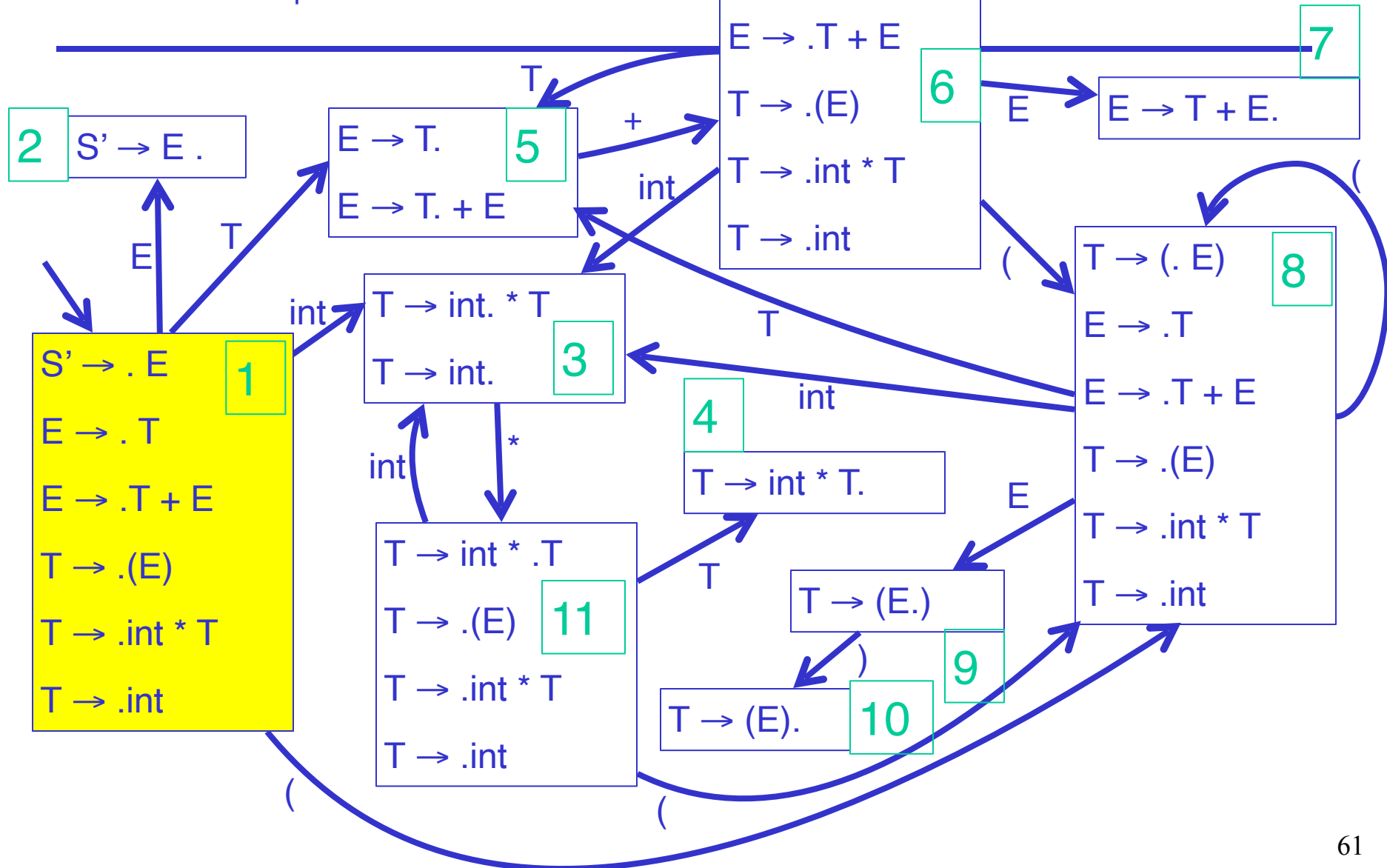
$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

SLR Example

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

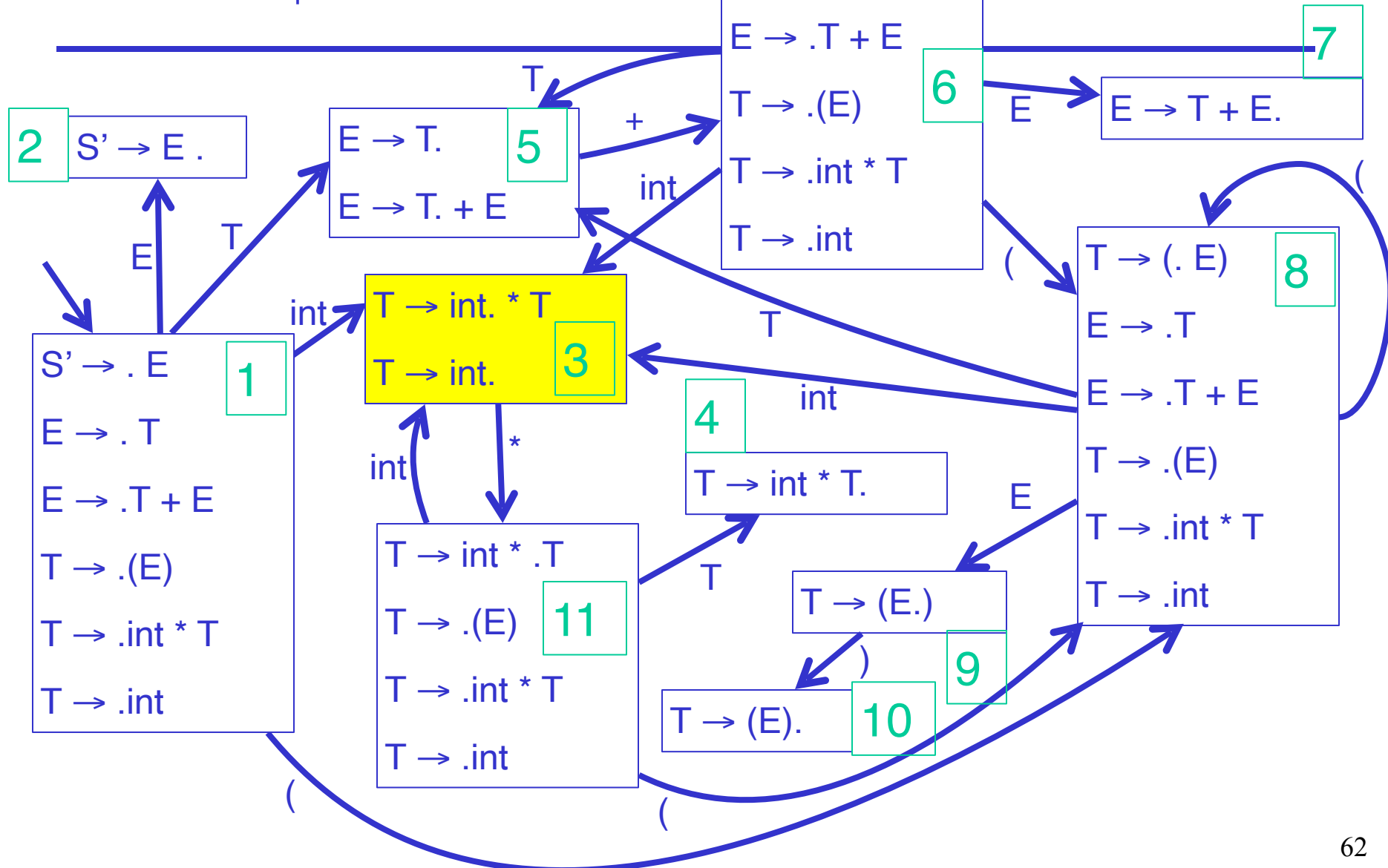
int | * int\$

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



int | * int\$

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



$$E \rightarrow T + E \mid T$$

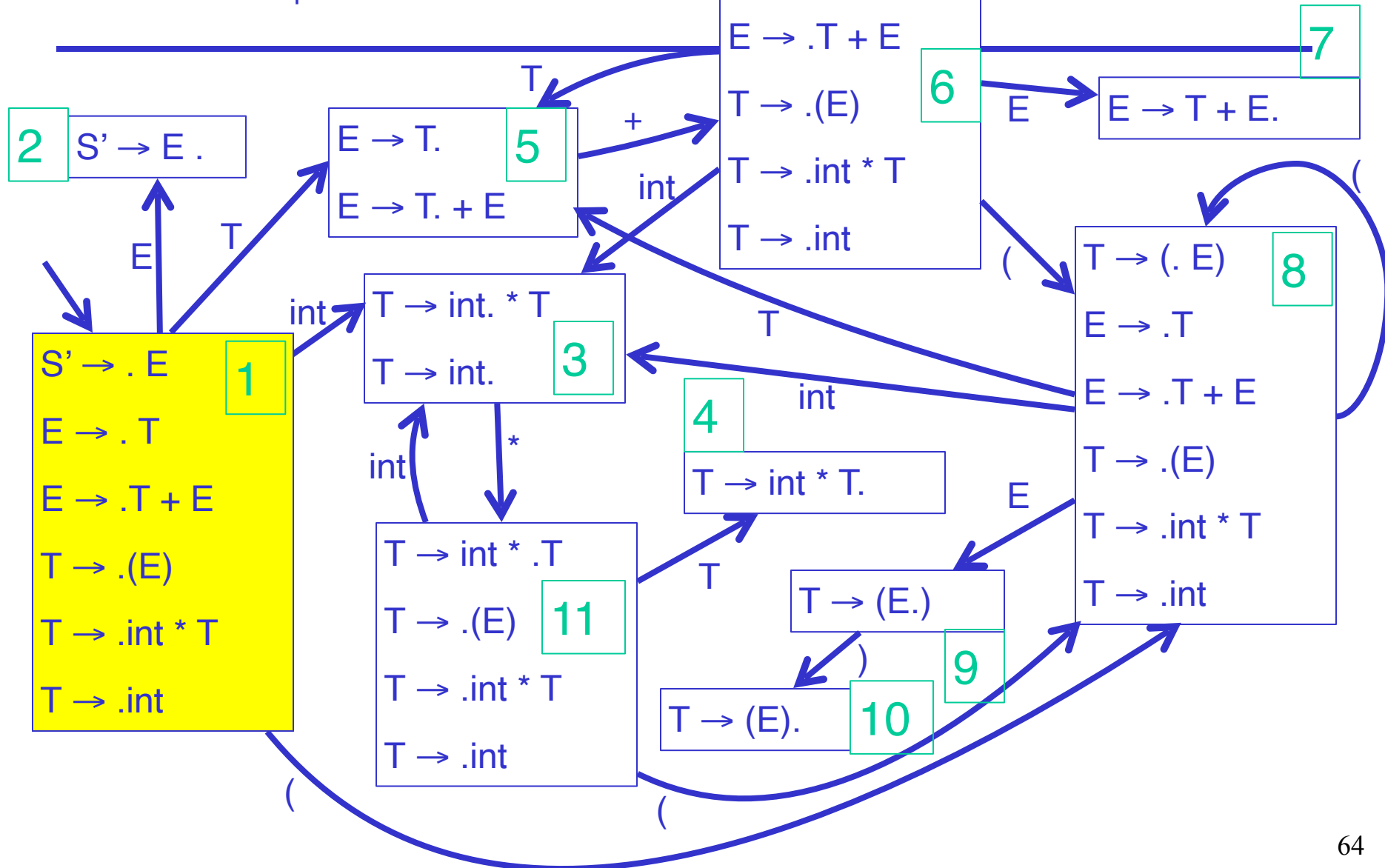
$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

SLR Example

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

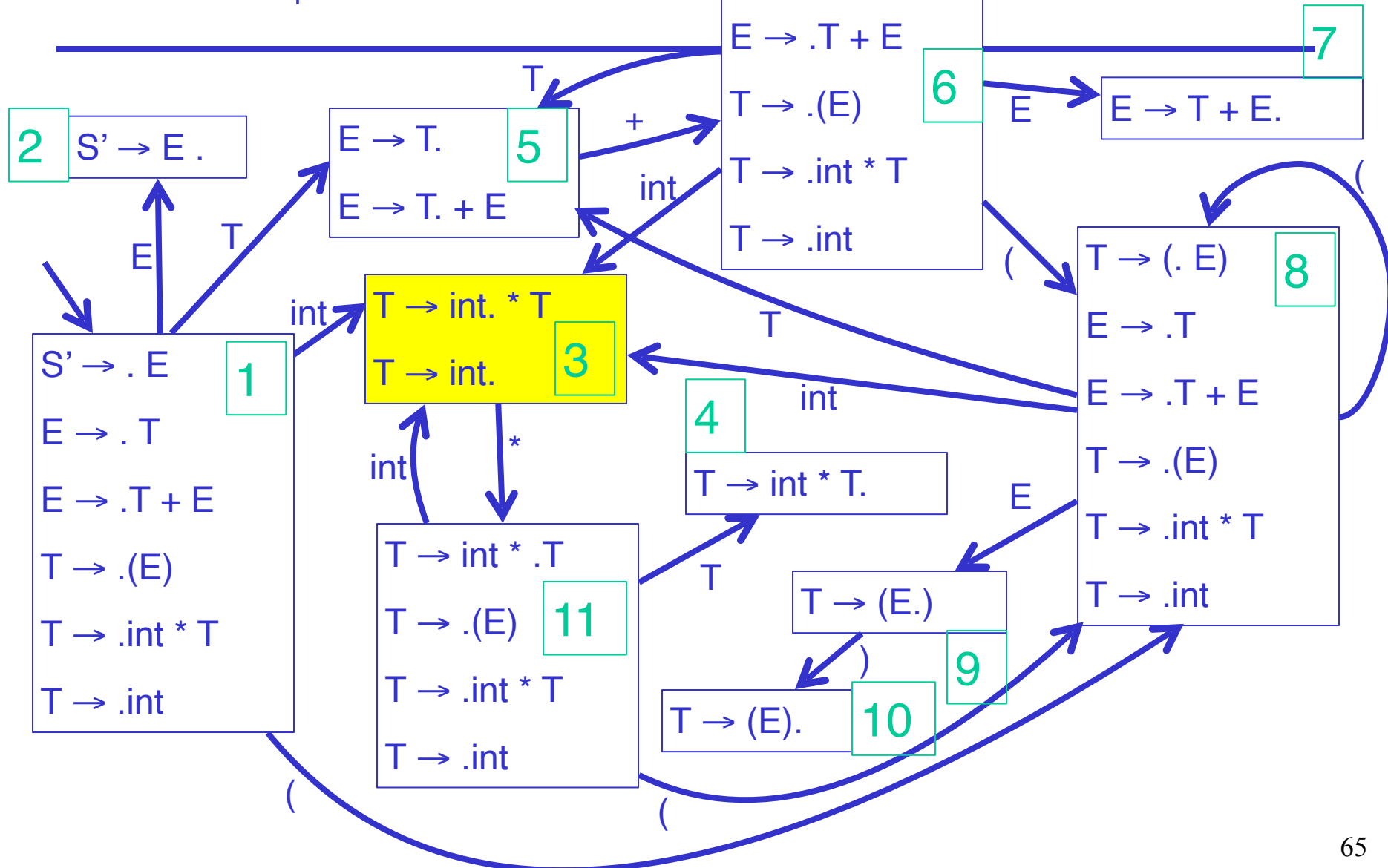
int * | int\$

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



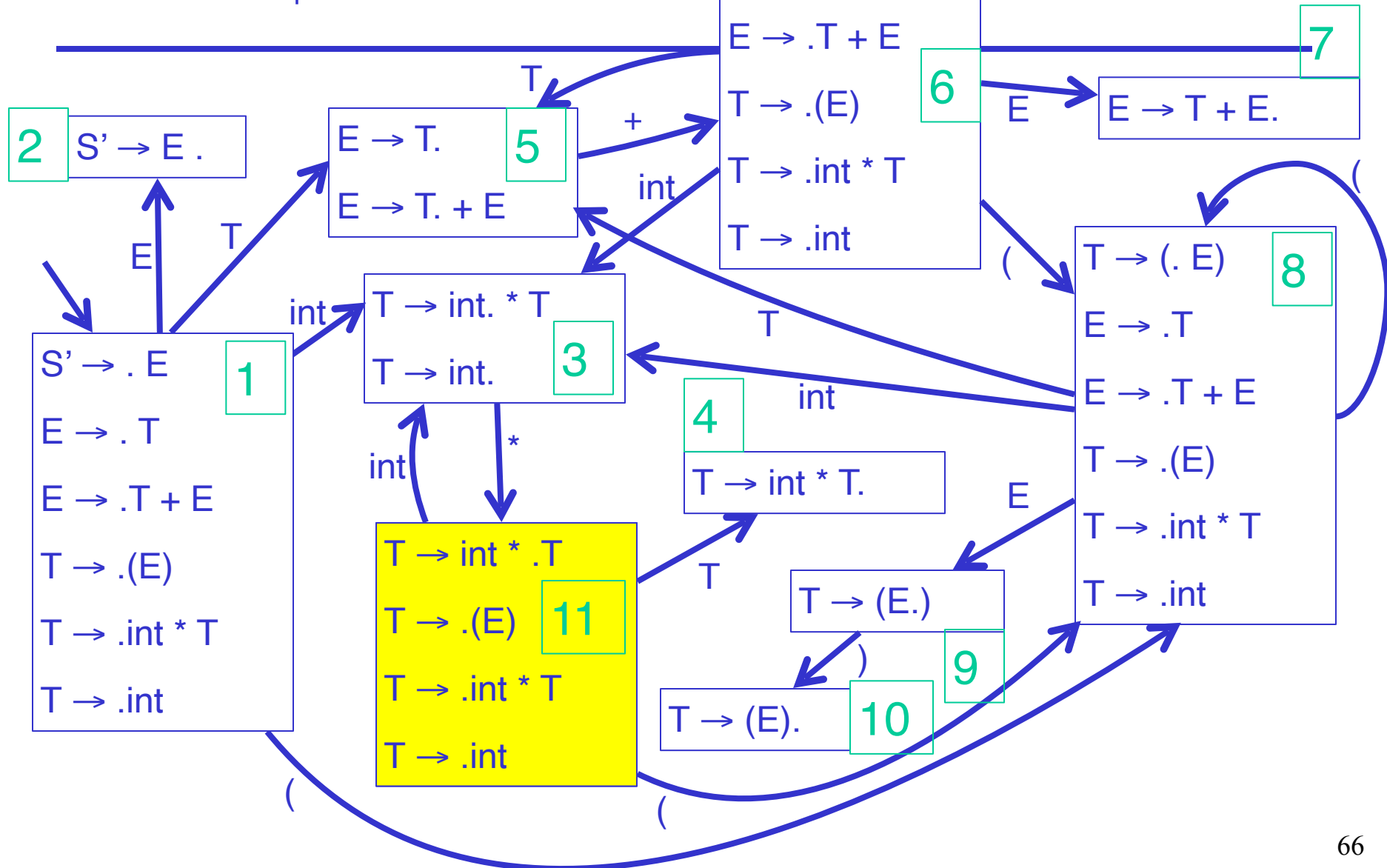
int * | int\$

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



int * | int\$

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



$$E \rightarrow T + E \mid T$$

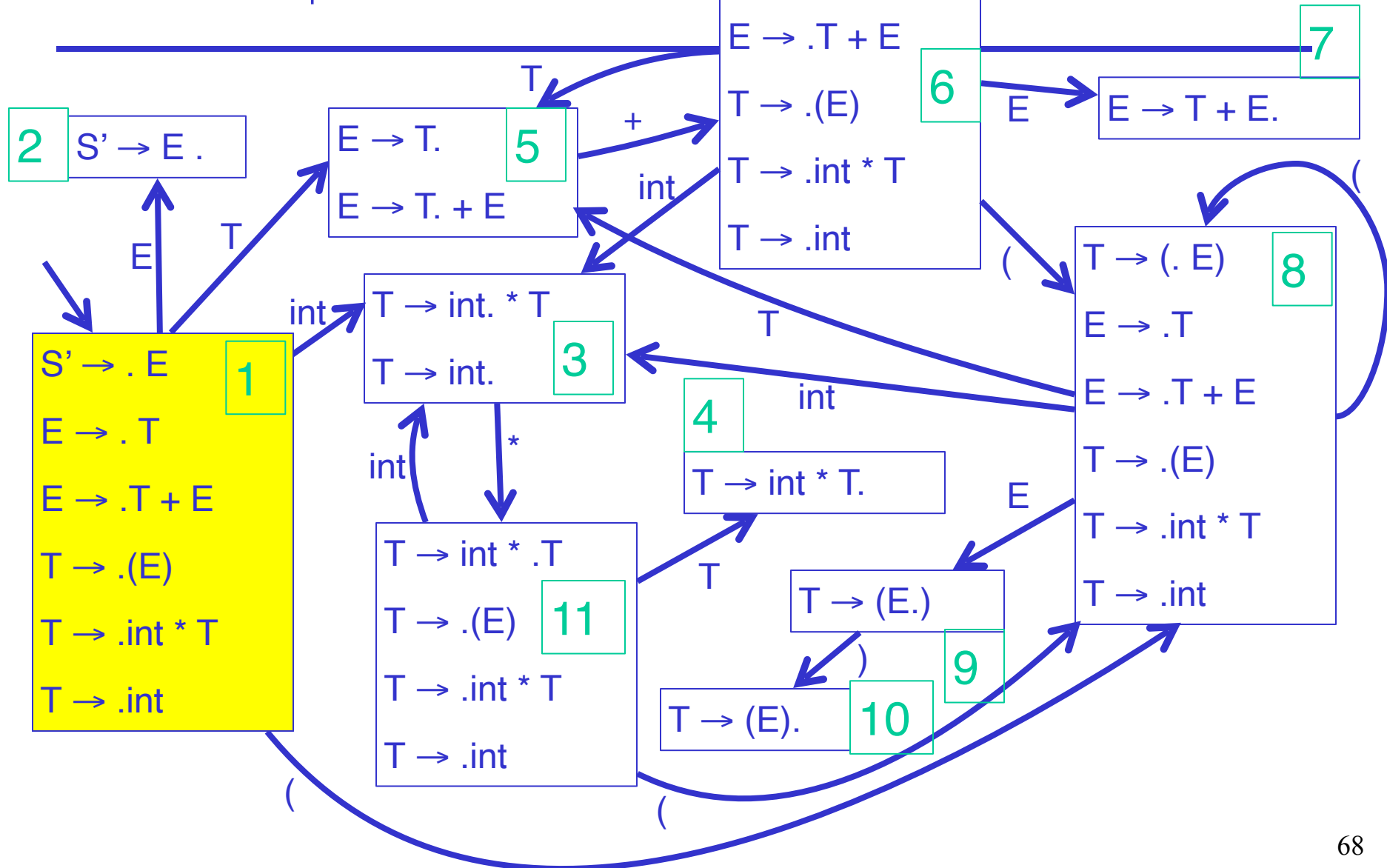
$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

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 $T \rightarrow \text{int}$

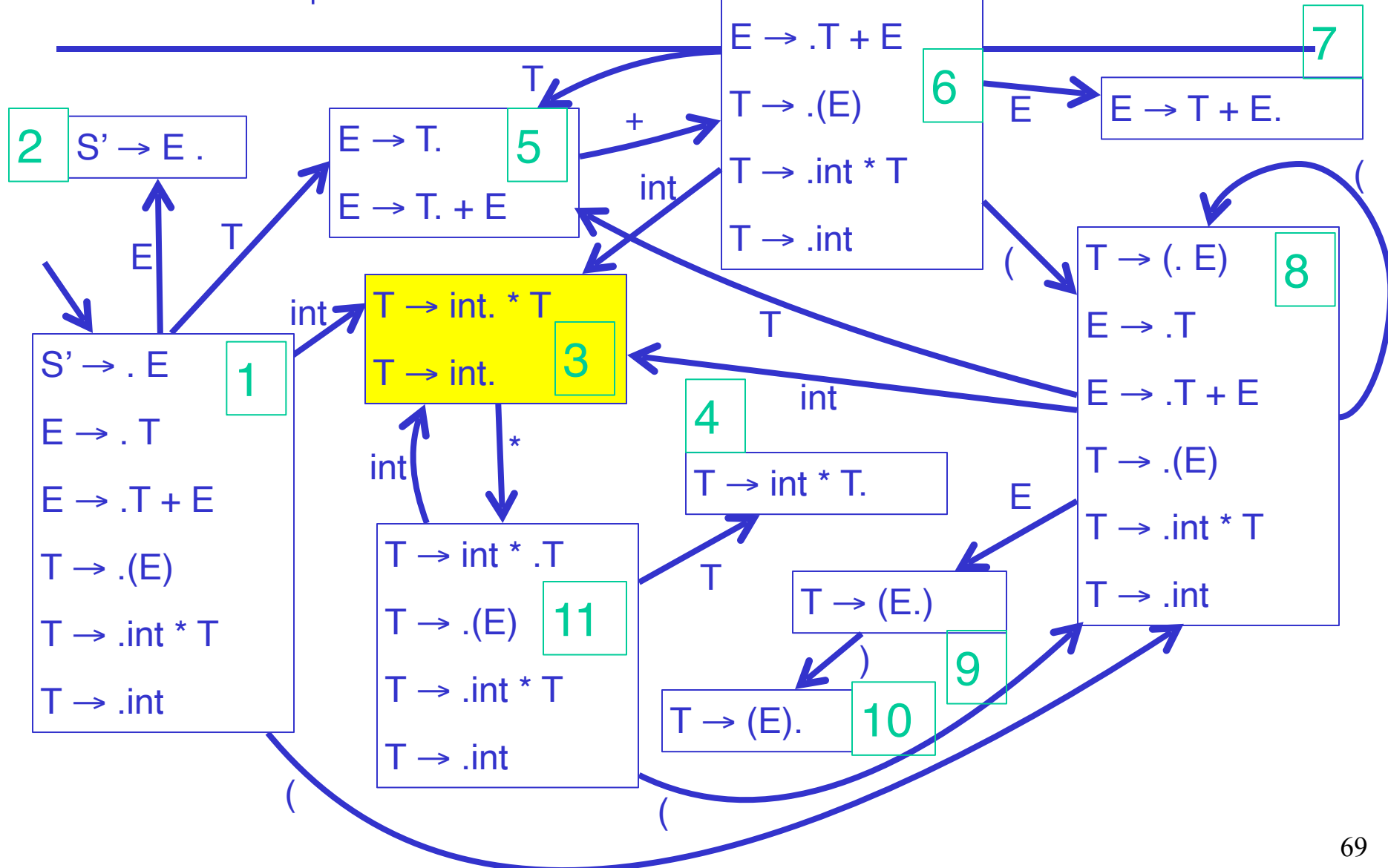
int * int | \$

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



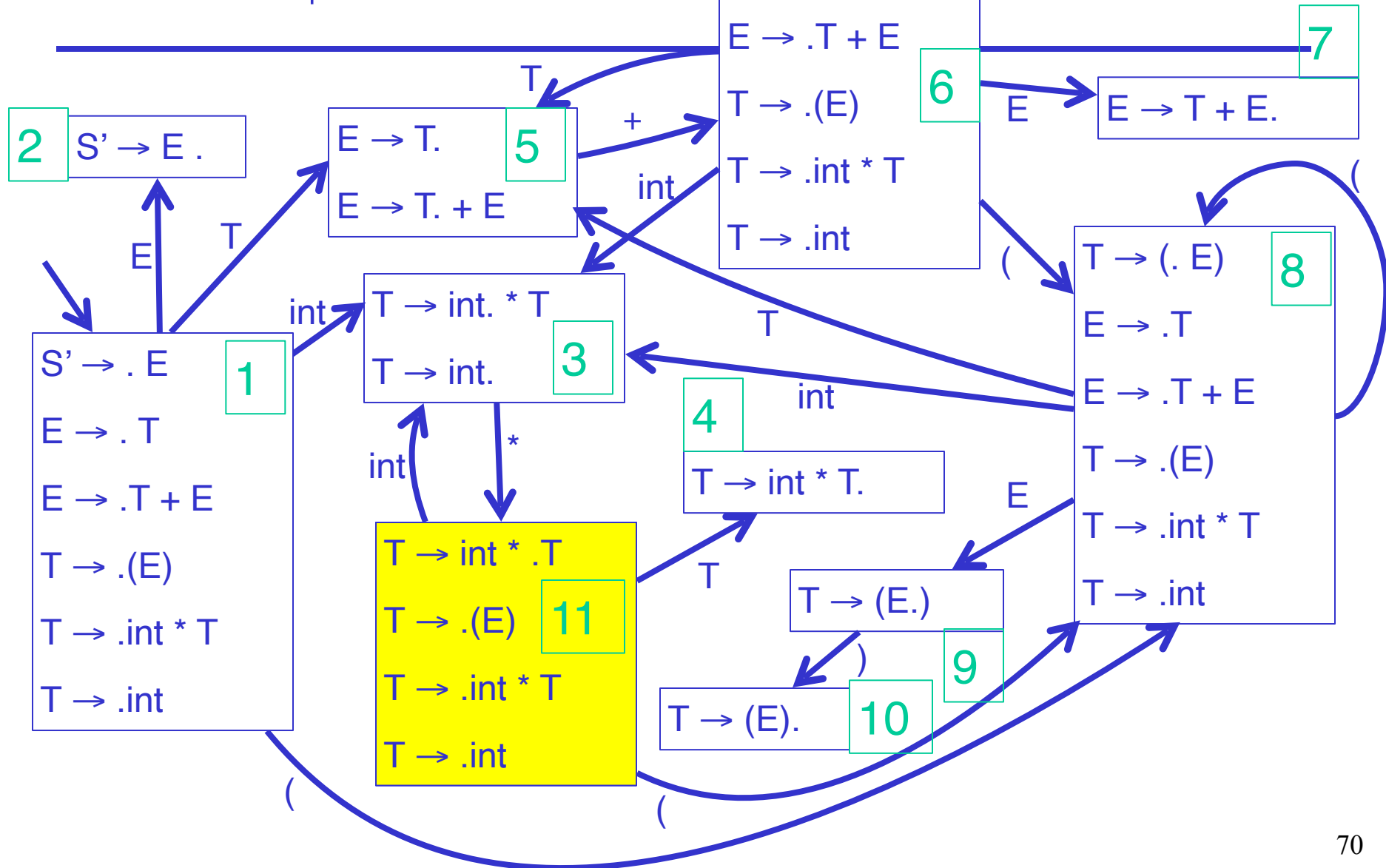
int * int | \$

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



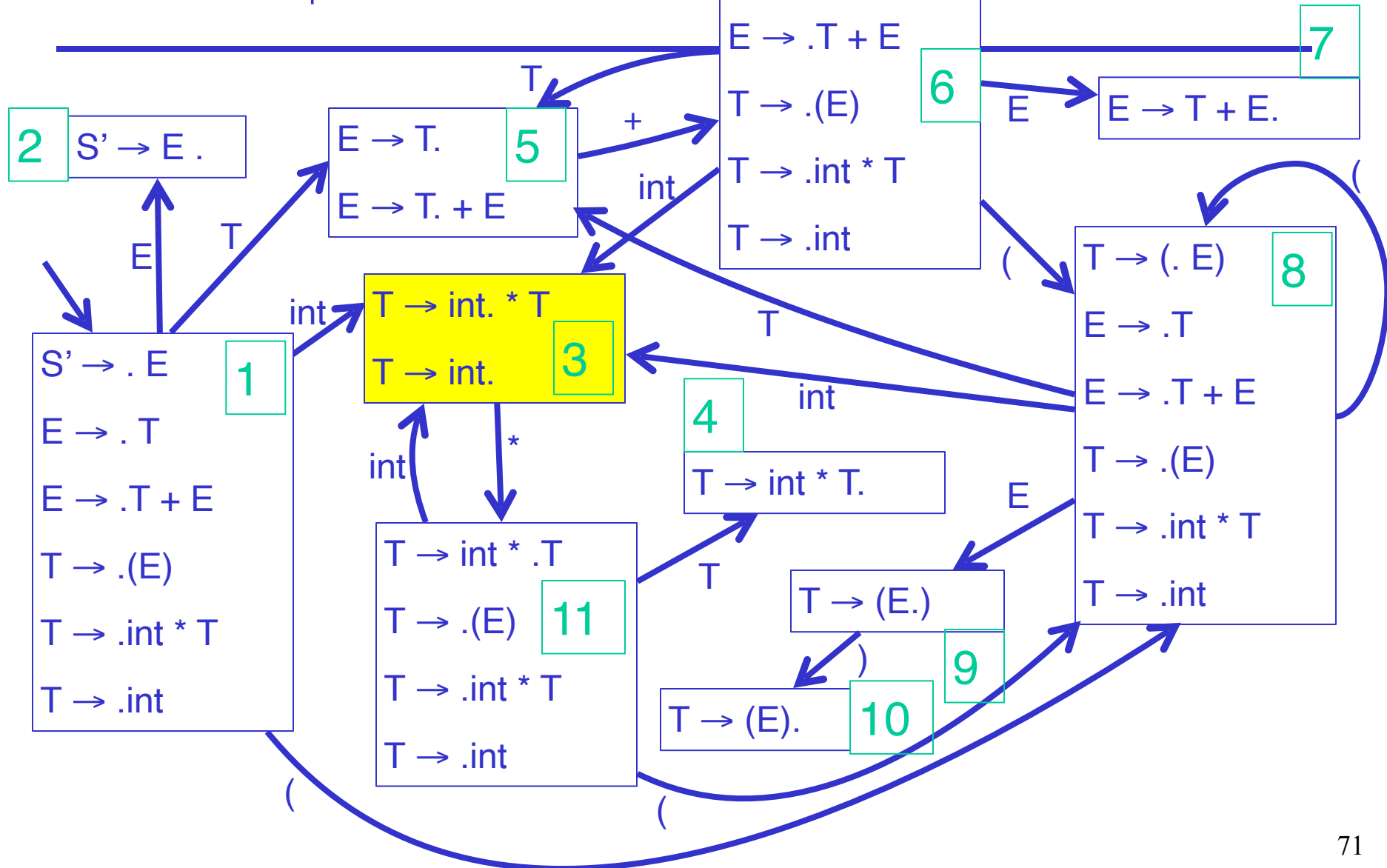
int * int | \$

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



int * int | \$

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



$$E \rightarrow T + E \mid T$$

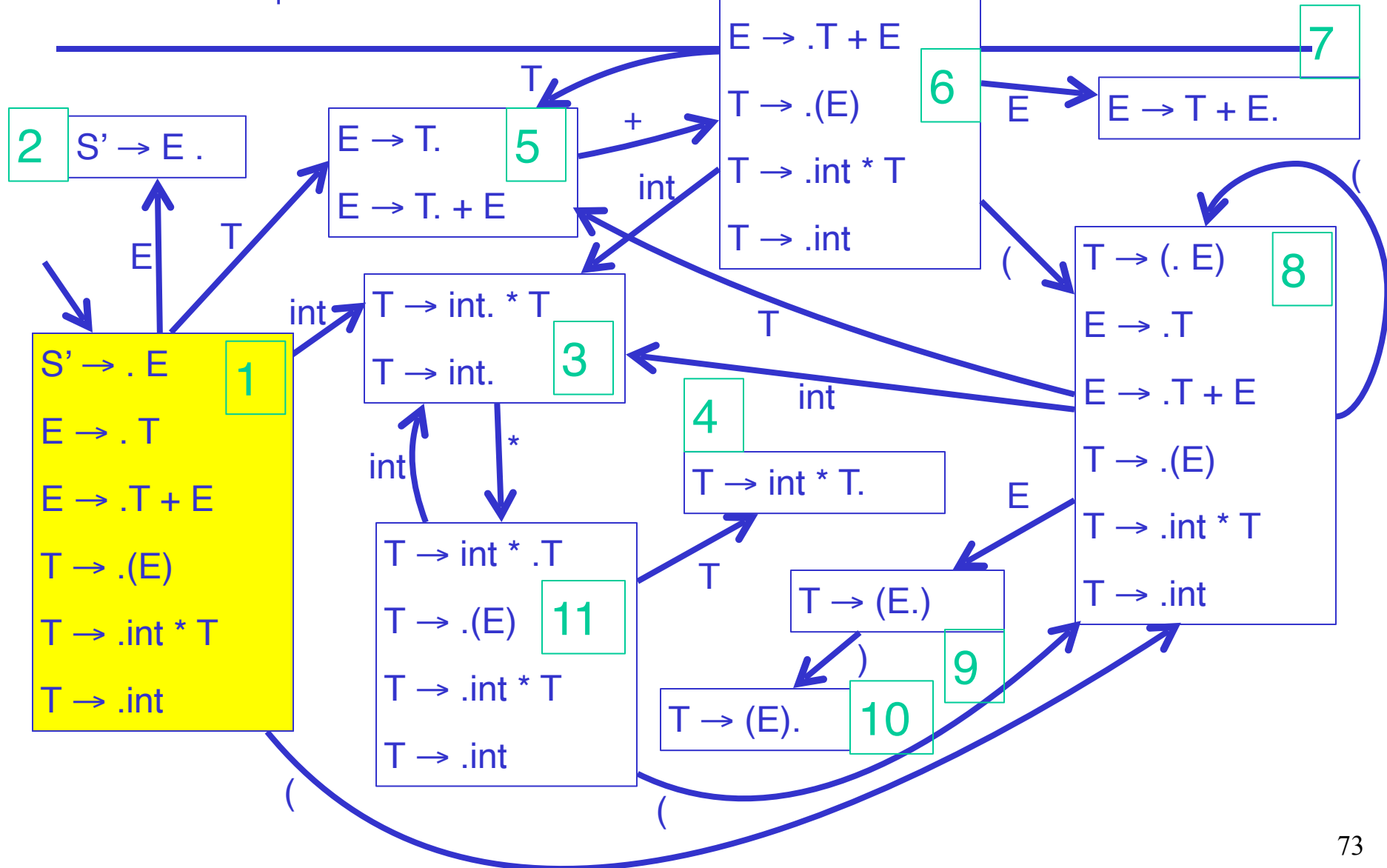
$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

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 $T \rightarrow \text{int}$
int * T \$	4 \$ ∈ Follow(T)	reduce $T \rightarrow \text{int} * T$

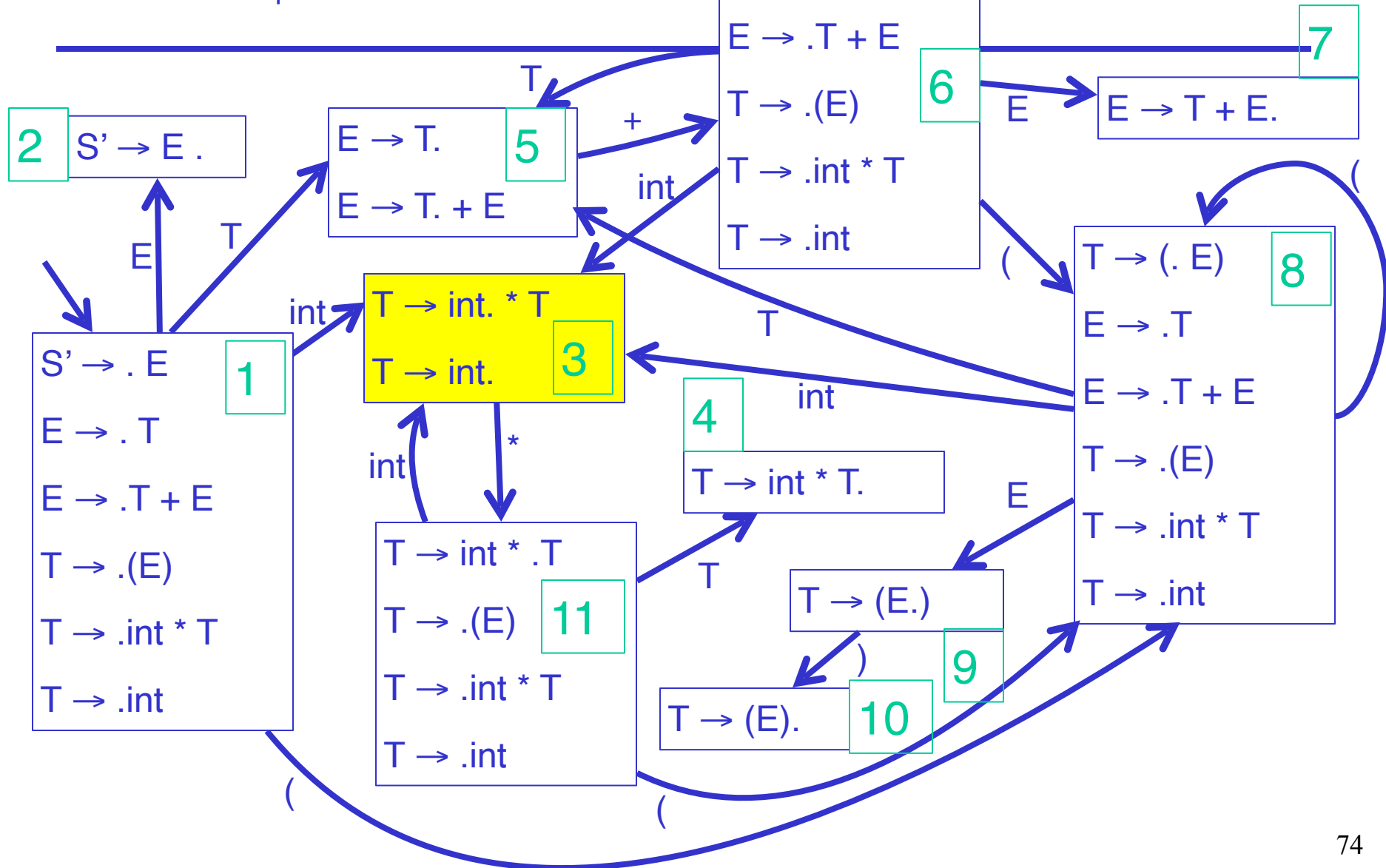
int * T | \$

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



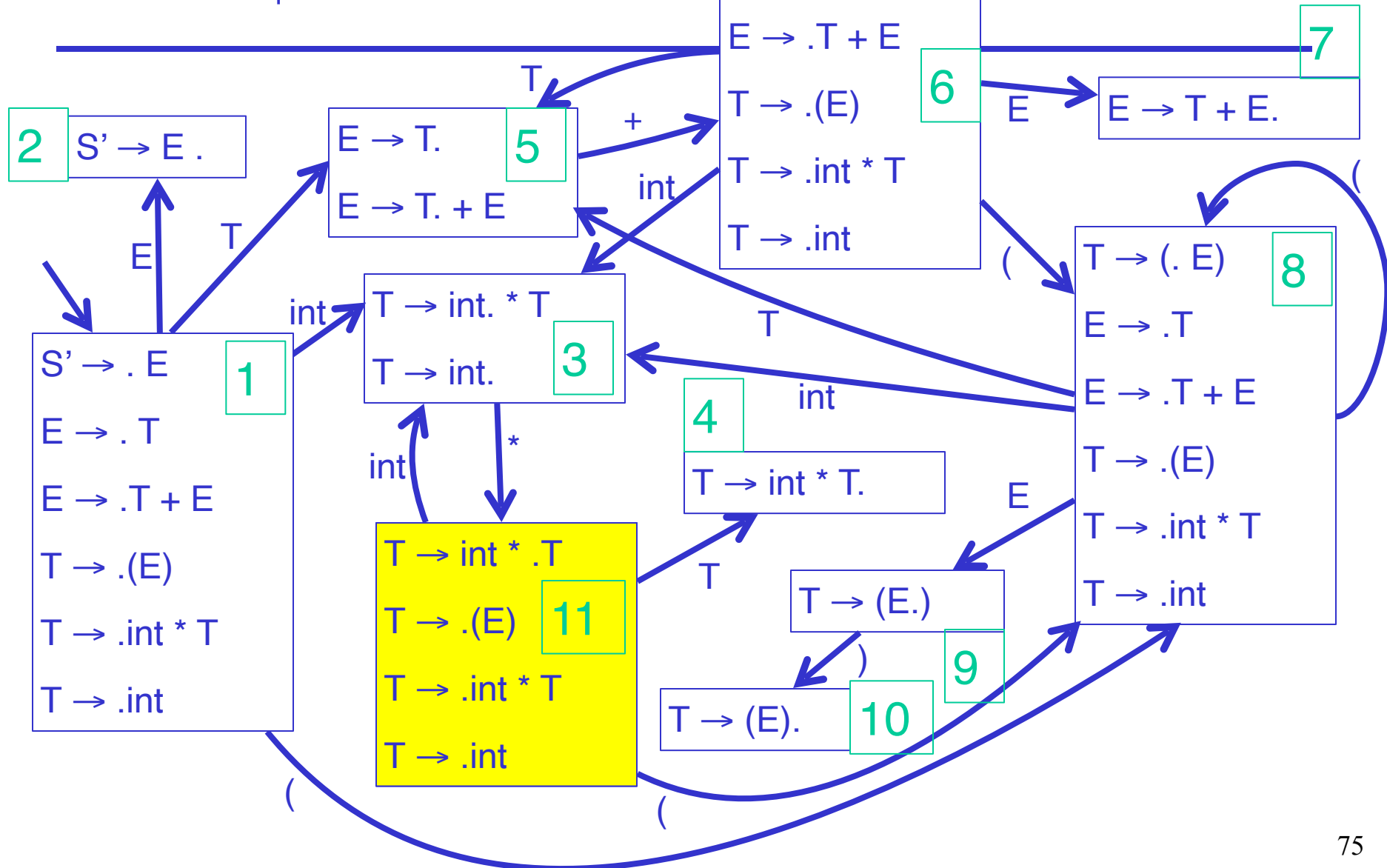
int * T | \$

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



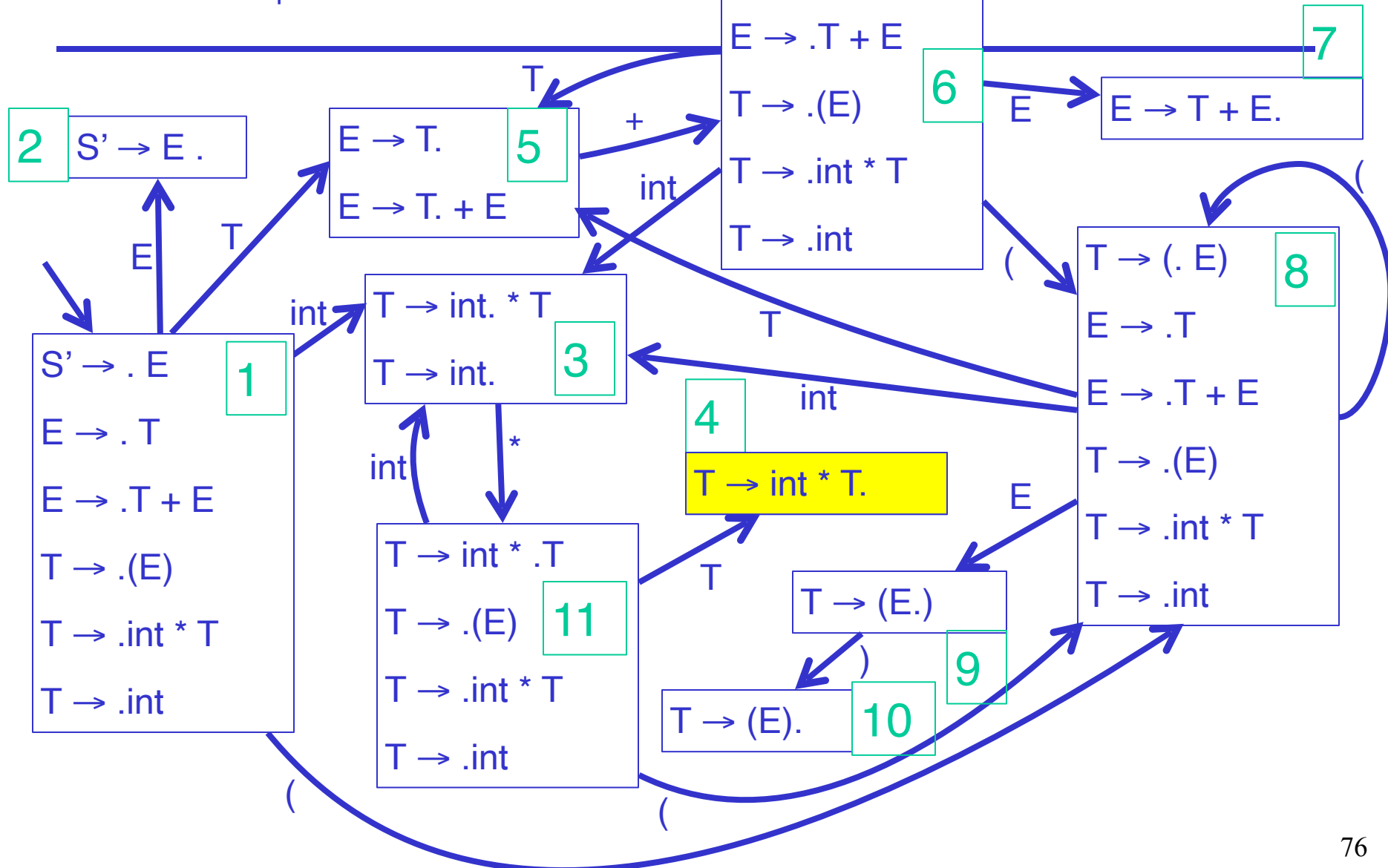
int * T | \$

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



int * T | \$

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



$$E \rightarrow T + E \mid T$$

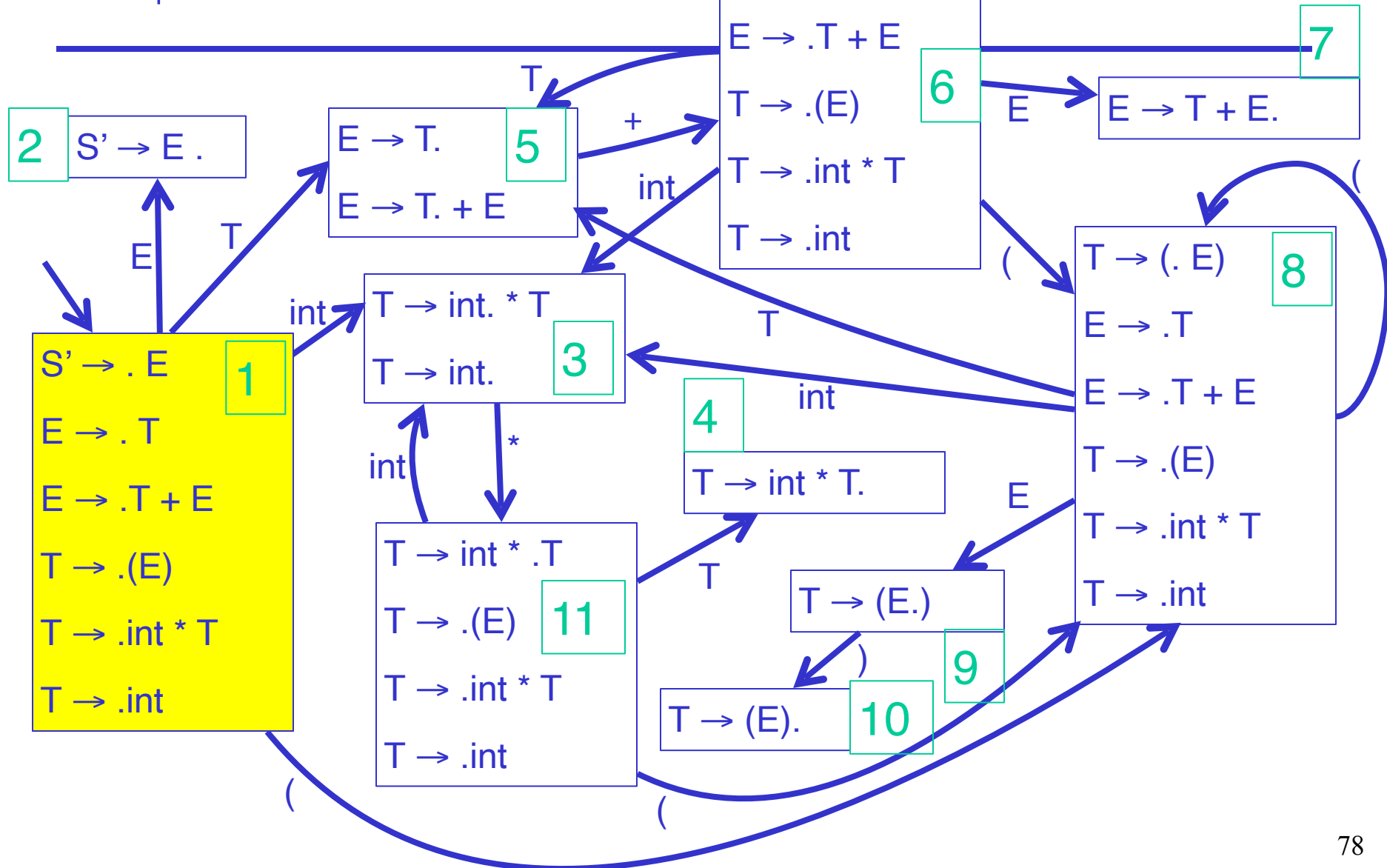
$$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$$

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 $\$ \in \text{Follow}(T)$	reduce $T \rightarrow \text{int}$
int * T \$	4 $\$ \in \text{Follow}(T)$	reduce $T \rightarrow \text{int} * T$
T \$	5 $\$ \in \text{Follow}(T)$	reduce $E \rightarrow T$

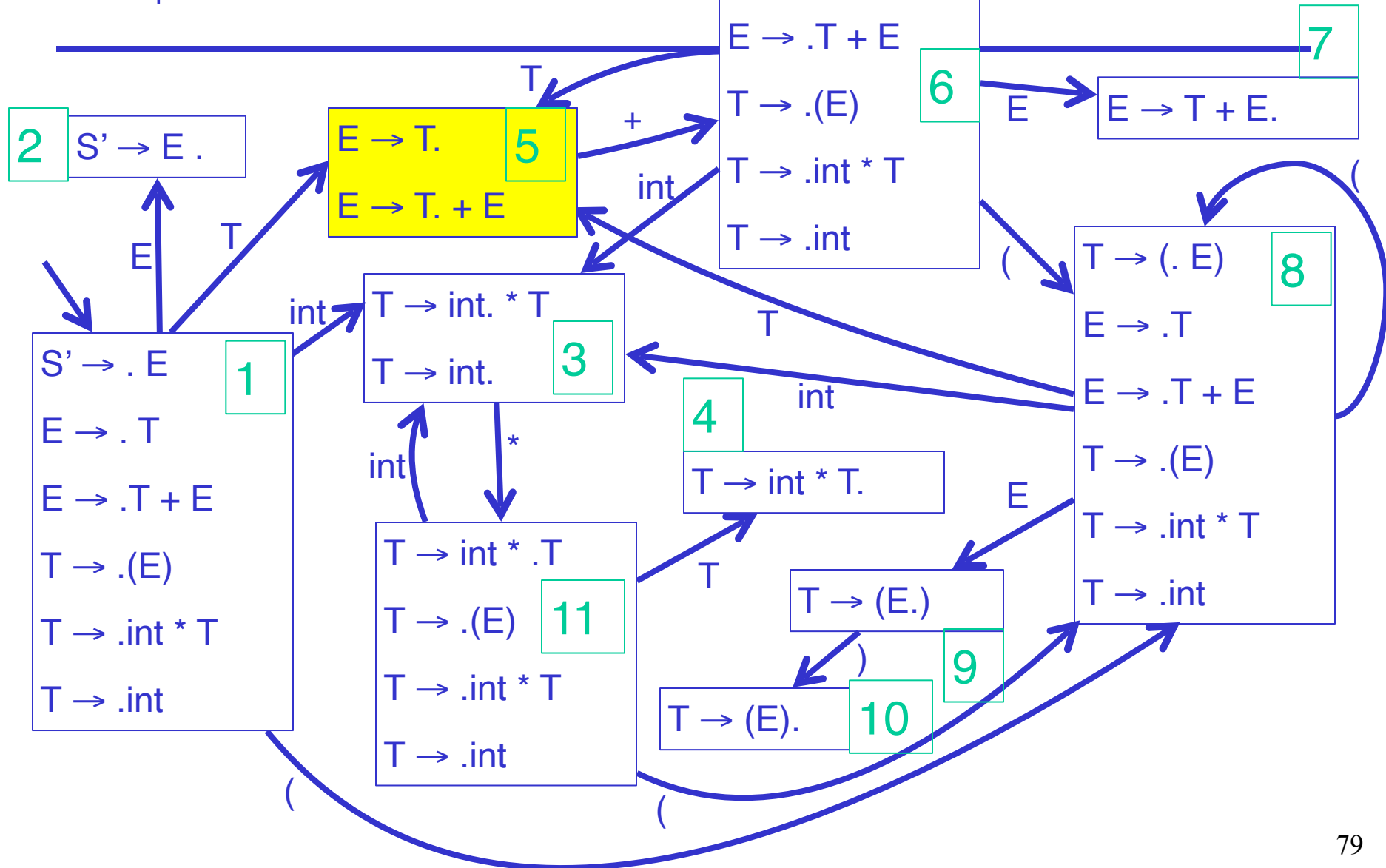
T | \$

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



T | \$

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



$$E \rightarrow T + E \mid T$$

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

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 $T \rightarrow \text{int}$
int * T \$	4 \$ ∈ Follow(T)	reduce $T \rightarrow \text{int} * T$
T \$	5 \$ ∈ Follow(T)	reduce $E \rightarrow T$
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

$\langle \text{symbol}_1, \text{state}_1 \rangle \dots \langle \text{symbol}_n, \text{state}_n \rangle$

state_n is the final state of the DFA on $\text{symbol}_1 \dots \text{symbol}_n$

- Detail: The bottom of the stack is $\langle \text{dummy}, \text{start} \rangle$ where
 - any is any dummy symbol
 - start is the start state of the DFA

Goto (DFA) Table

- Define $\text{goto}[i,A] = j$ if $\text{state}_i \xrightarrow{A} \text{state}_j$
- **goto** is just the transition function of the DFA
 - One of two parsing tables

Refined Parser Moves

- Shift x
 - Push $\langle a, x \rangle$ on the stack
 - a is current input
 - x is a DFA state
- Reduce $X \rightarrow \alpha$
 - 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 $\text{goto}[i,t] = k$ then $\text{action}[i,t] = \text{shift } k$
- If s_i has item $X \rightarrow \alpha.$ and $t \in \text{Follow}(X)$ and $X \neq S'$ then $\text{action}[i,t] = \text{reduce } X \rightarrow \alpha$
- If s_i has item $S' \rightarrow S.$ then $\text{action}[i,\$] = \text{accept}$
- Otherwise, $\text{action}[i,t] = \text{error}$

SLR Parsing Algorithm

Let input = $w\$$ be initial input

Let $j = 0$

Let DFA state 1 be the one with item $S' \rightarrow .S$

Let stack = $\langle \text{dummy}, 1 \rangle$

repeat

case action[top_state(stack), input[j]] of

shift k : push $\langle \text{input}[j++], k \rangle$ // $\langle \text{symbol}, \text{state} \rangle$

reduce $X \rightarrow \alpha$:

pop $|\alpha|$ pairs,

push $\langle X, \text{goto}[\text{top_state}(\text{stack}), X] \rangle$

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)
 - $[T \rightarrow \cdot \text{int} * T, \$]$ means
 - After seeing $T \rightarrow \text{int} * T$ reduce if lookahead is $\$$
 - More accurate than just using follow sets
 - See Dragon Book
 - Take a look at the LR(1) automaton for your parser!