Bottom-Up Parsing

CS143

Lecture 8

Instructor: Fredrik Kjolstad

Slide design by Prof. Alex Aiken, with modifications

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 the preferred method
- Concepts today, algorithms next time

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

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

The Idea

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

int * int + int	$T \to$ int
int * T + int	$T \to$ int * T
T + int	$T \to$ int
T + T	$E \to T$
T + E	$E \to T + E$
E	

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

Observation

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

int * int + int	$T \rightarrow$ int
int * T + int	$T \rightarrow$ int * T
T + int	$T \rightarrow$ int
T + T	$E \rightarrow T$
T + E	$E \rightarrow T + E$
E	

Important Fact #1

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse

A Bottom-up Parse

E → T + E | T
T → int * T | int | (E)

int * int + int

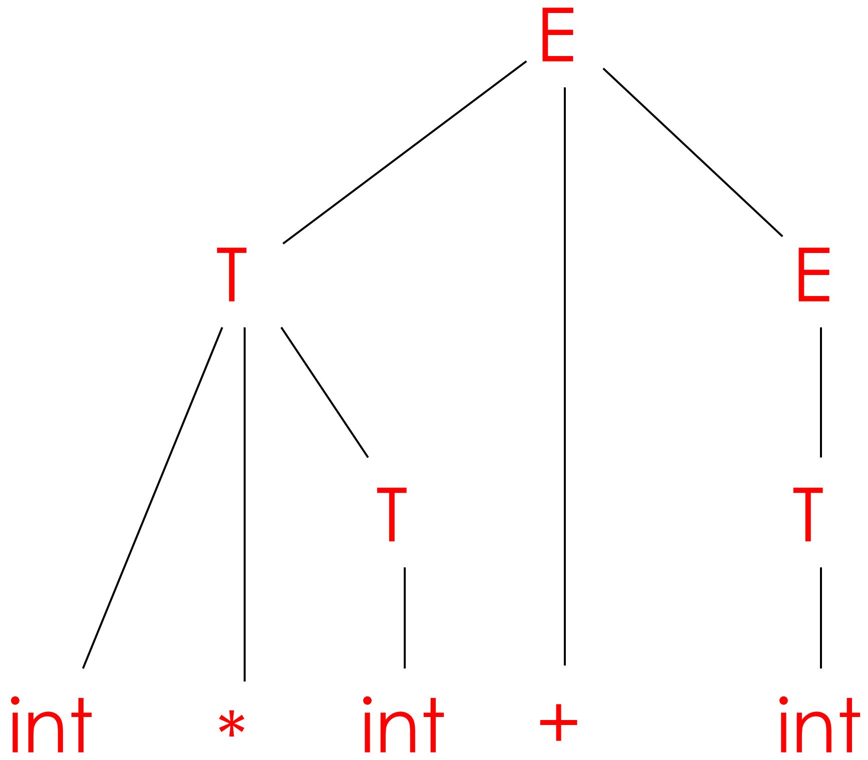
int * T + int

T + int

T + T

T + E

E



A Bottom-up Parse in Detail (1)

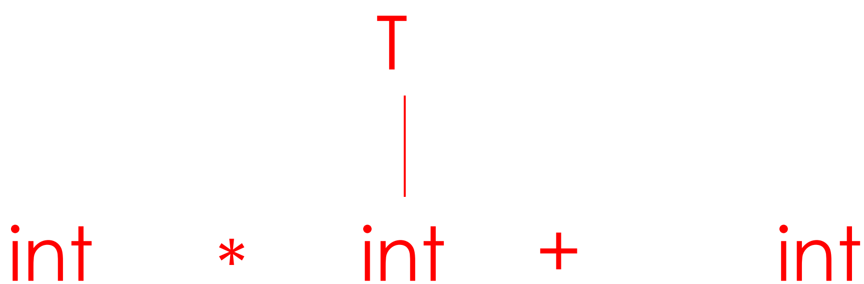
int * int + int

int * int + int

A Bottom-up Parse in Detail (2)

int * int + int

int * T + int

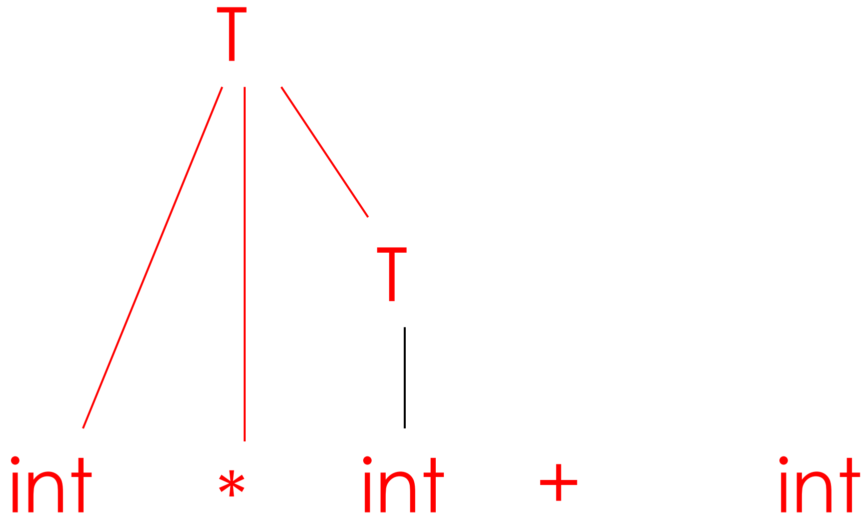


A Bottom-up Parse in Detail (3)

int * int + int

int * T + int

T + int



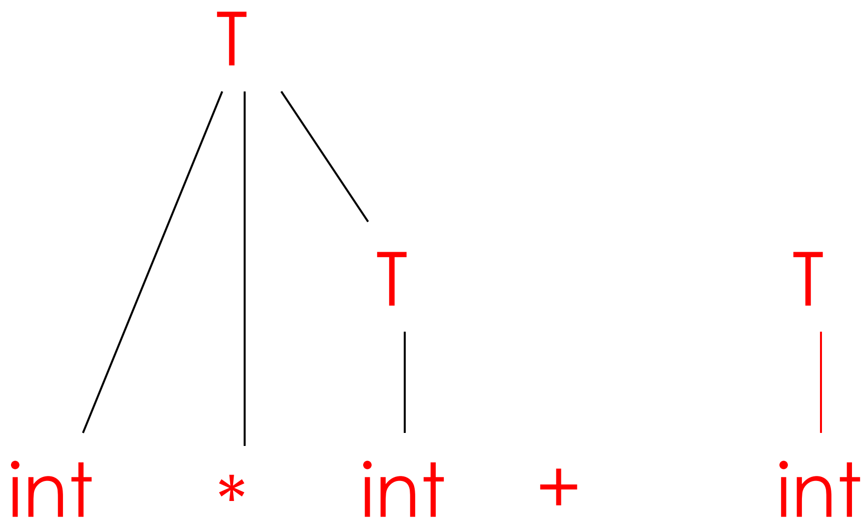
A Bottom-up Parse in Detail (4)

int * int + int

int * T + int

T + int

T + T



A Bottom-up Parse in Detail (5)

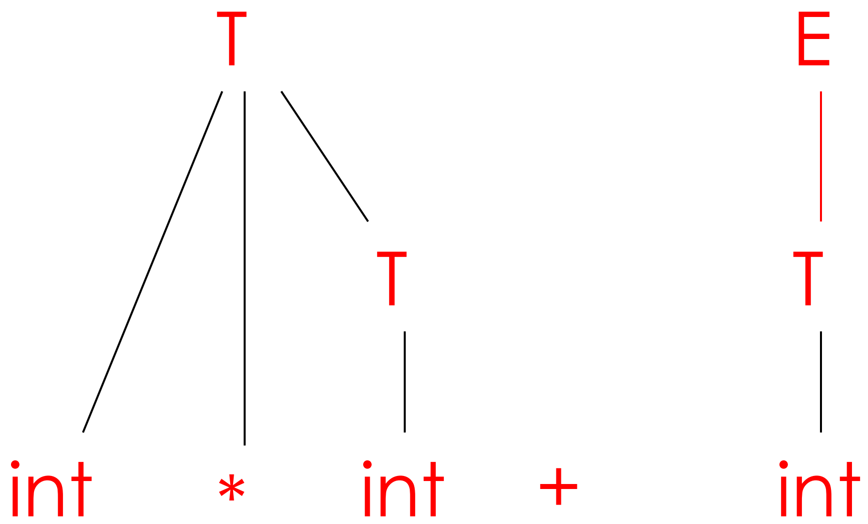
int * int + int

int * T + int

T + int

T + T

T + E



A Bottom-up Parse in Detail (6)

int * int + int

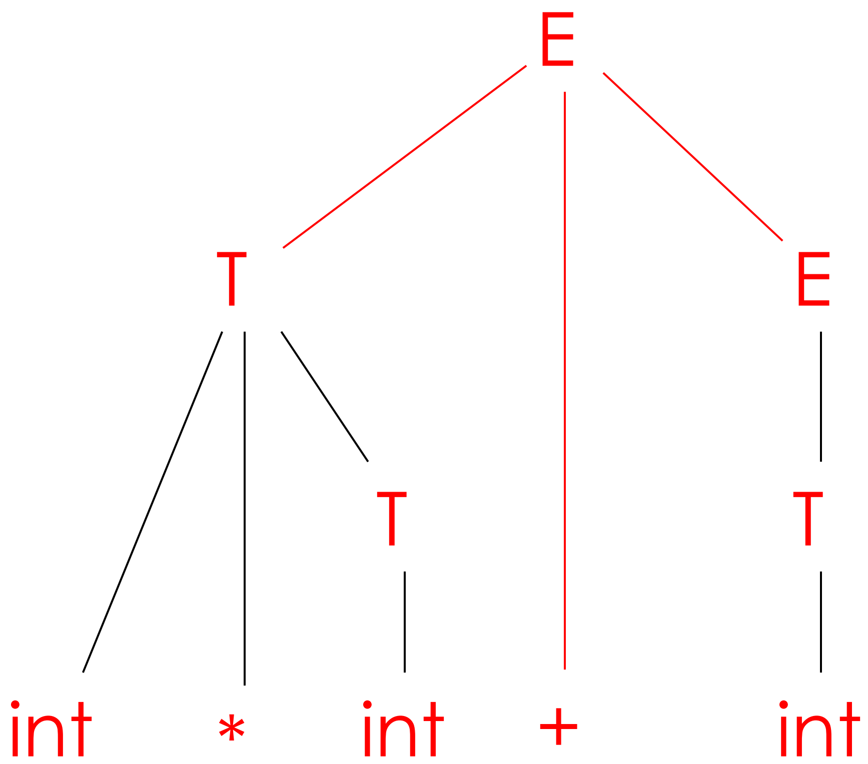
int * T + int

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 has terminals and non-terminals
- The dividing point is marked by a I
  - The I is not part of the string
- Initially, all input is unexamined $\mathrm{I}x_1 x_2 \dots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Shift

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

ABCIxyz $\Rightarrow$ ABCxIyz

Reduce

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

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

The Example with Reductions Only

int * int | + int	reduce T $\rightarrow$ int
int * T | + int	reduce T $\rightarrow$ int * T
T + int |	reduce T $\rightarrow$ int
T + T |	reduce E $\rightarrow$ T
T + E |	reduce E $\rightarrow$ T + E
E |	

The Example with Shift-Reduce Parsing

$\mid$ int * int + int	shift
int $\mid$ * int + int	shift
int * $\mid$ int + int	shift
int * int $\mid$ + int	reduce T $\rightarrow$ int
int * T $\mid$ + int	reduce T $\rightarrow$ int * T
T $\mid$ + int	shift
T + $\mid$ int	shift
T + int $\mid$	reduce T $\rightarrow$ int
T + T $\mid$	reduce E $\rightarrow$ T
T + E $\mid$	reduce E $\rightarrow$ T + E
E $\mid$	

A Shift-Reduce Parse in Detail (1)

| int * int + int

int * int + int

↑

A Shift-Reduce Parse in Detail (2)

I int * int + int

int I * int + int

int * int + int

↑

A Shift-Reduce Parse in Detail (3)

| int * int + int

int | * int + int

int * | int + int

int * int + int

A Shift-Reduce Parse in Detail (4)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * int + int

A Shift-Reduce Parse in Detail (5)

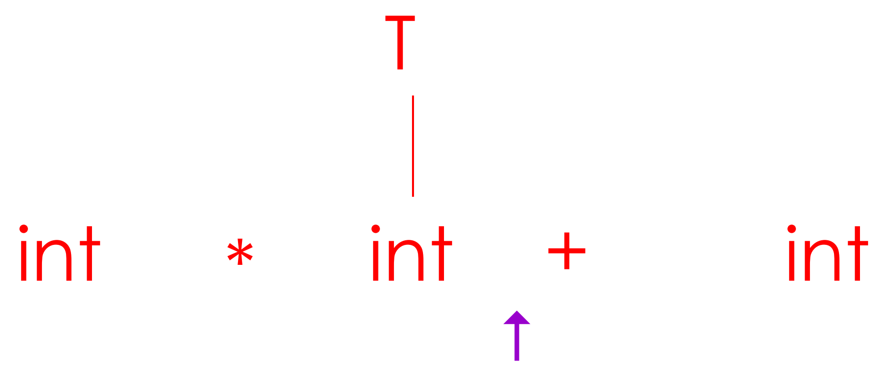
| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int



A Shift-Reduce Parse in Detail (6)

| int * int + int

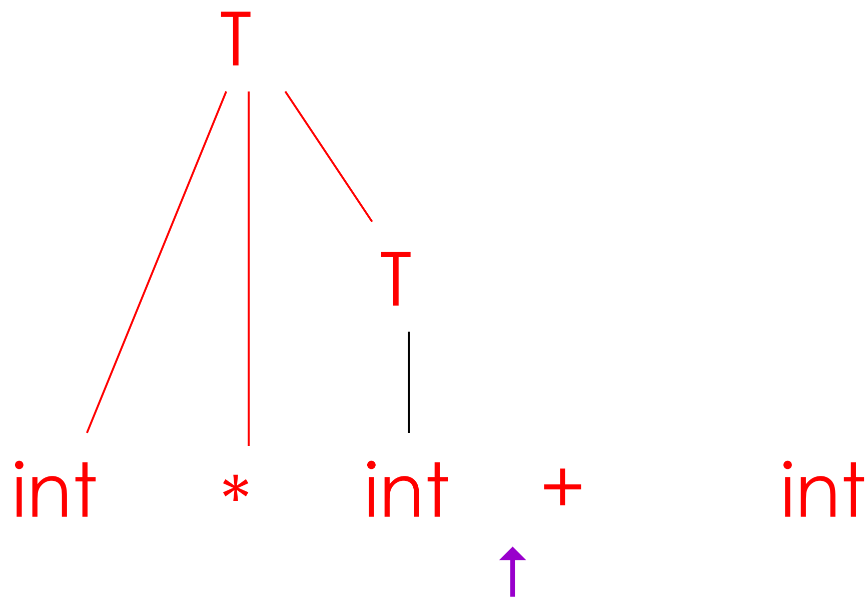
int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int



A Shift-Reduce Parse in Detail (7)

| int * int + int

int | * int + int

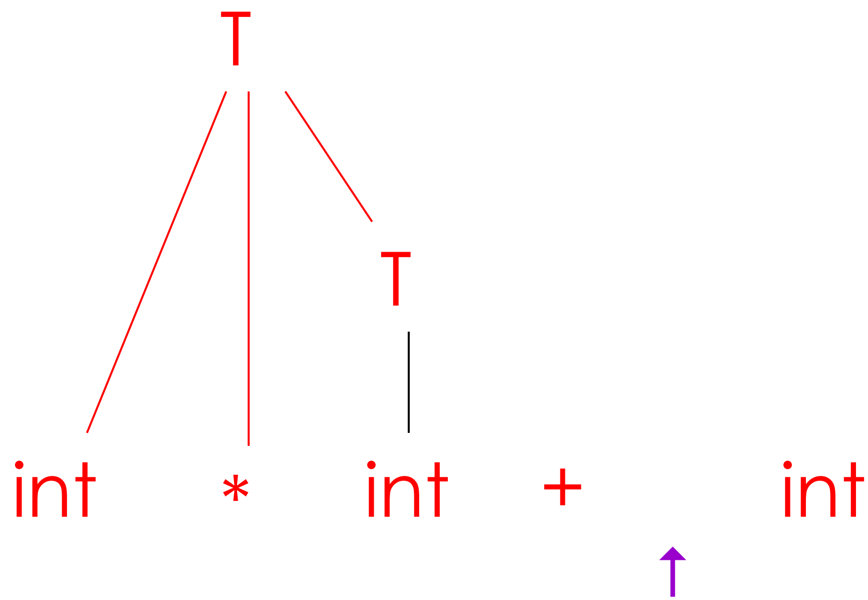
int * | int + int

int * int | + int

int * T | + int

T | + int

T + | int



A Shift-Reduce Parse in Detail (8)

| int * int + int

int | * int + int

int * | int + int

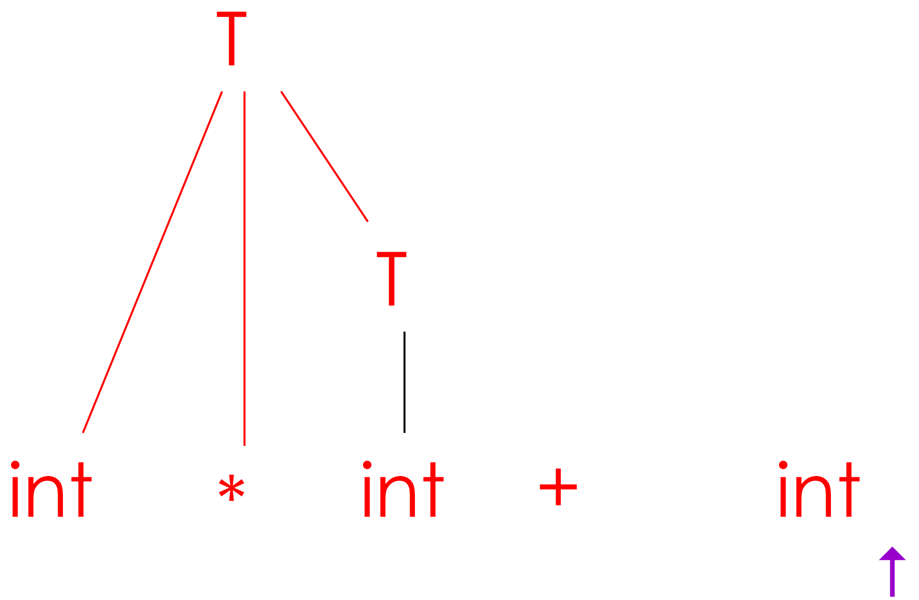
int * int | + int

int * T | + int

T | + int

T + | int

T + int |



A Shift-Reduce Parse in Detail (9)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

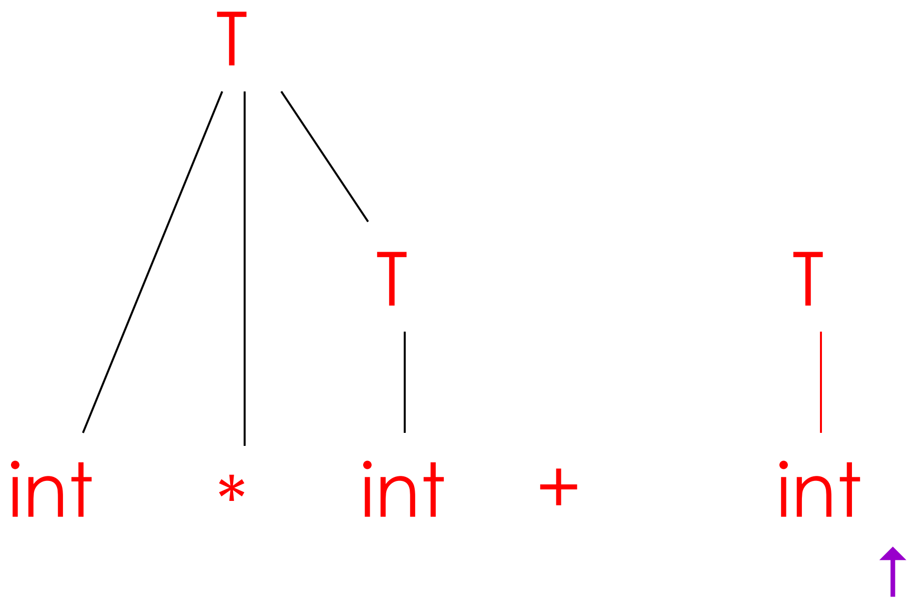
int * T | + int

T | + int

T + | int

T + int |

T + T |



A Shift-Reduce Parse in Detail (10)

I int * int + int

int I * int + int

int * I int + int

int * int I + int

int * T I + int

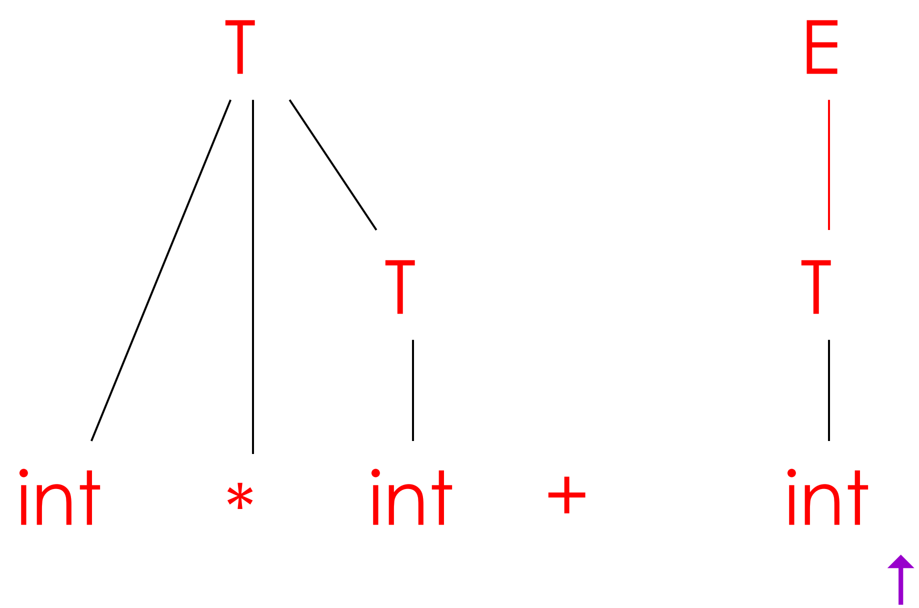
T I + int

T + I int

T + int I

T + T I

T + E I



A Shift-Reduce Parse in Detail (11)

I int * int + int

int I * int + int

int * I int + int

int * int I + int

int * T I + int

T I + int

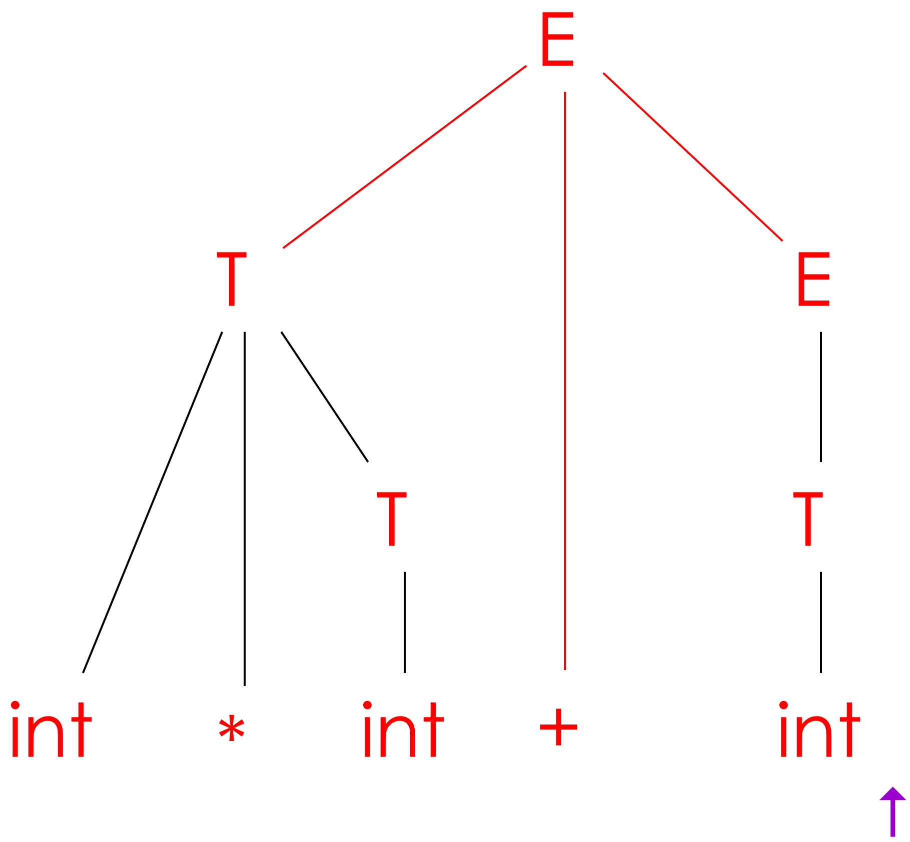
T + I int

T + int I

T + T I

T + E I

E I



The Stack

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

- 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

- You will see such conflicts in your project!
  – More next time . . .

Key Issue

- How do we decide when to shift or reduce?

- Example grammar:

  $E \rightarrow T + E \mid T$

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

- Consider step int | * int + int
  - We could reduce by $T \rightarrow int$ 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

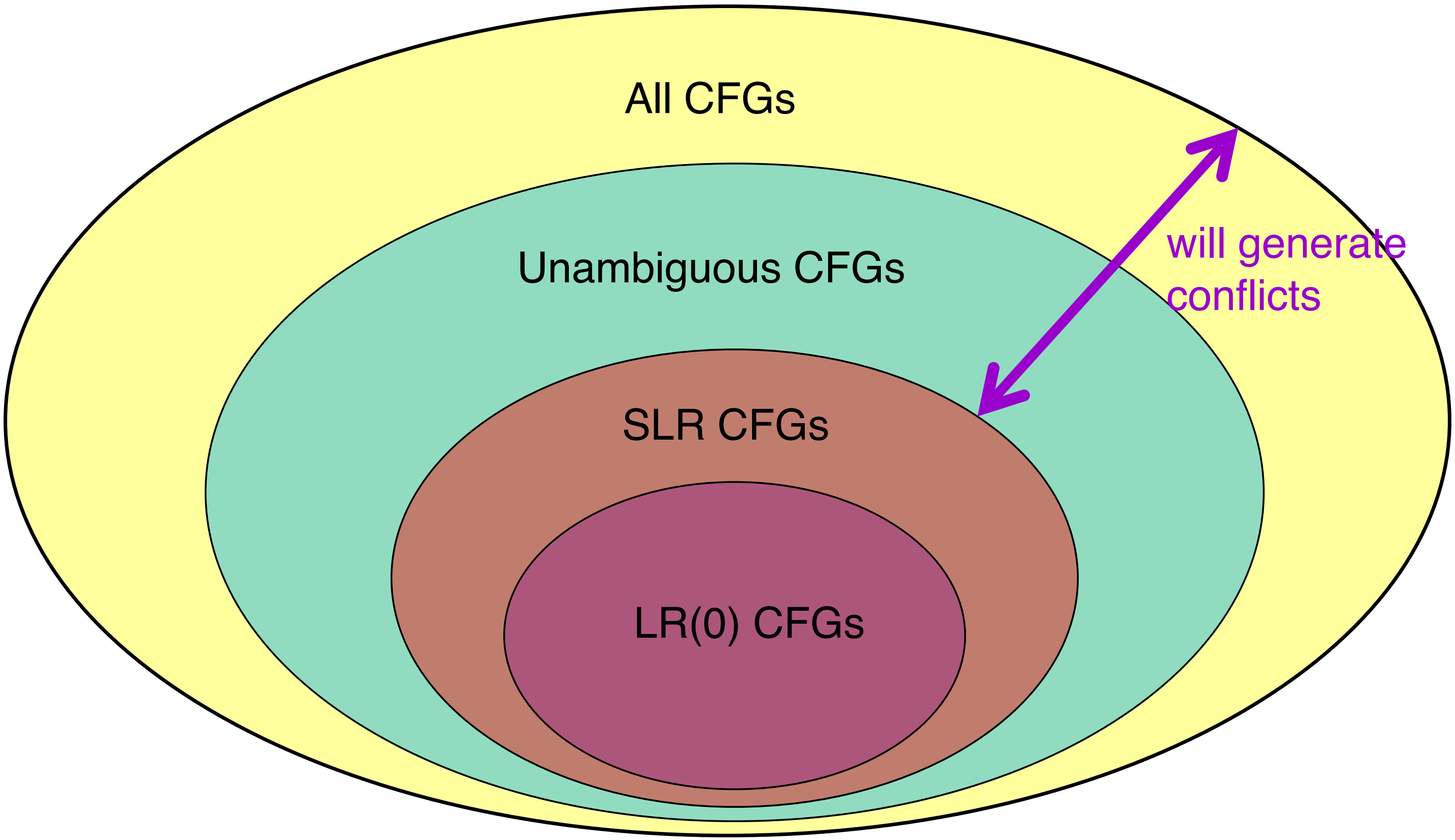
Summary of Handles

- In shift-reduce parsing, handles always appear at the top of the stack
- Handles are never to the left of the rightmost non-terminal
  - Therefore, shift-reduce moves are sufficient; the I 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


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

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 SLR 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

  $T \rightarrow .(E)$
  $T \rightarrow (.E)$
  $T \rightarrow (E.)$
  $T \rightarrow (E).$

Items (Cont.)

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

- 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 $X_i \to \alpha_i$
  - 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 $X_{i-1} \to \text{Prefix}_{i-1}\ X_i\ \beta$ for some β

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

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

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 $E \rightarrow T$
"(" is a prefix of the rhs of $T \rightarrow (E)$
"ε" is a prefix of the rhs of $E \rightarrow T$
"int *" is a prefix of the rhs of $T \rightarrow int * T$

An Example (Cont.)

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

The stack of items

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

$E \rightarrow .T$

$T \rightarrow (.E)$

Says

We've seen 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

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

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

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

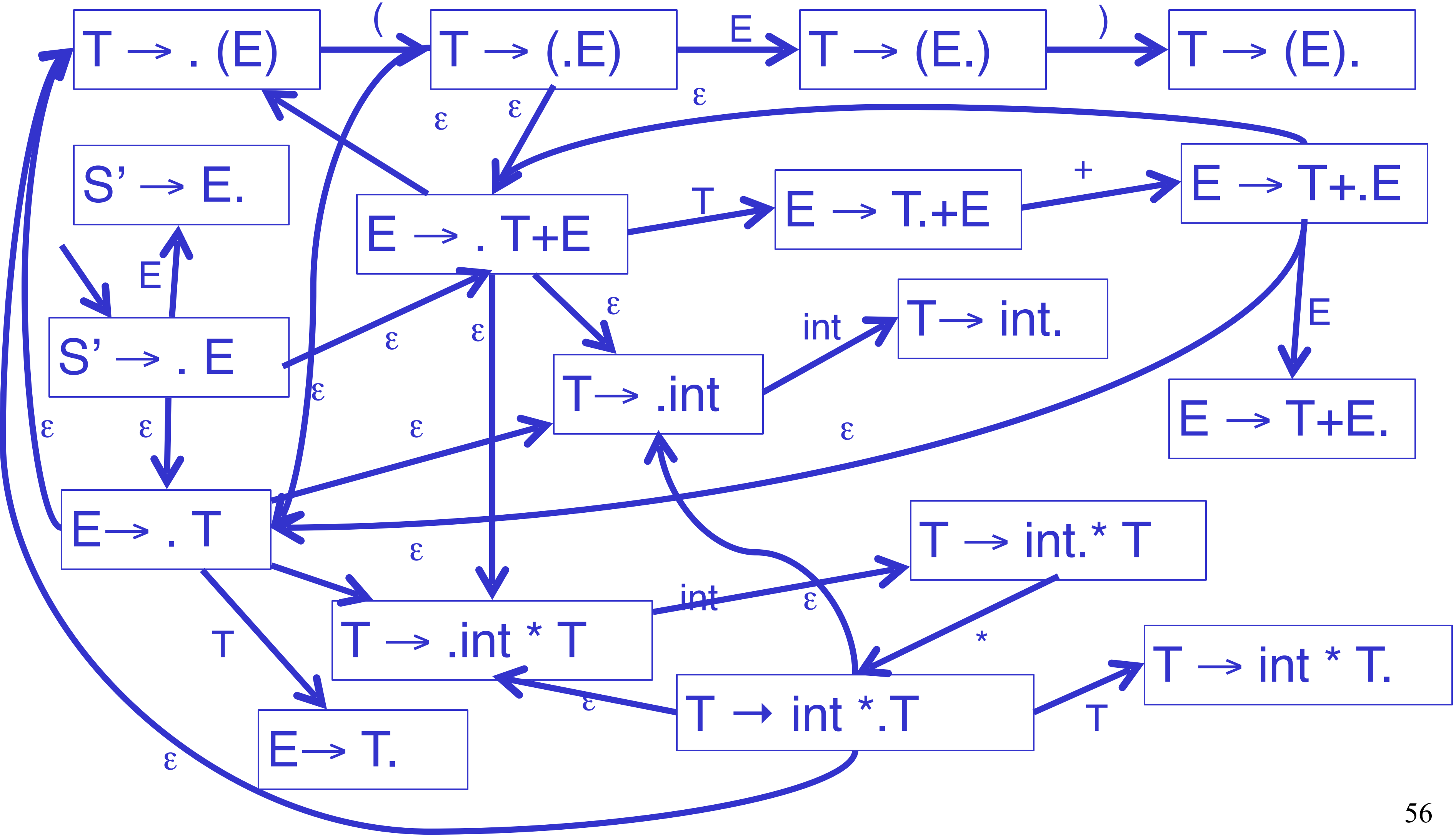
An NFA Recognizing Viable Prefixes (Cont.)

5. Every state is an accepting state

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

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

NFA for Viable Prefixes



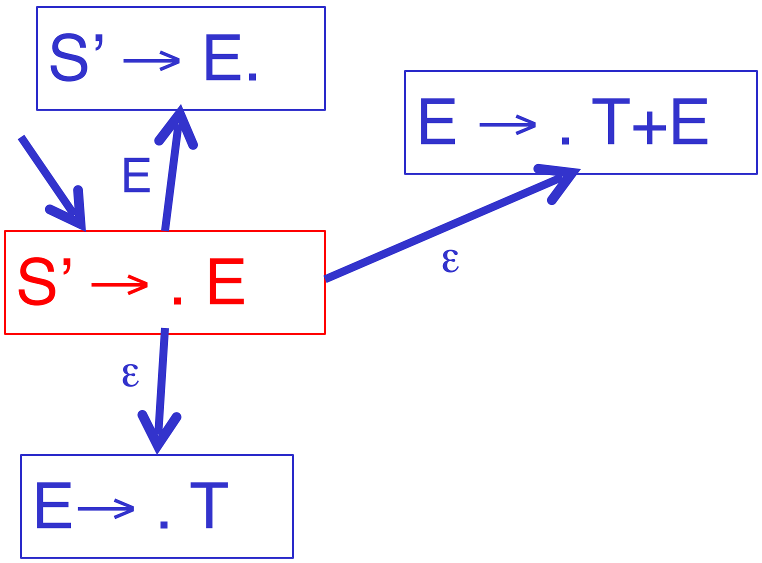
$E \rightarrow T + E \mid T$
$T \rightarrow int * T \mid int \mid (E)$

NFA for Viable Prefixes

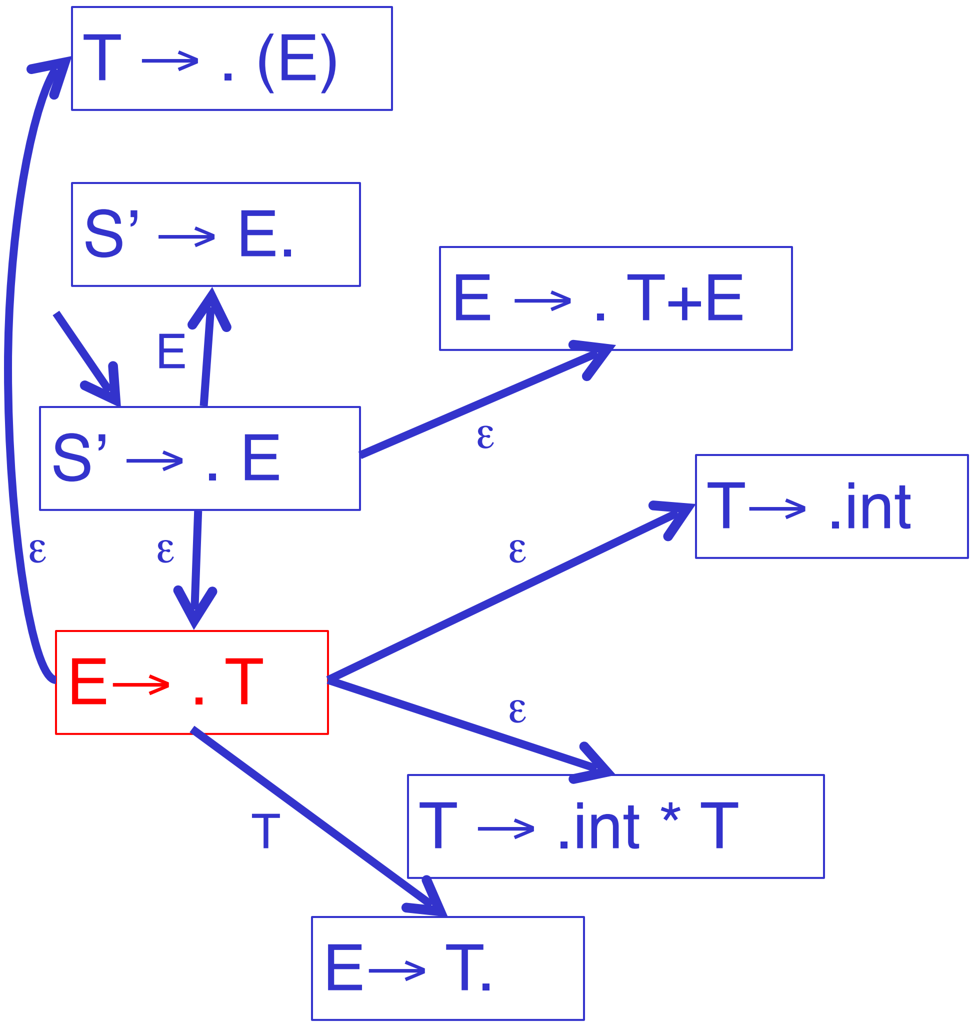
$\boxed{S' \rightarrow .\,E}$

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

NFA for Viable Prefixes

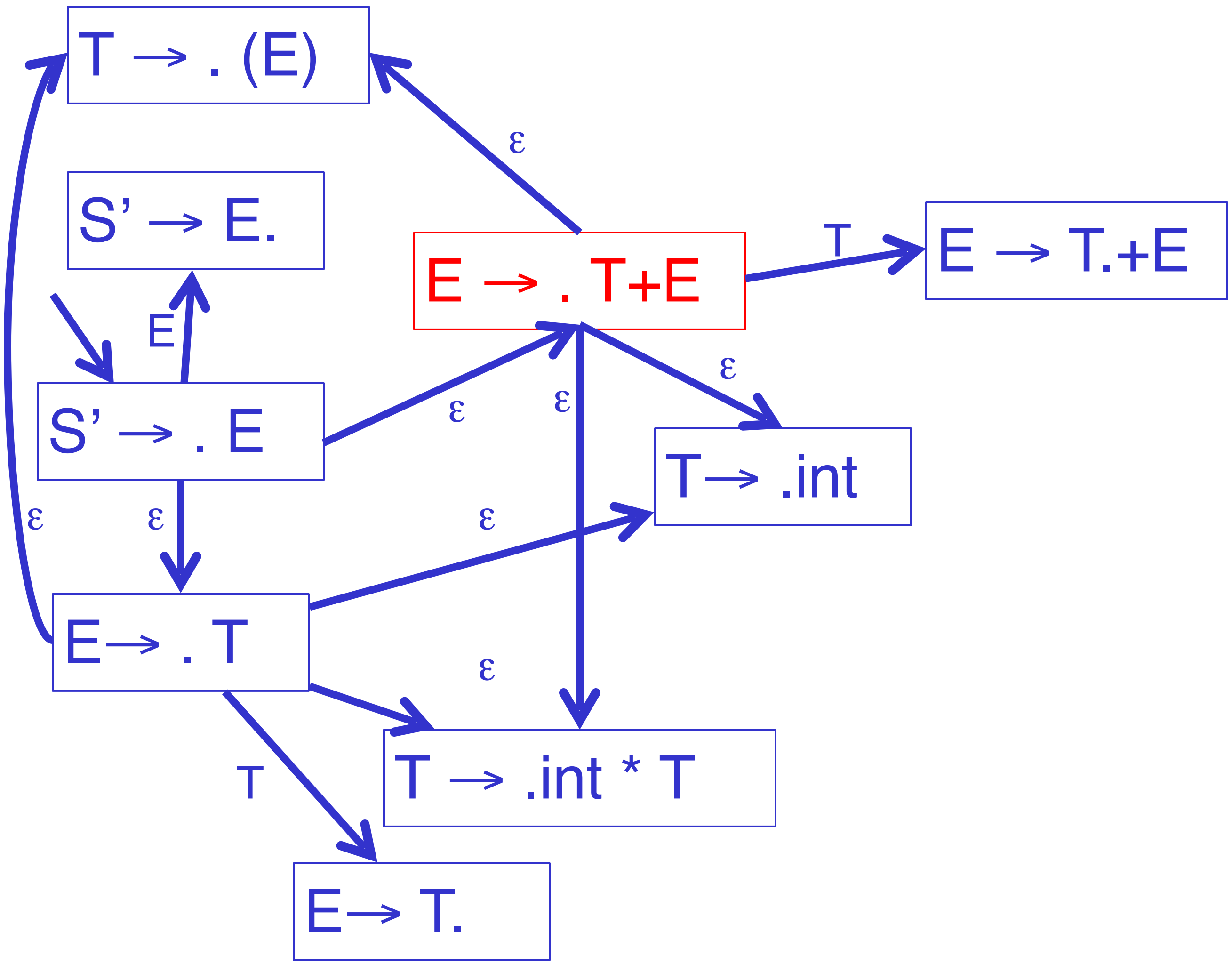


NFA for Viable Prefixes



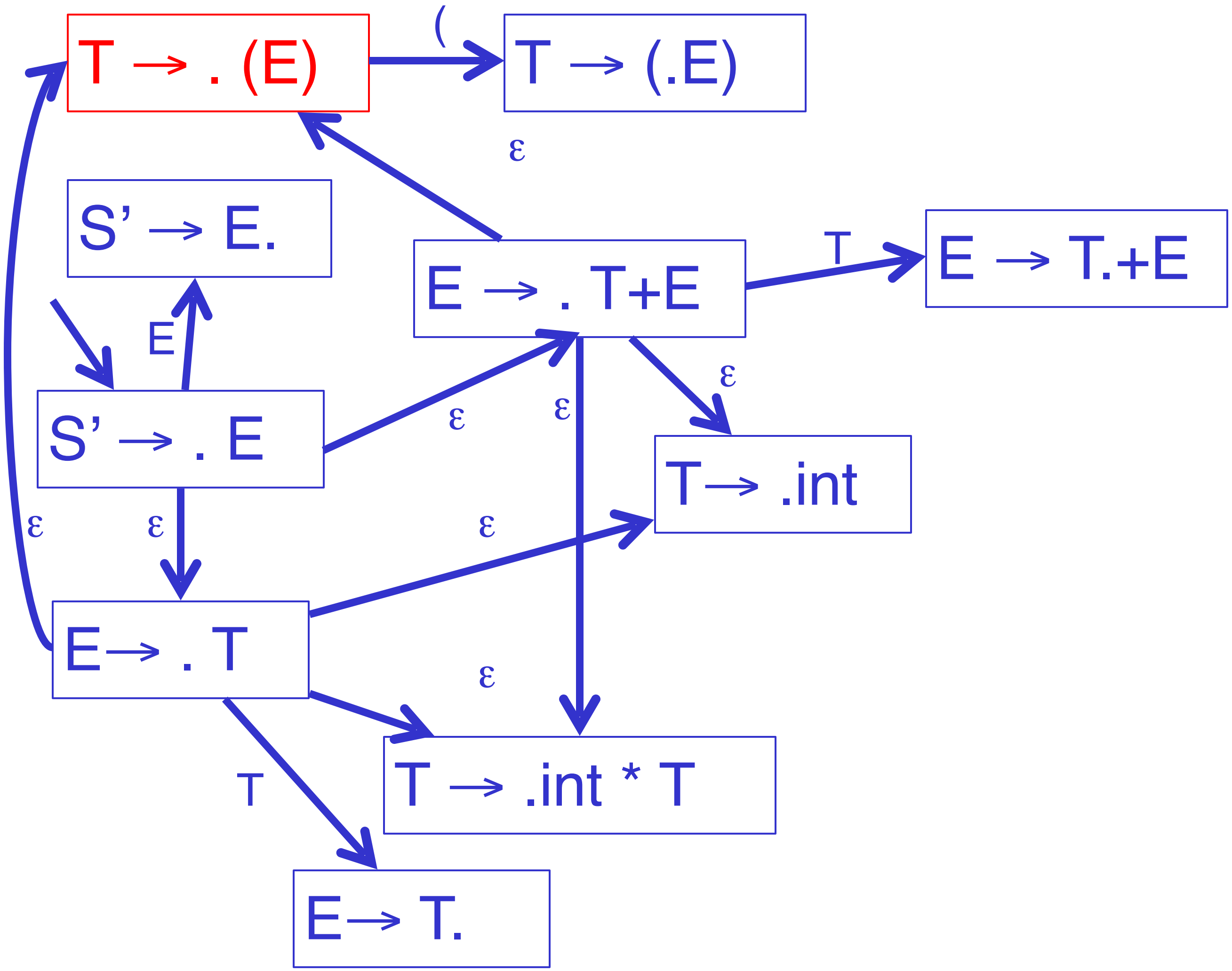
$E \to T + E \mid T$
$T \to int * T \mid int \mid (E)$

NFA for Viable Prefixes



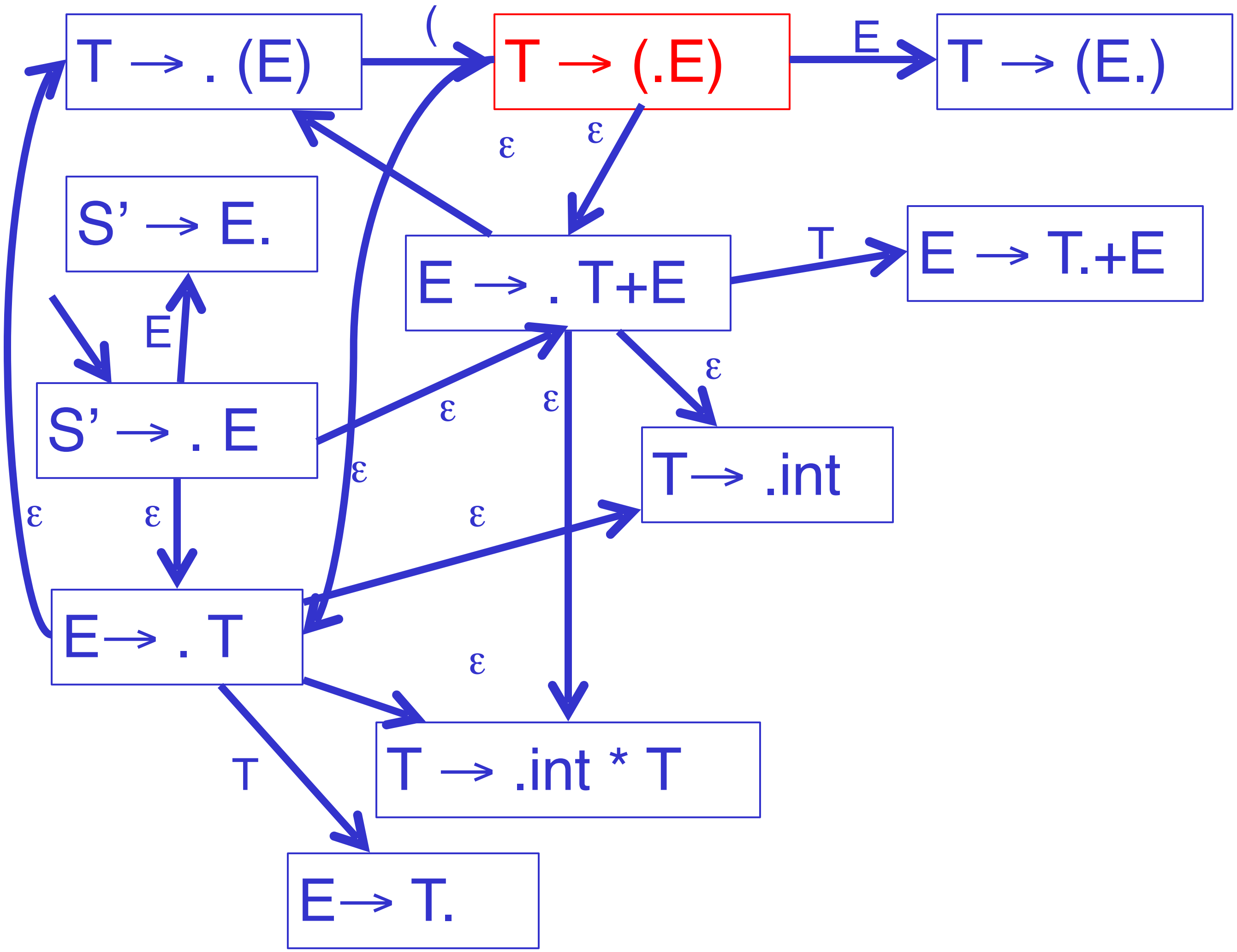
$E \rightarrow T + E \mid T$
$T \rightarrow int * T \mid int \mid (E)$

NFA for Viable Prefixes

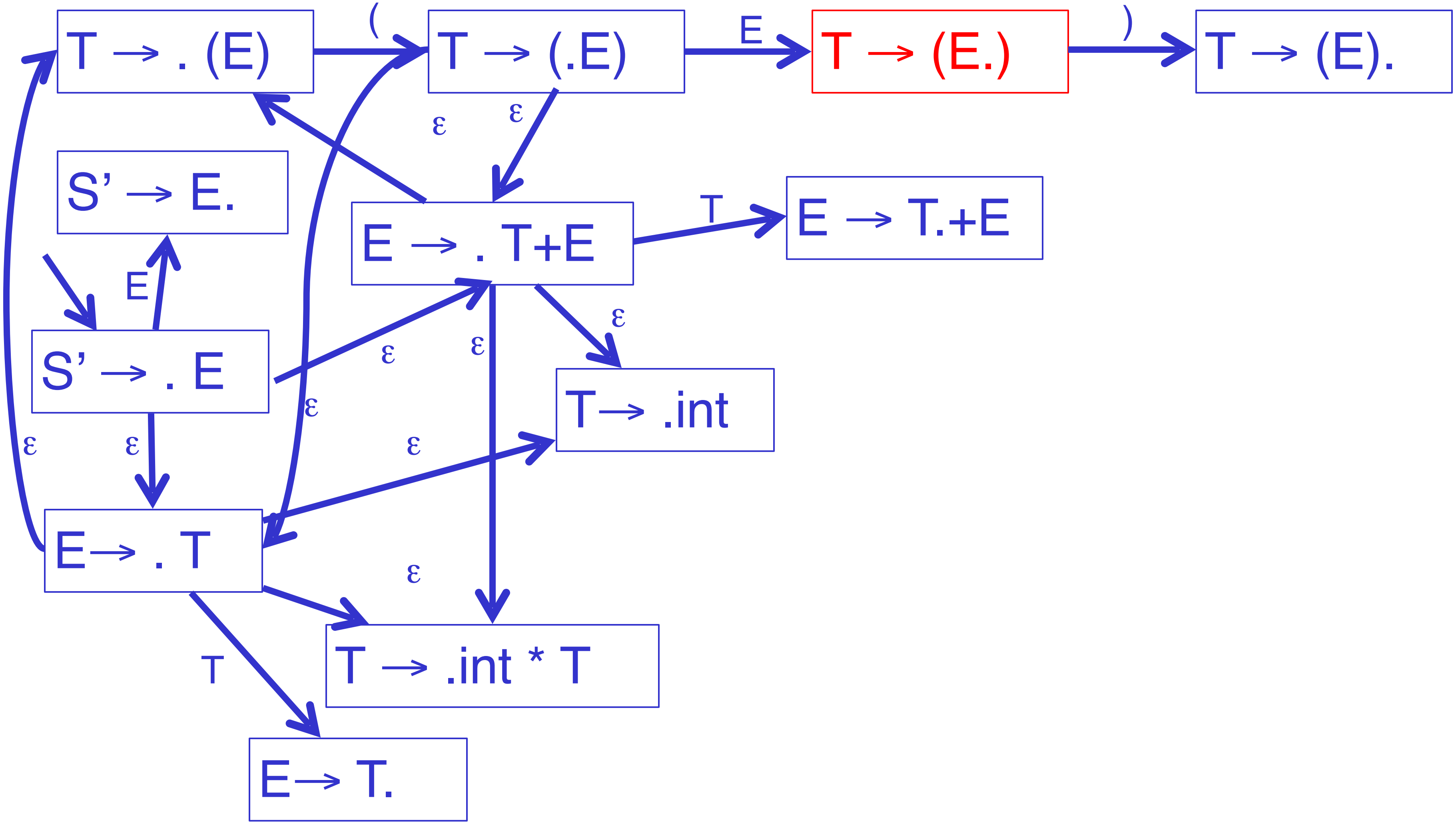


$E \rightarrow T + E \mid T$
$T \rightarrow int * T \mid int \mid (E)$

NFA for Viable Prefixes

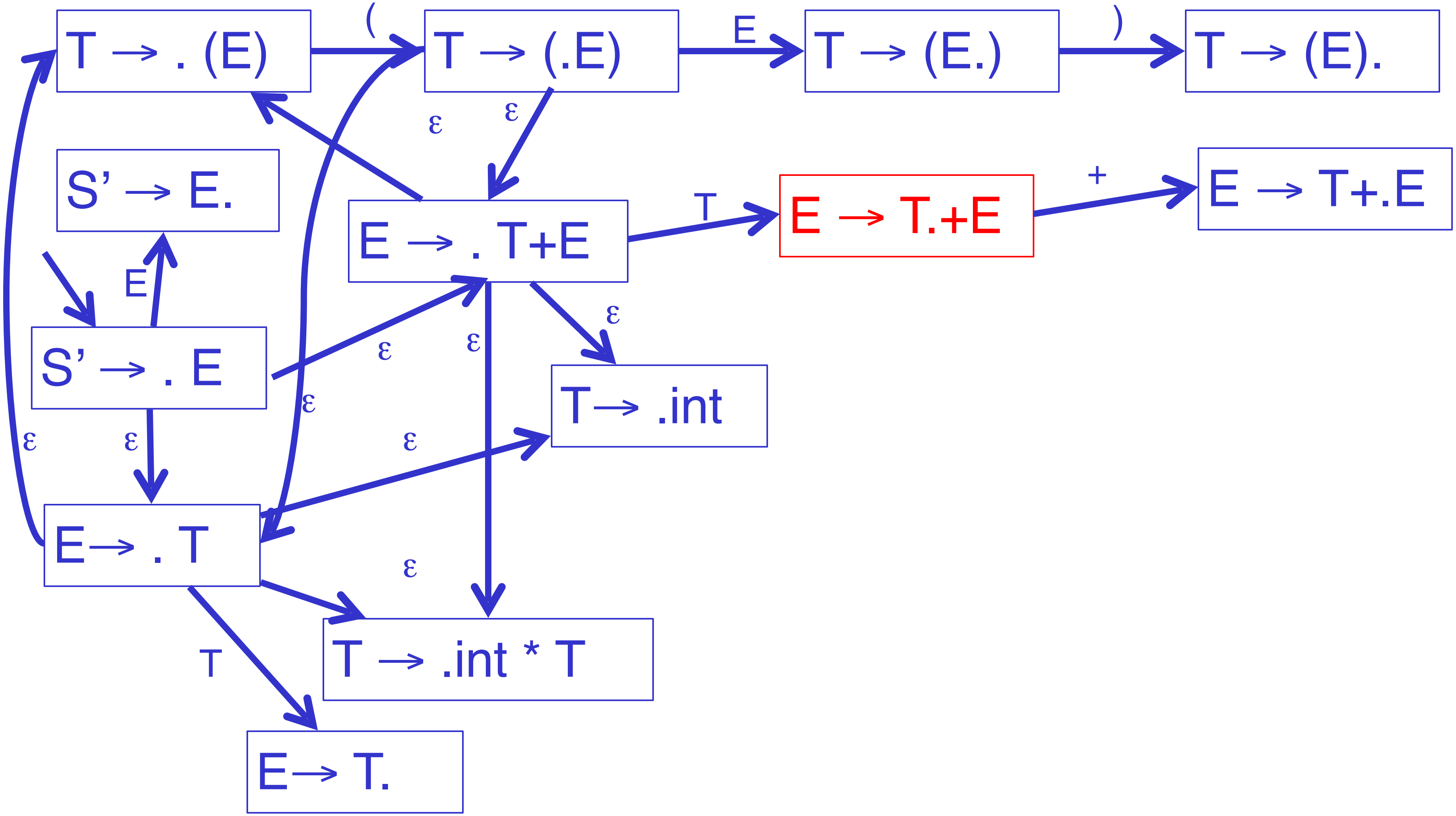


NFA for Viable Prefixes



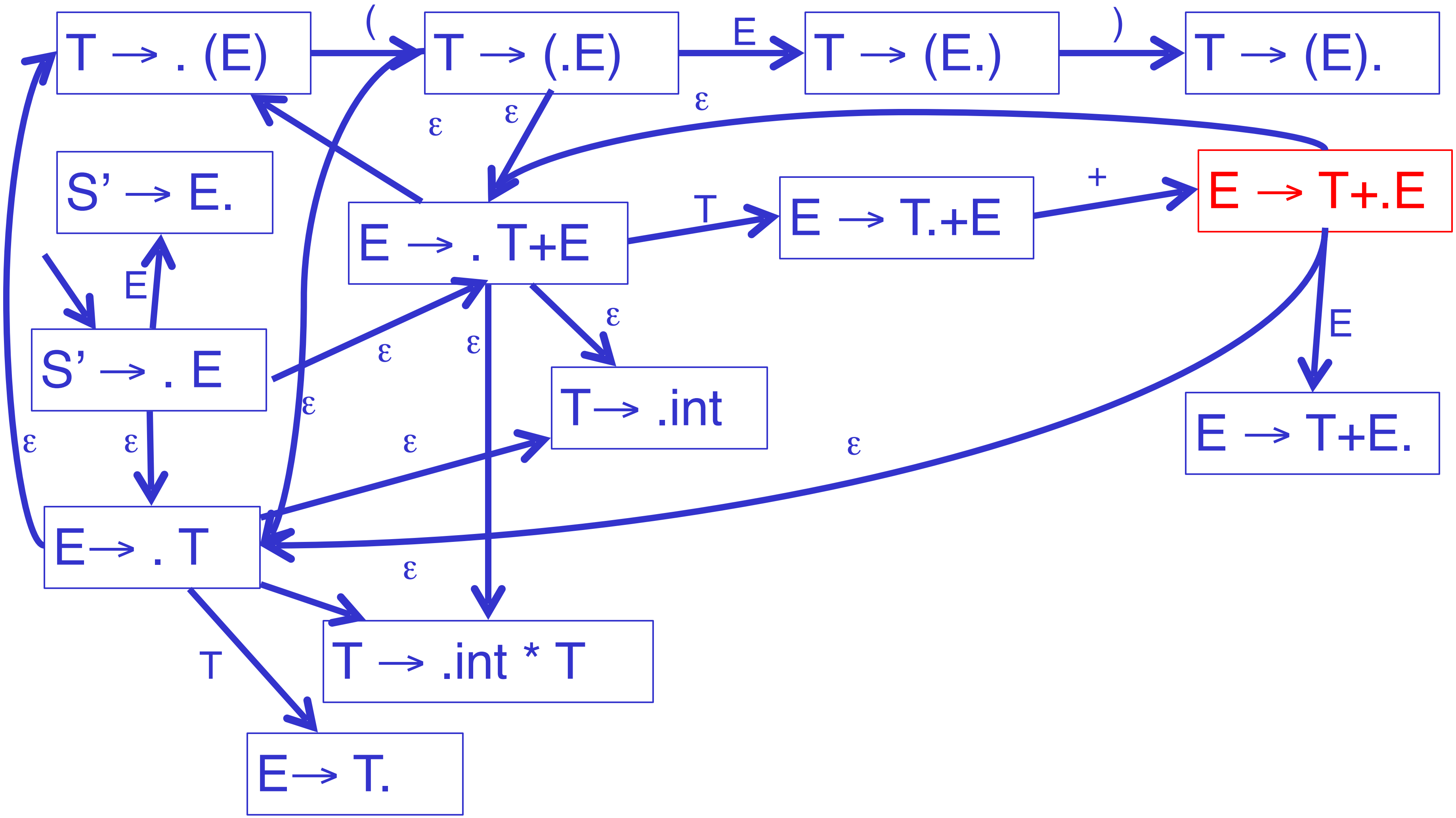
E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



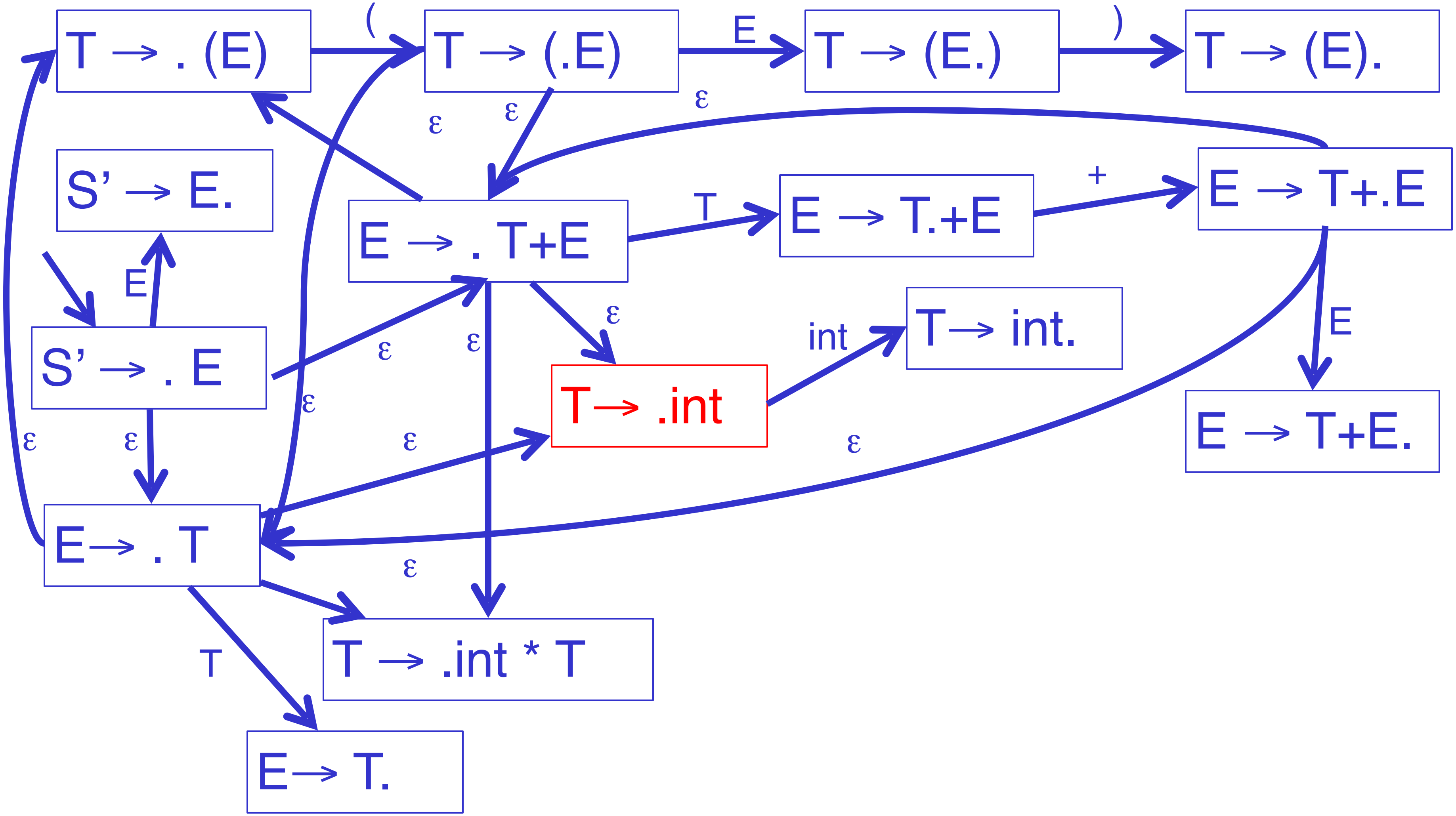
$E \rightarrow T + E \mid T$
$T \rightarrow int * T \mid int \mid (E)$

NFA for Viable Prefixes



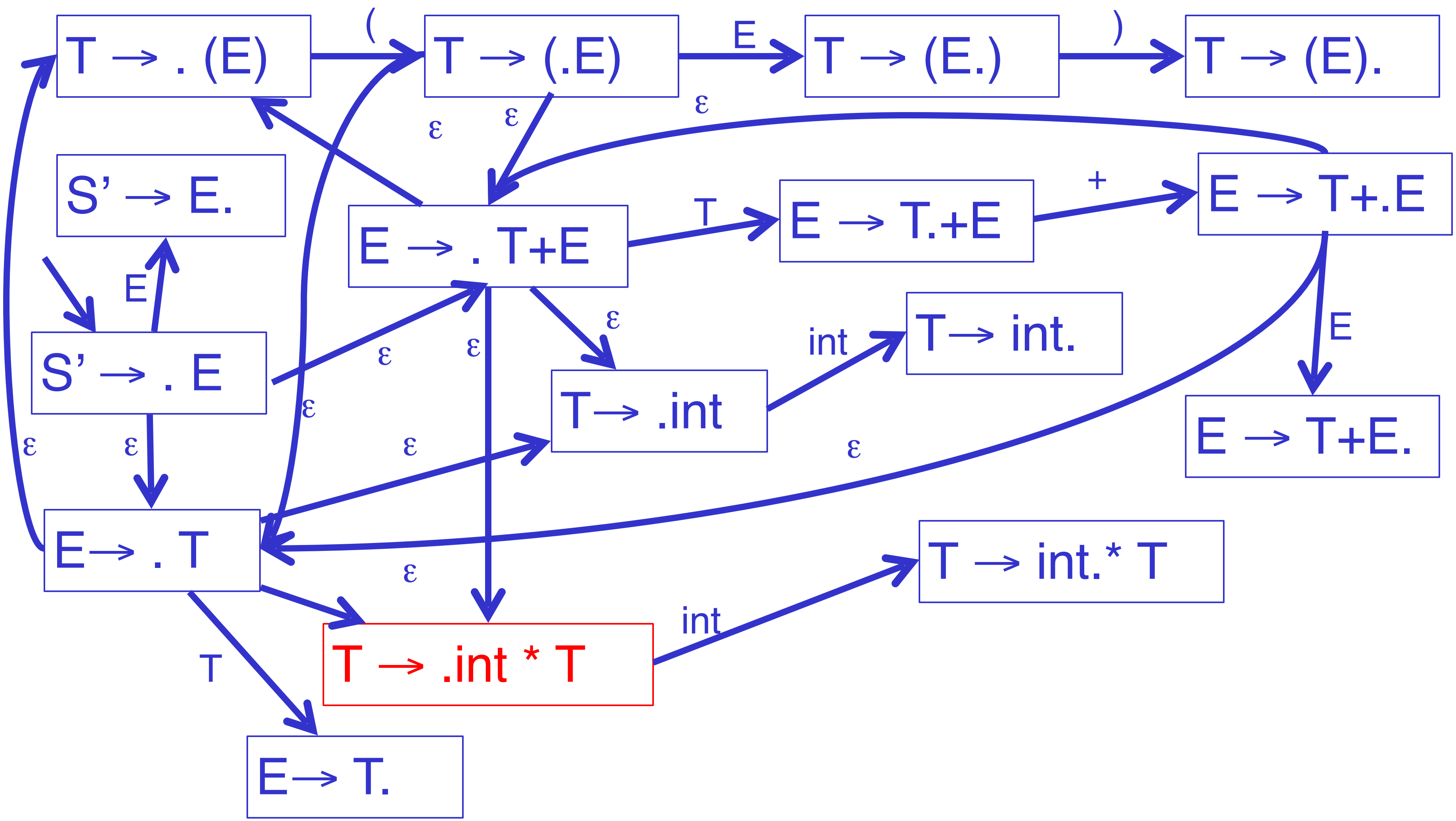
$E \rightarrow T + E \mid T$
$T \rightarrow int * T \mid int \mid (E)$

NFA for Viable Prefixes



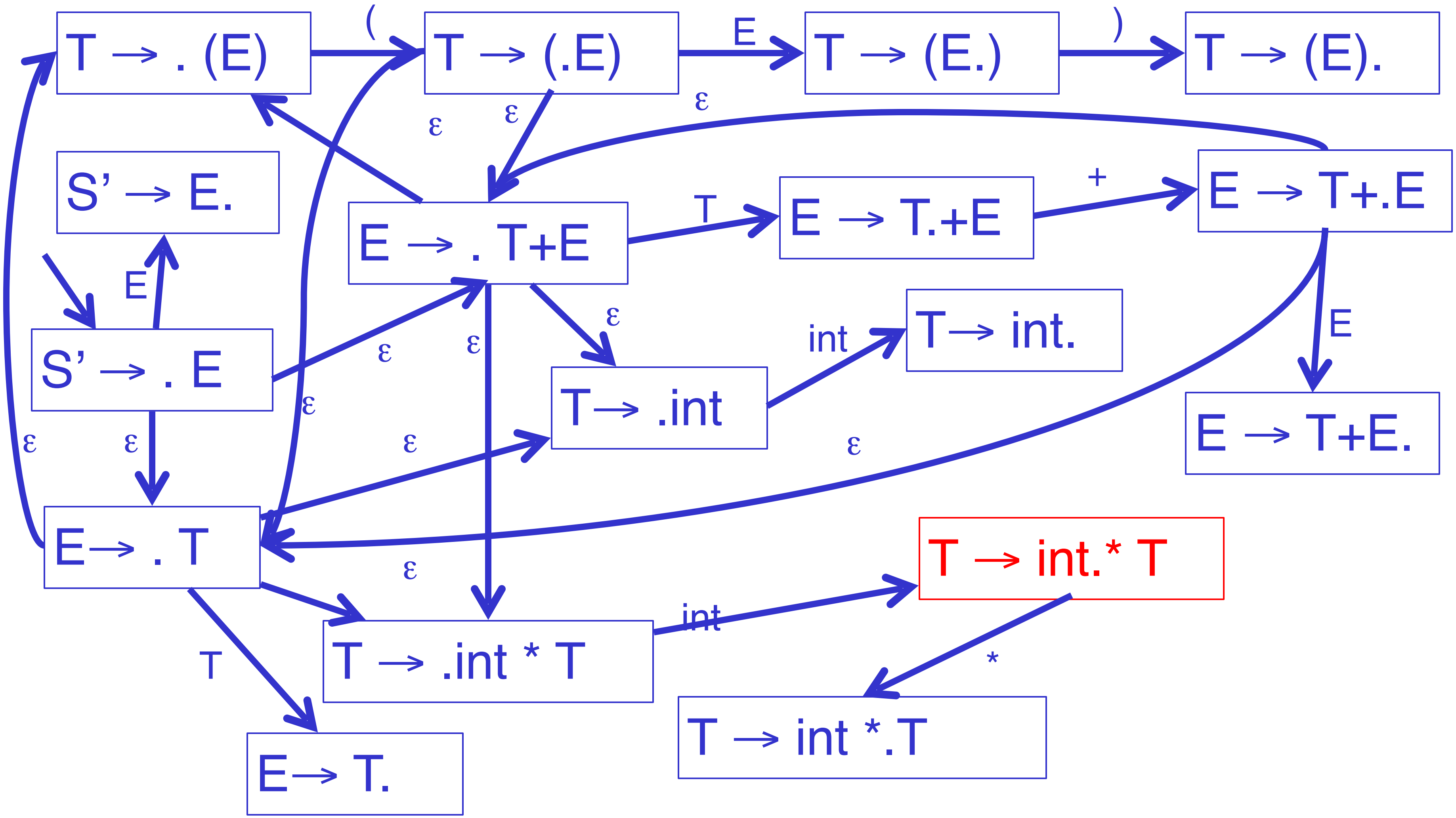
E → T + E | T
T → int * T | int | (E)

NFA for Viable Prefixes



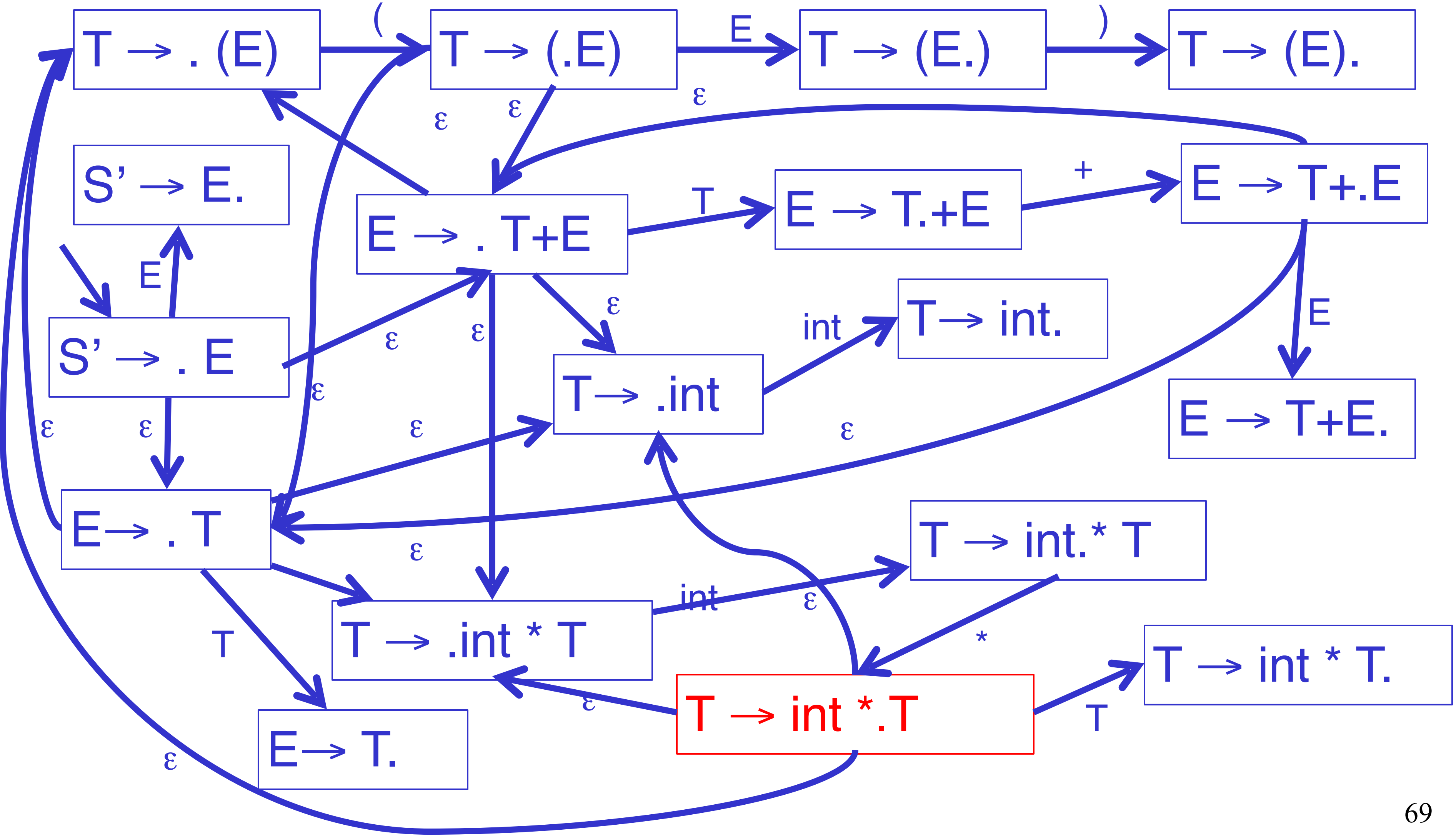
$E \to T + E \mid T$
$T \to int * T \mid int \mid (E)$

NFA for Viable Prefixes



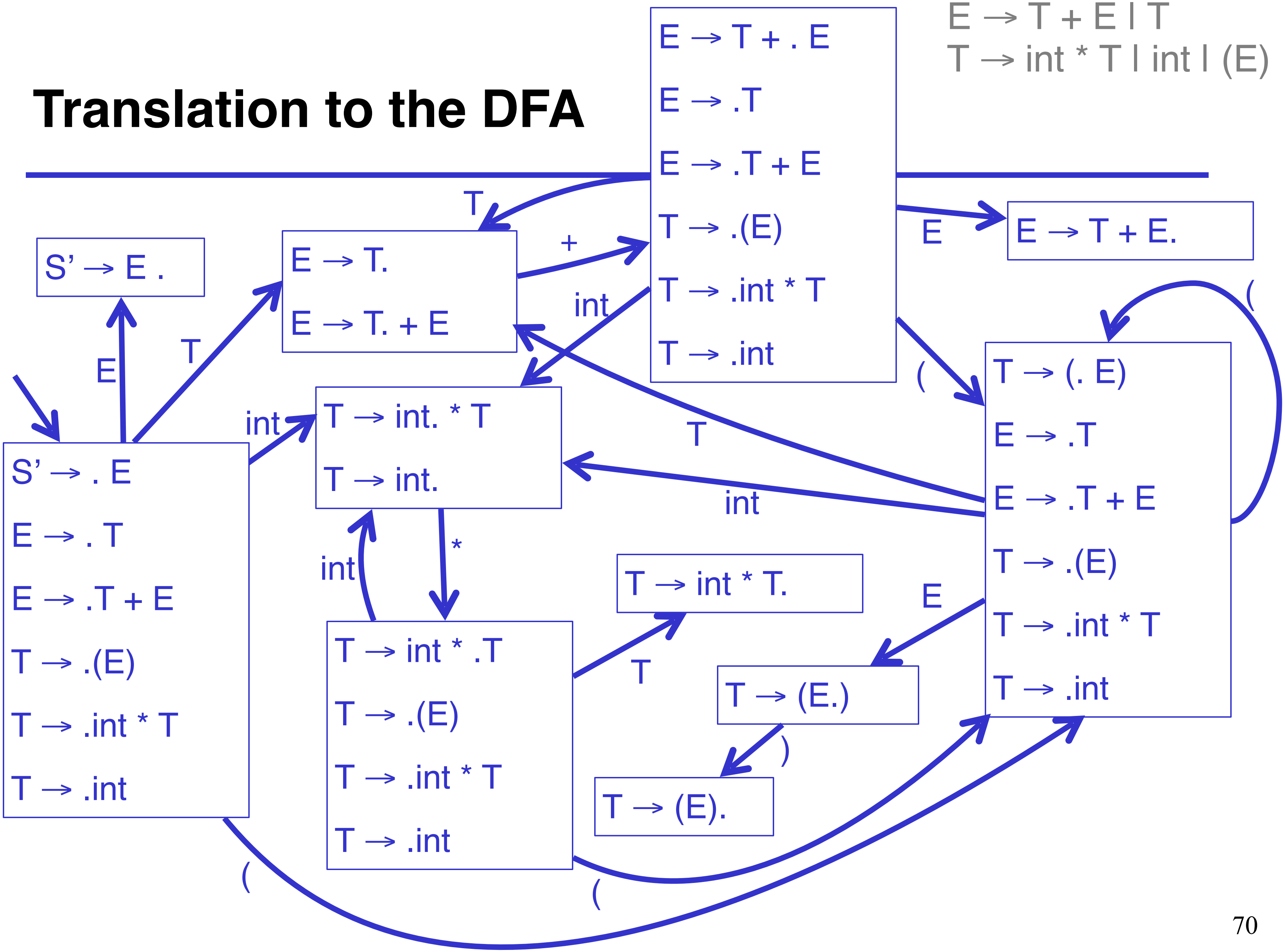
$E \rightarrow T + E \mid T$
$T \rightarrow int * T \mid int \mid (E)$

NFA for Viable Prefixes



Translation to the DFA

E → T + E | T
T → int * T | int | (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 $X \rightarrow \beta.\gamma$ 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

(

((

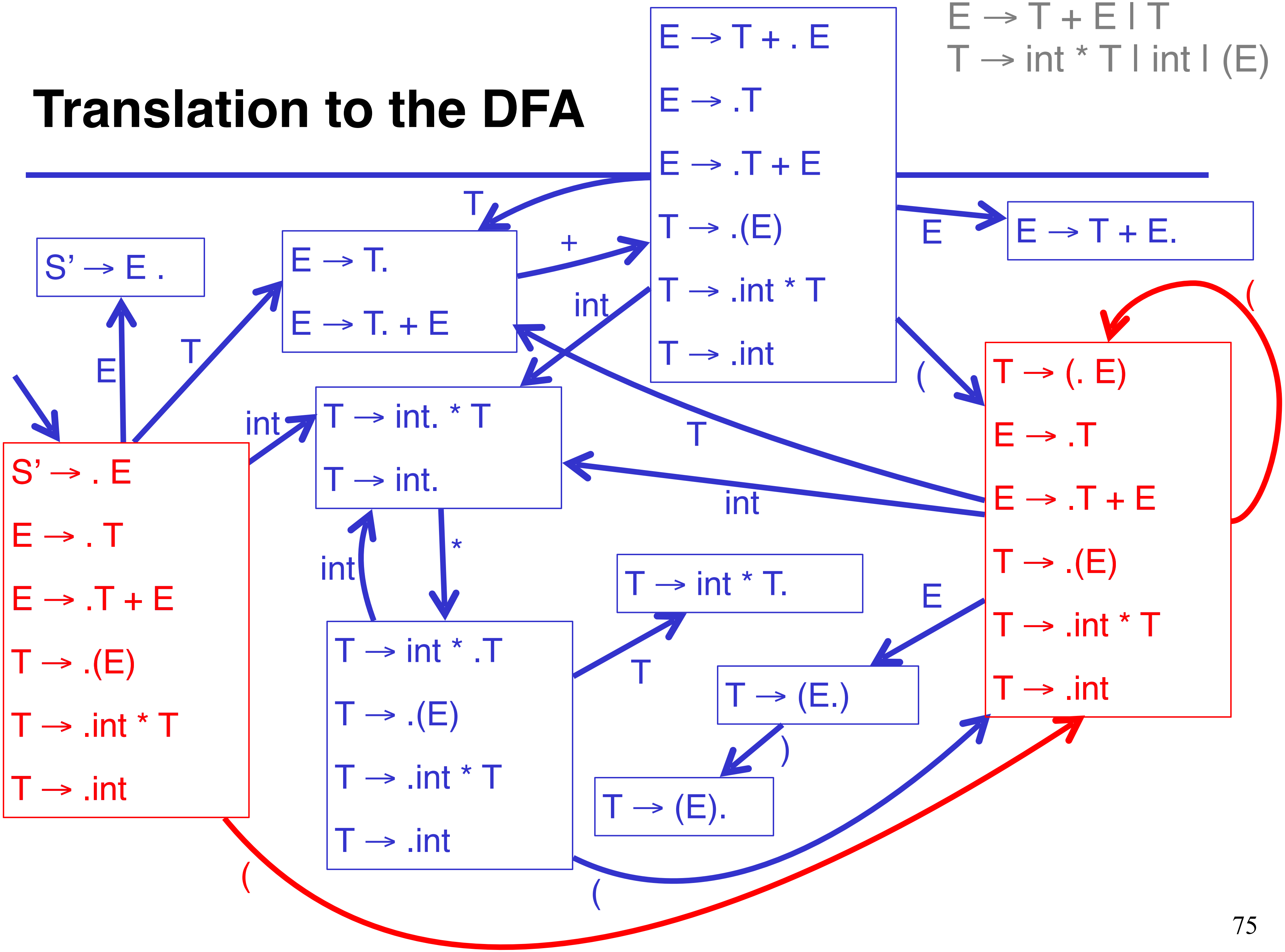
(((

((((

. . .

Translation to the DFA

E → T + E | T
T → int * T | int | (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

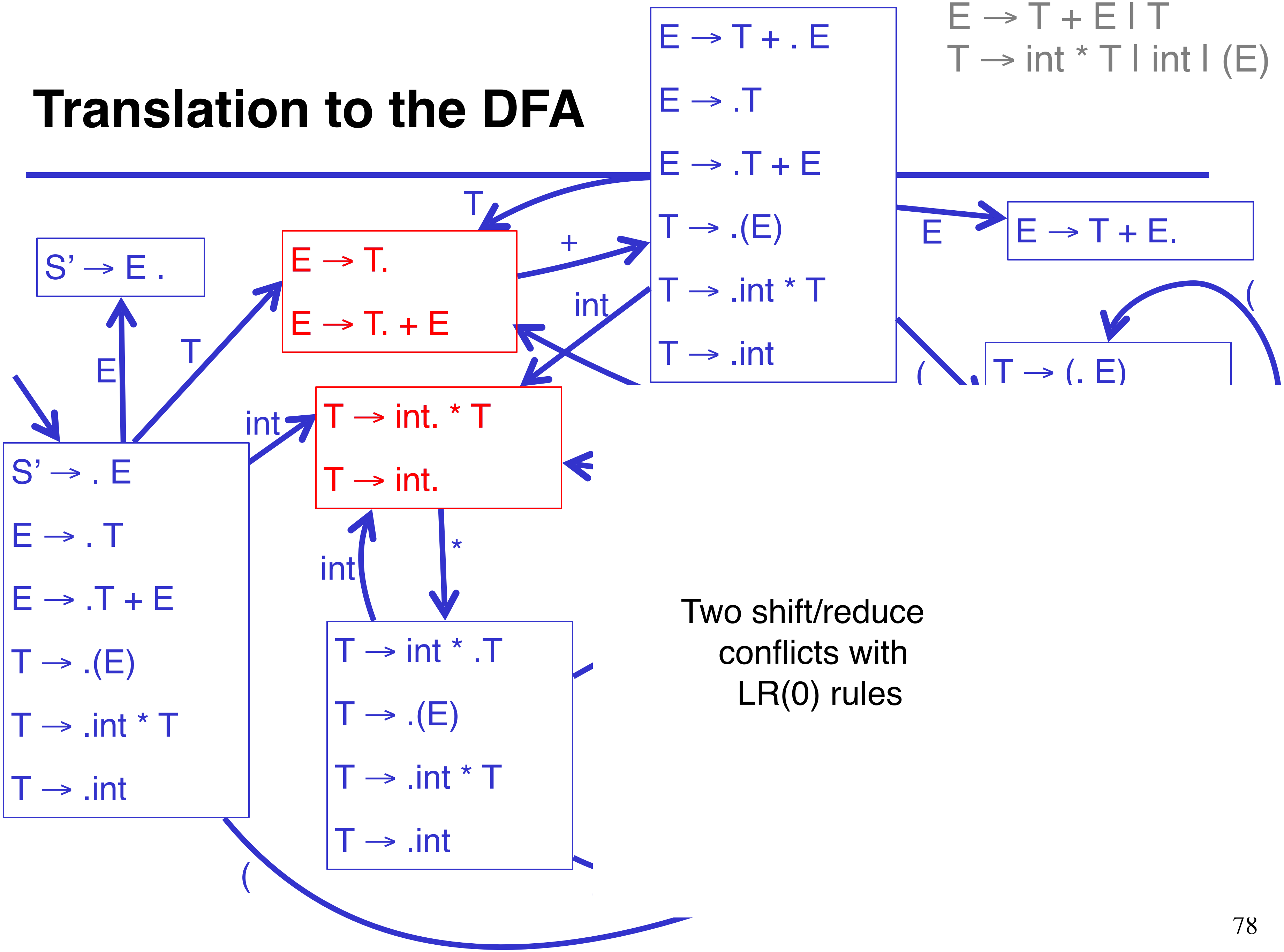
LR(0) Conflicts

- 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

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



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 $X \rightarrow \beta$ if
  - s contains item $X \rightarrow \beta.$
  - $t \in \text{Follow}(X)$
- Shift if
  - s contains item $X \rightarrow \beta.t\omega$

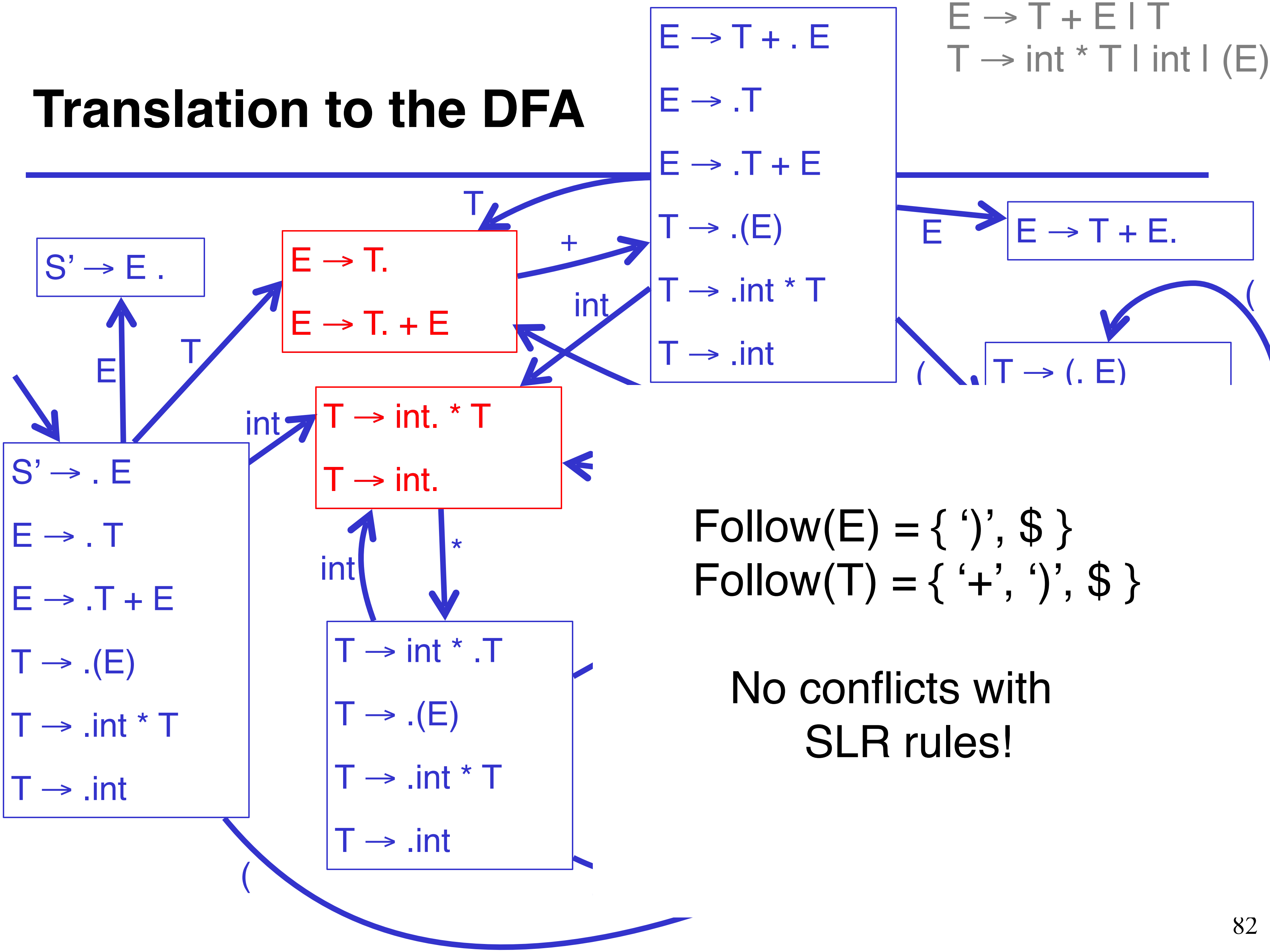
SLR Parsing (Cont.)

- 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

E → T + E | T
T → int * T | int | (E)

Translation to the DFA



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

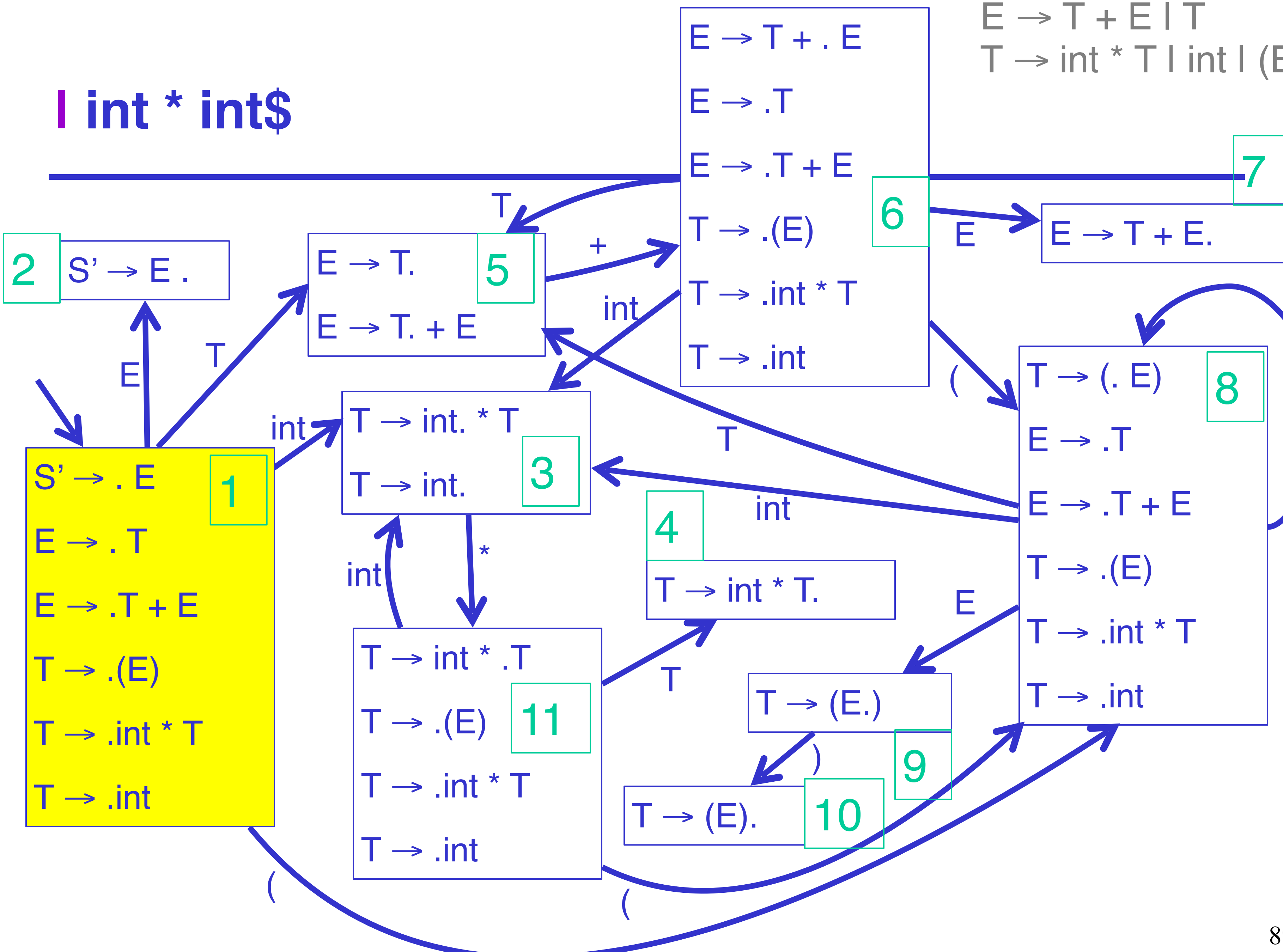
E → T + E | T
T → int * T | int | (E)

SLR Example

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

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

| int * int$



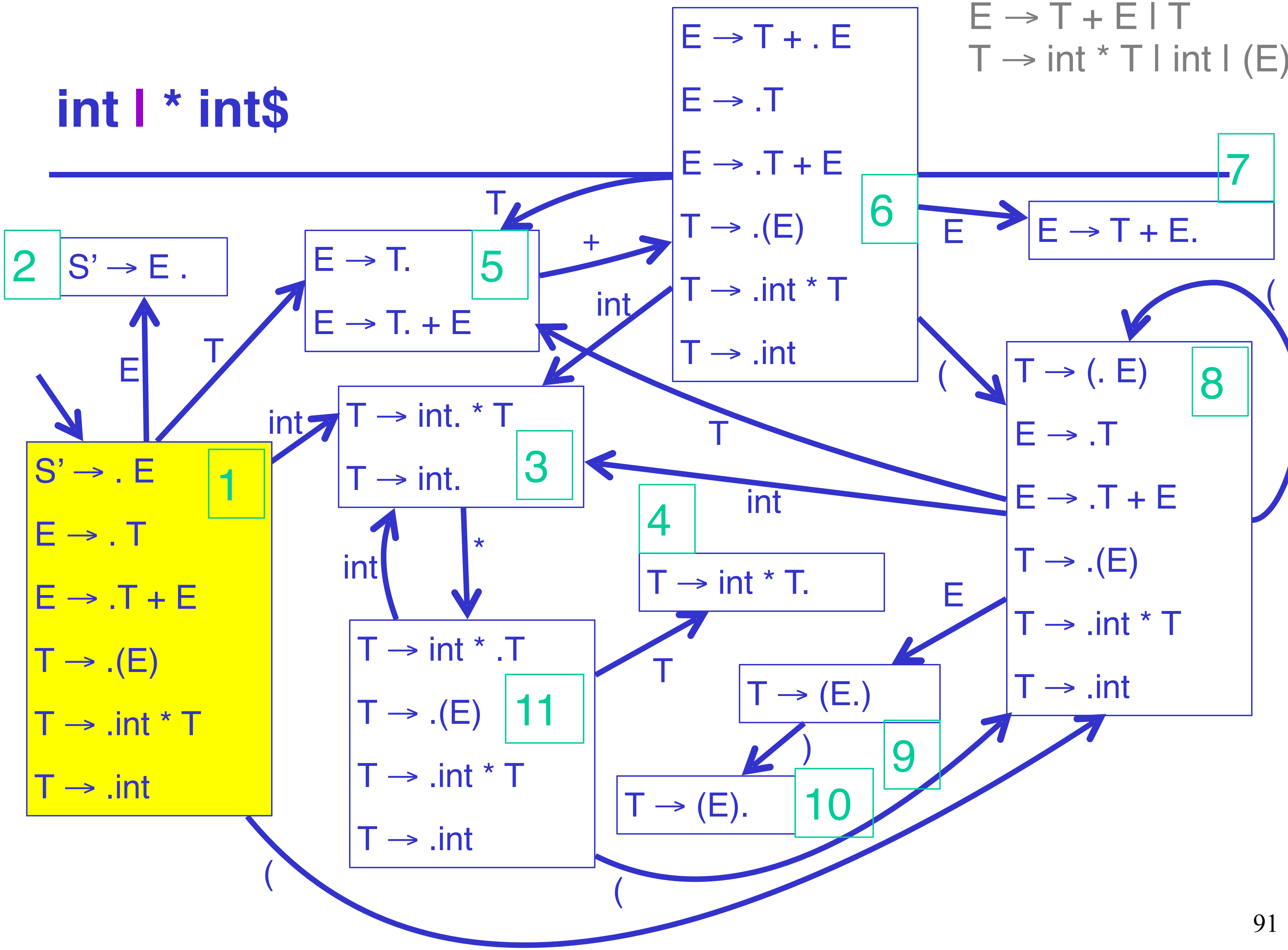
$E \to T + E \mid T$
$T \to int * T \mid int \mid (E)$

SLR Example

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

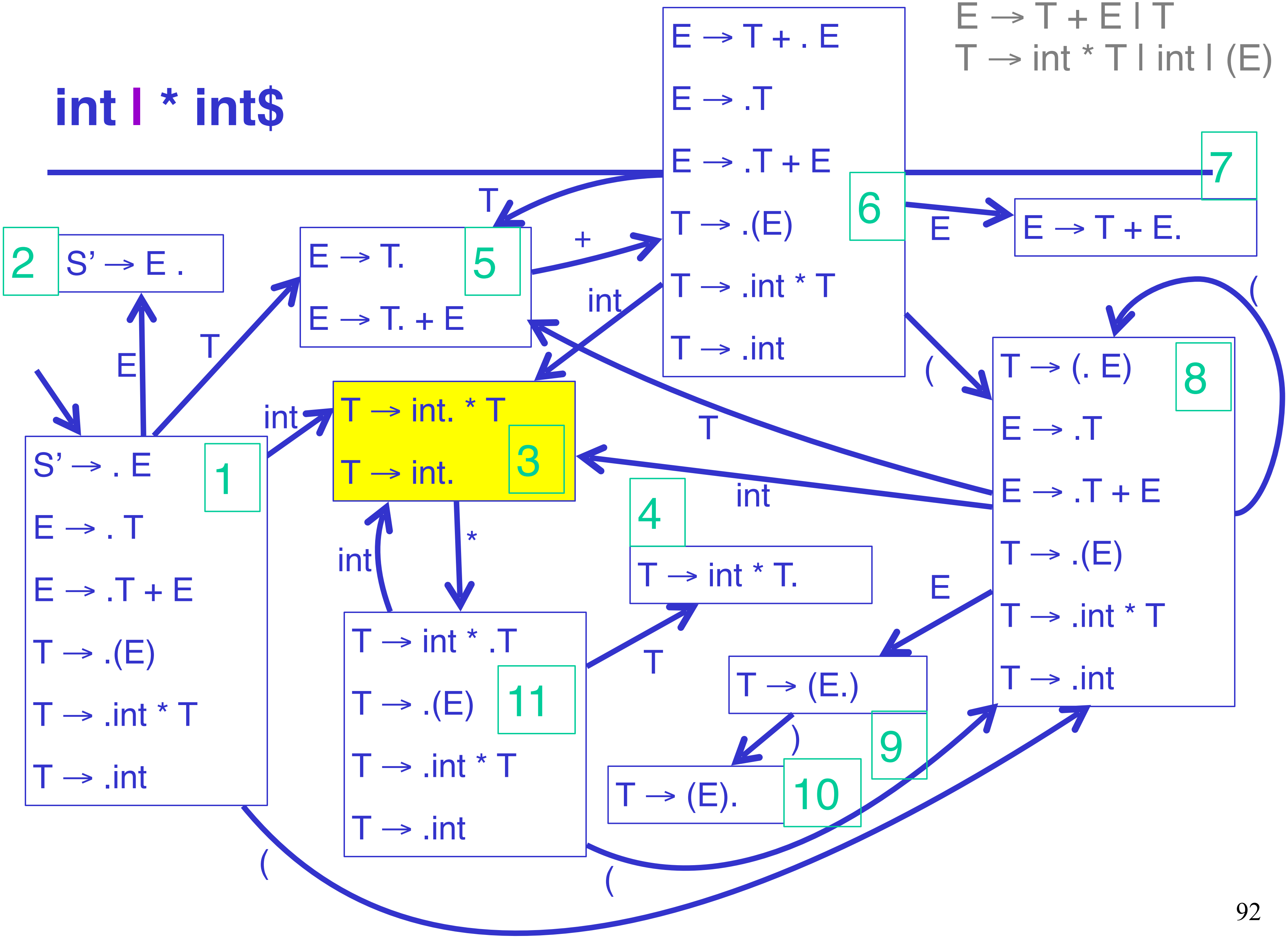
int | * int$

E → T + E | T
T → int * T | int | (E)



E → T + E | T
T → int * T | int | (E)

int | * int$



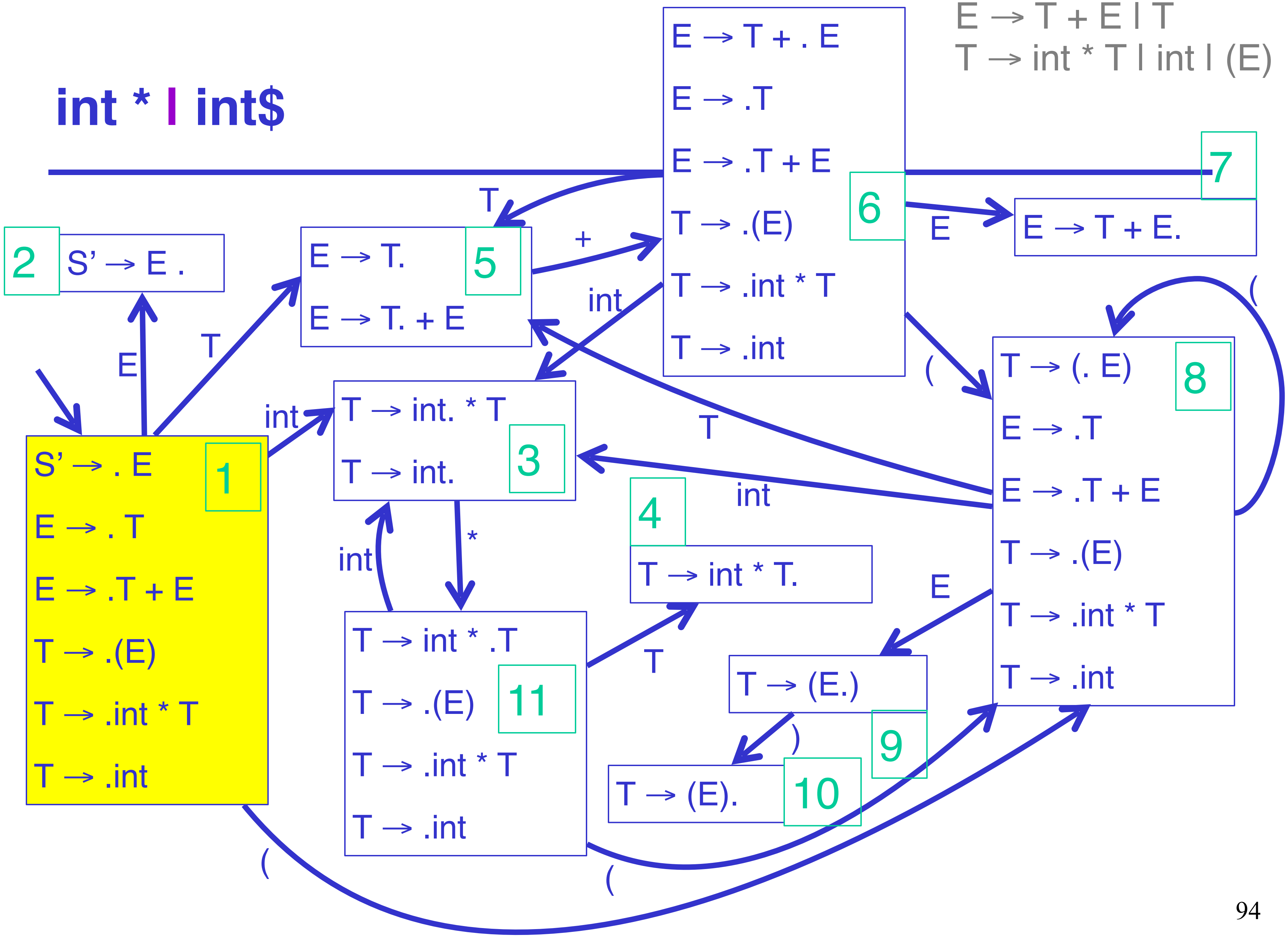
$E \rightarrow T + E \mid T$
$T \rightarrow int * T \mid int \mid (E)$

SLR Example

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

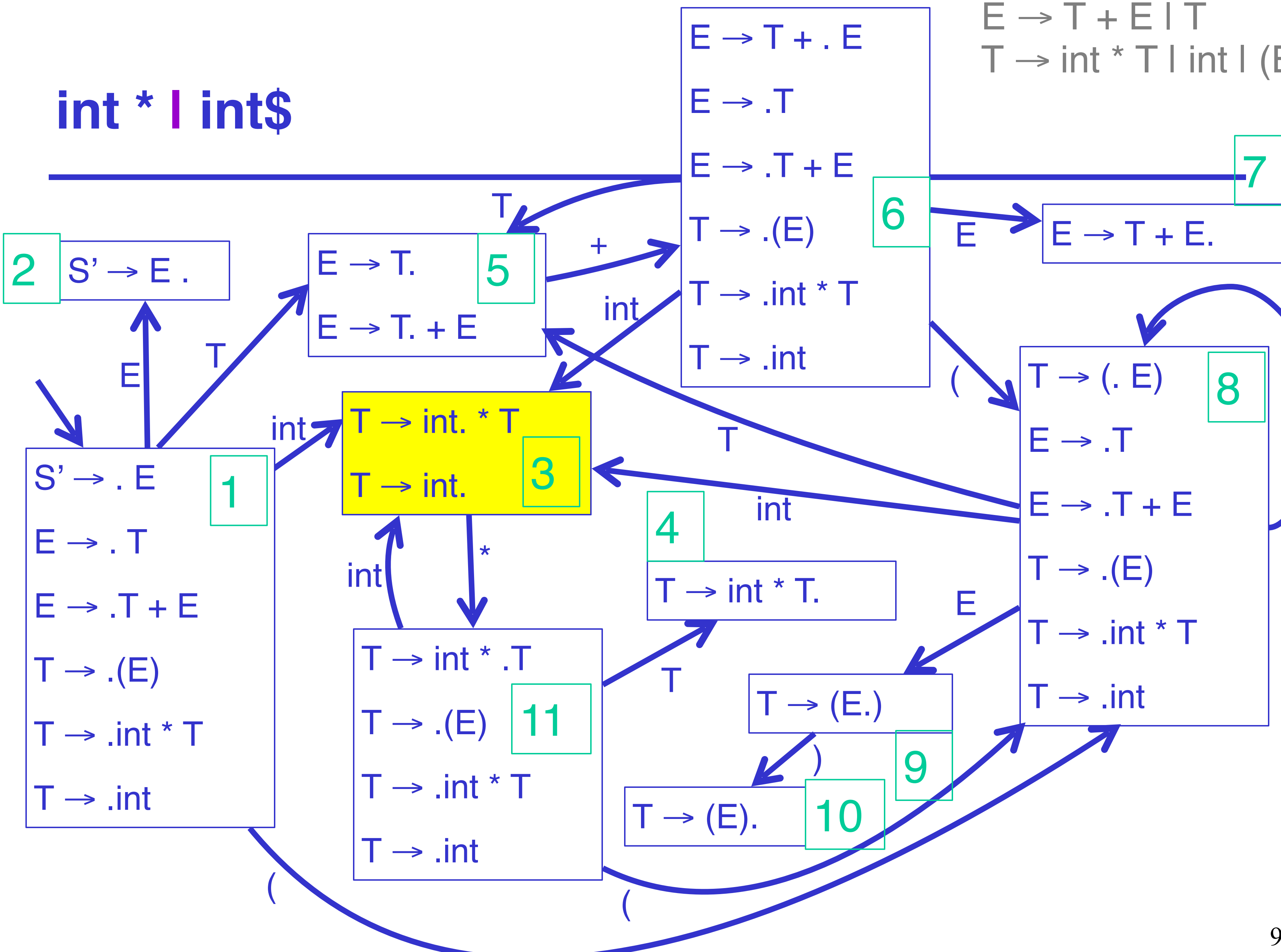
E → T + E | T
T → int * T | int | (E)

int * | int$



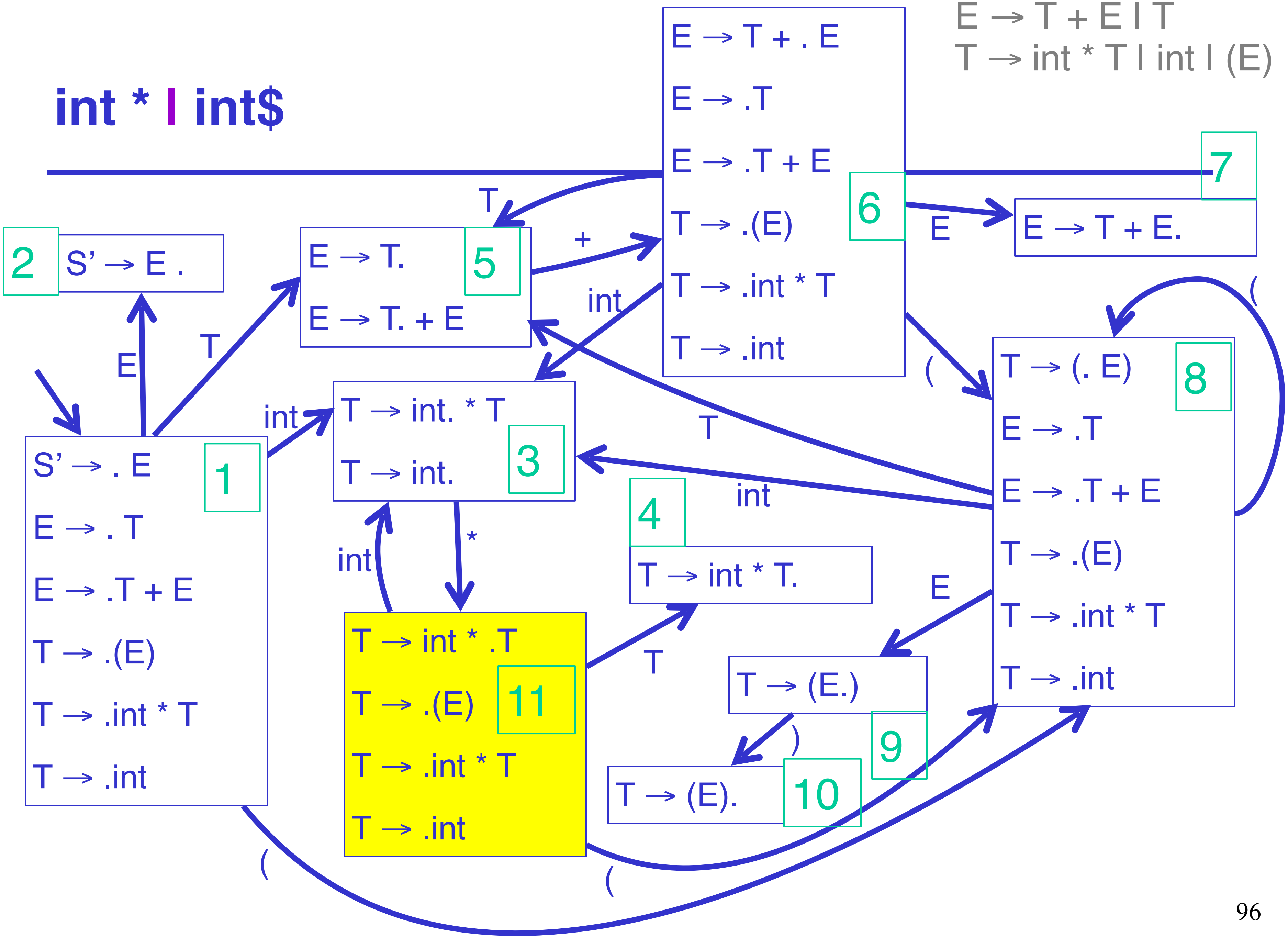
E → T + E | T
T → int * T | int | (E)

int * | int$



E → T + E | T
T → int * T | int | (E)

int * | int$



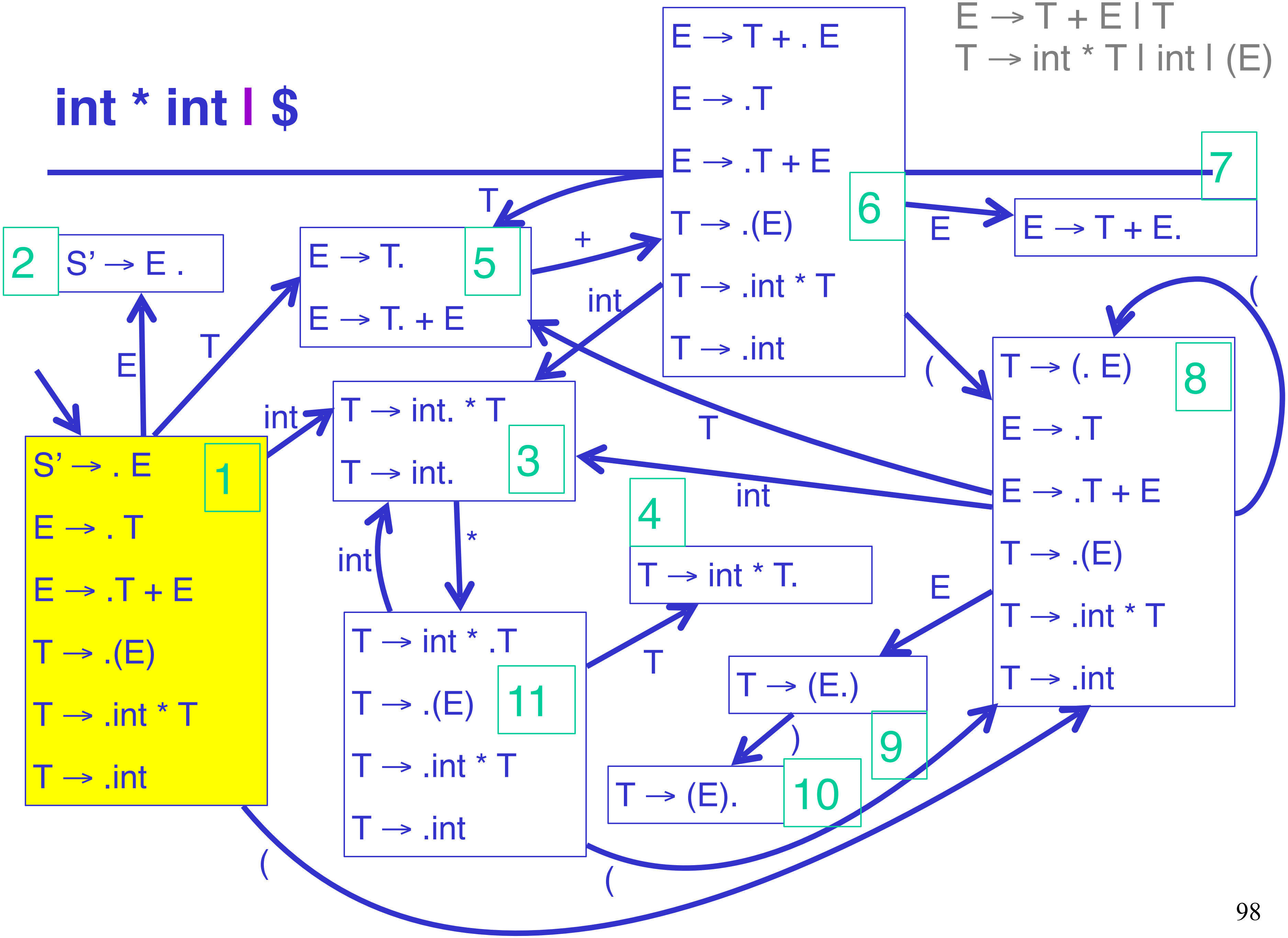
$E \to T + E \mid T$
$T \to int * T \mid int \mid (E)$

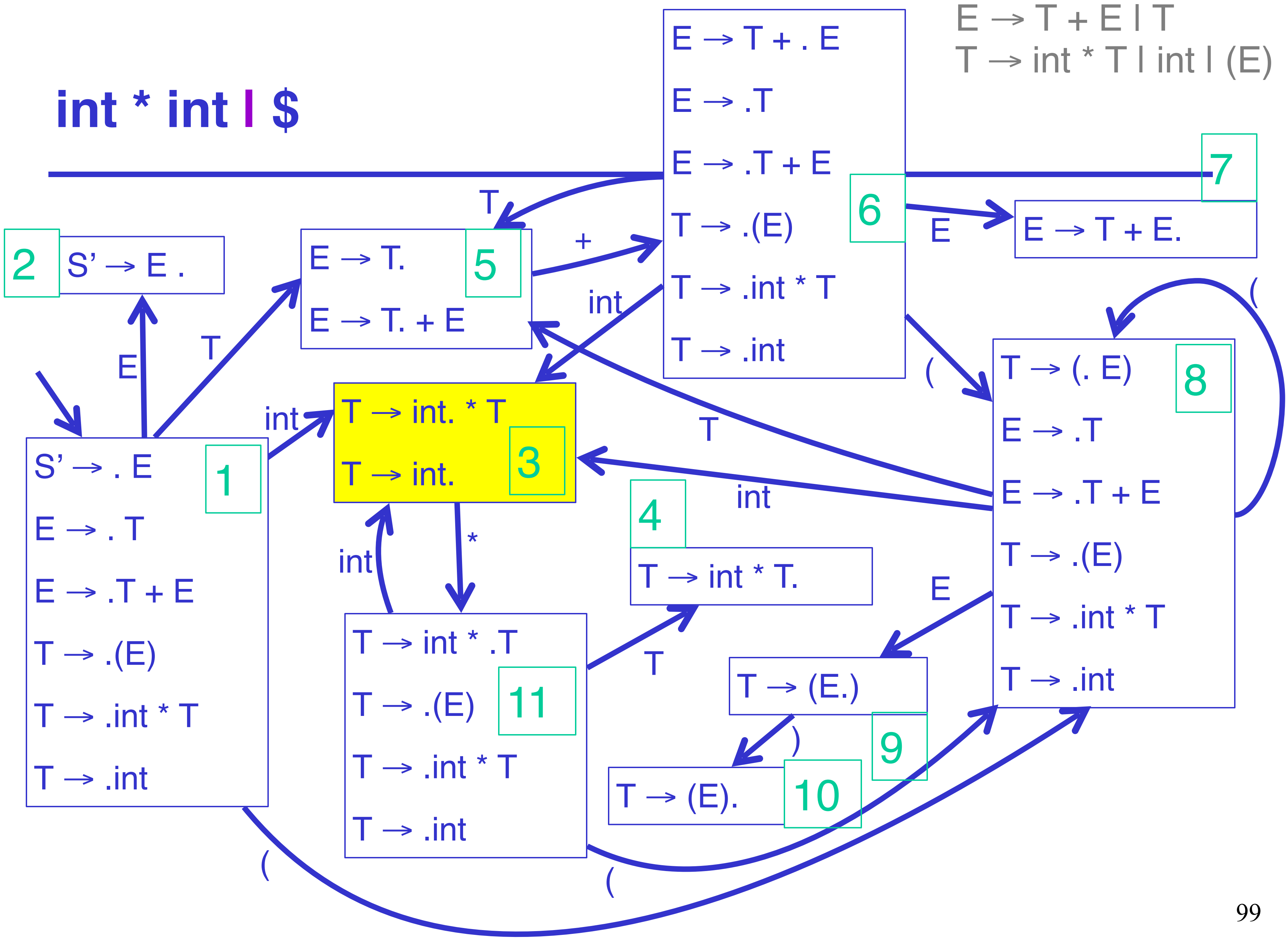
SLR Example

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→int

E → T + E | T
T → int * T | int | (E)

int * int | $

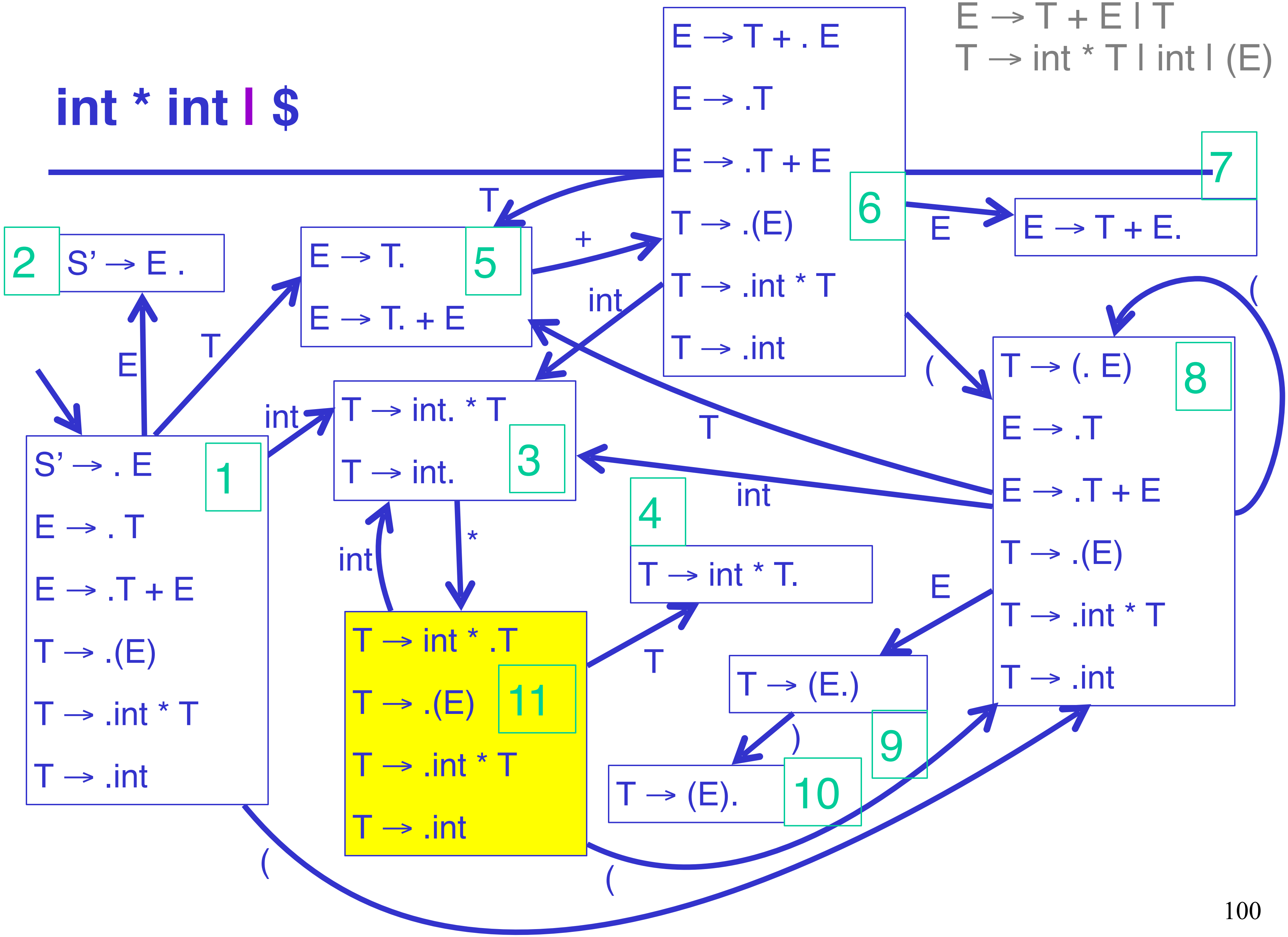




int * int I $
E → T + E I T
T → int * T I int I (E)
1
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int
2
S' → E .
3
T → int. * T
T → int.
4
T → int * T.
5
E → T.
E → T. + E
6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int
7
E → T + E.
8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int
9
T → (E.)
10
T → (E).
11
T → int * .T
T → .(E)
T → .int * T
T → .int
E
T
int
*
+
(
)

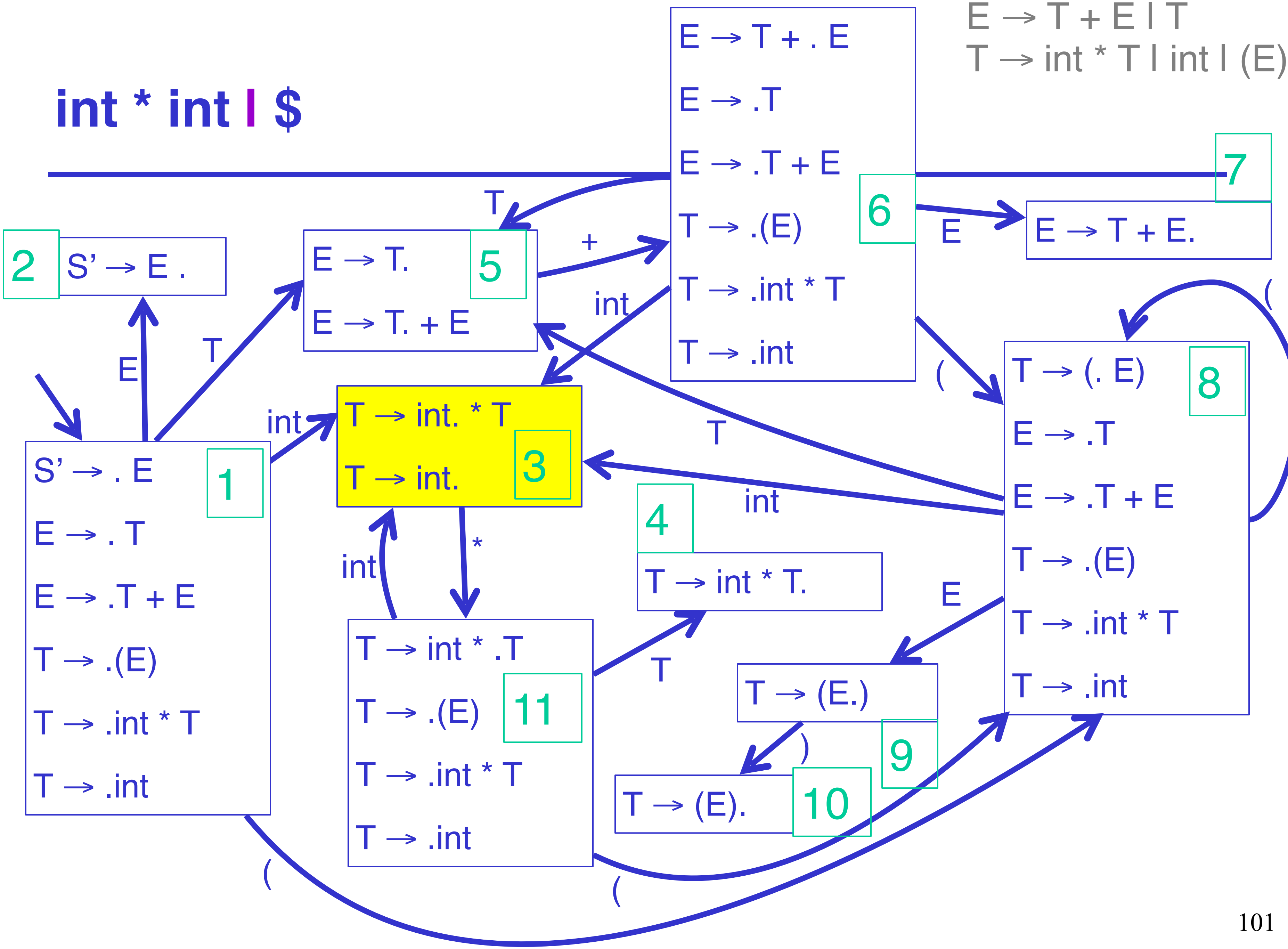
int * int | $

E → T + E | T
T → int * T | int | (E)



int * int | $

E → T + E | T
T → int * T | int | (E)



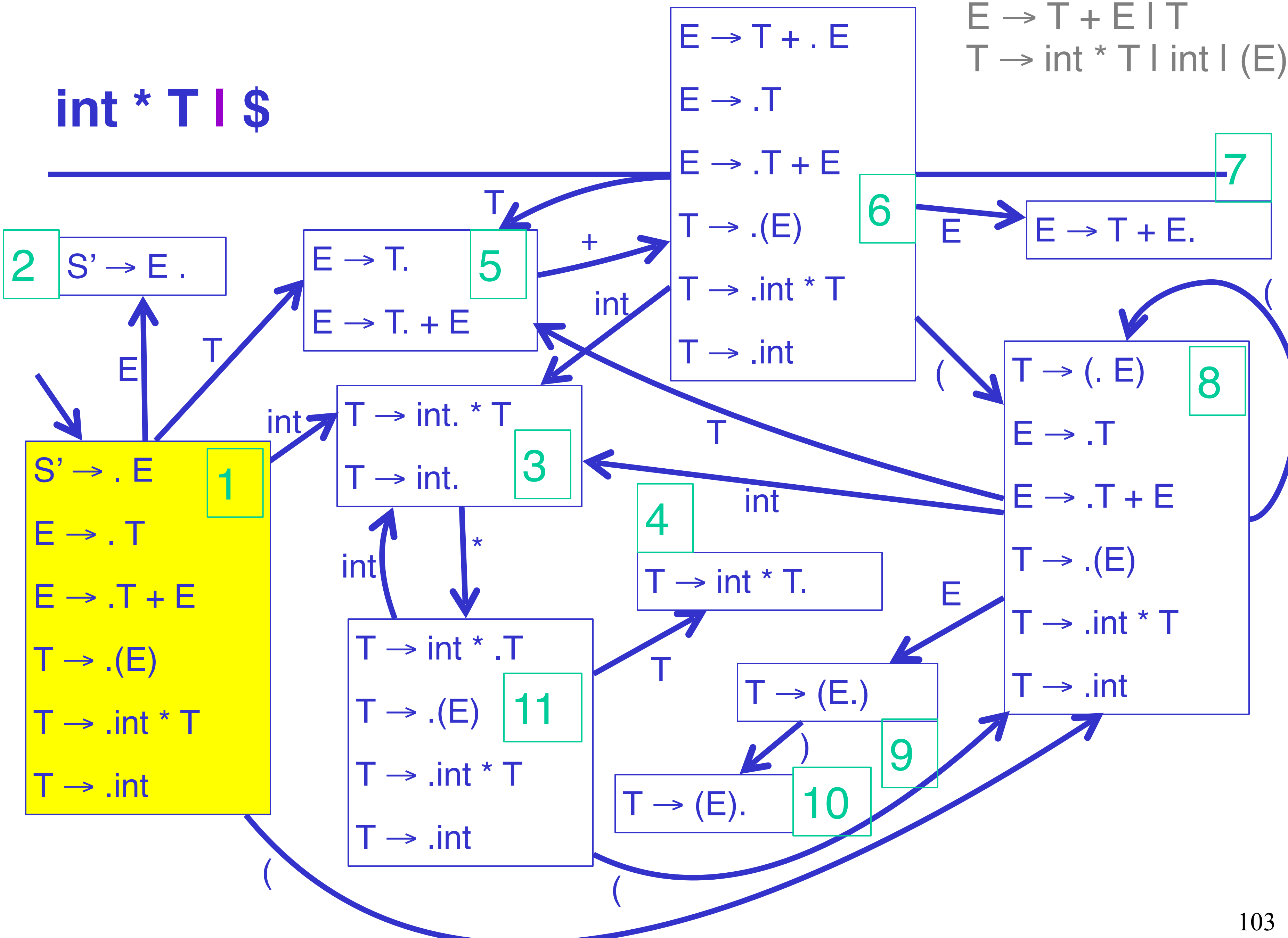
$E \to T + E \mid T$
$T \to int * T \mid int \mid (E)$

SLR Example

Configuration	DFA Halt State		Action
I int * int$	1		shift
int I * int$	3	* not in Follow(T)	shift
int * I int$	11		shift
int * int I$	3	$ ∈ Follow(T)	reduce T→int
int * T I$	4	$ ∈ Follow(T)	reduce T→int*T

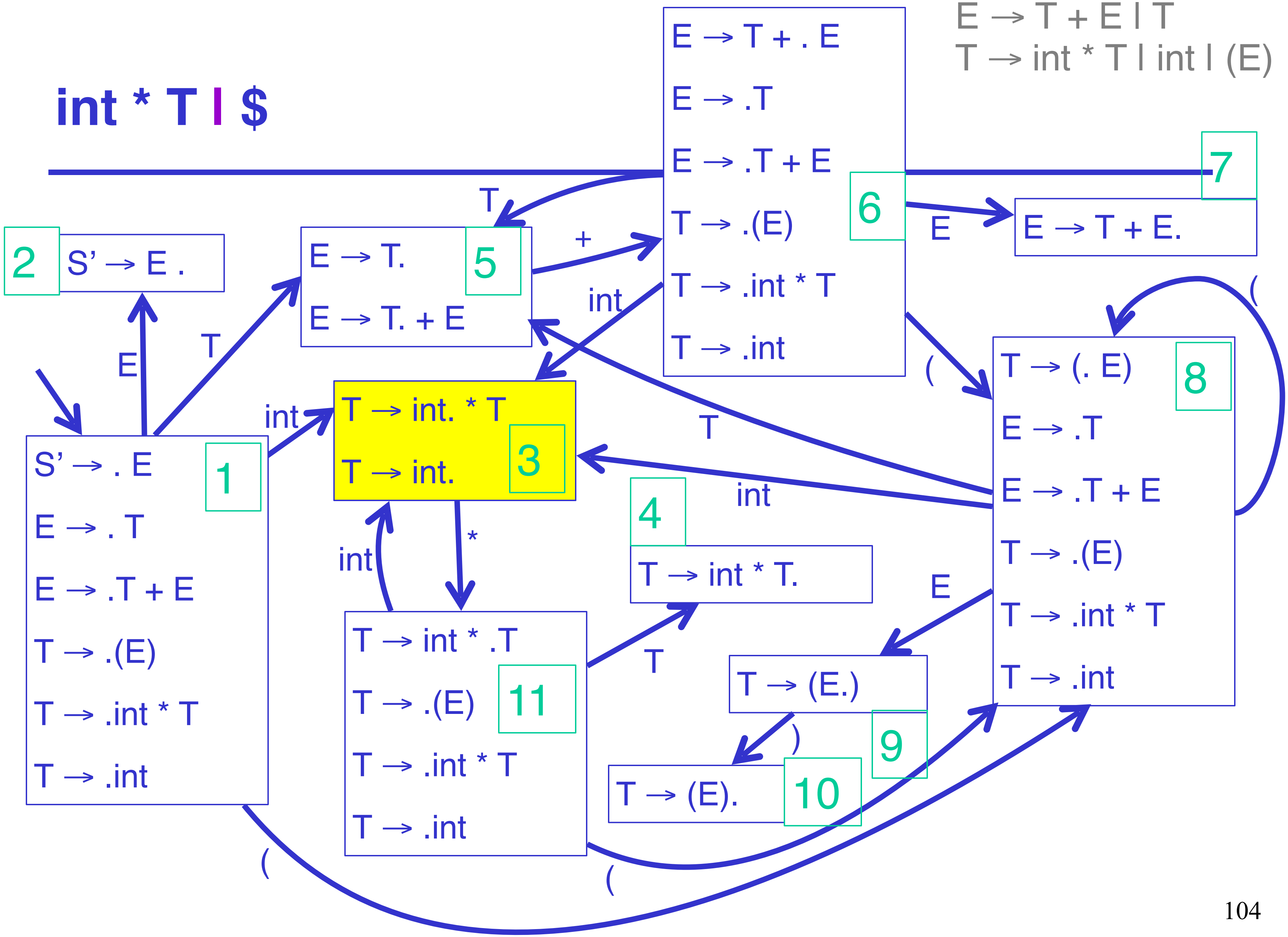
$E \rightarrow T + E \mid T$
$T \rightarrow int * T \mid int \mid (E)$

int * T | $



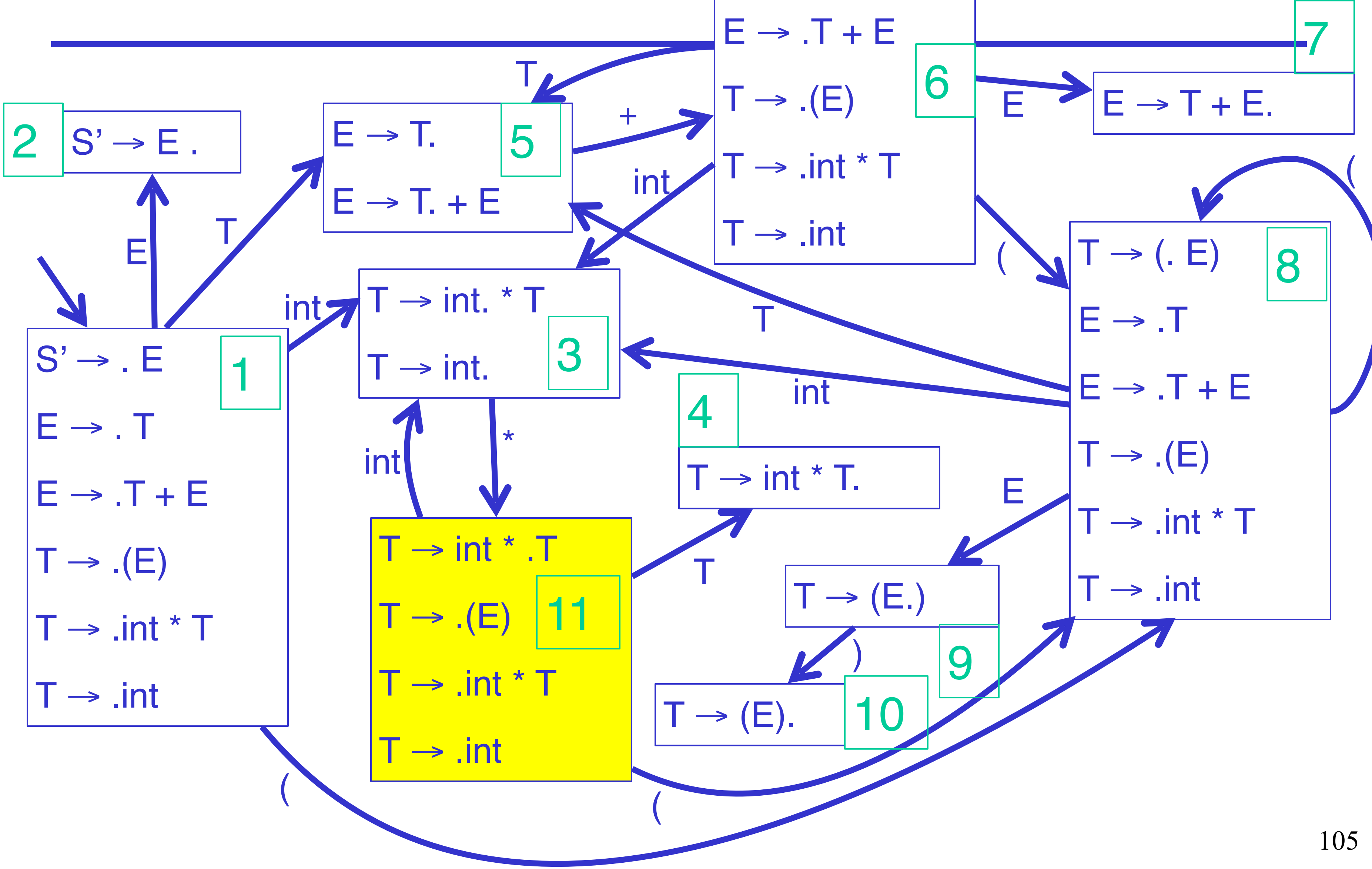
E → T + E | T
T → int * T | int | (E)

int * T | $



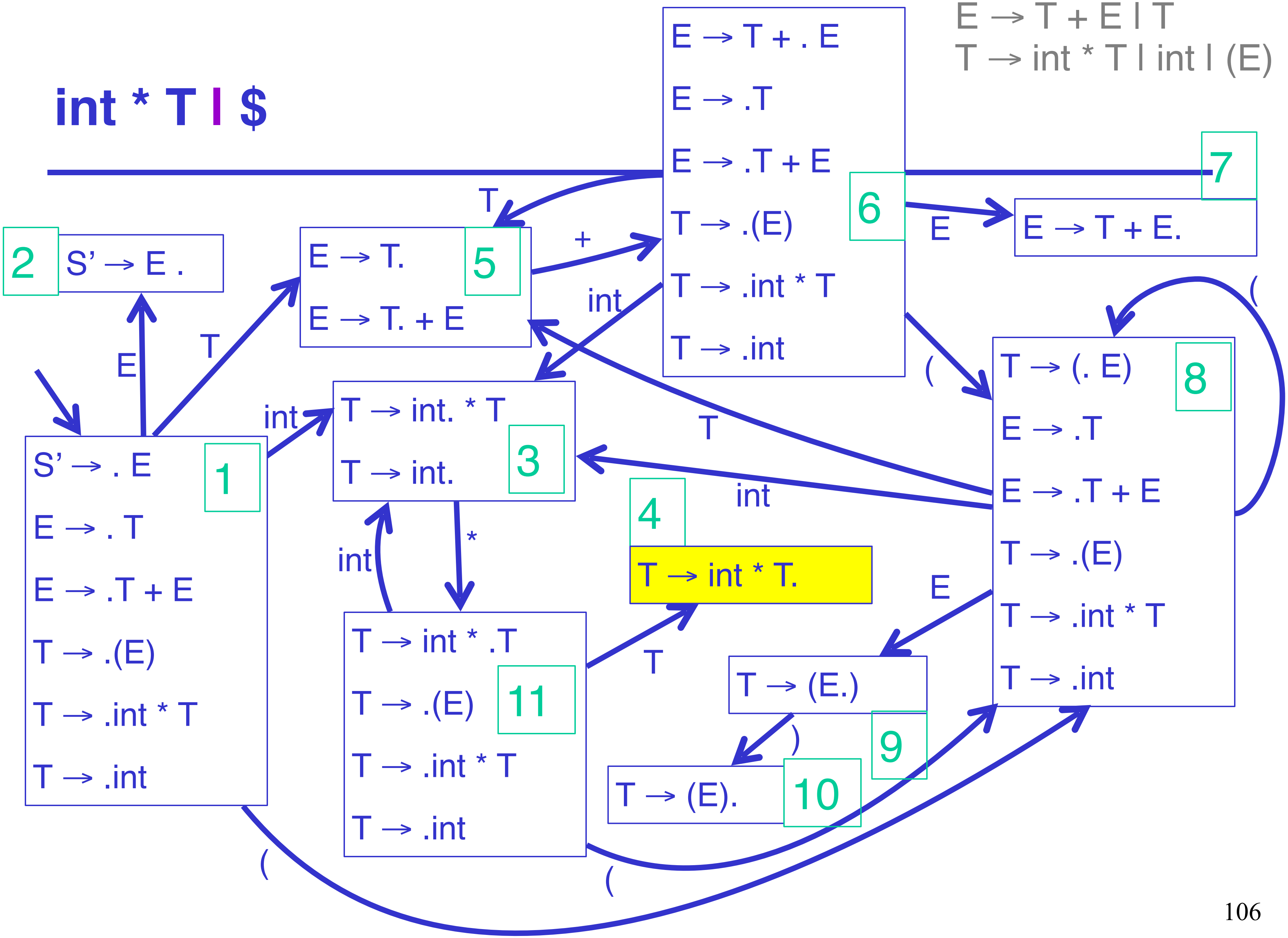
E → T + E | T
T → int * T | int | (E)

int * T | $



E → T + E | T
T → int * T | int | (E)

int * T | $



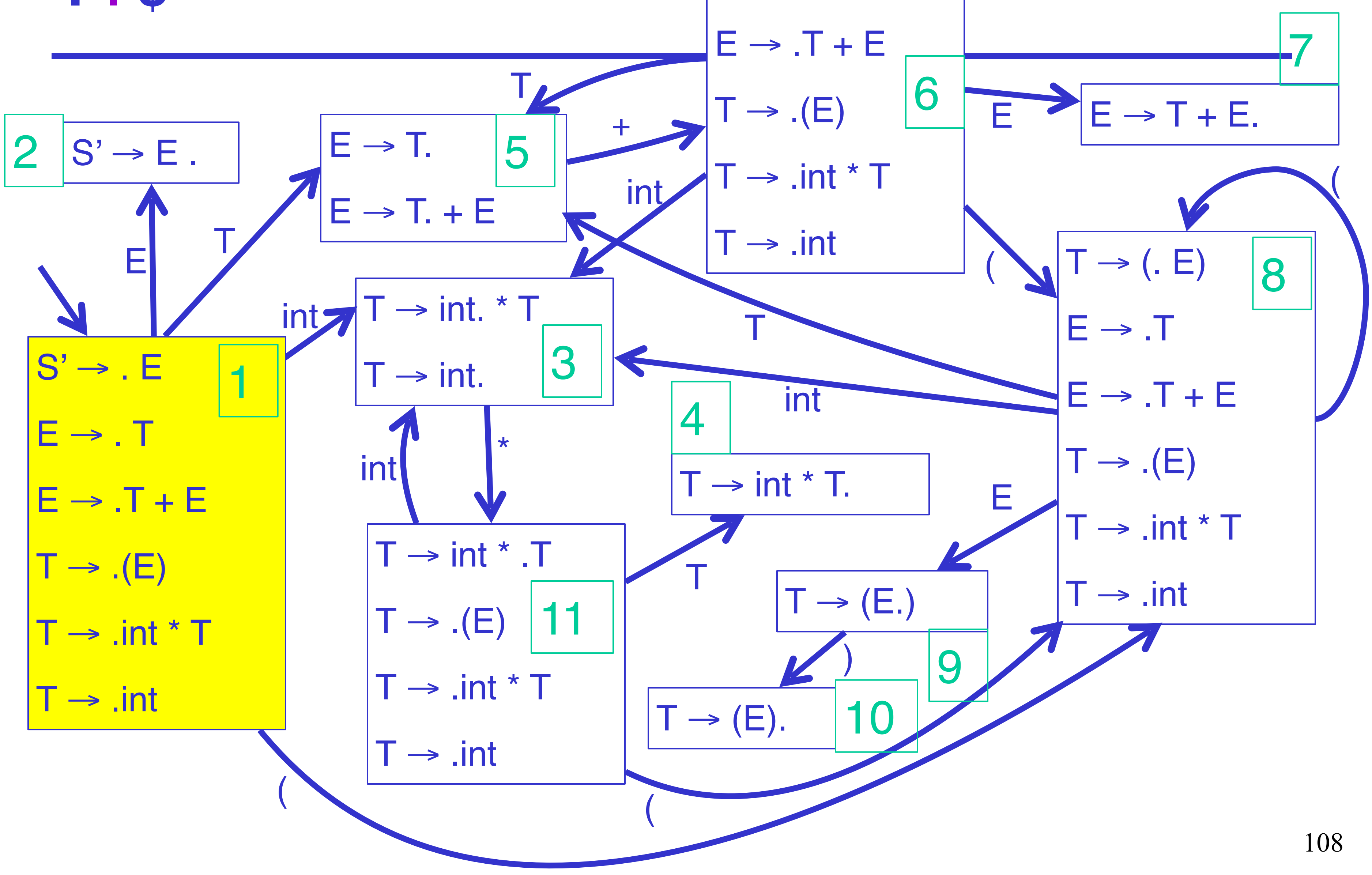
$E \rightarrow T + E \mid T$
$T \rightarrow int * T \mid int \mid (E)$

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→int
int * T |$	4	$ ∈ Follow(T)	reduce T→int*T
T |$	5	$ ∈ Follow(T)	reduce E→T

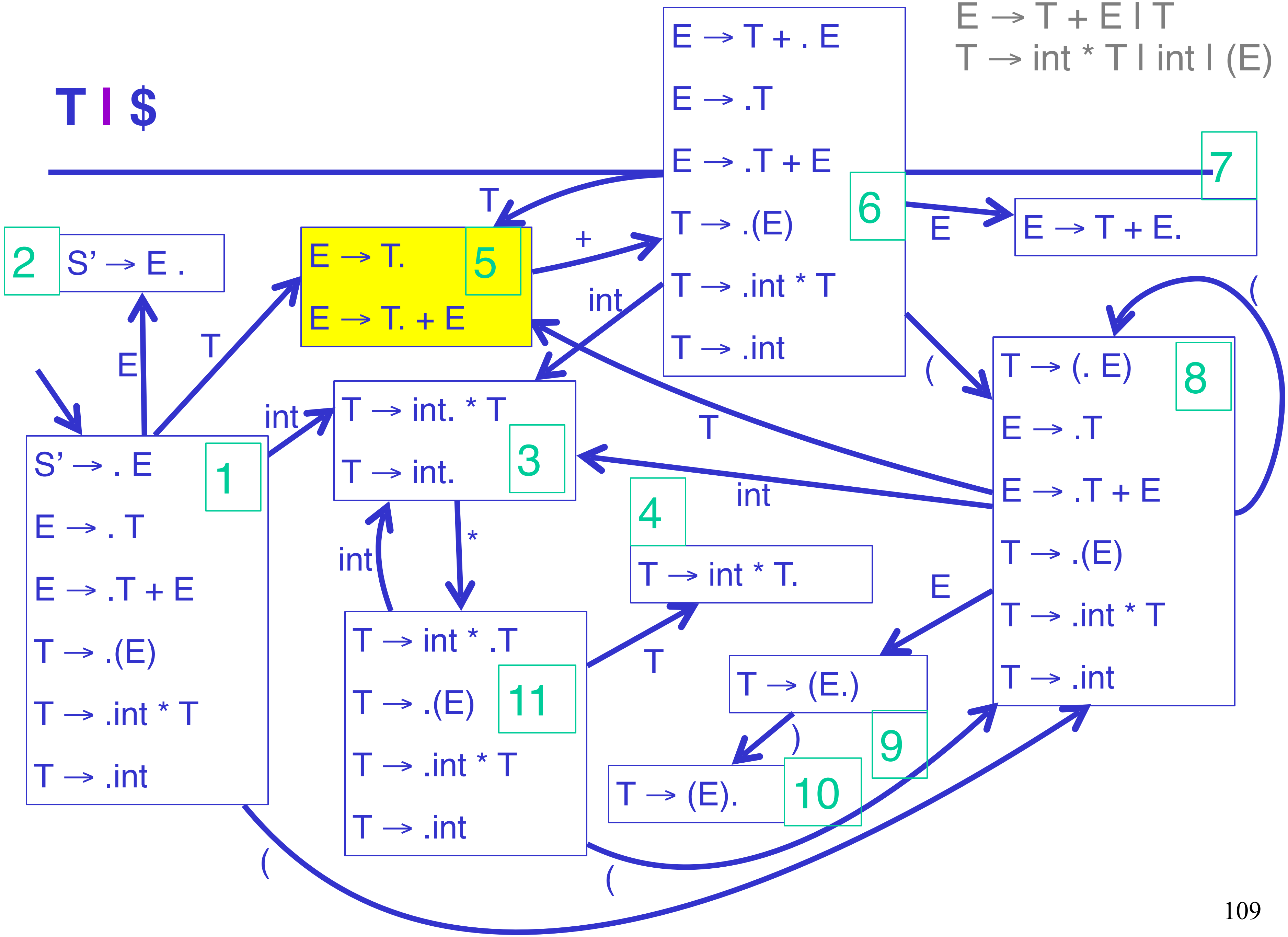
E → T + E | T
T → int * T | int | (E)

T | $



E → T + E | T
T → int * T | int | (E)

T | $



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

SLR Example

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

$$\langle \text{ symbol, DFA state } \rangle$$

An Improvement (Cont.)

- For a stack

  $\langle$ symbol$_1$, state$_1$ $\rangle$ . . . $\langle$ symbol$_n$, state$_n$ $\rangle$

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

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

Goto (DFA) Table

- Define $goto[i,A] = j$ if $state_i \rightarrow^{A} state_j$

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

SLR Parsing Algorithm

Let input = w$ be initial input

Let j = 0

Let DFA state 1 be the one with item S' → .S

Let stack = ⟨ dummy, 1 ⟩ // ⟨ symbol, state ⟩

repeat

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

    shift k: push ⟨ input[j++], k ⟩

    reduce X → α:

      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
  – Take a look at the LR(1) automaton for your parser!