Compiler

Parsing: Bottom-Up

An Introductory Example

- Bottom-up parsers neither fail on left-recursion nor need left-factored grammars
- Hence we can revert to the “natural” grammar for our example:

  $E \rightarrow E + T \mid T$

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

- Consider the string: int * int + int

The Idea

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

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

Observation

- Read the productions in this bottom-up parse in reverse (i.e., from bottom to top)
- This is a rightmost derivation!

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

A Bottom-up Parse

int * int + int

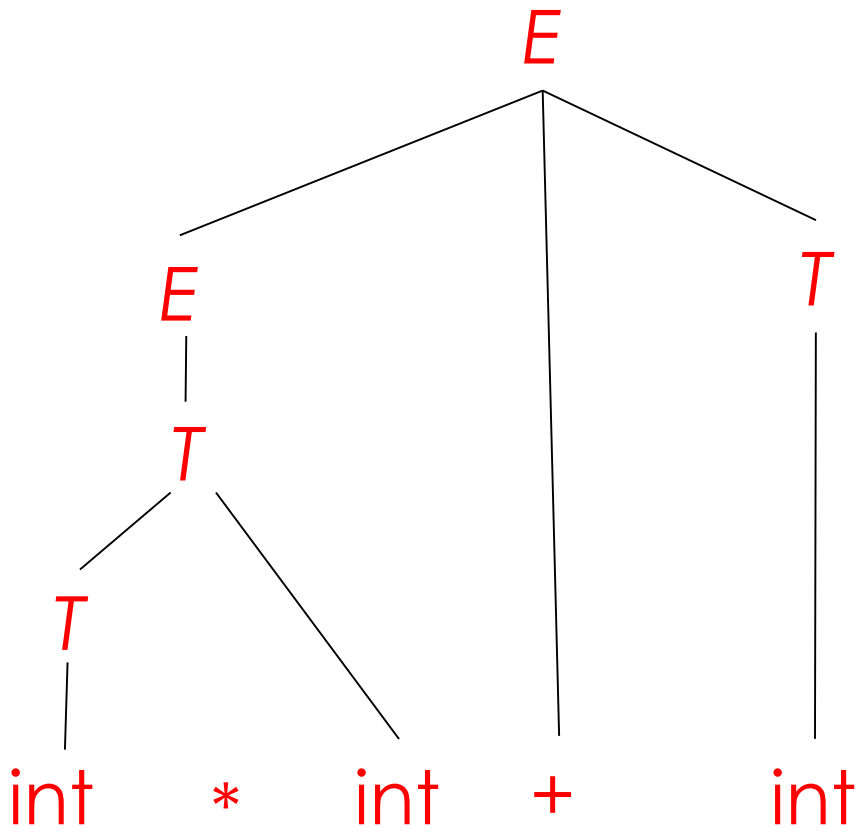
T * int + int

T + int

E + int

E + *T*

E



A bottom-up parser traces a rightmost derivation in reverse

A Bottom-up Parse in Detail (1)

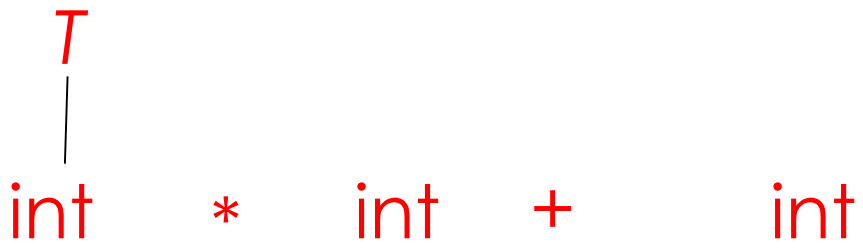
int * int + int

int * int + int

A Bottom-up Parse in Detail (2)

int * int + int

T * int + int

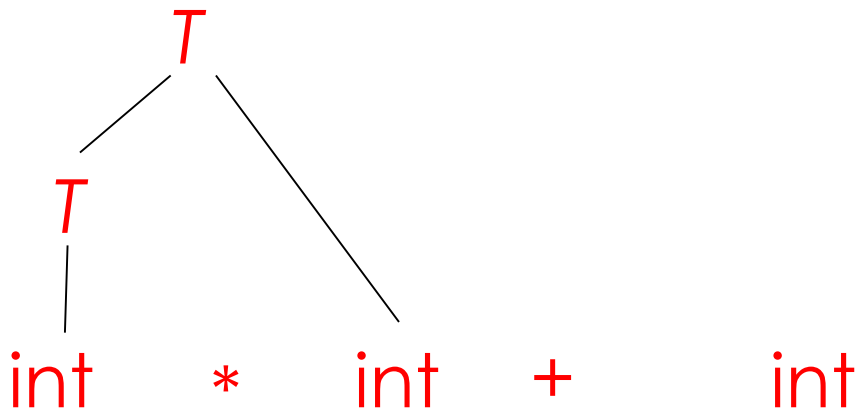


A Bottom-up Parse in Detail (3)

int * int + int

T * int + int

T + int



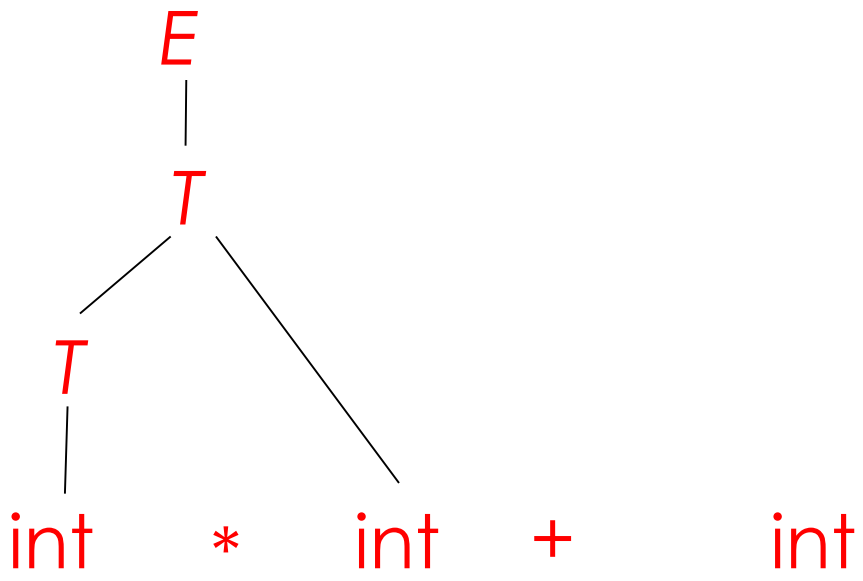
A Bottom-up Parse in Detail (4)

int * int + int

T * int + int

T + int

E + int



A Bottom-up Parse in Detail (5)

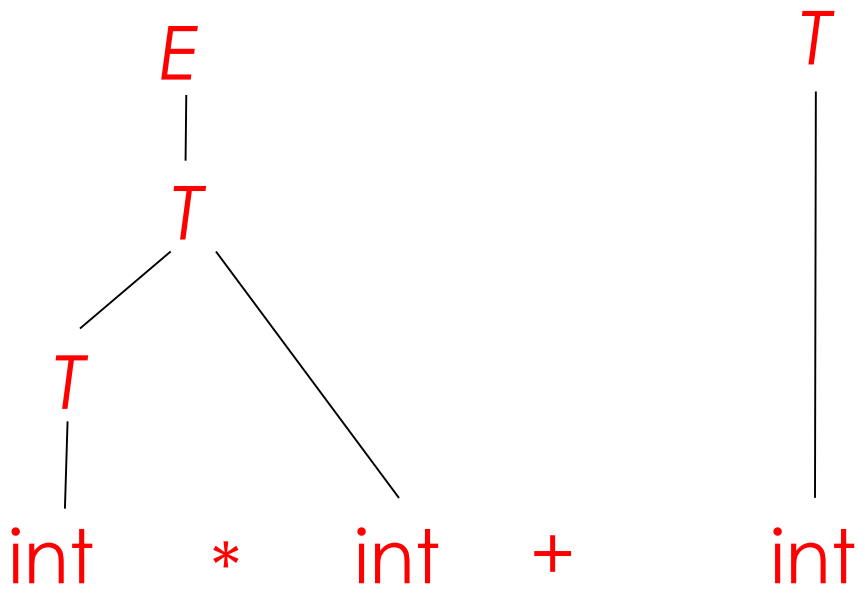
int * int + int

T * int + int

T + int

E + int

E + *T*



A Bottom-up Parse in Detail (6)

int * int + int

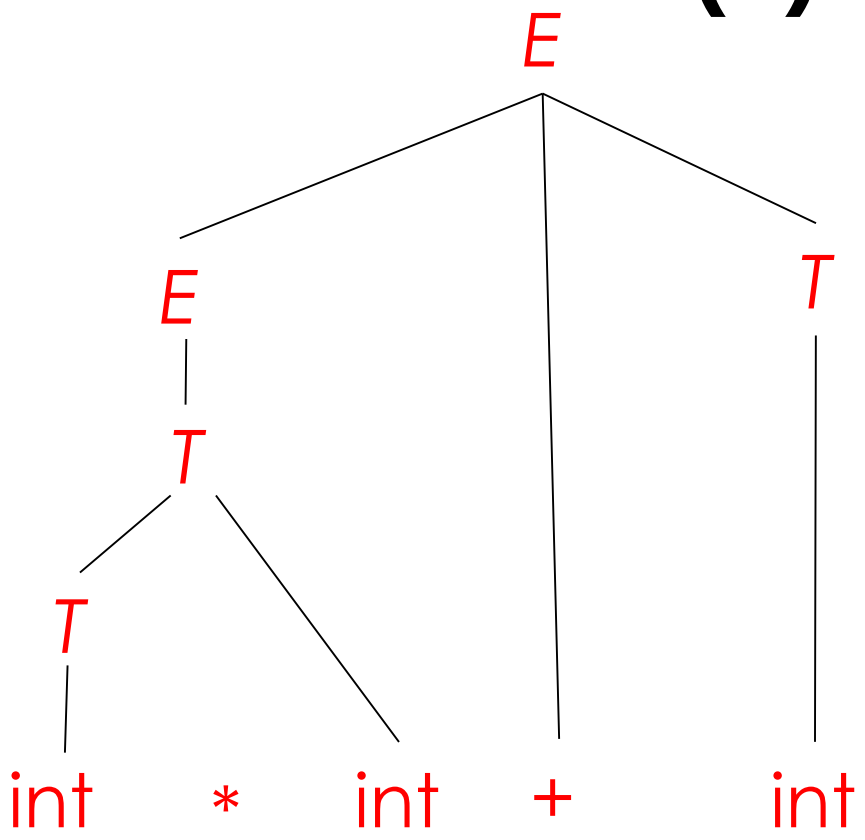
T * int + int

T + int

E + int

E + *T*

E



A Trivial Bottom-Up Parsing Algorithm

Let $\mathcal{I}$ = input string

repeat

pick a non-empty substring β of $\mathcal{I}$

where $X \rightarrow \beta$ is a production

if no such β, backtrack

replace one β by X in $\mathcal{I}$

until $\mathcal{I} = S$ (the start symbol) or all possibilities are exhausted

For You To Do

Do you see any problems with this algorithm?

Think about performance and completeness!

Where Do Reductions Happen

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, i.e. $\omega \in T^*$

Why?

Because $\alpha X\omega \Rightarrow \alpha\beta\omega$ is a step in a right-most derivation

Idea

- Split string into two substrings
  - Right substring : as yet unexamined by parsing (a string of terminals)
  - Left substring : has terminals and non-terminals
- The dividing point is marked by a |
  - The | is not part of the string
- Initially, all input is unexamined $|x_1 x_2 \ldots x_n$

Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Shift

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

$$ABC|xyz \Rightarrow ABCx|yz$$

Reduce

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

$$Cbxy|ijk \Rightarrow CbA|ijk$$

  - *xy* is called a *handle*

The Example with Shift-Reduce Parsing

|int * int + int

The Example with Shift-Reduce Parsing

|int * int + int	shift
int | * int + int	

The Example with Shift-Reduce Parsing

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

The Example with Shift-Reduce Parsing

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

The Example with Shift-Reduce Parsing

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

The Example with Shift-Reduce Parsing

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

The Example with Shift-Reduce Parsing

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

The Example with Shift-Reduce Parsing

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

The Example with Shift-Reduce Parsing

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

The Example with Shift-Reduce Parsing

|int * int + int	shift
int | * int + int	reduce $T \rightarrow$ int
T | * int + int	shift
T * | int + int	shift
T * int | + int	reduce $T \rightarrow T$ * int
T | + int	reduce $E \rightarrow T$
E | + int	shift, shift
E + int |	reduce $T \rightarrow$ int
E + T |	reduce $E \rightarrow E$+T
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)
- This is how a Pushdown Automaton works.

Key Issue

- How do we decide when to shift or reduce?
  - Consider step $T * \text{int} \mid + \text{int}$
  - We could reduce by $T \rightarrow \text{int}$ giving $T * T \mid + \text{int}$
  - A fatal mistake: No way to reduce to the start symbol *E*
- This is resolved by various bottom-up parsing algorithms

Conflicts

- But what if there is a choice?
  - 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
- Generic shift-reduce strategy:
  - If there is a handle on top of the stack, reduce
  - Otherwise, shift

Source of Conflicts

- Ambiguous grammars always cause conflicts
- But beware, so do many non-ambiguous grammars

Conflict Example

Consider our favorite ambiguous grammar:

$$
\begin{array}{lcl}
E & \rightarrow & E + E \\
 & | & E * E \\
 & | & (E) \\
 & | & \text{int}
\end{array}
$$

One Shift-Reduce Parse

|int * int + int	shift
. . .	. . .
E * *E* | + int	reduce $E \rightarrow E * E$
E | + int	shift
E + | int	shift
E + int|	reduce $E \rightarrow$ int
E + *E* |	reduce $E \rightarrow E + E$
E |	

Another Shift-Reduce Parse

|int * int + int	shift
. . .	. . .
E * *E* | + int	shift
E * *E* + | int	shift
E * *E* + int |	reduce $E \rightarrow$ int
E * *E* + *E*|	reduce $E \rightarrow E + E$
E * *E* |	reduce $E \rightarrow E * E$
E |	

Example Notes

- In the second step $E * E \mid + \text{int}$ we can either shift or reduce by $E \rightarrow E * E$
- Choice determines priority of + and *
- As noted previously, grammar can be rewritten to enforce precedence
- Precedence declarations are an alternative

Precedence Declarations Revisited

- Precedence declarations cause shift-reduce parsers to resolve conflicts in certain ways
- Declaring "* has greater precedence than +" causes parser to reduce at *E* * *E* | + int
- More precisely, precedence declaration is used to resolve conflict between reducing a * and shifting a + (and vice versa)

Precedence Declarations Revisited

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

Using Lookahead Strings

- Use lookahead for resolving conflicts
- Problem: *reduction history* remains disregarded
- Example: Consider the **unambigious** grammar

  $S \rightarrow aA \mid Ba$

  $A \rightarrow Ba$

  $B \rightarrow b$

  - For input ba we need to reduce Ba by $S \rightarrow Ba$.
  - For input aba we need to reduce Ba by $A \rightarrow Ba$.
  - But the lookahead string is the same (ε) is both cases.
- Solution: Encode the *reduction history* in the stack.

LR(1) Parsing

- Use lookahead for resolving conflicts
- DFA for memorizing *reduction history*
- Introduce new start symbol S' and unique production $S' \rightarrow S$.
  (Reason: S' does not occur in any other production)

LR(1) Parsing

- NFA
  - States are LR(1)-items $(A \rightarrow \alpha \bullet \beta, u)$

    where $A \rightarrow \alpha\beta \in P$, α is parsed prefix of handle

    $u \in T \cup \{\varepsilon\}$ is lookahead expected behind $\alpha\beta$

  - Transitions:
    - $(A \rightarrow \alpha \cdot X\beta, u) \xrightarrow{X} (A \rightarrow \alpha X \cdot \beta, u)$ $\quad X \in T \cup N$
    - $(A \rightarrow \alpha \cdot B\beta, u) \xrightarrow{\varepsilon} (B \rightarrow \cdot\gamma, v)$ $\quad B \in N$, $B \rightarrow \gamma \in P$, $v \in \text{First}(\beta u)$

LR(1) Parsing

- NFA
  - Start state ($S' \rightarrow \bullet S, \varepsilon$)
  - Accepting state ($S' \rightarrow S\bullet, \varepsilon$)
- DFA
  - NFA $\rightarrow$ DFA conversion (states = sets of LR(1)-items)
  - Actions associated with states and lookahead
    - Act(q,t) = shift iff $(A \rightarrow \alpha \bullet t\, \beta, u) \in q,\ t \in T$
    - Act(q,u) = ($A \rightarrow \beta$) iff $(A \rightarrow \beta \bullet, u) \in q,\ u \in T \cup \{\varepsilon\}$

LR(1) Parsing

- Parsing algorithm
  - Stack symbols: States (Sets of LR(1)-items)
  - Initial stack is $q_0 \mid \omega_c$ where ω_c is the complete input
  - **Parse steps**
    - $\rho\, q \mid t\, \omega \quad \rightarrow \quad \rho\, q\, \delta(q,t) \mid \omega$ iff Act(q,t) = shift
    - $\rho\, q\, \rho' q' \mid \omega \quad \rightarrow \quad \rho\, q\, \delta(q,A) \mid \omega$ iff Act(q',u) = $(A \rightarrow \beta)$
      $\mid \beta \mid = \mid \rho' q' \mid$
      u = First(ω)
  - Accept with stack $q_0\, q_f \mid \varepsilon$ where q_f is accepting state

LR(1) Parsing

- Example $E \rightarrow E + T \mid T$

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

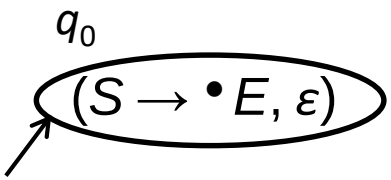
- With new start symbol

  $S \rightarrow E$

  $E \rightarrow E + T \mid T$

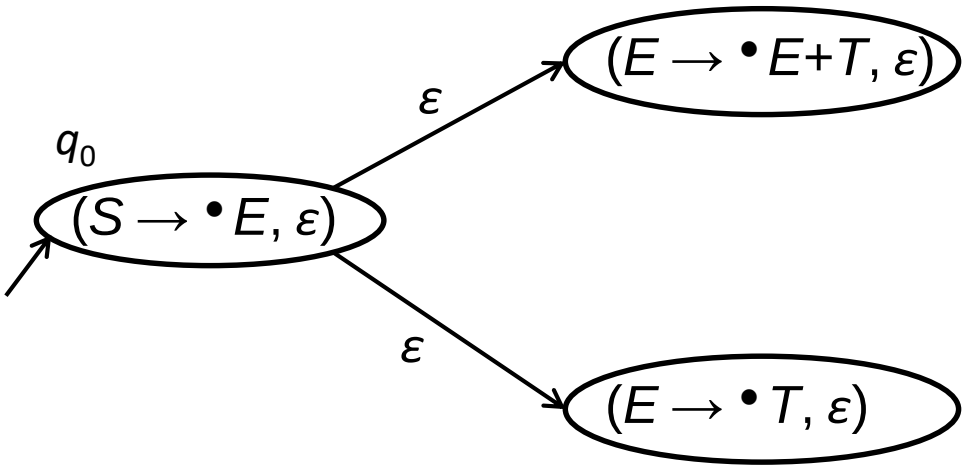
  $T \rightarrow T * \text{int} \mid \text{int}$

Example: LR(1) NFA Construction



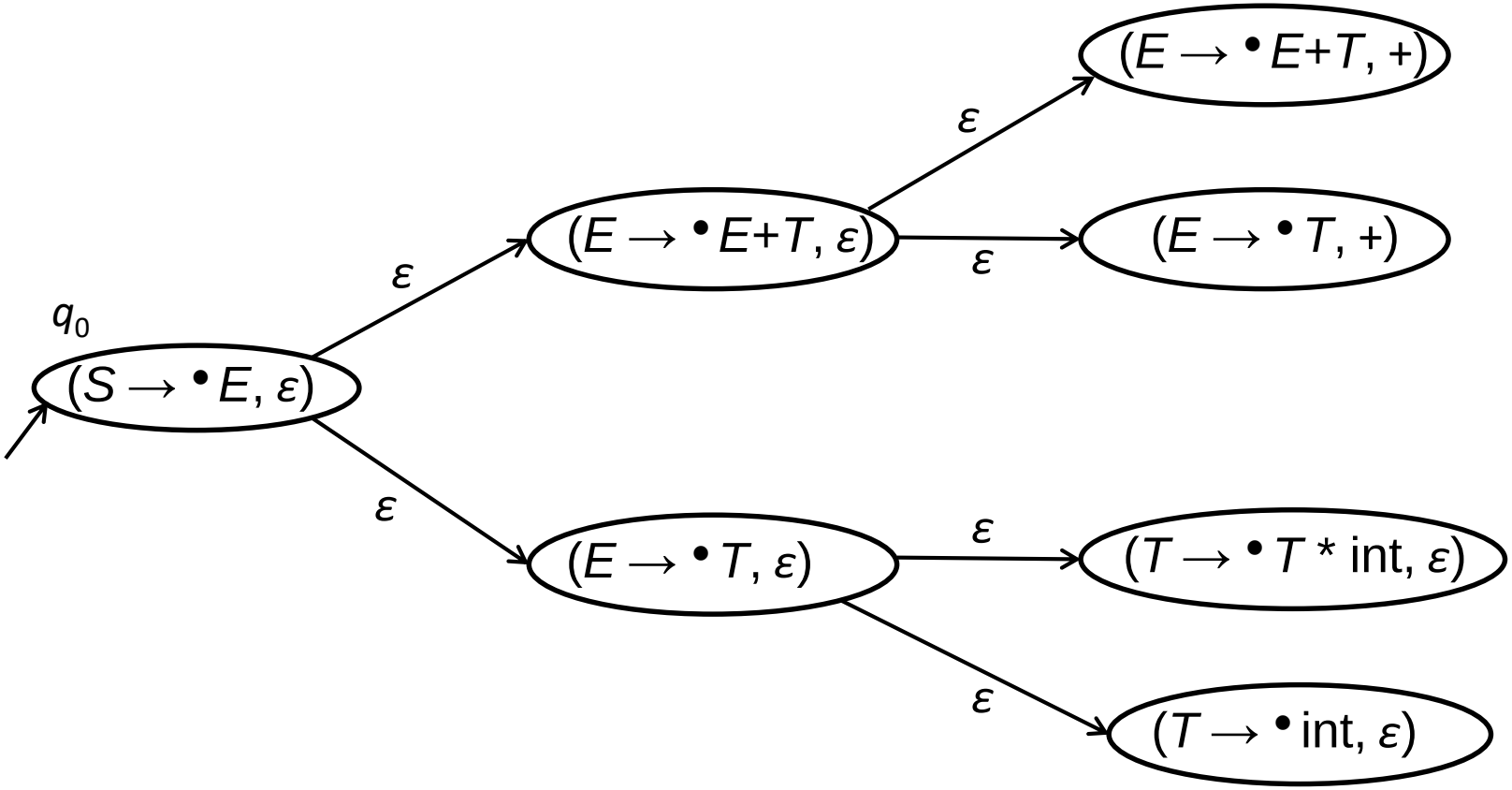
- $(A \rightarrow \alpha \bullet X \beta, u) \xrightarrow{X} (A \rightarrow \alpha X \bullet \beta, u) \quad X \in T \cup N$
- $(A \rightarrow \alpha \bullet B \beta, u) \xrightarrow{\varepsilon} (B \rightarrow \bullet \gamma, v) \quad B \in N, B \rightarrow \gamma \in P, v \in \text{First}(\beta u)$

Example: LR(1) NFA Construction



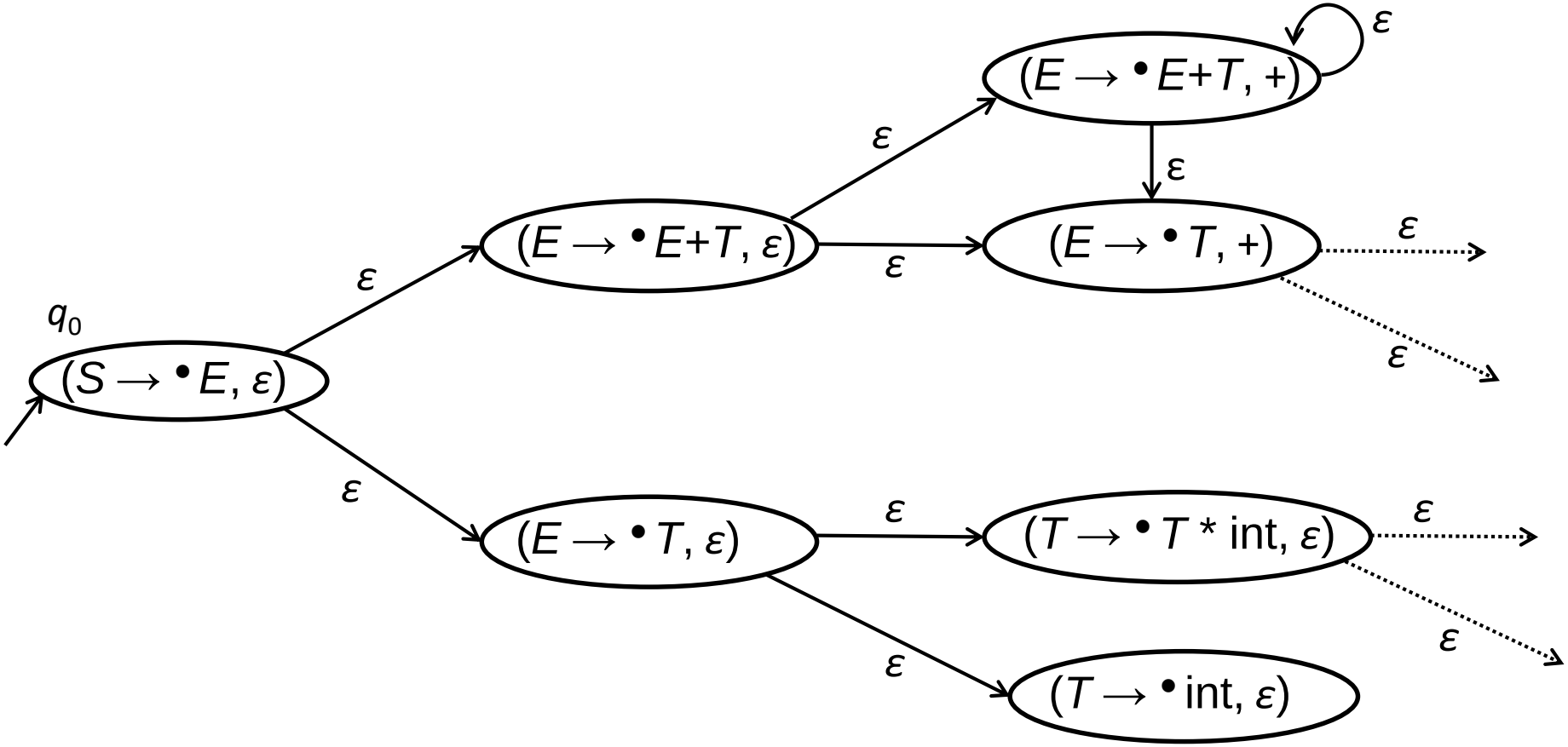
- $(A \rightarrow \alpha \bullet X \beta, u) \xrightarrow{X} (A \rightarrow \alpha X \bullet \beta, u) \quad X \in T \cup N$
- $(A \rightarrow \alpha \bullet B \beta, u) \xrightarrow{\varepsilon} (B \rightarrow \bullet \gamma, v) \quad B \in N, B \rightarrow \gamma \in P, v \in \text{First}(\beta u)$

Example: LR(1) NFA Construction



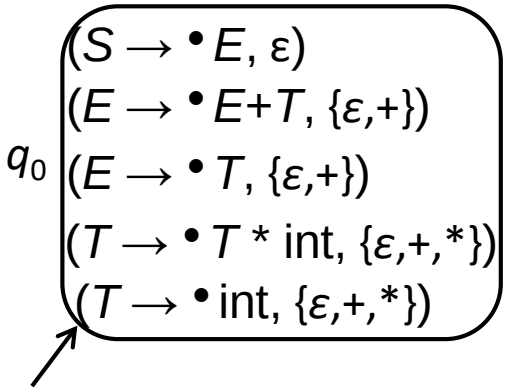
- $(A \rightarrow \alpha \bullet X \beta, u) \xrightarrow{X} (A \rightarrow \alpha X \bullet \beta, u) \quad X \in T \cup N$
- $(A \rightarrow \alpha \bullet B \beta, u) \xrightarrow{\varepsilon} (B \rightarrow \bullet \gamma, v) \quad B \in N, B \rightarrow \gamma \in P, v \in \text{First}(\beta u)$

Example: LR(1) NFA Construction

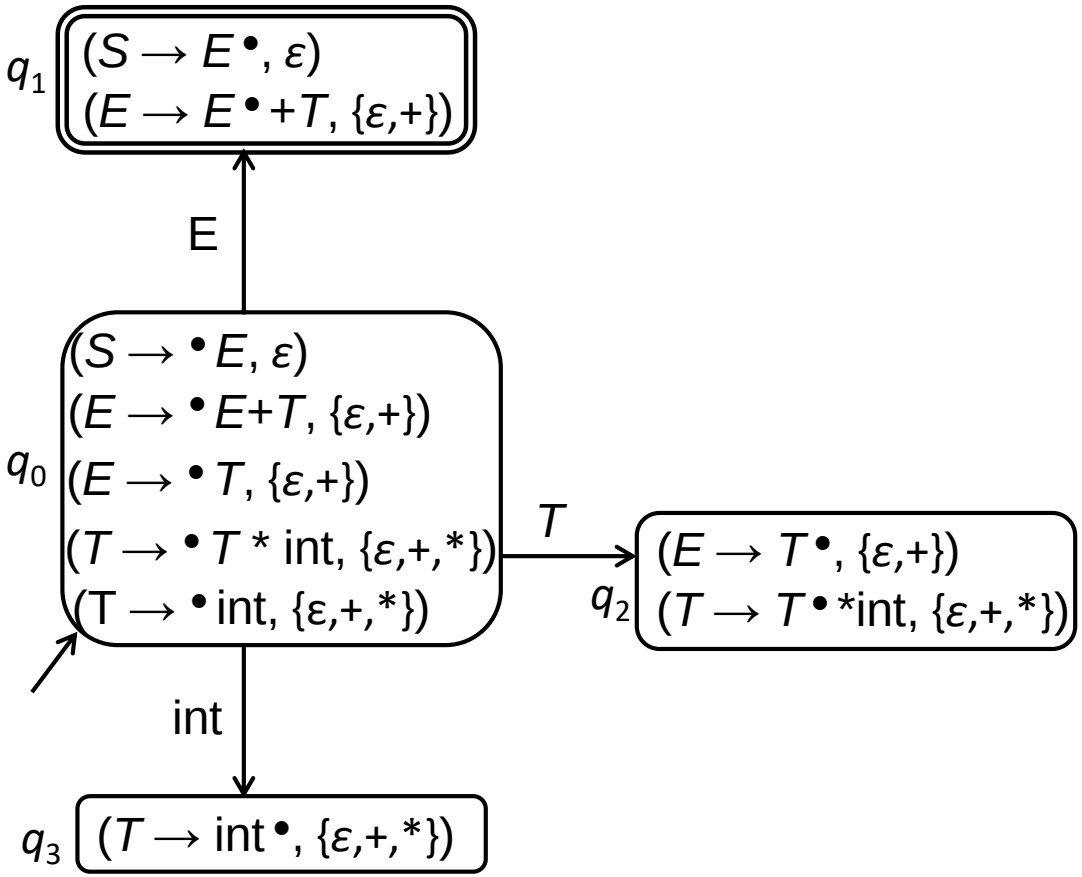


- $(A \rightarrow \alpha \bullet X \beta, u) \xrightarrow{X} (A \rightarrow \alpha X \bullet \beta, u) \quad X \in T \cup N$
- $(A \rightarrow \alpha \bullet B \beta, u) \xrightarrow{\varepsilon} (B \rightarrow \bullet \gamma, v) \quad B \in N, B \rightarrow \gamma \in P, v \in \text{First}(\beta u)$

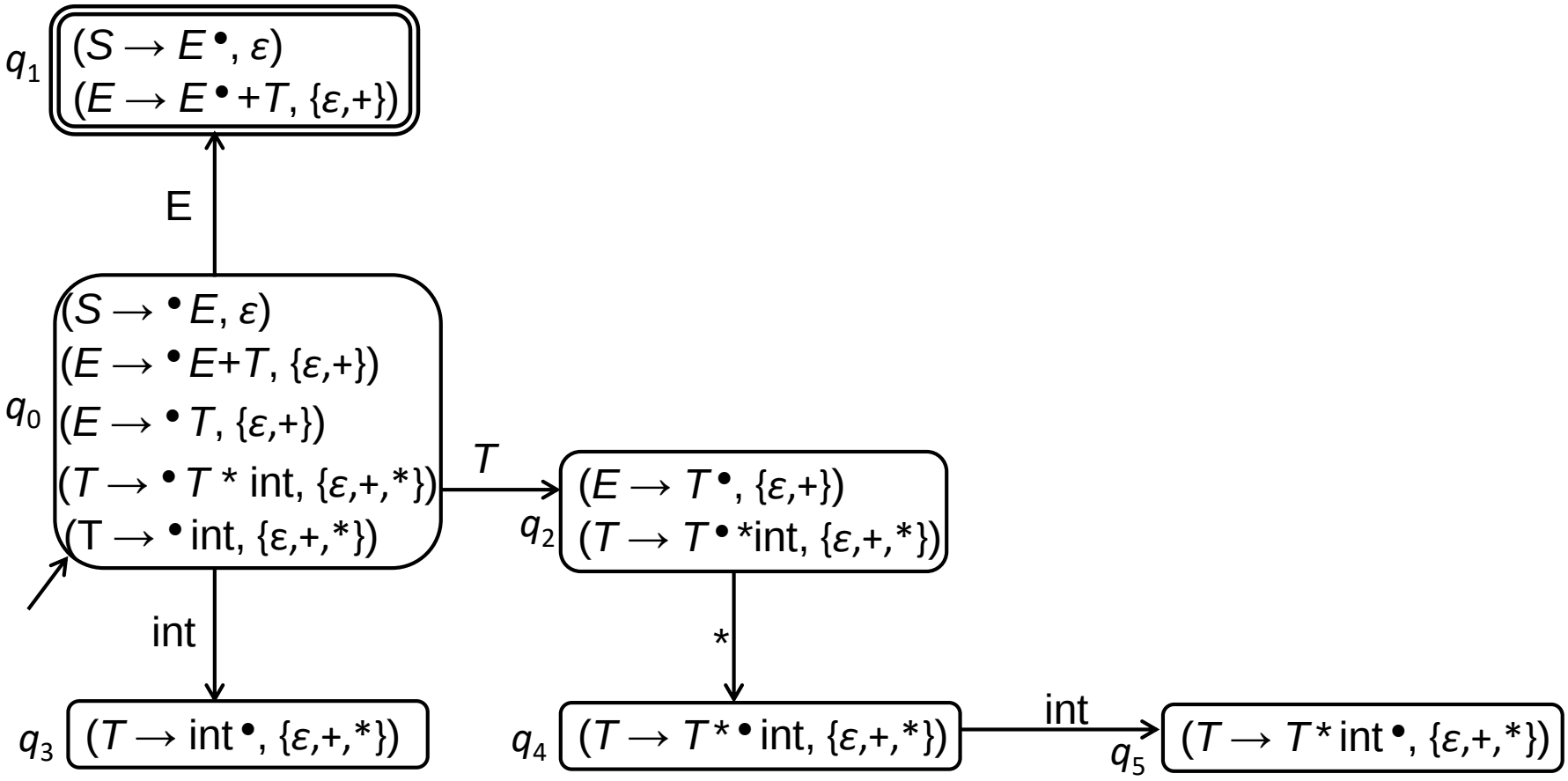
Example: LR(1) DFA Construction on the fly



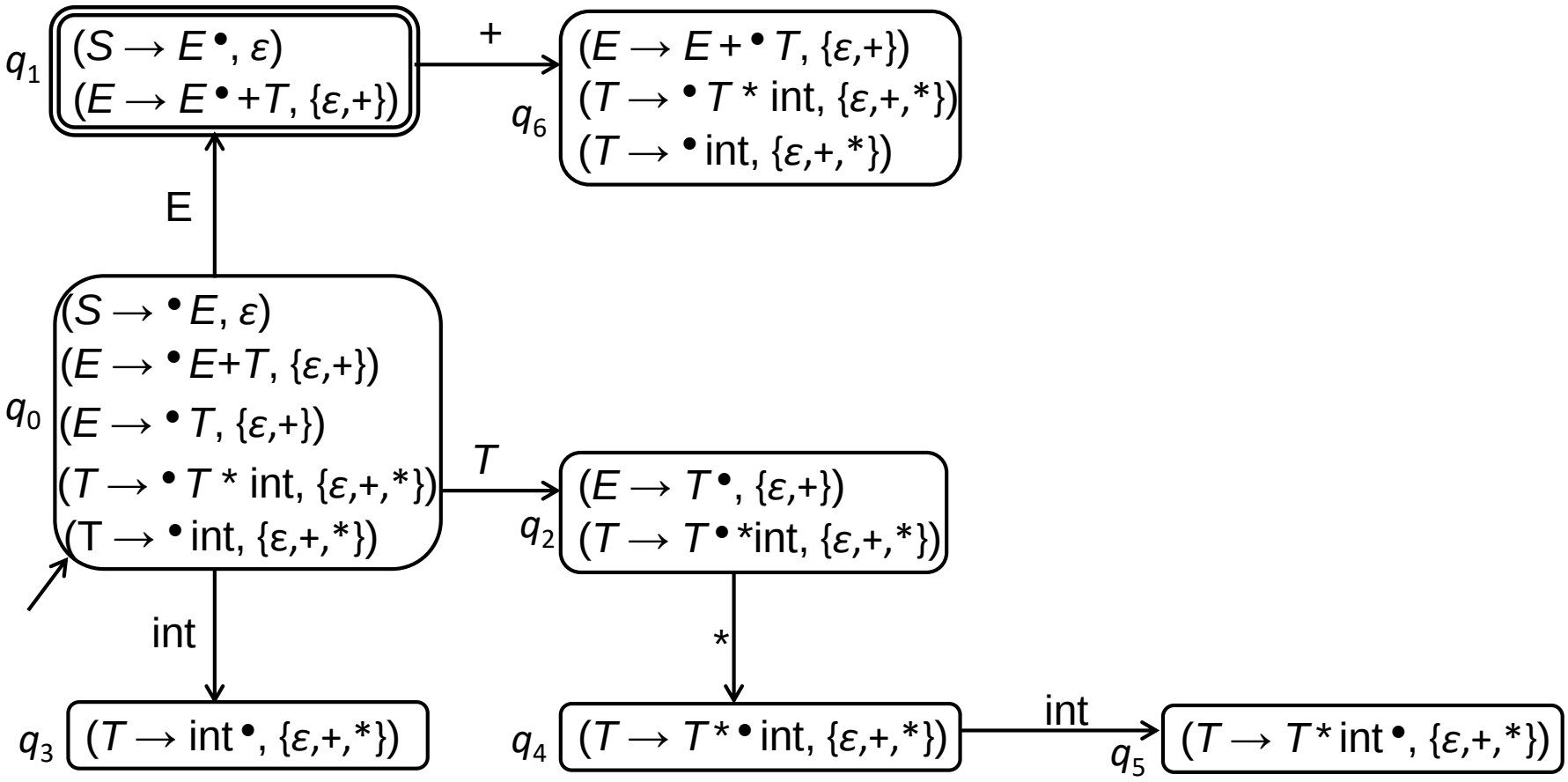
Example: LR(1) DFA Construction



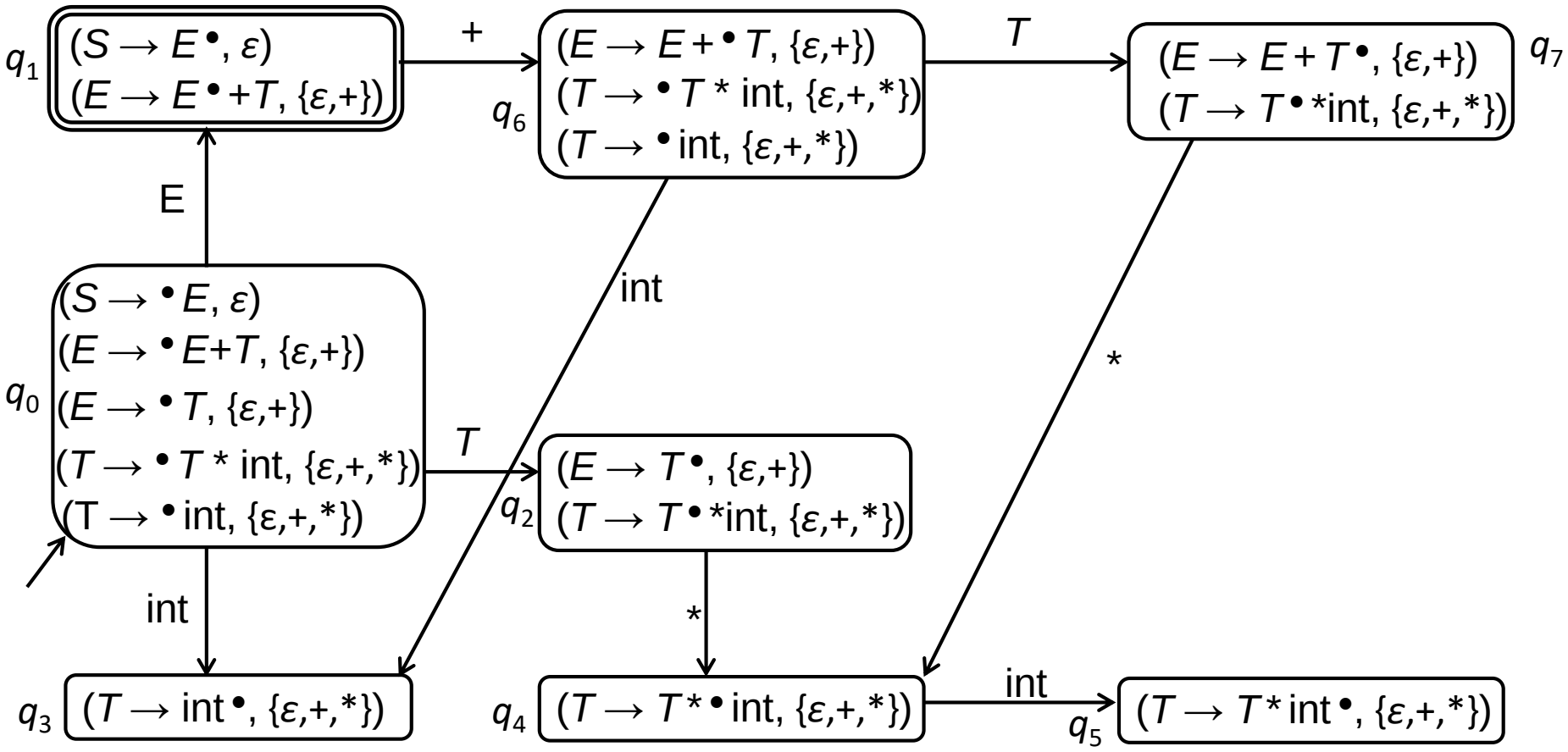
Example: LR(1) DFA Construction



Example: LR(1) DFA Construction

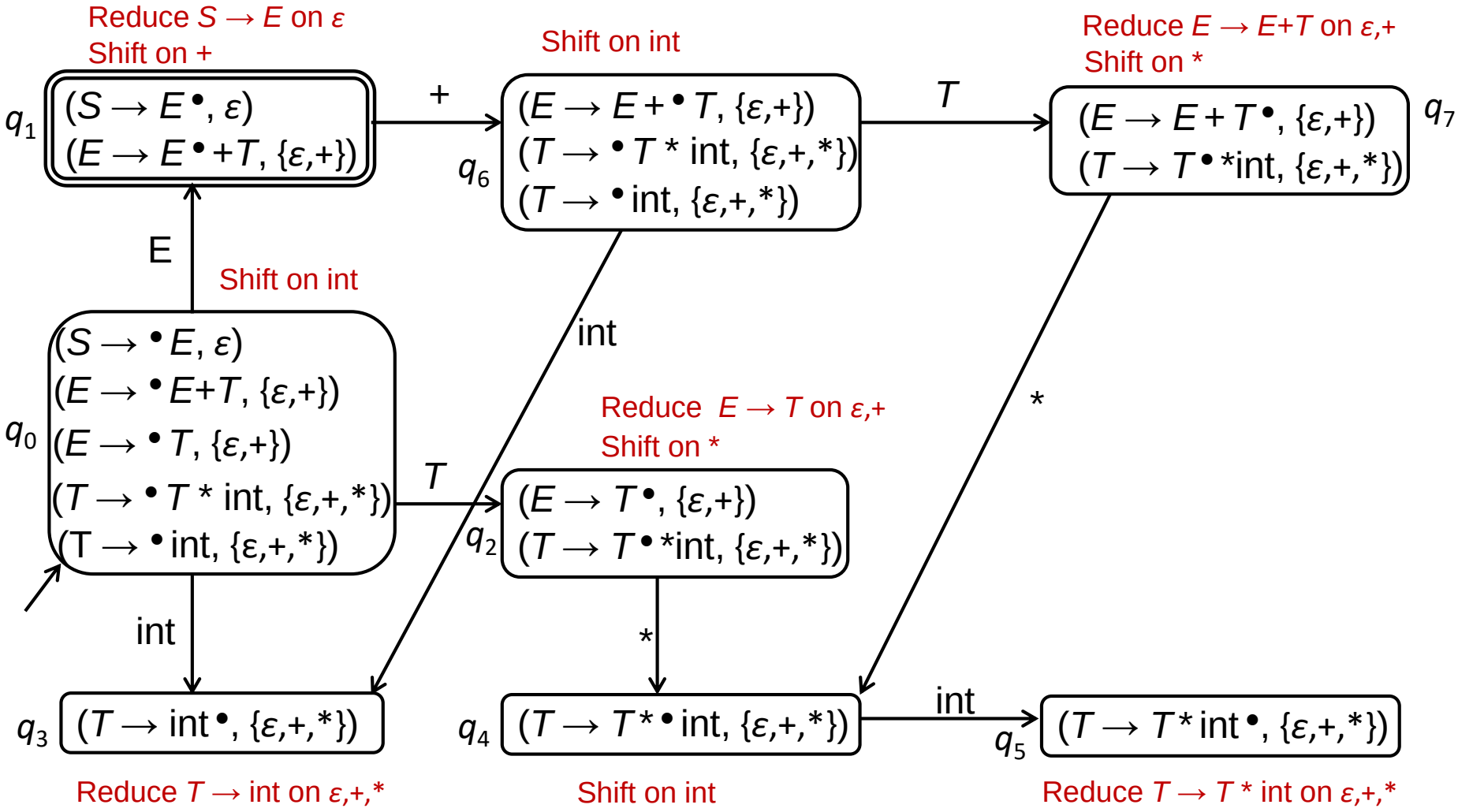


Example: LR(1) DFA Construction



LR(1) Parsing

- NFA
  - Start state ($S' \rightarrow \bullet S, \varepsilon$)
  - Accepting state ($S' \rightarrow S\bullet, \varepsilon$)
- DFA
  - NFA → DFA conversion (states = sets of $LR(1)$-items)
  - Actions associated with states and lookahead
    - $Act(q,t)$ = shift iff $(A \rightarrow \alpha \bullet t\, \beta, u) \in q,\ t \in T$
    - $Act(q,u) = (A \rightarrow \beta)$ iff $(A \rightarrow \beta \bullet, u) \in q,\ u \in T \cup \{\varepsilon\}$

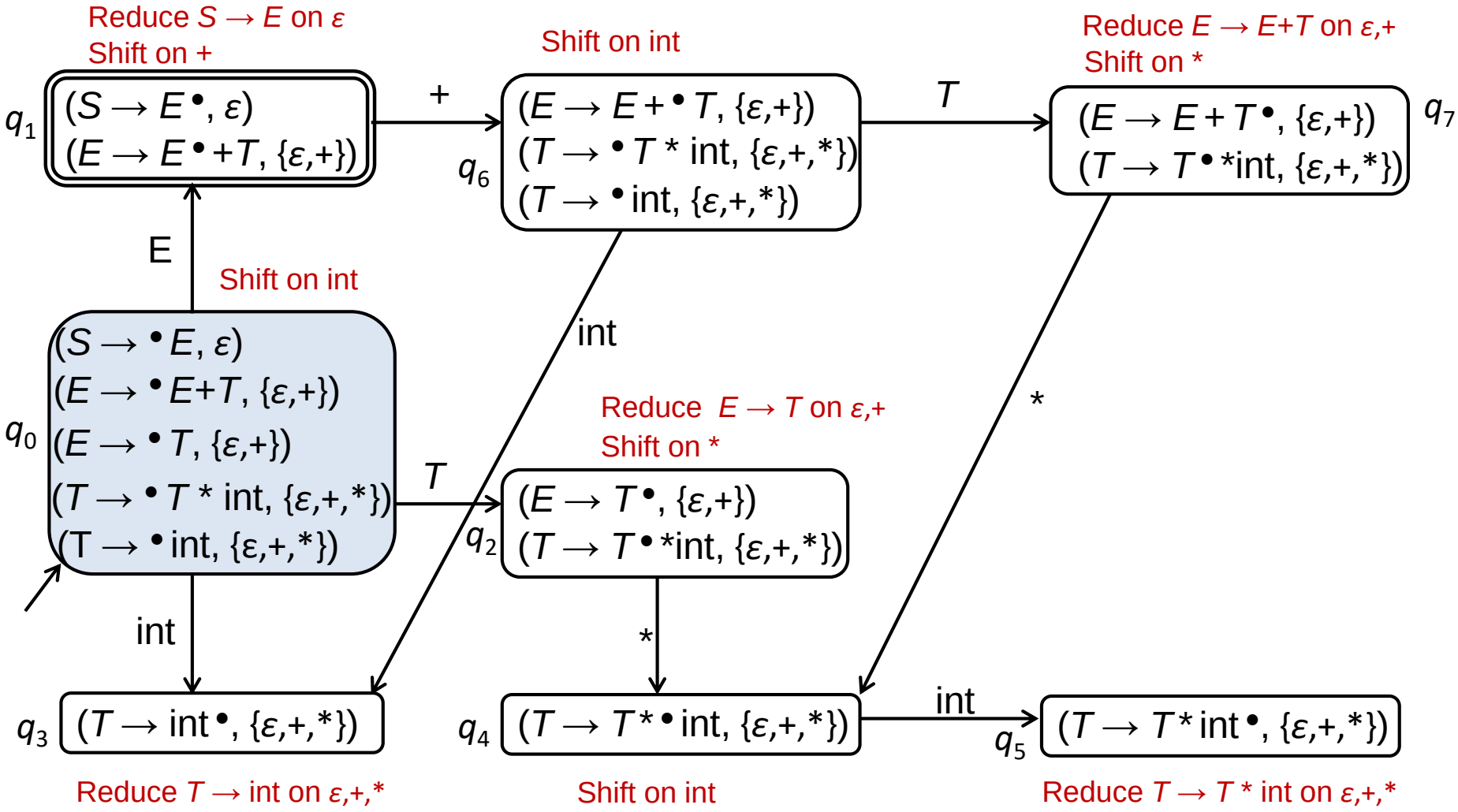

Reduce S → E on ε
Shift on +
q1
(S → E •, ε)
(E → E • + T, {ε,+})
+
Shift on int
q6
(E → E + • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
T
Reduce E → E+T on ε,+
Shift on *
q7
(E → E + T •, {ε,+})
(T → T • *int, {ε,+,*})
E
Shift on int
q0
(S → • E, ε)
(E → • E+T, {ε,+})
(E → • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
int
T
Reduce E → T on ε,+
Shift on *
q2
(E → T •, {ε,+})
(T → T • *int, {ε,+,*})
*
int
q3
(T → int •, {ε,+,*})
*
q4
(T → T * • int, {ε,+,*})
int
q5
(T → T * int •, {ε,+,*})
Reduce T → int on ε,+,*
Shift on int
Reduce T → T * int on ε,+,*

LR(1) Parsing

- Parsing algorithm
  - Stack symbols: States (Sets of LR(1)-items)
  - Initial stack is $q_0 \mid \omega_c$ where ω_c is the complete input
  - **Parse steps**
    - $\rho\, q \mid t\, \omega \quad \rightarrow \quad \rho\, q\, \delta(q,t) \mid \omega$ iff Act(q,t) = shift
    - $\rho\, q\, \rho' q' \mid \omega \quad \rightarrow \quad \rho\, q\, \delta(q,A) \mid \omega$ iff Act(q',u) = $(A \rightarrow \beta)$
      $\mid \beta \mid = \mid \rho' q' \mid$
      u = First(ω)
  - Accept with stack $q_0\, q_f \mid \varepsilon$ where q_f is accepting state

Example LR(1)-Parse

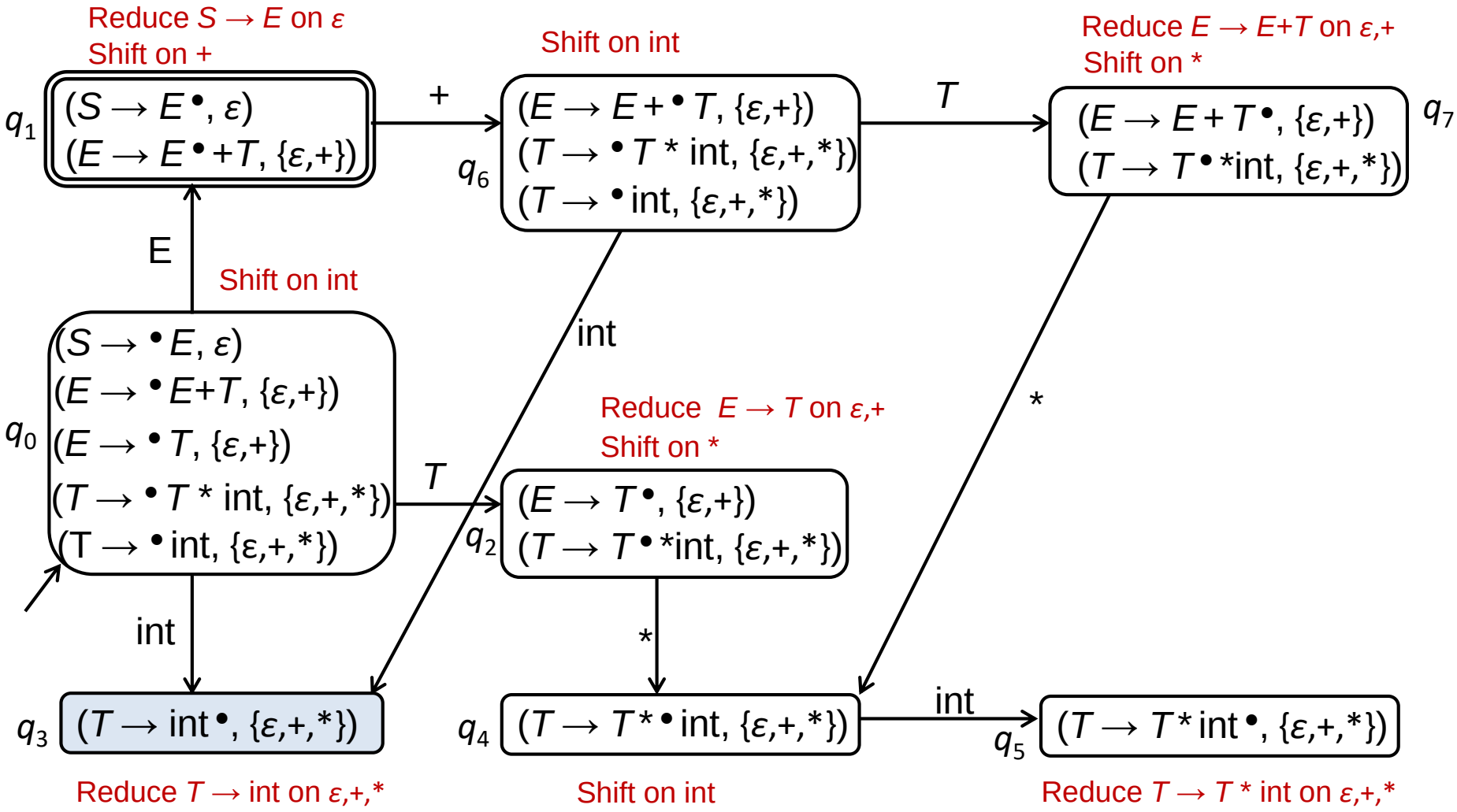
q_0 | int * int + int


Reduce S → E on ε
Shift on +
q1
(S → E •, ε)
(E → E • + T, {ε,+})
+
Shift on int
q6
(E → E + • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
T
Reduce E → E+T on ε,+
Shift on *
q7
(E → E + T •, {ε,+})
(T → T • *int, {ε,+,*})
E
Shift on int
q0
(S → • E, ε)
(E → • E+T, {ε,+})
(E → • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
int
T
Reduce E → T on ε,+
Shift on *
q2
(E → T •, {ε,+})
(T → T • *int, {ε,+,*})
*
int
q3
(T → int •, {ε,+,*})
*
q4
(T → T* • int, {ε,+,*})
int
q5
(T → T* int •, {ε,+,*})
Reduce T → int on ε,+,*
Shift on int
Reduce T → T * int on ε,+,*

Example LR(1)-Parse

q_0 | int * int + int — Shift

$q_0\ q_3$ | * int + int

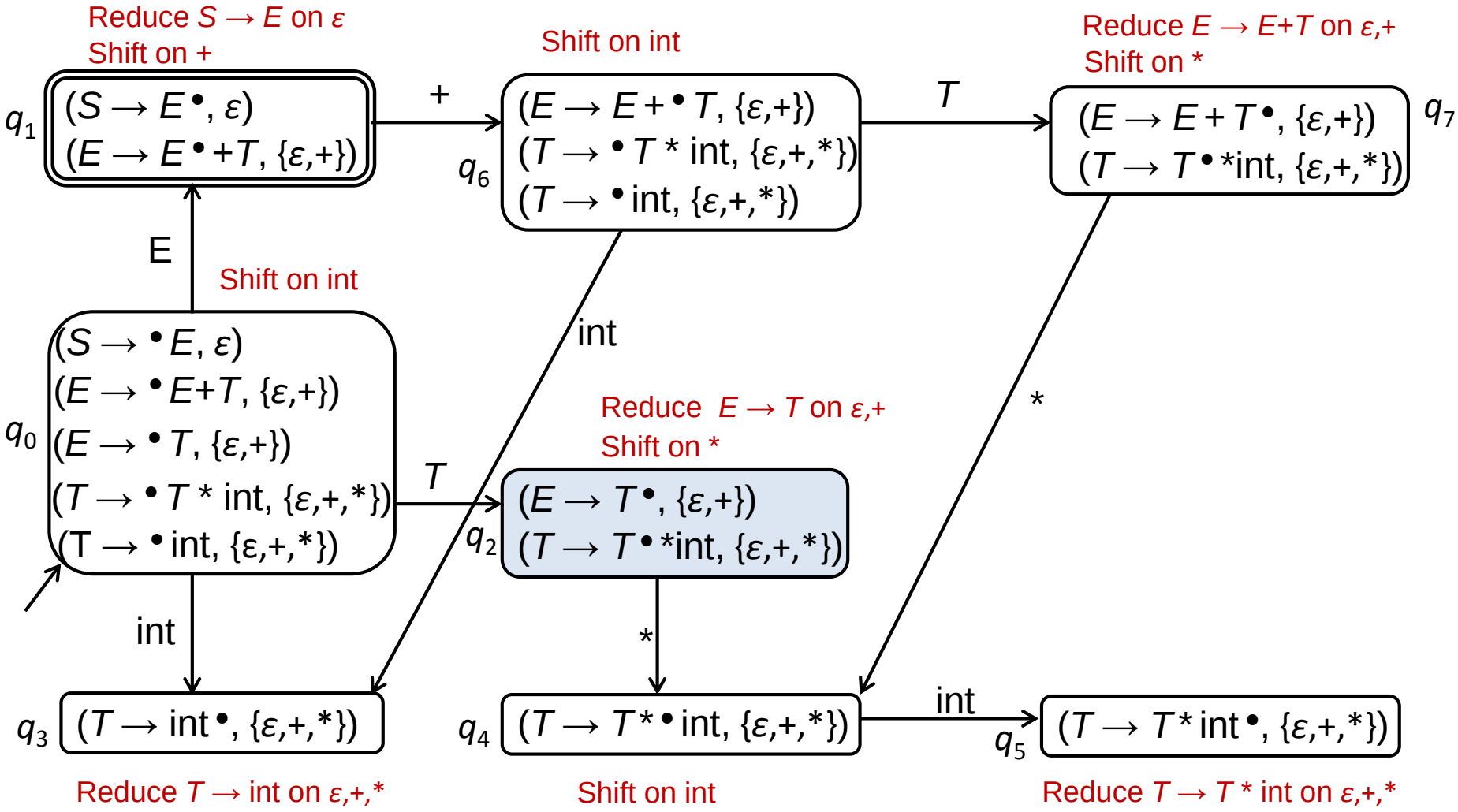

Reduce S → E on ε
Shift on +
q1
(S → E •, ε)
(E → E • + T, {ε,+})
+
Shift on int
q6
(E → E + • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
T
Reduce E → E+T on ε,+
Shift on *
q7
(E → E + T •, {ε,+})
(T → T • *int, {ε,+,*})
E
Shift on int
q0
(S → • E, ε)
(E → • E+T, {ε,+})
(E → • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
int
T
Reduce E → T on ε,+
Shift on *
q2
(E → T •, {ε,+})
(T → T • *int, {ε,+,*})
*
int
q3
(T → int •, {ε,+,*})
*
q4
(T → T * • int, {ε,+,*})
int
q5
(T → T * int •, {ε,+,*})
Reduce T → int on ε,+,*
Shift on int
Reduce T → T * int on ε,+,*

Example LR(1)-Parse

q_0 | int * int + int — Shift

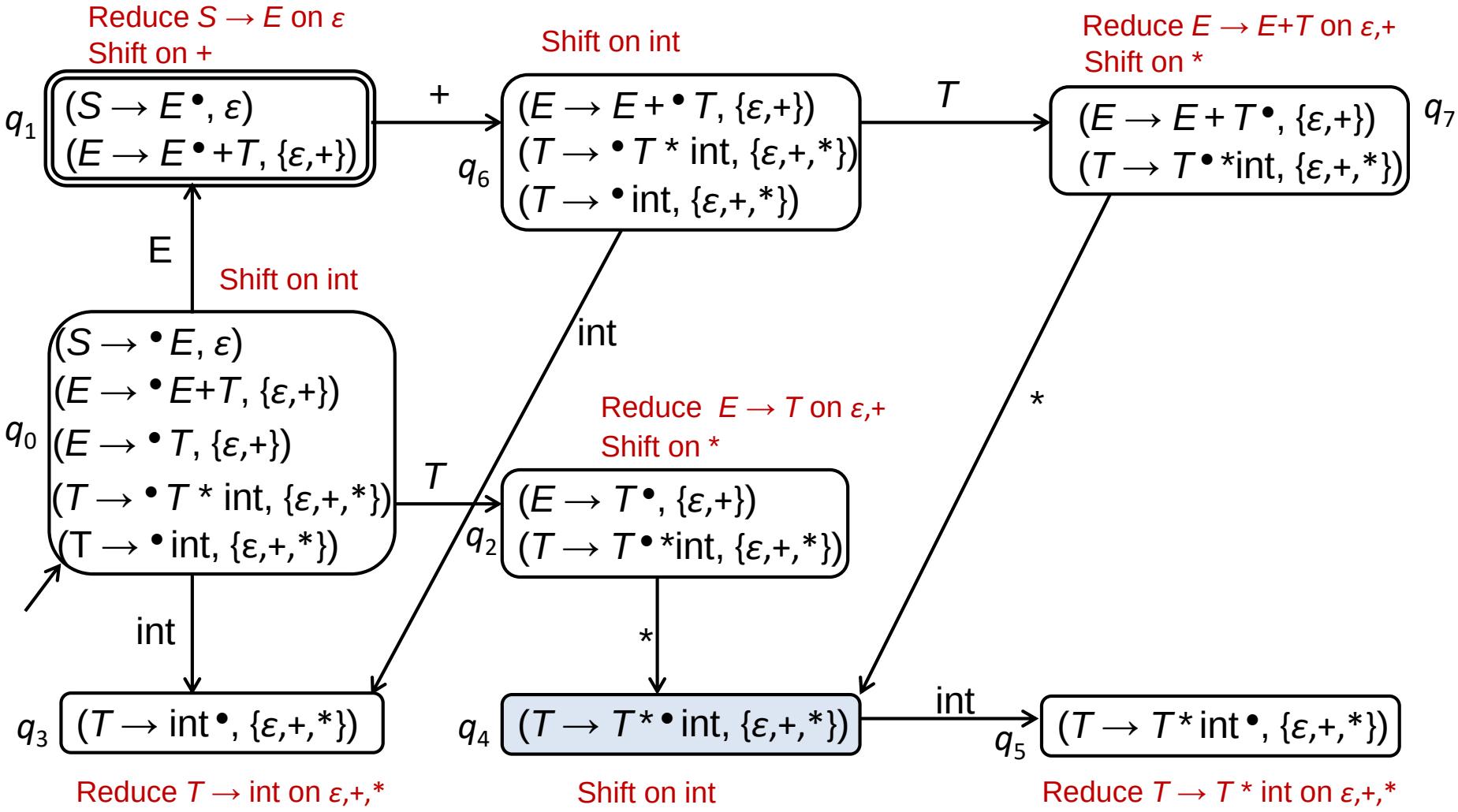
$q_0\ q_3$ | * int + int — Reduce $T \rightarrow$ int

$q_0\ q_2$ | * int + int


Reduce S → E on ε
Shift on +
q1
(S → E •, ε)
(E → E • +T, {ε,+})
+
Shift on int
q6
(E → E + • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
T
Reduce E → E+T on ε,+
Shift on *
q7
(E → E + T •, {ε,+})
(T → T • *int, {ε,+,*})
E
Shift on int
q0
(S → • E, ε)
(E → • E+T, {ε,+})
(E → • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
int
T
Reduce E → T on ε,+
Shift on *
q2
(E → T •, {ε,+})
(T → T • *int, {ε,+,*})
*
int
q3
(T → int •, {ε,+,*})
Reduce T → int on ε,+,*
*
q4
(T → T* • int, {ε,+,*})
Shift on int
int
q5
(T → T* int •, {ε,+,*})
Reduce T → T * int on ε,+,*

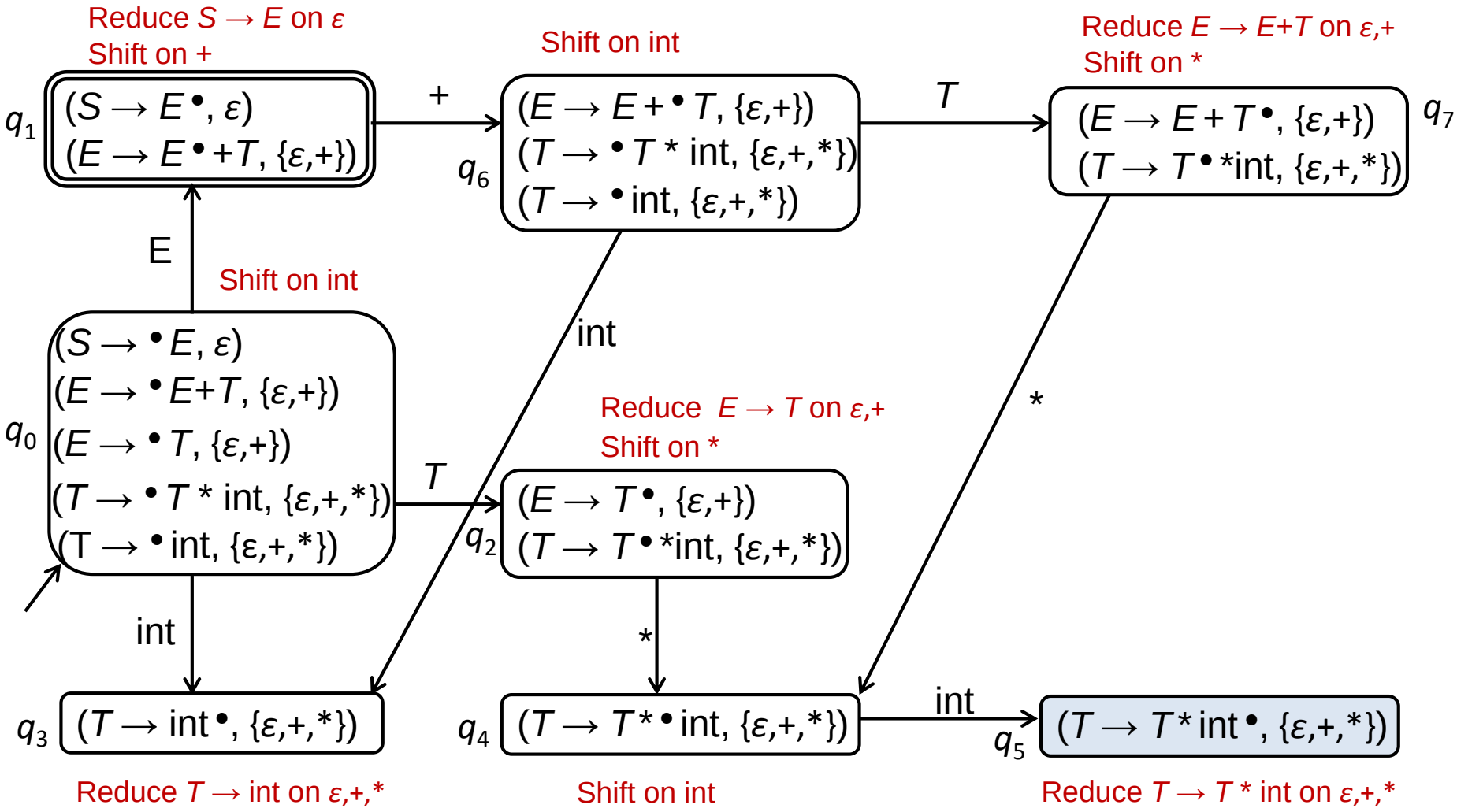
Example LR(1)-Parse

q_0 | int * int + int	Shift
$q_0\ q_3$ | * int + int	Reduce $T \rightarrow$ int
$q_0\ q_2$ | * int + int	Shift
$q_0\ q_2\ q_4$ | int + int	


Reduce S → E on ε
Shift on +
q1
(S → E •, ε)
(E → E • + T, {ε,+})
+
Shift on int
q6
(E → E + • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
T
Reduce E → E+T on ε,+
Shift on *
q7
(E → E + T •, {ε,+})
(T → T • *int, {ε,+,*})
E
Shift on int
q0
(S → • E, ε)
(E → • E+T, {ε,+})
(E → • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
int
T
Reduce E → T on ε,+
Shift on *
q2
(E → T •, {ε,+})
(T → T • *int, {ε,+,*})
*
int
q3
(T → int •, {ε,+,*})
*
q4
(T → T * • int, {ε,+,*})
int
q5
(T → T * int •, {ε,+,*})
Reduce T → int on ε,+,*
Shift on int
Reduce T → T * int on ε,+,*

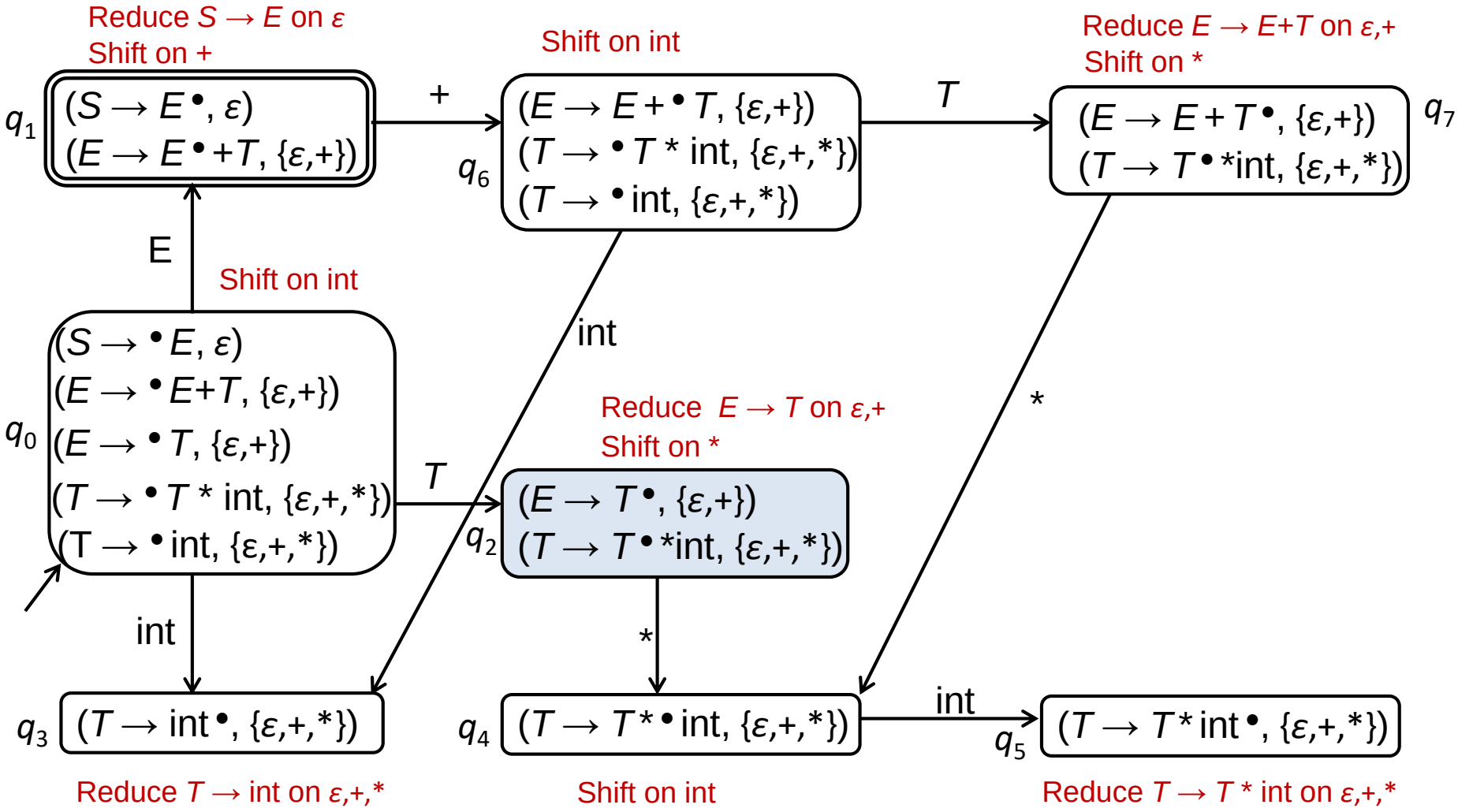
Example LR(1)-Parse

q_0 | int * int + int	Shift
$q_0\ q_3$ | * int + int	Reduce $T \rightarrow$ int
$q_0\ q_2$ | * int + int	Shift
$q_0\ q_2\ q_4$ | int + int	Shift
$q_0\ q_2\ q_4\ q_5$ | + int	


Reduce S → E on ε
Shift on +
q1
(S → E •, ε)
(E → E • + T, {ε,+})
+
Shift on int
q6
(E → E + • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
T
Reduce E → E+T on ε,+
Shift on *
q7
(E → E + T •, {ε,+})
(T → T • *int, {ε,+,*})
E
Shift on int
q0
(S → • E, ε)
(E → • E+T, {ε,+})
(E → • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
int
T
Reduce E → T on ε,+
Shift on *
q2
(E → T •, {ε,+})
(T → T • *int, {ε,+,*})
*
int
q3
(T → int •, {ε,+,*})
*
q4
(T → T * • int, {ε,+,*})
int
q5
(T → T * int •, {ε,+,*})
Reduce T → int on ε,+,*
Shift on int
Reduce T → T * int on ε,+,*

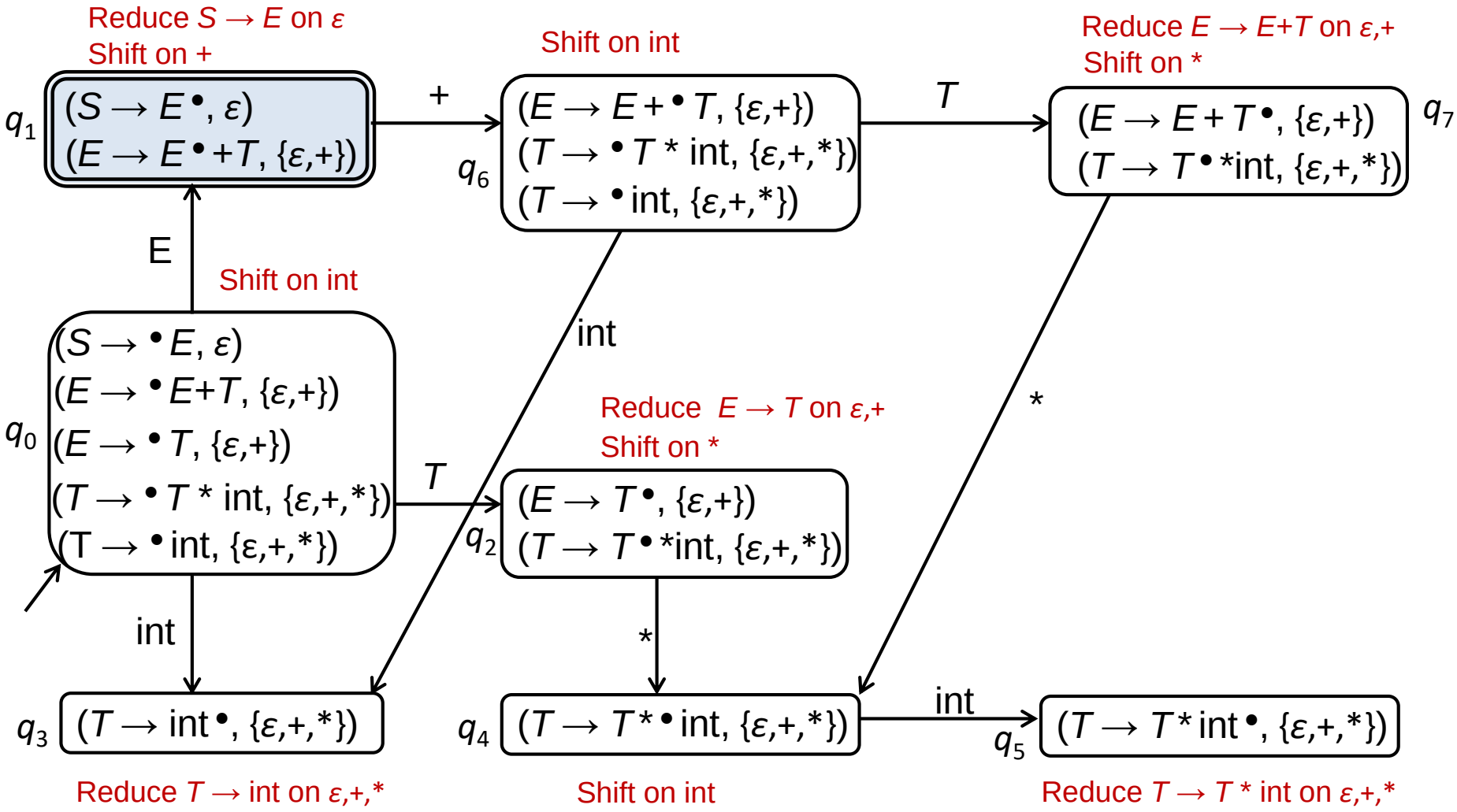
Example LR(1)-Parse

q_0 | int * int + int	Shift
$q_0\ q_3$ | * int + int	Reduce $T \rightarrow$ int
$q_0\ q_2$ | * int + int	Shift
$q_0\ q_2\ q_4$ | int + int	Shift
$q_0\ q_2\ q_4\ q_5$ | + int	Reduce $T \rightarrow T$ * int
$q_0\ q_2$ | + int	


Reduce S → E on ε
Shift on +
q1
(S → E •, ε)
(E → E • + T, {ε,+})
+
Shift on int
q6
(E → E + • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
T
Reduce E → E+T on ε,+
Shift on *
q7
(E → E + T •, {ε,+})
(T → T • *int, {ε,+,*})
E
Shift on int
q0
(S → • E, ε)
(E → • E+T, {ε,+})
(E → • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
int
T
Reduce E → T on ε,+
Shift on *
q2
(E → T •, {ε,+})
(T → T • *int, {ε,+,*})
*
int
q3
(T → int •, {ε,+,*})
*
q4
(T → T* • int, {ε,+,*})
int
q5
(T → T* int •, {ε,+,*})
Reduce T → int on ε,+,*
Shift on int
Reduce T → T * int on ε,+,*

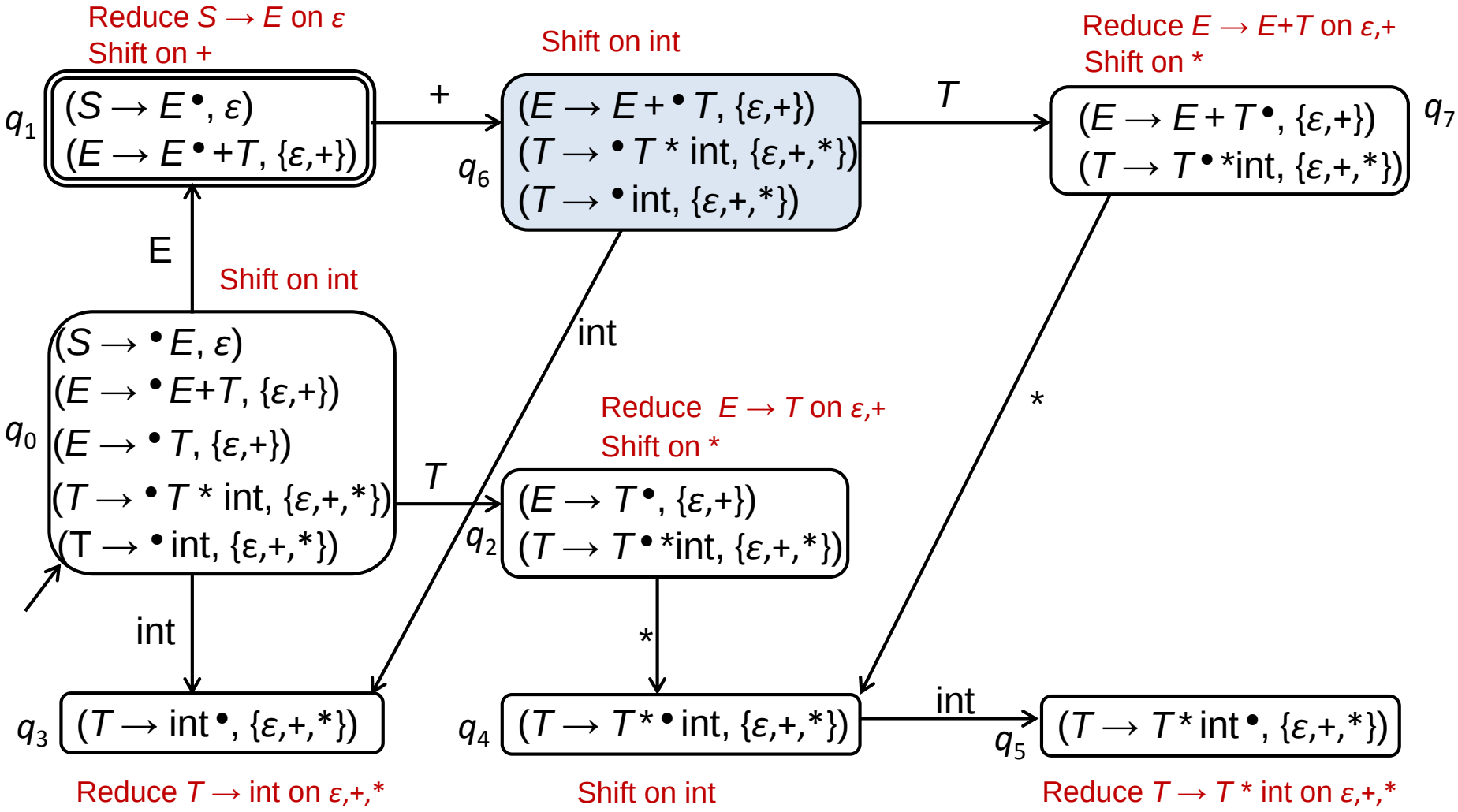
Example LR(1)-Parse

q_0 | int * int + int	Shift
$q_0\ q_3$ | * int + int	Reduce $T \rightarrow$ int
$q_0\ q_2$ | * int + int	Shift
$q_0\ q_2\ q_4$ | int + int	Shift
$q_0\ q_2\ q_4\ q_5$ | + int	Reduce $T \rightarrow T$ * int
$q_0\ q_2$ | + int	Reduce $E \rightarrow T$
$q_0\ q_1$ | + int	


Reduce S → E on ε
Shift on +
q1
(S → E •, ε)
(E → E • +T, {ε,+})
+
Shift on int
q6
(E → E + • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
T
Reduce E → E+T on ε,+
Shift on *
q7
(E → E + T •, {ε,+})
(T → T • *int, {ε,+,*})
E
Shift on int
q0
(S → • E, ε)
(E → • E+T, {ε,+})
(E → • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
int
T
Reduce E → T on ε,+
Shift on *
q2
(E → T •, {ε,+})
(T → T • *int, {ε,+,*})
*
int
q3
(T → int •, {ε,+,*})
*
q4
(T → T* • int, {ε,+,*})
int
q5
(T → T* int •, {ε,+,*})
Reduce T → int on ε,+,*
Shift on int
Reduce T → T * int on ε,+,*

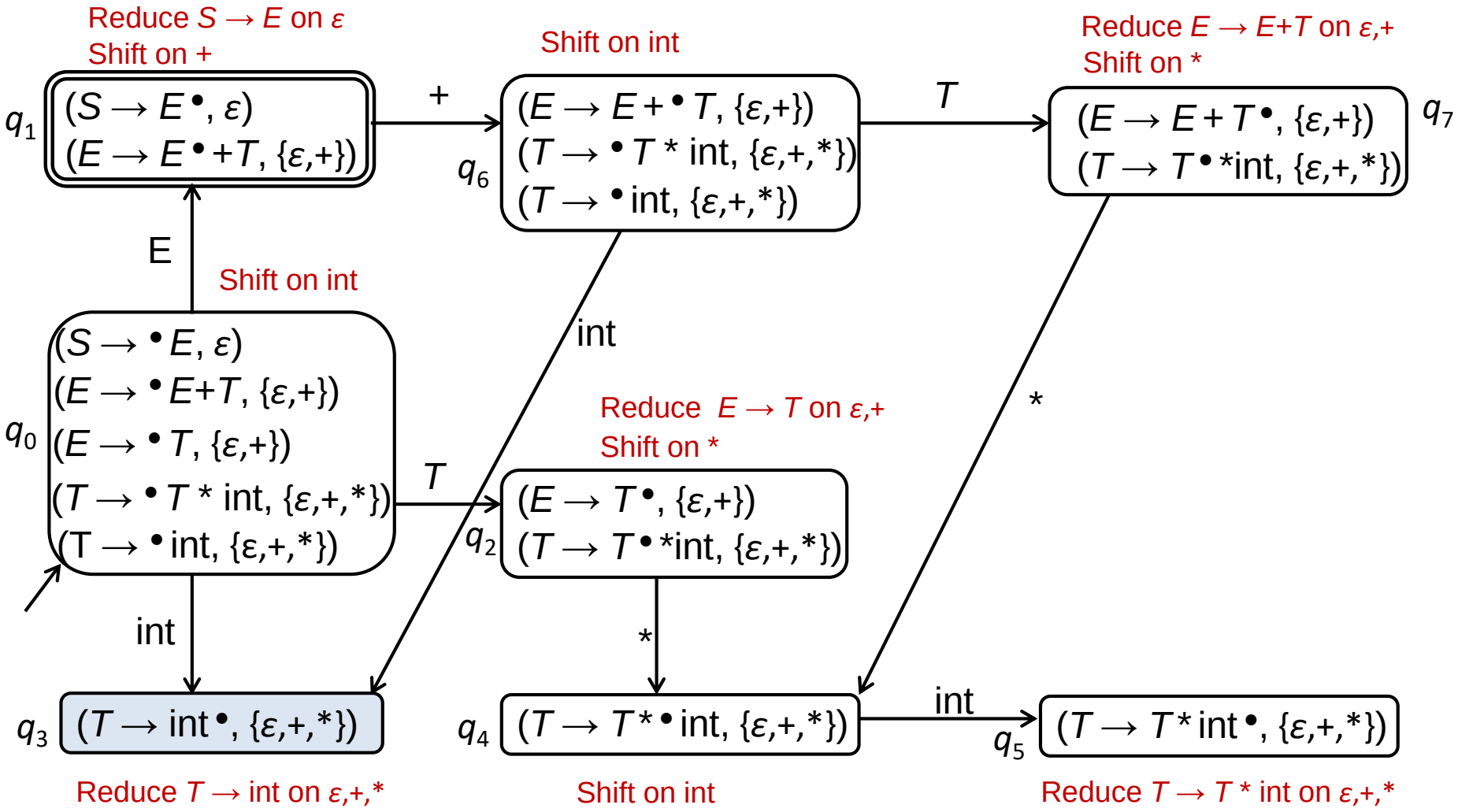
Example LR(1)-Parse

q_0 | int * int + int	Shift
$q_0\ q_3$ | * int + int	Reduce $T \rightarrow$ int
$q_0\ q_2$ | * int + int	Shift
$q_0\ q_2\ q_4$ | int + int	Shift
$q_0\ q_2\ q_4\ q_5$ | + int	Reduce $T \rightarrow T$ * int
$q_0\ q_2$ | + int	Reduce $E \rightarrow T$
$q_0\ q_1$ | + int	Shift
$q_0\ q_1\ q_6$ | int	


Reduce S → E on ε
Shift on +
q1
(S → E •, ε)
(E → E • +T, {ε,+})
+
Shift on int
q6
(E → E + • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
T
Reduce E → E+T on ε,+
Shift on *
q7
(E → E + T •, {ε,+})
(T → T • *int, {ε,+,*})
E
Shift on int
q0
(S → • E, ε)
(E → • E+T, {ε,+})
(E → • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
int
T
Reduce E → T on ε,+
Shift on *
q2
(E → T •, {ε,+})
(T → T • *int, {ε,+,*})
*
int
q3
(T → int •, {ε,+,*})
*
q4
(T → T* • int, {ε,+,*})
int
q5
(T → T* int •, {ε,+,*})
Reduce T → int on ε,+,*
Shift on int
Reduce T → T * int on ε,+,*

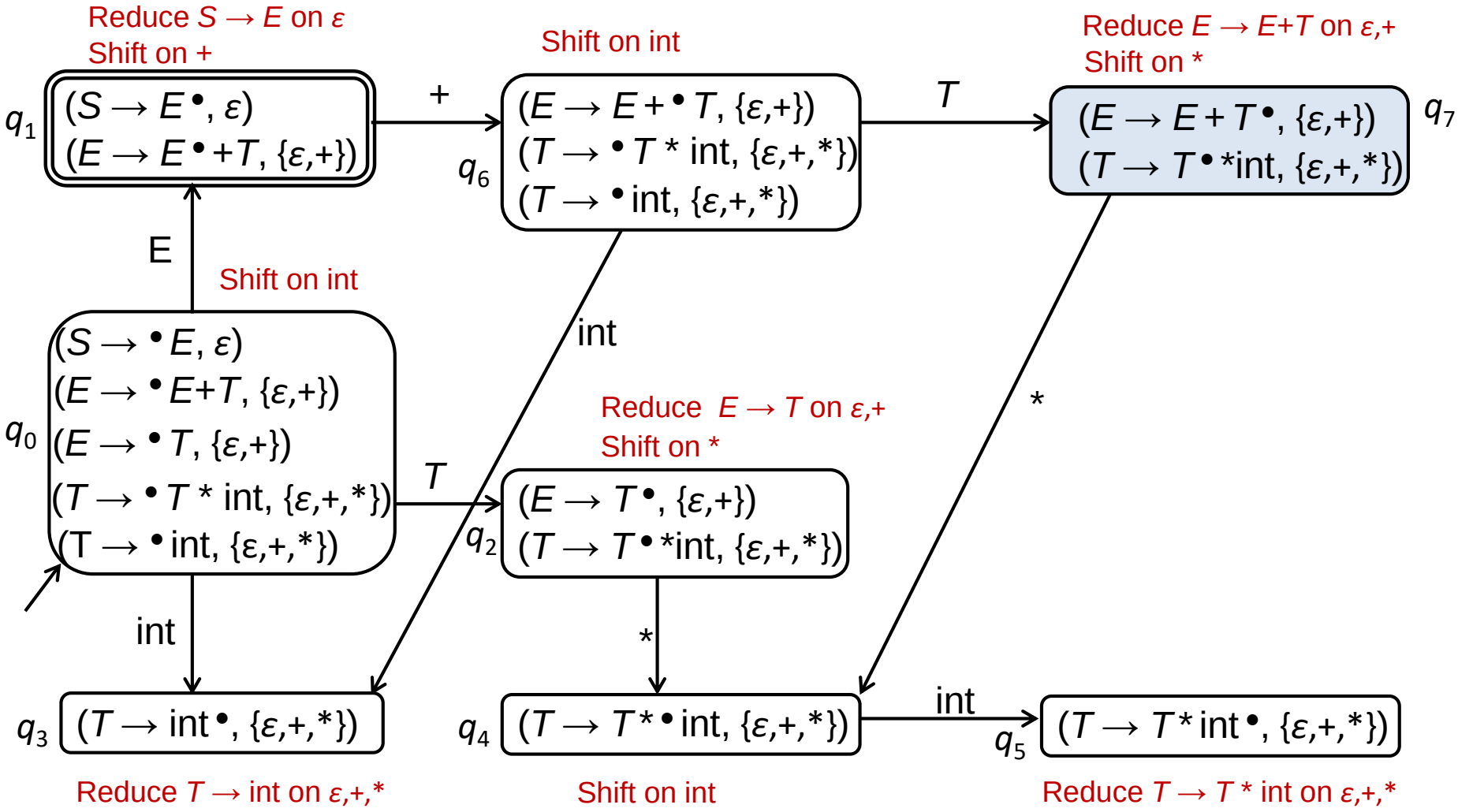
Example LR(1)-Parse

q_0 | int * int + int	Shift
$q_0\ q_3$ | * int + int	Reduce $T \rightarrow$ int
$q_0\ q_2$ | * int + int	Shift
$q_0\ q_2\ q_4$ | int + int	Shift
$q_0\ q_2\ q_4\ q_5$ | + int	Reduce $T \rightarrow T$ * int
$q_0\ q_2$ | + int	Reduce $E \rightarrow T$
$q_0\ q_1$ | + int	Shift
$q_0\ q_1\ q_6$ | int	Shift
$q_0\ q_1\ q_6\ q_3$ |	


Reduce S → E on ε
Shift on +
q1
(S → E •, ε)
(E → E • + T, {ε,+})
+
Shift on int
q6
(E → E + • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
T
Reduce E → E+T on ε,+
Shift on *
q7
(E → E + T •, {ε,+})
(T → T • *int, {ε,+,*})
E
Shift on int
q0
(S → • E, ε)
(E → • E+T, {ε,+})
(E → • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
int
T
Reduce E → T on ε,+
Shift on *
q2
(E → T •, {ε,+})
(T → T • *int, {ε,+,*})
*
int
q3
(T → int •, {ε,+,*})
*
q4
(T → T * • int, {ε,+,*})
int
q5
(T → T * int •, {ε,+,*})
Reduce T → int on ε,+,*
Shift on int
Reduce T → T * int on ε,+,*

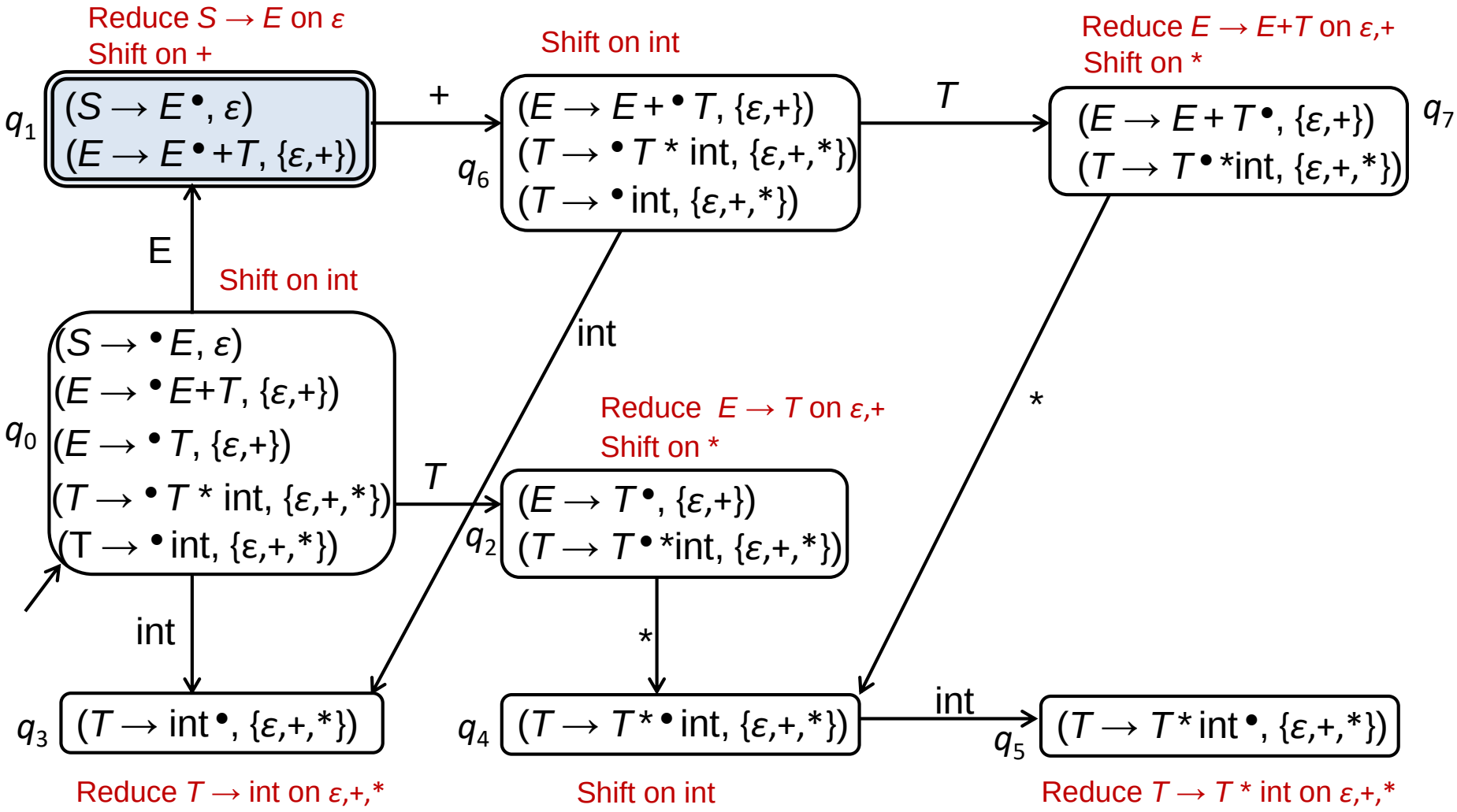
Example LR(1)-Parse

q_0 | int * int + int	Shift
$q_0\ q_3$ | * int + int	Reduce $T \rightarrow$ int
$q_0\ q_2$ | * int + int	Shift
$q_0\ q_2\ q_4$ | int + int	Shift
$q_0\ q_2\ q_4\ q_5$ | + int	Reduce $T \rightarrow T$ * int
$q_0\ q_2$ | + int	Reduce $E \rightarrow T$
$q_0\ q_1$ | + int	Shift
$q_0\ q_1\ q_6$ | int	Shift
$q_0\ q_1\ q_6\ q_3$ |	Reduce $T \rightarrow$ int
$q_0\ q_1\ q_6\ q_7$ |	


Reduce S → E on ε
Shift on +
q1
(S → E •, ε)
(E → E • + T, {ε,+})
+
Shift on int
q6
(E → E + • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
T
Reduce E → E+T on ε,+
Shift on *
q7
(E → E + T •, {ε,+})
(T → T • *int, {ε,+,*})
E
Shift on int
q0
(S → • E, ε)
(E → • E+T, {ε,+})
(E → • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
int
T
Reduce E → T on ε,+
Shift on *
q2
(E → T •, {ε,+})
(T → T • *int, {ε,+,*})
*
int
q3
(T → int •, {ε,+,*})
*
q4
(T → T * • int, {ε,+,*})
int
q5
(T → T * int •, {ε,+,*})
Reduce T → int on ε,+,*
Shift on int
Reduce T → T * int on ε,+,*

Example LR(1)-Parse

q_0 | int * int + int	Shift
$q_0\ q_3$ | * int + int	Reduce $T \rightarrow$ int
$q_0\ q_2$ | * int + int	Shift
$q_0\ q_2\ q_4$ | int + int	Shift
$q_0\ q_2\ q_4\ q_5$ | + int	Reduce $T \rightarrow T$ * int
$q_0\ q_2$ | + int	Reduce $E \rightarrow T$
$q_0\ q_1$ | + int	Shift
$q_0\ q_1\ q_6$ | int	Shift
$q_0\ q_1\ q_6\ q_3$ |	Reduce $T \rightarrow$ int
$q_0\ q_1\ q_6\ q_7$ |	Reduce $E \rightarrow E + T$
$q_0\ q_1$ |	


Reduce S → E on ε
Shift on +
q1
(S → E •, ε)
(E → E • + T, {ε,+})
+
Shift on int
q6
(E → E + • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
T
Reduce E → E+T on ε,+
Shift on *
q7
(E → E + T •, {ε,+})
(T → T • *int, {ε,+,*})
E
Shift on int
q0
(S → • E, ε)
(E → • E+T, {ε,+})
(E → • T, {ε,+})
(T → • T * int, {ε,+,*})
(T → • int, {ε,+,*})
int
T
Reduce E → T on ε,+
Shift on *
q2
(E → T •, {ε,+})
(T → T • *int, {ε,+,*})
*
int
q3
(T → int •, {ε,+,*})
*
q4
(T → T* • int, {ε,+,*})
int
q5
(T → T* int •, {ε,+,*})
Reduce T → int on ε,+,*
Shift on int
Reduce T → T * int on ε,+,*

Example LR(1)-Parse

q_0 | int * int + int	Shift
$q_0\ q_3$ | * int + int	Reduce $T \rightarrow$ int
$q_0\ q_2$ | * int + int	Shift
$q_0\ q_2\ q_4$ | int + int	Shift
$q_0\ q_2\ q_4\ q_5$ | + int	Reduce $T \rightarrow T$ * int
$q_0\ q_2$ | + int	Reduce $E \rightarrow$ T
$q_0\ q_1$ | + int	Shift
$q_0\ q_1\ q_6$ | int	Shift
$q_0\ q_1\ q_6\ q_3$ |	Reduce $T \rightarrow$ int
$q_0\ q_1\ q_6\ q_7$ |	Reduce $E \rightarrow E + T$
$q_0\ q_1$ |	**Accept**

LR Parsing

- Can be extended by increasing lookahead
  - $k > 1$ is not relevant in practice
  - LR(k) = LR(1) = Det. CFL (for languages!)
- Can be simplified by disregarding the Follow-Sets
  - LR-items have the form ($A \rightarrow \alpha \bullet \beta$)
  - Actions must not depend on lookahead
  - LR(0) = Prefix-Free Det. CFL (for languages!)

LR Parsing (2)

- Can be simplified by reducing state space
  - LALR(1): Collapse LR(1) states that only differ in lookahead components (*unify those* Follow-*sets*)
  - SLR(1): Compute LR(0) states and add lookahead information by using Follow-construction (*ignoring the context from former steps*).
  - LR(0) $\subset$ SLR(1) $\subset$ LALR(1) $\subset$ LR(1)
- Lots of Tools: E.g. Yacc (Bison), CUP, LPG, …