Bottom-Up Parsing: Algorithms, part 1

LR(0),

SLR

Lecture 12

Reading Assignment

- Reading Assignment
  - Bottom-Up Parsing: Sections 6.1 – 6.4
  - Also, you may want to read about Top-down parsing in Chapter 5
    - Sections 5.8 and 5.9 are optional

Overview

- LR(k) parsing
  - L: scan input Left to right
  - R: produce rightmost derivation
  - k tokens of lookahead
- LR(0)
  - zero tokens of look-ahead
  - “What? No lookahead?” See clarification in Section 6.2.5
- SLR
  - Simple LR: like LR(0) but uses FOLLOW sets to build more “precise” parsing tables
  - LR(0) is a toy, so we are going to use SLR in this lecture

Problem: when to shift, when to reduce?

- Recall our favorite grammar:

  $E \rightarrow T + E \mid T$

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

- The step

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

  is not part of any rightmost derivation

- Hence, reducing first int to T was a mistake
- ***how to know when to reduce and when to shift?***

What we need to know to do LR parsing

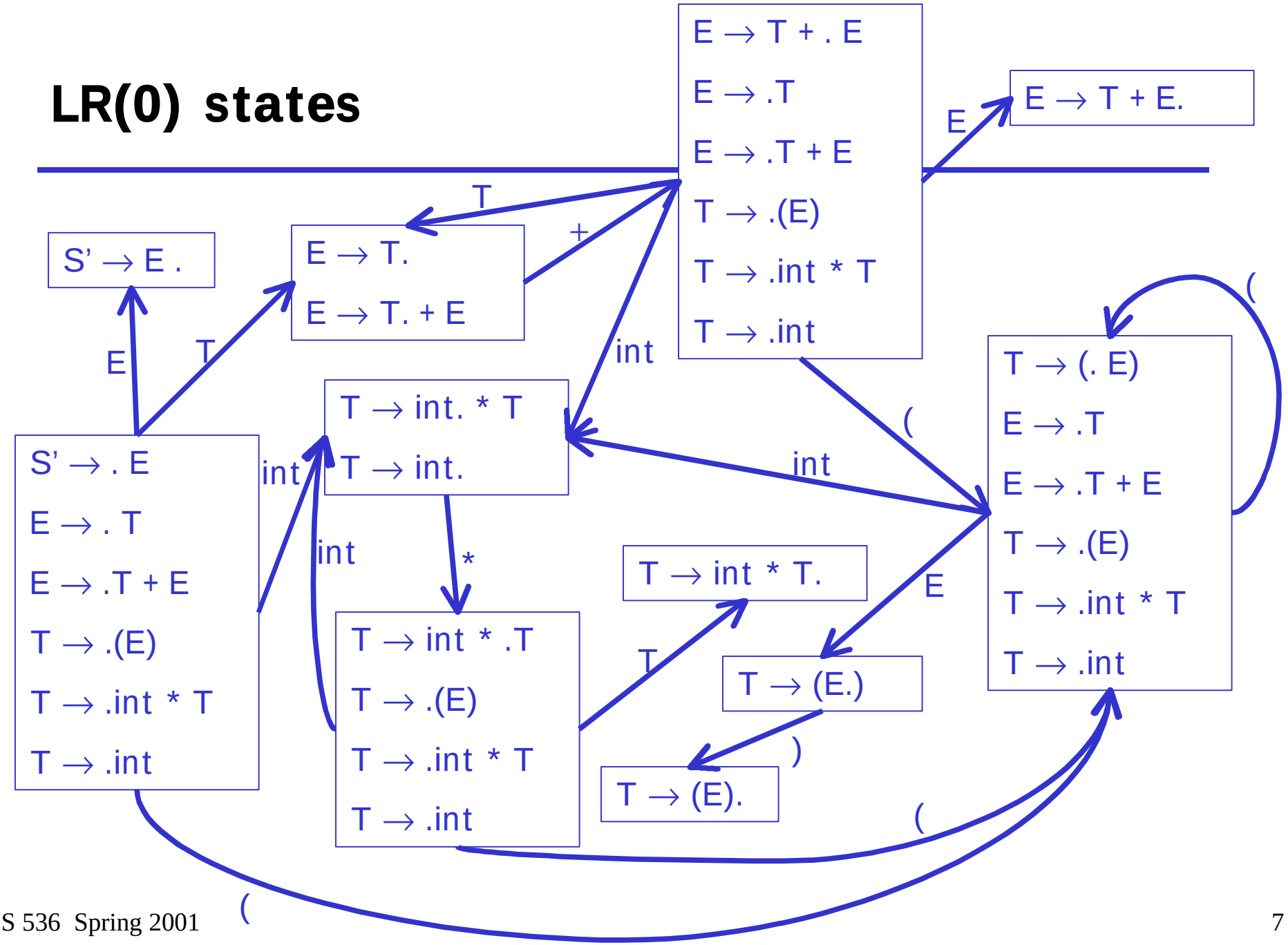
- LR(0) states
  - describe states in which the parser can be
  - Note: LR(0) states are used by both LR(0) and SLR parsers
- Parsing tables
  - transitions between LR(0) states,
  - actions to take when transiting:
    - shift, reduce, accept, error

- How to construct LR(0) states
- How to construct parsing tables
- How to drive the parser

An LR(0) state = a set of LR(0) items

- An LR(0) item $[X \rightarrow \alpha.\beta]$ says that
  - the parser is looking for an X
  - it has an α on top of the stack
  - expects to find in the input a string derived from β.
- Notes:
  - $[X \rightarrow \alpha.a\beta]$ means that if a is on the input, it can be shifted. That is:
    - *a* is a correct token to see on the input, and
    - shifting *a* would not "over-shift" (still a viable prefix).
  - $[X \rightarrow \alpha.]$ means that we could reduce X

LR(0) states


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

Naïve SLR Parsing Algorithm

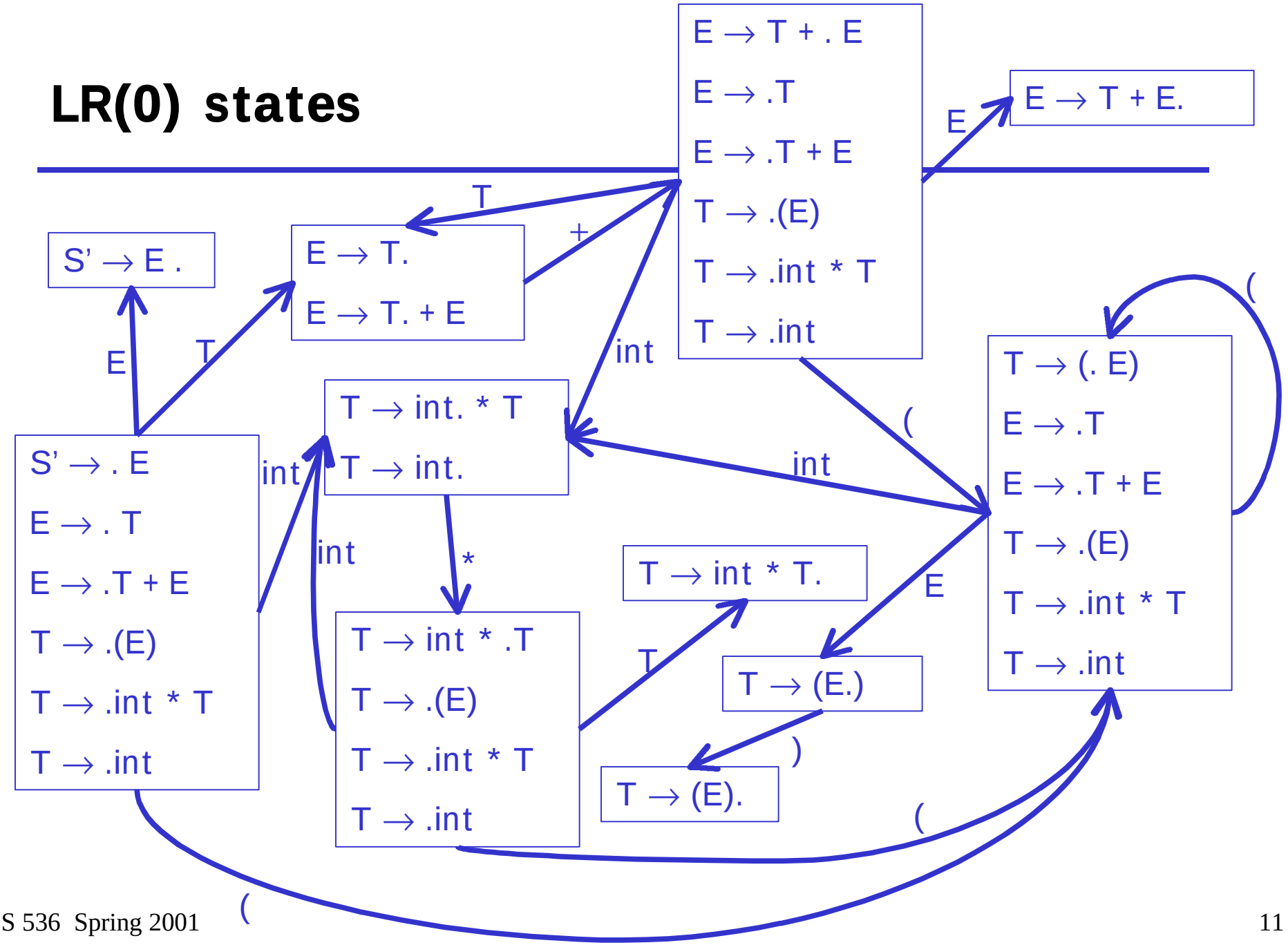
1. Let M be LR(0) state machine for G
   - each state contains a set I of LR(0) items
2. Let $|x_1 \ldots x_n\$$ be initial configuration
3. Repeat until configuration is $S|\$$
   - Let $\alpha|\omega$ be current configuration
   - Run M on current stack α
   - If M rejects α, report parsing error
   - If M accepts α with items I, let a be next input
     - Shift if $X \rightarrow \beta.\, a\, \gamma \in I$
     - Reduce if $X \rightarrow \beta.\, \in I$ and $a \in \text{Follow}(\alpha)$
       $\ldots \beta \mid a \ldots \rightarrow \ldots \mid X\, a \ldots$
     - 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

$int_1 * int_2$

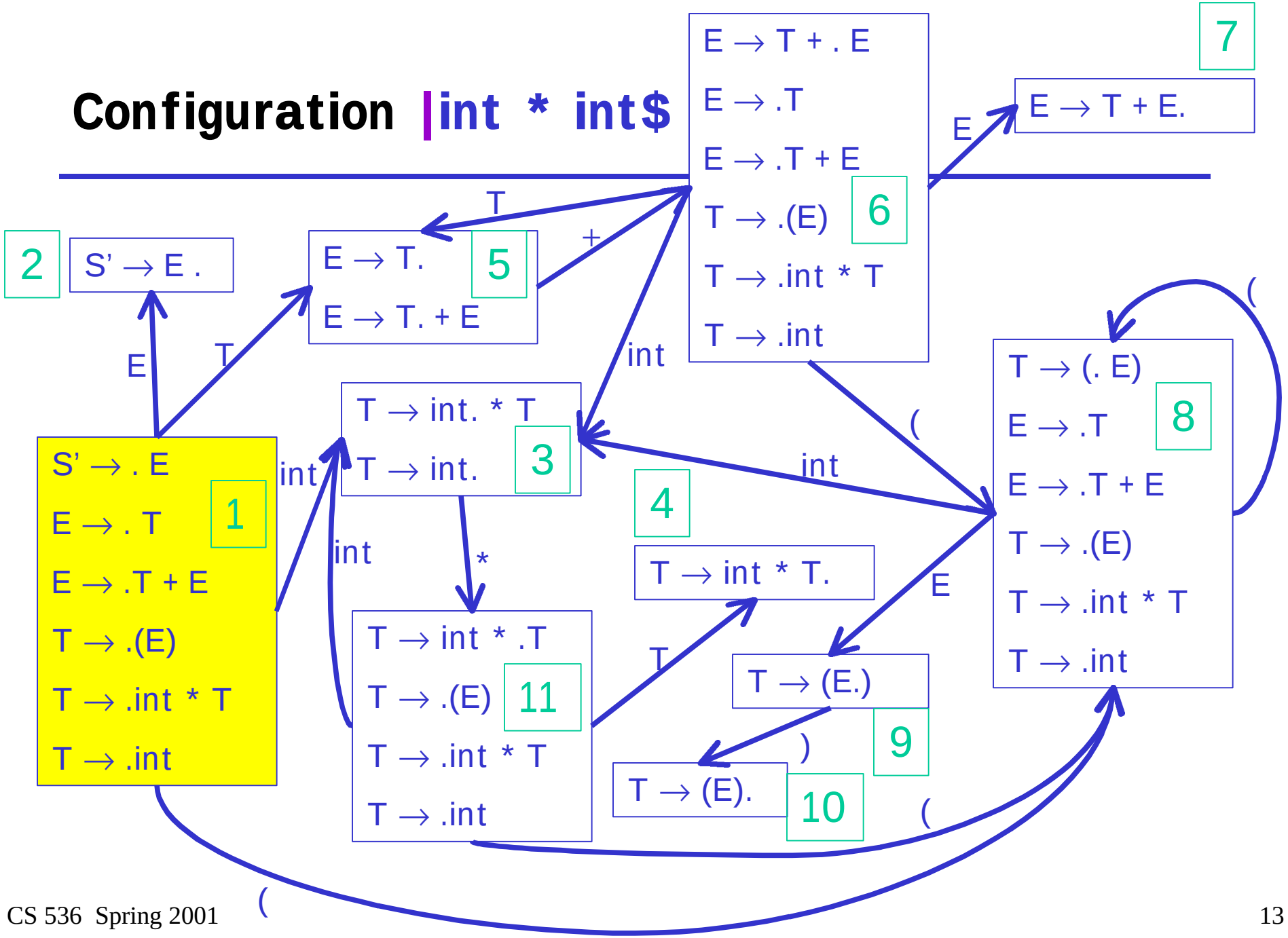
LR(0) states


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

SLR Example

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

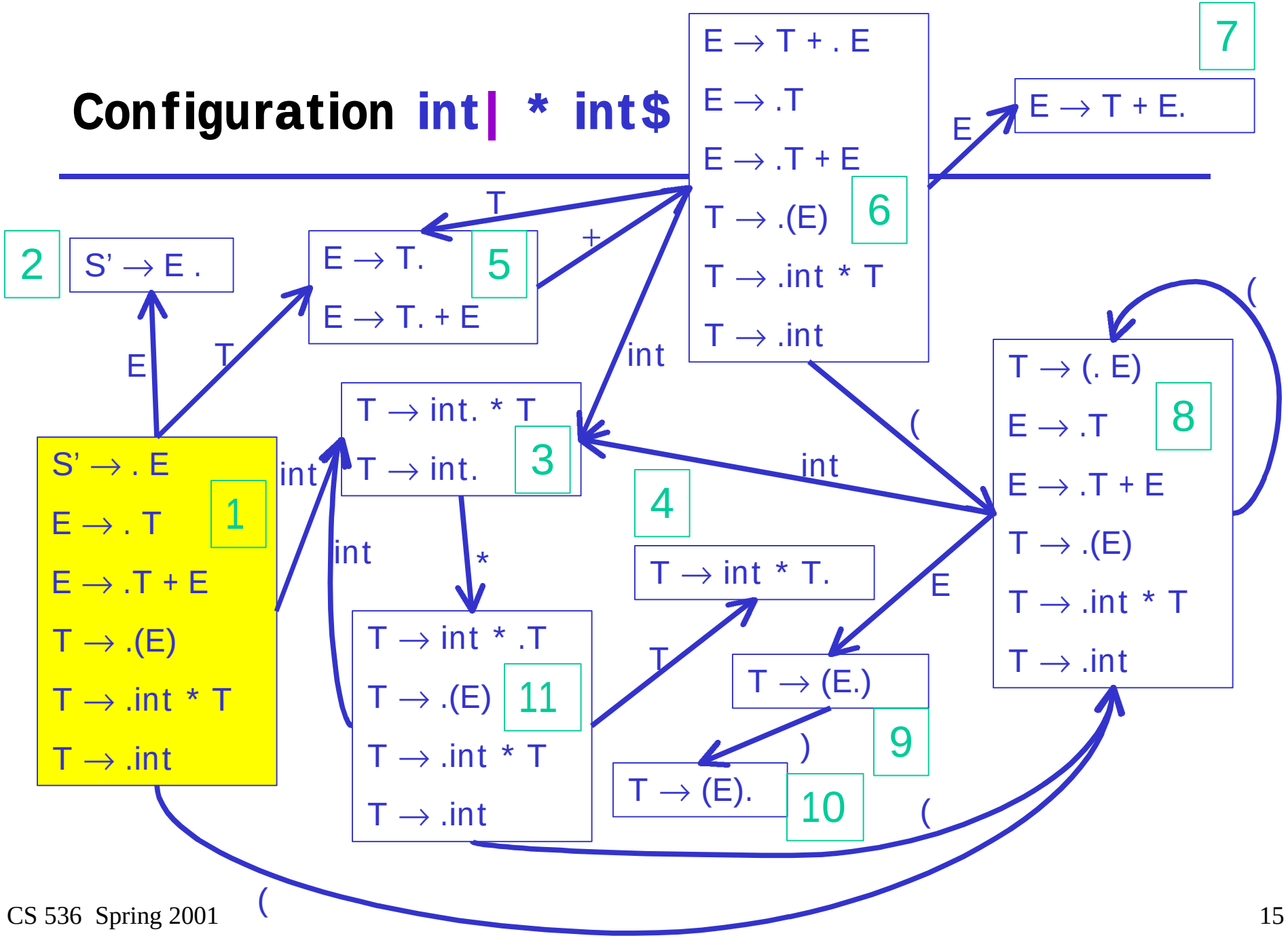
Configuration |int * int$

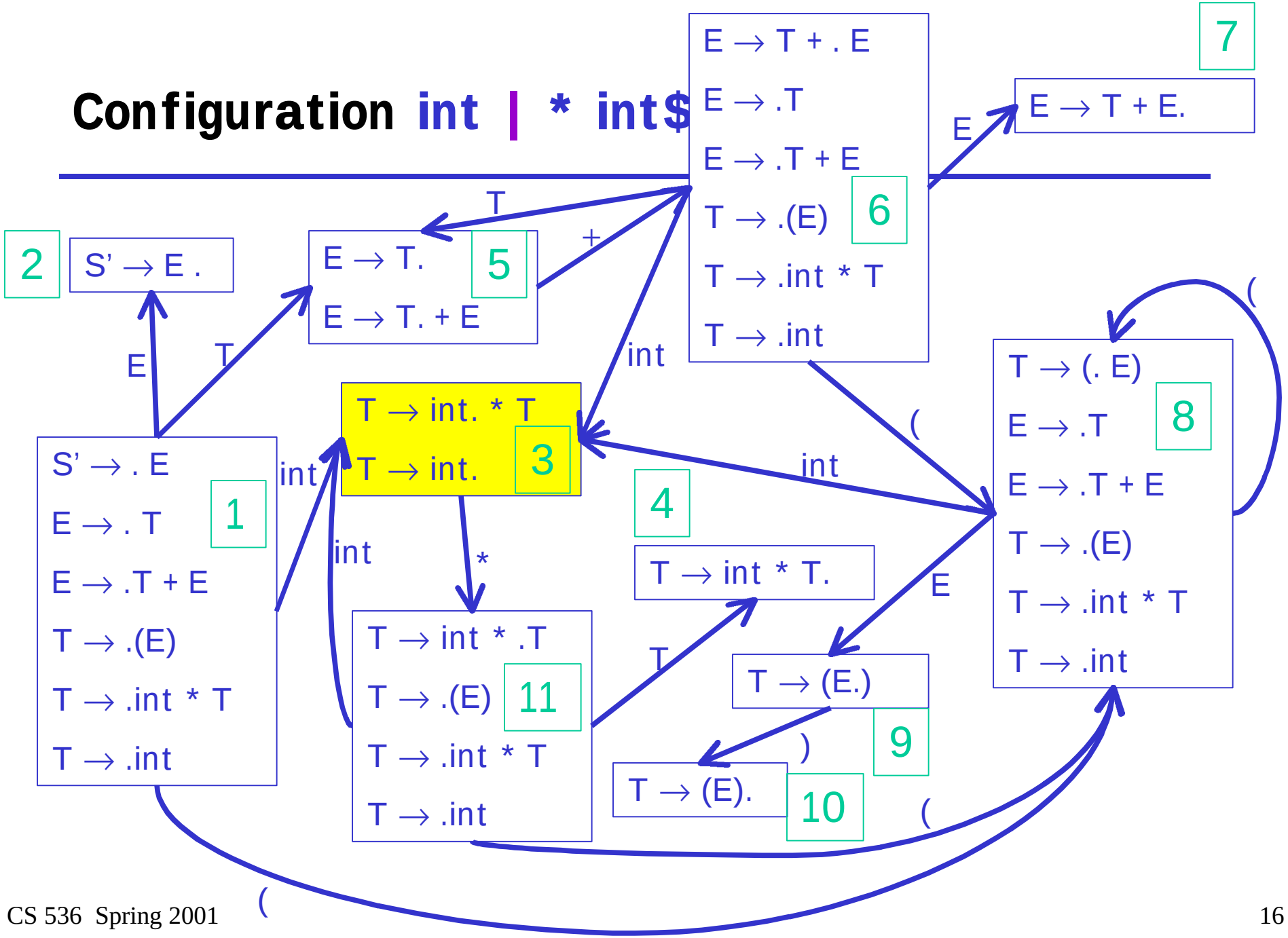


SLR Example

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

Configuration int| * int$

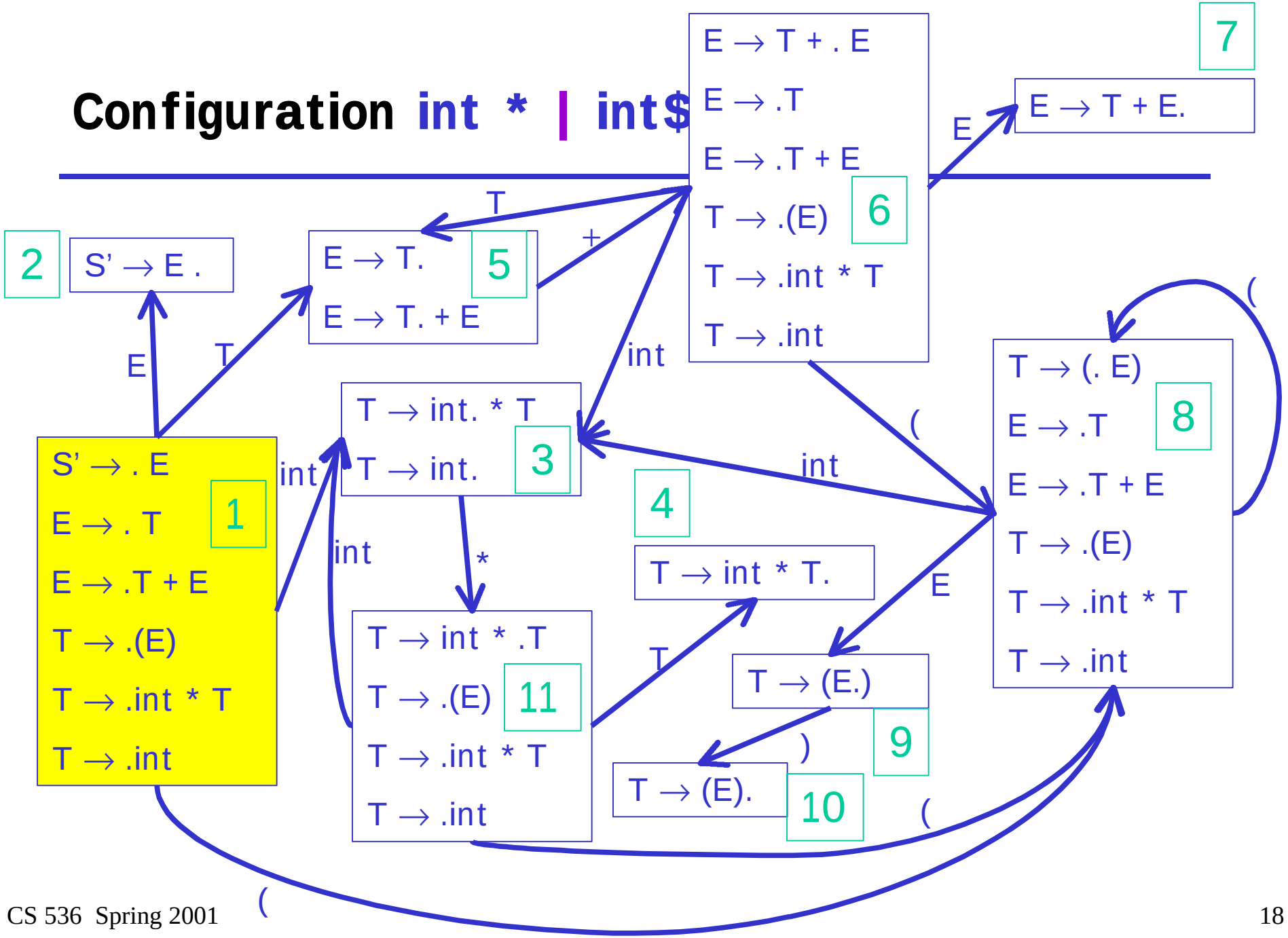



Configuration int | * int$
1
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int
2
S' → E .
3
T → int. * T
T → int.
4
T → int * T.
5
E → T.
E → T. + E
6
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int
7
E → T + E.
8
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int
9
T →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

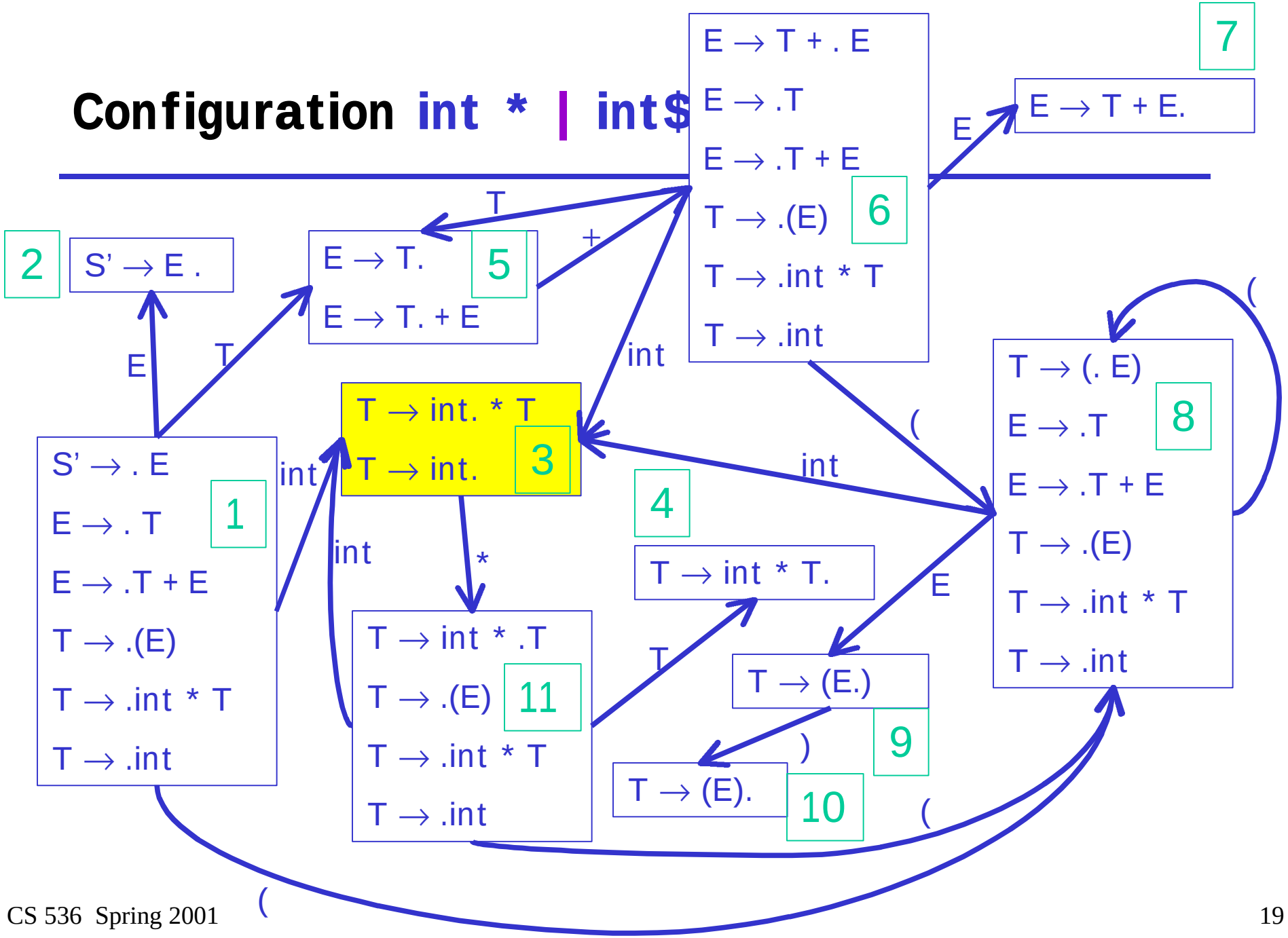
SLR Example

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

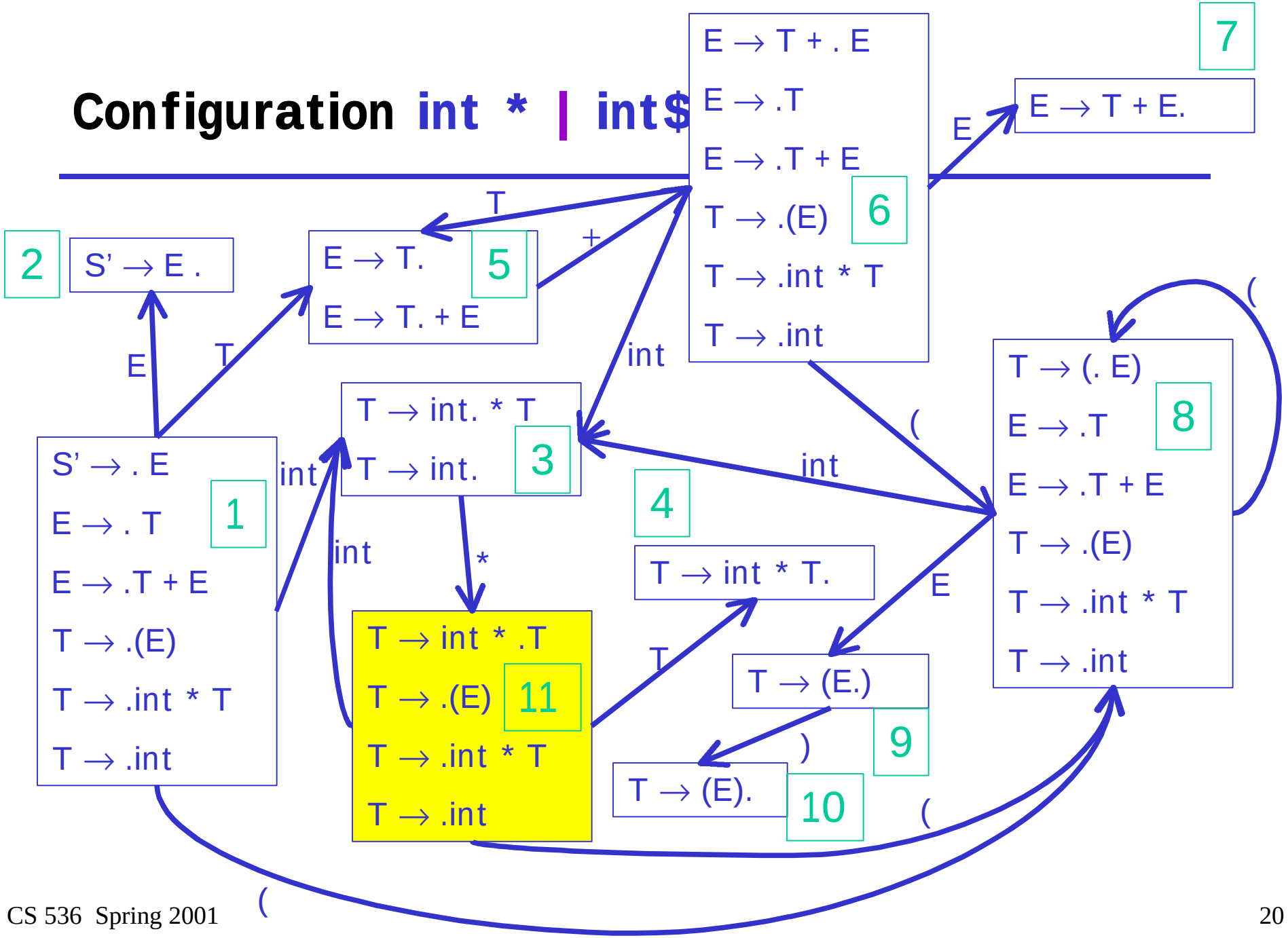
Configuration int * | int$



Configuration int * | int$



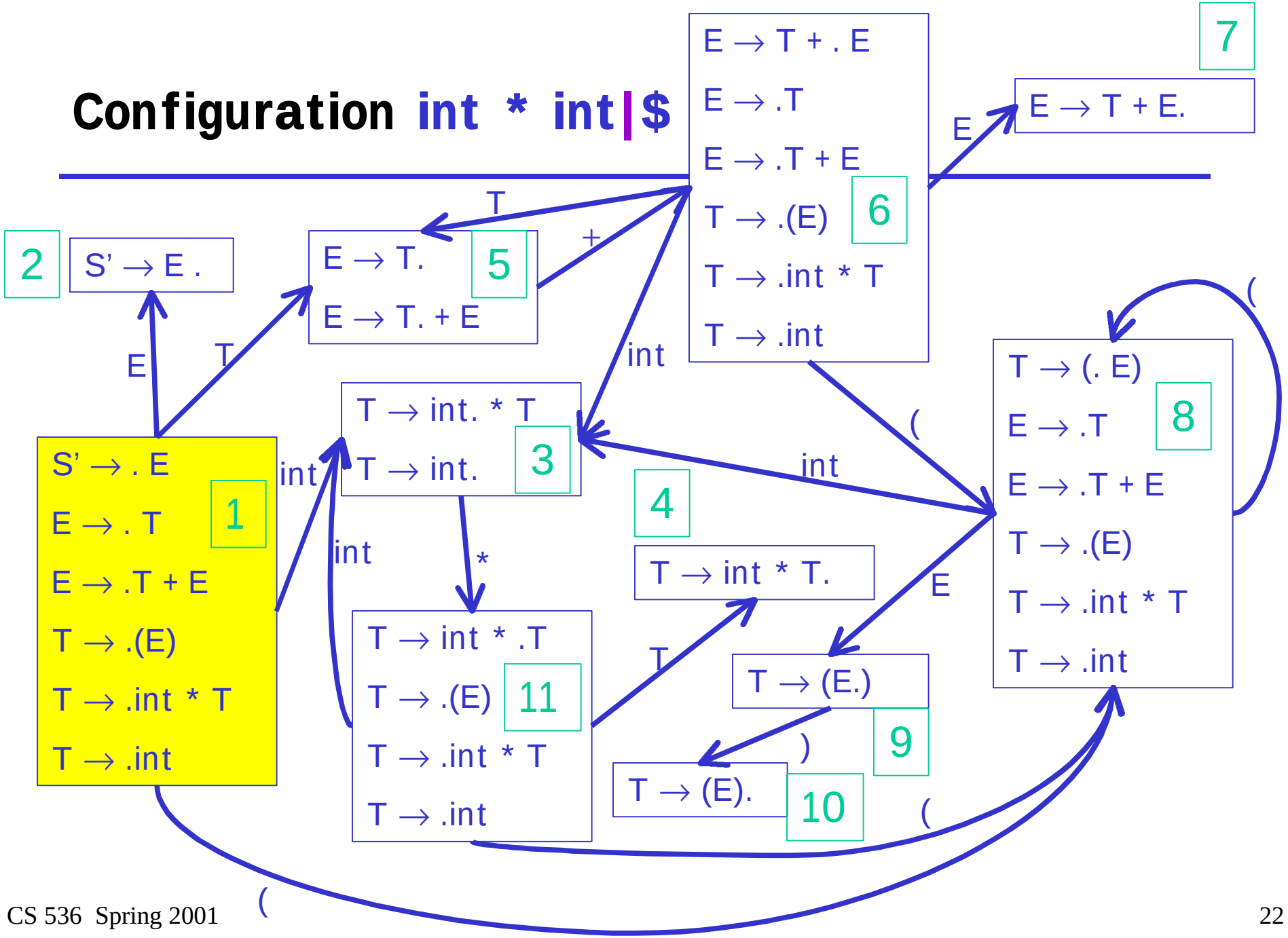
Configuration int * | int$



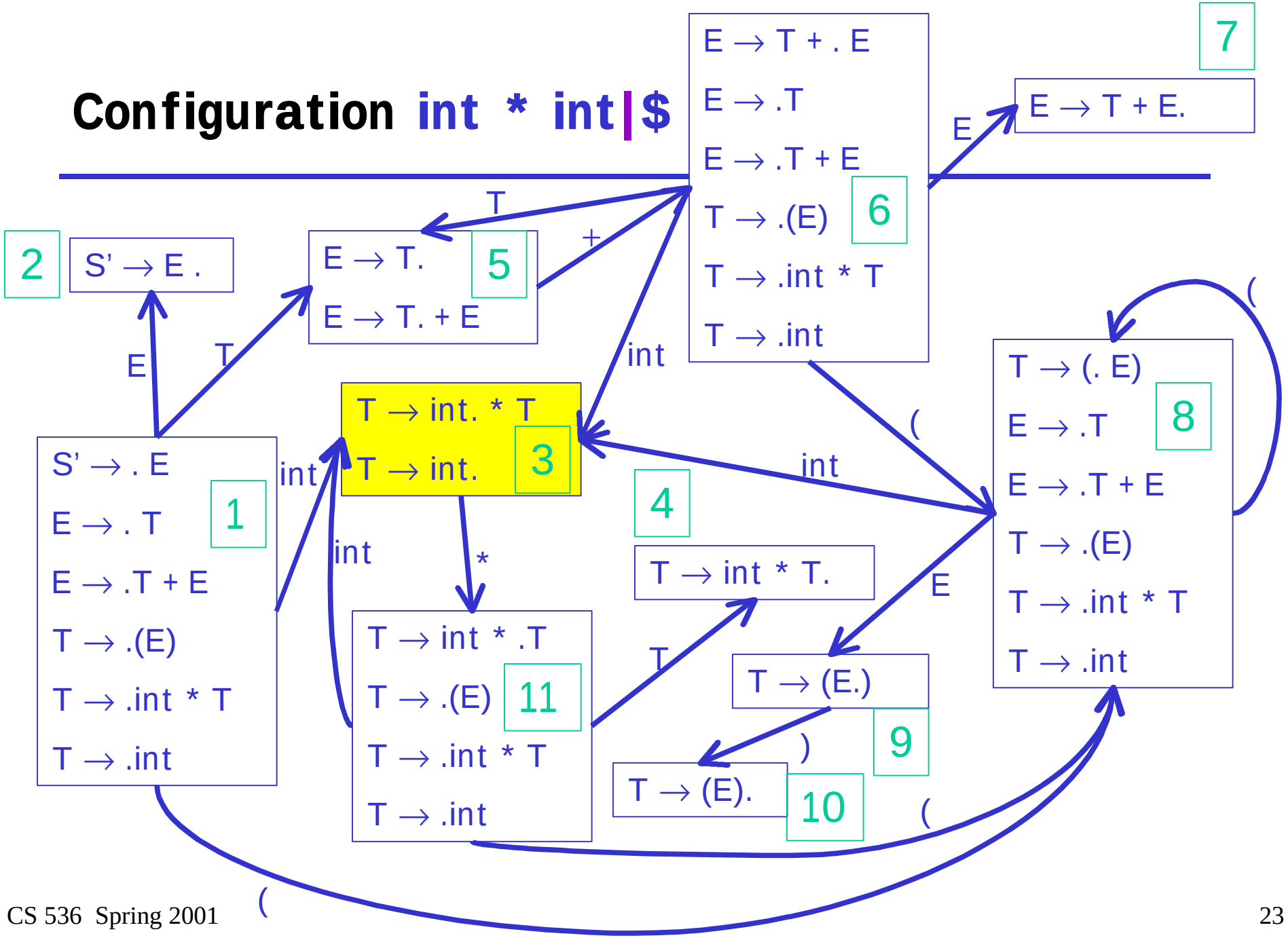
SLR Example

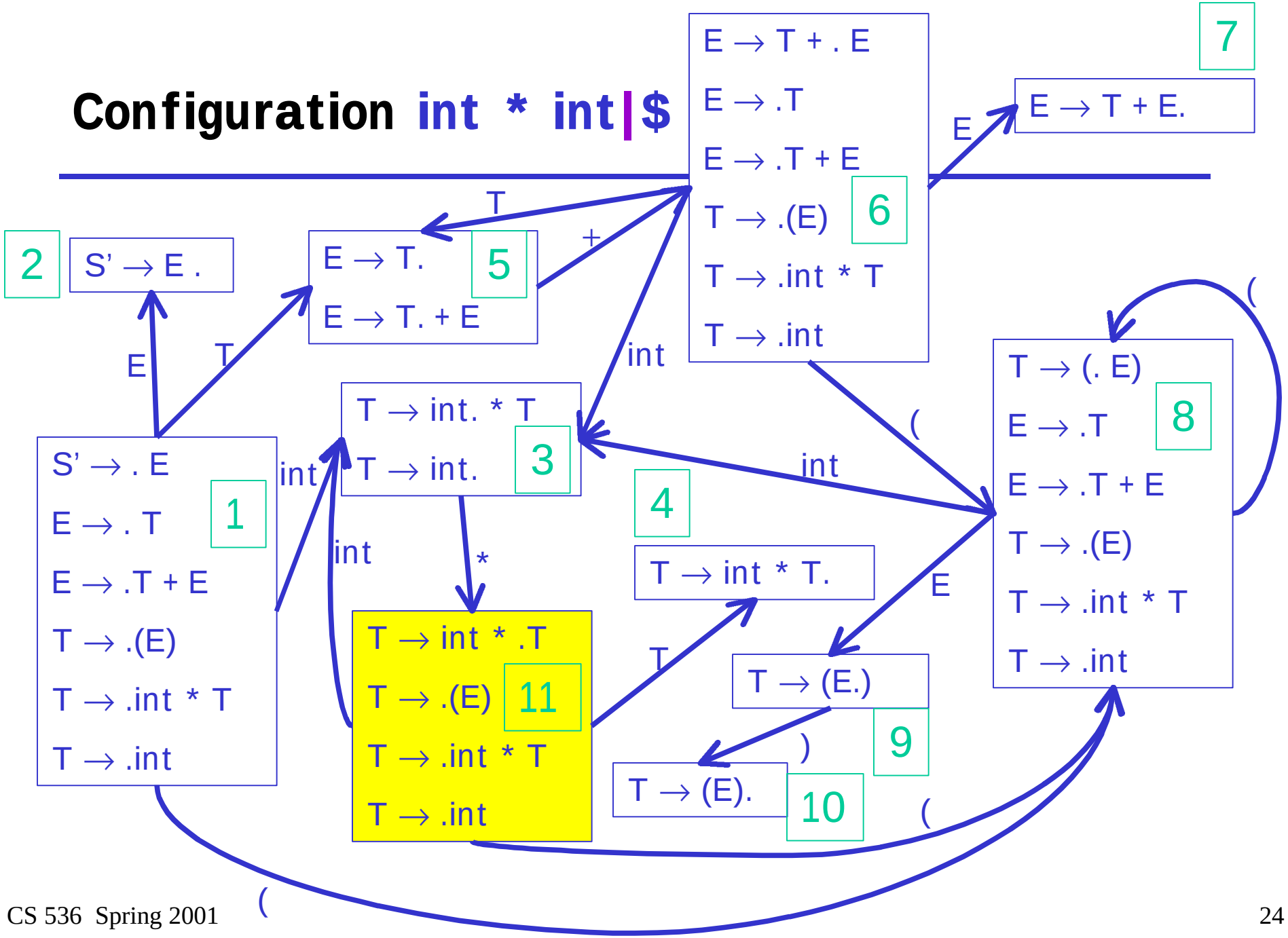
Configuration	*DFA Halt State*		*Action*
|int * int$	1		shift
int | * int$	3	* not in Follow(T)	shift
int * | int$	11		shift
int * int |$	3	$ ∈ Follow(T)	red. T→int

Configuration int * int|$

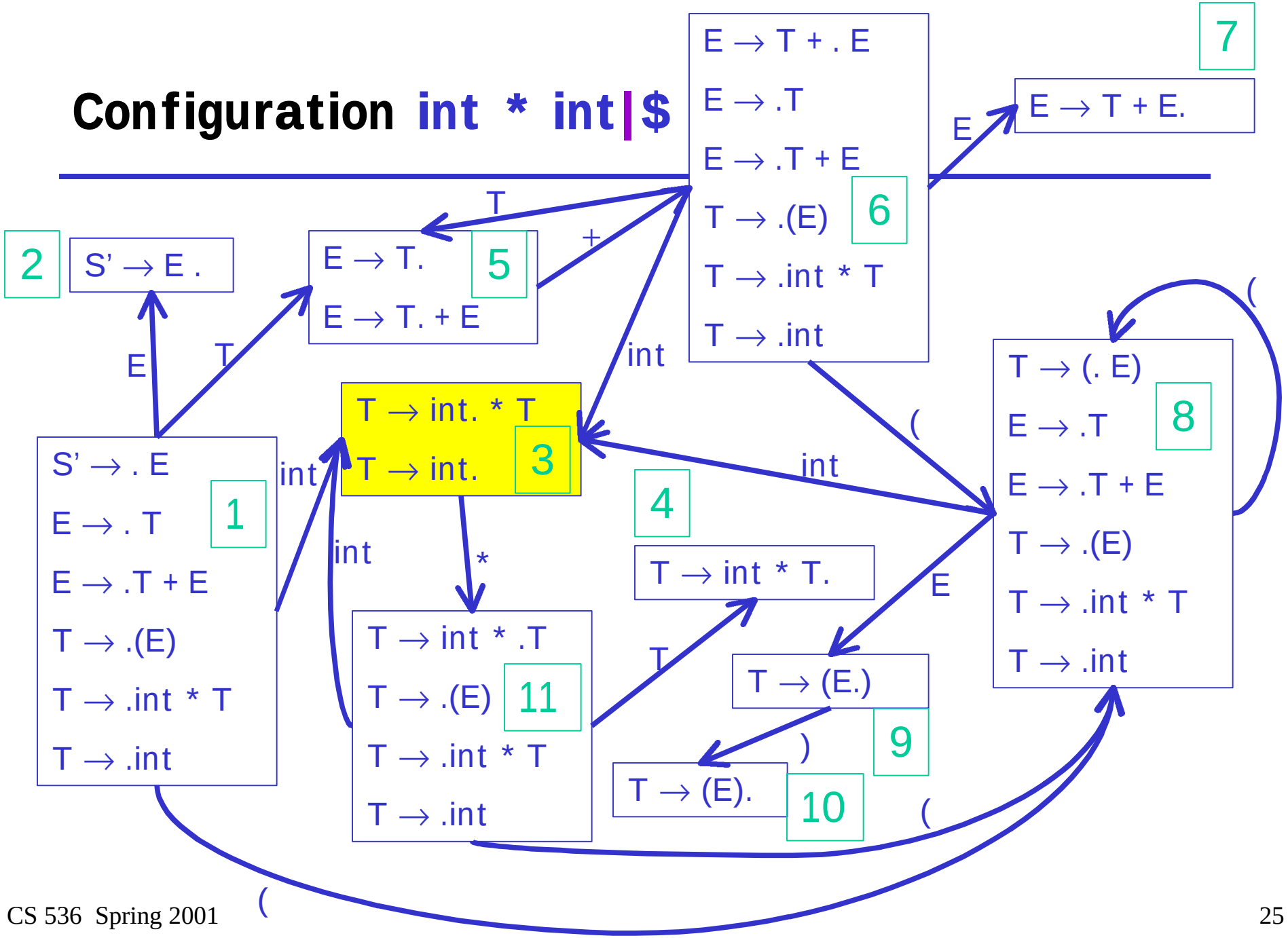


Configuration int * int|$




Configuration 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
int
(
+
*
)

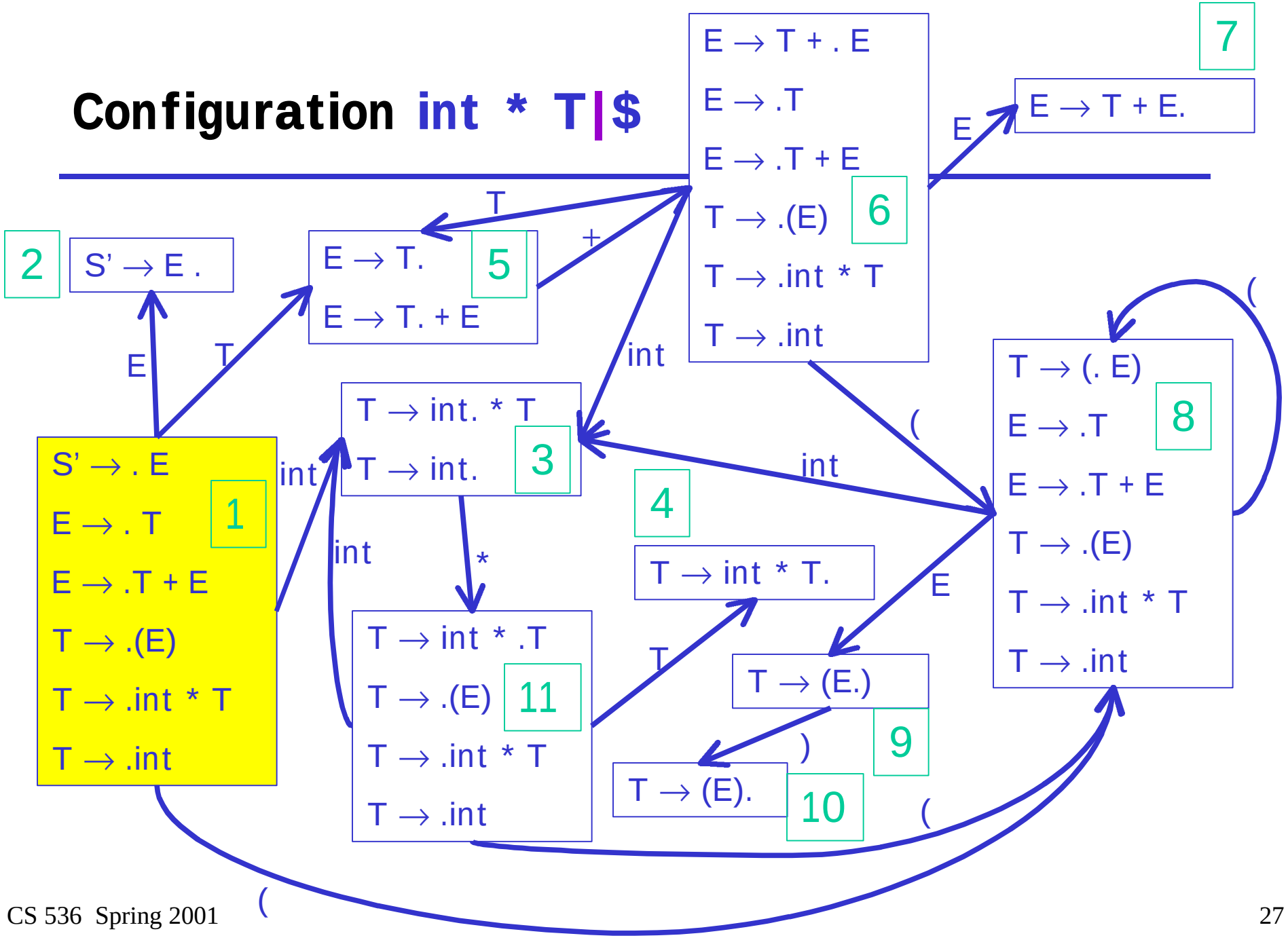
Configuration int * int|$



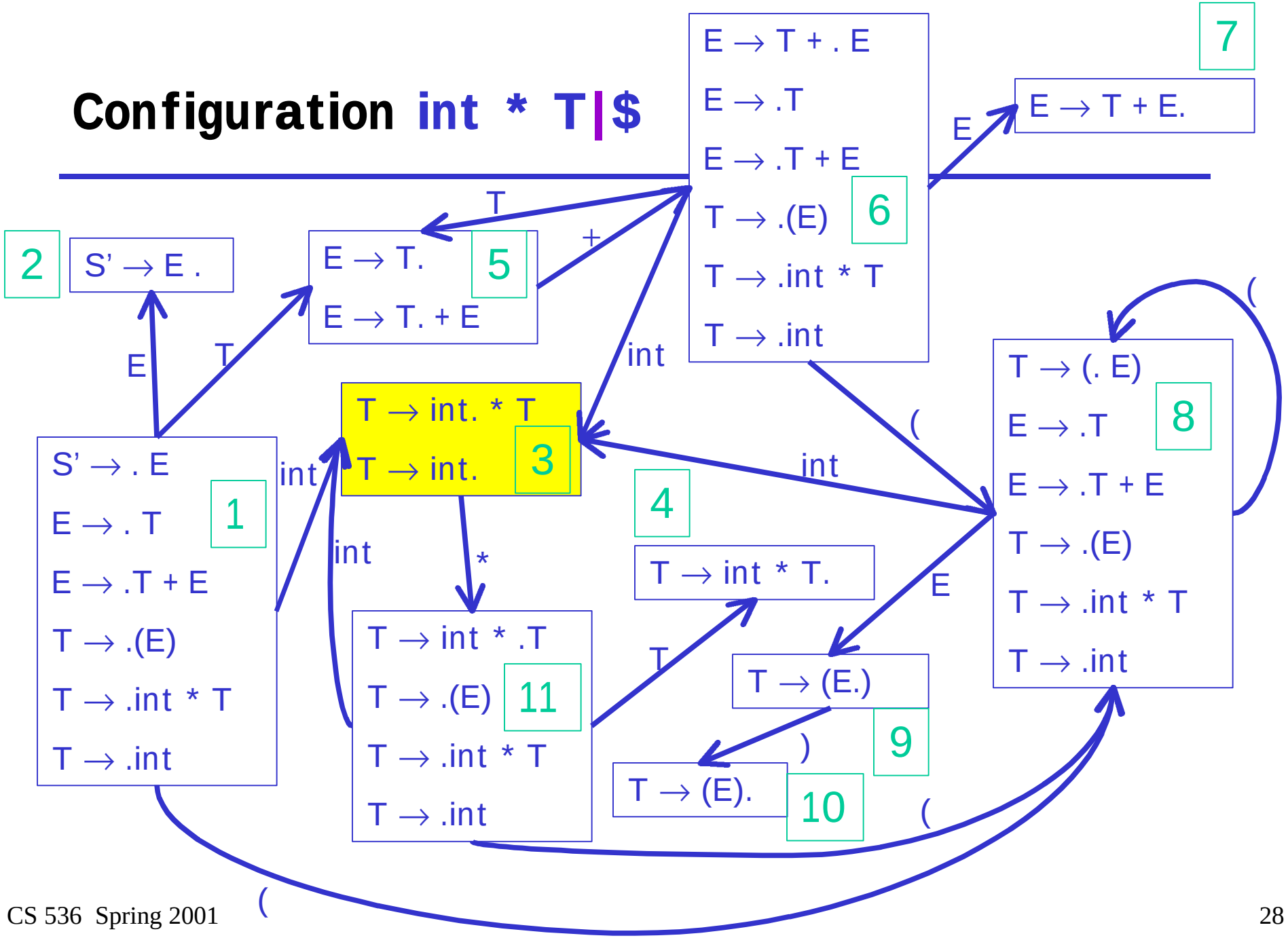
SLR Example

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)	red. T→int
int * | T $	11		shift
int * T |$	4	$ ∈ Follow(T)	red. T→int*T

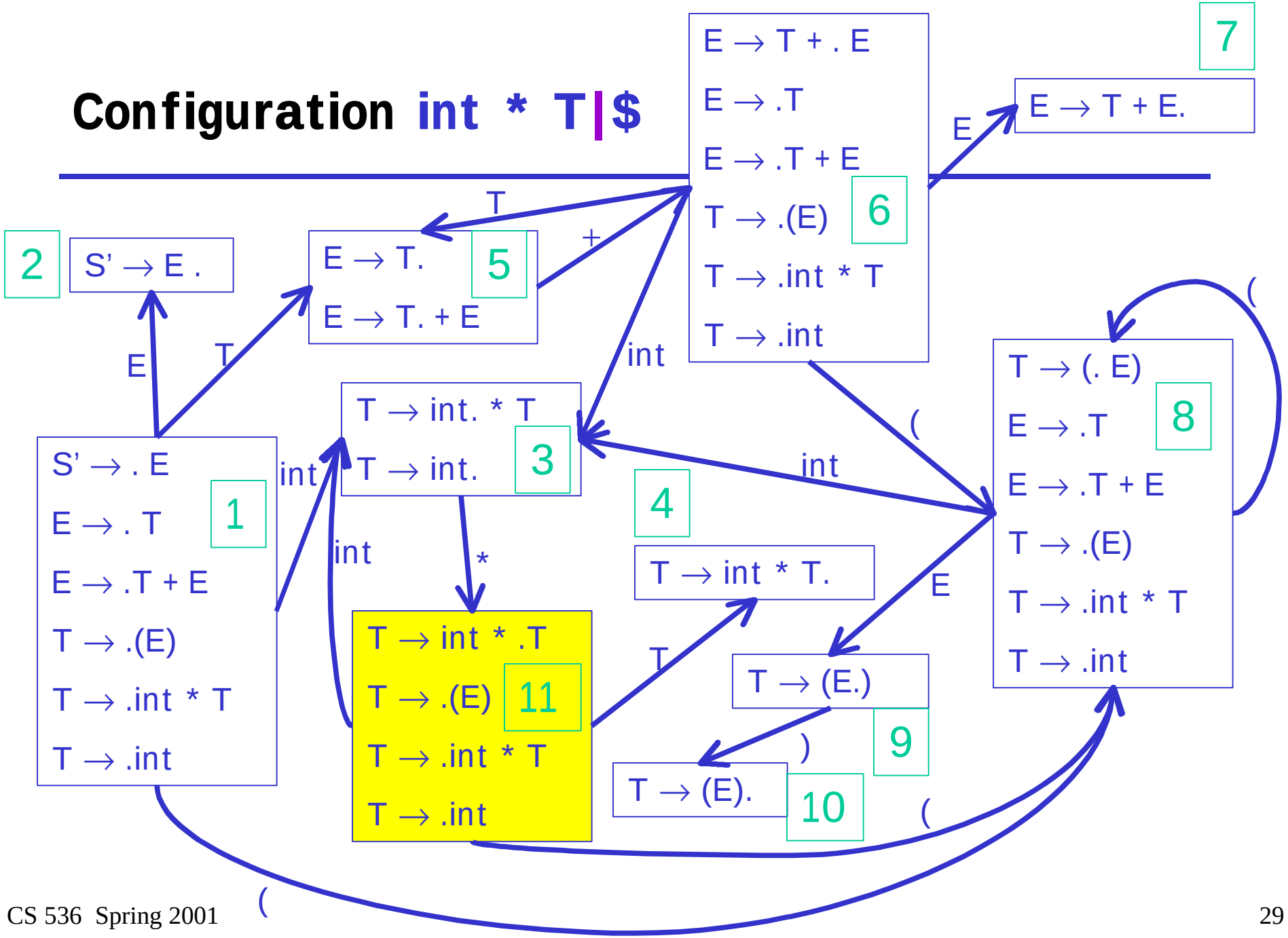
Configuration int * T|$



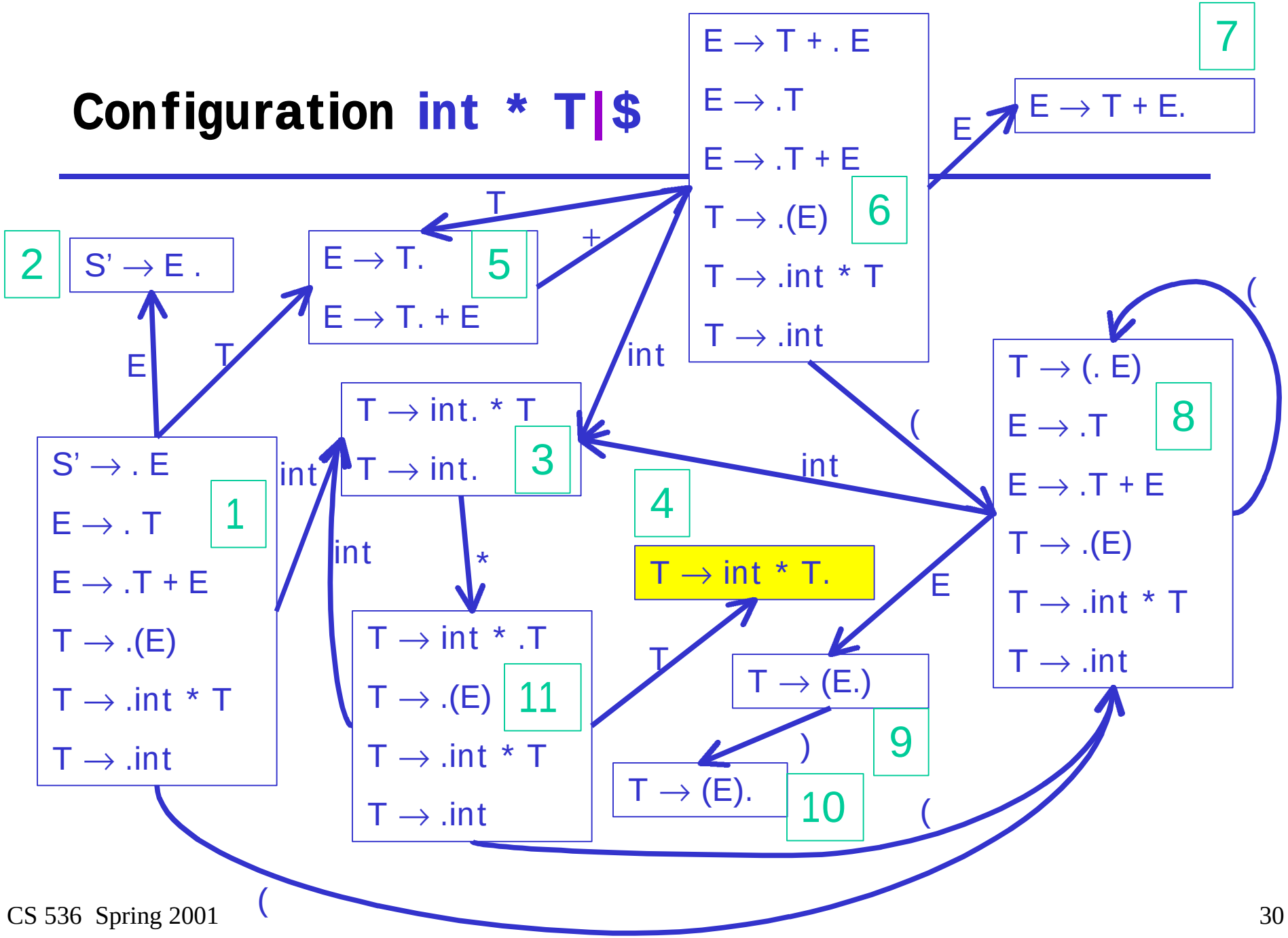
Configuration int * T|$



Configuration int * T|$



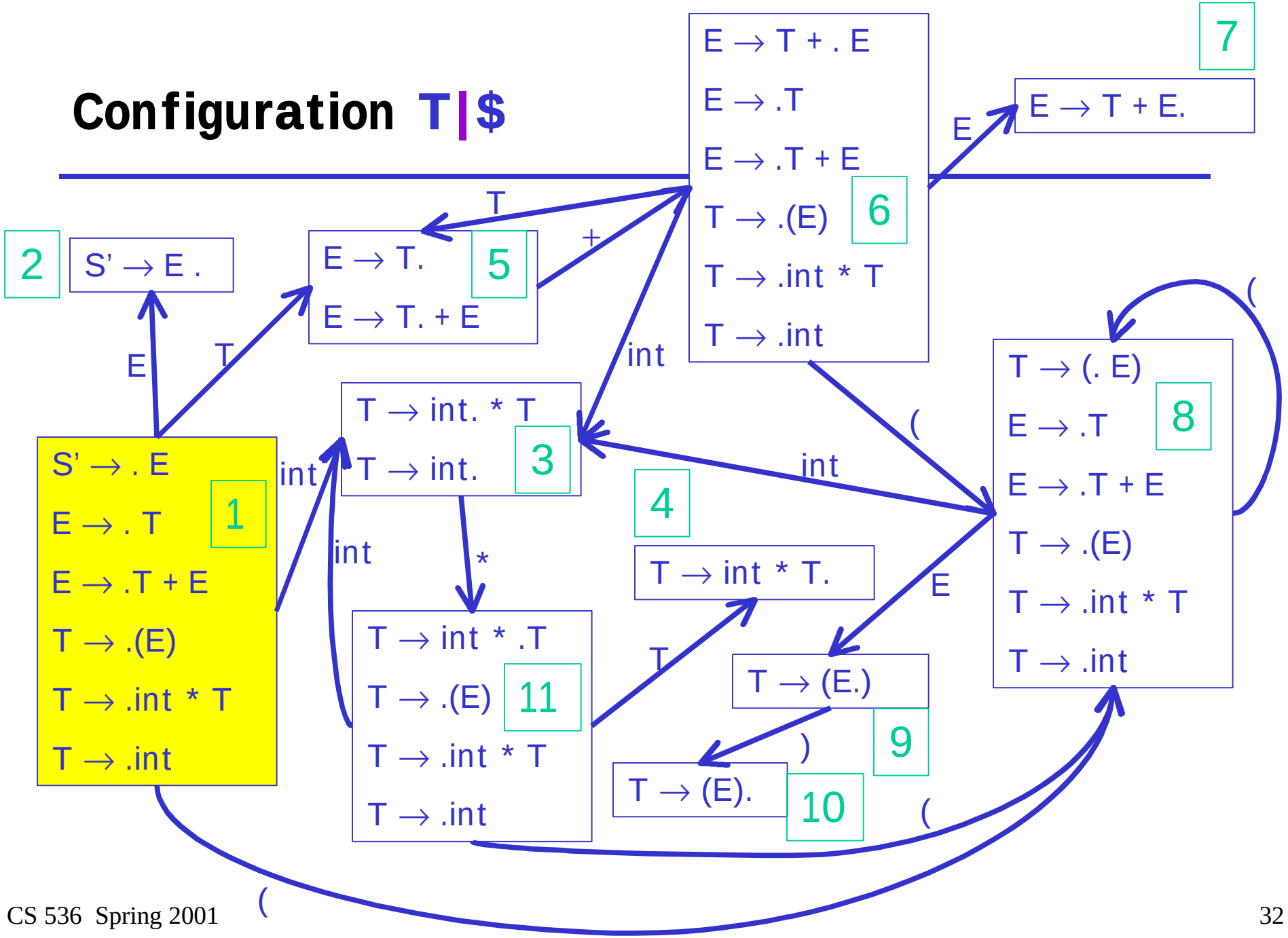
Configuration int * T|$



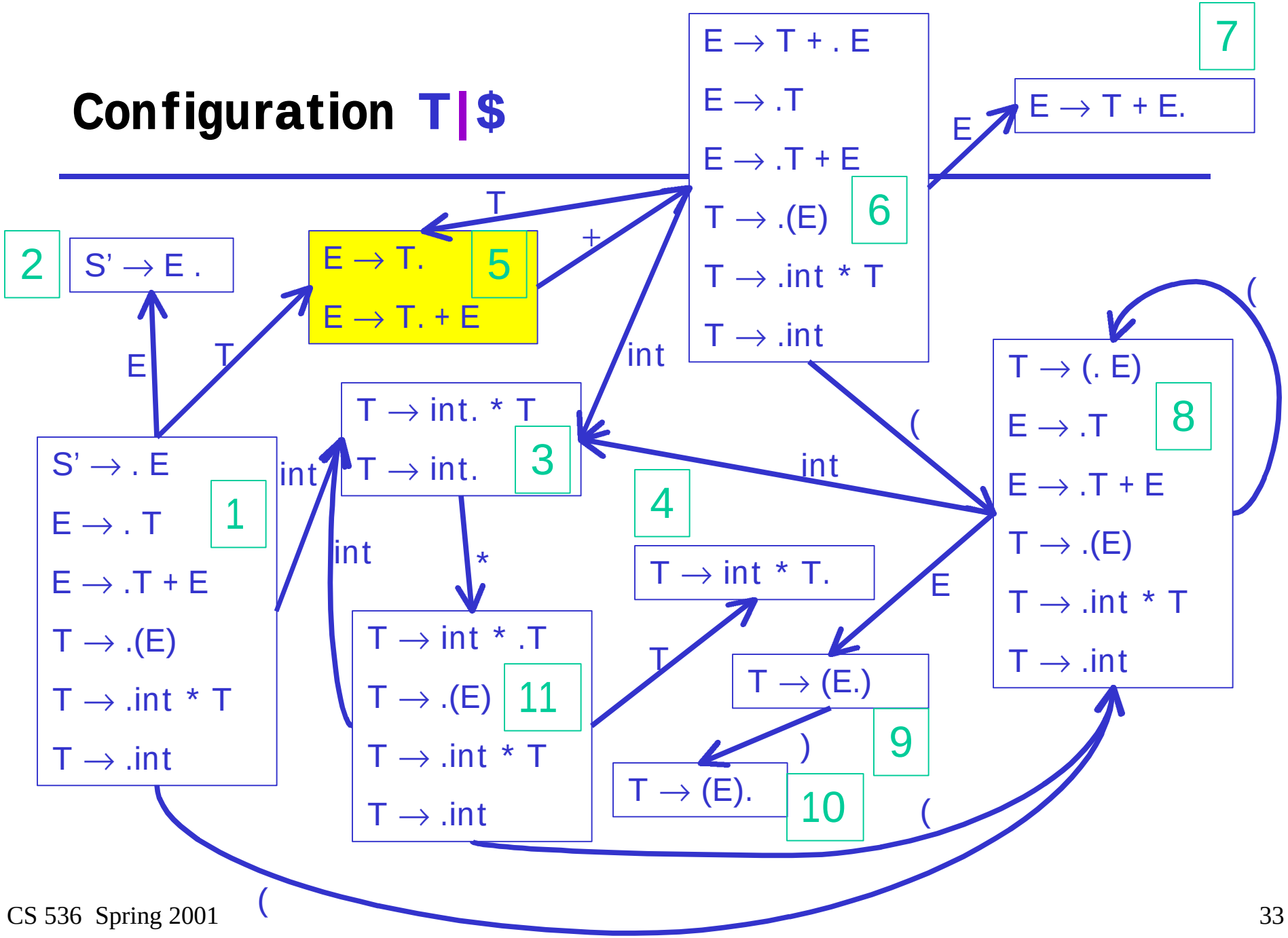
SLR Example

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)	red. T→int
int * | T $	11		shift
int * T |$	4	$ ∈ Follow(T)	red. T→int*T
|T$	1		shift
T |$	5	$ ∈ Follow(E)	red. E→T

Configuration T|$



Configuration T|$



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)	red. T→int
int * | T $	11		shift
int * T |$	4	$ ∈ Follow(T)	red. T→int*T
|T$	1		shift
T |$	5	$ ∈ Follow(E)	red. E→T
|E$	1		shift
E |$	2		accept

Notes

- Textbook uses one more state:
  - it accepts in state "S' → E $. "
  - I.e., it accepts in configuration E$|, not in E|$.

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

An Improvement

- 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

  $\langle sym_1, state_1 \rangle \ldots \langle sym_n, state_n \rangle$

  $state_n$ is the final state of the DFA on $sym_1 \ldots sym_n$

- Detail: The bottom of the stack is ⟨any,start⟩ where
  - any is any dummy state (textbook uses blank, Fig 6.6)
  - start is the start state of the DFA

Goto 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 a

- If s_i has item $X \rightarrow \alpha.a\beta$ and Goto[i,a] = j then Action[i,a] = shift j
- If s_i has item $X \rightarrow \alpha.$ and $a \in$ Follow(X) and $X \neq S'$ then Action[i,a] = reduce $X \rightarrow \alpha$
- If s_i has item $S' \rightarrow S.$ then action[i,$] = accept
- Otherwise, action[i,a] = error

SLR Parsing Algorithm (Fig 6.3)

```
Let I = w$ be initial input
Let j = 0
Let DFA state 1 have item S' → .S
Let stack = ⟨ dummy, 1 ⟩
    repeat
        case action[top_state(stack), I[j]] of
                shift k:  push ⟨ I[j++], k ⟩
                reduce X → A:
                    pop |A| pairs,
                    I[--j] = X          // prepend X to input
                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

Constructing LR states

- Read Section 6.3
  - we'll review it on Friday
  - today: an intuitive construction

- LR(0) state machine
  - encodes all strings that are valid on the stack.
  - each valid string is a configuration, and hence corresponds to a state of the LR(0) state machine
  - each state tells us what to do (shift or reduce?)

LR(0) states for the example grammar