

Bottom-Up Parsing II

(Different types of Shift-Reduce Conflicts)

Lecture 10

Review: Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
 - And just as efficient
 - Doesn't have issues with left-recursion
 - Doesn't require left-factoring
 - Many well-known parser generators (Yacc, Bison,...)
 - Can handle many more grammars without backtracking than otherwise
- LR(k) parsers are bottom-up

Review: 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: $\text{int} * \text{int} + \text{int}$

Review: The Idea

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

int * int + int

$T \rightarrow \text{int}$

int * T + int

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

T + int

$T \rightarrow \text{int}$

T + T

$E \rightarrow T$

T + E

$E \rightarrow T + E$

E

Review: Bottom-up Parsers and Rightmost derivations in reverse

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

int * int + int

$T \rightarrow \text{int}$

int * T + int

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

T + int

$T \rightarrow \text{int}$

T + T

$E \rightarrow T$

T + E

$E \rightarrow T + E$

E

Review: Notation useful for Shift-Reduce Parsers

- 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 |
 - The | is not part of the string
- Initially, all input is unexamined | $x_1 x_2 \dots x_n$

Review: Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

Shift

Reduce

Review: Shift

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

$ABC|xyz \Rightarrow ABCx|yz$

Review: Reduce

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

$$ABCx|yz \Rightarrow ABD|yz$$

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)

Review: The Example with Reductions Only

int * int | + int

reduce $T \rightarrow \text{int}$

int * T | + int

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

T + int |

reduce $T \rightarrow \text{int}$

T + T |

reduce $E \rightarrow T$

T + E |

reduce $E \rightarrow T + E$

E |

Review: The Example with Shift-Reduce Parsing

int * int + int	shift
int * int + int	shift
int * int + int	shift
int * int + int	reduce $T \rightarrow \text{int}$
int * T + int	reduce $T \rightarrow \text{int} * T$
T + int	shift
T + int	shift
T + int	reduce $T \rightarrow \text{int}$
T + T	reduce $E \rightarrow T$
T + E	reduce $E \rightarrow T + E$
E	

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

Key Issue: To Shift or Reduce?

- 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

Handles: Symbols replaced by Reduction

- **Intuition:** Want to reduce only if the result can still be reduced to the start symbol
- **Handle:** Informally, represents the RHS of a production. Let $X \rightarrow \beta$ be a production in G . Then β in the position after α is a *handle* of $\alpha\beta\omega$

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

Using the symbol already shifted into the stack (left context) and the next k lookahead symbols (right context), decide to shift or reduce

A Shift-Reduce Parse in Detail (1)

| int * int + int

int * *int* + *int*
↑

A Shift-Reduce Parse in Detail (2)

| int * int + int

int | * 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)

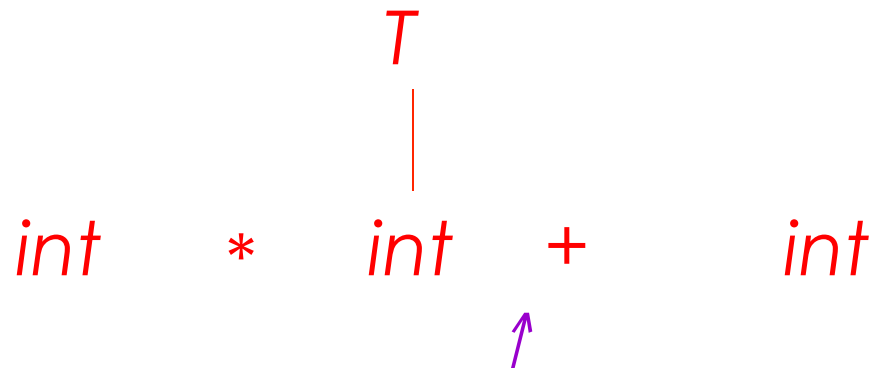
| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int



A Shift-Reduce Parse in Detail (6)

| int * int + int

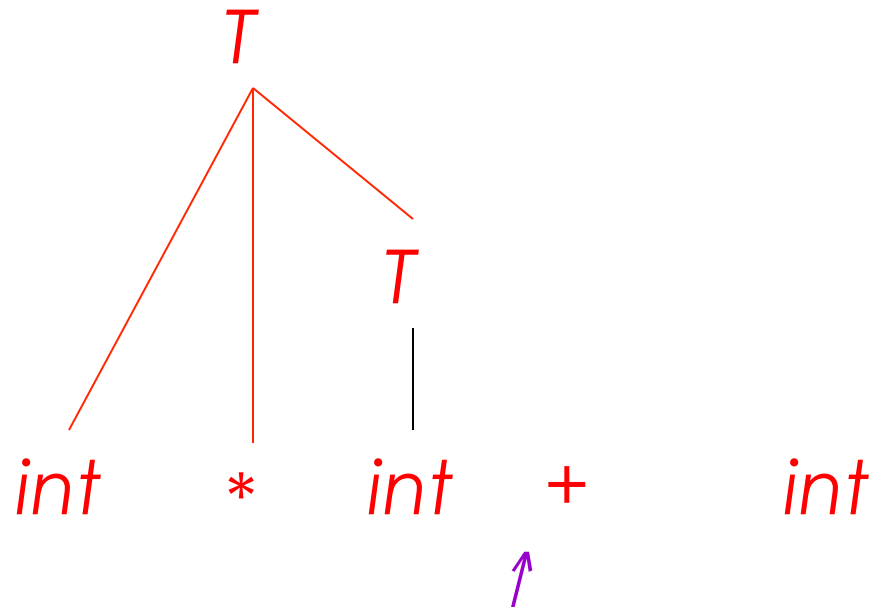
int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int



A Shift-Reduce Parse in Detail (7)

| int * int + int

int | * int + int

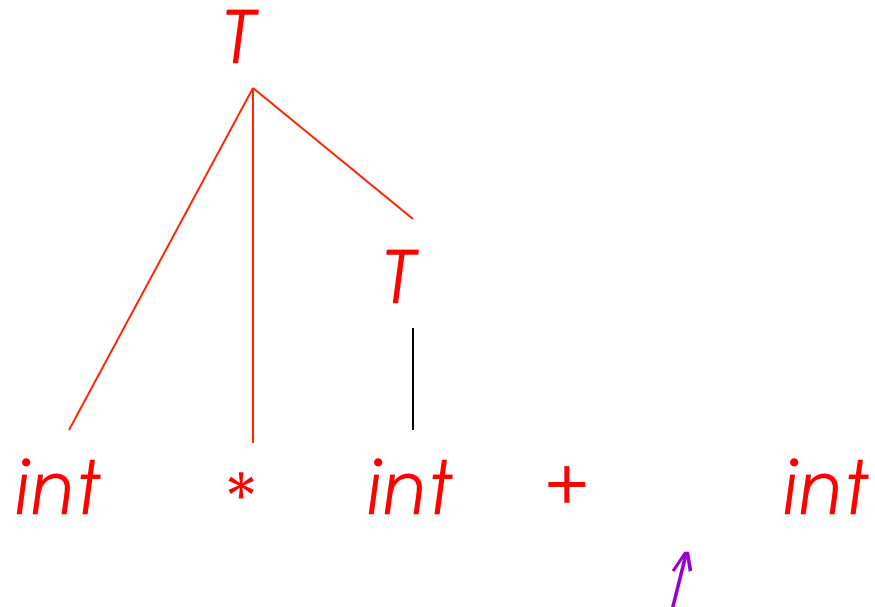
int * | int + int

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

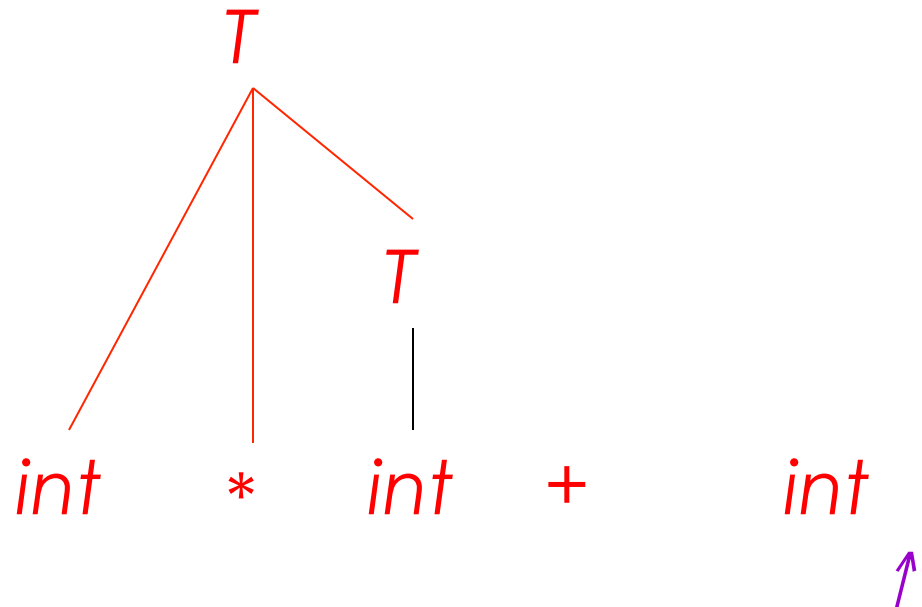
int * int | + int

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

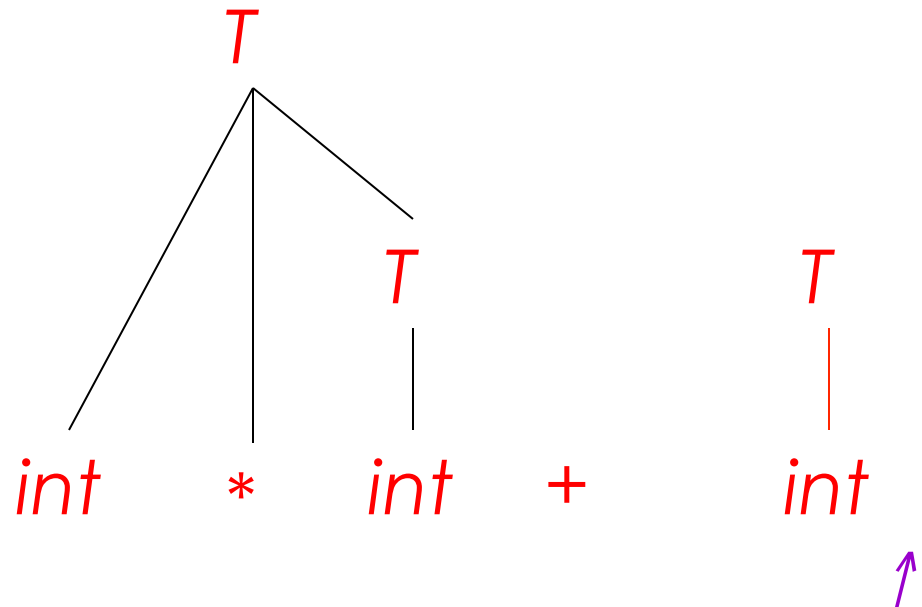
int * T | + int

T | + int

T + | int

T + int |

T + T |



A Shift-Reduce Parse in Detail (10)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

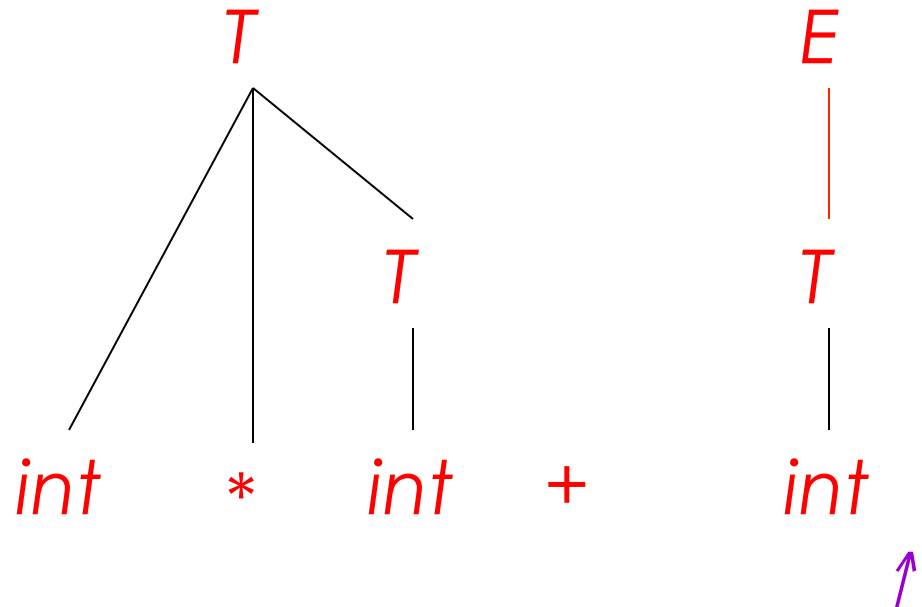
T | + int

T + | int

T + int |

T + T |

T + E |



A Shift-Reduce Parse in Detail (11)

| int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int

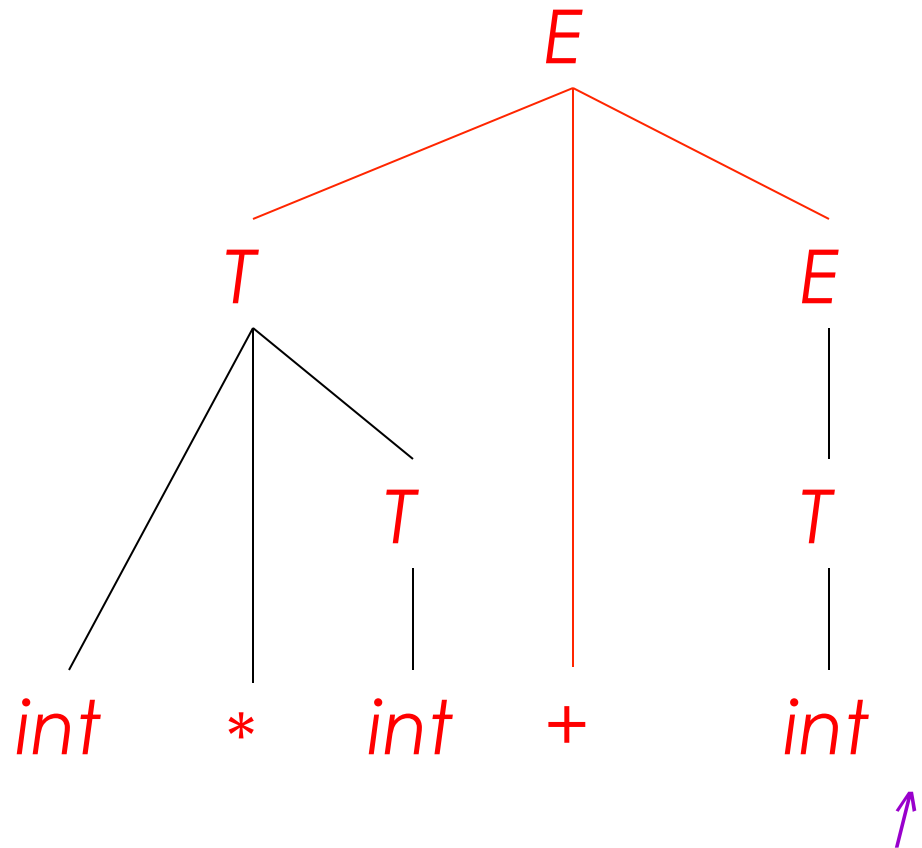
T + | int

T + int |

T + T |

T + E |

E |



Theorem: Handle always on top of Stack

- Proof Method: Informal induction on # of reduce moves
- Proof:
 - 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

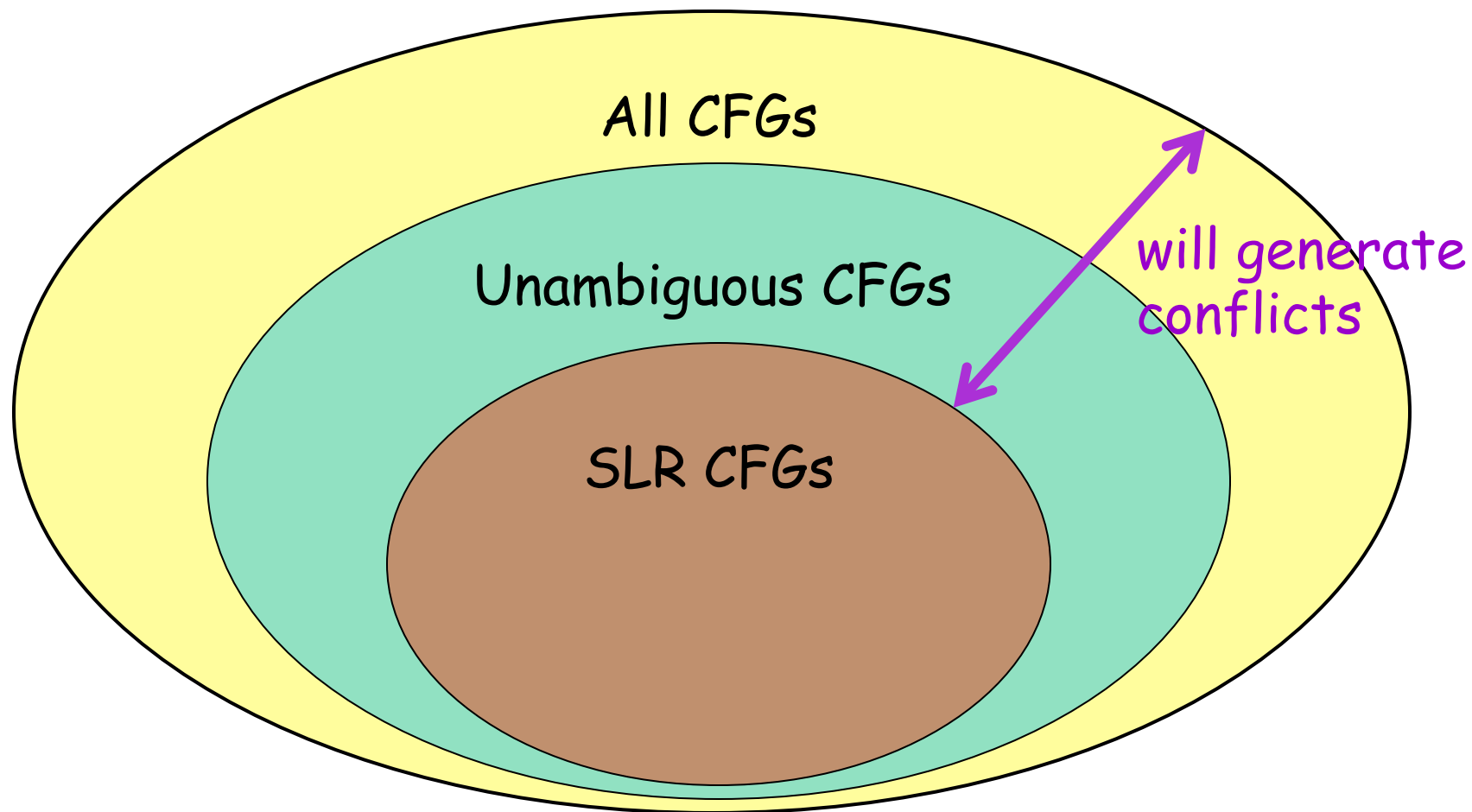
Summary of Handles

- 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
- 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 $X \rightarrow \cdot$
- 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

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 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

An Example

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

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

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

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

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

An Example (Cont.)

The “stack of items”

$T \rightarrow (.E)$

$E \rightarrow .T$

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

Says

We’ve seen “(” of $T \rightarrow (E)$

We’ve seen ϵ of $E \rightarrow T$

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

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

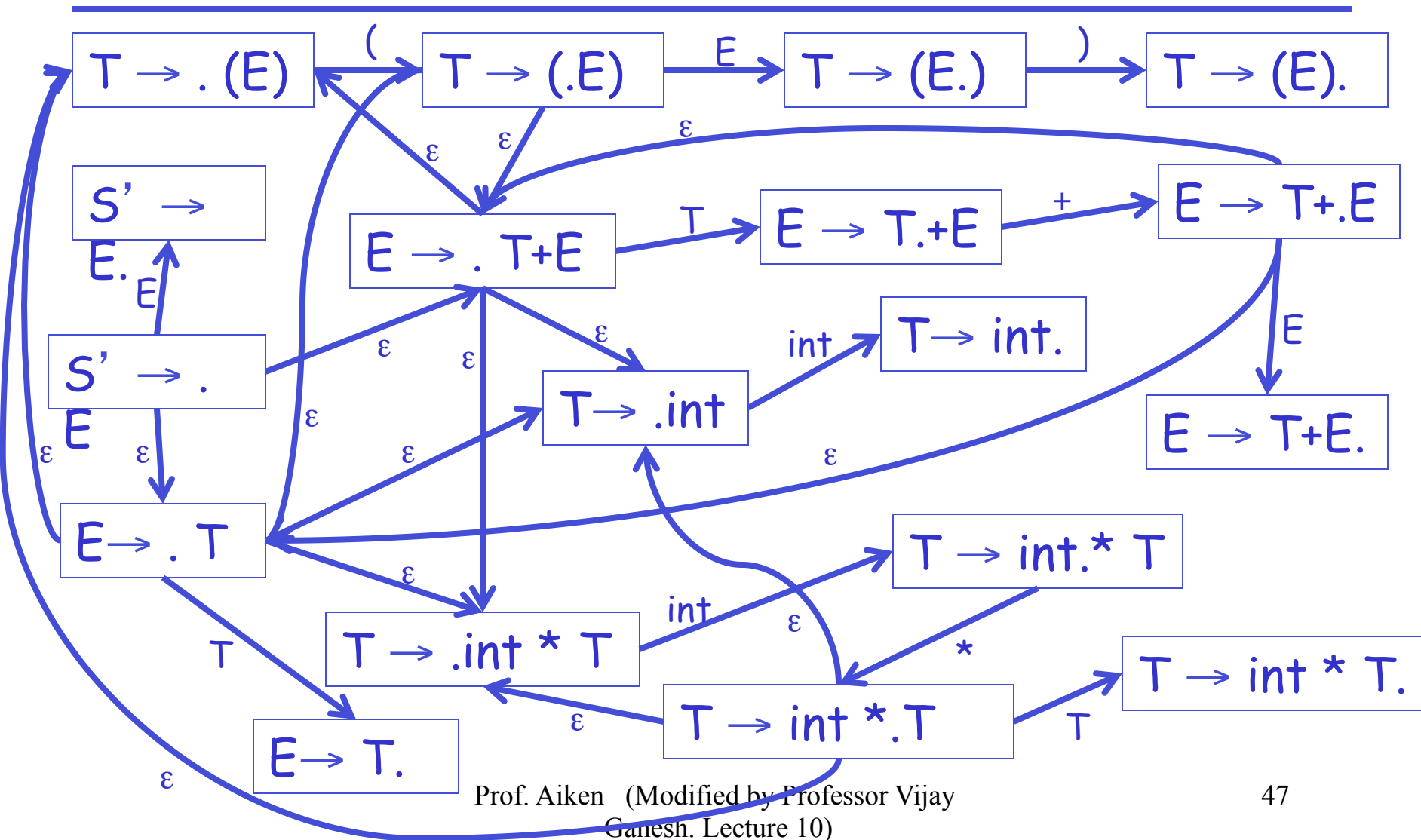
An NFA Recognizing Viable Prefixes

1. Add a dummy production $S' \rightarrow S$ to G
2. The NFA states are the items of G
 - Including the extra 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 $S' \rightarrow .S$

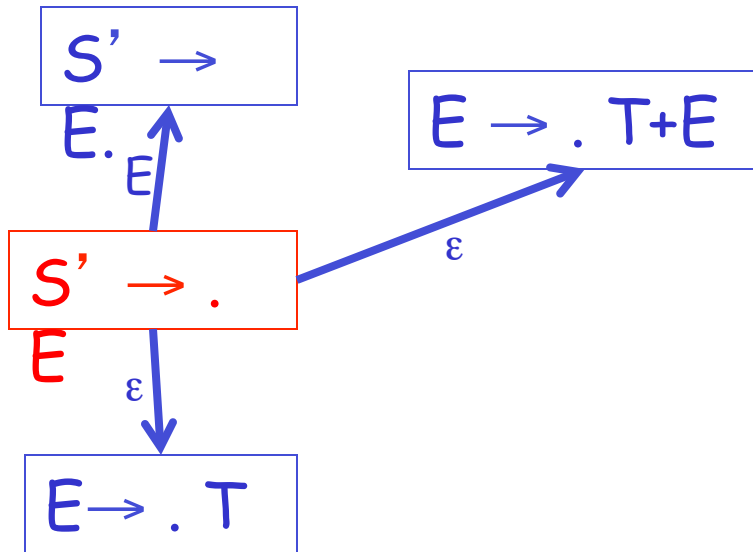
NFA for Viable Prefixes of the Example



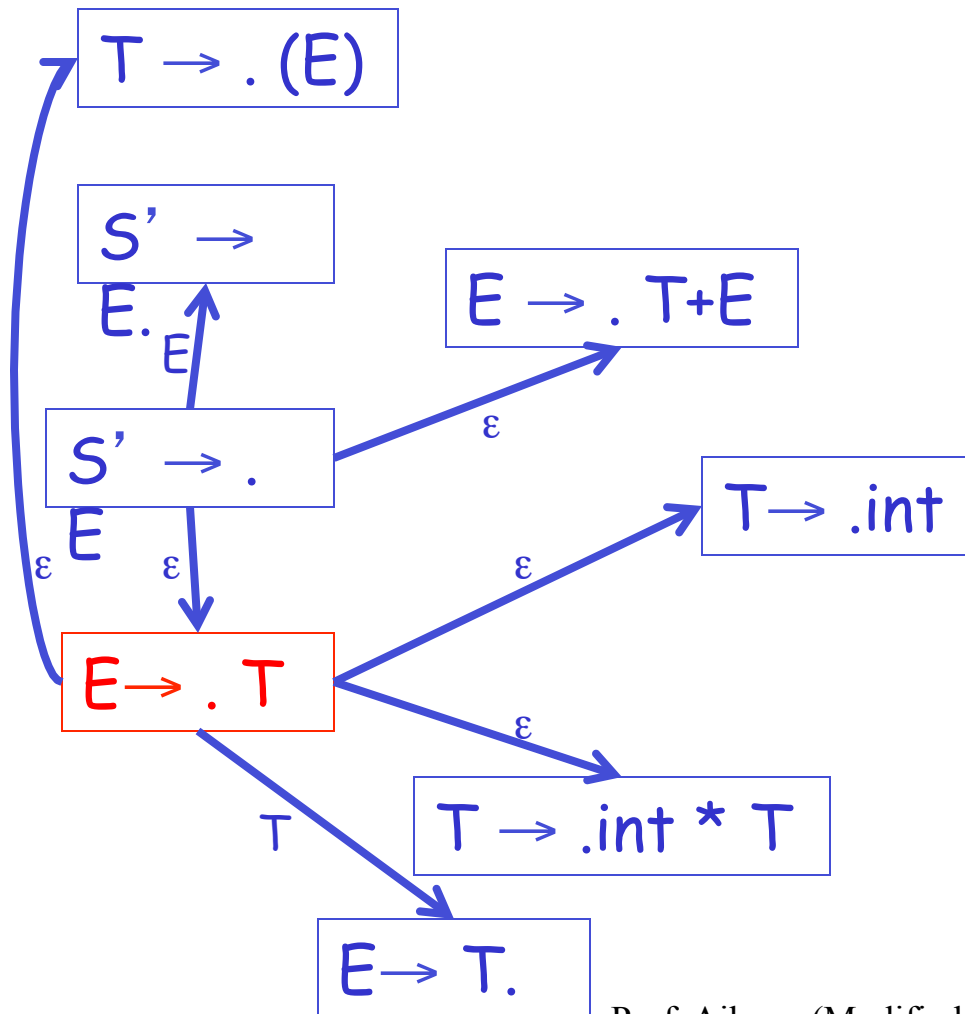
NFA for Viable Prefixes in Detail (1)

$S' \rightarrow .$
E

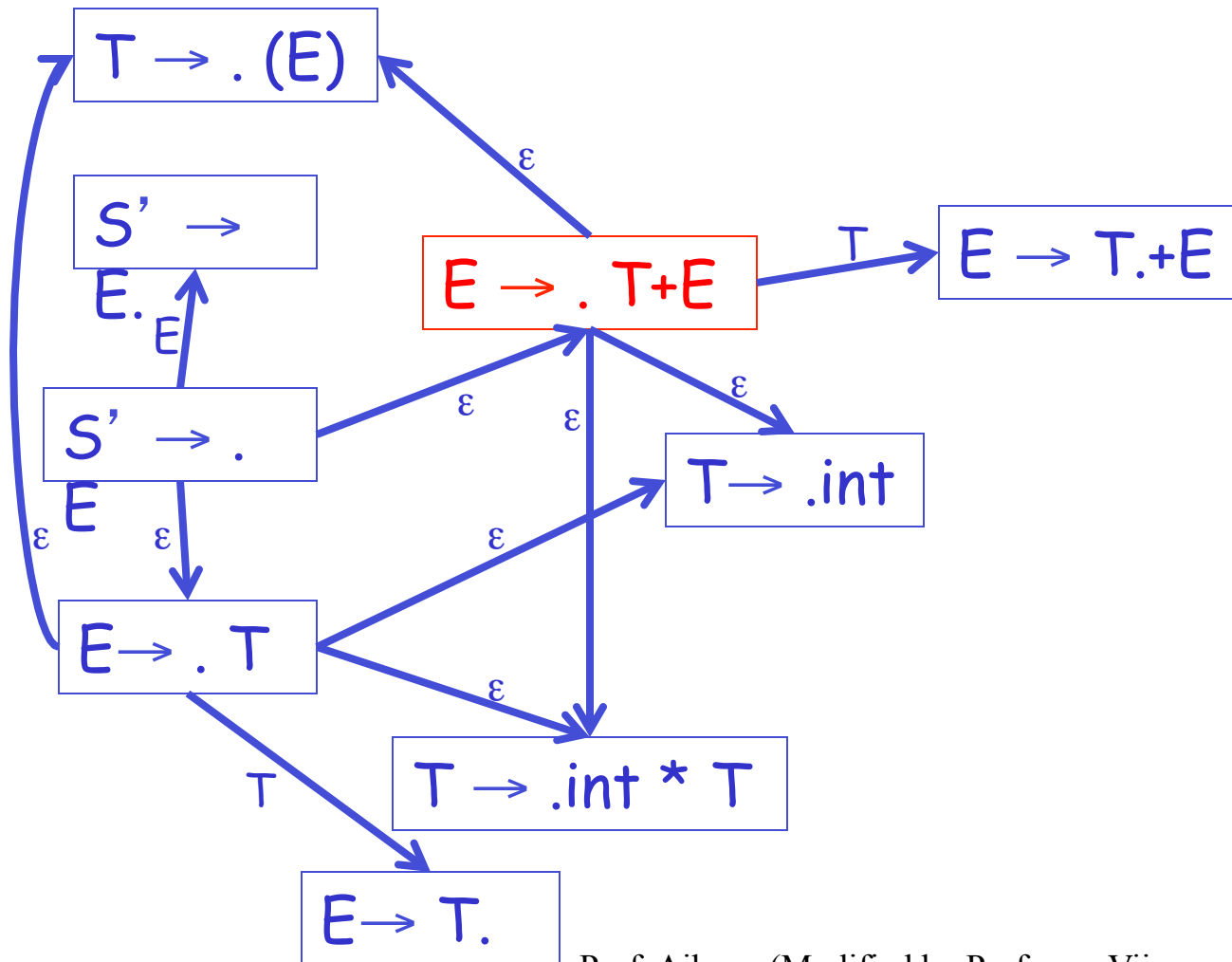
NFA for Viable Prefixes in Detail (2)



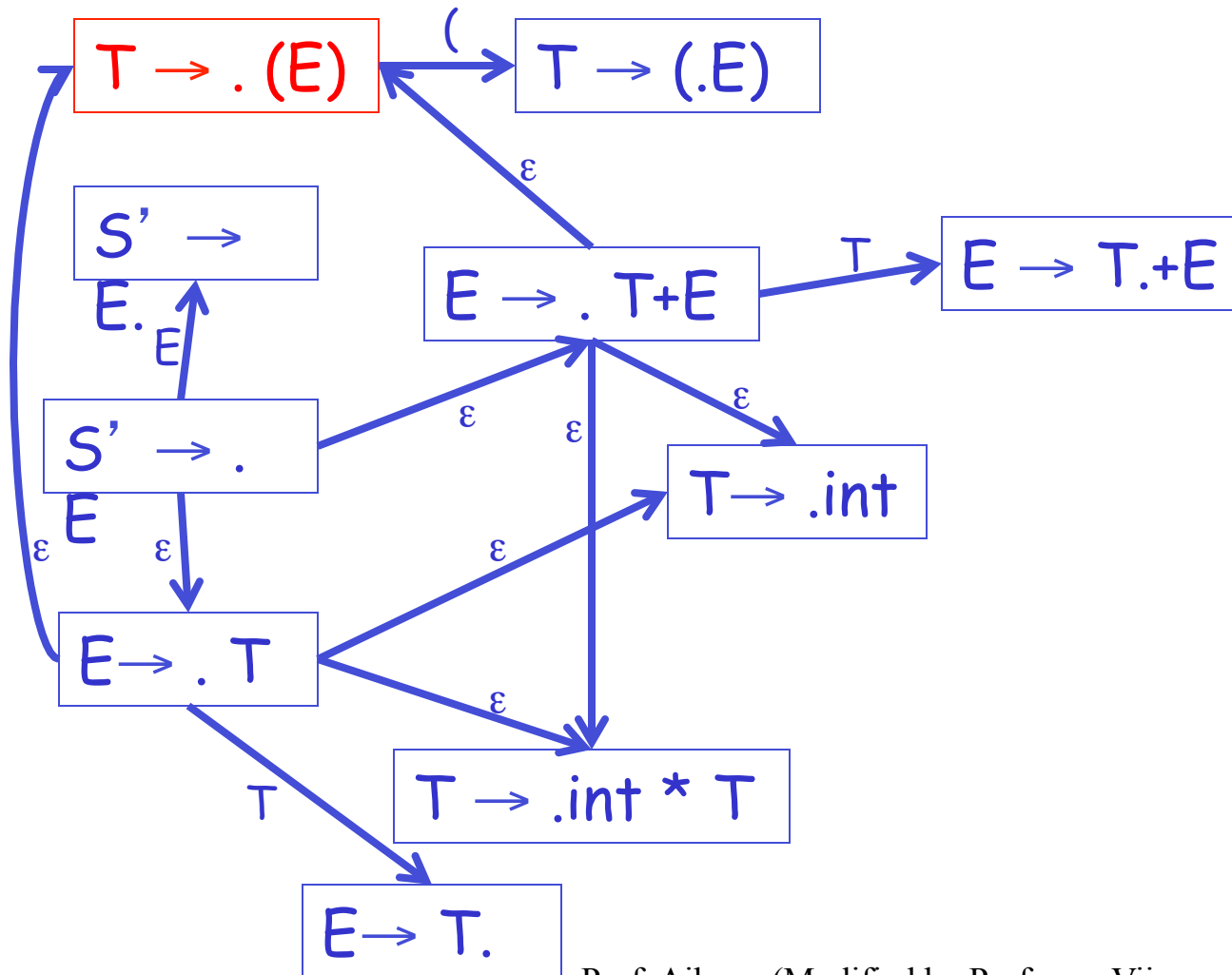
NFA for Viable Prefixes in Detail (3)



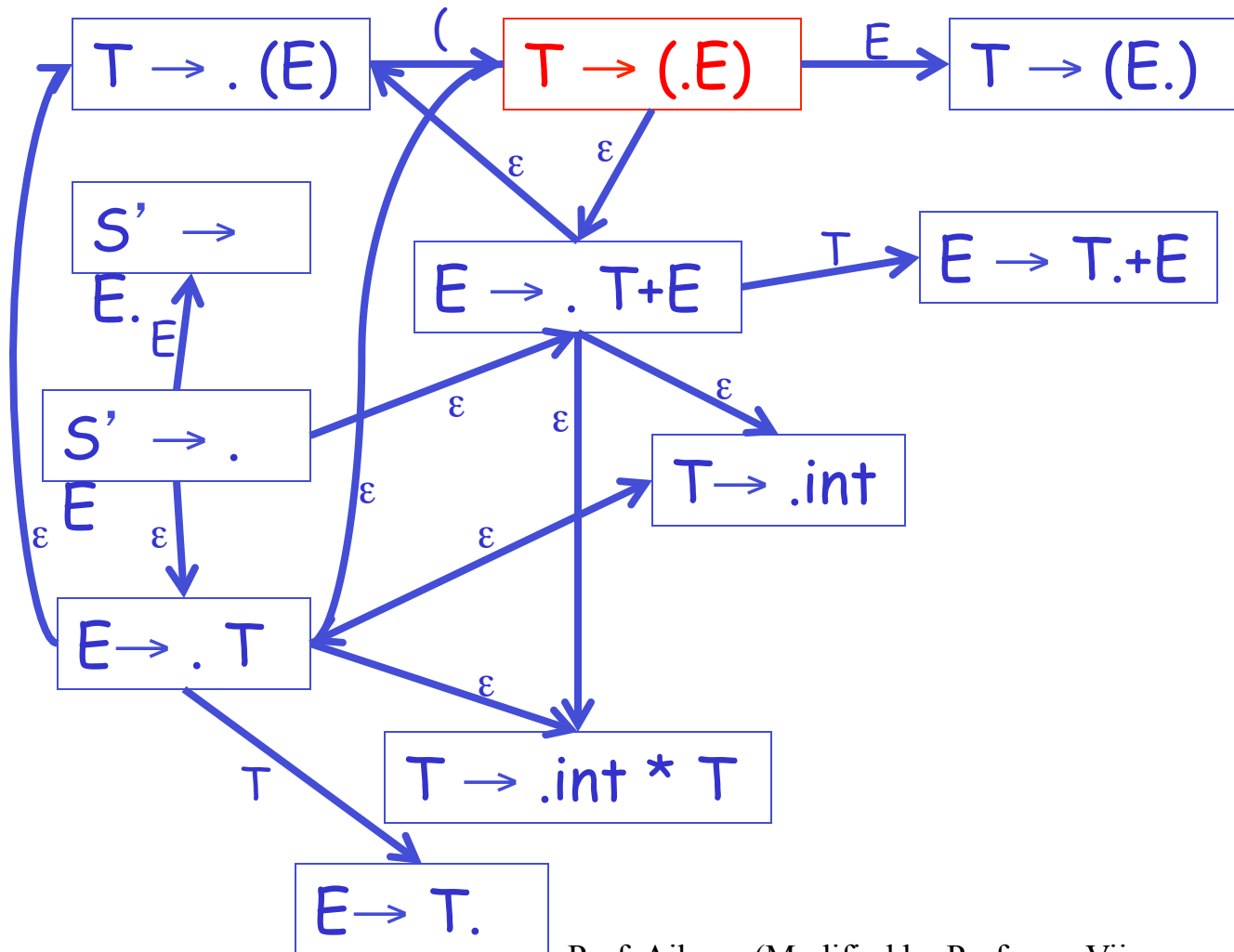
NFA for Viable Prefixes in Detail (4)



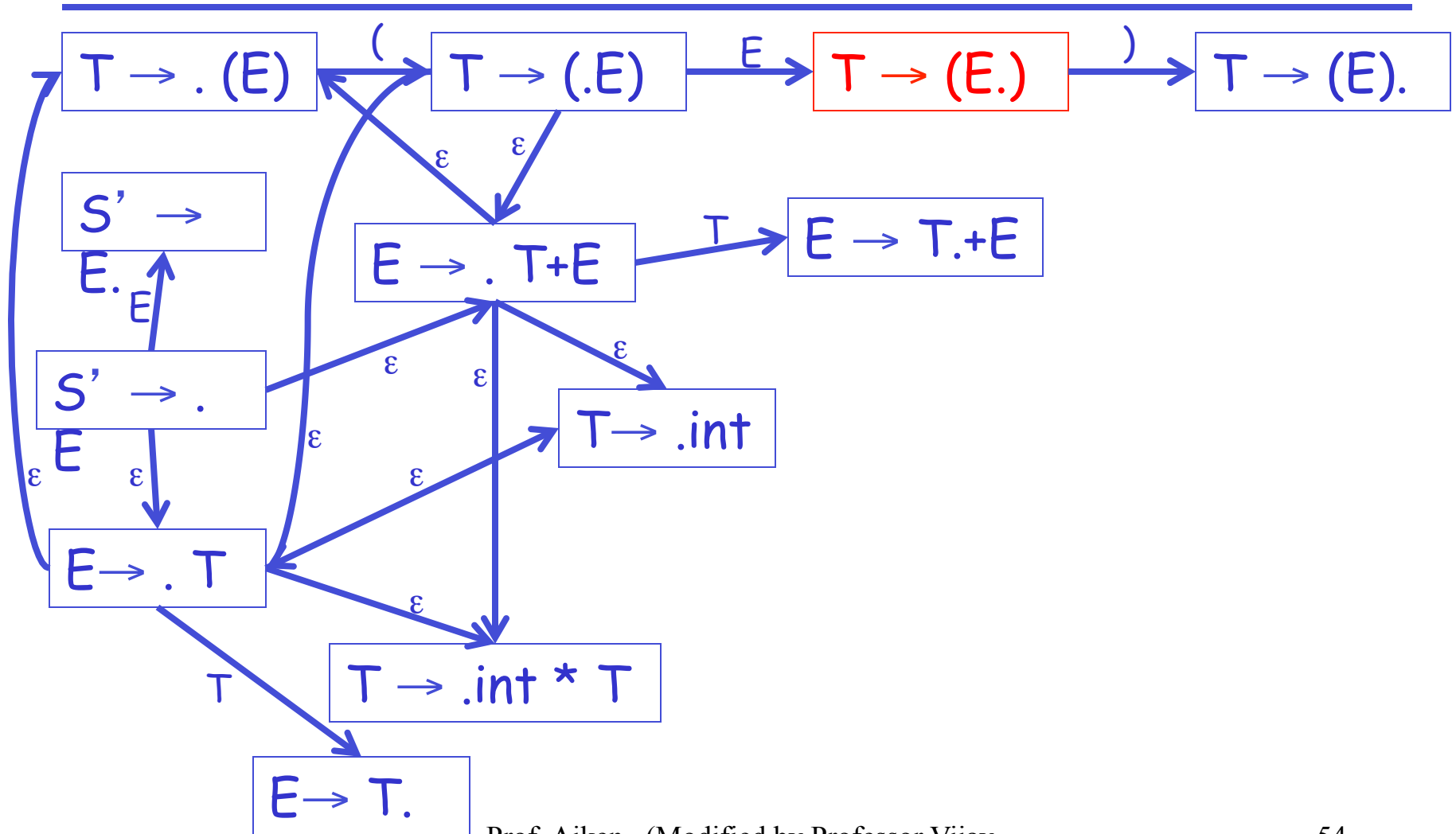
NFA for Viable Prefixes in Detail (5)



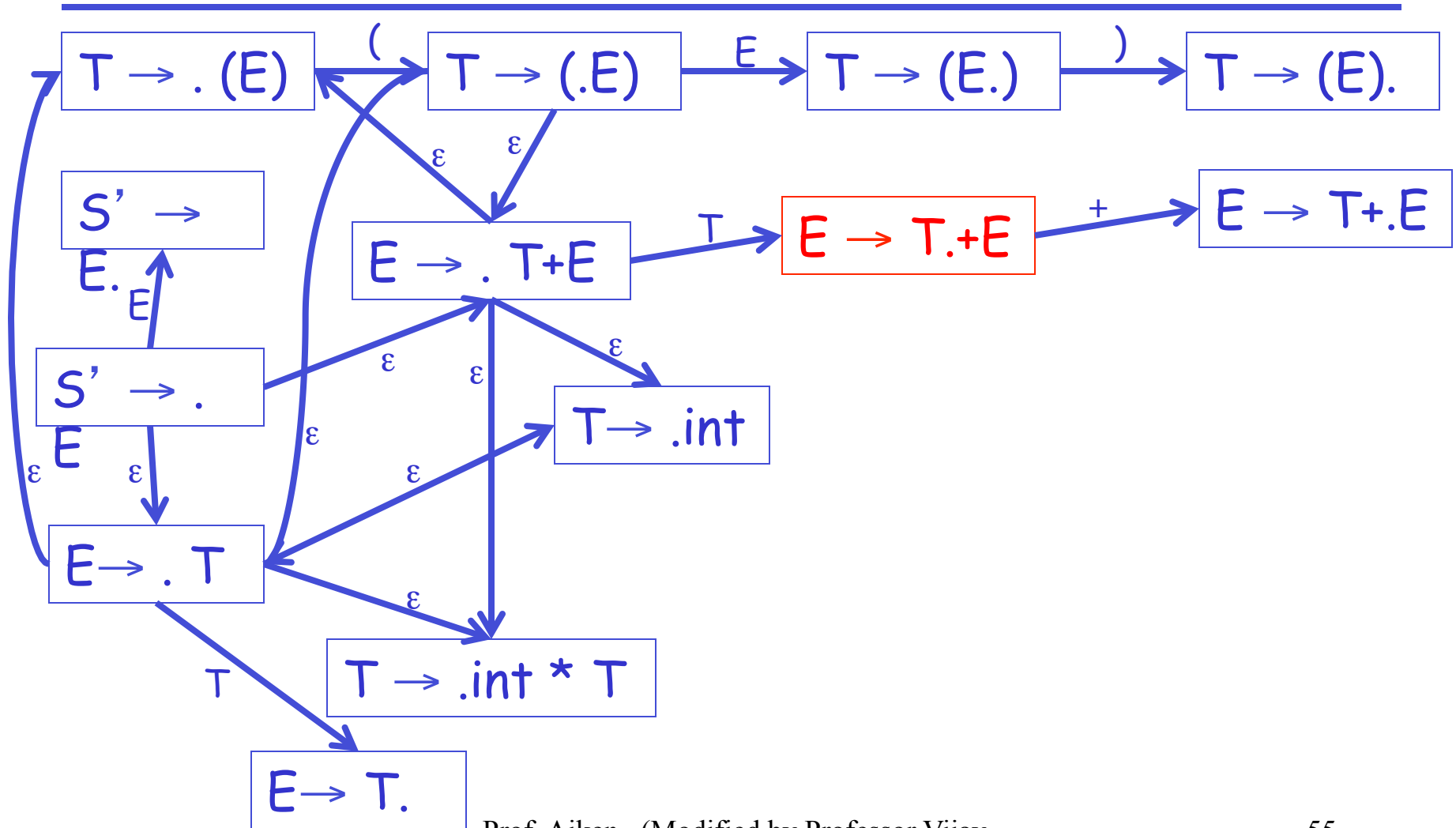
NFA for Viable Prefixes in Detail (6)



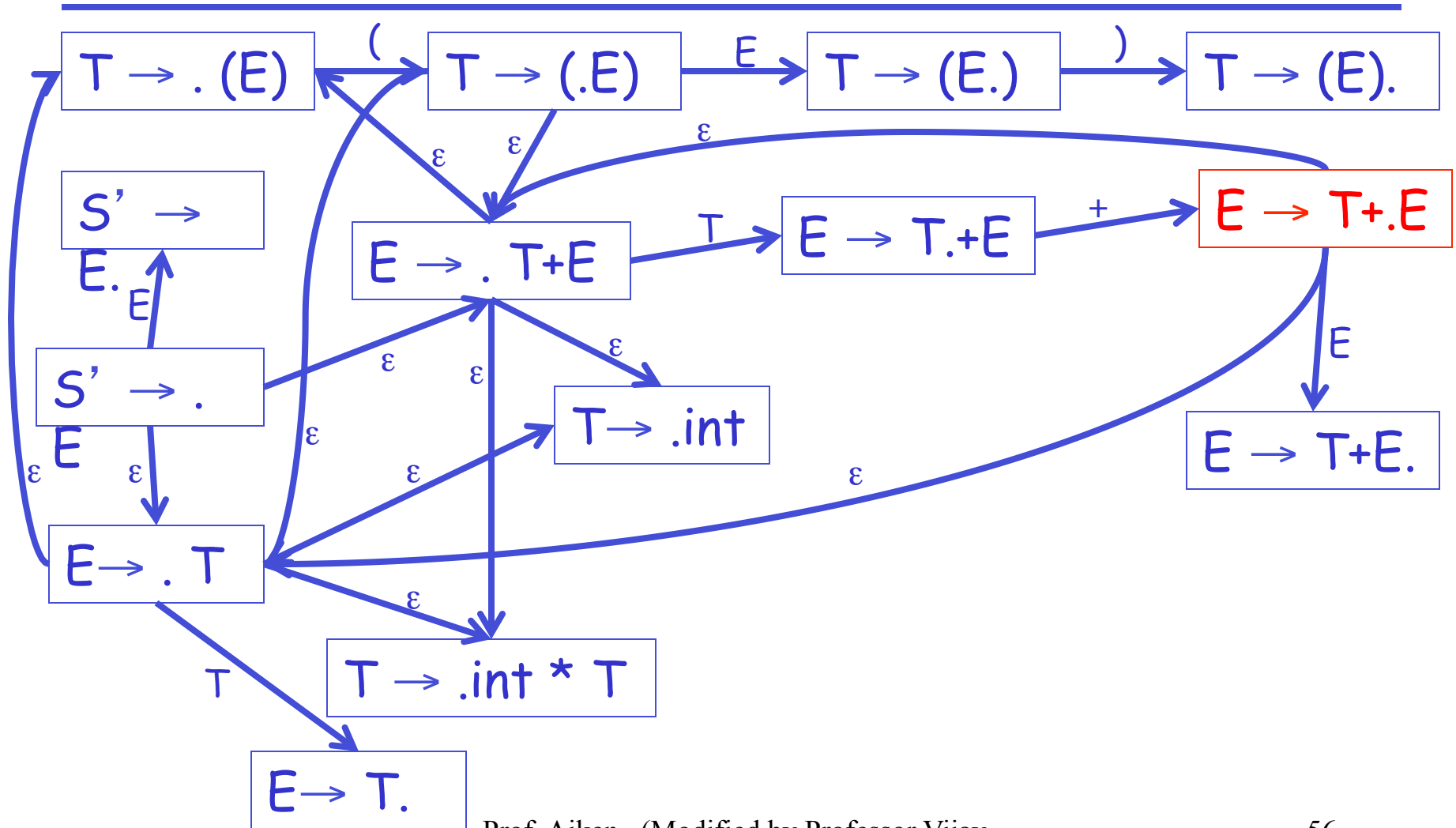
NFA for Viable Prefixes in Detail (7)



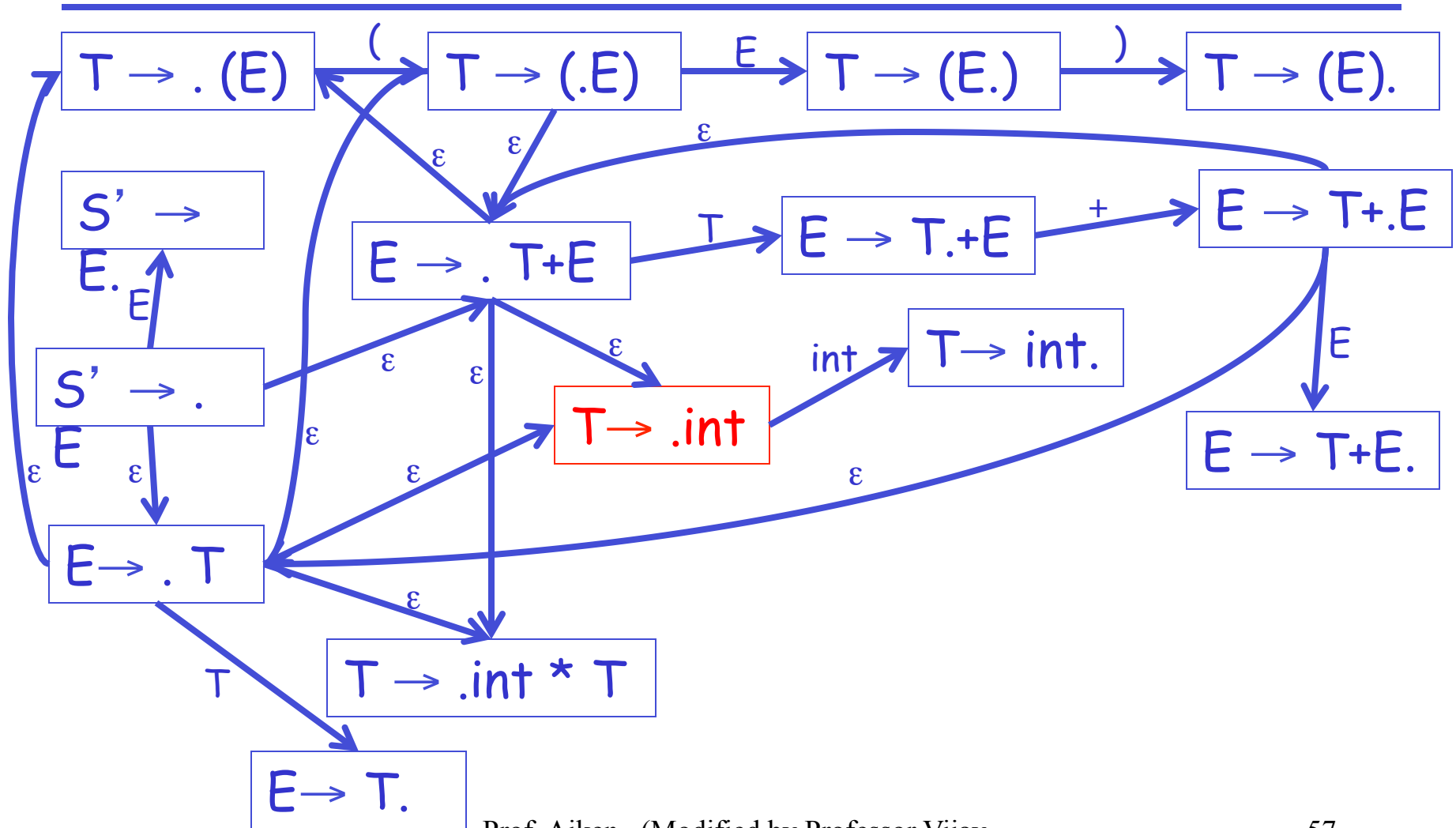
NFA for Viable Prefixes in Detail (8)



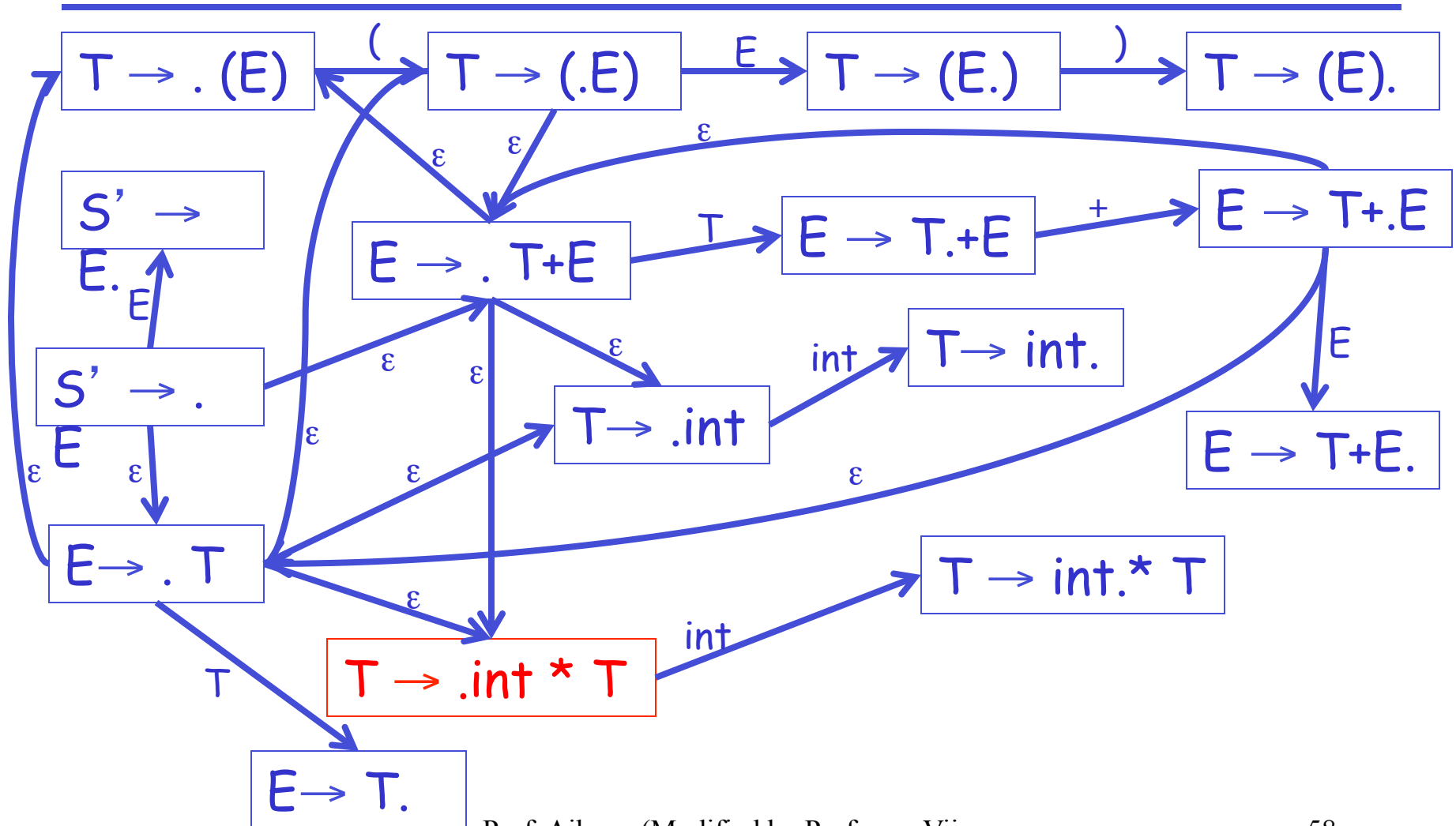
NFA for Viable Prefixes in Detail (9)



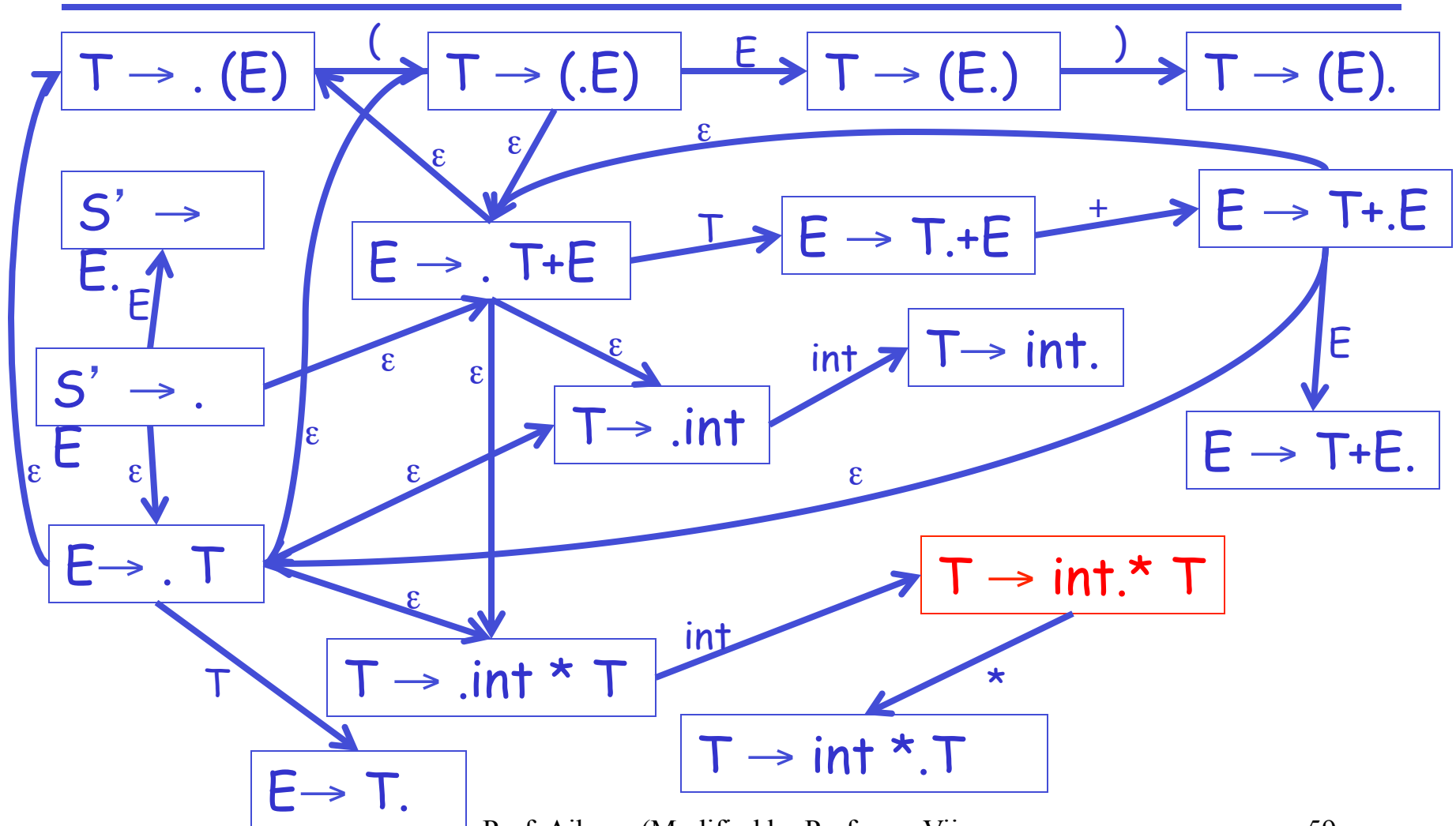
NFA for Viable Prefixes in Detail (10)



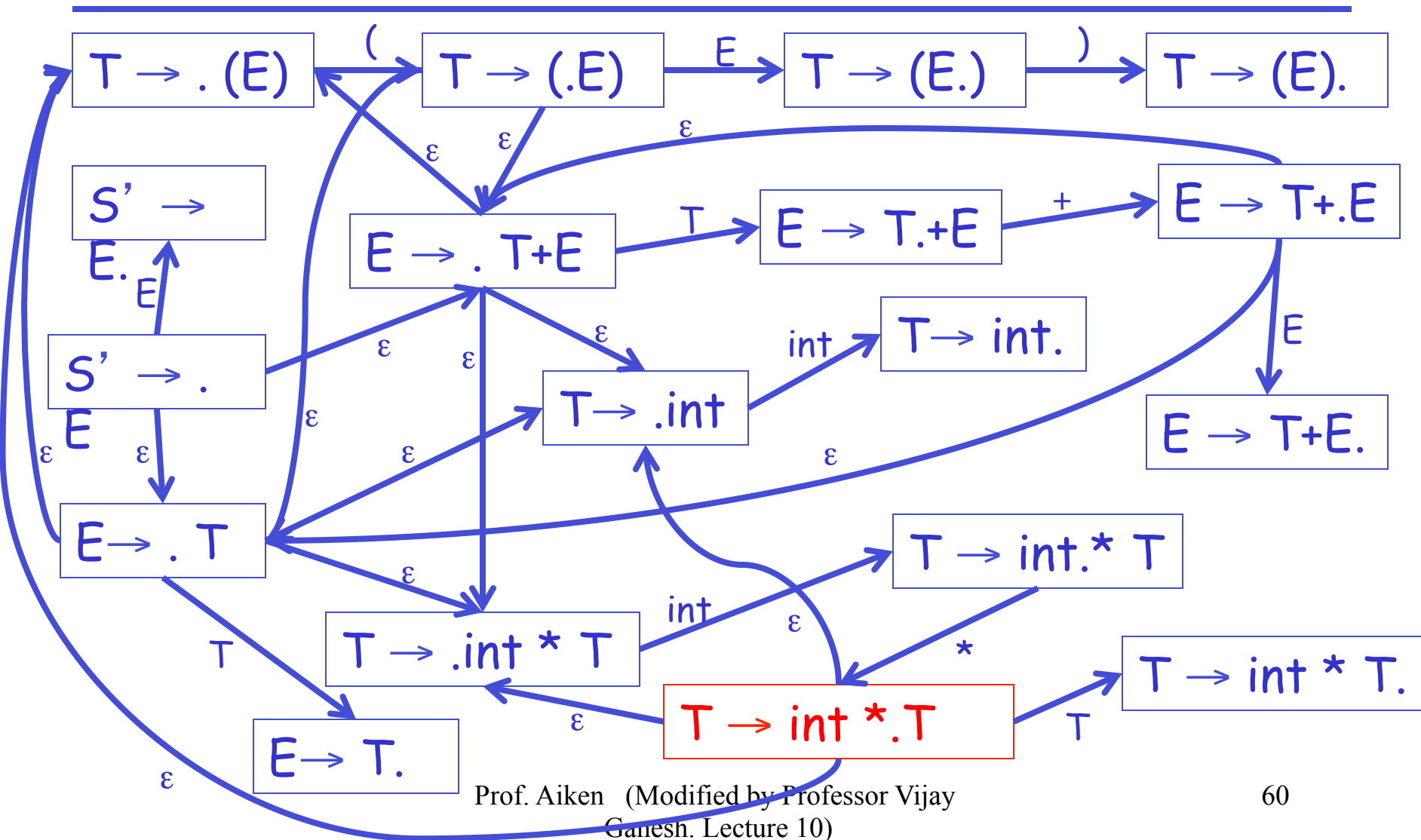
NFA for Viable Prefixes in Detail (11)



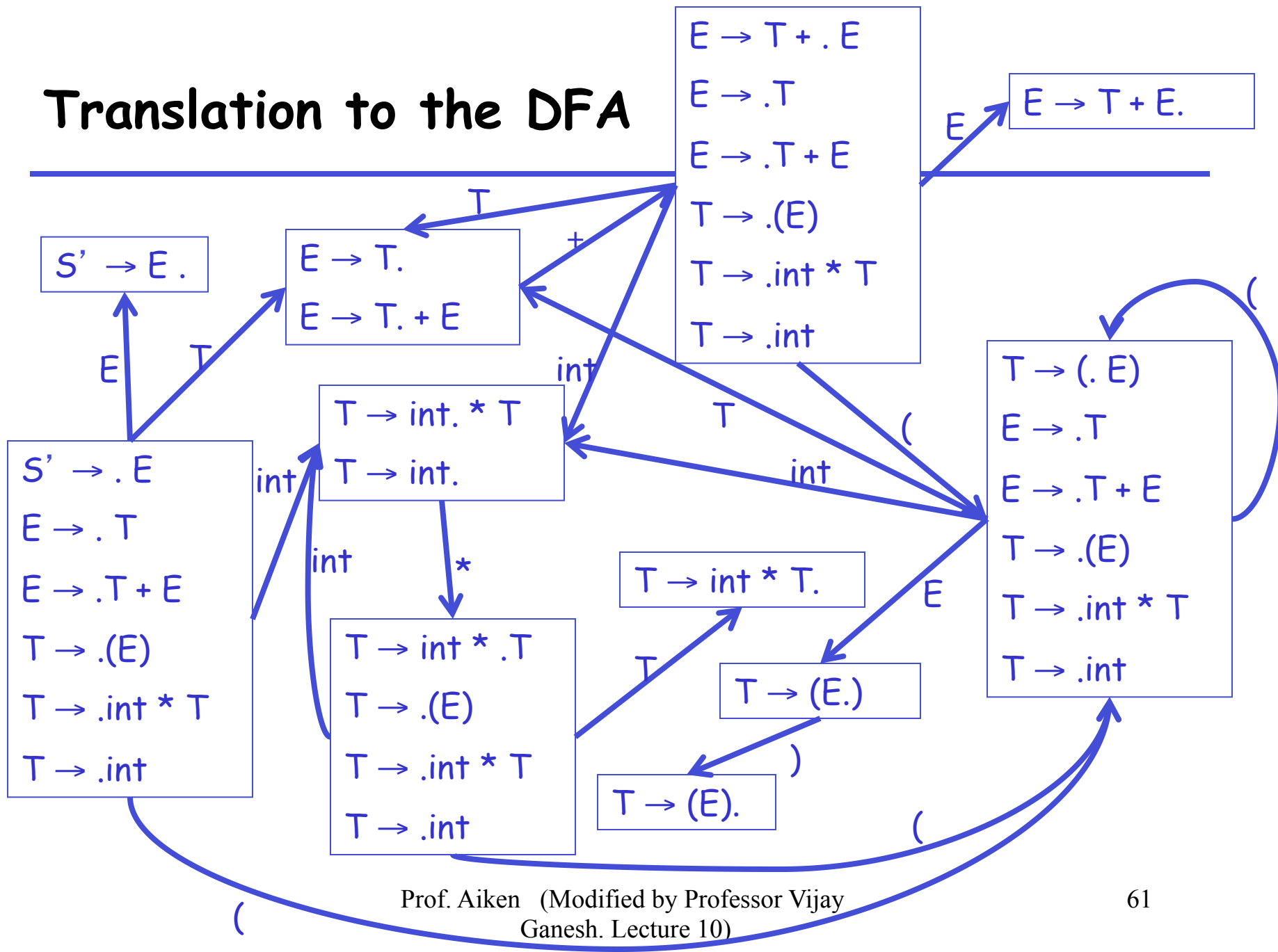
NFA for Viable Prefixes in Detail (12)



NFA for Viable Prefixes in Detail (13)



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

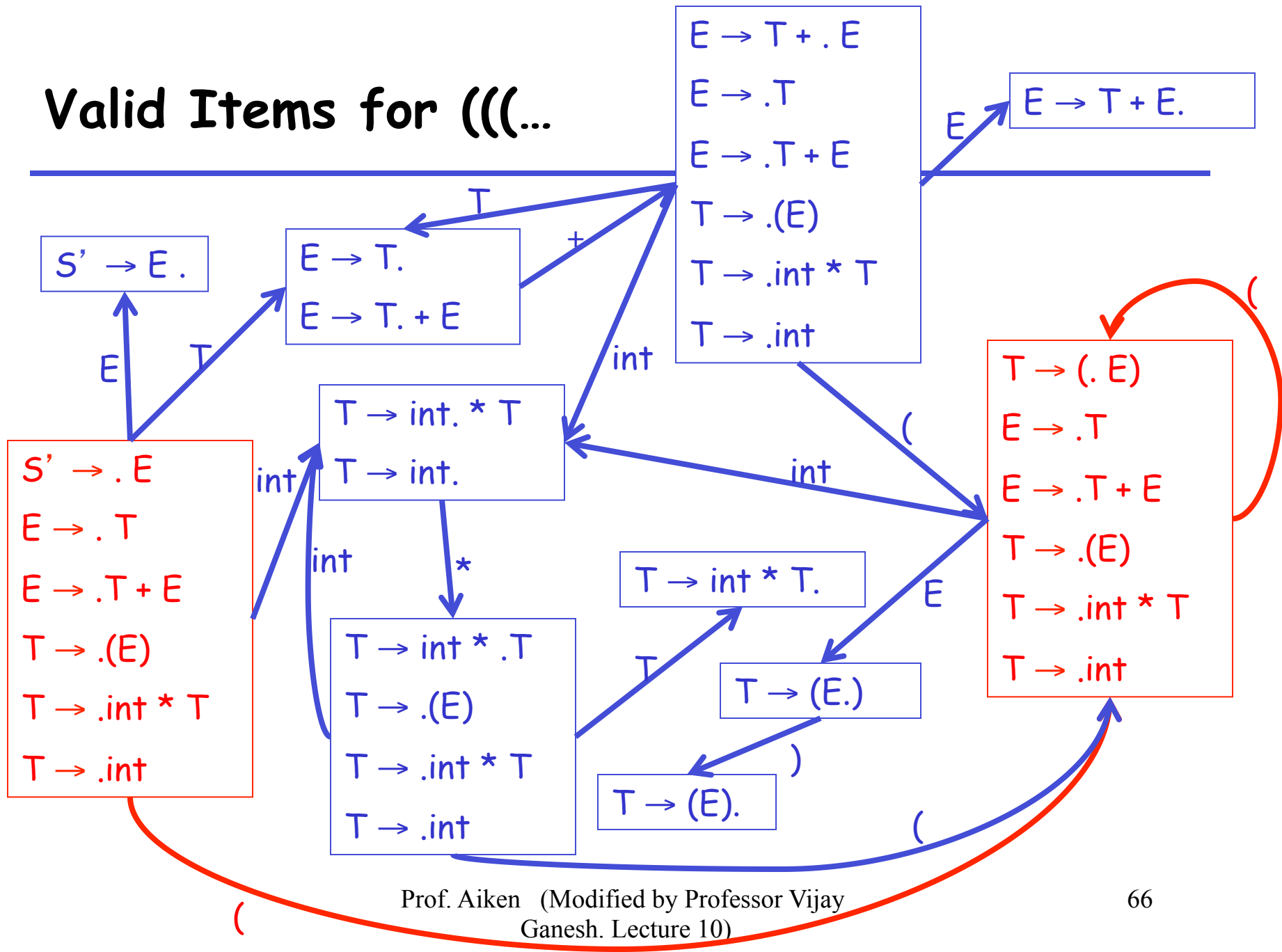
The items in s describe what the top of the item stack might be after reading input α

Valid Items Example

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

(
(
(
(
...

Valid Items for (((...



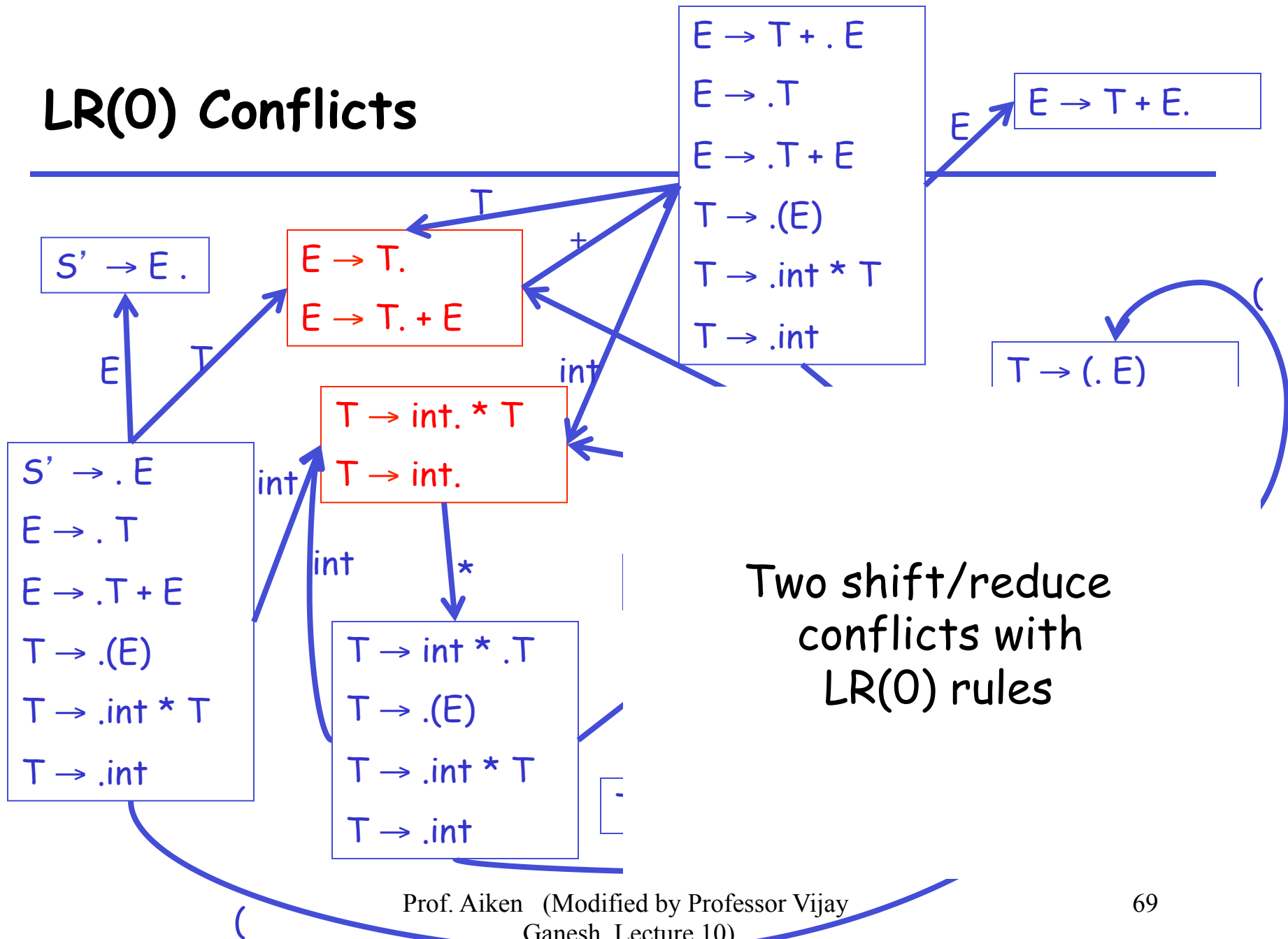
LR(0) Parsing

- 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 $Y \rightarrow \omega.t\delta$


LR(0) Conflicts



SLR

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

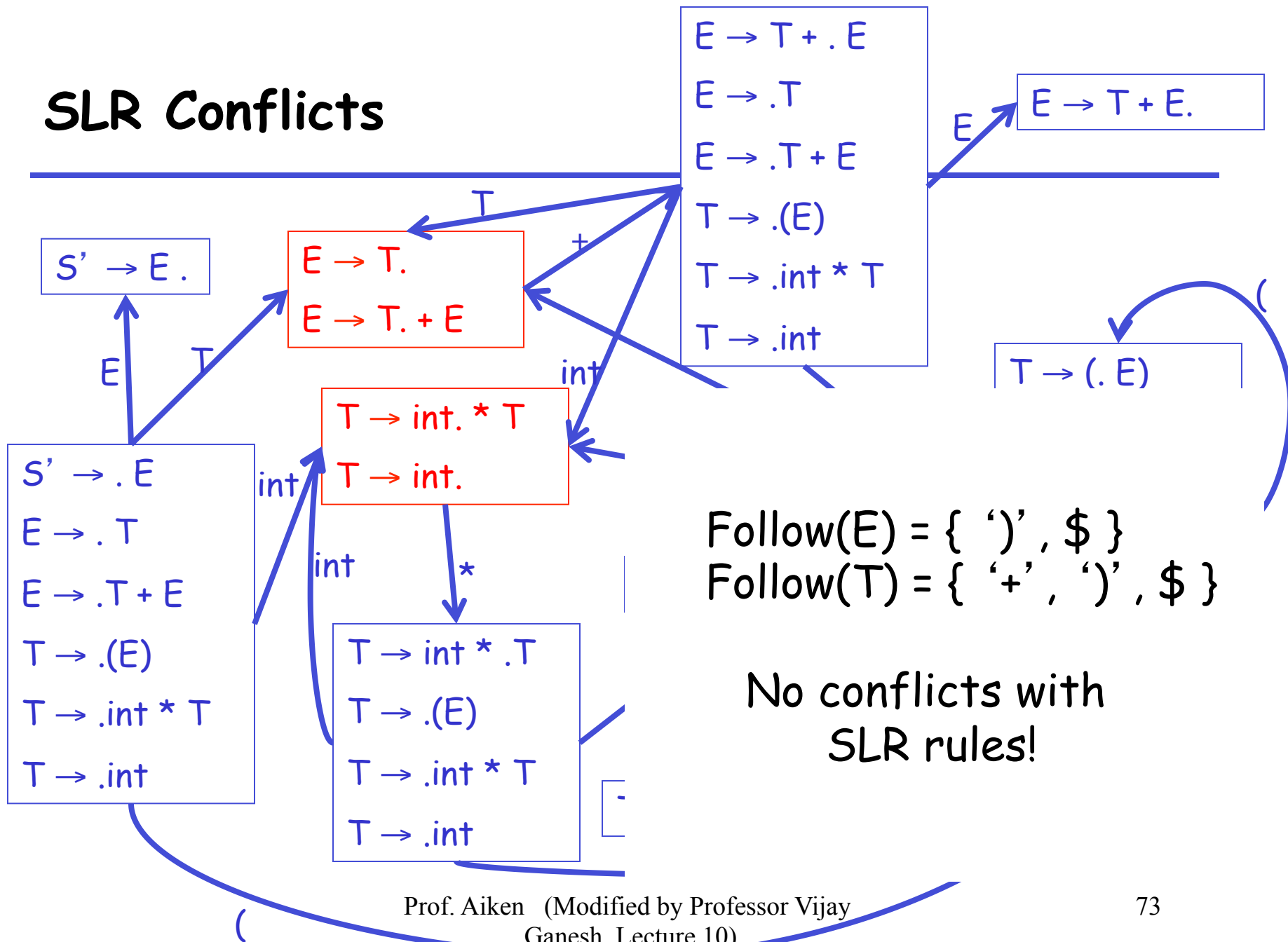
SLR Parsing

- 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

SLR Conflicts



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:
 - $E \rightarrow E * E . \quad E \rightarrow E . + E$
 - 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!

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 $S| \$$
 - 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 a be next input
 - Shift if $X \rightarrow \beta. a \gamma \in I$
 - Reduce if $X \rightarrow \beta. \in I$ and $a \in \text{Follow}(X)$
 - Report parsing error if neither applies

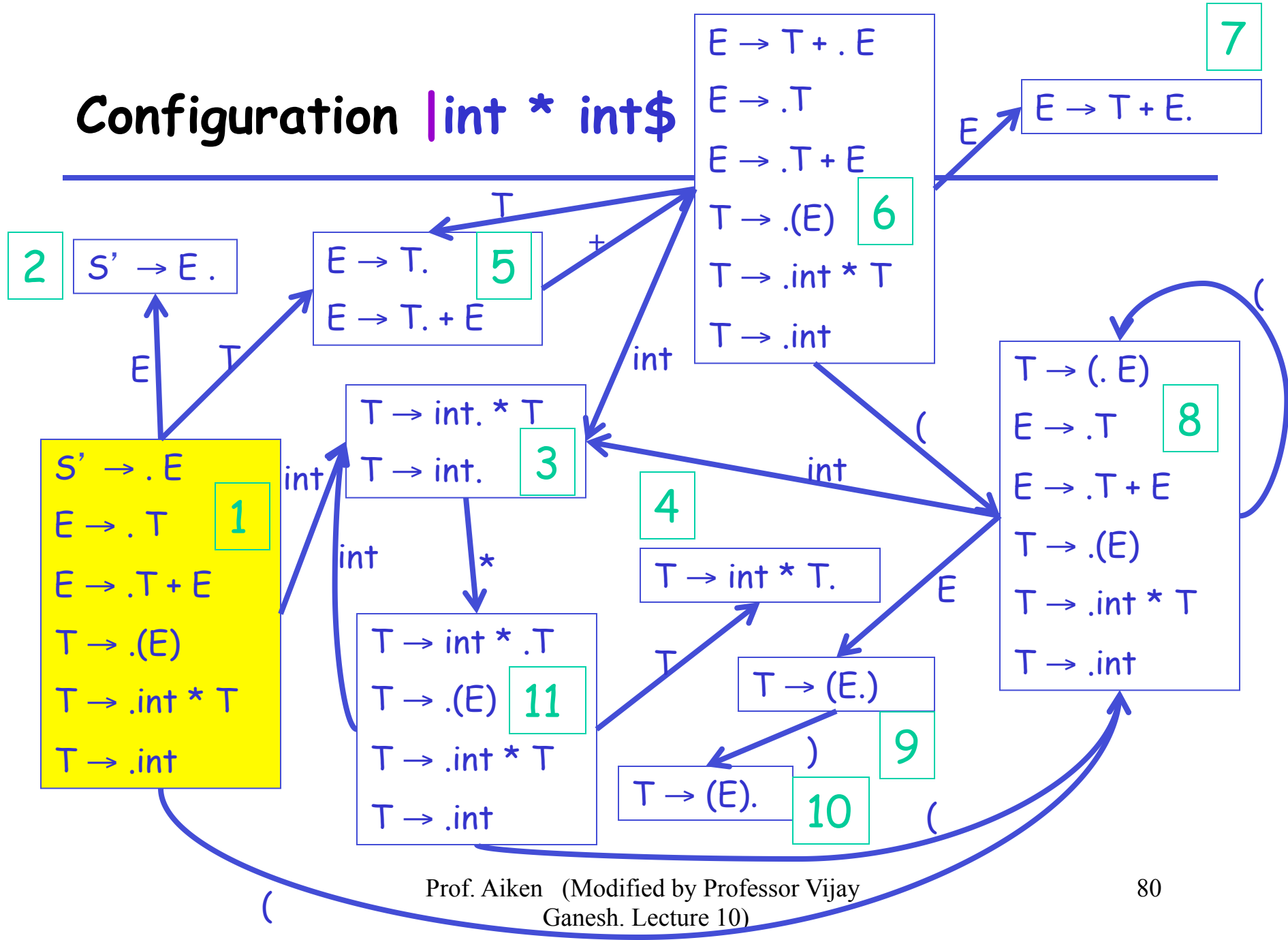
Notes

- If there is a conflict in the last step, grammar is not SLR(k)
- k is the amount of lookahead
 - In practice $k = 1$

SLR Example

<i>Configuration</i>	<i>DFA Halt State</i>	<i>Action</i>
int * int\$	1	shift

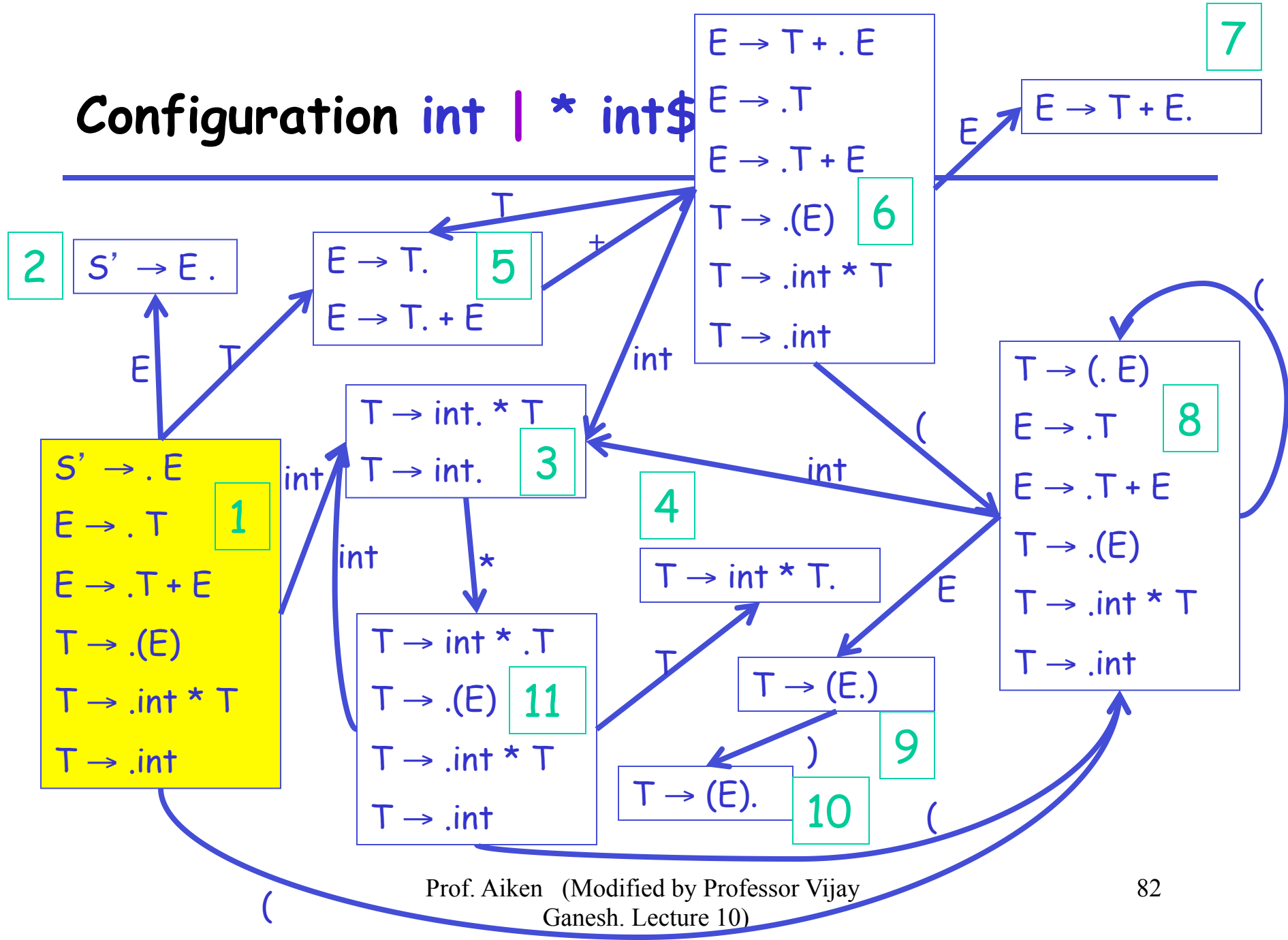
Configuration **|int * int\$**



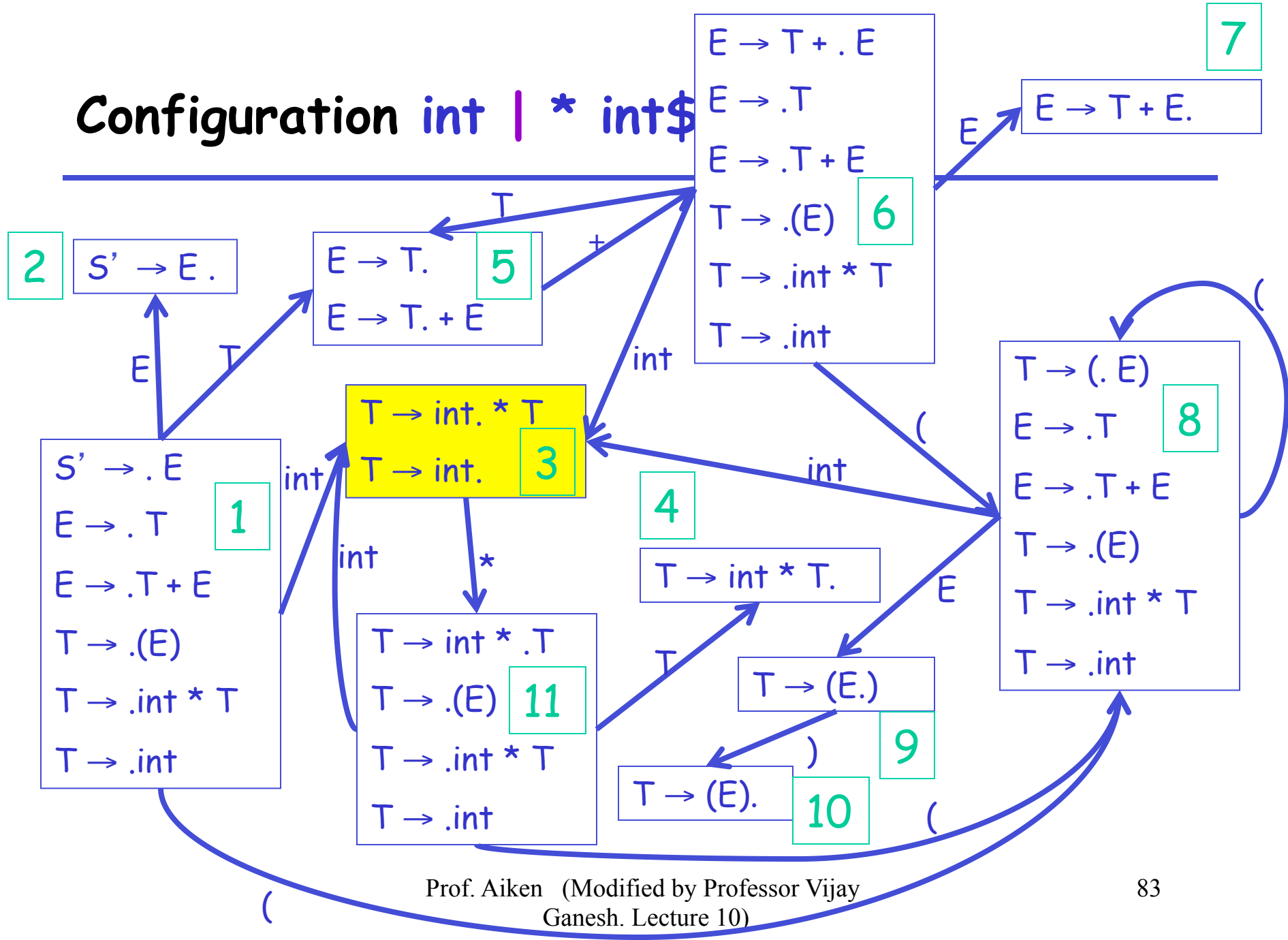
SLR Example

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

Configuration **int | * int\$**



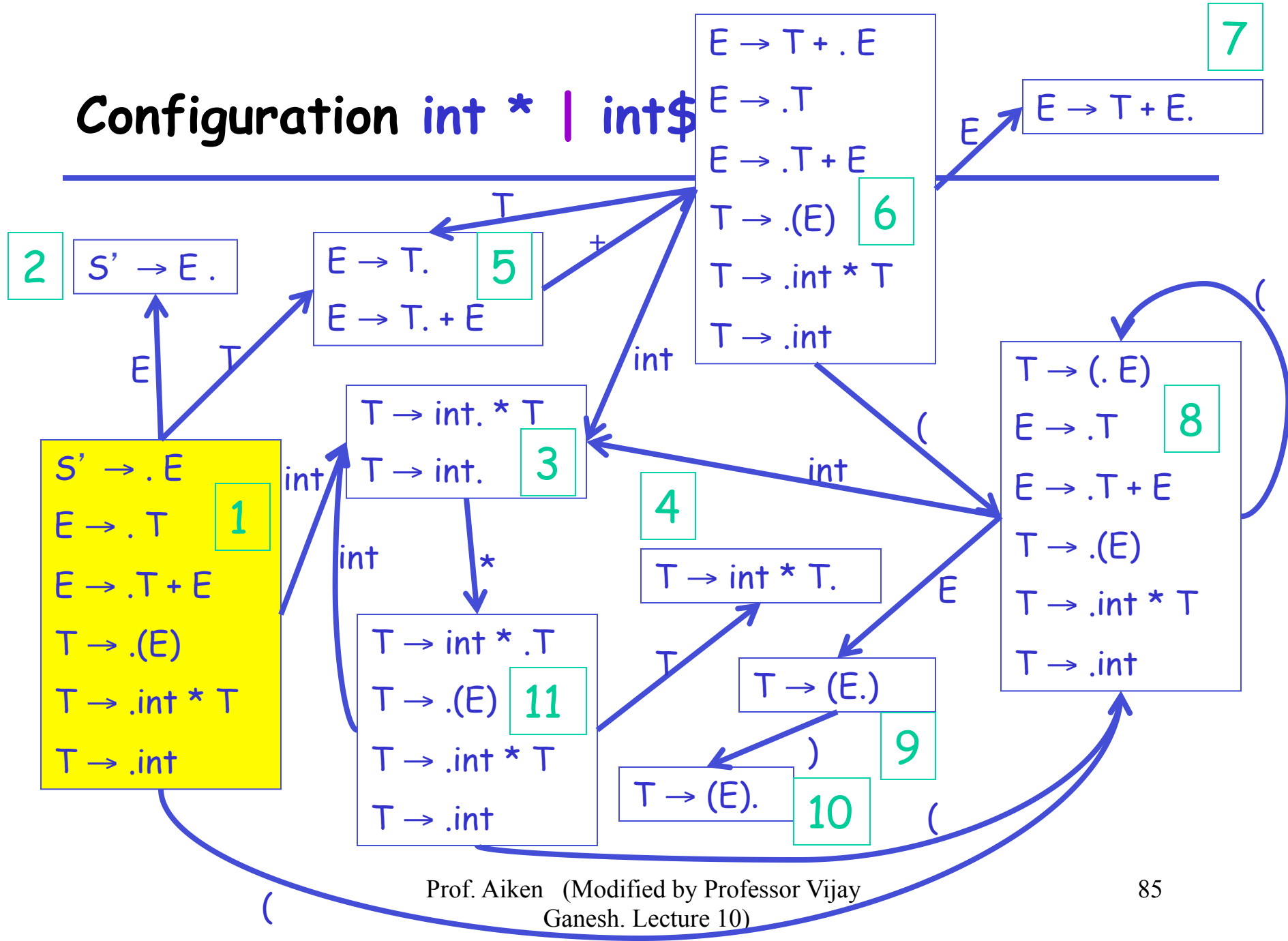
Configuration **int | * int\$**



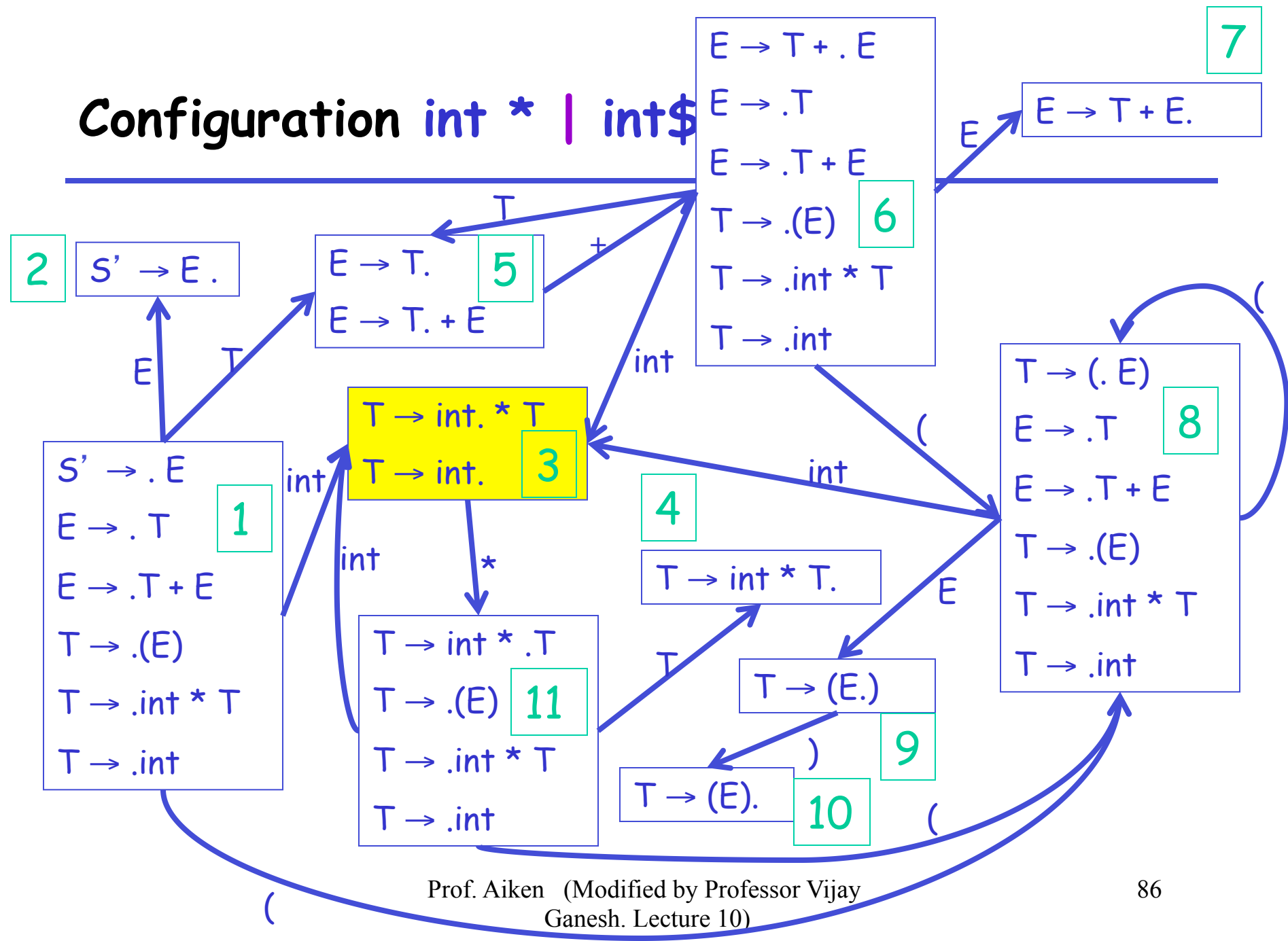
SLR Example

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

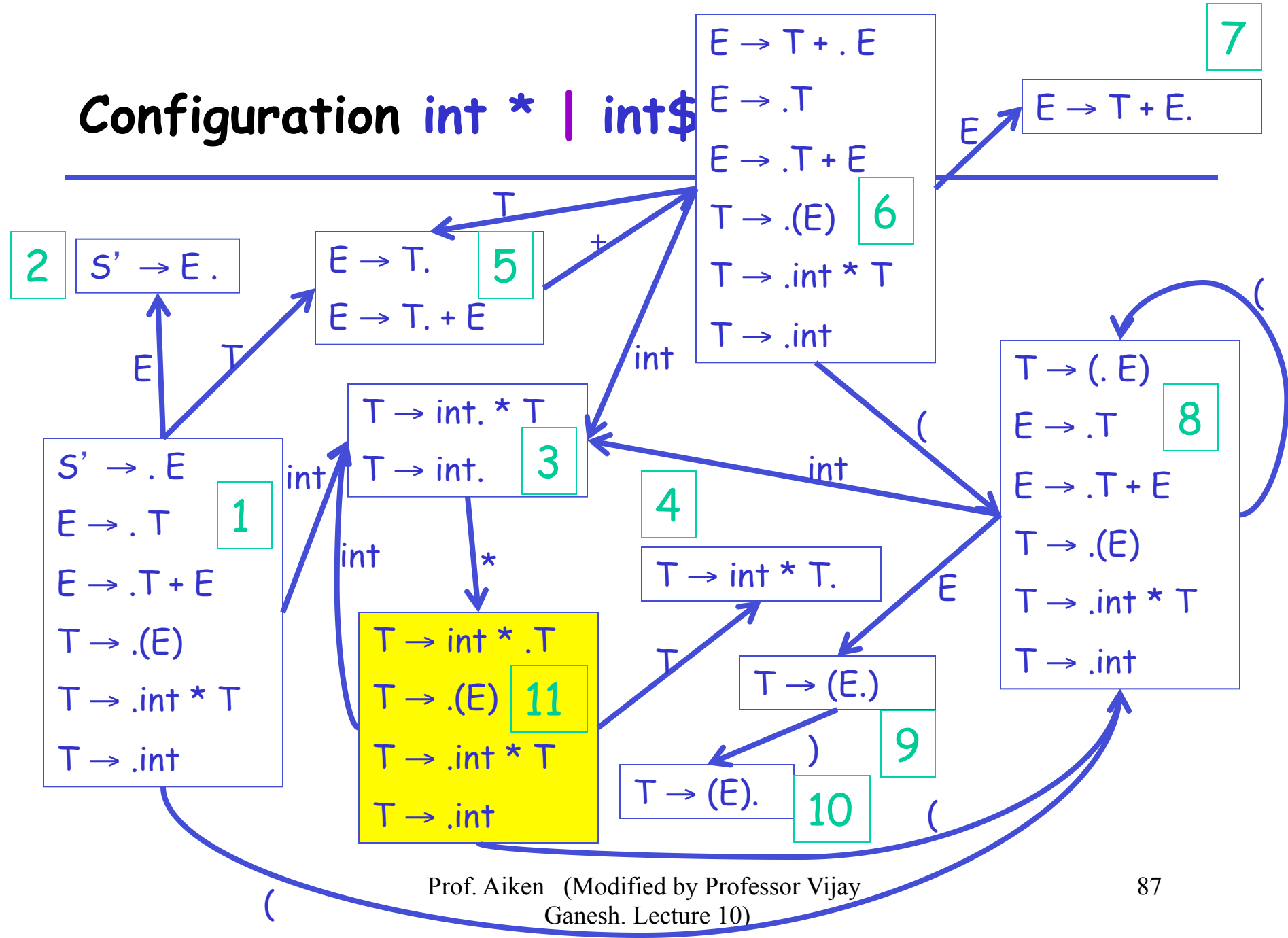
Configuration **int * | int\$**



Configuration $\text{int} * \mid \text{int}\$$



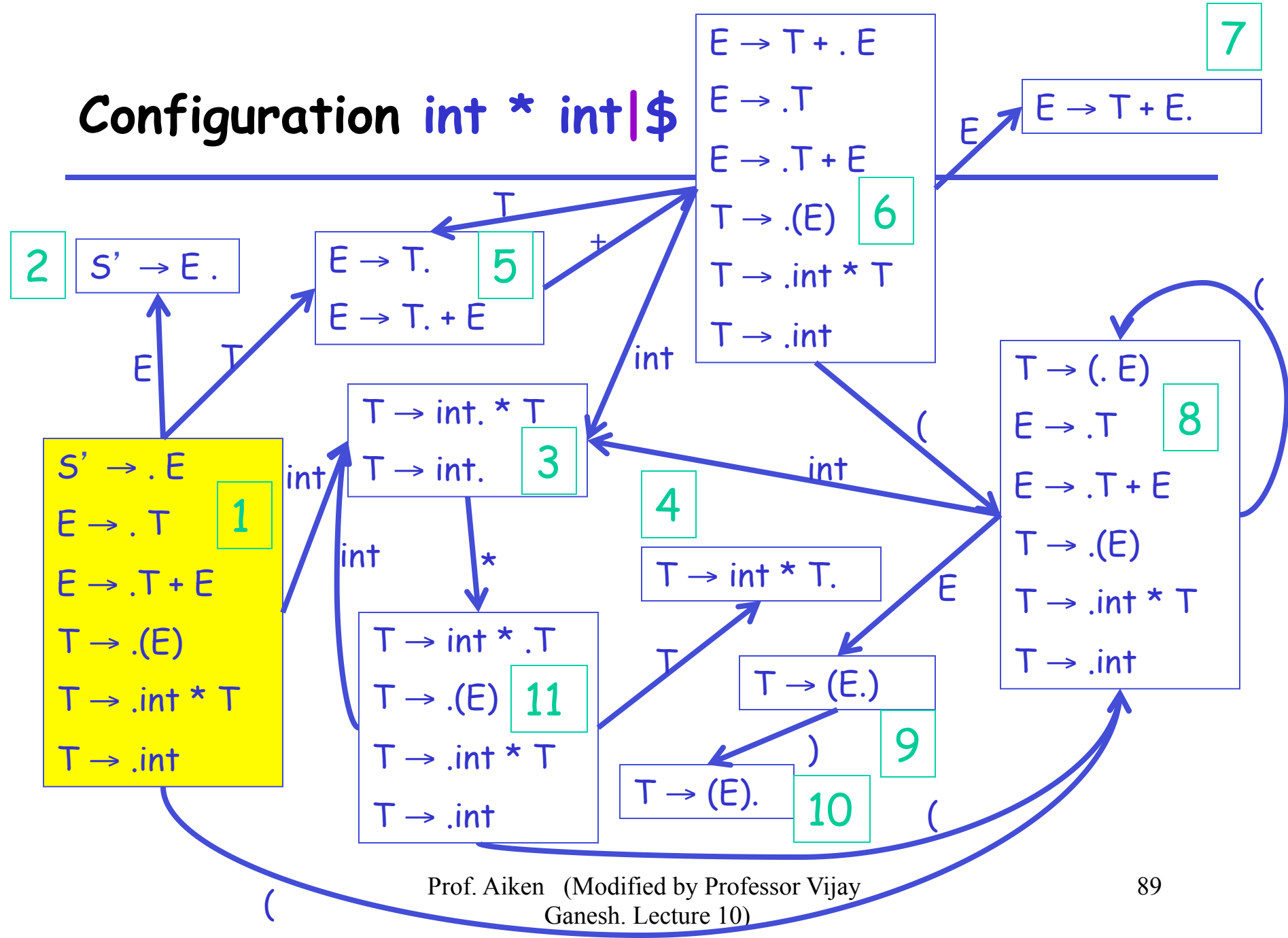
Configuration $\text{int} * \mid \text{int}\$$



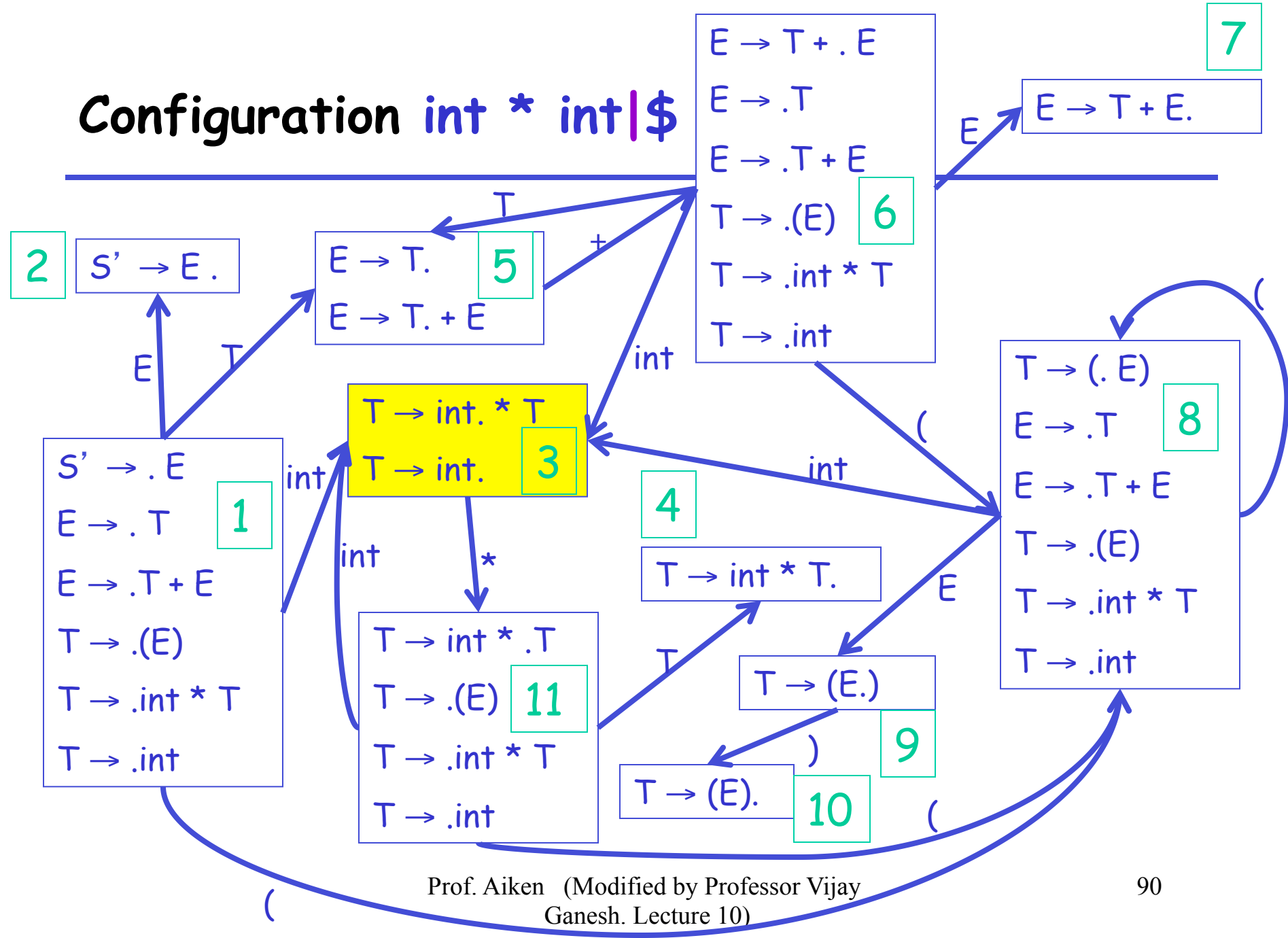
SLR Example

<i>Configuration</i>	<i>DFA Halt State</i>	<i>Action</i>
int * int\$	1	shift
int * int\$	3 * not in Follow(T)	shift
int * int\$	11	shift
int * int \$	3 $\$ \in \text{Follow}(T)$	red. $T \rightarrow \text{int}$

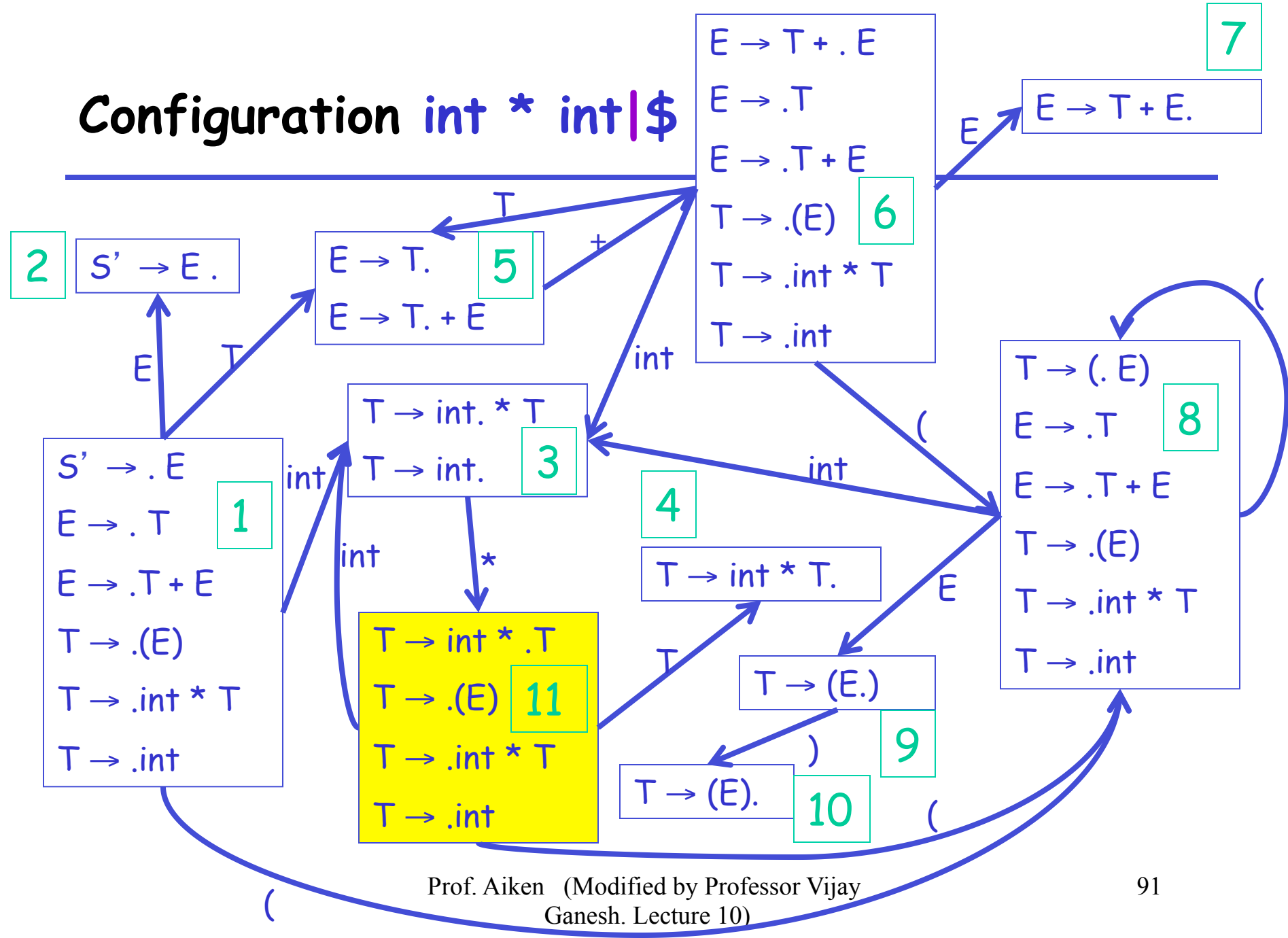
Configuration $\text{int} * \text{int} | \$$



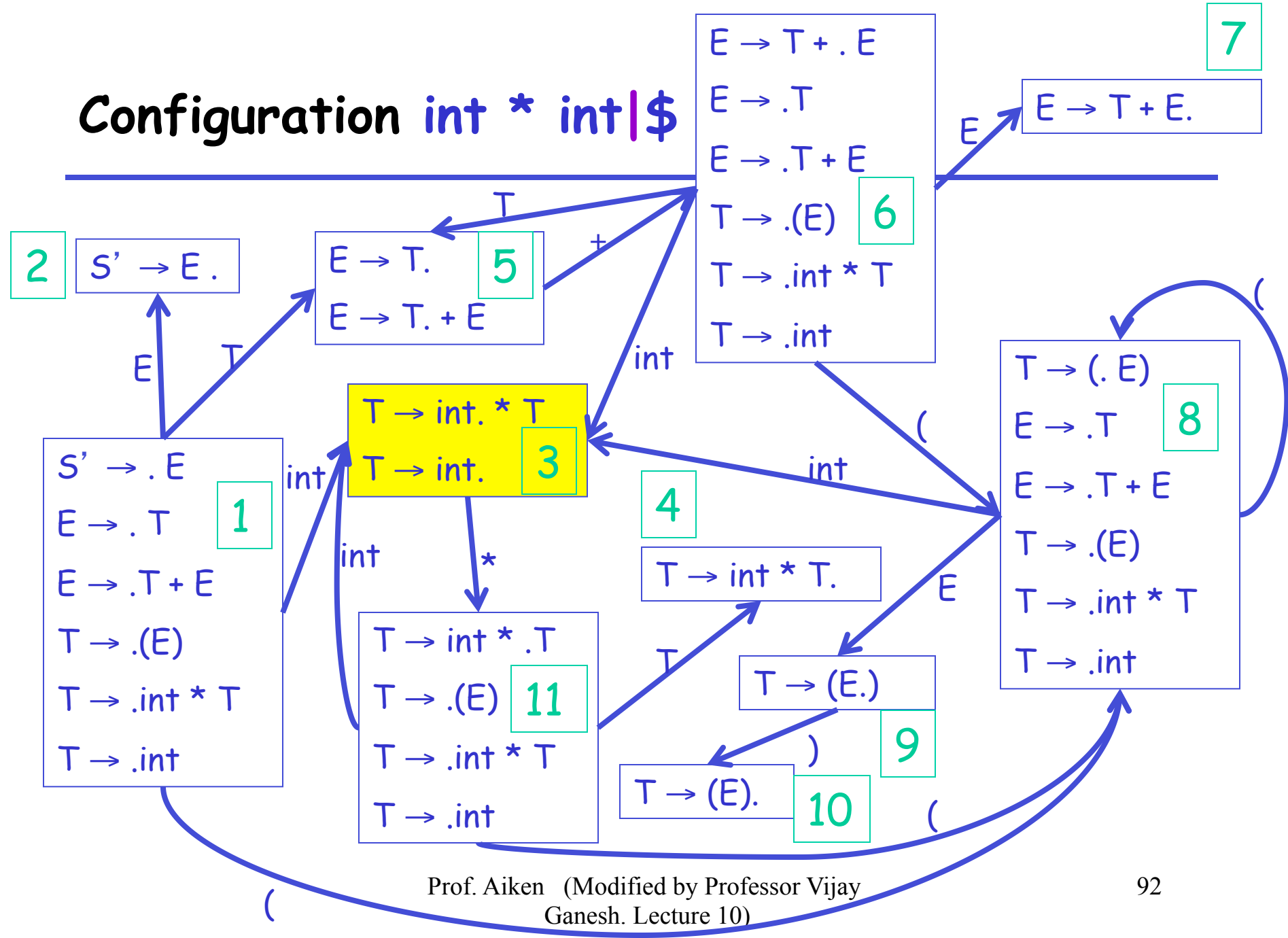
Configuration $\text{int} * \text{int} | \$$



Configuration $\text{int} * \text{int} | \$$



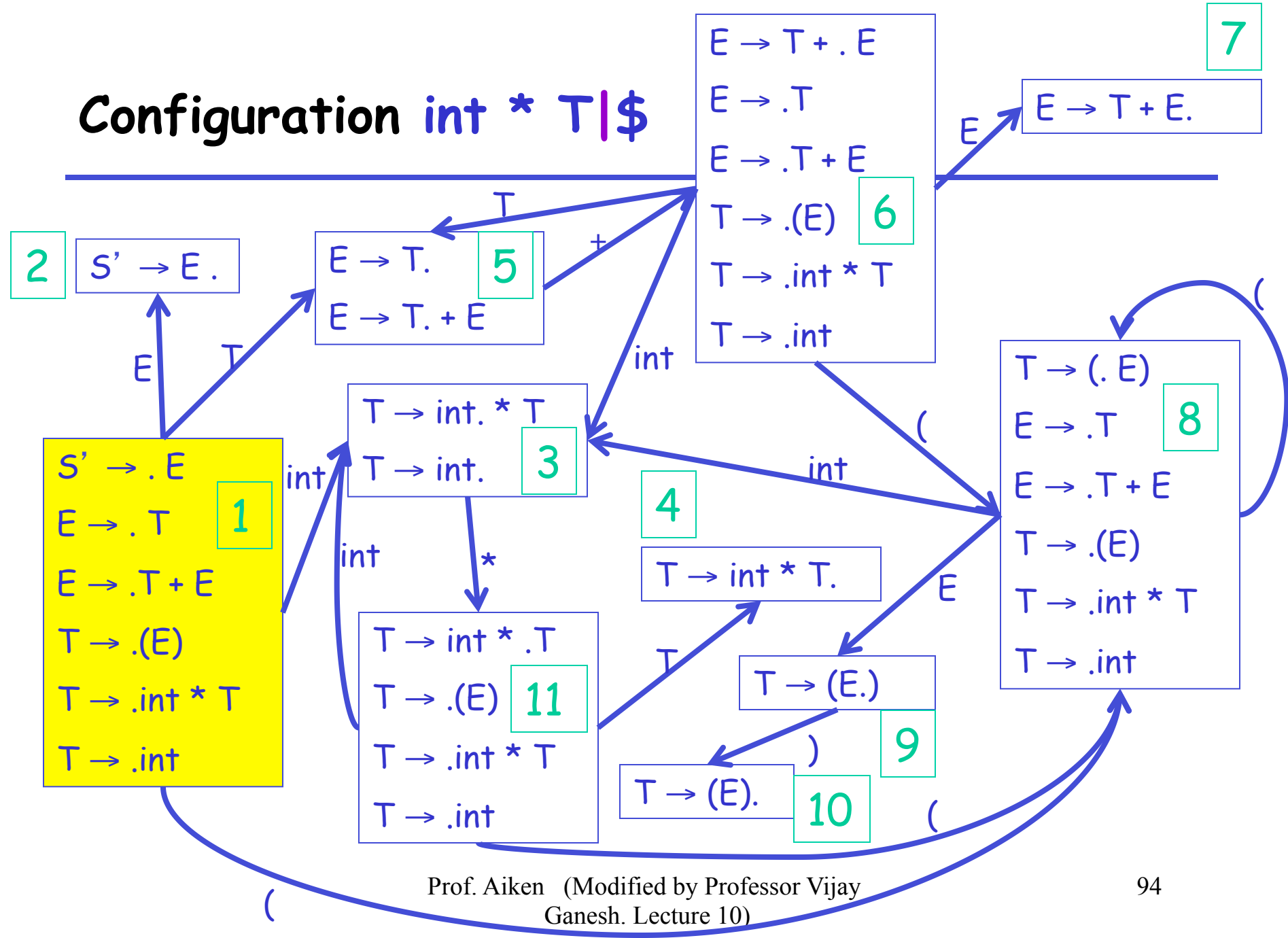
Configuration $\text{int} * \text{int} | \$$



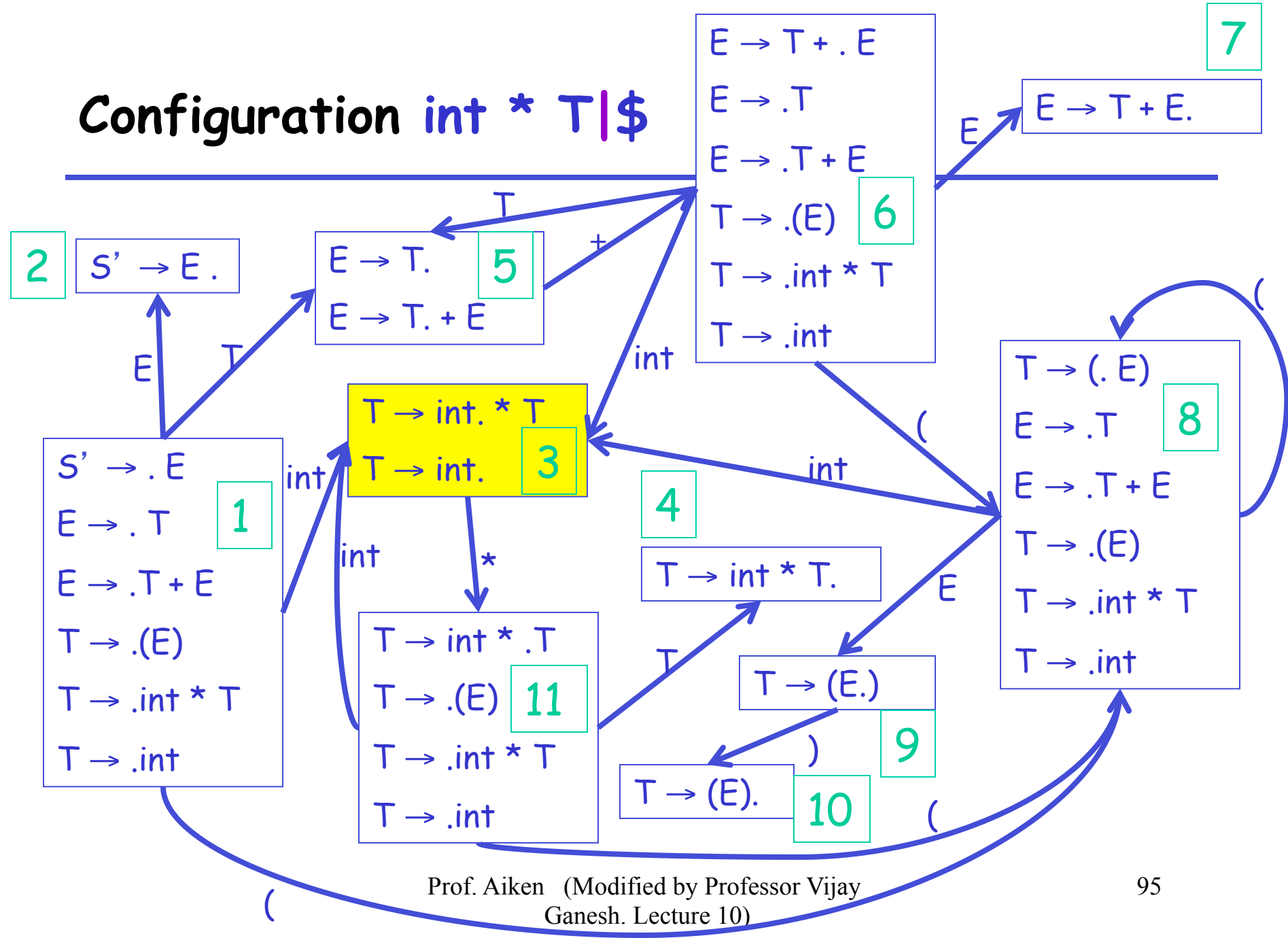
SLR Example

<i>Configuration</i>	<i>DFA Halt State</i>	<i>Action</i>
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 \$	4 \$ ∈ Follow(T)	red. T → int * T

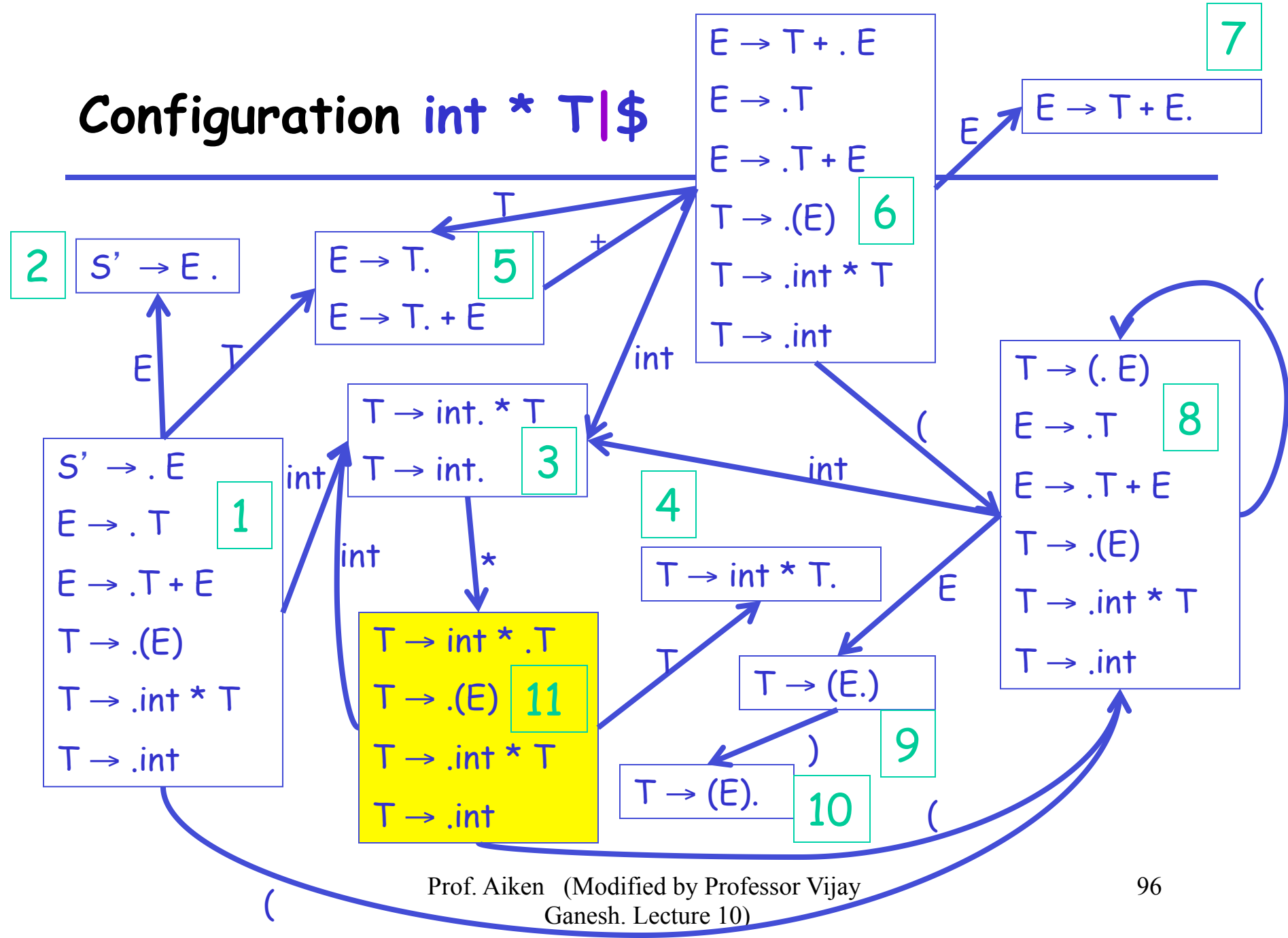
Configuration $\text{int} * T | \$$



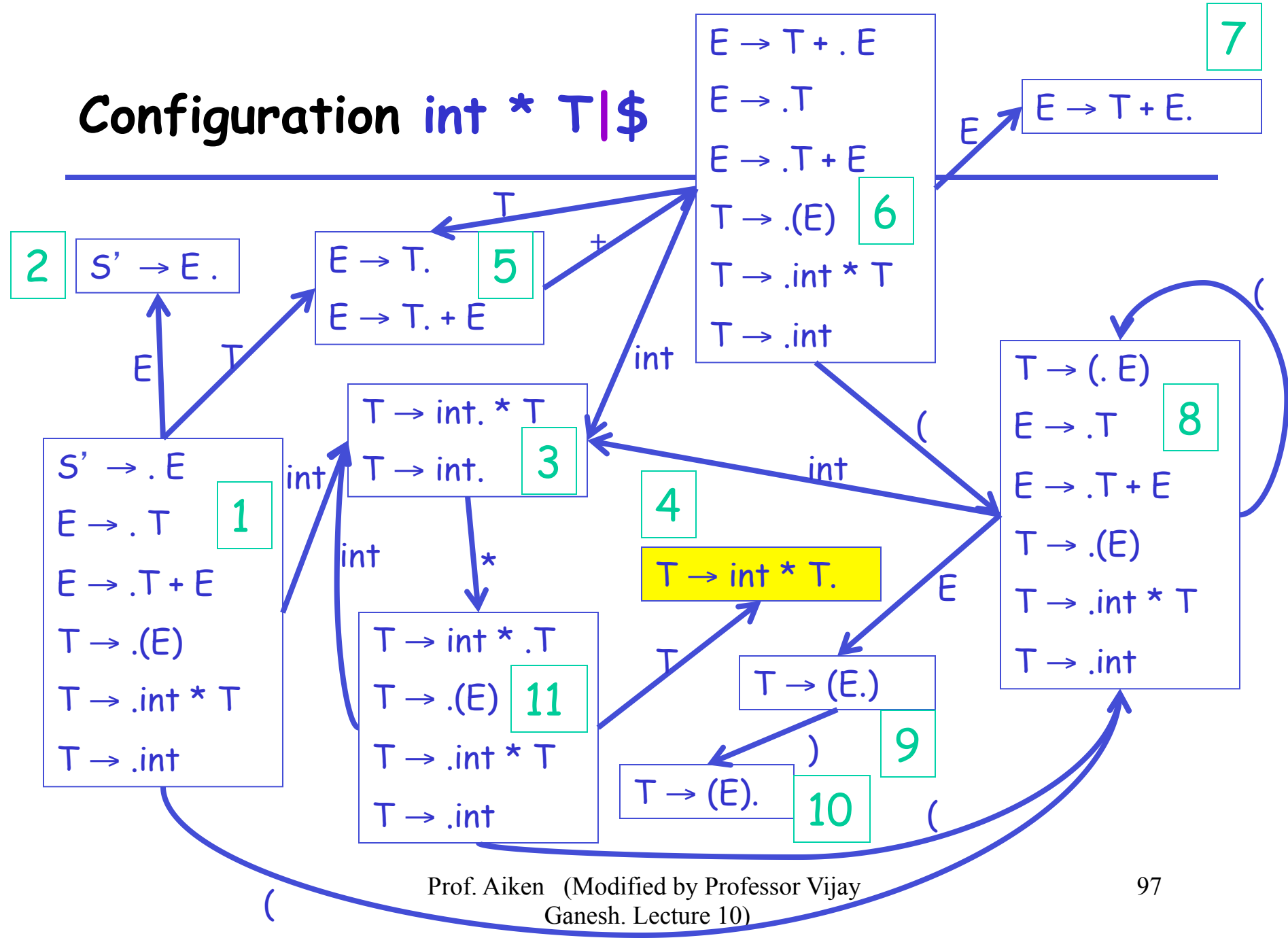
Configuration $\text{int} * T | \$$



Configuration $\text{int} * T | \$$



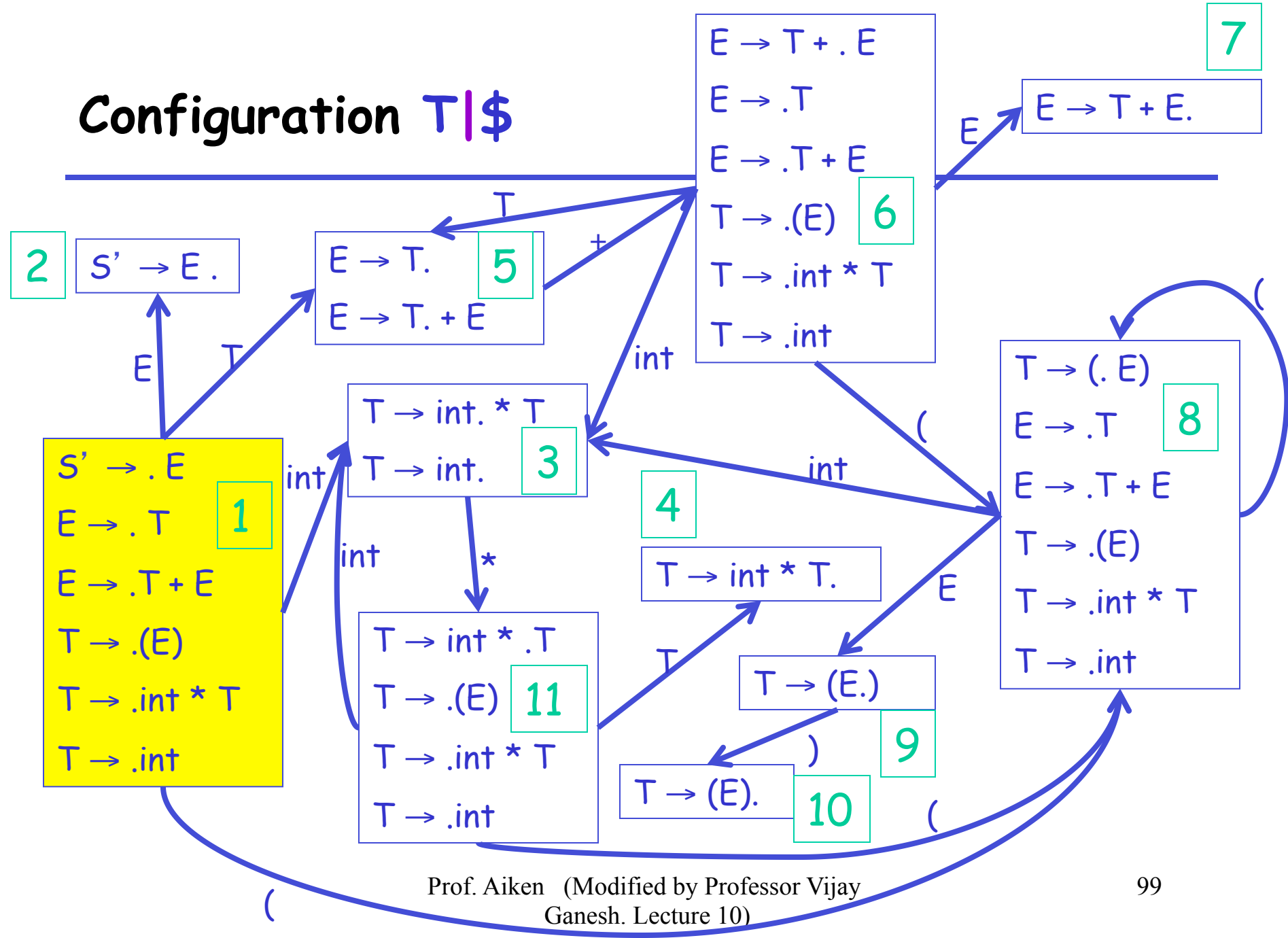
Configuration $\text{int} * T | \$$



SLR Example

<i>Configuration</i>	<i>DFA Halt State</i>	<i>Action</i>
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 \$	4 \$ ∈ Follow(T)	red. T → int * T
T \$	5 \$ ∈ Follow(E)	red. E → T

Configuration $T| \$$



7



SLR Example

<i>Configuration</i>	<i>DFA Halt State</i>	<i>Action</i>
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 \$	4 \$ ∈ Follow(T)	red. T → int * T
T \$	5 \$ ∈ Follow(T)	red. E → T
E \$		accept

Notes

- Skipped using extra start state S' in this example to save space on slides
- 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
 $\langle \text{Symbol, DFA State} \rangle$

An Improvement (Cont.)

- For a stack

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

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

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

Goto 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 a

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

SLR Parsing Algorithm

Let $I = w\$$ be initial input

Let $j = 0$

Let DFA state 1 have item $S' \rightarrow .S$

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

repeat

case $\text{action}[\text{top_state}(\text{stack}), I[j]]$ of

shift k : push $\langle I[j++], k \rangle$

reduce $X \rightarrow A$:

pop $|A|$ 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
 - Take a look at the LR(1) automaton for your parser!