Bottom-Up Parsing II

Lecture 8

Review: Shift-Reduce Parsing

Bottom-up parsing uses two actions:

Shift

ABC|xyz ⇒ ABCx|yz

Reduce

Cbxy|ijk ⇒ CbA|ijk

Recall: The Stack

- Left string can be implemented by a stack
  - Top of the stack is the |
- Shift pushes a terminal on the stack
- Reduce
  - pops 0 or more symbols off of the stack
    - production rhs
  - pushes a non-terminal on the stack
    - production lhs

Key Issue

- How do we decide when to shift or reduce?
- Example grammar:
  - $E \to T + E \mid T$
  - $T \to int * T \mid int \mid (E)$
- Consider step int | * int + int
  - We could reduce by $T \to int$ giving T | * int + int
  - A fatal mistake!
    - No way to reduce to the start symbol E

Handles

- Intuition: Want to reduce only if the result can still be reduced to the start symbol
- Assume a rightmost derivation

  $$S \to^* \alpha X \omega \to \alpha\beta\omega$$
- Then $X \to \beta$ 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

Why?

- Informal induction on # of reduce moves:
- True initially, stack is empty
- Immediately after reducing a handle
  - right-most non-terminal on top of the stack
  - next handle must be to right of right-most non-terminal, because this is a right-most derivation
  - Sequence of shift moves reaches next handle

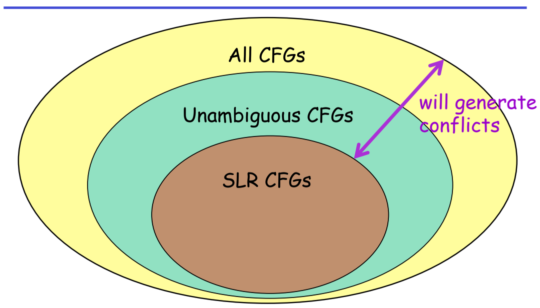
Summary of Handles

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

Recognizing Handles

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

Grammars



Viable Prefixes

- It is not obvious how to detect handles
- At each step the parser sees only the stack, not the entire input; start with that . . .

α is a viable prefix if there is an ω such that $\alpha|\omega$ is a state of a shift-reduce parser

Huh?

- What does this mean? A few things:
  - A viable prefix does not extend past the right end of the handle
  - It's a viable prefix because it is a prefix of the handle
  - As long as a parser has viable prefixes on the stack no parsing error has been detected

Important Fact #3

Important Fact #3 about bottom-up parsing:

For any grammar, the set of viable prefixes is a regular language

Important Fact #3 (Cont.)

- Important Fact #3 is non-obvious
- We show how to compute automata that accept viable prefixes

Items

- An item is a production with a "." somewhere on the rhs
- The items for $T \to (E)$ are
  $T \to .(E)$
  $T \to (.E)$
  $T \to (E.)$
  $T \to (E).$

Items (Cont.)

- The only item for $X \to \varepsilon$ is $X \to .$
- 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 $T \to (E)$
  - 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

  $Prefix_1\ Prefix_2\ \ldots\ Prefix_{n-1} Prefix_n$
- 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 α_{i-1} starts with X_i
  - i.e. there is a $X_{i-1} \to Prefix_{i-1}\ X_i\ \beta$ for some β
- Recursively, $Prefix_{k+1} \ldots Prefix_n$ eventually reduces to the missing part of α_k

An Example

Consider the string (int * int):

(int *|int) is a state of a shift-reduce parse

"(" is a prefix of the rhs of $T \to (E)$

"ε" is a prefix of the rhs of $E \to T$

"int *" is a prefix of the rhs of $T \to int * T$

An Example (Cont.)

The "stack of items"

$T \to (.E)$

$E \to .T$

$T \to int * .T$

Says

We've seen "(" of $T \to (E)$

We've seen ε of $E \to T$

We've seen int * of $T \to int * T$

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 $S' \to S$ to G
2. The NFA states are the items of G
   - Including the extra production
3. For item $E \to \alpha.X\beta$ add transition

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

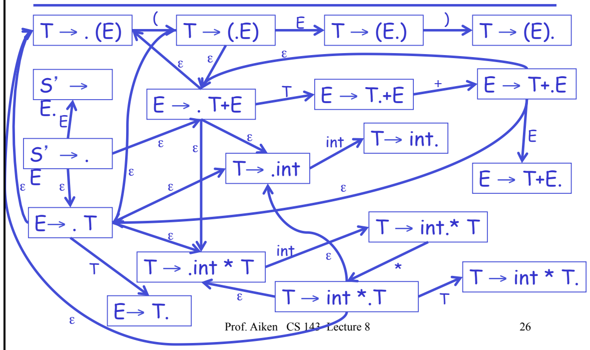
   $E \to \alpha.X\beta \ \to^\varepsilon\ X \to .\gamma$

An NFA Recognizing Viable Prefixes (Cont.)

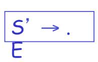
5. Every state is an accepting state

6. Start state is S' → .S

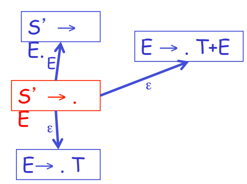
NFA for Viable Prefixes of the Example



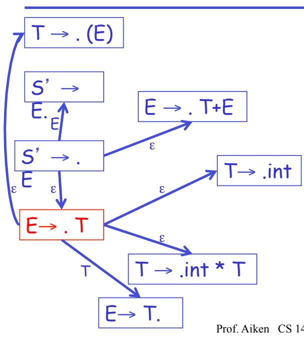
NFA for Viable Prefixes in Detail (1)



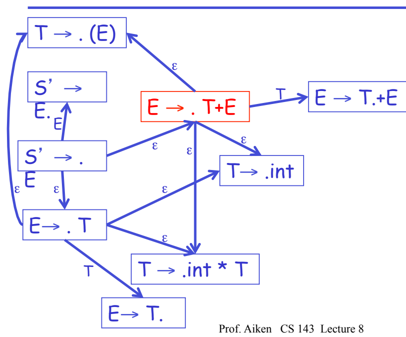
NFA for Viable Prefixes in Detail (2)



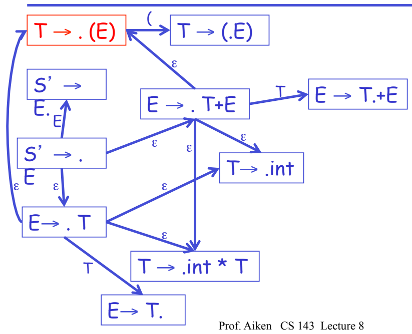
NFA for Viable Prefixes in Detail (3)



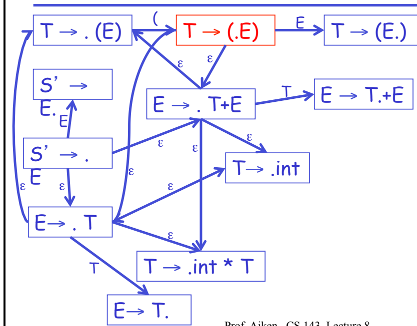
NFA for Viable Prefixes in Detail (4)



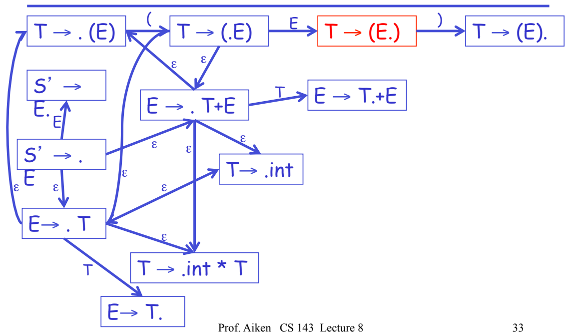
NFA for Viable Prefixes in Detail (5)



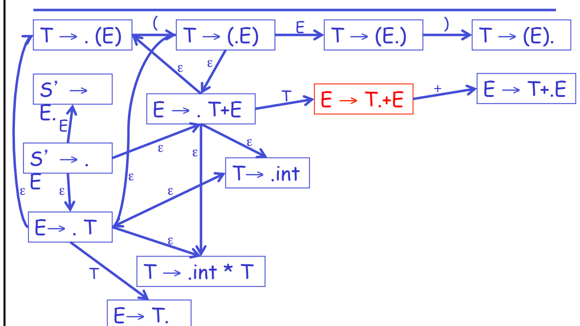
NFA for Viable Prefixes in Detail (6)



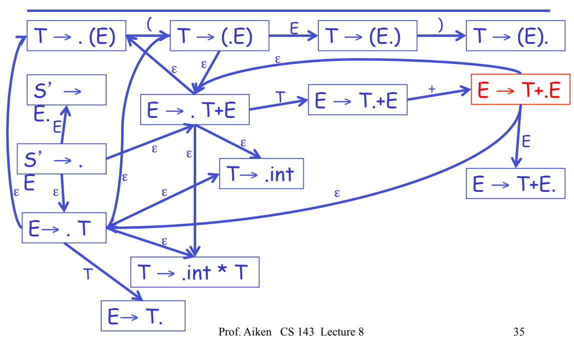
NFA for Viable Prefixes in Detail (7)



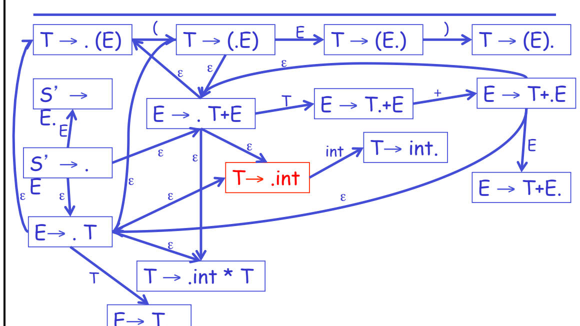
NFA for Viable Prefixes in Detail (8)



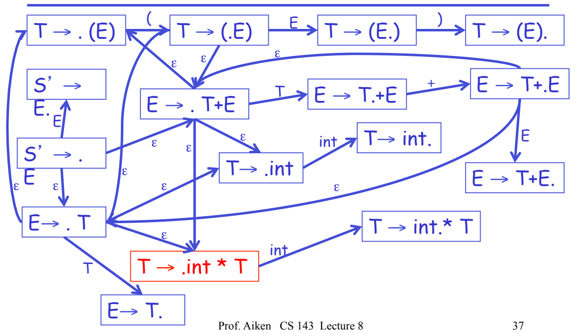
NFA for Viable Prefixes in Detail (9)



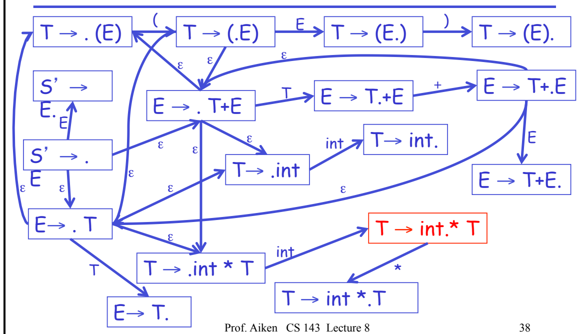
NFA for Viable Prefixes in Detail (10)



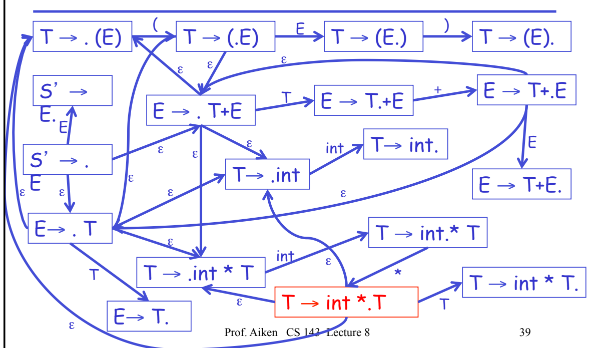
NFA for Viable Prefixes in Detail (11)



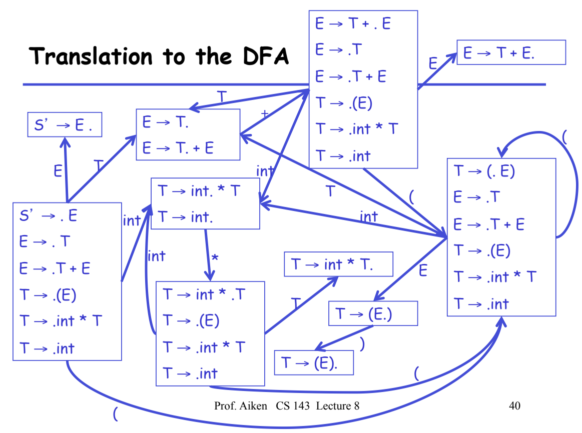
NFA for Viable Prefixes in Detail (12)



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

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

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

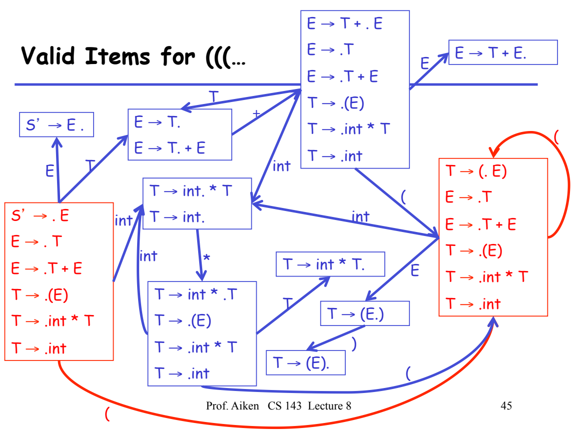
The items in s describe what the top of the item stack might be after reading input α

Valid Items Example

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

(
((
(((
((((
. . .

Valid Items for (((...



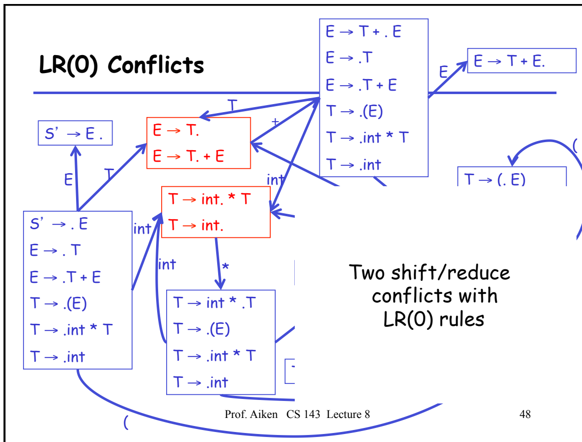
LR(0) Parsing

- Idea: Assume
  - stack contains α
  - next input is t
  - DFA on input α terminates in state s
- Reduce by $X \to \beta$ if
  - s contains item $X \to \beta.$
- Shift if
  - s contains item $X \to \beta.t\omega$
  - 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:
  - $X \to \beta.$ and $Y \to \omega.$
- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item:
  - $X \to \beta.$ and $Y \to \omega.t\delta$

LR(0) Conflicts



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

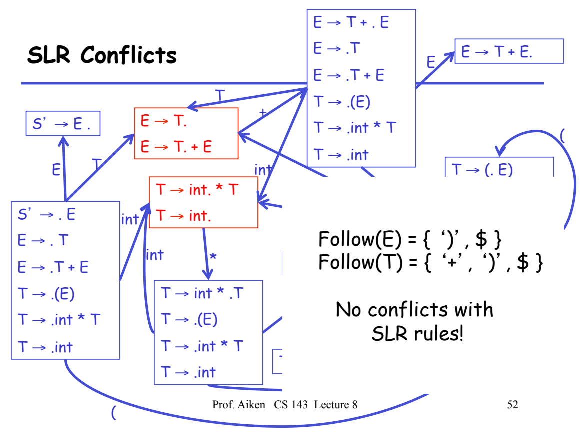
- Idea: Assume
  - stack contains α
  - next input is t
  - DFA on input α terminates in state s
- Reduce by $X \to \beta$ if
  - s contains item $X \to \beta.$
  - t ∈ Follow(X)
- Shift if
  - s contains item $X \to \beta.t\omega$

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



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!

Naïve 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 a be next input
     - Shift if $X \to \beta .\, a\, \gamma \in I$
     - Reduce if $X \to \beta . \in I$ and $a \in Follow(X)$
     - Report parsing error if neither applies

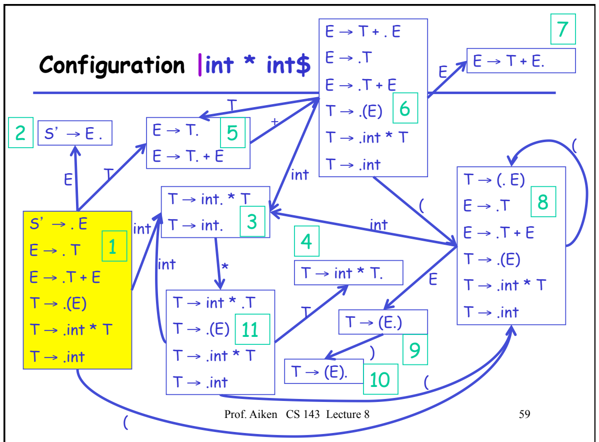
Notes

- 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

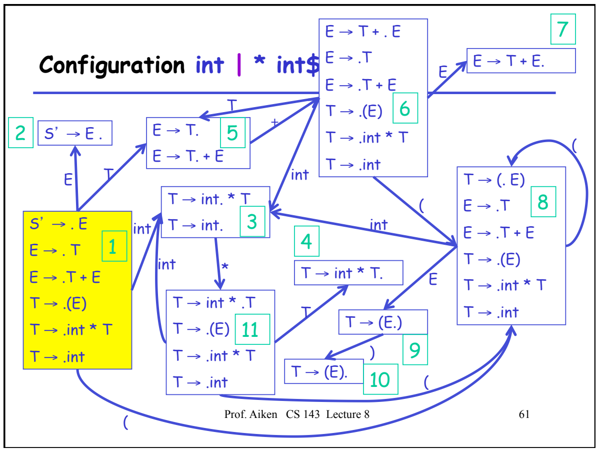
Configuration	*DFA Halt State*	*Action*
|int * int\$	1	shift

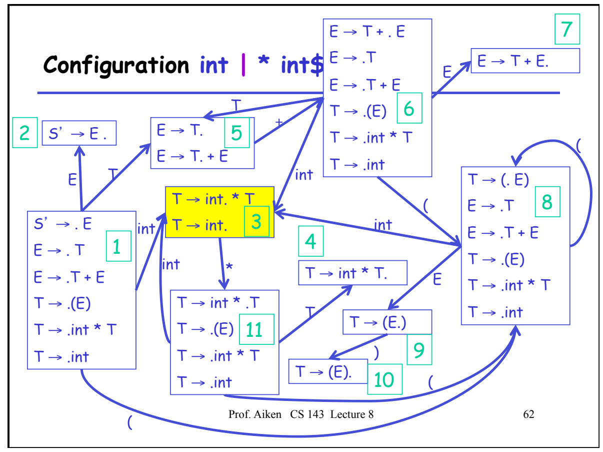
Configuration |int * int\$



SLR Example

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

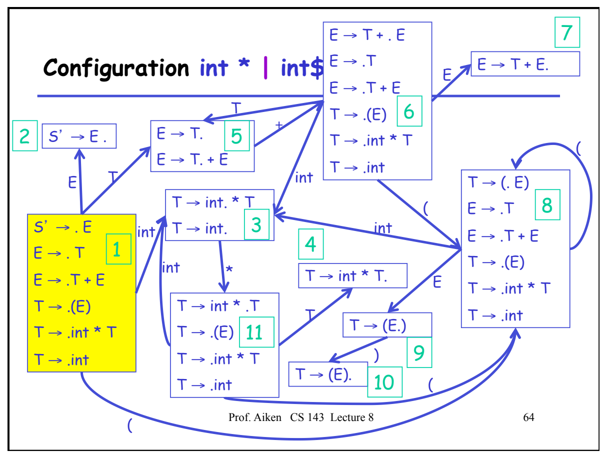


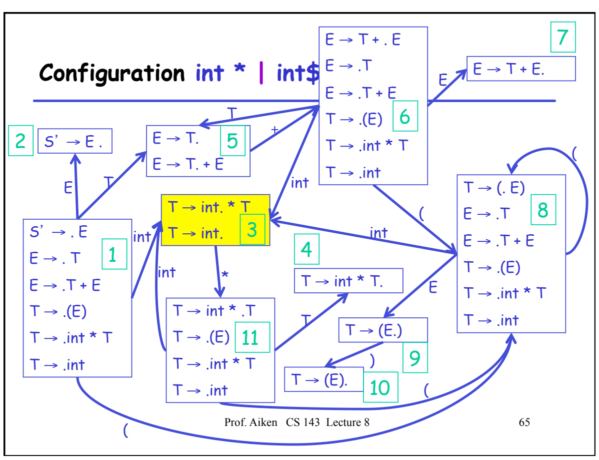


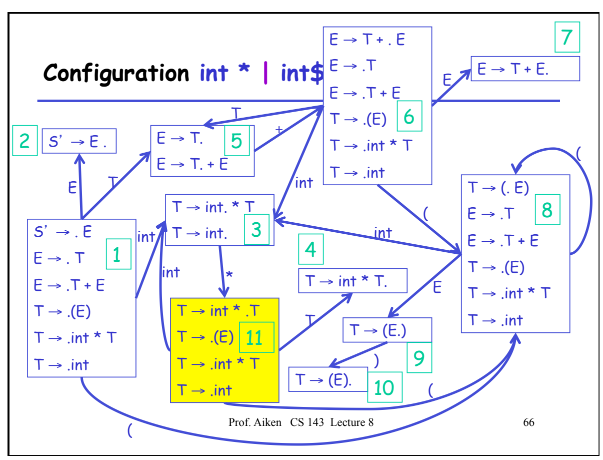
SLR Example

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

Prof. Aiken CS 143 Lecture 8 63



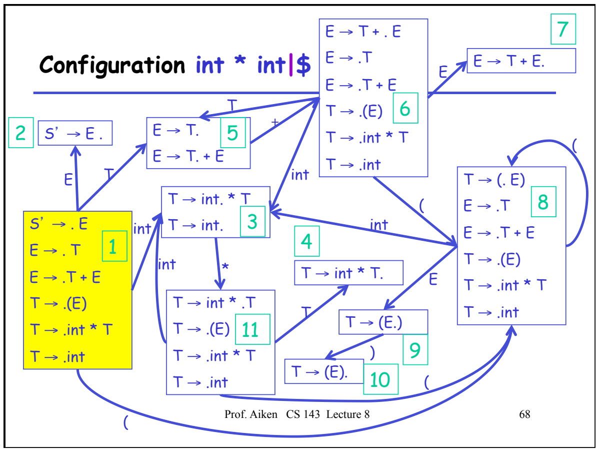




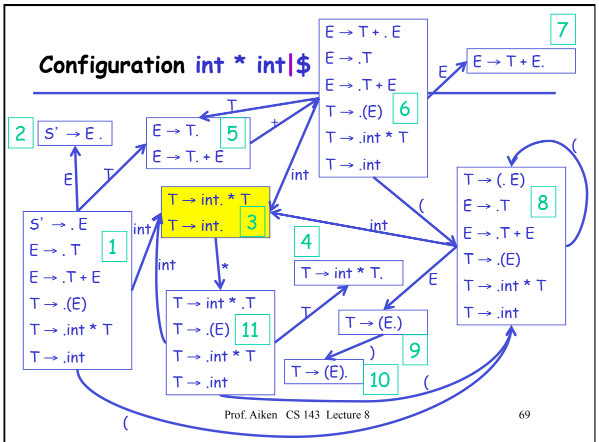
SLR Example

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

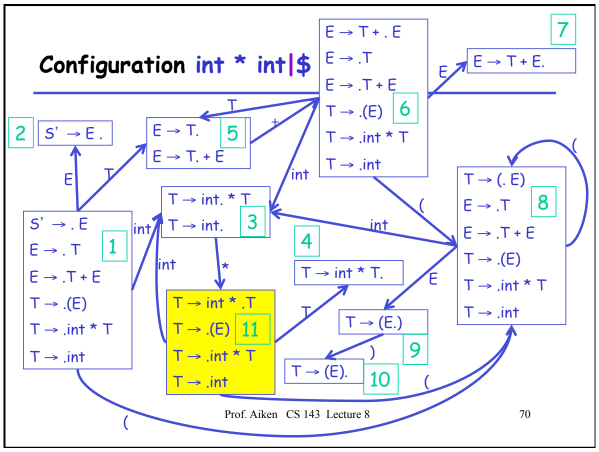
Configuration int * int|$



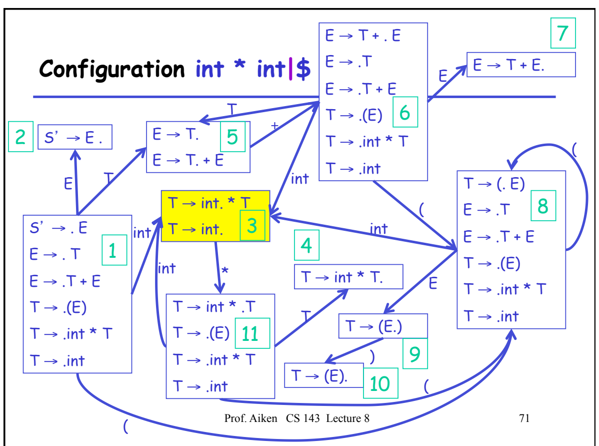
Configuration int * int|$



Configuration int * int|$



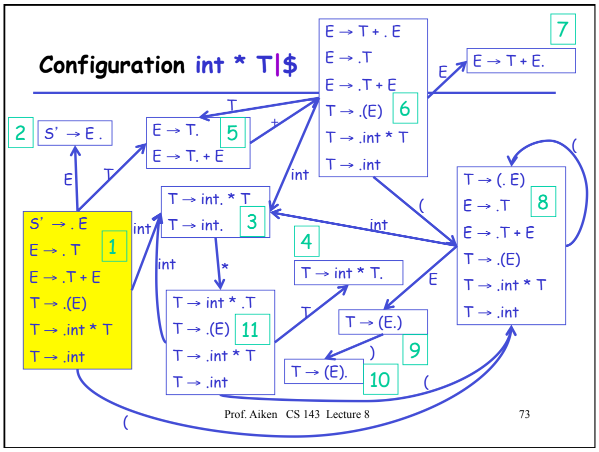
Configuration int * int|$



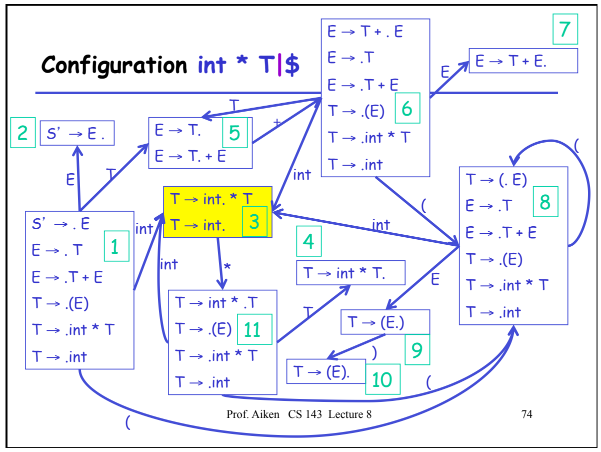
SLR Example

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 |$	4	$ ∈ Follow(T)	red. T→int*T

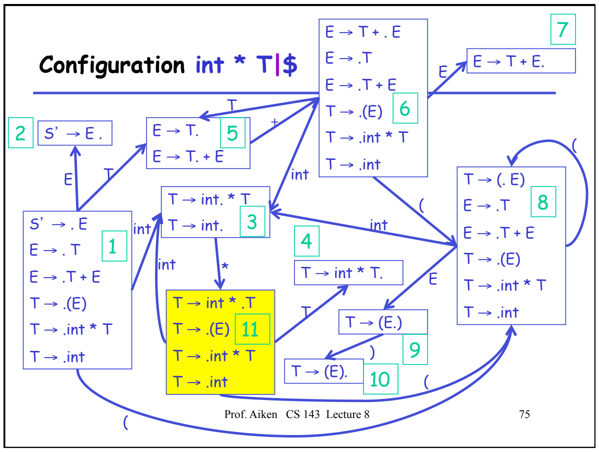
Configuration int * T|$



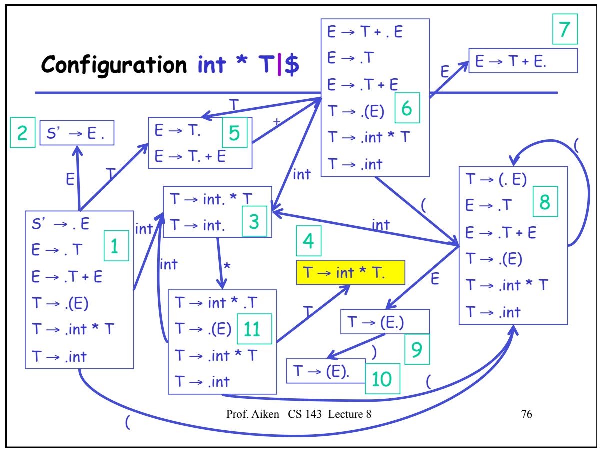
Configuration int * T|$



Configuration int * T|$



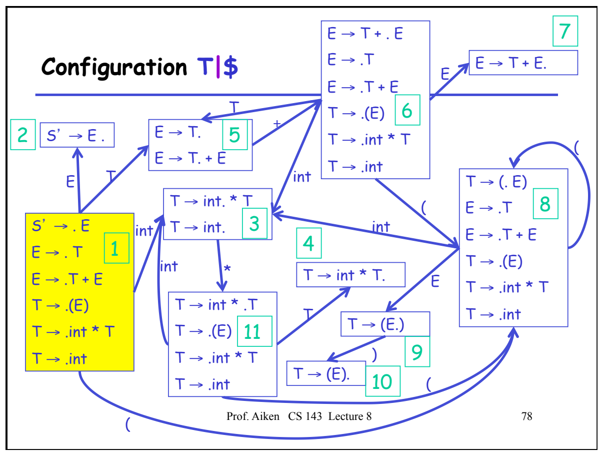
Configuration int * T|$



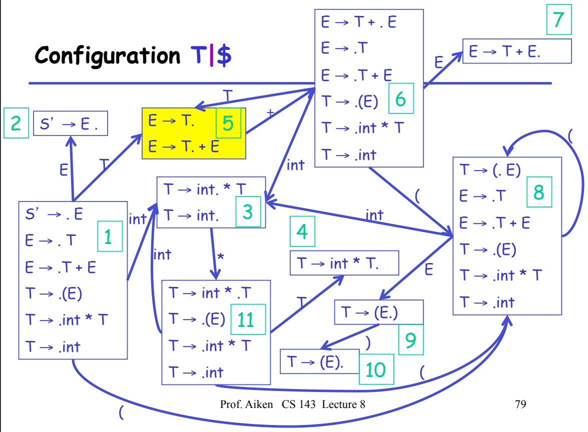
SLR Example

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 |$	4	$ ∈ Follow(T)	red. T→int*T
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 |$	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

  ⟨ 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 symbol
  - 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 $X \to \alpha$
  - As before
- Accept
- Error

Action Table

For each state s_i and terminal a

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

SLR Parsing Algorithm

```
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,
                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
  - Take a look at the LR(1) automaton for your parser!