Lecture #12

Syntax Analysis - VI

Bottom-Up Parsing

- Given a grammar *G*, a parse tree for a given *string* is constructed by starting at the *leaves* (*terminals* of the string) and working to the *root* (the start symbol *S*).
- They are able to accept a more general class of grammars compared to top-down predictive parsers.
- It builds on the concepts developed in top-down parsing.
- Preferred method for most of the parser generators including *bison*
- They don't need left-factored grammars
  - So, its valid to use the following grammar

$$E \rightarrow T + E \mid T$$

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

Bottom-Up Parsing

- A parse for a string generates a sequence of derivations of the form:

$$S \Rightarrow \delta_0 \Rightarrow \delta_1 \Rightarrow \delta_2 \Rightarrow \ldots \Rightarrow \delta_{n-1} \Rightarrow sentence$$

- Bottom-up parsing *reduces* a string to the start symbol by inverting productions
- Let $A \rightarrow b$ be a production and δ_{i-1} and δ_i be two consecutive derivations with sentential forms: $\alpha A\beta$ and $\alpha b\beta$
  - δ_{i-1} is derived from δ_i by matching the *RHS b* in δ_i, and then replacing *b* with its corresponding *LHS*, *A*. This is called a ***reduction***
- P**arse tree** is the result of the **tokens** and the **reductions**.

Bottom-Up Parsing

- Consider the parse for the input string: *int * int + int*

  $E \rightarrow T + E \mid T$

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

Sentential Form	Production
*int * <u>int</u> + int*	
<u>int * T</u> + int	$T \rightarrow$ int
T + <u>int</u>	$T \rightarrow$ int * T
T + <u>T</u>	$T \rightarrow$ int
<u>T + E</u>	$E \rightarrow T$
E	$E \rightarrow T + E$

- When we reduce, we only have terminals to the right.
  - If $\alpha b \beta$ to $\alpha A \beta$ is a step of a bottom-up parse
  - And the reduction is by $A \rightarrow b$
  - Then β is a string of terminals

- In other words, a *bottom-up parser traces a rightmost derivation in reverse*

Shift-Reduce Parsing

- Idea: Split string being parsed into two parts
  - Two parts are separated by a special character ‘|’
  - Left part is a string of terminals and non terminals
  - Right part is a string of terminals
    - Still to be examined

- Bottom up parsing has two actions
  - **Shift:** Move terminal symbol from right string to left string
    - **ABC | xyz ⇒ ABCx | yz**

- **Reduce:** Apply an inverse production at the right end of the left string
  - If $A \to xy$ is a production, then
    - **Cbxy | ijk ⇒ CbA | ijk**

Shift-Reduce Example

Sentential Form	Action
|int * int + int	Shift
int | * int + int	Shift
int * | int + int	Shift
int * int | + int	Reduce T $\rightarrow$ int
int * T | + int	Reduce T $\rightarrow$ int * T
T | + int	Shift
T + | int	Shift
T + int |	Reduce T $\rightarrow$ int
T + T |	Reduce E $\rightarrow$ T
T + E |	reduce E $\rightarrow$ T + E
E |	Accept

To Shift or Reduce?

- Symbols on the left of "|" are kept on a ***stack***
  - Top of the stack is at "|"
  - Shift pushes a terminal on the stack
  - Reduce pops symbols (RHS of production) and pushes a non terminal (LHS of production) onto the stack

- The most important issues are:
  - When to shift and when to reduce!
  - Which production to use for reduction?
  - Sometimes parser can reduce but it should not!
  - $X \rightarrow \epsilon$ can always be reduced!
  - Sometimes parser can reduce in different ways!

To Shift or Reduce? - 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:
    - ***Shift-reduce conflict***
  - If it is legal to reduce by two different productions:
    - ***Reduce-reduce conflict***

- *Reduce action should be taken only if the result can be reduced to the start symbol*

Shift-Reduce Parsing - Handles

- Handles
  - A substring that matches the right-side of a production that occurs as one step in the rightmost derivation. This substring is called a *handle*.
  - Because δ is a right-sentential form, the substring to the right of a handle contains only terminal symbols. Therefore, the parser doesn't need to scan past the handle.
  - If a grammar is unambiguous, then every right-sentential form has a unique handle in the reversed rightmost derivation
  - If we can find those handles, we can build a derivation

Recognizing Handles

- Given the grammar: $E \rightarrow T + E \mid T$
  $T \rightarrow int * T \mid int \mid (E)$
- Consider step int | * int + int
- We could reduce by $T \rightarrow$ int giving T | * int + int
  - But this is incorrect because:
    - No way to reduce to the start symbol E

- So, a handle is a reduction that also allows further reductions back to the start symbol
- *In shift-reduce parsing, handles appear only at the top of the stack, never inside*

Recognizing Handles

- **Handles always appear only at stack top:**
  - 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

- 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 a β such that α|β is a state of a shift-reduce parser

Viable Prefixes

- Therefore:
  - 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

- *For any grammar, the set of viable prefixes is a regular language*
- *So, we can generate an automata to recognize viable prefixes!*

Viable Prefixes

- α is a viable prefix of a given grammar if:
  - There is a ω such that $\alpha\omega$ is a right sentential form
- $\alpha \mid \omega$ is a state of the shift-reduce parser
- As long as the parser has viable prefixes on the stack no parser error has been seen
- The set of viable prefixes is a regular language (not obvious)
- Construct an automaton that accepts viable prefixes

LR(0) Items

- An LR(0) item of a grammar G is a production of G with a special symbol "." at some position of the right side
- Thus production A→XYZ gives four LR(0) items

  A → .XYZ
  A → X.YZ
  A → XY.Z
  A → XYZ.
- An item indicates how much of a production has been seen at a point in the process of parsing
  - Symbols on the left of "." are already on the stack **s**
  - Symbols on the right of "." are expected in the input
- The only item for X → ε is X → .

Viable Prefixes and LR(0) Items

- Consider the input: **(int)**

  E → T + E | T

  T → int * T | int | (E)

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

  - **(E** is a prefix of the rhs of **T → (E)**
    - Will be reduced after the next shift

  - Item **T → (E.)** says that so far we have seen **(E** of this production and hope to see **)**

Viable Prefixes and LR(0) Items

- The stack may have many prefixes of rhs's
  - **$\mathbf{Prefix_1}$ $\mathbf{Prefix_2}$ . . . $\mathbf{Prefix_{n-1}}$$\mathbf{Prefix_n}$**
- Let $\mathbf{Prefix_i}$ be a prefix of the rhs of $\mathbf{X_i} \rightarrow \boldsymbol{\alpha_i}$
  - $\mathbf{Prefix_i}$ will eventually reduce to $\mathbf{X_i}$
  - The missing part of $\boldsymbol{\alpha_{i-1}}$ starts with $\mathbf{X_i}$
  - i.e. there is a $\mathbf{X_{i-1}} \rightarrow \mathbf{Prefix_{i-1}}\ \mathbf{X_i}\ \boldsymbol{\beta}$ for some $\boldsymbol{\beta}$
- Recursively, $\mathbf{Prefix_{k+1}}$…$\mathbf{Prefix_n}$ eventually reduces to the missing part of $\boldsymbol{\alpha_k}$

Viable Prefixes and LR(0) Items

- Consider the string (int * int):
  - (int *|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 $E \rightarrow T$
    - "int *" is a prefix of the rhs of $T \rightarrow \text{int} * T$

- The "stack of items"
  - $T \rightarrow (.E)$ says, we have seen "(" of $T \rightarrow (E)$
  - $E \rightarrow .T$ says, we have seen ε of $E \rightarrow T$
  - $T \rightarrow \text{int} * .T$ says, we have seen int * of $T \rightarrow \text{int} * T$

Recognizing Viable Prefixes

- Therefore, we have to build the finite automata that recognizes this sequence of partial rhs's of productions
- Algorithm:

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
   - $E \rightarrow \alpha.X\beta \rightarrow^{X} E \rightarrow \alpha X.\beta$
4. For item $E \rightarrow \alpha.X\beta$ and production $X \rightarrow \gamma$ add
   - $E \rightarrow \alpha.X\beta \rightarrow^{\varepsilon} X \rightarrow .\gamma$
5. Every state is an accepting state
6. Start state is $S' \rightarrow .S$

Recognizing Viable Prefixes

- Therefore, we have to build the finite automata that recognizes this sequence of partial rhs's of productions
- Algorithm:

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
   - $E \rightarrow \alpha.X\beta \rightarrow^{X} E \rightarrow \alpha X.\beta$
4. For item $E \rightarrow \alpha.X\beta$ and production $X \rightarrow \gamma$ add
   - $E \rightarrow \alpha.X\beta \rightarrow^{\varepsilon} X \rightarrow .\gamma$
5. Every state is an accepting state
6. Start state is $S' \rightarrow .S$

Recognizing Viable Prefixes

- Given the grammar:

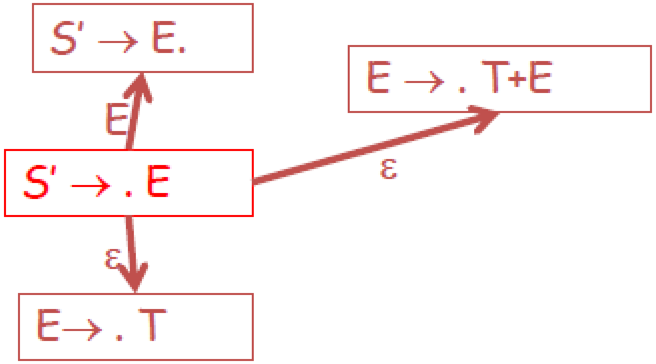
$$S' \rightarrow E$$
$$E \rightarrow T + E \mid T$$
$$T \rightarrow int * T \mid int \mid (E)$$

Create an NFA to recognize viable prefixes

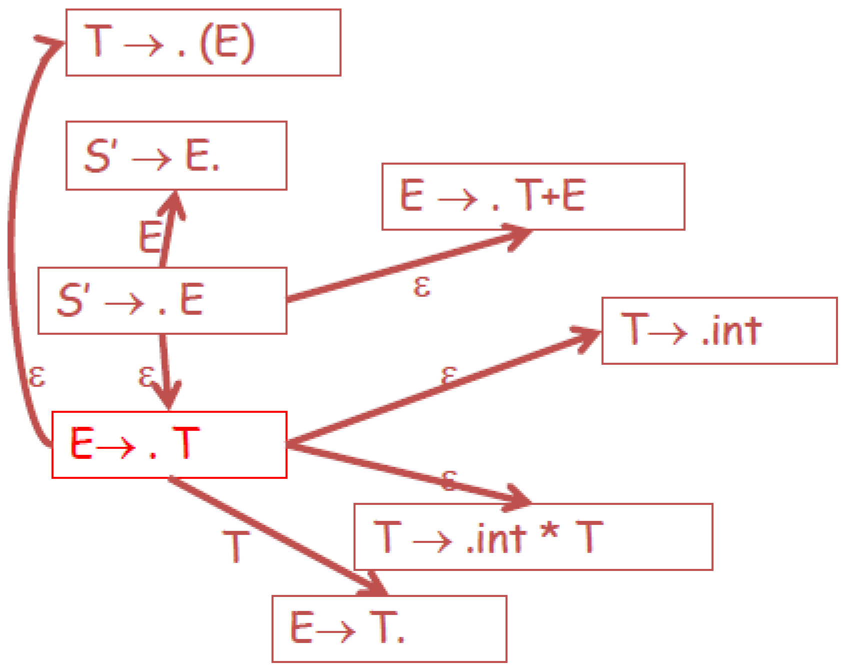
Recognizing Viable Prefixes

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

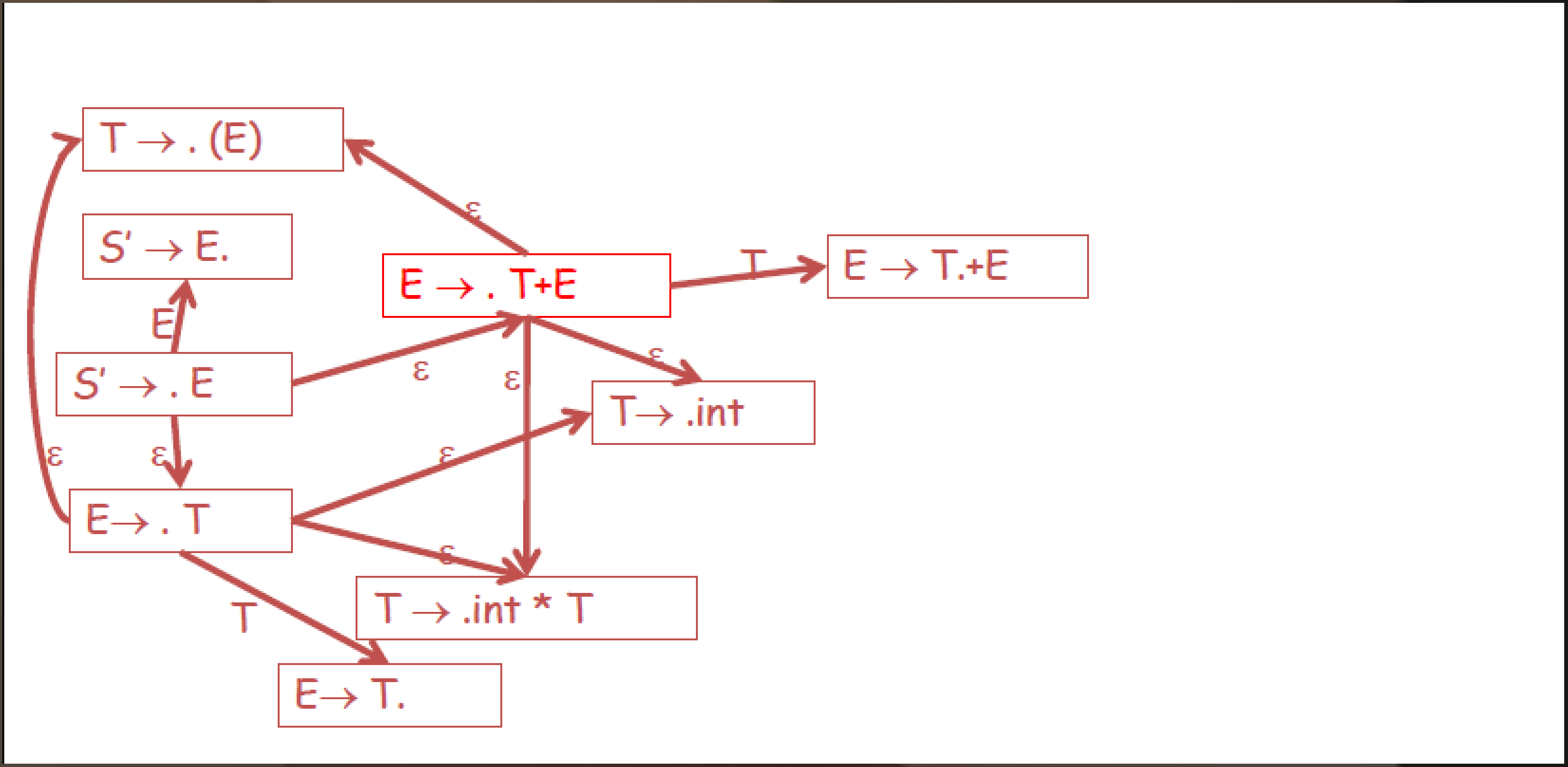
Recognizing Viable Prefixes



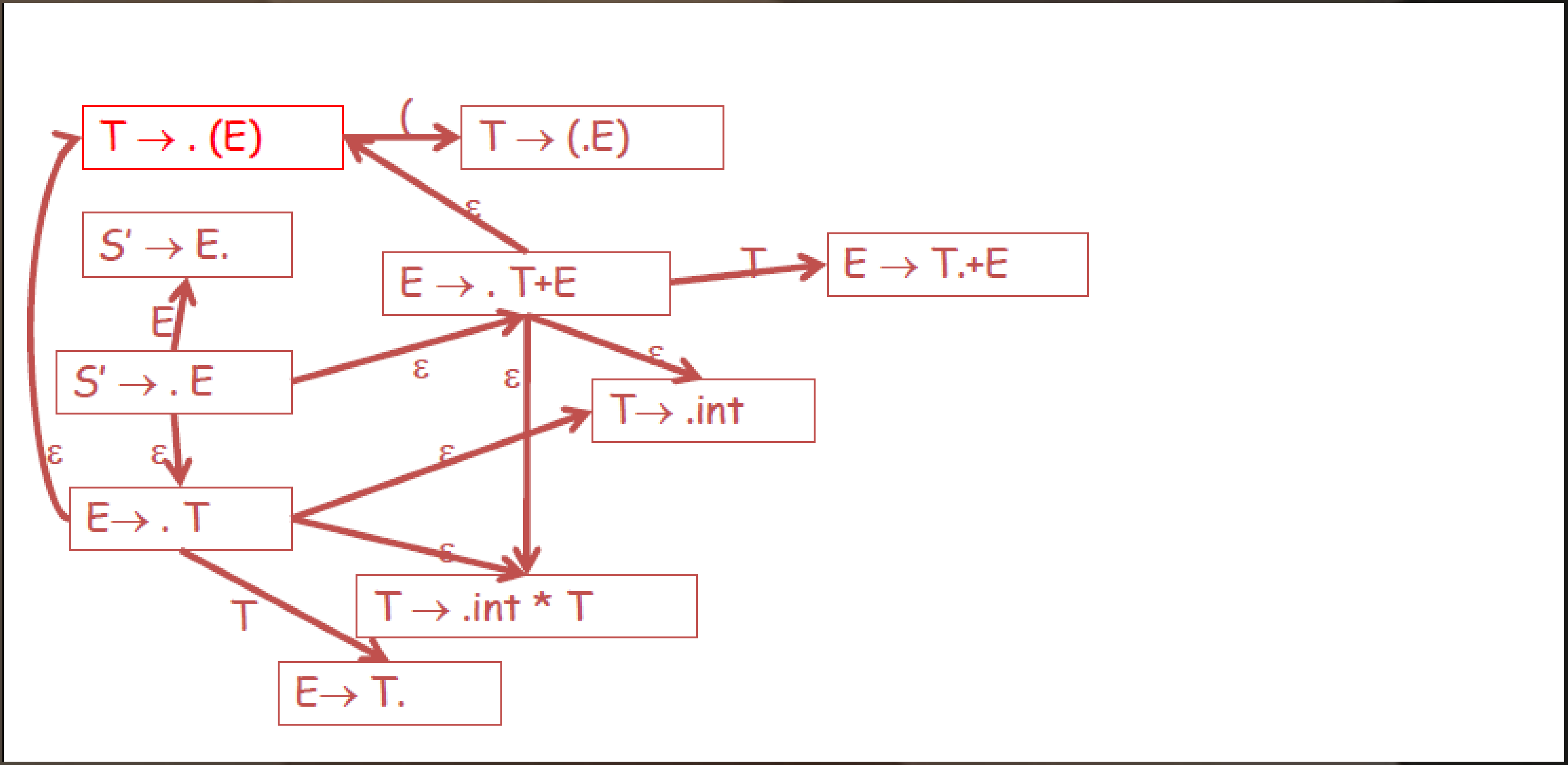
Recognizing Viable Prefixes



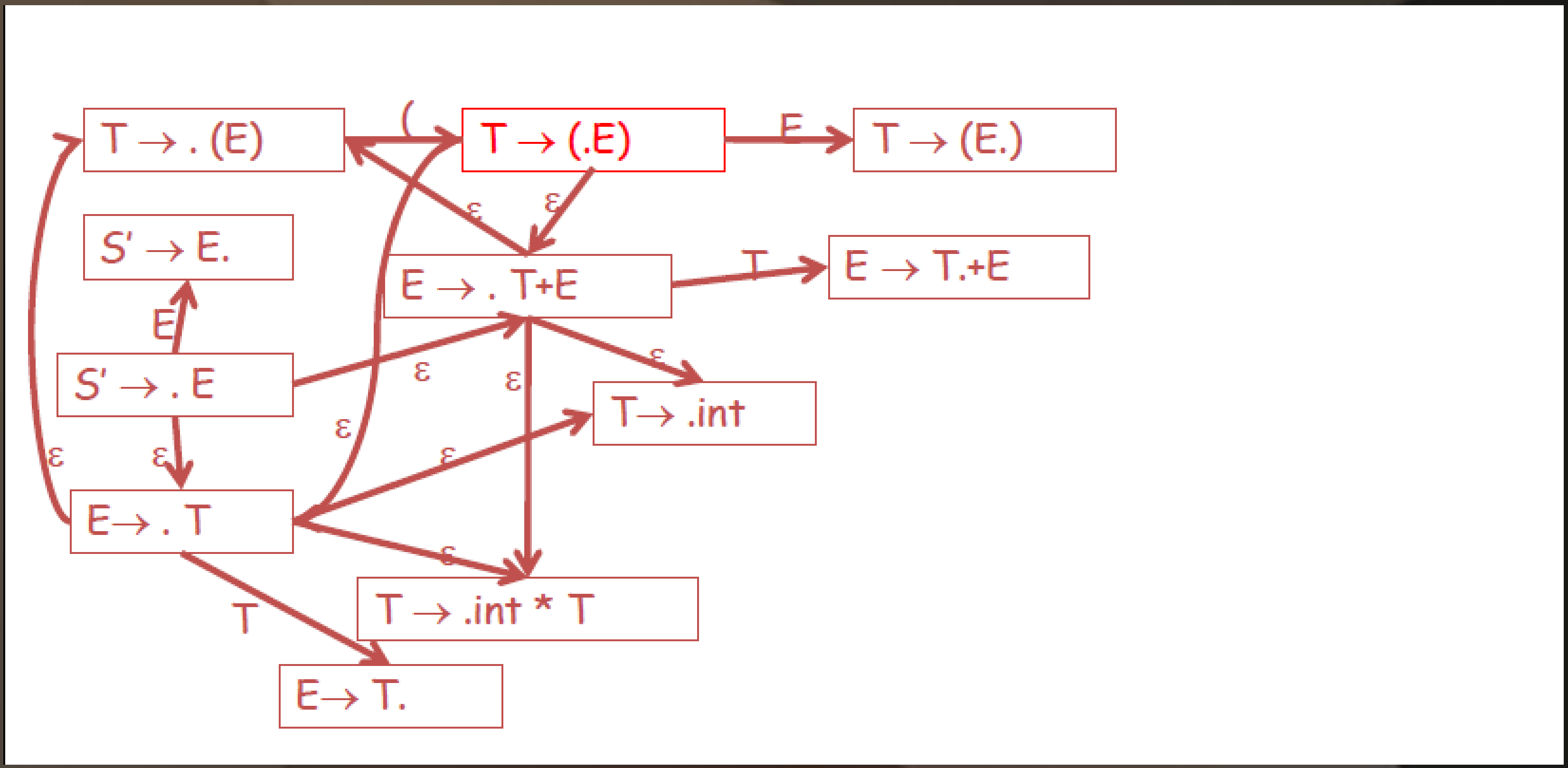
Recognizing Viable Prefixes



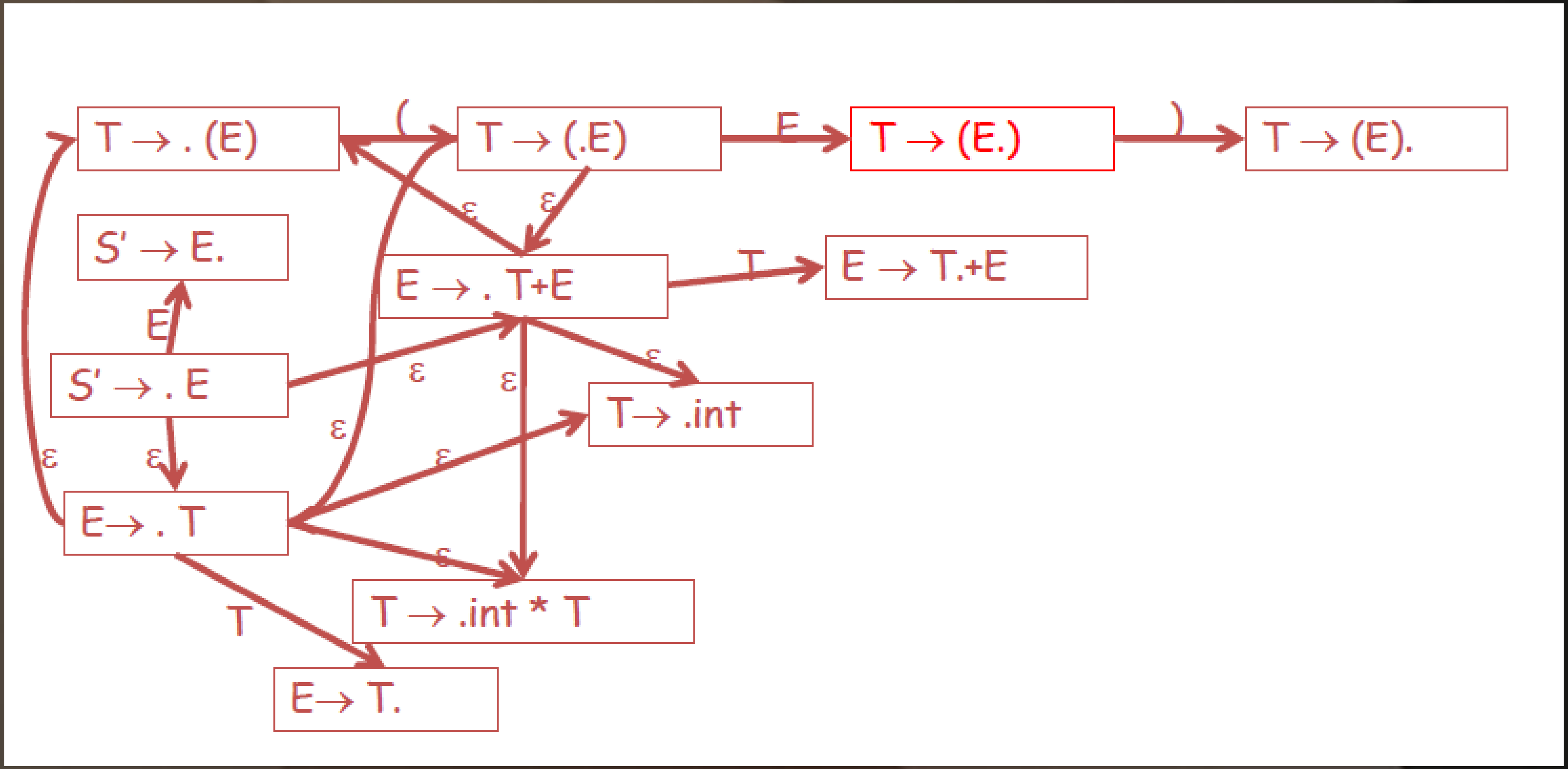
Recognizing Viable Prefixes



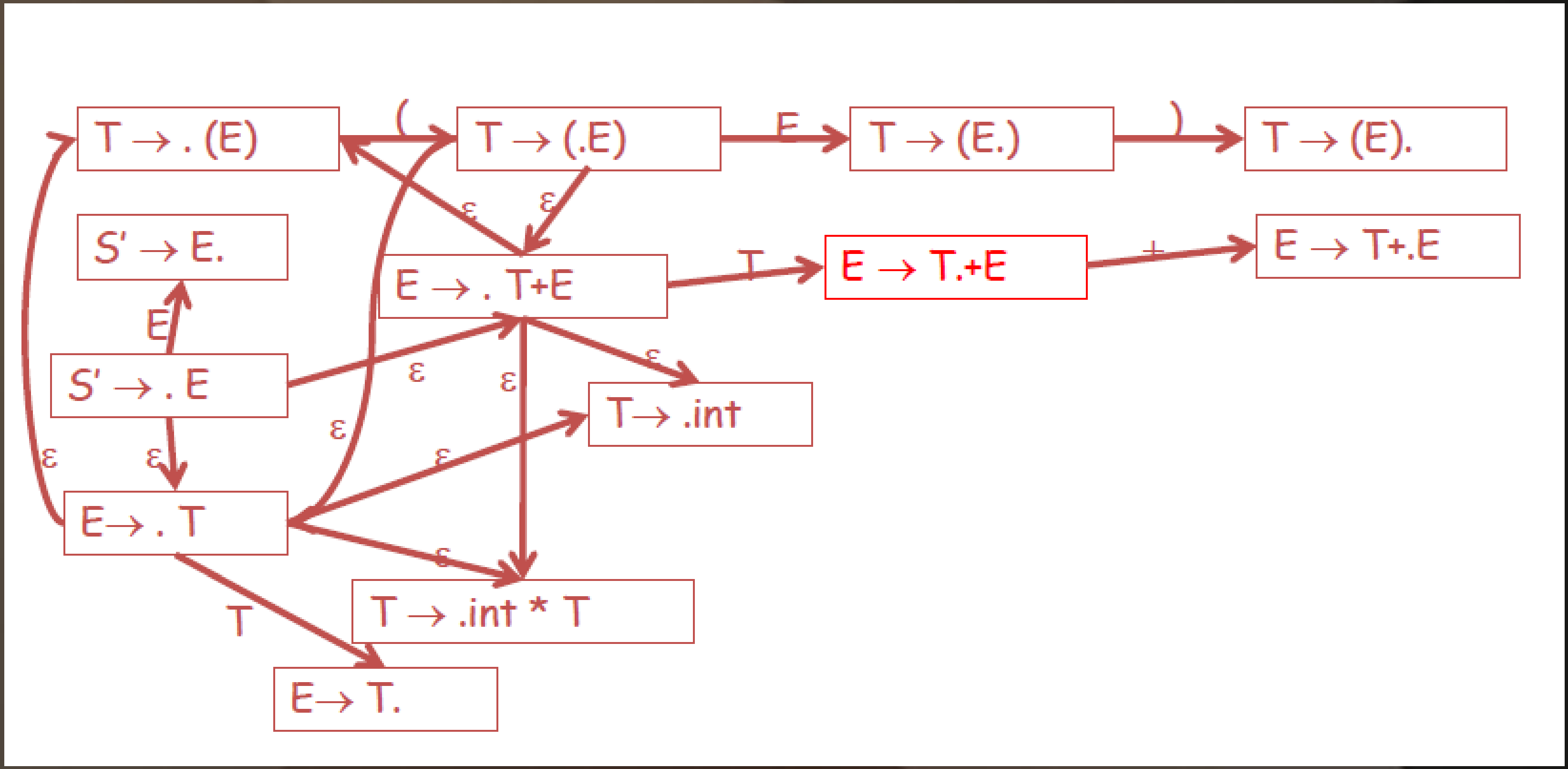
Recognizing Viable Prefixes



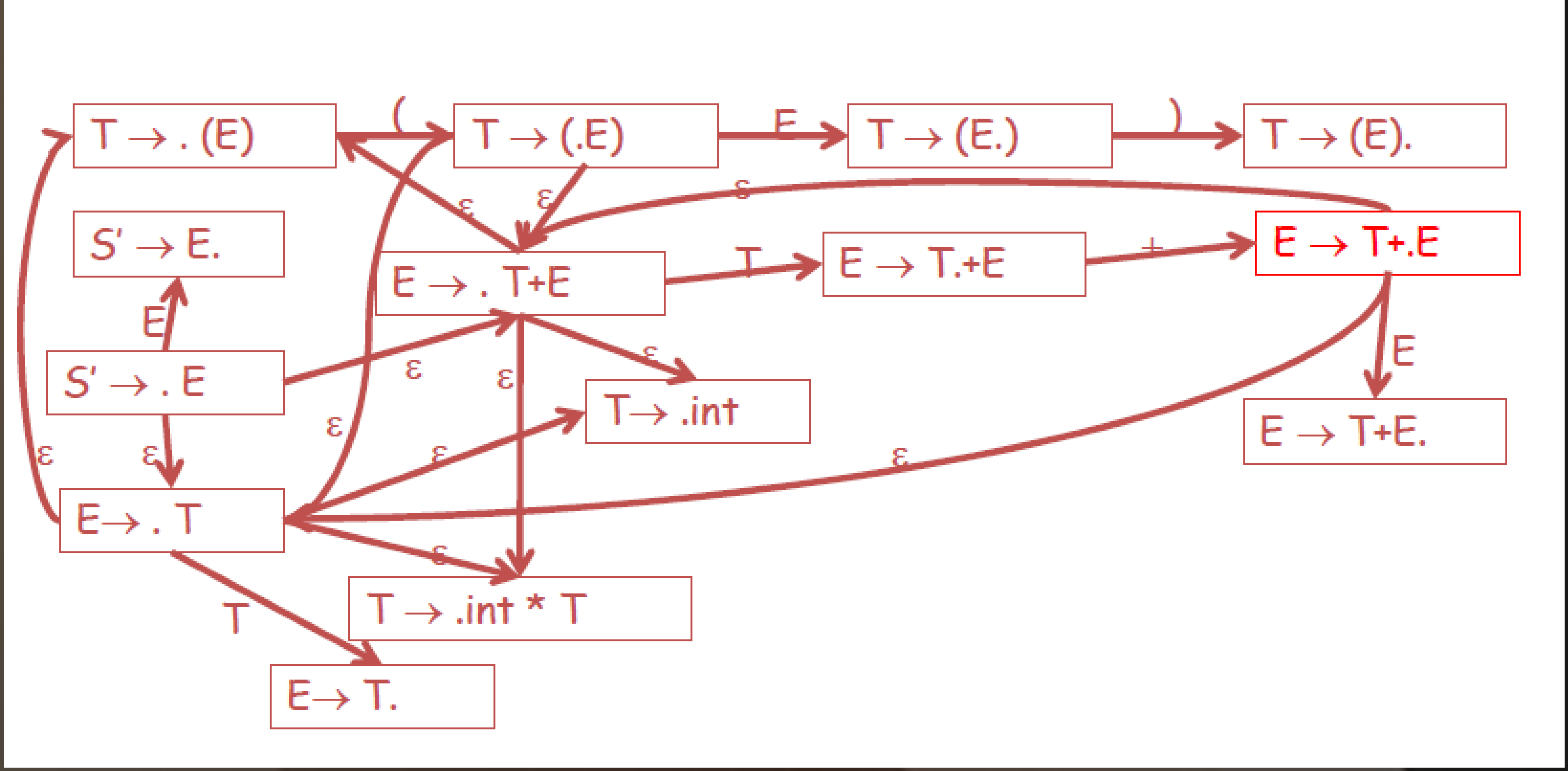
Recognizing Viable Prefixes



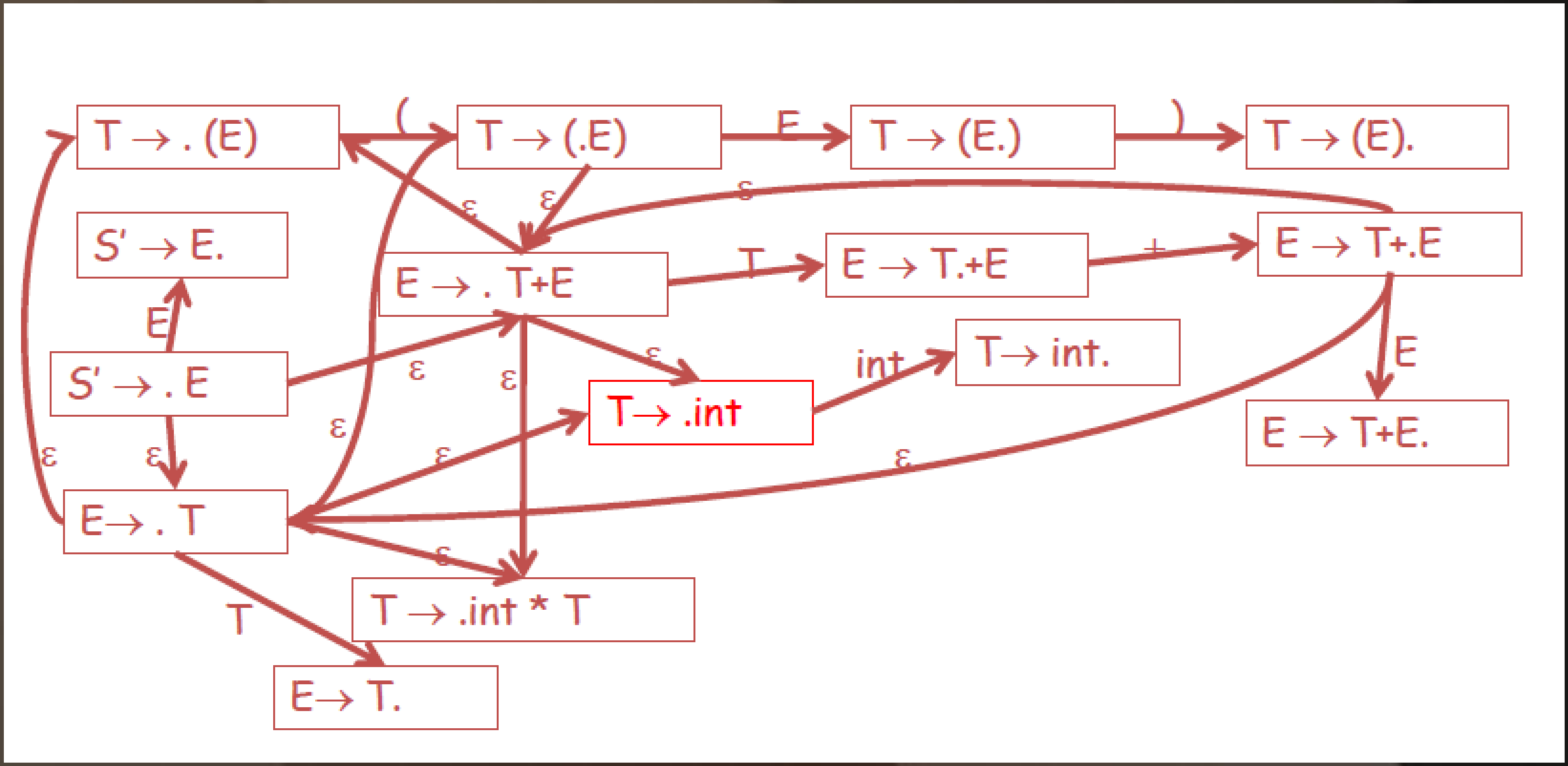
Recognizing Viable Prefixes



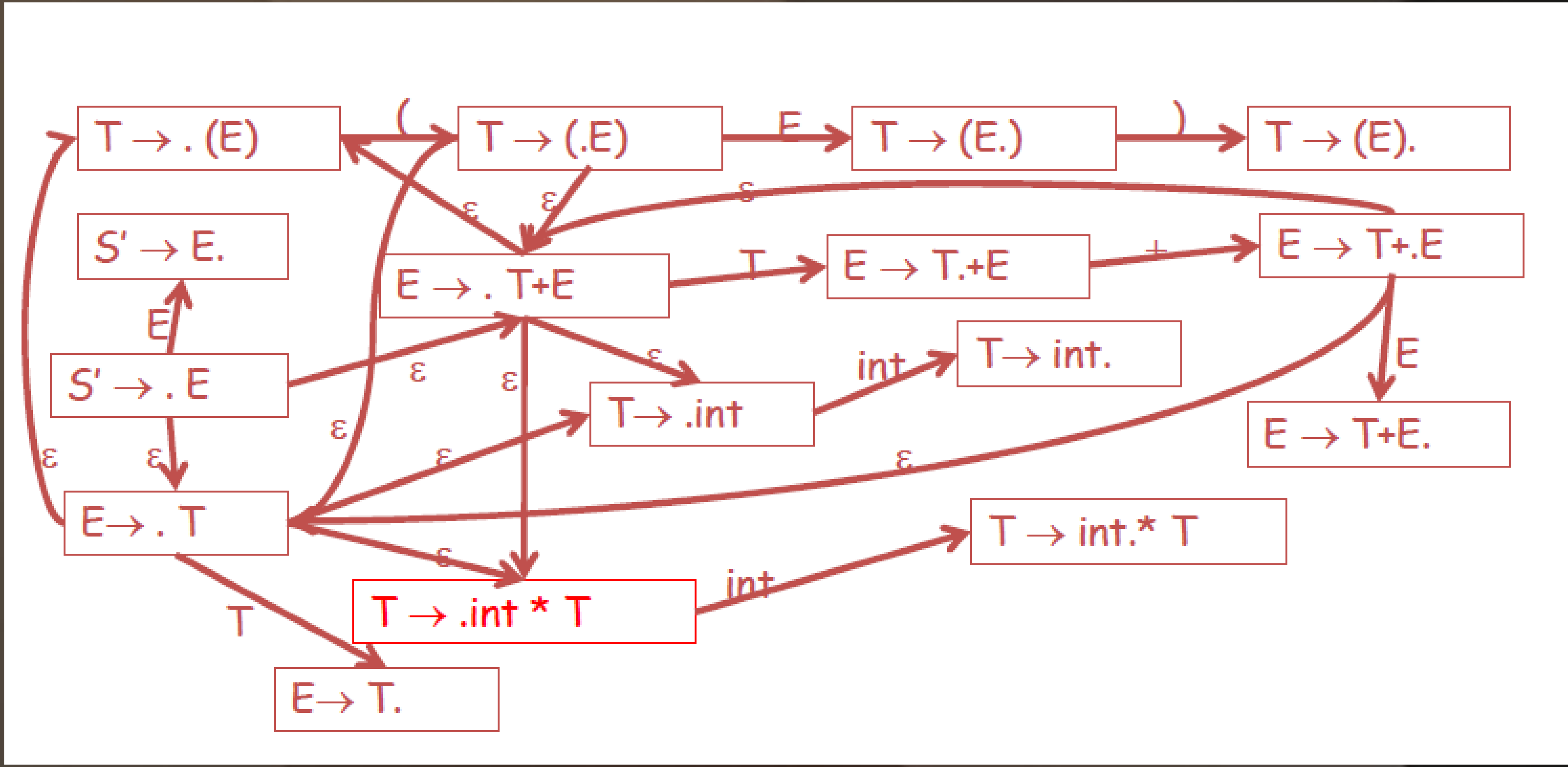
Recognizing Viable Prefixes



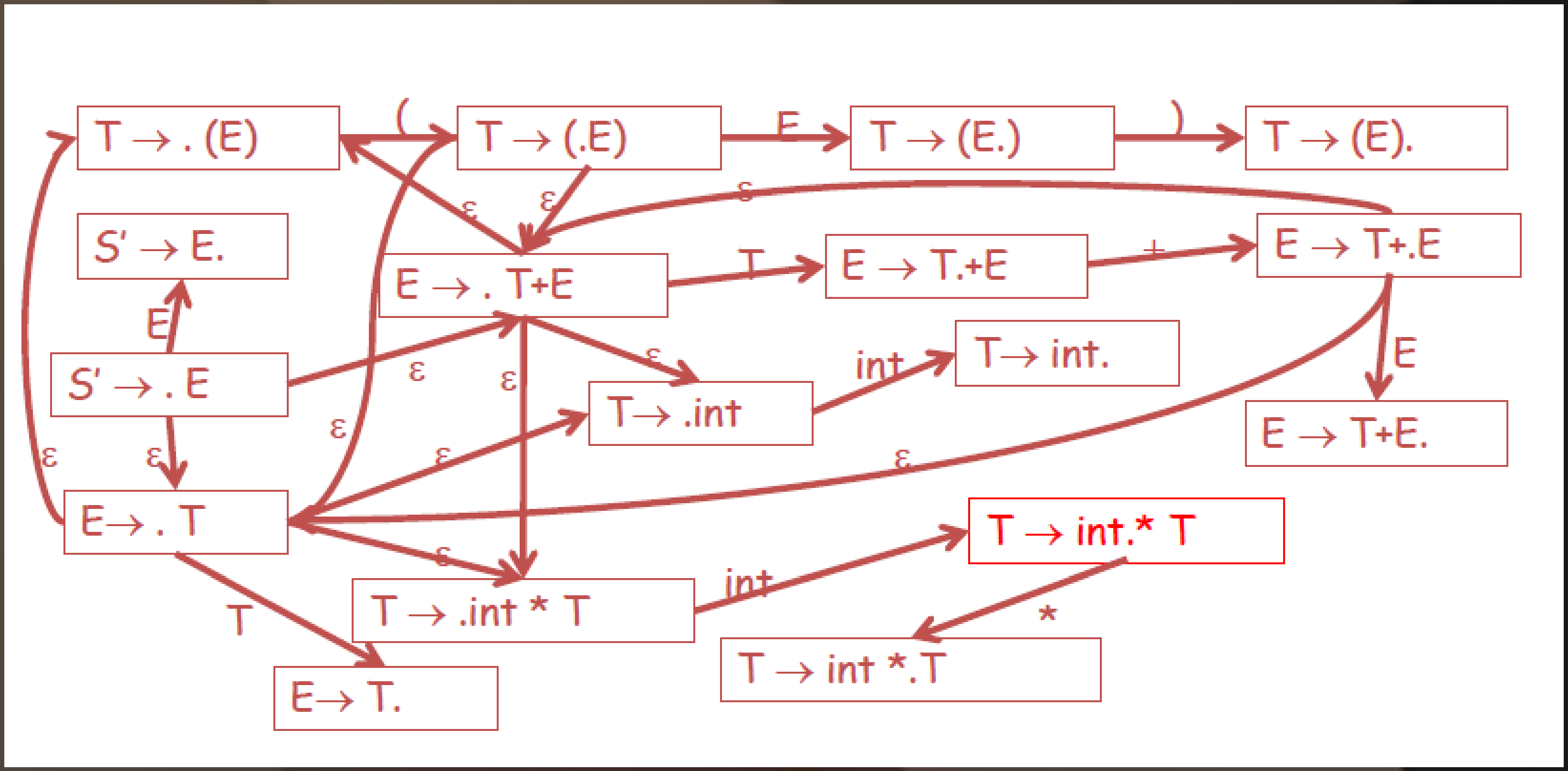
Recognizing Viable Prefixes



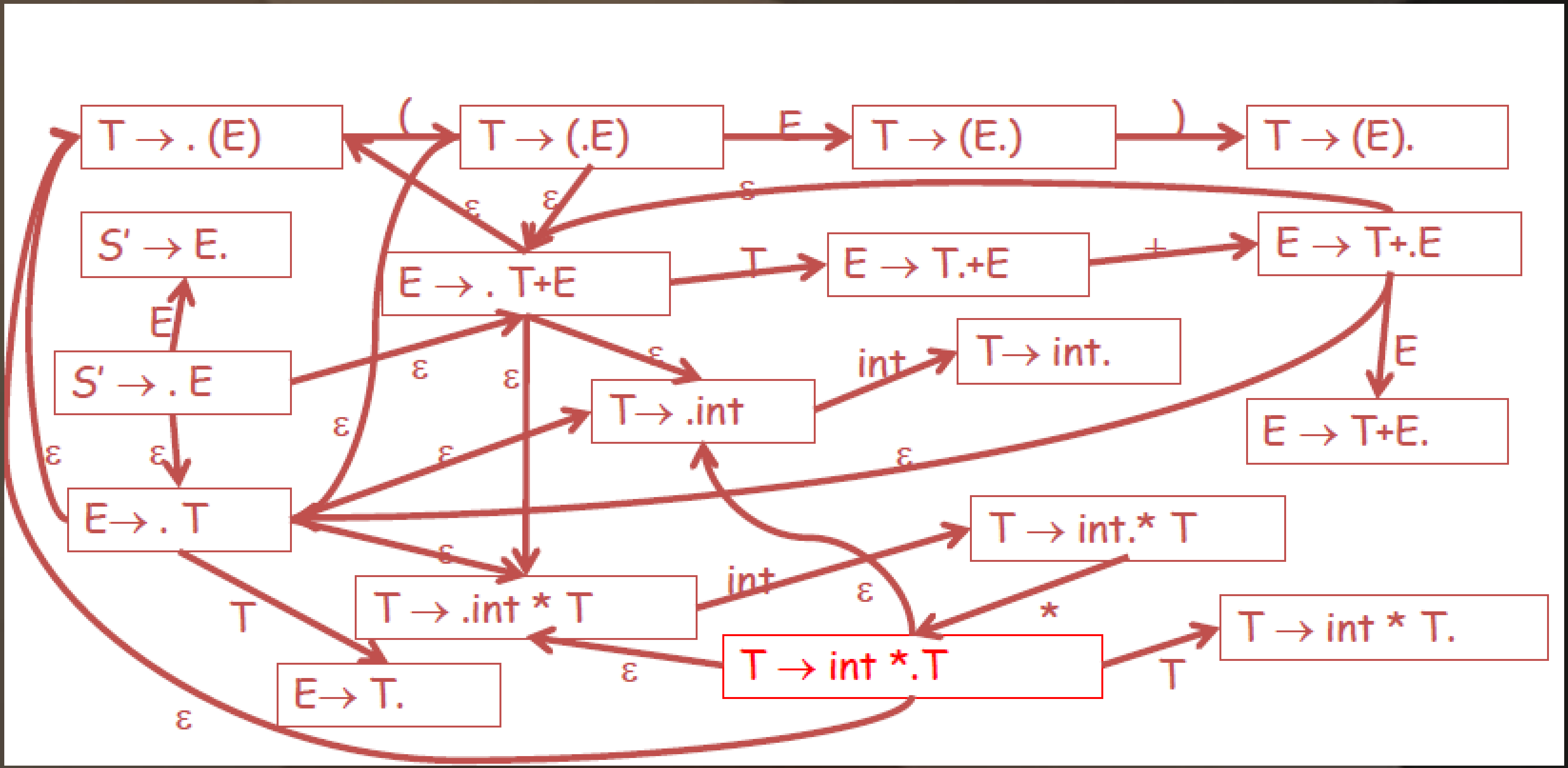
Recognizing Viable Prefixes



Recognizing Viable Prefixes



Recognizing Viable Prefixes



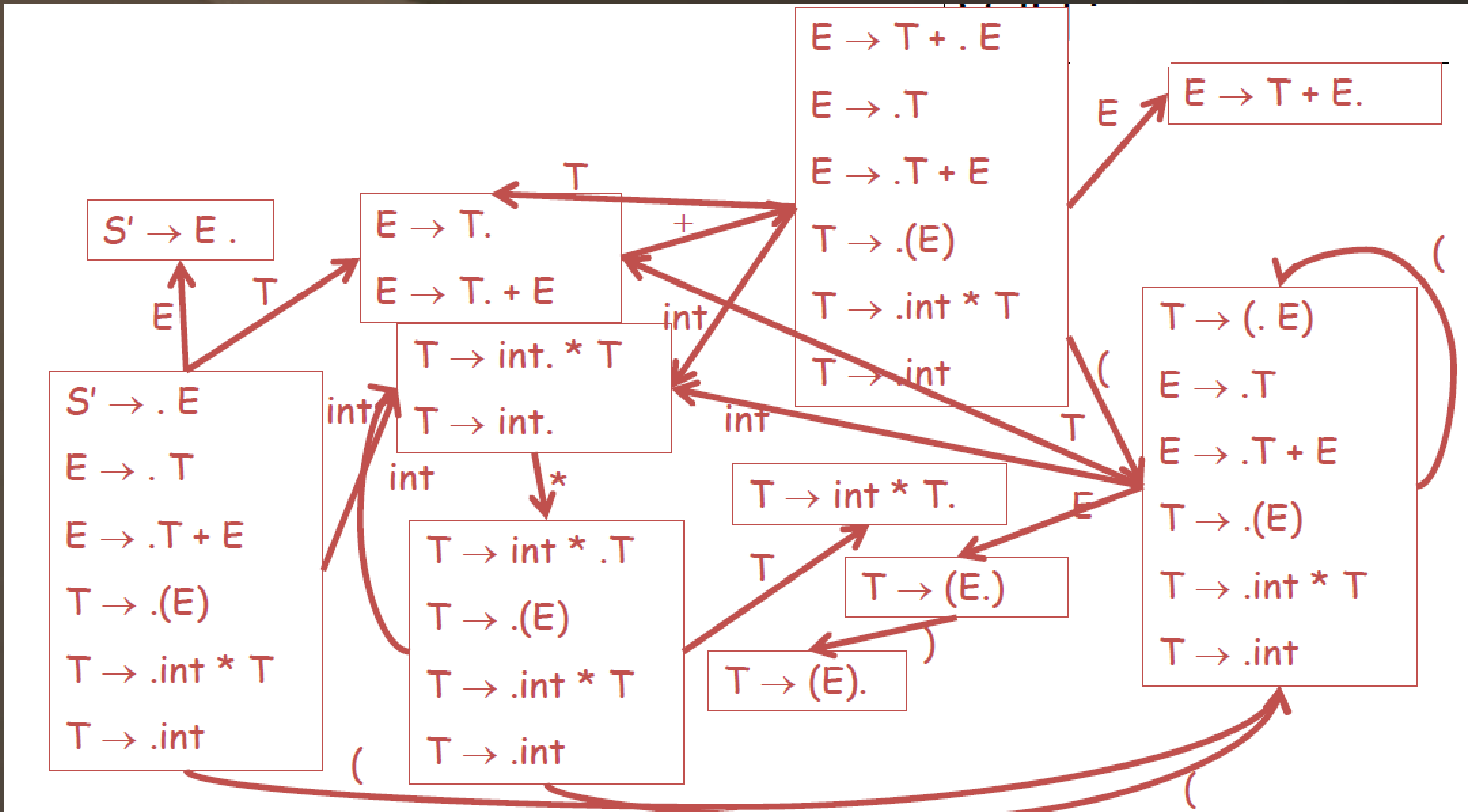
Recognizing Viable Prefixes

- Given the grammar:

$$S' \rightarrow E$$
$$E \rightarrow - E \mid id$$

Create an NFA to recognize viable prefixes

NFA to DFA – Subset Construction



- *Convert the NFA obtained in the previous slide to a DFA*

Valid Items

- The states of the DFA are:
  - canonical collections of LR(0) items

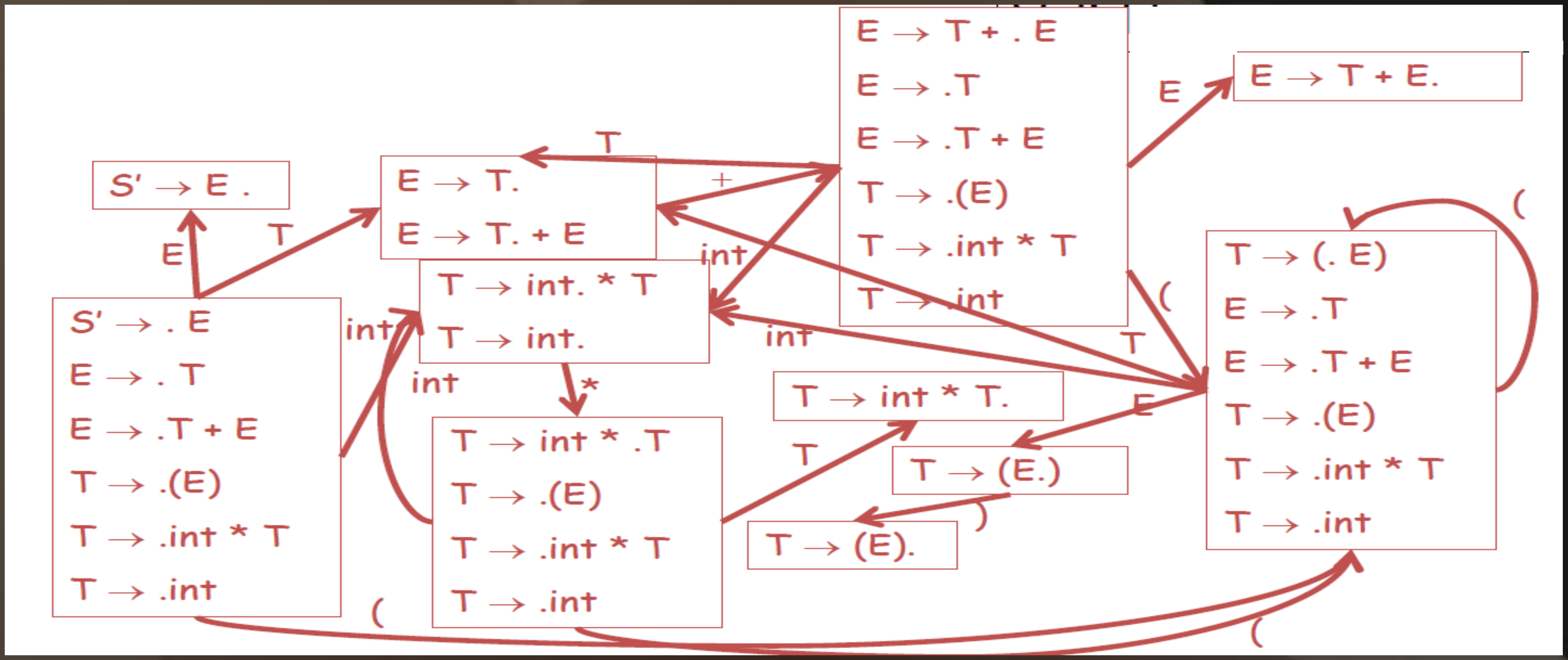
- Item $X \rightarrow \beta.\gamma$ is *valid for a viable prefix $\alpha\beta$ if*
  - $S' \rightarrow^{*} \alpha X \omega \rightarrow \alpha\beta\gamma\omega$ by a right-most derivation

- After parsing $\alpha\beta$, the valid items are the possible tops of the stack of items

- An item **I** is valid for a viable prefix $\boldsymbol{\alpha}$ if the DFA recognizing viable prefixes terminates on input $\boldsymbol{\alpha}$ in a state **S** containing **I**

Valid Items

- The items in **S** describe what the top of the item stack might be after reading input **α**
- An item is often valid for many prefixes
  - Example: The item **T → (.E)** is valid for prefixes
    - (, ((, (((, ((((, . . .



LR(0) Parsing

- Assume
  - stack contains **α**
  - next input token is **t**
  - DFA on input **α** terminates in state **s**

- Reduce by **X** → **β** if
  - **s** contains item **X** → **β**.

- Shift if
  - **s** contains item **X** → **β.tω**
  - **E**quivalent to saying **s** has a transition labeled **t**

LR(0) Parsing - Conflicts

- LR(0) has a reduce/reduce conflict if:
  - Any state has two reduce items:
  - $\mathbf{X} \rightarrow \boldsymbol{\beta}.$ and $\mathbf{Y} \rightarrow \boldsymbol{\omega}.$

- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item:
  - $\mathbf{X} \rightarrow \boldsymbol{\beta}.$ and $\mathbf{Y} \rightarrow \boldsymbol{\omega}.\mathbf{t}\boldsymbol{\delta}$

- SLR improves on LR(0) shift/reduce heuristics
  - Fewer states have conflicts

SLR Parsing

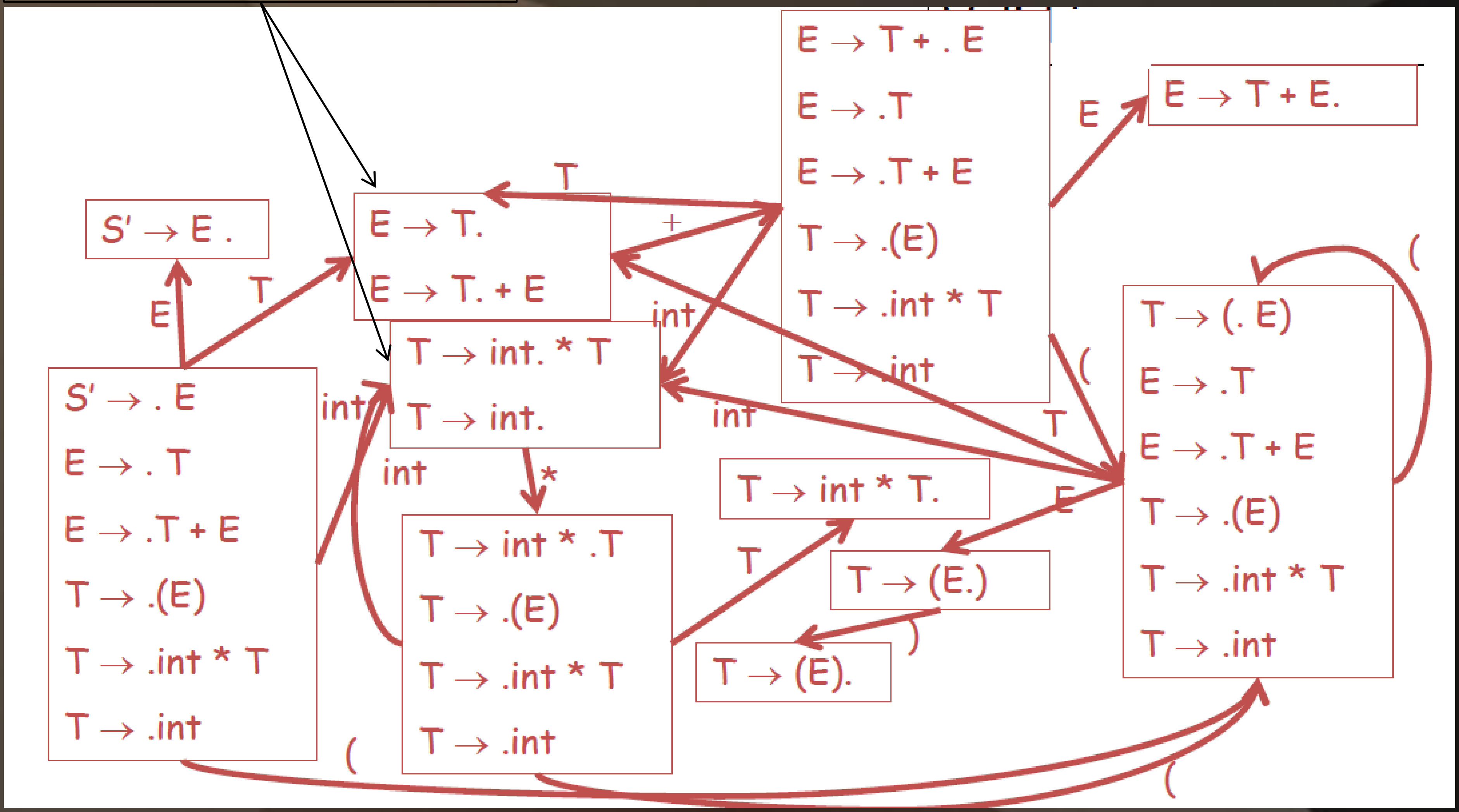
- Assume
  - stack contains **α**
  - next input token is **t**
  - DFA on input **α** terminates in state **s**

- Reduce by **X → β** if
  - **s** contains item **X → β.**
  - **t ϵ Follow (X)**

- Shift if
  - **s** contains item **X → β.tω**
  - **E**quivalent to saying **s** has a transition labeled **t**

The DFA Again

Shift-Reduce Conflicts



SLR Parsing Algorithm

1. Let **M** be DFA for viable prefixes of **G**
2. Let $|\mathbf{x_1 \ldots x_n \$}$ be initial configuration
3. Repeat until configuration is **S|$**
    1. Let $\boldsymbol{\alpha|\omega}$ be current configuration
    2. Run **M** on current stack $\boldsymbol{\alpha}$
    3. If M rejects $\boldsymbol{\alpha}$, report parsing error
        1. Stack $\boldsymbol{\alpha}$ is not a viable prefix
    4. If **M** accepts $\boldsymbol{\alpha}$ with items **I**, let **u** be next input
        1. Shift if $\mathbf{X \rightarrow \beta.u\gamma \in I}$
        2. Reduce if $\mathbf{X \rightarrow \beta. \in I}$ and $\mathbf{u \in Follow(X)}$
        3. Report parsing error if neither applies

SLR Parsing - Improvements

- Note that Step 3.3 is redundant
- If there is a conflict in the last step, grammar is not SLR(k)

- Lots of grammars are not SLR
    - Including all ambiguous grammars

- We can parse more grammars by using precedence declarations
    - Instructions for resolving conflicts

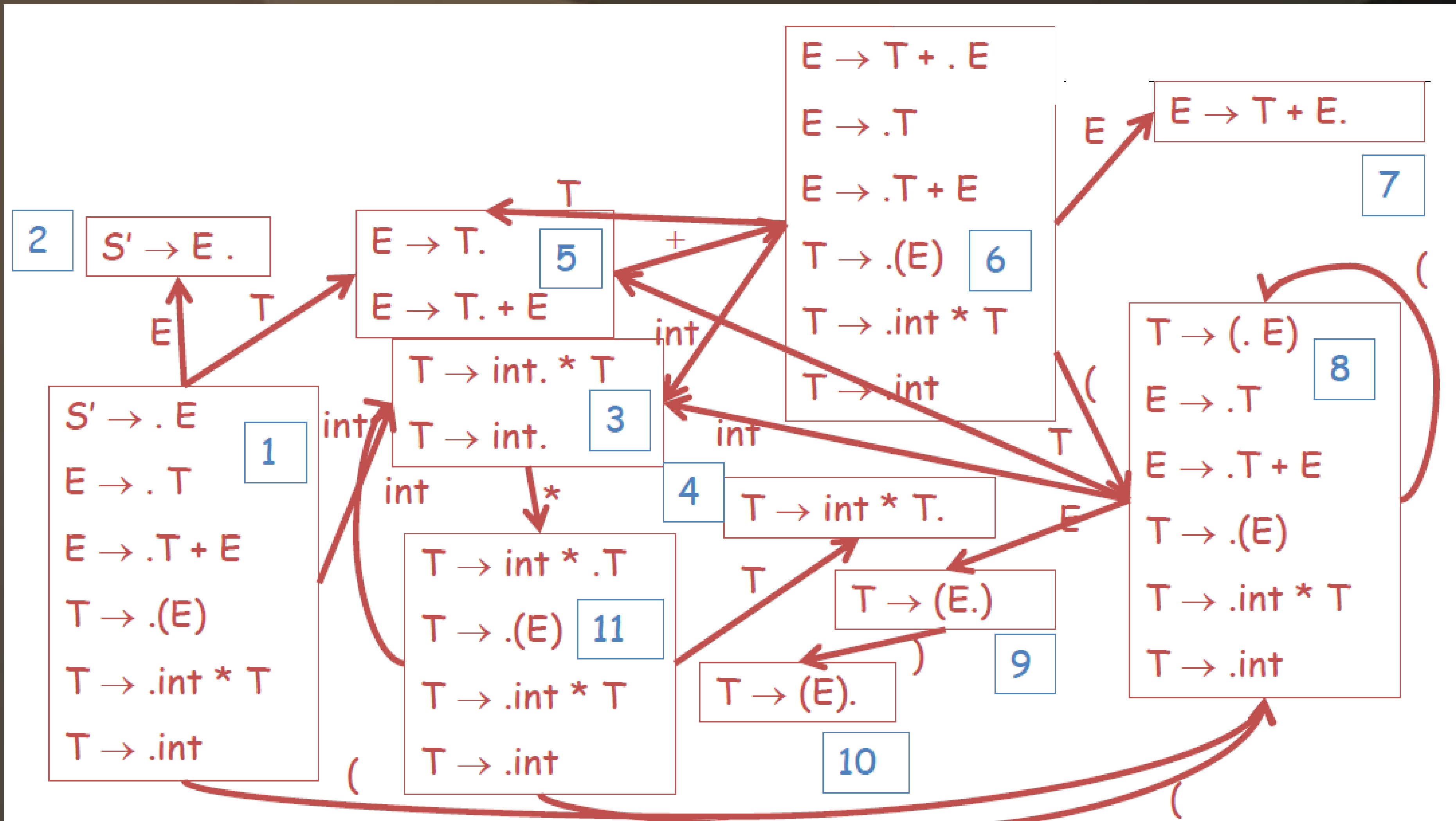
SLR Parsing

- Consider the ambiguous grammar:
  - **E → E + E | E * E | (E) | int**

- The DFA for this grammar contains a state with the following items:
  - **E → E * E.** and **E → E. + E**
  - There is a shift/reduce conflict

- Declaring "* has higher precedence than +" resolves this conflict in favour of reducing

SLR Parsing Example



SLR Parsing Example

Parse the token stream: **int * int$**

Configuration	*DFA Halt State*	*Action*
|int * int$	1	Shift
int | * int$	3 * not in Follow(T)	Shift
int * | int$	11	Shift
int * int |$	3 $ ϵ Follow(T)	Reduce. T→ int
int * T |$	4 $ ϵ Follow(T)	Reduce. T → int * T
T |$	5 $ ϵ Follow(T)	Reduce. E → T
E |$	Accept	

Next Lecture

Bottom-Up Parsing Continued…