Lecture 8: Bottom-up Analysis

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
School of Computer Science E301

In A Nutshell

What is Bottom-Up Parsing?

- Idea: Apply productions **in reverse** to convert the user's program to the start symbol.
  - We can think of bottom-up parsing as the process of "reducing" a string *w* to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

- Keywords
  - Reductions, handle, shift-reduce parsing, conflicts, LR grammars

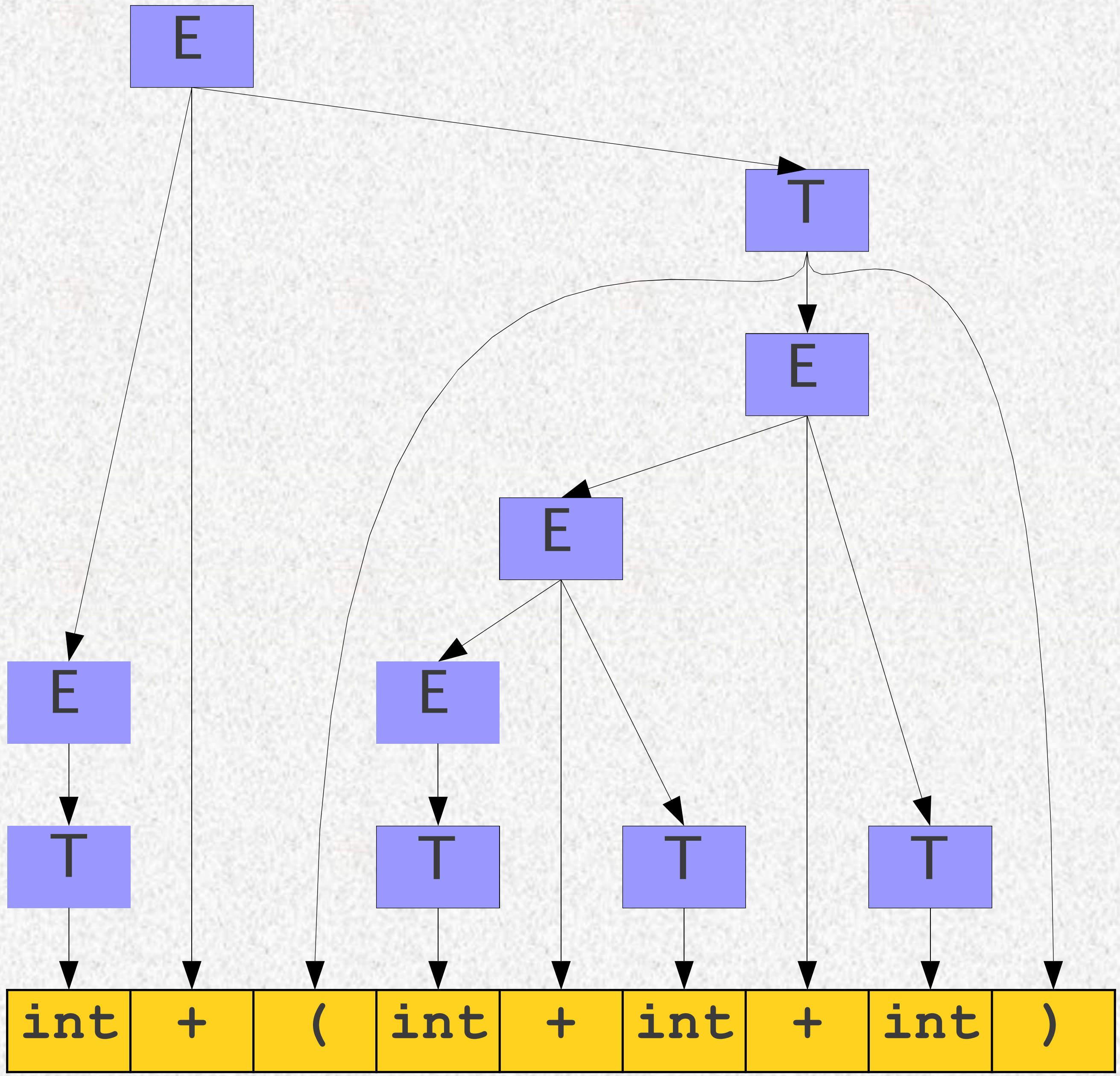
What is Bottom-Up Parsing?

- Four major **directional**, **predictive** bottom-up parsing techniques:
  - **Directional**: Scan the input from left-to-right.
  - **Predictive**: Guess which production should be inverted.
  - The largest class of grammars for which shift-reduce parsers can be built, the LR grammars: **LR(0), SLR(0), LR(1), LALR(1)**

A View of a Bottom-Up Parse

E → T
E → E + T
T → int
T → (E)

int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



A left-to-right, bottom-up parse is a rightmost derivation traced in reverse.

A View of a Bottom-Up Parse

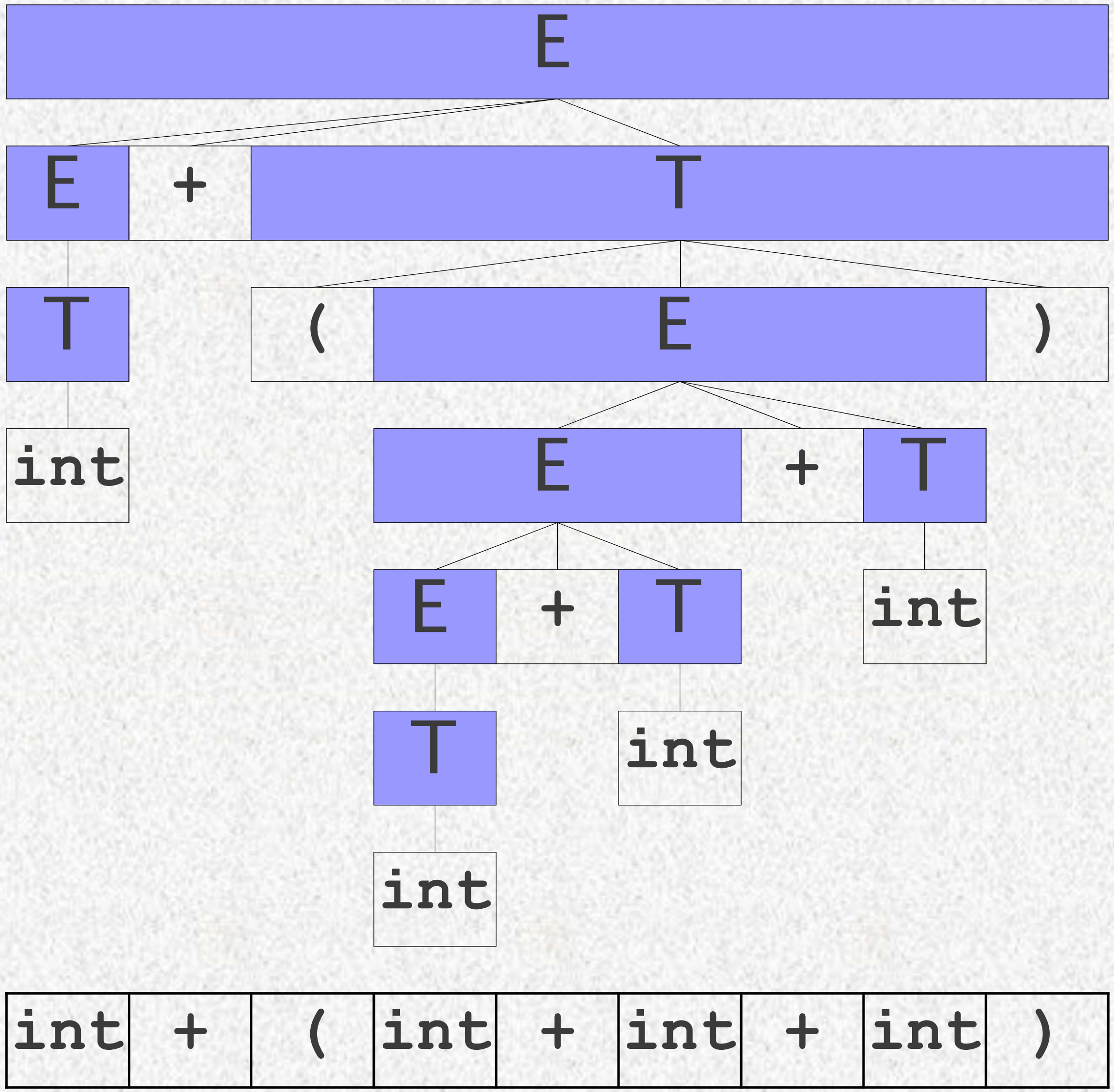
int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

Each step in this bottom-up parse is called a reduction. We reduce a substring of the sentential form back to a nonterminal (start symbol).

E → T
E → E + T
T → int
T → (E)

A View of a Bottom-Up Parse

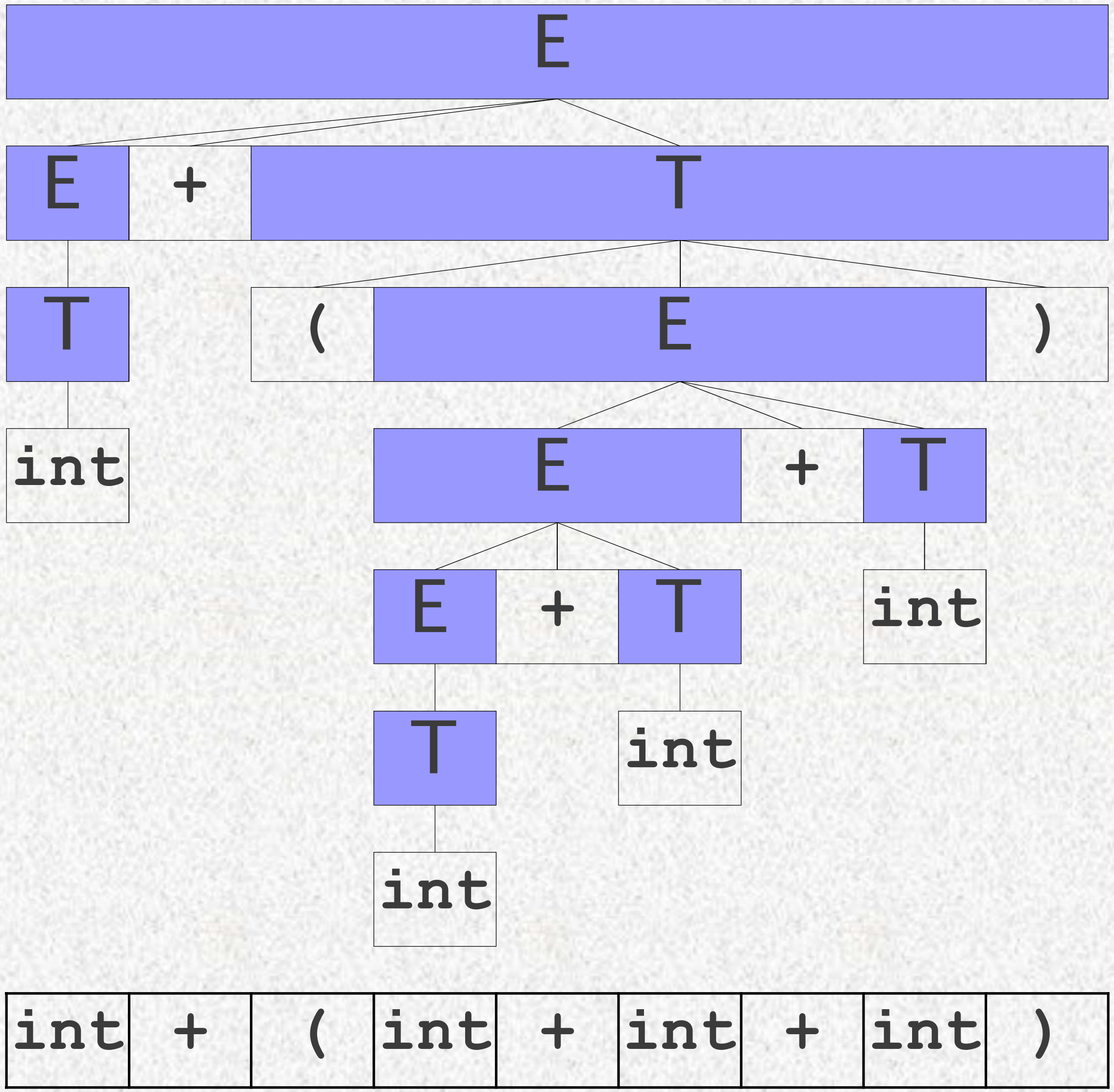
```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```



A View of a Bottom-Up Parse

E → T
E → E + T
T → int
T → (E)

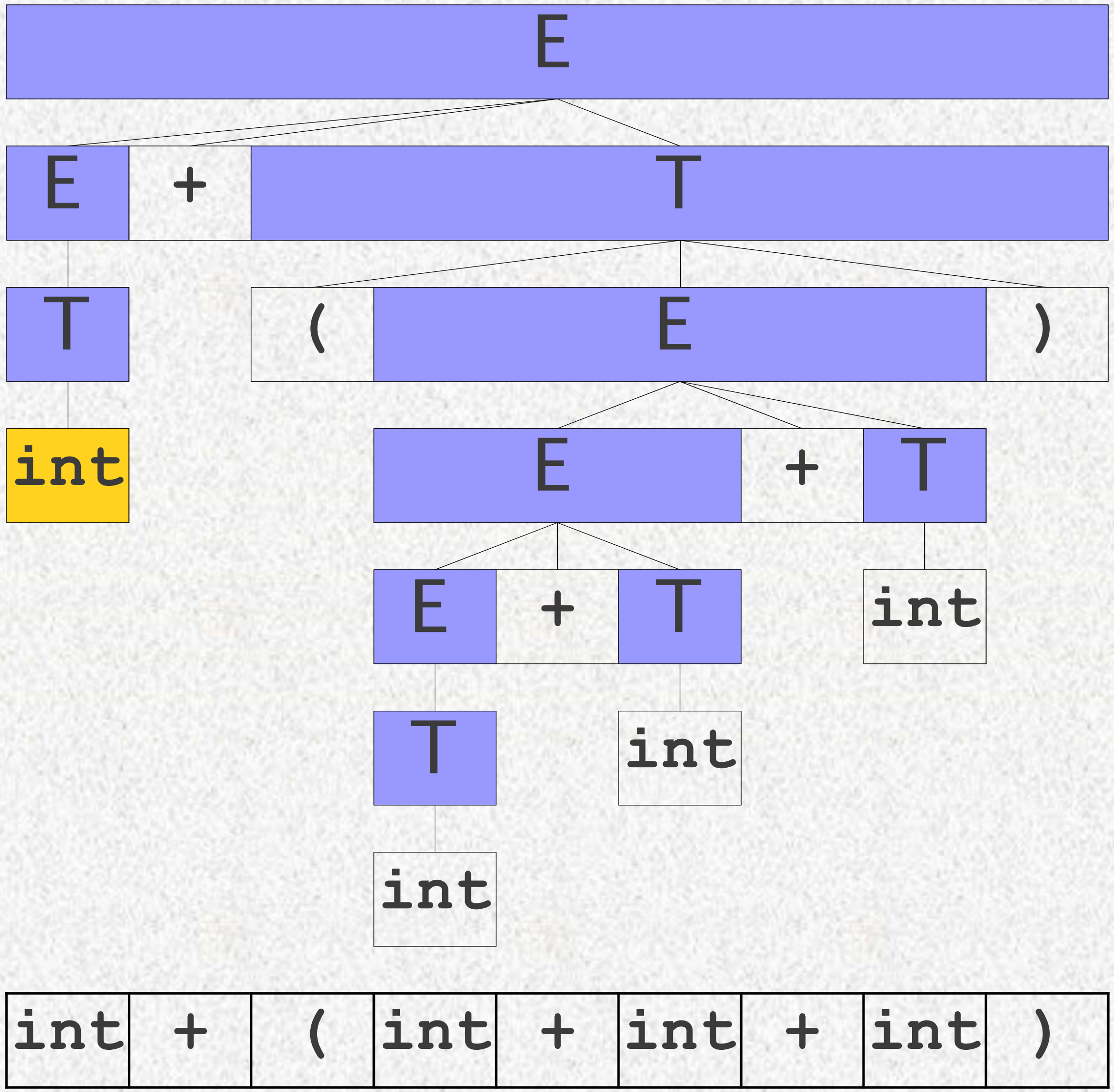
```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```



E → T
E → E + T
T → int
T → (E)

A View of a Bottom-Up Parse

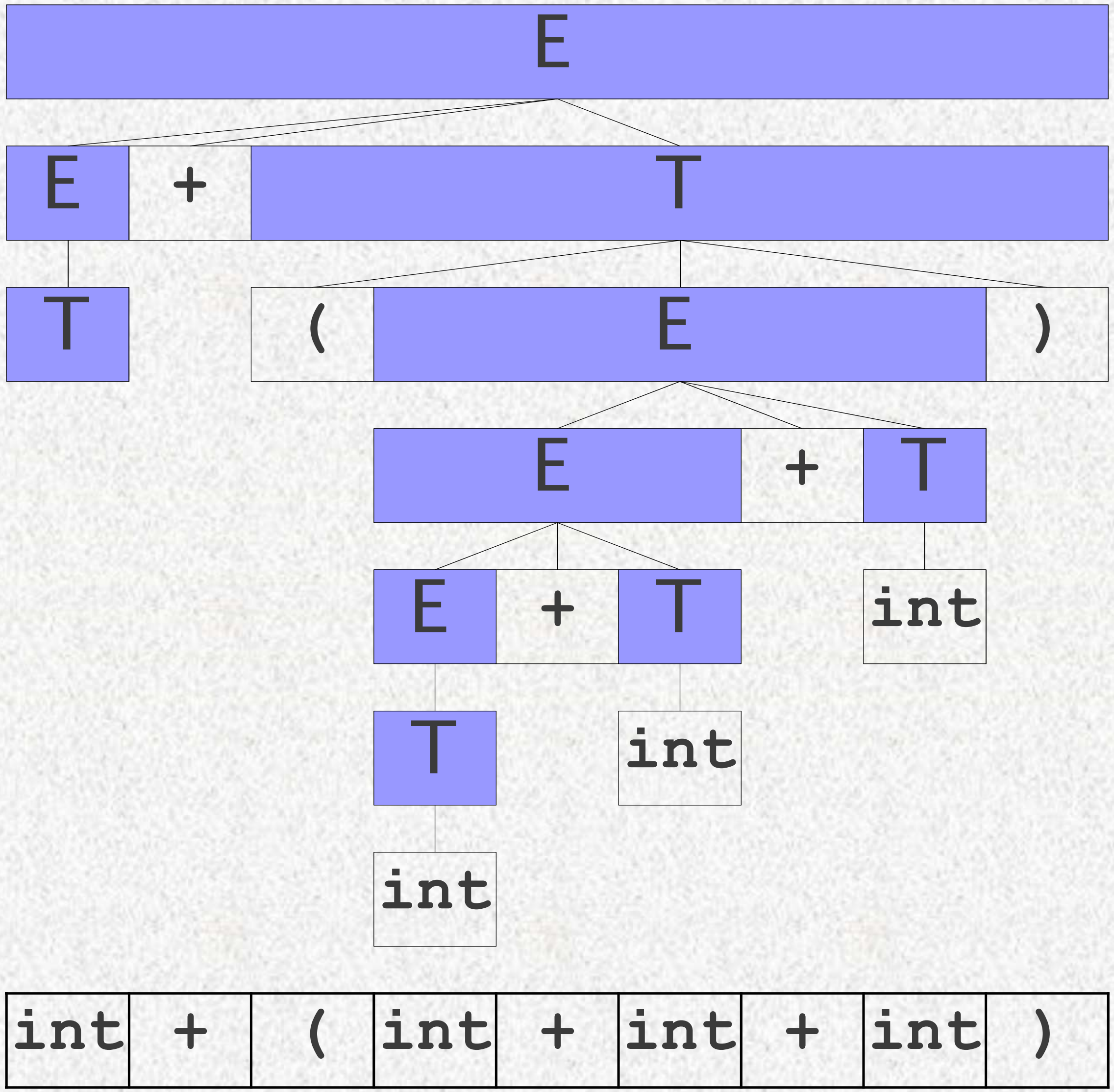
int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



A View of a Bottom-Up Parse

E → T
E → E + T
T → int
T → (E)

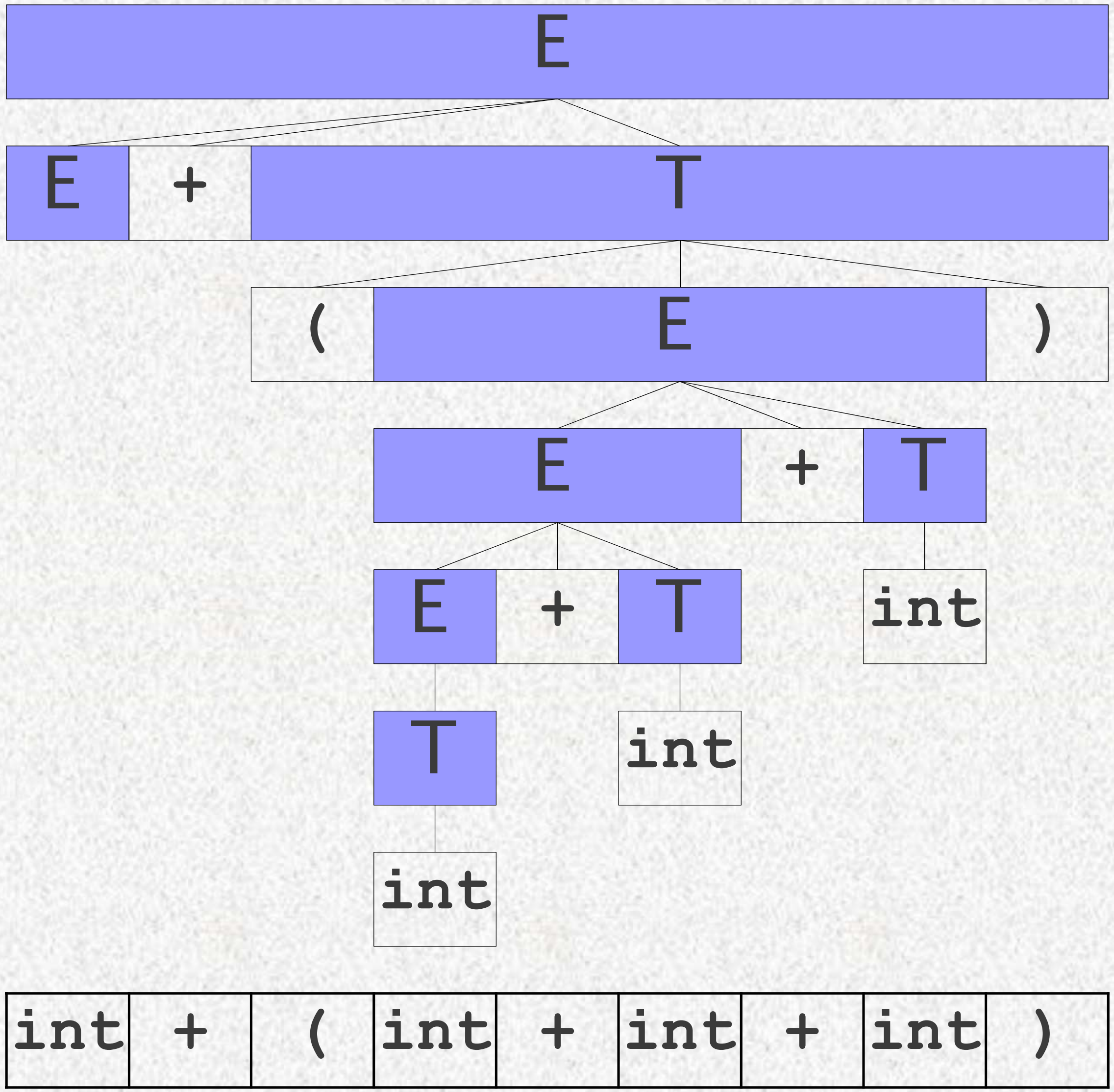
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



A View of a Bottom-Up Parse

E → T
E → E + T
T → int
T → (E)

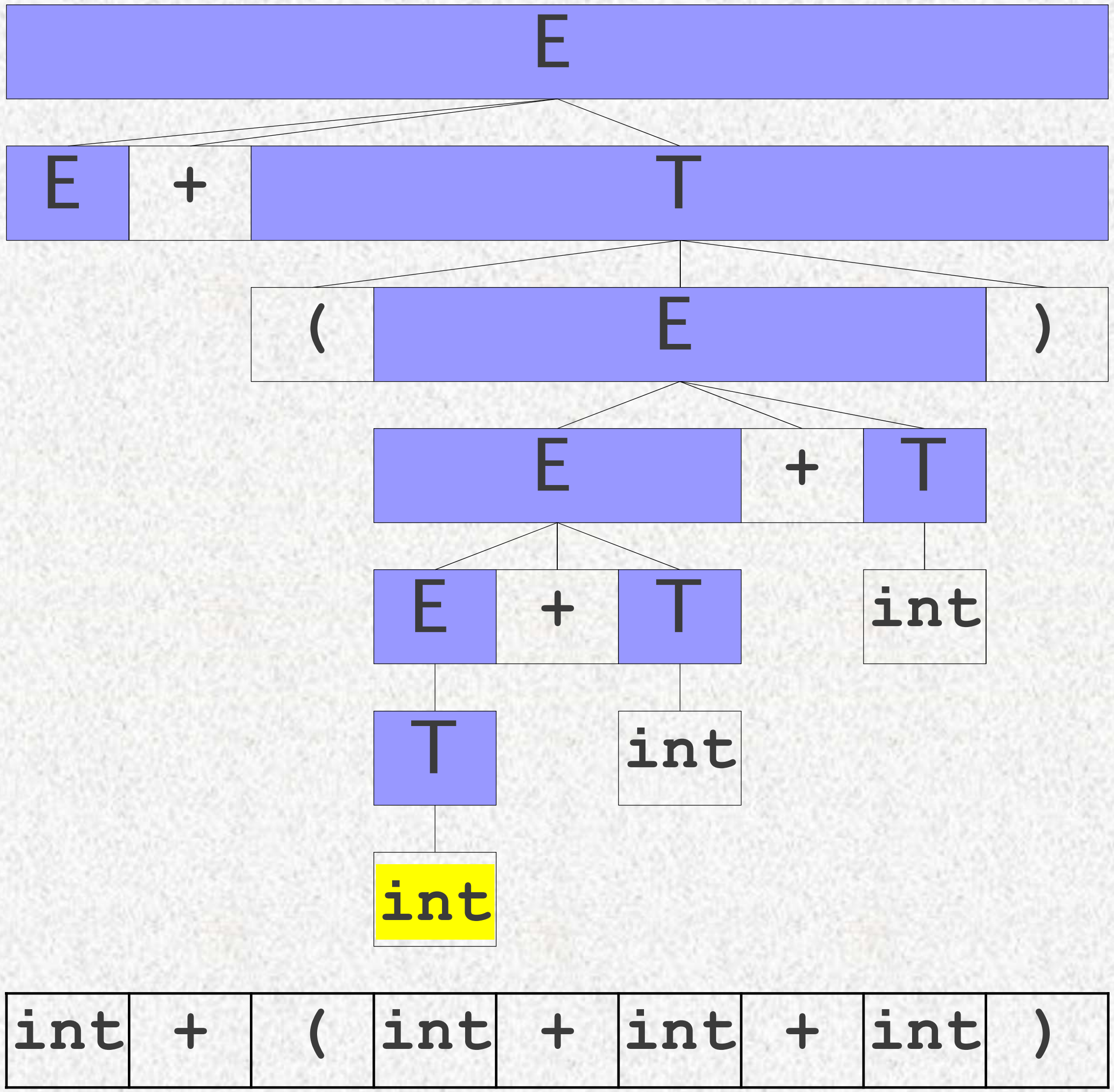
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



E → T
E → E + T
T → int
T → (E)

A View of a Bottom-Up Parse

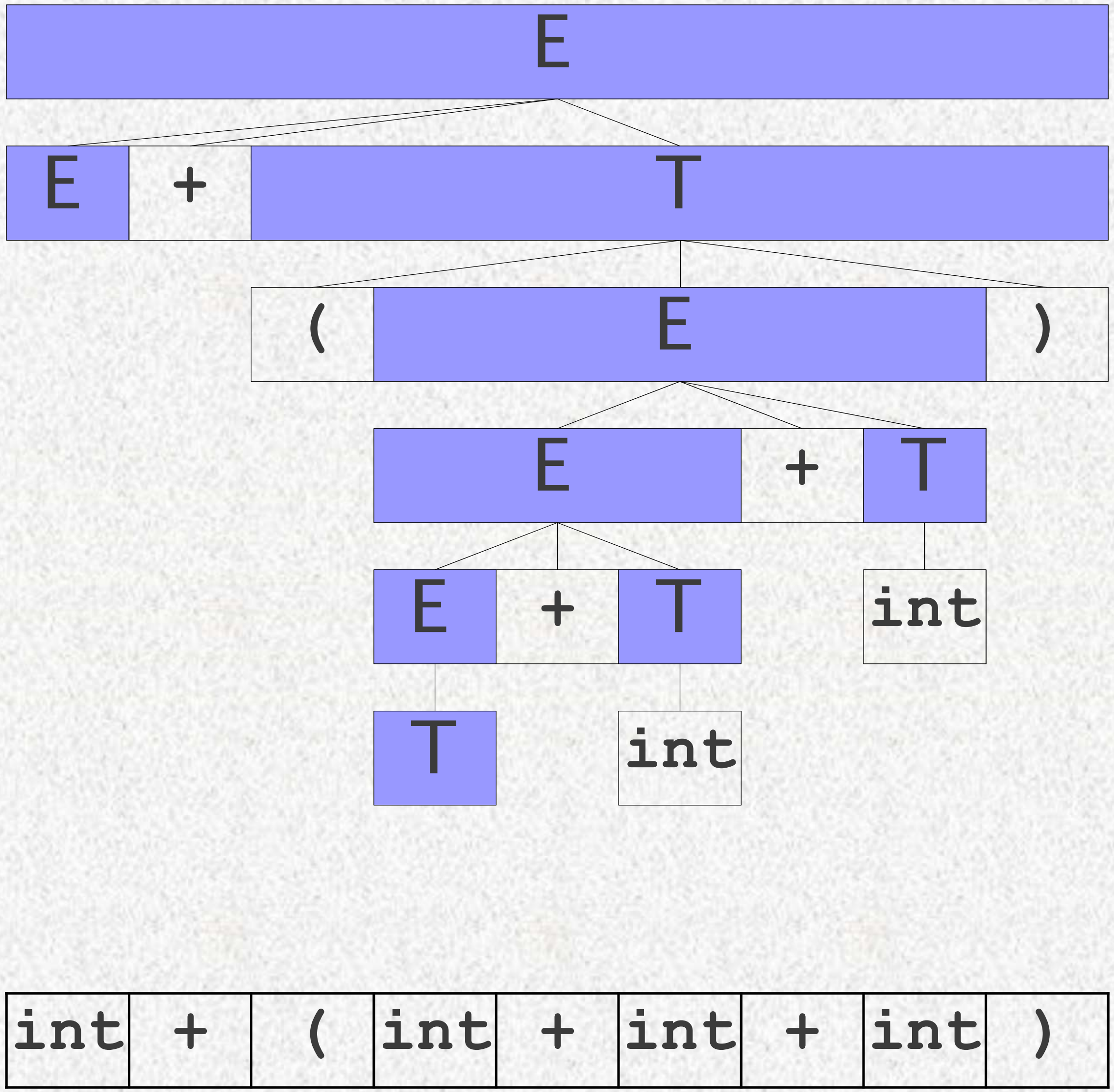
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



A View of a Bottom-Up Parse

E → T
E → E + T
T → int
T → (E)

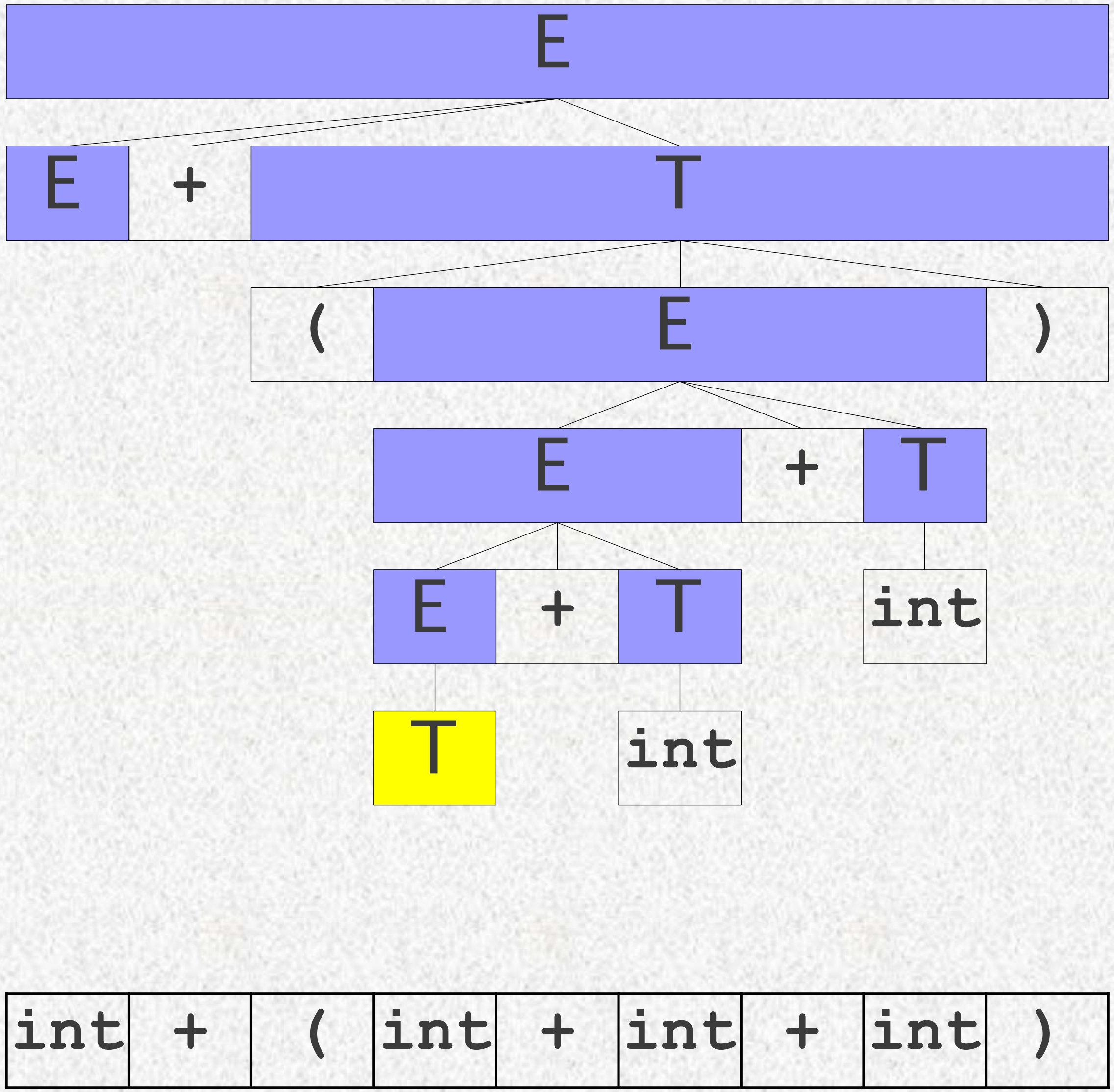
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



A View of a Bottom-Up Parse

E → T
E → E + T
T → int
T → (E)

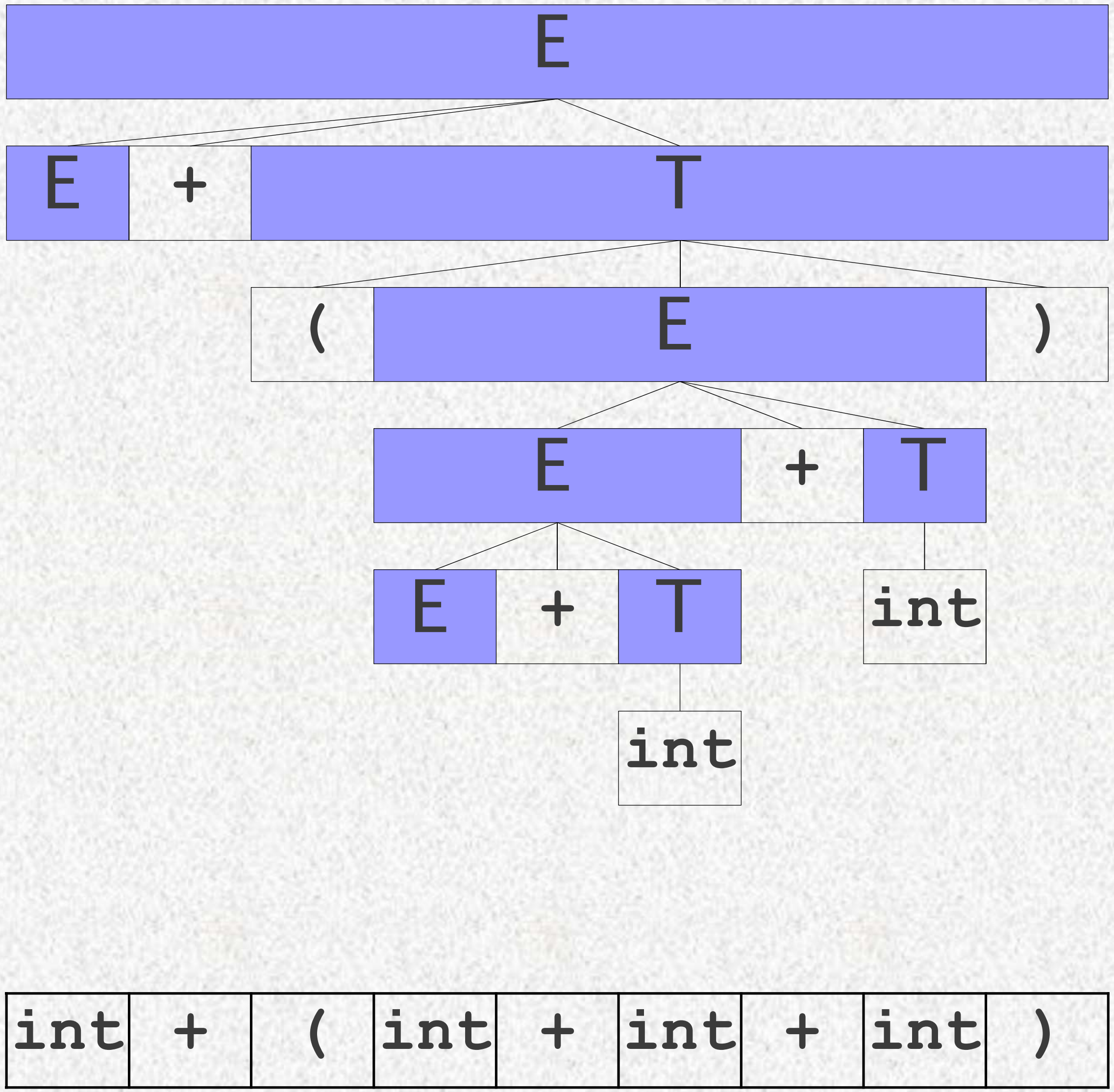
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



$E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow$ `int`
$T \rightarrow$ `(E)`

A View of a Bottom-Up Parse

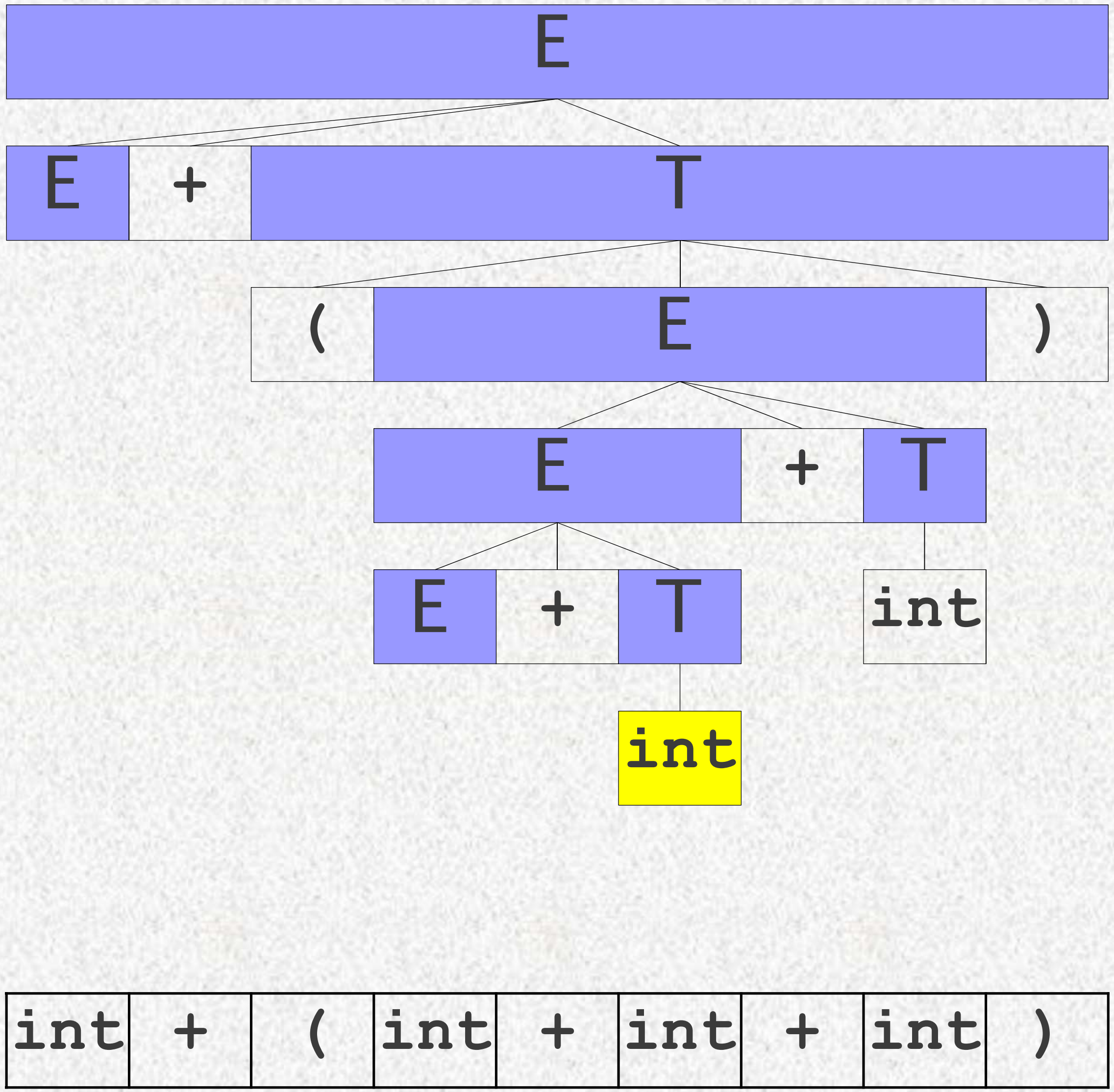
⇒ `E + (E + int + int)`
⇒ `E + (E + T + int)`
⇒ `E + (E + int)`
⇒ `E + (E + T)`
⇒ `E + (E)`
⇒ `E + T`
⇒ `E`



A View of a Bottom-Up Parse

E → T
E → E + T
T → int
T → (E)

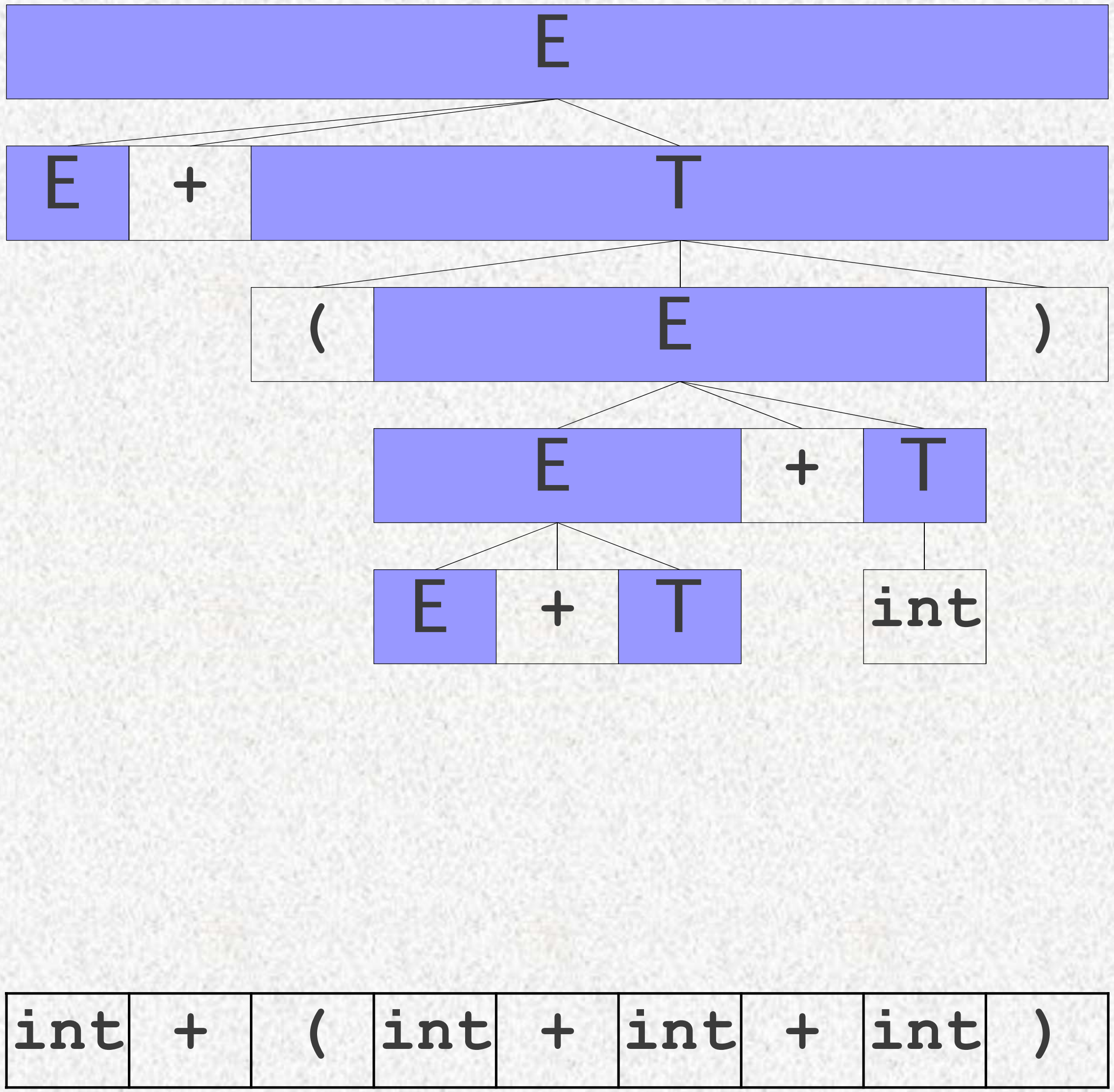
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



A View of a Bottom-Up Parse

E → T
E → E + T
T → int
T → (E)

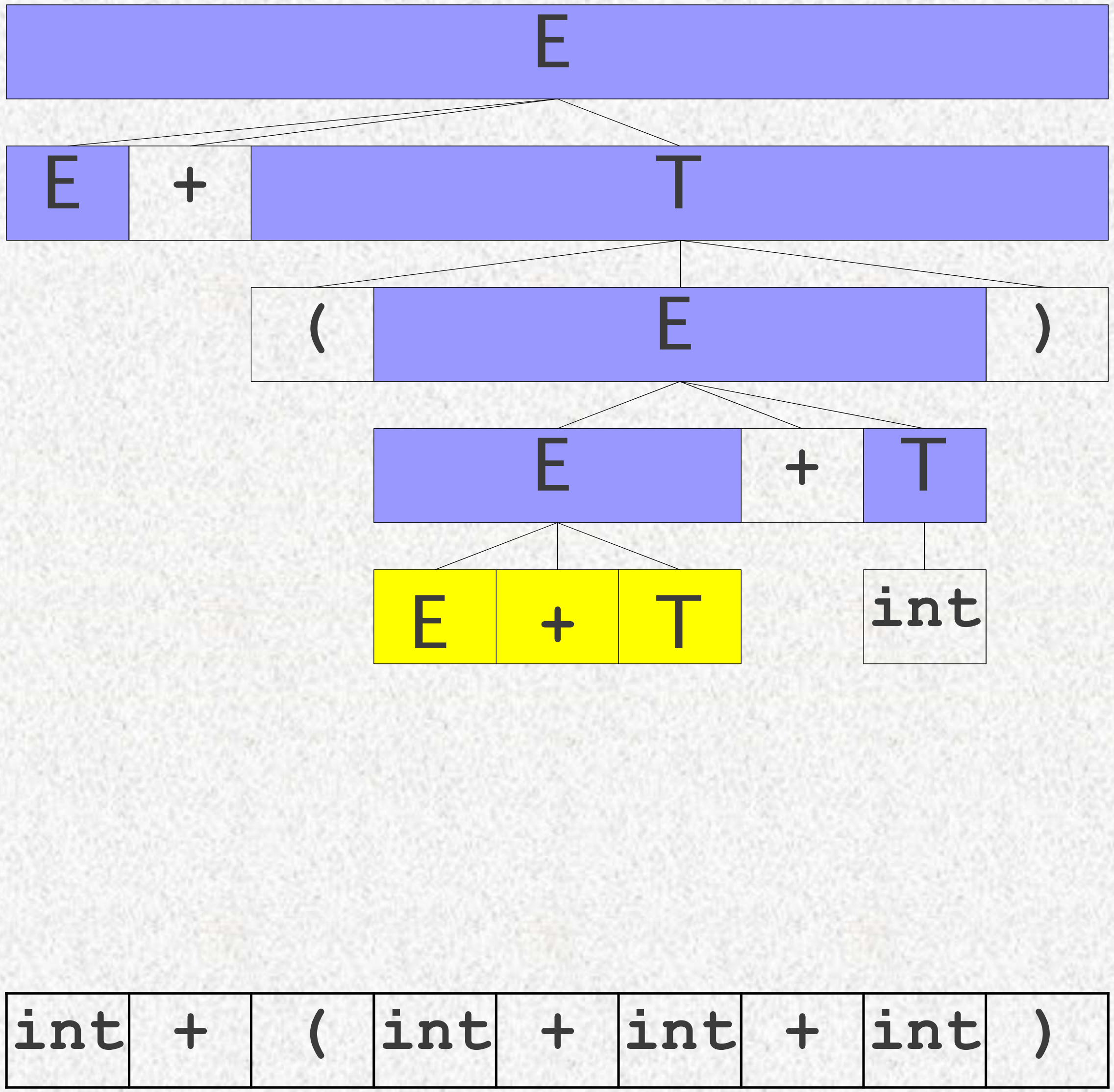
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



E → T
E → E + T
T → `int`
T → (E)

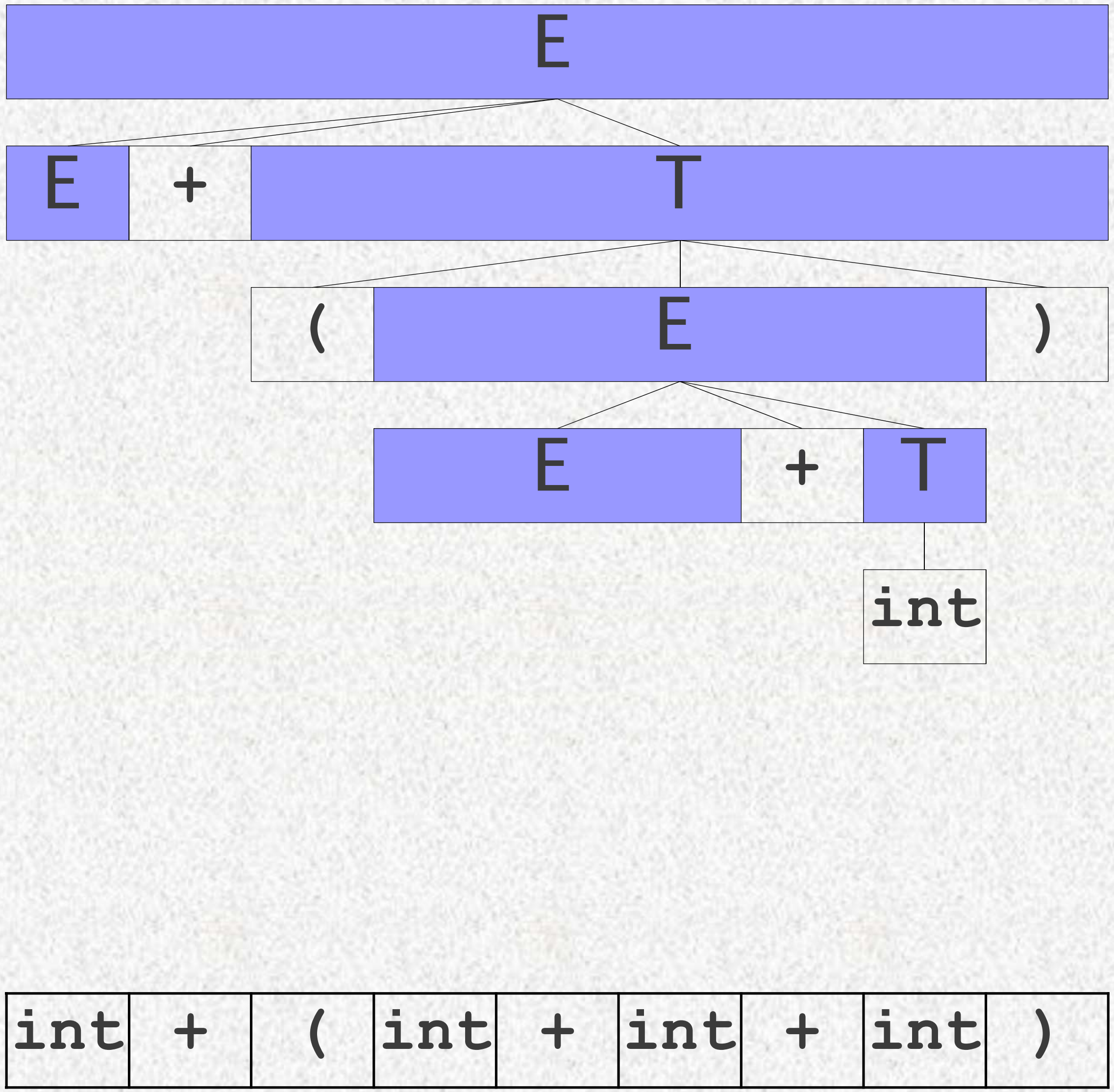
A View of a Bottom-Up Parse

⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



A View of a Bottom-Up Parse

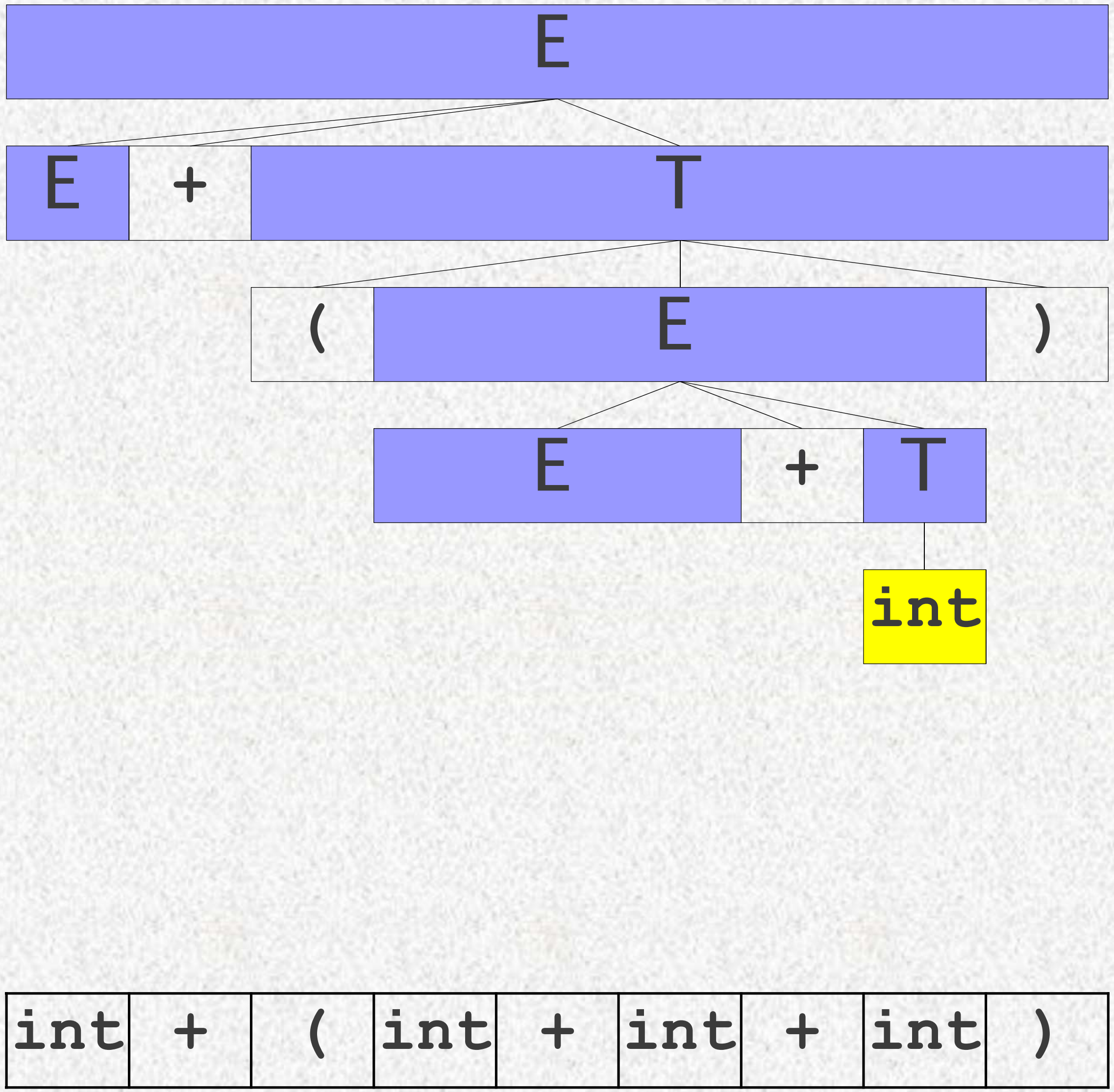
E → T
E → E + T
T → int
T → (E)



⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

A View of a Bottom-Up Parse

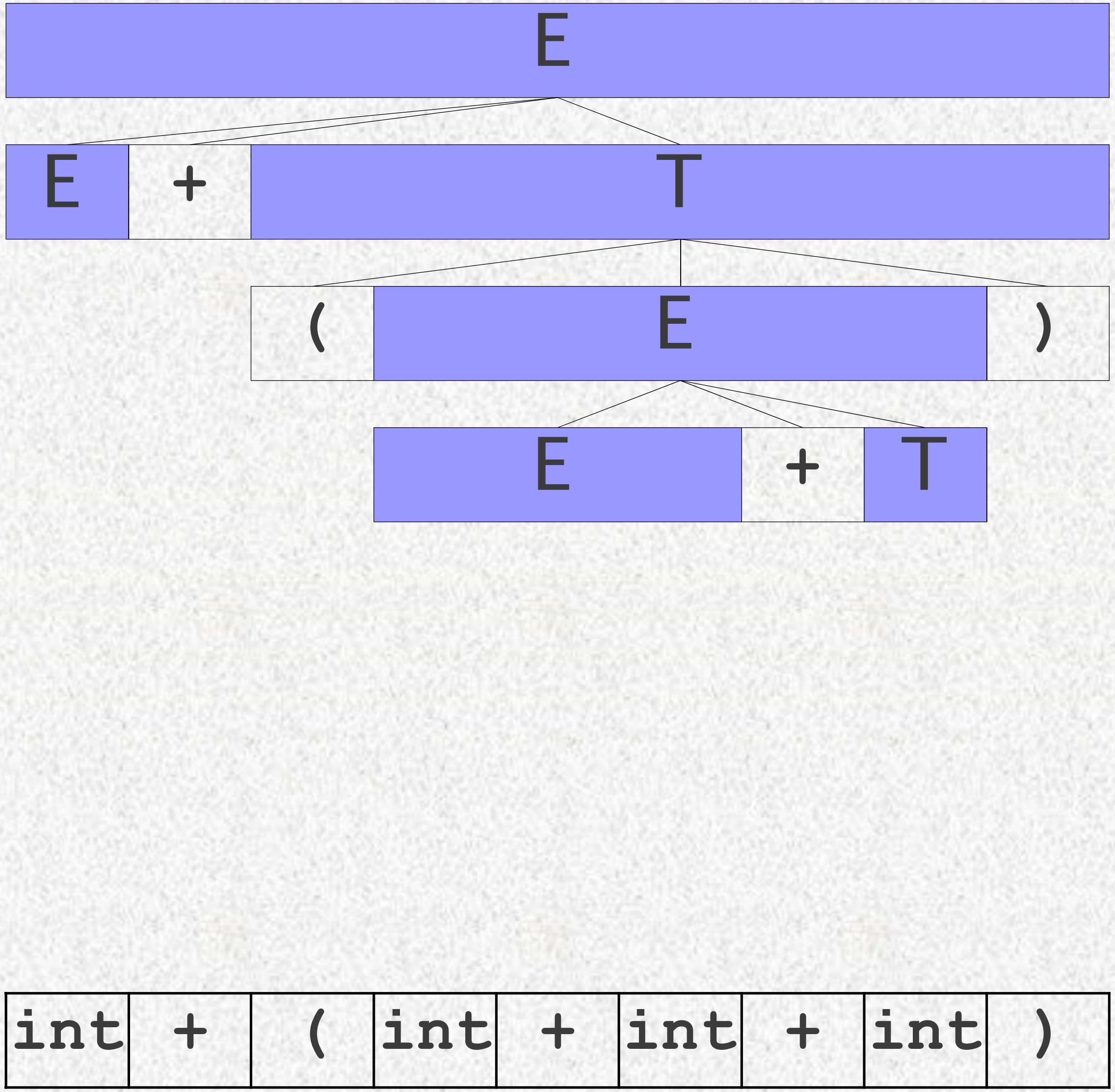
E → T
E → E + T
T → int
T → (E)



⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

A View of a Bottom-Up Parse

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

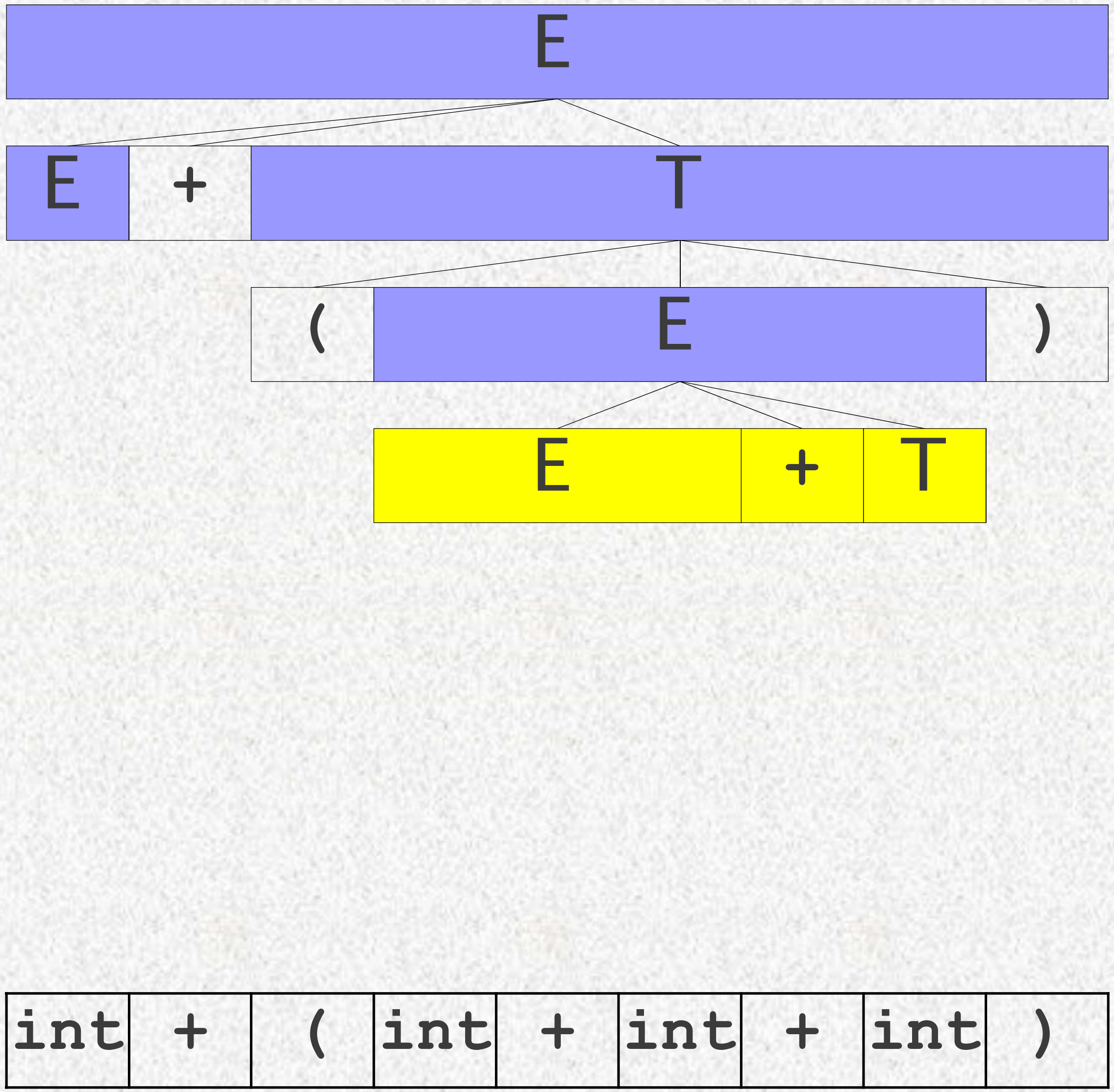


⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

A View of a Bottom-Up Parse

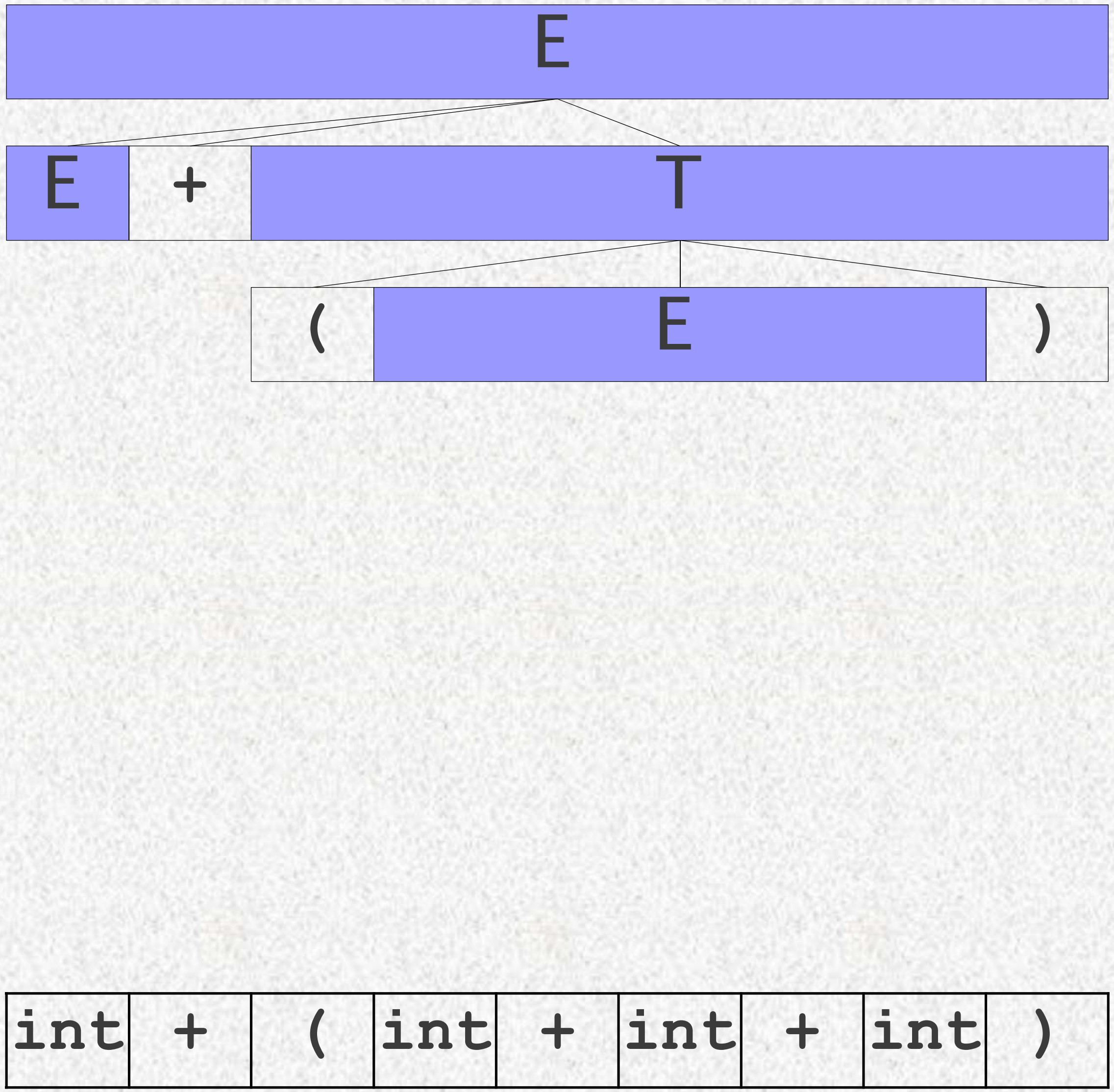
E → T
E → E + T
T → int
T → (E)

⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



A View of a Bottom-Up Parse

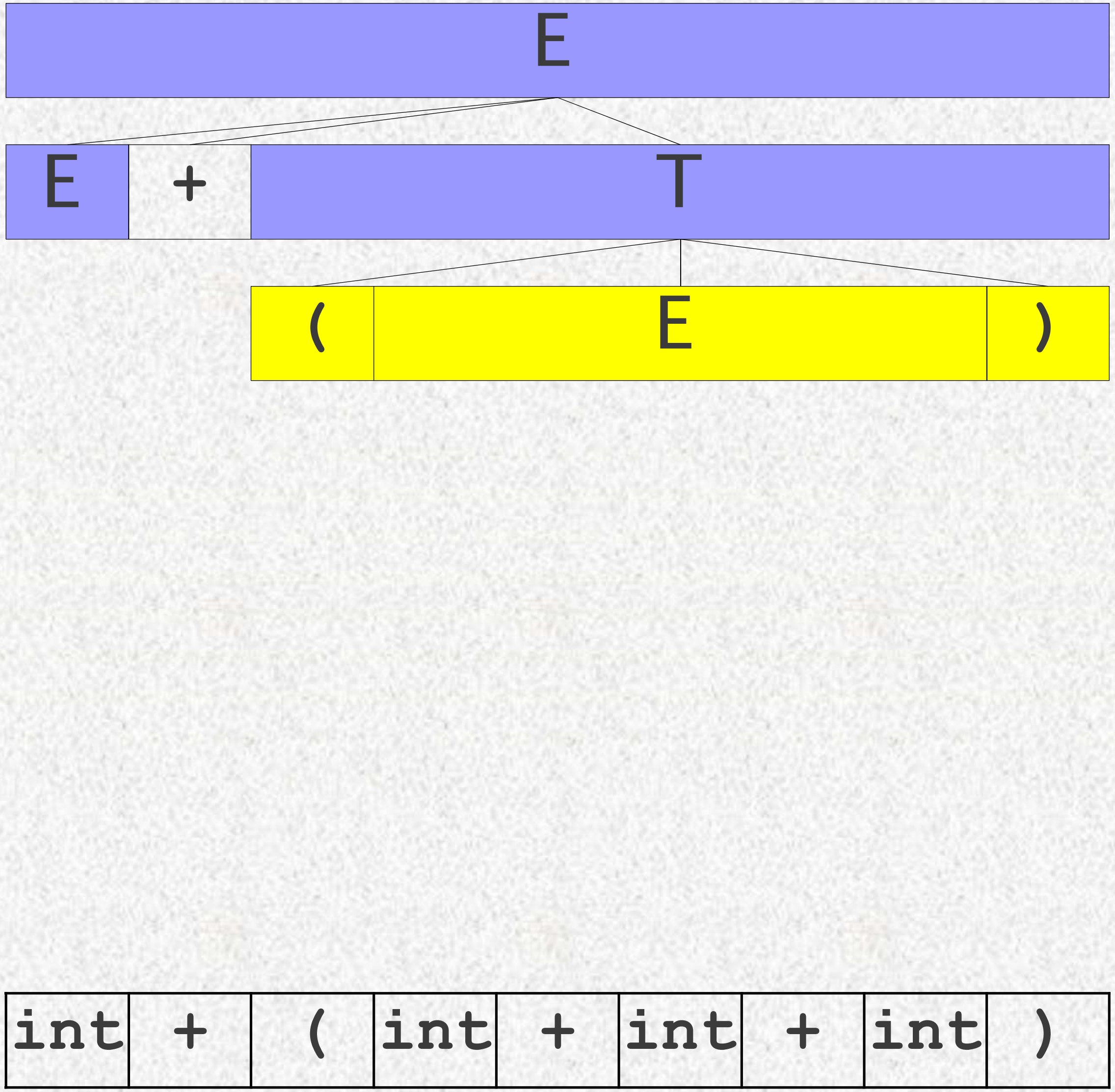
$E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow$ `int`
$T \rightarrow$ `(E)`



$\Rightarrow E + (E)$
$\Rightarrow E + T$
$\Rightarrow E$

A View of a Bottom-Up Parse

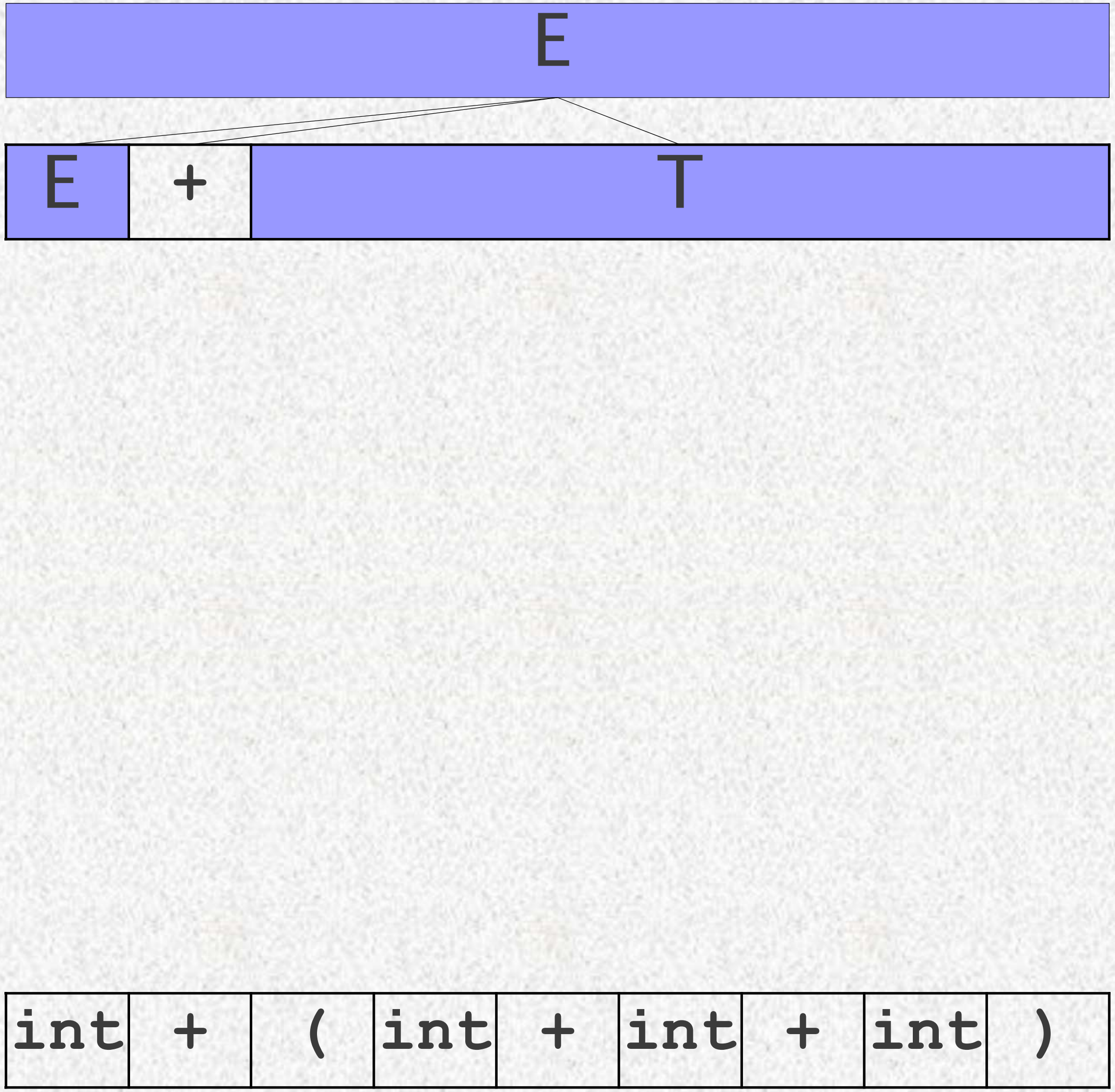
$E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow$ `int`
$T \rightarrow$ `(E)`



$\Rightarrow E + (E)$
$\Rightarrow E + T$
$\Rightarrow E$

A View of a Bottom-Up Parse

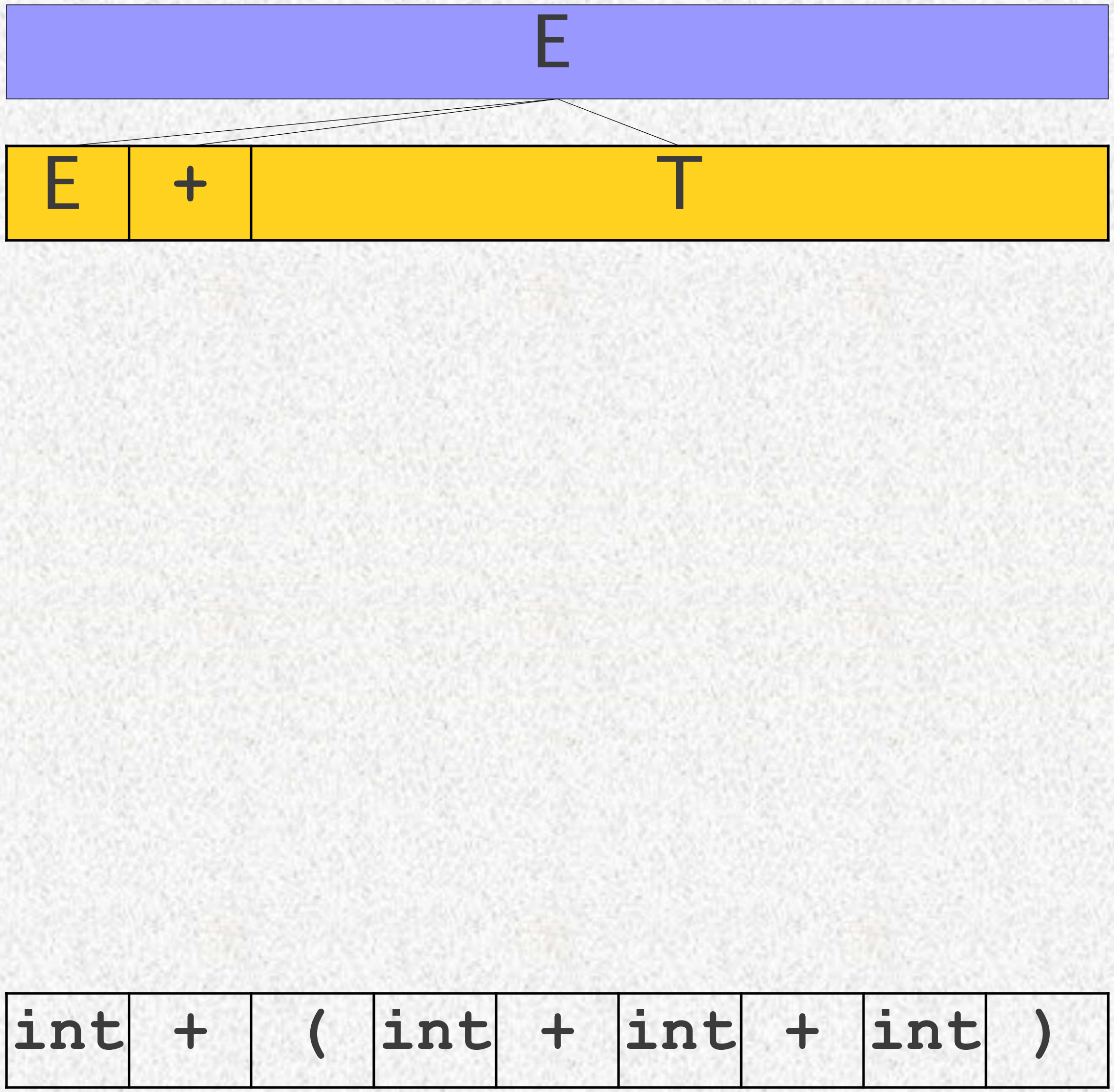
E → T
E → E + T
T → int
T → (E)



⇒ E + T
⇒ E

A View of a Bottom-Up Parse

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



⇒ **E + T**
⇒ **E**

A View of a Bottom-Up Parse

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

E

⇒ **E**

`int`	`+`	`(`	`int`	`+`	`int`	`+`	`int`	`)`

Preliminaries

Basic Concepts

- How to build a predictive bottom-up parser?
- Sentential form
  - For a grammar G with start symbol S

    A string α is a sentential form of G if $S \Rightarrow^* \alpha$
    - α may contain terminals and nonterminals
    - If α is in T^*, then α is a sentence of L(G)
  - Left sentential form: A sentential form that occurs in the leftmost derivation of some sentence
  - Right sentential form: A sentential form that occurs in the rightmost derivation of some sentence

Basic Concepts

- Example of the sentential form
  - $E \rightarrow E * E \mid E + E \mid (E) \mid id$
  - Leftmost derivation:

    $E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow id * E + E \Rightarrow id * id + E \Rightarrow$
    $id * id + E * E \Rightarrow id * id + id * E \Rightarrow id * id + id * id$

    - All the derived strings are of the left sentential form
  - Rightmost derivation

    $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * id \Rightarrow E + id * id \Rightarrow$
    $E * E + id * id \Rightarrow E * id + id * id \Rightarrow id * id + id * id$

    - All the derived strings are of the right sentential form

A Small example

E → E + T | T
T → T * F | F
F → (E) | id

A Rightmost Derivation:

$E \rightarrow \underline{E + T}$

$\rightarrow E + \underline{T * F}$

$\rightarrow E + T * \underline{id}$

$\rightarrow E + \underline{F} * id$

$\rightarrow E + \underline{id} * id$

$\rightarrow \underline{T} + id * id$

$\rightarrow \underline{F} + id * id$

$\rightarrow \underline{id} + id * id$

The Parsing Problem

- Given a right sentential form, α, determine what substring of α is the **right-hand side (RHS)** of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation
- The correct RHS is called the <u>handle</u>

Basic Concepts

- Informally, a **handle of a string is a substring that matches the right side of a production rule**.
  - *But not every substring matches the right side of a production rule is handle*
  - *Reduction of a handle represents one step along the reverse of a rightmost derivation*

Basic Concepts

- A **handle** of a right sentential form γ ($\equiv \alpha\beta\omega$) is a production rule $A \to \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.
  $S \overset{rm}{\Rightarrow}{}^* \alpha \mathbf{A} \omega \overset{rm}{\Rightarrow}{}^* \alpha\beta\omega$

- Alternatively, a handle of a right-sentential form γ is a production $A \to \beta$ and a position of γ where the string β may be found, such that replacing β at that position by A produces the previous right-sentential form in a rightmost derivation of γ.

Basic Concepts

- **Handle**
  - Given a rightmost derivation

    $S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_k \Rightarrow \gamma_{k+1} \Rightarrow \ldots \Rightarrow \gamma_n$
    - γ_i, for all i, are the right sentential forms
    - $\gamma_k = \alpha A w$; $\gamma_{k+1} = \alpha\beta w$
    - From γ_k to γ_{k+1}, production $A \rightarrow \beta$ is used
  - For convenience, we refer to the body β rather than $A \rightarrow \beta$ as a handle.

Basic Concepts

– Def: β is the handle of the right sentential form

$\gamma = \alpha\beta w$ if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$

$E \rightarrow \underline{E + T}$

$\rightarrow E + \underline{T * F}$

$\rightarrow E + T * \underline{id}$

$\rightarrow E + \underline{F} * id$

Let $\gamma = \alpha\beta w$ be

$E + \underline{F} * id$

What is β ?

What is w ?

What is α ? What is A?

Basic Concepts

– Def: β is a <u>phrase</u> of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$

$E \to \underline{E + T}$ Let $\gamma = \alpha_1 A \alpha_2$ be

$\to E + \underline{T * F}$ $\underline{E + T}$

$\to E + T * \underline{id}$ Let A be T. What is α_1 ? α_2 ?

What can β be?

Basic Concepts

– Def: β is a <u>simple phrase</u> of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$

$E \rightarrow \underline{E + T}$ Let $\gamma = \alpha_1 A \alpha_2$ be

$\rightarrow E + \underline{T * F}$ $\underline{E + T}$

$\rightarrow E + T * \underline{id}$ Let A be T. What is α_1 ? α_2 ?

What can β be?

- The handle of any rightmost sentential form is its leftmost simple phrase

Handles

- The **handle** of a parse tree *T* is the leftmost complete cluster of leaf nodes.
- A left-to-right, bottom-up parser works by iteratively searching for a handle, then reducing the handle.

Basic Concepts

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

Figure 4.26: Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

Basic Concepts

- Example

Sentential Form: T+ (E+T)*i

P:
(1) $E \rightarrow T$
(2) $E \rightarrow E + T$
(3) $T \rightarrow F$
(4) $T \rightarrow T * F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow i$
(7) $F \rightarrow n$

A derivation of this sentential form (not a rightmost derivation)
$E \Rightarrow E + T \Rightarrow E + T*F \Rightarrow E+T*i \Rightarrow$
$E+F*i \Rightarrow E+(E)*i \Rightarrow E+(E+T)*i$
$\Rightarrow T+(E+T)*i$

Phrases: T +(E+T)*i, T, E+T, i, (E+T), (E+T)*i

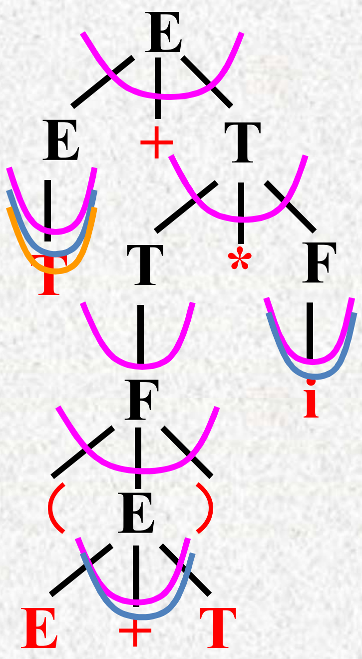
Simple phrases: T, E+T, i

Handle: T

Given a sentential form, build a parsing tree, then it will be easy to identify a handle

Basic Concepts

- Illustration via Parse Tree



<u>Sentential form</u>: leave nodes (from left to right)

T+ (E+T)*i

<u>Phrases: leave nodes of each subtree</u>

T+ (E+T)*i 、T、(E+T)*i、(E+T)、E+T、i

<u>Simple phrase: leave nodes of all simple subtree</u>

(i.e. a subtree with only one level of leaves)

T、E+T、i

<u>Handle: leave nodes of the leftmost simple subtree</u>

T

Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.

- $S=\gamma_0 \overset{rm}{\Rightarrow} \gamma_1 \overset{rm}{\Rightarrow} \gamma_2 \overset{rm}{\Rightarrow} \ldots \overset{rm}{\Rightarrow} \gamma_{n-1} \overset{rm}{\Rightarrow} \gamma_n = \omega$
- input string

- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until reach the start nonterminal *S*.

Homework

Page 240, 4.5.1
Page 241, 4.5.3(a)

LR Parsing

The Parsing Problem

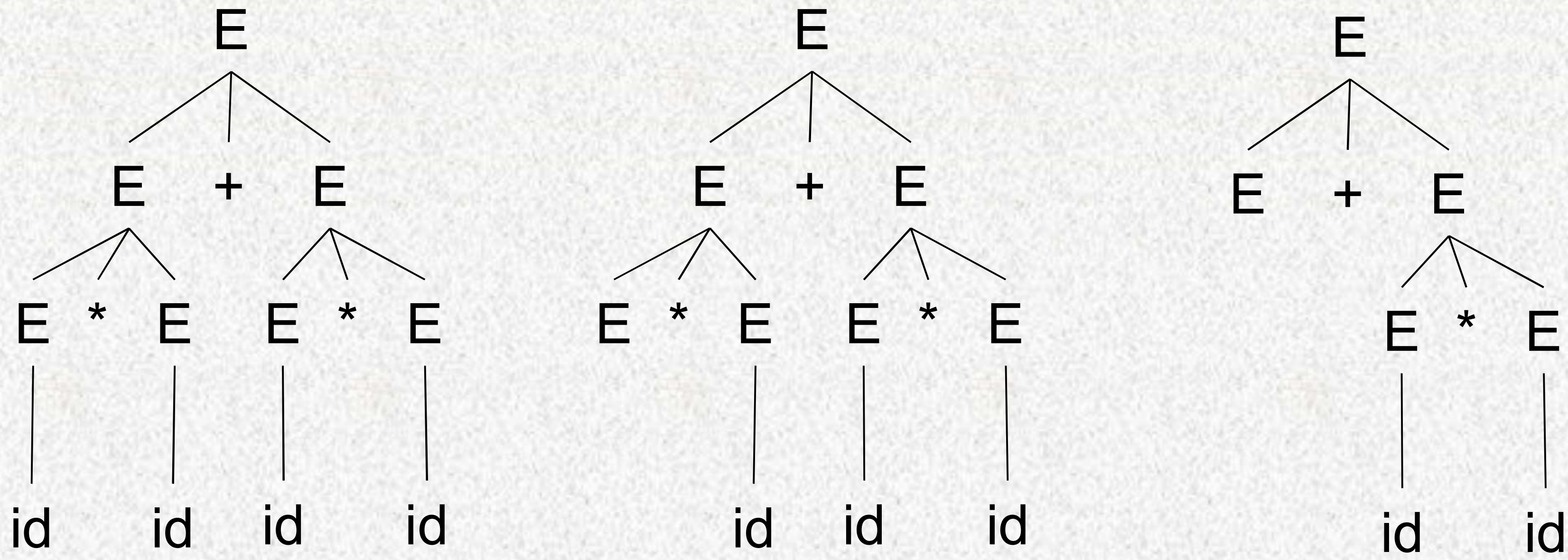
- Produce a parse tree starting at the leaves
- The order will be that of a rightmost derivation
- The most common bottom-up parsing algorithms are in the LR family

  L – Read the input left to right

  R – Trace out a rightmost parse tree

Meaning of LR

- L: Process input from left to right
- R: Use rightmost derivation, but in reversed order
- $E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * id \Rightarrow E + id * id$
  $\Rightarrow E * E + id * id \Rightarrow E * id + id * id \Rightarrow id * id + id * id$



LR Parsers Use Shift-Reduce

- Shift-Reduce Algorithms
  - Reduce: replace the handle on the top of the parse stack with its corresponding LHS
  - Shift: move the next token to the top of the parse stack

LR Parsers Use Shift-Reduce

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \,\$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2 \,\$$	reduce by $F \to \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2 \,\$$	reduce by $T \to F$
$\$\,T$	$* \mathbf{id}_2 \,\$$	shift
$\$\,T *$	$\mathbf{id}_2 \,\$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \to T * F$
$\$\,T$	$\$$	reduce by $E \to T$
$\$\,E$	$\$$	accept

Figure 4.28: Configurations of a shift-reduce parser on input $\mathbf{id}_1 * \mathbf{id}_2$

Shift/Reduce/Accept/Error

A Shift-Reduce Parser

- $E \rightarrow E+T \mid T$
- $T \rightarrow T*F \mid F$
- $F \rightarrow (E) \mid id$

Right-Most Derivation of id+id*id

$$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$$
$$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id$$

Right-Most Sentential Form	Reducing Production
id+id*id	$F \rightarrow id$
F+id*id	$T \rightarrow F$
T+id*id	$E \rightarrow T$
E+id*id	$F \rightarrow id$
E+F*id	$T \rightarrow F$
E+T*id	$F \rightarrow id$
E+T*F	$T \rightarrow T*F$
E+T	$E \rightarrow E+T$
E	

Handles are red and underlined in the right-sentential forms.

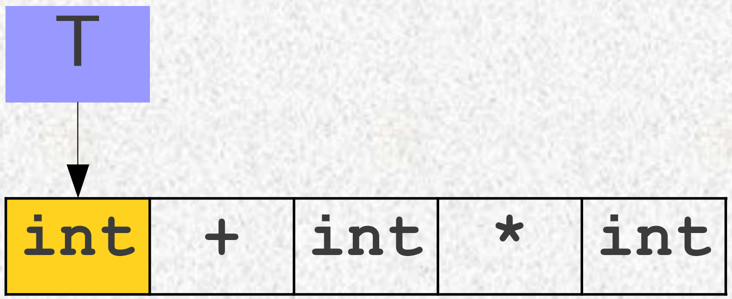
A Detail about Handles

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

int	+	int	*	int

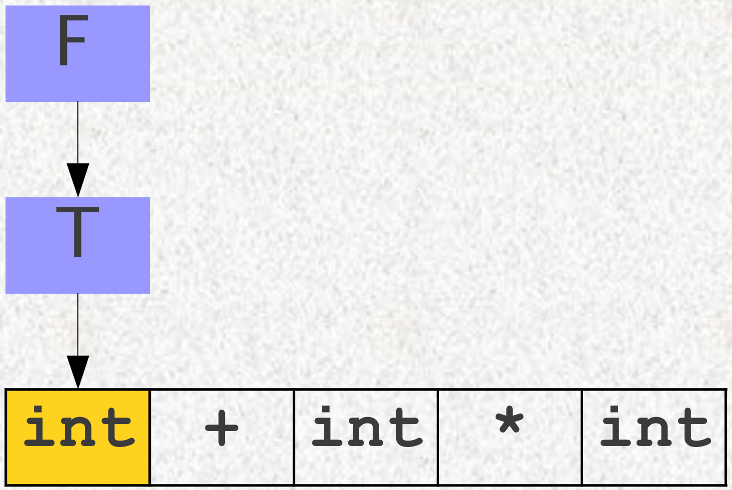
A Detail about Handles

$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$



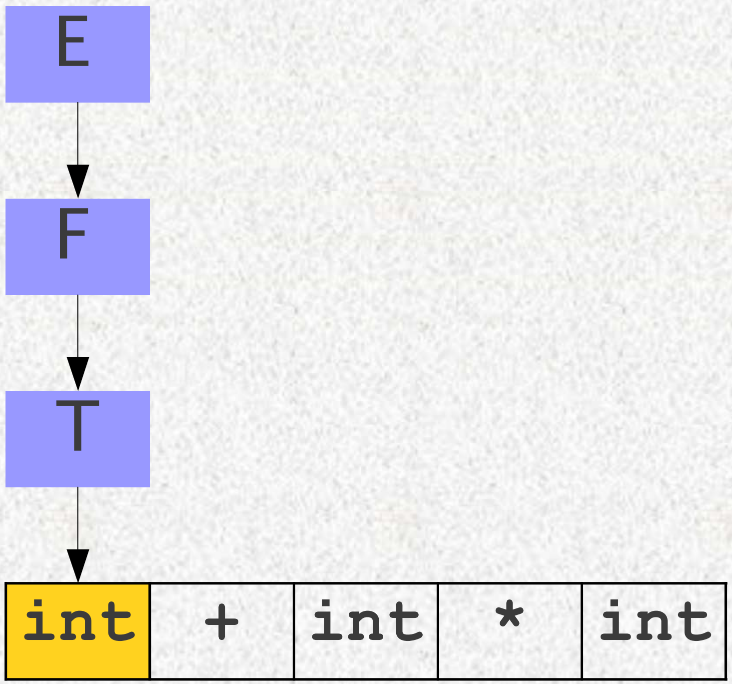
A Detail about Handles

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



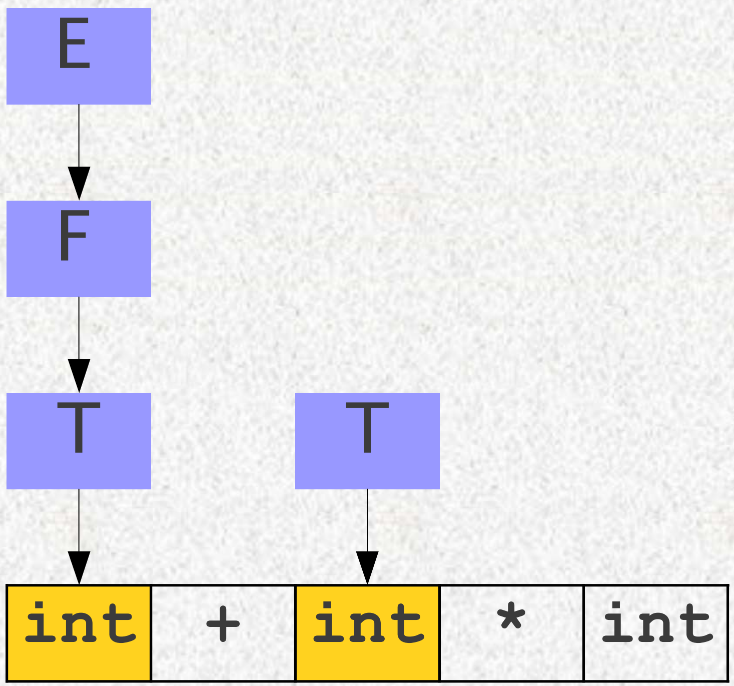
A Detail about Handles

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



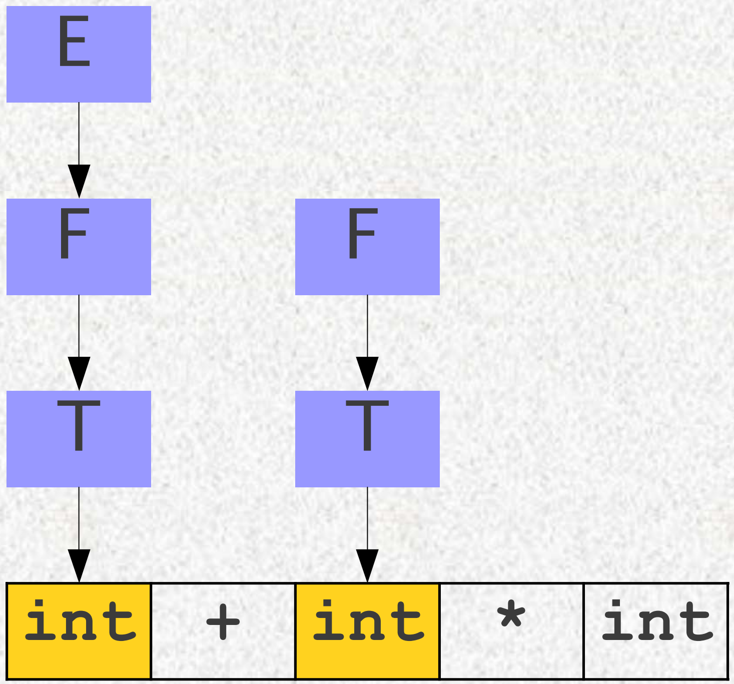
A Detail about Handles

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



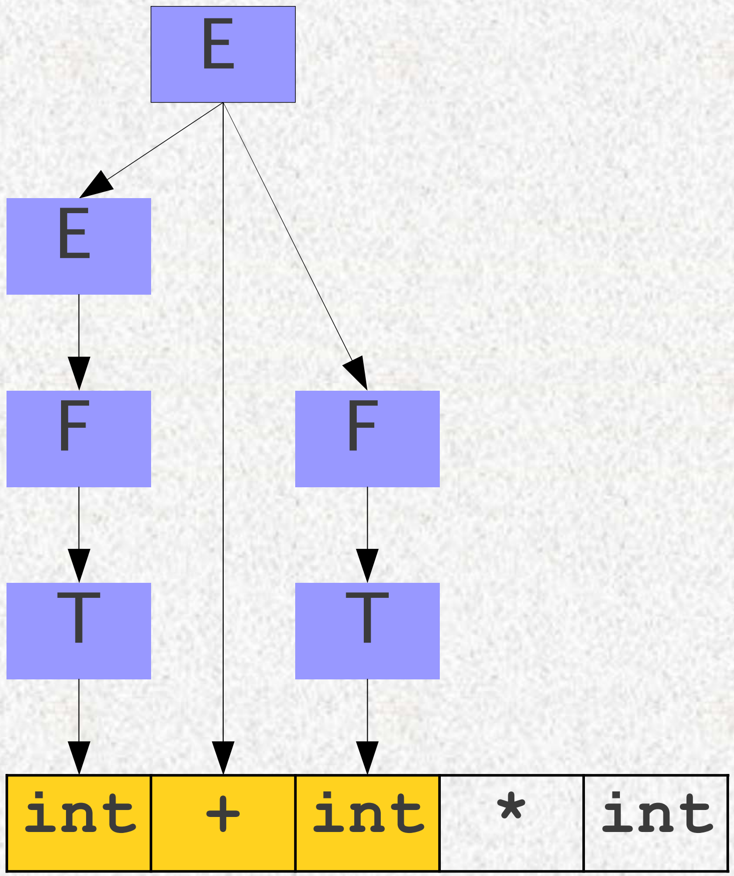
A Detail about Handles

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



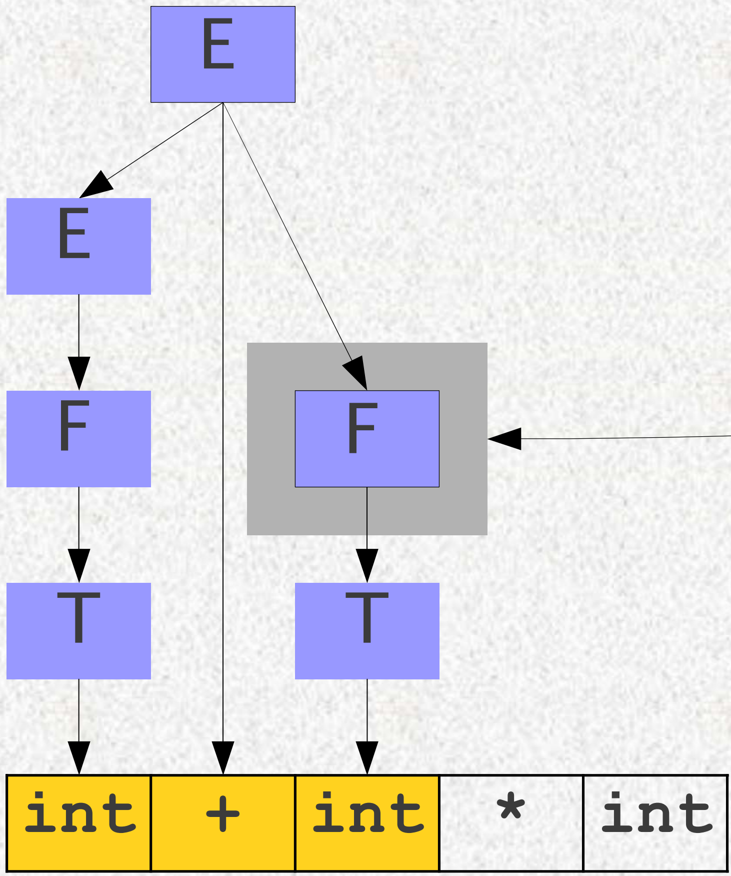
A Detail about Handles

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



A Detail about Handles

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



This reduction wasn't a handle!

Bottom-up Parsing

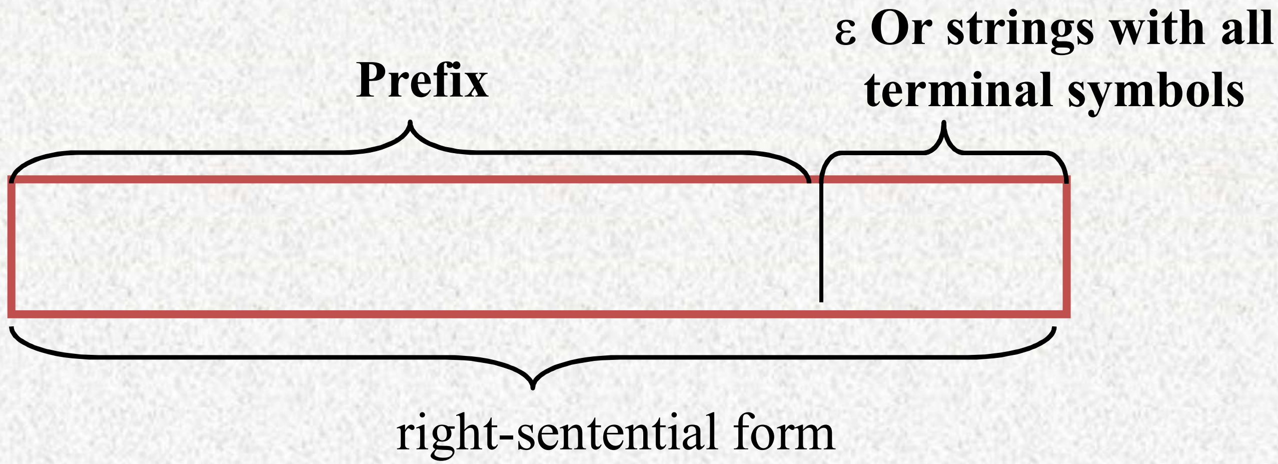
- Traverse rightmost derivation backwards
  - If reduction is done arbitrarily
    - It may not reduce to the starting symbol
    - Need backtracking
  - If we follow the path of rightmost derivation
    - All the reductions are guaranteed to be “correct”
    - Guaranteed to lead to the starting symbol without backtracking
  - That is: If it is always possible to correctly find the handle

Key: Finding Handles

- Where do we look for handles?
  - Where in the string might the handle be?
- How do we search for possible handles?
  - Once we know where to search, how do we identify candidate handles?
- How do we recognize handles?
  - Once we've found a candidate handle, how do we check that it really is the handle?
    - Use a stack to keep track of the viable prefix
    - The prefix of the handle will always be at the top of the stack

Viable prefix

- If a prefix of a right-sentential form:
  - $Z \Rightarrow ABb$: Consider prefixes AB, ABb
  - $Z \Rightarrow^{+} Acb$: Consider prefixes A, Ac, Acb



Viable prefix

- Viable prefixes are:
  - Prefixes that do not contain simple phrases; or
  - Prefixes containing one simple phrase that are at the end of this prefix --- that is, this simple phrase is the handle.
- A viable prefix does not contain any symbol after a handle.

Viable prefix

- **Eg.**

(1) $Z \rightarrow ABb$
(2) $A \rightarrow a$
(3) $A \rightarrow b$
(4) $B \rightarrow d$
(5) $B \rightarrow c$
(6) $B \rightarrow bB$

$Z \Rightarrow ABb$
Consider prefixes: AB, ABb
Viable prefixes are: AB(no simple phrase)
ABb (one simple phrase, which is at the end of the prefix)

$Z \Rightarrow^{+} abcb$

Consider prefixes: a, ab, abc, abcb

Viable prefix: a (contain one simple phrase)

ab, abc, abcd are not viable prefix

Viable prefix

- Two types of viable prefix
  - Nonreducible (for shift operation)： no simple phrase, need to shift more symbols to form the first leftmost simple phrase (i.e. handle)
  - Reducible (for reduction operation): contain one simple phrase, at the end of the

(1) $Z \rightarrow ABb$
(2) $A \rightarrow a$
(3) $A \rightarrow b$
(4) $B \rightarrow d$
(5) $B \rightarrow c$
(6) $B \rightarrow bB$

$Z \Rightarrow ABb$ Viable prefixes:
AB(no simple phrase) --- nonreducible
ABb (contain a simple phrase) --- reducible

Bottom-up Parsing

- Shift-reduce operations in bottom-up parsing
  - Shift the input into the stack
    - Wait for the current handle to complete or to appear
    - Or wait for a handle that may complete later
  - Reduce
    - Once the handle is completely in the stack, then reduce
  - The operations are determined by the parsing table

Build the Automata

- LR(0) Item of a grammar G
  - Is a production of G with a distinguished position
  - Position is used to indicate how much of the handle has already been seen (in the stack)
    - For production $S \rightarrow a\ B\ S$, items for it include

      $S \rightarrow \bullet\ a\ B\ S$

      $S \rightarrow a \bullet B\ S$

      $S \rightarrow a\ B \bullet S$

      $S \rightarrow a\ B\ S \bullet$

      - Left of $\bullet$ are the parts of the handle that has already been seen
      - When $\bullet$ reaches the end of the handle $\Rightarrow$ reduction
    - For production $S \rightarrow \varepsilon$, the single item is

      $S \rightarrow \bullet$

Building the Automata

- Closure function Closure(I)
  - I is a set of items for a grammar G
  - Every item in I is in Closure(I),

    if $A \rightarrow \alpha \bullet B\ \beta$ is in Closure(I) and $B \rightarrow \gamma$ is a production in G, then add $B \rightarrow \bullet\ \gamma$ to Closure(I)
    - If it is not already there
    - Meaning
      - When α is in the stack and B is expected next
      - One of the B-production rules may be used to reduce the input to B
        - » May not be one-step reduction though
  - Apply the rule until no more new items can be added

Building the Automata

– CLOSURE(IS)Example

$V_T = \{a, b, c\}$
$V_N = \{S, A, B\}$
$S = S$
P:
{ $S \to aAc$
$A \to ABb$
$A \to Ba$
$B \to b$
}

$IS = \{S \to \bullet aAc\}$
$CLOSURE(IS) = \{S \to \bullet aAc\}$

$IS = \{S \to a \bullet Ac\}$
$CLOSURE(IS)$
$= \{S \to a \bullet Ac,$
$A \to \bullet ABb, A \to \bullet Ba,$
$B \to \bullet b\}$

Building the Automata

- Goto function Goto(I,X)
  - X is a grammar symbol
  - If $A \rightarrow \alpha \bullet X \beta$ is in I then $A \rightarrow \alpha X \bullet \beta$ is in Goto(I, X)
    - Let J denote the set constructed by this step
  - All items in Closure(J) are in Goto(I, X)
  - Meaning
    - If I is the set of valid items for some viable prefix γ
    - Then goto(I, X) is the set of valid items for the viable prefix γX

Building the Automata

- Augmented grammar
  - G is the grammar and S is the staring symbol
  - Construct G' by adding production $S' \rightarrow S$ into G
    - S' is the new starting symbol
    - E.g.: G: $S \rightarrow \alpha \mid \beta$ $\Rightarrow$ G': $S' \rightarrow S$, $S \rightarrow \alpha \mid \beta$
  - Meaning
    - The starting symbol may have several production rules and may be used in other non-terminal's production rules
    - Add $S' \rightarrow S$ to force the starting symbol to have a single production
    - When $S' \rightarrow S \bullet$ is seen, it is clear that parsing is done

Building the Automata

- Complete process: Given a grammar G
  - Step 1: augment G
  - Step 2: initial state
    - Construct the valid item set “I” of State 0 (the initial state)
    - Add $S' \rightarrow \bullet\, S$ into I
      - All expansions have to start from here
    - Compute Closure(I) as the complete valid item set of state 0
      - All possible expansions S can lead into
  - Step 3:
    - From state I, for all grammar symbol X

      Construct J = Goto(I, X)

      Compute Closure(J)
    - Create the new state with the corresponding Goto transition
      - Only if the valid item set is non-empty and does not exist yet
  - Repeat Step 3 till no new states can be derived

Building the Automata -- Example

- Grammar G:

  $S \rightarrow E$

  $E \rightarrow E + T \mid T$

  $T \rightarrow id \mid (E)$

  – Step 1: Augment G

    $S' \rightarrow S \quad S \rightarrow E \quad E \rightarrow E + T \mid T \quad T \rightarrow id \mid (E)$

  – Step 2:

    - Construct Closure(I_0) for State 0
    - First add into I_0: $S' \rightarrow \bullet S$
    - Compute Closure(I_0)

      $S' \rightarrow \bullet S$

      $S \rightarrow \bullet E$

      $E \rightarrow \bullet E + T$

      $E \rightarrow \bullet T$

      $T \rightarrow \bullet id$

      $T \rightarrow \bullet (E)$

Building the Automata -- Example

I_0:

$S' \rightarrow \bullet S \quad S \rightarrow \bullet E$

$E \rightarrow \bullet E + T \quad E \rightarrow \bullet T$

$T \rightarrow \bullet id \quad T \rightarrow \bullet (E)$

- Step 3
  - I_1
    - Add into I_1: Goto(I_0, S) = $S' \rightarrow S \bullet$
    - No new items to be added to Closure (I_1)
  - I_2
    - Add into I_2: Goto(I_0, E) = $S \rightarrow E \bullet \qquad E \rightarrow E \bullet + T$
    - No new items to be added to Closure (I_2)
  - I_3
    - Add into I_3: Goto(I_0, T) = $E \rightarrow T \bullet$
    - No new items to be added to Closure (I_3)
  - I_4
    - Add into I_4: Goto(I_0, id) = $T \rightarrow id \bullet$
    - No new items to be added to Closure (I_4)

Building the Automata -- Example

I_0:

$S' \to \bullet S$ $\quad S \to \bullet E$

$E \to \bullet E + T$ $\quad E \to \bullet T$

$T \to \bullet id$ $\quad T \to \bullet (E)$

- Step 3
  - I_5
    - Add into I_5: Goto(I_0, "(") = $T \to (\bullet E)$
    - Closure(I_5)

      $E \to \bullet E + T$ $\quad E \to \bullet T$

      $T \to \bullet id$ $\quad T \to \bullet (E)$
  - No more moves from I_0
  - No possible moves from I_1
  - I_6
    - Add into I_6: Goto(I_2, +) = $E \to E + \bullet T$
    - Closure(I_5)

      $T \to \bullet id$ $\quad T \to \bullet (E)$
  - No possible moves from I_3 and I_4

Building the Automata -- Example

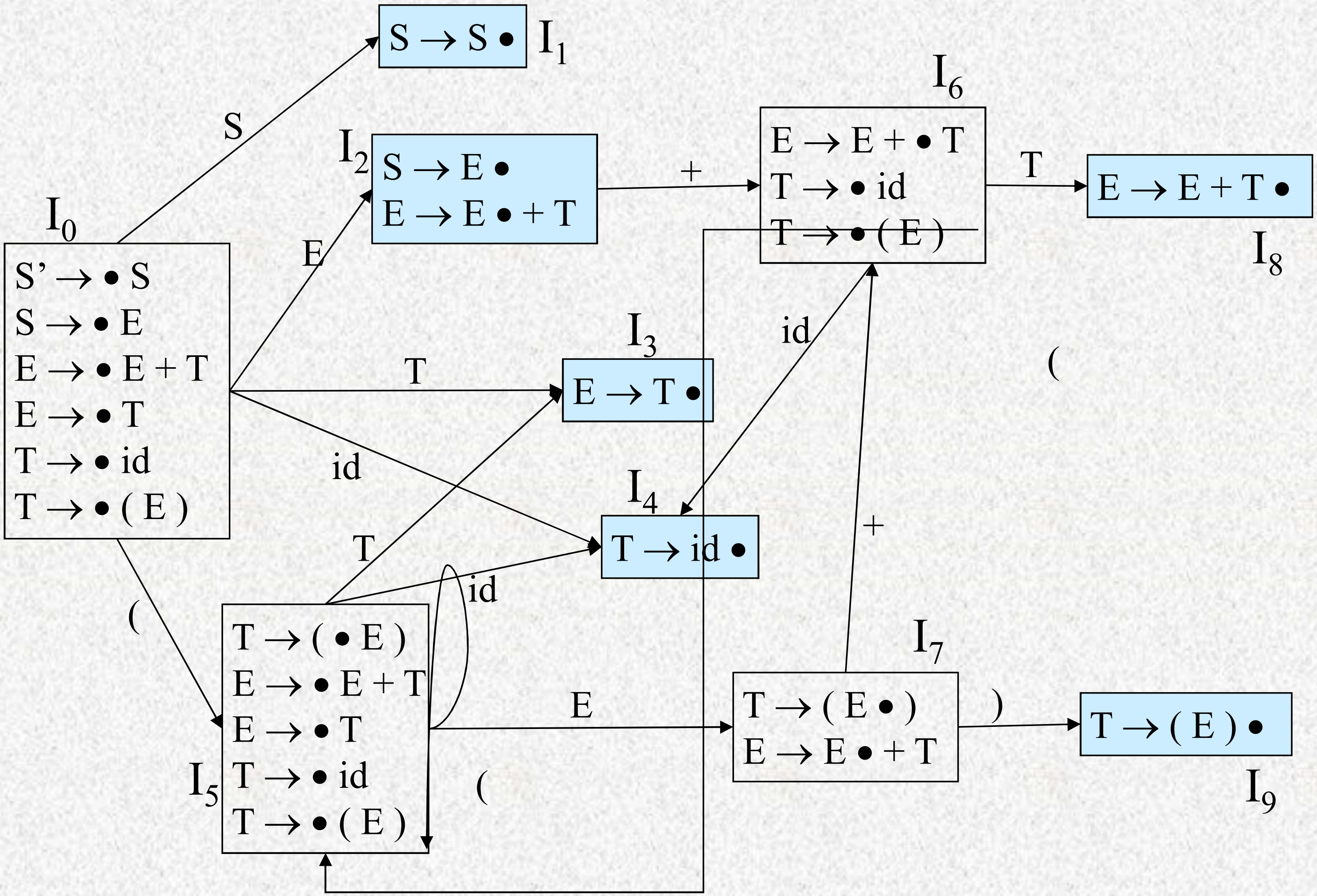
- Step 3
  - I_7
    - Add into I_7: Goto(I_5, E) =
      $T \rightarrow (E \bullet)$ $E \rightarrow E \bullet + T$
    - No new items to be added to Closure (I_7)
  - Goto(I_5, T) = I_3
  - Goto(I_5, id) = I_4
  - Goto(I_5, "(") = I_5
  - No more moves from I_5
  - I_8
    - Add into I_8: Goto(I_6, T) = $E \rightarrow E + T \bullet$
    - No new items to be added to Closure (I_8)
  - Goto(I_6, id) = I_4
  - Goto(I_6, "(") = I_5

Building the Automata -- Example

- Step 3
  - I_9
    - Add into I_9: Goto(I_7, ")") =

      $T \rightarrow (E) \bullet$
    - No new items to be added to Closure (I_9)
  - Goto(I_7, +) = I_6
  - No possible moves from I_8 and I_9

Building the Automata -- Example

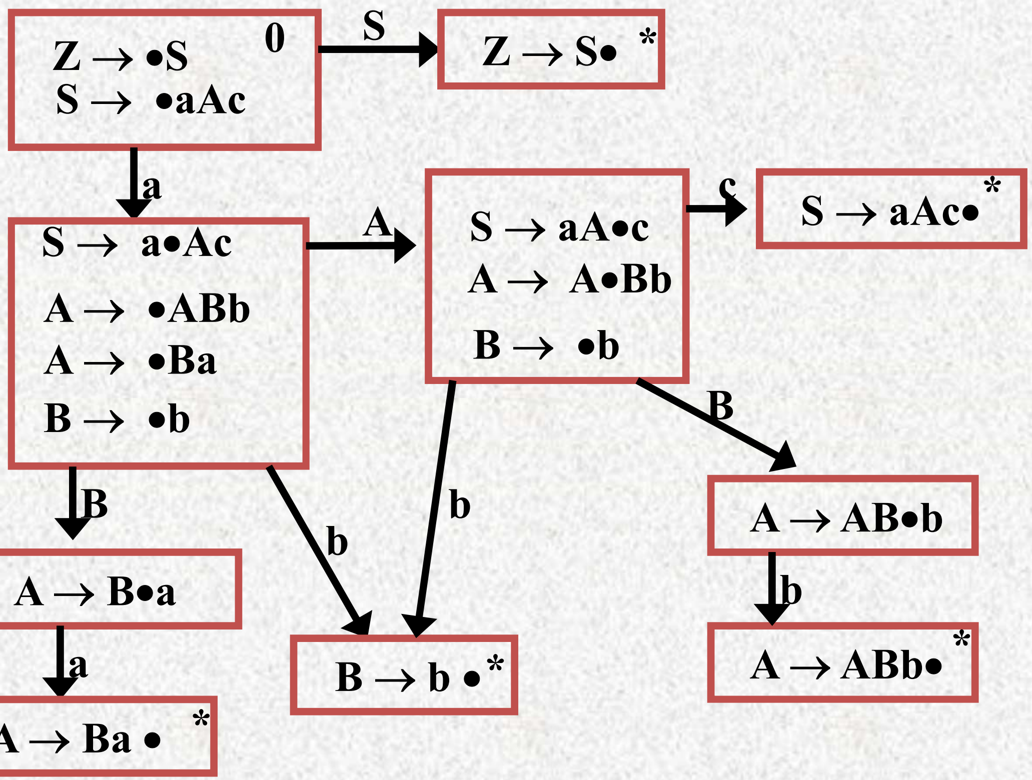


Reducible or Nonreducible

- LR(0) parser
  - Shift item: $A \rightarrow \alpha \bullet a\beta$, $a \in V_T$
  - Reducible item: $A \rightarrow \alpha \bullet$,
  - Accepted item: $Z \rightarrow S \bullet$, ($Z \rightarrow S$ is from the augmented grammar)
  - Shift status: include shift item
  - Reducible state: include reducible item
  - Conflict state:
    - A state contains different reducible items: reduce-reduce conflict;
    - A state contains both shift states and reducible items: shift-reduce conflict

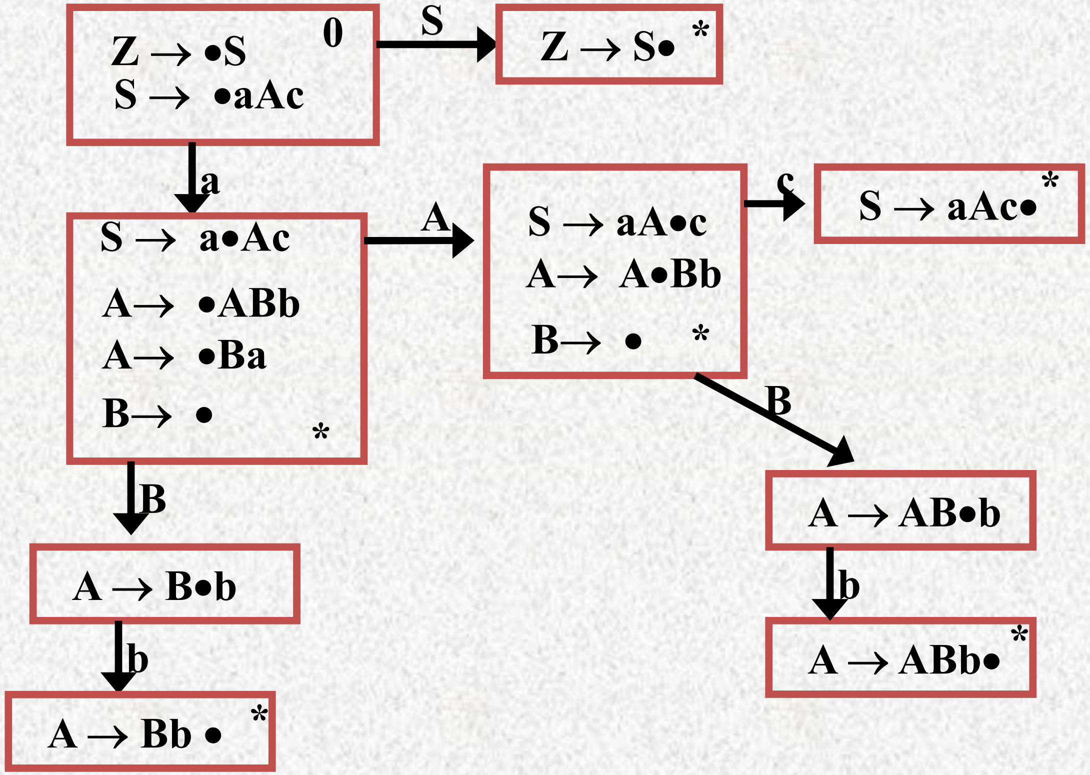
Building the Automata – Example 2

$V_T = \{a, b, c\}$
$V_N = \{S, A, B\}$
$S = S$
P:
{ $S \rightarrow aAc$
$A \rightarrow ABb$
$A \rightarrow Ba$
$B \rightarrow b$
}



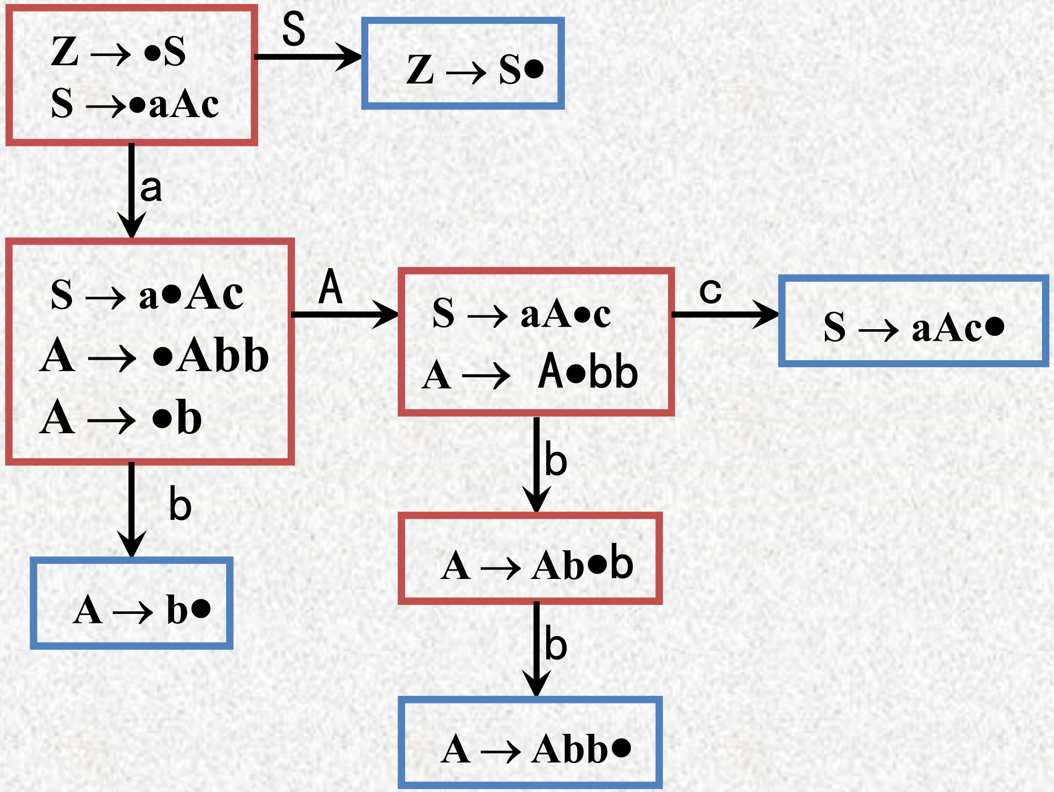
Building the Automata – Example 3

$V_T = \{a, b, c\}$
$V_N = \{S, A, B\}$
$S = S$
P:
{ $S \rightarrow aAc$
$A \rightarrow ABb$
$A \rightarrow Ba$
$B \rightarrow \varepsilon$
}



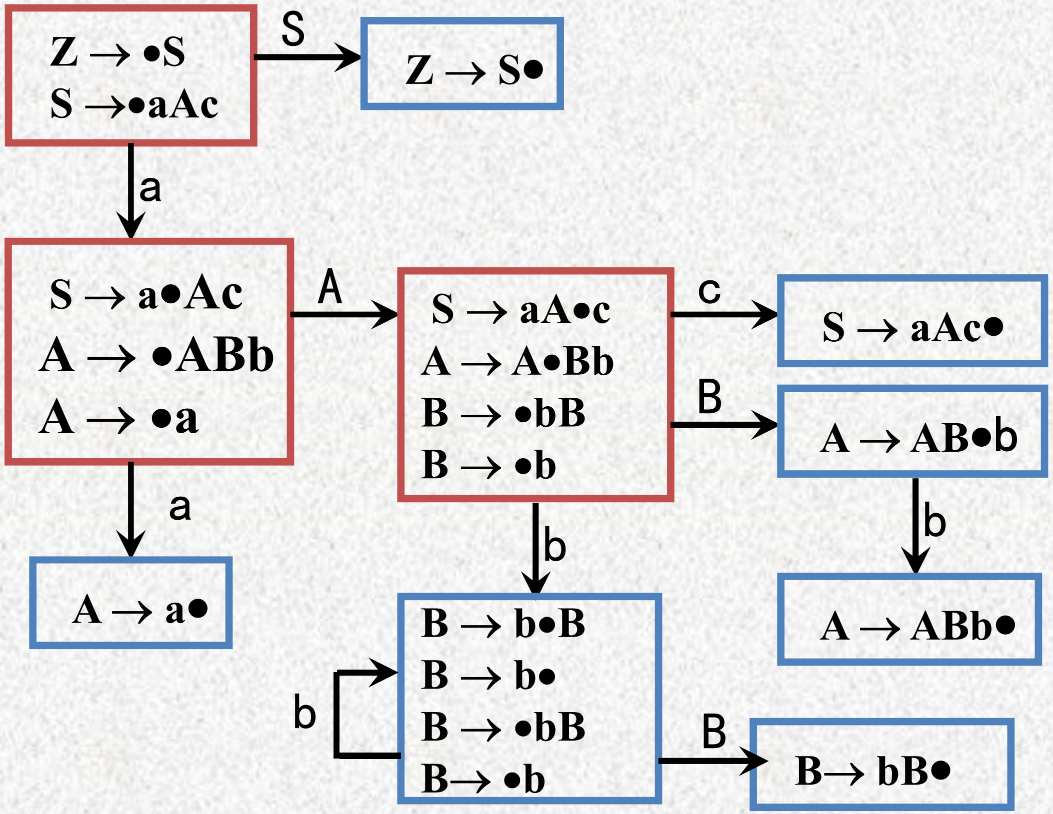
Building the Automata – Example 4

$V_T = \{a, b, c\}$
$V_N = \{S, A\}$
$S = S$
P:
$\{ \; S \rightarrow aAc$
$\quad A \rightarrow Abb$
$\quad A \rightarrow b$
$\}$



Building the Automata – Example 5

$V_T = \{a, b, c\}$
$V_N = \{S, A, B\}$
$S = S$
P:
$\{\ S \rightarrow aAc$
$A \rightarrow ABb$
$A \rightarrow a$
$B \rightarrow bB$
$B \rightarrow b$
$\}$



LR(0) algorithm

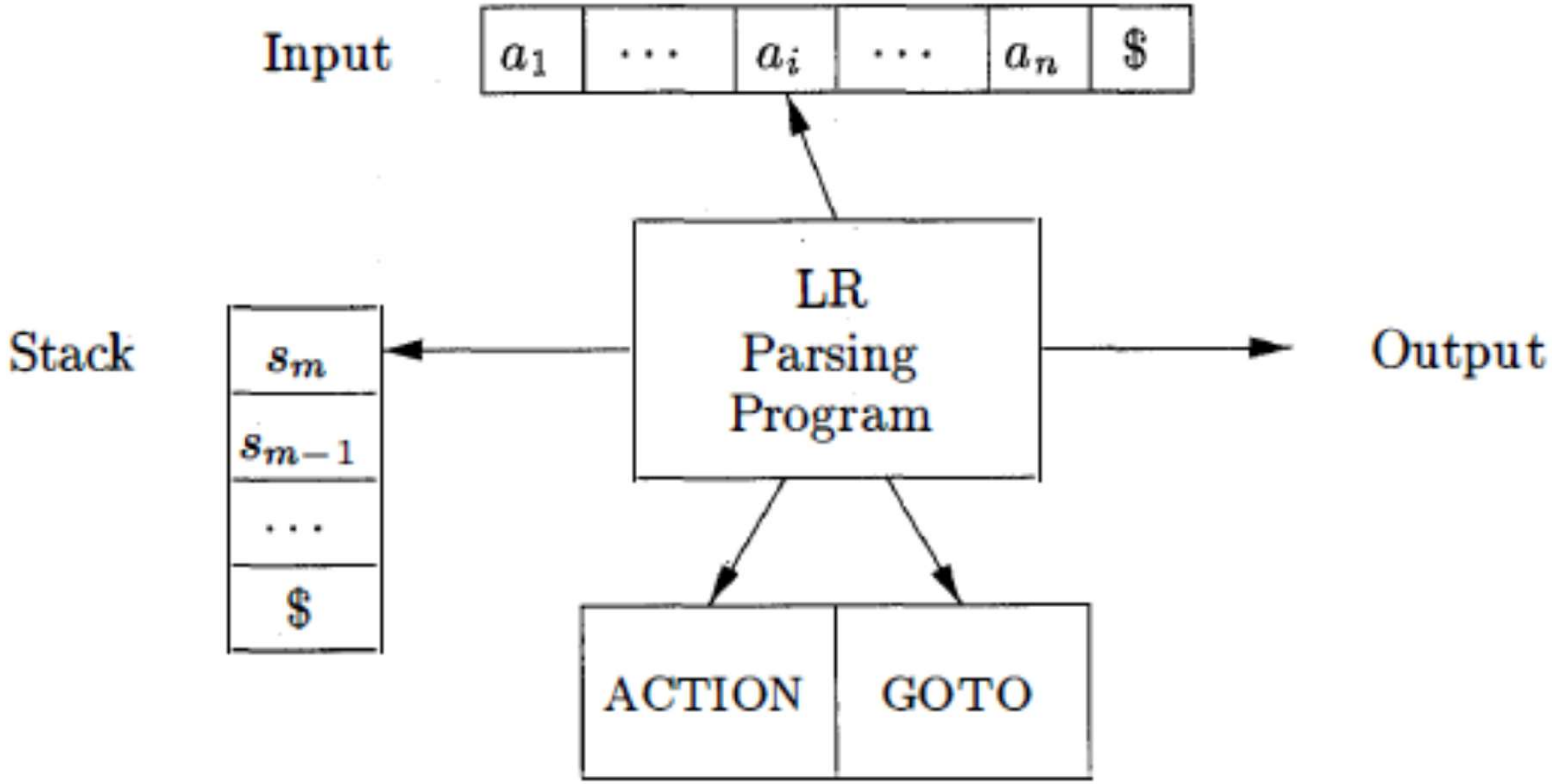


Figure 4.35: Model of an LR parser

Building the Action Table

- If state I_i has item $A \to \alpha \bullet a \beta$, and
  - Goto(I_i, a) = I_j
  - Next symbol in the input is a
- Then Action[I_i, a] = I_j
  - Meaning: Shift "a" to the stack and move to state I_j
    - Need to wait for the handle to appear or to complete
- If State I_i has item $A \to \alpha \bullet$
- Then Action[S, b] = reduce using $A \to \alpha$
  - For all b in Follow(A)
  - Meaning: The entire handle α is in the stack, need to reduce
  - Need to wait to see Follow(A) to know that the handle is ready
    - E.g. $S \to E \bullet$ $\quad E \to E \bullet + T$
    - Current input can be either Follow(S) or +

Building the Action Table

- If state has $S' \rightarrow S_0 \bullet$
- Then Action[S, $] = accept

- Current state
  - The action to be taken depends on the current state
    - In LL, it depends on the current non-terminal on the top of the stack
    - In LR, non-terminal is not known till reduction is done
  - Who is keeping track of current state?
  - The stack
    - Need to push the state also into the stack
    - The stack includes the viable prefix and the corresponding state for each symbol in the viable prefix

Building the Action Table

Action Table

$action(S_i,a) = S_j$, if there is an edge from S_i to S_j labeled as a

$action(S_i,c) = R_p$, if S_i is a reducible state, $c \in Vt \cup \{\#\}$

$action(S_i,\#)$ = accept, if S_i is acceptance state

$action(S_i,a)$ = error, otherwise

Terminal symbols / **States**	a_1	…	#
S_1			
…			
S_n			

Building the Goto Table

- If Goto(I_i, A) = I_j
- Then Goto[i, A] = j
- Meaning
  - When a reduction $X \rightarrow \alpha$ taken place
  - The non-terminal X is added to the stack replacing α
  - What should the state be after adding X
  - This information is kept in Goto table

Building the Goto Table

GOTO Table

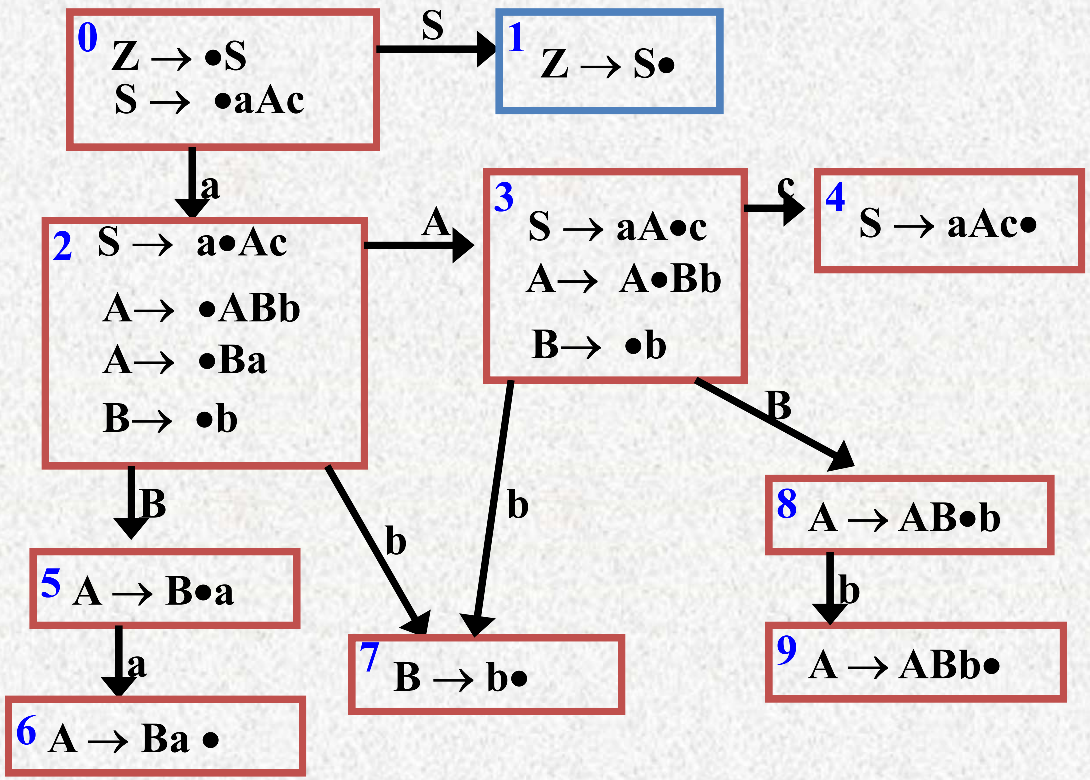
goto (S_i, A) = S_j, if there is an edge from S_i to S_j labeled as A
goto (S_i, A) = error, if there is no edge from S_i to S_j labeled as A

State \ non-terminal	A_1	…	#
S_1			
…			
S_n			

LR(0) Parsing algorithm

- Example

$V_T = \{a, b, c\}$
$V_N = \{S, A, B\}$
$S = S$
P:
{ (1) $S \rightarrow aAc$
(2)$A \rightarrow ABb$
(3)$A \rightarrow Ba$
(4)$B \rightarrow b$
}



LR(0) Parsing algorithm

action						goto		
	a	b	c	#		S	A	B
0	S2					1		
1				accept				
2		S7					3	5
3		S7	S4					8
4	R1	R1	R1	R1				
5	S6							
6	R3	R3	R3	R3				
7	R4	R4	R4	R4				
8		S9						
9	R2	R2	R2	R2				

LR(0) Parsing algorithm

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36. □

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s,a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s,a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s,a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

LR(0) Parsing algorithm

a	b	a	c

P: (0) Z → S; (1) S→ aAc; (2)A → ABb;
(3)A → Ba; (4)B → b

	action				goto		
	a	b	c	#	S	A	B
0	S2				1		
1				accept			
2		S7				3	5
3		S7	S4				8
4	R1	R1	R1	R1			
5	S6						
6	R3	R3	R3	R3			
7	R4	R4	R4	R4			
8		S9					
9	R2	R2	R2	R2			

Stack	Input	Actions
0	abac#	S2
02	bac#	S7
027	ac#	R4,Goto(2, B)=5
025	ac#	S6
0256	c#	R3,Goto(2, A)=3
023	c#	S4
0234	#	R1, Goto(0, S)=1
01	#	Accept

Homework

Page 257, 4.6.1

Limit of LR(0)

LR Conflicts

A **shift/reduce conflict** is an error where a shift/reduce parser cannot tell whether to shift a token or perform a reduction.

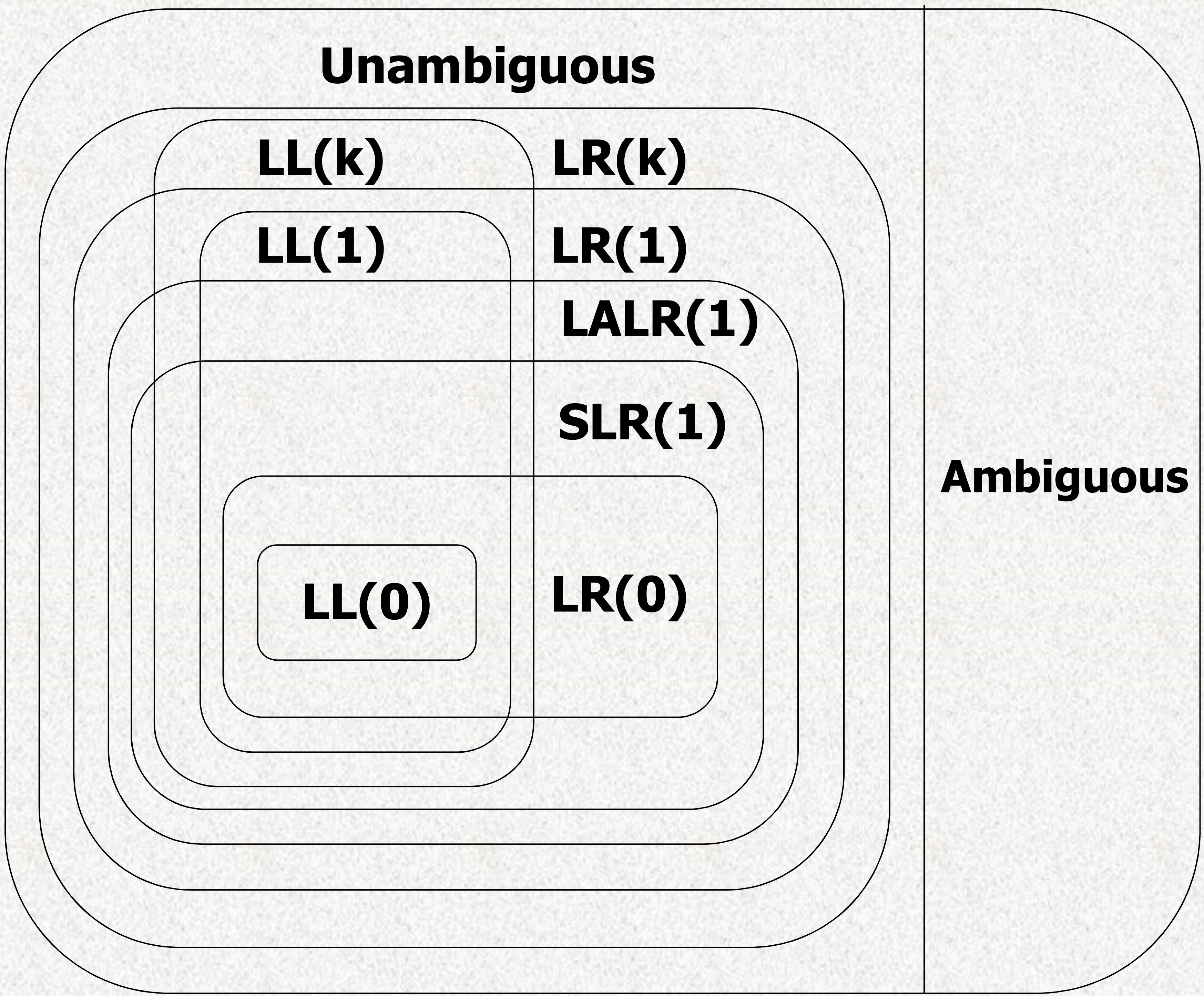
A **reduce/reduce conflict** is an error where a shift/reduce parser cannot tell which of many reductions to perform.

A grammar whose handle-finding automaton contains a shift/reduce conflict or a reduce/reduce conflict is not LR(0).

LR Family

- **LR Family**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser
  - LALR – intermediate LR parser (look-head LR parser)

  - SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.


Unambiguous
LL(k)
LR(k)
LL(1)
LR(1)
LALR(1)
SLR(1)
Ambiguous
LL(0)
LR(0)