Compiler Technology of Programming Languages

Chapter 4

Formal Grammars and Parsing

Prof. Farn Wang

Formal Grammars and Parsing Contents:

- Context-Free Grammars:
  Concepts and Notation
- Properties of CFG
- Parsers and Recognizers
- Grammar Analysis Algorithms

Introduction

- Why using CFG
  - CFG gives a precise syntactic specification of a programming language.
  - Enabling automatic translator generator
  - Language extension becomes easier
- The role of the parser
  - Taking tokens from scanner, parsing, reporting syntax errors
  - Not just parsing, in a *syntax-directed* translator, the parser also conducts type checking, semantic analysis and IR generation.

Example of CFG

- A C– program is made out of functions,
- a function out of declarations and blocks,
- a block out of statements,
- a statement out of expressions, … etc

<program> → <global_decl_list>
<global_decl_list> → <global_decl_list><global_decl> | λ
<global_decl> → <decl_list> <function_decl>
<function_decl> → <type> id (<param_list>) { <block> }
<block> → <decl_list> <statement_list> | λ
<decl_list> → <decl_list> <decl> | <decl> | λ
<decl> → <type_decl> | <var_decl>
<type> → void | int | float
<statement_list> → ….
<statement> → { <block> }

pay attention to
Repetitions and Recursions !!

Example of Statements

- Selection statements
  if (<expression>) <statement>
  if (<expression>) <statement> **else** <statement>

- Iteration statements
  while (<expression>) <statement>
  for (<expression>; <expression>; <expression>) <statement>

- Other C statements
  do <statement> **while** (<expression>) ;
  switch (<expression>) <statement>
  labeled statements, jump statement, … etc.

CFG

- A context-free grammar G = (Vt, Vn, S, P)
  - Vt: A finite terminal vocabulary
    - The token set produced by scanner
  - Vn: A finite set of nonterminal vocabulary
    - Intermediate symbols
  - S: A start symbol, S $\in$ Vn that starts all derivations
    - Also called goal symbol
  - P: a finite set of productions (rewriting rules) of the form A$\rightarrow$ X1X2 … Xm
    - A$\in$Vn, Xi $\in$ Vn$\cup$Vt, 1$\leq$i $\leq$m
    - A $\rightarrow\lambda$ is a valid production

How is CFG different from RE/DFA ?

CFG

- Other notation
  - Vocabulary V of G,
    - V= Vn∪Vt
  - L(G), the set of string s derivable from S
    - Context-free language of grammar G
  - Notational conventions
    - a,b,c, … denote symbols in Vt
    - A,B,C, … denote symbols in Vn
      (*or some names enclosed in* <>)
    - *U,V,W*, … denote symbols in V
    - α,β,γ,… denote *strings* in V*
    - u,v,w, … denote *strings* in Vt*

CFG

- Derivation
  - One step derivation
    - If $A \rightarrow \gamma$, then $\alpha A \beta \Rightarrow \alpha \gamma \beta$
  - One or more steps derivation $\Rightarrow^+$
  - Zero or more steps derivation $\Rightarrow^*$
- If $S \Rightarrow^* \beta$, then β is said to be sentential form of the CFG
  - SF(G) is the set of sentential forms of grammar G
- The language L(G) generated by G is the set of terminal strings *w* such that $S \Rightarrow^+ w$. The string *w* is called a sentence of G.

$$L(G) = \{w \in V_t^* | S \Rightarrow^+ w\}$$

CFG

Left-most derivation

denoted by $\Rightarrow_{lm}$, $\Rightarrow^{+}_{lm}$, $\Rightarrow^{*}_{lm}$

example of leftmost derivation of b(c+c)

G_0

$E \rightarrow Prefix(E)$
$E \rightarrow c\ Tail$
$Prefix \rightarrow b$
$Prefix \rightarrow \lambda$
$Tail \rightarrow +E$
$Tail \rightarrow \lambda$

$E \Rightarrow_{lm} Prefix(E)$
$\Rightarrow_{lm} b(E)$
$\Rightarrow_{lm} b(c\ Tail)$
$\Rightarrow_{lm} b(c+E)$
$\Rightarrow_{lm} b(c+c\ Tail)$
$\Rightarrow_{lm} b(c+c)$

- Top-down parsing is to come up with a *leftmost* derivation.

CFG

Right-most derivation

denoted by $\Rightarrow_{rm}$, $\Rightarrow^{+}_{rm}$, $\Rightarrow^{*}_{rm}$

example of rightmost derivation of b(c+c)

G_0

$E \rightarrow Prefix(E)$
$E \rightarrow c\ Tail$
$Prefix \rightarrow b$
$Prefix \rightarrow \lambda$
$Tail \rightarrow +E$
$Tail \rightarrow \lambda$

$E \Rightarrow_{rm} Prefix(E)$
$\Rightarrow_{rm} Prefix(c\ Tail)$
$\Rightarrow_{rm} Prefix(c+E)$
$\Rightarrow_{rm} Prefix(c+c\ Tail)$
$\Rightarrow_{rm} Prefix(c+c)$
$\Rightarrow_{rm} b(c+c)$

- Bottom-up parsing is to *discover* a *rightmost* derivation, it reverses the derivation.

Parse Tree and Derivation

A Parse tree can be viewed as a graphical representation for a derivation that ignore replacement order.

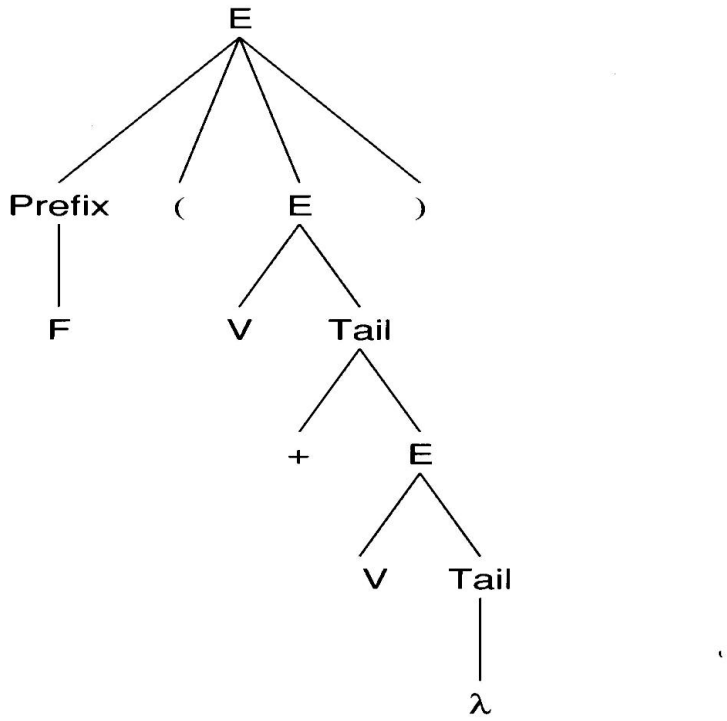


Figure 4.1 The Parse Tree Corresponding to F(V+V)

Phrases and Handles

- ***phrase***

  a sequence of symbols descended from a single nonterminal in the parse tree

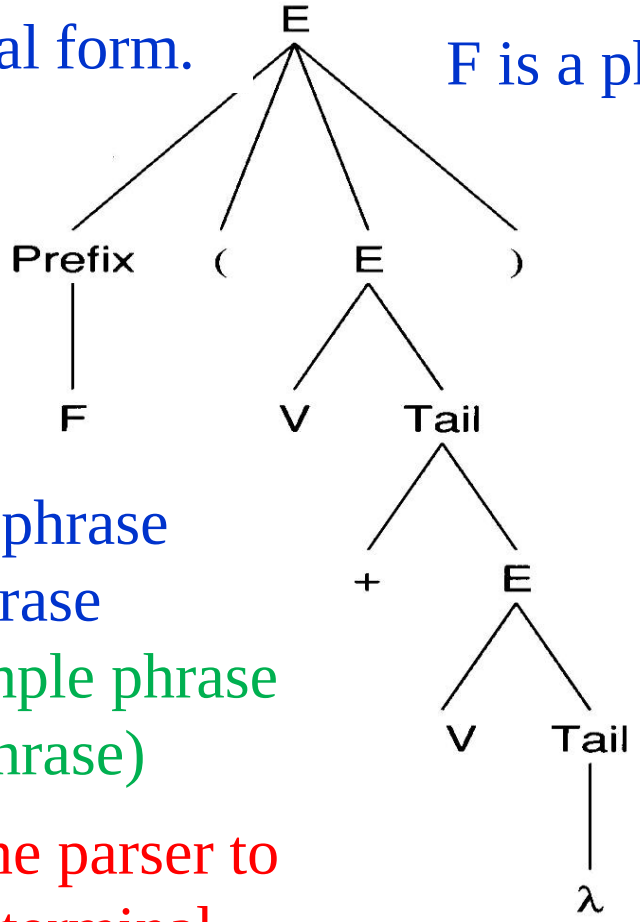
  – A ***simple*** or ***prime*** phrase is a phrase that contains no smaller phrase.

- The ***handle*** of a sentential form is the left-most simple phrase

Phrases and Handles

F (E) is a sentential form.

F is a phrase, also a handle



V Tail is a simple phrase

+ E is a simple phrase

V+E is NOT a simple phrase

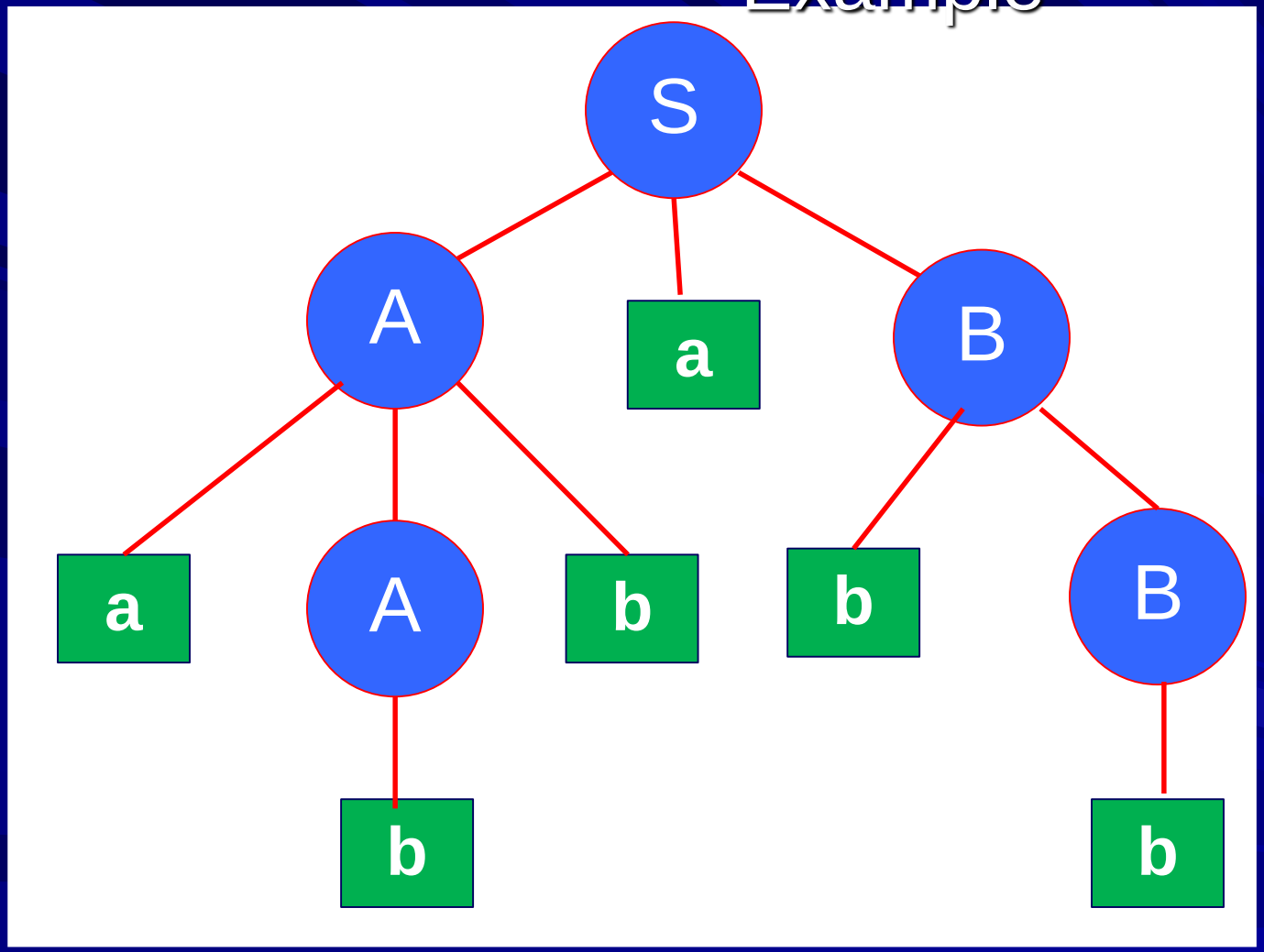
(+E is a smaller phrase)

A handle allows the parser to reduce it to a non-terminal

Figure 4.1 The Parse Tree Corresponding to F(V+V)

Phrases and Handles

Example



abbabb

a sentence

aAbabB

a sentential form

aAb

a handle

bB

a simple phrase

abb

a phrase of **A**

G_0

- E→Prefix(E)
- E→c Tail
- Prefix→b
- Prefix→λ
- Tail→+E
- Tail→ λ

$E \Rightarrow_{rm}$ Prefix(E)
$\Rightarrow_{rm}$ Prefix(c Tail)
$\Rightarrow_{rm}$ Prefix(c+E)
$\Rightarrow_{rm}$ Prefix(c+c Tail)
$\Rightarrow_{rm}$ Prefix(c+c)
$\Rightarrow_{rm}$ b(c+c)

$E \Rightarrow$ Prefix(E) $\Rightarrow$ Prefix(c Tail) $\Rightarrow$Prefix(c+E) $\Rightarrow$ Prefix(c+c Tail)
$\Rightarrow$ Prefix(c+c) $\Rightarrow$ b(c+c)

rightmost derivation

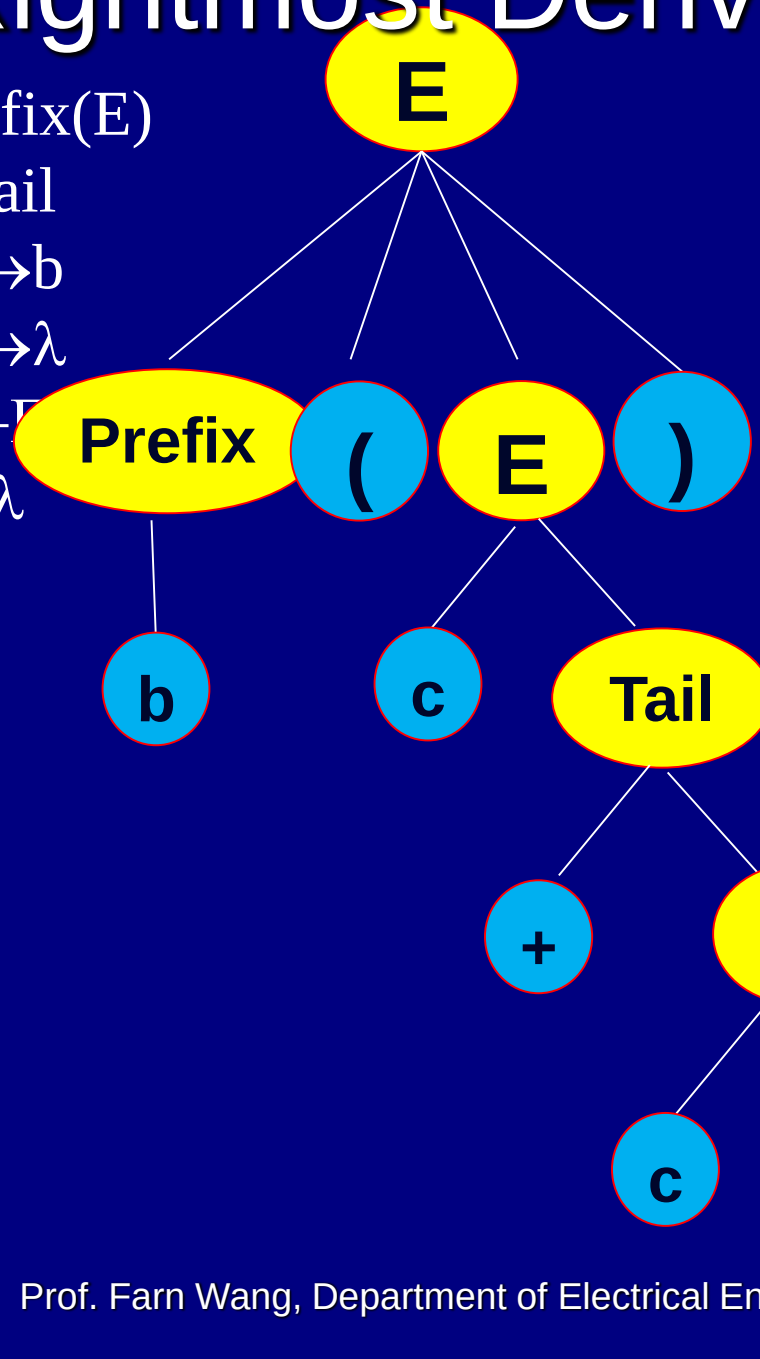
$E \Leftarrow$ Prefix(E) $\Leftarrow$ Prefix(c Tail) $\Leftarrow$ Prefix(c+E) $\Leftarrow$ Prefix(c+c Tail)
$\Leftarrow$ Prefix(c+c) $\Leftarrow$ b(c+c)

bottom-up parsing

Rightmost Derivation

G_0

- E→Prefix(E)
- E→c Tail
- Prefix→b
- Prefix→λ
- Tail→+E
- Tail→ λ

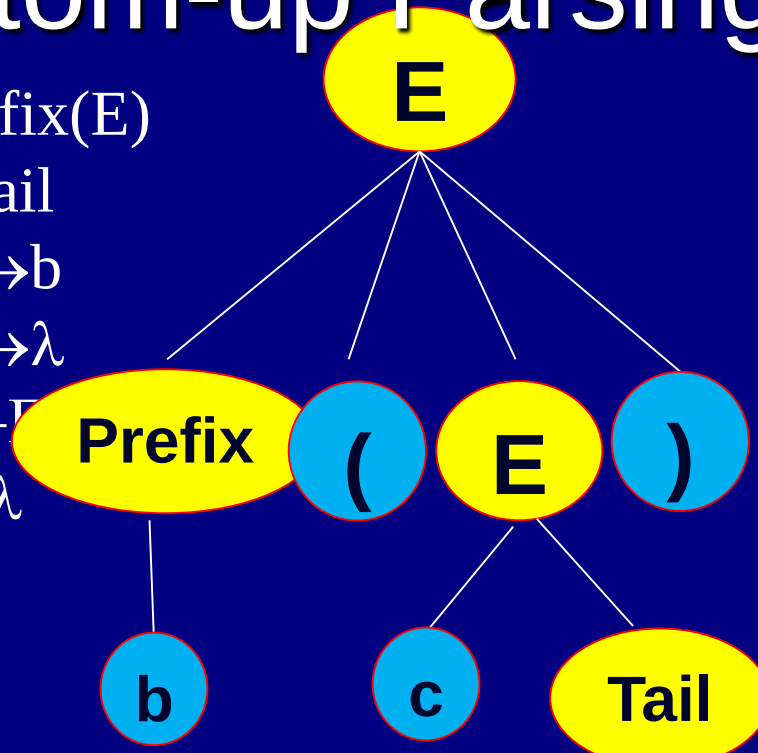


$E \Rightarrow_{rm}$ Prefix(E)
$\Rightarrow_{rm}$ Prefix(c Tail)
$\Rightarrow_{rm}$ Prefix(c+E)
$\Rightarrow_{rm}$ Prefix(c+c Tail)
$\Rightarrow_{rm}$ Prefix(c+c)
$\Rightarrow_{rm}$ b(c+c)

Bottom-up Parsing (1/13)

G_0

E→Prefix(E)
E→c Tail
Prefix→b
Prefix→λ
Tail→+E
Tail→ λ



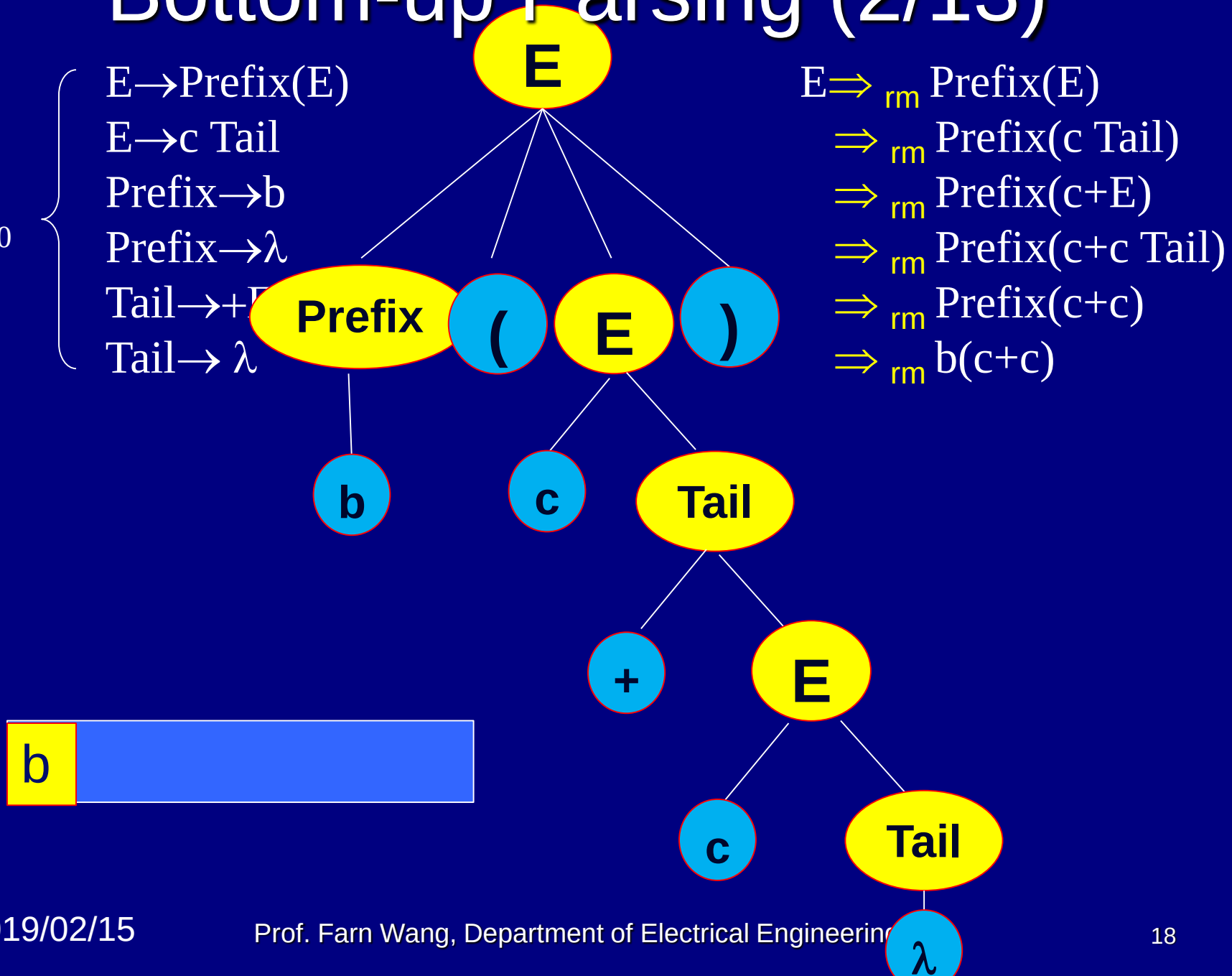
$E \Rightarrow_{rm}$ Prefix(E)
$\Rightarrow_{rm}$ Prefix(c Tail)
$\Rightarrow_{rm}$ Prefix(c+E)
$\Rightarrow_{rm}$ Prefix(c+c Tail)
$\Rightarrow_{rm}$ Prefix(c+c)
$\Rightarrow_{rm}$ b(c+c)

Bottom-up Parsing (2/13)

G_0

- E→Prefix(E)
- E→c Tail
- Prefix→b
- Prefix→λ
- Tail→+E
- Tail→ λ

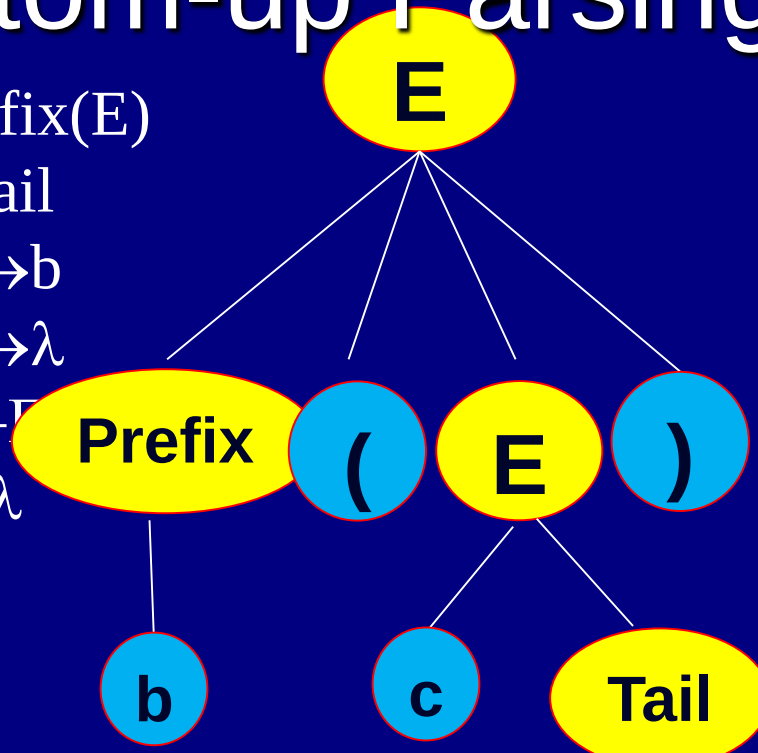
$E \Rightarrow_{rm}$ Prefix(E)
$\Rightarrow_{rm}$ Prefix(c Tail)
$\Rightarrow_{rm}$ Prefix(c+E)
$\Rightarrow_{rm}$ Prefix(c+c Tail)
$\Rightarrow_{rm}$ Prefix(c+c)
$\Rightarrow_{rm}$ b(c+c)



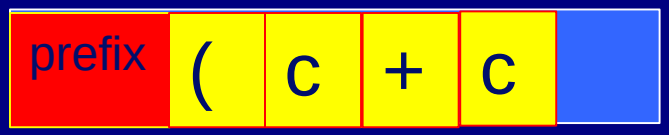
Bottom-up Parsing (3/13)

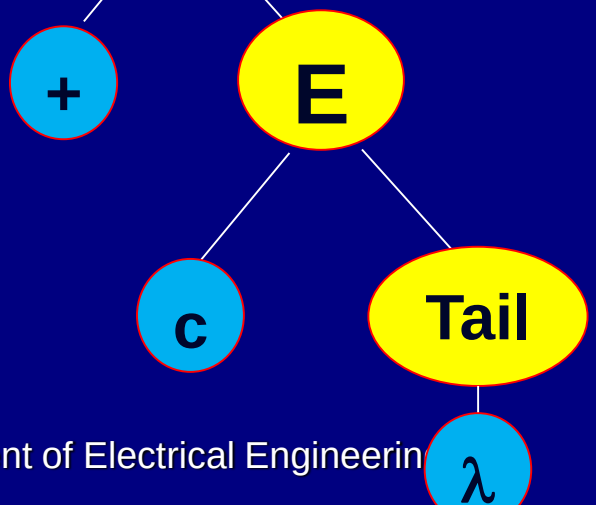
G_0

E→Prefix(E)
E→c Tail
Prefix→b
Prefix→λ
Tail→+E
Tail→ λ



$E \Rightarrow_{rm}$ Prefix(E)
$\Rightarrow_{rm}$ Prefix(c Tail)
$\Rightarrow_{rm}$ Prefix(c+E)
$\Rightarrow_{rm}$ Prefix(c+c Tail)
$\Rightarrow_{rm}$ Prefix(c+c)
$\Rightarrow_{rm}$ b(c+c)

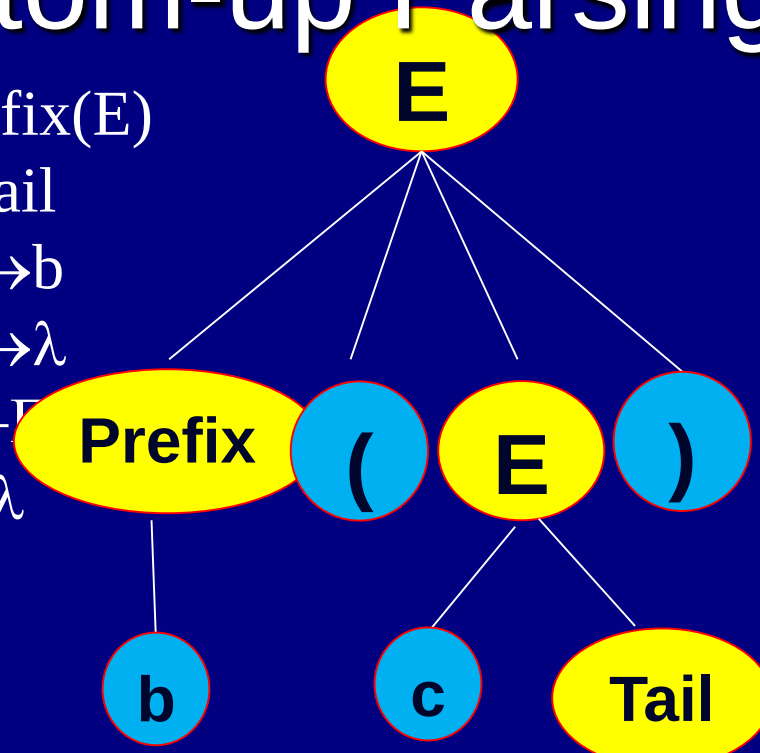




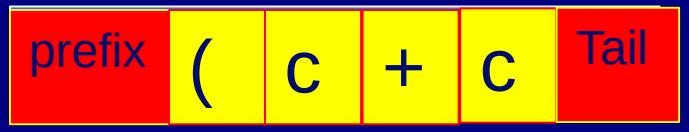
Bottom-up Parsing (4/13)

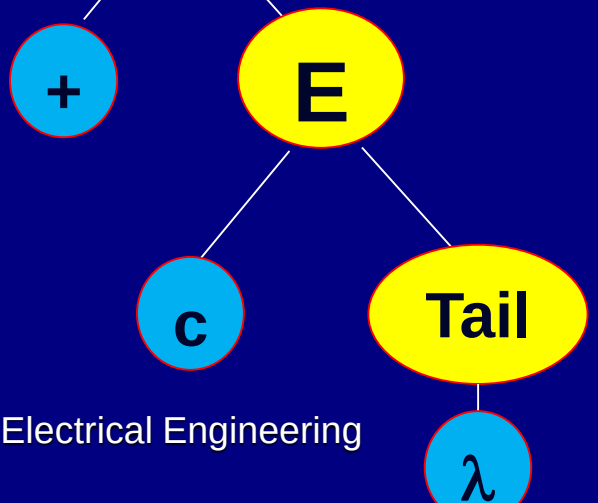
G_0

E→Prefix(E)
E→c Tail
Prefix→b
Prefix→λ
Tail→+E
Tail→ λ



$E \Rightarrow_{rm}$ Prefix(E)
$\Rightarrow_{rm}$ Prefix(c Tail)
$\Rightarrow_{rm}$ Prefix(c+E)
$\Rightarrow_{rm}$ Prefix(c+c Tail)
$\Rightarrow_{rm}$ Prefix(c+c)
$\Rightarrow_{rm}$ b(c+c)



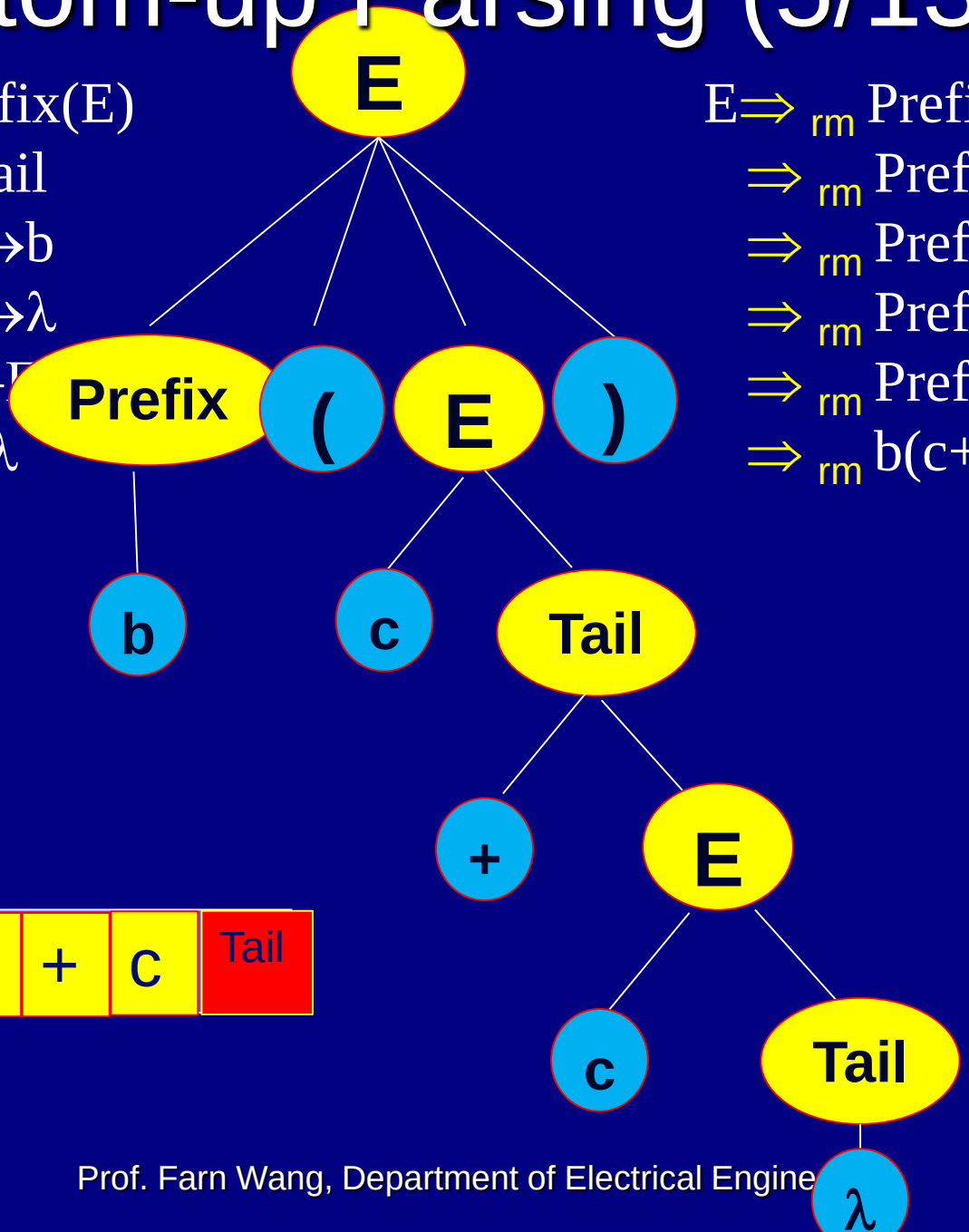


Bottom-up Parsing (5/13)

G_0

E→Prefix(E)
E→c Tail
Prefix→b
Prefix→λ
Tail→+E
Tail→ λ

$E \Rightarrow_{rm}$ Prefix(E)
$\Rightarrow_{rm}$ Prefix(c Tail)
$\Rightarrow_{rm}$ Prefix(c+E)
$\Rightarrow_{rm}$ Prefix(c+c Tail)
$\Rightarrow_{rm}$ Prefix(c+c)
$\Rightarrow_{rm}$ b(c+c)



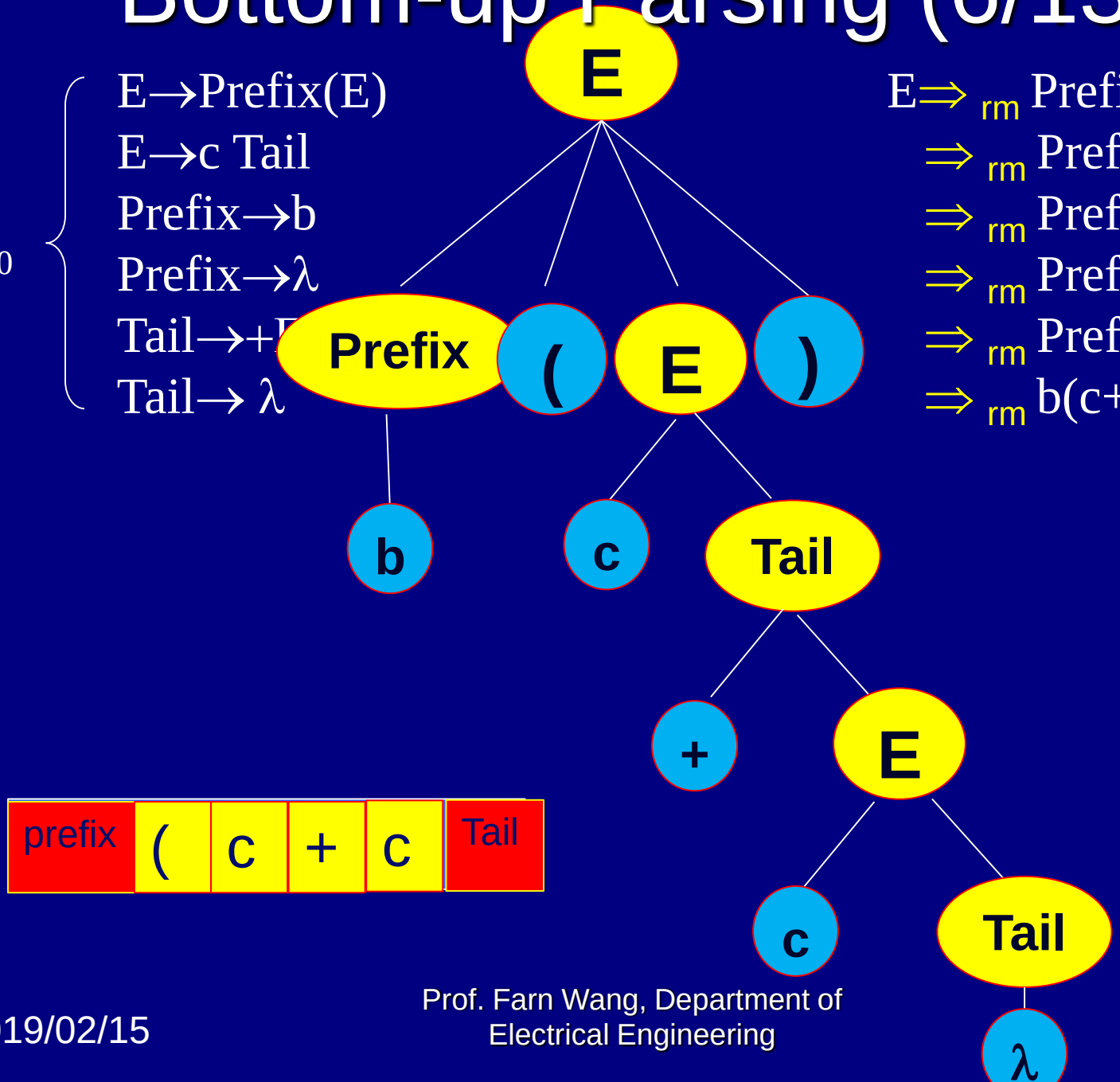
prefix	(	c	+	c	Tail

Bottom-up Parsing (6/13)

G_0

- E→Prefix(E)
- E→c Tail
- Prefix→b
- Prefix→λ
- Tail→+E
- Tail→ λ

$E \Rightarrow_{rm}$ Prefix(E)
$\Rightarrow_{rm}$ Prefix(c Tail)
$\Rightarrow_{rm}$ Prefix(c+E)
$\Rightarrow_{rm}$ Prefix(c+c Tail)
$\Rightarrow_{rm}$ Prefix(c+c)
$\Rightarrow_{rm}$ b(c+c)

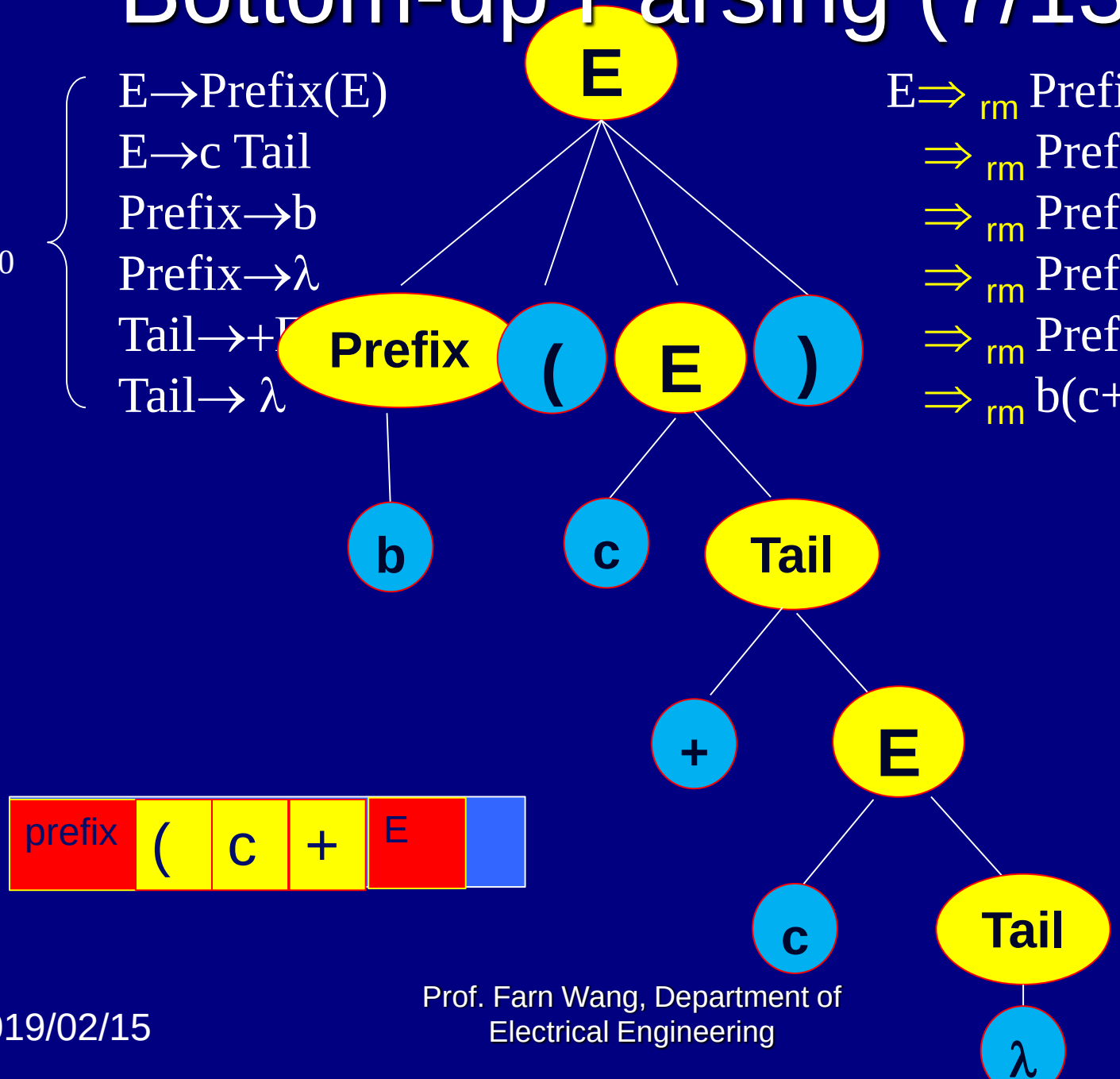


Bottom-up Parsing (7/13)

G_0

E→Prefix(E)
E→c Tail
Prefix→b
Prefix→λ
Tail→+E
Tail→ λ

$E \Rightarrow_{rm} Prefix(E)$
$\Rightarrow_{rm} Prefix(c\ Tail)$
$\Rightarrow_{rm} Prefix(c+E)$
$\Rightarrow_{rm} Prefix(c+c\ Tail)$
$\Rightarrow_{rm} Prefix(c+c)$
$\Rightarrow_{rm} b(c+c)$

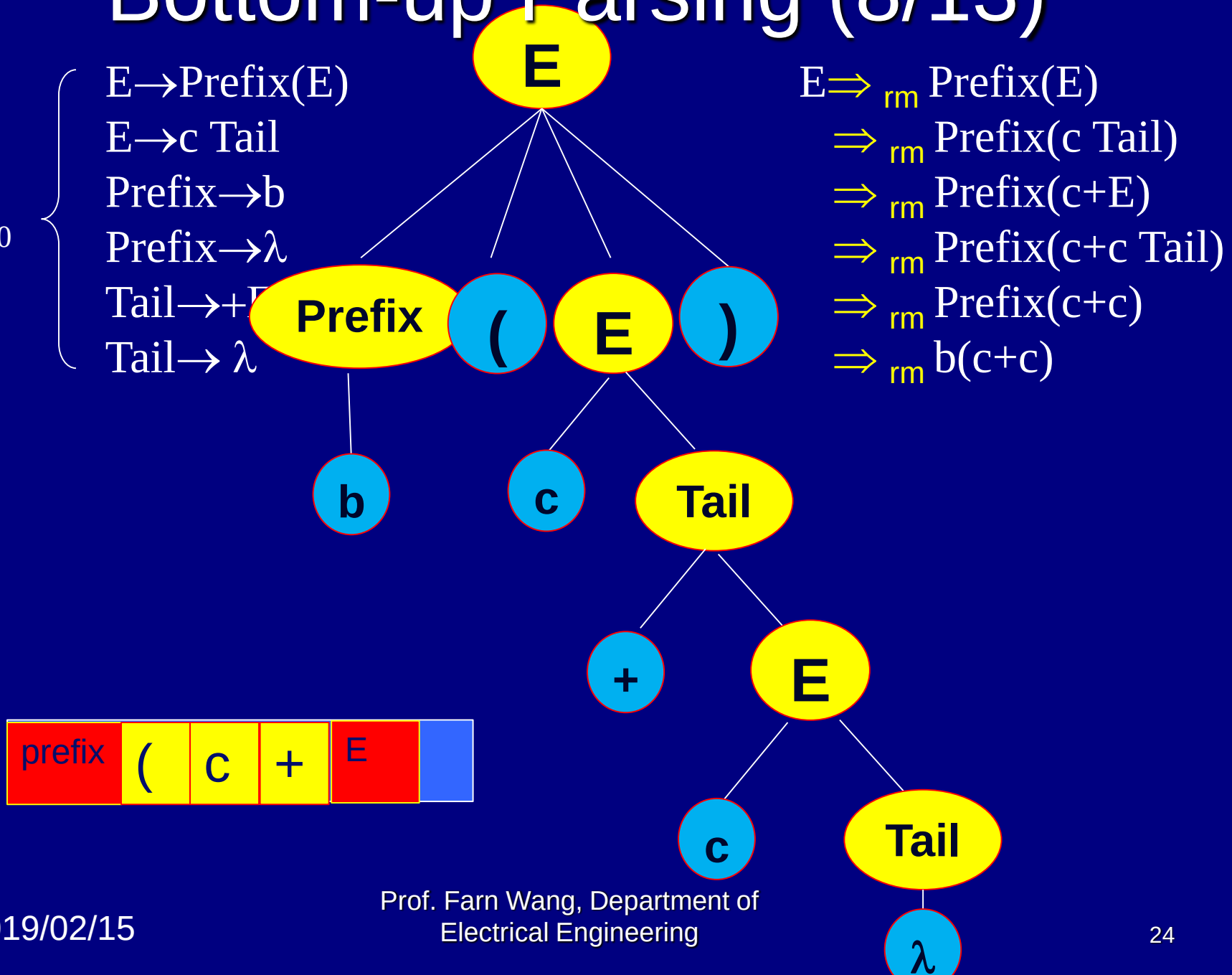


Bottom-up Parsing (8/13)

G_0

- E→Prefix(E)
- E→c Tail
- Prefix→b
- Prefix→λ
- Tail→+E
- Tail→ λ

$E \Rightarrow_{rm}$ Prefix(E)
$\Rightarrow_{rm}$ Prefix(c Tail)
$\Rightarrow_{rm}$ Prefix(c+E)
$\Rightarrow_{rm}$ Prefix(c+c Tail)
$\Rightarrow_{rm}$ Prefix(c+c)
$\Rightarrow_{rm}$ b(c+c)

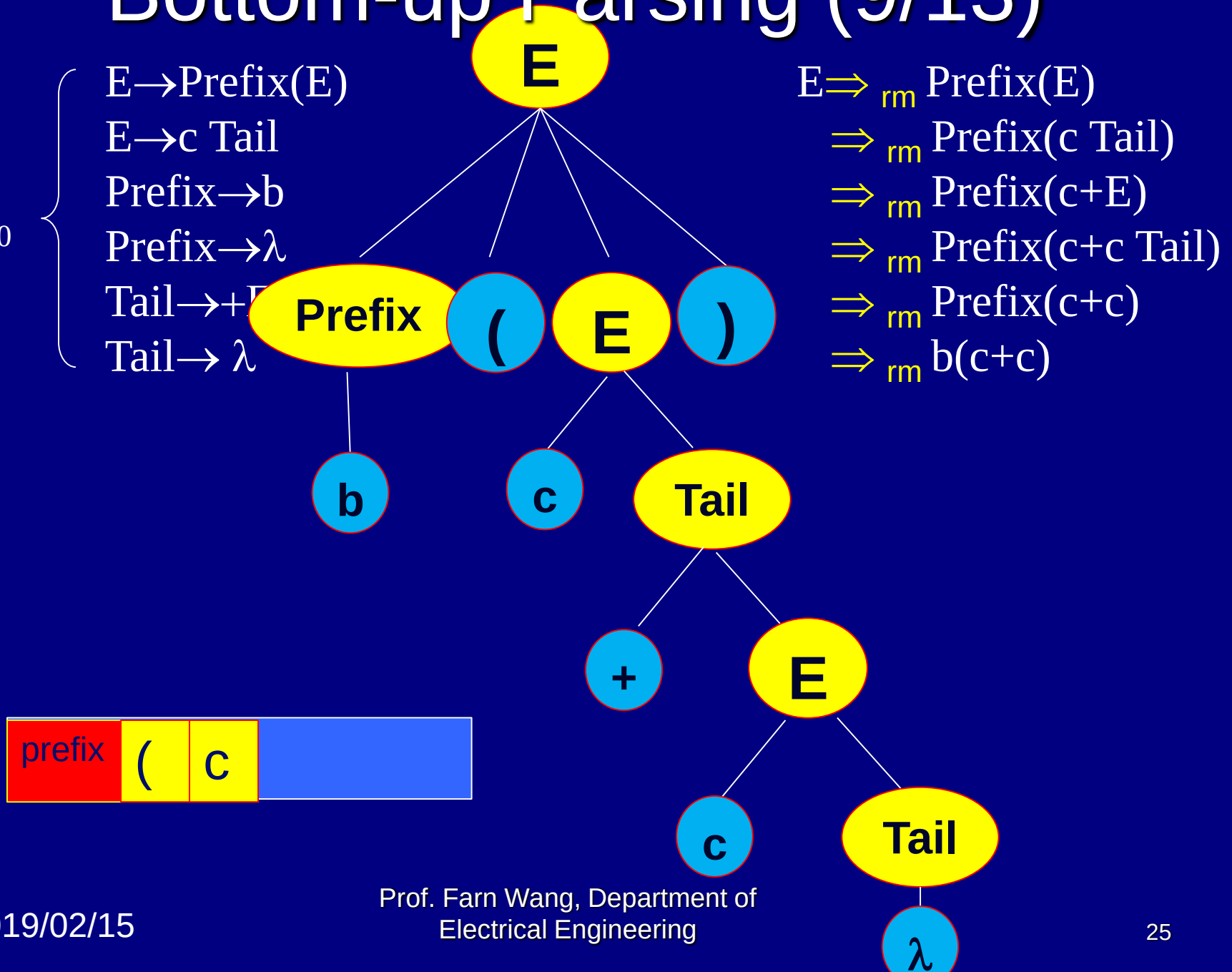


Bottom-up Parsing (9/13)

G_0

- E→Prefix(E)
- E→c Tail
- Prefix→b
- Prefix→λ
- Tail→+E
- Tail→ λ

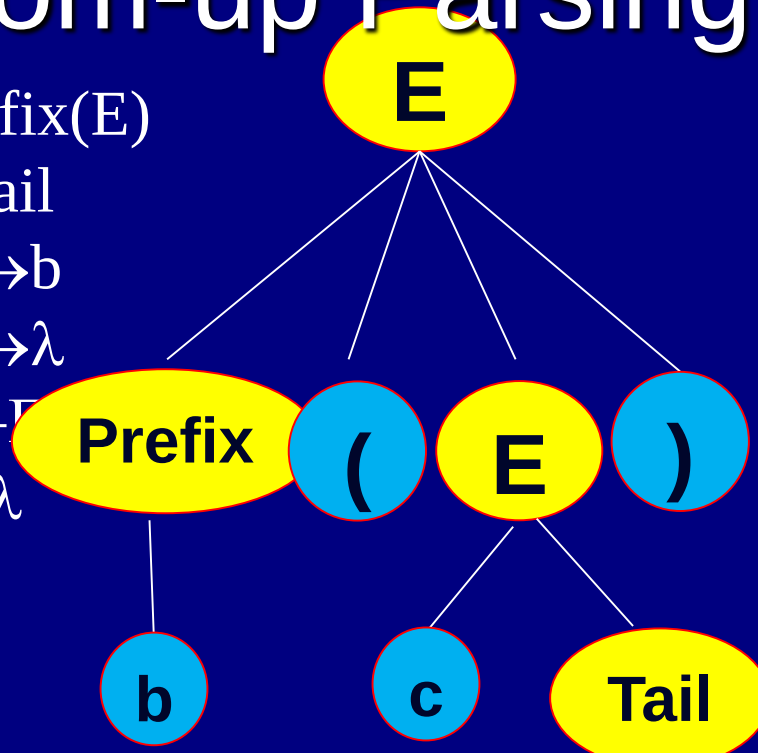
$E \Rightarrow_{rm}$ Prefix(E)
$\Rightarrow_{rm}$ Prefix(c Tail)
$\Rightarrow_{rm}$ Prefix(c+E)
$\Rightarrow_{rm}$ Prefix(c+c Tail)
$\Rightarrow_{rm}$ Prefix(c+c)
$\Rightarrow_{rm}$ b(c+c)



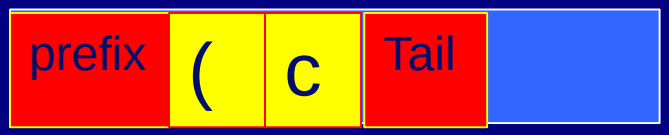
Bottom-up Parsing (10/13)

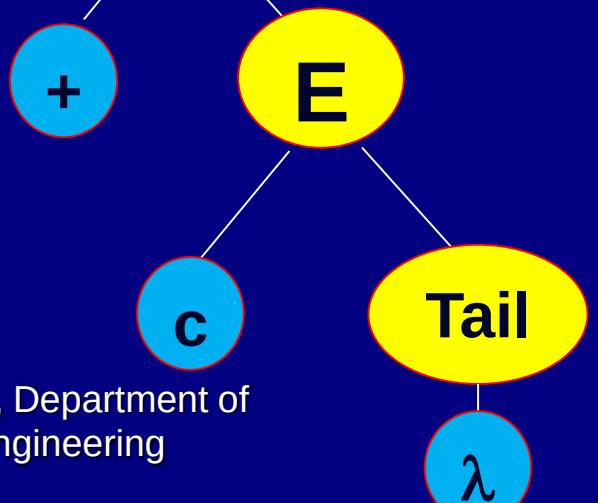
G_0

E→Prefix(E)
E→c Tail
Prefix→b
Prefix→λ
Tail→+E
Tail→ λ



$E \Rightarrow_{rm}$ Prefix(E)
$\Rightarrow_{rm}$ Prefix(c Tail)
$\Rightarrow_{rm}$ Prefix(c+E)
$\Rightarrow_{rm}$ Prefix(c+c Tail)
$\Rightarrow_{rm}$ Prefix(c+c)
$\Rightarrow_{rm}$ b(c+c)

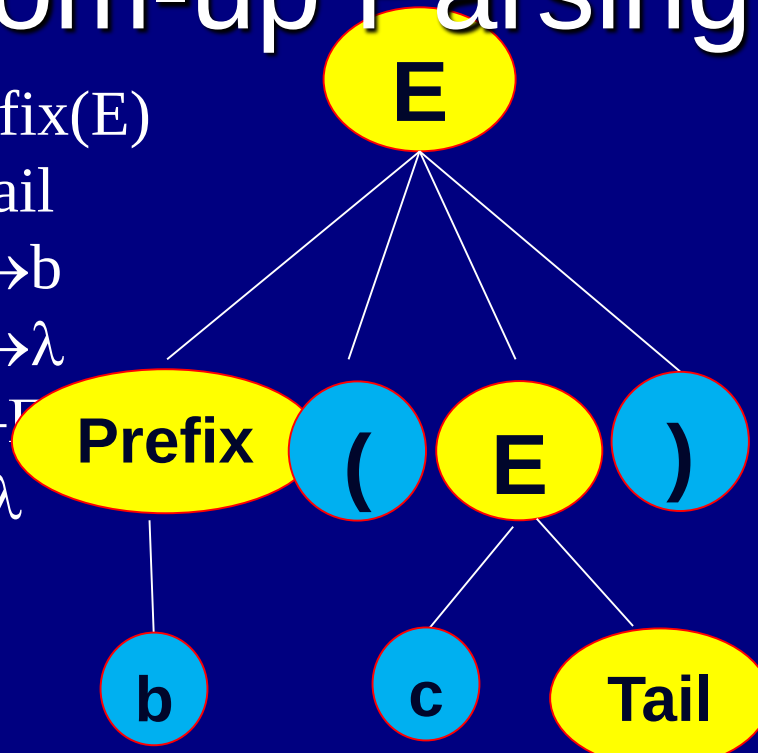




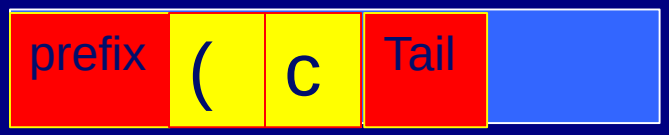
Bottom-up Parsing (11/13)

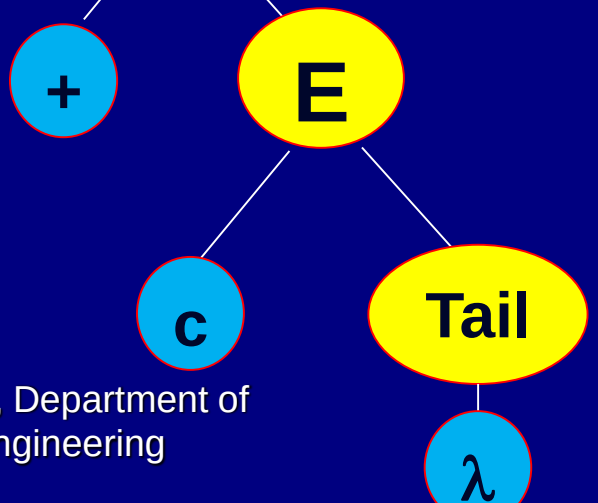
G_0

E→Prefix(E)
E→c Tail
Prefix→b
Prefix→λ
Tail→+E
Tail→ λ



$E \Rightarrow_{rm}$ Prefix(E)
$\Rightarrow_{rm}$ Prefix(c Tail)
$\Rightarrow_{rm}$ Prefix(c+E)
$\Rightarrow_{rm}$ Prefix(c+c Tail)
$\Rightarrow_{rm}$ Prefix(c+c)
$\Rightarrow_{rm}$ b(c+c)

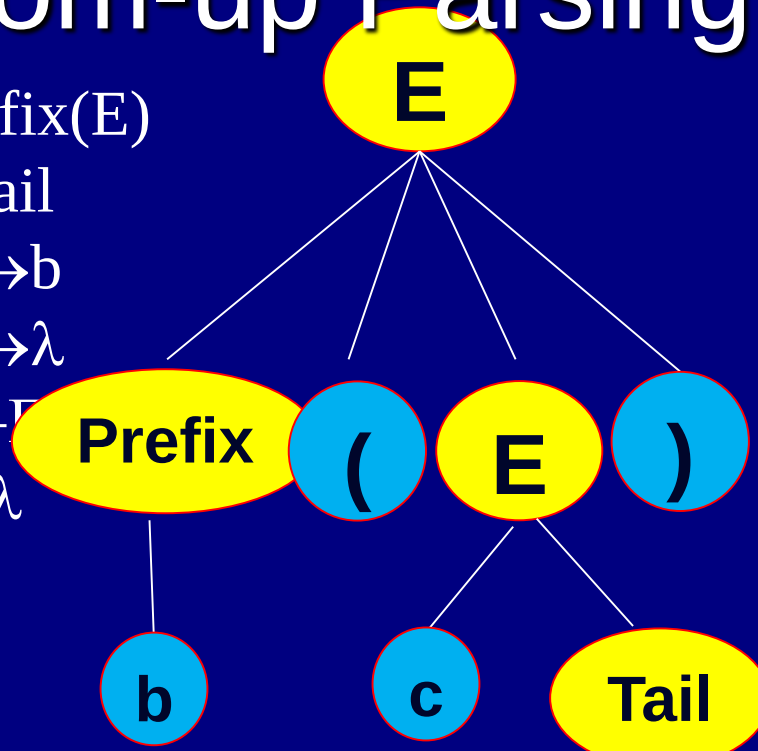




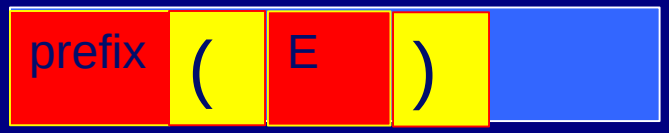
Bottom-up Parsing (12/13)

G_0

E→Prefix(E)
E→c Tail
Prefix→b
Prefix→λ
Tail→+E
Tail→ λ



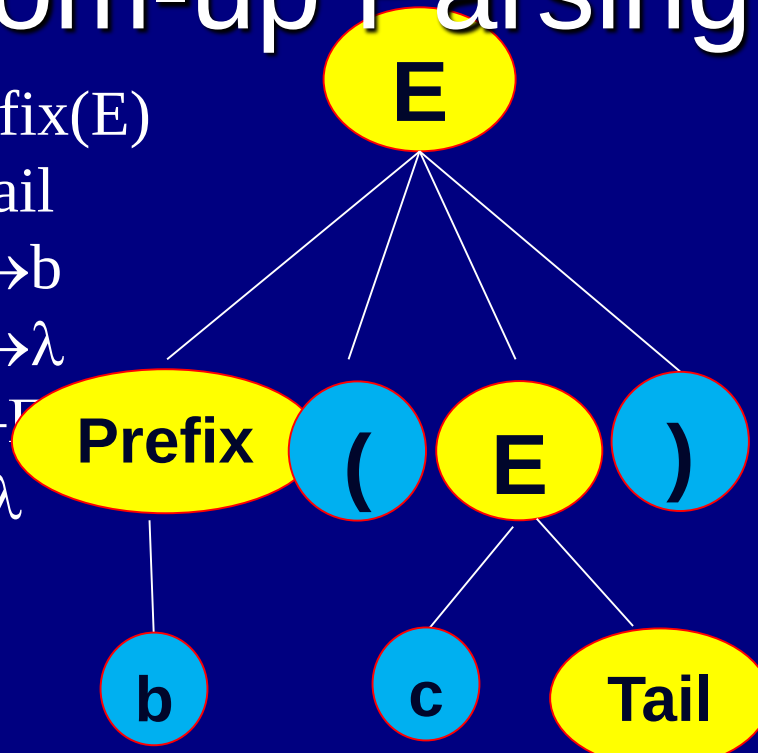
$E \Rightarrow_{rm}$ Prefix(E)
$\Rightarrow_{rm}$ Prefix(c Tail)
$\Rightarrow_{rm}$ Prefix(c+E)
$\Rightarrow_{rm}$ Prefix(c+c Tail)
$\Rightarrow_{rm}$ Prefix(c+c)
$\Rightarrow_{rm}$ b(c+c)



Bottom-up Parsing (13/13)

G_0

E→Prefix(E)
E→c Tail
Prefix→b
Prefix→λ
Tail→+E
Tail→ λ



$E \Rightarrow_{rm} Prefix(E)$
$\Rightarrow_{rm} Prefix(c\ Tail)$
$\Rightarrow_{rm} Prefix(c+E)$
$\Rightarrow_{rm} Prefix(c+c\ Tail)$
$\Rightarrow_{rm} Prefix(c+c)$
$\Rightarrow_{rm} b(c+c)$

Properties of CFG

- Useless nonterminals
  - S → A | B
    A → a
    B → Bb
    C → c

<derives no terminal string>

<unreachable>

- Reduced Grammar
  - A grammar is reduced after all useless non-terminals are removed
- Ambiguity

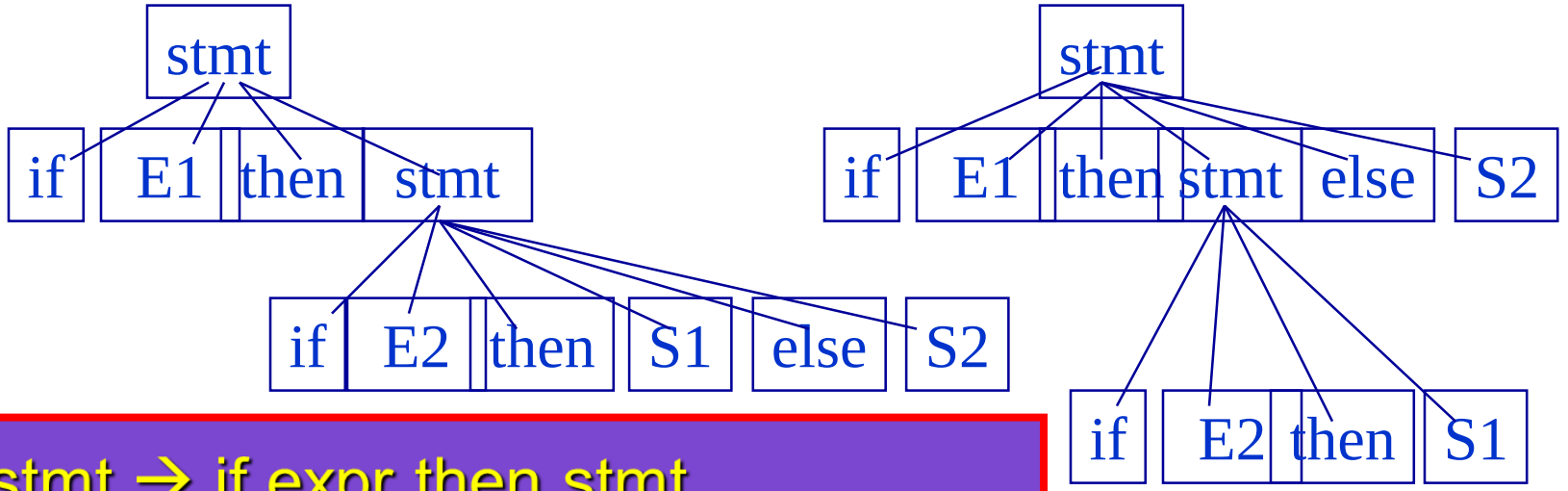
Ambiguity

- A grammar is *ambiguous* if it produces more than one parse tree for some sentences
- example 1: A+B+C (is it (A+B)+C or A+(B+C))
  - Improper production: expr → expr + expr | id
- example 2: A+B*C (is it (A+B)*C or A+(B*C))
  - Improper production: expr → expr + expr | expr * expr
- example 3: **if** E1 **then if** E2 **then** S1 **else** S2 (which **then** does the **else** match with)
  - Improper production:
    - stmt → if expr then stmt
      | if expr then stmt else stmt

Ambiguity

if E1 then if E2 then S1 else S2

Two parse trees of example 3



- stmt → if expr then stmt
  | if expr then stmt else stmt

Dangling Else Example

```
if ((k >=0) && (k < Table_SIZE))
        if (table[k] >=0)
                printf("Entry %d is %d\n", k, table[k]);
else printf("Error: index %d out of range \n"), k);
```

In ANSI C, an *else* is always assumed to belong to the *innermost if* statement possible.

```
if ((k >=0) && (k < Table_SIZE)) {
        if (table[k] >=0)
                printf("Entry %d is %d\n", k, table[k]); }
else printf("Error: index %d out of range \n"), k);
```

Eliminating Ambiguity

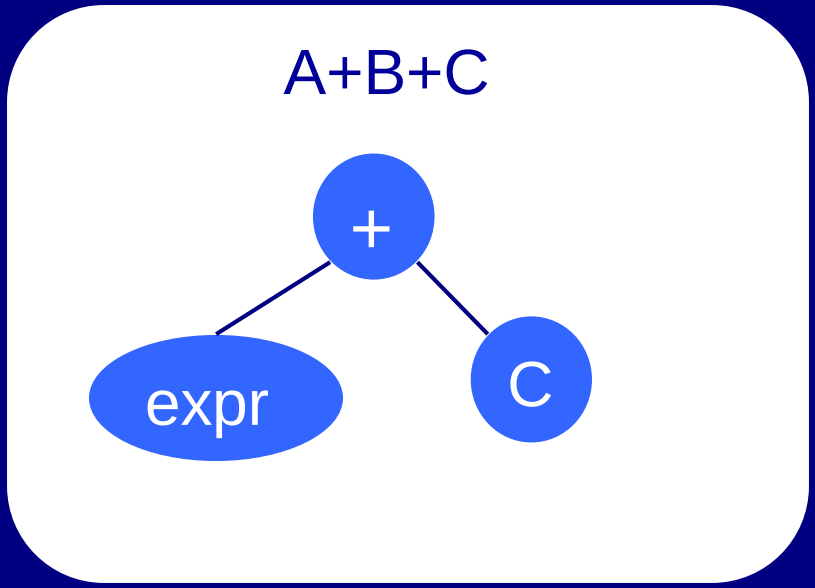
- Operator Associativity
  - expr → expr + term | term
- Operator Precedence
  - expr → expr + term | term

    term → term * factor | factor
- Dangling Else
  - In YACC, shift is favored over reduction to enforce the dangling “else” to associate with the innermost if.

Eliminating Ambiguity

- Operator Associativity
  - expr → expr + term | term
- Operator Precedence
  - expr → expr + term | term

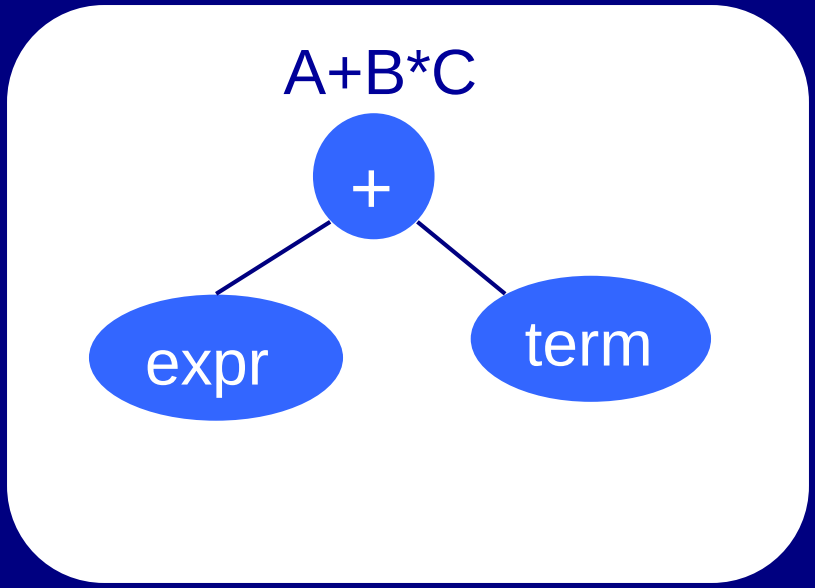
    term → term * factor | factor
- Dangling Else
  - In YACC, shift is favored over reduction to enforce the dangling “else” to associate with the innermost if.



Eliminating Ambiguity

- Operator Associativity
  - expr → expr + term | term
- Operator Precedence
  - expr → expr + term | term

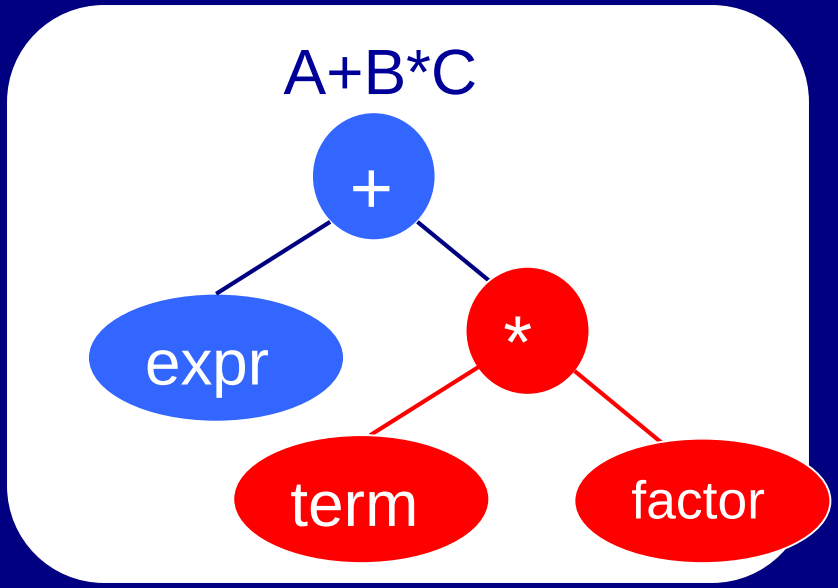
    term → term * factor | factor
- Dangling Else
  - In YACC, shift is favored over reduction to enforce the dangling “else” to associate with the innermost if.



Eliminating Ambiguity

- Operator Associativity
  - expr → expr + term | term
- Operator Precedence
  - expr → expr + term | term

    term → term * factor | factor
- Dangling Else
  - In YACC, shift is favored over reduction to enforce the dangling “else” to associate with the innermost if.



Left Factoring

Consider the following grammar

$A \rightarrow \alpha\beta1 \mid \alpha\beta2$

It is not easy to determine whether to expand A to $\alpha\beta1$ or $\alpha\beta2$

A transformation called left factoring can be applied. It becomes:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta1 \mid \beta2$

Exercise

stmt → if expr then stmt
| if expr then stmt else stmt

For the following grammar form:

A → $\alpha\beta 1 \mid \alpha\beta 2$

What is α? $\beta 1$? $\beta 2$?

α: if expr then stmt
$\beta 1$: λ
$\beta 2$: else stmt

Exercise

stmt → if expr then stmt
| if expr then stmt else stmt

For the following grammar form:

A → αβ1 | αβ2

What is α? β1? β2?

α: if expr then stmt
β1: λ
β2: else stmt

stmt → if expr then stmt S'
S' → λ | else stmt

Exercise

stmt → if expr then stmt
| if expr then stmt else stmt

For the following grammar form:

A → $\alpha\beta 1 \mid \alpha\beta 2$

What is α? $\beta 1$? $\beta 2$?

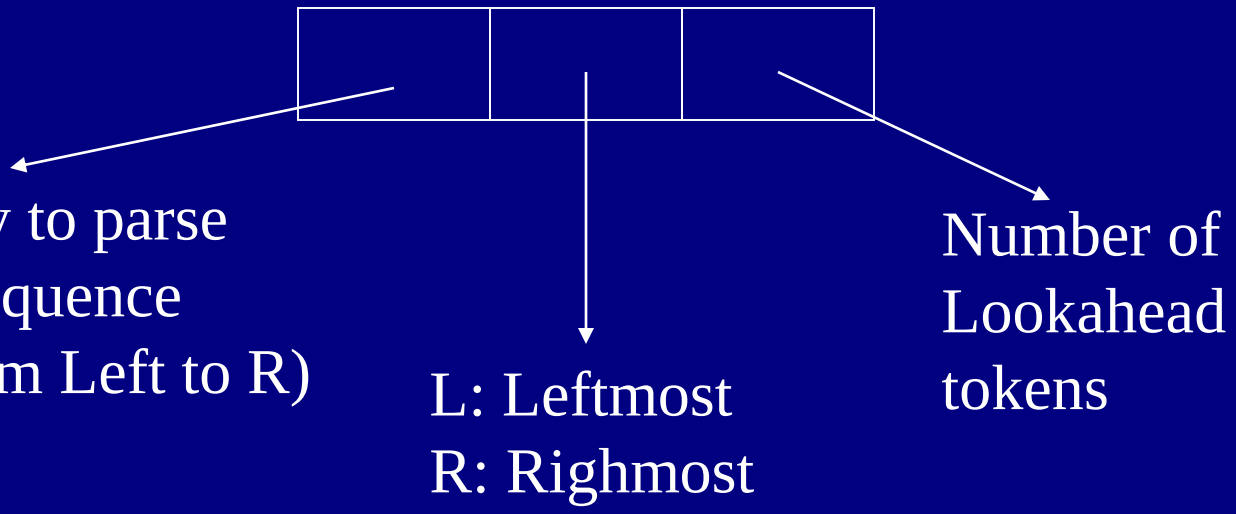
α: if expr then stmt
$\beta 1$: λ

Just one example --
in this case, left factoring does not solve the problem
you may try the new grammar on the example stmt in slide 30.

Parsers and Recognizers

- Recognizer
  - An algorithm that does boolean-valued test
    - "Is this input syntactically valid according to G?"
    - "Is this input in L(G)?"
- Parser
  - Answers more general questions
    - Is this input valid?
    - And, if it is, what is its structure (parse tree)?
- Two general approaches to parsing
  - Top-down and Bottom-up

Naming of parsing techniques



- Top-down
  - LL(1)
- Bottom-up
  - LR(1)

Parsing

CFG

Input program

G_0

$E \rightarrow Prefix(E)$

$E \rightarrow c\ Tail$

$Prefix \rightarrow b$

$Prefix \rightarrow \lambda$

$Tail \rightarrow +E$

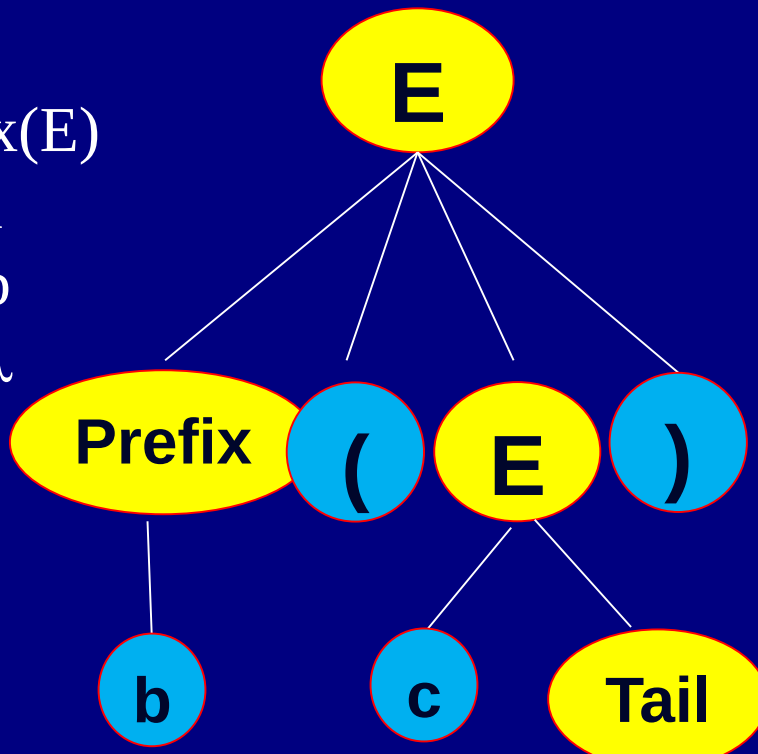
$Tail \rightarrow \lambda$

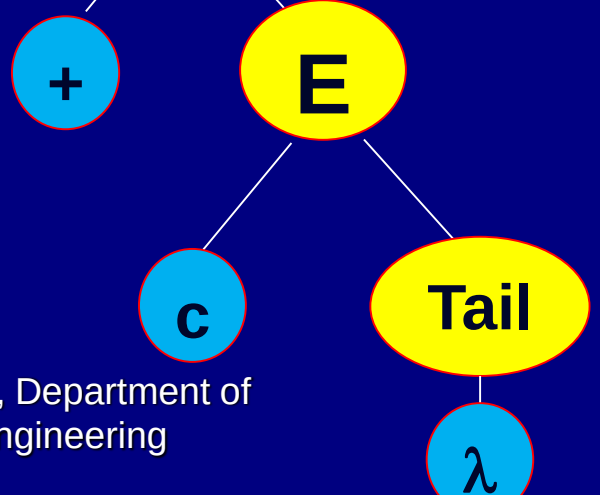
b (c + c)

Is the input program valid for the given CFG?

G_0

- E→Prefix(E)
- E→c Tail
- Prefix→b
- Prefix→λ
- Tail→+E
- Tail→ λ

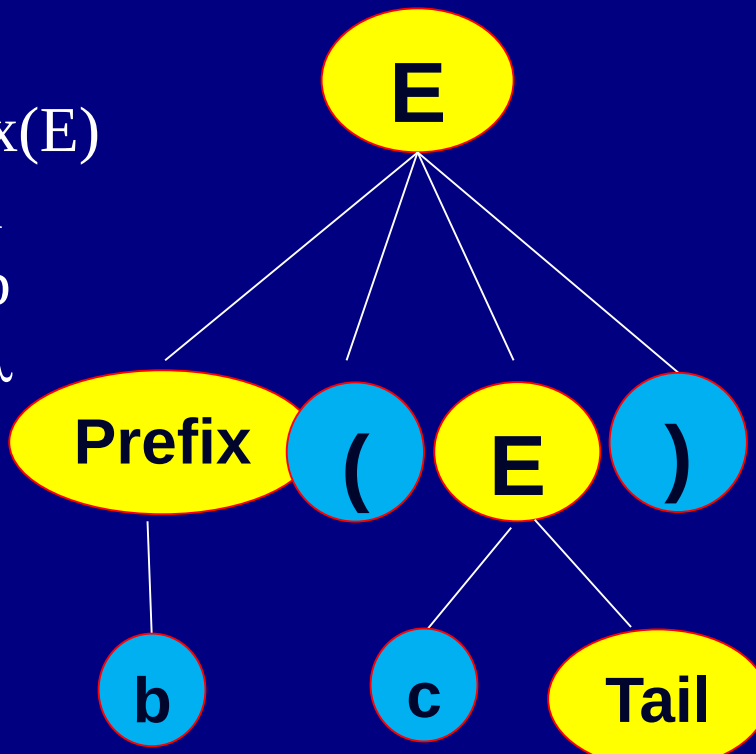




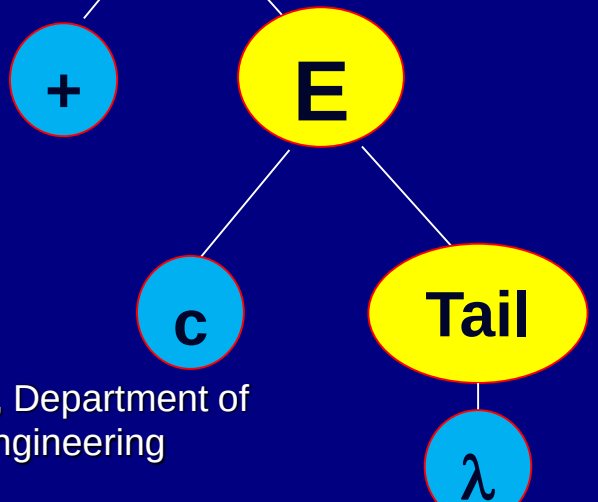
b (c + c)

G_0

- E→Prefix(E)
- E→c Tail
- Prefix→b
- Prefix→λ
- Tail→+E
- Tail→ λ



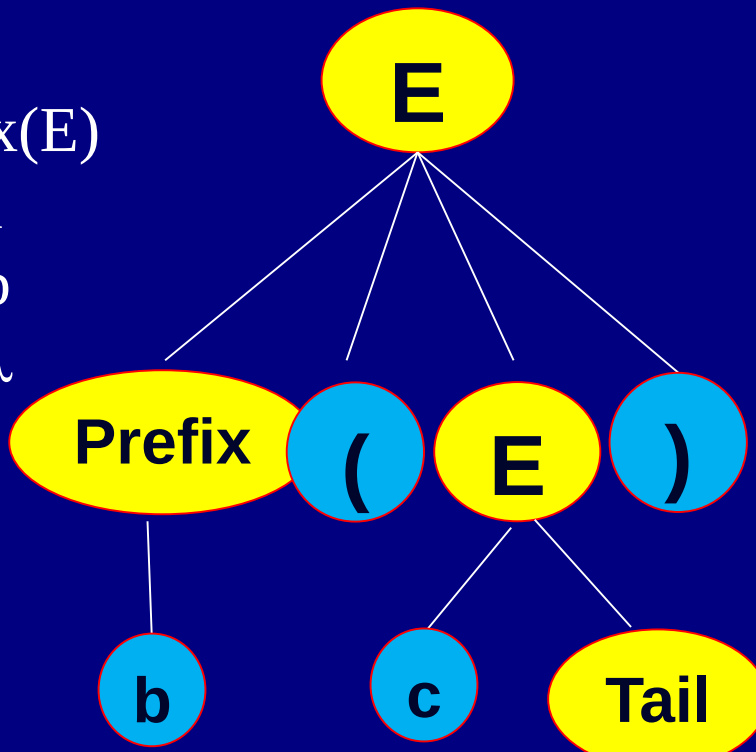
b (c + c)



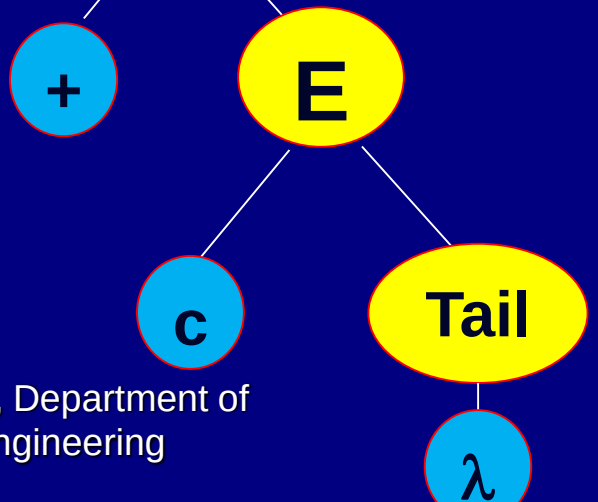
Top-Down Parsing:
Leftmost Derivation

G_0

- E→Prefix(E)
- E→c Tail
- Prefix→b
- Prefix→λ
- Tail→+E
- Tail→ λ



b (c + c)



Bottom-Up Parsing:
Leftmost Derivation

Grammar Analysis Algorithms

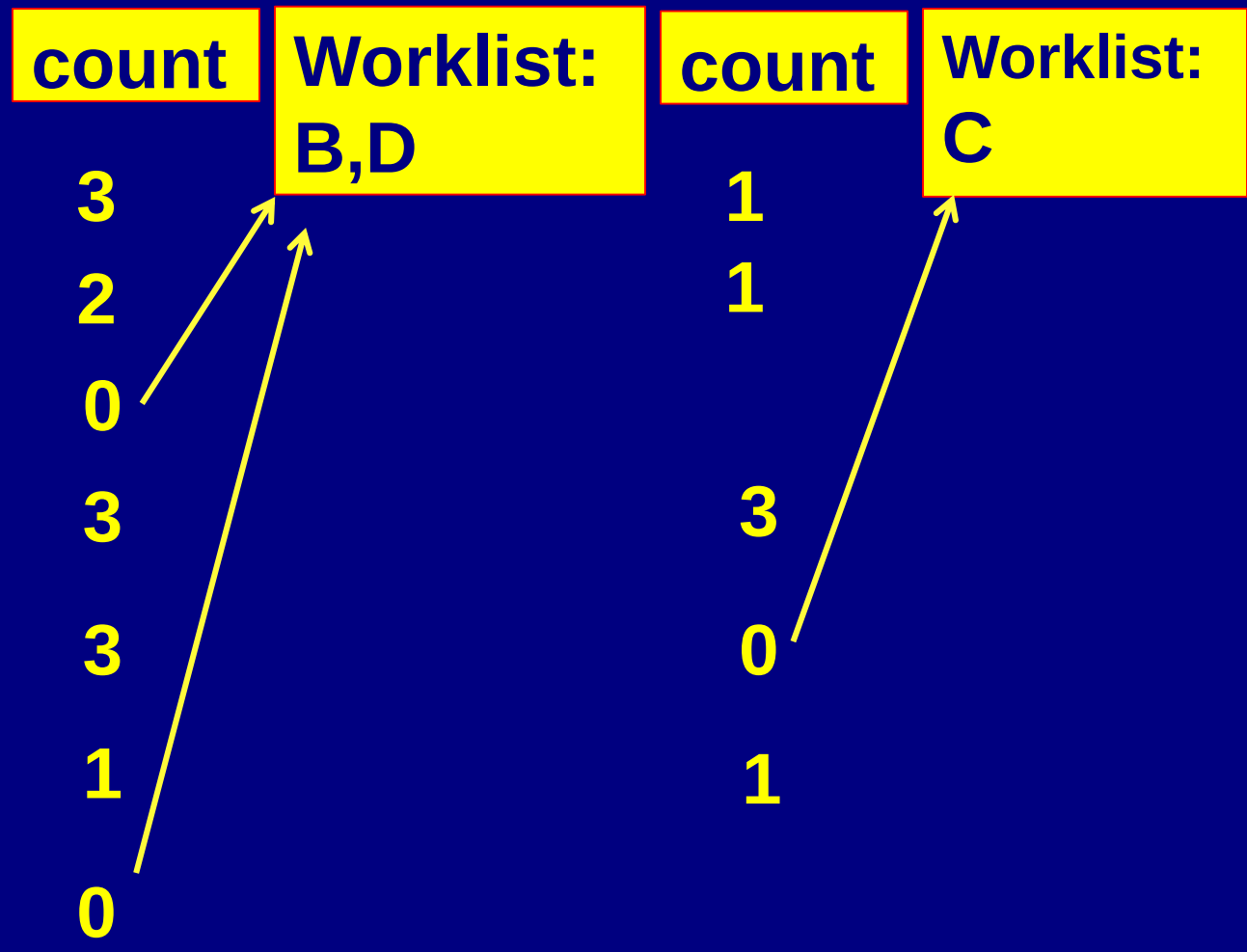
- Goal of this section:
  - Discuss a number of important analysis algorithms for Grammars
- What non-terminals can derive λ (*called nullable symbols*)?

  $A \Rightarrow BCD \Rightarrow BC \Rightarrow B \Rightarrow \lambda$

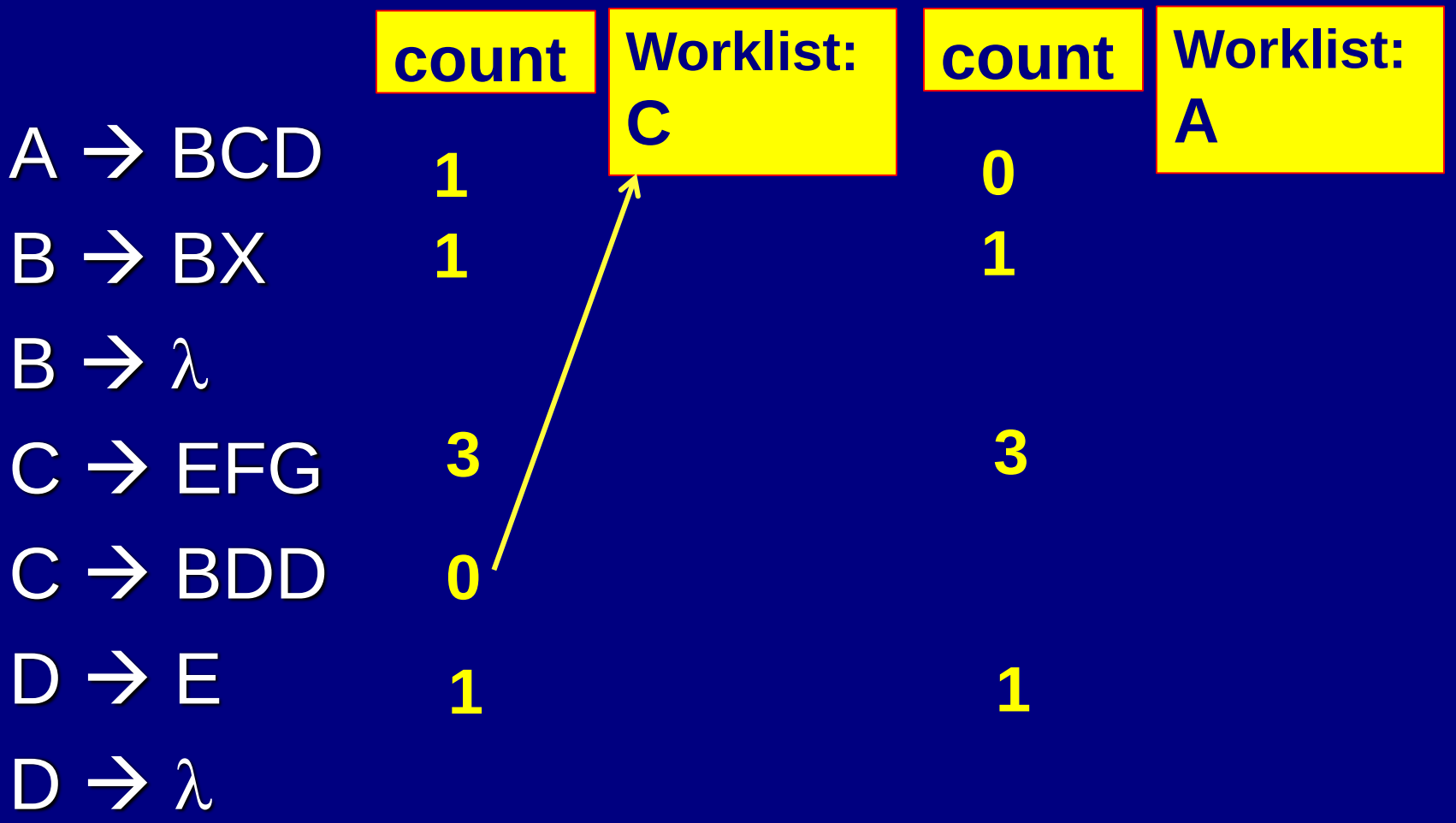
  - An iterative marking algorithm
    - Marking non-terminals that derive λ in one step, then marking non-terminals requires a parse tree of height 2, continue with increasing heights until no change.

What non-terminals can derive λ?

A → BCD
B → BX
B → λ
C → EFG
C → BDD
D → E
D → λ



What non-terminals can derive λ?



Definition of Follow and FIRST

- Follow(A)
  - Follow(A) is the set of terminals that may follow A in some sentential form. It provides the lookaheads that might signal the recognition of a production with A as the left hand side
  - Follow(A)={a∈Vt|S⇒* … Aa … }
- First(α)
  - The set of all the terminal symbols that can begin a sentential form derivable from α
  - If α is the right-hand side of a production, then First(α) contains terminal symbols that begin strings derivable from α
  - First(α)={a∈Vt| α ⇒* aβ} ∪ {λ| α ⇒* λ}

Definition of Follow and FIRST

- Follow(A)
  - Follow(A) is the set of terminals that may follow A in some sentential form. It provides the lookaheads that might signal the recognition of a production with A as the left hand side
  - Follow(A)={a∈Vt|S⇒* … Aa … }
- First(α)
  - The set of all the terminal symbols that can begin a sentential form derivable from α
  - If α is the right-hand side of a production, then First(α) contains terminal symbols that begin strings derivable from α
  - First(α)={a∈Vt| α ⇒* aβ} ∪ {λ| α ⇒* λ}

Example:
A → b X → WAa
B → b Y → WBc
……ba……
Should b be reduced to A or B?

Computing Follow and FIRST

- FIRST(Z)
  - If Z is a terminal, then FIRST(Z) = {Z}
  - If Z is a non-terminal, and Z→ Y1Y2…Yk for some k >=1, then place a in FIRST(Z) if for some *i*, a is in FIRST(Y*i*), and λ is in all of FIRST(Y1),… FIRST(Y*i-1*).
  - If Z→ λ is a production, then add λ to FIRST(Z)

When computing First(A), be careful of left recursion !
or any indirect recursion caused cycles !

- FOLLOW(A)
  - Place $ in FOLLOW(S)
  - If there is a production B → αAβ , then everything in FIRST(β) except λ, is in FOLLOW(A)
  - If there is a production B → αA, or a production B → αAβ, where FIRST(β) contains λ, then everything in FOLLOW(B) is in FOLLOW(A).

Example 1

$S \rightarrow aSe$
$S \rightarrow B$
$B \rightarrow bBe$
$B \rightarrow C$
$C \rightarrow cCe$
$C \rightarrow d$

First (S) = { ? }
First (B) = { ? }
First (C) = { ? }

Follow (S) = {? }
Follow (B) = {? }
Follow (C) = {? }

Example 1

$S \rightarrow aSe$
$S \rightarrow B$
$B \rightarrow bBe$
$B \rightarrow C$
$C \rightarrow cCe$
$C \rightarrow d$

First (S) = {a,b,c,d}
First (B) = {b,c,d}
First (C) = {c,d}

Follow (S) = {e, $}
Follow (B) = {e, $}
Follow (C) = {e, $}

Example 2

$S \rightarrow ABc$
$A \rightarrow a$
$A \rightarrow \lambda$
$B \rightarrow b$
$B \rightarrow \lambda$

First (S) = { ? }
First (A) = { ? }
First (B) = { ? }

Follow (S) = { ? }
Follow (A) = { ? }
Follow (B) = { ? }

Example 2

$S \rightarrow ABc$
$A \rightarrow a$
$A \rightarrow \lambda$
$B \rightarrow b$
$B \rightarrow \lambda$

First (S) = {a,b,c}
First (A) = {a, λ}
First (B) = {b, λ}

Follow (S) = {$}
Follow (A) = {b,c}
Follow (B) = {c}