Compilation

0368-3133

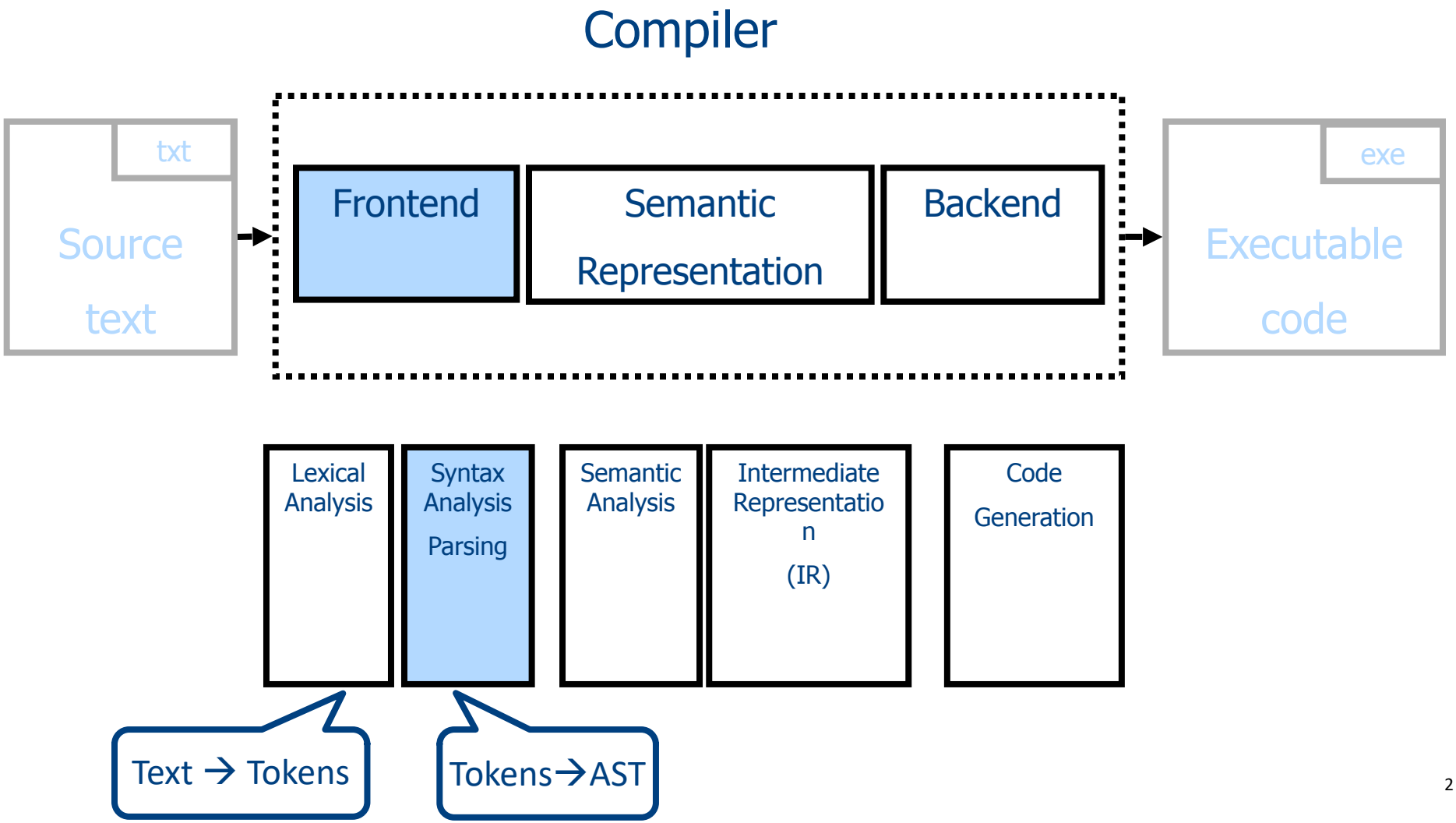
Lecture 2b:

Syntax Analysis:

Top-Down parsing

Slides' credit: Shachar Itzhaky, Eran Yahav

Conceptual Structure of a Compiler

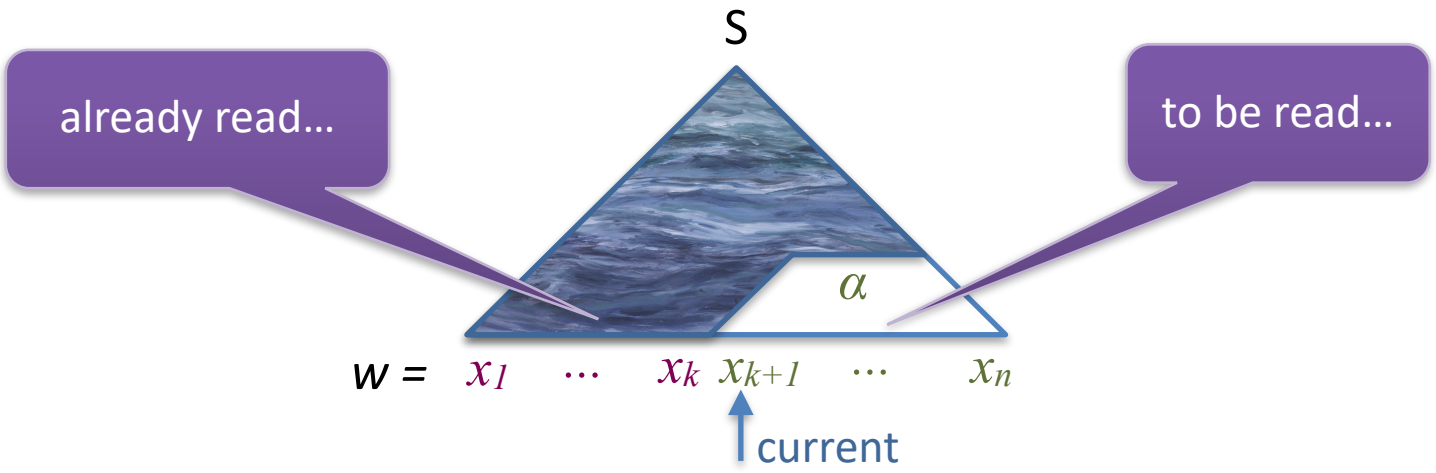


Parsing

- Goals
  - Decide whether the sequence of tokens a valid program in the source language
  - Construct a structured representation of the input
  - Error detection and reporting

Top-Down (Predictive) Parsers

- Construct the leftmost derivation
  - Read input from left to right
  - Apply rules "from left to right"
- Predict what rule to apply based on nonterminal and current token



$S \Longrightarrow AtB \Longrightarrow \beta tB \Longrightarrow^{*} x_1 \ldots x_k \alpha$

Top-Down Parsing

- Given a grammar G = ⟨V, T, P, S⟩ and a sentence *w*
- Goal: derive *w* using G
  - S $\Rightarrow^*$ w

- Idea
  - Apply production to leftmost nonterminal
  - Pick production rule based on next input token

Leftmost derivation

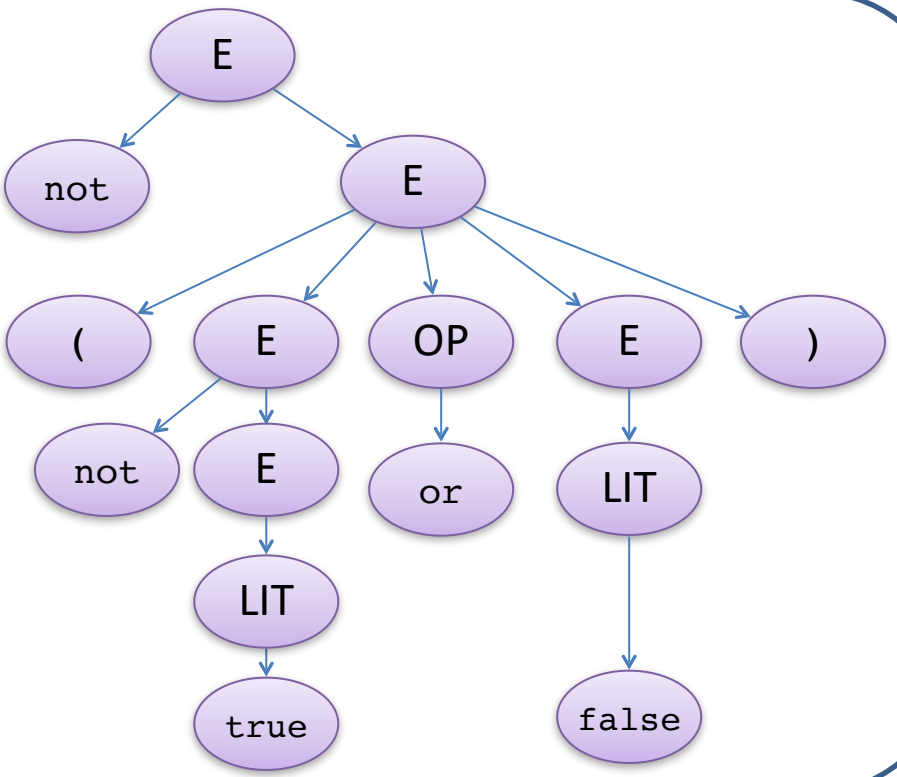
Lookahead

Boolean Expressions Example

```
not ( not true
      or false )
```

E → LIT | (E OP E) | not E
LIT → true | false
OP → and | or | xor

E ⟹
not E ⟹
not (E OP E) ⟹
not (not E OP E) ⟹
not (not LIT OP E) ⟹
not (not true OP E) ⟹
not (not true or E) ⟹
not (not true or LIT) ⟹
not (not true or false)



Production to apply is known from **next input token**

Challenges in top-down parsing

- Top-down parsing begins with virtually no information
  - Begins with just the start symbol, which matches every program

- How can we know which productions to apply?

Which productions to apply?

- In general, we can't guess effectively
  - There are some grammars for which the best we can do is guess and backtrack if we're wrong
    - Recall "Brute force" parsing

⇒ **We will sacrifice generality for predictability**

- Restricted grammars ($LL(k)$)
  - Know exactly which rule to apply
  - May require k lookahead tokens to decide

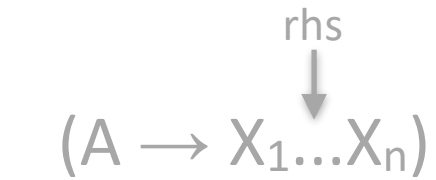
Predictive Parsing

- Recursive descent
- LL(k) grammars
- Generated LL(k) parsers
  - Using a pushdown automata

Recursive Descent Parsing

Recursive Descent Parsing

- Define a function for every nonterminal
- Every function work as follows:
  - Select a production rule
  - Go over rule's rhs "from left to right"
  - For terminal symbols — checks **match** with next input token
  - For nonterminal symbols — calls (recursively) their corresponding functions
- When there are several productions for a nonterminal, use lookahead



Matching tokens

```
void match(token t) {
  if (current == t)
    current = next_token();
  else
    error;
}
```

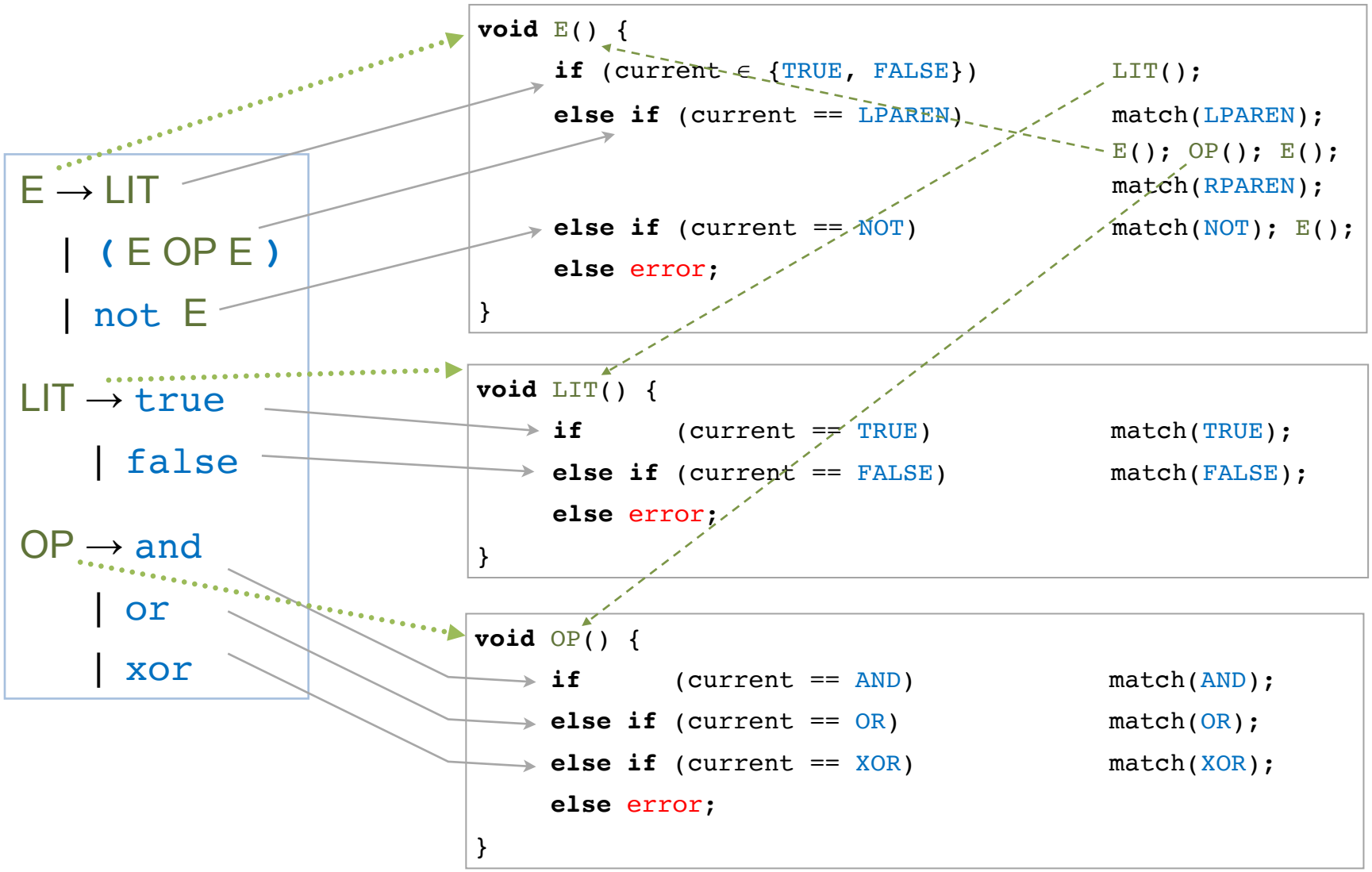
- Variable **`current`** holds the current input token

Functions for Nonterminals

```
E  → LIT | ( E OP E ) | not E
LIT→ true | false
OP → and | or | xor
```

```
void E() {
  if (current ∈ {TRUE, FALSE}) // E → LIT
    LIT();
  else if (current == LPAREN)  // E → ( E OP E )
    match(LPAREN); E(); OP(); E(); match(RPAREN);
  else if (current == NOT)     // E → not E
    match(NOT); E();
  else
    error;
}
void LIT() {
  if      (current == TRUE)  match(TRUE);
  else if (current == FALSE) match(FALSE);
  else error;
}
```

Functions for Nonterminals

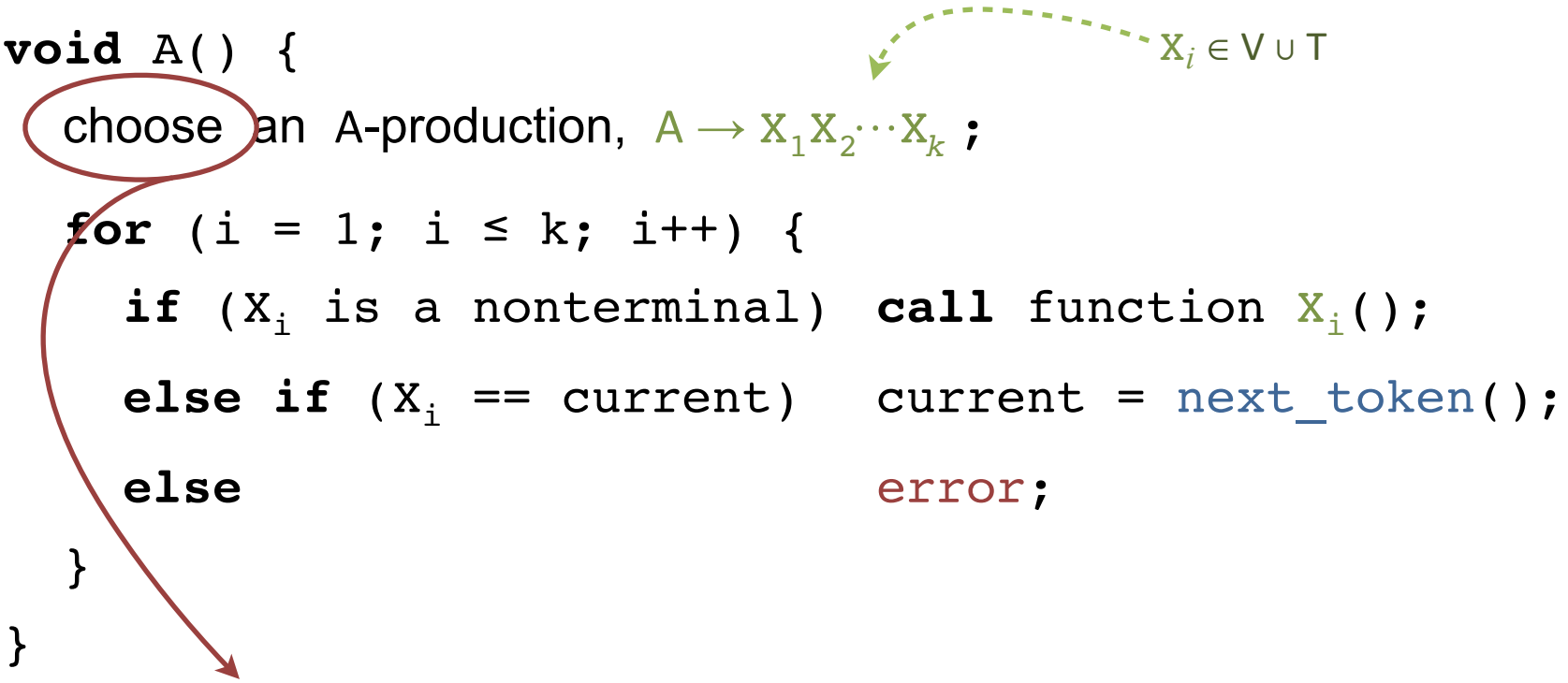


Recursive Descent — General Structure


```
void A() {
   choose an A-production, A → X1X2···Xk ;
   for (i = 1; i ≤ k; i++) {
     if (Xi is a nonterminal)  call function Xi();
     else call match(Xi);

   }
}
```

Recursive Descent — General Structure



- How do you pick the right A-production?
  - Can we generalize the rationale behind all those `if` statements we saw in the example?

LL(*k*) Grammars

LL(*k*) Grammars

- A grammar is in the class LL(*k*) when it can be parsed via:
  - Top-down analysis
    - Scanning the input from left to right (L)
    - Producing the leftmost derivation (L)
  - With lookahead of *k* tokens (*k*)
- A language is said to be LL(*k*) if it has an LL(*k*) grammar
- The simplest case is LL(1), which we discuss next

FIRST Sets

- To formalize the property (of a grammar) that we can determine a rule using a single lookahead token (LL(1)), we define the FIRST sets.
- For a sequence of symbols $\alpha \in (V \cup T)^*$
  - FIRST(α) = all terminals that α can start with
  - *i.e.*, $t \in \text{FIRST}(\alpha) \Leftrightarrow$ there exists a derivation sequence $\alpha \Rightarrow^* tw$
  - Special case: $\varepsilon \in \text{FIRST}(\alpha) \Leftrightarrow \alpha \Rightarrow^* \varepsilon$ (α is *nullable*)

(not really a terminal)

FIRST Sets - Example

```
E   → LIT | ( E OP E ) | not E |LIT maybe
LIT → true | false |ε
OP → and | or | xor
```

- In our Boolean expressions example:
  - FIRST(LIT) = ?
  - FIRST((E OP E)) = ?
  - FIRST(not E) = ?

These are all the alternatives for E → α

FIRST(E) = ?

Recursive Descent (revised)

Special case: No ε-productions

```
void A() {
   find an A-production, A → X1X2…Xk ,
                            (α)
         such that current ∈ FIRST(α)
   if no such production exists error;   // If there are multiple applicable
                                         // productions, the grammar is not LL(1)
   for (i = 1; i ≤ k; i++) {
     if (Xi is a nonterminal)    call function Xi();
     else if (Xi == current)     current = next_token();
     else                        error;
   }
}
```

FIRST Sets

- No intersection between FIRST sets of rules of the same nonterminal ⇒
  can *always** pick a single rule

- If the FIRST sets intersect, may need longer lookahead
  - LL(1) ⊂ LL(2) ⊂ LL(3) ⊂ ⋯
  - But the size of FIRST sets grow exponentially with *k*

* except when null (ε) productions exist — see next slides

Recursive Descent (Naive)

Handling ε-productions

```
void A() {
   find an A-production, A → X1X2…Xk ,        (α = X1X2…Xk)
         such that current ∈ FIRST(α) or α == ε
   if no such production exists error;
   if (α == ε) return;
   for (i = 1; i ≤ k; i++) {
     if (Xi is a nonterminal)    call function Xi();
     else if (Xi == current)     current = next_token();
     else                        error;
   }
}
```

(can be chosen regardless of current)

Handling null-productions

- What's the problem with A → ε?
- Example:

L(S) = ?

FIRST(A) = ?

FIRST(Ab) = ?

– Consider inputs: ab and b
– Need to know what comes after A in the sentence to select the right production

FOLLOW sets

FOLLOW Sets

- For *every* nonterminal A
  - FOLLOW(A) = set of ”terminals” that can immediately follow A (in some derivation)

FOLLOW Sets

- For *every* nonterminal A
  - FOLLOW(A) = set of "terminals" that can immediately follow A (in some derivation)
  - *i.e.*, t ∈ FOLLOW(A) ⇔ there exists some derivation sequence S $\Rightarrow^* \alpha A t \beta$
  - Special case: $ ∈ FOLLOW(A) ⇔ S $\Rightarrow^* \alpha A$ (i.e., A at the **end**)

(not really a terminal)

$$\text{FOLLOW}(A) = \{ t \in T \mid S \Rightarrow^* \alpha A t \beta \} \cup$$

$$\{ \$ \mid S \Rightarrow^* \alpha A \}$$

FOLLOW Sets - Example

$S \rightarrow Ab \mid c$

$A \rightarrow \varepsilon \mid a$

FOLLOW(A) = ?

$S \rightarrow Aab \mid c$

$A \rightarrow \varepsilon \mid a$

FOLLOW(A) = ?

Choosing Rules using FIRST & FOLLOW Sets - Example

$S \rightarrow Ab \mid c$

$A \rightarrow \varepsilon \mid a$

FIRST(ε) = { ε }

FIRST(a) = { a }

FOLLOW(A) = { b }

$S \rightarrow Aab \mid c$

$A \rightarrow \varepsilon \mid a$

FIRST(ε) = { ε }

FIRST(a) = { a }

FOLLOW(A) = { a }

FIRST-FOLLOW Conflict - Example

$S \rightarrow Ab \mid c$

$A \rightarrow \varepsilon \mid a$

FIRST(ε) = { ε }

FIRST(a) = { a }

FOLLOW(A) = { b }

$S \rightarrow Aab \mid c$

$A \rightarrow \varepsilon \mid a$

FIRST(ε) = { ε }

FIRST(a) = { a }

FOLLOW(A) = { a }

Consider ab and aab

LL(1) grammars

- A grammar is in the class LL(1) iff
  - For every two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ we have
    - $FIRST(\alpha) \cap FIRST(\beta) = \{\}$ // including ε
    - If $\varepsilon \in FIRST(\alpha)$ then $FIRST(\beta) \cap FOLLOW(A) = \{\}$
    - If $\varepsilon \in FIRST(\beta)$ then $FIRST(\alpha) \cap FOLLOW(A) = \{\}$

Recursive Descent (revised again)

```
void A() {
```

find an A-production, $A \rightarrow \overbrace{X_1 X_2 \ldots X_k}^{\alpha}$,

such that `current` $\in$ FIRST(α)

or $\varepsilon \in$ FIRST(α) and `current` $\in$ FOLLOW(A)

```
  if no such production exists error;
  if (α == ε) return;
  for (i = 1; i ≤ k; i++) {
    if (Xi is a nonterminal)   call function Xi();
    else if (Xi == current)    current = next_token();
    else                       error;
  }
}
```

This is the basis for an LL(1) parser.

(but it is still recursive — stay tuned)

Computing FIRST sets

FIRST Sets - Recursive Definition

$$\text{FIRST}(\alpha) = \{ t \in T \mid \alpha \Rightarrow^* t\beta \} \cup \{ \varepsilon \mid \alpha \Rightarrow^* \varepsilon \}$$

- For every terminal t and nonterminal N:
  - FIRST(ε) = $\{\varepsilon\}$
  - FIRST(t) = $\{t\}$ — t terminal
  - FIRST(N) = $\bigcup \{\text{FIRST}(\alpha) \mid N \rightarrow \alpha \in P\}$ — N nonterminal

FIRST Sets - Recursive Definition

$$\text{FIRST}(\alpha) = \{ t \in T \mid \alpha \Rightarrow^* t\beta \} \cup \{ \varepsilon \mid \alpha \Rightarrow^* \varepsilon \}$$

- For every terminal t and nonterminal N:
  - FIRST(ε) = $\{\varepsilon\}$
  - FIRST(t) = $\{t\}$ — t terminal
  - FIRST(N) = $\bigcup \{\text{FIRST}(\alpha) \mid N \rightarrow \alpha \in P\}$ — N nonterminal

β any sequence of terminals and nonterminals

- $\text{FIRST}(\mathcal{X}\beta) = \begin{cases} \text{FIRST}(\mathcal{X}) & \varepsilon \notin \text{FIRST}(\mathcal{X}) \\ \text{FIRST}(\mathcal{X}) \setminus \{\varepsilon\} \cup \text{FIRST}(\beta) & \varepsilon \in \text{FIRST}(\mathcal{X}) \end{cases}$

$\mathcal{X}$ is terminal or nonterminal

Possible that β=ε

FIRST Sets - Constraint Definition

$$\text{FIRST}(X) = \{ t \in T \mid X \Rightarrow^* t\beta \} \cup \{ \varepsilon \mid X \Rightarrow^* \varepsilon \}$$

- FIRST(ε) = $\{\varepsilon\}$
- FIRST(t) = $\{t\}$

- For every rule $X \rightarrow A_1 .. A_k \alpha$

  such that $\varepsilon \in \text{FIRST}(A_i)$ for every i=1…k

  $\text{FIRST}(\alpha) \subseteq \text{FIRST}(X)$

  - α can be ε

if $S \Rightarrow^* \gamma X \mu \Rightarrow \gamma A_1 \ldots A_k \alpha \mu \ \wedge A_1 \Rightarrow^* \varepsilon \wedge \ldots A_k \Rightarrow^* \varepsilon \ \wedge \ \alpha \Rightarrow^* t\beta$

then $S \Rightarrow^* \gamma X \mu \Rightarrow^* \gamma t \beta \mu$

Computing FIRST sets

$$G = (V, T, P, S)$$

1. For all $A \in V$, $\boldsymbol{FIRST}(\boldsymbol{A}) = \{t \mid \boldsymbol{A} \rightarrow t\boldsymbol{\beta} \in P\} \cup \{\epsilon \mid \boldsymbol{A} \rightarrow \epsilon \in P\}$

2. Repeat until no $FIRST(X)$ changes:
   foreach $A \in V$
   foreach $(\boldsymbol{A} \rightarrow \boldsymbol{B_1 B_2 \ldots B_k B_{k+1} \ldots B_n}) \in P$
   if $(\epsilon \in \boldsymbol{FIRST}(\boldsymbol{B_1}) \wedge \ldots \wedge \epsilon \in \boldsymbol{FIRST}(\boldsymbol{B_n}))$
   $\boldsymbol{FIRST}(\boldsymbol{A}) = \boldsymbol{FIRST}(\boldsymbol{A}) \cup \{\epsilon\}$
   for (k = 1 .. n-1)
   if $(\epsilon \in \boldsymbol{FIRST}(\boldsymbol{B_1}) \wedge \ldots \wedge \boldsymbol{FIRST}(\boldsymbol{B_k}))$ // k < n
   $\boldsymbol{FIRST}(\boldsymbol{A}) = \boldsymbol{FIRST}(\boldsymbol{A}) \cup (\boldsymbol{FIRST}(\boldsymbol{B_{k+1}}) \backslash \{\epsilon\})$

- This is known as fixed-point computation

FIRST sets computation example

STMT	EXPR	TERM

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

1. Initialization

STMT	EXPR	TERM
if	zero?	id
while	Not	constant
	++	
	--	

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

2. Iterate 1

STMT	EXPR	TERM
if while	zero? Not ++ --	id constant
zero? Not ++ --		

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

2. Iterate 2

STMT	EXPR	TERM
if while	zero? Not ++ --	id constant
zero? Not ++ --	id constant	

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

2. Iterate 3 – fixed-point

STMT	EXPR	TERM
if while	zero? Not ++ --	id constant
zero? Not ++ --	id constant	
id constant		

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant