Compilers

by
Marwa Yusuf

Lecture 6
Tues. 6-4-2021

Chapter 4 (4.4 to 4.4.3)

Syntax Analysis

Top-Down Parsing

- Constructing a parse tree for an input string starting from the root, and creating the nodes in preorder (depth-first), (finding a leftmost derivation for an input string).
- Ex: **id** + **id** * **id**

$$E \rightarrow T\,E'$$

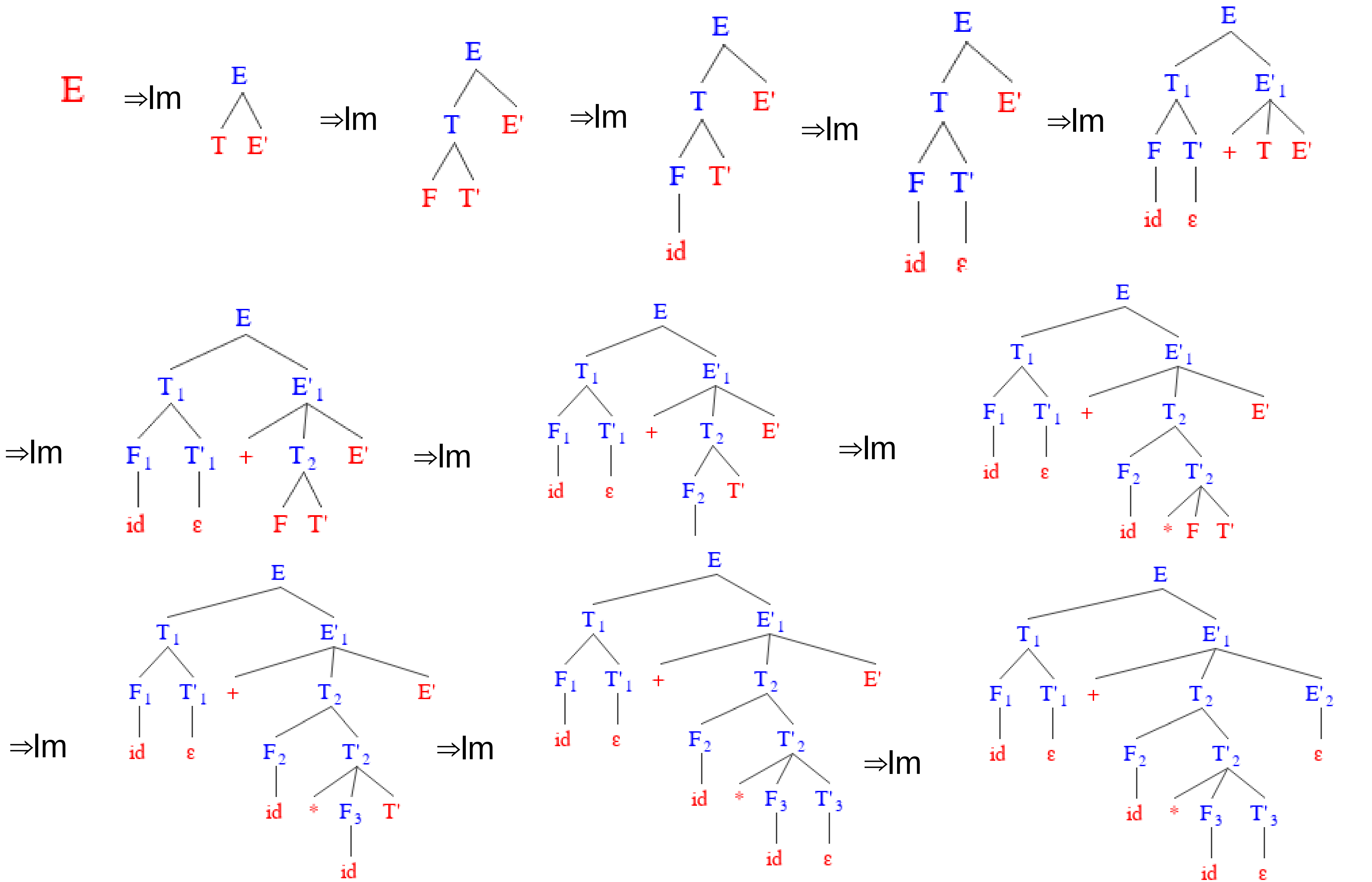
$$E' \rightarrow +\,T\,E' \mid \varepsilon$$

$$T \rightarrow F\,T'$$

$$T' \rightarrow *\,\mathrm{F}\,T' \mid \varepsilon$$

$$F \rightarrow (\,E\,) \mid \mathbf{id}$$

Top-Down Parsing



Top-Down Parsing

- At each step, the problem is choosing the next production.
- Topics:
  1) Recursive-decent parsing (backtracking).
  2) Predictive parsing (special case of recursive-decent parsing, no backtracking, lookahead)
     - LL(k): class of grammar for which we can build a predictive parser looking ahead k symbols. (ex: LL(1))
  3) Nonrecursive parsing (using stack).
  4) Error recovery.

Recursive-Decent Parsing

- One procedure for each non-terminal.
- Begin with the procedure of the start symbol, and halt announcing success if the entire input string is scanned.

```
void A{) {
  Choose an A-production, A → X1X2 … Xk;
  for ( i = 1 to k ) {
    if ( Xi is a nonterminal )
      call procedure Xi();
    else if ( Xi equals the current input symbol a )
      advance the input to the next symbol;
    else /* an error has occurred */;
  }
}
```

- Note: Nondeterministic (which production to choose at line 2?).

Recursive-Decent Parsing

- May require backtracking (not efficient), so not used frequently.
- To add backtracking to the previous code:
  - Try each of several productions in some order.
  - Failure at line 7 means going back to line 1 to try another production.
  - A local variable to store input string pointer to be able to backtrack.

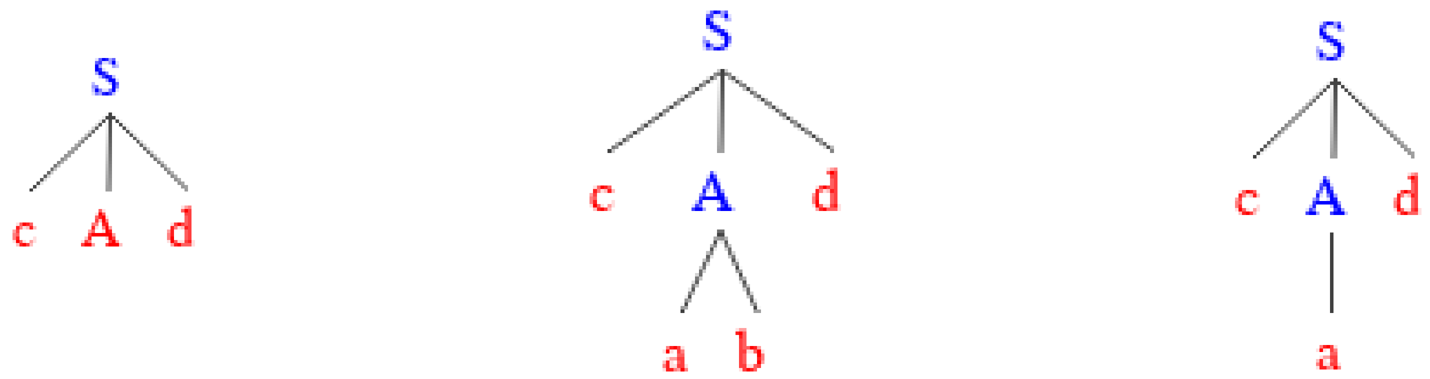
Recursive-Decent Parsing

- Ex: given grammar:

$$S \rightarrow c\,A\,d$$

$$A \rightarrow ab \mid a$$

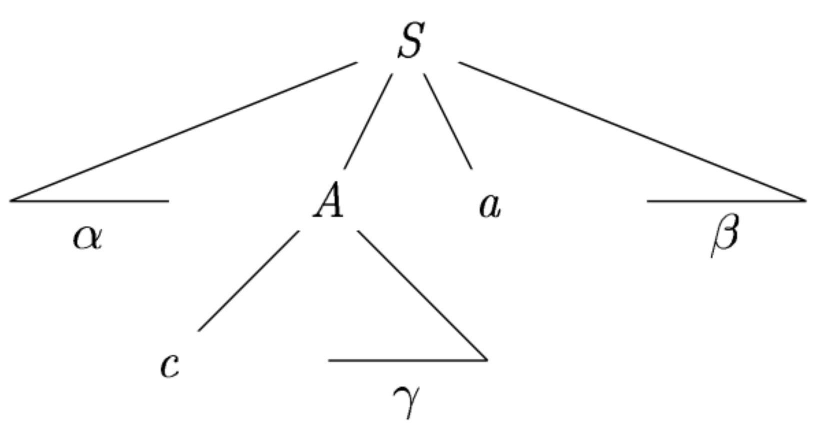
and input $w = cad$



- Left recursive grammar can cause infinite loop.

FIRST and FOLLOW

- Helps top-down (and bottom-up) parsing.
- To choose which production based on the next symbol and the FIRST sets of alternative productions available..
- Sets of tokens produced by FOLLOW can be used as synchronizing tokens in panic mode.
- FIRST(α) = the set of terminals that begin strings derived from α.
  - FIRST(A) = {c, ...}
- FOLLOW(A) = the set of terminals that can appear immediately to the right of A in some sentential form ($S \Rightarrow^* \alpha\ A\ a\ \beta$).
  - FOLLOW(A) = {a, ...}
- $: a special end symbol “endmarker” that is not a symbol of any grammar.
- If A can be the rightmost symbol in some sentential form, then $ is in the FOLLOW(A).



FIRST(X) computing

- Apply the following rules until no more terminals or ε can be added to any FIRST set:
  1) If X is a terminal, then FIRST(X) = {X}.
  2) If $X \rightarrow Y_1 Y_2 \ldots Y$
     a) If a is in FIRST(Y_i) and ε is in all of FIRST(Y_1), …,FIRST(Y_{i-1}) [i.e. $Y_1 \ldots Y_{i-1} \Rightarrow^* \varepsilon$] , then add a to FIRST(X).
     b) If ε is in FIRST(Y_j) for all j = 1, 2, …, k, then add ε to FIRST(X).
  3) If $X \rightarrow \varepsilon$ then add ε to FIRST(X).

FIRST(X) computing

- Apply the following rules until no more terminals or ε can be added to any FIRST set:
  1) If X is a terminal, then FIRST(X) = {X}.
  2) If $X \rightarrow Y_1 Y_2 \ldots Y$
     a) If a is in FIRST(Y_i) and ε is in all of FIRST(Y_1), …,FIRST(Y_{i-1}) [i.e. $Y_1 \ldots Y_{i-1} \Rightarrow^* \varepsilon$] , then add a to FIRST(X).
     b) If ε is in FIRST(Y_j) for all j = 1, 2, …, k, then add ε to FIRST(X).
  3) If $X \rightarrow \varepsilon$ then add ε to FIRST(X).

- Given Grammar:

  $E \rightarrow T\,E'$

  $E' \rightarrow +\,T\,E' \mid \varepsilon$

  $T \rightarrow F\,T'$

  $T' \rightarrow *\,\mathrm{F}\,T' \mid \varepsilon$

  $F \rightarrow (\,E\,) \mid \mathbf{id}$

- FIRST(E) = FIRST(T) = FIRST(F) = { (, **id** }
- FIRST(E') = { + , ε }
- FIRST(T') = { * , ε }

FOLLOW(X) computing

- Apply the following rules until nothing can be added to any FOLLOW set:
  1) Place $ in FOLLOW(S) where S is the start symbol.
  2) If $A \rightarrow \alpha B \beta$ then everything in FIRST(β) except ε is in FOLLOW(B).
  3) If $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε then everything in FOLLOW(A) is in FOLLOW(B).

FIRST(X) computing

- Apply the following rules until nothing can be added to any FOLLOW set:
  1) Place $ in FOLLOW(S) where S is the start symbol.
  2) If $A \rightarrow \alpha B \beta$ then everything in FIRST(β) except ε is in FOLLOW(B).
  3) If $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where FIRST(β) contains ε then everything in FOLLOW(A) is in FOLLOW(B).

- FIRST(E) = FIRST(T) = FIRST(F) = { (, **id** }
- FIRST(E') = { + , ε }
- FIRST(T') = { * , ε }

- Given Grammar:

  $E \rightarrow T\,E'$

  $E' \rightarrow +\,T\,E' \mid \varepsilon$

  $T \rightarrow F\,T'$

  $T' \rightarrow *\,\mathrm{F}\,T' \mid \varepsilon$

  $F \rightarrow (\,E\,) \mid \mathbf{id}$

- FOLLOW(E) = FOLLOW(E') = {) , $ }
- FOLLOW(T) = FOLLOW(T') = { + ,) , $ }
- FOLLOW(F) = { + , * ,) , $ }

Example

- Given Grammar:

  $E \rightarrow T\,E'$

  $E' \rightarrow +\,T\,E' \mid \varepsilon$

  $T \rightarrow F\,T'$

  $T' \rightarrow *\,\mathrm{F}\,T' \mid \varepsilon$

  $F \rightarrow (\,E\,) \mid \mathbf{id}$
- FIRST(F) = FIRST(T) = FIRST(E) = { (, **id** }
- FIRST(E') = { + , ε }
- FIRST(T') = { * , ε }
- FOLLOW(E) = FOLLOW(E') = {) , $ }
- FOLLOW(T) = FOLLOW(T') = { + ,) , $ }
- FOLLOW(F) = { + , * ,) , $ }

LL(1) Grammar

- LL(1): L for scanning input from left to right, L for using leftmost derivations, (1) for one lookahead symbol.
- Rich enough to cover most programming constructs, but take care in writing grammar. (no left-recursive or ambiguous grammar can be LL(1)).

LL(1) Grammar

- A grammar G is LL(1) iff whenever $A \rightarrow \alpha \mid \beta$:
  1) For no terminals a do both α and β derive strings beginning with a.
  2) At most one of α and β can derive the empty string.
  3) If $\beta \Rightarrow^* \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW (A). Likewise, if $\alpha \Rightarrow^* \varepsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

LL(1) Grammar

- Predictive parsers can be constructed for LL(1) since only current input symbol can determine the production to choose.
- Keywords for flow of control constructs generally satisfy LL(1) rules (**if, while, {**).
- Algorithm to construct parsing table M[A,a]:
    - Choose $A \rightarrow \alpha$ if next input symbol is in FIRST(α).
    - If $\alpha \Rightarrow^* \varepsilon$ then choose $A \rightarrow \alpha$ if next symbol is in FOLLOW(A) or if $ has been reached and $ is in FOLLOW(A).

Parsing Table Construction Algorithm

- **INPUT** : Grammar G.
- **OUTPUT** : Parsing table M.
- **METHOD** : For each production $A \rightarrow \alpha$ of the grammar, do the following:
  1) For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A, a]. (error in book page 224, l -4)
  2) If ε is in FIRSTS (α) , then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to M[A,b]. If ε is in FIRST (α) and $ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, $] as well.
  3) If, after performing the above, there is no production at all in M[A, a], then set M[A, a] to **error** (which we normally represent by an empty entry in the table).

Parsing Table Construction Example

$E \rightarrow T\,E'$

$E' \rightarrow +\,T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *\,\mathrm{F}\,T' \mid \varepsilon$

$F \rightarrow (\,E\,) \mid \mathbf{id}$

- FIRST(F) = FIRST(T) = FIRST(E) = { (, **id** }
- FIRST(E') = { + , ε }
- FIRST(T') = { * , ε }
- FOLLOW(E) = FOLLOW(E') = {) , $ }
- FOLLOW(T) = FOLLOW(T') = { + ,) , $ }
- FOLLOW(F) = { + , * ,) , $ }

Non-Terminal	Input Symbol					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

LL(1) Grammar

- If G is left recursive or ambiguous, M will have min. one multiply defined entry.
- Left-recursion elimination and left factoring may not be able to convert a given G to LL(1).
- Ex: $S \rightarrow i\ E\ t\ S\ S' \mid a$

  $S' \rightarrow e\ S \mid \varepsilon$

  $E \rightarrow b$
- To solve the conflict, we may always choose S' → eS on seeing an else.

Non-Terminal	Input Symbol					
	a	b	e	i	t	$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \varepsilon$ $S' \rightarrow eS$			$S' \rightarrow \varepsilon$
E		$E \rightarrow b$				