Top-Down Parsing

Syntax Analysis

- Parser = Syntax analyzer
  - Input: sequence of tokens from lexical analysis
  - Output: a parse tree of the program
    - E.g., AST
  - Process:
    - Try to derive from a starting symbol to the input string (How?)
    - Build the parse tree following the derivation
- Scanner and parser
  - Scanner looks at the lower level part of the programming language
    - Only the language for the tokens
  - Parser looks at the higher lever part of the programming language
    - E.g., if statement, functions
    - The tokens are abstracted

Recursive Descent Parsing

- How to parse
  - The easy way: try till it parses
- Algorithm
  - For a non-terminal
    - Generally, follow leftmost derivation
      - i.e., try to expand the first non-terminal
    - When there are more than one production rules for the non-terminal
      - Follow a predetermined order (easy for backtracking)
  - For the derived terminals
    - Compared against input
    - Match – advance input, continue
    - Not match – backtrack
  - Parsing fails if all possible derivations have been tried but still no match

Recursive Descent Parsing

- Example

  Rule 1: S → a S b

  Rule 2: S → b S a

  Rule 3: S → B

  Rule 4: B → b B

  Rule 5: B → ε

  - Parse: a a b b b
    - Has to use R1: S ⇒ a S b
    - Again has to use R1: a S b ⇒ a a S b b
    - Now has to use Rule 2 or 3, follow the order (always R2 first):
      - a a S b b ⇒ a a b S a b b ⇒ a a b b S a a b b ⇒ a a b b b S a a a b b
      - Now cannot use Rule 2 any more: ⇒ a a b b b B a a a b b ⇒ a a b b b B a a a b b ⇒ incorrect, backtrack
    - After some backtracking, finally tried
      - a S b ⇒ a a S b b ⇒ a a b B b b ⇒ a a b b b ⇒ worked

Recursive Descent Parsing

- Remarks
  - Why leftmost derivation
    - Generally, input is read from left to right
    - Allow matching with input easily after expansion
  - Doesn't work when the grammar is left recursive

    $E \rightarrow E + T$

    $T \rightarrow T * F \mid F$

    $F \rightarrow (E) \mid id \mid num$

    - Parse: x + 2 * y (id + num * id)
    - Will repeatedly apply $E \rightarrow E + T \Rightarrow$ won't terminate
  - Very inefficient
    - Since there is a large space to search for

$\Rightarrow$ Need *predictive parsing* that never backtracks

Predicative Parsing

- Need to immediately know which rule to apply when seeing the next input character
  - If for every non-terminal X
    - We know what would be the first terminal of each X's production
    - And the first terminal of each X's production is different
  - Then
    - When current leftmost non-terminal is X
    - And we can look at the next input character
    - $\Rightarrow$ We know exactly which production should be used next to expand X

Predicative Parsing

- Need to immediately know which rule to apply when seeing the next input character
  - If for every non-terminal X
    - We know what would be the first terminal of each X's production
    - And the first terminal of each X's production is different
  - Example

Rule 1: S → a S b

Rule 2: S → b S a

Rule 3: S → B

Rule 4: B → b B

Rule 5: B → ε

First terminal is a

First terminal is b

If next input is a, use R1
If next input is b, use R2

But, R3's first terminal is also b
Won't work!!!

Predicative Parsing

- Need to immediately know which rule to apply when seeing the next input character
  - If for every non-terminal X
    - We know what would be the first terminal of each X's production
    - And the first terminal of each X's production is different
  - What grammar does not satisfy the above?
    - If two productions of the same non-terminal have the same first symbol (N or T), you can see immediately that it won't work
      - $S \rightarrow b\ S\ a \mid b\ B$
      - $S \rightarrow B\ a \mid B\ C$
    - If the grammar is left recursive, then it won't work
      - $S \rightarrow S\ a \mid b\ B$, $B \rightarrow b\ B \mid c$
      - The left recursive rule of S can generate all terminals that the other productions of S can generate
        - $S \rightarrow b\ B$ can generate b, so, $S \rightarrow S\ a$ can also generate b

Predicative Parsing

- Need to rewrite the grammar
  - Left recursion elimination
    - This is required even for recursive descent parsing algorithm
  - Left factoring
    - Remove the leftmost common factors

Eliminate Left Recursion

- A grammar is left recursive
  - If it has at least one rule in the form $X \rightarrow X\alpha$
- How to eliminate left recursion?
  - Simple rule: $A \rightarrow A\alpha \mid \beta$
    - Derivations will always be: $A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow^* A\alpha\alpha\ldots\alpha \Rightarrow \beta\alpha\alpha\ldots\alpha$
    - Rewrite into:

      $A \rightarrow \beta A'$

      $A' \rightarrow \alpha A' \mid \varepsilon$

      - $A \Rightarrow \beta A' \Rightarrow \beta\alpha A' \Rightarrow \beta\alpha\alpha A' \Rightarrow^* \beta\alpha\alpha\ldots\alpha A' \Rightarrow \beta\alpha\alpha\ldots\alpha\varepsilon$

Eliminate Left Recursion

- How to eliminate left recursion?
  - In a general case:
    - Group A's production rules as follows

      $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

      - The left recursive ones and the non-left recursive ones
    - Rewrite A's production rules as follows

      $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$

      $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$

      - The derived string will always ending up with β_i in front
      - Followed by any combination of α_i's

Eliminate Left Recursion

- How to eliminate left recursion
  - Hidden left recursion

    $S \rightarrow A\alpha \mid b$

    $A \rightarrow A\beta \mid S\gamma \mid \varepsilon$

- Elimination steps
  - Index the non-terminals (A_1, A_2, …)

    for i := 1 to n do -- current production

    for j := 1 to i – 1 do -- previous non-terminals

    if A_j appears in A_i's production, like $A_i \rightarrow A_j\gamma$, then

    $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots$ (assume that $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots$)

    eliminate left recursion for A_i's productions

    Repeat this till no A_j appears in A_i's production rules (for $j < i$)

  - E.g., when processing A

    $A \rightarrow S\gamma$ is substituted by $A \rightarrow A\alpha\gamma \mid b\gamma$ first

    then eliminate left recursion for A

Eliminate Left Recursion

- For grammar

  $E \to E + T \mid T$

  $T \to T * F \mid F$

  $F \to (E) \mid id \mid num$

  $A \to A\alpha \mid \beta \Rightarrow$

  $A \to \beta A'$

  $A' \to \alpha A'$

- E

  $E \to TE'$

  $E' \to +TE' \mid \varepsilon$

- T

  $T \to FT'$

  $T' \to *FT' \mid \varepsilon$

- F

  $F \to (E) \mid id \mid num$

  - All start with non-terminals, no left recursion

Left Factoring

- Given a non-terminal A, represent its rules as:

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \gamma$

  - α is the longest matching prefix of several A productions
  - γ is the other productions that does not have leading α
  - α should be eliminated to achieve predictive parsing

- Rewrite the production rules

  $A \rightarrow \alpha A' \mid \gamma$

  $A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots$

Left Factoring

- Grammar

  $S \rightarrow$ if E then S else S

  $S \rightarrow$ if E then S

  $A \rightarrow \alpha\beta1 \mid \alpha\beta2 \mid \ldots \mid \gamma \Rightarrow$

  $A \rightarrow \alpha A' \mid \gamma$

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

- Rewrite the rules

  $S \rightarrow$ if E then S S'

  $S' \rightarrow$ else S $\mid \varepsilon$

  - Input: if a then if b then s1 else s2
  - $S \Rightarrow$ if E then S S' $\Rightarrow$ if a then S S' $\Rightarrow$ if a then if E then S S' S' $\Rightarrow$ if a then if b then S S' S' $\Rightarrow$ if a then if b then s1 S' S'
    - Could be: $\Rightarrow$ if a then if b then s1 else s2 ε
    - Could be: $\Rightarrow$ if a then if b then s1 ε else s2
  - Left factoring cannot eliminate ambiguity

Eliminate Left Recursion and Left Factoring

- Given a grammar
  - First eliminate left recursion
  - Then perform left factoring
  - Now, compute "First" -- first terminals of each production
- Grammar

  $E \rightarrow TE'$

  $E' \rightarrow +TE' \mid \varepsilon$

  $T \rightarrow FT'$

  $T' \rightarrow *FT' \mid \varepsilon$

  $F \rightarrow (E) \mid id \mid num$

  - No longer left recursive
  - No longer have left factors
  - Ready to compute first

A language can be expressed by an infinite number of grammars
You can rewrite a left recursive grammar into a totally different form to make it not left recursive
But such grammar rewriting is not left recursion elimination
Left recursion elimination is this specific process

First(α)

- First(α) = $\{ t \mid \alpha \Rightarrow^* t\beta \}$
  - Consider all possible terminal strings derived from α
  - The set of the first terminals of those strings
- For all terminals $t \in T$
  - First(t) = {t}

First(α)

- For all non-terminals $X \in N$
  - If $X \rightarrow \varepsilon \Rightarrow$ add ε to First(X)
  - If $X \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n$
    - α_i is either a terminal or a non-terminal (not a string as usual)

    $\Rightarrow$

    - Add all terminals in First(α_1) to First(X)
      - Exclude ε
    - If $\varepsilon \in$ First(α_1) $\wedge \ldots \wedge$ $\varepsilon \in$ First(α_{i-1}) then
      add all terminals in First(α_i) to First(X)
    - If $\varepsilon \in$ First(α_1) $\wedge \ldots \wedge$ $\varepsilon \in$ First(α_n) then
      add ε to First(X)
- Apply the rules until nothing more can be added
    - For adding t or ε: add only if t is not in the set yet

First(α)

- Grammar
  - E → TE'
  - E' → +TE' | ε
  - T → FT'
  - T' → *FT' | ε
  - F → (E) | id | num
- First
  - First(*) = {*}, First(+) = {+}, …
  - First(F) = {(, id, num}
  - First(T') = {*, ε}
  - First(T) = First(F) = {(, id, num}
  - First(E') = {+, ε}
  - First(E) = First(T) = {(, id, num}

First(α)

- Grammar
  - $S \rightarrow AB$
  - $A \rightarrow aA \mid \varepsilon$
  - $B \rightarrow bB \mid \varepsilon$
- First
  - First(A) = {a, ε}
  - First(B) = {b, ε}
  - First(S) = First(A) ={a, ε}

Is this complete?

First(α)

	If we see a	If we see b	If we see c	If we see d
When expanding S	Use R1	Use R2	Use R1	Use R2
When expanding A	Use R3	-	Use R4	-
When expanding B	-	Use R5	-	Use R6

- Grammar
  - $S \rightarrow AB \mid B \quad (R1 \mid R2)$
  - $A \rightarrow aA \mid c \quad (R3 \mid R4)$
  - $B \rightarrow bB \mid d \quad (R5 \mid R6)$

Input: acbd
Expands S, seeing a, use R1: $S \Rightarrow AB$
Expands A, seeing a, use R3: $AB \Rightarrow aAB$
Expands A, seeing c, use R4: $aAB \Rightarrow acB$
Expands B, seeing b, use R5: $acB \Rightarrow acbB$
Expands B, seeing d, use R6: $acbB \Rightarrow acbd$

- First
  - First(A) = {a, c}
  - First(B) = {b, d}
  - First(S) = First(A) $\cup$ First(B) = {a, b, c, d}
- Productions
  - First (R1) = {a, c}, First (R2) = {b, d}
  - First (R3) = {a}, First (R4) = {c}
  - First (R5) = {b}, First (R6) = {d}

First(α)

	If we see a	If we see b	If we see ε
When expanding S	Use R1	Use R1	Use R1
When expanding A	Use R2	-	Use R3
When expanding B	-	Use R4	Use R5

Input: aabb
Use R1: $S \Rightarrow AB$
Expands A, seeing a, use R2: $AB \Rightarrow aAB$
Expands A, seeing a, use R2: $aAB \Rightarrow aaAB$
Expands A, seeing b, What to do? Not in table!

- Grammar
  - $S \rightarrow AB$ (R1)
  - $A \rightarrow aA \mid \varepsilon$ (R2 | R3)
  - $B \rightarrow bB \mid \varepsilon$ (R4 | R5)
- First
  - First(A) = {a, ε}
  - First(B) = {b, ε}
  - First(S) = First(A) ∪ First(B) ={a, b, ε}
- Productions
  - First (R1) = {a, b, ε}
  - First (R2) = {a}, First (R3) = {ε}
  - First (R4) = {b}, First (R5) = {ε}

Follow(α)

- Follow(α) = $\{ t \mid S \Rightarrow^* \alpha t \beta \}$
  - Consider all strings that may follow α
  - The set of the first terminals of those strings
- Assumptions
  - There is a $ at the end of every input string
  - S is the starting symbol
- For all non-terminals only
  - Add $ into Follow(S)
  - If $A \rightarrow \alpha B \beta \Rightarrow$ add First(β) – $\{\varepsilon\}$ into Follow(B)
  - If $A \rightarrow \alpha B$ or

    $A \rightarrow \alpha B \beta$ and $\varepsilon \in$ First(β)
    $\Rightarrow$ add Follow(A) into Follow(B)

Follow(α)

Grammar

$S \rightarrow AB$ (R1)

$A \rightarrow aA \mid \varepsilon$ (R2 | R3)

$B \rightarrow bB \mid \varepsilon$ (R4 | R5)

- First
  - First(A) = {a, ε}
  - First(B) = {b, ε}
  - First(S) = First(A) ={a, b, ε}
- Productions
  - First (R1) = {a, b, ε}
  - First (R2) = {a}, Fi
  - First (R4) = {b}, Fi
- Follow
  - Follow(S) = {$}
  - Follow(B) = Follow(S) = {$}
  - Follow(A) = First(B) ∪ Follow(S) = {b, $}
    - Since ε ∈ First(B), Follow(S) should be in Follow(A)

	If we see a	If we see b
When expanding S	Use R1	Use R1
When expanding A	Use R2	?
When expanding B	-	Use R4

	If we see a	If we see b	If we see $
When expanding S	Use R1	Use R1	Use R1
When expanding A	Use R2	Use R3	Use R3
When expanding B	-	Use R4	Use R5

Construct a Parse Table

- Construct a parse table $M[N, T \cup \{\$\}]$
  - Non-terminals in the rows and terminals in the columns
- For each production $A \rightarrow \alpha$
  - For each terminal $a \in First(\alpha)$

    $\Rightarrow$ add $A \rightarrow \alpha$ to $M[A, a]$
    - Meaning: When at A and seeing input a, $A \rightarrow \alpha$ should be used
  - If $\varepsilon \in First(\alpha)$ then for each terminal $a \in Follow(A)$

    $\Rightarrow$ add $A \rightarrow \alpha$ to $M[A, a]$
    - Meaning: When at A and seeing input a, $A \rightarrow \alpha$ should be used
      - In order to continue expansion to ε
      - $X \rightarrow AC$ $A \rightarrow B$ $B \rightarrow b \mid \varepsilon$ $C \rightarrow cc$
  - If $\varepsilon \in First(\alpha)$ and $\$ \in Follow(A)$

    $\Rightarrow$ add $A \rightarrow \alpha$ to $M[A, \$]$
    - Same as the above

First(α) and Follow(α) – another example

Grammar

E → TE’
E’ → +TE’ | ε
T → FT’
T’ → *FT’ | ε
F → (E) | id | num

- First(*) = {*}
- First(F) = {(, id, num}
- First(T’) = {*, ε}
- First(T) = First(F) = {(, id, num}
- First(E’) = {+, ε}
- First(E) = First(T) = {(, id, num}

- Follow(E) = {$,)}
- Follow(E’) = Follow(E) = {$,)}
- Follow(T) = {$,), +}
  - Since we have TE’ from first two rules and E’ can be ε
  - Follow(T) = (First(E’)–{ε}) ∪ Follow(E’)
- Follow(T’) = Follow(T) = {$,), +}
- Follow(F) = {*, $,), +}
  - Follow(F) = (First(T’)–{ε}) ∪ Follow(T’)

Construct a Parse Table

Grammar
E → TE’
E’ → +TE’ | ε
T → FT’
T’ → *FT’ | ε
F → (E) | id | num

First(*) = {*}
First(F) = {(, id, num}
First(T’) = {*, ε}
First(T) {(, id, num}
First(E’) = {+, ε}
First(E) {(, id, num}

Follow(E) = {$,)}
Follow(E’) = {$,)}
Follow(T) = {$,), +}
Follow(T) = {$,), +}
Follow(T’) = {$,), +}
Follow(F) = {*, $,), +}

E → TE’: E’ → E’ –T → FT’: FT’ → ’T’ → ε: Follow(T’) = {$,), +}

	id	num	*	+	(	)	$
E	E → TE’	E → TE’			E → TE’		
E’				E’ → +TE’		E’ → ε	E’ → ε
T	T → FT’	T → FT’			T → FT’		
T’			T’ → *FT’	T’ → ε		T’ → ε	T’ → ε
F	F → id	F → num			F → (E)		

Predictive Parsing

- Now we can have a predictive parsing mechanism
  - Use a stack to keep track of the expanded form
  - Initialization
    - Put starting symbol S and $ into the stack
    - Add $ to the end of the input string
    - $ is for the recognition of the termination configuration
  - If a is at the top of the stack and a is the next input symbol then
    - Simply pop a from stack and advance on the input string
  - If A is on top of the stack and a is the next input symbol then
    - Assume that $M[A, a] = A \rightarrow \alpha$
    - Replace A by α in the stack
  - Termination
    - When only $ in the stack and in the input string
  - If A is on top of the stack and a is the next input but
    - M[A, a] = empty

Error!

Pop F from stack
Remove id from input

Pop T’ from stack
Input unchanged

+TE’: Only TE’ in stack
Remove + from input

Stack	Input	Action
E $	id + num * id $	E→TE’
T E’ $	id + num * id $	T→ FT’
F T’ E’ $	id + num * id $	F → id
T’ E’ $	+ num * id $	T’ → ε
E’ $	+ num * id $	E’ → +TE’
T E’ $	num * id $	T→ FT’
F T’ E’ $	num * id $	F → num
T’ E’ $	* id $	T’ → *FT’
F T’ E’ $	id $	F → id
T’ E’ $	$	T’ → ε
E’ $	$	E’ → ε

	id	num	*	+	(	)	$
E	E → TE’	E → TE’			E → TE’		
E’				E’ → +TE’		E’ → ε	E’ → ε
T	T → FT’	T → FT’			T → FT’		
T’			T’ → *FT’	T’ → ε		T’ → ε	T’ → ε
F	F → id	F → num			F → (E)		

Build the Parse Tree

- For each non-terminal in the stack
  - Keep a pointer to its location in the parse tree
- Initialization
  - After putting S in stack, create T(S) as the root of the tree and let S points to T(S)
- At each expansion of $X \rightarrow \alpha$
  - Create child nodes of T(X) for all terminals and nonterminals in α
  - For each non-terminals added, let it point back to its corresponding tree node (when expanding, knowing where the node is in the tree)
- Termination
  - When the parsing terminates, the tree is built

LL(1) Grammar

- The predictive parsing we had is LL(1) parsing
  - First L: scanning input from left to right
  - Second L: Leftmost derivation
  - 1: lookahead 1 input character
  - Similar to recursive descent
    - But use table to determine which production to use
    - Use stack to keep track of pending non-terminals

LL(1) Grammar

- Requirements for LL(1) grammar
  - $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots$
  - For all i, j, $i \neq j$, $First(\alpha_i) \cap First(\alpha_j) = \phi$
    - $A \rightarrow B \mid a$, $B \rightarrow ab$
    - First of $A \rightarrow B$ and $A \rightarrow a$ both has a
    - Expanding A, seeing input a, can't know which rule to use
  - If $\alpha_i = \varepsilon$, then, for all j, $i \neq j$, $First(\alpha_j) \cap Follow(A) = \phi$
    - $S \rightarrow AB$, $A \rightarrow ac \mid \varepsilon$, $B \rightarrow a$
    - First of $A \rightarrow ac$ and Follow(A) both has a
    - When seeing a while expanding A, not sure to use $A \rightarrow ac$ or $A \rightarrow \varepsilon$

More about LL Grammar

- What grammar is not LL(1)?
  - Left recursive
    - $A \rightarrow A\alpha \mid \beta$
      - First(β) $\subseteq$ First(A)
      - Two production rules of A: $A \rightarrow A\alpha$ and $A \rightarrow \beta$ have the same terminals in their "First" sets (or $A \rightarrow A\alpha$ has a super set)
  - Grammar that is not left factored
    - Two productions with the same left symbols have the same First set
    - $A \rightarrow \alpha\beta \mid \alpha\delta \Rightarrow$ both rules will get into M[A,f]
      - f is any terminal in First(α)

More about LL Grammar

	a	b	$
S	$S \to A$ $S \to B$	$S \to B$	$S \to A$
A	$A \to aaA$		$A \to \varepsilon$
B	$B \to abB$	$B \to b$	

- What grammar is not LL(1)?

  $S \to A \mid B$

  $A \to aaA \mid \varepsilon$

  $B \to abB \mid b$

  - First(A) = {a, ε}, First(B) = {a, b}, First(S) = {a, b, ε}
  - Follow(S) = {\$}, Follow(A) = {\$}, Follow(B) = {\$}

- But this grammar is LL(2)
  - If we lookahead 2 input characters, predictive parsing is possible
  - $First_2$(A) = {aa, ε}, $First_2$(B) = {ab, b\$}, $First_2$(S) = {aa, ab, b\$, ε}

	aa	ab	b$	$	ba, bb, a$
S	$S \to A$	$S \to B$	$S \to B$	$S \to A$	
A	$A \to aaA$			$A \to \varepsilon$	
B		$B \to abB$	$B \to b$		

More about LL Grammar

	a	b	$
S	$S \rightarrow AB$		$S \rightarrow AB$
A	$A \rightarrow ab$ $A \rightarrow \varepsilon$		
B	$B \rightarrow a$		

- What grammar is not LL(1)?

  $S \rightarrow AB$

  $A \rightarrow ab \mid \varepsilon$

  $B \rightarrow a$

  - First(B) = {a}, First(A) = {a, ε}, First(S) = {a, ε}
  - Follow(S) = {$}, Follow(B) = {$}, Follow(A) = {a}

- But this grammar is also LL(2)
  - $First_2$(B) = {a$}, $First_2$(A) = {ab, ε}, $First_2$(S) = {ab, a$}

	a$	ab	$	…
S	$S \rightarrow AB$	$S \rightarrow AB$		
A	$A \rightarrow \varepsilon$	$A \rightarrow ab$		
B	$B \rightarrow a$			

More about LL Grammar

LL(2) Parsing Example

$S \rightarrow AB$
$A \rightarrow abA \mid \varepsilon$
$B \rightarrow aB \mid \varepsilon$

$First_2(A) = \{ab, \varepsilon\}$
$First_2(B) = \{aa, a\$, \varepsilon\}$
$First_2(S) = \{ab, aa, a\$, \varepsilon\}$

$Follow_2(S) = \{\$\}$
$Follow_2(B) = \{\$\}$
$Follow_2(A) = \{aa, a\$, \$\}$

Input: abaaa

	a$	aa	ab	$
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow AB$
A	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$	$A \rightarrow abA$	$A \rightarrow \varepsilon$
B	$B \rightarrow aB$	$B \rightarrow aB$		$B \rightarrow \varepsilon$

Stack	Input	Action
S $	abaaa$	$S \rightarrow AB$
A B $	abaaa$	$A \rightarrow abA$
A B $	aaa$	$A \rightarrow \varepsilon$
B $	aaa$	$B \rightarrow aB$
B $	aa$	$B \rightarrow aB$
B $	a$	$B \rightarrow aB$
B $	$	$B \rightarrow \varepsilon$
$	$	

LL(k) Grammar

- LL(k) parsing
  - Allow to lookahead k input characters
  - Can extend LL(1) parsing method to LL(k) parsing
    - Build parsing table based on first k terminals – $First_k(X)$

	aaa	aa$	a$
S	S→A S→B	S→A	S→B
A	A→aaA	A→aa	
O	B→aaB		B→a

- What grammar is not LL(k)?

  S → A | B

  A → aaA | aa

  B → aaB | a

    - Even number of a's ⇒ parse with A production rule
    - Odd number of a's ⇒ parse with B production rule
    - Need to continue to lookahead till the end of the input string
    - $First_3(B)$ = {aaa, a$}, $First_3(A)$ = {aaa, aa$}, $First_3(S)$ = {aaa, aa$, a$}

LL(k) Grammar

❖ What grammar is not LL(k)?

$S \rightarrow A \mid B$

$A \rightarrow aaA \mid aa$

$B \rightarrow aaB \mid a$

❖ Can something be done? Rewrite the grammar

$S \rightarrow aaS \mid E \mid O$

$E \rightarrow aa$

$O \rightarrow a$

❑ Becomes LL(3)

	aaa	aa$	a$
S	S→aaS	S→E	S→O
E		E→aa	
O			O→a

❑ $First_3(E) = \{aa\$\}$, $First_3(O) = \{a\$\}$, $First_3(S) = \{aaa, aa\$, a\$\}$

LL(k) Grammar

- What grammar is not LL(k)?
  - Ambiguous grammars

    $S \rightarrow$ if E then S else S

    $S \rightarrow$ if E then S

LL(k) Grammar

- About LL(k) language
  - A language is LL(k) if there exists an LL(k) grammar for it
  - Check whether a grammar is LL(k)
    - If given an arbitrary k
    - Always can find the same $First_k$ substring for two X-productions
    - Then the grammar is not LL(k)
  - There are CFGs that are not LL(k)

    $S \rightarrow A \mid B$

    $A \rightarrow aAa \mid aa$

    $B \rightarrow aBb \mid ab$

    - No matter how big the k is, one can always find more than k aaa…a in the first set of $S \rightarrow A$ and $S \rightarrow B$
    - This is true for the language itself

LL(k) Grammar

- How about LL(0)?
  - Only one rule to use, no lookahead needed
  - Subsequently, only one word in the language

Top-Down Parsing -- Summary

- Top down parsing
  - Recursive descent parsing
  - Making it a predictive parsing algorithm
    - Left recursion elimination
    - Left factoring
  - LL parsing
    - First set and Follow set
    - Parse table construction
    - Parsing procedure
  - LL grammars and languages
    - LL(1) grammar
    - LL(k) grammar