CS412/CS413

Introduction to Compilers

Tim Teitelbaum

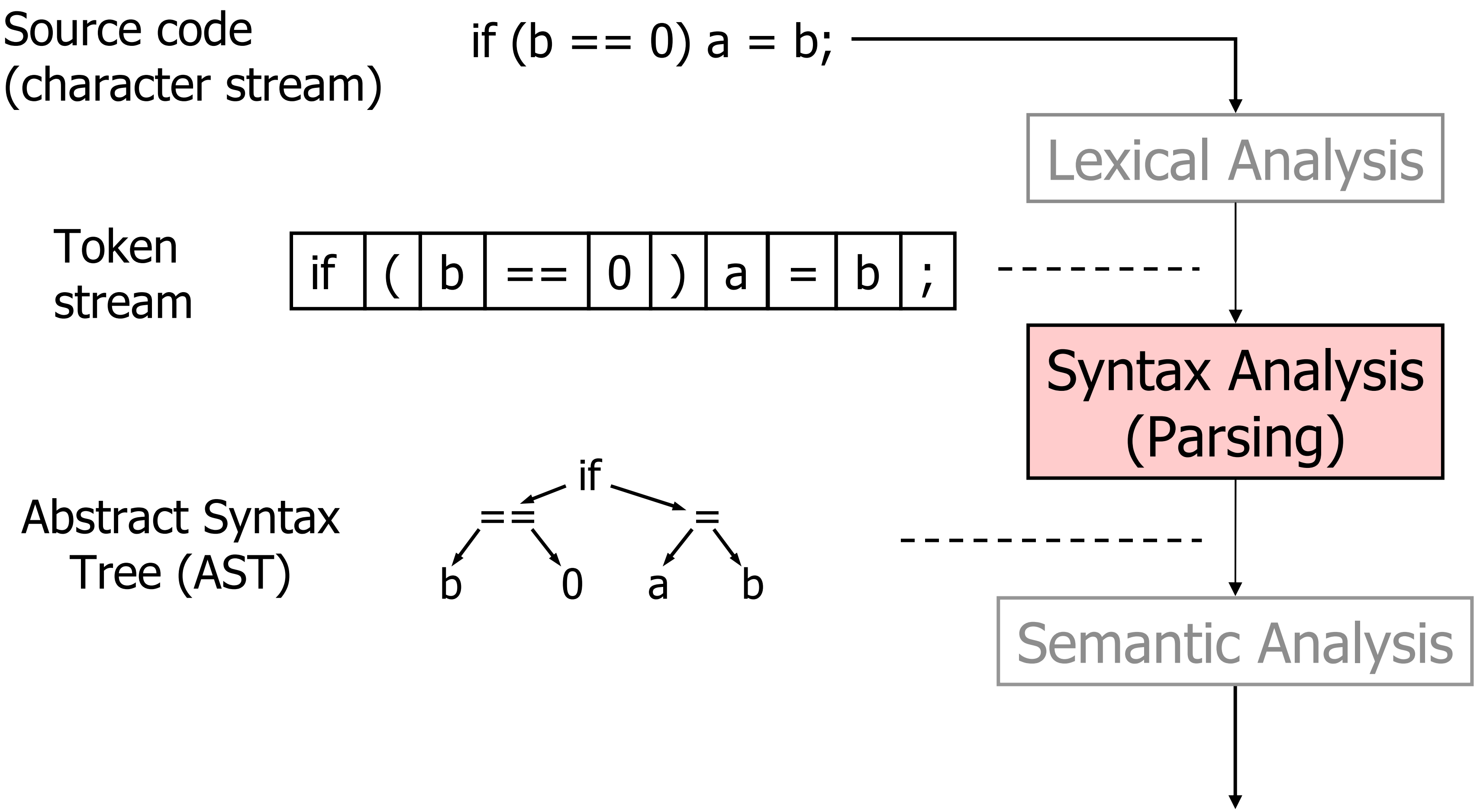
Lecture 6: Top Down Parsing

February 2, 2007

Outline

- Top-down parsing
- LL(k) grammars
- Transforming a grammar into LL form
- Recursive-descent parsing

Where We Are


Source code
(character stream)
if (b == 0) a = b;
Lexical Analysis
Token
stream
if (b == 0) a = b ;
Syntax Analysis
(Parsing)
Abstract Syntax
Tree (AST)
if
==
=
b
0
a
b
Semantic Analysis

LL(k) Parsing

- LL(k) parsing goal
  - Determine a Leftmost derivation of the input while reading the input from Left to right while looking ahead at most k input tokens
  - Beginning with the start symbol, grow the parse tree topdown in left-to-right preorder while looking ahead at most k input tokens beyond the input prefix matched by the parse tree

Sample Grammar

- Consider the grammar

  $S \rightarrow E + S \mid E$

  $E \rightarrow \textbf{num} \mid (S)$

- and the two derivations

  $S \Rightarrow E \Rightarrow (S) \Rightarrow (E) \Rightarrow (3)$

  $S \Rightarrow E+S \Rightarrow (S)+S \Rightarrow (E)+E \Rightarrow (3)+E \Rightarrow (3)+4$

- How could we decide between

  $S \Rightarrow E$

  $S \Rightarrow E+S$

  as the first derivation step based on finite number of lookahead symbols?

- We can't!
  - The sample grammar is not LL(1)
  - The sample grammar is not LL(k) for any k.

Making a grammar LL(1)

S → E+S
S → E
E → **num**
E → (S)

S → ES′
S′ → ε
S′ → +S
E → **num**
E → (S)

- Problem: can't decide which S production to apply until we see symbol after first expression
- Left-factoring: Factor common S prefix E, add new non-terminal S′ for what follows that prefix
- Also: convert left-recursion to right-recursion

Predictive Parsing

- LL(1) grammar $G = \langle V, \Sigma, S, \rightarrow \rangle$
  - For a given nonterminal, the look-ahead symbol uniquely determines the production to apply
  - Top-down parsing a.k.a. predictive parsing
  - Driven by predictive parsing table that maps $V \times (\Sigma \cup \{\varepsilon\})$ to the production to apply (or error)

Using Table

$S \to ES'$
$S' \to \varepsilon \mid +S$
$E \to$ **num** $\mid (S)$

S	(	(1+2+(3+4))+5
⇒ **E**S′	(	(1+2+(3+4))+5
⇒ (**S**)S′	1	(1+2+(3+4))+5
⇒ (**E**S′)S′	1	(1+2+(3+4))+5
⇒ (1**S′**)S′	+	(1+2+(3+4))+5
⇒ (1+**S**)S′	2	(1+2+(3+4))+5
⇒ (1+**E**S′)S′	2	(1+2+(3+4))+5
⇒ (1+2**S′**)S′	+	(1+2+(3+4))+5

	num	+	(	)	ε
S	→ ES′		→ ES′		
S′		→ **+**S		→ ε	→ ε
E	→ **num**		→ (S)		

How to Implement, Version 1

- A table-driven parser

```
void parse(nonterminal A) {
  int i;
  let A → X0X1...Xn = TABLE[A,token]
  for (i=0; i<=n; i++) {
        if (Xi in Σ)
              if (token == Xi) token = input.read();
              else throw new ParseError();
        else parse(Xi);
  }
  return;
}
```

How to Implement, Version 2

- Convert table into a recursive-descent parser

	num	+	(	)	ε
S	→ ES′		→ ES′		
S′		→ **+**S		→ ε	→ ε
E	→ **num**		→ (S)		

- Three procedures: parse_S, parse_S′, parse_E

Recursive-Descent Parser

```
void parse_S () {          lookahead token
  switch (token) {
    case num: parse_E(); parse_S′(); return;
    case '(': parse_E(); parse_S′(); return;
    default: throw new ParseError();
  }
}
```

	num	+	(	)	ε
→ S	→ ES′		→ ES′		
S′		→ +S		→ ε	→ ε
E	→ num		→ (S)		

Recursive-Descent Parser

```
void parse_S'() {
   switch (token) {
      case '+': token = input.read(); parse_S(); return;
      case ')': return;
      case EOF: return;
      default: throw new ParseError();
   }
}
```

	num	+	(	)	ε
S	→ ES'		→ ES'		
→ S'		→ +S		→ ε	→ ε
E	→ **num**		→ (S)		

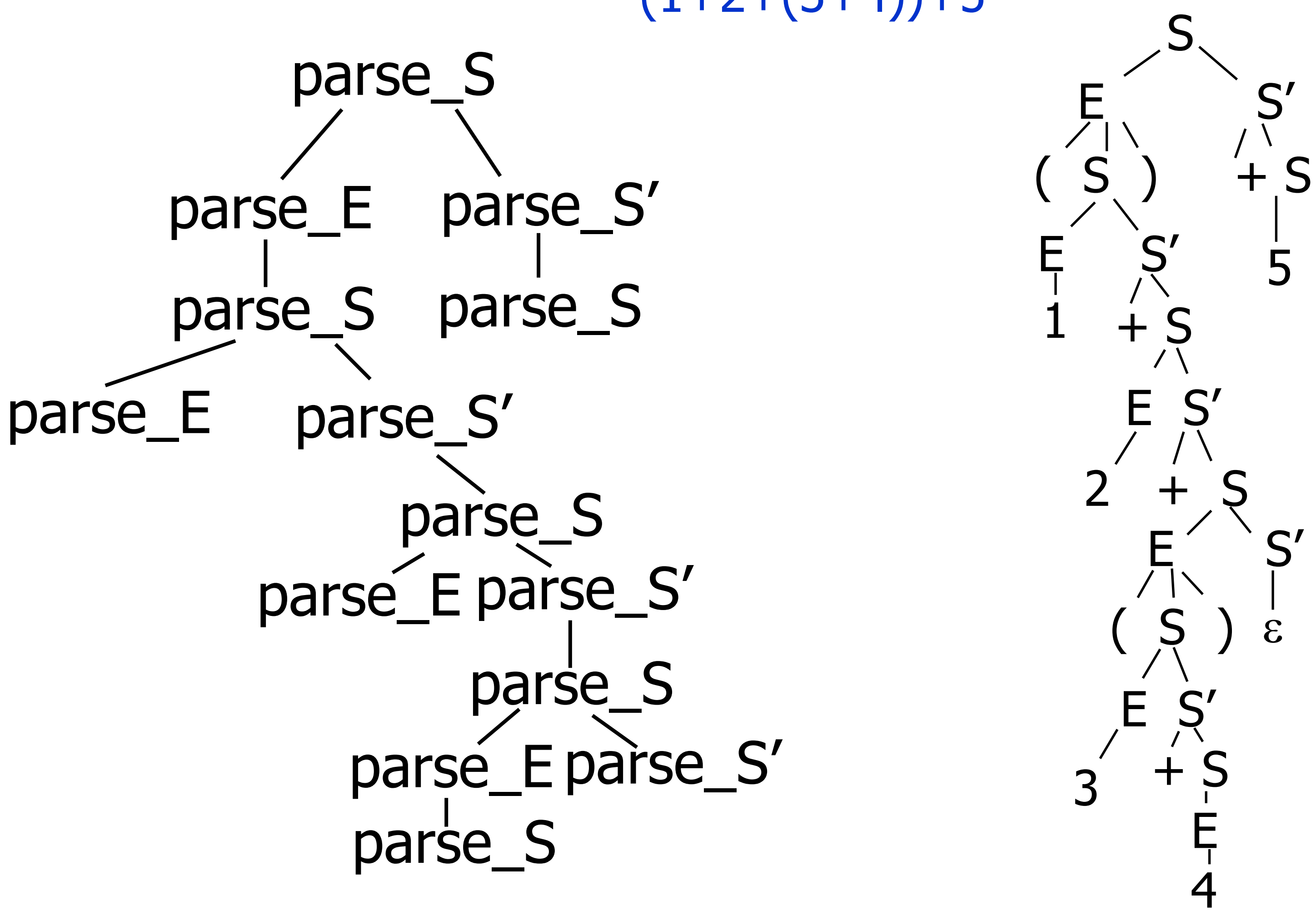
Recursive-Descent Parser

```
void parse_E() {
  switch (token) {
     case number: token = input.read(); return;
     case '(': token = input.read(); parse_S();
          if (token != ')') throw new ParseError();
          token = input.read(); return;
     default: throw new ParseError(); }
}
```

	num	+	(	)	ε
S	→ ES′		→ ES′		
S′		→ **+**S		→ ε	→ ε
→ E	→ **num**		→ (S)		

Call Tree = Parse Tree

(1+2+(3+4))+5



How to Construct Parsing Tables

- Needed: algorithm for automatically generating a predictive parsing table from a grammar

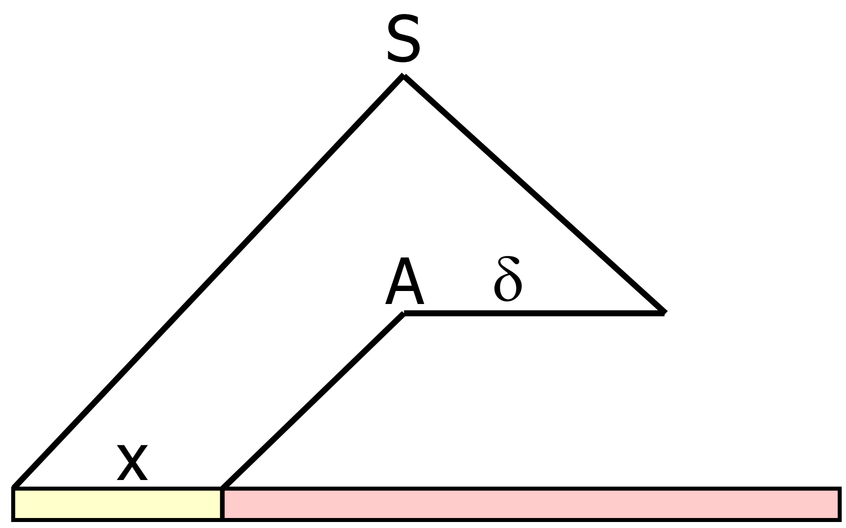
S → ES′
S′ → ε | +S
E → **num** | (S)

?

	N	+	(	)	ε
S	ES′		ES′		
S′		+S		ε	ε
E	N		(S)		

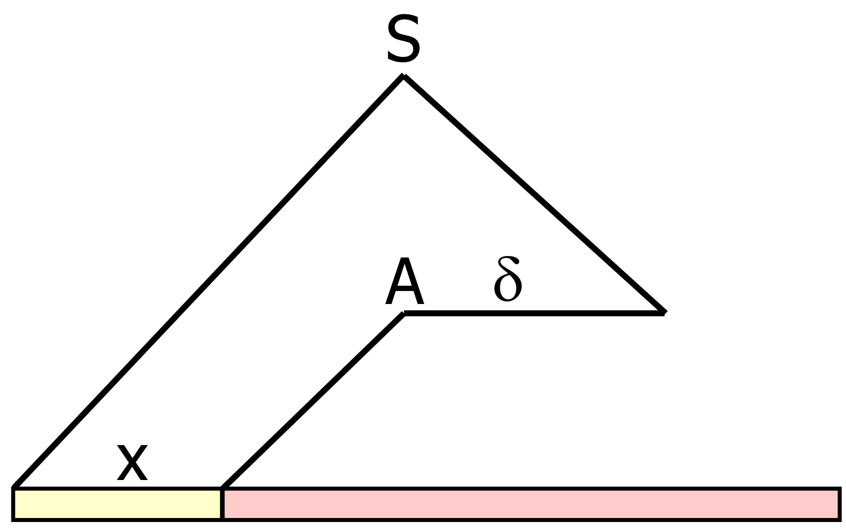
Intuition for LL(k)

- $S \Rightarrow^* xA\delta$



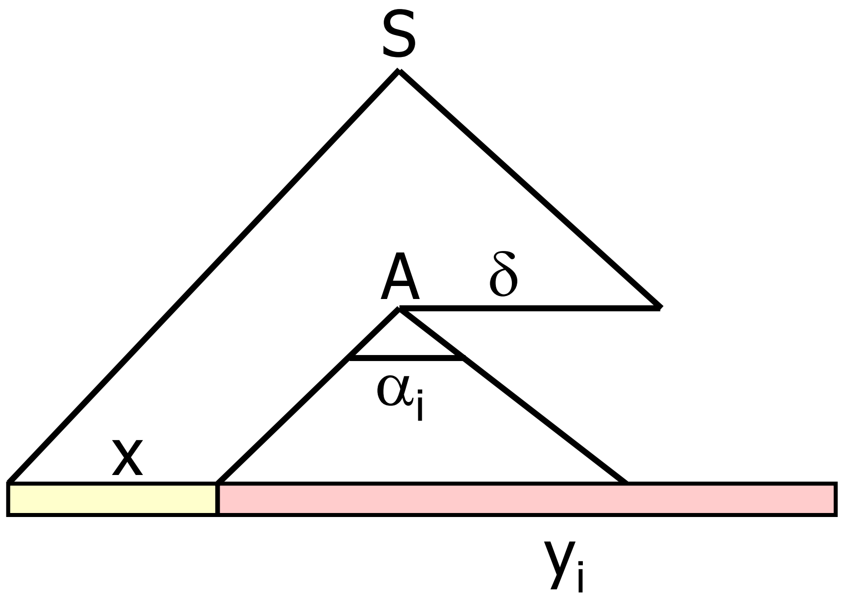
Intuition for LL(k)

- $S \Rightarrow^* xA\delta$
- $A \rightarrow \alpha_1$
- $A \rightarrow \alpha_2$



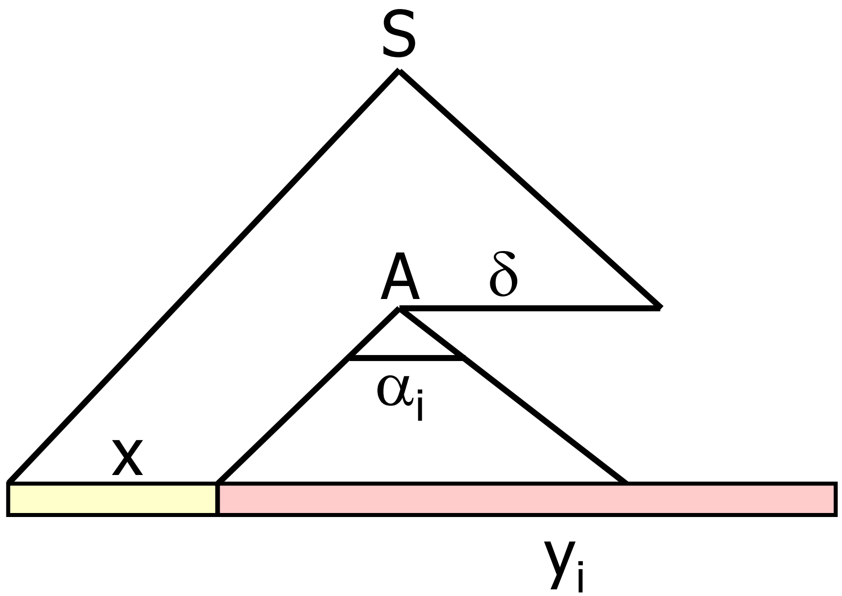
Intuition for LL(k)

- $S \Rightarrow^* xA\delta$
- $A \rightarrow \alpha_1$
- $A \rightarrow \alpha_2$
- $\alpha_1\delta \Rightarrow^* y_1$
- $\alpha_2\delta \Rightarrow^* y_2$



Intuition for LL(k)

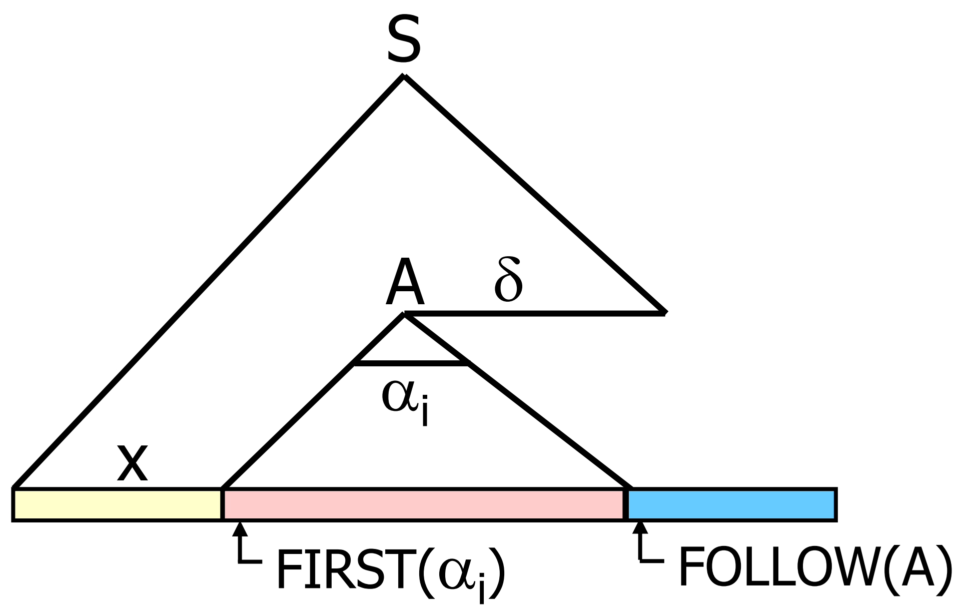
- $S \Rightarrow^* xA\delta$
- $A \rightarrow \alpha_1$
- $A \rightarrow \alpha_2$
- $\alpha_1\delta \Rightarrow^* y_1$
- $\alpha_2\delta \Rightarrow^* y_2$



- If $\alpha_1 \neq \alpha_2$ then y_1 and y_2 must differ in first k symbols

Intuition for LL(k)

- $S \Rightarrow^* xA\delta$
- $A \rightarrow \alpha_1$
- $A \rightarrow \alpha_2$
- $\alpha_1\delta \Rightarrow^* y_1$
- $\alpha_2\delta \Rightarrow^* y_2$
- If $\alpha_1 \neq \alpha_2$ then y_1 and y_2 must differ in first k symbols



Prerequisite Definitions

- Length of string w, denoted |w|, is the number of symbols in w
- k-limited prefix of string $w=a_1 \ldots a_n$, denoted k:w, is w (if $|w| \leq k$) and $a_1 \ldots a_k$ (if $|w| > k$)
- $FIRST_k(\alpha) = \{ k{:}w \mid \alpha \Rightarrow^* w \}$
- $FOLLOW_k(A) = \{ FIRST_k(\beta) \mid S \Rightarrow^* \alpha A \beta \}$

LL(k) Grammars

- A grammar is LL(k) if for every nonterminal A, if $S \Rightarrow^* xA\delta$ and $A \rightarrow \alpha_1$ and $A \rightarrow \alpha_2$ are distinct productions, then $FIRST_k(\alpha_1\delta) \cap FIRST_k(\alpha_2\delta) = \varnothing$
- A grammar is SLL(k), known as strong LL(k), if for every pair of distinct productions $A \rightarrow \alpha_1$ and $A \rightarrow \alpha_2$, $FIRST_k(\alpha_1 FOLLOW_k(A)) \cap FIRST_k(\alpha_2 FOLLOW_k(A)) = \varnothing$

LL vs SLL

- Grammars
  - LL(1) = SLL(1)
    - Proof?
  - LL(k) ≠ SLL(k), for k≥2
    - Example?

LL vs SLL

- LL(2) ≠ SLL(2)

  S → aAab | bAbb

  A → ε | a
- Language = {aab, aaab, bbb, babb}
- $FOLLOW_2(A)$ = {ab,bb}
- Is LL(2)
  - $FIRST_2$(ε ab) = {ab}; $FIRST_2$(a ab) = {aa} OK
  - $FIRST_2$(ε bb) = {bb}; $FIRST_2$(a bb) = {ab} OK
- But is not SLL(2)
  - $FIRST_2$(ε $FOLLOW_2(A)$) = {ab,bb}; BAD
  - $FIRST_2$(a $FOLLOW_2(A)$) = {aa,ab}

LL(1) Parse Table Construction

TABLE = ∅;
for each production $A \to \alpha$
  for each a in FIRST(α FOLLOW(A))
    if TABLE[A,a]==∅
      then TABLE[A,a] = $A \to \alpha$
      else fail("not LL(1)")
  if $S \Rightarrow^* wA$ **and** $\alpha \Rightarrow^* \varepsilon$ **then**
    if TABLE[A, ε]==∅
      then TABLE[A, ε] = $A \to \alpha$
      else fail("not LL(1)");

Nullability

- B is nullable if it can derive the empty string
- Algorithm

nullable = $\{ A \mid A \rightarrow \varepsilon \}$

while (nullable changed)

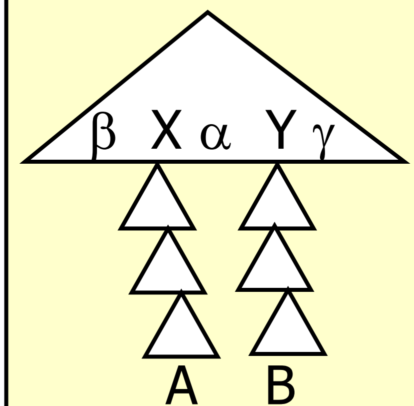
if there exists $A \rightarrow B_1 \ldots B_n$ with all B_i nullable

then nullable := nullable U {A}

- α is nullable if $\alpha = \varepsilon$ or $\alpha = A_1 \ldots A_n$ and each A_i is nullable

Computing FIRST and FOLLOW

- Let G = $\langle V, \Sigma, S, \rightarrow \rangle$ be a CFG
- Define three binary relations λ, μ, and ρ
  - $A \lambda X$ if $A \rightarrow \alpha X \beta$ and α is nullable
  - $X \mu Y$ if $A \rightarrow \beta X \alpha Y \gamma$ and α is nullable
  - $A \rho X$ if $A \rightarrow \beta X \alpha$ and α is nullable
- FIRST = $\lambda^{+} \cap (V \times \Sigma)$
- FOLLOW = $((\rho^{-1})^{*} \bullet \mu \bullet \lambda^{*}) \cap (V \times \Sigma)$



if
$X \rho^{*} A$, $X \mu Y$, $Y \lambda^{*} B$
then
A FOLLOW B

Example

- nullable
  - only S′ is nullable
- FIRST
  - FIRST(ES′) = {**num**,**(**}
  - FIRST(+S) = { **+** }
  - FIRST(**num**) = {**num**}
  - FIRST(**(S)**) = {**(**}, FIRST(S′) = {+}
- FOLLOW
  - FOLLOW(S) = {**)**}
  - FOLLOW(S′) = {**)**}
  - FOLLOW(E) = { **+**,**)**}

S → ES′
S′ → ε | **+**S
E → **num** | **(S)**

	num	**+**	**(**	**)**	ε
S	→ E S′		→ ES′		
S’		→**+**S		→ ε	→ ε
E	→ **num**		→ (S)		

Ambiguous grammars

- Construction of predictive parse table for ambiguous grammar results in conflicts

$S \rightarrow$ S+S | S*S | **num**

FIRST(S+S) = FIRST(S*S) = FIRST(**num**) = { **num** }

	num	+	*	ε
S	$\rightarrow$**num**, $\rightarrow$S+S, $\rightarrow$S*S			

Summary

- SLL(k) grammars
  - left-to-right scanning
  - leftmost derivation
  - can determine what production to apply from the next k symbols
  - Can automatically build predictive parsing tables
- Predictive parsers
  - Can be easily built for SLL(k) grammars from the parsing tables
  - Also called recursive-descent, or top-down parsers