Syntax Analysis – Part III

(Top-Down Parsers)

Yongjun Park

Hanyang University

From Last Time: Predictive Parsing

- **LL(1) grammar:**
  - For a given non-terminal, the lookahead symbol uniquely determines the production to apply
  - Top-down parsing = predictive parsing
  - Driven by predictive parsing table of
    - non-terminals x terminals → productions

From Last Time: Parsing with Table

S → ES' S' → ε | +S E → num | (S)

Partly-derived String	Lookahead	parsed part unparsed part
→ES'	(	(1+2+(3+4))+5
→(S)S'	1	(1+2+(3+4))+5
→(ES')S'	1	(1+2+(3+4))+5
→(1S')S'	+	(1+2+(3+4))+5
→(1+ES')S'	2	(1+2+(3+4))+5
→(1+2S')S'	+	(1+2+(3+4))+5

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

How to Construct Parsing Tables?

Needed: Algorithm for automatically generating a predictive parse table from a grammar

S → ES'
S' → ε | +S
E → number | (S)

⇨ ??

	num	+	(	)	$
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

Constructing Parse Tables

- **Can construct predictive parser if:**
  - For every non-terminal, every lookahead symbol can be handled by at most 1 production
- **FIRST(β) for an arbitrary string of terminals and non-terminals β is:**
  - Set of symbols that might begin the fully expanded version of β

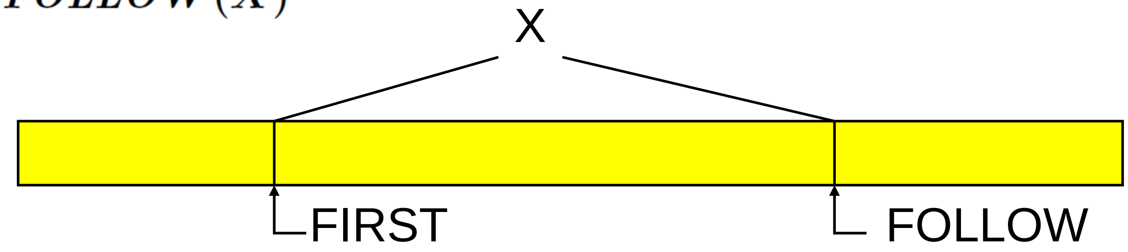
If $\alpha \Rightarrow^* c\beta$, then $c \in FIRST(\alpha)$.

If $\alpha \Rightarrow^* \epsilon$, $\epsilon \in FIRST(\alpha)$.

- **FOLLOW(X) for a non-terminal X is:**
  - Set of symbols that might follow the derivation of X in the input stream

If $S \Rightarrow^* \alpha X a \beta$, then $a \in FOLLOW(X)$.

If $S \Rightarrow^* \alpha X$, $\$ \in FOLLOW(X)$



Parse Table Entries

- Consider a production X → β
- Add → β to the X row for each symbol in FIRST(X)
- If β can derive ε (β is nullable), add → β for each symbol in FOLLOW(X)
- Grammar is LL(1) if no conflicting entries

S → ES'
S' → ε | +S
E → number | (S)

	num	+	(	)	$
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

Computing Nullable

- **X is nullable if it can derive the empty string:**
  - If it derives ε directly (X → ε)
  - If it has a production X → YZ ... where all RHS symbols (Y,Z) are nullable
- **Algorithm: assume all non-terminals are non-nullable, apply rules repeatedly until no change**

S → ES'
S' → ε | +S
E → number | (S)

Only S' is nullable

Computing FIRST

- **Determining FIRST(X)**
  1. if X is a terminal, then add X to FIRST(X)
  2. if X→ ε then add ε to FIRST(X)
  3. if X is a nonterminal and X → Y1Y2...Yk then a is in FIRST(X) if a is in FIRST(Yi) and ε is in FIRST(Yj) for j = 1...i-1 (i.e., its possible to have an empty prefix Y1 ... Yi-1
  4. if ε is in FIRST(Y1Y2...Yk) then ε is in FIRST(X)

FIRST Example

S → ES'
S' → ε | +S
E → number | (S)

Apply rule 1: FIRST(num) = {num}, FIRST(+) = {+}, etc.
Apply rule 2: FIRST(S') = {ε}
Apply rule 3: FIRST(S) = FIRST(E) = {}
FIRST(S') = FIRST('+') + {ε} = { ε, + }
FIRST(E) = FIRST(num) + FIRST('(') = {num, (}
Rule 3 again: FIRST(S) = FIRST(E) = {num, (}
FIRST(S') = {ε, + }
FIRST(E) = {num, (}

Computing FOLLOW

- **Determining FOLLOW(X)**
  1. if S is the start symbol then \$ is in FOLLOW(S)
  2. if $A \rightarrow \alpha B \beta$ then add all FIRST(β) != ε to FOLLOW(B)
  3. if $A \rightarrow \alpha B$ or ($\alpha B \beta$ and ε is in FIRST(β)) then add FOLLOW(A) to FOLLOW(B)

FOLLOW Example

S → ES'
S' → ε | +S
E → number | (S)

FIRST(S) = {num, (}
FIRST(S') = {ε, + }
FIRST(E) = { num, (}

Apply rule 1: FOL(S) = {$}
Apply rule 2: S → ES' FOL(E) += {FIRST(S') - ε} = {+}
S' → ε | +S -
E → num | (S) FOL(S) += {FIRST(')') - ε} = {$,) }
Apply rule 3: S → ES' FOL(E) += FOL(S) = {+,$,)}
(because S' is nullable)
FOL(S') += FOL(S) = {$,)}

Putting it all Together

FIRST(S) = {num, (}
FIRST(S') = {ε, + }
FIRST(E) = { num, (}

FOLLOW(S) = { $,) }
FOLLOW(S') = { $,) }
FOLLOW(E) = { +,), $ }

- Consider a production X → β
- Add → β to the X row for each symbol in FIRST(X)
- If β can derive ε (β is nullable), add → β for each symbol in FOLLOW(X)

S → ES'
S' → ε | +S
E → number | (S)

	num	+	(	)	$
S	ES'		ES'		
S'		+S		ε	ε
E	num		(S)		

Ambiguous Grammars

Construction of predictive parse table for ambiguous grammar results in conflicts in the table (ie 2 or more productions to apply in same cell)

$S \rightarrow S + S \mid S * S \mid num$

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

Class Problem

E → E + T | T
T → T * F | F
F → (E) | num | ε

1. Compute FIRST and FOLLOW sets for this G
2. Compute parse table entries

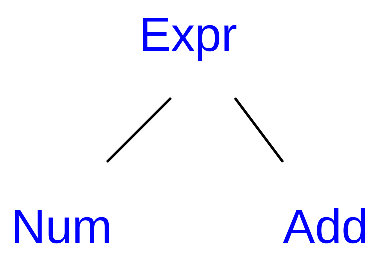
Top-Down Parsing Up to This Point

- **Now we know**
  - How to build parsing table for an LL(1) grammar (ie FIRST/FOLLOW)
  - How to construct recursive-descent parser from parsing table
  - Call tree = parse tree
- **Open question – Can we generate the AST?**

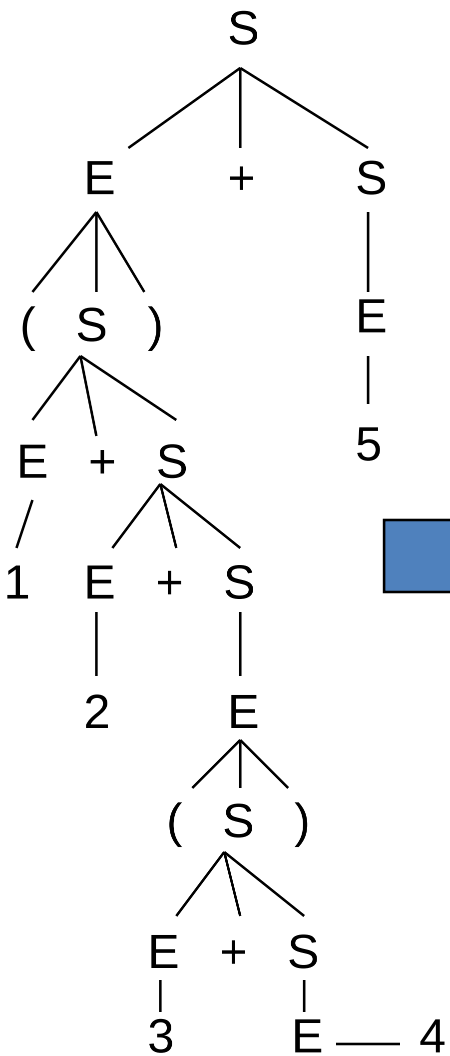
Creating the Abstract Syntax Tree

- **Some class definitions to assist with AST construction**
- **class Expr {}**
- **class Add extends Expr {**
  - Expr left, right;
  - Add(Expr L, Expr R) {
    - left = L; right = R;
  - }
- **}**
- **class Num extends Expr {**
  - int value;
  - Num(int v) {value = v;}
- **}**

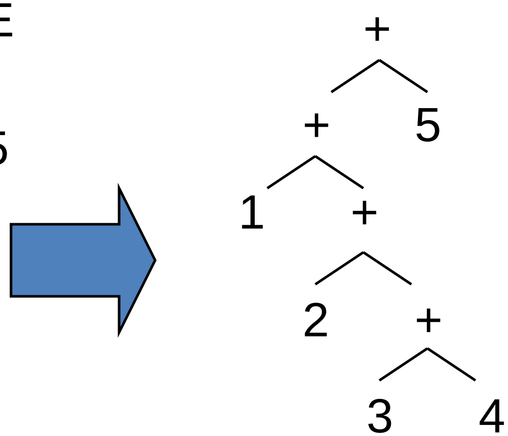
Class Hierarchy



Creating the AST



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



- We got the parse tree from the call tree
- Just add code to each parsing routine to create the appropriate nodes
- Works because parse tree and call tree are the same shape, and AST is just a compressed form of the parse tree

AST Creation: parse_E

Remember, this is lookahead token

- **Expr parse_E() {**
  - switch (token) {
    - case num: // E→ number
      - Expr result = Num(token.value);
      - token = input.read(); return result;
    - case '(': // E→ (S)
      - token = input.read();
      - Expr result = parse_S();
      - if (token != ')') ParseError();
      - token = input.read(); return result;
    - default: ParseError();
  - }
- **}**

$S \rightarrow ES'$
$S' \rightarrow \varepsilon \mid +S$
$E \rightarrow number \mid (S)$

AST Creation: parse_S

- **Expr parse_S() {**
  - switch (token) {
    - case num:
    - case '(': // S → ES'
      - Expr left = parse_E();
      - Expr right = parse_S'();
      - if (right == NULL) return left;
      - else return new Add(left,right);
    - default: ParseError();
  - }
- **}**

S → ES'
S' → ε | +S
E → number | (S)

Grammars

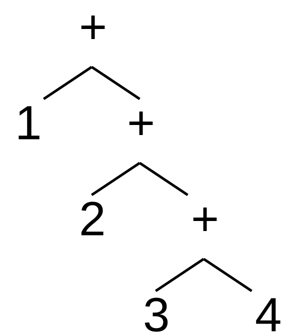
- **Have been using grammar for language "sums with parentheses" (1+2+(3+4))+5**
- **Started with simple, right-associative grammar**
  - S → E + S | E
  - E → num | (S)
- **Transformed it to an LL(1) by left factoring:**
  - S → ES'
  - S' → ε | +S
  - E → num (S)
- **What if we start with a left-associative grammar?**
  - S → S + E | E
  - E → num | (S)

Reminder: Left vs Right Associativity

Consider a simpler string on a simpler grammar: “1 + 2 + 3 + 4”

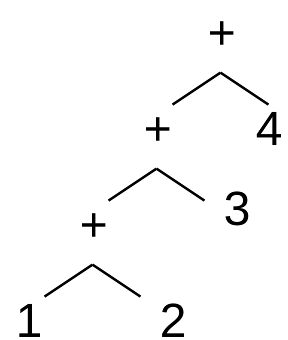
Right recursion : right associative

S → E + S
S → E
E → num



Left recursion : left associative

S → S + E
S → E
E → num



Left Recursion

S → S + E
S → E
E → num

"1 + 2 + 3 + 4"

derived string	lookahead	read/unread
S	1	1+2+3+4
S+E	1	1+2+3+4
S+E+E	1	1+2+3+4
S+E+E+E	1	1+2+3+4
E+E+E+E	1	1+2+3+4
1+E+E+E	2	1+2+3+4
1+2+E+E	3	1+2+3+4
1+2+3+E	4	1+2+3+4
1+2+3+4	$	1+2+3+4

Is this right? If not, what's the problem?

Left-Recursive Grammars

- **Left-recursive grammars don't work with top-down parsers: we don't know when to stop the recursion**
- **Left-recursive grammars are NOT LL(1)!**
  - $S \rightarrow S\alpha$
  - $S \rightarrow \beta$
- **In parse table**
  - Both productions will appear in the predictive table at row S in all the columns corresponding to FIRST(β)

Eliminate Left Recursion

- **Replace**
  - $X \rightarrow X\alpha 1 \mid \ldots \mid X\alpha m$
  - $X \rightarrow \beta 1 \mid \ldots \mid \beta n$

- **With**
  - $X \rightarrow \beta 1X' \mid \ldots \mid \beta nX'$
  - $X' \rightarrow \alpha 1X' \mid \ldots \mid \alpha mX' \mid \varepsilon$

- **See complete algorithm in Dragon book**

Class Problem

Transform the following grammar to eliminate left recursion:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Class Problem

Transform the following grammar to eliminate left recursion:

E → E + T | T

T → T * F | F

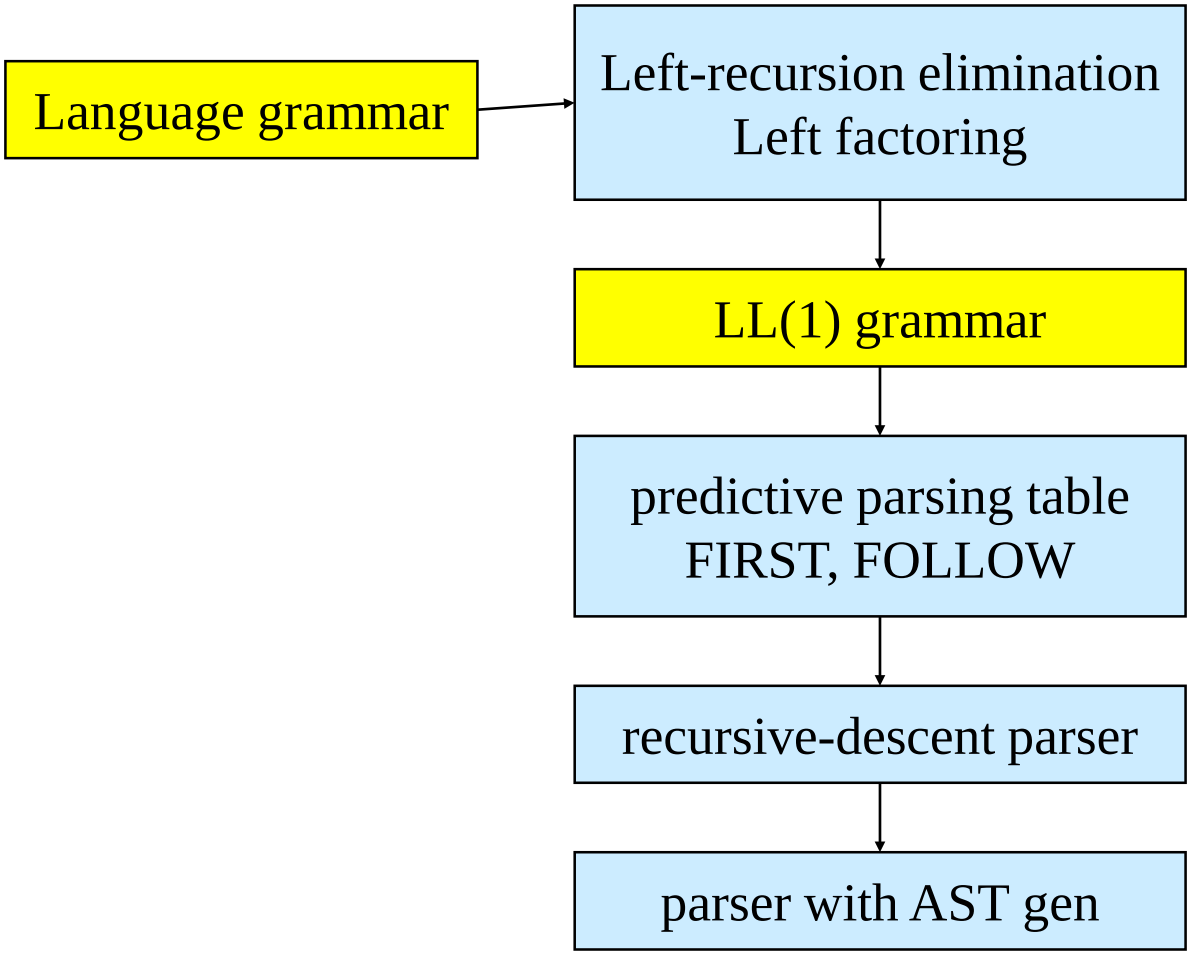
F → (E) | id

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \epsilon \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

Creating an LL(1) Grammar

- **Start with a left-recursive grammar**
    - S → S + E
    - S → E
  - and apply left-recursion elimination algorithm
    - S → ES’
    - S’ → +ES’ | ε
- **Start with a right-recursive grammar**
    - S → E + S
    - S → E
  - and apply left-factoring to eliminate common prefixes
    - S → ES’
    - S’ → +S | ε

Top-Down Parsing Summary



Next Topic: Bottom-Up Parsing

- **A more power parsing technology**
- **LR grammars – more expressive than LL**
  - Construct right-most derivation of program
  - Left-recursive grammars, virtually all programming languages are left-recursive
  - Easier to express syntax
- **Shift-reduce parsers**
  - Parsers for LR grammars
  - Automatic parser generators (yacc, bison)