# COMPILER (CSE 4120)

## (Lecture 4: Parsing 2 – Top-down Parsing )

Sungwon Jung

Mobile Computing & Data Engineering Lab
Dept. of Computer Science and Engineering
Sogang University
Seoul, Korea
Tel: +82-2-705-8930
Email : jungsung@sogang.ac.kr

Dept. of Computer Science at Sogang University

## Outline

- Eliminating ambiguity in CFGs
- Top-down parsing
- LL(1) grammars
- Transforming a grammar into LL form
- Recursive-descent parsing - parsing made simple

# Review of CFGs

- Context-free grammars can describe programming-language syntax
- Power of CFG needed to handle common PL constructs (e.g., parens)
- String is in language of a grammar if derivation from start symbol to string
- Top-down and bottom-up parsing correspond to left-most and right-most derivations
- Ambiguous grammars is a problem





# Parsing a String Top-down

$S \rightarrow S + E \mid E$
$E \rightarrow \text{number} \mid ( S )$

| Partly-derived String | Lookahead | String |
|---|---|---|
| ***S*** | ( | (1+2+(3+4))+5 |
| → ***S***+*E* | ( | (1+2+(3+4))+5 |
| → ***E***+*E* | ( | (1+2+(3+4))+5 |
| → (***S***)+*E* | 1 | (1+2+(3+4))+5 |
| → (***S***+*E*)+*E* | 1 | (1+2+(3+4))+5 |
| → (***S***+*E*+*E*)+*E* | 1 | (1+2+(3+4))+5 |
| → (***E***+*E*+*E*)+*E* | 1 | (1+2+(3+4))+5 |
| → (1+***E***+*E*)+*E* | 2 | (1+2+(3+4))+5 |
| → (1+2+***E***)+*E* | ( | (1+2+(3+4))+5 |

# Problem

$S \rightarrow S + E \mid E$
$E \rightarrow \text{number} \mid ( S )$

- Want to decide which production to apply based on next symbol
  - (1) $S \rightarrow E \rightarrow (S) \rightarrow (E) \rightarrow (1)$
  - (1)+2 $S \rightarrow S+E \rightarrow E+E \rightarrow (S)+E \rightarrow (E)+E \rightarrow (E)+E \rightarrow (1)+E \rightarrow (1)+2$
- Why is this hard?



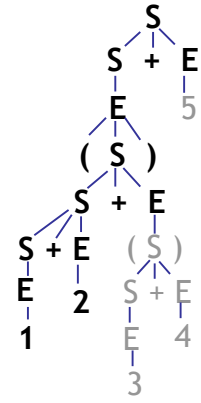


$S \rightarrow S + E \mid E$
$E \rightarrow \text{number} \mid ( S )$

# Top-down Parsing

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

- $S \rightarrow S+E \rightarrow E+E \rightarrow (S)+E \rightarrow (S+E)+E \rightarrow (S+E+E)+E \rightarrow (E+E+E)+E \rightarrow (1+E+E)+E \rightarrow (1+2+E)+E \ldots \rightarrow \ldots (1+2+(3+4))+5$
- Entire tree above a token (2) has been expanded when encountered



## Grammar is Problem

- This grammar cannot be parsed top-down with only a single look-ahead symbol
- Not **LL(1)**
- **L**eft-to-right-scanning, **L**eft-most derivation, **1** look-ahead symbol
- Is it LL(k) for some k?
- Can rewrite grammar to allow top-down parsing: create LL(1) grammar for same language





## Making an LL(1) grammar

$S \rightarrow E + S$
$S \rightarrow E$
$E \rightarrow \text{number}$
$E \rightarrow ( S )$

$S \rightarrow ES'$
$S' \rightarrow \varepsilon$
$S' \rightarrow + S$
$E \rightarrow \text{number}$
$E \rightarrow ( S )$

- **Problem**: can't decide which *S* production to apply until we see symbol after first expression
- **Left-factoring**: Factor common *S* prefix, add new non-terminal *S'* at decision point. *S'* derives (+*E*)*
- **Also:** convert left-recursion to right-recursion

# Eliminating Immediate Left-Recursion

- $A \rightarrow A\alpha_1 | \ A\alpha_2 | A\alpha_3 | \ldots | A\alpha_m | \beta_1 | \beta_2 | \beta_3 | \ldots | \beta_n$

  where no $\beta_i$ begins with an $A$
- Then replace the *A-production by*
  - $A \rightarrow \beta_1 A' | \beta_2 A' | \beta_3 A' | \ \ldots | \beta_n A'$
  - $A' \rightarrow \alpha_1 A' | \alpha_2 A' | \alpha_3 A' | \ldots | \alpha_m A' | \varepsilon$
- Example:

  $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

  $F \rightarrow (E) \mid id$

  **becomes**

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

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

  $F \rightarrow (E) \mid id$





# Eliminating General Left-Recursion

- Example:

  $S \rightarrow Aa \mid b$

  $A \rightarrow Ac \mid Sd \mid e$
- Algorithm to eliminate left-recursion from a grammar with no cycles or ε-production

1. Arrange the nonterminals of G in some order $A_1, A_2, A_3 \ldots, A_n$
2. for i = 1 to n do begin
   for j = 1 to i - 1 do
   replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma | \ \delta_2\gamma | \ \delta_3\gamma | \ \ldots | \delta_k\gamma$
   where $A_j \rightarrow \delta_1 | \ \delta_2 | \ \delta_3 | \ \ldots | \delta_k$ are all the current $A_j$-productions;
   Eliminate the immediate left-recursion among $A_i$-productions;

# Eliminating General Left-Recursion (Cont'd)

- Example:

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid e$$

- *becomes*

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid eA'$$
$$A' \rightarrow cA' \mid adA' \mid \varepsilon$$





# Parsing with new grammar

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

| | | |
|---|---|---|
| ***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+***E*** *S'*) *S'* | 2 | (1+2+(3+4))+5 |
| → (1+2 ***S'***) *S'* | + | (1+2+(3+4))+5 |
| → (1+2 + ***S***) *S'* | ( | (1+2+(3+4))+5 |
| → (1+2 + ***E*** *S'*) *S'* | ( | (1+2+(3+4))+5 |
| → (1+2 + (***S***) *S'* ) *S'* | 3 | (1+2+(3+4))+5 |
| → (1+2 + (***E*** *S'*) *S'* ) *S'* | 3 | (1+2+(3+4))+5 |
| → (1+2 + (3 ***S'***) *S'* ) *S'* | + | (1+2+(3+4))+5 |
| → (1+2 + (3 + ***E***) *S'* ) *S'* | 4 | (1+2+(3+4))+5 |

# Predictive Parsing Table

- **LL(1)** grammar:
  - for a given non-terminal, the look-ahead symbol uniquely determines the production to apply
  - top-down parsing = predictive parsing
  - Driven by ***predictive parsing table*** of
    - non-terminals × input symbols → productions





# Using Table

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

| | | |
|---|---|---|
| ***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 |

| | number | + | ( | ) | $ (EOF) |
|---|---|---|---|---|---|
| *S* | → *E S'* | | → *E S'* | | |
| *S'* | | → +*S* | | → ε | → ε |
| *E* | → number | | → ( *S* ) | | |

# How to Implement?

- Table can be converted easily into a **recursive-descent parser**

| | number | + | ( | ) | $ |
|---|---|---|---|---|---|
| *S* | → *E S'* | | → *E S'* | | |
| *S'* | | → +*S* | | → ε | → ε |
| *E* | → number | | → ( *S* ) | | |

- Three procedures: parse_S, parse_S', parse_E





# Recursive-Descent Parser

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

*lookahead token*

| | number | + | ( | ) | $ |
|---|---|---|---|---|---|
| *S* | → *ES'* | | → *ES'* | | |
| *S'* | | → +*S* | | → ε | → ε |
| *E* | → number | | → ( *S* ) | | |

# Recursive-Descent Parser (Cont'd)

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

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





# Recursive-Descent Parser (Cont'd)

```
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(); }
}
```

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

## Predictive Parsing Program

```
Repeat
 begin
  let X be the top stack symbol and a the next input symbol;
  if X is a terminal or $ then
     if X = a then pop X from the stack and remove a from the input
     else ERROR()
  else  /* X is a nonterminal */
    if M[X,a] = X→Y1 Y2 … Yk then begin
       pop X from the stack;
       push Yk Yk-1 … Y1 onto the stack, Y1 on top
    end
  else ERROR()
 end
until X = $ /*stack has emptied */
```





## How to Construct Parsing Tables ?

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

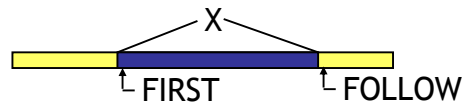
$S \rightarrow ES'$
$S' \rightarrow \varepsilon \mid +S$
$E \rightarrow \text{number} \mid (S)$


| | N | + | ( | ) | $ |
|---|---|---|---|---|---|
| S | $\rightarrow ES'$ | | $\rightarrow ES'$ | | |
| S' | | $\rightarrow +S$ | | $\rightarrow \varepsilon$ | $\rightarrow \varepsilon$ |
| E | $\rightarrow$ N | | $\rightarrow (S)$ | | |

# Constructing Parse Tables

- Can construct predictive parser if:

  For every non-terminal, every look-ahead symbol can be handled by at most one production
- FIRST(γ) for arbitrary string of terminals and non-terminals γ is:
  - set of symbols that might begin the fully expanded version of γ
- FOLLOW(X) for a non-terminal X is:
  - set of symbols that might follow the derivation of X in the input stream








# Computing nullable, FIRST

- X is nullable if it can derive the empty string:
  - it derives ε directly
  - it has a production X→ YZ... where all RHS symbols (Y, Z) are nullable
  - Algorithm: assume all non-terminals non-nullable, apply rules repeatedly until no change in status
- Determining FIRST(γ)
  - FIRST(X) ⊇ FIRST(γ) if X→ γ
  - FIRST(**a** β) = { **a** }
  - FIRST(X β) ⊇ FIRST(X)
  - FIRST(X β) ⊇ FIRST(β) if X is nullable
  - **Algorithm:** Assume FIRST(γ) = {} for all γ, apply rules repeatedly

# Computing FOLLOW

- FOLLOW($S$) $\supseteq$ { $ }
- If $X \rightarrow \alpha Y \beta$,
  FOLLOW(Y) $\supseteq$ FIRST($\beta$)
- If $X \rightarrow \alpha Y \beta$ and $\beta$ is nullable (or non-existent),
  FOLLOW(Y) $\supseteq$ FOLLOW(X)
- **Algorithm:** Assume FOLLOW(X) = { } for all X, apply rules repeatedly
- Common theme: iterative analysis. Start with initial assignment, apply rules until no change





# Applying Rules

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

- nullable
  - only $S'$ is nullable
- FIRST
  - FIRST(S) = { **number** , **(** }
  - FIRST($E$) = { **number** , **(** }
  - FIRST(S') = { + , ε}
- FOLLOW
  - FOLLOW($S$) = { **$**, **)**}
  - FOLLOW($S'$) = { ), $}
  - FOLLOW($E$) = { +, **)** ,**$**}

| | **num** | **+** | **(** | **)** | **$** |
|---|---|---|---|---|---|
| ***S*** | $\rightarrow ES'$ | | $\rightarrow ES'$ | | |
| ***S'*** | | $\rightarrow +S$ | | $\rightarrow \varepsilon$ | $\rightarrow \varepsilon$ |
| ***E*** | $\rightarrow$ num | | $\rightarrow$ ( $S$ ) | | |

# Parse Table Entries

- Consider a production X → γ
- Add → γ to the X row for each symbol in FIRST(γ)

| | N | + | ( | ) | $ |
|---|---|---|---|---|---|
| ***S*** | → *ES'* | | → *ES'* | | |
| ***S'*** | | → +*S* | | → ε | → ε |
| ***E*** | → N | | → ( *S* ) | | |

- If γ can derive ε (γ is *nullable*), add → γ for each symbol in FOLLOW(X)
- Grammar is LL(1) if no conflicts entries





# Detecting ambiguity

- Construction of predictive parse table results in *conflicts.*
  - *S* → *S* + *S* | *S* * *S* | **num**
  - FIRST(*S* + *S*) = FIRST(*S* * *S*) = FIRST(**num**) = { **num** }

| | **num** | + | * |
|---|---|---|---|
| S | →**num**, →*S* + *S*, →*S* * *S* | | |

# LL(1) Grammar

- Feature
  - input parsed from left to right
  - leftmost derivation
  - one token lookahead
- Definition
  - A grammar $G$ is LL(1) iff for each set of production $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \ldots \mid \alpha_n$
    1. FIRST($\alpha_1$), FIRST($\alpha_2$), … , FIRST($\alpha_n$) are all pairwise disjoint
    2. If $\alpha_j \Rightarrow^* \varepsilon$, then FIRST($\alpha_j$) $\cap$ FOLLOW($A$) = $\varnothing$, for all $1 \le j \le n$, $i \neq j$.

       *If G is $\varepsilon$-free, condition 1 is sufficient*





# LL(1) Grammars

- Provable facts about LL(1) grammars:
  - no left recursive grammar is LL(1)
  - no ambiguous grammar is LL(1)
- Not all grammar is LL(1)

  $S \rightarrow aS \mid a$ is not LL(1)

  since FIRST($aS$) = FIRST($a$) = {$a$}

  $S \rightarrow aS'$

  $S' \rightarrow aS' \mid \varepsilon$ accepts the same language and is LL(1)
- LL(1) grammars
  - may need to rewrite grammar (left recursion, left factoring)
  - resulting grammar larger, less maintainable
- LL($k$) grammars
  - $k$ token lookahead
  - more powerful than LL(1) grammars
  - example: $S \rightarrow ac \mid abc$ is LL(2)

# Summary

- We can build a recursive-descent parser for LL(1) grammars
  - Make parsing table from FIRST, FOLLOW
  - Translate to recursive-descent code
- Systematic approach avoids errors, detects ambiguities
- Next time: converting a grammar to LR(1) form, bottom-up parsing