Programming Paradigms

Syntax (Part 4)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2020

Overview of Syntax Analysis

[notes: raw code, scanner, seq of tokens, parser, parse tree/AST, rest of compiler or analysis]

Overview

- **Specifying syntax**
  - Regular expressions
  - Context-free grammars
- **Scanning**
- **Parsing** 
  - Top-down parsing
  - Bottom-up parsing

Top-down vs. Bottom-up Parsing

Two ways to construct a parse tree

- Top-down
  - Starting from root node, expand non-terminals until reaching terminals
  - If multiple rules apply: Predict which production rule to use
- Bottom-up
  - Combine incoming tokens into subtrees
  - Whenever subtrees can be further combined, add a parent node

Example: Grammar

P $\rightarrow$ **`begin` SS `end`**

SS $\rightarrow$ **S`;` SS**

SS $\rightarrow \epsilon$

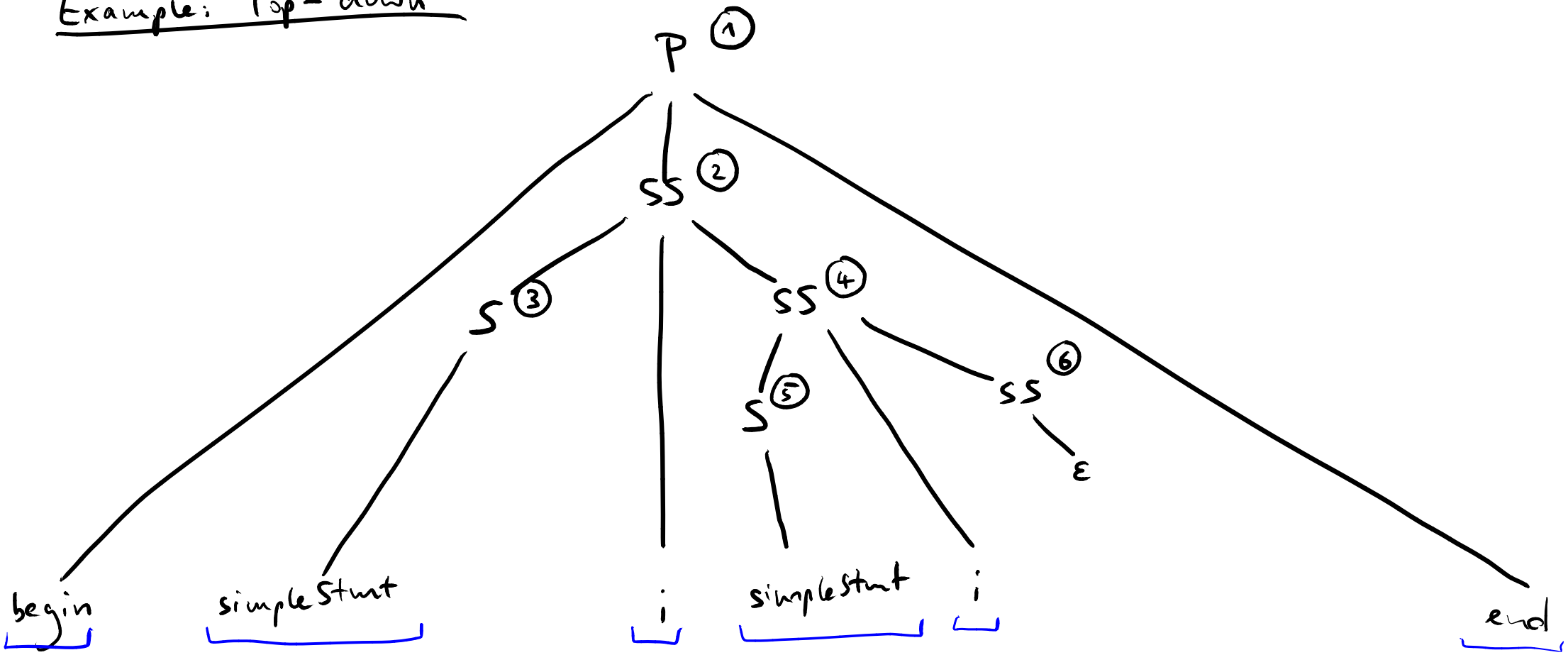
S $\rightarrow$ **`simplestmt`**

S $\rightarrow$ **`begin` SS `end`**

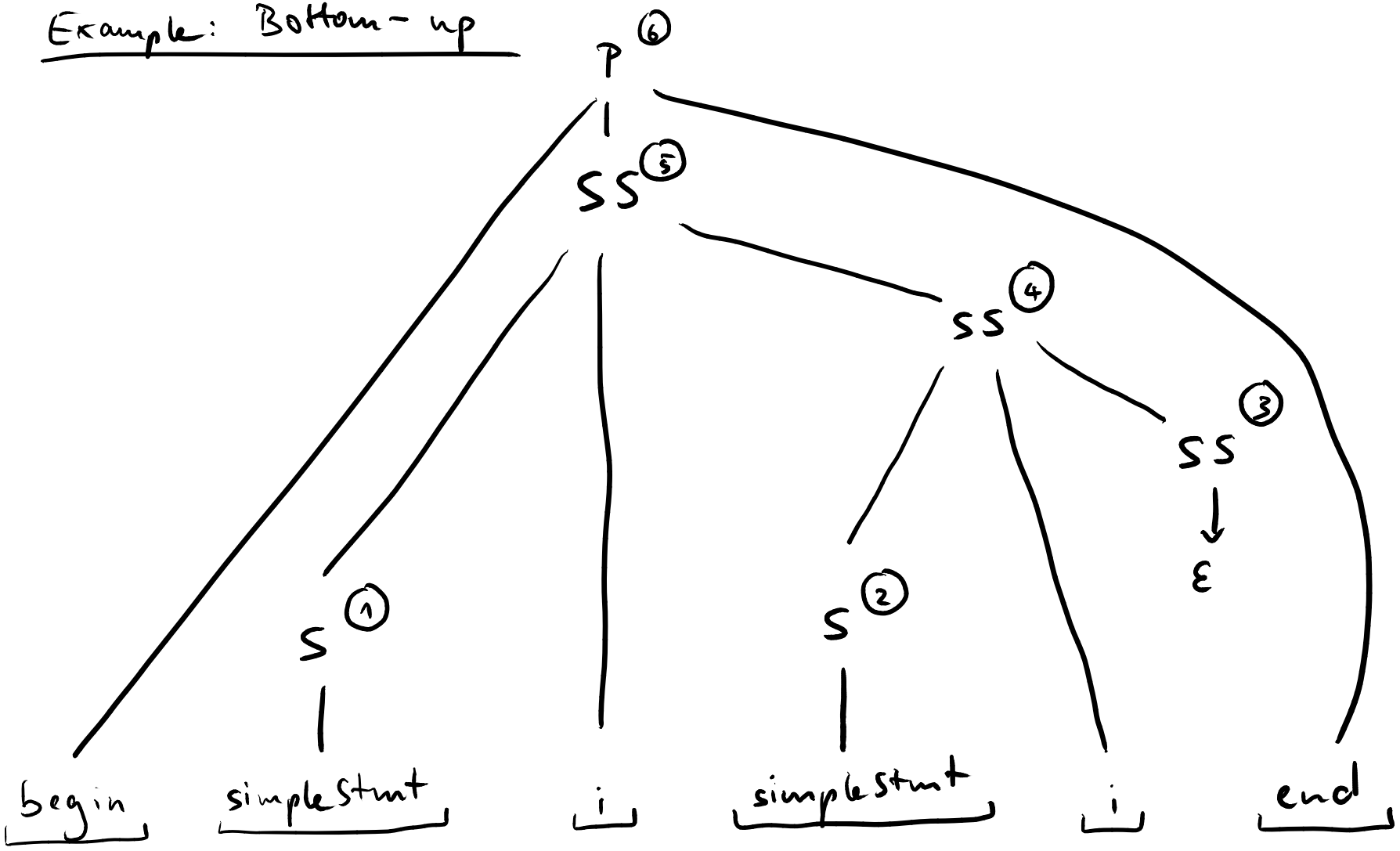
Example program:

```
begin simplestmt; simplestmt; end
```

Example: Top-down



Example: Bottom-up



Classes of Parsing Algorithms

	LL(k) parsers	LR(k) parsers
Parse tree construction	Top-down	Bottom-up
Scanning	Left-to-right	Left-to-right
Derivations	Left-most	Right-most
Algorithm	Predictive	Shift-reduce

Top-down Parsing

- **LL(k) parsers**
  - Left-to-right scanning, Left-most derivation, k tokens look-ahead
- **Two approaches**
  - Recursive descent parser
    - Easy to manually write (for simple languages)
  - Table-driven LL parser
    - Driver program and automatically generated table

General Algorithm

- **Initially, current non-terminal is start symbol**
- **Loop until no more input**
  - Given next k tokens and current non-terminal, choose a rule R
  - For each element X in rule R from left to right
    - If X is a non-terminal, we will need to expand X
    - If X is a terminal, see if next token matches X, and if so, move on to next token

Recursive Descent Parser

- **One function for each non-terminal N**
  - Mimics productions with N on left-hand side
  - Chooses production based on next k tokens
  - For non-terminals on right-hand side, call their function
  - For terminals on right-hand side, call *match* function
- ***match* function: Consumes input token (if expected) or raises error**

Example

Grammar:

S $\rightarrow$ `a` **B**

S $\rightarrow$ `b` **C**

B $\rightarrow$ `b b` **C**

C $\rightarrow$ `c c`

```
S() {
  if (inputToken == a)
    match(a); B();
  else if (inputToken == b)
    match(b); C();
  else error();
}
B() {
  if (inputToken == b)
    match(b); match(b); C();
  else error()
}
C() {
  if (inputToken == c)
    match(c); match(c);
  else error()
}
```

Example : Parsing "abbcc"

Step	Remaining input	Actions
1	abbcc	call S() from main() call match(a) call B()
2	bbcc	call match(b) call match(b) call C()
3	cc	call match(c) call match(c)