Programming Paradigms

Syntax (Part 2)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2023

From DFA to Scanner

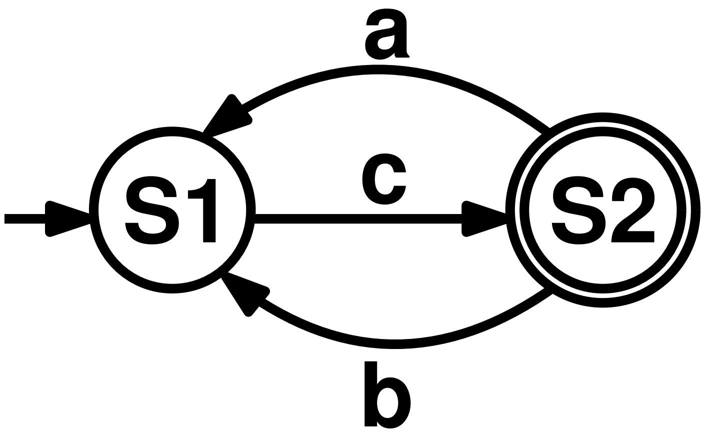
Two popular options

- Implement the DFA using switch statements
  - Mostly in hand-written scanners
- Table-based scanners
  - Table represents states and transitions
  - Driver program indexes the table
  - Mostly in auto-generated scanners

Switch Statement Style

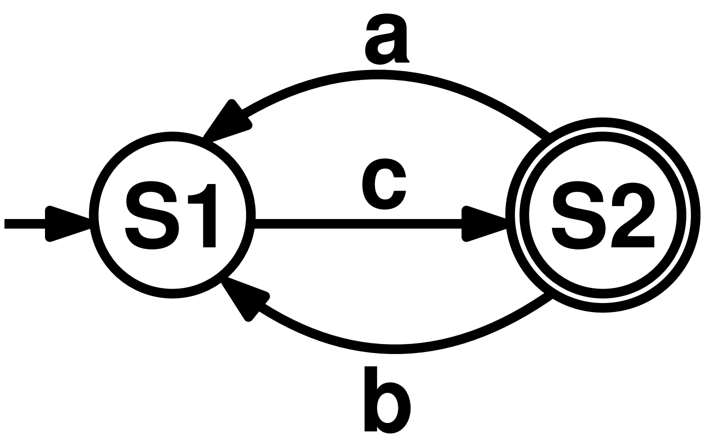
```
state = S1
```

Starting state: S1



Switch Statement Style

```
state = S1
token = ""
loop:
```

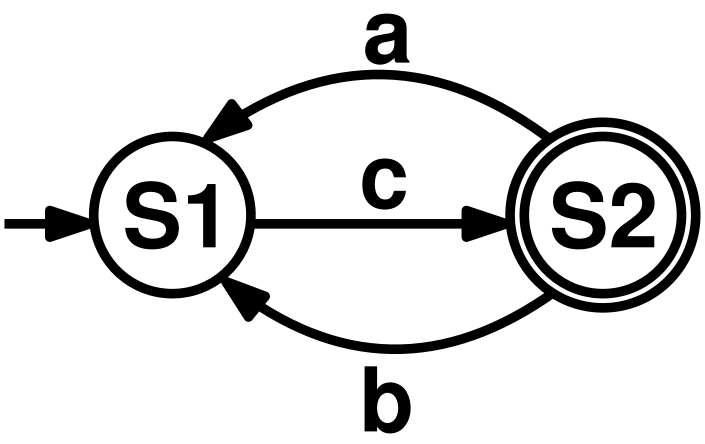


Loop reads one character at a time and builds the token

```
  token = token + in_char
  read next in_char
```

Switch Statement Style

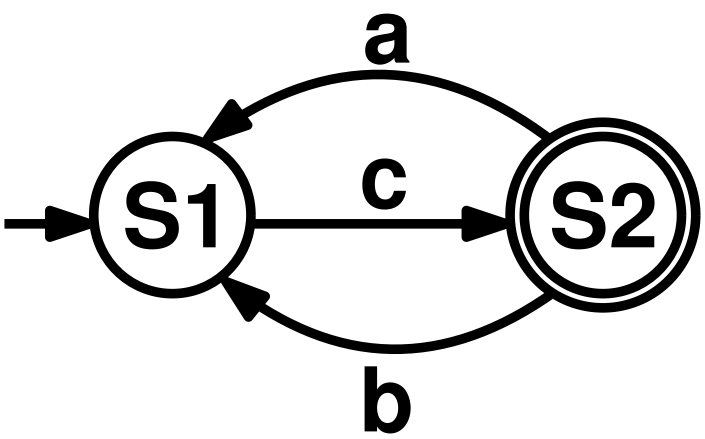
```
state = S1
token = ""
loop:
  switch state:
    case S1:



    case S2:



  token = token + in_char
  read next in_char
```



Switch statement that handles the current state

Switch Statement Style

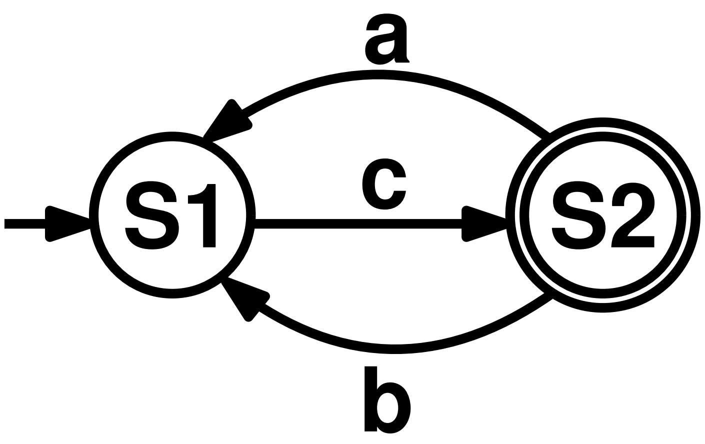
```
state = S1
token = ""
loop:
  switch state:
    case S1:
      switch in_char:
        case 'c': state = S2
        else error
    case S2:
      switch in_char:
        case 'a': state = S1
        case 'b': state = S1
        case ' ': return
        else error
  token = token + in_char
  read next in_char
```



Switch statements to handle the current character

Switch Statement Style

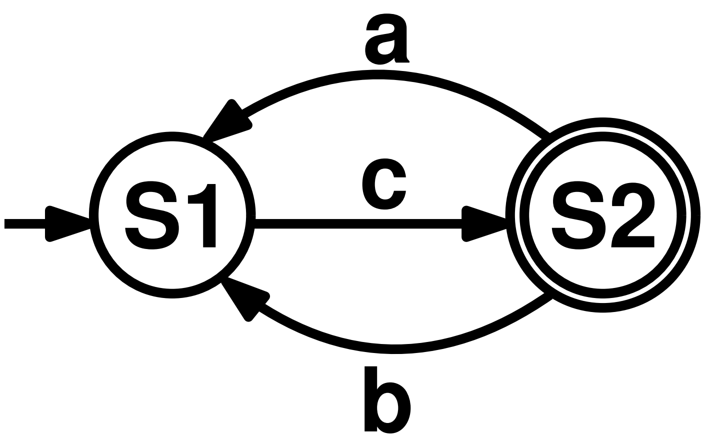
```
state = S1
token = ""
loop:
  switch state:
    case S1:
      switch in_char:
        case 'c': state = S2
        else error
    case S2:
      switch in_char:
        case 'a': state = S1
        case 'b': state = S1
        case ' ': return
        else error
  token = token + in_char
  read next in_char
```



Move to next state if character accepted

Switch Statement Style

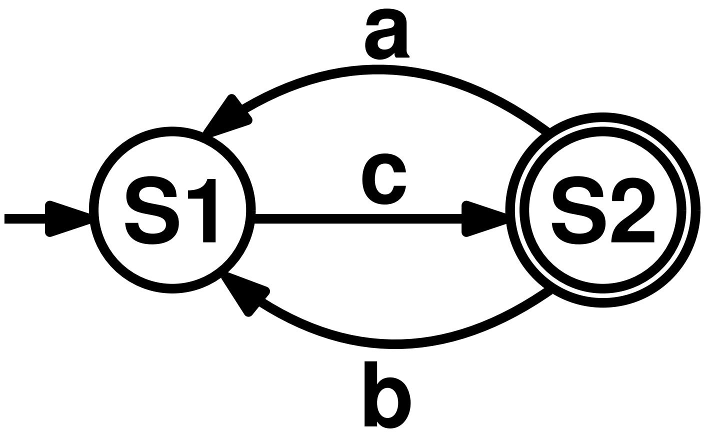
```
state = S1
token = ""
loop:
  switch state:
    case S1:
      switch in_char:
        case 'c': state = S2
        else error
    case S2:
      switch in_char:
        case 'a': state = S1
        case 'b': state = S1
        case ' ': return
        else error
  token = token + in_char
  read next in_char
```



Return the token when a space occurs

Switch Statement Style

```
state = S1
token = ""
loop:
  switch state:
    case S1:
      switch in_char:
        case 'c' : state = S2
        else error
    case S2:
      switch in_char:
        case 'a' : state = S1
        case 'b' : state = S1
        case ' ' : return
        else error
  token = token + in_char
  read next in_char
```

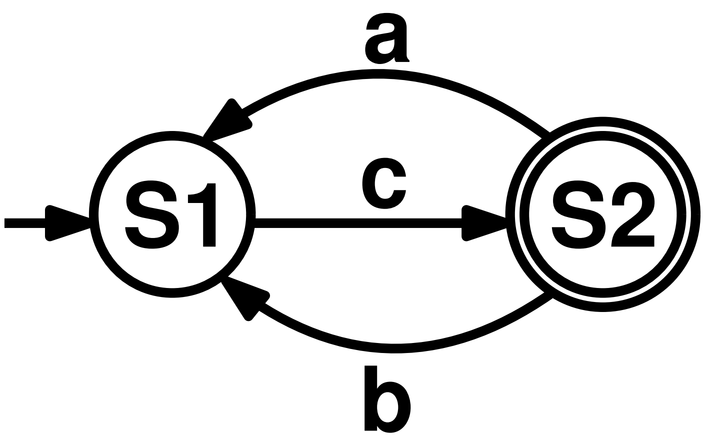


Raise an error for any illegal character

Table-based Scanning

Transition table indexed by state and input:

State	’a’	’b’	’c’	Return
S1	-	-	S2	-
S2	S1	S1	-	token

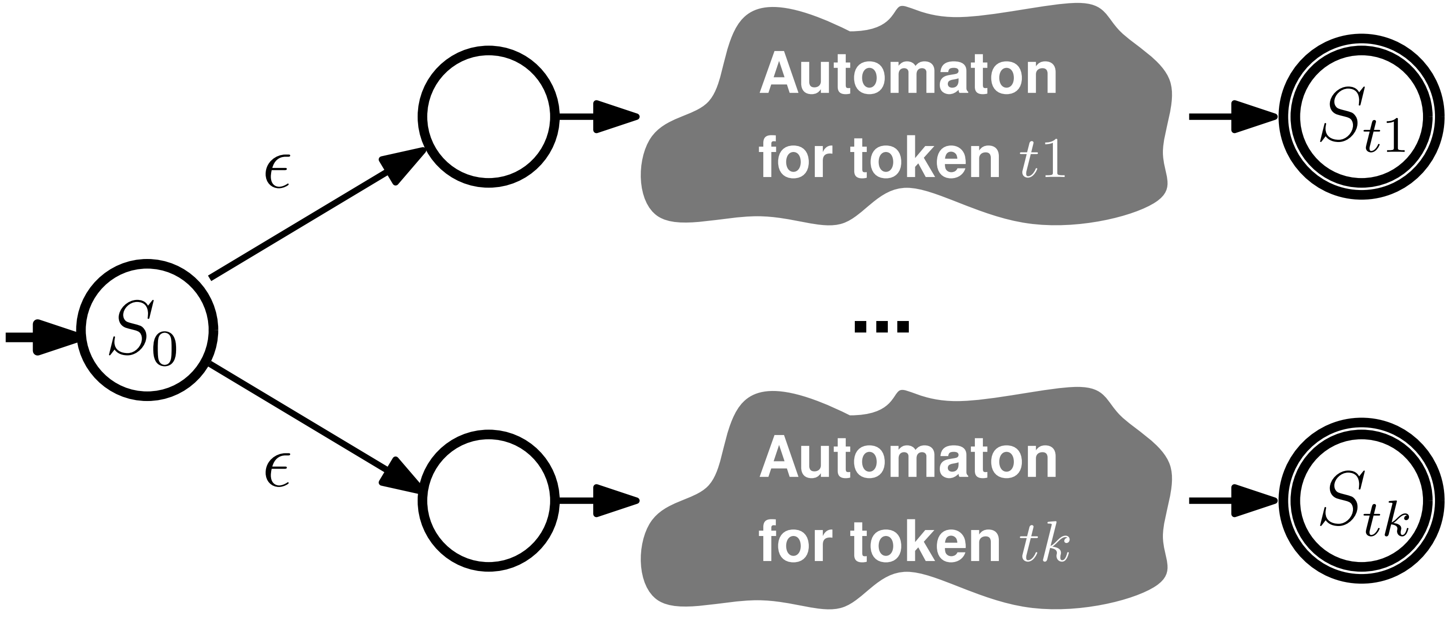


Driver program

- moves to a new state,
- returns a token, or
- raises an error

Recognizing Multiple Tokens

- So far: Recognize one kind of token
- To recognize **multiple kinds of tokens**
  - □ Merge multiple token-level automata into one



  - □ Apply NFA-DFA transformation afterwards

Longest Possible Token Rule

- **What if one token is a prefix of another?**
  - Number `3.1` vs. number `3.141`
- **Accept the longest possible token**
  - `3.141` for the above example
- **How to decide whether token has ended?**
  - Scanner looks ahead (at least one character)

Quiz: Automata and Scanners

Which of these statements is true?

- A parser produces a syntax tree.
- A parser produces a sequence of tokens.
- DFAs allow for more efficient scanning than NFAs.
- A scanner for Python will turn “ifWhile” into two tokens “if” and “while”.

Quiz: Automata and Scanners

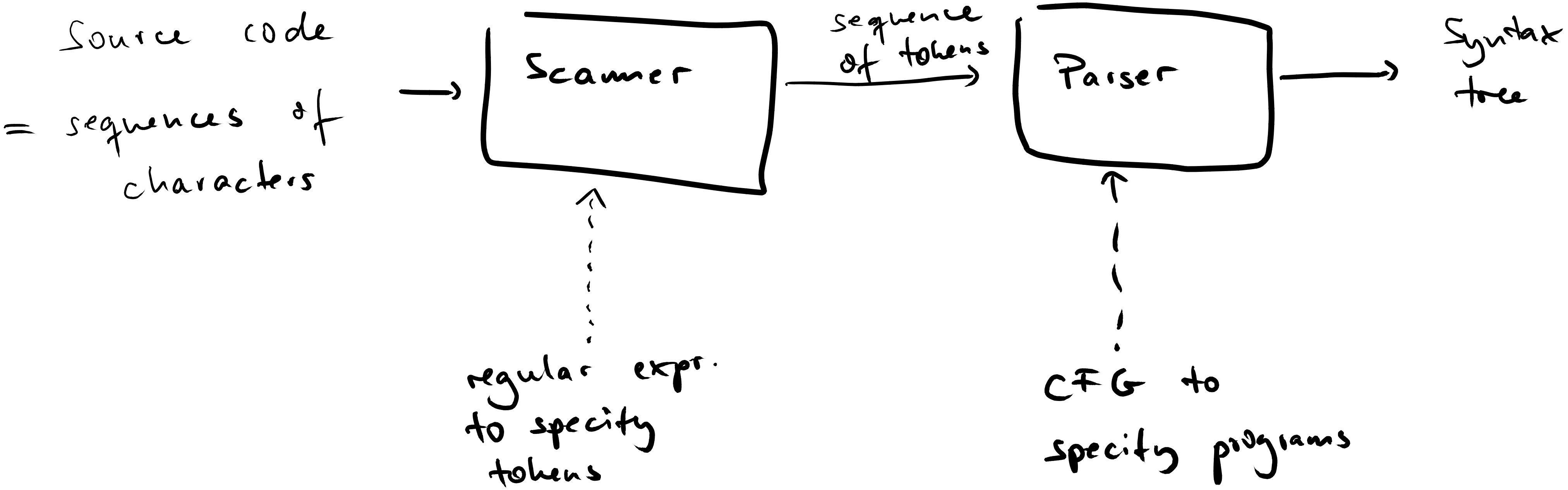
Which of these statements is true?

- A parser produces a syntax tree.
- ~~A parser produces a sequence of tokens.~~
- DFAs allow for more efficient scanning than NFAs.
- ~~A scanner for Python will turn “ifWhile” into two tokens “if” and “while”.~~

Overview

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

Big Picture


Source code
= sequences of characters
Scanner
sequence of tokens
Parser
Syntax tree
regular expr.
to specify
tokens
CFG to
specify programs

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 → `begin` SS `end`

SS → S`;` SS

SS → ϵ

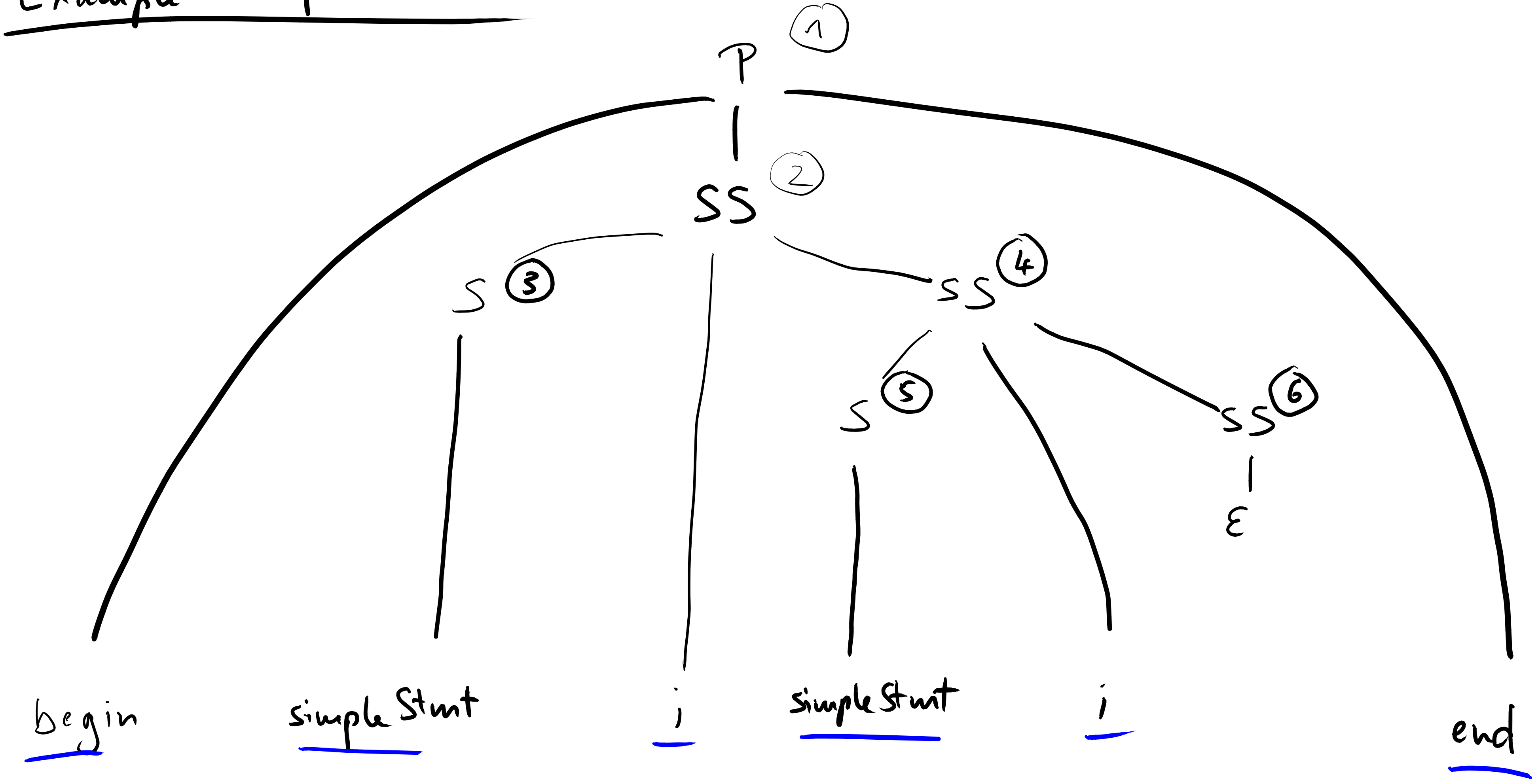
S → `simplestmt`

S → `begin` SS `end`

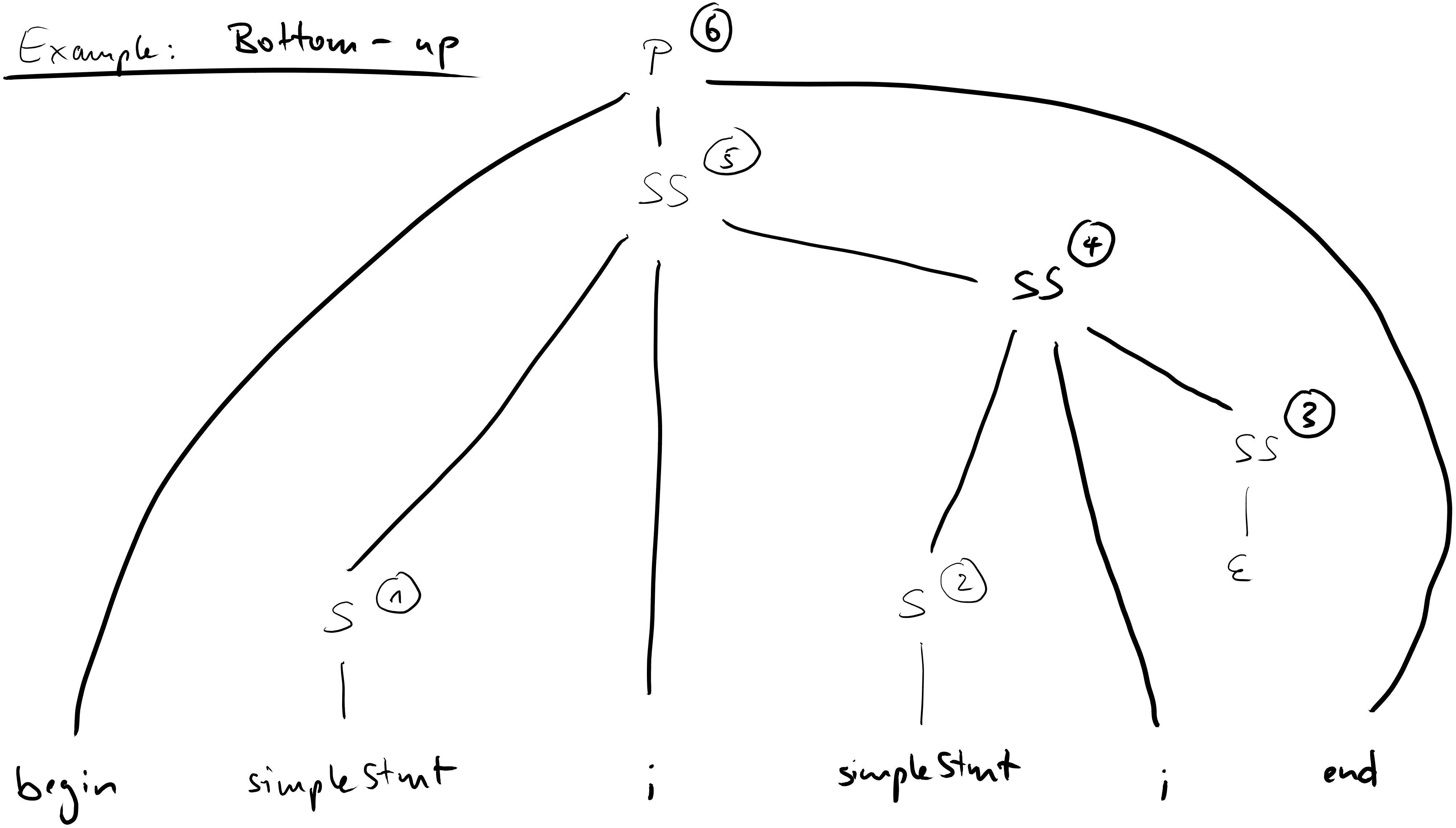
Example program:

```
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

Overview

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

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 → a B

S → b C

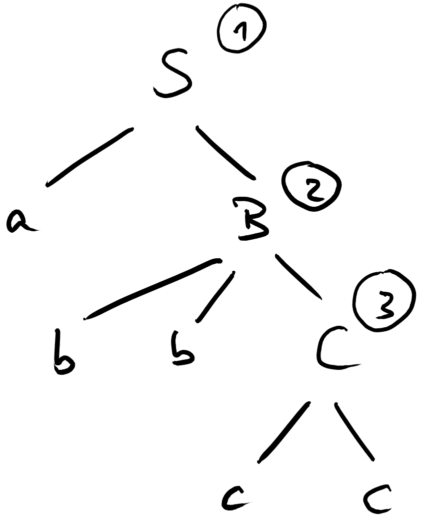
B → b b C

C → 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)



Generating a Top-Down Parser

- **To generate an LL(k) parser, need to predict which rule to apply**
- **Compute PREDICT sets for all productions, based on two helpers**
  - □ FIRST(N): What terminals come first when expanding non-terminal N?
  - □ FOLLOW(N): What terminals follow after non-terminal N?

FIRST Sets

FIRST(A): Set of all terminals that can begin a derivation starting with A

Example:

$\mathsf{S} \rightarrow \texttt{simple} \mid \texttt{begin}\ \mathsf{S}\ \texttt{end}$

$\mathsf{FIRST(S)} = \{\ \texttt{simple}, \texttt{begin}\ \}$

Computing FIRST sets

$$\text{FIRST}(\varepsilon) = \{\varepsilon\}$$

$$\text{FIRST}(A\alpha) = \begin{cases} \{A\} & \text{if } A \text{ is a terminal} \\ (\text{FIRST}(A) \setminus \{\varepsilon\}) \cup \text{FIRST}(\alpha) & \text{if } A \Rightarrow^* \varepsilon \\ \text{FIRST}(A) & \text{otherwise} \end{cases}$$

For a given grammar:

Apply above recursively until all FIRST sets remain constant

Example 1

$S \to aSe$

$S \to B$

$B \to bBe$

$B \to C$

$C \to cCc$

$C \to d$

$FIRST(S) = \{a, b, c, d\}$

$FIRST(B) = \{b, c, d\}$

$FIRST(C) = \{c, d\}$

Example 2

$P \rightarrow i \mid c \mid nTS$

$Q \rightarrow P \mid aS \mid dScST$

$R \rightarrow b \mid \varepsilon$

$S \rightarrow e \mid Rn \mid \varepsilon$

$T \rightarrow \underline{\underline{R}}Sq$

FIRST $(P) = \{i, c, n\}$

FIRST $(Q) = \{i, c, n, a, d\}$

FIRST $(R) = \{b, \varepsilon\}$

FIRST $(S) = \{e, \varepsilon, b, n\}$

FIRST $(T) = \{b, e, n, q\}$

Quiz: FIRST Sets

S $\rightarrow$ a X Y

X $\rightarrow$ b $|$ ϵ

Y $\rightarrow$ X c $|$ c

Compute the FIRST sets of all non-terminals. What is the sum of the sizes of these sets?

Quiz:

$\text{FIRST}(S) = \{a\}$

$\text{FIRST}(X) = \{b, \varepsilon\}$

$\text{FIRST}(Y) = \{b, c\}$

$$\Sigma = 5$$

FOLLOW Sets

FOLLOW(A): Set of all terminals that may follow A in some derivation

- Including special symbol EOF for “end of file”
- Never includes ϵ

Example:

S → `a` B `c`	**FOLLOW(S) = { EOF }**
B → d	**FOLLOW(B) = { c }**

Computing FOLLOW Sets

To compute FOLLOW(A), apply these rules until all FOLLOW sets constant

- If A is start symbol, put EOF in FOLLOW(A)
- Productions of the form B $\rightarrow$ α A β:

  Add FIRST(β) - { ϵ } to FOLLOW(A)
- Productions of the form

  B $\rightarrow$ α A, or

  B $\rightarrow$ α A β where $\beta \Rightarrow^* \epsilon$:

  Add FOLLOW(B) to FOLLOW(A)