

# Programming Paradigms

## Syntax (Part 2)


**Prof. Dr. Michael Pradel**

**Software Lab, University of Stuttgart**

**Summer 2022**

# Overview

---

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

# 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

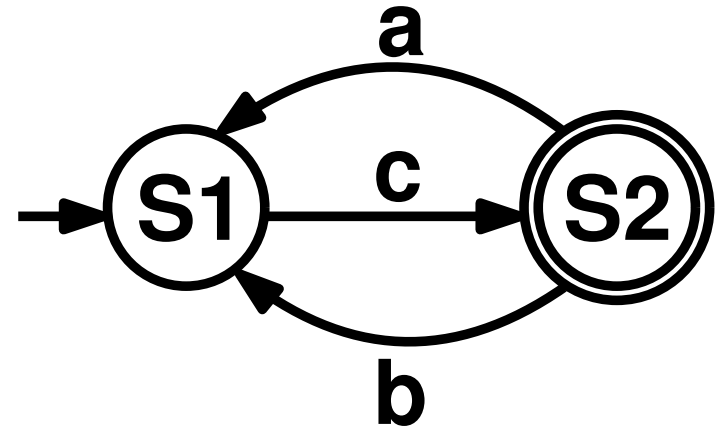
# Switch Statement Style

---

state = S1



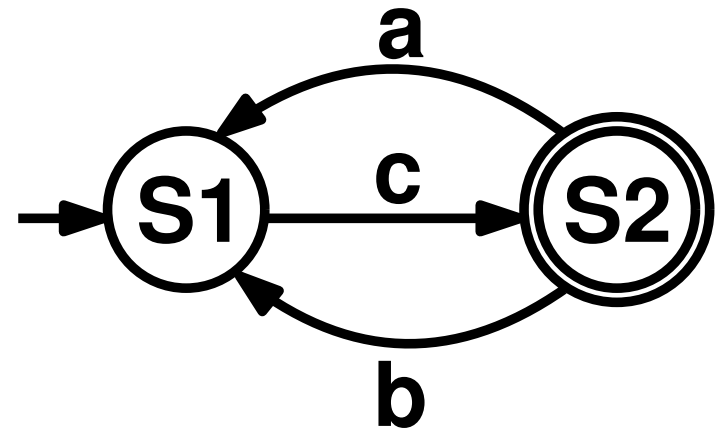
**Starting  
state: S1**



# Switch Statement Style

---

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



**Loop reads one  
character at a time  
and builds the  
token**

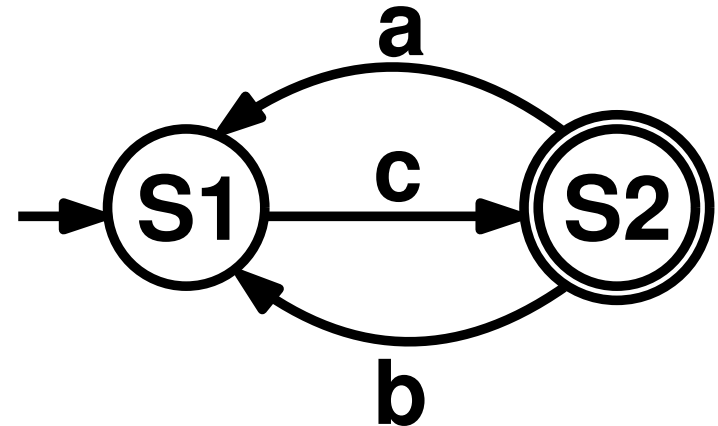
```
token = token + in_char  
read next in_char
```

# Switch Statement Style

---

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

```
    case S2:
```

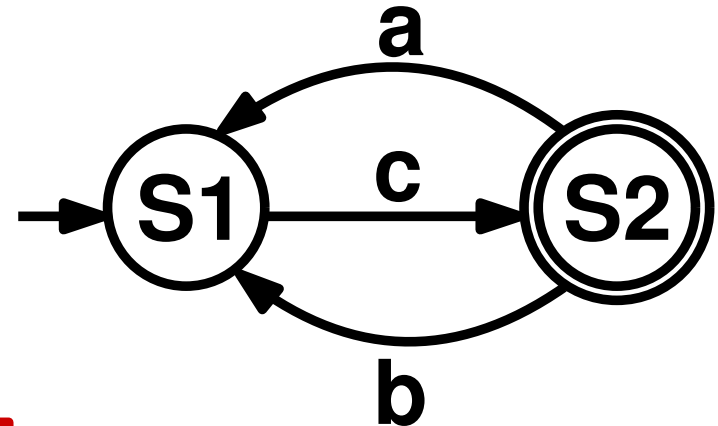


**Switch statement  
that handles the  
current state**

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

# Switch Statement Style

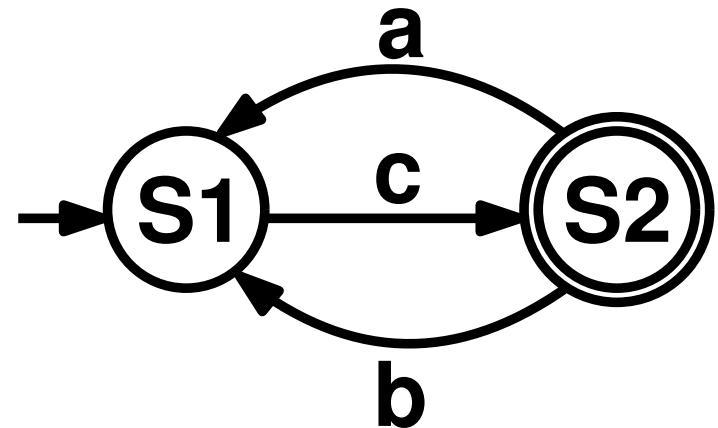
```
state = S1
token = ""
loop:
  switch state:
    case S1:
      switch in_char:
        case 'c': state = S2
        else error
    case S2:
      switch in_char:
        case 'a': state = S1
        case 'b': state = S1
        case ' ': return
        else error
  token = token + in_char
  read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

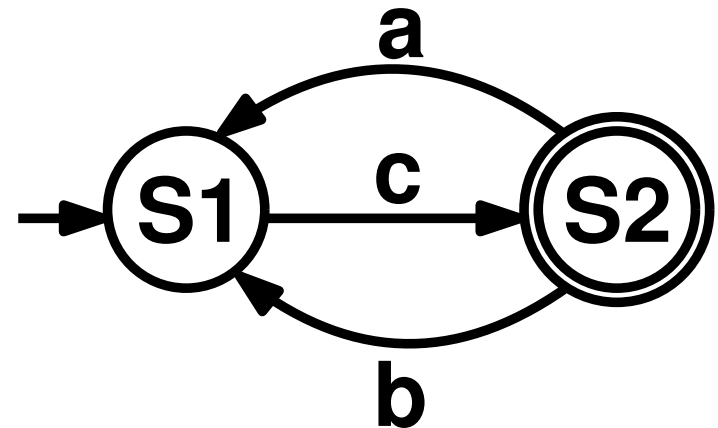


**Move to next  
state if  
character  
accepted**

# Switch Statement Style

---

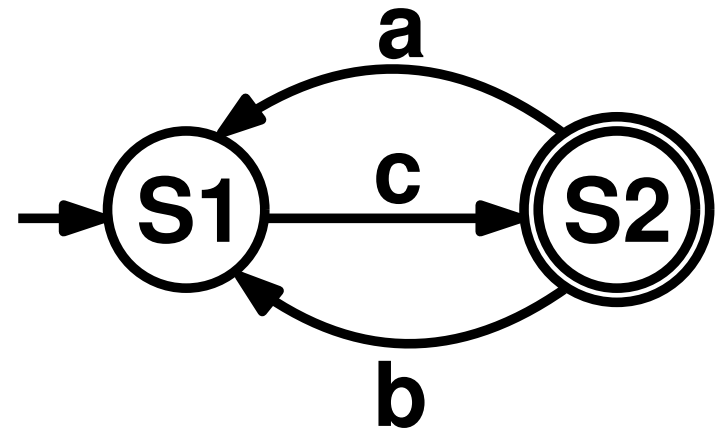
```
state = S1
token = ""
loop:
  switch state:
    case S1:
      switch in_char:
        case 'c' : state = S2
        else error
    case S2:
      switch in_char:
        case 'a' : state = S1
        case 'b' : state = S1
        case ' ' : return
        else error
  token = token + in_char
  read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

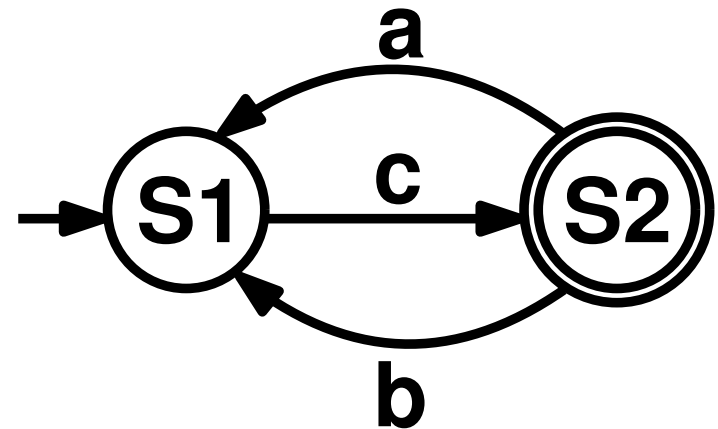
---

**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



# 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 scanner produces a syntax tree.
- A scanner produces a sequence of tokens.
- NFAs allow for more efficient scanning than DFAs.
- A scanner for Java will turn “ifClass” into two tokens “if” and “Class”.

# Quiz: Automata and Scanners


---

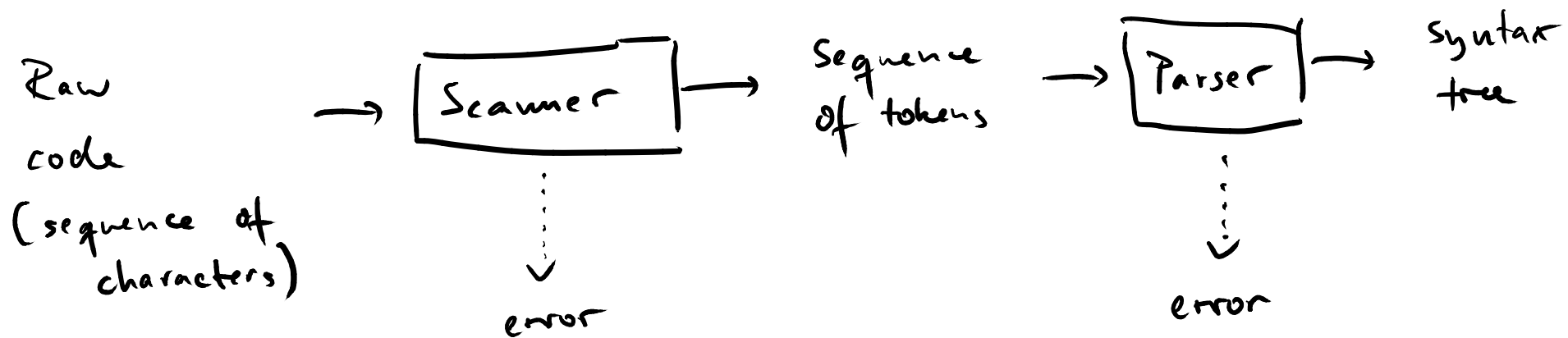
## Which of these statements is true?

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

# 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**  $\rightarrow$   $\epsilon$

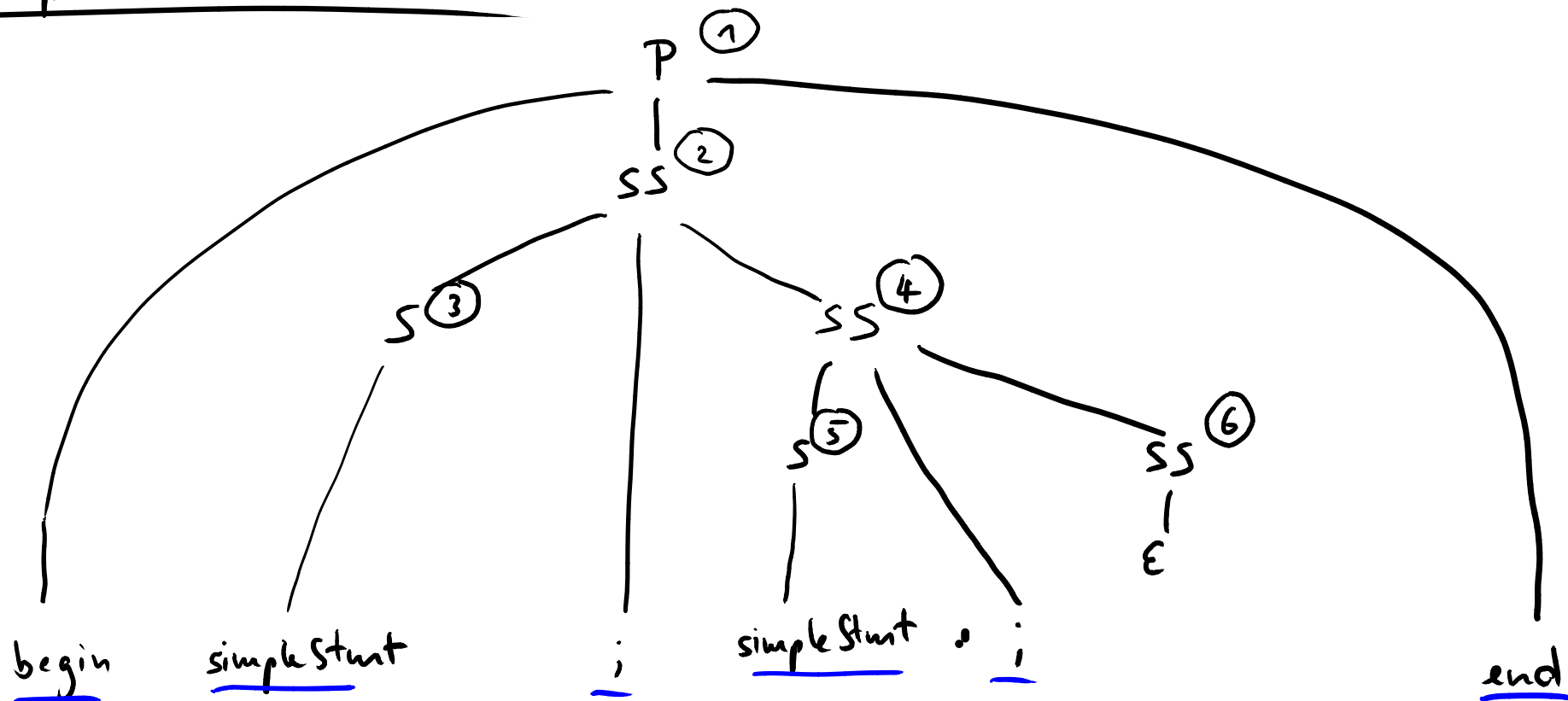
**S**  $\rightarrow$  **simplestmt**

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

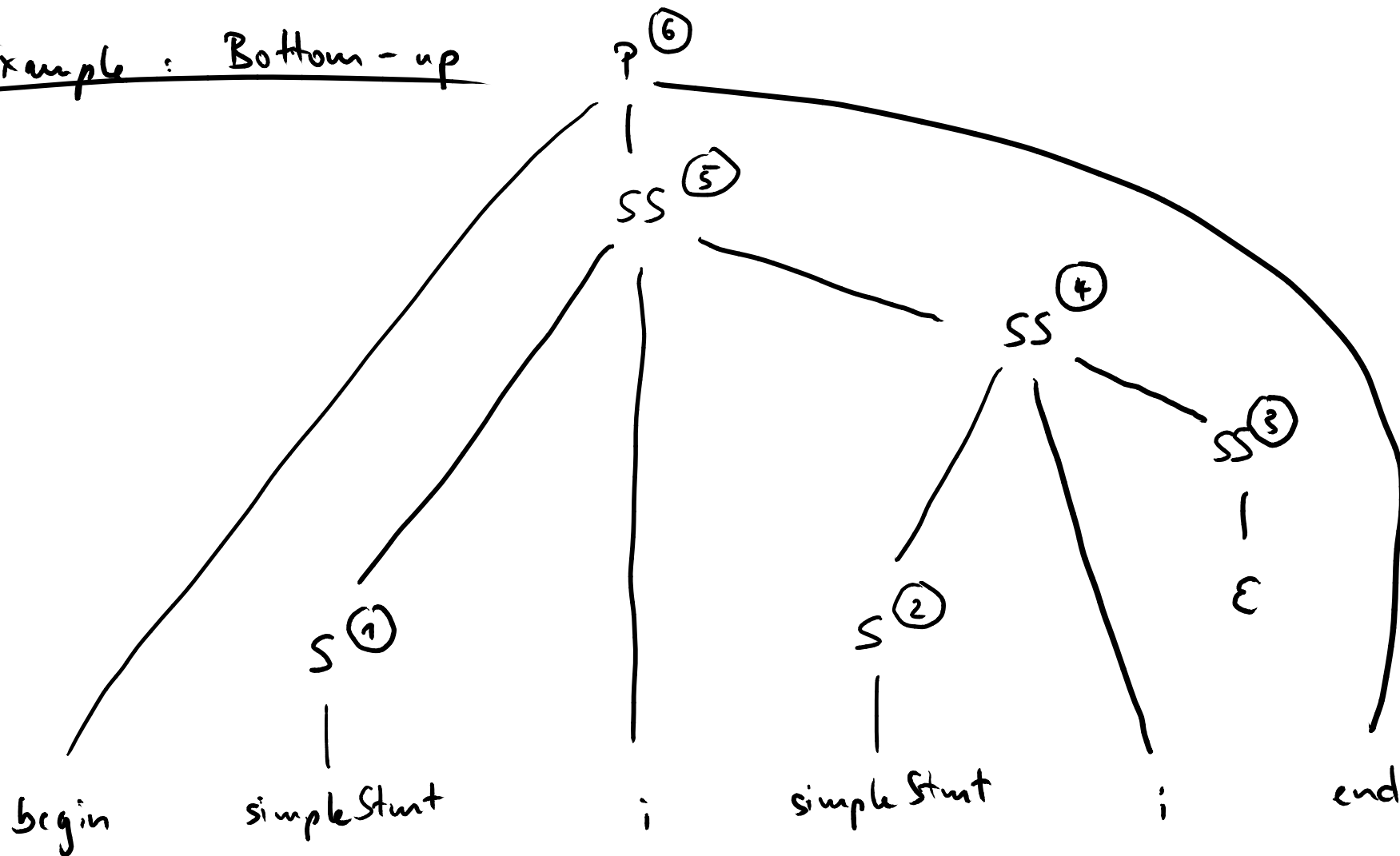
**Example program:**

**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**

---

# 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**

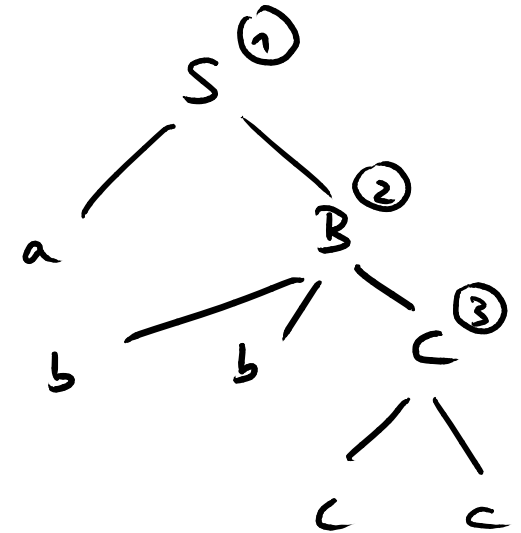
**B** → b b **C**

**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	<u>a</u> b b c c	Call S() from main() Call match(a) Call B()
2	<u>b</u> b c c	Call match(b) Call match(b) Call C()
3	<u>c</u> c	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:**

**S**  $\rightarrow$  **simple** | **begin S end**

**FIRST(S)** = { **simple**, **begin** }

## Computing FIRST sets

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

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

For a given grammar:

Apply above recursively until all FIRST sets remain constant

Example 1

$$S \rightarrow a S e$$

$$S \rightarrow B$$

$$B \rightarrow b B e$$

$$B \rightarrow C$$

$$C \rightarrow c C c$$

$$C \rightarrow d$$

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

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

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

## Example 2

$$P \rightarrow i | c | n | TS$$

$$Q \rightarrow P | a | S | d | S | c | S | T$$

$$R \rightarrow b | \epsilon$$

$$S \rightarrow e | R | n | \epsilon$$

$$T \rightarrow R | S | a$$

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

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

$$\text{FIRST}(R) = \{b, \epsilon\}$$

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

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

# Quiz: FIRST Sets

---

**S**  $\rightarrow$  **a S b** | **R c**

**T**  $\rightarrow$  **R S c** | **b**

**R**  $\rightarrow$  **c S b** |  $\epsilon$

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

Quiz

$$S \rightarrow aSb \mid Rc$$

$$T \rightarrow RSc \mid b$$

$$R \rightarrow cSb \mid \epsilon$$

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

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

$$\text{FIRST}(R) = \{c, \epsilon\}$$

# 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  $\epsilon$

**Example:**

**S**  $\rightarrow$  **a B c**

**B**  $\rightarrow$  **d**

**FOLLOW(S) = { EOF }**

**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  $B \rightarrow \alpha A \beta$ :  
Add  $\text{FIRST}(\beta) - \{ \epsilon \}$  to FOLLOW(A)
- Productions of the form  
 $B \rightarrow \alpha A$ , or  
 $B \rightarrow \alpha A \beta$  where  $\beta \Rightarrow^* \epsilon$ :  
Add FOLLOW(B) to FOLLOW(A)

Example 1

$$S \rightarrow aSe \mid B$$

$$B \rightarrow bBcf \mid C$$

$$C \rightarrow cCg \mid d \mid \epsilon$$

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

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

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

$$\text{FOLLOW}(S) = \{\text{EOF}, e\}$$

$$\text{FOLLOW}(B) = \{c, d, f, \text{EOF}, e\}$$

$$\text{FOLLOW}(C) = \{g, f, \text{EOF}, e, d, c\}$$

# Quiz: FOLLOW Sets

---

**S**  $\rightarrow$  **a S b** | **R c**

**T**  $\rightarrow$  **R S c** | **b**

**R**  $\rightarrow$  **c S b** |  $\epsilon$

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

Quiz

$$S \rightarrow aSb \mid Rc$$

$$T \rightarrow RSc \mid b$$

$$R \rightarrow cSb \mid \epsilon$$

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

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

$$\text{FIRST}(R) = \{c, \epsilon\}$$

$$\text{FOLLOW}(S) = \{b, c, \text{EOF}\}$$

$$\text{FOLLOW}(T) = \{\}$$

$$\text{FOLLOW}(R) = \{a, c\}$$

# PREDICT Sets

---

**PREDICT set for a rule:** Which terminals to look for in LL(1) parser

- If next input token is in PREDICT of rule, apply the rule

- **Computing the PREDICT set** for rule  $A \rightarrow \alpha$ :

- If  $\epsilon$  in  $\text{FIRST}(\alpha)$ :

$$\text{PREDICT}(A \rightarrow \alpha) = (\text{FIRST}(\alpha) - \{ \epsilon \}) \cup \text{FOLLOW}(A)$$

- Otherwise:

$$\text{PREDICT}(A \rightarrow \alpha) = \text{FIRST}(\alpha)$$

# Example

---

**Grammar:**

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

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

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

**C**  $\rightarrow$  **c c**



---

	<b>FIRST</b>	<b>FOLLOW</b>
<b>S</b>	<b>a, b</b>	<b>EOF</b>
<b>B</b>	<b>b</b>	<b>EOF</b>
<b>C</b>	<b>c</b>	<b>EOF</b>

---

# Example

---

Grammar:

$S \rightarrow a B$

$S \rightarrow b C$

$B \rightarrow b b C$

$C \rightarrow c c$



PREDICT:

{ a }

{ b }

{ b }

{ c }



---

FIRST

FOLLOW

---

S a, b

EOF

B b

EOF

C c

EOF

---

# Example

---

Grammar:

$S \rightarrow a B$

$S \rightarrow b C$

$B \rightarrow b b C$

$C \rightarrow c c$



**PREDICT:**

**{ a }**

**{ b }**

**{ b }**

**{ 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();  
}
```

---

	FIRST	FOLLOW
--	-------	--------

---

S	a, b	EOF
---	------	-----

B	b	EOF
---	---	-----

C	c	EOF
---	---	-----

---

# Example

---

Grammar:

$S \rightarrow a B$

$S \rightarrow b C$

$B \rightarrow b b C$

$C \rightarrow c c$



PREDICT:

{ a }

{ b }

{ b }

{ 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();
}
    
```

---

	FIRST	FOLLOW
S	a, b	EOF
B	b	EOF
C	c	EOF

---

# Example

---

Grammar:

$S \rightarrow a B$

$S \rightarrow b C$

$B \rightarrow b b C$

$C \rightarrow c c$



PREDICT:

{ a }

{ b }

{ b }

{ c }



---

FIRST

FOLLOW

---

S a, b

EOF

B b

EOF

C c

EOF

---

```
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();  
}
```

# Computing the Parse Table

---

## Computing an LL(1) parse table

- Given: PREDICT set of each rule
- Table is a **mapping M**:  
 $N \times T \rightarrow$  Production rule or error
- For all productions  $A \rightarrow \alpha$  do
  - For each terminal  $t$  in  $\text{PREDICT}(A \rightarrow \alpha)$ :  
 $M[A][t] = A \rightarrow \alpha$
  - Every undefined table entry is an error

## Example: Parse table

Termin.	a	b	c
Non-termin.			
S	1	2	-
B	-	3	-
C	-	-	4

Grammar:

$$1) S \rightarrow aB \quad \{a\}$$

$$2) S \rightarrow bC \quad \{b\}$$

$$3) B \rightarrow bbC \quad \{b\}$$

$$4) C \rightarrow cc \quad \{c\}$$

            
PREDICT

# Table-based, Predictive Parsing

---

```
stack.push(EOF) ; stack.push(startSymbol) ;
nextToken = lookAhead() ;
repeat
  x = stack.pop() ;
  if x is terminal or EOF
    if x == nextToken
      nextToken = lookAhead()
    else error()
  else // x is non-terminal
    if M[x][nextToken] == x -> y1 y2 .. ym
      stack.push(ym) ; ... ; stack.push(y1) ;
    else error()
until x is EOF
```

# Table-based, Predictive Parsing

---

Parse stack: Prediction of  
what will be seen in the future

```
stack.push(EOF); stack.push(startSymbol);  
nextToken = lookAhead();  
repeat  
  x = stack.pop();  
  if x is terminal or EOF  
    if x == nextToken  
      nextToken = lookAhead()  
    else error()  
  else // x is non-terminal  
    if M[x][nextToken] == x -> y1 y2 .. ym  
      stack.push(ym); ...; stack.push(y1);  
    else error()  
until x is EOF
```

# Table-based, Predictive Parsing

---

```
stack.push(EOF); stack.push(startSymbol);
nextToken = lookAhead();
repeat
    x = stack.pop();
    if x is terminal or EOF
        if x == nextToken
            nextToken = lookAhead()
        else error()
    else // x is non-terminal
        if M[x][nextToken] == x -> y1 y2 .. ym
            stack.push(ym); ...; stack.push(y1);
        else error()
until x is EOF
```

**Read one token after another, always  
looking only one token ahead**

# Table-based, Predictive Parsing

---

Check if expected terminal is indeed the next token

```
stack.push(EOF) ; stack.push(startSymbol) ;
nextToken = lookAhead() ;
repeat
  x = stack.pop() ;
  if x is terminal or EOF
    if x == nextToken
      nextToken = lookAhead()
    else error()
  else // x is non-terminal
    if M[x][nextToken] == x -> y1 y2 .. ym
      stack.push(y1) ; ... ; stack.push(ym) ;
    else error()
until x is EOF
```

# Table-based, Predictive Parsing

---

```
stack.push(EOF); stack.push(startSymbol);
```

```
nextToken = lookAhead();
```

```
repeat
```

```
  x = stack.pop();
```

```
  if x is terminal or EOF
```

```
    if x == nextToken
```

```
      nextToken = lookAhead()
```

```
    else error()
```

```
  else // x is non-terminal
```

```
    if  $M[x][nextToken] == x \rightarrow y_1 y_2 \dots y_m$ 
```

```
      stack.push( $y_m$ ); ...; stack.push( $y_1$ );
```

```
    else error()
```

```
until x is EOF
```

**Apply a production  
rule: Push right-hand  
side onto stack**

# Table-based, Predictive Parsing

---

```
stack.push(EOF); stack.push(startSymbol);
nextToken = lookAhead();
repeat
  x = stack.pop();
  if x is terminal or EOF
    if x == nextToken
      nextToken = lookAhead()
    else error()
  else // x is non-terminal
    if M[x][nextToken] == x -> y1 y2 .. ym
      stack.push(ym); ...; stack.push(y1);
    else error()
until x is EOF
```

**No entry in table:  
Raise error**


## Example: Table-based parsing

Input: bcc

stack	Remaining input	Steps
EOF, <u>S</u>	<u>b</u> , c, c, EOF	Pop S Use rule $S \rightarrow bC$
EOF, C, <u>b</u>	<u>b</u> , c, c, EOF	Push C, b Pop b Read next token
EOF, <u>C</u>	<u>c</u> , c, EOF	Pop C Use rule $C \rightarrow cc$ Push c, c
EOF, c, <u>c</u>	<u>c</u> , c, EOF	Pop c Read next token
EOF, <u>c</u>	<u>c</u> , EOF	Pop c Read next token
<u>EOF</u>	EOF	Pop EOF → Done

# Overview

---

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

# Bottom-up Parsing

---

- **LR(k) parsers**
  - Left-to-right scanning, Right-most derivation, k tokens look-ahead
- **Difficult to do by hand**
- **Mostly based on automatically generated table**

# Shift-reduce Algorithm

---

- **Repeat** until all tokens read and all symbols reduced to start symbol:
  - Shift (i.e., read) input tokens
  - Try to reduce a group of symbols into a single non-terminal

## Example: Shift-reduce parsing

$S \rightarrow aTRe$

$T \rightarrow Tbc \mid b$

$R \rightarrow d$

Input: a b b c d e ✓

### Steps:

Shift a, shift b

Reduce  $T \rightarrow b$

Shift b, shift c

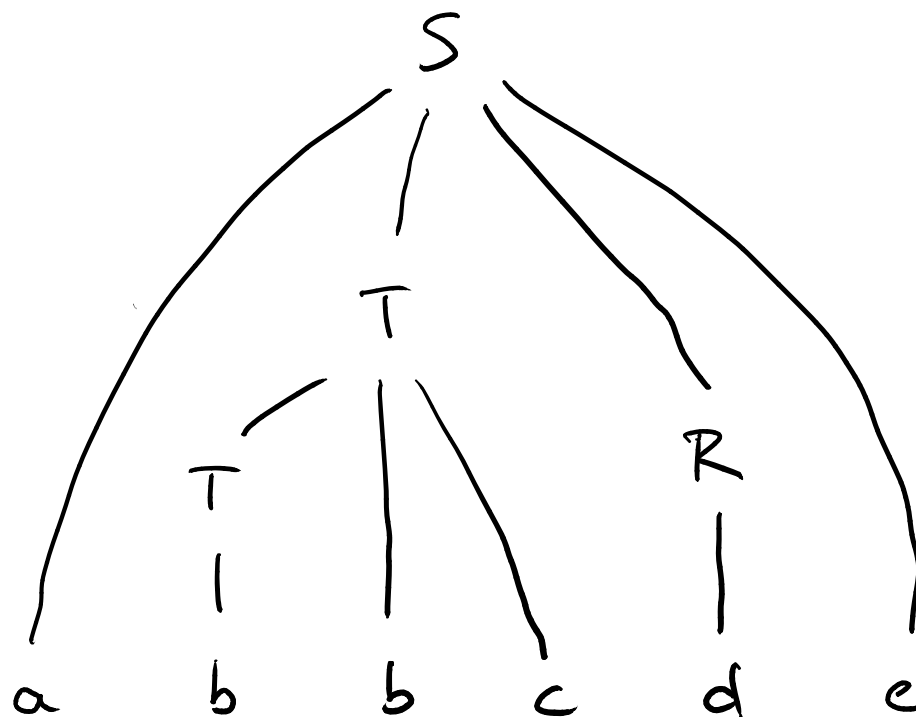
Reduce  $T \rightarrow Tbc$

Shift d

Reduce  $R \rightarrow d$

Shift e

Reduce  $S \rightarrow aTRe$



# Table-based LR Parsing

---

- **Two tables**

- **Action table:**

- state  $\times$  T  $\rightarrow$  reduce/shift/accept/error

- **Goto table:**

- state  $\times$  N  $\rightarrow$  state

- **Stack of symbol/state pairs**

- Record of what has been seen in the past

# Example: LR(1) Table

---

State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1		s3					2		
s2		s5		s6				4	
s3		r3		r3					
s4					s7				
s5			s8						
s6					r4				
s7						acc.			
s8		r2		r2					

# Example: LR(1) Table

---

State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1		s3					2		
s2		s5		s6				4	
s3		r3		r3					
s4					s7				
s5			s8						
s6					r4				
s7						acc.			
s8		r2		r2					

**Action table**

# Example: LR(1) Table

---

State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1		s3					2		
s2		s5		s6				4	
s3		r3		r3					
s4					s7				
s5			s8						
s6					r4				
s7						acc.			
s8		r2		r2					

**Goto table**

# Example: LR(1) Table

State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1	s3						2		
s2	s5		s6					4	
s3	r3		r3						
s4					s7				
s5		s8							
s6					r4				
s7						acc.			
s8	r2		r2						

**s means  
shift to  
some state**

# Example: LR(1) Table


State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1		s3						2	
s2		s5		s6					4
s3		r3		r3					
s4					s7				
s5			s8						
s6					r4				
s7						acc.			
s8		r2		r2					

**r means reduce using some production**

# Example: LR(1) Table

State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1		s3					2		
s2		s5		s6				4	
s3		r3		r3					
s4					s7				
s5			s8						
s6					r4				
s7						acc.			
s8		r2		r2					

**Accept input  
(i.e., done with  
parsing)**



# Example: LR(1) Table

State	a	b	c	d	e	EOF	S	T	R
s0	s1								
s1		s3					2		
s2		s5		s6				4	
s3		r3		r3					
s4					s7				
s5			s8						
s6					r4				
s7						acc.			
s8		r2		r2					

**No entry  
means error**

# Table-based LR(1) Parsing

---

```
stack.push(EOF, 0);
nextToken = lookAhead();
repeat
    s = state on top of stack
    if action[s, nextToken] == shift s'
        stack.push(nextToken, s');
        nextToken = lookAhead();
    else if action[s, nextToken] == reduce x -> y1 .. ym
        pop m pairs from stack
        s' = state on top of stack
        stack.push(x, goto[s', x])
    else if action[s, nextToken] == accept
        accept and return
    else error()
```

# Table-based LR(1) Parsing

---

Stack hold roots of partial trees found so far

```
stack.push(EOF, 0);  
nextToken = lookAhead();  
repeat  
  s = state on top of stack  
  if action[s, nextToken] == shift s'  
    stack.push(nextToken, s');  
    nextToken = lookAhead();  
  else if action[s, nextToken] == reduce x -> y1 .. ym  
    pop m pairs from stack  
    s' = state on top of stack  
    stack.push(x, goto[s', x])  
  else if action[s, nextToken] == accept  
    accept and return  
  else error()
```

# Table-based LR(1) Parsing

---

```
stack.push(EOF, 0);
nextToken = lookAhead();
repeat
```

```
  s = state on top of stack
```

```
  if action[s, nextToken] == shift s'
```

```
    stack.push(nextToken, s');
```

```
    nextToken = lookAhead();
```

```
  else if action[s, nextToken] == reduce x -> y1 .. ym
```

```
    pop m pairs from stack
```

```
    s' = state on top of stack
```

```
    stack.push(x, goto[s', x])
```

```
  else if action[s, nextToken] == accept
```

```
    accept and return
```

```
  else error()
```

**Reduce partial  
trees into a  
non-terminal  
by applying a  
rule**

# Table-based LR(1) Parsing

---

```
stack.push(EOF, 0);
nextToken = lookAhead();
repeat
  s = state on top of stack
  if action[s, nextToken] == shift s'
    stack.push(nextToken, s');
    nextToken = lookAhead();
  else if action[s, nextToken] == reduce x -> y1 .. ym
    pop m pairs from stack
    s' = state on top of stack
    stack.push(x, goto[s', x])
  else if action[s, nextToken] == accept
    accept and return
  else error()
```

**Read  
another  
token**

# Table-based LR(1) Parsing

---

```
stack.push(EOF, 0);
nextToken = lookAhead();
repeat
  s = state on top of stack
  if action[s, nextToken] == shift s'
    stack.push(nextToken, s');
    nextToken = lookAhead();
  else if action[s, nextToken] == reduce x -> y1 .. ym
    pop m pairs from stack
    s' = state on top of stack
    stack.push(x, goto[s', x])
  else if action[s, nextToken] == accept
    accept and return
  else error()
```

**All subtrees  
reduced to  
start symbol**

# How to Get the Table?

---

- Using a “**characteristic finite-state machine**” computed from the grammar
- Details differ for different kinds of LR parsers
  - SLR (simple LR)
  - LALR (look-ahead LR)
  - Full-LR
- Beyond the scope of this course

# Quiz: Parsing

---

## Which of these statements is true?

- The R in LR(k) stands for read-only.
- PREDICT sets are the union of FIRST and FOLLOW sets.
- The stack of a top-down parser contains the symbols expected in the future.
- Recursive descent parsers build a parse tree from the top down.

# Quiz: Parsing

---

## Which of these statements is true?

- ~~The R in LR(k) stands for read-only.~~
- ~~PREDICT sets are the union of FIRST and FOLLOW sets.~~
- The stack of a top-down parser contains the symbols expected in the future.
- Recursive descent parsers build a parse tree from the top down.

# Overview

---

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