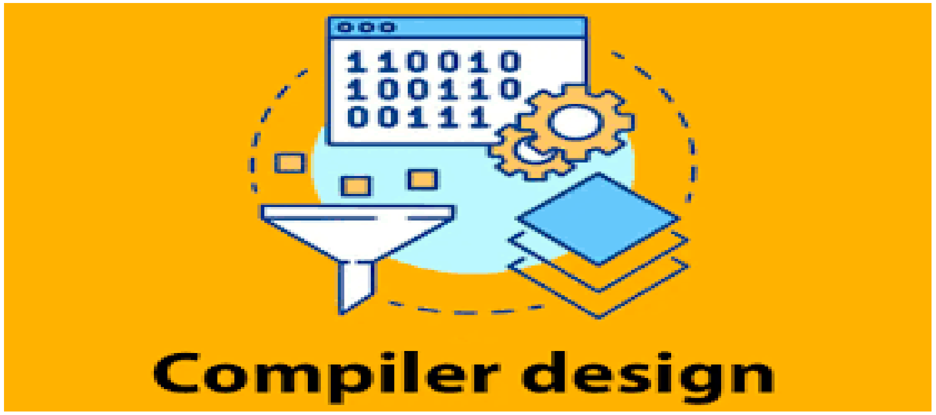
Menoufia University
Faculty of computers & Information
Computer Science Department.

Compiler Design

4 Year – first Semester

Lecture 5



DR. Eman Meslhy Mohamed
Lecturer at Computer Science department

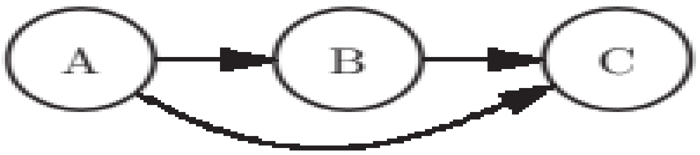
2023-2024

Outline

- **Parsing Algorithms.**
- Relations and Closure
- Simple Grammar.
- Pushdown Machine for Simple Grammar.
- Recursive Descent Parsers for Simple Grammar.
- Quasi-Simple Grammar.
- Pushdown Machine for Simple Grammar.
- Recursive Descent Parsers for Simple Grammar.

Relations and Closure

- A relation is a set of ordered pairs. Each pair may be listed in parentheses and separated by commas, as in the following example:
- **R1: (a,b) (c,d)**



- If R is a relation,

➔ then the **reflexive transitive closure of R is designated R*;**

- it is a relation made up of the same elements of R with the following properties:
  - 1. All pairs of R are also in R*.
  - 2. If (a,b) and (b,c) are in R*, then (a,c) is in R* (Transitive).
  - 3. If a is in one of the pairs of R, then (a,a) is in R* (Reflexive)

Example

- Show R1* the reflexive transitive closure of the relation
- R1*:
- (a,b) (c,d) (b,a) (b,c) (c,c) ➔ (from R1)
- (a,c) (b,d) (a,d) ➔ (transitive)
- (a,a) (b,b) (d,d) ➔ (reflexive)

R1:
(a,b)
(c,d)
(b,a)
(b,c)
(c,c)

Quiz

- Show the reflexive transitive closure of
- (a,b) (a,d) (b,c)

```
(a,b)
(a,d)
(b,c)
(a,c)
(a,a)
(b,b)
(c,c)
(d,d)
```

- (a,a) (a,b) (b, b)

```
(a,a)
(a,b)
(b,b)
```

Parsing Algorithms

Parsing algorithms

Parsing problem: Given a **grammar** and an **input string**, determine whether the string is in the language of the grammar, and, if so, determine **its structure**.

Parsing algorithms are usually classified based on the sequence in which a derivation tree is built or traversed into two categories:

- **Top-Down Parsing.**

It is a parsing technique that looks at the highest level of the parse tree initially, and then works its way down to the parse tree.

- **Bottom-Up Parsing.**

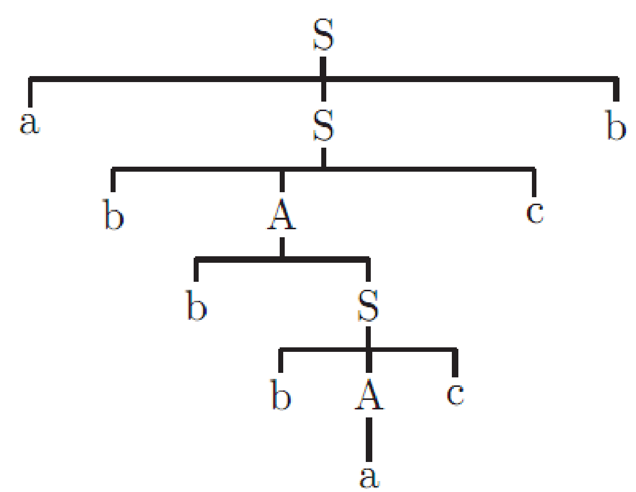
It is a parsing technique that looks at the lowest level of the parse tree and then works its way up to the parse tree.

Top-Down Parsing

A **top-down** algorithm will begin with the starting nonterminal and try to **decide which rule of the grammar should be applied**.

$$
\begin{aligned}
S &\rightarrow a\ S\ b \\
S &\rightarrow b\ A\ c \\
A &\rightarrow b\ S \\
A &\rightarrow a
\end{aligned}
$$

abbbaccb



$$S \Rightarrow \underset{1}{\underline{a}Sb} \Rightarrow \underset{2}{\underline{ab}Acb} \Rightarrow \underset{3}{\underline{abb}Scb} \Rightarrow \underset{2}{\underline{abbb}Accb} \Rightarrow \underset{4}{\underline{abbbaccb}}$$

Outline

- Parsing Algorithms.
- **Simple Grammar.**
- Pushdown Machine for Simple Grammar.
- Recursive Descent Parsers for Simple Grammar.
- Quasi-Simple Grammar.
- Pushdown Machine for Simple Grammar.
- Recursive Descent Parsers for Simple Grammar.

Simple Grammar

A grammar is a **simple grammar** if every rule is of the form:

$$\mathbf{A \rightarrow a\alpha}$$

(where **A** represents any **nonterminal**, **a** represents any **terminal**, and **α** represents any string of **terminals and nonterminals**) and every pair of rules defining the same nonterminal **begin with different terminals** on the right side of the arrow.

G9:
$$S \rightarrow aSb$$
$$S \rightarrow b$$

Simple

G10:
$$S \rightarrow aSb$$
$$S \rightarrow \epsilon$$

Not simple

G11:
$$S \rightarrow aSb$$
$$S \rightarrow a$$

Not simple

Selection Set

- The set of input symbols (i.e. terminal symbols) which imply the application of a grammar rule is called the **selection set** of that rule.

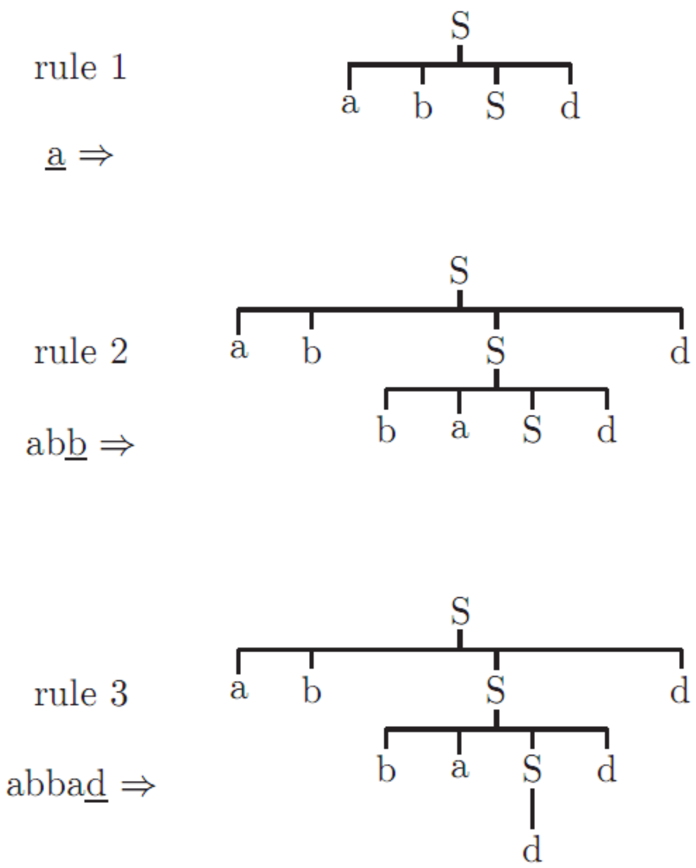
	Selection Set
$S \rightarrow a\ b\ S\ d$	$\{a\}$
$S \rightarrow b\ a\ S\ d$	$\{b\}$
$S \rightarrow d$	$\{d\}$

Selection Set

Selection Set

$S \rightarrow a\ b\ S\ d$	$\{a\}$
$S \rightarrow b\ a\ S\ d$	$\{b\}$
$S \rightarrow d$	$\{d\}$

abbaddd



- For **simple grammars**, the selection set of each rule always contains **exactly one terminal symbol** (i.e., the one beginning the right-hand side).

Quiz

Determine which of the following grammars are s*imple*

(a)

```
1.   S  →  a S b
2.   S  →  b
```

(b)

```
1.   Expr  →  Expr + Term
2.   Expr  →  Term
3.   Term  →  var
4.   Term  →  ( Expr )
```

(c)

```
1.   S  →  a A b B
2.   A  →  b A
3.   A  →  a
4.   B  →  b A
```

Outline

- Parsing Algorithms.
- Simple Grammar.
- **Pushdown Machine for Simple Grammar.**
- Recursive Descent Parsers for Simple Grammar.
- Quasi-Simple Grammar.
- Pushdown Machine for Simple Grammar.
- Recursive Descent Parsers for Simple Grammar.

Pushdown Machine for Simple Grammar

$$S \rightarrow aSB$$
$$S \rightarrow b$$
$$B \rightarrow a$$
$$B \rightarrow bBa$$

- It is always possible to construct a **one-state pushdown machine** to parse the language of a simple grammar.

As each input symbol is read, the machine will attempt to apply one of the four rules in the grammar.

- ✓ If the top stack symbol is S, the machine will apply either rule 1 or 2.
- ✓ If the top stack symbol is B, the machine will apply either rule 3 or rule 4.

Pushdown Machine for Simple Grammar

Pushdown Machine for Simple Grammar

1. Build a table with each column labeled by a terminal symbol (and **end-marker ←**) and each row labeled by a **nonterminal and terminal** symbol (and **bottom marker ∇**).

$S \rightarrow aSB$

$S \rightarrow b$

$B \rightarrow a$

$B \rightarrow bBa$

	a	b	↵
S			
B			
a			
b			
∇			

Pushdown Machine for Simple Grammar

2. For each grammar rule of the form **A → aα**, fill in the cell in row **A** and column **a** with: **REP(α^r a)**, **retain**, where α^r represents α **reversed** (here, a represents a terminal, and α represents a string of terminals and nonterminals).

S → aSB
S → b
B → a
B → bBa

	a	b	↲
S	Rep (Bsa) Retain	Rep (b) Retain	
B	Rep (a) Retain	Rep (aBb) Retain	
a			
b			
∇			

Pushdown Machine for Simple Grammar

3. Fill in the cell in row **a** and column **a** with **pop**, **advance**.

$S \rightarrow aSB$
$S \rightarrow b$
$B \rightarrow a$
$B \rightarrow bBa$

	a	b	↲
S	Rep (Bsa) Retain	Rep (b) Retain	
B	Rep (a) Retain	Rep (aBb) Retain	
a	pop Advance		
b		pop Advance	
∇			

Pushdown Machine for Simple Grammar

Pushdown Machine for Simple Grammar

4. Fill in the cell in row ∇ , and column ← with **Accept**.

$S \rightarrow aSB$

$S \rightarrow b$

$B \rightarrow a$

$B \rightarrow bBa$

	a	b	↵
S	Rep (Bsa) Retain	Rep (b) Retain	
B	Rep (a) Retain	Rep (aBb) Retain	
a	pop Advance		
b		pop Advance	
∇			Accept

Pushdown Machine for Simple Grammar

Pushdown Machine for Simple Grammar

5. Fill in all other cells with **Reject**.

6. Initialize the stack with ∇, and the starting nonterminal.

$S \rightarrow aSB$

$S \rightarrow b$

$B \rightarrow a$

$B \rightarrow bBa$

	a	b	↵
S	Rep (Bsa) Retain	Rep (b) Retain	Reject
B	Rep (a) Retain	Rep (aBb) Retain	Reject
a	pop Advance	Reject	Reject
b	Reject	pop Advance	Reject
∇	Reject	Reject	Accept

S
∇

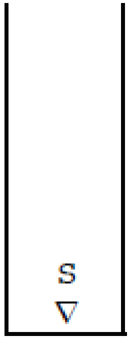
Initial

Pushdown Machine for Simple Grammar

Pushdown Machine for Simple Grammar

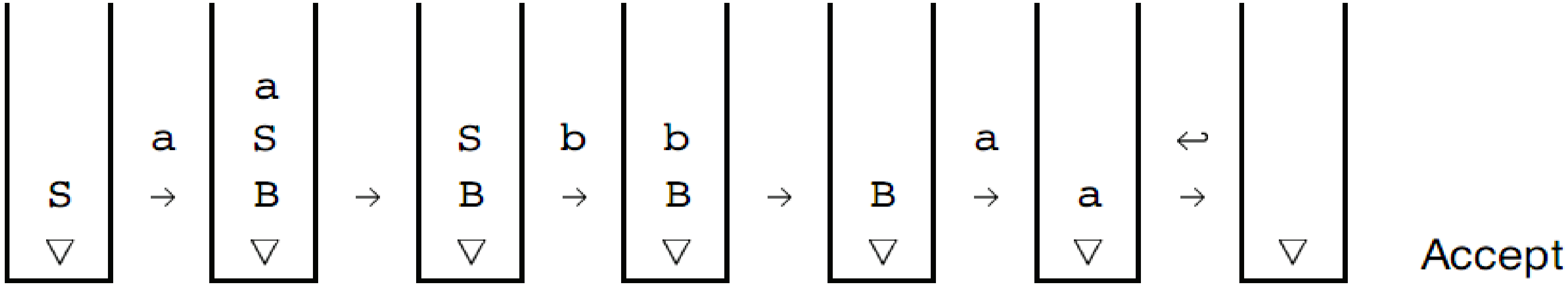
$S \rightarrow aSB$
$S \rightarrow b$
$B \rightarrow a$
$B \rightarrow bBa$

	a	b	↵
S	Rep (Bsa) Retain	Rep (b) Retain	Reject
B	Rep (a) Retain	Rep (aBb) Retain	Reject
a	pop Advance	Reject	Reject
b	Reject	pop Advance	Reject
∇	Reject	Reject	Accept



Initial

aba



Pushdown Machine for Simple Grammar

Pushdown Machine for Simple Grammar

Show a **one state pushdown** machine for the following grammar:

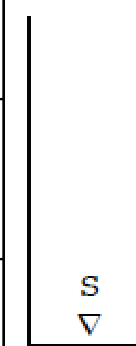
$S \rightarrow 0S1A$

$S \rightarrow 10A$

$A \rightarrow 0S0$

$A \rightarrow 1$

	1	0	↵
S	Rep(A01) Retain	Rep(A1S0) Retain	Reject
A	Rep(1) Retain	Rep(0S0) Retain	Reject
0	Reject	pop advance	Reject
1	pop advance	Reject	Reject
Δ	Reject	Reject	Accept



Initial

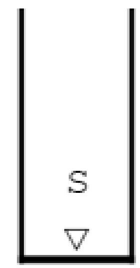
Quiz

(c)

1. S → a A b B
2. A → b A
3. A → a
4. B → b A

(c)

S1	a	b	↩
S	Rep (BbAa) Retain	Reject	Reject
A	Rep(a) Retain	Rep(Ab) Retain	Reject
B	Reject	Rep(Ab) Retain	Reject
a	pop adv	Reject	Reject
b	Reject	pop adv	Reject
▽	Reject	Reject	Accept



Initial Stack

Outline

- Parsing Algorithms.
- Simple Grammar.
- Pushdown Machine for Simple Grammar.
- **Recursive Descent Parsers for Simple Grammar.**
- Quasi-Simple Grammar.
- Pushdown Machine for Simple Grammar.
- Recursive Descent Parsers for Simple Grammar.

Recursive Descent Parsers for Simple Grammar

A second way of implementing a parser for simple grammars is **Recursive Descent**, in which the parser is written using a traditional programming language, such as Java or C++ and the **method** is written for **each nonterminal** in the grammar.

$S \rightarrow aSB$

$S \rightarrow b$

$B \rightarrow a$

$B \rightarrow bBa$

```
class RDP // Recursive Descent Parser
{
char inp;
public static void main (String[] args) throws
IOException
{ InputStreamReader stdin = new InputStreamReader
(System.in);
RDP rdp = new RDP();
rdp.parse();
}
```

Recursive Descent Parsers for Simple Grammar

Recursive Descent Parsers for Simple Grammar

```
void parse ()
{ inp = getInp();
   S ();                              // Call start nonterminal
   if (inp==' ↵') accept();           // end of string marker
   else reject();
}
```

$S \rightarrow aSB$

$S \rightarrow b$

$B \rightarrow a$

$B \rightarrow bBa$

```
char getInp()
{ try
{ return (char) System.in.read(); }
catch (IOException ioe)
{ System.out.println ("IO error " + ioe); }
return '#'; // must return a char
}
```

```
void accept() // Accept the input
{ System.out.println ("accept"); }
```

```
void reject() // Reject the input
{ System.out.println ("reject");
System.exit(0); // terminate parser
}
```

Recursive Descent Parsers for Simple Grammar

```
void parse ()
{ inp = getInp();
   S ();                              // Call start nonterminal
   if (inp==' ↵') accept();           // end of string marker
   else reject();
}
```

$S \rightarrow aSB$

$S \rightarrow b$

$B \rightarrow a$

$B \rightarrow bBa$

```
void S ()
{ if (inp=='a')                        // apply rule 1
      { inp = getInp();
        S ();
        B ();
      }                                 // end rule 1
else if (inp=='b') inp = getInp();  // apply rule 2
else reject();
}
```

Recursive Descent Parsers for Simple Grammar

```
void parse ()
{ inp = getInp();
   S ();                          // Call start nonterminal
   if (inp==’ ↵’) accept();      // end of string marker
   else reject();
}
```

$S \rightarrow aSB$

$S \rightarrow b$

$B \rightarrow a$

$B \rightarrow bBa$

```
void B ()
{ if (inp==’a’) inp = getInp();       // rule 3
  else if (inp==’b’)                        // apply rule 4
       { inp = getInp();
          B();
          if (inp==’a’) inp = getInp();
          else reject();
        }                                // end rule 4
  else reject();
}
```

Recursive Descent Parsers for Simple Grammar

Show a **recursive descent parser** for the following grammar:

$S \rightarrow 0S1A$

$S \rightarrow 10A$

$A \rightarrow 0S0$

$A \rightarrow 1$

```
void S()
  {  if (inp=='0')                    // apply rule 1
       {  getInp();
          S();
          if (inp=='1') getInp();
          else Reject();
          A();
       }                              // end rule 1
     else if (inp=='1')               // apply rule 2
       {  getInpu();
          if (inp=='0') getInp();
          else reject();
          A();
       }                              // end rule 2
     else reject();
  }
```

```
void A()
  {  if (inp=='0')              // apply rule 3
       {  getInp();
          S();
          if (inp=='0') getInp();
          else reject();
       }                        // end rule 3
     else if (inp=='1') getInp()  // apply rule 4
     else reject();
  }
```

Outline

- Parsing Algorithms.
- Simple Grammar.
- Pushdown Machine for Simple Grammar.
- Recursive Descent Parsers for Simple Grammar.
- **Quasi-Simple Grammar.**
- Pushdown Machine for Simple Grammar.
- Recursive Descent Parsers for Simple Grammar.

Quasi-Simple Grammar

Quasi-Simple Grammar

A quasi-simple grammar is a grammar which obeys the restriction of simple grammars, but which may also contain rules of the form:

$$\mathbf{N \rightarrow \epsilon}$$

$$S \rightarrow a\,A\,S$$
$$S \rightarrow b$$
$$A \rightarrow c\,A\,S$$
$$A \rightarrow \varepsilon$$

Quasi-Simple Grammar

Follow Set

- **Follow set** of **A** is the set of all terminals (or endmarker ← (which can immediately follow an A in an intermediate form derived from S←, where S is the starting nonterminal.

$$S \rightarrow a\,A\,S$$
$$S \rightarrow b$$
$$A \rightarrow c\,A\,S$$
$$A \rightarrow \varepsilon$$

- S ← ⇒aAS ← ⇒acASS ← ⇒acASaAS ←
  ⇒ acASb ←
- S ← ⇒aAS ← ⇒aAaAS ←
  ⇒ aAb ←

$\boxed{\text{Fol(S) = \{a,b,←\}}}$

$\boxed{\text{Fol(A) = \{a,b\}}}$

Given the following grammar:

1. S → a A b S
2. S → ε
3. A → a S b
4. A → ε

(a) Find the *follow set* for each nonterminal.

$$\text{Fol}(S) = \{b, \hookleftarrow\}$$

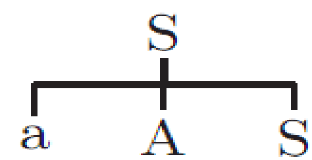
$$\text{Fol}(A) = \{b\}$$

Quasi-Simple Grammar

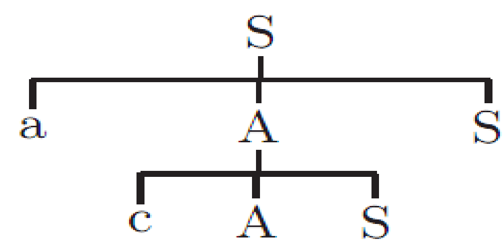
Selection & Follow Set

$S \rightarrow a\,A\,S$	{a}
$S \rightarrow b$	{b}
$A \rightarrow c\,A\,S$	{c}
$A \rightarrow \varepsilon$	{a, b}

rule 1
a $\Rightarrow$

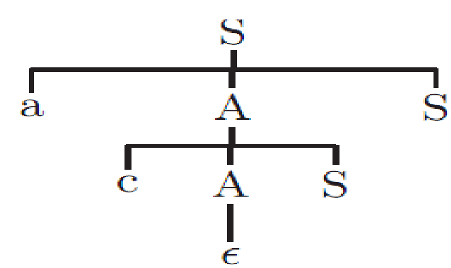


rule 3
ac $\Rightarrow$

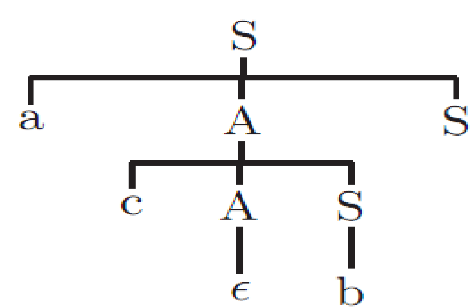


acbb

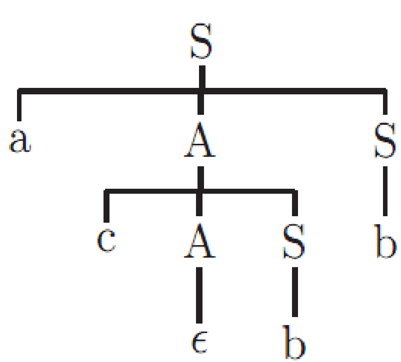
rule 4
acb $\Rightarrow$



rule 2
acb $\Rightarrow$



rule 2
acbb $\Rightarrow$



Outline

- Parsing Algorithms.
- Simple Grammar.
- Pushdown Machine for Simple Grammar.
- Recursive Descent Parsers for Simple Grammar.
- Quasi-Simple Grammar.
- **Pushdown Machine for Quasi-Simple Grammar.**
- Recursive Descent Parsers for Simple Grammar.

Pushdown Machine for Quasi-Simple Grammar

- To build a **pushdown machine** for **a quasi-simple grammar**. We need to **apply an ε rule** by simply **popping the nonterminal off** the stack and **retaining** the input pointer.

- **For each ε rule in the grammar**, fill in cells of the row corresponding to the nonterminal on the left side of the arrow, but only in those columns corresponding to elements of **the follow set** of the nonterminal. Fill in these cells with **Pop, Retain**.

Pushdown Machine for Quasi-Simple Grammar

Pushdown Machine for Quasi-Simple Grammar

$S \rightarrow a\,A\,S$

$S \rightarrow b$

$A \rightarrow c\,A\,S$

$A \rightarrow \epsilon$

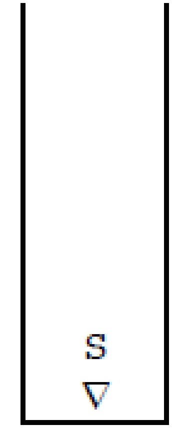
Fol(A) = {a,b}

	a	b	c	↵
S	Rep (SAa) Retain	Rep (b) Retain	Reject	Reject
A	Pop Retain	Pop Retain	Rep (SAc) Retain	Reject
a	Pop Advance	Reject	Reject	Reject
b	Reject	Pop Advance	Reject	Reject
c	Reject	Reject	Pop Advance	Reject
∇	Reject	Reject	Reject	Accept

S
∇

Initial
Stack

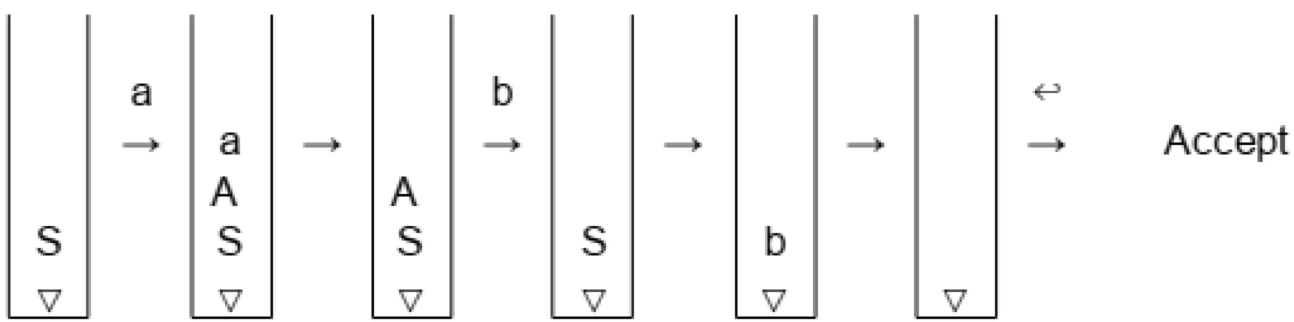
	a	b	c	↲
S	Rep (SAa) Retain	Rep (b) Retain	Reject	Reject
A	Pop Retain	Pop Retain	Rep (SAc) Retain	Reject
a	Pop Advance	Reject	Reject	Reject
b	Reject	Pop Advance	Reject	Reject
c	Reject	Reject	Pop Advance	Reject
∇	Reject	Reject	Reject	Accept



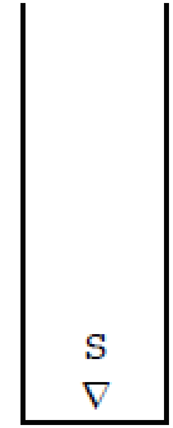
Initial Stack

ab

(a)
ab



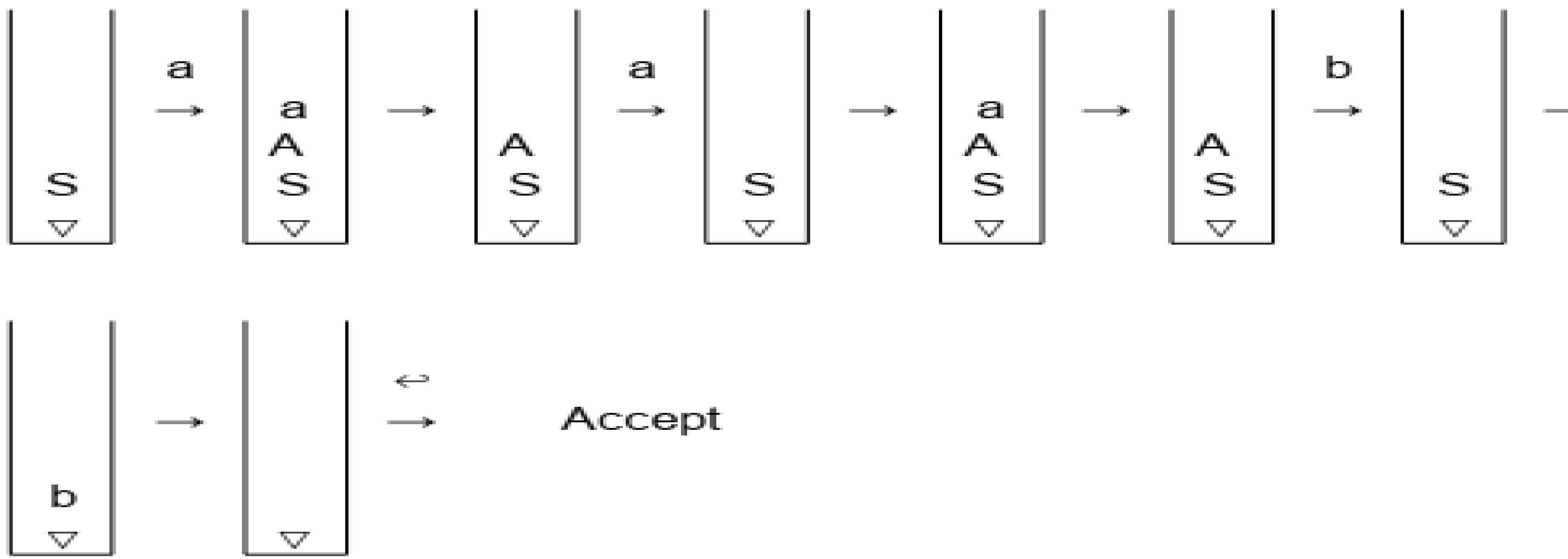
	a	b	c	↵
S	Rep (SAa) Retain	Rep (b) Retain	Reject	Reject
A	Pop Retain	Pop Retain	Rep (SAc) Retain	Reject
a	Pop Advance	Reject	Reject	Reject
b	Reject	Pop Advance	Reject	Reject
c	Reject	Reject	Pop Advance	Reject
∇	Reject	Reject	Reject	Accept



Initial Stack

aab

(c)
aab



Outline

- Parsing Algorithms.
- Simple Grammar.
- Pushdown Machine for Simple Grammar.
- Recursive Descent Parsers for Simple Grammar.
- Quasi-Simple Grammar.
- Pushdown Machine for Quasi-Simple Grammar.
- **Recursive Descent Parsers for Quasi-Simple Grammar.**

Recursive Descent Parsers for Quasi-Simple Grammar

For quasi-simple grammar, we need to check for all the input symbols in the **selection set of an ε rule**. If any of these are the current input symbol, we simply **return to the calling method without reading any input**.

Recursive Descent Parsers for Quasi-Simple Grammar

$S \rightarrow a\,A\,S$

$S \rightarrow b$

$A \rightarrow c\,A\,S$

$A \rightarrow \epsilon$

```
void parse ()
{ inp = getInp();
   S ();                                  // Call start nontermina
   if (inp==’ ↲’) accept();               // end of string marker
   else reject();
}
```

```
void S ()
{ if (inp==’a’)                           // apply rule 1
     { inp = getInp();
       A();
       S ();
     }                                    // end rule 1
else if (inp==’b’) inp = getInp();  // apply rule 2
else reject();
}
```

```
void A ()
{ if (inp==’c’)                           // apply rule 3
   { inp = getInp();
     A ();
     S ();
   }                    // end rule 3
  else if (inp==’a’ || inp==’b’)   {  } // apply rule 4
  else reject();
}
```

Quiz

10-11-2023

at 7 PM

Quiz

1. Show a **finite state machine** in either state graph or table form for Strings containing an **odd number of zeros or an even number of ones <u>but not both</u>**.

2. Describe the Strings containing **three sequential ones** by using **regular expressions**?

3. Consider the following grammar:

$$S \rightarrow A$$
$$A \rightarrow A{+}A \mid B{+}{+}$$
$$B \rightarrow y$$

a) Draw the **parse tree** for the input "**y + + + y + +**"

THANKS

for your attention