LECTURE 06

APPLICATIONS OF RECURSION

TODAY'S CLASS:

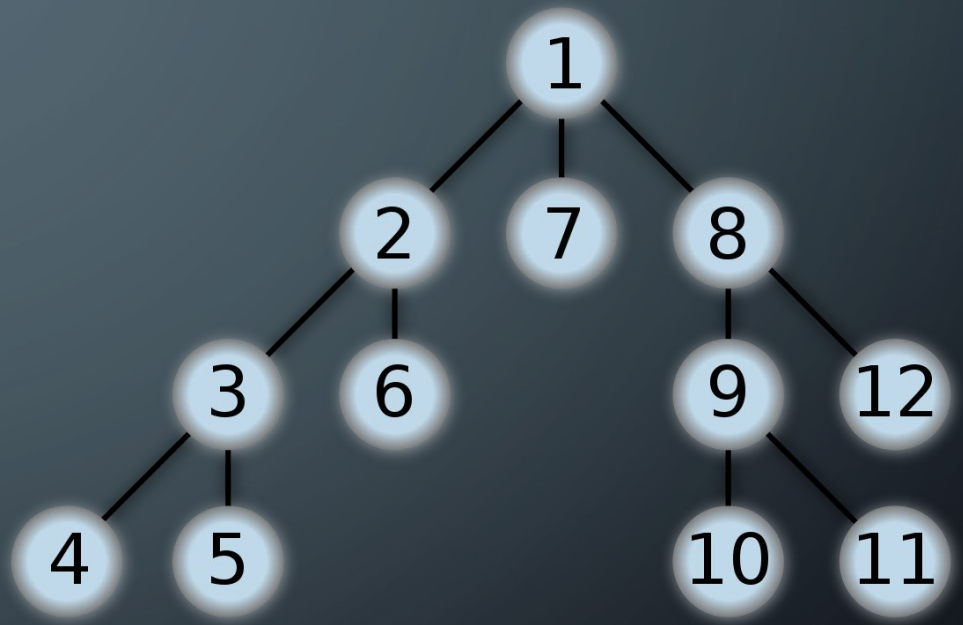
- Common applications of recursion

RECURSION IS A PROBLEM SOLVING TOOL

- Many problems are conveniently represented by a graph
  - E.g. Trees, grids, lists
  - A network of routers
  - A grid representing a physical space
  - A graph of states and connections among them
- If we can model a problem by a graph, we can apply recursion to search/traverse the graph

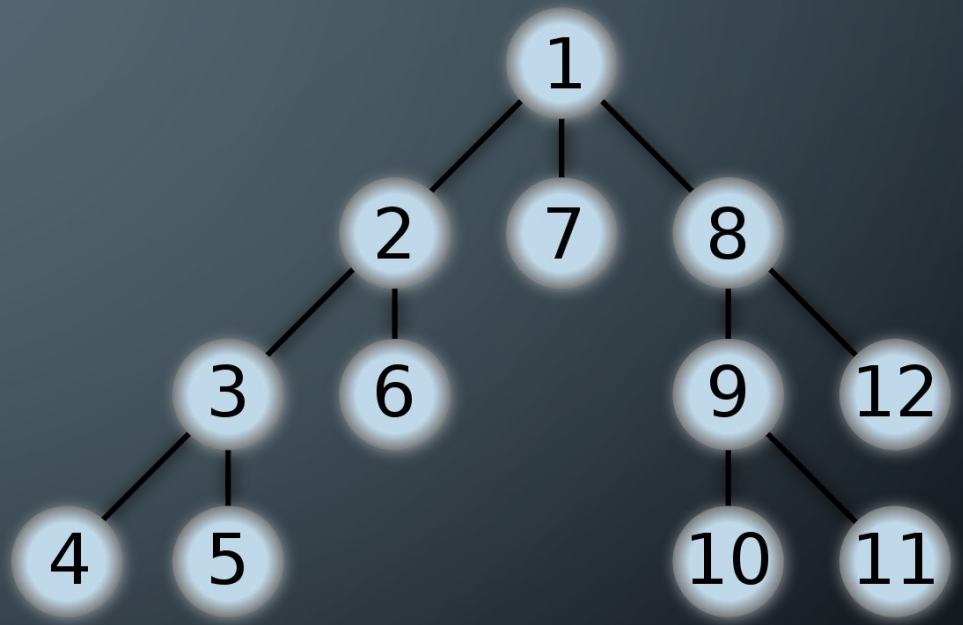
SPACE SEARCH

- We have the root of our graph, and we want to find a specific node
- How could we search this?
  - Randomly pick a branch (not great…)
  - Look at all the connected nodes (breadth-first)
  - Drill down as far as we can go, then backtrack (depth-first)



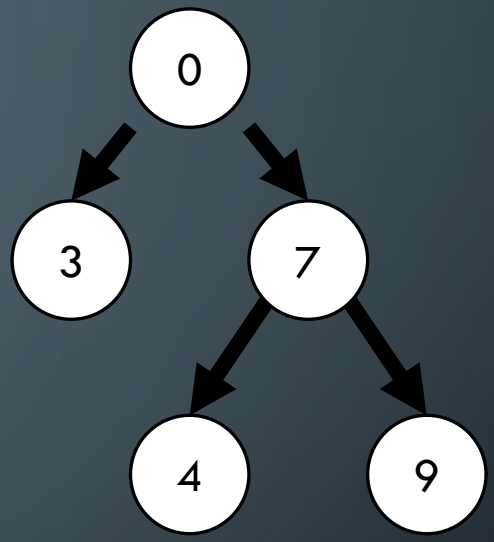
SPACE SEARCH & RECURSION

- At each node we reach, we make a decision about which branch to take next
- What we actually do at each node is the same—this is our opportunity to apply recursion



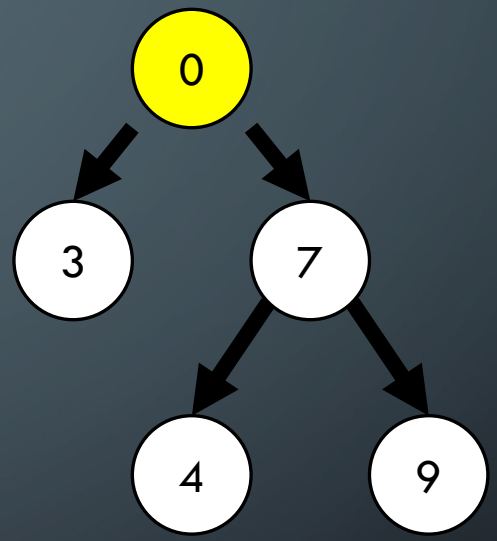
DEPTH FIRST SEARCH

- At each node, we do the following:
  - If this is the node we are looking for, we are done
  - Otherwise, we pick the first branch we have not visited, and recurse using that branch as the new root
  - If we check all branches and we have not found the node, we return false



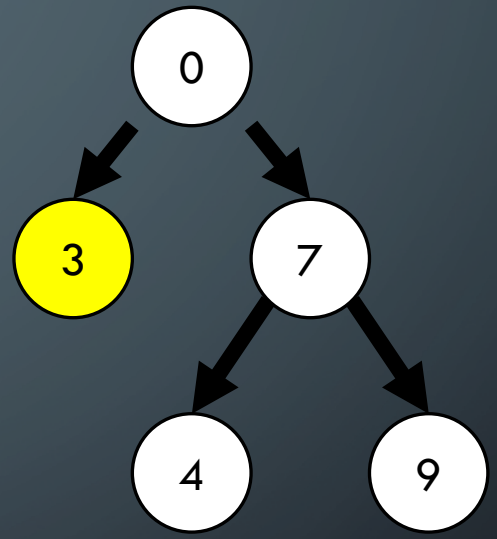
DEPTH FIRST SEARCH

- Finding value 4
- Current node: Root
- Current value: 0
- Children: Left, Right
- Check here: No match
- Check left: recurse left child:



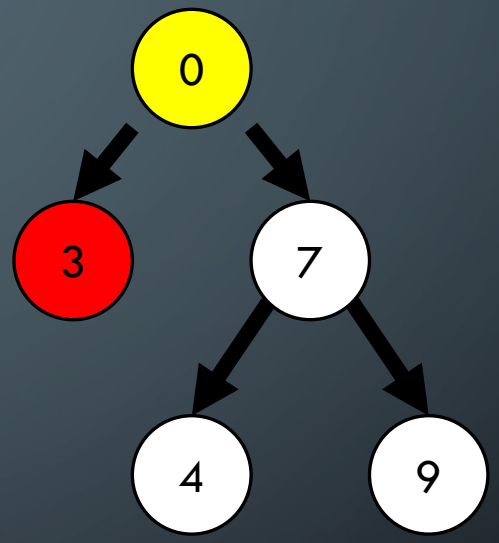
DEPTH FIRST SEARCH

- Finding value 4
- Current node: 3
- Current value: 3
- Children: None
- Check here: No match
- No children: return False



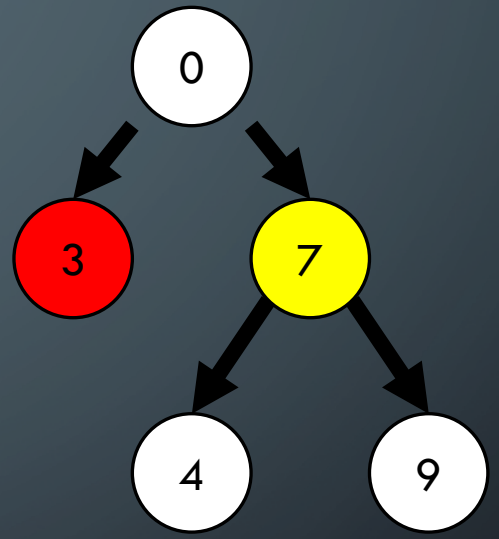
DEPTH FIRST SEARCH

- Finding value 4
- Current node: Root
- Current value: 0
- Children: Left, Right
- Check here: No match
- Check left: recurse left child: False
- Check right: recurse right child:



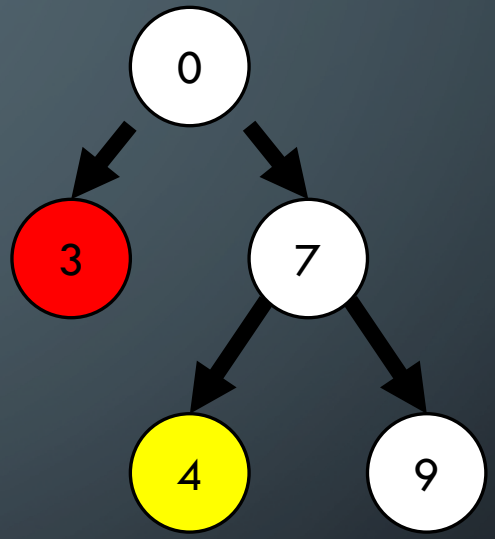
DEPTH FIRST SEARCH

- Finding value 4
- Current node: 7
- Current value: 7
- Children: left, right
- Check here: No match
- Check left: recurse left child:



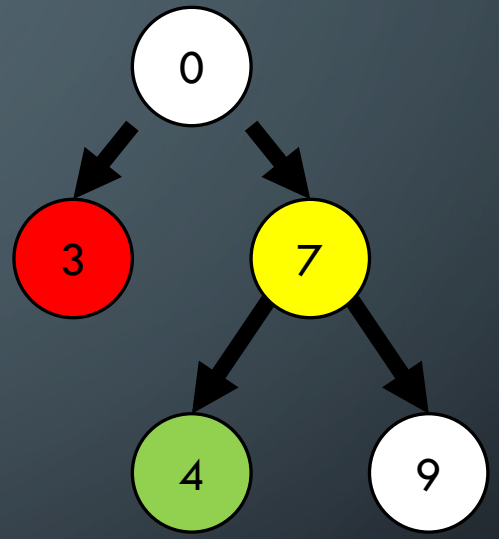
DEPTH FIRST SEARCH

- Finding value 4
- Current node: 4
- Current value: 4
- Children: None
- Check here: Match, return True



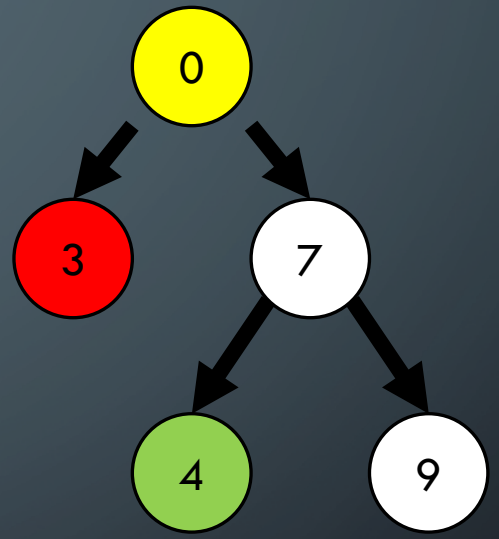
DEPTH FIRST SEARCH

- Finding value 4
- Current node: 7
- Current value: 7
- Children: left, right
- Check here: No match
- Check left: recurse left child: True
- Return True



DEPTH FIRST SEARCH

- Finding value 4
- Current node: Root
- Current value: 0
- Children: Left, Right
- Check here: No match
- Check left: recurse left child: False
- Check right: recurse right child: True
- Return True

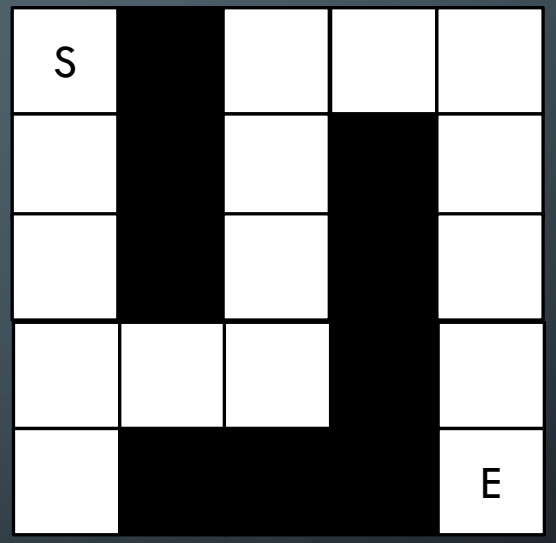


DEPTH FIRST SEARCH: PSEUDOCODE

```
bool dfs(node){
      if current node is search value{
            return True
      }
      else if dfs(left child){ return True )
      else if dfs(right child){ return True )
      return False
}
```

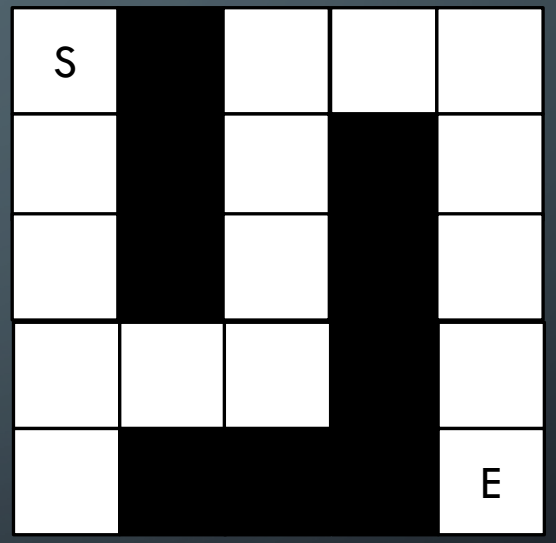
ANOTHER APPLICATION: PATHFINDING

- At each position, we can only travel on open tiles
- We can't walk through walls
- We wish to find the end tile



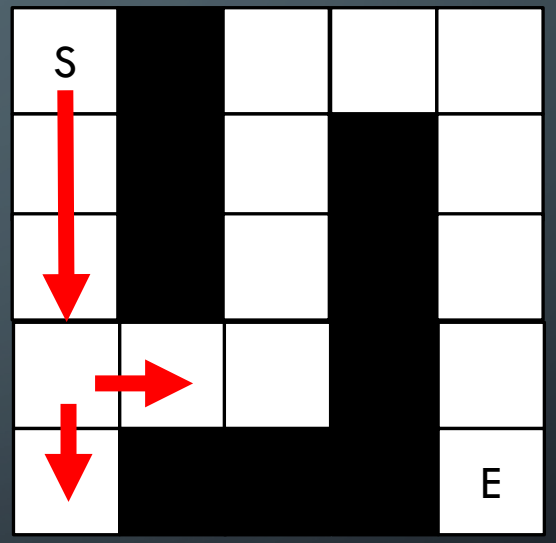
PATHFINDING

- What do we recurse over?
  - We decide where to move next
- What is the terminating condition?
  - We reach the end, or we have no valid moves
- How do we ensure progress?
  - We must track where we have been



PATHFINDING

- Steps 1-3 are straightforward: there is only one way to go
- Step 4 introduces a choice: we now have to track two possible paths taken
- Why do we have to track these separately?



BACKTRACKING

- Whenever we reached a dead end, we ‘backed up’ a level of recursion
- This concept is called backtracking
  - we enumerate a number of possible solutions, and try them independently
  - Whenever a possible solution fails, we backtrack to the previous step and try the next potential solution
- This comes up over an over again in recursive algorithms, but it’s not free

ALGEBRAIC EXPRESSIONS (INFIX EXPRESSION)

- Compiler must recognize and evaluate algebraic expressions

```
y = x + z * (w / k + z * (7 * 6));
```

  - Determine if legal expression
  - If legal, evaluate expression

KINDS OF ALGEBRAIC EXPRESSIONS

- Prefix expression
  - Operator appears before its operands

$a + b$ equivalent to $+ a\ b$

- Postfix expressions
  - Operator appears after its operands

$a + b$ equivalent to $a\ b +$

PREFIX NOTATION

- + 300 – 11 + 5 4
  - + has two arguments 300 and … oops, there was another operator –
    - – has two arguments 11 and … oops, there was another operator +
      - + has two arguments 5 and 4 so that makes 9
    - – has two arguments 11 and 9 so that makes 2
  - + has two arguments 300 and 2 so that makes 302
- Feels like recursion?

VERIFYING PREFIX EXPRESSIONS

- Assume we are compiling instructions for our machine:
  - How can we verify that an expression is correct?
  - What are the recursive steps?
  - What is our termination condition?

+ 300 – 11 + 5 4

THE PREFIX GRAMMAR

- A grammar defines all the legal statements in a language
- We use it here to understand the recursive behavior
- Notation:
  - '|' represents the OR operation
  - (characters)* represents any combination of the enclosed characters

<prefix> = <operand> | <operator> <prefix> <prefix>

<operator> = '+' | '-' | '*' | '/'

<operand> = (0|1|2|3|4|5|6|7|8|9)*

THE PREFIX GRAMMAR

- A grammar defines all the legal statements in a language
- We use it here to understand the recursive behavior
- Notation:
  - '|' represents the OR operation
  - (characters)* represents any combination of the enclosed characters

<prefix> = <operand> | <operator> <prefix> <prefix>

<operator> = '+' | '-' | '*' | '/'

<operand> = (0|1|2|3|4|5|6|7|8|9)*

- Recursive algorithm that recognizes whether string is a prefix expression
  - Check if first character is an operator
  - Remainder of string consists of two consecutive prefix expressions

EXAMPLE

/ * a b – c d

PSEUDOCODE: VALID PREFIX

```
int findPrefixEnd(string s, int startpos){
        if (startpos < 0 or startpos > s.length()-1) {return -1}
        char c = first char in s
        if isOperand( c ){
                return startpos
        }
        else if isOperator( c ){
                int thisEnd = findPrefixEnd(s, startpos + 1) //point X
                if (thisEnd >= 0){ return findPrefixEnd(s, thisEnd + 1) } // point Y
                return -1
        }
        return -1
}
```

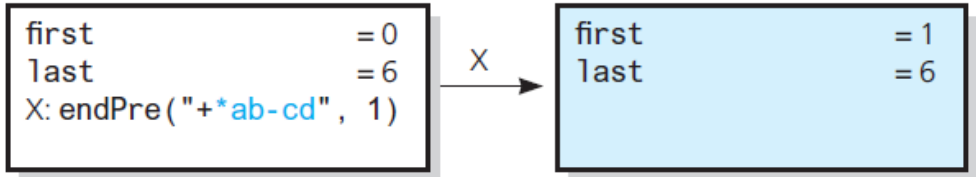
PSEUDOCODE: VALID EXPRESSION

```
boolean isValid(string s){
      char lastChar = findPrefixEnd(s, 0)
      if ( lastChar is positive AND
             lastChar is the end of the string){
            return true
      }
      return False
}
```

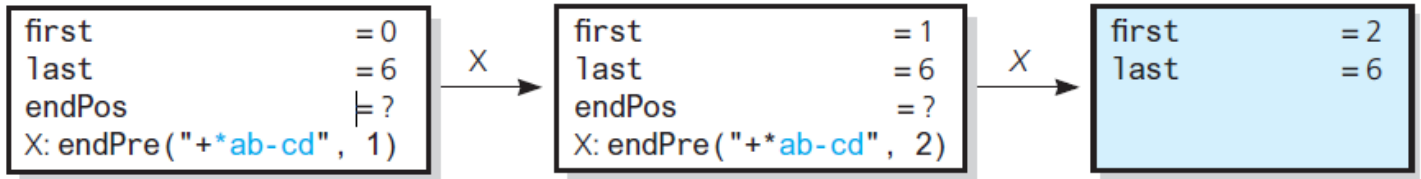
PREFIX EXPRESSIONS

The initial call **endPre("+*ab-cd", 0)** is made, and **endPre** begins execution:

First character of **strExp** is +, so at point *X*, a recursive call is made and the new invocation of **endPre** begins execution:

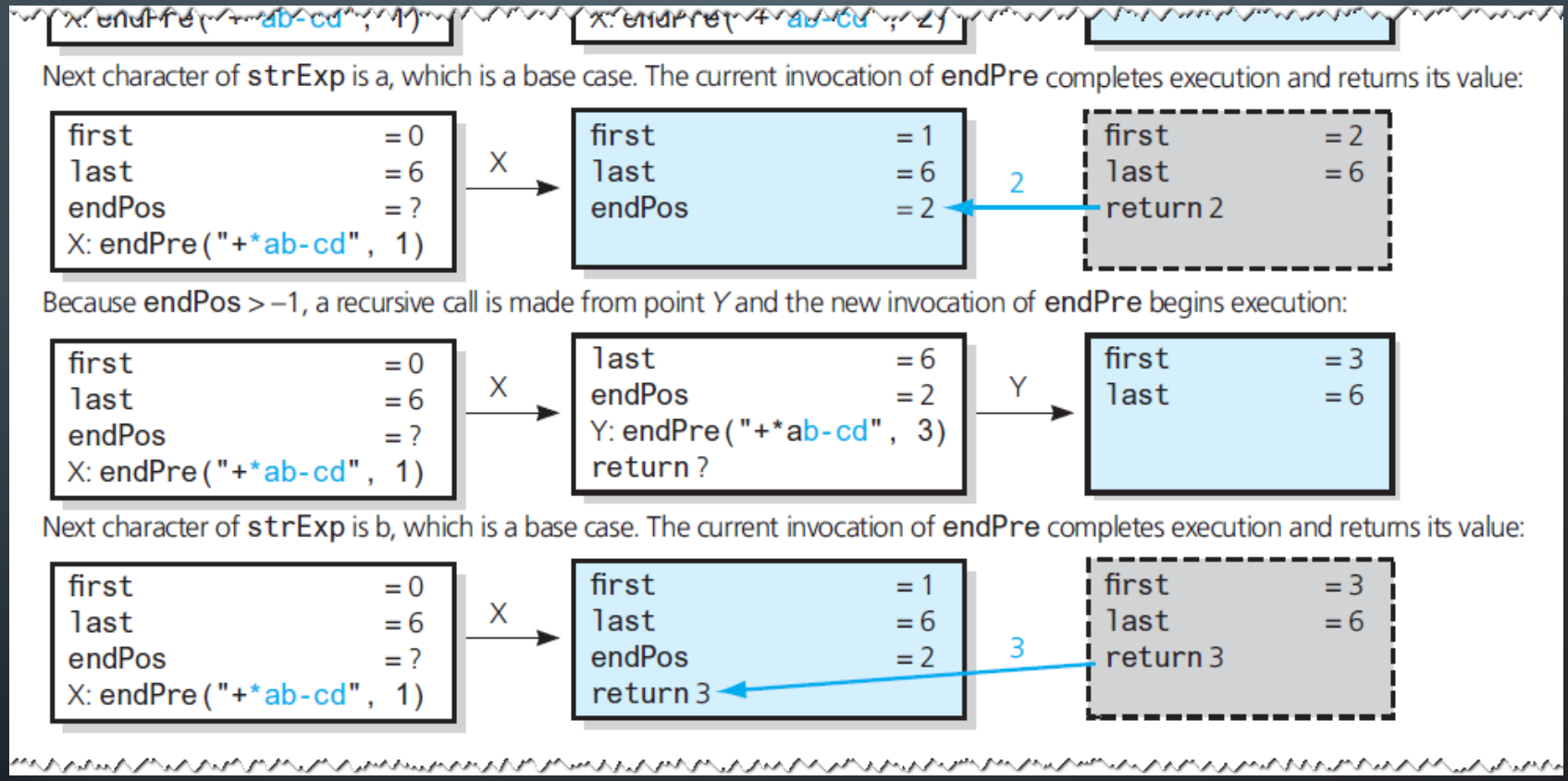


Next character of **strExp** is *, so at point *X*, a recursive call is made and the new invocation of **endPre** begins execution:

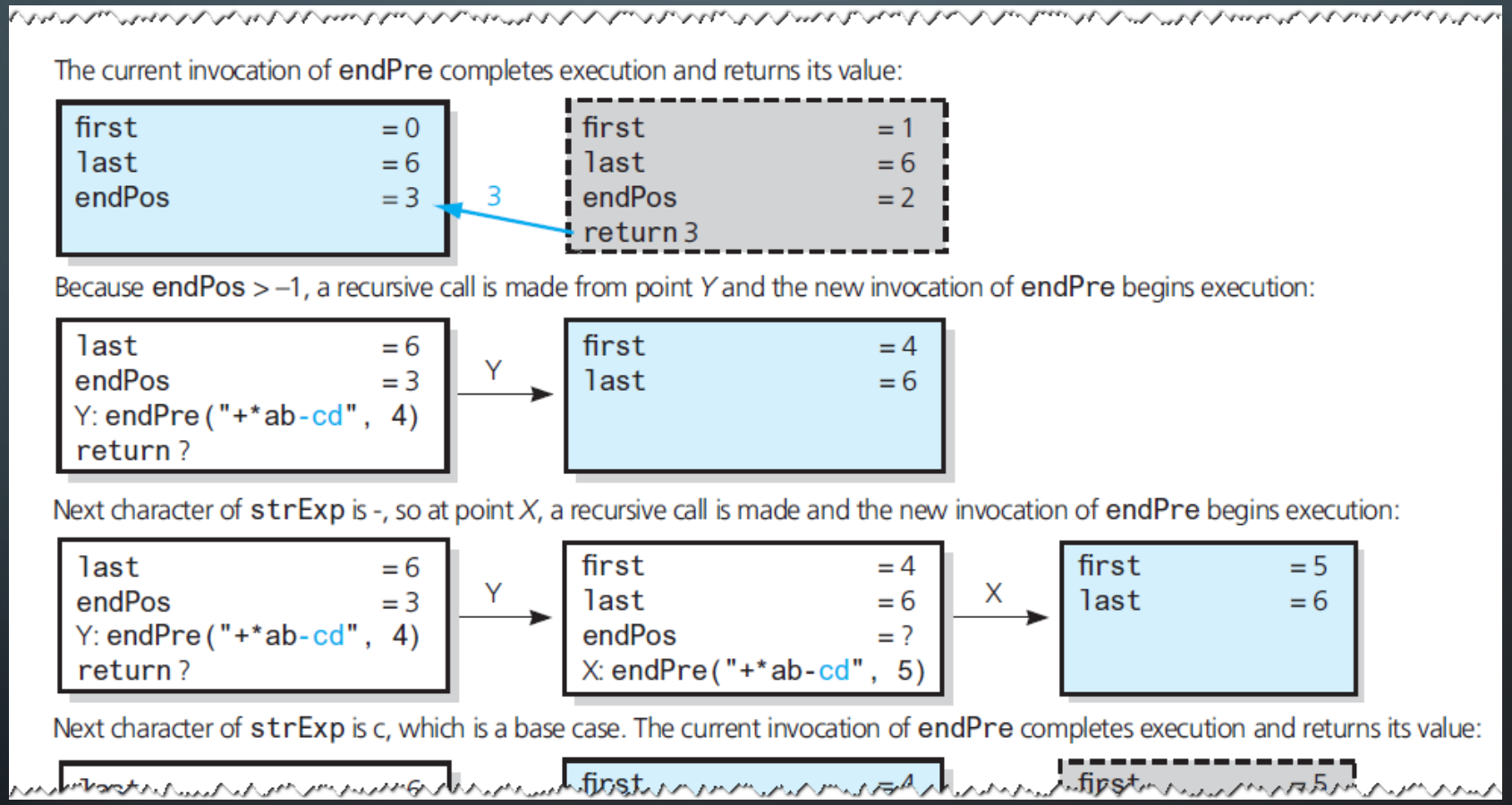


Next character of **strExp** is a, which is a base case. The current invocation of **endPre** completes execution and returns its value

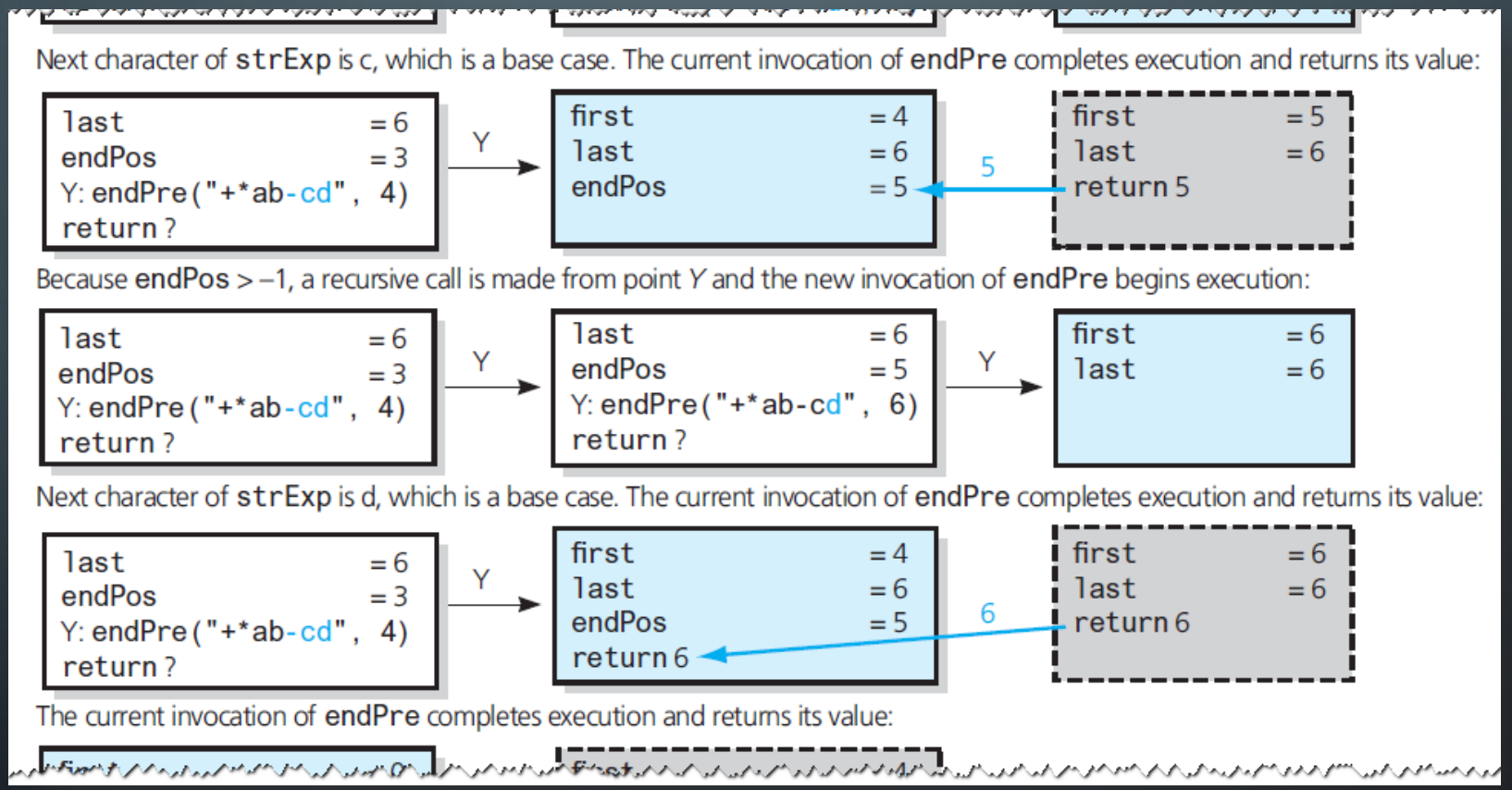
PREFIX EXPRESSIONS



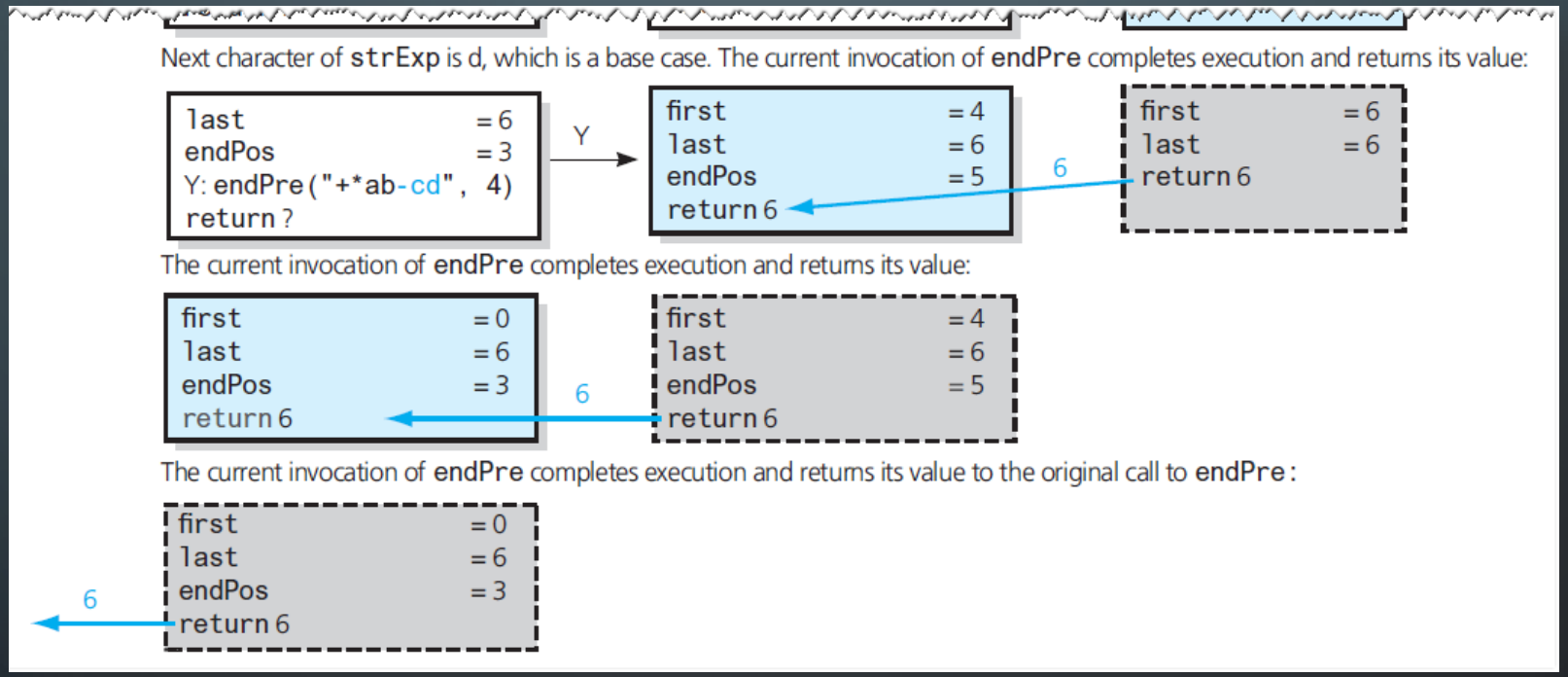
PREFIX EXPRESSIONS



PREFIX EXPRESSIONS



PREFIX EXPRESSIONS



EXAMPLES

- Are these valid expressions?

(A) / + a c d – e g

(B) * + a b c

(C) + a c – b – f

Assignment/Homework

- Reading: Carrano pp. 191-215, 225-239, 241-251
- ICE3 due Today
- HW3 and ICE4 due on Tuesday