Recursion as a Problem-Solving Technique

Chapter 5

Defining Languages

- A language a set of strings of symbols from a finite alphabet.
- Consider the C++ language

$$C{+}{+}Programs = \{\text{string } s : s \text{ is a syntactically correct C++ program}\}$$

- The set of algebraic expressions forms a language

$$AlgebraicExpressions = \{\text{string } s : s \text{ is an algebraic expression}\}$$

- A grammar states the rules of a language.

The Basics of Grammars

- A grammar uses several special symbols
  - x | y means x or y .
  - x y (and sometimes x • y) means x followed by y .
  - < word > means any instance of word , where word is a symbol that must be defined elsewhere in the grammar.

The Basics of Grammars

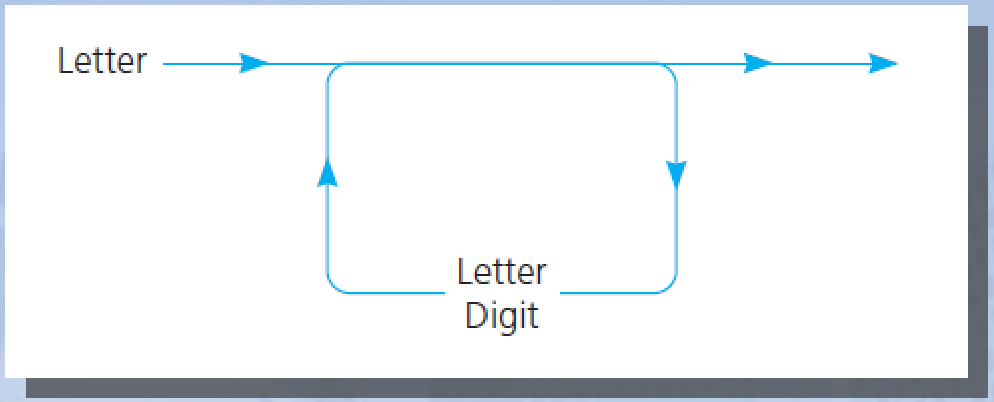


FIGURE 5-1 A syntax diagram for C++ identifiers

The Basics of Grammars

```
// Returns true if s is a legal C++ identifier; otherwise returns false.
isId(s: string): boolean
{
   if (s is of length 1)                          // Base case
      if (s is a letter)
         return true
      else
         return false
   else if (the last character of s is a letter or a digit)
      return isId(s minus its last character)     // Point X
   else
      return false
}
```

Pseudocode for a recursive valued function that determines whether a string is in the language C++Identifier s

The Basics of Grammars

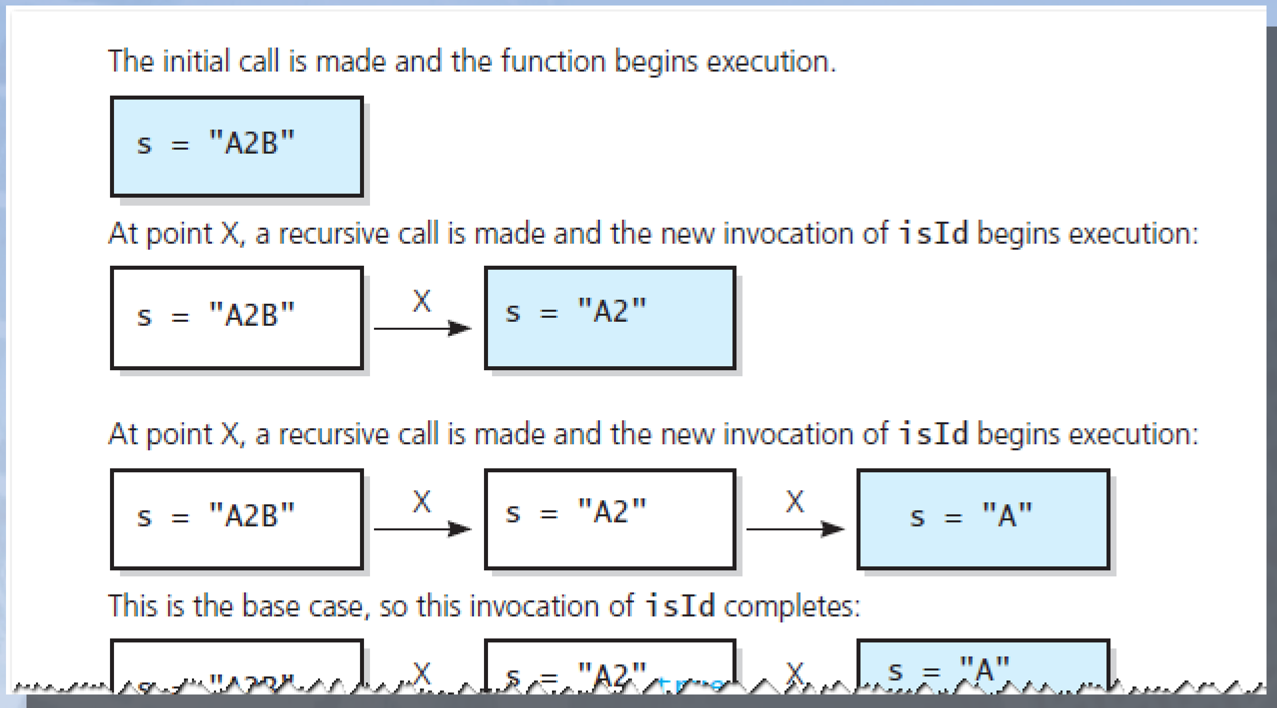


FIGURE 5-2 Trace of isId("A2B")

The Basics of Grammars

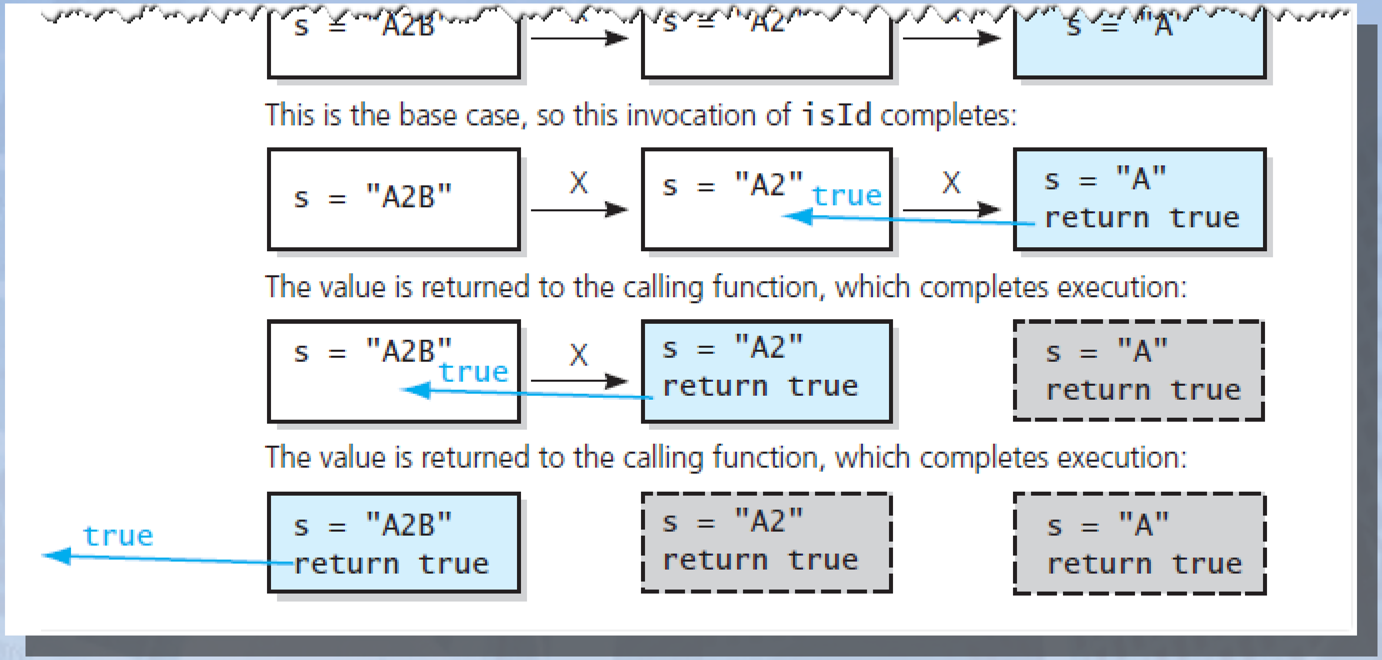


FIGURE 5-2 Trace of isId("A2B")

Two Simple Languages

- Palindromes

$$Palindromes = \{\text{string } s : s \text{ reads the same left to right as right to left}\}$$

- Recursive definition of palindrome
  - The first and last characters of *s* are the same
  - *s* minus its first and last characters is a palindrome

$$<pal> \; = \text{empty string} \mid <ch> \mid \text{a} <pal> \text{a} \mid \text{b} <pal> \text{b} \mid \ldots \mid \text{Z} <pal> \text{Z}$$
$$<ch> \; = \text{a} \mid \text{b} \mid \ldots \mid \text{z} \mid \text{A} \mid \text{B} \mid \ldots \mid \text{Z}$$

- G
palindromes

Two Simple Languages

- Strings of the form A^nB^n

$$AnBn = \{\text{string } s : s \text{ is of the form } A^nB^n \text{ for some } n \geq 0\}$$

- Grammar for the language A^nB^n is

$$\langle legal\_word \rangle = \text{empty string} \mid A \langle legal\_word \rangle B$$

Algebraic Expressions

- 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

- Infix expressions
  - Every binary operator appears between its operands
- This convention necessitates …
  - Associativity rules
  - Precedence rules
  - Use of parentheses

$$a + b * c$$

$$(a + b) * c$$

Kinds of Algebraic Expressions

- Prefix expression
  - Operator appears before its operands

$$a + b \text{ equivalent to } + a\ b$$

- Postfix expressions
  - Operator appears after its operands

$$a + b \text{ equivalent to } a\ b\ +$$

Prefix Expressions

- Grammar that defines language of all prefix expressions

$$<prefix> \; = \; <identifier> \,|\, <operator> <prefix> <prefix>$$
$$<operator> \; = \; + \,|\, - \,|\, * \,|\, /$$
$$<identifier> \; = \; a \,|\, b \,|\, \ldots \,|\, z$$

- 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

Prefix Expressions

```
// Finds the end of a prefix expression, if one exists.
// Precondition: The substring of strExp from the index first through the end of
// the string contains no blank characters.
// Postcondition: Returns the index of the last character in the prefix expression that
// begins at index first of strExp, or –1 if no such prefix expression exists.
endPre(strExp: string, first: integer): integer
{
   last = strExp.length() - 1
   if (first < 0 or first > last)
      return -1

   ch = character at position first of strExp
   if (ch is an identifier)
      return first            // Index of last character in simple prefix expression
   else if (ch is an operator)
   {
```

endPre determines the end of a prefix expression

Prefix Expressions

```
      return first        // Index of last character in simple prefix expression
   else if (ch is an operator)
   {
      // Find the end of the first prefix expression
      endPos = endPre(strExp, first + 1)    // Point X
     // If the end of the first prefix expression was found, find the end of the second
     // prefix expression
     if (endPos > -1)
        return endPre(strExp, endPos + 1) // Point Y
     else
        return -1
   }
   else
     return -1
}
```

endPre determines the end of a prefix expression

Prefix Expressions

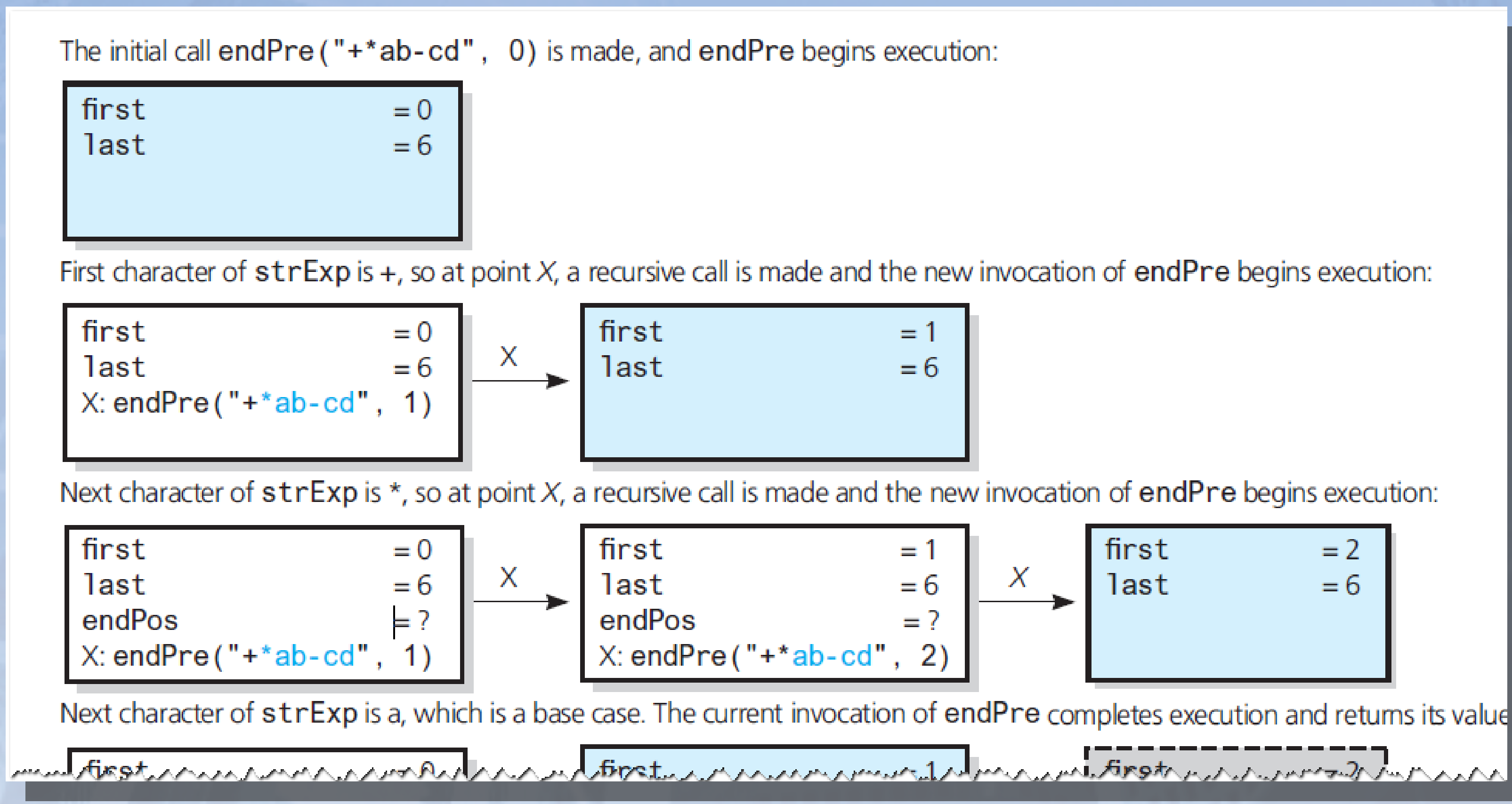


FIGURE 5-3 Trace of endPre("+*ab-cd", 0)

Prefix Expressions

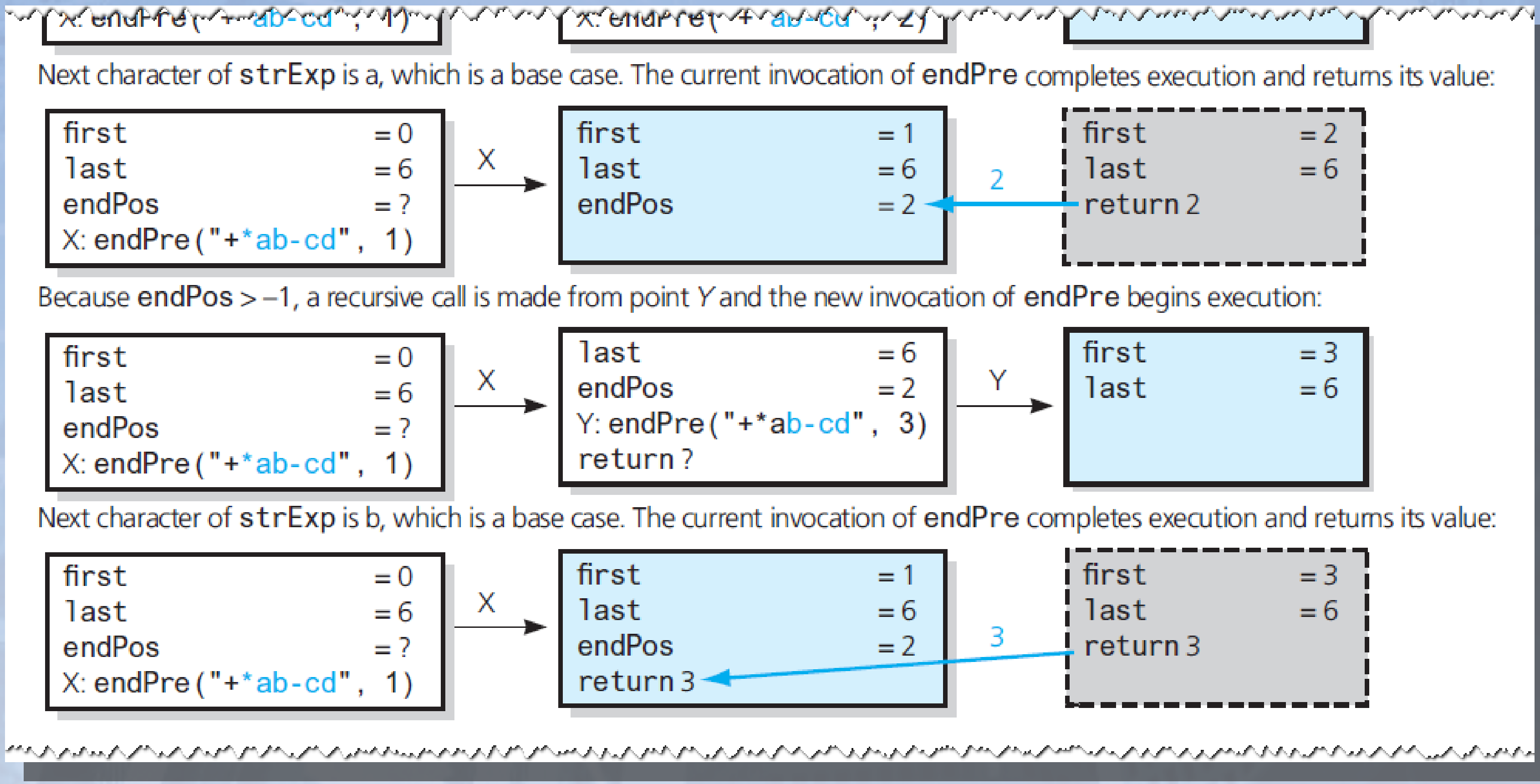


FIGURE 5-3 Trace of endPre("+*ab-cd", 0)

Prefix Expressions

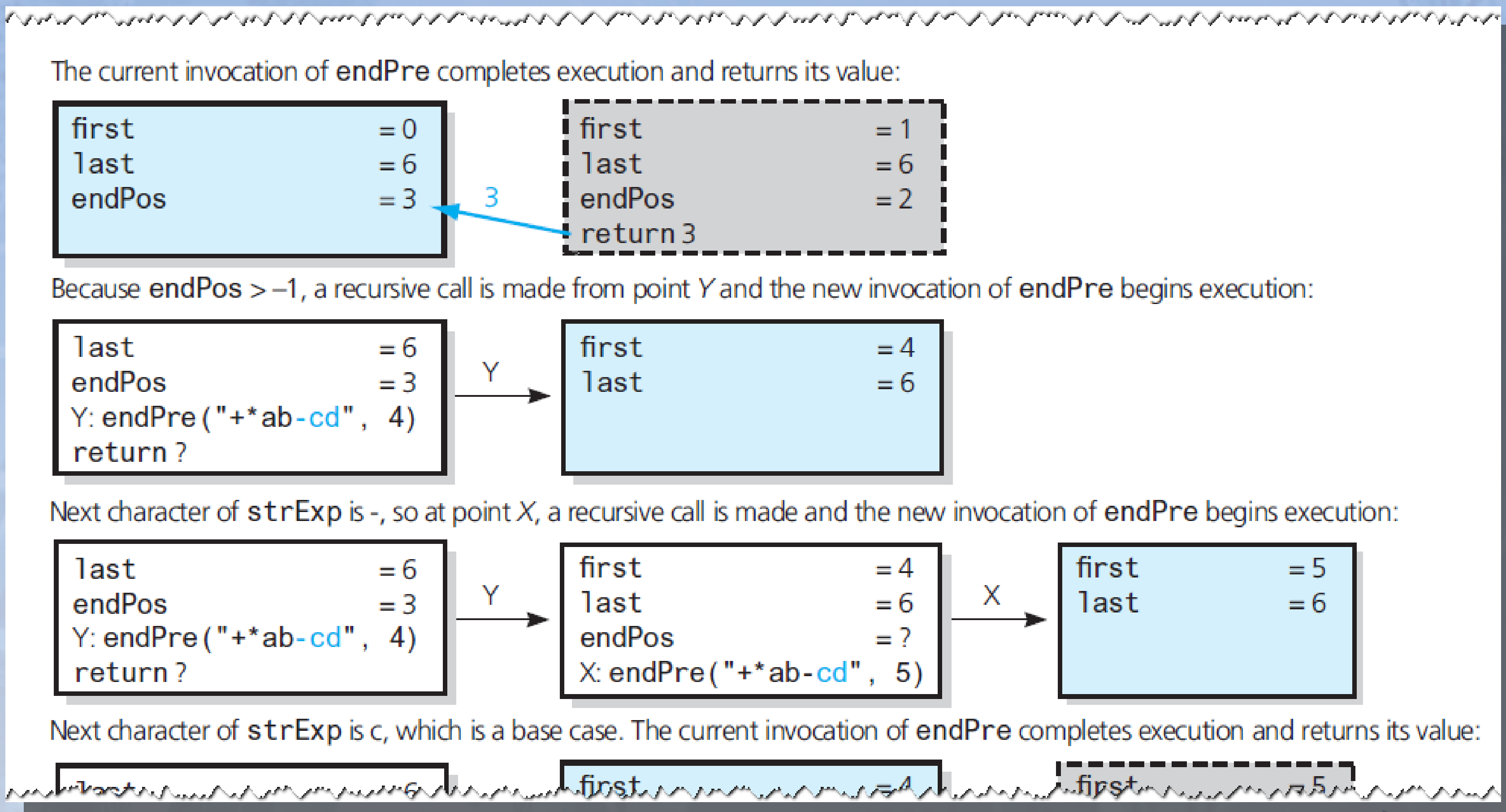


FIGURE 5-3 Trace of endPre("+*ab-cd", 0)

Prefix Expressions

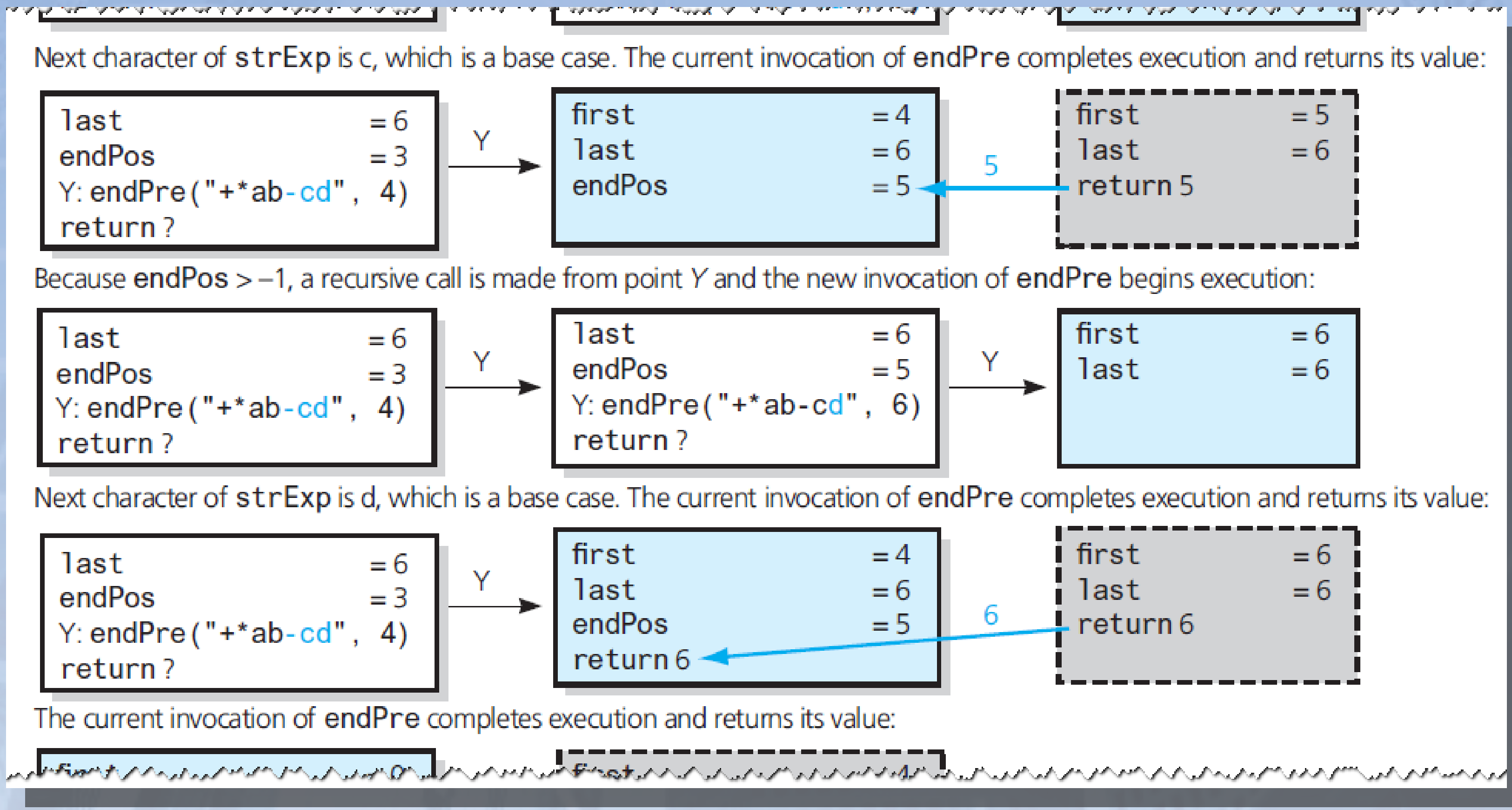


FIGURE 5-3 Trace of endPre("+*ab-cd", 0)

Prefix Expressions

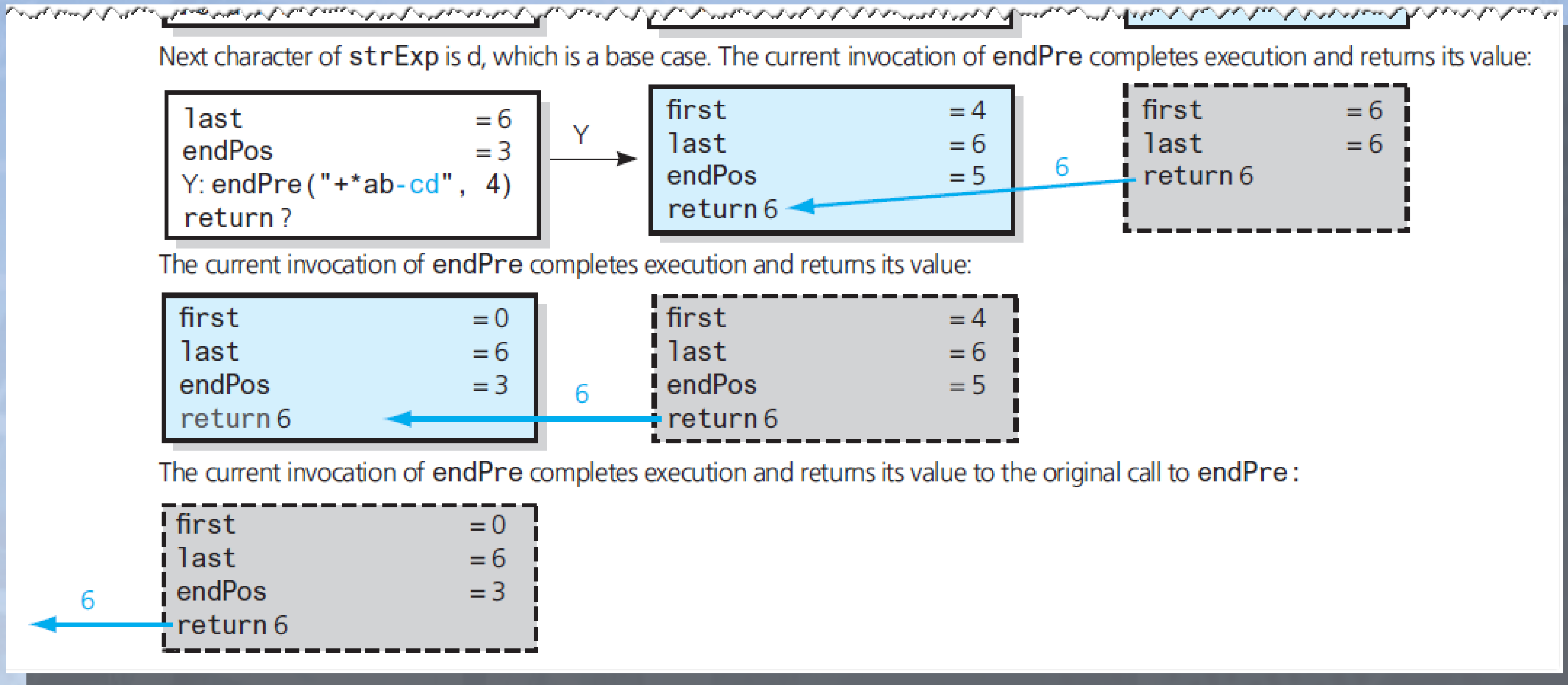


FIGURE 5-3 Trace of endPre("+*ab-cd", 0)

Prefix Expressions

```
// Sees whether an expression is a prefix expression.
// Precondition: strExp contains a string with no blank characters.
// Postcondition: Returns true if the expression is in prefix form; otherwise returns false.
isPrefix(strExp: string): boolean
{
   lastChar = endPre(strExp, 0)
   return (lastChar >= 0) and (lastChar == strExp.length() - 1)
}
```

A recognition algorithm for prefix expressions

Prefix Expressions

```
// Returns the value of a given prefix expression.
// Precondition: strExp is a string containing a valid prefix expression with no blanks.
evaluatePrefix(strExp: string): float
{
   strLength = the length of strExp
   if (strLength == 1)
      return value of the identifier // Base case—single identifier
   else
   {
      op = strExp[0]                 // strExp begins with an operator

      // Find the end of the first prefix expression—will be the first operand
      endFirst = endPre(strExp, 1)

      // Recursively evaluate this first prefix expression
```

An algorithm to evaluate a prefix expression

Prefix Expressions

```
        endFirst = endPre(strExp, 1)

        // Recursively evaluate this first prefix expression
        operand1 = evaluatePrefix(strExp[1..endFirst]);

        // Recursively evaluate the second prefix expression—will be the second operand
        endSecond = strLength - endFirst + 1
        operand2 = evaluatePrefix(strExp[endFirst + 1..endSecond])

        // Evaluate the prefix expression
        return operand1 op operand2
    }
}
```

An algorithm to evaluate a prefix expression

Postfix Expressions

Grammar that defines the language of all postfix expressions

$$<postfix> = <identifier> | <postfix><postfix><operator>$$
$$<operator> = + | - | * | /$$
$$<identifier> = a | b | \ldots | z$$

An algorithm that converts a prefix expression to postfix form

```
if (exp is a single letter)
    return exp
else
    return postfix(prefix1) • postfix(prefix2) • <operator>
```

Postfix Expressions

```
// Converts a prefix expression to postfix form.
// Precondition: The string preExp is a valid prefix expression with no blanks.
// Postcondition: Returns the equivalent postfix expression.
convertPreToPost(preExp: string): string
{
   preLength = the length of preExp
   ch = first character in preExp
   postExp = an empty string

   if (ch is a lowercase letter)
      // Base case—single identifier
      postExp = postExp • ch              // Append to end of postExp
   else // ch is an operator
   {
      // pre has the form <operator> <prefix1> <prefix2>
      endFirst = endPre(preExp, 1)        // Find the end of prefix1

      // Recursively convert prefix1 into postfix form
      postExp = postExp • convert(preExp[1..endFirst])

      // Recursively convert prefix2 into postfix form
      postExp = postExp • convert(preExp[endFirst + 1..preLength - 1))

      postExp = postExp • ch              // Append the operator to the end of postExp
   }
   return postExp
}
```

Recursive algorithm that converts a prefix expression to postfix form

Fully Parenthesized Expressions

- Grammar for language of fully parenthesized algebraic expressions

$$
\begin{aligned}
&<\textit{infix}> \; = \; <\textit{identifier}> \mid (<\textit{infix}> <\textit{operator}> <\textit{infix}>) \\
&<\textit{operator}> \; = \; + \mid - \mid * \mid / \\
&<\textit{identifier}> \; = \; a \mid b \mid \ldots \mid z
\end{aligned}
$$

- Most programming languages support definition of algebraic expressions
  - Includes both precedence rules for operators and rules of association

Backtracking

- Strategy for guessing at a solution and …
  - Backing up when an impasse is reached
  - Retracing steps in reverse order
  - Trying a new sequence of steps
- Combine recursion and backtracking to solve problems

Searching for an Airline Route

- Must find a path from some point of origin to some destination point
- Program to process customer requests to fly
  - From some origin city
  - To some destination city
- Use three input text files
  - names of cities served
  - Pairs of city names, flight origins and destinations
  - Pairs of names, request origins, destinations

Searching for an Airline Route

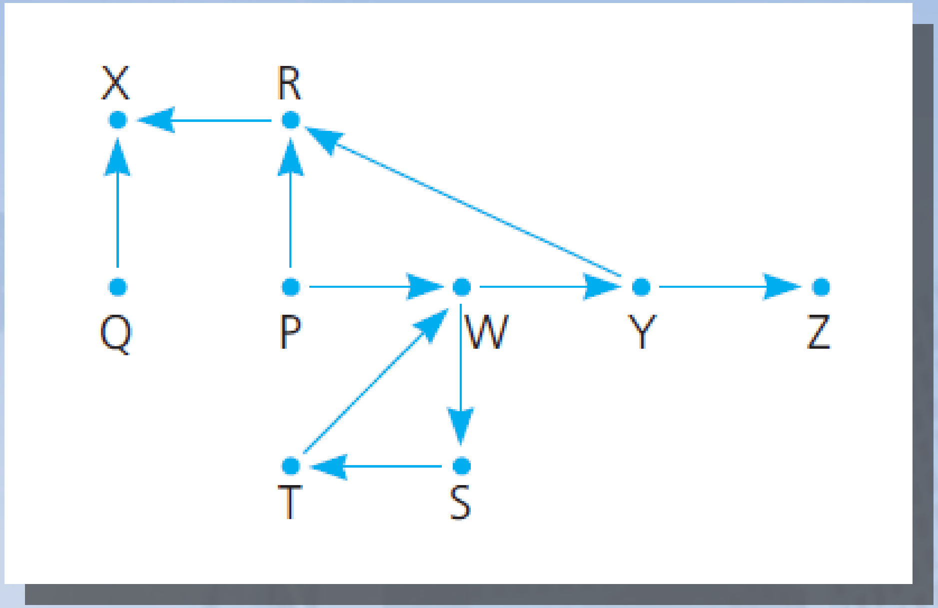


FIGURE 5-4 Flight map for HPAir

Searching for an Airline Route

```
To fly from the origin to the destination
{
   Select a city C adjacent to the origin
   Fly from the origin to city C
   if (C is the destination city)
      Terminate— the destination is reached
   else
      Fly from city C to the destination
}
```

A recursive search strategy

Searching for an Airline Route

- Possible outcomes of exhaustive search strategy
  1. Reach destination city, decide possible to fly from origin to destination
  2. Reach a city, *C* from which no departing flights
  3. You go around in circles
- Use backtracking to recover from a wrong choice (2 or 3)

Searching for an Airline Route

```
// Discovers whether a sequence of flights from originCity to destinationCity exists.
searchR(originCity: City, destinationCity: City): boolean
{
   Mark originCity as visited
   if (originCity is destinationCity)
      Terminate—the destination is reached
   else
      for (each unvisited city C adjacent to originCity)
         searchR(C, destinationCity)
}
```

Refinement of the recursive search algorithm

Searching for an Airline Route

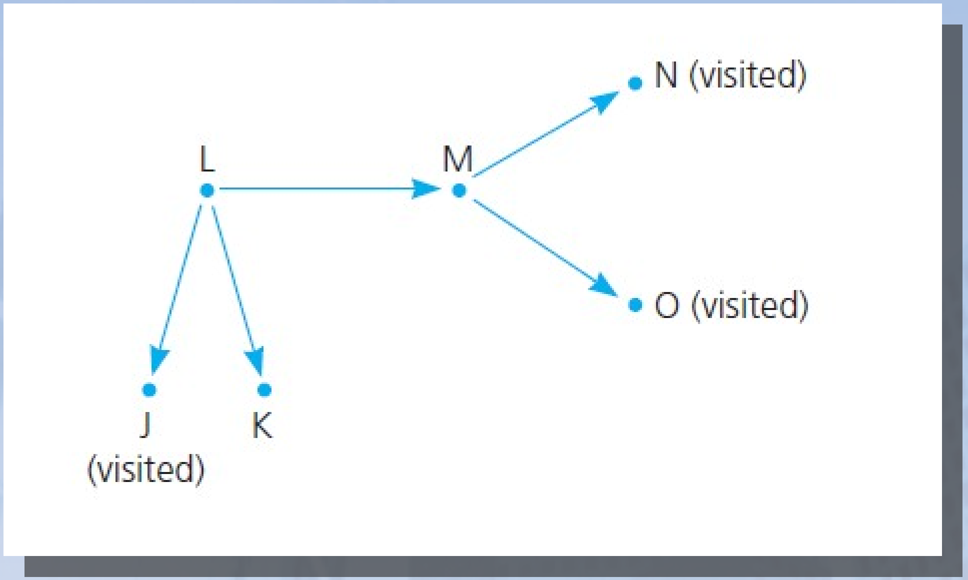


FIGURE 5-5 A piece of a flight map

Searching for an Airline Route

```
// Reads flight information into the flight map.
+readFlightMap(cityFileName: string, flightFileName: string): void

// Displays flight information.
+displayFlightMap(): void

// Displays the names of all cities that HPAir serves.
+displayAllCities(): void

// Displays all cities that are adjacent to a given city.
+displayAdjacentCities(aCity: City): void

// Marks a city as visited.
+markVisited(aCity: City): void

// Clears marks on all cities.
```

ADT flight map operations

Searching for an Airline Route

```
+markVisited(aCity: City): void

// Clears marks on all cities.
+unvisitAll(): void

// Sees whether a city was visited.
+isVisited(aCity: City): boolean

// Inserts a city adjacent to another city in a flight map.
+insertAdjacent(aCity: City, adjCity: City): void

// Returns the next unvisited city, if any, that is adjacent to a given city.
// Returns a sentinel value if no unvisited adjacent city was found.
+getNextCity(fromCity: City): City

// Tests whether a sequence of flights exists between two cities.
+isPath(originCity: City, destinationCity: City): boolean
```

ADT flight map operations

Searching for an Airline Route

```
/** Tests whether a sequence of flights exists between two cities.
 @pre  originCity and destinationCity both exist in the flight map.
 @post  Cities visited during the search are marked as visited
     in the flight map.
 @param originCity  The origin city.
 @param destinationCity  The destination city.
 @return  True if a sequence of flights exists from originCity
    to destinationCity; otherwise returns false. */
bool Map::isPath(City originCity, City destinationCity)
{
   // Mark the current city as visited
   markVisited(originCity);

   bool foundDestination = (originCity == destinationCity);
   if (!foundDestination)
   {
      // Try a flight to each unvisited city
```

C++ implementation of searchR

Searching for an Airline Route

```cpp
   {
      // Try a flight to each unvisited city
      City nextCity = getNextCity(originCity);
      while (!foundDestination && (nextCity != NO_CITY))
      {
         foundDestination = isPath(nextCity, destinationCity);
         if (!foundDestination)
            nextCity = getNextCity(originCity);
      }  // end while
   }  // end if

   return foundDestination;
}  // end isPath
```

C++ implementation of searchR

Searching for an Airline Route

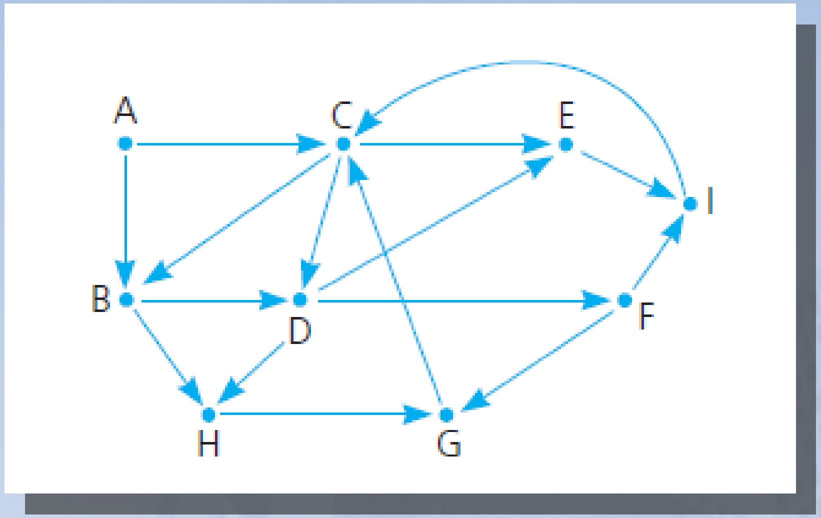


FIGURE 5-6 Flight map for Checkpoint Question 6

The Eight Queens Problem

- Chessboard contains 64 squares
  - Form eight rows and eight column
- Problem asks you to place eight queens on the chessboard …
  - So that no queen can attack any other queen.
- Note there are 4,426,165,368 ways to arrange eight queens on a chessboard
  - Exhausting to check all possible ways

The Eight Queens Problem

- However, each row and column can contain exactly one queen.
  - Attacks along rows or columns are eliminated, leaving only 8! = 40,320 arrangements
- Solution now more feasible

The Eight Queens Problem

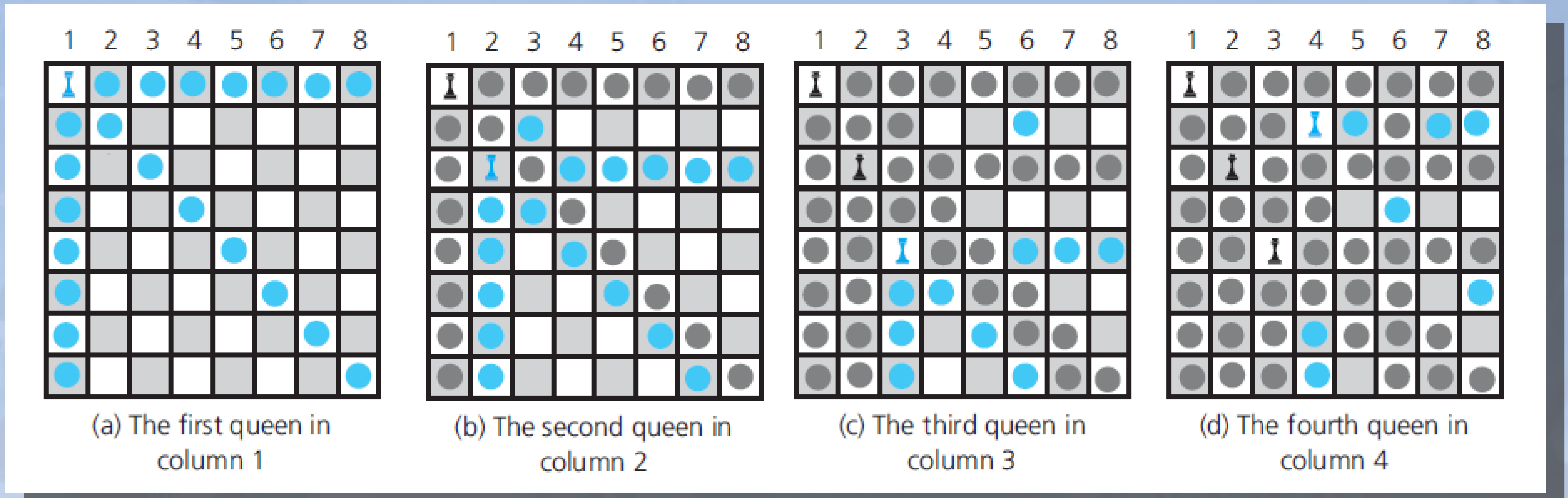
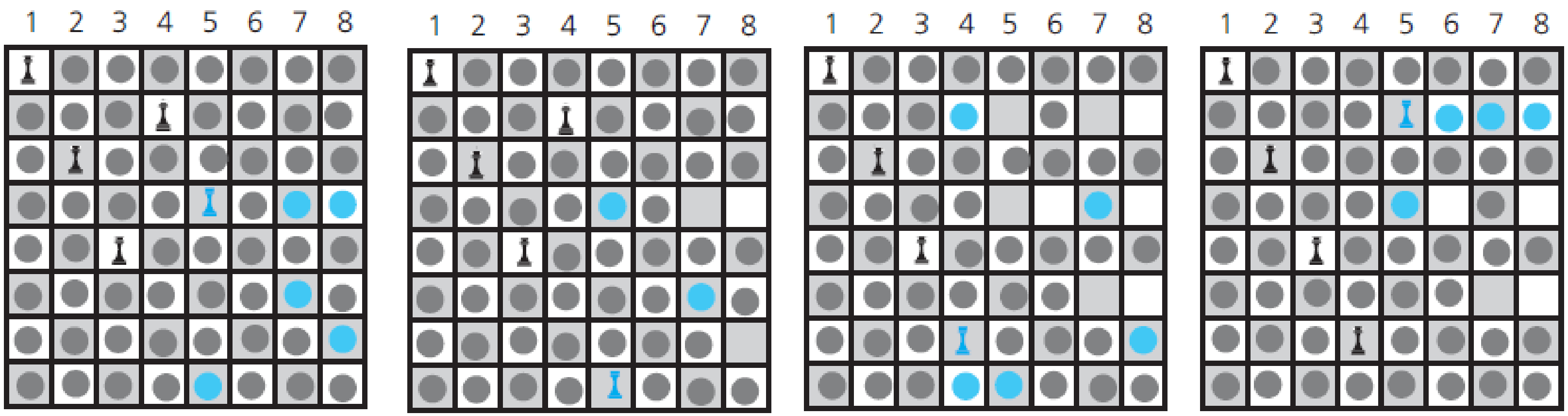


FIGURE 5-7 Placing one queen at a time in each column …

The Eight Queens Problem



(e) The five queens can attack all of column 6

(f) Backtracking to column 5 to try another square for the queen

(g) Backtracking to column 4 to try another square for the queen

(h) Considering column 5 again

FIGURE 5-7 Placing one queen at a time in each column …

The Eight Queens Problem

```
// Places queens in eight columns.
placeQueens(queen: Queen, row: integer, column: integer): boolean
{
   if (column > BOARD_SIZE)
      The problem is solved
   else
   {
      while (unconsidered squares exist in the given column and
               the problem is unsolved)
      {
         Find the next square in the given column that is
          not under attack by a queen in an earlier column
         if (such a square exists)
         {
            Place a queen in the square
```

Pseudocode describes the algorithm for placing queens

The Eight Queens Problem

```
            Place a queen in the square

            // Try next column
            if (!placeQueens(a new queen, firstRow, column + 1))
            {
                // No queen is possible in the next column
                Delete the new queen
                Move the last queen that was placed on the board
                   to the next row in that column
            }
         else
         {
            // Delete the new queen
            return true
         }
      }
    }
  }
}
```

Pseudocode describes the algorithm for placing queens

The Eight Queens Problem

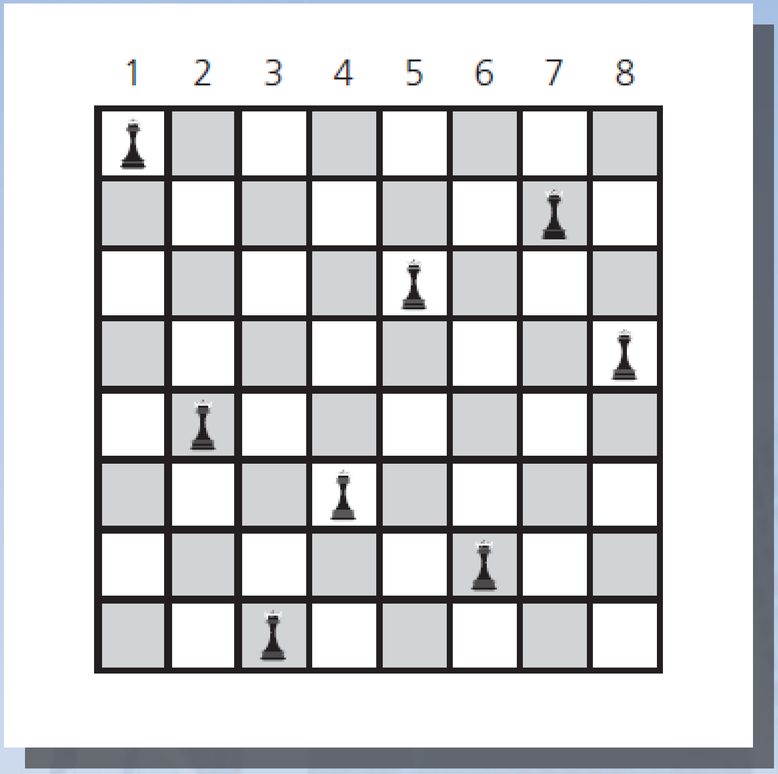


FIGURE 5-8 A solution to the Eight Queens problem

Relationship Between Recursion and Mathematical Induction

- Recursion solves a problem by
  - Specifying a solution to one or more base cases
  - Then demonstrating how to derive solution to problem of arbitrary size
  - From solutions to smaller problems of same type.
- Can use induction to prove
  - Recursive algorithm either is correct
  - Or performs certain amount of work

Correctness of Recursive Factorial Function

```
fact(n: integer): integer
{
   if (n is 0)
      return 1
   else
      return n * fact(n - 1)
}
```

Pseudocode describes recursive function that computes factorial

Correctness of Recursive Factorial Function

- Inductive hypothesis

$$factorial(k) = k! = k \times (k-1) \times (k-2) \times \ldots \times 2 \times 1$$

- Inductive conclusion

$$factorial(k+1) = (k+1) \times k \times (k-1) \times (k-2) \times \ldots \times 2 \times 1$$

The Cost of Towers of Hanoi

- Pseudocode to solution to the Towers of Hanoi problem

```
solveTowers(count, source, destination, spare)
{
   if (count is 1)
      Move a disk directly from source to destination
   else
   {
      solveTowers(count - 1, source, spare, destination)
      solveTowers(1, source, destination, spare)
      solveTowers(count - 1, spare, destination, source)
   }
}
```

- Consider: given *N* disks …
  - How many moves does solveTowers make?

The Cost of Towers of Hanoi

- Claim

$$moves(N) = 2^N - 1 \quad \text{for all } N \geq 1$$

- Basis
  - Show that the property is true for N = 1
- Inductive hypothesis
  - Assume that property is true for N = k
- Inductive conclusion
  - Show that property is true for N = k + 1

End

Chapter 5