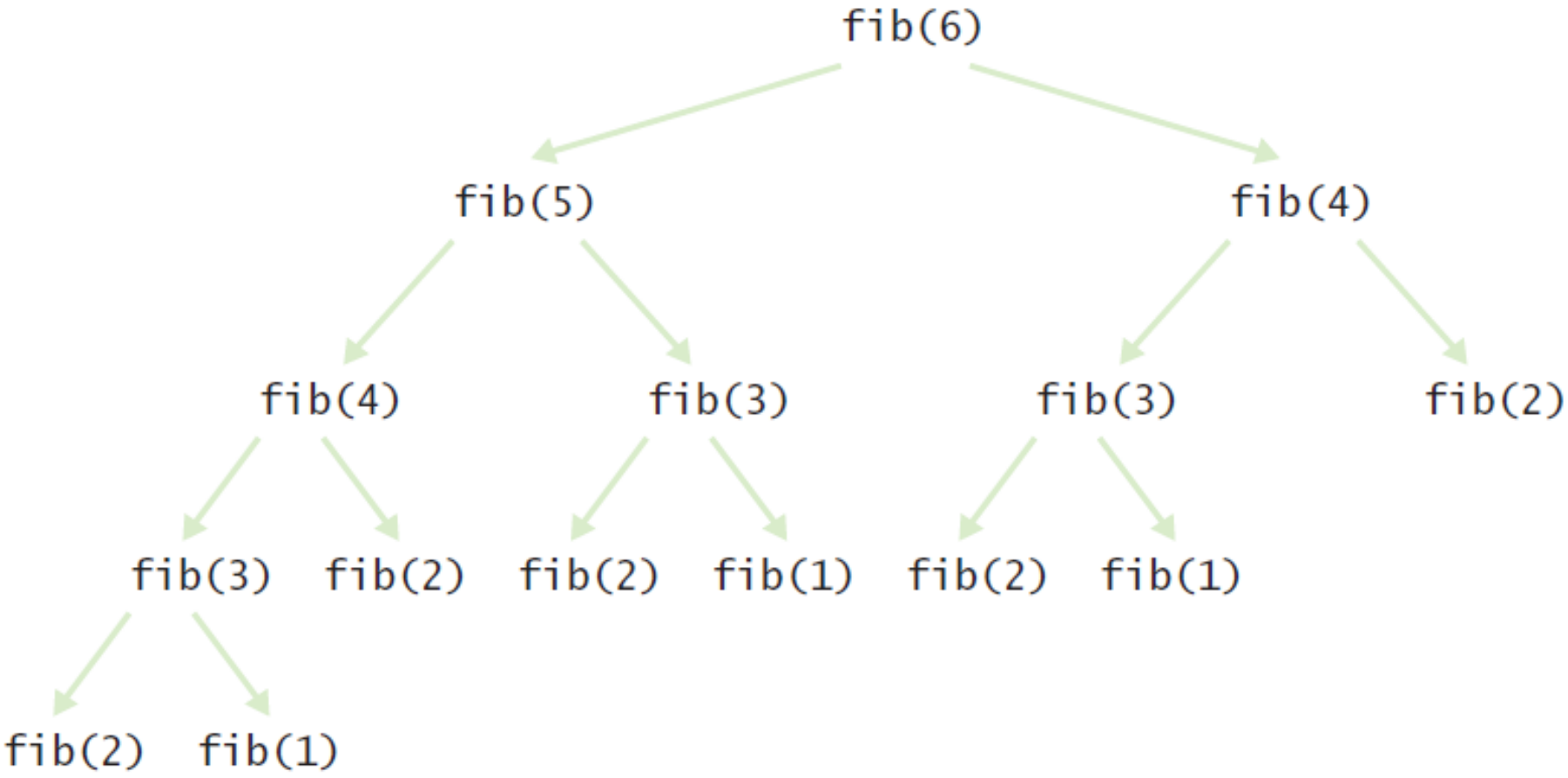
Call Pattern of Recursive `Fib()` Function



Efficiency of Recursion

- The function takes so long because it computes the same values over and over.
- Computation of `fib(6)` calls `fib(3)` three times.
- Imitate the pencil-and-paper process to avoid computing the values more than once.

Efficiency of Recursion

- Occasionally, a recursive solution runs much slower than its iterative counterpart.
- In most cases, the recursive solution is only slightly slower.
- The iterative `isPalindrome()` performs only slightly better than recursive solution.
  - *Each recursive function call takes a certain amount of processor time*

Loopfib.py (1)

```
1  ##
2  #  This program computes Fibonacci numbers using an iterative function.
3  #
4
5  def main() :
6     n = int(input("Enter n: "))
7     for i in range(1, n + 1) :
8        f = fib(i)
9        print("fib(%d) = %d" % (i, f))
10
```

Loopfib.py (2)

```
11  ## Computes a Fibonacci number.
12  #  @param n an integer
13  #  @return the nth Fibonacci number
14  #
15  def fib(n) :
16     if n <= 2 :
17        return 1
18     else :
19        olderValue = 1
20        oldValue = 1
21        newValue = 1
22        for i in range(3, n + 1) :
23           newValue = oldValue + olderValue
24           olderValue = oldValue
25           oldValue = newValue
26
27        return newValue
28
29  # Start the program.
30  main()
```

Program Run

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

Memoized Fibonacci

```
##
#  This program computes Fibonacci numbers using a recursive function.
#

def main() :
    n = int(input("Enter n: "))
    for i in range(1, n + 1) :
        f = fib(i,{})
        print("fib(%d) = %d" % (i, f))


## Computes a Fibonacci number.
#  @param n an integer
#  @return the nth Fibonacci number
#
def fib(n,memoize) :
    if n <= 2 :
        memoize[n] = 1
    else :
        if(not n in memoize):
            memoize[n] = fib(n - 1,memoize) + fib(n - 2,memoize)
    return memoize[n]
# Start the program.
main()
```

Efficiency of Recursion

- Smart compilers can avoid recursive function calls if they follow simple patterns.
- Most compilers don't do that
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution .
- '*To iterate is human, to recurse divine.*'
  - L. Peter Deutsch

Iterative `IsPalindrome()` Function

```
def isPalindrome(text) :
    start = 0
    end = len(text) - 1
    while start < end :
        first = text[start].lower()
        last = text[end].lower()
        if first.isalpha() and last.isalpha() :
            # Both are letters.
            if first == last :
                start = start + 1
                end = end - 1
            else :
                return False
        if not last.isalpha()
            end = end - 1
```

11.5 Permutations

- Design a class that will list all permutations of string, where a permutation is a rearrangement of the letters
- The string "eat" has six permutations:
  - "eat"
  - "eta"
  - "aet"
  - "ate"
  - "tea"
  - "tae"

Generate All Permutations (1)

- Generate all permutations that start with 'e', then 'a', then 't'
- The string "eat" has six permutations:
  - "eat"
  - "eta"
  - "aet"
  - "ate"
  - "tea"
  - "tae"

Generate All Permutations (2)

- Generate all permutations that start with 'e', then 'a', then 't'
- To generate permutations starting with 'e', we need to find all permutations of "at"
- This is the same problem with simpler inputs
- Use recursion

Implementing `Permutations()` Function

- Loop through all positions in the word to be permuted
- For each of them, compute the shorter word obtained by removing the i^{th} letter:

```
shorter = word[ : i] + word[i + 1 : ]
```

- Compute the permutations of the shorter word:

```
shorterPermutations = permutations(shorter)
```

Implementing `Permutations()` Function

- Add the removed letter to the front of all permutations of the shorter word:

```
for s in shorterPermutations :
    result.append(word[i] + s)
```

- Special case for the simplest string, the empty string, which has a single permutation - itself

Permutations.py (1)

```
1  ##
2  #  This program computes permutations of a string.
3  #

4
5  def main() :
6     for string in permutations("eat") :
7        print(string)
8

36
37 # Start the program.
38 main()
```

Permutations.py (2)

```
 9  ## Gets all permutations of a given word.
10  #  @param word the string to permute
11  #  @return a list of all permutations
12  #
13  def permutations(word) :
14     result = []
15
16     # The empty string has a single permutation: itself.
17     if len(word) == 0 :
18        result.append(word)
19        return result
20     else :
21        # Loop through all character positions.
22        for i in range(len(word)) :
23           # Form a shorter word by removing the ith character.
24
29           # Add the removed character to the front of each permutation
30           # of the simpler word.
31           for string in shorterPermutations :
32              result.append(word[i] + string)
33
34        # Return all permutations
35        return result
```

Program Run

```
eat
eta
aet
ate
tea
tae
```

Backtracking

- Backtracking examines partial solutions, abandoning unsuitable ones and returning to consider other candidates
- Can be used to
  - solve crossword puzzles
  - escape from mazes
  - find solutions to systems that are constrained by rules

Backtracking Characteristic Properties

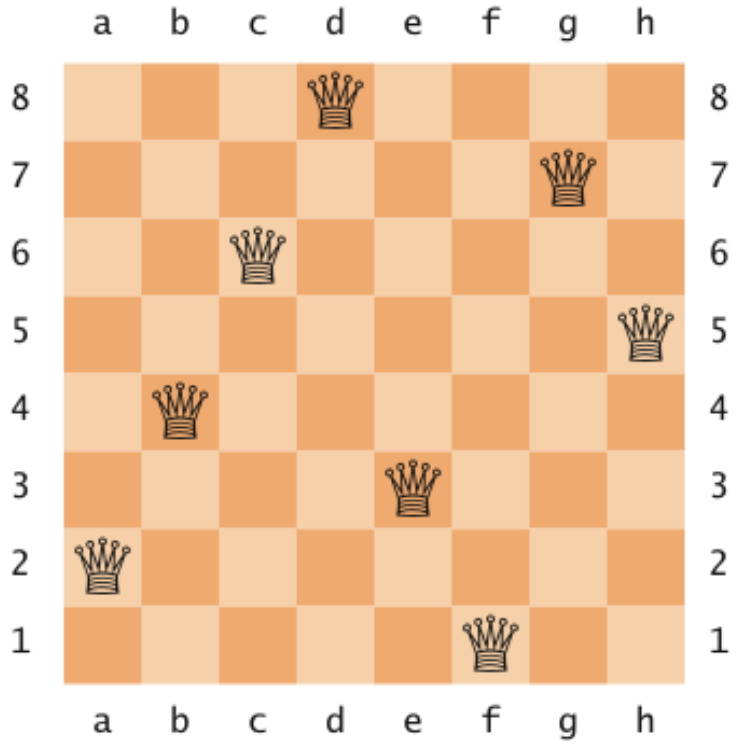
1. A procedure to examine a partial solution and determine whether to:
    I. accept it as an actual solution or,
    II. abandon it (because it either violates some rules or can never lead to a valid solution)
2. A procedure to extend a partial solution, generating one or more solutions that come closer to the goal

Recursive Backtracking Algorithm

```
Solve(partialSolution)
     Examine(partialSolution).
     If accepted
          Add partialSolution to the list of solutions.
     Else if not abandoned
          For each p in extend(partialSolution)
              Solve(p)
```

Eight Queens Problem

- **Problem:** position eight queens on a chess board so that none of them attacks another according to the rules of chess
- A solution:



Eight Queens Problem

- Easy to examine a partial solution:
  - If two queens attack one another, reject it
  - Otherwise, if it has eight queens, accept it
  - Otherwise, continue
- Easy to extend a partial solution:
  - Add another queen on an empty square
- Systematic extensions:
  - Place first queen on row 1
  - Place the next on row 2
  - Etc.

Function: `Examine()`

```
def examine(partialSolution) :
    for i in range(0, len(partialSolution)) :
        for j in range(i + 1, len(partialSolution)) :
            if attacks(partialSolution[i],
               partialSolution[j]) :
                return ABANDON
    if len(partialSolution) == NQUEENS :
        return ACCEPT
    else :
        return CONTINUE
```

Function: Extend()

```
def extend(partialSolution) :
    results = []
    row = len(partialSolution) + 1
    for column in "abcdefgh" :
        newSolution = list(partialSolution)
        newSolution.append(column + str(row))
        results.append(newSolution)
    return results
```

Diagonal Attack

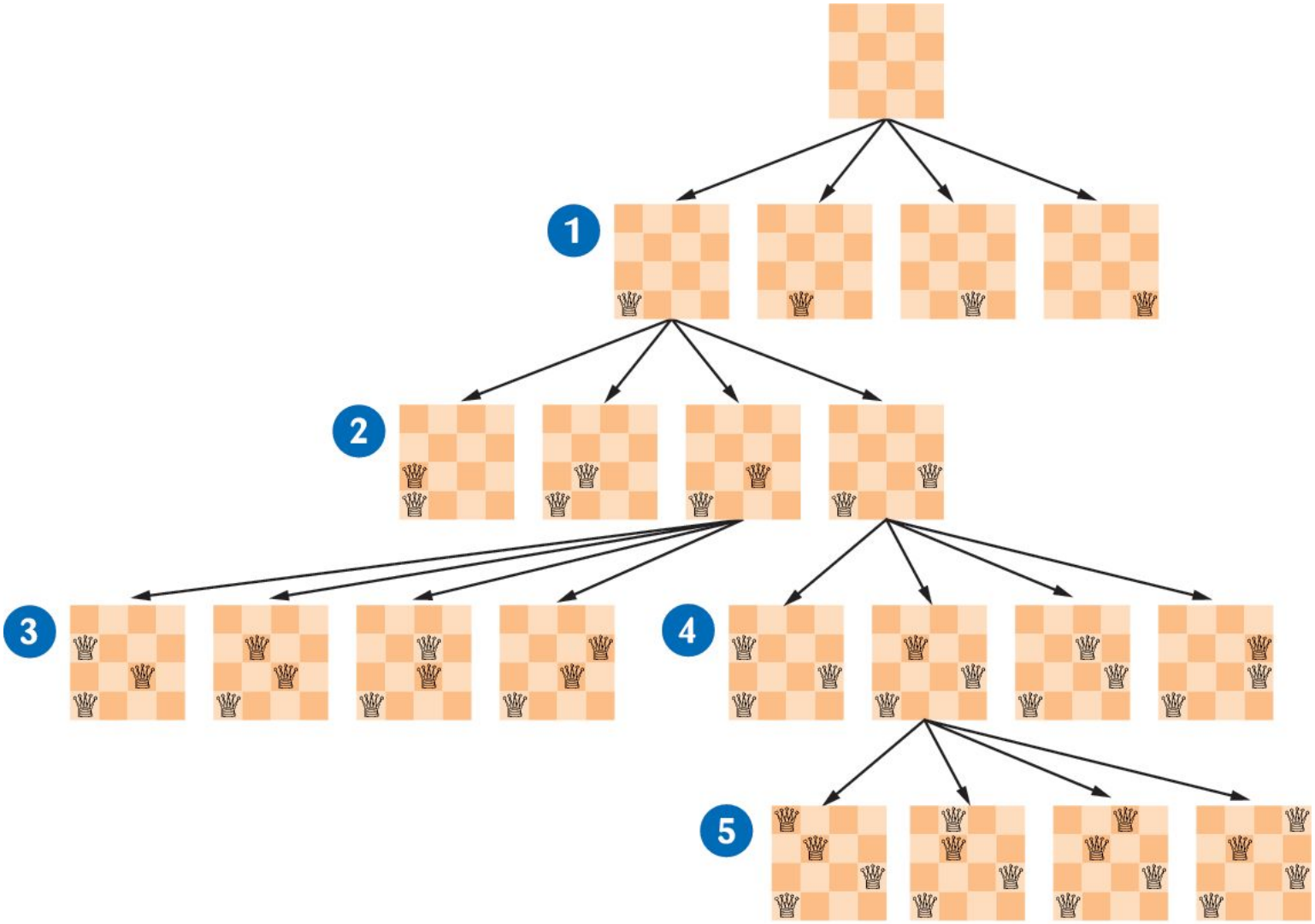
- To determine whether two queens attack each other diagonally:
  - Check whether slope is ±1

$$(row_2 - row_1)/(column_2 - column_1) = \pm 1$$

$$row_2 - row_1 = \pm(column_2 - column_1)$$

$$|row_2 - row_1| = |column_2 - column_1|$$

Backtracking in the Four Queens Problem (1)



Backtracking in the Four Queens Problem (2)

- Starting with a blank board, four partial solutions with a queen in row 1 (1)
- When the queen is in column 1, four partial solutions with a queen in row 2 (2)
  - Two are abandoned immediately
  - Other two lead to partial solutions with three queens (3) and (4), all but one of which are abandoned
- One partial solution is extended to four queens, but all of those are abandoned as well (5)

Queens.py

```
 5  def main() :
 6     solve([])
 7
 8  COLUMNS = "abcdefgh"
 9  NQUEENS = len(COLUMNS)
10  ACCEPT = 1
11  CONTINUE = 2
12  ABANDON = 3
```

```
18  def solve(partialSolution) :
19     exam = examine(partialSolution)
20     if exam == ACCEPT :
21        print(partialSolution)
22     elif exam != ABANDON :
23        for p in extend(partialSolution) :
24           solve(p)
```

```
69  # Start the program.
70  main()
```

Queens.py

```
31  def examine(partialSolution) :
32     for i in range(0, len(partialSolution)) :
33        for j in range(i + 1, len(partialSolution)) :
34           if attacks(partialSolution[i], partialSolution[j]) :
35              return ABANDON
36     if len(partialSolution) == NQUEENS :
37        return ACCEPT
38     else :
39        return CONTINUE
```

```
60  def extend(partialSolution) :
61     results = []
62     row = len(partialSolution) + 1
63     for column in COLUMNS :
64        newSolution = list(partialSolution)
65        newSolution.append(column + str(row))
66        results.append(newSolution)
67     return results
```

```
47  def attacks(p1, p2) :
48     column1 = COLUMNS.index(p1[0]) + 1
49     row1 = int(p1[1])
50     column2 = COLUMNS.index(p2[0]) + 1
51     row2 = int(p2[1])
52     return (row1 == row2 or column1 == column2 or
53        abs(row1 - row2) == abs(column1 - column2))
```

Queens.py

Program Run

```
['a1', 'e2', 'h3', 'f4', 'c5', 'g6', 'b7', 'd8']
['a1', 'f2', 'h3', 'c4', 'g5', 'd6', 'b7', 'e8']
['a1', 'g2', 'd3', 'f4', 'h5', 'b6', 'e7', 'c8']
 . . .
['f1', 'a2', 'e3', 'b4', 'h5', 'c6', 'g7', 'd8']
 . . .
['h1', 'c2', 'a3', 'f4', 'b5', 'e6', 'g7', 'd8']
['h1', 'd2', 'a3', 'c4', 'f5', 'b6', 'g7', 'e8']'
```

(92 solutions)

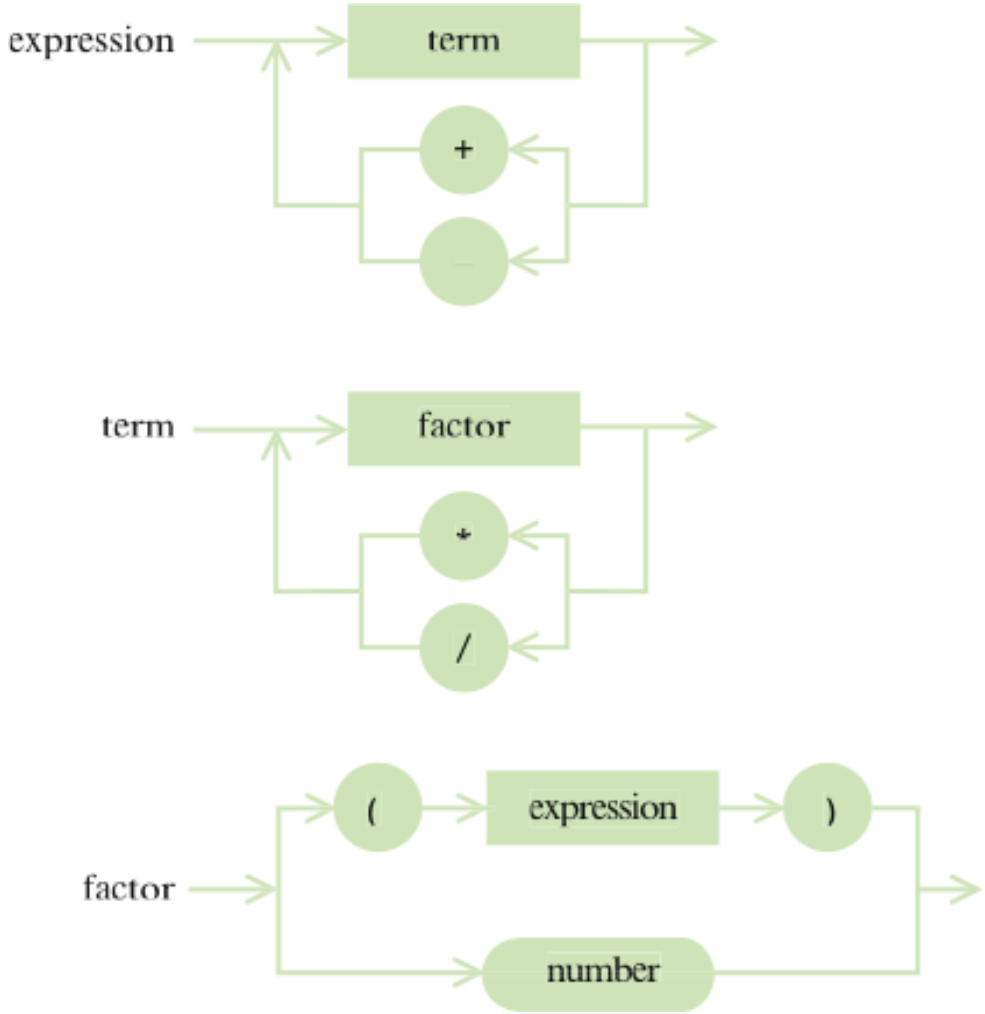
11.7 Mutual Recursion

- **Problem:** Compute the value of arithmetic expressions such as:

```
3 + 4 * 5
(3 + 4) * 5
1 - (2 - (3 - (4 - 5)))
```

- Computing the expression is complicated
  - * and / bind more strongly than + and –
  - Parentheses can be used to group sub-expressions

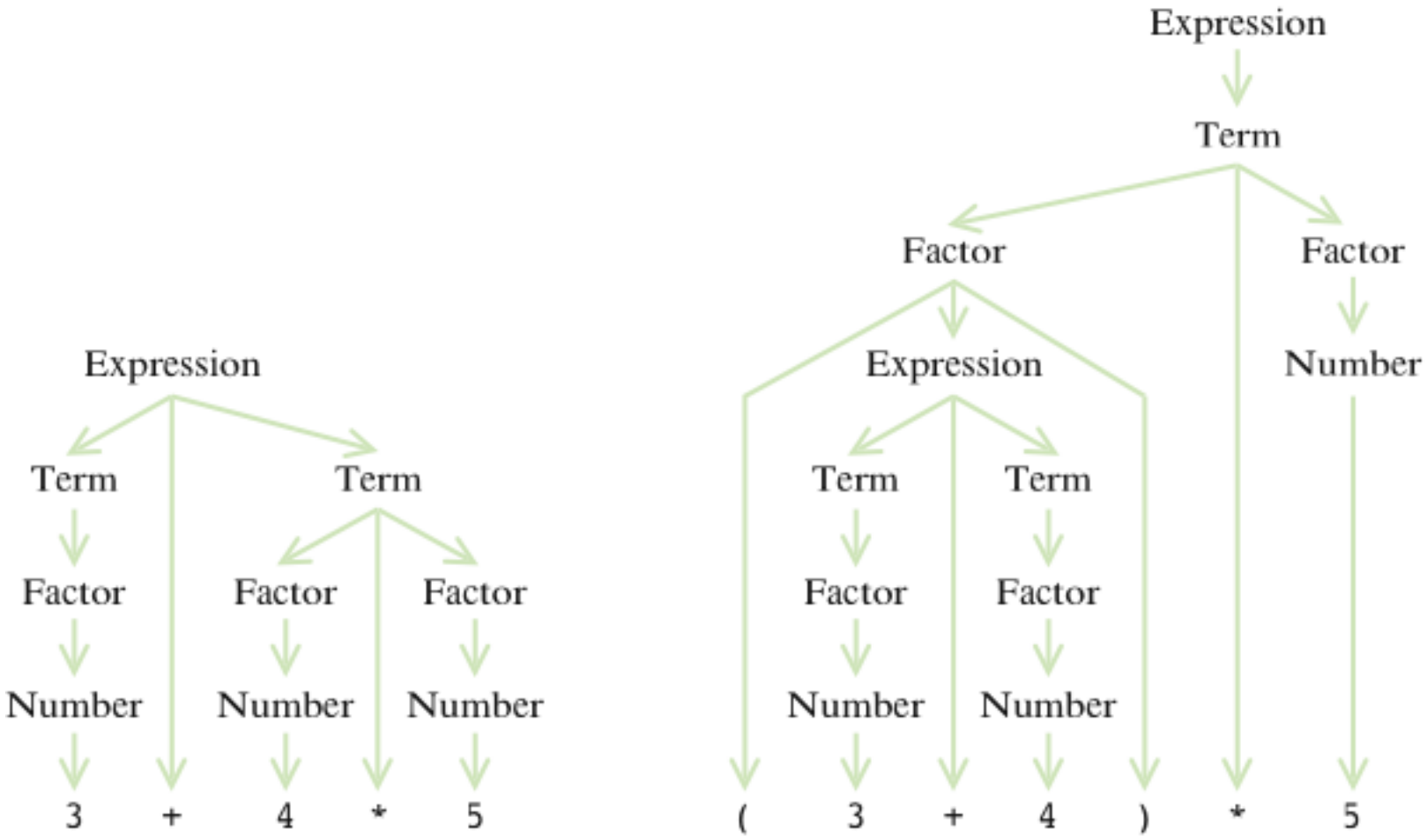
Syntax Diagrams for Evaluating an Expression



Mutual Recursion

- An *expression* can be broken down into a sequence of terms, separated by + or –
- Each *term* is broken down into a sequence of factors, separated by * or /
- Each *factor* is either a parenthesized expression or a number
- The syntax trees represent which operations should be carried out first

Syntax Trees for Two Expressions



Mutual Recursion

- In a mutual recursion, a set of cooperating functions calls each other repeatedly
- To compute the value of an expression, implement 3 functions that call each other recursively:
  - `expression()`
  - `term()`
  - `factor()`

Function: Expression()

```
def expression(tokens) :
    value = term(tokens)
    done = False
    while not done and len(tokens) > 0 :
        next = tokens[0]
        if next == "+" or next == "-" :
            tokens.pop(0)   # Discard "+" or "-"
            value2 = term(tokens)
            if next == "+" :
                value = value + value2
            else :
                value = value - value2
        else :
            done = True

return value
```

Function: `Term()`

- The `term()` function calls `factor()` in the same way, multiplying or dividing the factor values

```
def term(tokens) :
    value = factor(tokens)
    done = False
    while not done and len(tokens) > 0:
        next = tokens[0]
        if next == "*" or next == "/" :
            tokens.pop(0)
            value2 = factor(tokens)
            if next == "*" :
                value = value * value2
            else :
                value = value / value2
       else :
           done = True

return value
```

Function: `Factor()`

```
def factor(tokens) :
    next = tokens.pop(0)
    if next == "(" :
        value = expression(tokens)
        tokens.pop(0)   # Discard ")"
    else :
        value = next

    return value
```

Trace (3 + 4) * 5

To see the mutual recursion clearly, trace through the expression (3+4)*5:

- expression() calls term()
  - term() calls factor()
    - factor() consumes the (input
    - factor() calls expression()
      - expression() returns eventually with the value of 7, having consumed 3 + 4. This is the recursive call.
    - factor() consumes the) input
    - factor() returns 7
  - term() consumes the inputs * and 5 and returns 35
- expression() returns 35

Evaluator.py (1)

```
 5  def main() :
 6     expr = input("Enter an expression: ")
 7     tokens = tokenize(expr)
 8     value = expression(tokens)
 9     print(expr + "=" + str(value))
```

```
15  def tokenize(inputLine) :
16     result = []
17     i = 0
18     while i < len(inputLine) :
19        if inputLine[i].isdigit() :
20           j = i + 1
21           while j < len(inputLine) and inputLine[j].isdigit() :
22              j = j + 1
23           result.append(int(inputLine[i : j]))
24           i = j
25        else :
26           result.append(inputLine[i])
27           i = i + 1
28     return result
```

Evaluator.py

```
34  def expression(tokens) :
35      value = term(tokens)
36      done = False
37      while not done and len(tokens) > 0 :
38          next = tokens[0]
39          if next == "+" or next == "-" :
40              tokens.pop(0)    # Discard "+" or "-"
41              value2 = term(tokens)
42              if next == "+" :
43                  value = value + value2
44              else :
45                  value = value - value2
46          else :
47              done = True
48
49      return value
```

Evaluator.py

```
55  def term(tokens) :
56      value = factor(tokens)
57      done = False
58      while not done and len(tokens) > 0:
59          next = tokens[0]
60          if next == "*" or next == "/" :
61              tokens.pop(0)
62              value2 = factor(tokens)
63              if next == "*" :
64                  value = value * value2
65              else :
66                  value = value / value2
67          else :
68              done = True
69
70      return value
```

Evaluator.py

```
76  def factor(tokens) :
77      next = tokens.pop(0)
78      if next == "(" :
79          value = expression(tokens)
80          tokens.pop(0)    # Discard ")"
81      else :
82          value = next
83
84      return value
85
86  # Start the program.
87  main()
```

Program Run

```
Enter an expression: 3+4*5
3+4*5=23
```

Summary

- Understand the control flow in a recursive computation.
  - A recursive computation solves a problem by using the solution to the same problem with simpler inputs.
  - For a recursion to terminate, there must be special cases for the simplest values.
- Design a recursive solution to a problem.

Summary

- Identify recursive helper functions for solving a problem.
  - Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.
- Contrast the efficiency of recursive and non-recursive algorithms.
  - Occasionally, a recursive solution runs much slower than its iterative counterpart. However, in most cases, the recursive solution is only slightly slower.
  - In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

Summary

- Review a complex recursion example that cannot be solved with a simple loop.
  - The permutations of a string can be obtained more naturally through recursion than with a loop.
- Use backtracking to solve problems that require trying out multiple paths.
  - Backtracking examines partial solutions, abandoning unsuitable ones and returning to consider other candidates.

Summary

- Recognize the phenomenon of mutual recursion in an expression evaluator.
  - In a mutual recursion, cooperating functions or methods call each other repeatedly.