Chapter 5:
Recursion as a Problem-Solving Technique

Data Abstraction & Problem Solving with
C++
Fifth Edition
by Frank M. Carrano
Backtracking

- Backtracking
  - A strategy for guessing at a solution and backing up when an impasse is reached

- Recursion and backtracking can be combined to solve problems

- Eight-Queens Problem
  - Place eight queens on the chessboard so that no queen can attack any other queen
The Eight Queens Problem

• One strategy: guess at a solution
  – There are 4,426,165,368 ways to arrange 8 queens on a chessboard of 64 squares
• An observation that eliminates many arrangements from consideration
  – No queen can reside in a row or a column that contains another queen
    • Now: only 40,320 (8!) arrangements of queens to be checked for attacks along diagonals
The Eight Queens Problem

- Providing organization for the guessing strategy
  - Place queens one column at a time
  - If you reach an impasse, backtrack to the previous column

Figure 5-2
A solution to the Eight Queens problem
The Eight Queens Problem

• A recursive algorithm that places a queen in a column
  – Base case
    • If there are no more columns to consider
      – You are finished
  – Recursive step
    • If you successfully place a queen in the current column
      – Consider the next column
    • If you cannot place a queen in the current column
      – You need to backtrack
The Eight Queens Problem

Figure 5-1 (a) Five queens that cannot attack each other, but that can attack all of column 6; (b) backtracking to column 5 to try another square for the queen; (c) backtracking to column 4 to try another square for the queen and then considering column 5 again
Implementing Eight Queens Using the STL Class \texttt{vector}

- A \texttt{Board} object represents the chessboard
  - Contains a vector of pointers to \texttt{Queen} objects on the board
  - Includes operations to perform the Eight Queens problem and display the solution

- A \texttt{Queen} object represents a queen on the board
  - Keeps track of its row and column placement
  - Contains a static pointer to the \texttt{Board}
  - Has operations to move it and check for attacks
Defining Languages

• A language
  – A set of strings of symbols
  – Examples: English, C++

• A grammar
  – The rules for forming the strings in a language
  – Examples: English grammar, C++ syntax rules
Defining Languages

- If a C++ program is one long string of characters, the language of C++ programs is defined as
  \[
  \text{C++Programs} = \{ \text{strings } w : w \text{ is a syntactically correct C++ program} \}
  \]

- A language does not have to be a programming or a communication language
  - \text{AlgebraicExpressions} = \{ \text{strings } w : w \text{ is an algebraic expression} \}
The Basics of Grammars

• Symbols used in grammars
  – $x \mid y$ means $x$ or $y$
  – $x \cdot y$ or $x \cdot y$ means $x$ followed by $y$
    • The symbol $\cdot$ means concatenate (append)
  – $<\text{word}>$ means any instance of \textit{word} that the definition defines
The Basics of Grammars

• A C++ identifier begins with a letter and is followed by zero or more letters and digits

• Language of C++ identifiers
  C++Ids = \{w : w is a legal C++ identifier\}

• Grammar
  \[ < \text{identifier} > = \langle \text{letter} \rangle | \langle \text{identifier} \rangle \langle \text{letter} \rangle | \langle \text{identifier} \rangle \langle \text{digit} \rangle \]
  \[ < \text{letter} > = a | b | … | z | A | B | … | Z | _ \]
  \[ < \text{digit} > = 0 | 1 | … | 9 \]
The Basics of Grammars

• A recognition algorithm sees whether a given string is in the language
  – A recognition algorithm for a language is written more easily if the grammar is recursive
The Basics of Grammars

- Recognition algorithm for the language C++Ids

```cpp
isId(in w:string):boolean
  if (w is of length 1)
    if (w is a letter)
      return true
  else
    return false
else if (the last character of w is a letter or a digit)
  return isId(w minus its last character)
else
  return false
```
Two Simple Languages: Palindromes

• A string that reads the same from left to right as it does from right to left

• Language

\[ \text{Palindromes} = \{w : w \text{ reads the same left to right as right to left}\} \]

• Grammar

\[
<\text{pal}> = \text{empty string} \mid <\text{ch}> \mid a <\text{pal}> a \mid b <\text{pal}> b \mid \ldots \mid Z <\text{pal}> Z
\]

\[
<\text{ch}> = a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z
\]
Two Simple Languages: Palindromes

• Recognition algorithm
  
isPal(in w:string):boolean
  if (w is the empty string or
      w is of length 1)
    return true
  else if (w’s first and last characters are
          the same letter )
    return isPal(w minus its first and last
                characters)
  else
    return false
Two Simple Languages:
Strings of the Form $A^nB^n$

- $A^nB^n$
  - The string that consists of $n$ consecutive A’s followed by $n$ consecutive B’s

- Language
  $$L = \{w : w \text{ is of the form } A^nB^n \text{ for some } n \geq 0\}$$

- Grammar
  $$<\text{legal-word}> = \text{empty string} \mid A <\text{legal-word}> B$$
Two Simple Languages:
Strings of the form $A^nB^n$

- Recognition algorithm
  
  $\text{isAnBn}(\text{in } w: \text{string}): \text{boolean}$
  
  $\text{if } (\text{the length of } w \text{ is zero})$
  
  $\text{return } \text{true}$

  $\text{else if } (w \text{ begins with the character } A \text{ and }$
  
  $\text{ends with the character } B)$

  $\text{return } \text{isAnBn} (w \text{ minus its first and last }$
  
  $\text{characters})$

  $\text{else}$

  $\text{return } \text{false}$
Algebraic Expressions

• **Infix expressions**
  – An operator appears between its operands
    • Example: \( a + b \)

• **Prefix expressions**
  – An operator appears before its operands
    • Example: \( + a b \)

• **Postfix expressions**
  – An operator appears after its operands
    • Example: \( a b + \)
Algebraic Expressions

• To convert a fully parenthesized infix expression to a prefix form
  – Move each operator to the position marked by its corresponding open parenthesis
  – Remove the parentheses
  – Example
    • Infix expression: \(( (a + b) \times c )\)
    • Prefix expression: \(* + a b c\)
Algebraic Expressions

• To convert a fully parenthesized infix expression to a postfix form
  – Move each operator to the position marked by its corresponding closing parenthesis
  – Remove the parentheses
  – Example
    • Infix expression: ( (a + b) * c )
    • Postfix expression: a b + c *
Algebraic Expressions

• Advantages of prefix and postfix expressions
  – No precedence rules
  – No association rules
  – No parentheses
  – Simple grammars
  – Straightforward recognition and evaluation algorithms
Prefix Expressions

• Grammar
  \[
  \langle \text{prefix} \rangle = \langle \text{identifier} \rangle \mid \langle \text{operator} \rangle \langle \text{prefix} \rangle \langle \text{prefix} \rangle \\
  \langle \text{operator} \rangle = + \mid - \mid * \mid / \\
  \langle \text{identifier} \rangle = a \mid b \mid \ldots \mid z
  \]

• A recursive recognition algorithm
  – Base case: One lowercase letter is a prefix exp.
  – Recursive: \langle \text{operator} \rangle \langle \text{prefix} \rangle \langle \text{prefix} \rangle
Prefix Expressions

• If $E$ is a prefix expression
  If $Y$ is any nonempty string of nonblanks
  Then $E \cdot Y$ cannot be prefix

• Recognition algorithm

```java
isPre():boolean
    lastChar = endPre(0) // returns index of end
    return (lastChar >= 0 and
            lastChar == strExp.length() - 1)
```
Prefix Expressions

```plaintext
endPre(in first:integer):integer
  last = strExp.length() - 1
  if (first < 0 or first > last) return -1
  ch = strExp[first]
  if (ch is an identifier) return first
  else if (ch is an operator)
    { firstEnd = endPre(first + 1) //Point X
      if (firstEnd > -1)
        return endPre(firstEnd + 1) //Point Y
      else return -1
    }
  else return -1
```
Prefix Expressions

- Algorithm that evaluates a prefix expression
  
  ```
  evaluatePrefix(inout strExp:string):float
  
  ch = first character of expression strExp
  Delete first character from strExp
  if (ch is an identifier)
    return value of the identifier
  else if (ch is an operator named op)
  {  operand1 = evaluatePrefix(strExp)
      operand2 = evaluatePrefix(strExp)
      return operand1 op operand2
  }
  ```
Postfix Expressions

• Grammar

  \[
  \langle postfix \rangle = \langle identifier \rangle | \\
  \quad \langle postfix \rangle \langle postfix \rangle \langle operator \rangle \\
  \langle operator \rangle = + | - | * | / \\
  \langle identifier \rangle = a | b | \ldots | z
  \]

• The recursive case for conversion from prefix form to postfix form

  \[
  \text{postfix}(\text{exp}) = \text{postfix}(\text{prefix1}) + \\
  \quad \text{postfix}(\text{prefix2}) + \langle \text{operator} \rangle
  \]
Postfix Expressions

- Recursive algorithm that converts a prefix expression to postfix form

```cpp
convert(inout pre:string, out post:string) {
    ch = first character in pre
    Delete first character in pre
    if (ch is a lowercase letter) {
        post = post + ch
    } else {
        convert(pre, post)
        convert(pre, post)
        post = post + ch
    }
}
```
Fully Parenthesized Expressions

• Fully parenthesized infix expressions
  – Do not require precedence rules or rules for association
  – But are inconvenient for programmers

• Grammar
  
  \[ < \text{infix} > \ = \ < \text{identifier} > \mid \]  
  \[ ( < \text{infix} > < \text{operator} > < \text{infix} > ) \]  
  \[ < \text{operator} > \ = \ + \mid - \mid * \mid / \]  
  \[ < \text{identifier} > \ = \ a \mid b \mid \ldots \mid z \]
Recursion and Mathematical Induction

• Recursion and mathematical induction
  – Both use a base case to solve a problem
  – Both solve smaller problems of the same type to derive a solution

• Induction can be used to
  – Prove properties about recursive algorithms
  – Prove that a recursive algorithm performs a certain amount of work
The Correctness of the Recursive Factorial Function

• Pseudocode for recursive factorial

```java
if (n is 0)
    return 1
else
    return n * fact(n - 1)
```

• Induction on \(n\) proves the return values:
  - \(\text{fact}(0) = 0! = 1\)
  - \(\text{fact}(n) = n!(= n*(n - 1)*\ldots*1\) if \(n > 0\)
Organizing Data: The Towers of Hanoi

Figure 2-19a and b (a) The initial state; (b) move $n-1$ disks from A to C
The Towers of Hanoi

Figure 2-19c and d (c) move one disk from A to B; (d) move n - 1 disks from C to B
The Cost of Towers of Hanoi

• Solution to the Towers of Hanoi problem

```java
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)
    }
```
The Cost of Towers of Hanoi

• With $N$ disks, how many moves does `solveTowers` make?
• Let $moves(N)$ be the number of moves made starting with $N$ disks
• When $N = 1$
  – $moves(1) = 1$
• When $N > 1$
  – $moves(N) = moves(N - 1) + moves(1) + moves(N - 1)$
The Cost of Towers of Hanoi

• Recurrence relation for the number of moves that solveTowers requires for \( N \) disks

\[
\begin{align*}
\text{moves}(1) &= 1 \\
\text{moves}(N) &= 2 \times \text{moves}(N - 1) + 1 \\
& \quad \text{if } N > 1
\end{align*}
\]
The Cost of Towers of Hanoi

• A closed-form formula is more satisfactory
  – You can substitute any given value for $N$ and obtain the number of moves made
  – $\text{moves}(N) = 2^N - 1$, for all $N \geq 1$
  – Induction on $N$ can prove this
Summary

- Backtracking is a solution strategy that involves both recursion and a sequence of guesses that ultimately lead to a solution.
- A language is a set of strings of symbols.
  - A grammar is a device for defining a language.
  - A recognition algorithm for a language can often be based directly on the grammar of the language.
  - Grammars are frequently recursive.
Summary

• Different languages of algebraic expressions have their relative advantages and disadvantages
  – Prefix: simple grammar, hard to use
  – Postfix: simple grammar, hard to use
  – Infix: involved grammar, easy to use

• Induction can be used to prove properties about a recursive algorithm