Lecture
Chapter 6
Recursion
as a
Problem Solving Technique

Backtracking

1. **Select, i.e., guess, a path of steps that could possibly lead to a solution**
2. **If the path leads to a dead end then retrace steps in the reverse order**
3. **Select a new sequence of steps that could possibly lead to a solution**
4. **If need be, go to #2 above**

Eight Queens Problem

- **Chessboard**
  - **64 squares -- 8 rows x 8 columns**
  - **Queen can attack any other piece**
    - **within its row**
    - **within its column**
    - **along any diagonal**
  - **place eight queens on the board such that**
    no queen can attack any other queen,
  - **number of ways to arrange 8 queens on a 65 square board is c(64, 8)**
    - **c(64, 8) > 4 trillion**
      thus each row & column contains exactly one queen
    - **attacks along rows or columns are not possible**
    - **number of attacks to be checked along diagonals is 8! = 40,320**

- **Strategy 1**
  - **Placing queens on the board in the following sequence yields a strategy which fails after five queens since column six is totally blocked**

Q_1	•	•	•	•	•		
•	•	•	Q_4	•	•		
•	Q_2	•	•	•	•		
•	•	•	•	Q_5	•		
•	•	Q_3	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

  - **Backtrack by removing Q_5 and finding a new location on column five.**
  - **Placing Q_5 in any cell on column five still eliminates any queens being placed in column six.**
  - **Backtrack by removing Q_4 & Q_5 and finding new locations on both columns.**

Q_1	•	•	•	•	•		
•	•	•	•	Q_5	•		
•	Q_2	•	•	•	•		
•	•	•	•	•	•		
•	•	Q_3	•	•	•		
•	•	•	•	•	•		
•	•	•	Q_4	•	•		
•	•	•	•	•	•		

A solution to the Eight Queens Problem

Q_1	•	•	•	•	•	•	•
•	•	•	•	•	•	Q_7	•
•	•	•	•	Q_5	•	•	•
•	•	•	•	•	•	•	Q_8
•	Q_2	•	•	•	•	•	•
•	•	•	Q_4	•	•	•	•
•	•	•	•	•	Q_6	•	•
•	•	Q_3	•	•	•	•	•

Languages

- **Set of all Java Programs = $\left\{\text{all strings } \mathbf{w} : \mathbf{w} \text{ is a syntactically correct Java program}\right\}$**

  Java Compiler : program which determines
  if a string w is a syntactically correct Java program

- **Set of all English Sentences =**
  $\left\{\text{all strings } \mathbf{w} : \mathbf{w} \text{ is a syntactically correct English Sentences}\right\}$

  Set of all English Grammar Rules : determines
  if a sentence w is a syntactically correct English Sentence

- **Set of all Algebraic Expressions =**
  $\left\{\text{all strings } \mathbf{w} : \mathbf{w} \text{ is a syntactically correct Algebraic Expression}\right\}$

  Set of all Algebraic Rules : determines
  if a sentence w is a syntactically correct Algebraic Expression

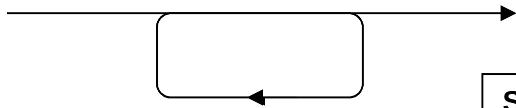
- **Grammar states the rules of a language**
  - **recursive rules – computer languages**
  - **non-recursive rules -- natural languages, e.g., English, Hungarian, etc.**

- **recognition algorithm – recursive algorithm based on the grammar**
  which determines whether a given string is in the grammar

Identifiers

- **x | y ⇔ x or y**
- **x y ⇔ x • y ⇔ x concatenated with y**
- **< symbol > ➔ any sequence of symbols defined by the grammar**

letter



identifier

Syntax Diagram

letter, digit
underscore __
dollar sign $

- **recursive grammar**
  - **<identifier> = <letter> | <identifier> <letter> | <identifier> <digit> |**
    $ <identifier> | _<identifier>
  - **<letter> = a | b | ... | z | A | B | ... | Z**
  - **<digit> = 0 | 1 | ... | 9**

- **recognition grammar**
  - **length(w) == 1 ➔ w is an identifier if w is a letter**
  - **length(w) ≥ 1 ➔ w is an identifier if**
    1. **last character of w is either a letter or a digit**
    2. **w minus the last character is an identifier**

```
isId(in w: string) : boolean
  if ( length(w) == 1 )
  {
        if ( w is a letter ) return true
        else return false
  }
  else if ( last character of w is either a letter or a digit )
 {
        return isId( w minus the last character )
 }
 else return false
```

Strings $A^n B^n$

- **n consecutive A's followed by n consecutive B's**
- **AAAAABBBBB**

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

- **Grammar** <legal-word > = empty string | A < legal-word > B

- **Recognition algorithm**

```
isAnBn(in w : string) : Boolean
{
   if ( length(w) == 0 ) return true
   else if ( w begins with A and ends with B )
   {
      return isAnBn( w minus first & last characters )
   }
   else return false
}
```

Algebraic Expressions

Binary Operators: +, -, *, /
Operands: single letter only

- **Infix, Prefix, Postfix Expressions**
  - **Infix**
    - operand$_1$ operator operand$_2$ e.g., x + y
    - associative rules
    - precedence rules
    - use of parentheses

    } avoid ambiguity
  - **Prefix**
    - operator operand$_1$ operand$_2$ e.g., + x y
    - infix a + (b * c) ➔ prefix + a * b c
    - infix (a + b) * c ➔ prefix * + a b c
  - **Postfix**
    - operand$_1$ operand$_2$ operator e.g., x y +
    - infix a + (b * c) ➔ postfix a b c * +
    - infix (a + b) * c ➔ postfix a b + c *

- **Conversion Infix to Prefix &/or Postfix**
  - **Infix ➔ fully parenthesized infix**

    a + b * c ➔ ((a + b) * c)
  - **Fully Parenthesized Infix ➔ Prefix**
    - **Move each operator to the position marked by its open parenthesis**

      ((a + b) * c) ➔ ((a   b)   c) ➔ * + a b c
      *** +**
  - **Fully Parenthesized Infix ➔ Postfix**
    - **Move each operator to the position marked by its closed parenthesis**

      ((a + b) * c) ➔ ((a   b)   c) ➔ a b + c *
      \+ *

> **Prefix & Postfix Expressions do not require**
> - **associative rules**
> - **precedence rules**
> - **use of parentheses**
>
> **to avoid ambiguity**

- **Prefix Grammar**

  <prefix> = <identifier> | <operator> <prefix><prefix>
  <operator> = + | - | * | /
  <identifier> = a | b | … | z

  > **To be Expanded Later in the Semester**

- **Postfix Grammar**

  <posfix> = <identifier> | <postfix> <postfix> <operator>
  <operator> = + | - | * | /
  <identifier> = a | b | … | z

  > **To be Expanded Later in the Semester**