Lecture 5: Problems Solving with Recursion

Read: Chapter 5, Carrano.

I. Recursive Programming & List Processing:

1. Traversing a list recursively:
   Approach:
      Visit the first node;
      Visit the list without the first node recursively.

Example: Output a sequence of characters stored in a list.

```
void writeString(Node *charPtr)
{
   if (charPtr != NULL)
   {  cout << charPtr->item;
      writeString(charPtr->next);
   }
}  // end writeString
```

Q: What if we need to output the string backward?

Approach:
   Output the list without the first char backward;
   Output the first char.

```
void writeBackward(Node *charPtr)
{
   if (charPtr != NULL)
   {
      writeBackWard(charPtr->next);
      cout << charPtr->item;
   }
}  // end writeBackward
```

2. Inserting a newItem into a sorted list:

Approach:

```
   if newItem < head.item
      insert newItem at the beginning of list
   else
      insert newItem to list without the first element recursively
```

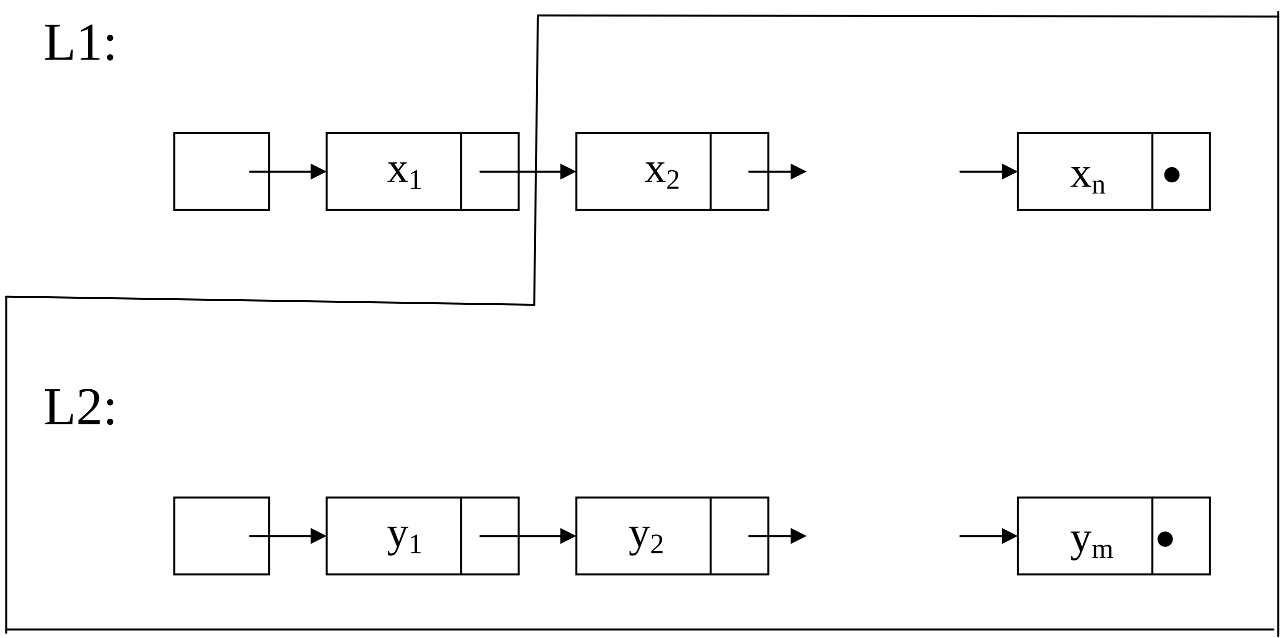
```
void linkedListInsert(Node *& headPtr,
                                        ListItemType newItem)
{
   if ( (headPtr == NULL) || (newItem <= headPtr->item) )
   {  // insert at the beginning of list
      Node *newPtr = new Node;
      if (newPtr == NULL)
         throw ListException(“ListException: No memory”);
      else
         {
            newPtr->item = newItem;
            newPtr->next = headPtr;
            headPtr = newPtr;
         }
   }  // endif
   else
      linkedListInsert(headPtr->next, newItem);

}  // end linkedListInsert
```

3. Concatenate two lists:

Q: Given two lists L1 and L2, how do we implement concat(ListNode L1,ListNode L2)?

Approach:

if L1 or L2 = ∅
    then return the other list
    else // strip off the first element in L1 and recursively
        // concatenate the remaining list with L2.
        L1->next = concat(L1->next,L2)



```
ListNode* concat(ListNode* L1, ListNode* L2)
{
    if (L1 == NULL)                    //base case
       return L2;
    else
    { if (L2 == NULL)
          return L1;
       else   //computing L1||L2 recursively
       {
          L1->next = concat(L1->next,L2);
          return L1;
        }
    }
}   //  end concat
```

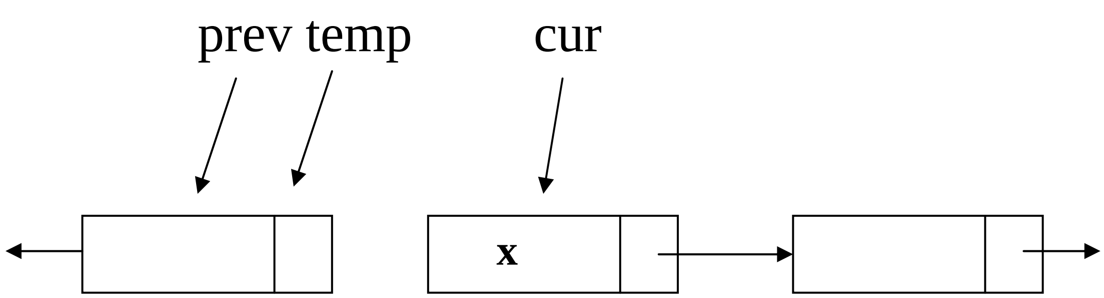
4. Reversing a list:

Q: How do we reverse a list L without copying the items in L?

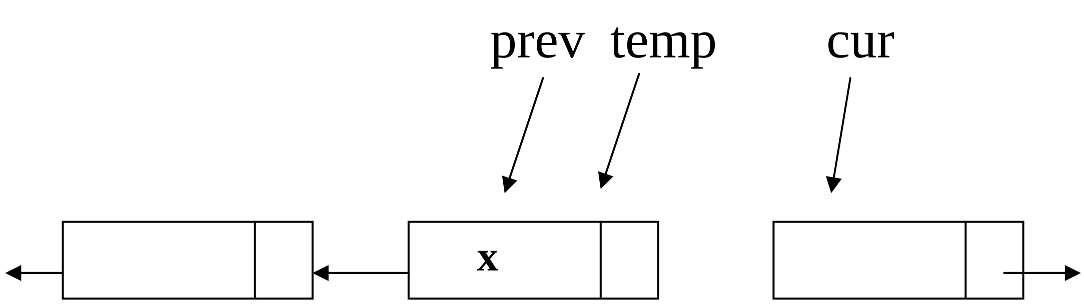
Iterative Algorithm for Reversing L:
Using 3 pointers:
    cur (pointing at node x whose link is to be reversed)
    prev, temp (pointing at the node preceding x, then x)

Initially,
    cur = head; prev = NULL; temp = NULL;

Before reversing link at current node x:



After reversing link at node x:




```
temp = cur;
cur = cur->next;
temp->next = prev;
prev = temp;
```

Iterative Reverse Algorithm:

```
void reverse(ListNode*& head)
{   ListNode *prev, *cur, *temp;
    cur = head;                 //cur points to the first node of L
    prev = NULL;                // initialize prev & temp pointer
    temp = NULL;

    while (cur != NULL)
    {   temp = cur;             // mark location of node
        cur = cur->next;   //mark next node
        temp->next = prev;  //reverse link of node
        prev = temp;            // advance prev to next node
    }
    head = prev;                //reset head of list
}
```

Q: Can you design a recursive algorithm for reversing L?
Let's try divide-and-conquer!

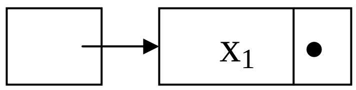
Given a list $L = <x_1, x_2, \ldots, x_t, x_{t+1}, \ldots, x_n>$.

- Divide L into two sublists L_1 and L_2 with $L_1 = <x_1, x_2, \ldots, x_t>$ and $L_2 = <x_{t+1}, x_{t+2}, \ldots, x_n>$, $1 \le t \le n-1$.
- If $L = \varnothing$ or $L = <x>$, then $L = L^R$.
- In general, $L^R = L_2^R \,\|\, L_1^R$.

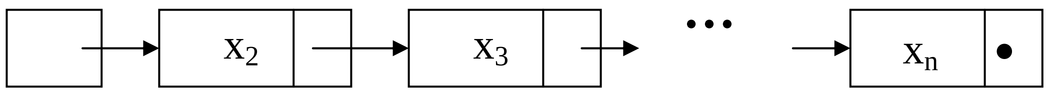
Approach:

- Strip off the first element in L to obtain L1.
- reverse(L) = concat(reverse(L2), L1);

L_1:



L_2:



reverse(L) = concat(reverse(L2), L1)

```
ListNode* reverse(ListNode* head)
{
 if (head == NULL)        //  base case: return an empty list
      return head;
 else                     //  reversing L recursively
       {  ListNode* L1 = head;              // define sublist L1
          ListNode* L2 = head->next;        // define sublist L2
          L1->next = NULL;
          //   compute L^R = L2^R || L1 recursively
          return concat(reverse(L2), L1);
       }
}
```

II. Backtracking and Recursive Algorithms:

Given a problem π and a set of properties/constraints P, find a solution s = $(x_1, x_2, ..., x_n)$, where s is an ordered n-tuples with x_i chosen from a finite set S_i with m_i elements, satisfying P.

Q: How do we compute a solution s of π?

1. Brute-Force Method (Exhaustive Search):
   Generate all possible n-tuples s = $(x_1, x_2, ..., x_n)$ and pick a solution that satisfies P.

Solution Space:

$$s \subseteq S_1 \times S_2 \times ... \times S_n$$

Any solution s must be in the form $(x_1, x_2, \ldots, x_k)$ such that $x_i \in S_i$, $1 \le i \le k \le n$. If there are m_1 choices for x_1, m_2 choices for x_2, ..., m_n choices for x_n, this method requires $m_1 * m_2 * ... * m_n$ steps in the worst case!

Q: Can we do it better?

Backtracking Solution Strategies:

An exhaustive search method allows us in exploring the solution space of a given problem in a systematic manner. For many problems, this method provides a simple recursive search algorithm that results in "good" average performance.

Backtracking Approach:

Compute $s = (x_1, x_2, \ldots, x_k)$ one component at a time. After the selection of the (i-1)th element x_{i-1}, if a solution has not yet been obtained, we will use a modified constraint function $P_i(x_1, x_2, ..., x_{i-1})$ to determine a set of possible candidates S_i^*, $S_i^* \subseteq S_i$, from which the next element x_i will be chosen such that $s = (x_1, x_2, ..., x_{i-1}, x_i)$ may still lead to a possible solution. If $S_i^* = \varnothing$ or the selection of x_i is not possible, we will then discard x_{i-1} and pick the next available element x_{i-1}^+ from the previously available set, S_{i-1}^*, $S_{i-1}^* \subseteq S_{i-1}$, to form $s = (x_1, x_2, ..., x_{i-1}^+)$, and to continue this process. If x_{i-1}^+ can not be chosen, we will then backtrack to the selection of the (i-2)th element and continue this process as before.

Characteristics of Backtracking Solution Strategies:

1. The solution S to the original problem π is composed of a sequence of solutions to a sequence of subproblems of π, Hence, $s = (x_1, x_2, ..., x_i)$.
2. Each subproblem of π has m_i possible (local) solutions x_i, $x_i \in S_i$.
3. During the selection of x_i, any selected subproblem solutions must be “compatible” with the existing partial solution $(x_1, x_2, ..., x_{i-1})$. Hence, $s = (x_1, x_2, ..., x_i)$ must permit satisfy the given constraints P and possibly leads to a gobal solution of π.

Generic Backtracking Algorithm:

for each subproblem
    while there are more candidate solutions to try do
        select a candidate solution if it is consistent with the
            candidate solutions for previous subproblems;
        move on to next subproblem;
    end while;
    backtrack to the previous subproblem and try another
        candidate solution;
end for;
report a (global) solution when found;

Some Applications of Backtracking Algorithms:

- Finding a path through a maze:
  Subproblems correspond to decisions to turn left, right, or go straight at certain points in the maze
- Placing n non-attacking queens on an n×n chessboard (n-Queen Problem)
  Subproblems correspond to placing a queen in each column.
- Sudoku:
  Subproblems correspond to selecting an integer for a square.

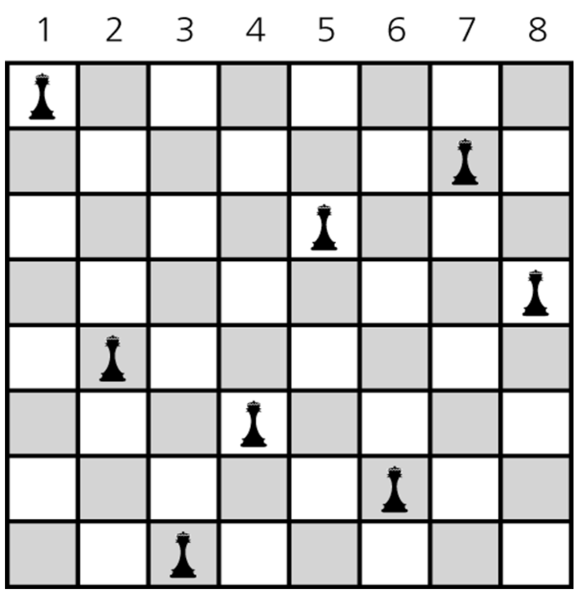
A Detailed Example: The n-Queen Problem.

Recall that a queen attacks everything on the same row, column and diagonals.

Q: For any given n $\in$N. How can you place n queens on an n$\times$n chessboard so that they will not attack each other?

Observe that a solution can be described by by s = (x_1, x_2, …, x_i, …, x_n) with the ith queen being placed on the (x_i,i)-position of the chessboard.

Example: Take n = 8. The solution for the following configuration is given by s = (1,5,8,6,3,7,2,4).



Q: How do we solve this n-queen problem?

Let's consider a brute-force approach for a simplified 8-queen problem.

Q: How many different configurations are there for placing 8 queens?

$$\binom{n^2}{n} = \binom{64}{8} = 4.4\times10^9$$

A Simplified Approach:

Since each column can only hold 1 queen, we need only consider

$$n^n \quad = \quad 1.7\times10^7 \text{ configurations.}$$

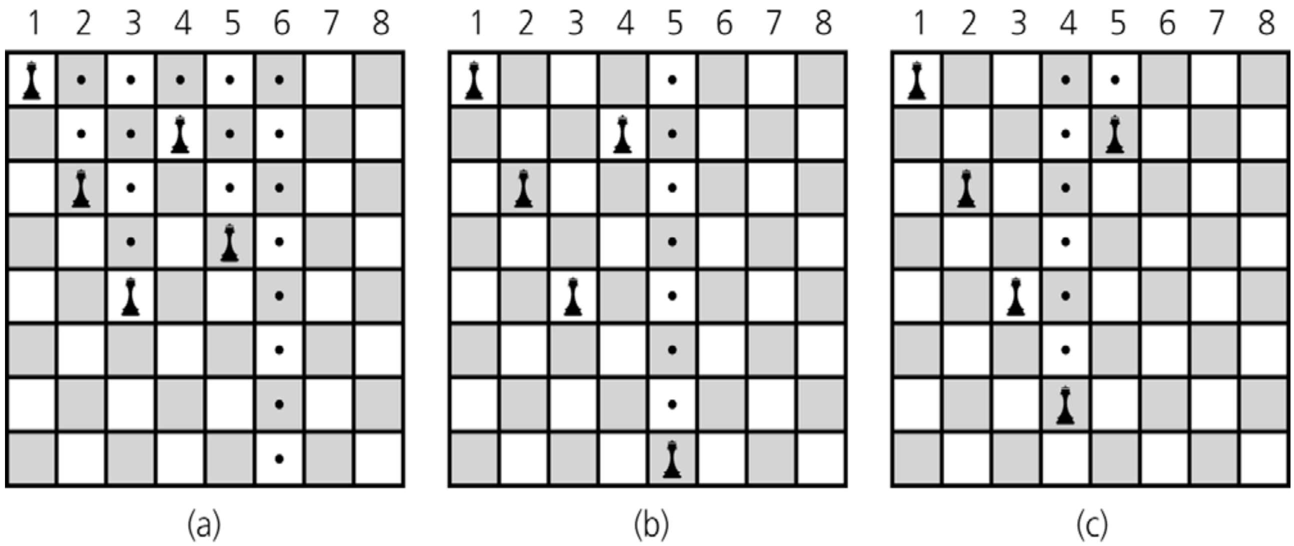
Also, since each row can only hold 1 queen, one can further reduce it to

$$n! \quad = \quad 4.0\times10^4 \text{ configurations.}$$

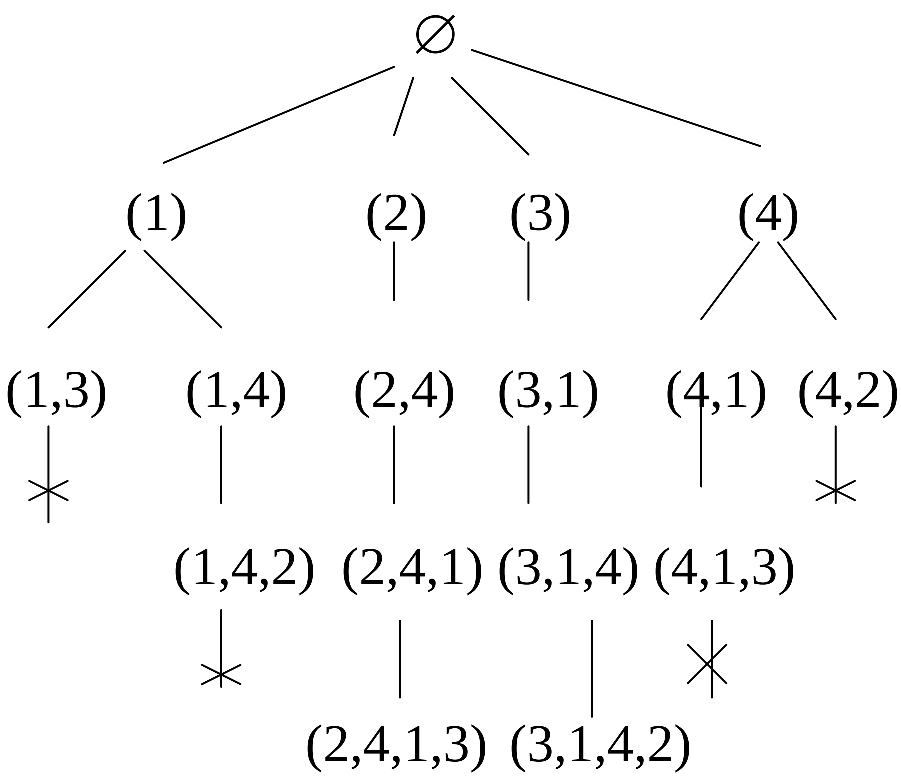
If one remembers the fact that each diagonal can only hold 1 queen, one can further reduce the number of solutions to just 2056 configurations!

This process is called ***pruning (preclusion)*** in backtracking.

Consider the general backtrack search algorithm for the simplified 8-queen problem.



A Complete Backtrack Search Tree for 4-Queen Problem:



```
const int BOARD_SIZE = 8; // squares per row or column
class Queens
{
public:
  Queens(); // Creates an empty square board.

  void clearBoard();     // Sets all squares to EMPTY.
  void displayBoard(); // Displays the board.

  bool placeQueens(int currColumn);
  // --------------------------------------------------
  // Places queens in columns of the board beginning at the
  // column specified.
  // Precondition: Queens are placed correctly in columns
  // 1 through currColumn-1.
  // Postcondition: If a solution is found, each column of
  // the board contains one queen and the function
  // returns true; otherwise, returns false (no solution
  // exists for a queen anywhere in column currColumn).
  // --------------------------------------------------

private:
  enum Square {QUEEN, EMPTY}; // states of a square
  Square board[BOARD_SIZE][BOARD_SIZE];

  void setQueen(int row, int column);
  // Sets the square on the board in a given row and column
  // to QUEEN.
   void removeQueen(int row, int column);
```

```
   // Sets the square on the board in a given row and column
   // to EMPTY.

   bool isUnderAttack(int row, int column);
   // Determines whether the square on the board at a given
   // row and column is under attack by any queens in the
   // columns 1 through column-1.
   // Precondition: Each column between 1 and column-1has
   // a queen placed in a square at a specific row. None of
   // these queens can be attacked by any other queen.
   // Postcondition: If the designated square is under
   // attack, returns true; otherwise, returns false.

   int index(int number);
   // Returns the array index that corresponds to
   // a row or column number.
   // Precondition: 1 <= number <= BOARD_SIZE.
   // Postcondition: Returns adjusted index value.
}; // end class
```

```
bool Queens::placeQueens(int currColumn)
// Calls: isUnderAttack, setQueen, removeQueen.
{
  if (currColumn > BOARD_SIZE)
    return true; // base case
  else
  {  bool queenPlaced = false;
    int row = 1; // number of square in column
    while ( !queenPlaced && (row <= BOARD_SIZE) )
    {  // if square can be attacked
      if (isUnderAttack(row, currColumn))
        ++row; // then consider next square in currColumn
      else // else place queen and consider next
      {  // column
        setQueen(row, currColumn);
        queenPlaced = placeQueens(currColumn+1);
        // if no queen is possible in next column,
        if (!queenPlaced)
        {  // backtrack: remove queen placed earlier
          // and try next square in column
          removeQueen(row, currColumn);
          ++row;
        } // end if
      } // end if
    } // end while
    return queenPlaced;
  } // end if
} // end placeQueens
```

III. Computing Recursively Defined Objects:

Let's consider a "grammar" G and the language L_G generated by G.

Language: A set of legal strings.
Grammar: A set of production rules specifying how a legal string can be formed.

Remark: Grammars define only the **syntax** (form) of a language, not its **semantics** (meaning).

Q: Why study formal grammar?

- It constitutes a recursive definition of a language. (More precisely, it serves as a set of rules by which words in a language are constructed.)
- It can be used to derive a recursive algorithm for recognizing or verifying that a particular string is "in the language" (i.e., it satisfies the rules of the grammar). (So-called "Recursive Descent" parsers are common in compilers.)

Defn: A **grammar** G = (N,T,S,π) consists of:

- N: a set of non-terminal symbols. (Non-terminal symbols are NOT elements of the language.)
- T: a set of terminal symbols (Atomic elements of the actual language)
- S: a designated non-terminal symbol in N, called the "starting symbol."
- π: a set of "production rules" which describe how non-terminal symbols are re-written in terms of other non-terminal symbols and terminal symbols.

Defn: A language defined by G, denoted by L_G, is the set of all strings, which are sequences of terminals, that can be derived from the starting symbol using only the production rules in G.

Dfn: Length of a string = # terminals in the string.

Empty String:

An empty string is a string with length 0.

Example: Consider a "language" consists of all strings with a sequence n a's followed by a sequence of 2n b's, where $n \in N$. Hence,

$L_G = \{abb, aabbbb, aaabbbbbb, \ldots\} = \{a^n b^{2n} \mid n \geq 1\}$.

This language can also be defined by the following grammar $G = (N,T,S,\pi)$:

$N = \{C, A, B\}$,

$T = \{a, b\}$.

$S = C$,

$\pi = \{$
$C \rightarrow ABB$
$C \rightarrow ACBB$
$A \rightarrow a$
$B \rightarrow b$
$\}$

Other Notation:

The production rules $C \rightarrow ABB$ and $C \rightarrow ACBB$ can be combined using $C \rightarrow ABB \mid ACBB$.

Q: How can we generate a given string using the production rules of a grammar?

Example: Consider a string aabbbb.

$C \to ACBB$	$(C \to ACBB)$
$\to AABBBB$	$(C \to ABB)$
$\to aABBBB$	$(A \to a)$
$\to aaBBBB$	$(A \to a)$
$\to aabBBB$	$(B \to b)$
$\to aabbBB$	$(B \to b)$
$\to aabbbB$	$(B \to b)$
$\to aabbbb$	$(B \to b)$

Q: Among all production rules in G, which rule should we use to generate a given string?

Q: How can a compiler determine whether a given string is a legal identifier in the language?

Q: Given a string s. How can we decide whether $s \in L_G$?
Need to construct a language recognizer for G!

Constructing Language Recognizer:

For the above language, observe that

(1) Any string $s \in L_G$ implies $s \in \{a^n b^{2n} \mid n \geq 1\}$.
(2) Any string s has 3 or more terminals.
(3) Every a in s corresponds to two b's in s.
(4) By deleting the first a and the last 2 b's of s, the remaining string must also be a legal string in L_G. Any legal string s can be recognized by (repeatedly) deleting the first a and the last 2 b's of the string and then apply the same recursive method to the remaining substring.

A Language Recognizer for G:

```
#include <string>
using namespace std;

bool inLanguage(string str)
{
   int len = str.length();         // find length of string s

   if (len < 3)                    // s must have ≥ 3 chars
      return false;

   if ( (str.at(0) ==‘a’) &&     // removing first a and last 2 b's
           (str.at(len-2) == ‘b’) &&
              (str.at(len-1) == ‘b’) )
      {  if (len == 3)             // s has the form abb
            return true;
         else //recursive call after deleting first a and last 2 b's
            return inLanguage(str.substring(1,len-3));
      }

   return false;
}  // end inLanguage
```

Another Example: Palindrome recognizer.

A palindrome is a string that reads the same from front to back as well as from back to front.

Example:

MOM
RACAR
RADAR
A MAN, A PLAN, A CANAL, PANAMA.
(Ignoring space and punctuations)

Grammar for Palindrome:

<pal> = empty_string | <ch> | a<pal>a | b<pal>b | …|
Z<pal>Z
<ch> = a | b | … | z | A | B | … | Z

Remark: We use < > to denote non-terminal symbols.

HW: Implement a palindrome recognizer with the following function:

```
bool isPal(string str)
```

Example: A grammar for positive decimal (integer).

<*positive decimal*> = <*decimal digit*> |
<*positive decimal*> <*decimal digit*>

<*decimal digit*> = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The grammar	The recursive recognizer
a positive decimal number is a single decimal digit or a positive decimal number followed by a single decimal digit	bool isPositiveDecimalNumber(string s) **if** s.length( ) == 1 return (s[0] is a decimal digit) **else** **if** s[s.length()-1] is a decimal digit **return** isPositiveDecimalNumber(s.substr(0,len-2)) **else** **return** false

Remark: Grammar-driven recursive algorithms may either

(1) Pass modified versions of the string to subsequent recursive calls, or

(2) Pass “first, last” indices (like in binary search).

Example: Grammars for Simple Algebraic Expressions.

Recall that we generally use *infix notation* for arithmetic expressions in which binary operator is placed between two operands to which it is being applied.

Example:

a ∗ b – c

a / b ∗ c

Problem: The meaning of an infix notation is not determined solely by the grammar, but also operator precedence rules and the use of parentheses when those rules need to be overridden.

Remedy:

Use postfix or prefix notations since they do not give rise to ambiguity as in infix notation.

(1) infix = infix operator infix | operand **|** (infix)
(2) prefix = operator prefix prefix | operand
(3) postfix = postfix postfix operator | operand

Example:

Infix	Prefix	Postfix
A ∗ B – C	–∗ABC	AB∗C–
A ∗(B–C)	∗A–BC	ABC–∗

Remark: Neither prefix nor postfix grammars are ambiguous. Hence *neither* requires parentheses *nor* operator precedence rules. This makes them preferred by compilers and interpreters. Postfix is generally the more useful, and hence common, to compilers and interpreters because its evaluation is simpler.

Problems:

We need to be able to

(1) Convert from infix to prefix (postfix), and

(2) Evaluate prefix (postfix) expression.

TBA.