Grammars, Stack Frames, and Recursion
(Preface for Chapter 6)

Hyunyoung Lee
Grammars

- Rewrite system / production system / rule-based system

- Each rule looks like this: (LHS) $\rightarrow$ (RHS)

- How to apply the rules:
  - start with a particular LHS
  - then continually do replacements according to the rules until you can’t do any more

- Applications: expert systems, logic programming, way to describe natural languages, way to describe programming languages, etc.
Example of a Grammar

Balanced parentheses: generate all (and only) strings of parentheses, where the parentheses are “balanced”

$S \rightarrow ( \; S \; )$

$S \rightarrow S \; S$

$S \rightarrow \text{empty string (i.e., stop)}$

The calculator example we will see in Chapter 6.
Stack Frames

When a program is executing, there are some important parts of the computer’s memory that we need to know about:

- to store the executable code
- to store data used and variables declared by the program and control information
- The data part is divided into two parts, the stack and the heap.

When a function begins executing, a chunk of (contiguous) memory on the stack is allocated to that function (called “stack frame” or “activation record”). It holds:

- local variables, the formal parameters, the “return value”
Example of Function Calls and Returns

main starts
main calls p
p calls q
q returns
p calls r
r calls s
s returns
r returns
p returns
Recursion

• A recursive function is a function that invokes itself, either directly or indirectly.

• An alternative to iteration.

• Recursive functions can be useful in solving problems that can be broken down into smaller or simpler subproblems of the same type. A base case should eventually be reached, at which time the breaking down (recursion) will stop.

• Implements the divide-and-conquer problem solving methodology.

• Examples: The calculator program in Chapter 6, string-related functions, computing the greatest common divisor, quicksort, and so on.
Consider a function for solving the count-down problem from some number \( n \) down to 0:

- The base case is when \( n \) is already 0: the problem is solved and we “blast off!”

- If \( n \) is greater than 0, we count off \( n \) and then recursively count down from \( n-1 \)
count_down (cont.)

#include <iostream>
using namespace std;

void count_down( int n );

int main(void)
{
    count_down(2);
    return 0;
}

void count_down( int n )
{
    if (n == 0)
    {
        cout << "\n*** BLAST OFF ***\n";
    }
    else if (n > 0) {
        cout << n << "! ";
        cout << count_down(n-1);
    }
    cout << "Exiting from count_down ...\n";
}
If a program contains a line like `count_down(2);`

1. `count_down(2)` generates the output `2 !`, then it calls `count_down(1)`
2. `count_down(1)` generates the output `1 !`, then it calls `count_down(0)`
3. `count_down(0)` generates the output `*** BLAST OFF ***` and Exiting from `count_down ...`, then returns to `count_down(1)`
4. Back in `count_down(1)`, it outputs Exiting from `count_down ...`, then returns to `count_down(2)`
5. Back in `count_down(2)`, it outputs Exiting from `count_down ...`, then returns to `main()`
Each time a recursive function is called, a new copy of the function runs, with new copies of parameters and local variables being created.

As each copy finishes executing, it returns to the copy of the function that called it.

When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function.
Stopping the Recursion

• A recursive function should include a test for the base cases.

• In the sample program, the test is:

  if (n == 0)

• With each recursive call, the parameter controlling the recursion should move closer to the base case.

• Eventually, the parameter reaches the base case and the chain of recursive calls terminates.
Types of Recursion

• Direct recursion
  a function calls itself

• Indirect recursion
  function A calls function B, and function B calls function A. Or,
  function A calls function B, which calls function C, which calls ..., which calls function A.
Example 2: Recursive gcd

- The greatest common divisor (gcd) of two positive integers is the largest integer that is a divisor of both of them.
- The Greek mathematician Euclid discovered that
  - If $y$ divides $x$, then $\text{gcd}(x, y)$ is just $y$
  - Otherwise, the $\text{gcd}(x, y)$ is the gcd of $y$ and the remainder of dividing $x$ by $y$
int gcd( int x, int y )
{
    int r;
    if ((r = x%y) == 0)
        return y;
    else
        return gcd(y, r);
}
Example 3: Recursive Factorial Function

• The factorial of a nonnegative integer \( n \) is the product of all positive integers less than or equal to \( n \)

• Factorial of \( n \) is denoted by \( n! \)

• The factorial of 0 is 1

\[ 0! = 1 \]

\[ n! = n \times (n-1) \times \ldots \times 2 \times 1 \quad \text{if} \quad n > 0 \]
Factorial of $n$ can be expressed in terms of the factorial of $n-1$

$$0! = 1$$

$$n! = n \times (n-1)!$$

Recursive function

```c
int factorial(int n)
{
    if (n == 0) return 1;
    else return n * factorial(n-1);
}
```
Example 4: The QuickSort Algorithm

- Recursive algorithm that can sort an array
- Determines an element to use as pivot_value
  
  pivot value

- Once pivot value is determined, values are shifted so that elements in sublist1 are < pivot and elements in sublist2 are >= pivot (“partition”)
- Then recursively sorts sublist1 and sublist2
- Base case: sublist has size <=1
- Three parameters: array to be sorted, start, end
The QuickSort Algorithm (1)

int partition(int arr[], int start, int end)
{
    // The pivot element is taken to be the element
    // at the start of the subrange to be partitioned
    int pivotValue = arr[start];
    int pivotPos = start, pos = start+1;

    // Rearrange the rest of the array elements to
    // partition the subrange from start to end
    for ( ; pos <= end; pos++)
        if (arr[pos] < pivotValue) // arr[pos] is the current item
            { swap(&arr[pivotPos+1], &arr[pos]);
              swap(&arr[pivotPos], &arr[pivotPos+1]);
              pivotPos++;
            }
    return pivotPos;
}
void quickSort(int arr[], int start, int end)
{
    if (start >= end) then arr has <= 1 elem (already sorted)
    if (start < end)
    {
        // Partition the array and get the pivot point
        int p = partition(arr, start, end);
        // Sort the portion before the pivot point
        quickSort(arr, start, p-1);
        // Sort the portion after the pivot point
        quickSort(arr, p+1, end);
    }
}
Recursion vs. Iteration

• Benefits (+), disadvantages(−) for recursion:
  + Natural formulation of solution to certain problems
  + Results in shorter, simpler functions
  − May not execute very efficiently

• Benefits (+), disadvantages(−) for iteration:
  + Executes more efficiently than recursion
  − May not be as natural as recursion for some problems