4. Recursion

4.1. Recursive Definitions

- There are many programming concepts that define themselves.
- As it turns out, formal restrictions imposed on definitions such as existence and uniqueness are satisfied and no violation of the rules takes place. These definitions are called **recursive definitions**.
- Recursive definitions are used primarily to define infinite sets.

4.1. Recursive Definitions contd…

- When defining such as set, giving a complete list of elements is impossible, and for large finite sets, it is inefficient.
- A recursive definition consists of 2 parts :
  - In the first part, called the *anchor* or *ground case* – the basic elements that are building blocks of all other elements of the set are listed.
  - In the second part, rules are given that allow for the construction of new objects out of basic elements or objects that have already been constructed.

4.1. Recursive Definitions contd…

- These rules are applied again and again to generate new objects.
- Example : To construct the set of natural numbers, one basic element, 0, is singled out, and the operation of incrementing by 1 is given as :
  - 0 € N;
  - if *n* € N, then (n+1) € N;
  - there are no other objects in the set N.

  N consists of the following items : 0,1,2,3,4,5,6,7,9

4.1. Recursive Definitions contd…

- Recursive Definitions serve two purposes:
  - Generating new elements
  - Testing whether an element belongs to a set.
- Recursive definitions are frequently used to define functions and sequence of numbers.

4.2. Method calls and recursion implementation

- What happens when a method is called? If the method has formal parameters, they have to be initialized to the values passed as actual parameters.
- In addition, the system has to know where to resume execution of the program after the method has finished.
- The method can be called by other methods or by the main program (main ()).

4.2. Method calls and recursion implementation contd…

- The information indicating where it has been called from has to be remembered by the system.
- This could be done by storing the return address in main memory in a place set aside for return addresses, but we do not know in advance how much space might be needed, and allocating too much space for that purpose alone is not efficient.
- For a method call, more information has to be stored than just a return address. Therefore, dynamic allocation using the run-time stack is a much better solution.

4.2. Method calls and recursion implementation contd…

- What information should be preserved when a method is called?
- First, automatic (local) variables must be stored.
- If method f1(), which contains a declaration of an automatic variable x, calls method f2(), which locally declares the variable x, the system has to make a distinction between these two variables x.
- If f2 () uses a variable x, then its own x is meant; if f2 () assigns a value to x, then x belonging to f1 () should be left unchanged.
- When f2 () is finished, f1 () can use the value assigned to its private x before f2 () was called.
- This is especially important in the context of the present chapter, when f1 () is the same as f2 (), when a method calls itself recursively.

4.2. Method calls and recursion implementation contd…

- The state of each method, including main ( ), is characterized by
  - the contents of all automatic variables,
  - the values of the method's parameters, and
  - the return address indicating where to restart its caller.
- The data area containing all this information is called an *activation record* or a *stack frame* and is allocated on the run-time stack.
- An activation record exists for as long as a method owning it is executing.
- This record is a private pool of information for the method, a repository that stores all information necessary for its proper execution and how to return to where it was called from.
- Activation records usually have a short lifespan because they are dynamically allocated at method entry and deallocated upon exiting.
- Only the activation record of main ( ) outlives every other activation record.

4.2. Method calls and recursion implementation contd…

- An activation record usually contains the following information:
  - Values for all parameters to the method, location of the first cell if an array is passed or a variable is passed by reference, and copies of all other data items.
  - Local (automatic) variables that can be stored elsewhere, in which case, the activation record contains only their descriptors and pointers to the locations where they are stored.
  - The return address to resume control by the caller, the address of the caller's instruction immediately following the call.
  - A dynamic link, which is a pointer to the caller's activation record.
  - The returned value for a method not declared as void. Because the size of the activation record may vary from one call to another, the returned value is placed right above the activation record of the caller.

4.2. Method calls and recursion implementation contd…

- If a method is called either by main () or by another method, then its activation record is created on the run-time stack.
- Creating an activation record whenever a method is called allows the system to handle recursion properly.
- **Recursion** is calling a method that happens to have the same name as the caller.
- Therefore, a recursive call is not literally a method calling itself, but rather an instantiation of a method calling another instantiation of the same original.
- These invocations are represented internally by different activation records and are thus differentiated by the system.

4.3. Anatomy of a Recursive Call

- The function that defines raising any number x to a nonnegative integer power *n* is good example of a recursive function.
- The most natural definition of this functions given by:

$$X^n = \begin{cases} 1 & \text{if } n = 0 \\ x.x^{n-1} & \text{if } n > 0 \end{cases}$$

4.3. Anatomy of a Recursive Call contd…

- A Java method for computing x^n can be written directly from the definition of a power:

```
double power (double x, int n) {
    if (n == 0)
        return 1.0;
    else
        return x* power (x, n-1);
}
```

4.4. The implementation of recursion

- Thee are several implementations of recursions such as
  - Tail recursion
  - NonTail Recursion
  - Indirect recursion
  - Nested Recursion
  - Excessive recursion

4.4.1. Tail recursion

- All recursive definitions contain a reference to a set or function being defined.
- There are, however, a variety of ways such a reference can be implemented.
- This reference can be done in a straightforward manner or in an intricate fashion, just once or many times.
- There may be many possible levels of recursion or different levels of complexity.

4.4.1. Tail recursion contd…

- Tail recursion is characterized by the use of only one recursive call at the very end of a method implementation.
- In other words, when the call is made, there are no statements left to be executed by the method; the recursive call is not only the last statement but there are no earlier recursive calls, direct or indirect.

4.4.1. Tail recursion contd…

- Example : The method tail() defined as

```
void tail (int i) {
    if (I > 0) {
        System.out.print (I + " ");
        tail (i-1);
    }
}
```

4.4.1. Tail recursion contd…

- Tail recursion is simply a glorified loop and can be easily replaced by one.
- In this example, it is replaced by substituting a loop for the if statement and incrementing or decrementing the variable i in accordance with the level of recursion.
- In this way, tail ( ) can be expressed by an iterative method:

4.4.1. Tail recursion contd…

```
void iterativeEquivalentOfTail ( int i ) {
    for (   ; i > 0; i-- )
    System. out. print (i+ "");
}
```

4.4.1. Tail recursion contd…

- Is there any advantage in using tail recursion over iteration?
  - For languages such as Java, there may be no compelling advantage, but in a language such as Prolog, which has no explicit loop construct (loops are simulated by recursion), tail recursion acquires a much greater weight.
  - In languages endowed with a loop or its equivalents, such as an if statement combined with a goto statement or labeled statement, tail recursion should not be used.

4.4.2. NonTail Recursion

- Another problem that can be implemented in recursion is printing an input line in reverse order.
- Here is a simple recursive implementation:

```
void reverse() {
        char ch = getChar();
        if (ch != '\n') {
                reverse() ;
                System.out.print(ch);
        }
}
```

4.4.3. Indirect recursion

- Direct recursion - where a method f ( ) called itself.
- f ( ) can call itself indirectly via a chain of other calls. For example, f ( ) can call *g(),* and g ( ) can call f ( ) . This is the simplest case of indirect recursion.
- The chain of intermediate calls can be of an arbitrary length, as in:

  f () → f1() → f2() → …….. → fn() → f()

- There is also the situation when f ( ) can call itself indirectly through different chains.
- Thus, in addition to the chain just given, another chain might also be possible. For instance

  f() → g1 () → g2() → …… → gm() → f()

4.4.3. Indirect recursion

- This situation can be exemplified by three methods used for decoding information.
  - **receive ()** stores the incoming information in a buffer
  - **decode ()** converts it into legible form
  - **store ()** stores it in a file
- receive () fills the buffer and calls decode (), which in turn, after finishing its job, submits the buffer with decoded information to store ().
- After store () accomplishes its tasks, it calls receive () to intercept more encoded information using the same buffer.
- Therefore, we have the chain of calls

  receive() → decode() → store() → receive() → decode() → ……

4.4.3. Indirect recursion contd…

- Above three methods work in the following manner:

```
receive (buffer)
        while buffer is not filled up
                if information is still incoming
                        get a character and store it in buffer;
                else exit( );
                decode (buffer);
decode (buffer)
        decode information in buffer;
        store (buffer);
store (buffer)
        transfer information from buffer to file;
        receive (buffer);
```

4.4.3. Indirect recursion contd…

- As usual in the case of recursion, there has to be an anchor in order to avoid falling into an infinite loop of recursive calls.

Nested Recursion

- A more complicated case of recursion is found in definitions in which a function is not only defined in terms of itself, but also is used as one of the parameters. The following definition is an example of such a nesting:

$$h(n) = \begin{cases} 0 & \text{if } n = 0 \\ N & \text{if } n > 4 \\ h(2 + h(2n)) & \text{if } n <= 4 \end{cases}$$

4.4.4. Nested Recursion contd…

- Function *h* has a solution for all *n* >= 0. This fact is obvious for all *n* > 4 and *n = 0,* but it has to be proven for n = 1,2,3, and 4. Thus, *h(2) = h(2 + h(4)) = h(2 + h(2 + h*(8))) = 12. (What are the values *of h(n)* for n = 1,3, and 4?)

4.4.4. Nested Recursion contd…

- Another example of nested recursion is a very important function originally suggested by Wilhelm Ackermann in 1928 and later modified by Rozsa Peter:

$$A(n,m) = \begin{cases} \mathbf{m+1} & \mathbf{if\ n = 0} \\ \mathbf{A(n-1,1)} & \mathbf{if\ n > 0,\ m = 0} \\ \mathbf{A(n-1,\ A(n,m-1))} & \mathbf{otherwise} \end{cases}$$

8.4.4. Nested Recursion contd…

- Above function is interesting because of its remarkably rapid growth.
- It grows so fast that it is guaranteed not to have a representation by a formula that uses arithmetic operations such as addition, multiplication, and exponentiation.
- To illustrate the rate of growth of the Ackermann function, we need only show that

$$A(3,m) = 2^{m+3} - 3$$

$$A(4,m) = 2^{2^{\cdot^{\cdot^{2}}}} - 3$$

  with a stack of m 2s in the exponent; $A(4,I) = 2^{2^{16}} - 3 = 2^{65536} - 3$, which exceeds even the number of atoms in the universe (which is 10^{80} according to current theories).
- The definition translates very nicely into Java, but the task of expressing it in a nonrecursive form is truly troublesome.

4.4.4. Excessive recursion

- Logical simplicity and readability are used as an argument supporting the use of recursion.
- The price for using recursion is slowing down execution time and storing on the run-time stack more things than required in a nonrecursive approach.
- If recursion is too deep (for example, computing $5.6^{100'000}$), then we can run out of space on the stack and our program terminates abnormally by raising an unrecoverable StackOverflowError.
- But usually, the number of recursive calls is much smaller than 100,000, so the danger of overflowing the stack may not be imminent. However, if some recursive function repeats the computations for some parameters, the run time can be prohibitively long even for very simple cases.

4.4.4. Excessive recursion contd…

- Consider Fibonacci numbers. A sequence of Fibonacci numbers is defined as follows:

$$Fib(n) = \begin{cases} n & \text{if } n = 0 \\ Fib(n-2)+Fib(n-1) & \text{otherwise} \end{cases}$$

- The definition states that if the first two numbers are 0 and 1, then any number in the sequence is the sum of its two predecessors. But these predecessors are in turn sums of their predecessors, and so on, to the beginning of the sequence. The sequence pro-duced by the definition is

  0,1,1,2,3, 5,8,13,21, 34, 55,89,...

4.4.4. Excessive recursion contd…

- How can this definition be implemented in Java? It takes almost term-by-term translation to have a recursive version, which is

```
int Fib (int n) {
   if (n < 2)
            return n;
   else return Fib(n-2) + Fib(n-l);
}
```

- The method is simple and easy to understand but extremely inefficient fibonacci heap.