### Defining Functions

- Functions with a **finite domain** can be described by specifying for each element in the domain the associated element in the **codomain**.

- **Examples:**
  - 
    \[
    f(x) = \begin{cases} 
      1 & \text{if } x = 1 \\
      0 & \text{if } x = 0 \text{ or } x = 3 
    \end{cases}
    \]
  - Let \( x \) a real. \( f(x) = 1 \) if \( 0 \leq x \leq 3 \)

- The two basic mechanisms for defining functions on **infinite** domains are
  - **explicit** definitions and
  - **recursive** definitions.

### Recursive Definitions

- A **recursive definition** of a function consists of
giving an expression for every domain element \( x \)
that indicates how \( f(x) \) is obtained from previously defined functions and values of \( f \) for "smaller" arguments (by composition).

  **Self-references**

- The **recursion principle** specifies under which conditions such definitions with self-references are **well-formed**.

- **Example**

  The number of **permutations** of \( n \) elements is \( n! \) (or \( \text{fact}(n) \), read \( n \) factorial).

  **Order**

  This function can be defined recursively by:

  \[
  \text{fact}(n) = \begin{cases} 
    n & \text{if } n = 0 \text{ then } 1 \\
    n \ast \text{fact}(n - 1) & \text{else}
  \end{cases}
  \]

  The values \( \text{fact}(n) \), for all \( n > 0 \), depend on values \( \text{fact}(k) \), where \( k \) is smaller than \( n \). Here \( k = n - 1 \).

  This case is called the **general case**.

  \( n = 0 \) is called the **exit condition** or the **basis condition**.
Well-Formed Recursive Definitions

- A **well-formed recursive definition** of a function \( f \) consists of two parts:
  - the **basis case** defines the function \( f \) for the “smallest” arguments in terms of previously defined functions (including constants), (no \( f \)).
  - the **general case** defines values \( f(x) \) in terms of previously defined functions and values \( f(y) \) for “smaller” arguments \( y \).

- In the case of definitions of functions over the natural numbers, smaller is interpreted in the usual sense.

Later on we will see recursive definitions of functions on other domains, such as lists, where “smaller” necessarily has to be interpreted differently. We use an **ordefining** on the elements we consider.

Example: Squares

- There are different ways to define a function.

  - For instance, the function that squares its argument can be defined **explicitly** in terms of multiplication,
    \[
    \text{square}(x) = x \times x,
    \]
  or by **recursion**:
    \[
    \text{square}(x) = \begin{cases} 
      0 & \text{if } x = 0 \\
      \text{square}(x-1) + 2x - 1 & \text{else} 
    \end{cases}
    \]
  From the recursive definition we get the following function values:
    \[
    \begin{align*}
    \text{square}(0) &= 0 \\
    \text{square}(1) &= \text{square}(0) + 1 = 1 \\
    \text{square}(2) &= \text{square}(1) + 3 = 4 \\
    \text{square}(3) &= \text{square}(2) + 5 = 9 \\
    \text{square}(4) &= \text{square}(3) + 7 = 16 \\
    \end{align*}
    \]

  The two definitions above define the same function, as
  \[
  x \times x = (x-1) \times (x-1) + 2x - 1.
  \]

Computing Values of Recursively Defined Functions

- The **evaluation** of a recursively defined function for a specific argument involves two kinds of operations:
  - **substitutions** use the function definition to “expand” an application, whereas
  - **simplifications** use knowledge about previously defined (or primitive) functions to “reduce” an expression.

- The evaluation process will **terminate** if the definition is **well-formed**.

Example:

\[
\begin{align*}
\text{fact}(5) &= 5 \times \text{fact}(4) \quad (\text{substitution}) \\
&= 5 \times (4 \times \text{fact}(3)) \quad (\text{substitution}) \\
&= 20 \times \text{fact}(3) \quad (\text{substitution}) \\
&= 120
\end{align*}
\]

Addition and gcd

- **Addition**
  \[
  \text{add}(a, b) = \begin{cases} 
    a & \text{if } b = 0 \\
    \text{add}(a, b-1) + 1 & \text{otherwise} 
  \end{cases}
  \]

- **Greatest Common Divisor**
  \[
  \text{gcd}(a, b) = \begin{cases} 
    a & \text{if } b = 0 \\
    \text{gcd}(b, a \mod b) & \text{otherwise} 
  \end{cases}
  \]
Fibonacci Numbers

- The recursive definition of the following well-known function (Fibonacci function) employs the function values for several smaller arguments:

\[fib(n) = \begin{cases} 
1 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
fib(n-1) + fib(n-2) & \text{if } n > 1 
\end{cases}\]

- The corresponding function values are called Fibonacci numbers:

\[
\begin{align*}
fib(0) &= 1 \\
fib(1) &= 1 \\
fib(2) &= fib(1) + fib(0) = 2 \\
fib(3) &= fib(2) + fib(1) = 3 \\
fib(4) &= fib(3) + fib(2) = 5 \\
fib(5) &= fib(4) + fib(3) = 8 \ldots
\end{align*}
\]

- The Fibonacci numbers were originally defined to count the number of rabbits after \(n\) generations, but they pop up in an amazing variety of places:
  - The Golden Ratio of architecture, \(\phi \approx \frac{fib(n)}{fib(n-1)} = (1 + \sqrt{5})/2 \approx 1.618\)
  - The angles in spiral pine cones grow as ratios of Fibonacci numbers.
  - They arise in the analysis of computer algorithms.

Well-defined Functions

- A key requirement of a recursive definition is that it be formulated in terms of function values for smaller arguments.

- A recursive function is said well-defined, if it is possible to compute \(f(n)\) for all \(n\) for which the function is defined. Otherwise it is said partially defined.

- Consider this definition,

\[
F(x) = \begin{cases} 
0 & \text{if } x = 0 \\
F(x+1) + 1 & \text{else}
\end{cases}
\]

and corresponding attempts at computing function values,

\[
\begin{align*}
F(0) &= 0 \\
F(1) &= F(2) + 1 = F(3) + 2 = F(4) + 3 \ldots
\end{align*}
\]

This function is defined for one argument only. So \(F\) is not well-defined.

- What about the function \(G\), defined for positive integers by

\[
G(n) = \begin{cases} 
0 & \text{if } n = 1 \\
1 + G(n/2) & \text{if } n \text{ is even} \\
G(3n-1) & \text{if } n \text{ is odd and } n > 1
\end{cases}
\]

A new function \(H\)

- It has been conjectured (and shown up to one trillion) that a slight modification,

\[
H(n) = \begin{cases} 
0 & \text{if } n = 1 \\
1 + H(n/2) & \text{if } n \text{ is even} \\
H(3n+1) & \text{if } n \text{ is odd and } n > 1
\end{cases}
\]

defines a well-defined function on all positive integers.

- \(H(2)\): \(H(1)\)
- \(H(10)\): \(H(5)\)→\(H(16)\)→\(H(8)\)→\(H(4)\)→\(H(2)\)→\(H(1)\)
- \(H(17)\): \(H(52)\)→\(H(26)\)→\(H(13)\)→\(H(40)\)→\(H(20)\)→\(H(10)\)→\(H(5)\)→\(H(16)\)→\(H(8)\)→\(H(4)\)→\(H(2)\)→\(H(1)\)
- \(H(21)\): \(H(64)\)→\(H(32)\)→\(H(16)\)→\(H(8)\)→\(H(4)\)→\(H(2)\)→\(H(1)\)
- \(H(35)\): \(H(106)\)→\(H(53)\)→\(H(160)\)→\(H(80)\)→\(H(40)\)→\(H(20)\)→\(H(10)\)→\(H(5)\)→\(H(16)\)→\(H(8)\)→\(H(4)\)→\(H(2)\)→\(H(1)\)

\(H\) counts the number of downward steps this path takes.

\(H(2) = 1\)
\(H(17) = 9\)
\(H(21) = 6\)
and \(H(35) = 10\).
More General Recursive Definitions

- **Example:**
  
  \[ M(n) = \begin{cases} 
  n - 10 & \text{if } n > 100 \\
  M(M(n + 11)) & \text{if } n \leq 100 
  \end{cases} \]

  This function is known as “McCarthy’s 91 function.”
  
  Its definition uses **nested** recursive function applications.

- Consider one instance,
  
  \[
  M(99) = M(M(110)) \quad \text{(since } 99 < 100) \\
  = M(100) \quad \text{(since } 110 > 100) \\
  = M(M(111)) \quad \text{(since } 100 \leq 100) \\
  = M(101) \quad \text{(since } 111 > 100) \\
  = 91 \quad \text{(since } 101 > 100) 
  \]

- Is this function defined for all arguments \( n \leq 100 \)?
  
  The function is in fact defined for all positive integers and remarkably takes the value 91 for all arguments less than or equal to 101.

Different evaluations of a recursive function

- **Example:**
  
  \[ f(x, y) = \begin{cases} 
  0 & \text{if } x = 0 \\
  f(x - 1, f(x, y)) & \text{otherwise} 
  \end{cases} \]

  - Consider \( f(1, 1) \).
    
    **Innermost evaluation**
    
    \[ f(1, 1) = f(0, f(1, 1)) = f(0, f(0, f(1, 1))) = \ldots \]
    
    **Outermost evaluation**
    
    \[ f(1, 1) = f(0, f(1, 1)) = 0 \]
    
    **Simultaneous**
    
    \[ f(1, 1) = f(0, f(1, 1)) = 0 \]

  - Innermost evaluation does not always terminate.
  
  - Outermost evaluation does always terminate.

  - Innermost evaluation is more efficient than outermost evaluation (Convergence).

Sorting

- We next design a function for **sorting a list of integers**.

- Sorting is an important problem for which a large variety of different algorithms have been proposed (List them?).

- The method we will explore is based on the following idea. To sort a list \( L \),
  
  - first **split** \( L \) into two disjoint sublists (of about equal size),
  
  - then (recursively) **sort** the sublists, and
  
  - finally **merge** the (now sorted) sublists.

  This recursive method is known as **Merge-Sort**.

- It evidently requires us to define suitable functions for
  
  - splitting a list into two sublists and
  
  - merging two sorted lists into one sorted list.

- A recursive definition appears natural.
Tracing Mergesort

- It is important to be able to trace the execution of the mergesort program to convince yourself that it works correctly.

In the course of executing the recursive algorithm, the computer has to keep track of what work still needs to be done as it is interrupted with additional recursive calls.

The Tower of Hanoi

- The tower of Hanoi consists of a fixed number of disks stacked on a pole in decreasing size, that is, with the smallest disk at the top.
- There are two other poles and the task is to transfer all disks from the first to the third pole, one at a time without ever placing a larger disk on top of a smaller one.
- There is an elegant solution to this problem by recursion.

Tower Moves

- First consider how many moves are needed, at the least, to transfer a tower of \( k \) disks.
- Observe that we need to get to the following intermediate configuration, so as to be able to move the largest disk.

That is, we have to transfer the \( k - 1 \) smaller disks to the middle pole, we can then move the largest disks from the first to the third pole, and finally the \( k - 1 \) smaller disks from the second pole to the third pole.
- Let \( M(k) \) be the minimum number of moves required to transfer \( k \) disks from one pole to another pole. This function \( M \) satisfies the recursive identity:
\[
M(k) = M(k - 1) + 1 + M(k - 1) = 2M(k - 1) + 1,
\]
for all \( k > 0 \).
In addition, we set \( M(0) = 0 \), so that by the above identity \( M(1) = 1 \), which is correct as one move suffices to transfer a tower containing only a single disk.
Minimum Number of Moves

- $M(0) = 0$
- $M(k) = M(k-1) + 1 + M(k-1) = 2M(k-1) + 1$ for all $k > 0$.

- Let us evaluate the function for some arguments:
  
  \[
  \begin{align*}
  M(0) &= 0 \\
  M(1) &= 2M(0) + 1 = 1 \\
  M(2) &= 2M(1) + 1 = 3 \\
  M(3) &= 2M(2) + 1 = 7 \\
  M(4) &= 2M(3) + 1 = 15 \\
  M(5) &= 2M(4) + 1 = 31 \\
  M(6) &= 2M(5) + 1 = 63
  \end{align*}
  \]

- The values grow fairly fast. In fact one can show that the function $M$ can be explicitly defined by
  
  $M(k) = 2^k - 1,$
  
  for all $k > 0$. That is, function values grow exponentially with the argument.

- This tells us that a lot of moves are needed to transfer a tall tower, though we don’t know the actual sequence of moves yet.

Poles are represented by $x$, $y$, and $z$.

- We represent a move as a pair of integers $(x, y)$. That is, $(x, y)$ is interpreted as moving a disk from pole $x$ to pole $y$.

- The function \texttt{tower} takes three integer arguments $k$, $x$, and $y$ such that $k \geq 0$, $1 \leq x \leq 3$ and $1 \leq y \leq 3$. It returns the moves that transfer a tower of $k$ discs from pole $x$ to pole $y$.

- Recursion permits us to solve the problem naturally.

- The actions to do are the following:
  
  (a) move $k - 1$ disks from $x$ to the “auxiliary” pole $z$,
  
  (b) move the largest disk from $x$ to $y$, and
  
  (c) move $k - 1$ disks from $z$ to $y$.

- Here are some simple sequences of moves:
  
  For 3 poles: $(1,2),(1,3),(2,3)$
  
  For 4 poles: $(1,3),(1,2),(3,2),(1,3),(2,1),(2,3),(1,3)$

Other examples

- Binary Search
- Trees – Evaluation of an expression represented by a tree.
- Parsing – Recursive Descent Parser for EBNF
- ...

Recursion versus iteration
Tail recursion

- Tail recursion is interesting because it is a form of recursion that can be implemented much more efficiently than general recursion.
- Tail recursion is the special case of recursion that is semantically equivalent to the iteration constructs normally used to represent repetition in programs. Because tail recursion is equivalent to iteration, tail-recursive programs can be compiled as efficiently as iterative programs.
- Every iterative algorithm can be implemented using only tail recursion. Any tail recursive algorithm can be transformed to an iterative algorithm.
- Form of a tail recursive function:
f(x) = if c(x) then a(x) else f(p(x))
  - Factorial is not tail recursive.
  - gcd (see previous slide) is tail recursive.

From tail recursion to iteration: (refactoring)
y=x;
while (not c(y))
{
y = p(y)
}
return a(y)

From recursion to tail recursion: Not always possible. If possible use an accumulator.

Example: Factorial
fact(n) = if n=0 then 1 else n*fact(n-1) is not tail recursive.
Tail recursive version:
tailfact(n,x) = if n=0 then x else tailfact(n-1,x*n);
Iterative version:
y=1;
while (n > 0)
{
y=n*y;
n=n-1;
}
return y;

Recursion — Summary

- Recursion is a general method for the definition of functions (and also a powerful technique for designing algorithms).
- Recursive definitions generally specify only partial functions (intuitively functions not defined everywhere).
- The evaluation of recursively defined functions is based on calculation by substitution and simplification.
- These two concepts,
  - definition by recursion and
  - evaluation by substitution and simplification,
are the foundation of functional programming languages such as ML (or SML). Recursion does not exist in FORTRAN and COBOL.
- Lots of natural applications.
- Efficiency and comparison with iteration.

- When should you use iteration, and when use recursion? There are (at least) these three factors to consider:
  (1) Iterative functions are typically faster than their recursive counterparts. So, if speed is an issue, you would normally use iteration.
  (2) If the stack limit is too constraining then you will prefer iteration over recursion.
  (3) Some procedures are very naturally programmed recursively, and all but unmanageable iteratively. Here, then, the choice is clear.