**Functional programming**

- **Function evaluation** (not assignment of variables) is the basic concept for a programming paradigm that has been implemented in such **functional programming languages** as ML.

- The language ML ("Meta Language") was originally introduced in the 1970's as part of a theorem proving system, and was intended for describing and implementing proof strategies. Standard ML of New Jersey (SML) is an implementation of ML.

- The basic mode of computation in ML, as in other functional languages, is the use of the **definition** and **application** of functions (explicit and recursive).

- The basic cycle of ML activity has three parts:
  - read input from the user,
  - evaluate it, and
  - print the computed value (or an error message).

---

**Why functional programming matters?**

- The key to understanding the importance of functional programming is to focus on what it adds, rather than what it takes away.

- Software becomes more and more complex. It is important to structure it well.
  Structured software is:
  - easy to write
  - easy to debug
  - easy to reuse

- Modular software is generally accepted to be the key to successful software.
  - Divide-and-conquer
  - The ways in which the original problem can be divided up depends directly on the ways in which solutions can be “glued” together.
  - New “glues” are provided in functional programming (Examples: higher-order functions, lazy evaluation, polymorphism, abstract data type).

---

**Applications**

- Software Prototyping.

- Industrial:
  - AnnoDomini - Year 2000 remediation for Cobol
  - Shop.com Merchant System - an e-commerce database
  - Combinators for financial derivatives

- Theorem provers.

- Natural language processing and speech recognition

- Network toolkits and applications.
### General features

- The functional ascetics forbid themselves facilities which less pious programmers regard as standard.
- No re-assignment.
- No side-effects.
  - When a value is assigned it does not change during the execution of the program ⇒ Property of referential transparency.
  - No global variable or instance of an object.
- No explicit flow of control.
- Higher level than third generation languages.
- Construction of more reliable software ⇒ Correctness.
  
  Proof of the correctness easiest than for imperative programs.

### Plan

- Recursion (sub-chapter)
- Expressions, values and simple types
- Functions (explicit, recursive)
- Types: tuples, lists
- Operations on lists
- Pattern matching
- Higher-order functions
- Mutual-recursion
- Currying
- Scope (let, local)
- Records, arrays, user defined types
- Exceptions

### Defining Functions

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

### Sub-chapter: Recursion

- 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.
Explicit definitions

- An explicit definition of a function \( f \) consists of giving an expression that indicates for each domain element \( x \) how \( f(x) \) is obtained from previously defined functions (including constants) by composition.

- Examples

\[
\begin{align*}
\text{zero}(x) &= 0 \\
\text{add3}(x) &= x + 3 \\
\text{gt}(x, y) &= \text{if } x > y \text{ then } 1 \text{ else } 0 \\
\lambda x(x) &= \text{if } x \in A \text{ then } 1 \text{ else } 0
\end{align*}
\]

Note
- The last function is called the characteristic function of the set \( A \).
- If-then-else may be used for case distinctions in function definitions.

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 ordering on the elements we consider.

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:

\[
\text{fact}(n) = \text{if } n = 0 \text{ then } 1 \text{ else } n \times \text{fact}(n - 1).
\]

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.

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:

\[
\begin{align*}
\text{fact}(5) &= 5 \times \text{fact}(5 - 1) \quad \text{(substitution)} \\
&= 5 \times \text{fact}(4) \quad \text{(substitution)} \\
&= 5 \times (4 \times \text{fact}(4 - 1)) \quad \text{(substitution)} \\
&= 20 \times \text{fact}(3) \quad \text{(substitution)} \\
&= 120 \quad \text{(simplification)}
\end{align*}
\]
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:
  \[ \text{square}(x) = \begin{cases} 
    0 & \text{if } x = 0 \\
    \text{square}(x-1) + 2x - 1 & \text{if } x > 0 
  \end{cases} \]

From the recursive definition we get the following function values:
- \( \text{square}(0) = 0 \)
- \( \text{square}(1) = \text{square}(0) + 1 = 1 \)
- \( \text{square}(2) = \text{square}(1) + 3 = 4 \)
- \( \text{square}(3) = \text{square}(2) + 5 = 9 \)
- \( \text{square}(4) = \text{square}(3) + 7 = 16 \)

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

Fibonacci Numbers

- The recursive definition of the following well-known function (Fibonacci function) employs the function values for several smaller arguments:
  \[ \text{fib}(n) = \begin{cases} 
    1 & \text{if } n = 0 \\
    1 & \text{if } n = 1 \\
    \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n > 1 
  \end{cases} \]

- The corresponding function values are called Fibonacci numbers:
  - \( \text{fib}(0) = 0 \)
  - \( \text{fib}(1) = 1 \)
  - \( \text{fib}(2) = \text{fib}(1) + \text{fib}(0) = 2 \)
  - \( \text{fib}(3) = \text{fib}(2) + \text{fib}(1) = 3 \)
  - \( \text{fib}(4) = \text{fib}(3) + \text{fib}(2) = 5 \)
  - \( \text{fib}(5) = \text{fib}(4) + \text{fib}(3) = 8 \ldots \)

- 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, \( \phi \approx \frac{\text{fib}(n)}{\text{fib}(n-1)} = (1 + \sqrt{5})/2 \approx 1.618 \)
  - The angles between leaves 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,
  \[ F(x) = \begin{cases} 
    0 & \text{if } x = 0 \\
    F(x+1) + 1 & \text{otherwise} 
  \end{cases} \]

and corresponding attempts at computing function values,
- \( F(0) = 0 \)
- \( F(1) = F(2) + 1 = F(3) + 2 = F(4) + 3 \ldots \)

This function is defined for one argument only. So \( F \) is not well-defined.
- What about the function \( G \), defined for positive integers by
  \[ G(n) = \begin{cases} 
    0 & \text{if } n = 1 \\
    1 + G(n/2) & \text{if } n \text{ is even} \\
    G(3n-1) & \text{if } n \text{ is odd and } n > 1 
  \end{cases} \]
Study of G

• G is not well-defined for all arguments. We have

\[
\begin{align*}
G(1) &= 0 \\
G(2) &= 1 + G(1) = 1 \\
G(3) &= G(8) = 1 + G(4) = 1 + (1 + G(2)) = 3 \\
G(4) &= 1 + G(2) = 2 \\
G(5) &= G(14) = 1 + G(7) = 1 + G(20) = 1 + (1 + G(10)) = 3 + G(5)
\end{align*}
\]

Thus, if G(5) was defined, we could infer the contradictory statement that 0 = 3! In other words, G(5) must be undefined.

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,

\[
\begin{align*}
M(99) &= M(M(110)) & (\text{since } 99 < 100) \\
&= M(100) & (\text{since } 110 > 100) \\
&= M(M(111)) & (\text{since } 100 < 100) \\
&= M(101) & (\text{since } 111 > 100) \\
&= 91 & (\text{since } 101 > 100)
\end{align*}
\]

• 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.

A new function \( H \)

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

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

defines a well-defined function on all positive integers.

- \( H(2) = 1 \)
- \( H(10) = H(5) = H(16) = H(8) = H(4) = H(2) = H(1) \)
- \( H(21) = H(64) = H(32) = H(16) = H(8) = H(4) = H(2) = H(1) \)

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

\( H(2) = 1 \)
\( H(17) = 9 \)
\( H(21) = 6 \)

and \( H(35) = 10. \)
Different evaluations of a recursive function

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

- 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)

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 function for specific arguments 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.

First SML example

- Not the “Hello World!” program!
- Here is a simple example:
  ```sml
  - 3;
  val it = 3 : int
  ```
- The first line contains the SML prompt, followed by an expression typed in by the user and ended by a semicolon.
- The second line is SML’s response, indicating the value of the input expression and its type.
Interacting with SML

- SML has a number of **built-in** operators and **data types**.
- SML provides the standard arithmetic operators.
  - `3+2;`
  - `val it = 5 : int`
  - `sqrt(2.0);`
  - `val it = 1.4142135623709 : real`
- The Boolean values `true` and `false` are available, as are logical operators such as `not` (negation), `andalso` (conjunction), and `orelse` (disjunction).
  - `not(true);`
  - `val it = false : bool`
  - `true andalso false;`
  - `val it = false : bool`

Binding Names to Values

- In SML one can associate identifiers with values,
  - `val three = 3;`
  - `val three = 3 : int`
  - and thereby establish a new **value binding**,
  - `three;`
  - `val it = 3 : int`
- More complex expressions can also be used to bind values to names,
  - `val five = 3+2;`
  - `val five = 5 : int`
- Names can then be used in other expressions,
  - `three + five;`
  - `val it = 8 : int`

Types in SML

- SML is a **strongly typed** language in that all (well-formed) expressions have a **type** that can be determined by examining the expression.
- As part of the evaluation process, SML determines the type of the output value.
  - → **Inference of type**
- Simple types are:
  - `real`
  - **Examples:** `~ 1.2` and `1.5e12` (1.5 × 10^{12}) are reals.
  - `int`
  - **Examples:** `~ 12` and `14` are integers. `3 + 5` is an integer.
  - `bool`
  - **Examples:** `true` and `not(true)` are booleans.
  - `string`
  - **Examples:** "nine", "" are strings.

Defining Functions in SML is a lot of fun!

- The general form of a function definition in SML is:
  - `fun (identifier) ([parameters]) = (expression);`
- The **type** of a function is expressed using →.
  - It is recursively defined by:
    - `type of the parameters → type of the result`
- **Example:**
  - `fun double(x) = 2*x;`
  - `val double = fn : int → int`
  - declares `double` as a function from integers to integers.
  - The type of the function `double` is: `int → int`.
    - `double(222);`
    - `val it = 444 : int`
  - The type of `double(222)` is `int`. 
If we apply `double` to an argument of the wrong type, we get an error message:

- `double(2.0);`
  Error: operator and operand don't agree [tycon mismatch]
  operator domain: int
  operand: real
  in expression: `double 2.0`

- The user may also explicitly specify types.

  **Example:**
  ```
  - fun max(x:int,y:int,z:int) =
    if ((x>y) andalso (x>z)) then x
    else (if (y>z) then y else z);
  val max = fn : int * int * int -> int
  - max(3,2,2); val it = 3 : int
  
  The type of the function `max` is:
  ```
  ```
  int * int * int -> int.
  ```

---

**Recursive Definitions**

- The use of recursive definitions is a main characteristic of functional programming languages.
- These languages strongly encourage the use of recursion as a structuring mechanism in preference to iterative constructs such as while-loops.

  **Example:**
  ```
  - fun factorial(x) = if x=0 then 1
    else x*factorial(x-1);
  val factorial = fn : int -> int
  
  The type of the function `factorial` is:
  ```
  ```
  int -> int.
  ```

  The definition is used by SML to evaluate applications of the function to specific arguments.
  ```
  - factorial(5);
    val it = 120 : int
  - factorial(10);
    val it = 3628800 : int
  ```

---

**Greatest Common Divisor**

- The calculation of the greatest common divisor (gcd) of two positive integers can also be done recursively based on the following observations:
  1. `gcd(n,n) = n`,
  2. `gcd(m,n) = gcd(n,m)`, and
  3. `gcd(m,n) = gcd(m-n,n)`, if `m > n`.

- A possible definition in SML is as follows:
  ```
  - fun gcd(m,n):int = if m=n then n
    else if m>n then gcd(m-n,n)
    else gcd(m,n-m);
  
  val gcd = fn : int * int -> int
  - gcd(12,30);
    val it = 6 : int
  - gcd(1,20);
    val it = 1 : int
  - gcd(126,2357);
    val it = 1 : int
  - gcd(125,56345);
    val it = 5 : int
  ```

---

**Tuples in SML**

- SML provides two ways of defining data types that represent sequences.
  - **Tuples** are finite sequences, where the length is arbitrary but fixed and the different components need not be of the same type.
  - **Lists** are finite sequences of elements of the same type.

- Some examples of tuples and the corresponding types are:
  ```
  - val t1 = (1,2,3);
    val t1 = (1,2,3) : int * int * int
  - val t2 = (4,5,6);
    val t2 = (4,5,6) : int * (real * int)
  - val t3 = (7,8.0,"nine");
    val t3 = (7,8.0,"nine") : int * real * string
  
  The type of `t1` is `int * int * int`. The type of `t2` is `int * (real * int)`. The type of `t3` is `int * real * string`.
  ```
• The components of a tuple can be accessed by applying the built-in function \#i, where i is a positive number.

- \#1(t1);
val it = 1 : int
- \#1(t2);
val it = 4 : int
- \#2(t2);
val it = (5,0,6) : real * int
- \#3(t3);
val it = 6 : int
- \#1(t3);
val it = "mine" : string

If a function \#i is applied to a tuple with fewer than i components, an error results:

- \#3(t3);
... Error: operator and operand don’t agree

• Another built-in data structure to represent sequences in SML are lists.

• A list in SML is essentially a finite sequence of objects, all of the same type.

• Examples:

- [1,2,3];
val it = [1,2,3] : int list
- [true,false,true];
val it = [true,false,true] : bool list
- [[1,2,3],[4,6],[6]];
val it = [[1,2,3],[4,6],[6]] : int list list

The last example is a list of lists of integers, in SML notation int list list.

• All objects in a list must be of the same type:

- [1,[2]];
Error: operator and operand don’t agree

• The empty list is denoted by the following symbols:

- [];
val it = [] : ’a list - nil; val it = [] : ’a list

Operations on Lists

• SML provides some predefined functions for manipulating lists.

• The function \#d returns the first element of its argument list.

- \#d[1,2,3];
val it = 1 : int
- \#d[[1,2],[3]];
val it = [1,2] : int list

Applying this function to the empty list will result in an exception (error).

• The function \#l removes the first element of its argument lists, and returns the remaining list.

- \#l[1,2,3];
val it = [2,3] : int list
- \#l[[1,2],[3]];
val it = [[3]] : int list list

The application of this function to the empty list will also result in an error.
• The **types** of the two functions are as follows:
  - hd;
    val it = fn : 'a list -> 'a
  - tl;
    val it = fn : 'a list -> 'a list

---

**More List Operations**

- Lists can be constructed by the (binary) function `::` (read `cons`) that adds its first argument to the front of the second argument.
  - 5::[];
    val it = [5] : int list
  - 1::[2,3];
    val it = [1,2,3] : int list
  - 1,2::[[3],[4,5,6,7]];
    val it = [[[1,2],[3],[4,5,6,7]]] : int list

Again, the arguments must be of the right type:
- [1]::[2,3];
  Error: operator and operand don't agree

- Lists can also be compared for equality:
  - [1,2,3]=[1,2,3];
    val it = true : bool
  - [1,2]=[2,1];
    val it = false : bool
  - tl[1] = [];
    val it = true : bool

---

**Defining List Functions**

- **Recursion** is particularly useful for defining list processing functions.

- For example, consider the problem of defining an SML function, call it **concat**, that takes as arguments two lists of the same type and returns the concatenated list.

- **What is the SML type of **concat**?**

- For example, the following applications of the function **concat** should yield the indicated responses.
  - `concat([1,2],[3]);`
    val it = [1,2,3] : int list
  - `concat([],[1,2]);`
    val it = [1,2] : int list
  - `concat([1,2],[]);`
    val it = [1,2] : int list

- In defining such list processing functions, it is helpful to keep in mind that a list is either
  - the empty list `[]` or
  - of the form `x::y`.

  The empty list and `::` are the constructors of the type list.

  For example,
  - `[1,2,3]=1::[2,3];`
    val it = true : bool
In **designing** a function for concatenating two lists \( x \) and \( y \) we thus distinguish two cases, depending on the form of \( x \):

- If \( x \) is an empty list, then concatenating \( x \) with \( y \) yields just \( y \).
- If \( x \) is of the form \( x_1 : x_2 \), then concatenating \( x \) with \( y \) is a list of the form \( x_1 : z \) where \( z \) is the results of concatenating \( x_2 \) with \( y \). In fact we can even be more specific by observing that \( x = \text{hd}(x) : \text{tl}(x) \).

This suggests the following recursive definition.

- \( \text{fun concat}(x,y) = \text{if } x=[] \text{ then } y \)
- \( \text{else } \text{hd}(x) : \text{concat}(\text{tl}(x),y) \);

val \( \text{concat} = \text{fn} : \ 'a \text{ list} \times \ 'a \text{ list} \rightarrow \ 'a \text{ list} \)

This seems to work (at least on some examples):

- \( \text{concat}([1,2],[3,4,5]) \);
val \( \text{it} = [1,2,3,4,5] : \text{int list} \)
- \( \text{concat}([], [1,2]) \);
val \( \text{it} = [1,2] : \text{int list} \)
- \( \text{concat}([1,2], []) \);
val \( \text{it} = [1,2] : \text{int list} \)

### More List Processing Functions

**Recursion** often yields simple and natural definitions of functions on lists.

The following function computes the length of its argument list by distinguishing between:

- the empty list (the basis case) and
- non-empty lists (the general case).

- \( \text{fun length}(L) = \)
- \( \text{if } L=\text{nil} \text{ then } 0 \)
- \( \text{else } 1+\text{length}(\text{tl}(L)) \);

val \( \text{length} = \text{fn} : \ 'a \text{ list} \rightarrow \text{int} \)

- \( \text{length}[1,2,3] \);
val \( \text{it} = 3 : \text{int} \)
- \( \text{length}[[5],[4],[3],[2,1]] \);
val \( \text{it} = 4 : \text{int} \)
- \( \text{length[]} \);
val \( \text{it} = 0 : \text{int} \)

The following function has a similar recursive structure. It **doubles** all the elements in its argument list (of integers).

- \( \text{fun doubleall}(L) = \)
- \( \text{if } L=[] \text{ then } [] \)
- \( \text{else } 2\times\text{hd}(L) : \text{doubleall}(\text{tl}(L)) \);

val \( \text{doubleall} = \text{fn} : \text{int list} \rightarrow \text{int list} \)

- \( \text{doubleall}[1,3,5,7] \);
val \( \text{it} = [2,6,10,14] : \text{int list} \)

\text{doubleall} is of type: \text{int list} \rightarrow \text{int list}. Why?
The Reverse of a List

- **Concatenation** of lists, for which we gave a recursive definition, is actually a built-in operator in SML, denoted by the symbol @.

- We use this operator in the following recursive definition of a function that produces the reverse of a list.

  ```ml
  fun reverse(L) = 
    if L = nil then nil 
    else reverse(tl(L)) @ [hd(L)];
  
  val reverse = fn : 'a list -> 'a list
  - reverse [1,2,3];
  val it = [3,2,1] : int list
  - reverse nil;
  stdIn:35.1-35.12 Warning: type vars not generalized
  because of value restriction are instantiated
  to dummy types (X1,X2,...)
  val it = [] : 'a list
  ```

Function Definition by Patterns

- In SML there is an alternative form of defining functions via **patterns**.

- The general form of such definitions is:

  ```ml
  fun <identifier>(<pattern1>) = <expression1> 
  | <identifier>(<pattern2>) = <expression2> 
  | ... 
  | <identifier>(<patternK>) = <expressionK>;
  
  where the identifiers, which name the function, are all the same, all patterns are of the same type, and all expressions are of the same type.

- For example, an alternative definition of the reverse function is:

  ```ml
  fun reverse(nil) = nil
  | reverse(x::xs) = reverse(xs) @ [x];
  
  val reverse = fn : 'a list -> 'a list
  ```

- In applying such a function to specific arguments, the patterns are inspected in order and the first match determines the value of the function.

Pattern Matching

- We informally use pattern matching all the time in real life.

- Informally, a **pattern** is an expression containing **variables**, for which other expressions may be substituted. The problem of matching a pattern against a given expression consists of finding a suitable substitution that makes the pattern identical to the desired expression, if one exists at all.

- For example, we may apply the commutativity of +,

  ```ml
  x + y = y + x
  ```

to the formula

  ```ml
  F = 1 + 2 + 3
  ```

to obtain an equivalent formula

  ```ml
  3 + 2 + 1
  ```

Here the "meta-variables" x and y were replaced by numbers. How?

Removing Elements from Lists

- The following function removes all occurrences of its first argument from its second argument list.

  ```ml
  fun remove(x,L) = 
    if (L = []) then [] 
    else if (x=hd(L)) 
      then remove(x,tl(L)) 
      else hd(L)::remove(x,tl(L));
  
  val remove = fn : 'a list -> 'a list
  - remove(1,[5,3,1]);
  val it = [5,3] : int list
  - remove(2,[4,2,4,2,4,2,2]);
  val it = [4,4,4] : int list
  - remove(2,nil); val it = [] : int list
  ```

- We use it as an auxiliary function in the definition of another function that removes all duplicate occurrences of elements from its argument list.

  ```ml
  fun removedupl(L) = 
    if (L = []) then [] 
    else hd(L)::remove(hd(L),removedupl(tl(L)));
  
  val removedupl = fn : 'a list -> 'a list
  ```
**Constructing Sublists**

- A sublist of a list $L$ is any list obtained by deleting some (i.e., zero or more) elements from $L$.
- For example, [], [1], [2], and [1,2] are all the sublists of [1,2].
- Let us design an SML function that constructs all sublists of a given list $L$. The definition will be recursive, based on a case distinction as to whether $L$ is the empty list or not.
- If $L$ is non-empty, it has a first element $x$. There are two kinds of sublists: those containing $x$, and those not containing $x$.
- For instance, in the above example we have sublists [1] and [1,2] on the one hand, and [] and [2] on the other hand.
- Note that there is a one-to-one correspondence between the two kinds of sublists, and that each sublist of the latter kind is also a sublist of $t1(L)$.

If we change the expression in the else-branch to

```sml
  = else insertL(hd(L), sublists(tl(L)))
  =  @ sublists(tl(L))
```

all sublists will still be generated, but in a different order.

**Constructing Sublists (continued)**

- These observations lead to the following definition.
  
  ```sml
  fun sublists(L) =
    if (L=[]) then []
    else sublists(tl(L))
    =  @ insertL(hd(L), sublists(tl(L)));
  
  val sublists = fn : "a list -> "a list list
  
  sublists[];
  stdIn:84.1-84.11 Warning: type vars not generalized because of value restriction are instantiated to dummy types (X1,X2,...)
  val it = [[],[],[1],[1,2],[2],[2,1],[1,3],[1,2,3],[1,2,3],[3],[2,3],[2,3,1],[3,1],[3,2],[3,2,1],[4],[4,1],...]
  
  Here @ denotes the (built-in) concatenation operation on lists, and the function insertL inserts its first argument at the front of all elements in its second argument (which must be a list). Its definition is left as an exercise.
```

**Higher-Order Functions**

- In functional programming languages, parameters may denote functions and be used in definitions of other, so-called higher-order, functions.
- One example of a higher-order function is the function apply defined below, which applies its first argument (a function) to all elements in its second argument (a list of suitable type).
  
  ```sml
  fun apply(f,L) =
    if (L=[]) then []
    else f(hd(L))::(apply(f,tl(L)));
  
  val apply = fn : ("a => 'b') * "a list -> 'b list
  
  We may apply apply with any function as argument.
  
  val square = fn : int -> int
  
  fun square(x) = (x:int)*x;
  val it = [4,9,16] : int list
  ```
- The function `doubleall` we defined may be considered a special case of supplying `apply` with first argument `double` (a function we defined in a previous lecture).

  ```sml
  apply(double,[1,3,5,7]);
  val it = [2,6,10,14] : int list
  ```

- `apply` is predefined in SML and is called `map`.

- Consider two functions, `take` and `skip`, both of which extract alternate elements from a given list, with the difference that `take` starts with the first element (and hence extracts all elements at odd-numbered positions), whereas `skip` skips the first element (and hence extracts all elements at even-numbered positions, if any).

  ```sml
  fun take(L) = 
      if L = nil then nil
      else hd(L)::skip(tl(L))
  and
  = skip(L) = 
      if L=nil then nil
      else take(tl(L))
  val take = fn : 'a list -> 'a list
  val skip = fn : 'a list -> 'a list
  ```

  ```sml
  - take[1,2,3];
  val it = [1,3] : int list
  - skip[1,2,3];
  val it = [2] : int list
  ```

- Sometimes the most convenient way of defining (two or more different) functions is in mutual dependence of each other.

- Consider the functions, `even` and `odd` that test if a number is even and odd. We can define them in the following way.

  ```sml
  fun even(0) = true
      | even(m) = odd(n-1)
      = and
  = odd(0) = false
      | odd(n) = even(n-1);
  val even = fn : int -> bool
  val odd = fn : int -> bool
  ```

  SML uses the keyword `and` (not to be confused with the logical operator `andalso`) for such mutually recursive definitions.

  Neither of the two definition is acceptable by itself.

  ```sml
  - even(2);
  val it = true : bool
  - odd(3);
  val it = true : bool
  ```

---

### Mutual Recursion

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

- More precisely, we want to define an SML function,

  ```sml
  sort : list int -> list int
  ```

  such that `sort(L)` is a sorted version (in non-descending order) of `L`.

- Sorting is an important problem for which a large variety of different algorithms have been proposed.

- 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.
Merging

- First we consider the problem of merging two sorted lists.
- A corresponding recursive definition can be easily defined by distinguishing between the different cases, as to whether one of the argument lists is empty or not.
- The following SML definition is formulated in terms of patterns (against which specific arguments in applications of the function will be matched during evaluation).

```ml
- fun merge([],M) = M
  | merge(L,[],) = L
  | merge(x::xl,y::yl) = if (x<int)<y then x::merge(xl,y::yl)
  | else y::merge(x::xl,y);
val merge = fn : int list * int list -> int list
- merge([1,5,7,9],[2,3,5,5,10]);
val it = [1,2,3,5,5,7,9,10] : int list
- merge([],[],[]);
val it = [1,2] : int list
- merge([1,2],[],[]);
val it = [1,2] : int list
```

- How do we split a list? Recursion seems to be of little help for this task, but fortunately we have already defined suitable functions that solve the problem.

Finally, some examples:

- `sort([]);
  val it = [] : int list`
- `sort([1]);
  val it = [1] : int list`
- `sort([1,2]);
  val it = [1,2] : int list`
- `sort([2,1]);
  val it = [1,2] : int list`
- `sort([1,2,3,4,5,6,7,8,9]);
  val it = [1,2,3,4,5,6,7,8,9] : int list`
- `sort([9,8,7,6,5,4,3,2,1]);
  val it = [1,2,3,4,5,6,7,8,9] : int list`
- `sort([1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2]);
  val it = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1] : int list`

Merge Sort

- Using `take` and `skip` to split a list, we obtain the following function for sorting.

```ml
- fun sort(L) = 
  | if L=[] then []
  | else merge(sort(take(L)),sort(skip(L)));
val sort = fn : int list -> int list
```

Don’t run this function, though, as it doesn’t quite work. Why?

- To see where the problem is, observe what the result is of applying `take` to a one-element list.

```ml
- take[1];
val it = [1] : int list
```

Thus in this case, the first recursive call to `sort` will be applied to the same argument!

- Here is a modified version in which one-element lists are dealt with correctly.

```ml
- fun sort(L) = 
  | if L=[] then []
  | else if tl(L)=[] then L
  | else merge(sort(take(L)),sort(skip(L)));
val sort = fn : int list -> int list
```

Tracing Mergesort

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

```
0 1 2 3 4 5 6
```

- 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.
### Tracing Mergesort

**How to split?**

![Diagram of Mergesort](image)

### 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.

![Diagram of Tower Moves](image)

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:

\[
\begin{align*}
M(0) &= 0 \\
M(1) &= 2M(0) + 1 = 1 \\
M(2) &= 2M(1) + 1 = 3 \\
M(3) &= 2M(2) + 1 = 7 \\
M(4) &= 2M(3) + 1 = 15 \\
M(5) &= 2M(4) + 1 = 31 \\
M(6) &= 2M(5) + 1 = 63 \\
&\vdots
\end{align*}
\]

- 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 \( k \geq 0 \). 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. For that purpose we will write an SML function.
Towers of Hanoi in SML

* Poles are represented by the numbers 1, 2, and 3.

* 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 pair \((x, y)\) is an example of a tuple of length 2 and type \(\text{int} \times \text{int}\).

* The function \(\text{ttower}\) takes three integer arguments \(k, x,\) and \(y\) such that \(k > 0, 1 \leq x \leq 3\) and \(1 \leq y \leq 3\). It returns a list of moves that transfer a tower of \(k\) discs from pole \(x\) to pole \(y\).

  The result returned by \(\text{ttower}\) is of type \((\text{int} \times \text{int})\) list.

* The function \(\text{comp}\), if provided with two of the numbers 1, 2, or 3 as arguments (2 poles), returns the third (pole).

  ```sml
  - fun comp(x,y) = 6-(x+y);
  val comp = fn : int * int -> int
  - comp(3,1);
  val it = 2 : int
  ```

* The function \(\text{ttower}\) is defined by:

  ```sml
  - fun ttower(k,x,y) =
    = if (k=0 or x=y) then []
    = else if k=1 then [(x,y)]
    = else ttower(k-1,x,comp(x,y))
    = @ ((x,y):ttower(k-1,comp(x,y),y));
  val ttower = fn : int * int * int
    . -> (int * int) list
  ```

  The second line indicates that no move is needed if \(k = 0\) or the tower is to remain at the same pole.

  The third line provides an explicit solution for moving a tower of one disk.

  The fourth and fifth line show that in the general case we can

  (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,

  ```sml
  - ttower(1,1,3);
  val it = [(1,3)] : (int * int) list
  
  - ttower(2,2,2);
  val it = [] : (int * int) list
  ```

  and a few longer ones,

  ```sml
  - ttower(2,1,3);
  val it = [(1,2),(1,3),(2,3)] : (int * int) list
  
  - ttower(3,1,3);
  val it = [(1,3),(1,2),(3,2),(1,3),(2,1),(2,3),(1,3)]
    : (int * int) list
  
  - ttower(4,1,3);
  val it = [(1,2),(1,3),(2,3),(1,2),(3,1),
    (3,2),(1,2),(1,3),(2,3),(2,1),
    (3,1),(2,3),(1,2),(1,3),(2,3)]
    : (int * int) list
  ```