**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.
- Programs are collections of function definitions.
- The basic mode of computation is the use of the definition and application of functions (explicit and recursive).
- The basic cycle of activity has three parts:
  - read input from the user,
  - evaluate it, and
  - print the computed value (or an error message).

---

**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.
- Recursion is the only method of repetition.
- Pattern matching.
- Strongly typed and type inference.
- Rule-based programming.
- The focus is more on what is to be computed, not how it should be computed.
  - No allocation of memory.
- Very high level languages.

---

**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, polymorphism, abstract data type).
The code is shorter, clearer, easy of understanding and there are no side-effects.

Functional programming plays an important role in symbolic processing.

Functional programming plays an important role in prototyping.
- Much of a software product's life is spent in specification, design and maintenance, and not in programming. Functional languages are superb for writing specifications which can actually be executed (and hence tested and debugged). Such a specification then is the first prototype of the final program.

Construction of more reliable software ⇒ Correctness.
Proof of the correctness easier than for imperative programs.

---

**Functional programming languages**

- SML
- CAML
- Erlang
- Haskell
- Miranda
- Scheme
- Logo
- Mathematica
- Matlab ...

---

**A comparison of function implementations in several FP languages**

<table>
<thead>
<tr>
<th>Language</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (Mathematics)</td>
<td>$f(x, y) = x^2 + y + 1.5$</td>
</tr>
</tbody>
</table>
| SML | ```
fun f(x, y) = x*x+y+1.5;
``` |
| Scheme | ```
(define f
  (lambda (x y)
    (+ (* x x) y 1.5))
)```
| Logo | ```
to f :x :y
output :x*x+x+y+1.5
end```
| Mathematica | ```
f[x_,y_]:=x*x+y+1.5```
| Matlab | ```
function o = f(x,y)
o=x*x+y+1.5;
```

---

**Functional Programming in the real world**


- Industrial:
  - Erlang is a functional programming language used for implementing very large scale, real-time, concurrent, industrial applications.
    - Used by Ericsson
  - AnnoDomini - Year 2000 remediation for Cobol
  - Shop.com Merchant System - an e-commerce database
    - Combinators for financial derivatives

- Theorem provers (HOL, Isabelle).

- Numerical applications.

- XML Stylesheet transformations.

- Natural language processing and speech recognition

- Network toolkits and applications.
QuickSort in Haskell

```haskell
qsort [] = []
qsort (x:xs) = qsort elts_lt x ++ [x] ++ qsort elts_greq x
  where
    elts_lt x = [y | y <- xs, y < x]
    elts_greq x = [y | y <- xs, y >= x]
```

QuickSort in C

```c
qsort( a, lo, hi ) int a[], hi, lo;
{
  int h, l, p, t;
  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];
    do
      while ((l < h) && (a[l] <= p))
        l = l+1;
      while ((h > l) && (a[h] >= p))
        h = h-1;
      if (l < h)
        t = a[l];
        a[l] = a[h];
        a[h] = t;
    while (l < h);
  }
  qsort( a, lo, l-1);
  qsort( a, l+1, hi);
}
```

Is functional programming hard to learn?

Functional programming does require a change in perspective, which some programmers find hard. But Ericsson's experience in training programmers in Erlang is that most find the transition easy - provided they take the training seriously rather than assuming that they can "pick it up on the day".

(http://www.haskell.org/aboutHaskell.html)
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.

- There are extensions of ML: Concurrent ML, Objective ML...
- http://cm.bell-labs.com/cm/cs/what/smlnj/

### Outline

- Expressions, values and simple types
- Scope (let, local)
- Functions (explicit, recursive)
- Evaluation of functions
- Types: tuples, lists
- Operations on lists
- Exceptions
- Pattern matching
- Higher-order functions
- Mutual-recursion
- Equality operators
- Curried functions
- Records, arrays, user defined types

---

### 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 = 1.41421356237309 : 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`

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.

  \[ \quad \text{Inference of type} \]

- Simple types are:
  - `real`
  - **Examples:** \( \approx 1.2 \) and \( 1.5 \times 10^{12} \) are reals.
  - `int`
  - **Examples:** \( \approx 12 \) and \( 14 \) are integers. \( 3 + 5 \) is an integer.
  - `bool`
  - **Examples:** `true` and `not(true)` are booleans.
  - `string`
  - **Examples:** "nine", "" are strings.

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`

Variables and environment

- A variable is represented by a name and a value.
- The environment is a list of pairs (variable, value).
- A new variable may be added to the environment using:
  - `val <variable> = <value>;`
- Example:
  - Initial environment: \( \rho_0 \)
  - `val x = 1;`
  - `val y = 3;`
  - `val z = y*x+3;`  
  - Environment: \( \rho_1 = \rho_0, (x, 1), (y, 3), (z, 7) \)
  - `val y = 1.2;`
  - Environment: \( \rho_2 = \rho_1, (y, 1.2) \)
Local environment

- It is possible to create local variables inside a function using `let ... in ... end`.

- Syntax:

```
let
val <var1> = <val1>; val <var2> = <val2>; ...
val <varn> = <valn> in <expression>
end
```

- Example:

\[ \rho_1 = \rho_0, (x, 1), (y, 2), (z, 3) \]

\[ \text{let val } u = x + y; \text{ val } v = x * y + z; \text{ val } z = 1 \]

\[ \text{in } u * z + v * z \]

\[ \text{end; val it } = 7 : \text{ int} \]

\[ \text{stdIn:160.1 Error: unbound variable or constructor: u} \]

\[ \rho_2 = \rho_1, (u, 3), (v, 3), (z, 1) \] (before end)

\[ \rho_3 = \rho_1 \] (after end)

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 domain: 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) andals0 (x>z)) then x
  = else (if (y>z) then y else z);
val max = fn : (int * int * int) \to int
- max(3,2,2);
val it = 3 : int
```

The type of the function `max` is:

```
int * int * int \to int.
```

Defining Functions in SML is a lot of fun!

- The general form of a function definition in SML is:

\[ \text{fun (identifier) ((parameters)) = (expression);} \]

- The type of a function is expressed using \( \to \). It is recursively defined by:

\[ \text{type of the parameters } \to \text{ type of the result} \]

- Example:

\[ \text{fun double(x) } = 2 \times x; \]

```
val double = fn : int \to int
```

declares `double` as a function from integers to integers.

The type of the function `double` is: `int \to int`.

\[ \text{fun double222; val it } = 444 : \text{ int} \]

The type of `double222` is `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:

\[ \text{fun factorial(x) } = \text{ if } x = 0 \text{ then 1 } \]

\[ = \text{ else } x \times \text{factorial}(x-1) ; \]

```
val factorial = fn : (int) \to int
```

The type of the function `factorial` is:

\[ \text{int } \to \text{ int} \]

The definition is used by SML to evaluate applications of the function to specific arguments.

\[ \text{fun factorial(x) } = \text{ if } x = 0 \text{ then 1 } \]

\[ = \text{ else } x \times \text{factorial}(x-1) ; \]

```
val factorial = fn : (int) \to int
```

\[ \text{fun factorial222; val it } = 444 : \text{ int} \]

The type of `factorial222` is `int`.

\[ \text{factorial}(5) ; \]

```
val it = 120 : int
```

\[ \text{factorial}(10) ; \]

```
val it = 3628800 : int
```
• 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:

```sml
fun gcd(m,n) = int = if m=n then n
  else if m>n then gcd(m-n,n)
  =
   else gcd(m,n-n);

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:
  
  ```sml
  - val t1 = (1,2,3);
  - val t1 = (1,2,3) : int * int * int
  - val t2 = (4,(5,0,6));
  - val t2 = (4,(5,0,6)) : 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 \(int * real * string\).

### Greatest Common Divisor

- SML parameter passing is **call-by-value**.

- **Eager evaluation** is used in SML.

  Eager evaluation: A term is reduced to a data value (i.e., is normalized) before it is passed as an argument to a function. This has the benefit that if an argument is used twice or more in the function body, we avoid having to normalize it twice or more.

- The opposite of eager evaluation is **lazy evaluation**.

  Lazy evaluation: A term is not normalized unless the value is required. This has the benefit that if a function argument is discarded, we did not perform the normalization in vain, as we would have in eager evaluation.

- **By-need evaluation** is like lazy evaluation, except for a relatively small modification: when an argument’s value is demanded, we normalize it the first time, store the value away somewhere, and if it is demanded again we use the use the normalized term which had being saved instead of normalizing it again.

### Evaluation of functions

- The components of a tuple can be accessed by applying the built-in function \(\#i\), where \(i\) is a positive number.

  ```sml
  - \#i(t1);
  - \#i(t2);
  - \#i(t3);
  ```

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

```sml
- \#4(t3);
... Error: operator and operand don't agree
```
Lists in SML

- 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 **hd** returns the first element of its argument list.
    - `hd[1,2,3];
      val it = 1 : int
    - `hd[[1,2],[3]];`
      val it = [1,2] : int list

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

  - The function **tl** removes the first element of its argument lists, and returns the remaining list.
    - `tl[1,2,3];
      val it = [2,3] : int list
    - `tl[[1,2],[3]];`
      val it = [[3]] : int list list

    The application of this function to the empty list will also result in an error.

- Note that the type is described in terms of a type variable `’a`, as a list of objects of type `’a`. Instantiating the type variable, by types such as `int`, results in (different) empty lists of corresponding types.

  - 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 = [6] : 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 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

Concatenation of Lists

- 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 x1::x2, then concatenating x with y is a list of the form x1::z, where z is the results of concatenating x2 with y. In fact we can even be more specific by observing that x = hd(x)::tl(x).

    - This suggests the following recursive definition.

      - fun concat(x,y) = if x=[] then y = else hd(x)::concat(tl(x),y);
      - val concat = fn : ''a list * ''a list -> ''a list

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

      - concat([1,2],[3,4,5])
      - val it = [1,2,3,4,5] : int list
      - concat([], [1,2])
      - val it = [1,2] : int list
      - concat([1,2], [])
      - val it = [1,2] : int list
The result of: `concat([],[])` is:

Warning: type vars not generalized because of
type restriction are instantiated to dummy types (X1,X2,...)
val it = [] : ? .X1 list

• 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).

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

  val length = fn : 'a list -> int

  - length[1,2,3];
  val it = 3 : int
  - length[[5],[4],[3],[2,1]];
  val it = 4 : int
  - length[];
  val it = 0 : int

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.

  - 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 = [] : ? .X1 list
Removing Elements from Lists

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

```sml
val remove = fn : 'a list -> 'a list
  - fun remove(x,L) = 
    = if (L=[]) then []
    = else if (x=hd(L))
    = then remove(x,tl(L))
    = else hd(L)::remove(x,tl(L));
```

```sml
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] : 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.

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

```sml
val removedupl = fn : 'a list -> 'a list
  - removedupl[];
```

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], [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 tl(L).

- These observations lead to the following definition.

```sml
val sublists = fn : 'a list -> 'a list list
  - sublists[];
```

```
stdIn:04.1-04.11 Warning: type var not generalized
because of value restriction are instantiated
to dummy types (X1,X2,...)
```

```sml
val it = [[],[]] : 'a list list - sublists[1,2];
val it = [[],[2],[1],[1,2]] : int list list
- sublists[1,2,3];
val it = [[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]] :
  int list list
- sublists[4,3,2,1];
val it = [[],[1],[2],[2,1],[3],[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.
Exceptions

- Many functions are partial, they do not produce a value for some of the possible arguments of the function's domain type. It is essential that we be able to catch such errors. This is done by generations and handling of exceptions.

- Examples of predefined exceptions in SML:
  
  ```sml
  5 div 0;
  uncaught exception Div
  
  hd(nil);
  uncaught exception Empty
  ```

- Declaration of an exception
  
  Syntax:
  ```sml
  exception < name of the exception > (of < type >)
  ```

  Examples:
  ```sml
  exception Foo;
  exception Alert of string;
  exception Number of int;
  exception Tuple of int*real;
  ```

- The type of an exception is exn.

- Raising an exception
  
  Syntax:
  ```sml
  raise < name of the exception(< parameters >) >
  ```

  Examples:
  ```sml
  raise Foo
  raise Alert('Stop!')
  raise Number(5)
  raise Tuple(1,5)
  ```

- Handling an exception

  Syntax:
  ```sml
  < expression > handle < match >
  ```

  Examples:
  ```sml
  Design a function g(n, m) that prints "Division by 0" in case m = 0 using exceptions.
  
  exception Division of int;
  exception Division of int
  
  fun f(n, m) =
  if m = 0
  then raise Division(0)
  else n div m;
  val f = fn : int * int -> int
  
  fun g(n, m) = f(n, m) handle
  Division(0) => (print "Division by 0";
  print "n"; print m;
  0);
  val g = fn : int * int -> int
  ```

  ```sml
  g(1,0);
  Division by 0
  val it = 0 : int
  g(2,1);
  val it = 2 : int
  ```