♦ The datatype declaration can be used for:
  • Making a new type
  • Enumerations
  • Alternatives (sum types)
  • Recursive data structures (e.g., lists, trees).

♦ Constructors are used in patterns to deconstruct values and in expressions to construct values.

♦ We can translate SML programs to Java programs (including design) almost mechanically.
Exceptions can carry values:

```haskell
exception BadInt of int
exception RottenPair of string * int list
```

BadInt and RottenPair are *exception constructors*. Like normal constructors they are used in patterns to deconstruct values (exception packages) and in expressions to construct values (with type exn).

Example:

```haskell
snuggle 42 handle BadInt n => n
   | RottenPair(_, xs) => hd xs
   | e => raise e
```
Representing integer expressions

datatype expr =
    Num of int
  | Add of expr * expr
  | Sub of expr * expr
  | Mult of expr * expr

val e1 = Mult(Num 6, Add(Num 5, Num 2))
val e2 = Add(Mult(Num 6, Num 5), Num 2)
We don’t need parentheses

\[
6 \times (5 + 2) \\
(6 \times 5) + 2
\]
Count the number of operators:

```plaintext
fun count e = 
  let fun helper(e1, e2) = 1 + count e1 + count e2 
in  case e of 
  | Num _     => 0 
  | Add arg   => helper arg 
  | Sub arg   => helper arg 
  | Mult arg  => helper arg 
  end 

val count = fn : expr -> int
```
Evaluating expressions

Expressions can be evaluated to their value:

```ml
fun eval e =
  case e of
    Num n => n
  | Add(e1, e2) => eval e1 + eval e2
  | Sub(e1, e2) => eval e1 - eval e2
  | Mult(e1, e2) => eval e1 * eval e2

val eval = fn : expr -> int
```
Simplified expressions and declarations:

\[
\begin{align*}
dcl & ::= \text{val } x = \text{expr} \\
& \quad | \text{fun } f(x_1, \ldots, x_n) = \text{expr} \\
expr & ::= n \\
& \quad | x \\
& \quad | \text{expr} + \text{expr} \\
& \quad | f(\text{expr}_1, \ldots, \text{expr}_n) \\
& \quad | \text{let } dcl \text{ in } \text{expr} \text{ end}
\end{align*}
\]

where \( n \) stands for numerals; and \( x \) and \( f \) for stands for identifiers.
Representing Mini-SML

Translating the grammar is almost routine:

```ocaml
type id = string

datatype expr =
  Num of int |
  Var of id |
  Add of expr * expr |
  Call of id * expr list |
  Let of decl * expr

and decl =
  Val of id * expr |
  Fun of id * id list * expr
```

The types `expr` and `decl` are *mutual recursive*. 
We represent the function declaration:

\[
\text{fun addFive } x = x + 5
\]

with the value:

\[
\text{val addFive = Fun("addFive", ["x"],}
\]

\[
\text{Add(Var "x", Num 5))}
\]

\[
\text{val addFive = Fun("addFive", ["x"],}
\]

\[
\text{Add(Var "x", Num 5)) : decl}
\]
A second look at trains

Recall the types for carriages and trains:

```plaintext
datatype carriage = Passenger of int
  | Goods       of real
infix 5 :-:
datatype train = Engine of string * int
  | :-:         of train * carriage
```

What if we sometimes want to use the following type for carriages:

```plaintext
datatype waggons = Chemical of int * int
  | Milk       of string
  | Cars       of int
```
Parameterised trains

We use a type variable to make a hole in the declaration of train:

```plaintext
datatype 'kind train = Engine of string * int
                   | :-: of 'kind train * 'kind
```
Now we can have different kind of trains:

```scala
val ursula = Engine("Ursula", 105)
val t1 = ursula :+: Passenger 67
val t2 = ursula :+: Milk "skimmed"

val 'a ursula = Engine("Ursula", 105) : 'a train
val t1 = :+: (Engine("Ursula", 105), Passenger 67)
     : carriage train
val t2 = :+: (Engine("Ursula", 105), Milk "skimmed")
     : waggons train
```
We can also parameterise the data carried by the Engine constructor:

```ocaml
datatype ('e, 'k) train =
  Engine of 'e
  | :-: of ('e, 'k) train * 'k

val t = Engine 42 :-: "Biker Jens" :-: "Sonny"
```
Nothing special about lists

The powerful list type is just the following:

```haskell
infix 5 ::

datatype 'a list = nil
| :: of 'a * 'a list
```

(And some magic with the [] syntax)
**Problem:** We often work with (large) collections of elements of the same type. For example, a set of countries. Lists are cool, but they are not efficient if we need to scan the elements often. Instead of list we use *binary search trees.*
First we declare a type for trees:

```haskell
datatype 'a tree = Leaf
    | Node of 'a tree * 'a * 'a tree
```

Node(Node(Node(Leaf, "Homer", Leaf), "Bart", 
    Node(Leaf, "Biker Jens", Leaf)) : string tree
**Invariant:** A binary search tree is a tree where for all nodes, \( \text{Node}(t_1, e, t_2) \), all the elements in \( t_1 \) are smaller than \( e \) and all the elements in \( t_2 \) are greater than \( e \).
Membership

Check if an integer \( k \) is in the binary search tree \( \text{tr} \):

\[
\text{fun member (k, Leaf) = false}
\mid \text{member (k, Node(l, e, r)) =}
\quad \text{case Int.compare(k, e) of}
\quad \quad \text{EQUAL => true}
\quad \mid \text{LESS => member(k, l)}
\quad \mid \text{GREATER => member(k, r)}
\]

\[
\text{val member = fn : int * int tree -> bool}
\]
Insert an integer $k$ into the binary search tree $tr$:

\[
\text{fun insert} \ (k, \ \text{Leaf}) = \ \text{Node}(\text{Leaf}, \ k, \ \text{Leaf}) \\
| \ \text{insert} \ (k, \ tr \ as \ \text{Node}(l, \ e, \ r)) = \ \\
\text{case} \ \text{Int.compare}(k, \ e) \ \text{of} \\
\hspace{1cm} \text{EQUAL} \ => \ tr \\
| \ \text{LESS} \ => \ \text{Node}(\text{insert}(k, \ l), \ e, \ r) \\
| \ \text{GREATER} \ => \ \text{Node}(l, \ e, \ \text{insert}(k, \ r)) \\
\]

\text{val insert} = fn : int * int tree \to int tree
There is nothing to stop silliness like this:

\[\text{member}(42, \text{Node}(\text{Leaf}, 99, \text{Node}(\text{Leaf}, 42, \text{Leaf})))\]

That is, trees can be build freely and there is nothing that enforces the search tree invariant.
abstype stree = Leaf
    | Node of stree * int * stree

with

    fun insert (k, Leaf) = Node(Leaf, k, Leaf)
    | insert (k, tr as Node(l, e, r)) = 

    : 

    fun member (k, Leaf) = false
    | member (k, Node(l, e, r)) = 

    : 

end
The datatype declaration can be used for:

- Recursive types (bulk data).
- Representing abstract syntax trees.
- Parameterised data types.

The abstype declaration is like the datatype declaration except:

- The constructors are hidden.
- The new type is not an equality type.

Binary search trees rocks.
Coffee.
Functions are first class citizens

We do not want to limit the whereabouts of functions. Therefore we allow:

♦ functions to be arguments for other functions,

♦ functions which produces functions as results (thus, we need expressions describing functions),

♦ data structures (such as lists or tuples) with functions as components.

In short: a function is a value.
Call in the substitutes

♦ Our old friend subst:

\[
\text{fun subst (_, _, [])} = [] \\
| \text{subst (n, k, x :: xs)} = \\
\text{(if x = n then k else x) :: subst(n, k, xs)}
\]

♦ Substitute all elements less than 5 with k:

\[
\text{fun substLess5 (_, [])} = [] \\
| \text{substLess5 (k, x :: xs)} = \\
\text{(if x < 5 then k else x) :: substLess5(k, xs)}
\]

♦ More in the same category:

\[
\text{fun substSmall (k, [])} = [] \\
| \text{substSmall (k, x :: xs)} = \\
\text{(if size x < 3 then k else x) :: substSmall(k, xs)}
\]
Take a predicate

Substitute all the elements that satisfy the predicate \( p \) with \( k \):

\[
\text{fun substitute (_, _, []) = []}
\]
\[
| \text{substitute (p, k, x :: xs) =}
\]
\[
(\text{if } p \ x \ \text{then } k \ \text{else } x) :: \text{substitute(p, k, xs)}
\]

\text{val substitute = fn}
\[
: ('a -> bool) * 'a * 'a list -> 'a list
\]
Using substitute

We can use substitute to declare substLess5 and substSmall:

fun less5 n = n < 5

fun substLess5(k, ls) = substitute(less5, k, ls)

fun substSmall(k, ls) =
    let fun isSmall s = size s < 3
    in substitute(isSmall, k, ls)
    end
Expressions denoting functions

♦ An identifier can denote a function:

\[
\text{val } l5 = \text{less5}\\
\text{val } l5 = fn : \text{int} \to \text{bool}
\]

♦ Using let-expressions:

\[
(\text{let } \text{fun add5 } n = n + 5 \text{ in add5 end}) \quad 37\\
42 : \text{int}
\]

♦ Using conditionals:

\[
\text{val } p = \text{if } x+1 < 5 \text{ then less5}\\
\quad \text{else let } \text{fun greater0 } n = n > 0 \text{ in greater0 end}\\
\text{val } p = fn : \text{int} \to \text{bool}
\]
No need to invent new names all the time

♦ Coming up with a new name that we don’t care about is cumbersome:

\[
\text{(let fun add5 n = n + 5 in add5 end)}
\]

♦ We need a function expression:

\[
\text{fn n => n+5}
\]

called a fn-expression
We can now use substitute and *anonymous functions* (fn-expressions) to declare `substLess5` and `substSmall` without any (named) helper functions:

```ocaml
fun substLess5(k, ls) =
    substitute(fn n => n < 5, k, ls)

fun substSmall(k, ls) =
    substitute(fn s => size s < 3, k, ls)
```
Given the datatype declaration:

```haskell
datatype suit = Diamonds | Clubs | Hearts | Spades
```

Substitute all spades with hearts:

```haskell
fun spades2hearts ls =
substitute(fn Spades => true
                | _ => false, Hearts, ls)
```

Example of usage:

```haskell
val ex = spades2hearts[Spades, Hearts, Clubs, Spades]
val ex = [Hearts, Hearts, Clubs, Hearts] : suit list
```
The general form of fn-expressions is:

\[
\text{fn } \textit{pat}_1 \Rightarrow \textit{exp}_1 \mid \cdots \mid \text{pat}_n \Rightarrow \textit{exp}_n
\]

Examples:

\[
\text{fn } n \Rightarrow n + 1
\]
\[
\text{fn } \textit{Diamonds} \Rightarrow 0 \mid \textit{Clubs} \Rightarrow 1 \mid \textit{Hearts} \Rightarrow 2 \mid _\Rightarrow 3
\]

Equivalent to the expression:

\[
\text{let fun } f \ x = \\
\quad \text{case } x \text{ of } \textit{pat}_1 \Rightarrow \textit{exp}_1 \mid \cdots \mid \textit{pat}_n \Rightarrow \textit{exp}_n \\
\quad \text{in } f \ \text{end}
\]

where \( f \) and \( x \) are \textit{fresh}. \]
fun is just shorthand

♦ For non-recursive functions fun is just shorthand for a val declaration and an fn-expression. Example:

```haskell
fun add10 n = n + 10
val add10 = fn n => n + 10
```

♦ For recursive functions we need an extra ingredient

```haskell
fun fact 0 = 1
    | fact n = n * fact(n - 1)
val rec fact = fn 0 => 1
    | n => n * fact(n - 1)
```
♦ Compute the function that for a given \( n \) adds \( n \) to its argument:

\[
\text{fun add n = fn x => x + n}
\]

\[
\text{val add = fn : int -> int -> int}
\]

♦ Examples of usage:

\[
\text{val addFive = add 5}
\]
\[
\text{val t1 = addFive 37}
\]
\[
\text{val t2 = add 20 3}
\]

\[
\text{val addFive = fn : int -> int}
\]
\[
\text{val t1 = 42 : int}
\]
\[
\text{val t2 = 23 : int}
\]
Another shorthand

♦ Instead of:

\[
\text{fun } f \ x = \text{fn } y \Rightarrow \text{exp}
\]

we write:

\[
\text{fun } f \ x \ y = \text{exp}
\]

♦ Examples:

\[
\begin{align*}
\text{fun add } n &= \text{fn } x \Rightarrow x + n \\
\text{fun add } n \ x &= x + n
\end{align*}
\]

\[
\begin{align*}
\text{fun triAdd } x &= \text{fn } y \Rightarrow \text{fn } z \Rightarrow x + y + z \\
\text{fun triAdd } x \ y \ z &= x + y + z
\end{align*}
\]
Substitute does not need to take all the arguments wrapped in a triple:

```plaintext
fun substitute _ _ [] = []
  | substitute p k (x :: xs) =
    (if p x then k else x) :: (substitute p k xs)
```

Now substLess5 and substSmall are one-liners:

```plaintext
val substLess5 = substitute (fn n => n < 5)
val substSmall = substitute (fn s => size s < 3)
```
Application:
List Transformation
Many list functions follow the same code-skeleton:

fun incrList [] = []
    | incrList (x :: xs) = x + 1 :: incrList xs

fun posList [] = []
    | posList (x :: xs) = (x > 0) :: posList xs

fun stepList [] = []
    | stepList (f :: fs) = oneStep f :: stepList fs

fun substitute _ _ [] = []
    | substitute p k (x :: xs) =
        (if p x then k else x) :: (substitute p k xs)
Apply a function \( f \) to each element of a list:

\[
\text{fun map } f \ [\ ] = [\ ] \\
| \text{map } f \ (x :: xs) = f \ x :: \text{map } f \ xs
\]

\[
\text{val map } = \text{fn : ('}a \rightarrow 'b) \rightarrow 'a \text{ list } \rightarrow 'b \text{ list}
\]

Usage:

\[
\text{val incrList } = \text{map } (\text{fn } n \Rightarrow n + 1)
\]

\[
\text{val posList } = \text{map } (\text{fn } i \Rightarrow i > 0)
\]

\[
\text{val stepList } = \text{map } \text{oneStep}
\]

fun substitute \( p \) \( k \) \( ls \) =

\[
\text{map } (\text{fn } x \Rightarrow \text{if } p \ x \text{ then } k \text{ else } x) \ ls
\]
Application:
List Searching
Does it exists?

♦ Check is the is an element in a list satisfying the predicate \( p \):

\[
\text{fun exists } p \ [\] = \text{false} \\
| \text{exists } p \ (x :: xs) = p \ x \ \text{orelse} \ \text{exists } p \ xs
\]

\[
\text{val exists } = \text{fn } : (\ 'a \rightarrow \text{bool} ) \rightarrow \ 'a \ \text{list} \rightarrow \text{bool}
\]

♦ Is there a positive element:

\[
\text{exists } (\text{fn } x \Rightarrow x > 0) \ [\sim3, \sim4, 0, 42, \sim6]
\]

\[
\text{true}
\]

♦ \text{exists} is more general than \text{member}:

\[
\text{fun member } x = \text{exists } (\text{fn } y \Rightarrow x = y)
\]

\[
\text{val member } = \text{fn } : \ 'a \rightarrow \ 'a \ \text{list} \rightarrow \text{bool}
\]
Can we find it?

♦ Find an element that satisfy the predicate \( p \):

\[
\text{fun find } p \ [\] \ = \ \text{NONE} \\
| \ \text{find } p \ (x :: \ xs) \ = \ \text{if } p \ x \ \text{then SOME } x \\
\hspace{1cm} \text{else find } p \ xs
\]

\text{val find} = \text{fn} : (\ 'a \rightarrow \text{bool} \) \rightarrow \ 'a \ \text{list} \rightarrow \ 'a \ \text{option}

♦ Find the data associated with an article code:

\[
\text{fun findArticle } (\text{code}, \ \text{reg}) = \\
\text{case find } (\text{fn}(c, \_, \_) \Rightarrow c = \text{code}) \ \text{reg of} \\
\hspace{1cm} \text{SOME } (_, n, p) \Rightarrow \text{SOME}(n, p) \\
\text{| NONE } \Rightarrow \text{NONE}
\]
Filter out the elements that do not satisfy the predicate \( p \):

\[
\text{fun filter } \ p \ [\ ] \ = \ [\ ] \\
| \text{filter } \ p \ (x : : \ xs) \ = \ \text{if } \ p \ x \ \text{then } x \ : : \ \text{filter } \ p \ xs \\
\text{else } \text{filter } \ p \ xs
\]

\text{val filter = fn : ('a -> bool) -> 'a list -> 'a list}

Only the positive can stay:

\[
\text{filter (fn } x \ => \ x > 0) \ [1, \ ~42, \ 2, \ 3, \ ~2, \ ~5] \\
[1, \ 2, \ 3] : \text{int list}
\]
Application: List Folding
Let’s get all the skeletons out

The basic skeleton for list functions:

```haskell
fun suml [] = 0
  | suml (x :: xs) = x + suml xs

fun multl [] = 1
  | multl (x :: xs) = x * multl xs

fun cat [] = ""
  | cat (x :: xs) = x ^ cat xs

fun length [] = 0
  | length (_ :: xs) = 1 + length xs

fun append ([], ys) = ys
  | append (x :: xs, ys) = x :: append(xs, ys)
```
The list skeleton

♦ The basic skeleton is:
  • For the empty list give the constant $c$.
  • For a list with head $x$ and tail $xs$: recurse on $xs$ and combine the result with $x$.

♦ Examples:
  • $\text{suml}$, here the constant $c$ is 0 and the combination function is $+$.  
  • $\text{cat}$, here the constant $c$ is "" and the combination function is $\wedge$.  

The list skeleton as a higher order function:

```haskell
fun foldr f c [] = c
| foldr f c (x :: xs) = let val res = foldr f c xs
  in  f(x, res)
end

val foldr = fn:('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Usage:

```haskell
val suml = foldr op+ 0
val multl = foldr op* 1
val cat = foldr op^ ""
```

(the op prefix gives an infix identifier nonfix status)
The dreaded value polymorphism

If we try to declare length like this:

```plaintext
val length = foldr (fn (_, res) => 1 + res) 0
```

we get the warning:

```
! Warning: Value polymorphism:
! Free type variable(s) at top level in value
! identifier length
```
The fix

We need to make the right-hand side an value (not an expression that demands computation):

\[
\text{val length} = \\
\quad \text{fn } ls \Rightarrow \text{foldr (fn } (_, \text{ res}) \Rightarrow 1 + \text{ res}) 0 \text{ ls}
\]

or use a \texttt{fun} declaration:

\[
\text{fun length } ls = \text{foldr (fn } (_, \text{ res}) \Rightarrow 1 + \text{ res}) 0 \text{ ls}
\]
Some polymorphic functions

- We can express append using foldr:

  ```haskell```
  ```
  fun append(xs, ys) = foldr op:: ys xs
  ```

- We can even express map:

  ```haskell```
  ```
  fun map f = foldr (fn (x, res) => f x :: res) []
  ```
Why is it called fold-right?

♦ Let us do some partial hand evaluation:

\[
\text{foldr \ op+ \ 0 \ [1,2,3,4]} \\
\sim 1 + (2 + (3 + (4 + 0)))
\]

♦ Hmm, but isn’t that the same as

\[
4 + (3 + (2 + (1 + 0)))
\]

♦ This correspond to the following definition of \text{suml}:

\[
\text{fun suml \ ls =} \\
\quad \text{let fun helper([], \ acc) \ = \ acc} \\
\quad \quad | \ \text{helper(x::xs, \ acc) \ = \ helper(xs, \ x + acc)} \\
\quad \text{in \ helper(ls, \ 0)} \\
\text{end}
\]
Hand evaluation

\[
\text{suml } [1, 2, 3] \\
\text{helper}([1, 2, 3], 0) \\
\text{helper}([2, 3], 1 + 0) \\
\text{helper}([2, 3], 1) \\
\text{helper}([3], 2 + 1) \\
\text{helper}([3], 3) \\
\text{helper}([], 3 + 3) \\
\text{helper}([], 6) \\
\text{6}
\]
Have we found a new skeleton?

Some interesting functions can be expressed using this new skeleton:

```haskell
fun length ls =  
  let fun helper([], acc) = acc  
      | helper(_ :: xs, acc) = helper(xs, 1 + acc)  
  in helper(ls, 0)  
end

fun rev ls =  
  let fun helper([], acc) = acc  
      | helper(x :: xs, acc) = helper(xs, x :: acc)  
  in helper(ls, [])  
end
```
Given a start value $c$ we accumulate the results of applying a function $f$ over the elements from start to end (left to right).

Examples:

- suml, the start value is 0 and the function is $\text{+}$.
- rev, the start value is $[]$ and the function is $::$. 
Our new skeleton as a higher order function:

```ml
fun foldl f c [] = c
  | foldl f c (x :: xs) = foldl f (f(x,c)) xs

val foldl = fn:('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

Usage:

```ml
val suml = foldl op+ 0

fun rev ls = foldl op:: [] ls

fun length ls = foldl (fn(_, acc) => 1+acc) 0 ls
```
List functions

♦ All the higher order list functions we have seen are available in the library List. Examples: List.filter and List.foldl.

♦ The functions map, foldr, and foldl are so useful that they are available in the toplevel without the List prefix.

♦ Check out the List library for other goodies such as List.tabulate.
Trees.
Traversing a tree

♦ Given the datatype declaration:

```plaintext
datatype 'a tree = Leaf
            | Node of 'a tree * 'a * 'a tree
```

♦ We can declare a higher order function for traversing trees:

```plaintext
fun treeFold f c Leaf = c
    | treeFold f c (Node(l, e, r)) = f(treeFold f c l, e, treeFold f c r)

val treeFold = fn : ('b * 'a * 'b -> 'b) -> 'b
                 -> 'a tree -> 'b
```
Count the number of nodes:

```ml
fun count t =
    treeFold (fn(c1, _, c2) => c1 + 1 + c2) 0 t
val count = fn : 'a tree -> int
```

Convert an int tree to a string tree:

```ml
fun makeNode(l, n, r) = Node(l, Int.toString n, r)
val intToStr = treeFold makeNode Leaf
```
Datastructures with functions in them

♦ A pair of list functions:

\[
(\text{map}, \text{fn} \ [\] \Rightarrow 0 \ | \ _ \Rightarrow 1)
\]
\[
(fn, fn) : (('a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list}) \ast ('c \text{ list} \rightarrow \text{int})
\]

♦ Two lists of functions:

\[
\text{val } fs1 = [\text{op}, \text{fn}(x, y) \Rightarrow 3\times x + y, \#1]
\]
\[
\text{val } fs2 = [\text{fact}, \text{fn} \ x \Rightarrow x, \text{add} \ 5]
\]
\[
\text{val } fs1 = [\text{fn}, \text{fn}, \text{fn}] : (\text{int} \ast \text{int} \rightarrow \text{int}) \text{ list}
\]
\[
\text{val } fs2 = [\text{fn}, \text{fn}, \text{fn}] : (\text{int} \rightarrow \text{int}) \text{ list}
\]

♦ What is the result of:

\[
\text{foldr } (\text{fn}(f, \text{res}) \Rightarrow f \ \text{res}) \ 2 \ fs2
\]
Functional vs. Object-oriented Programming.

(Again.)
A shape is an object with the methods:

- `toString` gives a textual representation of the shape.
- `getPos` gives the position of the shape.
- `move` moves the shape.

We represent objects as a record of functions:

```haskell
datatype shape =
  Shape of {toString : unit -> string,
             getPos : unit -> int * int,
             move : int * int -> shape}
```

Circle shape:

val i2s = Int.toString
local
fun cirHelper r (x,y) () =
   "Circle("^i2s r^") at ("^i2s x^", "^i2s y^")"
in
fun newCircle radius (pos as (x, y)) = Shape{toString = cirHelper radius pos,
   getPos = fn() => pos,
   move = fn(a,b) => newCircle radius (x+a, y+b)}
end

val newCircle = fn : int -> int * int -> shape
Rectangle shape:

```haskell
local
fun recHelper (w, h) (x,y) () =
  "Rect("^i2s w^", "^i2s h^"
    "") at ("^i2s x^", "^i2s y^")"
in
fun newRect sides (pos as (x, y)) =
  Shape{toString = recHelper sides pos,
    newPos = fn() => pos,
    move = fn(a,b) => newRect sides (x+a, y+b)}
end

val newRect = fn : int * int -> int * int -> shape
```

Functional Design and Programming, E2004
A small program using shapes

```javascript
val shapes = [newCircle 5 (0,0), newRect (3,42) (1,1),
              newCircle 1 (~1,0)]

val newShapes = map (fn (Shape s) => #move s (1,0)) shapes

val toStrings = map (fn (Shape s) => #toString s ())
val s1 = toStrings shapes
val s2 = toStrings newShapes
```
A first class function is an object with an apply method.

For functions with type \( \text{int} \rightarrow \text{int} \):

```java
interface FirstClassIntToInt {
    public int apply(int x);
}
```

For functions with type \( 'a \rightarrow 'b \):

```java
interface FirstClassObjToObj {
    public Object apply(Object x);
}
```
map in Java

Coding map is not that difficult:

```java
static LinkedList map(FirstClassObjToObj f, List ls) {
    if (ls.isEmpty())
        return new LinkedList();
    else {
        Object x = ls.get(0);
        List xs = ls.subList(1, ls.size());
        LinkedList res = map(f, xs);
        Object r = f.apply(x);
        res.addFirst(r);
        return res;
    }
}
```
Using `map` is a bit cumbersome:

```java
FirstClassObjToObj incr = new FirstClassObjToObj() {
    public Object apply(Object x) {
        int n = ((Integer)x).intValue();
        return new Integer(n + 1);
    }
};

LinkedList res = map(incr, xs);
```
Closures.

Function composition: $f \circ g$.

Type inference for higher order functions.

Static binding.
Functions are first class citizens.

- functions to be arguments for other functions,
- functions which produces functions as results (thus, we need expressions describing functions),
- data structures (such as lists or tuples) with functions as components.

We can use higher order functions to express code skeletons (design patterns).

We can write an *function value* using the \texttt{fn} notation.