FUNCTIONAL PROGRAMMING, PART II

ABSTRACTION WITH FUNCTIONS (PROCEDURES)

Greatest common divisor

- Our next example calculates the greatest common divisor of a and b with Euclid's algorithm.
- The basic thought is that if the remainder of a divided by b is r, then the divisors of a and b are equal to the divisors of b and r.
- The SML-function follows the mathematical definition precisely again.

$$\begin{aligned} \gcd(a, 0) &= a \\ \gcd(a, b) &= \gcd(b, a \bmod b) \end{aligned}$$

```
fun gcd (a, 0) = a
  | gcd (a, b) = gcd(b, a mod b)
```

- The *process* is iterative. The number of steps grows logarithmically.

  More precisely – according to the *Lamé-theorem* – if Euclid's algorithm calculates the greatest common divisor of two numbers in k steps, then the smaller number cannot be less then kth Fibonacci-number. (See SICP, section 1.2.5)

  Let n be the smaller parameter of the algorithm. If k steps are needed for the calculation of the greatest common divisor, then $n \geq F(k) \approx \Phi^k/\sqrt{5}$. So the k steps are really rational to the (Φ based) logarithm of n.

Prime test

- The predicate `prime` tests wheter the number n is prime. The function `findDivisor` searches for the smallest divisor of n starting from 2. n is prime if the smallest divisor is itself.
- The divisors of n should be searched from 2 to $\sqrt{n}$, so the number of steps is $O(\sqrt{n})$.

```
fun prime n =
  let
      infix divides
      fun smallestDivisor n = findDivisor(n, 2)
      and findDivisor (n, testDivisor) =
            if square testDivisor > n
            then n
            else if testDivisor divides n
            then testDivisor
            else findDivisor(n, testDivisor+1)
      and square x = x * x
      and a divides b = b mod a = 0
in
    n = smallestDivisor n
end
```

Exercise

`prime` searches for the smallest divisor of `n` using steps of one difference. Write a faster solution!

Prime test (continued)

- The next SML-predicate tests the primality of a number with *probability method*. The number of the steps is $O(\lg n)$.
- The algorithm is based on Fermat's Little Theorem, which says:

  if n is prime and $0 < a < n$, then a^n is *congruent* to a modulo n, that is $a^n \bmod n = a$.

  - Two numbers are *congruent* to each other modulo n, if they have the same remainder divided by n. The remainder of a divided by n is called the modulo n-based remainder of a, or shortly just a modulo n.
- If n isn't prime, the above relation does not apply to most of the numbers $0 < a < n$.
- So the algorithm of the prime test follows:
  - For a given n let's choose a number $0 < a < n$ randomly: if $a^n \bmod n \neq a$, then n isn't prime. Otherwise there is a great probability of n being prime.
  - Let's choose another number $a < n$ randomly: if $a^n \bmod n = a$, then the probability of n being prime has grown. Choosing further values for a rises the probability of n's primality.

Prime test (continued)

- The auxiliary function `expmod` returns the modulo `m` based remainder of the `exp`th power of the number `base`.

```
(* expmod (base, exp, m) = base pow exp modulo m
*)
fun expmod (_, 0, _) = 1
  | expmod (b, e, m) =
      if even e
      then square(expmod(b, e div 2, m)) mod m
      else b * expmod(b, e-1, m) mod m
and even n = n mod 2 = 0
and square x = x * x;
```

- It's very similar to `exptFast`. The number of steps is proportional to the exponent's logarithm.
- Generating of random numbers is needed. Details from the SML base library:

```
Random.range (min, max) gen = an integral random number in the
                              range [min, max).  Raises Fail if min > max.
Random.newgen () = a random number generator, taking the seed
                              from the system clock.
```

Prime test (continued)

- We load the `Random` library:

```
load "Random";
```

- `fermatTest` generates a pseudo-random number, and does the test once:

```
(* fermatTest n = false if n is not prime, true otherwise *)
fun fermatTest n =
      let fun tryIt a = expmod(a, n, n) = a
      in  tryIt(Random.range (1, n) (Random.newgen()) )
      end
```

- `fastPrime` repeats the test `times` times:

```
(* fastPrime (n, times) = true if n passes the prime test
                          times times
*)
fun fastPrime (n, 0) = true
  | fastPrime (n, t) = fermatTest n andalso fastPrime(n, t-1)
```

- Remark: This test gives a good answer at really high probability, but not for sure. For example 561 passes the test, however it's not prime.

Functions as general calculation methods

- We have already seen that a function (or more generally, a procedure) is an *abstraction* which – independetly from the value of the data passed as parameters – describes complex operations.
- A higher-order function which has a function as parameter, is an *even higher* abstraction, because the operation implemented by it is also independent from some exact operations, not just some exact data.
- So a higher-order function (procedure) expresses some kind of *general computational method*.
- On the next pages we introduce some bigger examples: a general method for finding the *zero* and *fixed points* of a function.

Finding the roots of an equation with half-interval method

- The half-interval method is an efficient way of finding the roots of the equation $f(x) = 0$, where f is a continuous function.
- The well-known core of the algorithm is:
    - If $f(a) < 0 < f(b)$, f has at least one zero-point between a and b.
    - Let $x = (a + b)/2$. If $f(x) > 0$, then f has (at least) one zero-point between a and x, else (if $f(x) < 0$), f has a root between x and b.
    - The search – the iteration – is stopped when the *difference* of two consecutive values becomes less then a pre-defined value.
- Because the difference is halved in every step, the number of necessary steps for findig one root of f is $O(L/T)$, where L is the lenght of the initial interval, and T is the allowed difference.
- The algorithm described above is implemented by the `search` function (see on next page):

```
(* search (f, negPoint, posPoint) = root of f x in the
                    negPoint <= x <= posPoint interval
   PRE: f negPoint <= 0 and f posPoint >= 0
*)
```

Finding the roots of an equation with half-interval method (cont.)

```
fun search (f, negPoint, posPoint) =
      let val midPoint = average(negPoint, posPoint)
      in
          if closeEnough(negPoint, posPoint)
              then midPoint
          else let val testValue = f midPoint
               in
                   if positive(testValue)
                   then search(f, negPoint, midPoint)
                   else if negative(testValue)
                   then search(f, midPoint, posPoint)
                   else midPoint
                end
      end
and average (x, y) = (x+y)/2.0
and closeEnough (x, y) = abs(x-y) < 0.001
and positive x = x >= 0.0
and negative x = x < 0.0
```

Finding the roots of an equation with half-interval method (cont.)

- It is recommended to verify the existence of the preconditions before applying `search` to avoid bad answers from the SML interpreter.

  -
    ```
    - search(Math.sin, 4.0, 2.0)  (* Good solution *);
    > val it = 3.14111328125 : real
    ```

  -
    ```
    - search(Math.sin, 2.0, 4.0)  (* Bad solution *);
    > val it = 2.00048828125 : real
    ```
- The function `halfIntervalMethod` does the verification, and signals the bad initial value of `negPoint` and `posPoint`.

  ```
  (* halfIntervalMethod (f, a, b) = root of f x in the
                                    a <= x <= b interval
  *)
  ```
- Let’s have a look at the principle of the *separation of concerns*: `search` implements the strategy of finding roots, while `halfIntervalMethod` verifies the preconditions.

Finding the roots of an equation with half-interval method (cont.)

-
  ```
  fun halfIntervalMethod(f, a, b) =
        let val aValue = f a; val bValue = f b
        in
            if negative aValue andalso positive bValue
            then search(f, a, b)
            else if negative bValue andalso positive aValue
            then search(f, b, a)
            else print ("Values " ^ makestring a ^ " and " ^
                   makestring b ^ " are not of opposite sign.\n")
        end
  ```
- The function `makestring` (type: *`numtxt`* `-> string`) converts an arbitrary value of numeric (`int`, `real`, `word`, `word8`), `char` and `string` type to `string` type.
- This version of the function is faulty, because all branches of the `if-then-else` conditional expression *must* have the *same return type*, while the return value of `print` doesn't have `int` type.
- The solution is the use of the so-called *sequential expression* of the form `(e; f)`: the interpreter evaluates `e` and `f` in the written order, then it returns the value of `f`.

Finding the roots of an equation with half-interval method (cont.)

```
fun halfIntervalMethod(f, a, b) =
      let val (aValue, bValue) = (f a, f b)
      in
          if negative aValue andalso positive bValue
          then search(f, a, b)
          else if negative bValue andalso positive aValue
          then search(f, b, a)
          else (print ("Values " ^ makestring a ^ " and " ^
                  makestring b ^ " are not of opposite sign.\n");
                0.0)
      end;
```

```
- halfIntervalMethod(Math.sin, 2.0, 4.0);
> val it = 3.14111328125 : real
- halfIntervalMethod(fn x => x*x*x-2.0*x-3.0, 1.0, 2.0);
> val it = 1.89306640625 : real
- halfIntervalMethod(Math.sin, 2.0, 2.5);
Values 2.0 and 2.5 are not of opposite signs
> val it = 0.0 : real
```

Finding the fixed point of a function

- The value x satisfying the $f(x) = x$ equation is the *fixed point* of the function f.
- A fixed point of a function f can be found by recursively applying f, starting from an applicable value:

  $f x, f(f x), f(f(f x)), f(f(f(f x))), \ldots$

  The recursion can be finished when the difference is insignificant between two steps.
- The parameter of the function `fixedPoint` is a pair; which first element is a function (of which the fixed point is needed), and the second element is the first guess of the fixed point.

```
(* fixedPoint (f, firstGuess) = fixpoint of f in the proximity
                                 of firstGuess with tolerance tolerance
*)
```

- We also need the tolerance of the approximation:

```
val tolerance = 0.00001;
```

Finding the fixed point of a function (cont.)

```
fun fixedPoint (f, firstGuess) =
      let
          fun closeEnough (v1, v2) = abs(v1-v2) < tolerance
          fun try guess =
                let
                    val next = f guess
                in
                    if closeEnough(guess, next)
                    then next
                    else try next
                end
      in
          try firstGuess
      end;

load "Math";
fixedPoint(Math.cos, 1.0);
fixedPoint(fn y => Math.sin y + Math.cos y, 1.0);
```

Finding the fixed point of a function (cont.)

- The calculation of a fixed point is similar to the method for calculating the square root (discussed earlier): both are based on the refinement of the approximation until a condition is satisfied.
- Extracting the square root can easily be considered as a calculation of fixed point: if the square root of x is y, then $y * y = x$, which means $y = x/y$. So the fixed point of the function $f y = x/y$ is the square root of x.

  ```
  fun sqrt x = fixedPoint (fn y => x/y, 1.0);
  ```

- Our solution is bad, because it doesn't converge! It can be easily verified:

  Let the first approximation of x's square root be $y1$, the second $y2 = x/y1$, the third $y3 = x/y2 = x/(x/y1) = y1$. It's clear that this process is endless.
- The oscillation can be blocked by *limiting* the value of difference between two approximate values.
- Because of the sound result is always between the approximation y and x/y, we can choose a new approximate value which is closer to y than x/y: the average of y and x/y. So the new approximation will be $(y + x/y)/2$.

  ```
  fun sqrt x = fixedPoint (fn y => (y+x/y)/2.0, 1.0);
  ```

- This commonly useful method is called *average damping*.

Function as return value

- When speaking of functions as abstraction tools we used functions having other functions as parameters.
- Now we introduce such higher-order functions that return *function* (more precisely *function-value*).
- The recently seen *average damping* is so useful that it should be written as a separate function: if the function f is given, the average of $f(x)$ and x has to be calculated.

```
(* averageDamp f = applying to an arbitrary value x of f
                   it calculates the average of x and f x *)
fun averageDamp f = fn x => (x + f x) / 2.0;
```

- It's clear that if applied to only one parameter, `averageDamp` returns a function-value. `averageDamp` is a partially applicable function.
- Example for using `averageDamp`:

```
(averageDamp (fn x => x*x)) 10.0; (* average of 10.0 and 100.0 *)
```

- Because of the precedence of the function-application operator, the outer brackets can be omitted:

```
averageDamp (fn x => x*x) 10.0;
```

Function as return value (cont.)

- The definition of `averageDamp` can be written with (*syntactic sugar*).

```
fun averageDamp f x = (x + f x) / 2.0;
```

- The version of `sqrt` written with `averageDamp` makes the methods *fixed-point calculation*, *average damping* and the *use of the equation* $y = x/y$ explicit.

```
fun sqrt x = fixedPoint(averageDamp (fn y => x/y), 1.0);
sqrt 4.0;
```

- Conclusion: a process can be described by lots of procedures, but the *essence* is much more comprehensible when introducing *properly selected abstractions*.
- Another example for the application of the principles demonstrated above: the cube root of x is the fixed point of $y \mapsto x/y^2$ – with SML-notation `fn y => x/(y*y)`. We already have the solution!

```
fun cubeRoot x  = fixedPoint(averageDamp (fn y => x/y/y), 1.0);
cubeRoot 8.0;
```

Function as return value (cont.): the general Newton-method

- If $x \mapsto g(x)$ is a differentiable function, then the equation $g(x) = 0$ is the fixed point of $x \mapsto f(x)$, where $f(x) = x - g(x)/g'(x)$ and $g'(x)$ is the derivative of g by x.
- The *general Newton-method* is an application of the fixed point method for finding the fixed point of the function f. For numerous g functions and appropriately chosen x values the Newton-method converges fast.
- At first, the function `deriv` should be defined, which (similarly to `averageDamp`) has a function as parameter and it returns function.
- If g is a function and dx is a small number, then the derivative of g is that g' function, which has the following value for an arbitrary number x: $g'(x) = (g(x + dx) - g(x))/dx$.

```
(* deriv g = derivative of g
*)
val dx = 0.00001;
fun deriv g = fn x => (g(x+dx) - g x) / dx;
```

- For example the derivative of the function $x \mapsto x^3$ for $x = 5$ (the exact value is 75):

```
let fun cube x = x*x*x in deriv cube 5.0 end;
```

Function as return value (cont.): the general Newton-method

- With the help of `deriv` the general Newton-method can be defined as a *fixed-point process*:

```
fun newtonTransform g x = x - (g x / deriv g x)
and newtonsMethod g guess = fixedPoint(newtonTransform g, guess)
```

- Example for using `newtonsMethod`:

```
fun sqrt x = newtonsMethod (fn y => y*y-x) 1.0;
sqrt 16.0;
```

- Two general methods were shown for extracting the square root of a number: one was the fixed-point method and the other was the Newton-method.
- Because of the last is based on the fixed-point method, in fact we have seen two applications of the fixed-point method.
- In both cases a fixed point of a transformation on the original function is calculated.
- Even this general method can also be defined as a procedure (function), as we can see on the next slides.

Function as return value (cont.): two ways of applying the fixed-point method

-
  ```
  (* fixedPointOfTransform (g, transform, guess) =
       a fixed point of (transform g) with the initial guess guess
  *)
  fun fixedPointOfTransform (g, transform, guess) =
                                    fixedPoint(transform g, guess)
  ```
- This was the first version of `sqrt` based on finding a fixed point:

  ```
  fun sqrt x = fixedPoint(averageDamp (fn y => x/y), 1.0)
  ```
- After rewriting with the function implementing the general method:

  ```
  fun sqrt x = fixedPointOfTransform (fn y => x/y,
                                      averageDamp, 1.0)
  ```
- This was the second version of `sqrt` using the general Newton-method:

  ```
  fun sqrt x = newtonsMethod (fn y => y*y-x) 1.0;
  ```
- After rewriting with the function implementing the general method:

  ```
  fun sqrt x = fixedPointOfTransform (fn y => y*y-x,
                                      newtonTransform, 1.0)
  ```

ABSTRACTION WITH DATA

Data abstraction: rational numbers

- On the next few lectures we will consider compound data and data abstraction.
- Base of data abstraction: we build our programs working on compound data that
  - the program parts using the data shouldn't suppose anything of the data structure, only the predefined operations should be used,
  - the program parts defining the data should be independent from the program parts using it,
  - the interface between these two parts of the program should consist of *constructors* and *selectors*.
- From the compound data we have met tuples and lists before.
- In our first bigger example we introduce the implementation of the rational numbers and the operations on them.
- A rational number can be represented with a pair, which first member is the *numerator* and the second is the *denominator*.
- The four basic arithmetic operations will be implemented: `addRat`, `subRat`, `mulRat`, `divRat`, and the test for equality: `equRat`.

Data abstraction: rational numbers (cont.)

- Suppose that
  - we have a *constructor operation* which generates the rational number from a numerator `n` and a denominator `d`: `makeRat(n,d)`, andalso
  - we have a *selector operation* which generates the numerator and one which generates the denominator of a rational number `q`: `num q`, `den q`.
- Let's write an SML-program with the well-known operations:

$n_1/d_1 + n_2/d_2 = (n_1d_2 + n_2d_1)/(d_1d_2)$, $n_1/d_1 - n_2/d_2 = (n_1d_2 - n_2d_1)/(d_1d_2)$,
$(n_1/d_1)(n_2/d_2) = (n_1n_2)/(d_1d_2)$, $(n_1/d_1)/(n_2/d_2) = (n_1d_2)/(d_1n_2)$,
$n_1/d_1 = n_2/d_2$ if and only if $n_1d_2 = n_2d_1$.

```
fun addRat(x, y) =
   makeRat(num x * den y + num y * den x, den x * den y)
fun subRat(x, y) =
   makeRat(num x * den y - num y * den x, den x * den y)
fun mulRat(x, y) = makeRat(num x * num y, den x * den y)
fun divRat(x, y) = makeRat(num x * den y, den x * num y)
fun equRat(x, y) = num x * den y = den x * num y
```

Data abstraction: rational numbers (cont.)

- In the SML we have a *constructor operation* for generating a *tuple*: we enumerate the members between round brackets separated by commas, and
- we have a *selector operation* for selecting one element of a *tuple*: `# i`, where `i` is the *positional label* of the ith element starting from 1.
- Examples: `(3, 4); #1(3, 4); #2(3, 4);`
- The members of a *tuple* can also be bound to a name by *pattern matching*: for example `val (n, d) = (3, 4)`.
- The type, the constructor and the selectors of the rational number will be implemented by *weak* abstraction:

```
type rat = int * int;
fun makeRat (n, d) = (n, d) : rat;
fun num (q : rat) = #1 q;
fun den q = #2 (q : rat);
```

- The *weak abstraction* gives name to an object, but *it does not hide* the details of the implementation.

- An output operation is also needed for printing a rational number of the form `n/d`.

```
fun printRat q =
      print(makestring(num q) ^ "/" ^ makestring(den q) ^ "\n");
```

Data abstraction: rational numbers (cont.)

- Now the first version of our program implementing the rational numbers is ready. The full program:

```
type rat = int * int;
fun makeRat (n, d) = (n, d) : rat;
fun num (q : rat) = #1 q;
fun den q = #2 (q : rat);

fun addRat(x, y) =
   makeRat(num x * den y + num y * den x, den x * den y)
fun subRat(x, y) =
   makeRat(num x * den y - num y * den x, den x * den y)
fun mulRat(x, y) = makeRat(num x * num y, den x * den y)
fun divRat(x, y) = makeRat(num x * den y, den x * num y)
fun equRat(x, y) = num x * den y = den x * num y

fun printRat q =
      print(makestring(num q) ^ "/" ^ makestring(den q) ^ "\n");
```

Data abstraction: rational numbers (cont.)

- Some examples for using the program:

```
val oneHalf  = makeRat(1,2);
val oneThird = makeRat(1,3);
val twoThird = makeRat(2,3);

printRat oneHalf;
printRat(addRat(oneHalf, oneThird));
printRat(mulRat(oneHalf, oneThird));
printRat(addRat(oneThird, oneThird));

equRat(addRat(oneThird, oneThird), twoThird);

oneThird = oneThird;
addRat(oneThird, oneThird) = twoThird;
```

Data abstraction: rational numbers (cont.)

- After trying the last example we can observe that our program does not *normalise* the rational numbers, which means they aren’t stored and printed in their simplest form.
- We can help this problem by dividing the numerator and the denominator with their greatest common divisor in the constructor operation:

```
fun makeRat (n, d) =
      let val g = gcd(n, d) in (n div g, d div g) : rat end;
```

  The selector operations aren’t changed.

- The rational numbers are stored in their normalised form, so not only the printing, but the test for equality gives also a correct result:

```
printRat(addRat(oneThird, oneThird));
addRat(oneThird, oneThird) = twoThird;
```

- There was only one location in the program where we had to make changes for the normalisation!

Data abstraction: rational numbers (cont.)

Data abstraction barriers in the rational numbers package

```
         -------------------------------------------------
---------         Programs using rational numbers         ---------
         -------------------------------------------------
               Rational number in the problem space
         -------------------------------------------------
---------       addRat subRat mulRat divRat equRat        ---------
         -------------------------------------------------
           Rational number like numerator and denominator
         -------------------------------------------------
---------    constructor: makeRat; selectors: num, den    ---------
         -------------------------------------------------
                    Rational number as a pair
         -------------------------------------------------
---------      constructor: ( , ) ; selectors:  #1, #2    ---------
         -------------------------------------------------
                 Implementation of pair in SML
```

Data abstraction: rational numbers (cont.)

- Abstraction barriers isolate certain parts of the program from each other.
- Its advantage is that the programs are easier to maintain and to modify, for example changing the representation of the data.
- For example a rational number can be normalised lazily, only when we need its numerator or denominator, instead of when it's created. If lots of rational numbers are generated, but their numerators or denominators are rarely needed, then the lazy solution is more efficient.

```
fun num (q : rat) =
       let val (n, d) = q; val g = gcd(n, d) in n div g end;
fun den (q : rat) =
       let val (n, d) = q; val g = gcd(n, d) in d div g end;
```

- The non-normalising version of `makeRat` will be used, the rest of the program isn't changed.

```
printRat(addRat(oneThird, oneThird));
addRat(oneThird, oneThird) = twoThird;
equRat(addRat(oneThird, oneThird), twoThird) = true;
```

Data abstraction: rational numbers (cont.)

- Speaking of *data* we can’t only say that „data is that the given constructors and selectors implement”.
- It’s obvious that only a certain set of constructors and selectors are applicable for example for implementing the rational numbers.
- In the case of rational numbers the constuctors and selectors must grant the fulfillment of the following conditions (axioms):

```
(* PRE : d > 0 *)
 x = makeRat(n, d);
 n = num x
 d = den x
```

- One abstraction stage lower the pair representation must also fulfill the following conditions:

```
 q = (x, y)
 x = #1 q
 y = #2 q
```

Data abstraction: rational numbers (cont.)

- Every implementation which satisfy these requirements is applicable, like this next example:

```
exception Cons of string;
fun cons (x, y) =
      let fun dispatch 0 = x
            | dispatch 1 = y
            | dispatch _ = raise Cons "argument not 0 or 1"
      in  dispatch
      end;
fun fst z = z 0;
fun snd z = z 1;
```

- Equations describing a property

```
q = cons(n, d)
n = fst q
d = snd q
```

- Let’s notice that object implementing the rational number is a *function*! `fst` and `snd` *send* a message to the object. So this style of programming is called *message passing*.

Data abstraction: rational numbers (cont.)

- Example:

```
val q = cons(1, 2);
fst q = 1; snd q = 2;
```

- Constuctor and selectors implemented by message passing:

```
fun makeRat (n, d) =
      let val g = gcd(n, d) in cons(n div g, d div g) end;
fun num q = fst q;
fun den q = snd q;
```

- Our package implementing rational numbers has a big problem: it uses *weak abstraction*, so it doesn’t hide the details of the implementation; it’s up to the programmer how deep he/she keeps the abstraction barriers. This is the source of bugs.
- The details of the implementation can be hidden from the outer world using *strong abstraction*, with the help of modules. The name of the „implementation” module in SML is `structure`, and the name of the (optional) „interface” module is `signature`.

```
structure name = struct ... end
signature name = sig ... end
```

Data abstraction with modules: rational numbers

```
structure Gcd = struct
  fun gcd (a, 0) = a
    | gcd (a, b) = gcd(b, a mod b) end

structure Rat =
struct
  type rat = int * int;
  fun makeRat (n, d) = let val g = Gcd.gcd(n, d) in (n div g, d div g) : rat end
  fun num (q : rat) = #1 q
  fun den q = #2 (q : rat)
  fun addRat(x, y) = makeRat(num x * den y + num y * den x, den x * den y)
  fun subRat(x, y) = makeRat(num x * den y - num y * den x, den x * den y)
  fun mulRat(x, y) = makeRat(num x * num y, den x * den y)
  fun divRat(x, y) = makeRat(num x * den y, den x * num y)
  fun equRat(x, y) = num x * den y = den x * num y
  fun printRat q = print(makestring(num q) ^ "/" ^ makestring(den q) ^ "\n");
  val one      = makeRat(1,1)
  val zero     = makeRat(0,1)
  val oneHalf  = makeRat(1,2)
  val oneThird = makeRat(1,3)
  val twoThird = makeRat(2,3)
end;
```

The abstracction isn’t strong enough: the details aren’t hidden enough!

Data abstraction with modules: rational numbers (cont.)

This is the real signature of the implemented `Rat` structure:

```
> structure Rat :
  {type rat = int * int,
   val addRat : (int * int) * (int * int) -> int * int,
   val den : int * int -> int,
   val divRat : (int * int) * (int * int) -> int * int,
   val equRat : (int * int) * (int * int) -> bool,
   val makeRat : int * int -> int * int,
   val mulRat : (int * int) * (int * int) -> int * int,
   val num : int * int -> int,
   val one : int * int,
   val oneHalf : int * int,
   val oneThird : int * int,
   val printRat : int * int -> unit,
   val subRat : (int * int) * (int * int) -> int * int,
   val twoThird : int * int,
   val zero : int * int}
```

The `int` type of the two components of the `rat` type can be seen.

Data abstraction with modules: rational numbers (cont.)

The creation of the signature and its binding to the structure *limit* the visibility of the implemented values:

```
signature Rat =
sig
   type rat
   val makeRat : int * int -> rat
   val num : rat -> int
   val den : rat -> int
   val addRat : rat * rat -> rat
   val subRat : rat * rat -> rat
   val mulRat : rat * rat -> rat
   val divRat : rat * rat -> rat
   val equRat : rat * rat -> bool
   val printRat : rat -> unit
   val one : rat
   val oneHalf : rat
   val oneThird : rat
   val twoThird : rat
   val zero : rat
end;
```

Data abstraction with modules: rational numbers (cont.)

```
structure Rat_1 :> Rat = (* this is the so-called opaque signature binding *)
struct
  type rat = int * int;
  fun makeRat (n, d) = let val g = Gcd.gcd(n, d)
                       in
                           (n div g, d div g) : rat
                       end
  fun num (q : rat) = #1 q
  fun den q = #2 (q : rat)

  fun addRat(x, y) = makeRat(num x * den y + num y * den x, den x * den y)
  fun subRat(x, y) = makeRat(num x * den y - num y * den x, den x * den y)
  fun mulRat(x, y) = makeRat(num x * num y, den x * den y)
  fun divRat(x, y) = makeRat(num x * den y, den x * num y)
  fun equRat(x, y) = num x * den y = den x * num y
  fun printRat q = print(makestring(num q) ^ "/" ^ makestring(den q) ^ "\n");

  val one      = makeRat(1,1)
  val zero     = makeRat(0,1)
  val oneHalf  = makeRat(1,2)
  val oneThird = makeRat(1,3)
  val twoThird = makeRat(2,3)
end;
```

Data abstraction with modules: rational numbers (cont.)

This is the real signature of `Rat1` (bound to `Rat` with *opaque signature binding*):

```
> New type names: rat
  structure Rat1 :
  {type rat = rat,
   val addRat : rat * rat -> rat,
   val den : rat -> int,
   val divRat : rat * rat -> rat,
   val equRat : rat * rat -> bool,
   val makeRat : int * int -> rat,
   val mulRat : rat * rat -> rat,
   val num : rat -> int,
   val one : rat,
   val oneHalf : rat,
   val oneThird : rat,
   val printRat : rat -> unit,
   val subRat : rat * rat -> rat,
   val twoThird : rat,
   val zero : rat}
```

Data abstraction with modules: rational numbers (cont.)

- Examples of the use of the structure `Rat`:

```
open Rat_1;
printRat oneHalf;
printRat(addRat(oneHalf, oneThird));
printRat(mulRat(oneHalf, oneThird));
printRat(addRat(oneThird, oneThird));
equRat(addRat(oneThird, oneThird), twoThird);

addRat(oneThird, oneThird) = twoThird;
! addRat(oneThird, oneThird) = twoThird;
! ^^^^^^^^^^^^^^^^^^^^^^^^^^
! Type clash: expression of type
!   rat
! cannot have equality type ''a
```

- Hey! The relation = cannot be used!
- If needed, the MoSML interpreter should be told with the declaration `eqtype` that the equality test of `rat` type values is allowed; which means `rat` is a so-called *equality type*.

Data abstraction with modules: rational numbers (cont.)

```
signature Rat =
sig
   eqtype rat
   val makeRat : int * int -> rat
   val num : rat -> int
   val den : rat -> int
   val addRat : rat * rat -> rat
   val subRat : rat * rat -> rat
   val mulRat : rat * rat -> rat
   val divRat : rat * rat -> rat
   val equRat : rat * rat -> bool
   val printRat : rat -> unit
   val one : rat
   val oneHalf : rat
   val oneThird : rat
   val twoThird : rat
   val zero : rat
end;
```

A version of the structure `Rat` can also be produced in the name of `Rat2` with the `Rat` signature above using equality type:

```
structure Rat2 :> Rat = Rat;
```

```
> signature Rat =
  /\=rat.
    {type rat = rat,
     val makeRat : int * int -> rat,
     val num : rat -> int,
     val den : rat -> int,
     val addRat : rat * rat -> rat,
     val subRat : rat * rat -> rat,
     val mulRat : rat * rat -> rat,
     val divRat : rat * rat -> rat,
     val equRat : rat * rat -> bool,
     val printRat : rat -> unit,
     val one : rat,
     val oneHalf : rat,
     val oneThird : rat,
     val twoThird : rat,
     val zero : rat}
```

Data abstraction with modules: rational numbers (cont.)

- The values defined in the `Rat` structure must be referenced with their full name:

```
Rat.printRat(Rat.mulRat(Rat.oneHalf, Rat.oneThird));
Rat.printRat(Rat.addRat(Rat.oneThird, Rat.oneThird));
```

- The content of the structure can be made visible – in measures limited by the signature – with `open`:

```
open Rat2;
equRat(addRat(oneThird, oneThird), twoThird);
addRat(oneThird, oneThird) = twoThird;
```

- The visibility can be local (declaration, or expression with local declaration):

```
local open Rat2
    val q1 = addRat(oneThird, oneThird); val q2 = twoThird
in  val ratPair = (q1, q2)
end;
```

```
let open Rat2
in  printRat(addRat(oneThird, oneThird))
end;
```

Data abstraction with modules: rational numbers (cont.)

- Let’s choose names more close to those in mathematics for the functions:

```
signature Rat =
sig
   eqtype rat
   val rat : int * int -> rat
   val num : rat -> int
   val den : rat -> int
   val ++ : rat * rat -> rat
   val -- : rat * rat -> rat
   val ** : rat * rat -> rat
   val // : rat * rat -> rat
   val == : rat * rat -> bool
   val toString : rat -> string
   val one : rat
   val oneHalf : rat
   val oneThird : rat
   val twoThird : rat
   val zero : rat
end;
```

Data abstraction with modules: rational numbers (cont.)

```
structure Rat3 :> Rat =
struct
  type rat = int * int;
  fun rat (n, d) =
        let val g = Gcd.gcd(n, d) in (n div g, d div g) : rat end
  fun num (q : rat) = #1 q
  fun den q = #2 (q : rat)
  fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
  fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
  fun op**(x, y) = rat(num x * num y, den x * den y)
  fun op//(x, y) = rat(num x * den y, den x * num y)
  fun op==(x, y) = num x * den y = den x * num y
  fun toString r = makestring(num r) ^ "/" ^ makestring(den r)
  val one      = rat(1,1)
  val zero     = rat(0,1)
  val oneHalf  = rat(1,2)
  val oneThird = rat(1,3)
  val twoThird = rat(2,3) end;
```

Data abstraction with modules: rational numbers (cont.)

- The new operators can be used in prefix position:

```
let open Rat3
in
   print(toString(++( **(oneThird, oneHalf),oneThird)) ^ "\n");
   ++(oneThird, oneThird) = twoThird
end;
```

At least one space is needed between `(` and `**`, or the MoSML considers it as beginning of a *comment*!

- Or they can be converted to infix position:

```
let open Rat3
    infix 6 ++ --
    infix 7 ** //
in
   print(toString(oneThird ** oneHalf ++ oneThird) ^ "\n");
   oneThird ++ oneThird = twoThird
end;
```

Data abstraction with modules: rational numbers (cont.)

- The common basic operators can also be redefined.
- Their original meaning doesn’t get lost, but the full name of the operations has to be used in *prefix* position:

```
load "Int";
let open Rat3
    val op+ = ++
    val op- = --
    val op* = **
    val op/ = //
in
   print(toString oneHalf ^ "\n");
   print(toString(oneHalf + oneThird) ^ "\n");
   print(toString(oneHalf * oneThird) ^ "\n");
   print(toString(oneThird - oneThird) ^ "\n");
   print(toString(twoThird / oneThird) ^ "\n");
   oneThird + oneThird = twoThird;
   Int.+(1,2)
end;
```

Note that `Int.+` cannot be used in infix position: `1 Int.+ 2 (* faulty!  *)`

Data abstraction with modules: rational numbers (cont.)

- *New type* and *new constructors* can be generated using the `datatype` declaration:

```
structure Rat4 :> Rat =
struct
  datatype rat = Rat of int * int
  fun rat (n, d) = let val g = Gcd.gcd(n, d) in Rat(n div g, d div g)
  fun num (Rat q) = #1 q
  fun den (Rat q) = #2 q
  fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
  fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
  fun op**(x, y) = rat(num x * num y, den x * den y)
  fun op//(x, y) = rat(num x * den y, den x * num y)
  fun op==(x, y) = num x * den y = den x * num y
  fun toString r = makestring(num r) ^ "/" ^ makestring(den r);
  val one       = rat(1,1)
  val zero      = rat(0,1)
  val oneHalf   = rat(1,2)
  val oneThird  = rat(1,3)
  val twoThird  = rat(2,3)  end;
```

Data abstraction with modules: rational numbers (cont.)

- The data constructor can (and should) be used for pattern matching as *selector*:

```
structure Rat5 :> Rat =
struct
  datatype rat = Rat of int * int;
  fun rat (n, d) = let val g = Gcd.gcd(n, d) in Rat(n div g, d div g)
  fun num (Rat(n, _)) = n
  fun den (Rat(_, d)) = d
  fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
  fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
  fun op**(x, y) = rat(num x * num y, den x * den y)
  fun op//(x, y) = rat(num x * den y, den x * num y)
  fun op==(x, y) = num x * den y = den x * num y
  fun toString r = makestring(num r) ^ "/" ^ makestring(den r);
  val one      = rat(1,1)
  val zero     = rat(0,1)
  val oneHalf  = rat(1,2)
  val oneThird = rat(1,3)
  val twoThird = rat(2,3)  end;
```

Data abstraction with modules: rational numbers (cont.)

- The data constructor function can *really* be used for the generation of a new useful value:

```
structure Rat6 :> Rat =
struct
  datatype rat = Rat of int * int;
  val rat = Rat;
  fun num (Rat(n, _)) = n
  fun den (Rat(_, d)) = d
  fun op++(x, y) = rat(num x * den y + num y * den x, den x * den y)
  fun op--(x, y) = rat(num x * den y - num y * den x, den x * den y)
  fun op**(x, y) = rat(num x * num y, den x * den y)
  fun op//(x, y) = rat(num x * den y, den x * num y)
  fun op==(x, y) = num x * den y = den x * num y
  fun toString r = makestring(num r) ^ "/" ^ makestring(den r);
  val one      = rat(1,1)
  val zero     = rat(0,1)
  val oneHalf  = rat(1,2)
  val oneThird = rat(1,3)
  val twoThird = rat(2,3)  end;
```

WEAK AND STRONG ABSTRACTION

Summary: weak and strong data abstraction

- Weak abstraction: the name is a synonim, the parts of the data structure are still accesible.
- Strong abstraction: the name stands for a new thing (entity, object), the parts of the data structure can only be accessed with restrictions.
- `type`: weak abstraction; ex. `type rat = {num : int, den : int}`
  - Gives new name to a type expression (see value declaration).
  - Helps understanding the program.
- `abstype`: strong abstraction
  - Creates a new type: name, operations, representation, notation.
  - Outworn, there's better: `datatype` + modules
- `datatype`: without modules weak, with modules strong abstraction;

  ex. `datatype ’a perhaps = Nothing | Something of ’a`

  Built-in version in SML: `datatype ’a option = NONE | SOME of ’a`
  - Creates a new entity.
  - Can be recursive and polymorphic.

Declaration with loccal declaration: `local` declaration

- Ún. `local`-deklarációt használunk, ha egyes deklarációkat fel akarunk használni más deklarációkban, miközben *el akarjuk rejteni* őket a program többi része elől.
- Szintaxisa: `local` *`d1`* `in` *`d2`* `end` ahol *`d1`* egy nemüres deklarációsorozat, *`d2`* egy másik nemüres deklarációsorozat.
- Példa:

```
(* length : 'a list -> int
   length zs = a zs lista hossza
*)
local
      (* len : 'a list * int -> int
         len (zs, n) = az n és a zs lista hosszának összege
      *)
      fun len ([], n)     = n
        | len (_::zs, n) = len(zs, n+1)
in
   fun length zs = len(zs, 0)
end
```

User-defined datatypes: about the `datatype` declaration again

- A new compound type called `person` is created:

```
datatype person = King
                | Peer of string * string * int
                | Knight of string
                | Peasant of string
```

- The new type has four *data constructor* (shortly: *constructor*):

  `King`, `Peer`, `Knight` and `Peasant`.

- `King` is a so-called *data constructor constant*, the rest are the so-called *data constructor functions*.

- The data constructors has type as well:

```
King :    person
Peer :    string * string * int -> person
Knight :  string -> person
Peasant : string -> person
```

The `datatype` declaration (continued)

```
King :    person
Peer :    string * string * int -> person
Knight :  string -> person
Peasant : string -> person
```

- There’s only one `King`, so it could be defined as a constructor constant.
- `Peer` is identified by his title (`string`), the name of his estate (`string`) and his ordinal number (`int`).
- `Knight` and `Peasant` are only identified by their name (`string`).
- Example on using the datatype `person`:

```
val persons = [King, Peasant "Jack Cade", Knight "Gawain",
                 Peer("Duke", "Norfolk", 9)];

> val persons = [King, Peasant "Jack Cade", ...] : person list
```

- Certain cases may be distinguished by pattern matching.
- All cases must be covered by a pattern; otherwise we are warned by the interpreter.
- Patterns can be arbitrarily complex.

The `datatype` declaration (continued)

- In the example below one of the four is the `Peasant name` *pattern*, and the `name` inside is the *pattern identifier*.

```
(* title : person -> string
   title p = title of p *)
fun title King = "His Majesty the King "
  | title (Peer (deg, ter, _)) = "The " ^ deg ^ " of " ^ ter
  | title (Knight name) = "Sir " ^ name
  | title (Peasant name) = name
```

- The function `sirs` gathers the names of all `Knight`-s from a list of people (`person`-s). (The order of the clauses is *important* because of the _ !):

```
(* sirs : person list -> string list
   sirs ps = the list of the names of all Knights *)
fun sirs [] = []
  | sirs ((Knight s)::ps) = s::sirs ps
  | sirs (_::ps) = sirs ps
```

The `datatype` declaration (continued)

- If the order of the clauses was different, the `_::ps` pattern would match `Knight` as well, not only `King`, `Peer` and `Peasant` (it stands for them in the example).
- Enumerating all disjunct cases helps proving the soundness of the algorithm.
- The three cases are closed up in one because their detailing would expand the code and the execution as well.
- Proving the soundness isn't problematic if the third line of the function (`sirs (_::ps) = sirs ps`) is considered a *conditional equation*:

$$\texttt{sirs(p::ps) = sirs ps}\ \textbf{if}\ \forall s \cdot p \neq \texttt{Knight}\ s.$$

The `datatype` declaration (continued)

- Order is more important in the following example, where hierarchy of people is observed. Instead of 16 only 7 cases have to be distinguished: which return *true*.

```
(* superior : person * person -> bool
   superior (p, r)= true if p has higher rank than r *)
fun superior (King, Peer _) = true
  | superior (King, Knight _) = true
  | superior (King, Peasant _) = true
  | superior (Peer _, Knight _) = true
  | superior (Peer _, Peasant _) = true
  | superior (Knight _, Peasant _) = true
  | superior _ = false
```

Enumeration type with `datatype` declaration

- It’s frequent that a name can take only few different values (the cardinality of the set of the values which can be taken by the name is small). In this case it’s useful to create an *enumeration type* with `datatype` declaration. For example

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron
```

- An enumeration type has only *constructor constants*. In order to use the new type the type `person` has to be declared again:

```
datatype person = King
                | Pear of degree * string * int
                | Knight of string
                | Peasant of string
```

Enumeration type with `datatype` declaration (continued)

- In case of data with type `degree` the cases have to be handled separately, for example

```
(* lady : degree -> string
   lady p = rank of p peer’s wife *)
fun lady Duke      = "Duchess "
  | lady Marquis   = "Marchioness"
  | lady Earl      = "Countess"
  | lady Viscount = "Viscountess"
  | lady Baron     = "Baroness"
```

- Type `Bool` with `Not function` similar to the internal `bool` could be declared/defined:

```
datatype Bool = True | False
(* Not : Bool -> Bool
   Not b = b negáltja *)
fun Not True = False | Not False = True
```

Polymorphic datatypes

- We have seen that `list` is a *postfix* positioned *type operator*, not a type: the `datatype` declaration also generates a *type constructor* beside the data constructors.
- `’a List` – similar to the internal `’a list` – with `Nil` and `Cons` *data constructors* can be defined in this way:

```
datatype ’a List = Nil | Cons of ’a * ’a List;
```

- Using the `Cons` *data constructor function* for creating lists is very inconvinient. For example the 1, 2, 3, 4 sequence has to be created in this way:

```
Cons(1, Cons(2, Cons(3, Cons(4, Nil))));
```

- The *infix* positioned `:::` data constructor operator can be introduced:

```
infix 5 ::: ; val op ::: = Cons
```

- The infix *triple-colon* can also be defined in the type declaration itself:

```
infix 5 ::: ; datatype ’a List = Nil | ::: of ’a * ’a List
```

Polymorphic datatypes: disjunct union

- Our next example is the *disjunct union* of two types:

```
datatype ('a, 'b) disun = In1 of 'a | In2 of 'b
```

- Three things can be defined:
  1. the `disun` type operator with two arguments,
  2. the `In1 : 'a -> ('a, 'b) disun` and
  3. the `In2 : 'b -> ('a, 'b) disun` data constructor functions.
- `('a, 'b) disun` is the disjunct union of types `'a` and `'b`. The union is called discunct because the base type of one or another element of the pair with type `('a, 'b) disun` can be determined later. The values of the new type has the form `In1 x` if `x` has `'a` type, and `In1 y` if `y` has `'b` type.
- The `In1` and `In2` constructor functions can be considered as such *labels* that distinguish type `'a` from type `'b`.

Disjunct union (continued)

- The disjunct union allows to use different types where only one type is allowed otherwise (see object oriented programming, where a *shape* class can have descendants like *rectangle*, *triangle* or *circle*).
- In SML lists with *elements of different types* can be created with disjunct union:

```
[In2 King, In1 "Scotland"] : ((string, person) disun) list;
[In1 "tyranne", In2 1040] : ((string, int) disun) list
```

- Possible cases can be processed with *pattern matching* as usual, for example

```
(* concat : (string, ’a) disun list -> string
   concat d = concatenation of elements with In1
              label of d disjunct union        *)

fun concat [] = ""
  | concat (In1 s :: ls) = s ^ concat ls
  | concat (In2 _ :: ls) = concat ls;
```

Disjunct union (continued)

- An example on using `concat`:

  ```
  concat [In1 "Oh! ", In2 King, In1 "Scotland"];
  ```

  ```
  > val it = "Oh! Scotland" : string
  ```

- The type of the `In1` constructor function is `'a -> ('a, 'b) disun`, so applied to the `"Oh!"` argument of type `string` its result has `(string, 'b) disun` type.
- The type of the `In2` constructor function is `'b -> ('a, 'b) disun`, so applied to the `"King"` argument of type `person` its result has `(string, 'a) disun` type.
- In the expression `[In1 "Oh!", In2 King, In1 "Scotland"]` all two base types are bound, so the type of this list is: `((string, person) disun) list`.
- The evaluation of the expression `[In2 "Ó", In2 King, In1 "Skócia"]` results in error, because the `'b` type variable can't be bound differently in the same expression.

CASE-STRUCTURE, OPTIONAL VALUE

Case-structure (`case`)

```
case E of P1 => E1 | P2 => E2 |  ···  | Pn => En
```

The SML-interpreter tries to match – from left to right and from up to down – `E` to `P1`, or – if it fails – to `P2` and so on. The result of the `case`-structure will be that `Ei` which belongs to the first `Pi` matching `E`.

`case` is also just a syntactic sugar becuase it can be replaced by `fn`-notation:

```
(fn P1 => E1 | P2 => E2 |  ···  | Pn => En) E
```

For examples the function `lady` could have been defined in this way:

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron
```

```
(* lady : degree -> string
   lady p = rank of p peer’s wife *)
fun lady p =
      case p of
        Duke     => "Duchess "
      | Marquis  => "Marchioness"
      | Earl     => "Countess"
      | Viscount => "Viscountess"
      | Baron    => "Baroness"
```

```
(* lady : degree -> string
   lady p = rank of p peer’s wife *)
fun lady p =
      (fn
         Duke     => "Duchess "
       | Marquis  => "Marchioness"
       | Earl     => "Countess"
       | Viscount => "Viscountess"
       | Baron    => "Baroness"
      ) p
```

Handling optional values (`’a option`)

```
datatype ’a option = NONE | SOME of ’a
```

Functions from the `Option` library:

```
val getOpt         : ’a option * ’a -> ’a
val isSome         : ’a option -> bool
val valOf          : ’a option -> ’a
val filter         : (’a -> bool) -> ’a -> ’a option
val map            : (’a -> ’b) -> ’a option -> ’b option
val mapPartial     : (’a -> ’b option) -> (’a option -> ’b option)
```

```
getOpt (xopt, d) = x if xopt is SOME x, d otherwise.

isSome xopt = true if xopt is SOME x, false otherwise.

valOf xopt = x if xopt is SOME x, raises Option otherwise.

filter p x = SOME x if p x is true, NONE otherwise.

map f xopt = SOME(f x) if xopt is SOME x, NONE otherwise.

mapPartial f xopt = f x if xopt is SOME x, NONE otherwise.
```

Examples on handling optional values

- Selecting the greatest element from an integer list

  An empty list doesn’t have a greatest element; the greatest element of a list with a single element is that single element; the greatest element of a list with at least two elements is the greatest among the first element and the rest of the list.

```
(* maxl : int list -> int option
   maxl ns = the greatest element of the ns integer list *)
fun maxl []      = NONE    (* empty *)
  | maxl [n]     = SOME n  (* one element *)
  | maxl (n::ns) =         (* at least two elements *)
               SOME(Int.max(n, valOf(maxl ns)))
```

- Converting a the beginning character sequence of a string to an integer

```
  val Int.fromString : string -> int option (* Overflow *)

  Int.fromString s = SOME i if a decimal integer numeral can be scanned
            from a prefix of string s, ignoring any initial whitespace;
            NONE otherwise.  A decimal integer numeral, after any initial
            whitespace, must have the form: [+~-]?[0-9]+

  Int.fromString "1234"; Int.fromString "-1234"; Int.fromString "~1234";
  Int.fromString "+1234"; Int.fromString "+007"; Int.fromString "alma"
```

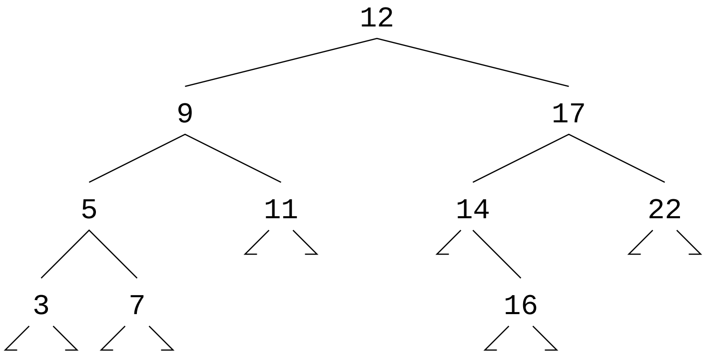
BINARY TREES

Binary trees with `datatype` declaration

- *Tree* is a recursive datatype similar to the list
- At first, the following binary tree is declared: its leaves are empty, and in the nodes the left subtree, the value of type `’a` and the right subtree is defined in this order.

```
datatype ’a tree = L | B of ’a tree * ’a * ’a tree;
```

- Let’s see the following tree:

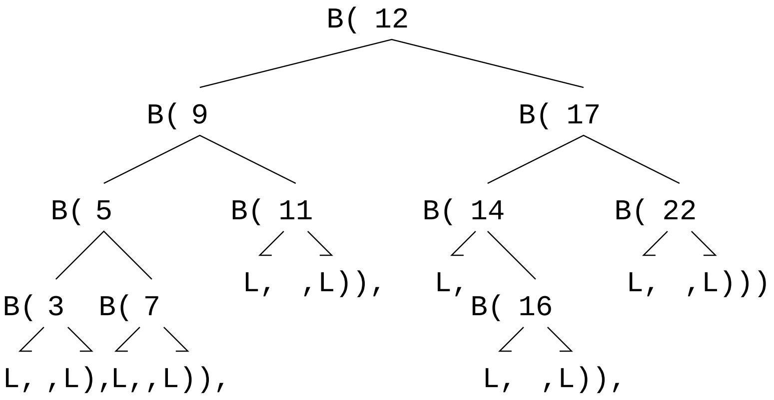


- This tree can be described with the `L` and `B` data constructors of the datatype `’a tree` as introduced on next page.

Binary trees with `datatype` declaration (continued)

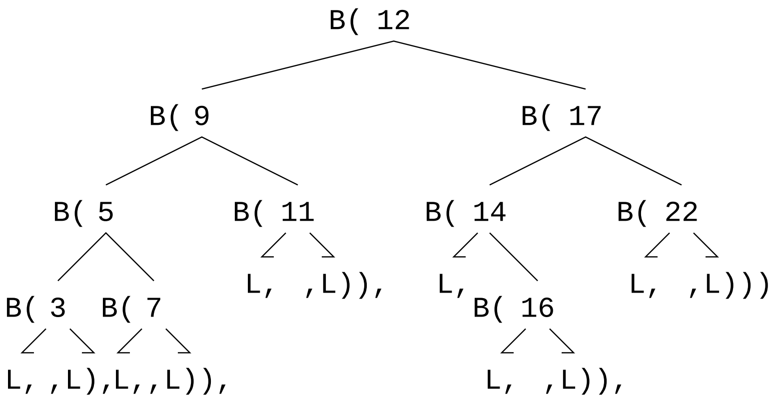
```
B(B(B(B(L,3,L),
      5,
      B(L,7,L)
     ),
    9,
    B(L,11,L)
   ),
  12,
  B(B(L,
      14,
      B(L,16,L)
     ),
    17,
    B(L,22,L)
   )
 );
```

The expression on the left side can’t be easily read. The textual description of the tree structure becomes more convinient when the corresponding data constructors are written into the figure.



Binary trees with `datatype` declaration (continued)

- The textual representation of the tree structure is more readable when the subtrees are given names, and the complete tree is built from subtrees:



```
                    B( 12
            B( 9                    B( 17
     B( 5        B( 11        B( 14        B( 22
                  L,  ,L)),    L,           L,  ,L)))
  B( 3  B( 7                     B( 16
  L, ,L),L,,L)),                  L,  ,L)),
```

```
val tr3  = B(L,3,L);          val tr7  = B(L,7,L);
val tr5  = B(tr3,5,tr7);      val tr11 = B(L,11,L);
val tr9  = B(tr5,9,tr11);     val tr16 = B(L,16,L);
val tr14 = B(L,14,tr16);      val tr22 = B(L,22,L);
val tr17 = B(tr14,17,tr22);   val tr12 = B(tr9,12,tr17);
```

Binary trees with `datatype` declaration (continued)

- Declaration of other tree structures is also possible, for example
  - it can be started with the value of type `’a`, followed by the left then the right subtree,
  - leaves can also store values,
  - empty stubs not containing a value can be described by `E`
- The following declaration declares a binary tree according to the properties described above:

```
datatype ’a tree = E | L of ’a | B of ’a * ’a tree * ’a tree
```

- Like the recursive functions, recursive data structures must have a non-recursive branch in the declaration (trivial case).
- Because of the absence of the non-recursive branch, the following – sintactically correct – declarations are useless:

```
datatype ’a badtree = B of ’a badtree * ’a * ’a badtree
datatype ’a badtree = L of ’a badtree
                    | B of ’a badtree * ’a * ’a badtree
```

Simple operations on binary trees

- `nodes` counts the nodes in a tree. Let

```
datatype 'a tree = L | N of 'a * 'a tree * 'a tree

(* nodes : 'a tree -> int
   nodes f = number of the nodes in the tree f *)
fun nodes (N(_, t1, t2)) = 1 + nodes t2 + nodes t1
  | nodes L = 0
```

- Accumulator-using version of `nodes` (`nodesa`):

```
fun nodesa f =
      let (* nodes0(f, n) = n + number of the nodes in the tree f
             nodes0 : 'a tree * int -> int *)
          fun nodes0 (N(_, t1, t2), n) =
                                   nodes0(t1, nodes0(t2, n+1))
            | nodes0 (L, n) = n
      in  nodes0(f, 0)
      end
```

Simple operations on binary trees (continued)

- The number of the edges on the path (the length of the path) from the root to a leaf in a tree is called the level of the leaf. The biggest of the levels is called the *depth* of the tree.
- `depth` calculates the depth of a tree

```
(* depth : 'a tree -> int
   depth f = depth of the tree f *)
fun depth  (N(_, t1, t2)) = 1 + Int.max(depth t2, depth t1)
  | depth L = 0
```

- Accumulator-using version of `depth` (`deptha`):

```
fun deptha f = let fun depth0 (N(_, t1, t2), d) =
                        Int.max(depth0(t1, d+1), depth0(t2, d+1))
                    | depth0 (L, d) = d
              in
                 depth0(f, 0)
              end
```

Simple operations on binary trees (cont'd.)

- `fulltree` builds a *full binary tree* of depth n and numbers each node from 1 to $2^n - 1$. In a full binary tree, exactly two edges start from each node, and each leaf is on the same level.

```
(* fulltree : int -> 'a tree
   fulltree n = full tree of depth n *)
fun fulltree n =
      let fun ftree (_, 0) = L
            | ftree (k, n) = N(k, ftree(2*k, n-1), ftree(2*k+1, n-1))
      in
          ftree(1, n)
      end
```

- `reflect` reflects the tree about the vertical axis.

```
(* reflect : 'a tree -> 'a tree
   reflect t = the tree t reflected about the vertical axis *)
fun reflect L = L
  | reflect (N(v,t1,t2)) = N(v, reflect t2, reflect t1)
```

Creating a list from the elements of a binary tree

- All three functions create *lists from binary trees*. They differ in when they take the elements stored in the nodes, and the traversal order.
  - `preorder` takes the element first, then traverses the left subtree, afterwards the right one;
  - `inorder` first traverses the left subtree, then takes the element, finally traverses the right one;
  - `postorder` first traverses the left subtree, then the right one, and takes the element in the end.
- The versions not using an accumulator are simple, comprehensible but inefficient due to the use of the operator @.

```
(* preorder : 'a tree -> 'a list
   preorder t = preorder list of the elements of the tree t *)
fun preorder L = []
  | preorder (N(v,t1,t2)) = v :: preorder t1 @ preorder t2
(* inorder : 'a tree -> 'a list
   inorder t = inorder list of the elements of the tree t *)
fun inorder L = []
  | inorder (N(v,t1,t2)) = inorder t1 @ (v :: inorder t2)
(* postorder : 'a tree -> 'a list
   postorder t = postorder list of the elements of the tree t *)
fun postorder L = []
  | postorder (N(v,t1,t2)) = postorder t1 @ (postorder t2 @ [v])
```

Creating a list from the elements of a binary tree (cont’d.)

- In the previous version of `inorder`, if we don’t put the subexpression `v :: inorder t2` of the expression `inorder t1 @ (v ::  inorder t2)` into brackets, the compiler gives an error message, because `::` and @ have the same precedence, so without the brackets it would try to evaluate the obviously incorrect subexpression `inorder t1 @ v`.
- In the following version of `inorder`, which is roughly equivalent to its previous implementation, we prepend `[v]`, a list with one element, instead of the element `v` to `inorder t2`:

  ```
  fun inorder L = []
    | inorder (N(v,t1,t2)) = inorder t1 @ ([v] @ inorder t2)
  ```

  However, this version is *very volatile*, because its efficiency depends on the brackets. If we don’t put the subexpression `[v] @ inorder t2` into brackets, the compiler will first evaluate the subexpression `inorder t1 @ [v]`, i.e. it appends a (usually) much longer list to one with a single element!
- For reasons similar to those mentioned, the presented version of `postorder` is also *extremely volatile!* For if we don’t put the anyway inefficient subexpression `postorder t2 @ [v]` of the expression `postorder t1 @ (postorder t2 @ [v])` into brackets, then the compiler first evaluates the subexpression `postorder t1 @ postorder t2`, i.e. appends the two presumably long lists, and then appends the result list to the list with a single element.

Creating a list from the elements of a binary tree (cont’d.)

The versions using an accumulator are more difficult to understand, but they are *more efficient*, mainly in terms of stack usage.

```
(* preord : ’a tree * ’a list -> ’a list
   preord(t, vs) = preorder list of the elements of the tree t,
                   prepended to the list vs *)
fun preord (L, vs) = vs
  | preord (N(v,t1,t2), vs) = v::preord(t1, preord(t2,vs))

(* inord : ’a tree * ’a list -> ’a list
   inord(t, vs) = inorder list of the elements of the tree t,
                  prepended to the list vs *)
fun inord (N(v,t1,t2), vs) = inord(t1, v::inord(t2,vs))
  | inord (L, vs) = vs

(* postord : ’a tree * ’a list -> ’a list
   postord(t, vs) = postorder list of the elements of the tree t,
                    prepended to the list vs *)
fun postord (N(v,t1,t2), vs) = postord(t1, postord(t2, v::vs))
  | postord (L, vs) = vs
```

Creating a binary tree from the elements of a list: `balPreorder`

- The following functions transform a list into a *balanced binary tree*: `balPreorder`, `balInorder` and `balPostorder`; the difference between them is the traversal order also this time.

-
  ```
  (* balPreorder: 'a list -> 'a tree
     balPreorder xs = preorder balanced tree
                      consisting of the elements of the list xs
  *)
  fun balPreorder [] = L
    | balPreorder (x::xs) =
        let val k = length xs div 2
        in
           N(x, balPreorder(List.take(xs, k)),
                balPreorder(List.drop(xs, k)))
        end
  ```
- Efficiency is slightly decreased by the fact that `List.take` and `List.drop` sweep through the first part of the list independently *twice*.

`take` and `drop` with one function: `take’ndrop`

- Let’s write a function named `take’ndrop`, whose argument is a pair consisting of a list and an integer, and whose result is a pair with the first member as the first `k` elements of the list, and the second member as the rest of the list.

```
(* take’ndrop : ’a list * int -> ’a list * ’a list
   take’ndrop(xs, k) = a pair with the first member as
                       the first k elements of xs,
                       and the second member as the rest of xs
*)
fun take’ndrop (xs, k) =
      let fun td (xs, 0, ts) = (rev ts, xs)
            | td ([], _, ts) = (rev ts, [])
            | td (x::xs, k, ts) = td(xs, k-1, x::ts)
      in
          td(xs, k, [])
      end
```

- Due to the usage of `take’ndrop`, specifically the pair returned, we need to modify the structure of `balPreorder`.

Creating a binary tree from the elements of a list: `balPreorder` revisited

- There was this:

```
fun balPreorder [] = L
  | balPreorder (x::xs) =
      let val k = length xs div 2
      in  N(x, balPreorder(List.take(xs, k)),
               balPreorder(List.drop(xs, k)))
      end
```

- ... which became this:

```
(* balPreorder: ’a list -> ’a tree
   balPreorder xs = preorder ... of the list xs *)
fun balPreorder [] = L
  | balPreorder (x::xs) =
      let val k = length xs div 2
          val (ts, ds) = take’ndrop(xs, k)
      in  N(x, balPreorder ts, balPreorder ds)
      end
```

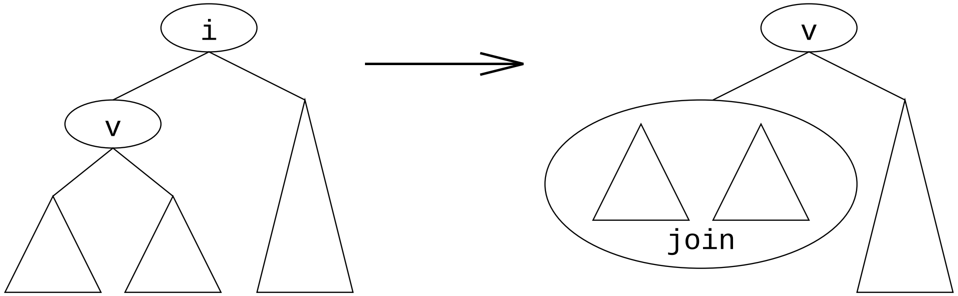
Creating a binary tree from the elements of a list

-
  ```
  (* balInorder: 'a list -> 'a tree
     balInorder xs = inorder balanced tree
                     consisting of the elements of the list xs
  *)
  fun balInorder [] = L
    | balInorder (xxs as x::xs) =
        let val k = length xxs div 2
            val ys = List.drop(xxs, k)
        in
            N(hd ys, balInorder(List.take(xxs, k)),
                     balInorder(tl ys))
        end
  ```

-
  ```
  (* balPostorder: 'a list -> 'a tree
     balPostorder xs = postorder balanced tree
                       consisting of the elements of the list xs
  *)
  fun balPostorder xs = balPreorder(rev xs)
  ```
- Defining `balInorder` with `take'ndrop` is an exercise.

Deleting an element from a binary tree

- *Finding* an *element* of a given value with a recursive method is an easy task.
- Neither is *inserting a new element* difficult: we seek a leaf with a recursive method, and replace it with the new element. If the tree is sorted, we must pay attention to keep the sorting.
- *Removing an element* or *elements* of a given value with a recursive method is somewhat harder: if the value to be deleted is in the root of the subtree being examined, then we need to *join* the subtrees of the tree falling into two pieces after we've performed the deletion on both subtrees.



- It is possible to join the two subtrees before deleting the element of the given value from the resulting tree.

Recursive deletion of an element from a binary tree (cont'd.)

- We join the two trees resulting from the deletion with `join`: it destroys the left subtree, and meanwhile puts its elements one-by-one into the right one.

```
(* join : 'a tree * 'a tree -> 'a tree
   join(l, r) = tree created by joining the trees l and r *)
fun join (L, tr) = tr
  | join (N(v, lt, rt), tr) = N(v, join(lt, rt), tr)
```

- `remove` removes *all* occurrences of the element of value `i` from an unsorted binary tree.

```
(* remove : 'a * 'a tree -> 'a tree
   remove(i, t) = removes all occurrences of i from t *)
fun remove (i, L) = L
  | remove (i, N(v,lt,rt)) =
      if i<>v
      then N(v, remove(i,lt), remove(i,rt))
      else join(remove(i,lt), remove(i,rt))
```

Binary search trees: `blookup`, `binsert`

- Usually we search for an element of a given key in a sorted binary tree, therefore we need to compare values, therefore the key searched for must be of *equality type* (in this example, we use the type `string`).
- The functions raise an *exception*, if the element of the key searched for isn’t present in the tree:

```
exception Bsearch of string
```

- The function `blookup` returns a value corresponding to a given key:

```
(* blookup : (string * 'a) tree * string -> 'a
   blookup(t, b) = the value corresponding to the key b
                   in the tree t
*)
fun blookup (L, b) = raise Bsearch("LOOKUP: " ^ b)
  | blookup (N((a,x), t1, t2), b) =
      if b < a      then blookup(t1,b)
      else if a < b then blookup(t2, b)
      else x;
```

Binary search trees: `bupdate`

- The function `binsert` inserts an element of a new key into a sorted binary tree, if it doesn't exist:

```
(* binsert : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   binsert(t, (b,y)) = the tree t extended with the new key-value pair (b,y) *)
fun binsert (L, (b,y)) = N((b,y), L, L)
  | binsert (N((a, x), t1, t2), (b,y)) =
      if b < a      then N((a, x), binsert(t1, (b,y)), t2)
      else if a < b then N((a, x), t1, binsert(t2, (b,y)))
      else (* a=b *) raise Bsearch("INSERT: " ^ b);
```

- The function `bupdate` writes a new value into an element of an existing key in a sorted binary tree:

```
(* bupdate : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   bupdate(t, (b,y)) = the tree t with the value y in place of
                       the value corresponding to the key b *)
fun bupdate (L, (b,y)) = raise Bsearch("UPDATE: " ^ b)
  | bupdate (N((a,x), t1, t2), (b,y)) =
      if b < a      then N((a,x), bupdate(t1, (b,y)), t2)
      else if a < b then N((a,x), t1, bupdate(t2, (b,y)))
      else (* a=b *) N((b,y), t1, t2);
```

- Making the functions *generic* is an exercise.

EXCEPTION HANDLING

Exception handling

- An exception is declared with the keyword `exception`, raised with the keyword `raise`, handled in the expression introduced with the keyword `handle`.
- Exceptions are usually used for indicating errors, but we can use them for handling backtracks as well (example for the latter can be seen in the function `change` on one of the following slides).
- Exception declaration reminds us of `datatype`-declaration: `exception` *`name`*`;` `exception` *`name`* `of` *`ty`*.
- Examples for declaring exceptions: `exception Change; exception Error of char * int`.
- The exception constructor can be a constant or a function. Examples: `Change : exn`, `Error : char * int -> exn`.
- The exception declaration is a special `datatype`-declaration, because in contrast to the latter it *extends* the set of exception constructors dynamically.
- For raising an exception, we must use the special expression beginning with the keyword `raise`.
- Examples for raising an exception: `raise Change`, `raise Error(#"N", 4)`.
- (Hypothetic) type of `raise` is `exn -> ’a`.

Exception handling (cont’d.)

- The outcome of applying `raise` is the so-called *exception pack*. Since the exception pack is of polymorphic type, it is compatible with all other types.
- Handling exceptions reminds us of the `case`-structure: `E handle P1 => E1 | ··· | Pn => En`
- If `E` returns a ”common” value, the exception handler simply forwards the result.
- If the result of `E` is an *exception pack*, SML tries to match it with the pattern `P1, ···, Pn`.
  - If `Pi` (`1 <= i <= n`) is the first matching pattern, the result of the exception handler is `Ei`.
  - If no patterns match the exception pack, the exception handler passes it on.
- Examples for handling exceptions:

  -
    ```
    coin :: change (coin::coinlist) (sum-coin)
              handle Change => change coinlist sum
    ```
  - `(fn i => exHan i handle Error(c, i) => (print(str c); i-1)) 0`
- (Hypothetic) type of `handle` is `exn -> ’a`.
- Let `Ex` be an exception of type `exn`, and let `e` be any expression; then `c` and `e` in the expression `e handle Ex => c` (containing an exception handler) must be of same type.

Exception handling (cont'd.)

- The next extract of program is an example for declaring, raising and handling an exception

```
exception Error of char * int;

fun exHan  0 = raise Error(#"N", 4)
  | exHan ~9 = raise Error(#"M", 9)
  | exHan  n = n;

fun exHandle i =
         exHan i handle Error(#"N", i) => (print "N"; i)
                      | Error(#"M", i) => (print "M"; i-1);

exHandle 0  = 4;
exHandle ~9 = 8;
exHandle 7  = 7;
```

Exception handling (cont'd.)

- Example for programming backtrack using exception handling

```
exception Change;

(* change : int list -> int -> int list
   change coinlist sum = the coin-list containing the fewest possible coins
                               whose sum is 'sum'
   PRE : coinlist = the coins for changing in decreasing order of value
         sum >= 0
*)
fun change _ 0  = []
  | change [] _ = raise Change
  | change (coin::coinlist) sum =
      if (* the actual coin is too large, we try the next one *)
         coin > sum then change coinlist sum
         (* if we manage to change starting with the actual coin, good;
            if not, we restart at the actual point with the next coin *)
      else coin :: change (coin::coinlist) (sum-coin)
                                      handle Change => change coinlist sum;

change [50, 20, 10, 5, 2] 197 = [50, 50, 50, 20, 20, 5, 2];
```

Exception handling (cont'd.)

- The most frequent built-in exceptions

Name	*Operation that might evoke it*
`Bind`	In value declaration, the right side expression doesn't match the left side pattern.
`Chr`	`chr pred succ`
`Div`	`/ div mod`
`Domain`	Value is out of domain.
`Empty`	`hd tl last`
`Fail`	`compile load loadOne` `Fail : string -> exn`
`Interrupt`	Interrupt by ctrl/c.
`Io`	Input/output error. `Io : {cause : exn, function : string, name : string}`
`Match`	Pattern matching error in `case` and `handle`, or in function application.
`Option`	Error when applying a function of the library `Option`.
`Overflow`	`~ + - * / div mod abs ceil floor round trunc`
`Size`	`^ array concat fromList implode tabulate translate vector`
`Subscript`	`copy drop extract nth sub substring take update`

- `Fail` and `Io` are ex. constr. functions, the others are ex. constr. constants of type `exn`.
- `Option` can be used only with the name `Option.Option` unless we open the library `Option`.

BACKTRACKING

n queens on a chessboard

How many ways are there, to place n queens to a chessboard of size $n \times n$, such that none of them attack each other?

- There is exactly one queen in each column. We describe the chessboard with a vector of length `n`, which's `i`th element, `s` is the row index of the queen in the `i`th column ($0 <=$`s`$< n$, $0 <=$`i`$< n$).
- Example for `n=4`:

```
+---+---+---+---+       +---+---+---+---+
| 2 | 0 | 3 | 1 |     0 |   | q |   |   |
+---+---+---+---+       +---+---+---+---+
  0   ---->  n-1      | |   |   |   | q |
                      | +---+---+---+---+
                      V | q |   |   |   |
           ===>         +---+---+---+---+
                     n-1|   |   | q |   |
                        +---+---+---+---+
```

- We implement vectors with lists.
- It is easy to append an element to a list from the left, for this reason we will flip our vector horizontally.

```
...-+---+---+---+       ...-+---+---+---+
    | 3 | 0 | 2 |     0     |   | q |   |
...-+---+---+---+       ...-+---+---+---+
   n-1  <----   0     |     |   |   |   |
                      | ...-+---+---+---+
                      V     |   |   | q |
        ===>            ...-+---+---+---+
                     n-1    | q |   |   |
                        ...-+---+---+---+
```

- The `i`th element in a vector of length `n` is the `n-(i+1)`th element in the list.

n queens on a chessboard

We can decide whether the new queen is attacked by the others with examining the vector.
We can place a new queen if:

1. The new queen is not in the same row as any of the others, so the new element of the list cannot show up in the already built part (tail) of the list.
2. The new queen cannot be attacked diagonally either. This means that if the newly placed queen is in the sth row, the new (0th) element in the list is `s`, then the `i`th element in the list cannot be nor `s-i`, neither `s+i`.

   The following example makes it clear:

If we want to place the new queen in the 1st row, then we have to check the placed marked with an `x`. Including the new element, the list has 3 elements. The element having index 1 cannot be neither $s - 1$, nor $s + 1$, the element having index 2 cannot be neither $s - 2$ nor $n + 2$.

```
...-+---+---+---+       ...-+---+---+---+
      1 |   |   |     0     |   | x |   |
...-+---+---+---+       ...-+---+---+---+
   n-1 <--- 1   0     1     | q |   |   |
                        ...-+---+---+---+
       ===>           |     |   | x |   |
                      V ...-+---+---+---+
                     n-1    |   |   | x |
                        ...-+---+---+---+
```

The list can be built by a recursive algorithm.

n queens on a chessboard: „attacked”-check

```
(* attacked : int list -> bool
   attacked zs = true, if the (hd zs) queen is attacked by at least
                 one other queen in (tl zs)
 *)
fun attacked [] = false
  | attacked (z::zs) =
      let (* att : int -> int -> int list -> bool
             att s1 s2 rs = true, if the queen z is attacked
                            by a queen in rs
          *)
          fun att _ _ [] = false
            | att s1 s2 (r::rs) = z = r orelse
                                  s1 = r orelse
                                  s2 = r orelse
                                  att (s1-1) (s2+1) rs
      in
          att (z-1) (z+1) zs
      end
```

n queens on a chessboard: producing a solution

```
exception Dead_end

(* queens0 : int -> int list
   queens0 n = a solution for the "n queens problem" *)
fun queens0 n =
  let (* queen : int -> int list -> int list
         queen z zs = a solution,
searching starts from placing queen to zth row, and the already placed queens are i
      fun queen z zs =
            if z = n (* backtracking is needed, if each row has been tried *)
            then raise Dead_end
            else if   (* z+1 should be tried, if z::zs is attacked *)
               attacked (z::zs) then queen (z+1) zs
            else if length (z::zs) = n
            then rev (z::zs) (* we have a solution *)
            else   (* continues placing the new queen from the 0th row,
                      and backtracks to the next row, if finds a dead-end *)
                 queen 0 (z::zs) handle Dead_end => queen (z+1) zs
  in
       (* starts with the 0th row *)
      queen 0 []
  end
```

n queens on a chessboard: producing a solution

```
exception Dead_end

(* queens0 : int -> int list
   queens0 n = a solution for the "n queens problem" *)
*)
fun queens0 n =
  let (* queen : int -> int list -> int list
         queen z zs = a solution,
      fun queen z zs =
            if (* backtracking is needed, if z=0 and is attacked *)
               z = 0 andalso attacked zs orelse
                 (* backtracking is needed, if each row has been tried *)
               z = n
            then raise Dead_end
            else if length zs = n
            then rev zs (* we have a solution *)
            else   (* continues placing the new queen from the 0th row,
                     and backtracks to the next row, if finds a dead-end *)
                 queens0 0 (z::zs) handle Dead_end => queen (z+1) zs
  in
       (* starts with the 0th row *)
      queen 0 []
  end
```

n queens on a chessboard: producing all the solutions with backtracking

```
(* queens1 : int -> int list list
   queens1 n = the list of all the solutions for the "n queens problem" *)
fun queens1 n =
  let (* queen: int -> int list -> int list list
         queen z zs: the list of all the solutions for the "n queens problem"
searching starts from placing queen to zth row,
and the already placed queens are in zs *)
      fun queen z zs =
            if
(* backtracking is needed, if z=0 and is attacked or if each row has been tried *)
               z = 0 andalso attacked zs orelse z = n
            then raise Dead_end
            else if length zs = n
            then [rev zs] (* we have a solution, we return it in a list *)
            else
(* continues with the next row, then appends the solution list... *)
                  (queen (z+1) zs handle Dead_end => []) @
(* ...to the solutions which comes from placing the next queen from the 0th row *)
                  (queen 0 (z::zs) handle Dead_end => [])
  in
       (* starts with the 0th row *)
      queen 0 []
  end
```

n queens on a chessboard: producing all the solutions in a list of lists

The pattern used in the previous example can be used many times, but in this simple case, it is unnecessary. Instead of using exceptions, we could simply return an empty list: the exception handlers did the same.

```
(* queens2 : int -> int list list
   queens2 n = all the solutions for the "n queens problem"
*)
fun queens2 n =
  let (* queen: int -> int list -> int list list
         queen z zs: all the solutions for the "n queens problem"
         searching starts from placing queen to zth row,
         and the already placed queens are in zs *)
      fun queen z zs =
          if z = 0 andalso attacked zs orelse z = n
          then []
          else if length zs = n
          then [rev zs]
          else queen (z+1) zs @ queen 0 (z::zs)
  in
      queen 0 []
  end
```

n queens on a chessboard: producing all the solutions in a list of lists

With using accumulator:

```
(* queens3 : int -> int list list
   queens3 n = a feladvány összes megoldásának listája
               n vezér esetén
*)
fun queens3 n =
  let (* queen: int -> int list -> int list list
         queen z zs zss: all the solutions for the "n queens problem"
         searching starts from placing queen to zth row,
         and the already placed queens are in zs, appended before zss *)
      fun queen z zs zss =
            if z = 0 andalso attack zs orelse z = n
            then zss
            else if length zs = n
            then rev zs :: zss
            else queen 0 (z::zs) (queen (z+1) zs zss)
  in
     queen 0 [] []
  end
```

SET OPERATIONS

Set operations: „is member?” (`isMem`) and „new member” (`newMem`)

- `isMem` returns true, if the element is found in the set

```
(* isMem : ''a * ''a list -> bool
   isMem(x, ys) = x is an element of ys
*)
fun op isMem (_, []) = false
  | op isMem (x, y::ys) = x = y orelse op isMem(x, ys)
infix isMem
```

Remark: without the `op` operator, after the `infix` declaration the function definition couldn’t be re-compiled.

- `newMem` inserts a new element in the list, if it isn’t yet a member of it.

```
(* newMem : ''a * ''a list -> ''a list
   newMem(x, xs) = union of [x] and xs sets as a list
*)
fun newMem (x, xs) = if x isMem xs
                     then xs
                     else x::xs
```

`newMem` creates a set.

Set operations: „set from list” (`setof`)

- `setof` makes a set from a list, with filtering out multiple occurences. Not efficient.

```
(* setof : ''a list -> ''a list
   setof xs = set of elements found in xs
*)
fun setof []        = []
  | setof (x::xs) = newMem(x, setof xs)
```

- We define five set operations:
  - union (`union`, $S \cup T$),
  - intersection (`inter`, $S \cap T$),
  - is-subset? (`isSubset`, $T \subseteq S$),
  - are-equal? (`isSetEq`, $S = T$),

Set operations: „union” (`union`) and „intersection” (`inter`)

- We store our sets as lists, later we’ll choose better representation, for example ordered lists or trees.
- Union of two sets:

```
(* union : ''a list * ''a list -> ''a list
   union(xs, ys) = union of the sets xs and ys
*)
fun union ([], ys)     = ys
  | union (x::xs, ys) = newMem(x, union(xs, ys))
```

- Intersection of two sets:

```
(* inter : ''a list * ''a list -> ''a list
   inter(xs, ys) = intersection of the sets xs and ys
*)
fun inter ([], _)      = []
  | inter (x::xs, ys) = let val zs = inter(xs, ys)
                        in
                           if x isMem ys then x::zs else zs
                        end
```

Set operations: „is subset of?” (`isSubset`) és „are equal?” (`isSetEq`)

- Is a set a subset of another?

```
(* isSubset : ''a list * ''a list -> bool
   isSubset (xs, ys) = true, if the set of the elements in xs
                       are a subset of the set ys
*)
fun op isSubset ([], _)      = true
  | op isSubset (x::xs, ys) = (x isMem ys) andalso
                              op isSubset(xs, ys)

infix isSubset
```

- Checking the equality of two sets. Checking the equality of two lists is built-in SML, but we can’t use that, because for example the sets `[3,4]` and `[4,3]` are equal, although their lists aren’t.

```
(* isSetEq : ''a list * ''a list -> bool
   isSetEq(xs, ys) = the set of elements in xs is equal to
                     the set of elements in ys
*)
fun isSetEq (xs, ys) = (xs isSubset ys) andalso (ys isSubset xs)
```

SIMULTANEOUS DECLARATION

Simultaneous declaration

- Types and values can be declared simultaneously, with using the `and` keyword.
- See the following declarations:

```
type row = int; type column = int;
datatype fa = L | B of fa * fa;
datatype 'a stack = >| | >> of 'a * 'a stack;
val v1 = "a"; val v2 = "z";
fun f1 i = i +1; fun f2 i = i - 1;
```

SML evaluates these in their order in the source code.

```
type sor = int and osz = int;

datatype fa = L | B of fa * fa and
'a verem = >| | >> of 'a * 'a verem;

val v1 = "a" and v2 = "z";
fun f1 i = i +1 and f2 i = i - 1;
```

The declarations separated by the `and` keyword are evaluated simultaneously.

Simultaneous declaration

- We have to use simultaneous declaration for defining mutually recursive functions. Example:

```
fun even 0 = true | even n = odd(n-1)
and odd 0 = false | odd n = even(n-1);
```

- We can use simultaneous declaration to exchange two or more name bindings. Example:

```
val v1 = "a"; val v2 = "z"; val v1 = v2 and v2 = v1;
```

- We use simultaneous declaration if we want top-down design in the source code. Example:

```
fun length zs = len zs 0
and len [] i = i | len (_ :: xs) i = len xs (i+1);
```

- Polymorphic functions are treated differently by sequential and simultaneous declaration, because SML carries out its type-deriving method for the whole expression. Example:

```
fun id x = x; fun hi () = id 3; fun nr () = id 4.0;
fun id x = x  and hi () = id 3  and nr () = id 4.0;
```

  After evaluating the first line, `id` has type `’a -> ’a`. In the case of the second line, `id` should have types `int -> int` and `real -> real` simultaneously, which leads to a type error.

THE ORDER TYPE

The `order` type

The definition of the `order` type: (see `General.sig`)

```
datatype order = LESS | EQUAL | GREATER

[order] is used as the return type of comparison functions.
```

Examples from the SML Basis Library:

```
Int.compare     : int * int -> order
Char.compare    : char * char -> order
Real.compare    : real * real -> order
String.compare  : string * string -> order
Time.compare    : time * time -> order
```

SORTING LISTS

Sorting lists

- **`inssort` (Insertion sort),**
- **`selsort` (selection sort),**
- **`quicksort` (quicksort),**
- **`tmsort` (top-down merge sort),**
- `bmsort` (bottom-up merge sort),
- `smsort` (smooth sort).

Insertion sort

- The `ins` auxiliary function inserts the element `x` to its proper place in a list:

```
(* ins : real * real list -> real list
   ins (x, ys) = x inserted in ys to its proper place
                 according to the <= relation
   PRE: ys is sorted according to the <= relation *)
fun ins (x, y::ys) = if x <= y then x::y::ys else y::ins(x, ys)
  | ins (x : real, []) = [x]
```

- We use `inssort` recursively to sort the tail of the list. Execution time is $O(n^2)$:

```
(* inssort : ('a * 'b list -> 'b list) -> 'a list -> 'b list
   inssort f xs =
           the sorted list consisting of the elements of xs
           and f used as an insertion function *)
fun inssort f (x::xs) = f(x, inssort f xs)
  | inssort _  [] = [];
```

- Example for using `inssort`:

```
insort ins [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0];
```

Insertion sort, generic variant

- We make the `ins` function generic:

```
(* ins : ('a * 'a -> bool) -> 'a * 'a list -> 'a list
ins cmp (x, ys) = x inserted in ys to its proper place
                 according to the cmp relation
   PRE: ys is sorted according to the cmp relation *)
fun ins cmp (x, ys) =
      let fun ins0 (y::ys) =
                    if cmp(x, y) then x::y::ys else y::ins0 ys
            | ins0 [] = [x]
      in  ins0 ys
      end
```

- With this, a new variant of `inssort`:

```
(* inssort : ('a * 'a -> bool) -> 'a list -> 'a list
   inssort cmp xs = the sorted list consisting of
the elements of xs, according to cmp *)
fun inssort cmp (x::xs) = ins cmp (x, inssort cmp xs)
  | inssort _ [] = []
```

Insertion sort, generic variant

- The previous variants of `inssort` first take the list apart to elements, then they build the result list from the end.
- This right-recursive variant (`inssort2`) uses less stack, because it inserts the elements into the result list while walking on the list from the left to the right. (later we’ll compare execution times)

```
(* inssort2 : ('a * 'a -> bool) -> 'a list -> 'a list
   inssort2 cmp xs =
           the sorted list consisting of the elements of xs
           according to cmp *)
fun inssort2 cmp xs =
      let (* sort :  'a list -> 'a list -> 'a list
             sort xs zs = the elements of xs inserted into zs,
             in their proper place according to the cmp relation
             PRE: zs is sorted according to cmp *)
          fun sort (x::xs) zs = sort xs (ins cmp (x, zs))
            | sort [] zs = zs
      in
          sort xs []
      end
```

Insertion sort with `foldr`-rel and `foldl`

- `foldl` uses its second argument as an accumulator, it uses less stack, it can sort longer lists.

```
fun inssortR cmp = foldr (ins cmp) [];
fun inssortL cmp = foldl (ins cmp) [];
```

- Examples for `insort` and `insort2`:

```
inssort op<= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0];
inssort2 op>= [4, 4, 5, 1, 0, 8];
inssort op< (explode "qwerty");
```

- Examples for using `foldr` and `foldl`:

```
fun inssortRi cmp = foldr (ins cmp) [];
fun inssortLr cmp = foldl (ins cmp) ([] : real list);

inssortRi op>= [4, 4, 5, 1, 0, 8];
inssortLr op>= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0];
```

Measuring and comparing execution times

- Sorting lists of length 2000, one filled with random elements and a reversed sorted list.
  - `Random.randomlist (n, gen)` returns a list of n random numbers in the interval [0,1)

    ```
    val xs2000R =
          Random.rangelist (1, 100000) (2000, Random.newgen());
    ```

  - The `---` operator generates a list of increasing numbers:

    ```
    infix ---;
    fun fm --- to =
          let fun upto to zs =
                    if to < fm then zs else upto (to-1) (to::zs)
          in
              upto to []
          end;

    val xs2000N = 1 --- 2000;
    ```

Measuring and comparing execution times

- We measure time with the following functions:

```
app load ["Timer","Time","Int"];

fun runTime (sort, sortFn) (cmp, cmpFn) (xs, kind) =
      let val starttime = Timer.startCPUTimer()
          val zs = sort cmp xs
          val usr=tim,... = Timer.checkCPUTimer starttime
      in
          "Int sort with " ^ sortFn ^ ", " ^ cmpFn ^
          ", length = " ^ Int.toString(length xs) ^ " (" ^
          kind ^ "), time = " ^ Time.fmt 2 tim ^ " sec\n"
      end;

val t1N =
      runTime (inssort, "inssort") (op>=, "op>=") (xs2000N, "increasing");
val t2N =
      runTime (inssort2, "inssort2") (op>=, "op>=") (xs2000N, "increasing");
val t1R =
      runTime (inssort, "inssort") (op>=, "op>=") (xs2000R, "random");
val t2R =
      runTime (inssort2, "inssort2") (op>=, "op>=") (xs2000R, "random");
```

Measuring and comparing execution times

- Soring the reversely sorted list with 2000 elements with the non-right-recursive version of `inssort` takes more than 5s, while with the right-recursive version it takes only 0.01s. (linux, 233 MHz-es Pentium)

```
Int sort with inssort, op>=, length = 2000 (increasing), time = 5.18 sec
Int sort with inssort2, op>=, length = 2000 (increasing), time = 0.01 sec
Int sort with inssortRi, op>=, length = 2000 (increasing), time = 5.14 sec
Int sort with inssortLi, op>=, length = 2000 (increasing), time = 0.01 sec
```

- The difference disappears, if we sort the lists with random elements.

```
Int sort with inssort, op>=, length = 2000 (random), time = 2.39 sec
Int sort with inssort2, op>=, length = 2000 (random), time = 2.26 sec
Int sort with inssortRi, op>=, length = 2000 (random), time = 2.40 sec
Int sort with inssortLi, op>=, length = 2000 (random), time = 2.24 sec
```

Selection sort

```
(* selsort : (’a * ’a -> order) -> ’a list -> ’a list
   selsort cmp xs = the elements of xs in increasing order
*)
fun selsort cmp xs =
      let
          (* max : ’a * ’a -> ’a
             max (x, y) = the max of x and y, according to cmp
          *)
          fun max (x, y) = if cmp(x, y) = GREATER then x else y

          (* min : ’a * ’a -> ’a
             min (x, y) = the min of x and y, according to cmp
          *)
          fun min (x, y) = if cmp(x, y) = LESS then x else y

          (* maxSelect : ’a * ’a list * ’a list -> ’a * ’a list
             maxSelect (x, ys, zs) =
a pair, consisting of the largest element of (x::ys) according to cmp,
and a list consisting of the elements of x::ys and zs *)
          fun maxSelect (x, [], zs) = (x, zs)
            | maxSelect (x, y::ys, zs) =
                           maxSelect(max(x, y), ys, min(x,y)::zs);
```

Selection sort, continued

```
          (* sSort : ’a list * ’a list -> ’a list
             sSort (xs, ws) =
the elements of xs appended in front of ws, in increasing order *)
          fun sSort ([], ws) = ws
            | sSort (x::xs, ws) =
                let val (z, zs) = maxSelect(x, xs, [])
                in
                   sSort (zs, z::ws)
                end
      in
          sSort (xs, [])
      end;

app load ["Int","Char","Real"];

selsort Int.compare [1,2,3,4,5,6,7,8,9];
selsort Int.compare [9,8,7,6,5,4,3,2,1];
selsort Real.compare [4.5,6.7,3.6,4.3,1.2,0.9,8.9,9.8,2.0];
selsort Char.compare (explode "Apple Pear Plum");
```

Quicksort without using accumulator

```
(* quicksort1 cmp xs = the elements of xs sorted according to cmp
   quicksort1 : ('a * 'a -> order) -> 'a list -> 'a list *)
fun quicksort1 cmp xs =
      let (* qs : 'a list -> 'a list
             qs ys =
the elements of ys sorted according to cmp
           *)
          fun qs (m::ys) =
                let (* partition : 'a list * 'a list * 'a list -> 'a list
                       partition (xs, ls, rs) = a pair consisting of
the elements of xs which are smaller than m, appended in front of ls,
and the rest, appended in front of rs *)
                    fun partition (x::xs, ls, rs) =
                          if cmp(x, m) = LESS then partition(xs, x::ls, rs)
                                              else partition(xs, ls, x::rs)
                      | partition ([], ls, rs) = qs ls @ (m::qs rs)
                in
                    partition (ys, [], [])
                end
            | qs [] = []
      in
          qs xs
      end;
```

Quicksort with accumulator

```
(* quicksort2 cmp xs = the elements of xs sorted according to cmp
   quicksort2 : ('a * 'a -> order) -> 'a list -> 'a list *)
fun quicksort2 cmp xs =
      let (* qs : 'a list -> 'a list -> 'a list
             qs ys zs =
the elements of ys sorted according to cmp, appended in front of zs
           *)
          fun qs (m::ys) zs =
                let (* partition : 'a list * 'a list * 'a list -> 'a list
                       partition (xs, ls, rs) = a pair consisting of
the elements of xs which are smaller than m, appended in front of ls,
and the rest, appended in front of rs *)
                    fun partition (x::xs, ls, rs) =
                          if cmp(x, m) = LESS then partition(xs, x::ls, rs)
                                              else partition(xs, ls, x::rs)
                      | partition ([], ls, rs) = qs ls (m :: qs rs zs)
                in
                    partition (ys, [], [])
                end
            | qs [] zs = zs
      in
          qs xs []
      end;
```

Measuring and comparing execution times

```
app load ["Listsort","Int"];
val t1 = futIdo (inssort2, "inssort2") (op>=, "op>=") (xs2000R, "random");
                                                    (* ~ 2 M comparisons! *)
val t3 = futIdo (quicksort2, "quicksort2")
                (Int.compare, "Int.compare") (xs2000R, "random");
val t4 = futIdo (Listsort.sort, "Listsort.sort")
                (Int.compare, "Int.compare") (xs2000R, "random");
                                                    (* ~ 300 E comparisons *)

Int sort with inssort2, op>=, length = 2000 (random), time = 2.30 sec

Int sort with quicksort1, Int.compare, length = 20000 (random), time = 2.18 sec
Int sort with quicksort2, Int.compare, length = 20000 (random), time = 1.72 sec
Int sort with Listsort.sort, Int.compare, length = 20000 (random), time = 1.76 sec

Int sort with quicksort2, Int.compare, length = 200000 (random), time = 27.13 sec
Int sort with quicksort1, Int.compare, length = 200000 (random), time = 32.59 sec

val t7 = futIdo (Listsort.sort, "Listsort.sort") (Int.compare, "Int.compare")
         (Random.rangelist (1, 100000) (200000, Random.newgen()), "random");
! Uncaught exception:
! Out_of_memory
```

Merge sort

- For the merge sort, we need a function which unifies two sorted lists (merges the lists).

```
(* merge(xs, ys) = xs and ys merged according to <=
   merge : int list * int list -> int list
*)
fun merge (xxs as x::xs, yys as y::ys)=
      if x <= y
      then x::merge(xs, yys)
      else y::merge(xxs, ys)
  | merge ([], ys) = ys
  | merge (xs, []) = xs;
```

- Inefficient, if we store the partial results in the stack.
- The result must be reversed if we use an accumulator.

Top-down merge sort

- The „top-down merge sort” is efficient, if the two lists are of nearly the same length.

```
(* tmsort xs = the elements of xs sorted according to <=
   tmsort : int list -> int list
*)
fun tmsort xs = let val h = length xs
                    val k = h div 2
                in
                   if h > 1
                   then merge(tmsort(List.take(xs, k)),
                              tmsort(List.drop(xs, k)))
                   else xs
                end;
```

- It needs $O(n \cdot log\, n)$ steps in the worst case.