Lars Birkedal

♦ Cand. Scient. from University of Copenhagen, 1994
  ✺ Advisors: Neil Jones and Mads Tofte
  ✺ Partial Evaluation of SML
  ✺ The ML Kit with Regions

♦ Ph.D. from Carnegie Mellon University, 1999
  ✺ Advisor: Dana Scott
  ✺ Realizability models of type theories and logics

♦ Started at ITU March 2000 as Assistant Professor

♦ Now Associate Professor and Head of Theory Department

♦ Head of LaCoMoCo research theme
Two concepts: *expression* and *declaration*.

Expressions evaluate to a *value* (or diverge).

A declaration contains one or more expressions:

```sml
val five = 5
fun addFive x = x + five
```

An *environment* consists of *bindings*.

We can calculate the *meaning* of SML programs by hand.
New kind of expression.

Package values together.

The type constructor for tuples is an asterisk \( * \).

A 2–tuple is called a *pair* and a 3–tuple is called a *triple*.

Examples:

\[
(5, \text{false}) : \text{int} * \text{bool} \\
(\"John\", 42, \text{true}) : \text{string} * \text{int} * \text{bool} \\
(1, (2, (3, 4))) : \text{int} * (\text{int} * (\text{int} * \text{int}))
\]
Taking tuples apart

♦ Selectors:

#2 (1, (2, 3), 4) \[\leadsto\] (2, 3)

♦ Patterns:

``` OCaml
val tup = (1, ("John", 42, true))
val (n, _) = tup
val tup = (1, ("John", 42, true))
    : int * (string * int * bool)
val n = 1 : int
```
fun power(x, 0) = 1.0
  | power(x, n) = x * power(x, n - 1)

fun power' arg =
  if #2 arg = 0 then 1.0
  else #1 arg * power'(#1 arg, #2 arg - 1)
Declare your own infix

♦ The form of infix directives

\[
\text{infix } d \ id_1 \ldots \ id_n \\
\text{infixr } d \ id_1 \ldots \ id_n
\]

where \( d \) is an optional integer denoting operator priority. After the directive we say that \( id_1 \ldots id_n \) has \textit{infix status}.

♦ The directive \texttt{nonfix } id cancel the infix status of \( id \).

♦ Infix status affect both expressions and patterns.
Some functions looks nicer when they are written infix

```haskell
infix isBetween
fun n isBetween (x, y) = x <= n andalso n <= y
```

Don’t you think?

```haskell
if x isBetween (0, 25) then "Young" else "Old"
vs.
if isBetween(x, (0, 25)) then "Young" else "Old"
```
Don’t overuse *infix* directives

Careful use of *infix* directives can make code easier to read. But it can also make it completely unreadable.

Think of *infix* like hot chili spice.
Records

♦ New kind of expression.
♦ Package values together.
♦ Generalised tuples, components are called *fields* and is identified by a *label* instead of a position.
♦ Example:

```scala
val iron9 = {nr = 9, range = (101, 119)}
val iron9 = {nr = 9, range = (101, 119)}
: {nr : int, range : int * int}
```
♦ The order of fields and labels does not matter.
♦ (Think of a Java object with no methods).
Taking records apart

♦ Selectors:

\[
#x \{x = 1.0, y = 42.0\} \sim 1.0
\]

♦ Patterns:

\[
\text{val iron9} = \{\text{nr} = 9, \text{range} = (101, 119)\}
\]
\[
\text{val } \{\text{nr} = n, \text{range} = r\} = \text{iron9}
\]
\[
\text{val } \{\text{nr}, \text{range}\} = \text{iron9}
\]
\[
\text{val } \{\text{nr}, \ldots\} = \text{iron9}
\]
Type declarations

Sometimes we want to bind a name to certain type.

♦ For abbreviation purposes.
♦ For documentation.
♦ For maintenance.

Examples:

```haskell
type iron = {nr : int, range : int * int, 
             brand : string}

type age    = int

type person = {name : string, age : age, sex : string}
```
Example: Quadratic equations

♦ All quadratic equations can be brought on a *canonical normal form*:

\[ ax^2 + bx + c = 0 \]

with coefficients \( a, b, \) and \( c. \)

♦ If \( b^2 - 4ac \geq 0\) and \( a \neq 0 \) then there are two solutions:

\[
x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}
\]

and

\[
x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}
\]

♦ What is the essential?
1. Important concepts:

   type equation = {a : real, b : real, c : real}
   type solution = real * real

2. Function interface(s):

   solve : equation -> solution
   (solve : {a : real, b : real, c : real} -> real * real)

3. Tests:

   The equation $2x^2 + 2x - 4 = 0$ has the solutions $x_1 = 1$ and $x_2 = -2$.

   val test1 = solve{a=2.0, b=2.0, c= ~4.0} = (1.0, ~2.0)
Example: Quadratic equations(3)

The code:

```javascript
fun solve{a, b, c} =
  ( (~b + Math.sqrt(b*b - 4.0 * a * c))/(2.0 * a), (~b - Math.sqrt(b*b - 4.0 * a * c))/(2.0 * a) )
```
 Exceptions

♦ New kind of expressions.
♦ Used to signal errors (*exceptional* events).
♦ New type `exn`.
♦ New kind of declaration. Example:
  ```plaintext
  exception OhNoooo
  
  exn OhNoooo = OhNoooo : exn
  ```
♦ An exception can be *raised* to signal an error. This is called a *raise* expression. Example:
  ```plaintext
  raise OhNoooo
  ```
We want solve e to signal an error, if there are no solutions to the equation e, or if e is an invalid quadratic equation.

```plaintext
exception Solve
fun solve{a, b, c} =
  if b*b - 4.0 * a * c < 0.0
    orelse
    Real.==(a, 0.0)
  then raise Solve
  else
    ( (~b + Math.sqrt(b*b - 4.0 * a * c))/(2.0 * a), (~b - Math.sqrt(b*b - 4.0 * a * c))/(2.0 * a) )
```
Local identifiers

♦ Two kinds:
  ✤ let expression
  ✤ local declaration

♦ Handy for making large expressions more readable and to signal intend.

♦ If you can only learn one, pick let.
local declaration

♦ Compose several declarations into one declaration
♦ Form:

\[
\text{local } \textit{dec}_1 \text{ in } \textit{dec}_2 \text{ end}
\]

where \( \textit{dec}_1 \) and \( \textit{dec}_2 \) are declarations.

♦ Example:

\[
\text{local val five = 4+1 in fun addFive x = x + five end}
\]

\[
\text{val addFive = fn : int -> int}
\]
local declaration, environments

[]
local
val five = 4+1
in
[five ↦→ 5]
fun addFive x = x + five
[[ five    ↦→  5 ]
 [ addFive ↦→  ... ]] end
[addFive ↦→ ...]
let expression

♦ Compose some declarations and an expression into an expression.

♦ Form:

let \textit{dec} in \textit{exp} end

where \textit{dec} is zero or more declarations and \textit{exp} is an expression.

♦ Example: \texttt{let val x = 42 in x * x end} \Rightarrow 1764
local

    infix ==
    fun x == y = Real.== (x, y)

in

fun solve{a, b, c} =
    let val d = b*b - 4.0 * a * c
    in  if d < 0.0 orelse a == 0.0 then raise Solve
        else ( (~b + Math.sqrt d)/(2.0 * a)
        , (~b - Math.sqrt d)/(2.0 * a) )
    end

end
<table>
<thead>
<tr>
<th>Activity</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem analysis</td>
<td>Interface definition</td>
</tr>
<tr>
<td>Programming</td>
<td>Program</td>
</tr>
<tr>
<td>Testing</td>
<td>Test</td>
</tr>
</tbody>
</table>

End result: *Technical documentation*
Interface definition

♦ The most important part.

♦ User documentation (who is the user?)

♦ Contains a list of specifications:
  ✴ types
  ✴ exceptions
  ✴ constants
  ✴ functions

♦ An interface definition is called a *signature* in SML.
Example: Rational numbers, interface

```ocaml
type qnum = int * int
exception QDiv
mkQ : int * int -> qnum
++  : qnum * qnum -> qnum
--  : qnum * qnum -> qnum
**  : qnum * qnum -> qnum
//  : qnum * qnum -> qnum
==  : qnum * qnum -> bool
toString : qnum -> string
```

Testing

A program (module) test consists of two parts:

♦ Test cases
♦ Test suite

Side note: This kind of test is called a *structural test* (or *white-box test*). For a good introduction to testing, see:

http://www.dina.kvl.dk/~sestoft/programmering/struktur.pdf
Example: Rational numbers, test cases

<table>
<thead>
<tr>
<th>Case</th>
<th>Function</th>
<th>Branch</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mkQ</td>
<td>(_, 0)</td>
<td>Denominator is zero</td>
</tr>
<tr>
<td>2</td>
<td>mkQ</td>
<td>pr</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>gcd</td>
<td>(0, n)</td>
<td>Final value</td>
</tr>
<tr>
<td>11</td>
<td>gcd</td>
<td>(m, n)</td>
<td>Recursive call</td>
</tr>
</tbody>
</table>
Example: Rational numbers, test suite

<table>
<thead>
<tr>
<th>Test nr</th>
<th>Case</th>
<th>Test</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>val q4 = q2 ** q3</td>
<td>val q4 = ((\sim 175), (\sim 300))</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>9</td>
<td>8, 10, 11</td>
<td>toString q4</td>
<td>val it = &quot;7/12&quot;</td>
</tr>
</tbody>
</table>
Ken’s detailed method for micro-development

For one function:

1. Write short problem statement
2. Figure out name
3. Write some tests (functional tests)
4. Figure out type (the function’s signature)
5. Write function (possibly introduce helper functions)
6. Write tests based on implementation
7. Test function
8. Write (short)documentation for function
What have we learned today

♦ $3 \frac{1}{2}$ new kinds of expressions: tuples, records, exceptions, let-expressions.
  With associated types and patterns.

♦ infix directive

♦ 3 new kinds of declarations: local-declarations, exception declarations, type declarations.
  We can name our types.

♦ Program development method. Technical documentation.
  ★ Interface definition
  ★ Rigorous testing
Exercise: Date library

A date can be represented by *a year, a month, and a day in the month*. Every day has a successor. Dates have an ordering. Some dates are in *leap years*. A leap year is divisible by four and not divisible by 100 except if it is divisible by 400; for example, 1900 is not a leap year, but 2000 is a leap year. The ISO format for dates is `YYYY-MM-DD`; example: `2002-09-04`. 