

Introduction to SML

Basic Types, Tuples and Records

Michael R. Hansen

mrh@imm.dtu.dk

Informatics and Mathematical Modelling
Technical University of Denmark

Overview

Today

- Integers, Reals, Characters, Strings, Booleans
- Expression, Precedence, Association
- Tuples and Patterns (**Records: read by yourselves**)
- Locally declared identifiers

Overview

Today

- Integers, Reals, Characters, Strings, Booleans
- Expression, Precedence, Association
- Tuples and Patterns (**Records: read by yourselves**)
- Locally declared identifiers

Next programming lecture

- Lists
- Applications in connection with languages and automata

Last programming exercises

- A file can contain several declarations
- The use directive is useful:

```
use "file.sml";
```

includes all declarations contained in the "file.sml".

- An identifier may only occur once in a pattern, e.g.:

```
fun bin(n,0) = 1
  | bin(n,k) = if k=n then 1
                else bin(n-1,k-1) + bin(n-1,k);
```

The following is not allowed:

```
...
| bin(n,n) = 1
...
```

Basic Types: Integers

A data type comprises

- a set of **values** and
- a collection of **operations**

Basic Types: Integers

A data type comprises

- a set of **values** and
- a collection of **operations**

Integers

Type name : `int`

Values : `~27, 0, 1024`

Operations: (A few selected)

Operator	Type	Precedence	Association
<code>~</code>	<code>int -> int</code>	Highest	
<code>* div mod</code>	<code>int * int -> int</code>	7	Left
<code>+ -</code>	<code>int * int -> int</code>	6	Left
<code>= <> < <=</code>	<code>int * int -> bool</code>	4	Left

See also the library **Int**

Reals

Type name : `real`

Values : `~27.0`, `0.0`, `1024.71717`, `23.4E~11`

Operations: (A few selected)

Operator	Type	Precedence	Association
<code>abs</code>	<code>real -> real</code>	Highest	
<code>*</code> <code>/</code>	<code>real*real -> real</code>	7	Left
<code>+</code> <code>-</code>	<code>real*real -> real</code>	6	Left
<code>=</code> <code><></code> <code><</code> <code><=</code>	<code>real*real -> bool</code>	4	Left

See also the libraries `Real` and `Math`

Reals

Type name : `real`

Values : `~27.0`, `0.0`, `1024.71717`, `23.4E~11`

Operations: (A few selected)

Operator	Type	Precedence	Association
<code>abs</code>	<code>real -> real</code>	Highest	
<code>*</code> <code>/</code>	<code>real*real -> real</code>	7	Left
<code>+</code> <code>-</code>	<code>real*real -> real</code>	6	Left
<code>=</code> <code><></code> <code><</code> <code><=</code>	<code>real*real -> bool</code>	4	Left

See also the libraries `Real` and `Math`

Some built-in operators are *overloaded*. `*` :

<code>real*real</code>	<code>-> real</code>
<code>int * int</code>	<code>-> int</code>

Default is `int`

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
<code>fun square x = x * x</code>	<code>int -> int</code>	Default

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
<code>fun square x = x * x</code>	<code>int -> int</code>	Default

A squaring function on reals: `square: real -> real`

Declaration	

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
<code>fun square x = x * x</code>	<code>int -> int</code>	Default

A squaring function on reals: `square: real -> real`

Declaration	
<code>fun square(x:real) = x * x</code>	Type the argument

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
<code>fun square x = x * x</code>	<code>int -> int</code>	Default

A squaring function on reals: `square: real -> real`

Declaration	
<code>fun square x:real = x * x</code>	Type the result

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
<code>fun square x = x * x</code>	<code>int -> int</code>	Default

A squaring function on reals: `square: real -> real`

Declaration	
<code>fun square x = x * x: real</code>	Type expression for the result

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
<code>fun square x = x * x</code>	<code>int -> int</code>	Default

A squaring function on reals: `square: real -> real`

Declaration	
<code>fun square x = x:real * x</code>	Type a variable

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
<code>fun square x = x * x</code>	<code>int -> int</code>	Default

A squaring function on reals: `square: real -> real`

Declaration	
<code>fun square x = x:real * x</code>	Type a variable

Choose any mixture of these possibilities

Characters

Type name `char`

Values `"a"`, `" "`, `"\"` (escape sequence for `"`)

Operator	Type	
<code>ord</code>	<code>char -> int</code>	ascii code of character
<code>chr</code>	<code>int -> char</code>	character for ascii code
<code>= < <= ...</code>	<code>char*char -> bool</code>	comparisons by ascii codes

Examples

```
- ord "a";  
> val it = 97 : int
```

```
- ord "A";  
> val it = 65 : int
```

```
- "a" < "A";  
> val it = false : bool;
```

```
- chr 88;  
> val it = "X" : char
```

Strings

Type name `string`

Values `"abcd"`, `" "`, `""`, `"123\"321"` (escape sequence for `"`)

Operator	Type	
<code>size</code>	<code>string -> int</code>	length of string
<code>^</code>	<code>string*string -> string</code>	concatenation
<code>= < <= ...</code>	<code>string*string -> bool</code>	comparisons
<code>Int.toString</code>	<code>int -> string</code>	conversions

Examples

```
- "auto" < "car";  
> val it = true : bool
```

```
- "abc" ^ "de";  
> val it = "abcde" : string
```

```
- size("abc" ^ "def");  
> val it = 6 : int
```

```
- Int.toString(6+18);  
> val it = "24" : string
```

Booleans

Type name `bool`

Values `false`, `true`

Operator	Type	
<code>not</code>	<code>bool -> bool</code>	negation

`not true = false`
`not false = true`

Expressions

`e1 andalso e2`

“conjunction $e_1 \wedge e_2$ ”

`e1 or else e2`

“disjunction $e_1 \vee e_2$ ”

— are lazily evaluated, e.g.

`1 < 2 or else 5 / 0 = 1`
`~> true`

Precedence: `andalso` has higher than `or else`

Tuples

An ordered collection of n values $(v_1, v_2, \dots, v_n)$ is called an n -tuple

Examples

<pre>- (); > val it = () : unit</pre>	0-tuple
<pre>- (3, false); > val it = (3, false) : int * bool</pre>	2-tuples (pairs)
<pre>- (1, 2, ("ab", true)); > val it = (1, 2, ("ab", true)) : ?</pre>	3-tuples (triples)

Selection Operation: $\#i(v_1, v_2, \dots, v_n) = v_i$. $\#2(1, 2, 3) = 2$

Equality defined componentwise

```
- (1, 2.0, true) = (2-1, 2.0*1.0, 1<2);  
> val it = true : bool
```

provided = is defined on components

Tuple patterns

Extract components of tuples

```
- val ((x,_), (_,y,_)) = ((1,true), ("a","b",false));  
> val x = 1 : int  
    val y = "b" : string
```

Pattern matching yields bindings

Restriction

```
- val (x,x) = (1,1);  
! Toplevel input:  
! val (x,x) = (1,1);  
!      ^  
! identifier is bound twice in a pattern
```

Infix functions

Directives: `infix d f` and `infixr d f`. `d` is the precedence of `f`

Example: exclusive-or

```
infix 0 xor      (* or just infix xor
                  -- lowest precedence *)
```

```
fun false xor true  = true
   | true  xor false = true
   | _     xor _     = false
```

type ?

```
- 1 < 2+3 xor 2.0 / 3.0 > 1.0;
> val it = true : bool
```

Infix status can be removed by `nonfix xor`

```
- xor(1 < 2+3, 2.0 / 3.0 > 1.0);
> val it = true : bool
```

Let expressions — `let dec in e end`

Bindings obtained from `dec` are valid only in `e`

Example: Solve $ax^2 + bx + c = 0$

```
type equation = real * real * real
type solution = real * real
```

```
exception Solve; (* declares an exception *)
```

```
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;
```

The type of `solve` is `equation -> solution`

`d` is declared once and used 3 times

readability, efficiency

Local declarations — `local $dec_2$ in $dec_2$ end`

Bindings obtained from dec_1 are valid only in dec_2

```
local
  fun disc(a,b,c) = b*b - 4.0*a*c
in
  exception Solve;

  fun hasTwoSolutions(a,b,c) = disc(a,b,c)>0.0
                                andalso a<>0.0;

  fun solve(a,b,c) =
    let val d = disc(a,b,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;
```

Example: Rational Numbers

Specification	Comment
type qnum = int * int	rational numbers
exception QDiv	division by zero
mkQ: int * int -> qnum	construction of rational numbers
++: qnum * qnum -> qnum	addition of rational numbers
--: qnum * qnum -> qnum	subtraction of rational numbers
**: qnum * qnum -> qnum	multiplication of rational numbers
//: qnum * qnum -> qnum	division of rational numbers
==: qnum * qnum -> bool	equality of rational numbers
toString: qnum -> string	String representation of rational numbers

Intended use

```
val q1 = mkQ(2, 3);
```

```
val q2 = mkQ(12, ~27);
```

```
val q3 = mkQ(~1, 4) ** q2 -- q1;
```

```
val q4 = q1 -- q2 // q3;
```

```
toString q4;
```

```
> val it = "~2/15" : string
```

Operators are infix with usual precedences

Representation: (a, b) , $b > 0$ and $\gcd(a, b) = 1$

Example $-\frac{12}{27}$ is represented by $(-4, 9)$

Greatest common divisor (Euclid's algorithm)

```
fun gcd(0, n) = n
  | gcd(m, n) = gcd(n mod m, m);
```

```
- gcd(12, 27);
> val it = 3 : int
```

Function to cancel common divisors:

```
fun canc(p, q) =
  let val sign = if p*q < 0 then ~1 else 1
      val ap = abs p
      val aq = abs q
      val d = gcd(ap, aq)
  in (sign * (ap div d), aq div d)
  end;
- canc(12, ~27);
> val it = (~4, 9) : int * int
```

Program for rational numbers

```
exception QDiv;
```

```
fun mkQ (_, 0) = raise QDiv  
  | mkQ pr    = canc pr;
```

```
infix 6 ++ --  
infix 7 ** //  
infix 4 ==
```

```
fun (a,b) ++ (c,d) = canc(a*d + b*c, b*d);  
fun (a,b) -- (c,d) = canc(a*d - b*c, b*d);  
fun (a,b) ** (c,d) = canc(a*c, b*d);  
fun (a,b) // (c,d) = (a,b) ** mkQ(d,c);  
fun (a,b) == (c,d) = (a,b) = (c,d);  
fun toString(p:int,q:int) =  
    (Int.toString p) ^ "/" ^ (Int.toString q);
```

Exercise

A time of day (ignore midnight) can be represented by a triple

(hour, minutes, f)

where *f* is either *am*, *noon* or *pm*.

- Declare a function to test whether one time of day comes before another. (Be careful, e.g. 12.15 pm comes before 1.15 pm.)

A solution

```
infix <<
```

```
fun (h1,m1,"am") << (h2,m2,"am") = h1<h2 orelse h1=h2 andalso m1<m2
  | (_,_,"am") << _ = true
  | (_,_,"noon") << (_,_,"pm") = true
  | (_,_,"noon") << _ = false
  | (12,m1,"pm") << (12,m2,"pm") = m1<m2
  | (12,_,"pm") << (_,_,"pm") = true
  | (h1,m1,"pm") << (h2,m2,"pm") = h1<h2 orelse h1=h2 andalso m1<m2
  | (_,_,"pm") << _ = false
```

Another solution

```
infix <<
```

```
fun t1 << t2 =
```

```
  let fun cvt(h,m,"am")    = (h,m)
        | cvt(_,_, "noon") = (12,0)
        | cvt(12,m, _)     = (12,m)
        | cvt(h,m,_)       = (12+h,m)
```

```
    fun lessThan((h1,m1),(h2,m2)) = h1<h2 orelse h1=h2 andalso m1<m2
```

```
  in lessThan(cvt t1, cvt t2)
```

```
end;
```