Lecture 2: The Basis of SML

Jean-Noël Monette

Functional Programming 1

September 9, 2013

Based on notes by Pierre Flener, Sven-Olof Nyström

Today

More on SML:

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Basic types

- `int`: integers
- `real`: real numbers
- `char`: characters
- `string`: character sequences
- `bool`: truth values (or: Booleans) `true` and `false`
- `unit`: only one possible value: `()`

More types

Some expressions may have a compound type.

- functions: e.g. `int -> int`
- tuples: e.g. `int * int`
- lists: e.g. `int list`

```
> ([abs ,~],( "cool" ,3.5));
val it = ([fn, fn], ("cool", 3.5)):
      (int -> int) list * (string * real)
```

We will see later in the course that one can even declare his own datatypes.

Type inference

- SML is *strongly typed*, meaning that all expressions have a well-defined type that can be determined statically (without running the program).
- It is (most of the time) not necessary to declare the type of an expression.
- The compilers are able to infer the type of all expressions.
- It is necessary to give the right operands to an operator or function.

```
fun double x = 2 * x; (* infers type int -> int *)
double 3.0; (* Error-Type error in function application. *)
2 * 3.0; (* Error-Type error in function application. *)
```

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Integer and real literals

Integers:

- In base 10: A sequence of digits (0-9), preceded by ~ for the negation. ~12
- Base 16: A sequence of digits (0-9, A-F) preceded by 0x or 0X. `~0xA17f`

Reals:

1. An optional ~.
2. A sequence of one or more digits.
3. One or both of:
   - A point and one or more digits.
   - `E` or e, optional ~, one or more digits.

0.0
~15.5e3
15E~2

Built-in operators

- SML has several built-in operators that work on the basic types.
- Several of them are overloaded for convenience. e.g.

```
2 + 3;
2.0 + 3.0;
```

- There is no explicit conversion.
- Operators are only a special case of functions.

Operators on Integers

op	:	*type*	*form*	*precedence*
$+$	:	int $\times$ int $\rightarrow$ int	infix	6
$-$	:	int $\times$ int $\rightarrow$ int	infix	6
$*$	:	int $\times$ int $\rightarrow$ int	infix	7
`div`	:	int $\times$ int $\rightarrow$ int	infix	7
`mod`	:	int $\times$ int $\rightarrow$ int	infix	7
$=$	:	int $\times$ int $\rightarrow$ bool *	infix	4
$<>$	:	int $\times$ int $\rightarrow$ bool *	infix	4
$<$	:	int $\times$ int $\rightarrow$ bool	infix	4
$<=$	:	int $\times$ int $\rightarrow$ bool	infix	4
$>$	:	int $\times$ int $\rightarrow$ bool	infix	4
$>=$	:	int $\times$ int $\rightarrow$ bool	infix	4
`~`	:	int $\rightarrow$ int	prefix	
`abs`	:	int $\rightarrow$ int	prefix	

(* the exact type will be defined later)

Infix operators associate to the left.

Operators on Reals

op	:	*type*	*form*	*precedence*
$+$	:	real $\times$ real $\rightarrow$ real	infix	6
$-$	:	real $\times$ real $\rightarrow$ real	infix	6
$*$	:	real $\times$ real $\rightarrow$ real	infix	7
$/$	:	real $\times$ real $\rightarrow$ real	infix	7
$<$, $<=$	:	real $\times$ real $\rightarrow$ bool	infix	4
$>$, $>=$	:	real $\times$ real $\rightarrow$ bool	infix	4
`~`	:	real $\rightarrow$ real	prefix	
`abs`	:	real $\rightarrow$ real	prefix	
`Math.sqrt`	:	real $\rightarrow$ real	prefix	
`Math.ln`	:	real $\rightarrow$ real	prefix	

(* the exact type will be defined later)

Infix operators associate to the left.

Remark the absence of = and <>.

Characters and Strings

- A *character* value is written as the symbol # immediately followed by the character enclosed in double-quotes "
- A *string* is a character sequence enclosed in double-quotes "
- Control characters can be included:
  end-of-line: `\n` double-quote: `\"` backslash: `\\`

```
#"a"
"Hello!\nGoodbye"
```

Operators on Characters and Strings

Let 'strchar × strchar' be 'char × char' or 'string × string'

op	:	*type*	*form*	*precedence*
=	:	strchar × strchar → bool *	infix	4
<>	:	strchar × strchar → bool *	infix	4
<	:	strchar × strchar → bool	infix	4
<=	:	strchar × strchar → bool	infix	4
>	:	strchar × strchar → bool	infix	4
>=	:	strchar × strchar → bool	infix	4
^	:	string × string → string	infix	6
`size`	:	string → int	prefix	

(* the exact type will be defined later)

Use of the *lexicographic order*, according to the ASCII code

Infix operators associate to the left.

Type Conversions

op	:	*type*
`real`	:	int → real
`ceil`	:	real → int
`floor`	:	real → int
`round`	:	real → int
`trunc`	:	real → int
`chr`	:	int → char
`ord`	:	char → int
`str`	:	char → string
`Int.toString`	:	int → string

Conversions of `chr` and `ord` are done according to the ASCII code.

Eager and Lazy Evaluation

- All the previous operators, as well as functions, are evaluated after all their operands are evaluated.
- This is called “eager evaluation”.
- Sometimes, it is however not necessary to evaluate all the operands to know the result of an operation.
- In such a case, we use “lazy evaluation”. The operands are only evaluated if needed.
- We will see several examples of lazy evaluation.

Operators on Booleans

op	:	*type*	*form*	*precedence*
`andalso`	:	bool $\times$ bool $\rightarrow$ bool	infix	3
`orelse`	:	bool $\times$ bool $\rightarrow$ bool	infix	2
`not`	:	bool $\rightarrow$ bool	prefix	
$=$	:	bool $\times$ bool $\rightarrow$ bool *	infix	4
$<>$	:	bool $\times$ bool $\rightarrow$ bool *	infix	4

(* the exact type will be defined later)

Infix operators associate to the left.

`and` and `or` are *not* boolean operators!

`andalso` and `orelse` are evaluated lazily.

Lazy evaluation: Examples

```
> ( 34 < 649 ) orelse ( Math.ln(12.4) * 3.4 > 12.0 ) ;
val it = true : bool
```

The second operand is *not* evaluated because the first operand evaluates to `true`.

```
> ( 34 < 649 ) orelse ( 0.0 / 0.0 > 999.9 ) ;
val it = true : bool
```

The second operand `( 0.0 / 0.0 > 999.9 )` is *not* evaluated, even though by itself it would lead to an error:

```
> ( 0.0 / 0.0 > 999.9 ) ;
! Uncaught exception: Div
```

if..then..else

`if B then E1 else E2`

- This is an expression, not a control structure.
- `B` must be a boolean expression.
- The type of `E1` and `E2` must be the same.
- `E1` is only evaluated if `B` evaluates to `true`.
- `E2` is only evaluated if `B` evaluates to `false`.
- There is no `if B then E` expression. What would be the value of the expression when `B` is false?
- if-then-else as a lower precedence than all the other operators.

Exercises

1. Express the following expressions as if-then-else expressions:
    1. `E orelse F`
    2. `E andalso F`
2. Evaluate by reduction the following expression:
   `if 1 + 2 < 4 then size (''sal'' ^ ''ut!'') else 4 div 2`

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Identifiers

- Alphanumeric identifiers
- Symbolic identifiers made from `+ - / * < > = ! @ # $ % ^ & ` ` ` `~ \ | ? :`
- Do not mix alphanumeric and symbolic characters
- The identifier `it` always has the result of the last unidentified expression evaluated.
- `3 +~ 2` is *different* from `3 + ~ 2`
  One must separate the symbols `+` and `~` with a space, otherwise they form a new symbolic identifier

Bindings and environments

- The execution of a declaration, say `val x = ` *expr*,
  creates a *binding*:
  the identifier `x` is *bound* to the value of the expression *expr*
- A collection of bindings is called an *environment*

```
> val sum = 24 ;
val sum = 24 : int
> val sum = 3.51 ;
val sum = 3.51 : real
```

Evaluation order

Evaluation and declaration from left to right

```
> val a = 1 ;
val a = 1 : int
> val b = 2 ;
val b = 2 : int
> val a = a+b val b = a+b ;
val a = 3 : int
val b = 5 : int
```

`and`: Simultaneous evaluation of the right-hand sides of the declarations

```
> val a = 1 val b = 2 ;
val a = 1 : int
val b = 2 : int
> val a = a+b and b = a+b ;
val a = 3 : int
val b = 3 : int
```

Identifiers are *not* variables

```
> val x = 10;
val x = 10: int
> fun addX y = x+y;
val addX = fn: int -> int
> addX 5;
val it = 15: int
> x = 100;
val it = false : bool
> val x = 100;
val x = 100: int
> addX 5;
val it = 15: int
```

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Tuples

- Group $n(\neq 1)$ values of possibly different types into *n-tuples* by enclosing them in parentheses, say: `(22>5, "abc", 123)`
- Particular cases of n-tuples: pairs (or: couples), triples, ...
- Careful: There are *no* 1-tuples in ML! `(1)` is just 1 in parentheses.
- Selector $\#i$ returns the i^{th} *component* of a tuple
- It is possible to have tuples of tuples
- The value `()` is the only 0-tuple, and it has type `unit`

Tuples: examples

```
> (2.3, 5) ;
val it = (2.3, 5) : real * int
```

Operator * here means the Cartesian product of types

```
> val bigTuple = ((2.3, 5), "two", (8, true)) ;
val bigTuple = ((2.3, 5), "two", (8, true)) :
               (real * int) * string * (int * bool)
> #3 bigTuple ;
val it = (8, true) : int * bool
> #2(#1 bigTuple) + #1(#3 bigTuple) ;
val it = 13 : int
```

Records

- A *record* is a generalised tuple where each component is identified by a *label* rather than by its integer position, and where curly braces are used instead of parentheses
- A record component is also called a *field*
- Selector $\#$*label* returns the value of the component identifed by *label*
- It is possible to have records of records
- n-tuples are just records with integer labels (when $n \neq 1$)

Records: examples

```
> {course = "FP", year = 2} ;
val it = {course = "FP", year = 2} : {course : string , year : int}
> #a {a=1, b="xyz"} ;
val it = 1 : int
> {a=1, b="xyz"} = {b="xyz", a=1} ;
val it = true : bool
> (1, "xyz") = ("xyz", 1) ;
Error-Type error in function application .
...
> {2=1, 1="xyz"} = ("xyz", 1) ;
val it = true : bool
```

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Functions

```
fun even x = x mod 2 = 0;
val even = fn x => x mod 2 = 0; (* anonymous function *)
fun odd x = not (even x); (* using another function *)
```

Evaluation:

```
> odd 17 orelse even 17;
-> (fn x=> not (even x)) 17 orelse even 17;
-> not (even 17) orelse even 17;
-> not ((fn x => x mod 2 = 0) 17) orelse even 17;
-> not (17 mod 2 = 0) orelse even 17;
-> not (1 = 0) orelse even 17;
-> not false orelse even 17;
-> true orelse even 17;
-> true;
```

Functions (cont.)

- Function application has the highest precedence, and is evaluated from left to right.
- Functions are values, just as integers, tuples, etc.
- They have a type (that can be inferred by the system).
  - If a type cannot be inferred from the context, then the default is that an overloaded operator symbol refers to the function on integers.
- Any identifier can be bound to them.
- They can be arguments or return values of other functions.

```
> fun even x = x mod 2 = 0;
val even = fn: int -> bool
> val plop = even;
val plop = fn: int -> bool
> plop 3;
val it = false: bool
> (fn x => x mod 2 = 1) 3;
val it = true: bool
```

Functions of several parameters

- In SML, functions only have one parameter, and one result.
- How can we implement e.g. the mathematical function $\max(a, b)$?

Functions of several parameters

- In SML, functions only have one parameter, and one result.
- How can we implement e.g. the mathematical function max(a, b)?
- Two ways:
  - The parameter is a tuple (a pair, here).

    > **fun** max (a,b) = **if** a > b **then** a **else** b;

  - Use curried functions.

    > **fun** max a b = **if** a > b **then** a **else** b;

- Does it look the same? Does that small difference matter?

Functions of several parameters (cont.)

```
> fun max1 (a,b) = if a > b then a else b;
val max1 = fn: int * int -> int
>val max1 = fn (a,b) => if a > b then a else b;
val max1 = fn: int * int -> int
> fun max2 a b = if a > b then a else b;
val max2 = fn: int -> int -> int
> val max2 = fn a => fn b => if a > b then a else b;
val max2 = fn: int -> int -> int
> val posOrZero = max2 0;
val posOrZero = fn: int -> int
> posOrZero 3;
val it = 3: int
> posOrZero ~3;
val it = 0: int
```

Currying

There is equivalence of the types of the following functions:

$$f : A \times B \to C$$

$$g : A \to (B \to C)$$

H.B. Curry (1958): $f\ (a, b)\ =\ g\ a\ b$
Currying = passing from the first form to the second form
Let a be an object of type A, and b an object of type B

- $f\ (a, b)$ is an object of type C, the application of the function f to the pair (a, b)
- $g\ a$ is an object of type $B \to C$: $g\ a$ is thus a *function*, the result of a function can thus also be a function!
- $(g\ a)\ b$ is an object of type C, the application of the function $g\ a$ to b
- Attention: $f\ (a, b)$ is different from $f\ a\ b$

Currying (cont.)

Every function on a Cartesian product can be curried:

$$g : A_1 \times A_2 \times \cdots \times A_n \to C$$

$$\downarrow$$

$$g : A_1 \to (A_2 \to \cdots \to (A_n \to C))$$
$$g : A_1 \to A_2 \to \cdots \to A_n \to C$$

The symbol $\to$ associates to the *right*.
Usefulness of currying:

- The rice tastes better ...
- Partial application of a function for getting other functions
- Easier design and usage of higher-order functions
  (functions with functional arguments)

Currying, examples

```
> fun greet word name = word ^ ", " ^ name ^ "!";
val greet = fn: string -> string -> string
> val greetEng = greet "Hello";
val greetEng = fn: string -> string
> val greetSwe = greet "Hej";
val greetSwe = fn: string -> string
> greetEng "Tjark";
val it = "Hello, Tjark!": string
> greetSwe "Kjell";
val it = "Hej, Kjell!": string
> greet "Salut" "Jean-Noel";
val it = "Salut, Jean-Noel!": string
```

More on functions

- Functions can return tuples when several results are needed.
- Functions can take or return the unit argument: `fun bof () = ();`
- The type of functions can be *polymorphic*:

```
> fun id x = x;
val id = fn: 'a -> 'a
```

The type variable `'a` can be instantiated to any type:

```
> id 5;
val it = 5: int
> id 3.5;
val it = 3.5: real
```

Polymorphism limitations

- When an arithmetic operator is encountered, if the type is not determined by another mean, the operator refers to integers.

```
> fun sqr x = x * x;
val sqr = fn: int -> int
> fun sqr x = (x:real) * x;
val sqr = fn: real -> real;
> fun sqr (x:real) = x * x;
val sqr = fn: real -> real;
> fun sqr x:real = x * x; (* (sqr x):real *)
val sqr = fn: real -> real;
> fun sqr x = x:real * x;
Error-Type constructor (x) has not been declared
   Found near x : real * x
```

Polymorphism limitations (cont.)

- There is a complication with type variables. Fortunately the compiler will warn you about it.

```
> fun id x = x;
val id = fn: 'a -> 'a
> val iidd = id id;
Warning-The type of (iidd) contains a free type variable.
   Setting it to a unique monotype.
val iidd = fn: _a -> _a
> iidd 1;
Error-Type error in function application.
   Function: iidd : _a -> _a
   Argument: 1 : int
   Reason:
      Can't unify int (*In Basis*) with
      _a (*Constructed from a free type variable.*)
      (Different type constructors)
```

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Specifying Functions

- *Function name* and *argument*
- *Type* of the function: types of the argument and result (can be infered by the compiler).
- *Pre-condition* on the argument:
  - If the pre-condition does not hold, then the function *may* return *any* result!
  - If the pre-condition does hold, then the function *must* return a result satisfying the post-condition!
- *Post-condition* on the result: its description and meaning
- *Side effects* (if any): printing of the result, ...
- *Examples* and *counter-examples* (if useful)

Specification: Example

```
(* PRE: n >= 0
   POST: sum_{0 <= i <= n}(i) *)
fun triangle n = ...
```

Beware: The post-condition and side effects *should* involve *all* the components of the argument

Role of well-chosen examples and counter-examples

In theory,they are redundant with the pre/post-conditions.
In practice:

- They often provide an intuitive understanding that no assertion or definition could achieve
- They often help eliminate risks of ambiguity in the pre/post-conditions by illustrating delicate issues
- If they contradict the pre/post-conditions, then we know that something is wrong somewhere!

```
(* PRE: (none)
POST: the largest integer m such that m <= n
EXAMPLES: floor(23.65) = 23, floor(~23.65) = ~24
COUNTER-EXAMPLE: floor(~23.65) <> ~23 *)
fun floor n = ...
```

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Pattern Matching

```
> val x = (18, true);
val x = (18,true): int * bool
> val (n,b) = x;
val n = 18: int
val b = true: bool
```

- The left-hand side of a value declaration is called a *pattern*.
- The value on the right must respect that pattern.
- An identifier can match anything.
- _ matches anything and has no name.

```
> val (n,_) = x;
val n = 18: int
> val (18,b) = x;
val b = true;
> val (17,b) = x;
Exception- Bind raised
```

Pattern Matching (cont.)

- Each identifier can occur at most once in a pattern (*linearity*)
- as introduces pattern alias for identifiers.

```
> val t = ((3.5, true), 4);
val t = ((3.5, true), 4): (real * bool) * int
> val (d as (a,b), c) = t;
val a = 3.5: real
val b = true: bool
val c = 4: int
val d = (3.5, true): real * bool
> val (d,c) = t;
> val ((a,b),c) = t;
> val s as (d,c) = t;
> val s as u as v = t;
> val (t,d) = t; (* t is bound to a different value after this *)
```

Pattern or not Pattern

Not every expression can be a pattern:

- Intuitively, only an irreducible expression can be a pattern.
  - literals, identifiers, constructors
- Real constants can not be involved in patterns.
- Function calls can not be involved in patterns.

```
val nil = x; (*OK*)
val niil = x; (*OK, but attention*)
val y::ys = x;(*OK*)
val a+1 = x; (*NOT OK*)
val abs y = x; (*NOT OK*)
val 0.0 =x; (*NOT OK*)
```

Case analysis

The case expression allows to match an expression against several patterns.

case Expr
  of Pat1 => Expr1
  | Pat2 => Expr2
  | ...
  | Patn => Exprn

- `case ...of ...` is an expression.
- `Expr1, ..., Exprn` must be of the *same* type.
- `Expr, Pat1, ..., Patn` must be of the *same* type.
- If `Pati` is selected, then *only* `Expri` is evaluated (lazy evaluation).

Case Analysis (cont.)

```
> case 17 mod 2
    of 0 => "even"
     | 1 => "odd";
Warning-Matches are not exhaustive.
val it = "odd": string
> case 17 mod 2
    of 0 => "even"
     | _ => "odd";
val it = "odd": string
```

- If the patterns are not exhaustive over their type, then there is an ML *warning* at the declaration.
- If none of the patterns is applicable during an evaluation, then there is an ML pattern-matching *exception*.
- The patterns need *not* be mutually exclusive: If several patterns are applicable, then ML selects the *first* applicable pattern.

Function with Case Analysis

```
fun fname Pat1 = Expr1
  | fname Pat2 = Expr2
  |  ...
  | fname Patn = Exprn

fn Pat1 => Expr1
 | Pat2 => Expr2
 |  ...
 | Patn => Exprn
```

Exercices

- How to express `if-then-else` as a `case-of` expression?
- Write a function of two integer arguments that returns the number of arguments that are equal to zero.

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Local declarations

- It is possible to declare variables locally with `let ... in ... end`.
- The variables declared between `let` and `in` only exist inside the expression.
- They hide any other declaration of the same identifier.

```
> val x = 5.5;
> val y = let
    val x = 1
  in
    x + 10
  end;
> (x,y);
val it = (5.5,11): real * int;
```

Local declarations (cont.)

- The let value is computed only once.
- Functions can also be local.

```
> fun discount unitPrice quantity =
    let val price = unitPrice * real(quantity)
    in if price < 100.0 then price else price * 0.95
    end;
> fun leapYear year =
    let fun isDivisible b = year mod b = 0
    in isDivisible 4 andalso
       (not (isDivisible 100) orelse isDivisible 400)
    end;
```

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

New Infix Operators

- It is possible to define new operators.
- `infix n id` makes `id` an infix operator with precedence `n`
- `rinfix n id` does the same but with right association.
- `nonfix id` makes return to the prefix form.
- Beware: operators take a pair of argument (no curried form).
- Still possible to call an infix operator in prefix form with op.

```
> fun x (a,b) = a*b;
> infix 5 x;
> 2 x 4;
val it = 8: int
> op + (2,4);
val it = 6: int
```

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Side Effects

Like most functional languages, ML has some functions with side effects:

- Input / output
- Variables (in the imperative-programming sense)
- Explicit references
- Tables (in the imperative-programming sense)
- Imperative-programming-style control structures (sequence, iteration, ...)

In these lectures we only consider printing.

The print function

Type: `print` string → unit

Side effect: The argument of `print` is printed on the screen

```
> fun welcome name = print ("Hello, " ^ name ^ "!\n") ;
val welcome = fn : string -> unit
> welcome "world" ;
Hello, world!
val it = () : unit
```

Sequential composition

Sequential composition is necessary when one wants to print intermediate results.

```
fun relError a b =
   let val diff = abs (a-b)
   in
     ( print (Real.toString diff) ;
       print "\n" ;
       diff / a )
   end;
```

- Sequential composition is an *expression* of the form
  ($Expr_1$; $Expr_2$; ... ; $Expr_n$)
- The value of this expression is the value of $Expr_n$.

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. **Exceptions**
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Exceptions

Execution can be interrupted immediately upon an error

```
> 1 div 0;
Exception- Div raised
```

Exceptions can be caught:

```
> 1 div 0 handle Div => 42;
val it = 42: int
```

You can declare and throw your own exceptions:

```
exception errorDiv;
fun safeDiv a b =
    if b = 0 then raise errorDiv
    else a div b;
```

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Modules

- Modules group related functionalities together.
- SML defines a basic library in standard modules.
- Access a function inside a module by typing the name of the module followed by the name of the function.
- Some functions are available at the top-level.

```
Int.toString
Int.+
Int.abs
Real.Math.sqrt
```

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Functional languages vs. imperative languages

Example: greatest common divisor of natural numbers We know from Euclid that:

$$\begin{array}{rcll} gcd(0,n) & = & n & \text{if } n > 0 \\ gcd(m,n) & = & gcd(n \bmod m, m) & \text{if } m > 0 \end{array}$$

In an imperative language (C)

```
int gcd(int m, int n) {

/* PRE: m, n >= 0 and m+n > 0
   POST: the greatest common divisor of m and n
*/

  int a=m, b=n, prevA;

  /* INVARIANT: gcd(m,n) = gcd(a,b) */
  while (a != 0) {
    prevA = a;
    a = b % a;
    b = prevA;
  }
  return b;
}
```

In a functional language (SML)

```
(* PRE: m,n >= 0 and m+n > 0
   POST: the greatest common divisor of m,n *)
fun gcd1 (m, n) =
    if  m = 0 then n
    else  gcd1 (n mod m, m)
fun gcd2 (0, n) = n
  | gcd2 (m, n) = gcd2 (n mod m, m)
```

Features of imperative programs

- Close to the hardware
  - Sequence of instructions
  - Modification of variables (memory cells)
  - Test of variables (memory cells)
  - Transformation of states (automata)
- Construction of programs
  - Describe what has to be computed
  - Organise the sequence of computations into steps
  - Organise the variables
- Correctness
  - Specifications by pre/post-conditions
  - Loop invariants
  - Symbolic execution
- Expressions: `f(z) + x / 2` can be different from `x / 2 + f(z)` namely when `f` modifies the value of `x` (by side effect)
- Variables: The assignment `x = x + 1` modifies a memory cell as a side effect

Features of functional programs

- Execution by evaluation of expressions
- Basic tools: expressions and recursion
- Handling of *values* (rather than states)
- The expression $e_1 + e_2$ *always* has the same value as $e_2 + e_1$
- Identifiers
  - Value via a *declaration*
  - No assignment, no "modification"
- Recursion: series of values from recursive calls
- Functions are "first-class", they can be arguments to other functions.

Table of Contents

1. Types and type inference
2. Literals and built-in operators
3. Value declarations
4. Tuples and record
5. Functions
6. Specifications of functions
7. Pattern Matching
8. Local declarations
9. New operators
10. Side Effects
11. Exceptions
12. Modules
13. Comparison with Imperative Programming
14. Exercises

Exercices

- Write a function `h` that takes a time as a pair (hour, minutes) and returns the number of minutes elapsed since midnight. Raise an exception if the pair does not represent a legit time. Make this function infix to be able to call it like `16 h 40` (should return 1000).
- Write a function that returns a boolean telling if a date (year, month, day) exists or not.
- Write a function that takes an integer between 1 and 7 and returns the day of the week (1 is Monday, 7 is Sunday).
- Write a function of an integer that returns 0, -1 or 1 if the argument is respectively equal to, smaller or larger than zero. Use this to reimplement `abs`.
- Write a function `surround tag content`, whose result is `<tag>content</tag>`, with the parameters names replaced. Use this function to create the functions `makeParagraph content` (`<p>` in html), and `makeHeader n content` (`<h1>...<h6>` in html).