# AU Compilation – Introduction to SML

This document is a supplement to the lecture on introduction to SML/NJ. It is recommended you work through this document interactively. Please report any errors or inconsistencies to `aslan@cs.au.dk`.

## 1 GETTING STARTED

### 1.1 GETTING IN AND OUT OF SML INTERACTIVE SESSION

We start our interactive SML session by typing in the shell

```
> sml
Standard ML of New Jersey v110.82 [built: Tue Jan  9 20:54:24 2018]
```

The SML/NJ provides us with a REPL (read-eval-print loop) environment, where we can evaluate SML expressions:

```
- 1 + 1;
val it = 2 : int
```

This is an explicit declaration. It is treated as implicit declaration of a standard variable `it`. We can examine its content by evaluating `it`

```
- it;
val it = 2 : int
```

To exit the shell, we type `^D`.

```
- ^D
>
```

Use `rlwrap` program to improve your experience of the SML interactive session.

```
> rlwrap sml
-
```

We suggest you edit your `.bashrc` file and add line `alias sml=rlwrap sml` to that file.

**Equality concerns** When comparing two values for equality, SML/NJ reports a warning if the values have polymorphic types:

```
- fn (x, y) => x = y;
stdIn:1.21 Warning: calling polyEqual
val it = fn : ''a * ''a -> bool
```

These equality warning can be suppressed using a dedicated option.

```
> rlwrap sml -Ccontrol.poly-eq-warn=false
...
- fn (x, y) => x = y;
val it = fn : ''a * ''a -> bool
- ^D
```

Not all types can be compared for equality:

```
- 42.0 = 42.0;
stdIn:1.2-1.13 Error: operator and operand don't
agree [equality type required]
  operator domain: ''Z * ''Z
  operand:         real * real
  in expression:
    42.0 = 42.0
```

To compare real values for equality, we need to use a dedicated comparison operator from `Real` library.

```
- Real.== (42.0, 42.0);
val it = true : bool
```

# 2 BASIC EXPRESSIONS

## 2.1 EXPRESSIONS AND DECLARATIONS

We can use the REPL to evaluate SML expressions. We can bind the results of the evaluation explicitly using `val` keyword. Here are a few examples:

```
- 3 + 4;
val it = 7 : int

- val res = 3 + 4;
val res = 7 : int

- res * 2 ;
val it = 14 : int

- res div 2: int;
val it = 3 : int

- (res: int) div (2 : int);
val it = 3 : int
```

**Arithmetic and logical expressions** Here are some more examples that illustrate mathematical and logical operations.

```
- val y = Math.sqrt 2.0;
[autoloading]
[library $SMLNJ-BASIS/basis.cm is stable]
[autoloading done]
val y = 1.41421356237 : real

- val large = 10 < res;
val large = false : bool

- y > 0.0 andalso 1.0/y > 7.0;
val it = false : bool
```

**String values and string operations** String values have type `string`. To concatenate two strings we use the caret operator ^. The size and substrings can be extracted using functions `size` and `substring`.

```
- val title = "Mr";
val title = "Mr" : string

- val name = "Tambourine Man";
val name = "Tambourine Man" : string
```

```
- val junkmail = "Dear " ^ title ^ " " ^ name ^ ", you have won $$$!";
val junkmail = "Dear Mr Tambourine Man, you have won $$$!" : string

- val n = size junkmail;
val n = 41 : int

- substring (junkmail, 0, 22);
val it = "Dear Mr Tambourine Man" : string
```

For other string operations, check out the documentation for `String` module at `http://sml-family.org/Basis/string.html`. You may need to prefix the calls to these with the explicit mention of `String` module.

```
- String.isPrefix "Hello" "HelloWorld" ;
val it = true : bool
```

**Conditionals** The conditional expressions in SML have the form `if e0 then e1 else e2`, where `e0` is the boolean, and `e1` and `e2` are the bodies of the then and else-branches respectively. Note that `e1` and `e2` must have the same type, and that is the type of the whole `if` expression. This also means that both branches need to be provided.

```
- if 2 > 1 then "Hi" else "Bye";
val it = "Hi" : string
```

**Bindings and their scope** Expressions evaluated in REPL are bound globally. We can bind results of evaluating some expressions using `let ... in ... end` keywords. In the examples below, the text in between `(*` and `*)` is comments and need not be typed in.

```
- let val res = 7 in res div 2 end        (* local binding *)
val it = 3 : int

- val x = 5;                              (* global binding that of x to 5: int *)
val x = 5 : int

- let val x = 3 < 4                       (* new x is true: bool *)
= in if x then 117 else 118               (* uses the new x *)
= end;
val it = 117 : int

- x;                                      (* old x is still 5: int *)
val it = 5 : int
```

## 2.2 TYPE ERRORS

Every expression in SML needs to be well-typed. SML REPL performs rigorous type checking of the provided expressions before evaluating them (the same applies to the compiler). When type checking fails, we get an error message:

```
- Math.sqrt "Hello";
stdIn:1.2-8.1 Error: operator and operand don't agree [tycon mismatch]
  operator domain: real
  operand:         string
  in expression:
    Math.sqrt "hello"
```

Taking a square root of a string value is clearly meaningless. The way to read the above message is that there is a mismatch between the expected type (`real`) and the type of the provided operand (`string`). Generally, when you encounter such messages, keep in mind the following mapping:

**operator domain** = expected (required) type
**operand** = actual type

# 3 FUNCTIONS

## 3.1 FUNCTION DECLARATIONS

Functions are declared using `fun` keyword:

```
- fun circleArea r = Math.pi * r * r;
val circleArea = fn : real -> real

- val a = circleArea 10.0;
val a = 314.159265359 : real

- fun doubl x = 2.0 * x ;
val doubl = fn : real -> real
```

In the above, `real -> real` is the *type* of the function. The first occurrence of `real`, to the left of the arrow, is the type of the function's input. The second occurrence of `real`, to the right of the arrow, is the type of the function's result.

Naturally, we can apply the function `doubl` several times.

```
- doubl (doubl 3.5);
val it = 14.0 : real
```

Here is an example of a function that takes two arguments:

```
- fun junkmail title name = "Dear " ^ title ^ " " ^ name ^ ", You have won $$$!";
val junkmail = fn : string -> string -> string

- junkmail "Mr" "Tambourine Man";
val it = "Dear Mr Tambourine Man, You have won $$$!" : string
```

**Example: swapping arguments and partial application** We define a function that calls function `junkmail` defined above, but with the arguments swapped.

```
- fun swapped x y = junkmail y x;
val swapped = fn : string -> string -> string
```

Functions with multiple arguments may be *partially* applied. The result is a function that takes the remaining arguments.

```
- val f = swapped "Mr";
val f = fn : string -> string

- f "Alice";
val it = "Dear Alice Mr, You have won $$$!" : string
```

## 3.2 FUNCTION COMPOSITION

We can compose functions using the function composition operator `o`. Consider the following two declarations.

```
- fun half x = x / 2.0;
val half = fn : real -> real

- half (Math.sqrt 2.0);
val it = 0.707106781187 : real
```

The following is the same as above:

```
- (half o Math.sqrt) 2.0;
val it = 0.707106781187 : real
```

Function composition is a function itself:

```
- half o Math.sqrt;
val it = fn : real -> real
```

### 3.3 FIRST-CLASS FUNCTIONS, RECURSION, AND ANONYMOUS FUNCTIONS

In SML, functions are "first-class values"[1]. This means that functions can be passed as arguments, returned as results of computations, or get bound/assigned to variables.

In the following, because `Math.sqrt` function is a value, we can bind `also_sqrt` to that value.

```
- val also_sqrt = Math.sqrt;
val also_sqrt = fn : real -> real

- also_sqrt 2.0;
val it = 1.41421356237 : real
```

**Recursive function declarations** Functions defined via `fun` keyword can be recursively called.

```
- fun fac n = if n = 0 then 1 else n * fac (n - 1);
val fac = fn : int -> int

- fac 10;
val it = 3628800 : int
```

**Mutually recursive functions** In general, functions can only be used after they have been declared. To declare two or more *mutually* recursive functions, use `fun ... and ...` syntax.

```
- fun even n = if n = 0 then true else odd (n - 1)
= and odd n = if n = 0 then false else even (n - 1);
val even = fn : int -> bool
val odd = fn : int -> bool
```

**Anonymous functions** Not every function needs a name. We define function values using `fn` keyword. Such values are typically called anonymous functions.

```
- fn x => x * 10;
val it = fn : int -> int
```

Here is an example of applying the anonymously declared function

```
- (fn x => x * 10) 42;
val it = 420 : int
```

Anonymous functions can be `val`-bound just as any other value.

```
- val g = fn x => x * 10;
val g = fn : int -> int
- g 42;
val it = 420 : int
```

Note, however, that anonymous functions cannot be called recursively, even if they are `val`-bound.

```
val fac' = fn n => if n = 0 then 1 else n * fac' (n -1);
stdIn:14.27-14.31 Error: unbound variable or constructor: fac'
```

[1] In programming languages terminology, the term "first-class citizens" is used interchangeably with "first-class values"

# 4 TYPE CONSTRAINTS, PAIRS AND TUPLES

## 4.1 TYPE CONSTRAINTS

In most examples so far, we have avoided specifying exact types. Indeed, SML's type system *infers* types of expressions for us. While the inference mechanism is very powerful, on occasion, it is helpful and sometimes necessary to constrain the types. For example, in the following, we constrain the type of the argument `x` to reals.

```
- fun isLarge (x : real): bool = 10.0 < x ;
val isLarge = fn : real -> bool

- isLarge 89.0;
val it = true : bool
```

Type constraints can be applied to values as well as expressions.

```
- val x = 6;
val x = 6 : int

- (x : int) div ( 2: int) ;
val it = 3 : int
```

To constrain function types, we use the arrow `->` notation.

```
- (op div: int * int -> int) (x : int, 2 :int);
val it = 3 : int
```

Note here that the `op` keyword allow us to use an infix function as a function that takes a tuple (more on tuples below).

## 4.2 PAIRS AND TUPLES

The following examples demonstrate how to construct pairs and tuples, and how to project values out of them.

```
- val p = (2,3);
val p = (2,3) : int * int

- val w = (2, true, (7, false), "blah");
val w = (2,true,(7,false),"blah") : int * bool * (int * bool) * string

- #2 w;                          (* projection *)
val it = true : bool

- #4 w;
val it = "blah" : string

- #1 (#3 w);
val it = 7 : int

- fun add (x,y) = x + y ;  (* "argument tuple" is a single argument*)
val add = fn : int * int -> int

- add (2,3);
val it = 5 : int
-
- val noon = (12,0);       (* tuples are values  *)
val noon = (12,0) : int * int

- val talk = (09,00);
```

```
val talk = (9,0) : int * int

- print "Hello\n";
Hello
val it = () : unit         (* the empty tuple: no info *)

- fun earlier ((h1, m1), (h2, m2)) =
=   h1 < h2 orelse (h1 = h2 andalso m1 < m2 );
val earlier = fn : (int * int) * (int * int) -> bool

- earlier noon talk;
stdIn:166.1-166.18 Error: operator and operand don't agree [tycon mismatch]
  operator domain: (int * int) * (int * int)
  operand:         int * int
  in expression:
    earlier noon
- earlier (noon, talk);
val it = false : bool
```

### 4.3 EXAMPLE FROM THE LECTURE

The following is a slightly modified version of the example from the lecture.

```
- fun mapPair (f: int -> int) (x :int, y:int) = (f x, f y);
val mapPair = fn : (int -> int) -> int * int -> int * int
- fun increment x = x + 1;
val increment = fn : int -> int
- mapPair increment (10, 20);
val it = (11,21) : int * int
- mapPair (increment o increment) (10, 20);
val it = (12,22) : int * int
- exception BadArgument;
exception BadArgument
- fun iterate (f: int -> int) n =
=     if n <= 0 then raise BadArgument
=     else
=       if n = 1 then f
=       else let val g = iterate f (n - 1) (* recursive call *)
=            in f o g                      (* composition    *)
=            end;
val iterate = fn : (int -> int) -> int -> int -> int
(iterate increment 10) 9;
val it = 19 : int
-
- mapPair (iterate increment 10) (10,20);
val it = (20,30) : int * int
```

# 5 BASIC PATTERN MATCHING

A neat feature of SML that allows for writing concise code is *pattern matching*. Pattern matching can be used at the level of function definitions on in special `case` expressions. Let us rewrite the definition of the factorial function in a way that illustrates these.

Version one, using pattern matching at the level of function definitions. Observe how in the second line of the `fac`, we binds the argument of the function to variable `n`.

```
- fun fac 0 = 1                    (* use first case that matches *)
```

```
=   | fac n = n * fac (n - 1);
val fac = fn : int -> int

- fac 10 ;
val it = 3628800 : int
```

Version two, using `case` keyword

```
- fun fac n =                      (* same meaning *)
=   case n of
=     0 => 1
=   | _ => n * fac (n - 1);
val fac = fn : int -> int

- fac 10 ;
val it = 3628800 : int
```

Patterns are matched in the order they are declared in the code. In the last example, note the use of the *wildcard* pattern `_` that matches anything. Because of the order in which patterns are matched, if the wildcard pattern is used, it makes no sense to have other patterns after it.

**Patterns and bindings** Observe that patterns are just that – the patterns. This means that if a variable is used in a pattern it is automatically rebound. More crucially (and this a common a source of confusion for beginners!), the value of the variable prior to the pattern is irrelevant and should not be relied upon. Pay attention to the result of evaluating the following expression.

```
- let val zero = 0 in
= case (1,1) of
= (zero, 1) => "first match"
= | (_, _) => "second match"
= end;
val it = "first match" : string
```

# 6 LISTS, RECORDS, EXCEPTIONS

```
- val x1 = [7,9,13];
val x1 = [7,9,13] : int list

- val x2 = 7:: 9 :: 13 :: [];
val x2 = [7,9,13] : int list

- val equal = ( x1 = x2);
val equal = true : bool

- val ss = ["Dear ", title, "", name, ", You have won $$$!"];
val ss = ["Dear ","Mr","","Tambourine Man",", You have won $$$!"]
  : string list

- val junkmail2 = String.concat ss;
val junkmail2 = "Dear MrTambourine Man, You have won $$$!" : string
```

**Functions on lists**

```
- fun len [] = 0
=   | len (x::xr) = 1 + len xr;
val len = fn : ''a list -> int   (* ''a means that the list is polymorphic, i.e., we *)
                                 (*     can apply this function to lists of any  *)
```

```
- val x2len = len x2;
val x2len = 3 : int

- val sslen = len ss;
val sslen = 5 : int

- val x3 = [47,11];
val x3 = [47,11] : int list

- val x1x3 = x1@x3;        (* list concatenation *)
val x1x3 = [7,9,13,47,11] : int list

- List.hd [1,2,3];         (* returns the head element of the list *)
val it = 1 : int

- List.tl [1,2,3];         (* returns the tail of the list *)
val it = [2,3] : int list

- val z = ListPair.zip ([1,2,3], ["a", "b", "c"]);
val z = [(1,"a"),(2,"b"),(3,"c")] : (int * string) list

- ListPair.unzip z;
val it = ([1,2,3],["a","b","c"]) : int list * string list
```

### 6.1 RECORDS

```
- val x = { name = "Alice", phone = 777 };
val x = {name="Alice",phone=777} : {name:string, phone:int}
- #name x;  (* projection *)
val it = "Alice" : string
-
- #phone x;
val it = 777 : int

(* pattern matching on the record fields *)
- fun show {name, phone} = name ^ " has extension " ^ Int.toString phone;
val show = fn : {name:string, phone:int} -> string
-
- show x;
val it = "Alice has extension 777" : string
```

### 6.2 EXCEPTIONS

```
- exception IllegalHour;
exception IllegalHour
- fun mins h = if h < 0 orelse h > 23 then raise IllegalHour else h * 60;
val mins = fn : int -> int
- mins 25; (* run this one in the REPL directly *)

uncaught exception IllegalHour
  raised at: stdIn:18.48-18.59
```

```
- datatype person = Student of string
=                  | Teacher of string * int;
datatype person = Student of string | Teacher of string * int
- val people = [ Student "Alice"
=               , Teacher ("Bob", 444) ];
val people = [Student "Alice",Teacher ("Bob",444)] : person list
- fun getphone (Teacher (_, p)) = p
=   | getphone (Student name) = raise Fail "no phone";
val getphone = fn : person -> int
-
- getphone (List.hd people); (* run this one in the REPL directly *)
uncaught exception Fail [Fail: no phone]
  raised at: stdIn:26.37-26.52
```

**The option datatype**

```
- datatype intopt =
=          SOME of int
=        | NONE;
datatype intopt = NONE | SOME of int
```

A different version of `getphone` that uses the option datatype instead of the exception.

```
- fun getphone (Teacher (name, phone)) = SOME phone
=   | getphone (Student name) = NONE;
val getphone = fn : person -> intopt

- getphone (Student "Alice");
val it = NONE : intopt
```