

## *2.3: Introduction to Forlan*

The Forlan toolset is implemented as a set of Standard ML (SML) modules. It's used interactively. In fact, a Forlan session is nothing more than a Standard ML session in which the Forlan modules are available.

Instructions for installing and running Forlan on machines running Linux, macOS and Windows can be found on the Forlan website: <https://alleystoughton.us/forlan/>.

We begin this section by giving a quick introduction to SML.

We then show how symbols, strings, finite sets of symbols and strings, and finite relations on symbols can be manipulated using Forlan.

## *Invoking Forlan*

To invoke Forlan, type the command `forlan` to your shell (command processor):

```
% forlan
Forlan Version m (based on Standard ML of New Jersey
Version n)
val it = () : unit
-
```

SML's prompt is “-”. To exit SML, type *CTRL-d* under Linux and macOS, and *CTRL-z* under Windows. To interrupt back to the SML top-level, type *CTRL-c*.

## *Invoking Forlan (Cont.)*

On Windows, you may find it more convenient to invoke Forlan by double-clicking on the Forlan icon.

Actually, a much more flexible and satisfying way of running Forlan is as a subprocess of the Emacs text editor. See the Forlan website for information about how to do this.

## *Evaluating Expressions*

The simplest way of using SML is as a calculator:

```
- 4 + 5;  
val it = 9 : int  
- it * it;  
val it = 81 : int  
- it - 1;  
val it = 80 : int  
- 5 div 2;  
val it = 2 : int  
- 5 mod 2;  
val it = 1 : int  
- ~4 + 2;  
val it = ~2 : int
```

SML responds to each expression by printing its value and type (`int` is the type of integers), and noting that the expression's value has been bound to the identifier `it`. Expressions must be terminated with semicolons. Negative numbers begin with `~`.

## More Types

In addition to the type `int` of integers, SML has types `string` and `bool`, product types  $t_1 * \dots * t_n$ , and list types `t list`.

```
- "hello" ^ " " ^ "there";
val it = "hello there" : string
- not true;
val it = false : bool
- true andalso (false orelse true);
val it = true : bool
- if 5 < 7 then "hello" else "bye";
val it = "hello" : string
- (3 + 1, 4 = 4, "a" ^ "b");
val it = (4,true,"ab") : int * bool * string
- [1, 3, 5] @ [7, 9, 11];
val it = [1,3,5,7,9,11] : int list
- rev it;
val it = [11,9,7,5,3,1] : int list
- length it;
val it = 6 : int
```

## *More Types (Cont.)*

```
- null[];  
val it = true : bool  
- null[1, 2];  
val it = false : bool  
- hd[1, 2, 3];  
val it = 1 : int  
- tl[1, 2, 3];  
val it = [2,3] : int list
```

## More Types (Cont.)

It also has option types `t option`:

```
- NONE;  
val it = NONE : 'a option  
- SOME 3;  
val it = SOME 3 : int option  
- SOME true;  
val it = SOME true : bool option
```

`NONE` is an example of a *polymorphic* value. It has all of the types that can be formed by instantiating the type variable `'a` with a type: `int option`, `bool option`, etc.

## *Value Declarations*

In SML, it is possible to bind the value of an expression to an identifier using a value declaration:

```
- val x = 3 + 4;  
val x = 7 : int  
- val y = x + 1;  
val y = 8 : int
```

One can even give names to the components of a tuple, or give a name to the data of a non-**NONE** optional value:

```
- val (x, y, z) = (3 + 1, 4 = 4, "a" ^ "b");  
val x = 4 : int  
val y = true : bool  
val z = "ab" : string  
- val SOME n = SOME(4 * 25);  
val n = 100 : int
```



## *Let Expressions*

One can use a `let` expression to carry out some declarations in a local environment, evaluate an expression in that environment, and yield the result of that evaluation:

```
- val x = 3;
val x = 3 : int
- val z = 10;
val z = 10 : int
- let val x = 4 * 5
=      val y = x * z
= in (x, y, x + y) end;
val it = (20,200,220) : int * int * int
- x;
val it = 3 : int
```

When a declaration or expression spans more than one line, SML prints its secondary prompt, `=`, on all of the lines except for the first one. SML doesn't process a declaration or expression until it is terminated with a semicolon.

## Function Declarations

One can declare functions, and apply those functions to arguments:

```
- fun f n = n + 1;
val f = fn : int -> int
- f(4 + 5);
val it = 10 : int
- fun g(x, y) = (x ^ y, y ^ x);
val g = fn : string * string -> string * string
- val (u, v) = g("a", "b");
val u = "ab" : string
val v = "ba" : string
```

The function `f` maps its input `n` to its output `n + 1`. All function values are printed as `fn`. A type `t1 -> t2` is the type of all functions taking arguments of type `t1` and producing results of type `t2`. Note that SML infers the types of functions, and that the type operator `*` has higher precedence than the operator `->`.

## *Anonymous Functions*

Forlan has *anonymous* functions, which may also be given names using value declarations:

```
- (fn x => x + 1)(3 + 4);  
val it = 8 : int  
- val f = fn x => x + 1;  
val f = fn : int -> int  
- f(3 + 4);  
val it = 8 : int
```

## *Functions as Data*

Functions are data: they may be passed to functions, returned from functions (a function that returns a function is called *curried*), be components of tuples or lists, etc. For example,

```
val map : ('a -> 'b) -> 'a list -> 'b list
```

is a polymorphic, curried function. The type operator `->` associates to the right, so that `map`'s type is

```
val map : ('a -> 'b) -> ('a list -> 'b list)
```

`map` takes in a function  $f$  of type `'a -> 'b`, and returns a function that when called with a list of elements of type `'a`, transforms each element using  $f$ , forming a list of elements of type `'b`.

## *Functions as Data (Cont.)*

```
- val f = map(fn x => x + 1);  
val f = fn : int list -> int list  
- f[2, 4, 6];  
val it = [3,5,7] : int list  
- f[~2, ~1, 0];  
val it = [~1,0,1] : int list  
- map (fn x => x mod 2 = 1) [3, 4, 5, 6, 7];  
val it = [true,false,true,false,true] : bool list
```

In the last use of `map`, we are using the fact that function application associates to the left, so that  $f \times y$  means  $(f x)y$ , i.e., apply  $f$  to  $x$ , and then apply the resulting function to  $y$ .

## *Recursive Functions*

It's also possible to declare recursive functions, like the factorial function:

```
- fun fact n =  
=       if n = 0  
=       then 1  
=       else n * fact(n - 1);  
val fact = fn : int -> int  
- fact 4;  
val it = 24 : int
```

## *Loading the Contents of Files*

One can load the contents of a file into SML using the function

```
val use : string -> unit
```

The type `unit` has the single element `()`. For example, if the file `fact.sml` contains the declaration of the factorial function, then this declaration can be loaded into the system as follows:

```
- use "fact.sml";  
[opening fact.sml]  
val fact = fn : int -> int  
val it = () : unit  
- fact 4;  
val it = 24 : int
```

## Pattern Matching

We can define functions using pattern matching. E.g., we could (inefficiently) define the list reversal function like this:

```
- fun rev nil          = nil
  =   | rev (x :: xs) = rev xs @ [x];
val rev = fn : 'a list -> 'a list
```

Calling `rev` with the empty list (`[]` or `nil`) will result in the empty list being returned. And calling it with a nonempty list will temporarily bind `x` to the list's head, bind `xs` to its tail, and then evaluate the expression `rev xs @ [x]`, making a recursive call to `rev xs`, and then returning the result of appending the result of this call and `[x]`. The official definition of `rev` is more efficient; see the book.



## *Recursive Datatypes*

We can define recursive datatypes, and define functions by structural recursion on recursive datatypes. E.g., here's how we can define the datatype of labeled binary trees:

```
- datatype ('a, 'b) tree =  
=   Leaf of 'b  
=   | Node of 'a * ('a, 'b) tree * ('a, 'b) tree;  
datatype ('a,'b) tree  
  = Leaf of 'b  
  | Node of 'a * ('a,'b) tree * ('a,'b) tree  
- Leaf;  
val it = fn : 'a -> ('b,'a) tree  
- Node;  
val it = fn  
  : 'a * ('a,'b) tree * ('a,'b) tree -> ('a,'b) tree
```

## *Recursive Datatypes (Cont.)*

```
- val tr =  
=       Node(true,  
=         Node(false, Leaf 7, Leaf ~1),  
=         Leaf 8);  
val tr =  
  Node (true,Node (false,Leaf 7,Leaf ~1),Leaf 8)  
  : (bool,int) tree
```

## *Recursive Datatypes (Cont.)*

Then we can define a function for reversing a tree, and apply it to `tr`:

```
- fun revTree (Leaf n)                = Leaf n
=   | revTree (Node(m, tr1, tr2)) =
=       Node(m, revTree tr2, revTree tr1);
val revTree = fn : ('a,'b) tree -> ('a,'b) tree
- revTree tr;
val it =
  Node (true,Leaf 8,Node (false,Leaf ~1,Leaf 7))
  : (bool,int) tree
- revTree it;
val it =
  Node (true,Node (false,Leaf 7,Leaf ~1),Leaf 8)
  : (bool,int) tree
```

## *Symbols*

The Forlan module `Sym` defines the abstract type `sym` of symbols, as well as some functions for processing symbols, including:

```
val input  : string -> sym
val output : string * sym -> unit
val equal  : sym * sym -> bool
```

These functions behave as follows:

- `input fil` reads a symbol from file `fil`; if `fil = ""`, then the symbol is read from the standard input;
- `output(fil, a)` writes the symbol `a` to the file `fil`; if `fil = ""`, then the string is written to the standard output;
- `equal` tests whether two symbols are equal.

## *Symbols (Cont.)*

The type `sym` is bound in the top-level environment; on the other hand, one must write `Sym.f` to select the function `f` of module `Sym`.  
Whitespace characters are ignored by Forlan's input routines.  
Interactive input is terminated by a line consisting of a single `."` (dot, period). Forlan's prompt is `@`.

## *Symbols (Cont.)*

The module `Sym` also provides the functions

```
val fromString : string -> sym
val toString   : sym -> string
```

where

- `fromString` is like `input`, except that it takes its input from a string;
- `toString` is like `output`, except that it writes its output to a string.

In the future, whenever a module/type has `input` and `output` functions, you may assume that it also has `fromString` and `toString` functions.

## *Symbols (Cont.)*

Here are some example uses of the functions of `Sym`:

```
- val a = Sym.input "";
@ <i
@ d>
@ .
val a = - : sym
- val b = Sym.fromString "<num>";
val b = - : sym
- Sym.output("", a);
<id>
val it = () : unit
- Sym.equal(a, b);
val it = false : bool
- Sym.equal(a, Sym.fromString "<id>");
val it = true : bool
```

## Sets

The module `Set` defines the abstract type

```
type 'a set
```

of finite sets of elements of type `'a`. It is bound in the top-level environment. E.g., `sym set` is the type of sets of symbols. `Set` also provides various constants and functions for processing sets, but we will only make direct use of a few of them:

```
val toList   : 'a set -> 'a list
val size     : 'a set -> int
val empty    : 'a set
val isEmpty  : 'a set -> bool
val sing     : 'a -> 'a set
```

These values are polymorphic: `'a` can be `int`, `sym`, etc. The function `sing` makes a value `x` into the singleton set `{x}`.



## *Sets*

More functions:

```
val filter : ('a -> bool) -> 'a set -> 'a set
val all    : ('a -> bool) -> 'a set -> bool
val exists : ('a -> bool) -> 'a set -> bool
```

`filter` keeps selected elements of a set, `all` tests whether all elements of a set satisfy a predicate, and `exists` tests whether at least one element of a set satisfies a predicate.

## *Sets of Symbols*

The module `SymSet` defines various functions for processing finite sets of symbols (alphabets), including:

```
val input      : string -> sym set
val output    : string * sym set -> unit
val fromList  : sym list -> sym set
val memb      : sym * sym set -> bool
val subset    : sym set * sym set -> bool
val equal     : sym set * sym set -> bool
val union     : sym set * sym set -> sym set
val inter     : sym set * sym set -> sym set
val minus     : sym set * sym set -> sym set
val genUnion  : sym set list -> sym set
val genInter  : sym set list -> sym set
```

Sets of symbols are expressed in Forlan as sequences of symbols, separated by commas.

## *Sets of Symbols (Cont.)*

Here are some example uses of the functions of `SymSet`:

```
- val bs = SymSet.input "";
@ a, <id>, 0, <num>
@ .
val bs = - : sym set
- SymSet.output("", bs);
0, a, <id>, <num>
val it = () : unit
- val cs = SymSet.input "";
@ a, <char>
@ .
val cs = - : sym set
- SymSet.subset(cs, bs);
val it = false : bool
- val ds = SymSet.fromString "<char>, <>";
val ds = - : sym set
```

## *Sets of Symbols (Cont.)*

More examples:

```
- SymSet.output("", SymSet.union(bs, cs));  
0, a, <id>, <num>, <char>  
val it = () : unit  
- SymSet.output("", SymSet.inter(bs, cs));  
a  
val it = () : unit  
- SymSet.output("", SymSet.minus(bs, cs));  
0, <id>, <num>  
val it = () : unit  
- SymSet.output("", SymSet.genUnion[bs, cs, ds]);  
0, a, <>, <id>, <num>, <char>  
val it = () : unit  
- SymSet.output("", SymSet.genInter[bs, cs, ds]);  
  
val it = () : unit
```

## Strings

We will be working with two kinds of strings:

- SML strings, i.e., elements of type `string`;
- The strings of formal language theory, which we call “formal language strings”, when necessary.

The module `Str` defines the type `str` of formal language strings, which is bound in the top-level environment, and is equal to `sym list`, the type of lists of symbols.

Because strings are lists, we can use SML’s list processing functions on them.

Strings are expressed in Forlan’s input syntax as either a single `%` or a nonempty sequence of symbols.

## *Strings (Cont.)*

The module `Str` defines some functions for processing strings, including:

```
val input      : string -> str
val output    : string * str -> unit
val alphabet  : str -> sym set
val equal     : str * str -> bool
val prefix    : str * str -> bool
val suffix    : str * str -> bool
val substr    : str * str -> bool
val power     : str * int -> str
val last      : str -> sym
val allButLast : str -> str
```

## Strings (Cont.)

Here are some example uses of the functions of `Str`:

```
- val x = Str.input "";
@ hello<there>
@ .
val x = [-,-,-,-,-] : str
- length x;
val it = 6 : int
- Str.output("", x);
hello<there>
val it = () : unit
- SymSet.output("", Str.alphabet x);
e, h, l, o, <there>
val it = () : unit
- Str.output("", Str.power(x, 3));
hello<there>hello<there>hello<there>
val it = () : unit
```

## *Strings (Cont.)*

```
- val y = Str.fromString "ello";  
val y = [-,-,-,-] : str  
- Str.equal(y, x);  
val it = false : bool  
- Str.prefix(y, x);  
val it = false : bool  
- Str.substr(y, x);  
val it = true : bool  
- val z = Str.fromString "h" @ y;  
val z = [-,-,-,-,-] : sym list  
- Str.prefix(z, x);  
val it = true : bool
```



## Strings (Cont.)

```
- val x = Str.fromString "hellothere";  
val x = [-,-,-,-,-,-,-,-,-] : str  
- null x;  
val it = false : bool  
- Sym.output("", hd x);  
h  
val it = () : unit  
- Str.output("", tl x);  
ellothere  
val it = () : unit  
- Sym.output("", Str.last x);  
e  
val it = () : unit  
- Str.output("", Str.allButLast x);  
hellother  
val it = () : unit
```

## *Sets of Strings*

The module `StrSet` defines various functions for processing finite sets of strings, including:

```
val input      : string -> str set
val output    : string * str set -> unit
val fromList  : str list -> str set
val memb      : str * str set -> bool
val subset    : str set * str set -> bool
val equal     : str set * str set -> bool
val union     : str set * str set -> str set
val inter     : str set * str set -> str set
val minus     : str set * str set -> str set
val genUnion  : str set list -> str set
val genInter  : str set list -> str set
val alphabet  : str set -> sym set
```

Sets of strings are expressed in Forlan as sequences of strings, separated by commas.

## *Sets of Strings (Cont.)*

Here are some example uses of the functions of `StrSet`:

```
- val xs = StrSet.input "";
@ hello, <id><num>, %
@ .
val xs = - : str set
- val ys = StrSet.input "";
@ <id><num>, another
@ .
val ys = - : str set
- val zs = StrSet.union(xs, ys);
val zs = - : str set
- Set.size zs;
val it = 4 : int
- StrSet.output("", zs);
%, <id><num>, hello, another
val it = () : unit
```

## *Sets of Strings (Cont.)*

More examples:

```
- val us = Set.filter (fn x => length x mod 2 = 0) zs;  
val us = - : sym list set  
- StrSet.output("", us);  
%, <id><num>  
val it = () : unit  
- SymSet.output("", StrSet.alphabet zs);  
a, e, h, l, n, o, r, t, <id>, <num>  
val it = () : unit
```

## *Relations on Symbols*

The module `SymRel` defines the type `sym_rel` of finite relations on symbols. It is bound in the top-level environment, and is equal to `(sym * sym)set`, i.e., its elements are finite sets of pairs of symbols.

`SymRel` also defines various functions for processing finite relations on symbols, including:

```
val input      : string -> sym_rel
val output    : string * sym_rel -> unit
val fromList  : (sym * sym)list -> sym_rel
val memb      : (sym * sym) * sym_rel -> bool
val subset    : sym_rel * sym_rel -> bool
val equal     : sym_rel * sym_rel -> bool
```

Relations on symbols are expressed in Forlan as sequences of ordered pairs  $(a, b)$  of symbols, separated by commas.

## *Relations on Symbols (Cont.)*

More functions:

```
val union      : sym_rel * sym_rel -> sym_rel
val inter     : sym_rel * sym_rel -> sym_rel
val minus     : sym_rel * sym_rel -> sym_rel
val genUnion  : sym_rel list -> sym_rel
val genInter  : sym_rel list -> sym_rel
```

## Relations on Symbols (Cont.)

More functions:

```
val domain      : sym_rel -> sym set
val range       : sym_rel -> sym set
val reflexive   : sym_rel * sym set -> bool
val symmetric   : sym_rel -> bool
val transitive  : sym_rel -> bool
val function    : sym_rel -> bool
val applyFunction : sym_rel -> sym -> sym
```

`reflexive(rel, bs)` tests whether *rel* is reflexive on *bs*.

The function `applyFunction` is *curried*. Given a relation *rel*, it checks that *rel* is a function, issuing an error message, otherwise. If *rel* is a function, it returns a function of type `sym -> sym` that, when called with a symbol *a*, will apply the function *rel* to *a*.

## *Relations on Symbols (Cont.)*

Here are some example uses of the functions of `SymRel`:

```
- val rel = SymRel.input "";  
@ (1, 2), (2, 3), (3, 4), (4, 5)  
@ .  
val rel = - : sym_rel  
- SymSet.output("", SymRel.domain rel);  
1, 2, 3, 4  
val it = () : unit  
- SymSet.output("", SymRel.range rel);  
2, 3, 4, 5  
val it = () : unit
```



## *Relations on Symbols (Cont.)*

More examples:

```
- SymRel.reflexive(rel, SymSet.fromString "1, 2");  
val it = false : bool  
- SymRel.symmetric rel;  
val it = false : bool  
- SymRel.transitive rel;  
val it = false : bool  
- SymRel.function rel;  
val it = true : bool
```

## *Relations on Symbols (Cont.)*

More examples:

```
- val f = SymRel.applyFunction rel;
val f = fn : sym -> sym
- Sym.output("", f(Sym.fromString "3"));
4
val it = () : unit
- Sym.output("", f(Sym.fromString "4"));
5
val it = () : unit
- Sym.output("", f(Sym.fromString "5"));
argument not in domain

uncaught exception Error
-
```