

Section 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, Mac OS X and Windows can be found on the Forlan WWW site: <http://people.cis.ksu.edu/~stough/forlan/>.

Copyright © 2003–5 Alley Stoughton

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

The L^AT_EX source of these slides, the associated book, and the distribution of the Forlan toolset are available on the WWW at <http://people.cis.ksu.edu/~stough/forlan/>.

1

(2.3) Introduction (Cont.)

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.

2

(2.3) Invoking Forlan

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

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

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 an IDE (integrated development environment). See the Forlan WWW site for information about how to do this.

3

(2.3) Invoking Forlan (Cont.)

The identifier `it` is normally bound to the value of the most recently evaluated expression. Initially, though, its value is the empty tuple `()`, the single element of the type `unit`. The value `()` is used in circumstances when a value is required, but it makes no difference what that value is. SML's prompt is `"-"`. To exit SML, type `CTRL-d` under Linux and Mac OS X, and `CTRL-z` under Windows. To interrupt back to the SML top-level, type `CTRL-c`.

4

(2.3) 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
```

SML responds to each expression by printing its value and type, and noting that the expression's value has been bound to the identifier `it`. Expressions must be terminated with semicolons.

5

(2.3) More Types

SML also has the types `string` and `bool`, as well as product types $t_1 * \dots * t_n$, whose values consist of n -tuples:

```
- "hello" ^ " " ^ "there";  
val it = "hello there" : string  
- 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
```

6

(2.3) 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:

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

7

(2.3) 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 `->`.

8

(2.3) 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
```

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.

9

(2.3) Loading the Contents of Files

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

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

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
```

(2.3) Symbols in Forlan

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

```
val input   : string -> sym
val output  : string * sym -> unit
val compare : sym * sym -> order
```

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;
- `compare` compares two symbols, yielding `LESS`, `EQUAL` or `GREATER`.

(2.3) Symbols in Forlan (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 `@`.

(2.3) Symbols in Forlan (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.

These functions are especially useful when defining functions.

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

13

(2.3) Symbols in Forlan (Cont.)

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

```
- val a = Sym.input "";
@ <id>
@ .
val a = - : sym
- val b = Sym.fromString "<num>";
val b = - : sym
- Sym.output("", a);
<id>
val it = () : unit
- Sym.compare(a, b);
val it = LESS : order
```

14

(2.3) Symbols in Forlan (Cont.)

Expressions in SML are evaluated from left to right, which explains why the following transcript results in the value `GREATER`, rather than `LESS`:

```
- Sym.compare(Sym.input "", Sym.input "");  
@ <can>  
@ .  
@ <be>  
@ .  
val it = GREATER : order
```

15

(2.3) Sets in Forlan

The module `Set` defines an abstract type

```
type 'a set
```

of finite sets of elements of type `'a`. It is bound in the top-level environment. E.g., `int set` is the type of sets of integers. `Set` also defines a variety of functions for processing sets. But we will only make direct use of a few of them, including:

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

These functions are “polymorphic”: they are applicable to values of type `int set`, `sym set`, etc. The function `sing` makes a value `x` into the singleton set `{x}`.

16

(2.3) Sets of Symbols in Forlan

The module `SymSet` defines various functions for processing finite sets of symbols (elements of type `sym set`; 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
```

Sets of symbols are expressed in Forlan as sequences of symbols, separated by commas. When a set is outputted, or converted to a list, its elements are listed in ascending order.

17

(2.3) Sets of Symbols in Forlan (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
```

18

(2.3) Sets of Symbols in Forlan (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
```

19

(2.3) Strings in Forlan

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, as well as some functions for processing strings, including:

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

20

(2.3) Strings in Forlan (Cont.)

The type `str` is bound in the top-level environment, and is equal to `sym list`, the type of lists of symbols. Every value of type `str` has the form $[a_1, \dots, a_n]$, where $n \in \mathbb{N}$ and the a_i are symbols. The usual list processing functions, such as `@` (append) and `length`, are applicable to elements of type `str`, and the empty string can be written as either `[]` or `nil`.

Every string can be expressed in Forlan's input syntax as either a single `%` or a nonempty sequence of symbols. For convenience, though, string expressions may be built up from symbols and `%` using parentheses (for grouping) and concatenation. During input processing, the parentheses are removed and the concatenations are carried out, producing lists of symbols. E.g., `%(hell)%o` describes the same string as `hello`.

21

(2.3) Strings in Forlan (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
```

22

(2.3) Strings in Forlan (Cont.)

```
- val y = Str.input "";
@ %(hell)%o
@ .
val y = [-,-,-,-] : str
- Str.output("", y);
hello
val it = () : unit
- Str.compare(x, y);
val it = GREATER : order
- Str.output("", x @ y);
hello<there>hello
val it = () : unit
- Str.prefix(y, x);
val it = true : bool
- Str.substr(y, x);
val it = true : bool
```

23

(2.3) Sets of Strings in Forlan

The module `StrSet` defines various functions for processing finite sets of strings (elements of type `str set`; finite languages), 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 alphabet   : str set -> sym set
```

Sets of strings are expressed in Forlan as sequences of strings, separated by commas. When a set is outputted, or converted to a list, its elements are listed in ascending order.

24

(2.3) Sets of Strings in Forlan (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>, ano%ther
@ .
val ys = - : str set
- val zs = StrSet.union(xs, ys);
val zs = - : str set
```

25

(2.3) Sets of Strings in Forlan (Cont.)

More examples:

```
- Set.size zs;
val it = 4 : int
- StrSet.output("", zs);
%, <id><num>, hello, another
val it = () : unit
- SymSet.output("", StrSet.alphabet zs);
a, e, h, l, n, o, r, t, <id>, <num>
val it = () : unit
```

26

(2.3) Relations on Symbols in Forlan

The module `SymRel` defines a 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
val union      : sym_rel * sym_rel -> sym_rel
val inter      : sym_rel * sym_rel -> sym_rel
val minus      : sym_rel * sym_rel -> sym_rel
```

27

(2.3) Relations on Symbols in Forlan (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 it 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*.

28

(2.3) Relations on Symbols in Forlan (Cont.)

Relations on symbols are expressed in Forlan as sequences of ordered pairs (a,b) of symbols, separated by commas. When a relation is outputted, or converted to a list, its pairs are listed in ascending order, first according to their left-sides, and then according to their right sides.

29

(2.3) Relations on Symbols in Forlan (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  
- SymRel.output("", rel);  
(1, 2), (2, 3), (3, 4), (4, 5)  
val it = () : unit  
- SymSet.output("", SymRel.domain rel);  
1, 2, 3, 4  
val it = () : unit  
- SymSet.output("", SymRel.range rel);  
2, 3, 4, 5  
val it = () : unit
```

30

(2.3) Relations on Symbols in Forlan (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
```

31

(2.3) Relations on Symbols in Forlan (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  
-
```

32