150Basis Reference

Principles of Functional Programming

Contents

1 Preamble

Standard ML defines a standard Basis Library which contains various commonly used utilities, interfacing with the operating system, etc. Much of this library is entirely unnecessary for learning Standard ML, and some of it may actually be considered bad style or make learning harder. There are also some common practices and utilities that you may find useful for your learning.

To accomodate this we've restricted the basis library to a small subset of core structures that you may use, as well as added a few of our own.

2 MakeString

Often when testing code we want to convert things to a human readable format. So, this structure defines ways to convert common types into strings which can then be outputted in some way.

You can access any of these functions using `MakeString.function_name`.

2.1 Basic Types

```
val bool   : bool   -> string
val int    : int    -> string
val real   : real   -> string
val string : string -> string
val char   : char   -> string
val unit   : unit   -> string
val order  : order  -> string
```

2.2 Lists

```
val bool_list   : bool list   -> string
val int_list    : int list    -> string
val string_list : string list -> string
```

2.3 List Lists

```
val bool_list_list   : bool list list   -> string
val int_list_list    : int list list    -> string
val string_list_list : string list list -> string
```

2.4 Tuples

```
val int_int : int * int -> string
val string_int : string * int -> string
val int_string : int * string -> string
val string_string : string * string -> string
val int_list_int_list : int list * int list -> string
```

2.5 Options

```
val int_option      : int option      -> string
val string_option   : string option   -> string
val int_list_option : int list option -> string
```

2.6 Polymorphic Types and Other Utilities

The following function indents a `string` at all the new lines as well as the start:

```
val indent : string -> string
```

All of the following convert a polymorphic type to a `string`:

```
val list : ('a -> string) -> 'a list -> string
```

```
val list_list : ('a -> string) -> 'a list list -> string
```

```
val pair : ('a -> string) * ('b -> string) -> 'a * 'b -> string
```

```
val option : ('a -> string) -> 'a option -> string
```

3 Test

This structure implements a quick way to verify that our code works as intended. Most of these testers looks for simple equality. Meaning that permutations or other things are not checked.

Most tests adhere to the following format:

```
val () = Test.function_name (name, expected, actual)
```

For example, if we wanted to test that a `fact` function works as expected, we might write the test:

```
val () = Test.int ("base case", 1, fact 0)
```

If `fact` is implemented correctly, then this testcase will be passed and there won't be any visible output.

Suppose our test case was instead (assuming `fact` is correct):

```
val () = Test.int ("base case", 0, fact 0)
```

Then upon evaluating, we will receive the following output:

```
uncaught exception Fail [Fail: base case!
First argument:
        0

Second argument:
        1
]
```

Indicating that the test called `"base case"` failed.

3.1 Basic Types

```
val bool   : string * bool * bool -> unit
val int    : string * int * int -> unit
val string : string * string * string -> unit
val char   : string * char * char -> unit
val order  : string * order * order -> unit
```

3.2 Lists

```
val bool_list_eq   : (string * bool list * bool list) -> unit
val int_list_eq    : (string * int list * int list) -> unit
val string_list_eq : (string * string list * string list) -> unit
```

3.3 List Lists

```
val bool_list_list_eq   : (string * bool list list * bool list
 list) -> unit
val int_list_list_eq    : (string * int list list * int list list)
  -> unit
```

```
val string_list_list_eq : (string * string list list * string list
  list)
                          -> unit
```

3.4 Tuples

```
val int_int : string * (int * int) * (int * int) -> unit
val string_int : string * (string * int) * (string * int) -> unit
val int_string : string * (int * string) * (int * string) -> unit
val string_string : string * (string * string) * (string * string)
  -> unit
val int_list_int_list : string
                        * (int list * int list)
                        * (int list * int list)
                      -> unit
```

3.5 Options

```
val int_option      : string * int option * int option -> unit
val string_option   : string * string option * string option ->
 unit
val int_list_option : string * int list option * int list option
 -> unit
```

3.6 Polymorphic Types and Other Utilities

To pass or fail a test, we define the following functions

```
val fail : string -> unit
val pass : string -> unit
```

For arbitrary types we define the following function:

```
val test_eq : (('a * 'a -> bool) * ('a -> string))
            -> string * 'a * 'a
            -> unit
```

`test_eq (eq, toString) (name, expected, actual)` passes the test if `expected` and `actual` are equal according to `eq`, otherwise the test fails.

```
val test_property : (('a -> bool) * ('a -> string))
                  -> string * string * 'a
                  -> unit
```

`test_property (p, toString) (name, desc, value)` passes the test if `value` has property `p`, otherwise the test fails.

```
val general_eq : string * ''a * ''a -> unit
```

`general_eq (name, expected, actual)` passes the test if `expected = actual`, otherwise the test fails.

The following are for testing equality of more specific, but still polymorphic types. Their arguments are similar to `test_eq`

```
  val list_eq : ('a * 'a -> bool) * ('a -> string)
              -> string * 'a list * 'a list
              -> unit
```

```
  val list_list_eq : ('a * 'a -> bool) * ('a -> string)
                   -> string * 'a list list * 'a list list
                   -> unit
```

```
  val pair : (('a * 'a -> bool) * ('b * 'b -> bool))
              * (('a -> string) * ('b -> string))
           -> string * ('a * 'b) * ('a * 'b)
           -> unit
```

```
  val option : ('a * 'a -> bool) * ('a -> string)
             -> string * 'a option * 'a option
             -> unit
```