STANDARD ML

EXCEPTIONS

EXCEPTIONS - WHY?

- an extensive part of code is error handling
- a function can return an answer, or fail to find one, or signal that a solution does not exists

EXCEPTIONS - ALTERNATIVE

```
datatype int_sol = Success of int | Failure | Impossible;

case methodA(problem) of
    Success s  => Int.toString s
  | Failure    => (case methodB(problem) of
                      Success s  => Int.toString s
                    | Failure    => "Both failed"
                    | Impossible => "No Good")
  | Impossible => "No Good"
;
```

it can be tedious and requires explicit handling

sometimes we don't really know what to do with the error, so we'll simply return it

EXCEPTIONS

- when an error is discovered we will **raise** an exception
- the exception will propagate up until someone **handles** it
- the caller of a function doesn't have to check any error values

in pseudo code:

```
fun inner = do_calculation
    if local_error then raise local_error,
    if global_error then raise global_error;

fun middle = inner(…) handle local_error;

fun outer = middle(…) handle global_error;
```

THE EXCEPTION TYPE `exn`

- we can **raise** only a specific type: `exn`
- `exn` is a special datatype with an **extendable** set of constructors and values

```
exception Failure;
Failure;

exception Problem of int;
Problem;
```

values of type `exn` have all the privileges of other values

```
val p = Problem 1;
map Problem [0, 1, 2];
fun whats_the_problem (Problem p) = p;
```

... except

```
val x = Failure;
x = x;
```

RAISING EXCEPTIONS - SEMANTICS

```
raise Exp
```

- the expression `Exp` of type `exn` evaluates to `e`
- `raise Exp` evaluates to an exception packet containing `e`
- packets are not ML values!

- exception packets propagate under the call by value rule
- all of the following evaluate to `raise Exp`

```
f (raise Exp)

(raise Exp) arg

raise (Exp1 (raise Exp)) (* Exp1 is a constructor *)

(raise Exp, raise Exp1)  (* or {a=Exp, b=Exp1} *)

let val name = raise Exp in some_expression end

local val name = raise Exp in some_declaration end
```

FIXING `hd` AND `tl`

```
exception Empty;

fun hd (x::_) = x
  | hd []      = raise Empty;

fun tl (_::xs) = xs
  | tl []       = raise Empty;
```

HANDLING EXCEPTIONS - SYNTAX

```
Exp_0 handle
     P1 => Exp_1
   | ...
   | Pn => Exp_n
```

- all `Exp_i`s must be type-compatible
- all `Pi`s must be valid patterns for the type `exn`

```
fun len l = 1 + len (tl l) handle Empty => 0;
```

HANDLING EXCEPTIONS - SEMANTICS

```
Exp_0 handle Cons1 x => Exp_1
```

- assume `Exp_0` evaluates to some value `V` then the value of this expressions is
  - `Exp_1` if `Exp_0` evaluates to `raise Cons1 x`
  - `V` otherwise (`V` may be either a normal value or a non-matching raised exception)
- `handle` is short-circuiting
- exactly equivalent to familiar notions from C++

THE TYPE OF `raise Exp`

- the expression `raise Exp` is of type `'a`
- it is **not** an expression of type `exn`
- it simply puts no restrictions on other parts of the expression

```
fun throw _ = raise Empty;
exception Underflow;
fun bar x = if x>0 then x else raise Underflow;
```

USING EXCEPTION HANDLING

```
case methodA(problem) of
    Success s  => Int.toString s
  | Failure    => (case methodB(problem) of
                      Success s  => Int.toString s
                    | Failure    => "Both failed"
                    | Impossible => "No Good")
  | Impossible => "No Good"
```

and now with exceptions:

```
toString (methodA problem handle Failure => methodB problem)
  handle Failure => "Both failed"
    | Impossible => "No Good"
```