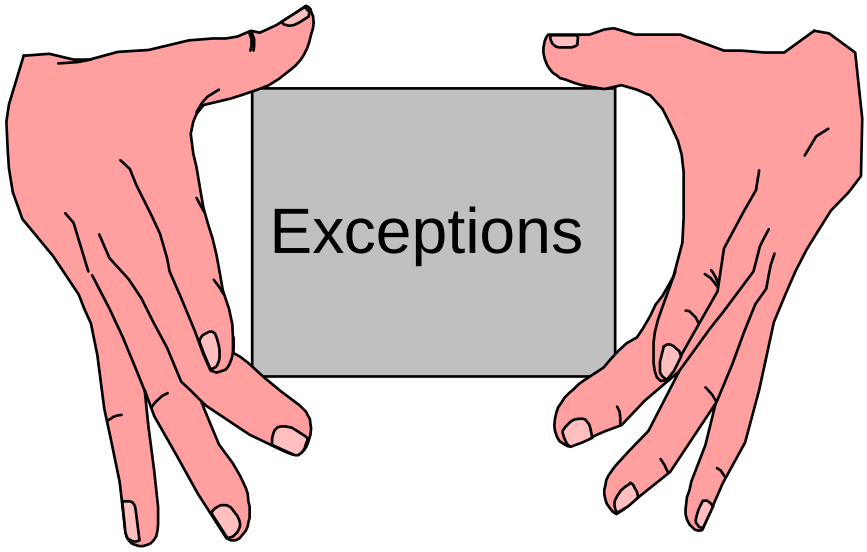
Standard ML



ML Exceptions.1

Exceptions – The Need

- An extensive part of the code is error handling
- A function F can return a problem solution (like int) or fail to find the solution or find that a solution does not exists.
- We can try to use ML datatypes:

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

- Using the failure return values can be tedious

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

ML Exceptions.2

Exceptions

- Instead of using the return value to report an error we will use a mechanism outside the return value: Exceptions.
- When an error is discovered we will ***raise*** an exception
- The exception will propagate up the stack until someone ***handles*** it.
- The caller of a function doesn't have to check all or any of the error values.
- In VERY pseudo-code:

```
fun inner = do calculation
            if local error raise local_error,
            if global error raise global_error
fun middle = inner(…) handle local_error
fun outer = middle(…) handle global_error
```

ML Exceptions.3

Exceptions in ML

- We can ***raise*** only a specific type: the built-in type `exn`.
- `exn` is a datatype with an **extendable** set of constructors.
- Declaring exceptions
  - `exception Failure;`
  - `exception Impossible;`
  - `exception Problem of int;`
- Defines the following constructors:
  - `Failure : exn;`
  - `Impossible : exn;`
  - `Problem : int -> exn;`
- Can be declared locally using **`let`**
- Values of type `exn` have all the privileges of values of other types, and in addition, a special role in the operations ***raise*** and ***handle***

ML Exceptions.4

Raising Exceptions

- **raise** Exp
  - The expression Exp of type "exn **of** 'a", is evaluated to e
  - **raise** Exp evaluates to an *exception packet* containing e
- Packets are *not* ML values
- Packets propagate under the call-by-value rule
  - If E returns a packet then that is the result of f(E)
  - f(**raise** Ex) is equivalent to **raise** Ex
  - **raise**(Badvalue(**raise** Failure))
    is equivalent to:
    raise Failure

ML Exceptions.5

Raising Exceptions

- Expressions are evaluated from left to right.
- If a packet is returned during the evaluation then it is returned as the result, and the rest of the expression is not evaluated.
  - ... let val D = E1 **in** E2 **end**;
    If E1 evaluates to an exception packet so does the entire **let** expression

ML Exceptions.6

Handling Exceptions

- Block handle exp1 => Block1 | . . . | expN => BlockN
- If the result of Block is a packet:
  - The packet's contents are examined.
  - If no pattern matches, the exception is propagated
- If the result is not a packet, the value is passed on as usual
- Fixing ***hd*** and ***tl***

```
- exception Hd;
- fun hd (x::_) = x
    | hd []  = raise Hd;
val hd = fn : 'a list -> 'a
```

```
- exception Tl;
- fun tl (_::xs) = xs
    | tl []   = raise Tl;
val tl = fn : 'a list -> 'a list
```

- Calculating length using exceptions
  - `fun len l = 1 + len(tl l) handle Tl => 0;`

ML Exceptions.7

Another Example

Sum of a list's elements in positions i,f(i),f(f(i)),...

```
- exception Nth;
- fun nth(x::_ ,0) = x
    | nth(x::xs,n) = if n>0 then nth(xs,n-1)
                            else raise Nth
    | nth _        = raise Nth;
val nth = fn : 'a list * int -> 'a
- fun sumchain (l,f,i) =
   nth(l,i)+sumchain(l,f,f(i)) handle Nth => 0;
> val sumchain = fn : int list * int -> int
```

ML Exceptions.8

Using Exception Handling - More

- Example: If methodA fails then methodB is tried

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

- Exceptions give a shorter and clearer program.
  Error propagation does not clutter the code.

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

ML Exceptions.9

Question from Exam

- Given

```
exception E1;
exception E2;
fun f(1) = raise E1 | f(2) = raise E2
  | f(n) =
    ( let val x = f(n-2) handle E2 => 2
        in f(n-1) + x end  ) handle E1 => 1;
```

- What will be returned for f(5)?

ML Exceptions.10

Question from Exam

```
exception E1;
exception E2;
    fun f(1) = raise E1 | f(2) = raise E2
    | f(n) =
        ( let val x = f(n-2) handle E2 => 2
          in f(n-1) + x end  ) handle E1 => 1;
```

f(1) = **raise E1**

f(2) = **raise E2**

f(3) = *(let val x = (* raise E1 *f(1))...) handle E1 => 1* **= 1**

f(4) = *(let val x = (* raise E2 *f(2)) handle E2 => 2* in f(3) + x end **= 3**
(1 2)

f(5) = *(let val x = (* 1 *f(3))...* 3 in f(4) + x end 1 **= 4**

ML Exceptions.11

Standard Exceptions

Built-in exceptions

- `Chr` is raised by chr(k) if k<0 or k>255
- `Match` is raised for failure of pattern-matching (e.g. when an argument matchs none of the function's patterns, if a case expression has no pattern that matches, etc.)
- `Bind` is raised if the value of E does not match pattern P in the declaration `val P = E`

ML Exceptions.12