Exceptions – The Need

◆ An extensive part of the code is error handling
◆ A function can return an answer, or fail to find one, or signal that a solution does not exist.
◆ We can try to use ML datatypes:

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

◆ Using special return values can be tedious, and requires explicit handling every time

```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"
```

◆ Sometimes we don’t really know what to do with the error, so we we’ll return a sol...

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:

```ml
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
```
The Exception type \texttt{exn}

- We can \texttt{raise} only a specific type: the built-in type \texttt{exn}
- The type \texttt{exn} is a special \texttt{datatype} with an \texttt{extendable} set of constructors and values
- Declaring exceptions - values of type \texttt{exn}
  - exception \texttt{Failure};
  - \texttt{Failure};
  
  \texttt{val it = Failure(-) : exn}
- Declaring exceptions – Constructors:
  - exception \texttt{Problem} of \texttt{int};
  - \texttt{Problem};
  
  \texttt{val it = fn : int -> exn}

- Can be declared locally using \texttt{let/local}
The Exception type \texttt{exn}

- In normal expressions, \texttt{exn} behaves pretty much like a regular datatype (though it is not an equality type)
- **Values** of type \texttt{exn} have all the privileges of values of other types
  
  - val \texttt{x} = \texttt{Failure}; (* Failure is a value *)
  
  val \texttt{x} = \texttt{Failure(-)} : \texttt{exn}
  
  - val \texttt{p} = \texttt{Problem 1}; (* Problem is a constructor *)
  
  val \texttt{p} = \texttt{Problem(-)} : \texttt{exn}
  
  - map \texttt{Problem [0, 1, 2]};
  
  val \texttt{it} = [\texttt{Problem(-)},\texttt{Problem(-)},\texttt{Problem(-)}] : \texttt{exn list}
  
  - fun what's_the_problem (\texttt{Problem} \texttt{p}) = \texttt{p};
  Warning: match nonexhaustive
    Problem \texttt{x} => ...
  
  val \texttt{what's_the_problem} = fn : \texttt{exn} -> \texttt{int}
Raising Exceptions - Semantics

◆ **raise** Exp
  - The expression Exp of type exn is evaluated to e
  - **raise** Exp evaluates to an *exception packet* containing e
  - Packets are not ML values

◆ Packets propagates under the call-by-value rule

◆ Which means that all of the following evaluate to **raise** Exp
  - f (**raise** Exp)
  - (**raise** Exp) arg
  - **raise** (Exp1 (**raise** Exp)) (* Exp1 is 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**
Raising Exceptions - Example

◆ Fixing \texttt{hd} and \texttt{tl}

- exception \texttt{Empty};

- fun \texttt{hd} (x::_) = x
  
  | \texttt{hd} [] = \texttt{raise Empty};

\texttt{val hd} = \texttt{fn : 'a list -> 'a}

- fun \texttt{tl} (_::xs) = xs
  
  | \texttt{tl} [] = \texttt{raise Empty};

\texttt{val tl} = \texttt{fn : 'a list -> 'a list}
Handling Exceptions - Syntax

◆ General form:
\[
\text{Exp}_0 \ \text{handle} \ P_1 \Rightarrow \text{Exp}_1 \\
\vdots \\
\text{P}_n \Rightarrow \text{Exp}_n
\]

◆ All Exp_\text{i} must be type-compatible
◆ All P_i must be valid pattern for the type exn

◆ Calculating length using exceptions
  
  - fun \ len \ l = 1 + len (tl \ l) \ \text{handle} \ \text{Empty} \Rightarrow 0;
Handling Exceptions - Semantics

$$\text{Exp}_0 \ handle \ \text{Cons1} \ x \Rightarrow \text{Exp}_1$$

- Assume $\text{Exp}_0$ evaluates to some value $V$
  Then the value of this expression is:
    - $\text{Exp}_1$ if $\text{Exp}_0$ evaluate to $\text{raise Cons1} \ x$
    - $V$ otherwise. $V$ may be either a normal value, or a non-matching raised exception.

- Handle is a short-circuit operator, like if-then-else
- All this is exactly equivalent to the familiar notions from C++
The Type of \texttt{raise Exp}

- The expression \texttt{raise Exp} is of type \texttt{\textquote{\textquote{a}}}
- It is \textbf{NOT} an expression of type \texttt{exn}!

- Within context, it simply puts no restriction on other parts of the expression:
  - \texttt{fun throw _ = raise Empty;}
  - \texttt{val throw = fn : \textquote{\textquote{a}} -> \textquote{\textquote{b}}}
  - \texttt{fun bar x = if \texttt{x}>0 then x else raise Underflow;}
  - \texttt{val bar = fn : \textquote{int} -> \textquote{int}}
Example: If methodA fails then methodB is tried

```haskell
  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"
```

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

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

- `raise` Problem;
Error: argument of raise is not an exception [tycon mismatch]
  raised: int -> exn
in expression:
  raise Problem
- `hd ["good"] handle nil => "bad";
Error: handler domain is not exn [tycon mismatch]
  handler domain: 'Z list
in expression:
  hd ("good" :: nil)
handle
  nil => "bad"
  | exn => raise exn
- `hd ["good"] handle Empty => false;
Error: expression and handler don't agree [tycon mismatch]
  body: string
handler range: bool
in expression:
  hd ("good" :: nil)
handle
  Empty => false
  | exn => raise exn
Exam Questions

◆ What is the response

1. local
   exception E of string;
   in
   fun f (E "Hello", E x) = x;
   end
   stdIn:9.7-9.29 Warning: match nonexhaustive
   (E "Hello",E x) => ...
   val f = fn : exn * exn -> string

2. let
   exception Exy of int
   in
   (fn 2 => raise Exy 4
    | x => x+x) 2 handle Exy n => n
   end
   val it = 4 : int
Some 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 matches 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`