Exception Handling in SML

- Exceptions are datatypes of error values used to minimize explicit testing.
- Exceptions are raised when the failures are discovered and appropriately handled elsewhere.
- General form of exception declaration is:

  exception *exception_name*

- Exception name is a new constructor of the built-in type exn similar to value constructor in datatype declaration.
- Exception constructor can be used in pattern or expression in the same way as value constructors are used.
- We use a convention that exception_name should start with capital letter.

- Exception can also be a function.

```
-       exception Fail;
>       exception Fail = Fail : exn
-       exception Failure of string;
>       exception Failure  = fn : string -> exn
-       exception Badvalues of int;
>       exception Badvalues = fn :  int -> exn
```

- In SML, the exception declaration can declare one or more exception names.
- These can be raised by a construct called **raise** to force a computation to terminate with an error signal.

Raising of an exception

- The general form of exception raising is:

  raise *exception_name*

- It creates an exception packet containing a value of built-in type exn

```
-       exception Bad;
>       exception Bad = Bad : exn
-       fun  divide (x, y) =     if y = 0 then raise Bad else
                                 x div y;
>       val divide = fn : int * int -> int
-       divide (12,3);
>       val it = 4 : int
-       divide(34,0);
>       uncaught exception Bad raised
```

- Let us define our own functions **head** and **tail** for getting head and tail of a given list.
- These functions should take care of the situations when applied on empty list.
- For this purpose we define exceptions and raise them suitably.

```
-       exception Head;
>       exception Head = Head : exn
-       exception Tail;
>       exception Tail = Tail : exn
-       fun     head (x::_) = x
                | head []  = raise Head;
>       val head = fn : 'a list -> 'a
-       head [2,3,4];
>       val it = 2 : int
```

```
-       head [];
>       uncaught exception Head  raised
-       hd [];              system defined head function
>       uncaught exception Empty raised  system defined
        exception name
-       fun     tail (_::xs)= xs
                | tail [] = raise Tail;
>       val head = fn : 'a list -> 'a list
-       tail [2,3,4,5];
>       val it = [3,4,5] : int list
-       tail [];
>       uncaught exception Tail  raised
-       tl [];          system defined function for tail
>       uncaught exception Empty raised  system defined
        exception name
```

Exception handling

• An exception handler tests whether the result of an expression is an exception packet or not.

• The exception handler is always placed after an expression.

• The general form is:

```
exp handle  <pat1>  =>  exp1
          | <pat2>  =>  exp2
          | <patn>  =>  expn
```

• An exception handler catches a raised exception if one of the pattern matches the value of an exception and then corresponding expression is evaluated under this binding.

• Hence if **exp** returns a normal value, then the handler simply passes this value on.

- If **exp** returns an exception packet's content (because of raising exception) then its contents are matched against the pattern.
- If patk is the first pattern to match then the result is the value of an expression expk (1≤ k ≤ n).
- It should be noted that exp, exp1 ,… and expn must be of the same type.
- If we define our own exception names then declare them before their use otherwise built in exceptions can also be directly raised.
- The handlers are provided in the functions which use other functions directly or indirectly having exception raised in them.

```
-       fun len x = 1 + len (tail x) handle Tail =>0;
>       val len = fn : 'a list -> int
-       len [2,3,4];
>       val it = 3 : int
-       len [];
>       val it = 0 : int
-       fun  head_list  x = head x handle Hd=>0;
>       val head_list = fn : int list -> int
-       head_list [3,4,5];
>       val it = 3 : int
-       head_list [];
>       val it = 0 : int
-       fun tail_list x = tail x handle Tail =>[0];
>       val tail_list = fn : int list -> int list
-       tail_list [];
>       val it = [0] : int list
```

- *Function for finding nth element of the list assuming first element is stored at 0th index*

```
-       exception Subscript;
>       exception Subscript = Subscript :exn
-       fun     nth (x::_, 0) = x
              | nth (x::xs, n) = if n>0 then nth(xs, n-1) else
                                          raise Subscript
              | nth _ =  raise Subscript;
>       val nth = fn : 'a list * int -> 'a
-       nth ([2,3,4,5],0);
>       val it = 2 : int
-       nth ([1,3,5,6], 5);
>       uncaught exception Subscript
```

- *Handling of raised exception*

```
-       fun findnth (l,n)=nth (l,n) handle Subscript =>0;
>       val findnth = fn : int list * int -> int

-       findnth ([34,56,12,33],6);
>       val it = 0 : int

-       findnth ([34,56,12,33],2);
>       val it = 12 : int
```

- *Function for computing the sum of a list's elements at position n, f(n), f(f(n), …. The sequence of integer terminates at the first value out of range using exception.*

```
-       fun f (n) = n-2;
>       val f = fn : int -> int
-       fun chain (x, n)=nth(x,n) + chain(x, f(n)) handle
                                Subscript => 0;
>       val chain = fn : int list * int -> int
-       chain ([23,45,65,12],2);
>       val it = 88 : int
-       chain ([23,45,67],1);
>       val it = 45 : int
```

- *Check whether a given positive integer is a square.*

Display false if number is negative integer

```
-       exception Neg;
>       exception Neg = Neg : exn
-       fun sq x:int = x*x;
>       val sq = fn : int -> int
-       fun issq i =   if i > 0 then
                              sq (round  (Math.sqrt (real i))) = i
                                      else raise Neg;
>       val issq = fn : int -> bool
-       issq ~45;          exception is raised in this function
>       uncaught exception Neg
-       fun is_sq i = issq i handle Neg => false;
>       val is_sq = fn : int -> bool
```

Benefits of the exception mechanism

- We are forced to consider the exceptional cases otherwise we will get an uncaught exception at run time.
- We can separate the special cases from the normal case in the code rather than putting explicit checks.
- Another typical use of exception is to implement backtracking which requires exhaustive search of a state space.

Backtracking using Exception Mechanism

- Let us consider the following example which implements backtracking.
- Exceptions are raised and handled in the same function.
- When exception is raised, it backtracks to previous solution with the help of handle exception.
- The function convert converts a given amount in number of coins starting from the highest to the lowest in the best possible way.
- The list of coins is passed as an argument along with the amount.

```
-       exception Invalid of int;
>       exception  Invalid = fn :  int -> exn
-       fun     convert (xs,0) =[]
              | convert ([], amount)  = raise Invalid (1)
              | convert (x::xs, amount) =
                     if amount < 0 then raise Invalid (2)
                     else
                            if x > amount then
                                   convert (xs, amount)
                     else x::convert(c::xs,(amount-x)))
                     handle Invalid (1) =>
                                   convert (xs, amount);
                                          ↑
                                   for backtracking
>       val  convert = fn : int list * int -> int list
```

- convert([5,3,2],56);
> **val it = [5,5,5,5,5,5,5,5,5,5,3,3] : int list**
- convert([5],~23);
> **uncaught exception Invalid raised**
- convert([],23);
> ***uncaught exception Invalid raised***
- convert([5,3],4);
> **uncaught exception Invalid raised**

• This function raises exception when amount is negative.
• It is handled in another function which makes use of a function convert.

```
-       fun     convert1 ([], amount) = []
                | convert1 (list,amount) = convert(list,amount)
                                        handle Invalid (2) => [];
>       val  convert1 = fn : int list * int -> int list

-       convert1 ([5,3,2],56);
>       val it = [5,5,5,5,5,5,5,5,5,5,3,3] : int list

-       convert1([5],~23);
>       val it = [] : int list

-       convert1([],23);
>       val it = [] : int list

-       convert1([5,3],4);
>       uncaught exception Invalid raised
```

• The function convert1 raises exception when the amount can't be expressed by any combination of coins.
• This situation can further be handled in yet another function which calls convert1 and handles the exception raised in convert1.

```
-       fun     convert2 ([], amount) = []
                | convert2 (list, amount) =
                        convert1(list, amount)
                                handle Invalid (1) => [];
>       val  convert2 = fn : int list * int -> int list
-       convert2 ([5,3,2],56);
>       val it = [5,5,5,5,5,5,5,5,5,5,3,3] : int list
-       convert2 ([5],~23);
>       val it = [ ] : int list
-       convert2([],23);
>       val it = [] : int list
```