```
datatype money
pounds(3.0)   =  pounds of real
dollars(3.0)  | dollars of real
yen(3.0)      | yen of real
```

Enumerated type

```
datatype day = mon | tues | weds | thurs | fri | sat | sun
fun activity Fri = "party"
  | activity sat = "shop"
  | activity sun = "church"
  | activity D = "work"
```

D is a variable

capital letter? variable

non capital?

activity: day → string

```
activity(Fri) = "party"
activity(mon) = "work"
activity(sun) = "church"
```

Let expressions

Local Variables

Local functions

```
Let val x1 = expr1
    and x2 = expr2
    :
    and xn = exorn
```

meaning:

first calculate $expr_1 \dots expr_n$

then assign these values to variables $x_1, x_2, \dots, x_n$

resa

in Expr Then evaluate expr & return its value

end

```
fun f(x, y)=
    let val D=X - Y
        and S= x + Y
    in D * D + S * S
    end
```

$f(3, 1) = (3-1)^2 + (3+1)^2$

last 3[1, 2, 3, 4, 5, 6, 7]

⇒ [5, 6, 7]

```
fun last3 L
    let val RL = reverse L
    in [(third RL), (second RL), (first RL)]
    end
```

given reverse: 'a list → 'a list

Lexical Scope

```
Let val y=2 in
    expr then y
         else -y
```

end

⟹ 2 or -2, always, even if expr redefines y

```
let val y = 2 in expr
    if  y > (let val y=3) in y * y end)  then  y
                                         else  -y
    end
```

⟹ -y ⟹ -2 (not -3)

local functions

```
sum Cube(A, B, C) = A^3 + B^3 + C^3
fun cube(x) = X * X * X
    in cube(A) + cube(B) + cube(C)
end
```

local recursion

```
fun revers (L) =
    let fun reverse2 ([], L2) = L2
          | reose2 (h::t, L2) = reveres2(t, h:: L2)
    in reverse2(L, [])
end
```

reverse2 (L1, L2)

y reverse 2([1,2,3],[a,b,c])

⟹ [3, 2, 1, a, b, c]

reverse2(L, []) = reverse(L)

reverse2([2,3],[1,a,b,c])

reverse2([3],[2,1,a,b,c])

reverse([],[3,2,1,a,b,c])

```
fun reverse2([],L2) = L2
    | reverse2(h::t, L2) = reverse2(t, h::L2)
end

fun reverse(L) = reverse2(L,[])
```

Local Mutual Recursion

```
let fun even 0 = true
    | even N = odd(N-1)
    and odd 0 = false
    | odd N = even(N-1)
in even(4) end;
```

```
odd N = mod(N,2)
even N = 1- odd(N)
```

Picture

Higher order functions unnamed functions (lambda-like)

(fn x => expr) ≡ (lambda (x) expr) in scheme

(fn x => x+1)(2) => 3

map(sq, [1,2,3]) => [1², 2², 3²] = [1, 4, 9]

map (fn x => x+1, [1, 2, 3]) => [2,3,4]

map('a -> 'b) * ('a list) -> ('b list)

map(F, L)

```
fun map(f,[]) = []
```

```
  | map (f, h::t)) = f(h) :: map(f, t) ;
```

```
val F = (fn x => x + 2)
F 3 => 5
≡ fun f x = x + 2;
```

local recursion

```
reverse(L) =
    let fun reverse2([], L2) = L2
        | reverse2(h::t, L2) = reverse2(t, h::L2)
    in reverse2(L, [])
end
```

```
fun double F = (fn x => F(F x))
double : ('a → 'a) → ('a → 'a)
```

Exception Handling

fun f(x) = 1.0/x	f : real ↦ real	f 0 => error
fun g(x) = hd(x)	g : 'a list → 'a	g([]) => error
fun h(x) = tl(x)	h : 'a list → 'a list	h [] => error

```
n!  require  n ≥ 0
exception NegArg
fun fact N =
      if N = 0 then 1
```

```
else if N > 0 then N * fact(N-1)
else raise NegArg;
```

```
fact : int -> int
```

when $0 \leq m \leq n$ then $\binom{n}{m} = \frac{n!}{m!(n-m)!}$

(1) $n < 0$ negative(n)

(2) $m < 0$ negative(m)

(3) $m > n$ TooBig(m)

```
exception Negative of int
exception TooBig of int
```

```
fun comb(N, M) =
    if N < 0 then raise Negative(N)
    else if M < 0 then raise Negative(M)
    else if M > N then raise TooBig(M)
    else fact(N) div (fact(M) * fact(N - M));
                 int division
```

```
fact (3 + comb(2, 4))
    uncaught exception TooBig(4)
comb(7.5, 6.0)
    type error, don't need to catch
```

Exception Handling

<expr$_1$> handle <exception> = <expr$_2$>

meaning: when evaluating <expr$_1$>

(1) if no expections are raised then return values of <expr$_1$>

(2) if <exception> is raised, then return value of <expr$_2$>