Evaluation Strategy

Strict (or eager) evaluation.

Also known as call-by-value

Given an expression, which is a function application

$$f(E_1, \ldots, E_n)$$

evaluate $E_1, \ldots, E_n$ and then apply f to the resulting values.

Call-by-name:

Substitute the expressions $E_1, \ldots, E_n$ into the definition of f and then evaluate the resulting expression.

Lazy Evaluation

Also known as call-by-need.

Like call-by-name, but sub-expressions that appear more than once are not copied. Pointers are used instead.

Potentially more efficient, but difficult to implement.

Standard ML uses strict evaluation.

Lists

A list is an ordered collection (of any length) of elements of the same type

```
- [1,2,4];
> val it = [1, 2, 4] : int list
- ["a" , "", "abc", "a"];
> val it =  . . .  : string list
- [[1],[],[2,3]];
> val it = . . . : int list list
- [];
> val it = [] : 'a list
- 1::[2,3];
> val it = [1, 2, 3] : int list
```

Lists

There are two kinds of list:

`nil` or `[]` is the empty list

`h::t` is the list with head `h` and tail `t`

`::` is an infix operator of type

```
fn : 'a * 'a list -> 'a list
```

$[x_1, \ldots, x_n]$ is shorthand for

$$x_1\texttt{::}(\cdots(x_n\texttt{::nil})\cdots)$$

Built-in Functions 1

`null`

```
fn : 'a list -> bool
```

determines if a list is empty

`hd`

```
fn : 'a list -> 'a
```

gives the first element of the list

`tl`

```
fn : 'a list -> 'a list
```

gives the tail of the list

Built-in Functions 2

```
length
```

```
fn : 'a list -> int
```

gives the number of elements in a list

```
rev
```

```
fn : 'a list -> 'a list
```

gives the list in reverse order

```
@
```

appends two lists NB: infix!

List Functions

```
fun null l =
      if l = [] then true else false;
```

or, using pattern matching:

```
fun null []     = true
  | null (_::_) = false;
```

```
fun hd (x::_) = x;
```

```
fun tl (_::l) = l;
```

NB: these functions are built-in and do not need to be defined

Recursive definitions

```
fun rlength []      = 0
  | rlength (h::t) = 1 + rlength(t);
```

```
fun append ([], l)    = l
  | append (h::t, l) = h::append(t,l);
```

```
fun reverse []      = []
  | reverse (h::t) = reverse(t)@[h];
```

Purely recursive definitions can be very inefficient

Iterative Definitions

```
fun addlen ([],n)     = n
  | addlen (h::t, n) = addlen (t, n+1);
```

```
fn : 'a list * int -> int
```

```
fun ilength l = addlen(l,0);
```

```
fun revto ([],l)     = l
  | revto (h::t, l) = revto (t, h::l);
```

```
fn : 'a list * 'a list -> 'a list
```

Library List Functions

```
load "List";
```

We can then use `List.take, List.drop`

```
fun take (k, [])   = []
  | take (k, h::t) =
       if k > 0 then h::take(k-1,t)
                else [];

fun drop (k, [])   = []
  | drop (k, h::t) =
       if k > 0 then drop(k-1,l)
                else h::t;
```

```
fn : int * 'a list -> 'a list
```