Sets: Without structures

```
val emptyset = [];

val rec isin =
  fn x =>
    (fn [] => false
    | y::l =>
      if (x=y) then
         true
      else
       isin x l);

val addin =
    fn x =>
      fn l =>
       if (isin x l)
          then
            l
       else
           x::l;
```

Continued

```
val rec removefrom
    = fn x =>
         (fn [] => []
          | y::l =>
            if (x=y) then
                l
           else
     y::(removefrom x l));
```

Sets

- Sets are still based on type list
- We introduce a signature for set type

```
signature SET = sig
    type 'a set
    val emptyset : 'a set
    val isin : ''a -> ''a set -> bool
    val addin : ''a -> ''a set -> a set
    val removefrom : ''a -> ''a set -> ''a set
end;
```

Sets

- With the signature, we can then instantiate it as follows

```
structure Set = struct
   val emptyset = [];

   val rec isin =
      fn x =>
        (fn [] => false
         | y::l =>
            if (x=y) then
                 true
            else
                isin x l);

   val addin =
      fn x =>
          fn l =>
            if (isin x l)
              then
                 l
              else
                 x::l;
```

Continued

```
val rec removefrom =
       fn x =>
           (fn [] => []
          | y::l =>
            if (x=y) then l
            else y::(removefrom x l));
end :> SET;
```

Towers of Hanoi

- Put the following into a file

```
fun move n from to via =
    if n = 0 then
        " ; "
   else
    (move (n - 1) from via to) ^
   "Move disk from " ^ from ^ " to " ^ to ^
    (move (n - 1) via to from);
```

Testing it

- Import

```
use "hanoi.sml";
```

- Example

```
move 3 "A" "B" "C";
```

Alternative: Induction from 1 instead of 0

```
fun move n from to via =
    if n = 1 then
        " Move_disk_from " ^ from ^ " to " to
   else
    (move (n - 1) from via to) ^
     move 1 from to via ^
    (move (n - 1) via to from);
```

Fibonacci: Tail-recursive

```
val rec fibrc  =
   fn (n,res1,res2) =>
   if n=0 then res2
   else if n=1 then res2
   else fibrc(n-1,res2,res1+res2);
```

Lists

- List: Recursive type
- Construction of lists

```
datatype list = empty | cons of (int * list );

cons (1, cons (2, empty));
```

- `cons` (and `car`, `cdr`) come from LISP

Operations on lists: Exercises

- `car`: Return head of list
- `cdr`: Return tail of list

Operations on lists

- `car`: Return head of list

```
val car = fn cons (v,_ ) => v;
```

- `cdr`: Return tail of list

```
val cdr = fn cons (_,l) => l;
```

- Matches are not exhaustive. Why? Can you get an exception?

Answer

```
car empty;
```

Other functions: Exercises

- Is a list empty?
- Length of a list

Solutions

- Is a list empty?

```
val isempty = fn empty => true
   | _=> false ;

isempty (cons (1, empty));
```

- Length of a list

```
val rec length = fn empty => 0
     | cons (_, l)=>1+length l;

val a = cons (1, cons (2, cons (3, empty)));
length a;
```

More on lists

- Concatenate two lists

Solution

- Concatenate two lists

```
val rec concat = fn empty => (fn b => b)
  | cons (e, l) => (fn b => cons (e, concat l b));
```

Exercise

- Write a function to insert an item into an *ordered* list

Solution

```
val rec insert =
   fn n => fn l =>
      if (isempty l) orelse (n < car l)
        then cons (n, l)
    else cons (car l, insert n (cdr l ));
```

Exercise

Extend the currency example to include Danish crowns (dkk)

Solution

```
datatype currency = eur | usd | dkk | ounce_gold;
datatype money = Eur of real | Usd of real | Dkk of real | Ounce_gold of real;

fun convert (amount, to) =
    let val toeur = fn
       Eur x => x
     | Usd x => x / 1.05
     | Dkk x => x / 7.45
     | Ounce_gold x => x * 1113.0
   in
     ( case to of
        eur => toeur amount
      | usd => toeur amount * 1.05
      | dkk => toeur amount * 7.45
      | ounce_gold => toeur amount / 1113.0
      , to)
  end;
```