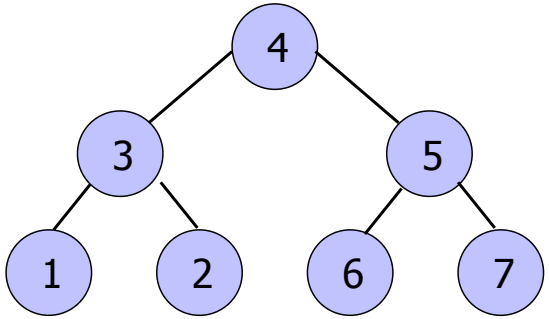
Esempi ML

```
datatype tree = leaf of int | node of int*tree*tree
```

```
node(4, node(3,leaf(1), leaf(2)),
        node(5,leaf(6), leaf(7))
     )
```



Problema: scrivere una funzione
`ap: tree*int -> bool`
per decidere se un intero
occorre in un albero

```
fun ap(leaf(n),x) = (x = n)|
    ap(node(n,t,u),x) = (x=n)
                  orelse ap(t,x) orelse ap(u,x);
```

```
$ sml
Standard ML of New Jersey v110.71 [built: Thu Sep 17 16:48:42 2009]
- datatype tree = nil | leaf of int | node of int*tree*tree;
datatype tree = leaf of int | nil | node of int * tree * tree
- fun ap(leaf(n),x) = (x = n)|
      ap(node(n,t,u),x) = (x=n) orelse ap(t,x) orelse ap(u,x);
val ap = fn : tree * int -> bool
- val t = node(4,
node(3,leaf(1), leaf(2)),
node(5,leaf(6), nil)
);
val t = node (4,node (3,leaf #,leaf #),node (5,leaf #,nil)) : tree
- ap(t,5);
val it = true : bool
val it = true : bool
- ap(t,1);
val it = true : bool
- ap(t,9);
val it = false : bool
-
```

definire alberi binari etichettati con interi:
non è detto che un nodo abbia due figli!

datatype btree = nil | leaf of int | node of int*btree*btree

datatype btree = nil | node of int*btree*btree

Problema: scrivere una funzione

```
ap1: btree*int -> bool
```

per decidere se un intero occorre in un albero

datatype btree = nil | node of int*btree*btree

```
fun ap(nil,x) = false |
    ap(node(n,t,u),x) = (x=n)
                  orelse ap(t,x) orelse ap(u,x);
```

LISTE

```
[];
nil;
[1,3,4];
2 :: [4,5];
```

Problema: calcolare la lunghezza di una lista

```
fun length nil = 0
    length (x::s) = 1 + length(s);
```

```
length:'a list → int

lenght: ∀α∈TYPES. α list → int
```

```
fun length nil = 0|
    length ((x:int)::s) = 1 + length(s);
```

```
length: int list → int
```

Problema: calcolare il massimo di una lista di **naturali**

```
fun max [] = 0|
    max (h::t) =
        if (h > max(t)) then h
                        else max(t) ;
```

```
max: int list → int
```

```
fun max [] = 0|
    max (h::t) =
          if (h > max(t)) then h
                          else max(t) ;
```

```
fun max1 [] = 0
| max1 (h::t) = let val z = max1(t)
                in
                if (h > z) then h
                           else z
                end;
```

```
max1: int list → int
```

```
fun max [] = 0.0
    max (h::t) =
          if (h > max(t)) then h
                          else max(t) ;
```

```
max: real list → int
```

Problema: calcolare il massimo di una lista di **interi**

```
fun max [] = 0|
    max (h::t) =
          if (h > max(t)) then h
                          else max(t) ;
```

NO! errato se la lista contiene valori negativi

```
- max([~1]);
val it = 0 : int
-
```

```
fun max1[] = 0|
    max1(h::nil) = h |
    max1(h::t) =
          if (h > max1(t)) then h
                           else max1(t) ;
```

Problema: estrarre da una lista la sottolista degli elementi in posizione pari

```
fun extPP nil = nil                                    |
    extPP (t::nil) = nil                               |
    extPP (a::(b::r)) = b :: extPP(r);
```

```
extPP:'a list -> 'a list
```

Problema: sommatoria delle etichette di un albero binario di interi

datatype btree = nil | node of int*btree*btree

```
fun sm(nil) = 0 |
    sm(node(n,t,u)) = n+sm(t)+sm(u);
```

Definire le espressioni aritmetiche basate su letterali, variabili, somma e prodotto.
Scrivere un valutatore.

```
infixr 5 ++;
infixr 6 **
datatype aexpi = c of int |v of int |
  ++ of aexpi*aexpi | ** of aexpi*aexpi;
```

```
infixr 5 ++;
infixr 6 **
datatype aexpi = c of int |v of int |
               ++ of aexpi*aexpi | ** of aexpi*aexpi;
```

```
fun Eval(e,s)=
  case e of
    c(n)          => n                                    |
    v(n)          => s(n)                                 |
    (e1 ++ e2)    => (Eval(e1,s)) + (Eval(e2,s))          |
    (e1 ** e2)    => (Eval(e1,s)) * (Eval(e2,s)) ;

fun s(x:int) = 0;

Eval(c(2) ++ (c(3) ** c(4)), s);
```

```
(* versione con identificatori rappresentati costruttori *)

infixr 5 ++;
infixr 6 --;
infixr 7 **;
datatype aexp =  C of int          |
                 R of int          |
                 ++ of aexp*aexp   |
                 ** of aexp*aexp   |
                 -- of aexp* aexp;

infixr  << ;
infixr  == ;
infixr  &&;

datatype bexp = << of aexp * aexp |
                == of aexp * aexp |
                !! of bexp        |
                && of bexp * bexp;
```

```
datatype Lval = L of int;

fun  aeval (C(n),s) = n                                    |
     aeval (R(t),s) = s(t)                                 |
     aeval (e1 ++ e2, s) = aeval(e1,s) + aeval(e2,s)       |
     aeval (e1 -- e2, s) = aeval(e1,s) - aeval(e2,s)       |
     aeval (e1 ** e2, s) = aeval(e1,s) * aeval(e2,s)       ;

fun  beval ( e1 << e2, s) = aeval(e1,s) < aeval(e2,s)          |
     beval ( e1 == e2, s) = aeval(e1,s) = aeval(e2,s)          |
     beval ( !!(b), s) = not(beval(b,s))                       |
     beval ( b1 && b2, s) = beval(b1,s) andalso beval(b2,s)    ;
```

```
infixr 1 ^ ;
infixr 2 <- ;
datatype com = <- of Lval * aexp   |
               SKIP                |
               IF of bexp*com*com  |
               WHILE of bexp*com   |
               ^ of com*com        ;

fun s0 (x:int) = 0;

fun upd (s:int->int,x,n) = fn z => if (z = x) then n else s(z);


fun E(SKIP, s)          = s                                          |
    E(L(i) <- e , s ) = upd(s,i,aeval(e,s))                          |
    E(IF(b,c1,c2), s) = if (beval(b,s)) then E(c1,s)
                                        else E(c2,s)                 |
    E(c1^c2 , s)        = E(c2, E(c1,s))                             |
    E(WHILE(b,c),s)   = if (beval(b,s)) then E(WHILE(b,c),E(c,s))
                                        else s;
```

```
val t = L(1) <- C(3) ^ L(1) <- (R(1)**R(1));

val prog =
L(1) <- C(1) ^
L(2) <- C(0) ^
WHILE((R(1)<<C(11)),
   (L(2) <- R(2) ++ R(1) ^ L(1) <- R(1) ++ C(1))
);
```