15-150
Fall 2024

Dilsun Kaynar

LECTURE 12

Programming with Continuations

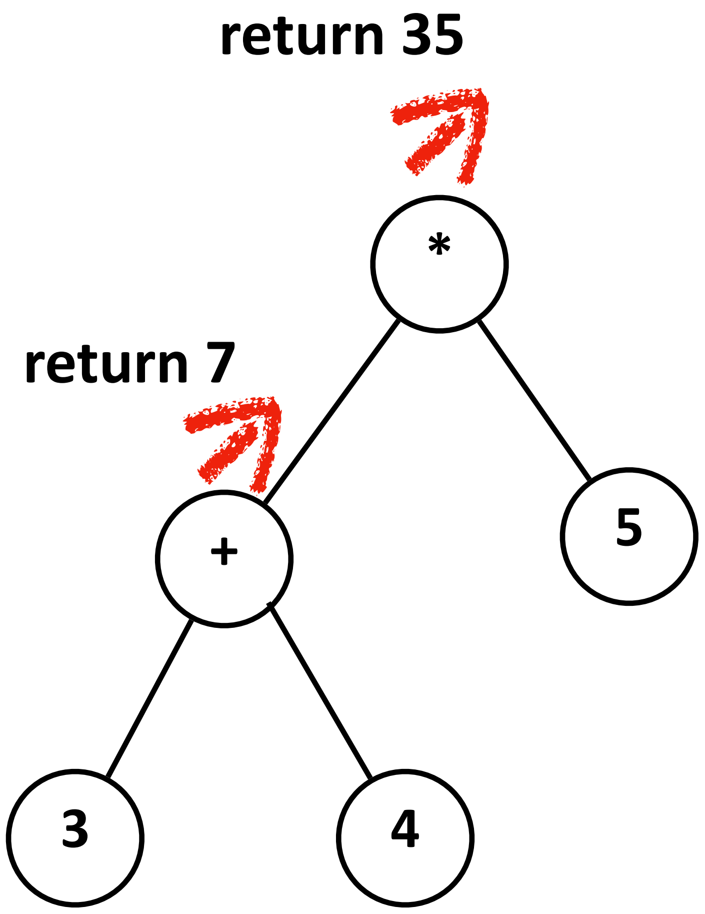
Intuition

- A continuation is a functional argument that controls flow of expression evaluation. It is often used to abstract away the “rest of the computation”.

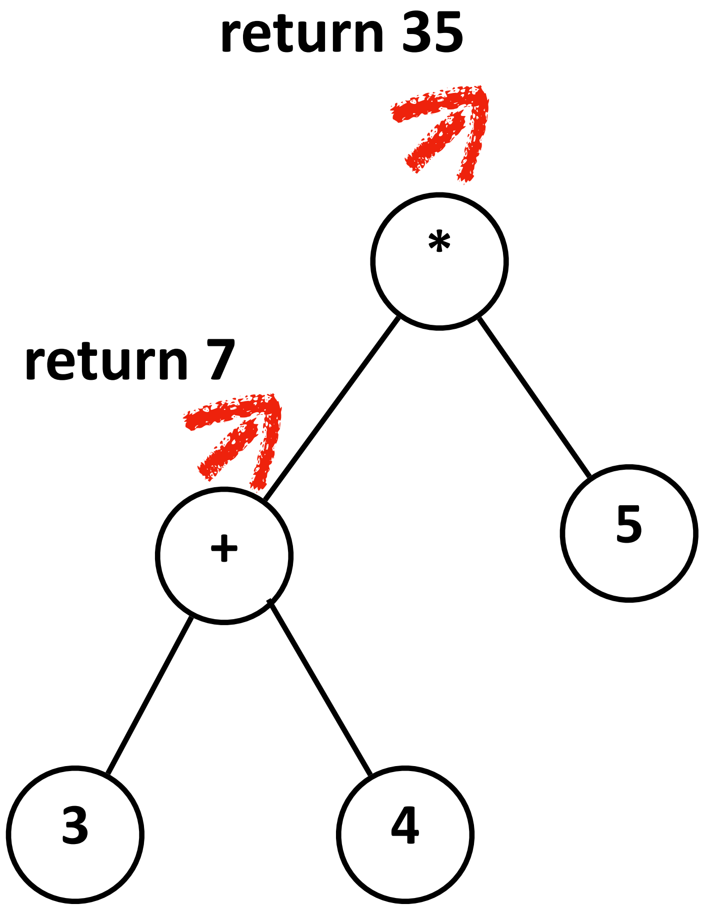
- They act like “functional accumulators” in tail-recursive functions.

- Useful in backtracking search

Evaluating (3+4)*5

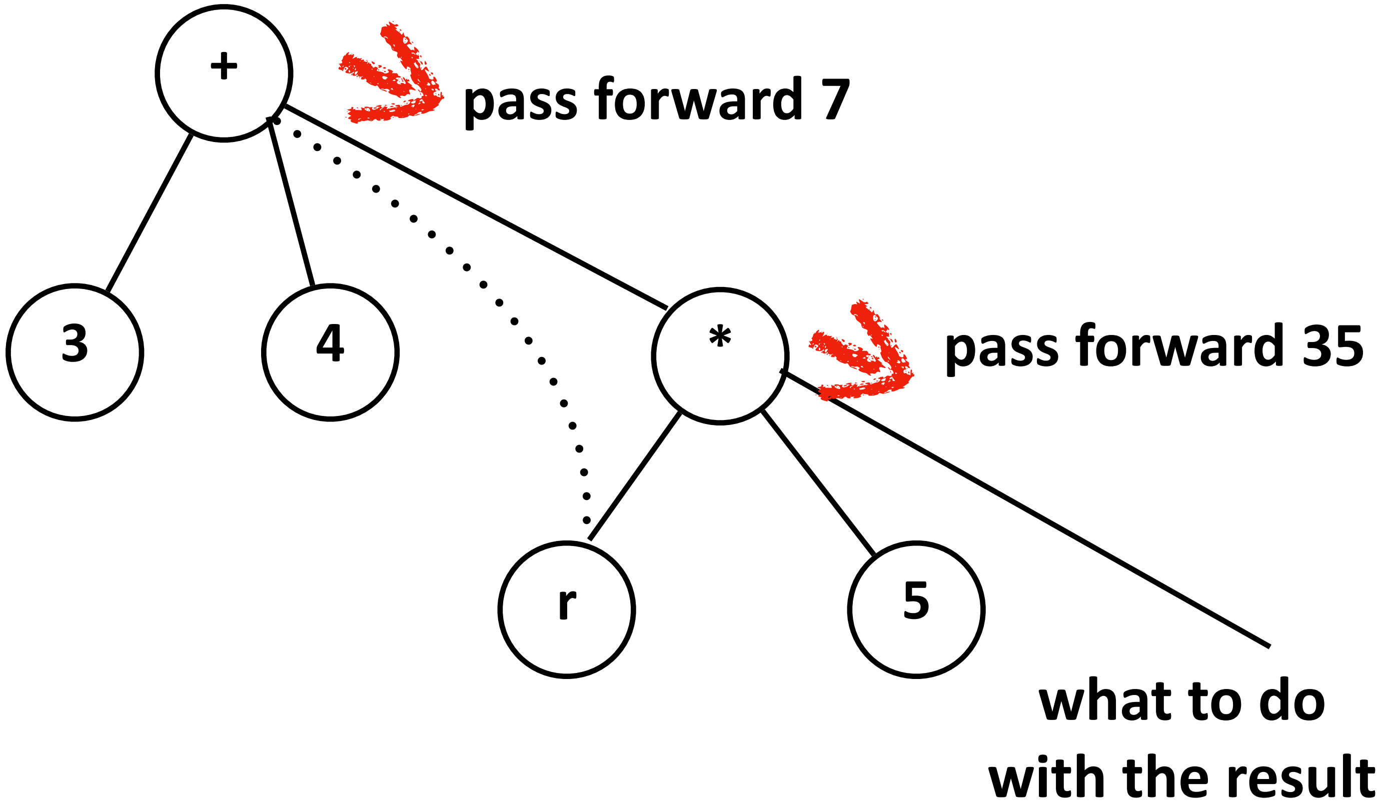


Evaluating (3+4)*5



return 35

return 7



pass forward 7

pass forward 35

what to do
with the result

Simple cps functions

```
fun add (x,y,k) = k (x + y)
```

continuation

```
fun mult (x,y,k) = k (x * y)
```

We could write them in curried form as well.

Alternatively

```
fun add (x,y) k = k (x + y)
```

```
fun mult (x,y) k = k (x * y)
```

Type of add

```
fun add (x,y,k) = k (x + y)
```

- `add :  int * int * (int -> 'a) -> 'a`

the overall return type is the return type of the continuation

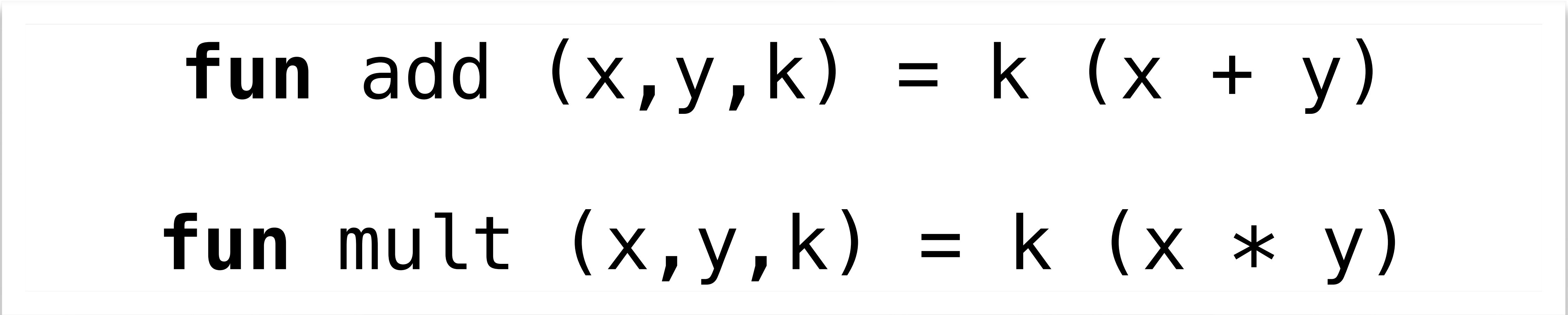
Using a cps function

```
fun add (x,y,k) = k (x + y)
```

↑

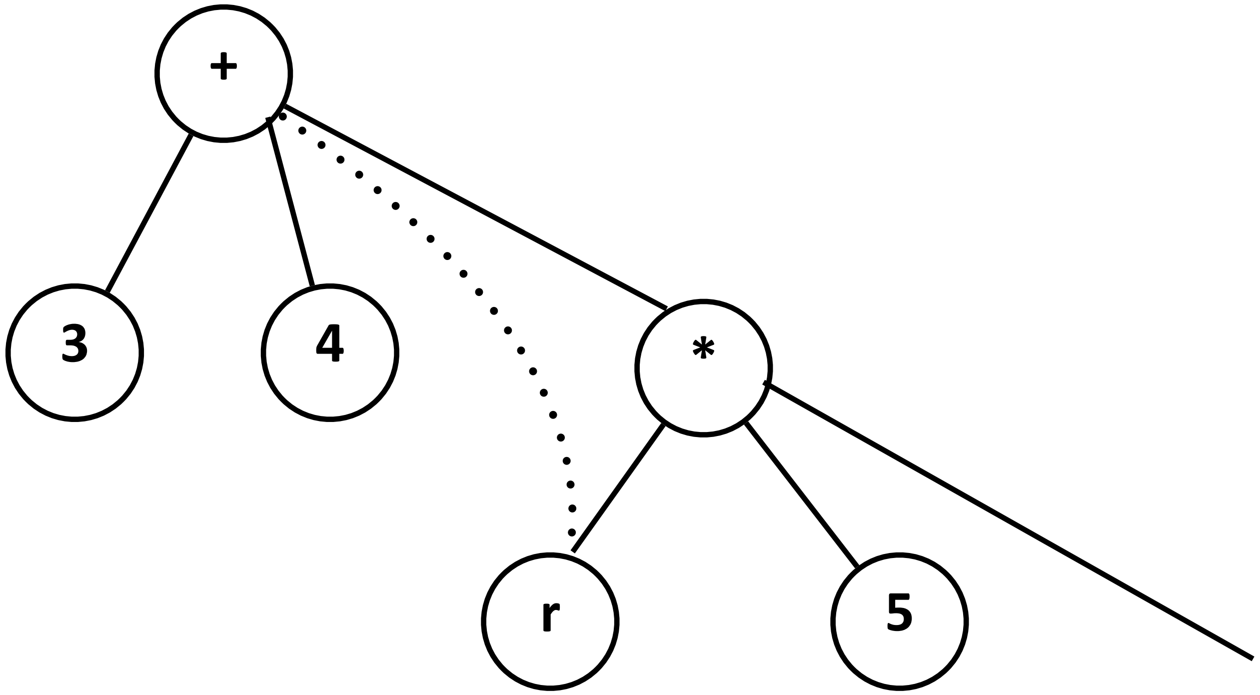
continuation

```
add (3, 4, fn r => r)
=> [3/x,4/y,(fn r => r)/k] k (x + y)
=> (fn r => r) (3 + 4)
=> (fn r => r) (7)
=> […7/r] r
=> 7
```



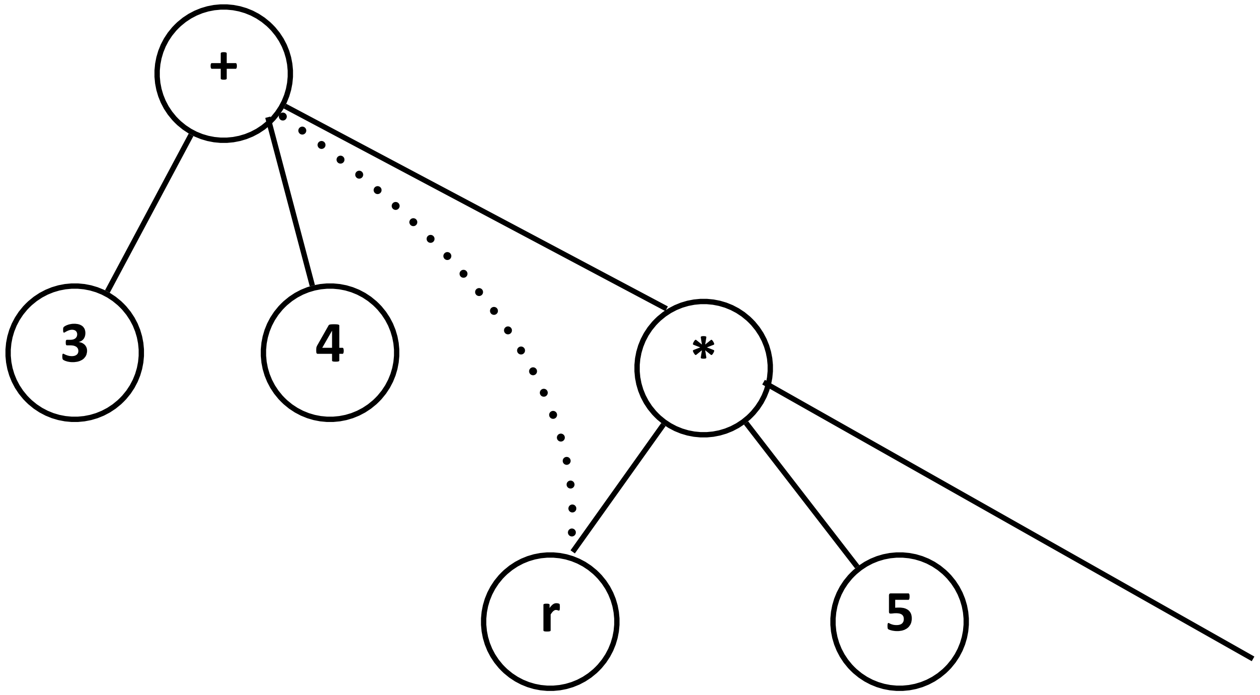
```
fun add (x,y,k) = k (x + y)

fun mult (x,y,k) = k (x * y)
```

cps function to compute (3+4)*5



```
fun add (x,y,k) = k (x + y)

fun mult (x,y,k) = k (x * y)
```

cps function to compute (3+4)*5



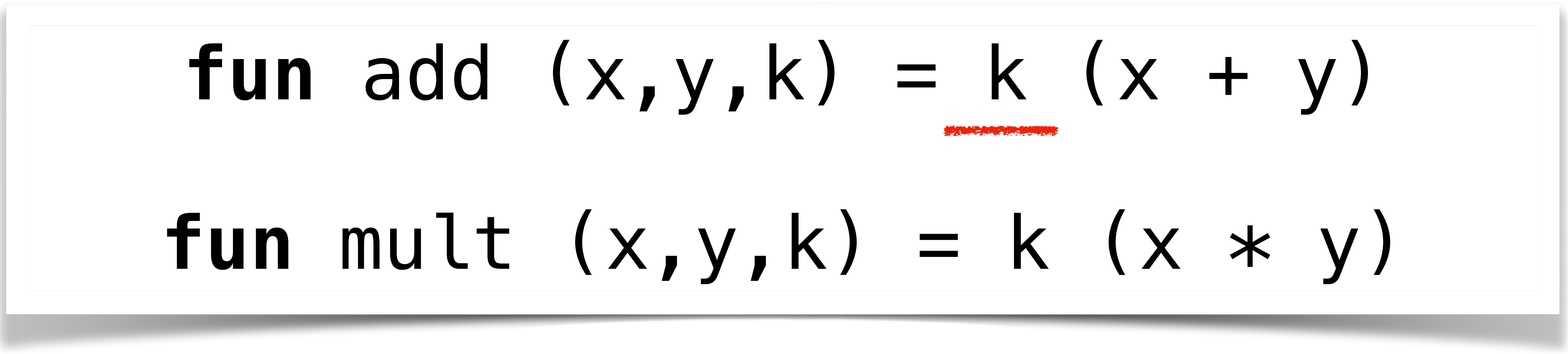
```
add (3, 4, fn r => mult (r,5, _______ ))
```

```
fun add (x,y,k) = k (x + y)

fun mult (x,y,k) = k (x * y)
```

cps function to compute (3+4)*5

```
add (3, 4, fn r => mult (r,5, fn x=>x))
```

What if we wanted to return the result as a string?



```
fun add (x,y,k) = k (x + y)

fun mult (x,y,k) = k (x * y)
```

cps function to compute (3+4)*5

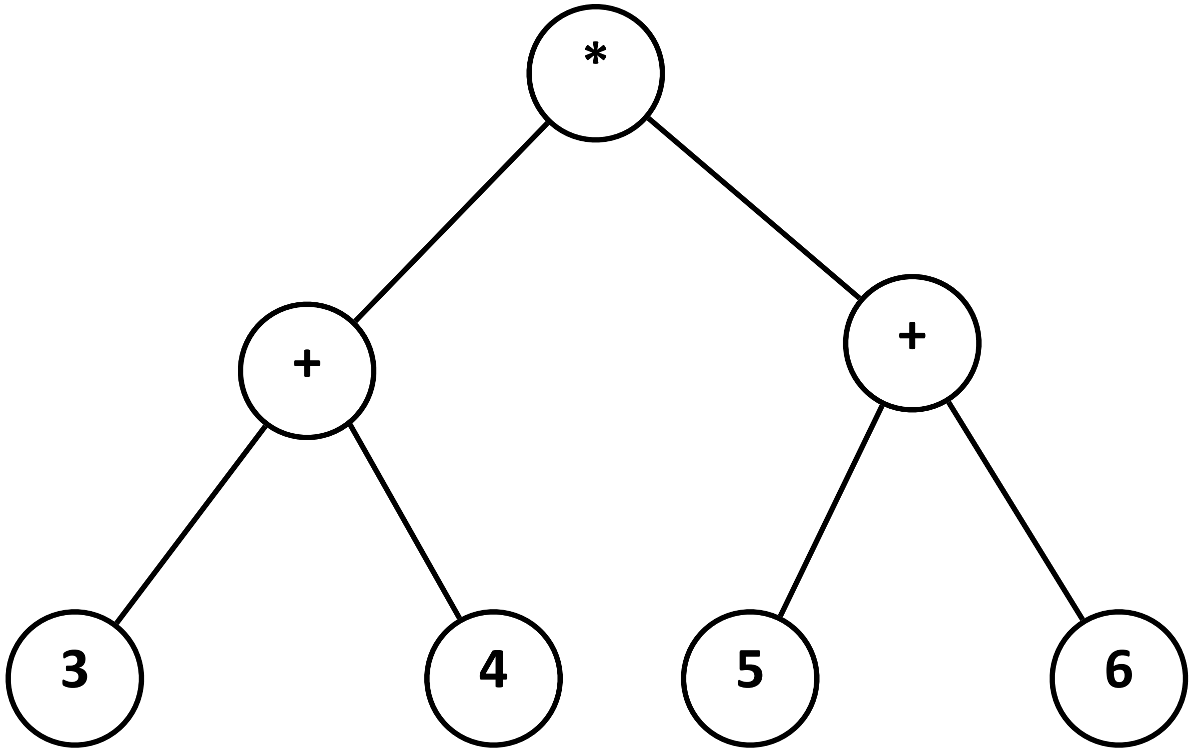
```
add (3, 4, fn r => mult(r,5,Int.toString))


==> (fn r => mult (r, 5, Int.toString))(3 + 4)
==> [7/r] mult (r, 5, Int.toString)
==> [7/r] Int.toString(r*5)
==> Int.toString(35) ==> "35"
```

```
fun add (x,y,k) = k (x + y)

fun mult (x,y,k) = k (x * y)
```

```
(3 + 4) * (5 + 6)
```

```
add (3,4, fn r1 =>
              add (5, 6, fn r2 =>
                             mult (r1, r2, fn r3 => r3)))
```

`(3 + 4) * (5 + 6)`



```
 add 3 4 r1
 add 5 6 r2
 mult  r1 r2 r3
return   r3
```

`add` 3 and 4 and put the result in `r1` …

makes control flow and intermediate results explicit

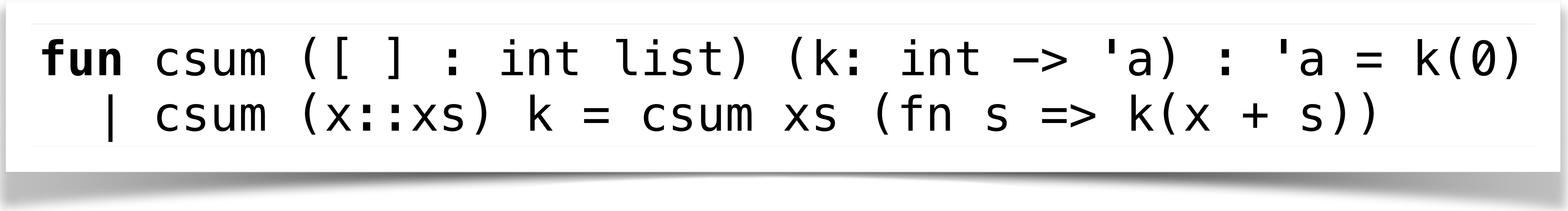
```
(* sum : int list -> int
   REQUIRES: true
   ENSURES:  sum L returns the sum of all the elements in L
*)

    fun sum ([ ] : int list) : int = 0
      | sum (x::xs) = x + sum xs

(* Using tail-recursion: *)
(* tsum : int list * int -> int
   REQUIRES: true
   ENSURES:  tsum (L, acc) ≅ (sum L) + acc
*)
   fun tsum ([ ] : int list, acc : int) : int = acc
     | tsum (x::xs, acc) = tsum(xs, x + acc)


(* Using foldr: *)
   val Lsum = foldr (op +) 0
```

```
(* sum : int list -> int
   REQUIRES: true
   ENSURES:  sum L returns the sum of all the elements in L
*)
    fun sum ([ ] : int list) : int = 0
      | sum (x::xs) = x + sum xs

(* Using tail-recursion: *)
(* tsum : int list * int -> int
   REQUIRES: true
   ENSURES:  tsum (L, acc) ≅ (sum L) + acc
*)
   fun tsum ([ ] : int list, acc : int) : int = acc
     | tsum (x::xs, acc) = tsum(xs, x + acc)
(* Using continuation-passing style: *)
(* csum : int list -> (int -> 'a) -> 'a
   REQUIRES: k is total
   ENSURES:  csum L k ≅ k (sum L)
*)
```

```
(* sum : int list -> int
   REQUIRES: true
   ENSURES:  sum L returns the sum of all the elements in L
*)
    fun sum ([ ] : int list) : int = 0
      | sum (x::xs) = x + sum xs

(* Using tail-recursion: *)
(* tsum : int list * int -> int
   REQUIRES: true
   ENSURES:  tsum (L, acc) ≅ (sum L) + acc
*)
   fun tsum ([ ] : int list, acc : int) : int = acc
     | tsum (x::xs, acc) = tsum(xs, x + acc)
(* Using continuation-passing style: *)
(* csum : int list -> (int -> 'a) -> 'a
   REQUIRES: k is total
   ENSURES:  csum L k ≅ k (sum L)
*)

   fun csum ([ ] : int list) (k: int -> 'a) : 'a = _______
     | csum (x::xs) k = __________________________________
```

```
(* sum : int list -> int
   REQUIRES: true
   ENSURES:  sum L returns the sum of all the elements in L
*)

    fun sum ([ ] : int list) : int = 0
      | sum (x::xs) = x + sum xs

(* Using tail-recursion: *)
(* tsum : int list * int -> int
   REQUIRES: true
   ENSURES:  tsum (L, acc) ≅ (sum L) + acc
*)
   fun tsum ([ ] : int list, acc : int) : int = acc
     | tsum (x::xs, acc) = tsum(xs, x + acc)
(* Using continuation-passing style: *)
(* csum : int list -> (int -> 'a) -> 'a
   REQUIRES: k is total
   ENSURES:  csum L k ≅ k (sum L)
*)

   fun csum ([ ] : int list) (k: int -> 'a) : 'a =  k(0)
     | csum (x::xs) k = ________________________________
```

```
(* sum : int list -> int
   REQUIRES: true
   ENSURES:  sum L returns the sum of all the elements in L
*)

    fun sum ([ ] : int list) : int = 0
      | sum (x::xs) = x + sum xs

(* Using tail-recursion: *)
(* tsum : int list * int -> int
   REQUIRES: true
   ENSURES:  tsum (L, acc) ≅ (sum L) + acc
*)
   fun tsum ([ ] : int list, acc : int) : int = acc
     | tsum (x::xs, acc) = tsum(xs, x + acc)
(* Using continuation-passing style: *)
(* csum : int list -> (int -> 'a) -> 'a
   REQUIRES: k is total
   ENSURES:  csum L k ≅ k (sum L)
*)

   fun csum ([ ] : int list) (k: int -> 'a) : 'a = k(0)
     | csum (x::xs) k = csum xs (fn s => k(x + s))
```



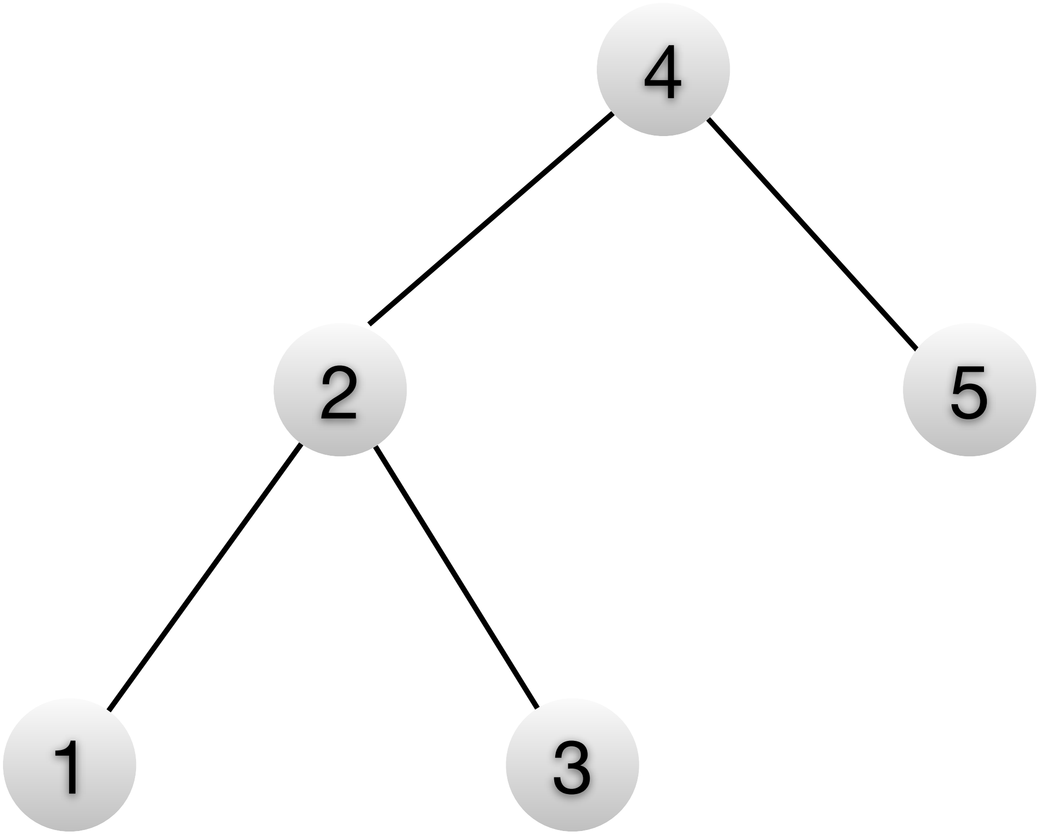
```
fun csum ([ ] : int list) (k: int -> 'a) : 'a = k(0)
  | csum (x::xs) k = csum xs (fn s => k(x + s))
```

```
   csum [2,3] (fn s => s)

≅  csum [3]  (fn s’ => (fn s => s)(2 + s’))

≅  csum []  (fn s’’=> (fn s’ => (fn s => s)(2 + s’)) (3 + s’’))

≅  (fn s’’=> (fn s’ => (fn s => s) (2 + s’)) (3 + s’’)) 0

≅  (fn s’ => (fn s => s) (2 + s’)) (3 + 0))

≅  (fn s => s) (2 + 3)

≅  5
```

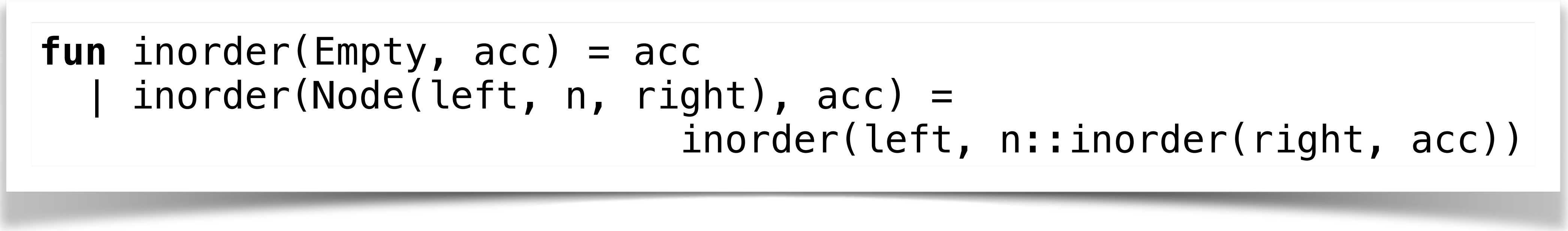
Comparing tree contents to list contents

```
datatype tree = Empty | Node of tree * int * tree

(* inorder :  tree * int list --> int list
   REQUIRES: true
   ENSURES:  inorder(T, acc) ≅ L @ acc, where L consists of the
             elements of T as encountered in an in-order traversal of T.
*)
fun inorder(Empty, acc) = acc
  | inorder(Node(left, n, right), acc) =
                              inorder(left, n::inorder(right, acc))


(* treematch : tree -> int list -> bool
   REQUIRES: k is total
   ENSURES:  treematch T L returns true if L consists of the
             elements of T as encountered in an in-order traversal of T,
             and returns false otherwise.
*)
```

Suppose T is bound to the following tree:



```
val true = treematch T [1,2,3,4,5]

val false = treematch T [1,4,5,2]
```



```
fun inorder(Empty, acc) = acc
  | inorder(Node(left, n, right), acc) =
                          inorder(left, n::inorder(right, acc))
```

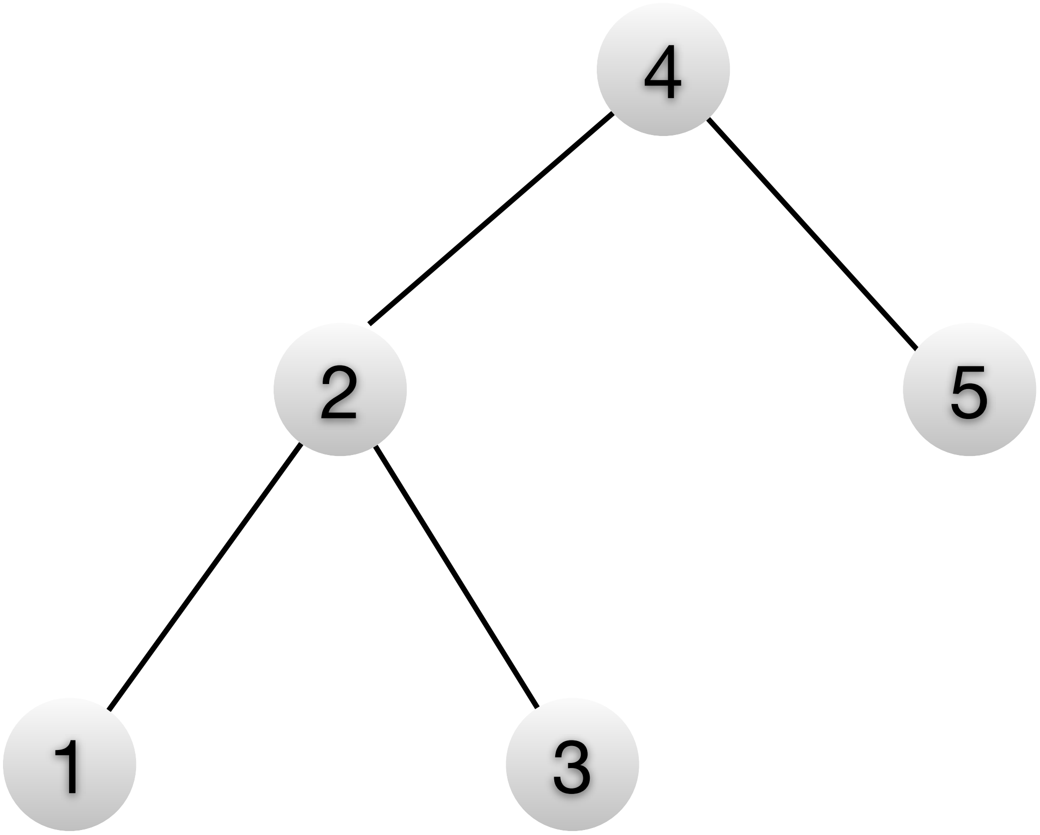
```
(* same : int list * int list -> bool *)
```

```
fun same ([ ],[ ]) = true
  | same (x::xs, y::ys) = (x = y) andalso same(xs,ys)
  | same _ = false
```

```
(* treematch : tree -> int list -> bool
   REQUIRES: k is total
   ENSURES:  treematch T L returns true if L consists of the
             elements of T as encountered in an in-order
             traversal of T, and returns false otherwise.
*)
```

```
fun treematch T L =  same(inorder(T, nil) ,L))
```

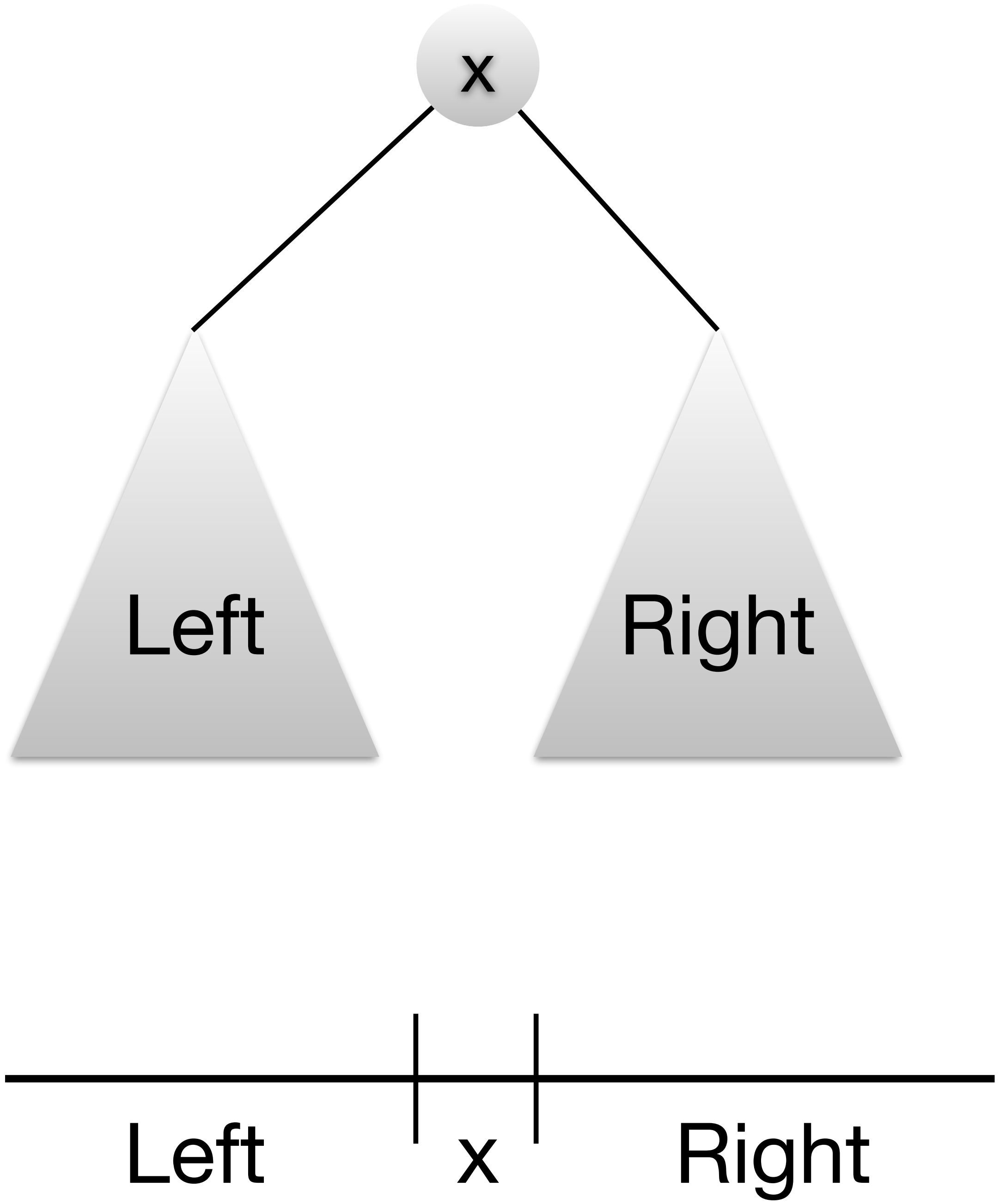
Suppose T is bound to the following tree:



Could we make treematch be faster?

```
val false = treematch T [1,4,5,2]
```

```
prefix : tree -> int list -> (int list -> bool) -> bool
```



Matching tree contents to prefix of a list

```
(* prefix : tree -> int list -> (int list -> bool) -> bool

   REQUIRES: k is total
   ENSURES:  prefix T L k ==>  true,  if L ≅ L1 @ L2, such that
             the inorder traversal of T is equal to L1, and k(L2) ≅ true.
             false, otherwise.

 *)


 fun prefix (Empty) L k = k(L)
   | prefix (Node(left, n, right)) L k = prefix left L _________________


(* treematch' : tree -> int list -> bool
   REQUIRES: true
   ENSURES:  treematch' T L returns true if L consists of the
             elements of T as encountered in an in-order traversal of T,
             and returns false otherwise.
*)

 fun treematch' T L =  prefix T L List.null
```

Matching tree contents to prefix of a list

```
(* prefix : tree -> int list -> (int list -> bool) -> bool


   REQUIRES: k is total
   ENSURES:  prefix T L k ==>  true,  if L ≅ L1 @ L2, such that
             the inorder traversal of T is equal to L1, and k(L2) ≅ true.
             false, otherwise.

*)


fun prefix (Empty) L k = k(L)
  | prefix (Node(left, n, right)) L k =

        prefix left L (fn []  => false
                       |(y::ys) => (n=y) andalso (prefix right ys k))

(* treematch' : tree -> int list -> bool
   REQUIRES: true
   ENSURES:  treematch' T L returns true if L consists of the
             elements of T as encountered in an in-order traversal of T,
             and returns false otherwise.
*)


fun treematch' T L =  prefix T L List.null
```

Search problems

- Implemented using 2 continuations:
  - success (what to do if search is successful)
  - failure (takes unit as argument, implements backtracking)

```
search : ('a -> bool) -> 'a tree -> ('a -> 'b) -> (unit -> 'b) -> 'b
```

success continuation

failure continuation

```
(* search : ('a -> bool) -> 'a tree -> ('a -> 'b) -> (unit -> 'b) -> 'b
   REQUIRES: p, sc, and f are total.
   ENSURES:  search p T sc fc ≅ sc(x), if p(x) ≅ true for some x in T
                              ≅ fc(),  otherwise
            (if more than one x satisfies p(x) ≅ true, then use the first
             encountered in a pre-order traversal of T).
*)

  fun search p Empty sc fc = fc()
    | search p (Node(left, x, right)) sc fc =
                             if p(x) then sc(x)
                             else
                               search p left sc (fn () => search p right sc fc)

(* findeven : int tree -> string
   REQUIRES: true
   ENSURES:  findeven(T) returns the string representation of the
                   first even integer found in a pre-order traversal of T,
                   if there is such an integer.  Otherwise, findeven(T)
                   returns "none found".
*)


fun findeven T = search (fn n => n mod 2 = 0) T Int.toString (fn () => "none found")
```

```
(* search : ('a -> bool) -> 'a tree -> ('a -> 'b) -> (unit -> 'b) -> 'b
   REQUIRES: p, sc, and f are total.
   ENSURES:  search p T sc fc ≅ sc(x), if p(x) ≅ true for some x in T
                              ≅ fc(),  otherwise
            (if more than one x satisfies p(x) ≅ true, then use the first
             encountered in a pre-order traversal of T).
*)

  fun search p Empty sc fc = fc()
    | search p (Node(left, x, right)) sc fc =
                              if p(x) then sc(x)
                              else
                                search p left sc (fn () => search p right sc fc)

(* findeven : int tree -> string
   REQUIRES: true
   ENSURES:  findeven(T) returns SOME x where x is the
                   first even integer found in a pre-order traversal of T,
                   if there is such an integer.  Otherwise, findeven(T)
                   returns NONE.
*)


   fun findeven T =   search (fn n => n mod 2 = 0) T SOME (fn () => NONE)
```