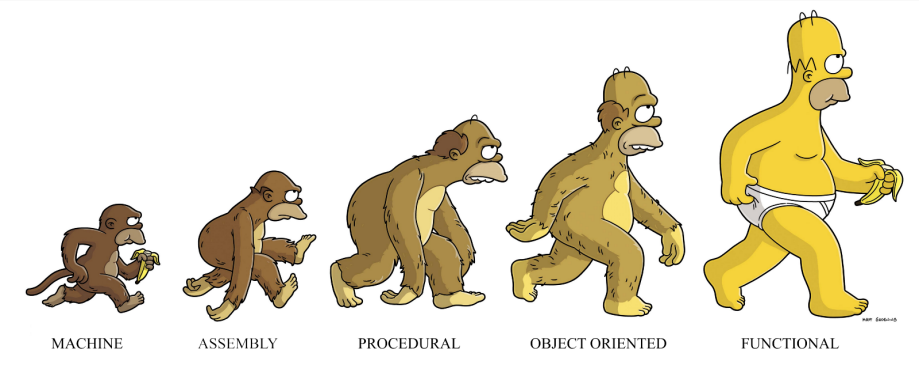
COMP302: Programming Languages and Paradigms

Week 7: Continuations - Part 1

Prof. Brigitte Pientka <bpientka@cs.mcgill.ca>

School of Computer Science, McGill University



Functional Tidbit: Programming with Style!

Get ready for

continuations!

What is a continuation?

A continuation is a representation of the execution state of a program (for example a call stack) at a certain point in time.

Save the current state of execution into some object and restore the state from this object at a later point in time resuming its execution.

First-class Support for Continuations

C#	`async / wait`
Racket	`call-with-current-continuation`
Ruby	`callcc`
Scala	`shift / reset`
Scheme	`callcc`

TODAY: Functions as Continuations

Tail-Recursion Revisited

A function is tail-recursive if no computation occurs after any recursive call.

Consequence: the compiler can reuse stack frames, leading to $O(1)$ (stack) space complexity.

Can every function be written tail-recursively?

Yes, using continuations!

Revisiting Append

```
1 (* append: 'a list -> 'a list -> 'a list *)
2 let rec append l k = match l with
3   | [] -> k
4   | h::t -> h::append t k
```

- not tail-recursive, since we use `h::append t k` (consing `h` occurs *after* the recursive call).
- Nonetheless, quite efficient.

Can we rewrite it tail-recursively?

Recipe: How to re-write a function tail-recursively?

- Add an additional argument, a continuation, which acts like an accumulator
- In the base case, we call the continuation
- In the recursive case, we build up in the continuation the work to do *after* the recursive call.

A continuation is a stack of functions modelling the call stack, i.e. the work we still need to do upon returning.

Let's see what this means in practice!

Implementing a tail-recursive append:

```
1  let rec app' l k =
2
3   (* app_tr: 'a list -> 'a list -> ('a list -> 'a list) -> 'a list *)
4
5    let rec app_tr l k cont = match l with
6
7      | []    -> cont k          <- call the continuation
8
9      | h::t -> app_tr t k (fun r -> cont (h::r))   <- build up the
10                                                       continuation
11   in
12
13     app_tr l k (fun r -> r)    <- initial continuation
```

Step-by-step, how does this work?

```
1 let rec app_tr l1 l2 k =
2   match l1 with
3   | []   -> k l2
4   | h::t -> app_tr t l2 (fun r -> k (h::r))
```

Consider this example; it's the base case.

(Notice: `let id = fun x -> x` is a shorthand for the identity function here.)

```
      app_tr [] [1;2] id
⟹    id [1;2]
⟹    [1;2]
```

Nothing too strange.

Step-by-step, how does this work?

```
1 let rec app_tr l1 l2 k =
2   match l1 with
3   | []    -> k l2
4   | h::t -> app_tr t l2 (fun r -> k (h::r))
```

Now let's try with a nonempty list. This is trickier!

`app_tr [1;2] ls id`

$\Longrightarrow$ `app_tr [2] ls (fun r -> id (1 :: r))`

$\Longrightarrow$ `app_tr [] ls (fun r' -> (fun r -> id (1 :: r)) (2 :: r'))`

A "stack" of pending operations has been built up explicitly in the continuation!

What's next? – Collapse the function stack

```
app_tr [1;2] ls id
⟹ app_tr [2] ls (fun r -> id (1 :: r))
⟹ app_tr [] ls (fun r' -> (fun r -> id (1 :: r)) (2 :: r'))
```

build up function stack

A "stack" of pending operations has been built up explicitly in the continuation!

Continue with executing the function stack (the continuation)

```
⟹ (fun r -> id (1 :: r)) (2 :: ls)
⟹ id (1 :: (2 :: ls))
⟹ 1 :: 2 :: ls
```

collapse the function stack

Remark: Performance

- `append` will outperform `append_tr` on small lists.
  On (very) large lists, `append` will crash whereas `append_tr` will run.
- Both `append` and `append_tr` use $O(n)$ memory, but it's the *type* of memory used that is different.

  append uses the *stack* which is not very big (8 MiB)
  append_tr uses the *heap* which is plentiful (several GiB)
- However, the use of continuations in the form of *closures* (functions capturing an environment) incurs an extra time and space penalty.

Let's see what this means in practice! – Try it out!

```
1 (* Constructs a list of numbers. *)
2 let rec genList n acc = if n > 0 then genList (n-1) (n::acc) else acc;;
3
4 (* Experiment: *)
5
6 let l1 = genList 800000 [];;
7 let l2 = genList 400000 [];;
```

Choose your own lists and carefully try this out on the top level

1. `# append l1 l2;;`
2. `# l1 @ l2;;`

} stack overflow

3. `# app_tr l1 l2;;` → succeed

Take-Away

- A continuation is a functional argument representing the work that still needs to be done when we finished recursively traversing our data; it represents the call stack that is normally built up by the runtime system – but you're in control!

Recap: how to convert to continuation-passing style

1. Change the type signature: add the continuation.

   e.g. `append : 'a list -> 'a list -> 'a list.`

   `app_tr : 'a list -> 'a list -> ('a list -> 'r) -> 'r.`

2. All the work that should happen after the recursive call gets moved inside the continuation; e.g.

```
| x :: xs ->
  let ys = append xs l2 in
  x :: ys
```

   becomes

```
| x :: xs ->
  app_tr xs l2
    (fun ys -> x :: ys)
```

What we use continuations for

- Tail-recursion: the continuation is a functional accumulator; it represents the call stack built when recursively calling a function and builds the final result
- Failure Continuation: the continuation keeps track of what to do upon failure and defers control to the continuation
- Success Continuation: the continuation keeps track of what to do upon success, defers control to the continuation, and builds the final result

Functional Programming Ninjas [OPTIONAL]

Implement `app_tr_gen` such that we build the call stack upfront while traversing the first list and return in each of the cases a function of type `('a list -> 'r) -> 'a list -> 'r`.

To make it a bit easier, we already give you a code template to fill in below.

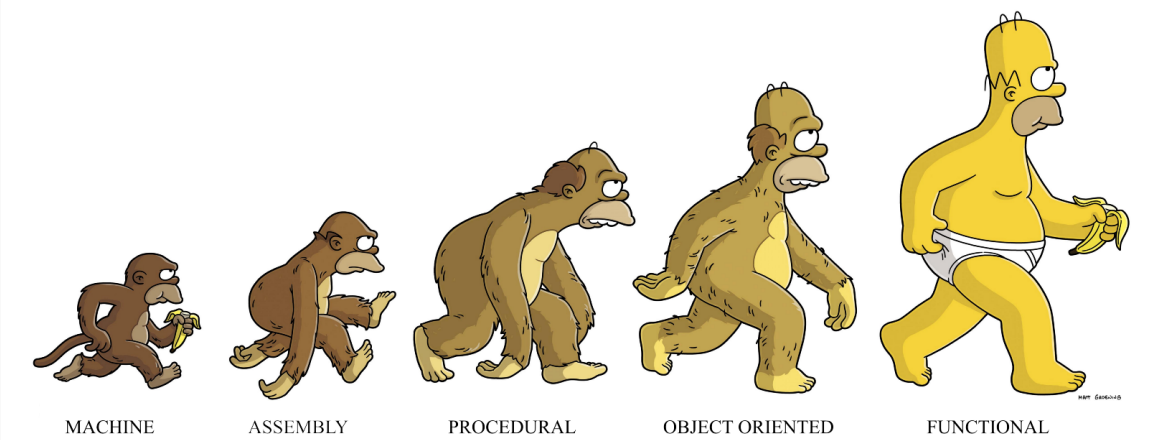
```
1  (* app_tr_gen : 'a list -> ('a list -> 'r) -> 'a list -> 'r *)
2  let rec app_tr_gen l1 = match l1 with
3
4    | [] -> fun cont l2 -> ____cont l2____
5
6    | h::t ->
7       let cc = app_tr_gen t in
8       (* NOTE: cc : ('a list -> 'r) -> 'a list -> 'r *)
9
10      ____fun cont l2 -> cc cont t____
```

COMP302: Programming Languages and Paradigms

Week 7: Continuations - Part 2

Prof. Brigitte Pientka <bpientka@cs.mcgill.ca>

School of Computer Science, McGill University



Functional Tidbit: Programming with Style!

Let's

continue!

What is a continuation?

A continuation is a representation of the execution state of a program (for example a call stack) at a certain point in time.

Save the current state of execution into some object and restore the state from this object at a later point in time resuming its execution.

When to use continuations?

- Tail-recursion: the continuation is a functional accumulator; it represents the call stack built when recursively calling a function and builds the final result
- Failure Continuation: the continuation keeps track of what to do upon failure and defers control to the continuation
- Success Continuation: the continuation keeps track of what to do upon success, defers control to the continuation, and builds the final result

How to handle success and failure?

Finding an element in a binary tree

```
(* VERSION 1: Using option types *)
let rec find p t = match t with
  | Empty -> None
  | Node (l, d, r) ->
   if  (p d) then Some d
   else (match find p l with
     | None -> find p r
     | Some d' -> Some d')
```

```
(* VERSION 2: Exceptions *)
let rec find_exc p t = match t with
  | Empty -> raise Fail
  | Node (l,d,r) ->
    if (p d) then Some d
    else (try find_exc p l with Fail -> find_exc p r)

let find_ex p t =  (try find_exc p t with Fail -> None)
```

How to handle success and failure?

Finding an element in a binary tree

```
1  (* VERSION 3: Success & failure continuation *)
2             ('a -> bool) -> 'a tree -> (unit -> 'b) -> ('a -> 'b) -> 'b
3  let rec find_tr p t fail succeed = match t with
4
5    | Empty -> fail ()                          (* call the failure continuation *)
6
7    | Node(_, d, _) when p d -> succeed d       (* call the success continuation *)
8
9    | Node(l, _, r) ->
10                                   (fail succeed)
11     find_tr p l (fun () -> find_tr p r) succeed
                   (* build the failure continuation *)
```

```
1  (** A driver function that calls the continuation-passing
       style function and wraps the result in an option.
2   *)
3  let find' p t =
4    find_tr p t (fun () -> None) (fun x -> Some x)
```

Deferring control using continuations vs exceptions

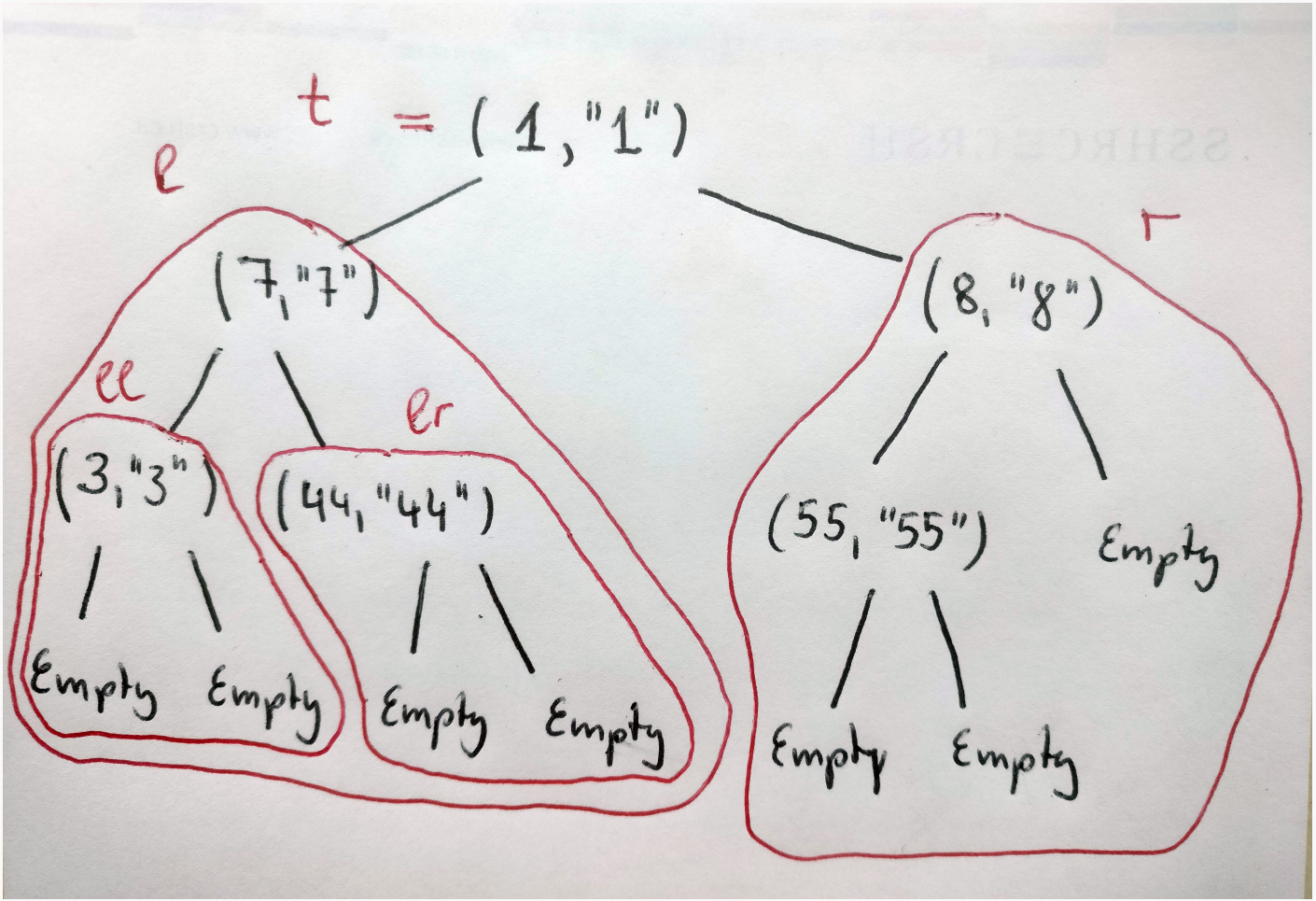
```
(* VERSION 2: Exceptions *)
let rec find_exc p t = match t with
  | Empty -> raise Fail
  | Node (l,d,r) ->
    if (p d) then Some d
    else (try find_exc p l with Fail -> find_exc p r)

let find_ex p t = (try find_exc p t with Fail -> None)
```

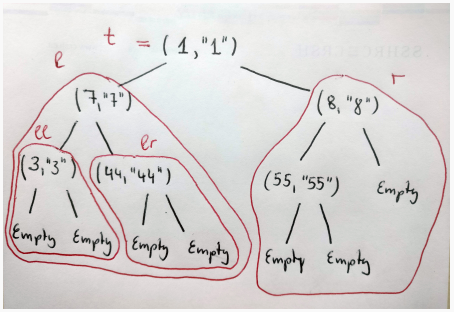
```
(* VERSION 3: Success & failure continuation *)
let rec find_tr p t fail succeed = match t with
  | Empty -> fail ()
  | Node(_, d, _) when p d -> succeed d
  | Node(l, _, r) ->
     find_tr p l
        (fun () -> find_tr p r fail succeed)
        succeed

let find' p t =
  find_tr p t (fun () -> None) (fun x -> Some x)
```

Find element in a tree



Finding elements in a tree



let is44 (x,k) = x = 44

```
1 (* VERSION 3: Success & failure continuation *)
2 let rec find_tr p t fail succeed = match t with
3   | Empty -> fail ()
4   | Node(_, d, _) when p d -> succeed d
5   | Node(l, _, r) ->
6       find_tr p l
7         (fun () -> find_tr p r fail succeed)
8         succeed
```

find_tr is44 t fc sc

⇒ find_tr is44 l (fun() → find_tr is44 r fc sc) sc (fc1)

⇒ find_tr is44 ll (fun() → find_tr is44 lr sc) sc (fc2)

⇒ find_tr is44 Empty (fun() → find_tr is44 Empty sc) sc

⇒ find_tr is44 Empty fc2 sc

⇒ find_tr is44 lr fc1 sc

⇒ sc (44, "44")

⇒ Some (44, "44")

Collecting all elements in a tree

```
1 (* VERSION 1: Straightforward recursive program          *)
2 let rec findAll p t = match t with
3   | Empty -> []
4   | Node(l,d,r) ->
5     if (p d) then (findAll p l) @(d ::(findAll p r))
6     else
7       (findAll p l) @ (findAll p r)
```

findAll : ('α -> bool) -> 'α tree -> 'α list

Collecting all elements in a tree using success continuations

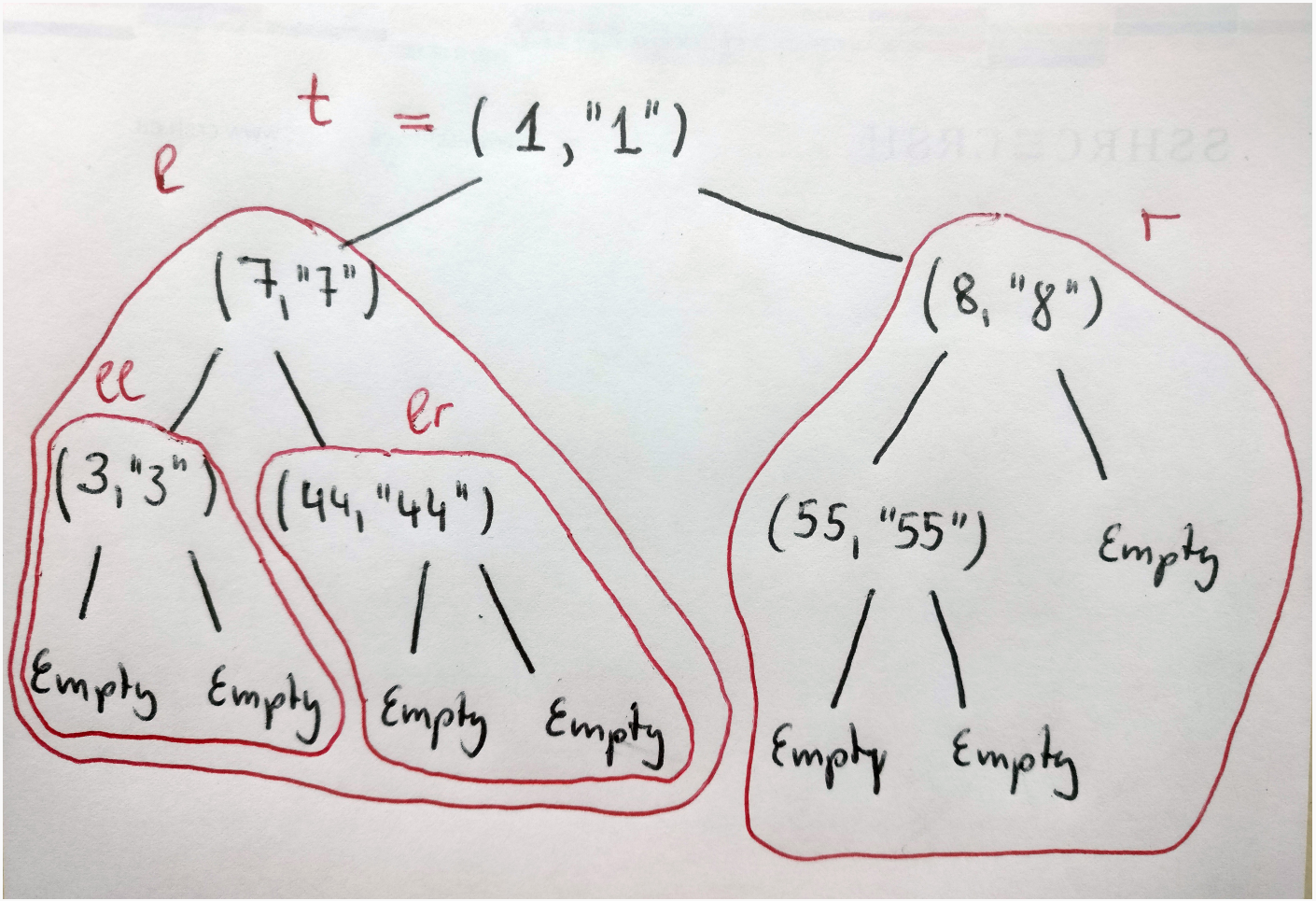
```
1  findAll : ('a -> bool) -> 'a tree -> ('a list -> 'b) -> 'b
2  let rec findAll p t sc = match t with
3
4    | Empty -> sc []
5
6    | Node(l,d,r) ->
7
8      findAll p l
9
10       (fun el -> findAll p r
11
12                  (fun er -> if (p d) then
13
14                               sc (el @ (d::er))
15
16                             else sc (el @ er)))
```

Alternative Solution

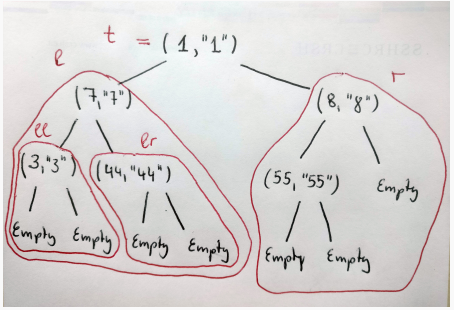
```
let rec findAll' p t sc = match t with
  | Empty -> sc []
  | Node(l,d,r) ->
     (if (p d) then
       findAll' p l
          (fun el -> findAll' p r
                        (fun er ->  sc (el@(d::er))))
     else
       findAll' p l
          (fun el -> findAll' p r
                        (fun er ->  sc (el@er)))

     )
```

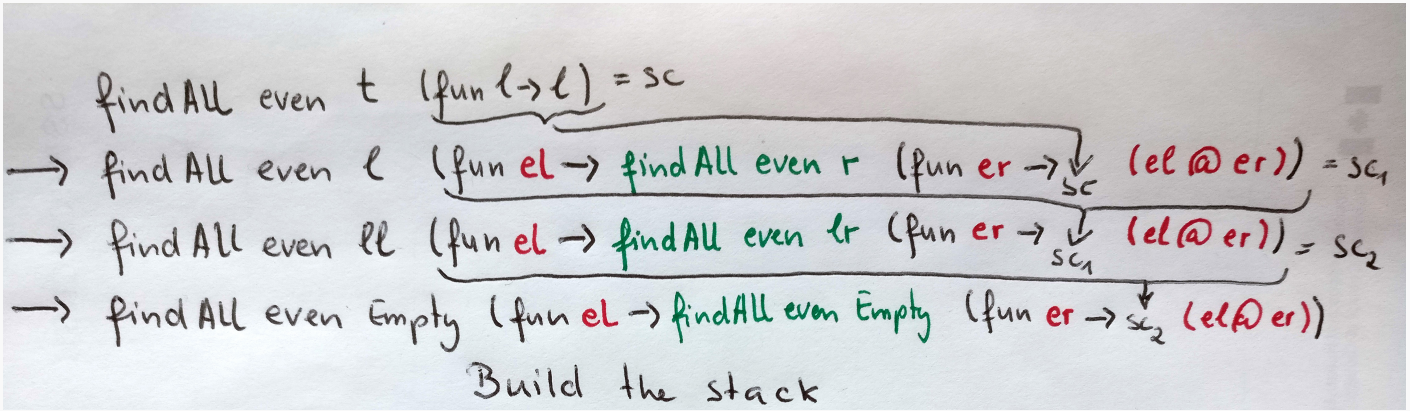
Collecting elements in a tree



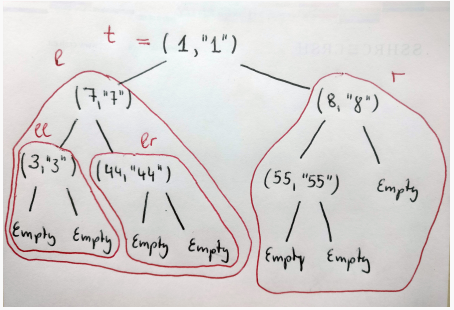
Collecting elements in a tree



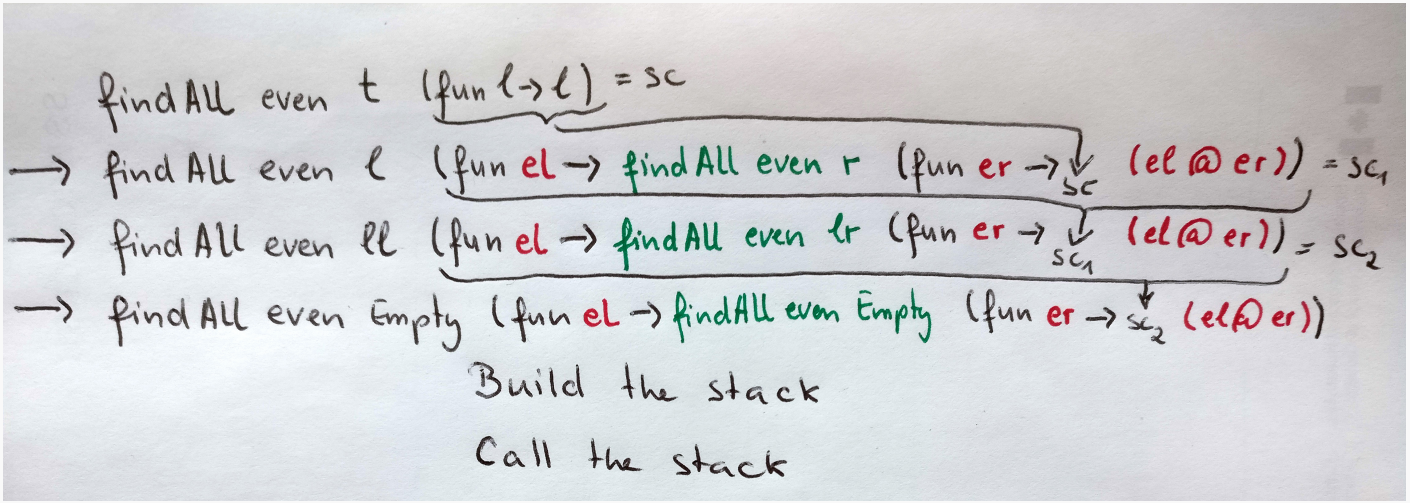

```
1  let rec findAll' p t sc = match t with
2    | Empty -> sc []
3    | Node(l,d,r) ->
4       (if (p d) then
5         findAll' p l
6            (fun el -> findAll' p r
7                          (fun er ->  sc (el@(d::er))))
8       else
9         findAll' p l
10           (fun el -> findAll' p r
11                         (fun er ->  sc (el@er)))
12
13      )
```



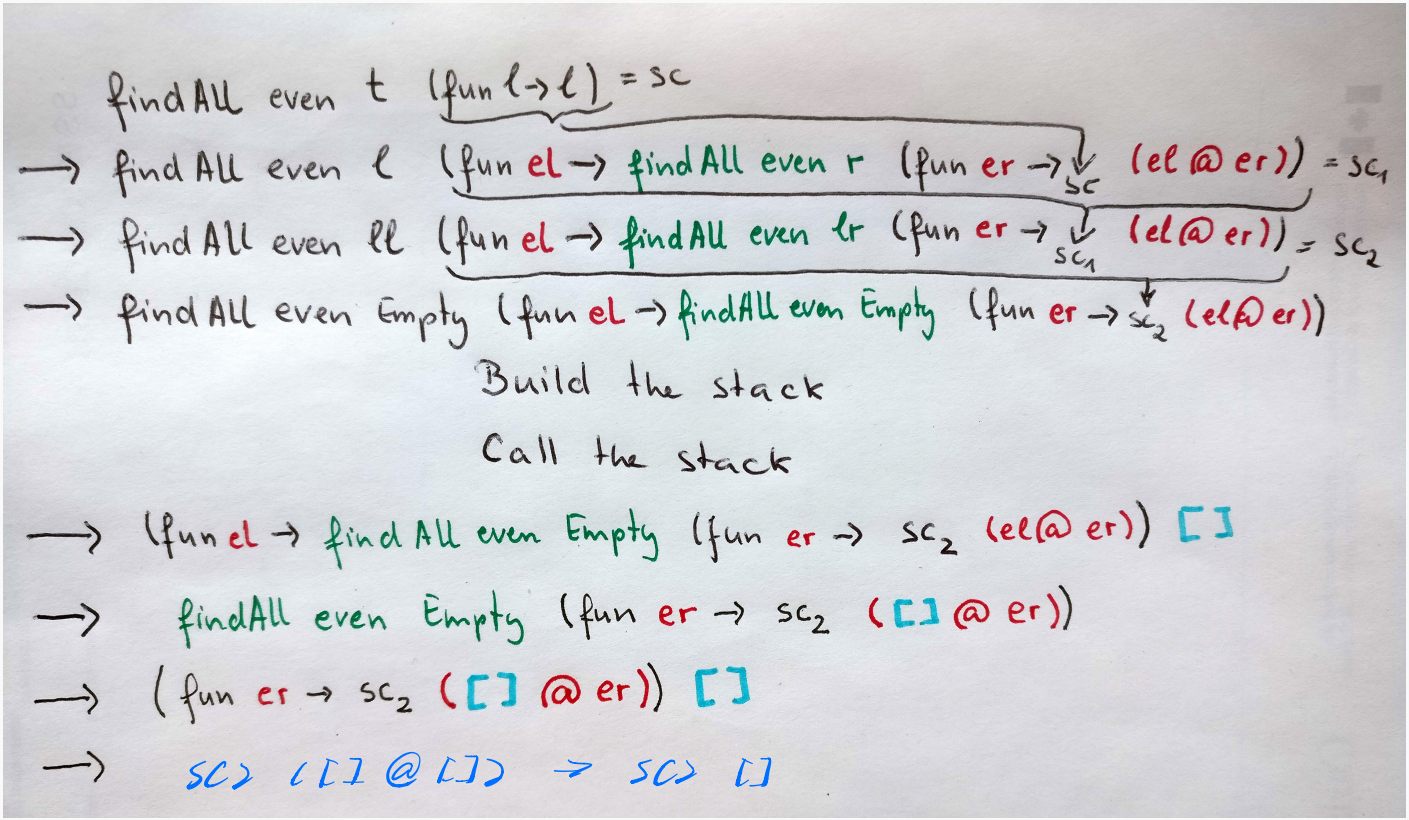
Collecting elements in a tree



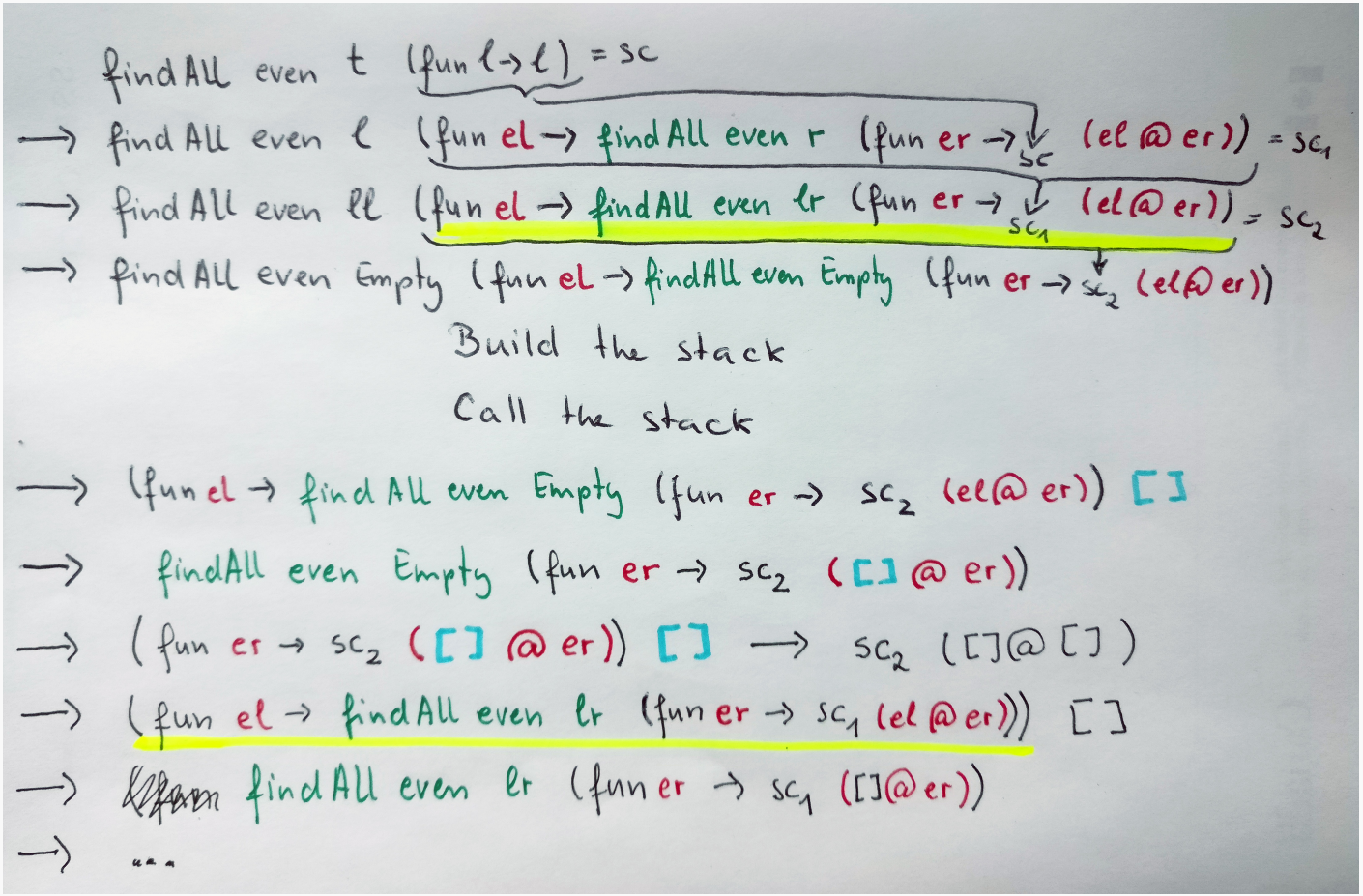

```
1  let rec findAll' p t sc = match t with
2    | Empty -> sc []
3    | Node(l,d,r) ->
4       (if (p d) then
5         findAll' p l
6            (fun el -> findAll' p r
7                          (fun er ->  sc (el@(d::er))))
8       else
9         findAll' p l
10           (fun el -> findAll' p r
11                         (fun er ->  sc (el@er)))
12
13      )
```



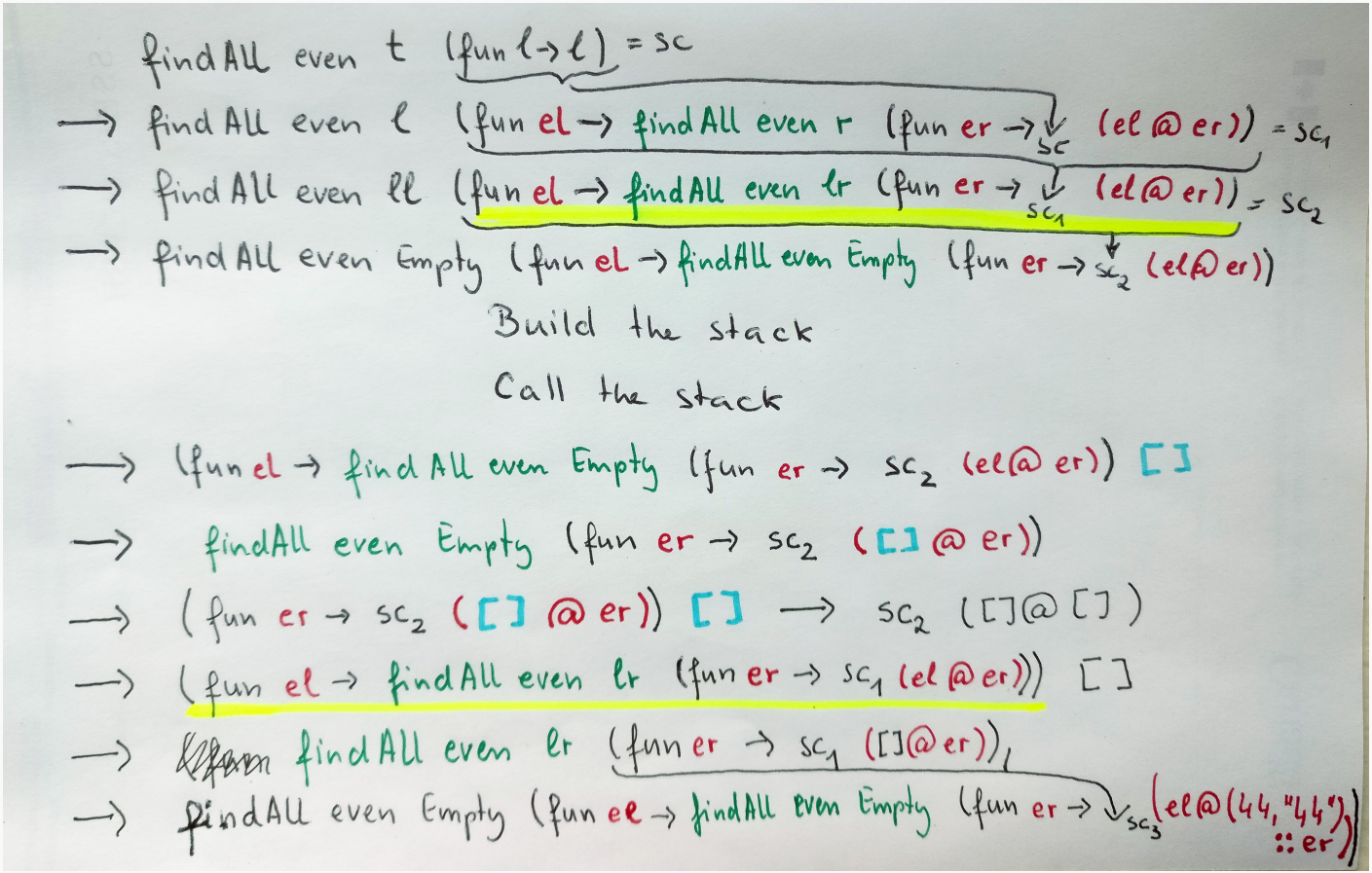
Collecting elements in a tree



Collecting elements in a tree



Collecting elements in a tree



Take-Away

- A continuation is a functional argument representing the work that still needs to be done when we finished recursively traversing our data; it represents the call stack that is normally built up by the runtime system – but you're in control!

Take-Away

- A continuation is a functional argument representing the work that still needs to be done when we finished recursively traversing our data; it represents the call stack that is normally built up by the runtime system – but you're in control!
- Tail-recursion: the continuation is a functional accumulator; it represents the call stack built when recursively calling a function and builds the final result
- Failure Continuation: the continuation keeps track of what to do upon failure and defers control to the continuation
- Success Continuation: the continuation keeps track of what to do upon success, defers control to the continuation, and builds the final result

Next

Implementing a regular expression matcher with continuations