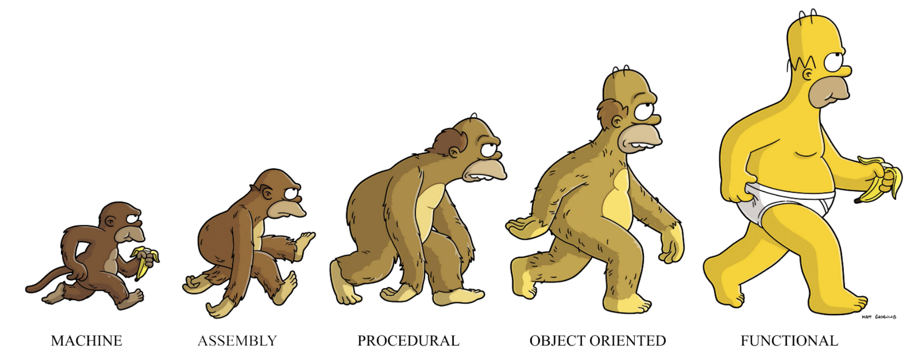
COMP302: Programming Languages and Paradigms

Week 6: Exceptions

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

School of Computer Science, McGill University



Exceptions: What are they good for?

Primary benefits:

- Force you to consider the exceptional case
- Allows you to segregate the special case from other cases in the code (avoids clutter!)
- **Diverting control flow!**

Computation and Effects

So far:

Expressions in OCaml:

- An expression has a type.
- An expression evaluates to a value (or diverges).
- An expression may have an effect (update memory, raise an exception, print, etc.).

Warm-Up: Type, Values, and Effect

Expression `3 / 0`

Type `int`

Value ✗

Effect raises run-time exception `Division_by_zero`

Expression

```
let head_of_empty_list =
  let head (x::t) = x in
    head []
```

not defined on all possible inputs

Type `'a`

Value ✗

Effect raises run-time exception `Match_failure`

In the top-level (REPL)

```
# let head (x::t) = x;;
Characters 9-19:
  let head (x::t) = x;;
           ^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
val head : 'a list -> 'a = <fun>
# head [] ;;
Exception: Match_failure ("//toplevel//", 2825, -11498).
# 3 / 0;;
Exception: Division_by_zero.
```

don't have value
⇓
type not shown.

User-defined Exceptions

```
exception Domain

let fact n =
  let rec f n =
    if n = 0 then 1
    else n * f (n-1)
  in
    if n < 0 then raise Domain
    else f(n)

let runFact n =
  try
    print_string ("Factorial of " ^  string_of_int n ^
                  " is " ^ string_of_int (fact n) ^ "\n")
  with Domain -> print_string "Error: Given input is less than 0 \n"
```

Exceptions for Backtracking: How to Handle Failure?

"Ever tried. Ever failed. No matter.
Try Again. Fail again. Fail better."

Samuel Beckett

A Binary Search Tree

```
type key = int

type 'a btree =
  | Empty
  | Node of 'a btree * (key * 'a) * 'a btree

```

Find an element in a tree (which is a binary search tree) :

Given a binary tree `t` and a key `k`,

- Return the element stored with that key
- Otherwise ...

Traversing a Binary Search Tree

```
(* find : 'a btree -> int -> 'a opt *)
let rec  find t k = match t with


  | Empty -> ____None________


  | Node(l, (k',d), r) ->


      if k = k' then ____Some  d____


      else

        (if k < k' then find l k else find r k)
```

Traversing a Binary Search Tree Using Exceptions

exception NotFound

```
(* find : 'a btree -> int -> 'a *)
let rec  find t k = match t with


  | Empty -> raise NotFound


  | Node(l, (k',d), r) ->


      if k = k' then d


      else

        (if k < k' then find l k else find r k)
```

A Binary Tree – Not Ordered!

```
type key = int

type 'a btree =
  | Empty
  | Node of 'a btree * (key * 'a) * 'a btree

```

Find an element in a tree (which is NOT a binary search tree) :

Given a binary tree `t` and a key `k`,

- Return the element stored with that key
- Otherwise . . .

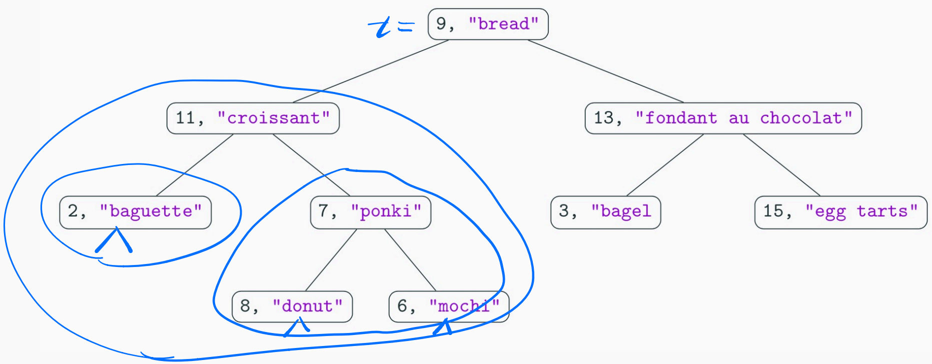
Backtracking through a tree using options

```
(* 'a btree -> key -> 'a option *)
let rec find t k = match t with
  | Empty -> None
  | Node (l, (k',d), r) ->
      if k = k' then Some d
      else


          match (find l k) with


            | None -> find r k


            | Some v -> Some v
```

Backtracking through a tree using exceptions

```
(* find : 'a btree -> int -> 'a opt *)
let rec  find t k = match t with
  | Empty -> raise NotFound
  | Node(l, (k',d), r) ->
      if k = k' then d
      else
        try find l k with NotFound -> find r k
```

Backtracking in Action



find t 7 → try find l 7 with NotFound → find r 7

find l 7 → try find ll 7 with NotFound → find lr 7

find (Node(Empty, (2, "baguette"), Empty)) 7

→ try find Empty 7 with NotFound → find Empty 7

find Empty 7 → raise NotFound

Giving Change – Another Example for Backtracking

Implement a function `change:int list -> int -> int list`. It takes in a list of coins in decreasing order and an amount, and returnse a list of "coins" (also in increasing order) which in total add up to the required amount.

Example:

```
# change [50;25;10;5;2;1] 43;;
- : int list = [25; 10; 5; 2; 1]
# change [50;25;10;5;2;1] 13;;
- : int list = [10; 2; 1]
# change [5;2;1] 13;;
- : int list = [5; 5; 2; 1]
```

Giving Change – Another Example for Backtracking

Implement a function `change:int list -> int -> int list`. It takes in a list of coins in decreasing order and an amount, and returnse a list of "coins" (also in increasing order) which in total add up to the required amount.

```
(* change: : int list -> int -> int list *)
let rec change coins amt =
  if amt = 0 then ____[]____________
  else
    begin match coins with
      | [] -> ____raise Change________
      | coin::cs -> if coin > amt then
                    ____change cs amt________
                    else
                     try ____coin:: change coins (amt-coin)
                     with Change -> ____change cs amt________
    end
```

check if this coin is useful.

How does backtracking work here?

```
change [3;2] 4
```

⇒ try 3 :: change [3;2] 1 with Change → change [2] 4 ⇒ [2;2]

change [2] 1 → change [] 1 → raise Change

change [2] 4 [2;2]

⇒ try 2 :: change [2] 2 with Change → change [] 4

[2]

try 2 :: change [2] 0 with Change → change [] 0

[] []

Take Away : Exceptions

- Force you to consider the exceptional case
- **Diverting control flow!**
- Use exception for Backtracking