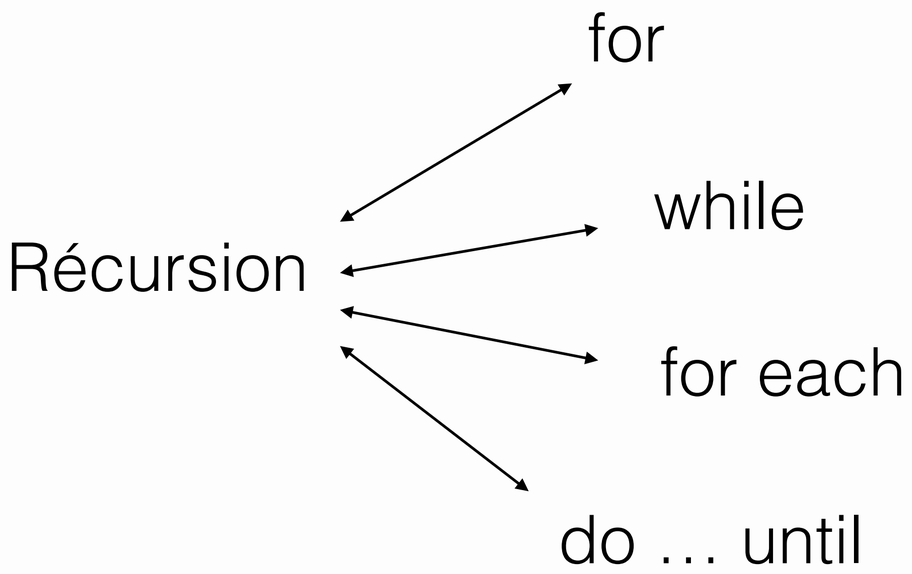
Continuations

Vincent Archambault-B
IFT 2035 - Université de Montréal

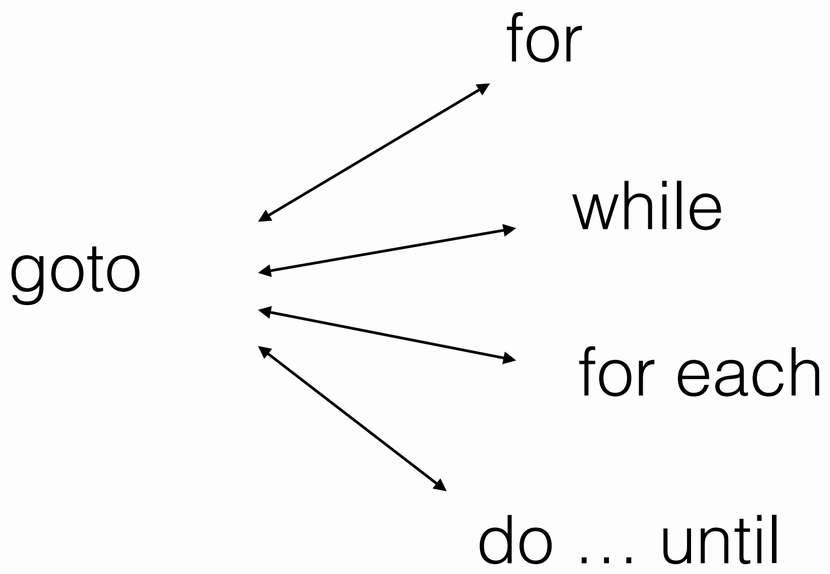
Pour obtenir le code source de ce document

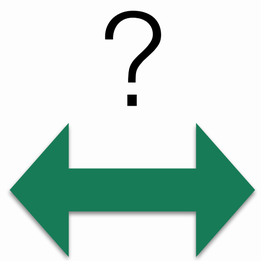
- https://github.com/archambaultv/IFT2035-UdeM
- vincent.archambault-bouffard@umontreal.ca

La récursion peut remplacer les instructions de boucle



L’instruction goto peut aussi remplacer les instructions de boucle



goto  Récursion

NON

goto ⟷ Récursion

goto (en assembleur) peut sauter n’importe où dans le programme

goto (en c) peut sauter n’importe où dans la fonction

la récursion revient au début de la fonction

goto  ?

goto  continuation

Continuation

- goto indique, à l'aide d'une adresse mémoire ou d'un label, où poursuivre le programme
- Une continuation indique, à l'aide d'une fonction, comment poursuivre le programme
- Cette fonction est souvent passée en paramètre, continuation passing style (CPS)

Conversion CPS

```haskell
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Sans continuation, map retourne à l'appelant

```haskell
map :: (a -> b) -> [a] -> ([b] -> c) -> c
map _ [] k = k []
map f (x:xs) k =
  map f xs (\r -> k (f x : r))
```

Nécessite une fermeture

Avec continuation, map poursuit avec la fonction k

Conversion CPS

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Sans continuation, map retourne à l'appelant

```
map :: (a -> b) -> [a] -> ([a] -> c) -> c
map _ [] k = k []
map f (x:xs) k =
  map f xs
  (\r -> f x (\rf -> k (rf : r)))
```

Si f est aussi en CPS

Avec continuation, map poursuit avec la fonction k

Conversion CPS

```
x = map (+ 1) [1, 2, 3]
main = putStrLn x
```

Sans continuation, map retourne à l'appelant

```
main = map (+ 1) [1, 2, 3]
           (\x -> putStrLn x)
```

Avec continuation, map poursuit avec la fonction k

Try catch

```
import Control.Exception

notZero ::  Int -> Int
notZero 0 = error "Pas de zéro"
notZero x = x

program :: Int -> IO Int
program arg =
  catch
  (let a = notZero arg
   in return (a * a))
  (\err ->
     (putStrLn (show (err :: SomeException)))
     >> return 0)
```

Avec try catch

```
notZeroCPS :: Int ->
              (Int -> a) ->
              (String -> a) ->
              a
notZeroCPS 0 _ kErr = kErr "Pas de zéro"
notZeroCPS x k _ = k x

programCPS :: Int -> IO Int
programCPS arg =
  notZeroCPS arg
  (\a -> return (a * a))
  (\err -> putStrLn err >> return 0)
```

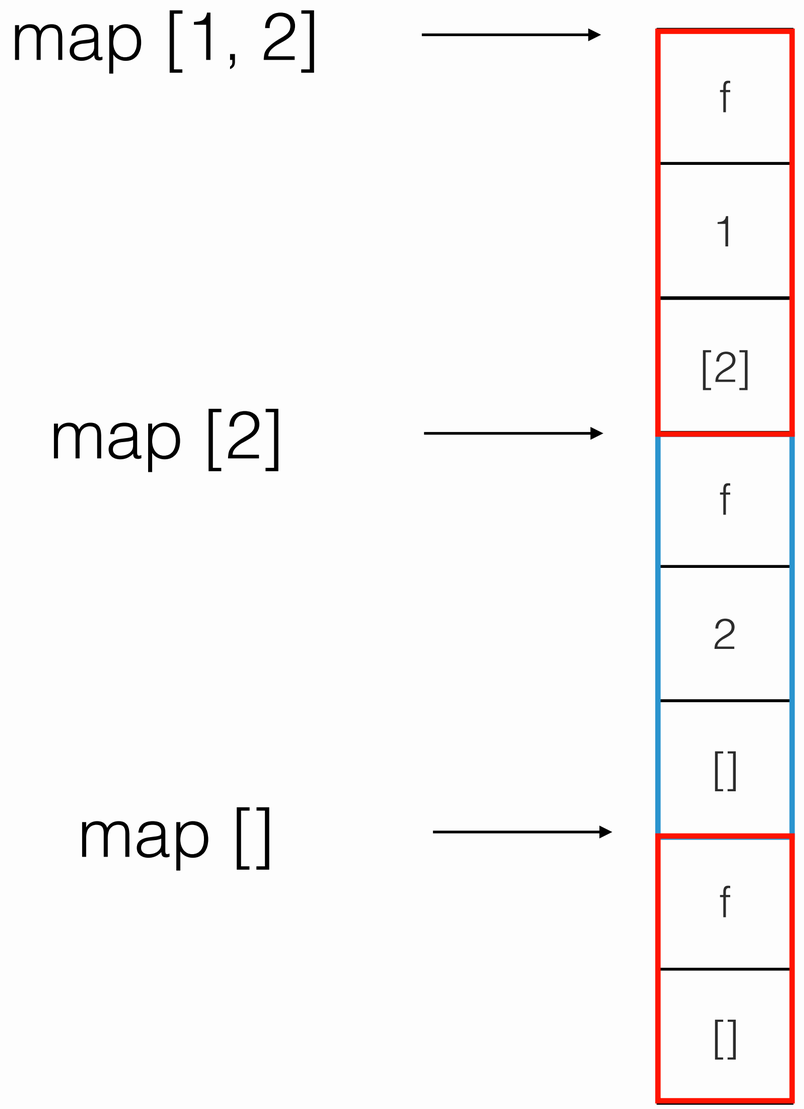
Avec continuation

Un programme en CPS n’a pas besoin de pile

- Tous les appels sont terminaux
- Les fermetures remplacent la pile

Sans continuation, utilisation d'une pile

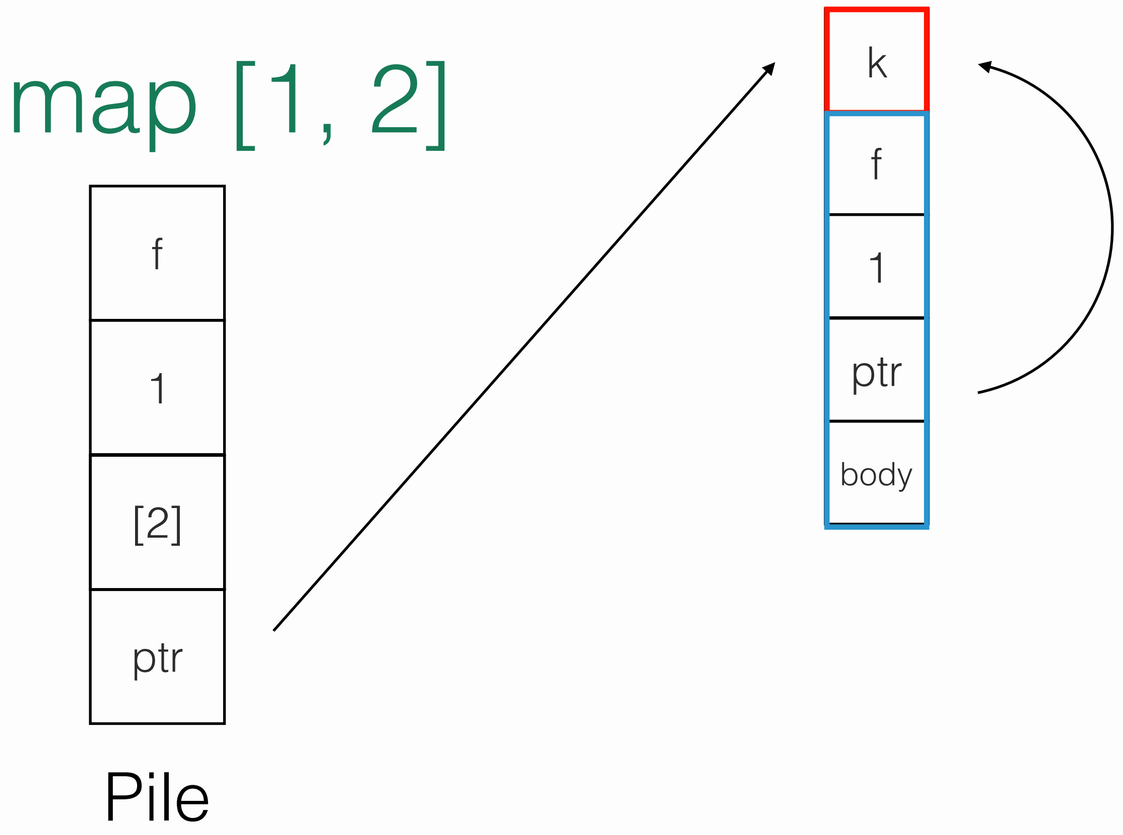
```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs

map [1, 2]
```



Sans continuation, map retourne à l'appelant

Avec continuation, utilisation des fermetures

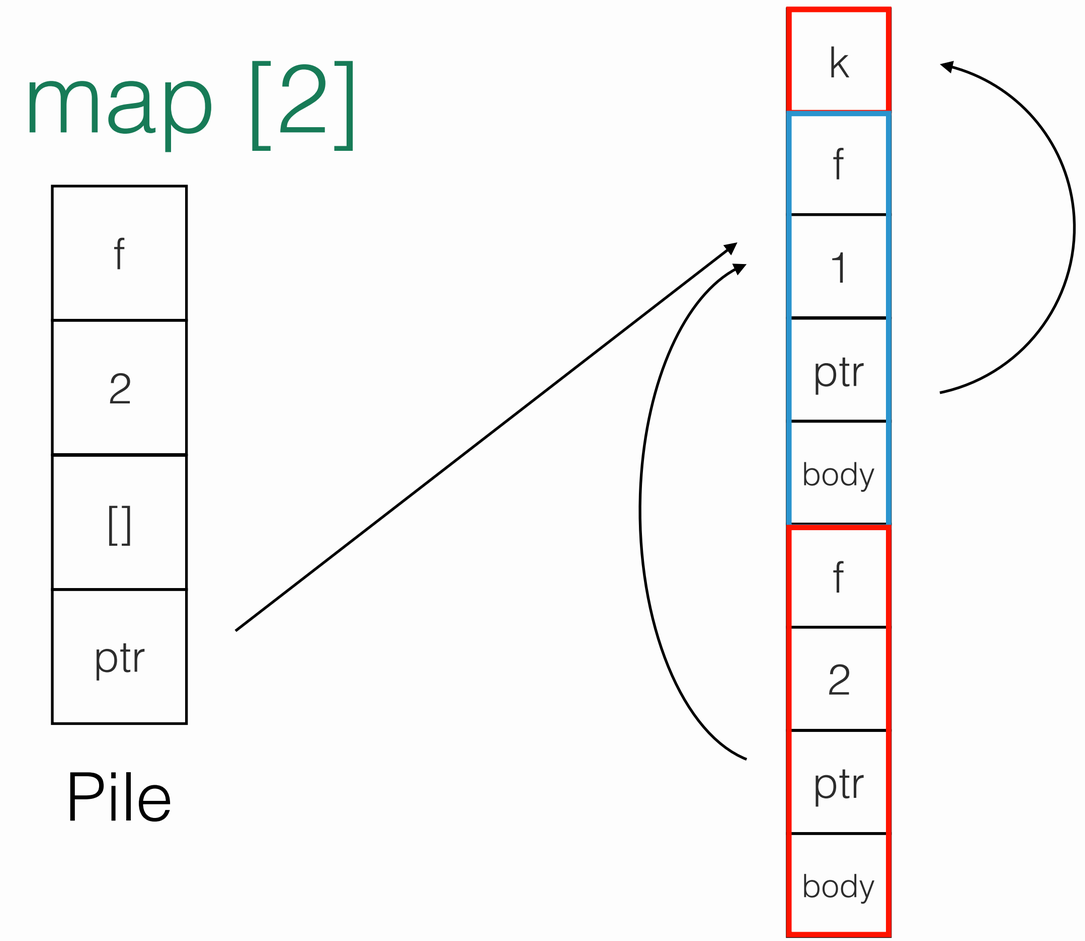
```
map :: (a -> b) -> [a] -> ([a] -> c) -> c
map _ [] k = k []
map f (x:xs) k =
  map f xs (\r -> k (f x : r))
```



Avec continuation, map poursuit avec la fonction k

Avec continuation, utilisation des fermetures

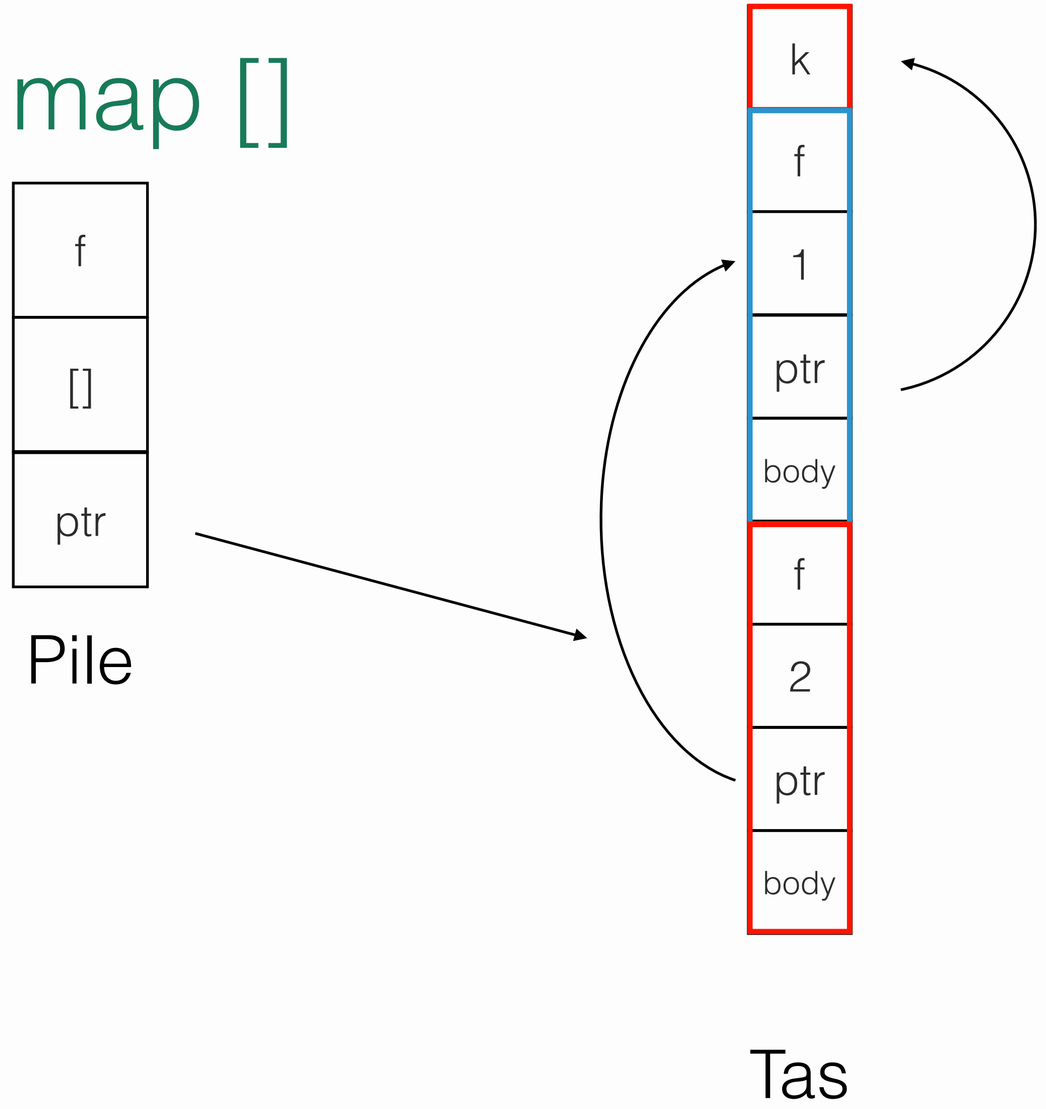
```
map :: (a -> b) -> [a] -> ([a] -> c) -> c
map _ [] k = k []
map f (x:xs) k =
  map f xs (\r -> k (f x : r))
```



Avec continuation, map poursuit avec la fonction k

Avec continuation, utilisation des fermetures

```
map :: (a -> b) -> [a] -> ([a] -> c) -> c
map _ [] k = k []
map f (x:xs) k =
  map f xs (\r -> k (f x : r))
```



Avec continuation, map poursuit avec la fonction k