Recursion versus Iteration

Recursive Factorial

```
factR 0  =  1
factR n  =  n * factR (n-1)
```

Recursion versus Iteration

Recursive Factorial

```
factR 0  =  1
factR n  =  n * factR (n-1)
```

NB!

The context of **factR** increases with every recursive call:

```
factR 4  ==>  4 * factR 3
```

Recursion versus Iteration

Recursive Factorial

```
factR 0  =  1
factR n  =  n * factR (n-1)
```

NB!

The context of **factR** increases with every recursive call:

```
factR 4  ==>  4 * factR 3
         ==>  4 * (3 * factR 2)
```

Recursion versus Iteration

Recursive Factorial

```
factR 0  =  1
factR n  =  n * factR (n-1)
```

NB!

The context of **factR** increases with every recursive call:

```
factR 4  ==>  4 * factR 3
         ==>  4 * (3 * factR 2)
         ==>  4 * (3 * (2 * factR 1))
```

Recursion versus Iteration

Recursive Factorial

```
factR 0  =  1
factR n  =  n * factR (n-1)
```

NB!

The context of **factR** increases with every recursive call:

```
factR 4  ==>  4 * factR 3
         ==>  4 * (3 * factR 2)
         ==>  4 * (3 * (2 * factR 1))
         ==>  4 * (3 * (2 * (1 * factR 0)))
```

Recursion versus Iteration

Recursive Factorial

```
factR 0  =  1
factR n  =  n * factR (n-1)
```

NB!

The context of **factR** increases with every recursive call:

```
factR 4  ==>  4 * factR 3
         ==>  4 * (3 * factR 2)
         ==>  4 * (3 * (2 * factR 1))
         ==>  4 * (3 * (2 * (1 * factR 0)))
         ==>  4 * (3 * (2 * (1 * 1)))
```

Recursion versus Iteration

Recursive Factorial

```
factR 0  =  1
factR n  =  n * factR (n-1)
```

NB!

The context of **factR** increases with every recursive call:

```
factR 4  ==>  4 * factR 3
         ==>  4 * (3 * factR 2)
         ==>  4 * (3 * (2 * factR 1))
         ==>  4 * (3 * (2 * (1 * factR 0)))
         ==>  4 * (3 * (2 * (1 * 1)))
         ==>  24
```

Recursion versus Iteration

Tail-recursive factorial

```
factA  =  factAcc 1
       where  factAcc a 0  =  a
              factAcc a n  =  factAcc (a*n) (n-1)
```

Recursion versus Iteration

Tail-recursive factorial

```
factA  =  factAcc 1
       where  factAcc a 0  =  a
              factAcc a n  =  factAcc (a*n) (n-1)
```

NB!

All calls in the same context:

```
factA 4  ==>  factAcc 1 4
         ==>  factAcc 4 3
         ==>  factAcc 12 2
         ==>  factAcc 24 1
         ==>  factAcc 24 0
         ==>  24
```

Recursion versus Iteration

Iterative control behavior

Function has

- iterative behavior if it uses O(1) memory,
- iterative control behavior if it uses O(1) of memory to store contexts,
- recursive control behavior, if does not have iterative control behavior.

Example

```
reverse  = revApp []
      where  revApp as []       =  as
             revApp as (x:xs)   =  revApp (x:as) xs
```

Recursion versus Iteration

Question

When does a function have iterative control behavior?

Recursion versus Iteration

Question

When does a function have iterative control behavior?

NB!

Hard question!

Recursion versus Iteration

Question

When does a function have iterative control behavior?

NB!

Hard question!

Example

```
factQ n  =  if  strangePredicate n
            then  factR n
            else  factA n
```

Recursion versus Iteration

Tail position/recursion/form

- Subexpression in tail position, evaluating it immediately gives value to the whole expression.

```
if expr0 then expr1 else expr2
let x1 = expr1 in expr0
```

- Function application in tail position is a tail call.

Recursion versus Iteration

Tail position/recursion/form

- Subexpression in tail position, evaluating it immediately gives value to the whole expression.

```
if expr0 then expr1 else expr2
let x1 = expr1 in expr0
```

- Function application in tail position is a tail call.
- Expression is in tail form if all non-tail-position subexpressions are "simple".

Recursion versus Iteration

Tail position/recursion/form

- Subexpression in tail position, evaluating it immediately gives value to the whole expression.

```
if expr0 then expr1 else expr2
let x1 = expr1 in expr0
```

- Function application in tail position is a tail call.
- Expression is in tail form if all non-tail-position subexpressions are "simple".

NB!

Tail form expressions have iterative control behavior!

Continuation

Contexts and Continuations

- `factR 4` is computed as `4 * (3 * (2 * factR 1))`
- We can say that, `factR 1` is computed in context `4 * (3 * (2 * □))`

Continuation

Contexts and Continuations

- `factR 4` is computed as `4 * (3 * (2 * factR 1))`
- We can say that, `factR 1` is computed in context `4 * (3 * (2 * □))`
- Context are an expressions with one hole; they can be expressed as lambda-terms: `\ v -> 4 * (3 * (2 * v))`

Continuation

Contexts and Continuations

- `factR 4` is computed as `4 * (3 * (2 * factR 1))`
- We can say that, `factR 1` is computed in context `4 * (3 * (2 * □))`
- Context are an expressions with one hole; they can be expressed as lambda-terms: `\ v -> 4 * (3 * (2 * v))`
- Such lambda-terms are called continuations.

Continuation

Contexts and Continuations

- `factR 4` is computed as `4 * (3 * (2 * factR 1))`
- We can say that, `factR 1` is computed in context `4 * (3 * (2 * □))`
- Context are an expressions with one hole; they can be expressed as lambda-terms: `\ v -> 4 * (3 * (2 * v))`
- Such lambda-terms are called continuations.
- All computations of `factR` can be represented as `k (factR n)` where `k` is a continuation.

Continuation

Contexts and Continuations

- `factR 4` is computed as `4 * (3 * (2 * factR 1))`
- We can say that, `factR 1` is computed in context `4 * (3 * (2 * □))`
- Context are an expressions with one hole; they can be expressed as lambda-terms: `\ v -> 4 * (3 * (2 * v))`
- Such lambda-terms are called continuations.
- All computations of `factR` can be represented as `k (factR n)` where `k` is a continuation.
- CPS = *Continuation Passing Style*

Continuations

Factorial in CPS

```
factCPS  0  k  =  k 1
factCPS  n  k  =  factCPS (n-1) (\ v -> k (n*v))
```

Continuations

Factorial in CPS

```
factCPS  0  k  =  k 1
factCPS  n  k  =  factCPS (n-1) (\ v -> k (n*v))
```

Claim

For every `k` and `n`,

```
k (factR n) === factCPS n k
```

Continuations

Factorial in CPS

```
factCPS  0  k  =  k 1
factCPS  n  k  =  factCPS (n-1) (\ v -> k (n*v))
```

Claim

For every `k` and `n`,

```
k (factR n) === factCPS n k
```

Proof for (`n = 0`)

```
k (factR 0)  ===  k 1  ===  factCPS 0 k
```

Continuations

Factorial in CPS

```
factCPS  0  k  =  k 1
factCPS  n  k  =  factCPS (n-1) (\ v -> k (n*v))
```

Claim

For every `k` and `n`,

```
k (factR n) === factCPS n k
```

Proof for (`n > 0`)

```
k (factR n)  ===  k (n * factR (n-1))
             ===  (\ v -> k (n*v)) (factR (n-1))
             ===  factCPS (n-1) (\ v -> k (n*v))
             ===  factCPS n k
```

Jätkud

NB!

`factCPS` has iterative control behavior

```
      factCPS 4 k
==>   factCPS 3 (\ v -> k (4*v))
==>   factCPS 2 (\ v -> k (4*(3*v)))
==>   factCPS 1 (\ v -> k (4*(3*(2*v))))
==>   factCPS 0 (\ v -> k (4*(3*(2*(1*v)))))
==>   k (4*(3*(2*(1*1))))
==>   k 24
```

Jätkud

NB!

`factCPS` has iterative control behavior

```
      factCPS 4 k
==>   factCPS 3 (\ v -> k (4*v))
==>   factCPS 2 (\ v -> k (4*(3*v)))
==>   factCPS 1 (\ v -> k (4*(3*(2*v))))
==>   factCPS 0 (\ v -> k (4*(3*(2*(1*v)))))
==>   k (4*(3*(2*(1*1))))
==>   k 24
```

NB!

We get the normal “context independent” factorial if we take **`id`** as the continuation.

```
factC n = factCPS n id
```

Continuations

Recursive `length`

```
length []       =  0
length (x:xs)   =  1 + length xs
```

`length` in CPS

```
lengthC xs = lengthCPS xs id
 where lengthCPS []     k = k 0
       lengthCPS (x:xs) k = lengthCPS xs (\v ->
                                          k (1+v))
```

Continuations

CPS transformation

How to transform expressions into CPS:

- Identification of non-trivial intermediate results
- Serialize their computation.
- Introduce contexts for serial computations.

Continuations

CPS transformation

How to transform expressions into CPS:

- Identification of non-trivial intermediate results
- Serialize their computation.
- Introduce contexts for serial computations.

Example: S combinator

```
f x (g x)
```

Continuations

CPS transformation

How to transform expressions into CPS:

- Identification of non-trivial intermediate results
- Serialize their computation.
- Introduce contexts for serial computations.

Example: S combinator

```
f x (g x)

let   v1  =  f x
      v2  =  g x
      v3  =  v1 v2
in v3
```

Continuations

CPS transformation

How to transform expressions into CPS:

- Identification of non-trivial intermediate results
- Serialize their computation.
- Introduce contexts for serial computations.

Example: S combinator

```
f x (g x)
```

```
let   v1  =  f x          \ k ->  f x     (\ v1  ->
      v2  =  g x                  g x     (\ v2  ->
      v3  =  v1 v2                v1 v2   (\ v3  ->
in v3                             k v3)))
```

Continuations

CPS transformation

How to transform expressions into CPS:

- Identification of non-trivial intermediate results
- Serialize their computation.
- Introduce contexts for serial computations.

Example: Fibonacci

```
fibo 0  =  0
fibo 1  =  1
fibo n  =  fibo (n-1) + fibo (n-2)
```

Continuations

CPS transformation

How to transform expressions into CPS:

- Identification of non-trivial intermediate results
- Serialize their computation.
- Introduce contexts for serial computations.

Example: Fibonacci

```
fibo 0  =  0
fibo 1  =  1
fibo n  =  let  v1  =  fibo (n-1)
               v2  =  fibo (n-2)
           in v1 + v2
```

Continuations

CPS transformation

How to transform expressions into CPS:

- Identification of non-trivial intermediate results
- Serialize their computation.
- Introduce contexts for serial computations.

Example: Fibonacci

```
fibo 0   k   =   k 0
fibo 1   k   =   k 1
fibo n   k   =   fibo   (n-1)   (\   v1   ->
                 fibo   (n-2)   (\   v2   ->
                 k (v1+v2)))
```

Continuations

CPS transformation

How to transform expressions into CPS:

- Identification of non-trivial intermediate results
- Serialize their computation.
- Introduce contexts for serial computations.

Example: `map`

```
map f []     =  []
map f (x:xs) =  f x : map f xs
```

Continuations

CPS transformation

How to transform expressions into CPS:

- Identification of non-trivial intermediate results
- Serialize their computation.
- Introduce contexts for serial computations.

Example: `map`

```
map f []       =  []
map f (x:xs)   =  let  v1  =  f x
                       v2  =  map f xs
                  in v1 : v2
```

Continuations

CPS transformation

How to transform expressions into CPS:

- Identification of non-trivial intermediate results
- Serialize their computation.
- Introduce contexts for serial computations.

Example: `map`

```
map f []       k  =  k []
map f (x:xs)   k  =  f x (\ v1 ->
                        map f xs (\ v2 ->
                        k (v1:v2)))
```

Continuations

CPS

In CPS all non-trivial computations are explicitly revealed and serialized.

- Simplifies semantics and translation
- Instead of using Higher-order function, data structures can be used as continuations. (defunktsionaliseerimine)
- Continuations can be used (in a non-standard way) to model control structures.

CPS example: exceptions

Finding the product (ver. 1)

```
prod []     =  1
prod (x:xs) =  x * prod xs
```

CPS example: exceptions

Finding the product (ver. 1)

```
prod []     =  1
prod (x:xs) =  x * prod xs
```

NB!

If the list contains 0, the result is 0!

```
prod [1,2,3,0,1,2,3,4,5,6,7]  ==>  0
```

NB!

The used definition still multiplies all list elements.

CPS example: exceptions

Finding the product (ver. 2)

```
prod []     =  1
prod (0:xs) =  0
prod (x:xs) =  x * prod xs
```

CPS example: exceptions

Finding the product (ver. 2)

```
prod []     = 1
prod (0:xs) = 0
prod (x:xs) = x * prod xs
```

NB!

Traverses the list until the first zero.

NB!

Multiplies as many elements as it traverses.

CPS example: exceptions

Continuation version

```
prod xs = prodC xs id  where
    prodC []       k  =  k 1
    prodC (0:xs)   k  =  0
    prodC (x:xs)   k  =  prodC xs (\ v -> k (x*v))
```

CPS example: exceptions

Continuation version

```
prod xs = prodC xs id  where
    prodC []      k  =  k 1
    prodC (0:xs)  k  =  0
    prodC (x:xs)  k  =  prodC xs (\ v -> k (x*v))
```

NB!

Traverses the list until the first zero.

NB!

No multiplication if it finds a zero!

CPS Example: multiple results

Compute list length and sum of the elements

```
sumLength = sumLen 0 0  where
    sumLen s l []       =  (s,l)
    sumLen s l (x:xs)   =  sumLen (s+x) (l+1) xs
```

CPS Example: multiple results

Compute list length and sum of the elements

```
sumLength = sumLen 0 0  where
    sumLen s l []       =  (s,l)
    sumLen s l (x:xs)   =  sumLen (s+x) (l+1) xs
```

NB!

Pairs are constructed to be immediately destructed!

```
average xs =  let (s,l) = sumLength xs
              in s `div` l
```

CPS Example: multiple results

Continuation version

```
sumLengthC xs k = sumLen 0 0 xs    where
    sumLen s l []       =  k s l
    sumLen s l (x:xs)   =  sumLen (s+x) (l+1) xs

averageC xs = sumLengthC xs (\ s l -> s `div` l)
```

CPS Example: multiple results

Continuation version

```
sumLengthC xs k = sumLen 0 0 xs    where
    sumLen s l []       =  k s l
    sumLen s l (x:xs)   =  sumLen (s+x) (l+1) xs

averageC xs = sumLengthC xs (\ s l -> s `div` l)
```

NB!

Results are passed on directly.

Continuations using data structures

Factorial in CPS

```
factC n = factCPS n id  where
  factCPS 0 k = k 1
  factCPS n k = factCPS (n-1)
                        (\ v -> k (n*v))
```

Continuations using data structures

Representation Independent CPS

```
factC n = factCPS n mkFinalCont where
   factCPS  0  k  =  applyCont k 1
   factCPS  n  k  =  factCPS  (n-1)
                             (mkNewCont k n)
```

Continuations using data structures

Representation Independent CPS

```
factC n = factCPS n mkFinalCont where
   factCPS  0  k  =  applyCont k 1
   factCPS  n  k  =  factCPS  (n-1)
                             (mkNewCont k n)
```

Using functions

```
mkFinalCont     =  id
mkNewCont k n   =  \ v -> applyCont k (n*v)
applyCont k v   =  k v
```

Continuations using data structures

Representation using data structures

```
data FactCont  =  FinalFactCont
               |  NewFactCont Int FactCont

mkFinalCont    =  FinalFactCont
mkNewCont k n  =  NewFactCont n k
applyCont k v  =  case k of
                    FinalFactCont    ->  v
                    NewFactCont n k  ->  applyCont k (n*v)
```

Continuations using data structures

Representation using data structures

```
data FactCont  =  FinalFactCont
               |  NewFactCont Int FactCont

mkFinalCont    =  FinalFactCont
mkNewCont k n  =  NewFactCont n k
applyCont k v  =  case k of
            FinalFactCont    ->  v
            NewFactCont n k  ->  applyCont k (n*v)
```

NB!

Type `FactCont` is isomorphic to lists of ints!

Continuations using data structures

Representation using data structures

```
mkFinalCont     =  []
mkNewCont k n   =  n : k
applyCont k v   =  foldl (\ v n -> n*v) v k
```

Continuations using data structures

Representation using data structures

```
mkFinalCont     =  []
mkNewCont k n   =  n : k
applyCont k v   =  foldl (\ v n -> n*v) v k
```

NB!

Substitute back into **`factC`** definition ...

CPS factorial — using lists

```
factC n = factCPS n [] where
   factCPS  0  k  =  foldl (\ v n -> n*v) 1 k
   factCPS  n  k  =  factCPS (n-1) (n:k)
```

Continuations for IO

Example

```
getCharCPS :: (Char -> a) -> a
putCharCPS :: Char -> a -> a
```

Continuations for IO

Example

```
getCharCPS :: (Char -> a) -> a
putCharCPS :: Char -> a -> a
```

Converting IO to CPS

```
f :: a -> IO b
```

```
fCPS :: a -> (b -> c) -> c
```