Monad P3 : Continuation Passing Style (1D)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Application operator

($) is a curious **higher-order operator**.

Its type is:

($) :: (a -> b) -> a -> b

It takes a **function** as its first **argument**,

and all it does is to apply the **function** **(a -> b)**

to the second **argument** **a**

for instance, **(head $ "abc") == (head "abc")**.

https://en.wikibooks.org/wiki/Haskell/Higher-order_functions#Function_manipulation

Application operator as a function

Furthermore, as **($)** is just a **function**
which happens to apply **functions**,
and **functions** are just **values**,
we can write intriguing expressions such as:

map ($ 2) [(2*), (4*), (8*)]

($) :: (a -> b) -> a -> b
($ a) :: (a -> b) -> b

https://en.wikibooks.org/wiki/Haskell/Higher-order_functions#Function_manipulation

Application operator ($)

First, **($)** has very low **precedence**,
unlike regular **function application**
which has the highest **precedence**.

can avoid confusing nesting of **parentheses**
by breaking **precedence** with **$**.

https://en.wikibooks.org/wiki/Haskell/Higher-order_functions#Function_manipulation

Application operator ($) example – (1)

We write a non-point-free version of **myInits**

without adding new parentheses:

```
myInits :: [a] -> [[a]]
myInits xs = map reverse . scanl (flip (:)) [] $ xs
```

```
myInits [a1, a2, an]
[[a1], [a2], [an]]
```

https://en.wikibooks.org/wiki/Haskell/Higher-order_functions#Function_manipulation

Application operator ($) example – (2)

```
myInits :: [a] -> [[a]]
myInits xs = map reverse . scanl (flip (:)) [] $ xs


(:) :: a -> [a] -> [a]


scanl :: (a -> b -> a) -> a -> [b] -> [a]


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

https://en.wikibooks.org/wiki/Haskell/Higher-order_functions#Function_manipulation

Application operator ($) example – (3)

```
myInits :: [a] -> [[a]]
myInits xs = map reverse . scanl (flip (:)) [] $ xs


xs :: [a]                         -- [a1, a2, an]     [[a1], [a2], [an]]


(:) :: a -> [a] -> [a]
flip (:) :: [a] -> a -> [a]
scan (flip (:)) :: [a] -> [b] -> [[a]]


flip :: (a -> b -> c) -> b -> a -> c
scanl :: (a -> b -> a) -> a -> [b] -> [a]
```

https://en.wikibooks.org/wiki/Haskell/Higher-order_functions#Function_manipulation

scanl

scanl :: (a -> b -> a) -> a -> [b] -> [a]

it takes the second argument **a**

and the first item of the list **[b]**

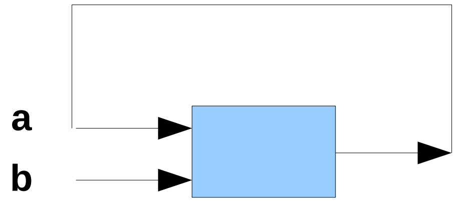
and applies the function to them, **a -> b -> a**

then feeds the function with this result

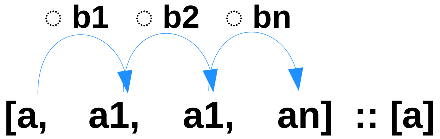
and the second argument and so on.

It returns the list of intermediate and final results.

[a, a1, … an] :: [a]



[b1, … bn] :: [b]



[a, a1, a1, an] :: [a]

http://zvon.org/other/haskell/Outputprelude/scanl_f.html

scanl examples

Input: scanl (/) 64 [4,2,4]

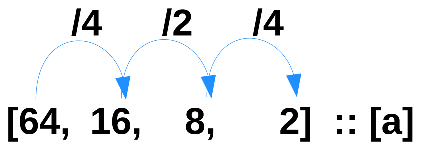
Output: [64.0,16.0,8.0,2.0]

Input: scanl (/) 3 []

Output: [3.0]

Input: scanl max 5 [1,2,3,4]

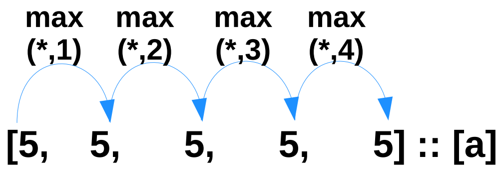
Output: [5,5,5,5,5]



/4 /2 /4

[64, 16, 8, 2] :: [a]

[3] :: [a]



max (*,1) max (*,2) max (*,3) max (*,4)

[5, 5, 5, 5, 5] :: [a]

http://zvon.org/other/haskell/Outputprelude/scanl_f.html

flip

flip takes a **function** of two **arguments** and
returns a version of the same function
with the **arguments** swapped.

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

```
(flip (/)) 3 1
0.3333333333333333
```

```
(flip map) [1,2,3] (*2)                  map (*2) [1,2,3]
[2,4,6]
```

https://en.wikibooks.org/wiki/Haskell/Higher-order_functions#Function_manipulation

Point-free style programming

tacit programming, also called **point-free style**,
is a programming paradigm in which function definitions
do not identify the **arguments** (or "**points**") on which they operate.

Instead the **definitions** merely compose other **functions**,
among which are **combinators** that manipulate the **arguments**.

Tacit programming is of theoretical interest,
because the strict use of **composition** results in programs
that are well adapted for equational reasoning

https://en.wikipedia.org/wiki/Tacit_programming

Combinator

here are two distinct meanings of combinator

The first is a narrow, technical meaning, namely:

A function or definition with **no** **free variables**.

a pure lambda-expression that refers only to its **arguments**, like

\a -> a

\a -> \b -> a

\f -> \a -> \b -> f b a

The study of such things is called combinatory logic.

the examples above are **id**, **const**, and **flip** respectively.

https://wiki.haskell.org/Combinator

Free variable

A **variable** that is not bound.

(\x -> x y)

In the above expression, **y** is a free variable.

Whether a variable is free or not depends largely **on context**.

It often helps to describe a **variable** as being free
within a particular expression.

https://wiki.haskell.org/Free_variable

Point-free style programming (1)

Conventional (specify the arguments explicitly):

```
sum (x:xs) = x + (sum xs)
sum [] = 0
```

Point-free (no explicit arguments)

```
sum = foldr (+) 0
```

it's just a **fold** with **+** starting with **0**

https://en.wikipedia.org/wiki/Tacit_programming

Point-free style programming (2)

Conventional (specify the arguments explicitly):

g(x) = f(x)

Point-free (no explicit arguments)

g = f

It's closely related to **currying**

(or operations like **function composition**).

https://en.wikipedia.org/wiki/Tacit_programming

Point-free style programming (3)

to compute **x*x+1**

Conventional (specify the <u>arguments</u> <u>explicitly</u>):

```
f :: a -> a
f x = inc (square x)
```

Point-free (no <u>explicit</u> <u>arguments</u>)

```
f :: a -> a
f = inc . square
```

```
square :: a -> a
square x = x*x

inc :: a -> a
inc x = x+1
```

https://en.wikipedia.org/wiki/Tacit_programming

Like a value is applied to a function

```
map ($ 2) [ (2*), (4*), (8*) ]


[($ 2) (2*), ($ 2) (4*), ($ 2) (8*)]
[ (2*) $ 2, (4*) $ 2, (8*) $ 2 ]


[4,8,16]


map (*2) [ 2, 4, 8 ]


[ (*2) 2, (*2) 4, (*2) 8 ]
```

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

Reversal of a value and a function

map ($ 2) [(2*), (4*), (8*)]

[4,8,16]

[(2*) $ 2, (4*) $ 2, (4*) $ 2]

map (*2) [2, 4, 8]

[(*2) 2, (*2) 4, (*2) 8]

The **($) section** makes the code appear backwards,

as if we are applying a **value** to the **functions**

rather than the other way around.

such an **reversal** is at heart of

continuation passing style!

($) :: (a -> b) -> a -> b

($ a) :: (a -> b) -> b

map ($ 2) [(2*), (4*), (8*)]

[($ 2) (2*), ($ 2) (4*), ($ 2) (8*)]

[~~(2*) ($ 2)~~, ~~(4*) ($ 2)~~, ~~(8*) ($ 2)~~]

[~~$ 2 (2*)~~, ~~$ 2 (4*)~~, ~~$ 2 (8*)~~]

[(2*) $ 2, (4*) $ 2, (8*) $ 2]

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

Suspended computation and continuation

From a CPS perspective, **($ 2)** is a **suspended computation**:

a **function** with general type

(a -> r) -> r

takes another function as **argument** **(a -> r)**

produces a final result. **r**

the **(a -> r) argument** is the **continuation**;

it specifies how the **computation**

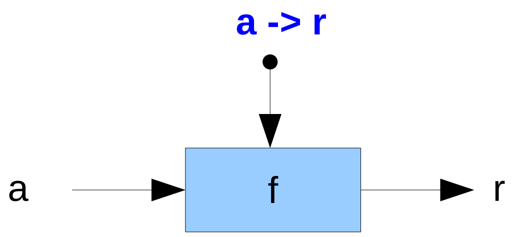
will be brought to a conclusion.

map ($ 2) [(2*), (4*), (8*)]

($) :: (a -> b) -> a -> b

($ a) :: (a -> b) -> b

($ 2) (2*)	**4**
($ 2) (4*)	**8**
($ 2) (8*)	**16**



https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

Continuation functions for conclusions

map ($ 2) [(2*), (4*), (8*)]

Suspended Computation

continuations

the **functions** in the list are **(2*), (4*), (8*)**

supplied as **continuations** via **map**,

to the **suspended computation** **($ 2)**

producing three distinct results.

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

CPS (Continuation Passing Style)

map ($ 2) [(2*), (4*), (8*)]

- **suspended computations** are largely <u>interchangeable</u> with plain values: **($ 2)**
- **flip ($)** converts any **value** into a **suspended computation**
- passing **id** as its **continuation** gives back the original value. **($ 2) id**

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

Continuation Passing Style

Example I : Factorial Computation

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

Recursive Calling

```
fact x =
  if x <= 1 then 1 else x * fact (x - 1)

 fact 4
 4 * fact 3
 4 * (3 * fact 2)
 4 * (3 * (2 * fact 1))
 4 * (3 * (2 * 1))
 4 * (3 * 2)
 4 * 6
 24
```

Each call of fact is made
with the **promise**
that the **value** returned will be multiplied
by the **value** of the parameter
at the time of the call.

Thus **fact** is invoked
with larger and larger
control contexts
as the calculation proceeds.

https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

Continuation passing style

```
fact_cps x k =
  if x <= 1 then k 1 else fact_cps (x - 1) (\v -> k (x * v))
```

```
• fact_cps 4 id
• fact_cps 3 (\v -> id (4 * v))                                          -- v
• fact_cps 2 (\v' -> (\v -> id (4 * v)) (3 * v'))                         -- v’
• fact_cps 1 (\v'' -> (\v' -> (\v -> id (4 * v)) (3 * v')) (2 * v''))     -- v’’
• (\v'' -> (\v' -> (\v -> id (4 * v)) (3 * v')) (2 * v'')) 1              -- v’’
  (\v' -> (\v -> id (4 * v)) (3 * v')) (2 * 1)                             -- v’
  (\v -> id (4 * v)) (3 * (2 * 1))                                         -- v
  id (4 * (3 * (2 * 1)))
  (4 * (3 * (2 * 1)))
  24
```

Continuations	**k**
Suspended computation	**x * v**

each step remembers

what to do with the result

At the **bottom** of the **recursion**,

these **continuations** are **evaluated**.

using 'id' as the first continuation.

https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

continuations are supplied
to the **suspended computation**

Continuation passing

```
fact_cps x k =              x          k
  if x <= 1  then k 1
             else fact_cps (x - 1) (\v -> k (x * v))
```

v :: a

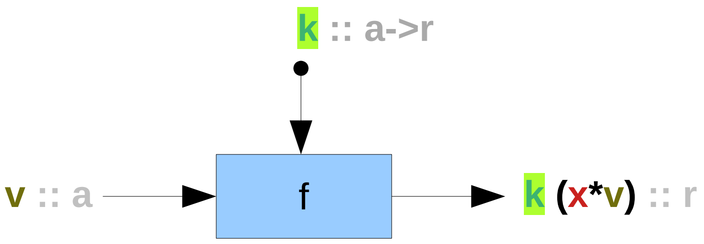
x :: a

k :: a -> r

k x*v :: r

(a -> r) -> r

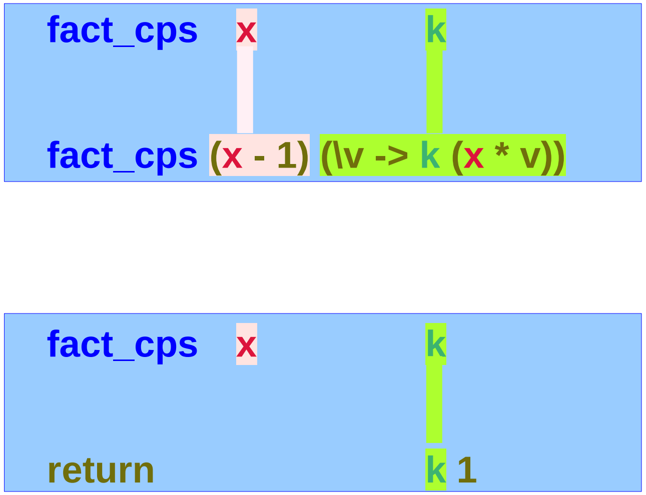
f :: a -> (a -> r) -> r



Continuations	k
Suspended computation	x * v

https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

Passing and evaluating continuations



```
fact_cps x                k

fact_cps (x - 1) (\v -> k (x * v))
```

each step remembers

what to do with the result

```
fact_cps x                k

return                    k 1
```

At the **bottom** of the **recursion**,

these **continuations** are **evaluated**.

Continuations	**k**
Suspended computation	**x * v**

https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

Steps of passing and evaluating continuations

```
fact_cps x k =
  if x <= 1 then k 1 else fact_cps (x - 1) (\v -> k (x * v))
```

```
• fact_cps 4 id
• fact_cps 3 (\v -> id (4 * v))                                         -- v
• fact_cps 2 (\v' -> (\v -> id (4 * v)) (3 * v'))                       -- v’
• fact_cps 1 (\v'' -> (\v' -> (\v -> id (4 * v)) (3 * v')) (2 * v''))   -- v’’
• (\v'' -> (\v' -> (\v -> id (4 * v)) (3 * v')) (2 * v'')) 1            -- v’’
   (\v' -> (\v -> id (4 * v)) (3 * v')) (2 * 1)                         -- v’
   (\v -> id (4 * v)) (3 * (2 * 1))                                     -- v
   id (4 * (3 * (2 * 1)))
   (4 * (3 * (2 * 1)))
   24
```

Continuations	**k**
Suspended computation	**x * v**

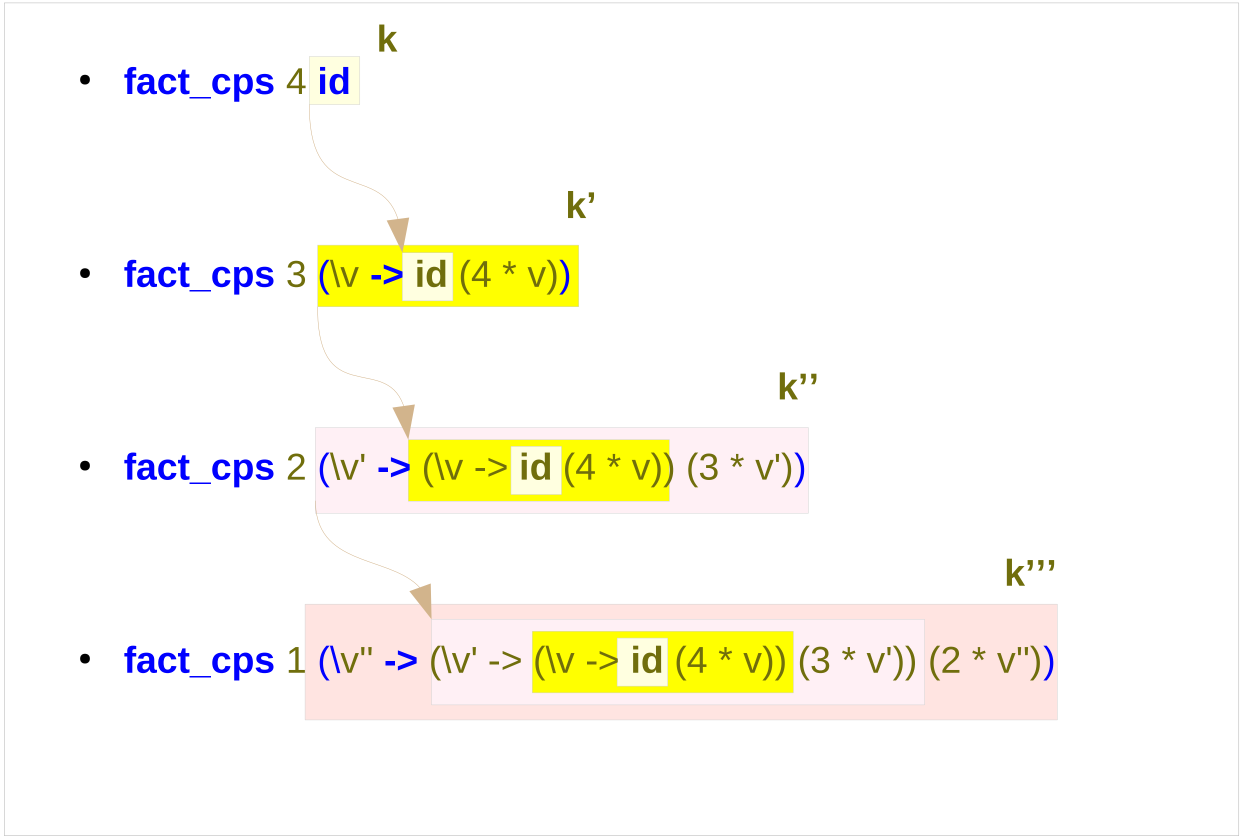
each step remembers
what to do with the result

At the **bottom** of the **recursion**,
these **continuations** are **evaluated**.

using 'id' as the first continuation.

https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

Passing continuations



- **fact_cps** 4 **id**
- **fact_cps** 3 (\v **->** **id** (4 * v))
- **fact_cps** 2 (\v' **->** (\v -> **id** (4 * v)) (3 * v'))
- **fact_cps** 1 (\v'' **->** (\v' -> (\v -> **id** (4 * v)) (3 * v')) (2 * v''))

Continuations	**k**
Suspended computation	**x * v**

Let’s name the continuations at each step as k’, k’’, k’’’

the **control context** is made explicit

in the **continuation argument** to

fact_cps

https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

Suspended computation

```
fact_cps x k =
  if x <= 1 then k 1 else fact_cps (x - 1) (\v -> k (x * v))            Suspended computation   x * v

fact_cps 4 k      fact_cps 4 id                      k = id

fact_cps 3 k’     fact_cps 3 (\v -> k (4*v))         k’ =   (\v -> k (4*v))       = (\v -> id (4*v))
                                                                    v = (3*(2*1))
fact_cps 2 k’’    fact_cps 2 (\v’ -> k’ (3*v’))      k’’ =  (\v’ -> k’ (3*v’))    = (\v' -> (\v -> id (4*v)) (3*v')
                                                                    v’ = (2*1)
fact_cps 1 k’’’   fact_cps 1 (\v’’ -> k’’ (2*v’’))   k’’’ = (\v’’ -> k’’ (2*v’’)) = (\v'' -> (\v' -> (\v -> id (4*v)) (3*v')) (2*v''))
                                                                    v’’ = 1

                  return      k’’’ 1
```

https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

Suspended computation

```
fact_cps x k =
  if x <= 1 then k 1 else fact_cps (x - 1) (\v -> k (x * v))
```

- fact_cps 4 id k = id
- fact_cps 3 (\v -> k (4*v)) k’ = (\v -> k (4*v))
- fact_cps 2 (\v’ -> k’ (3*v’)) k’’ = (\v’ -> k’ (3*v’))
- fact_cps 1 (\v’’ -> k’’ (2*v’’)) k’’’ = (\v’’ -> k’’ (2*v’’))

- return k’’’ 1 = (\v’’ -> k’’ (2*v’’)) 1 = k’’ (2*1) v’’ = 1

  k’’ (2*1) = (\v’ -> k’ (3*v’)) (2*1) = k’ (3*(2*1)) v’ = (2*1)

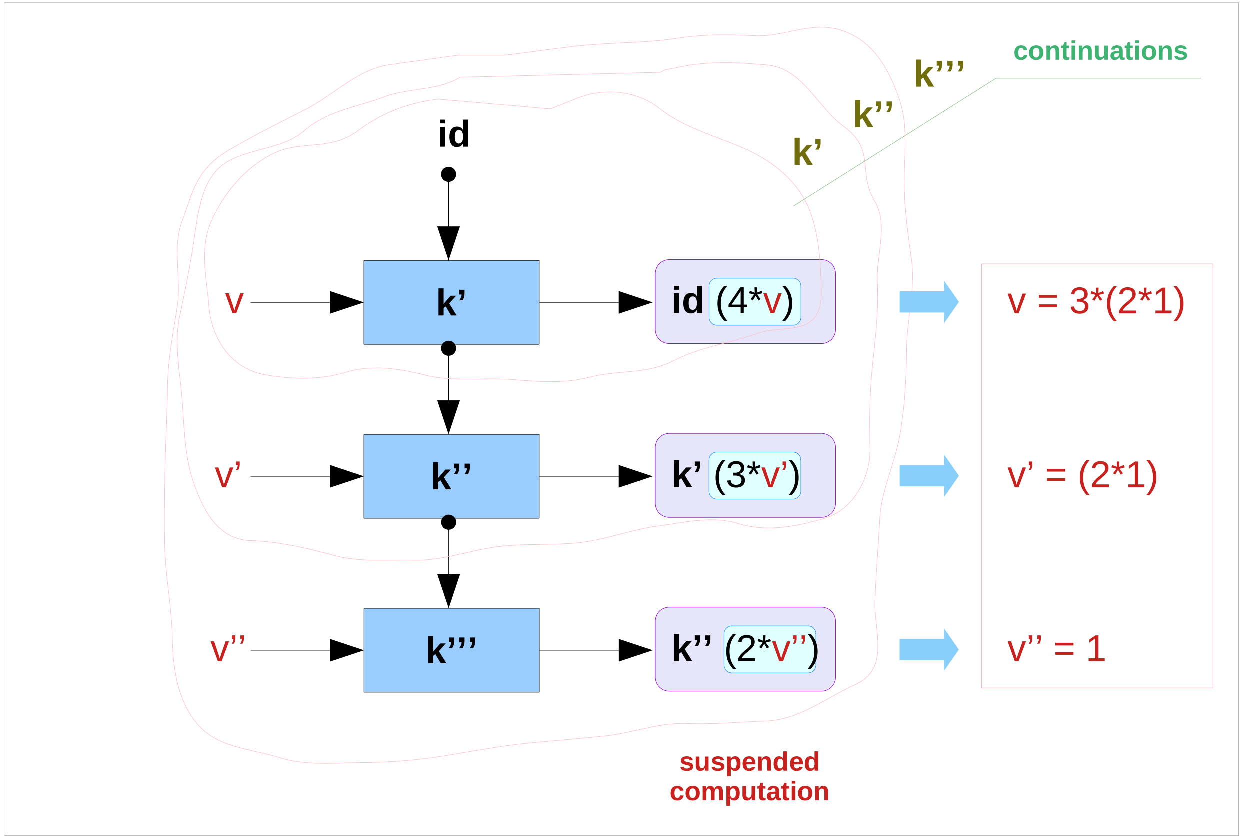
  k’ (3*(2*1)) = (\v -> k (4*v)) (3*(2*1)) = k (4*(3*(2*1))) v = (3*(2*1))

fact_cps 4 k
fact_cps 3 k’
fact_cps 2 k’’
fact_cps 1 k’’’
k’’’ 1
➡ k’’ (2*1)
k’’ (2*1)
➡ k’ (3*(2*1))
k’ (3*(2*1))
➡ k (4*(3*(2*1)))
➡ id (4*(3*(2*1)))

Suspended computation x * v

https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

Continuations and suspended computations

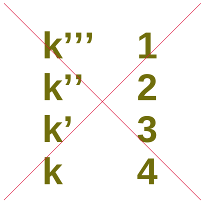


Let's name the **lambda expression** at each step as **k', k'', k'''**

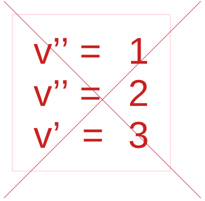
k' = \v -> **id** (4*v)

k'' =\v' -> (\v -> **id** (4*v)) (3*v')

k''' =\v'' -> (\v' -> (\v -> **id** (4*v)) (3*v')) (2*v'')

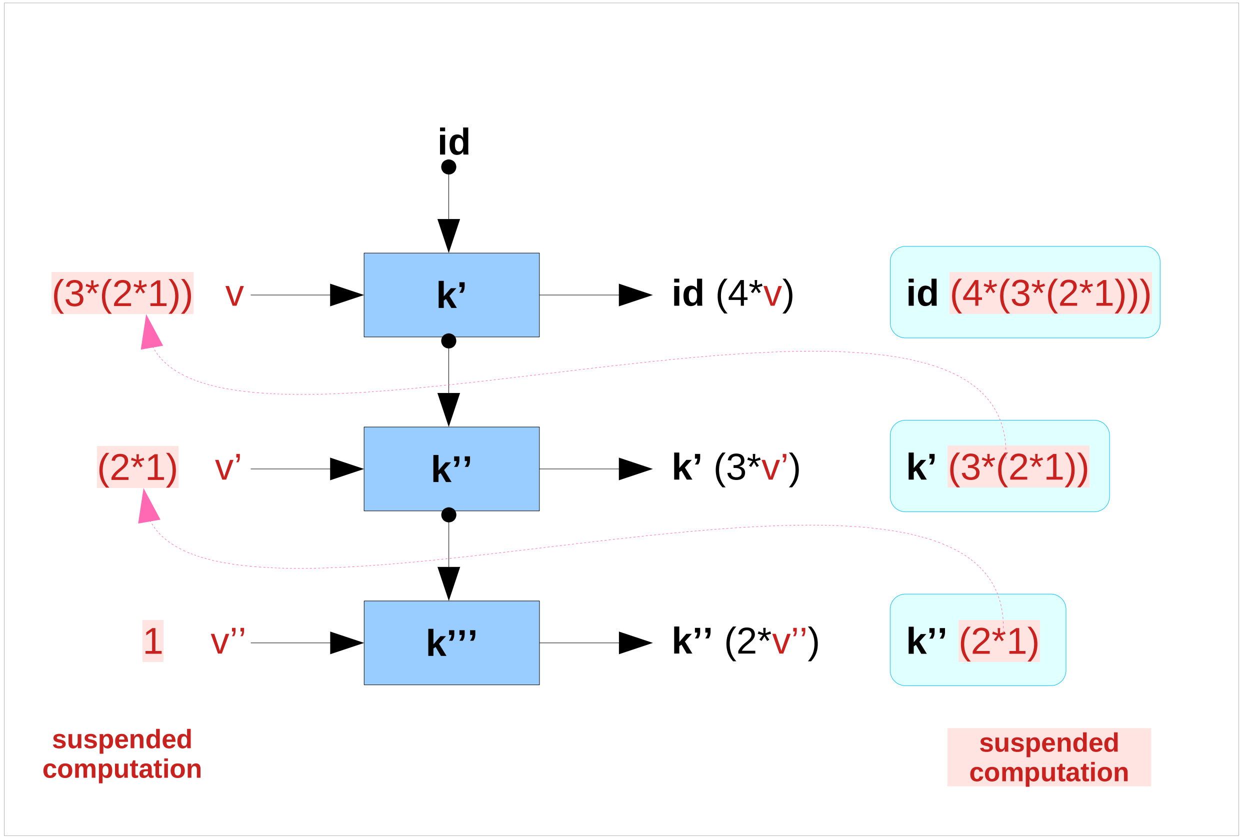


suspended computation



https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

Evaluating continuations



id

(3*(2*1)) v → k' → id (4*v)

(2*1) v' → k'' → k' (3*v')

1 v'' → k''' → k'' (2*v'')

id (4*(3*(2*1)))

k' (3*(2*1))

k'' (2*1)

suspended computation

suspended computation

id (4*(3*(2*1)))

(\v -> id (4*v)) (3*(2*1))

(\v' -> (\v -> id (4*v)) (3*v')) (2*1)

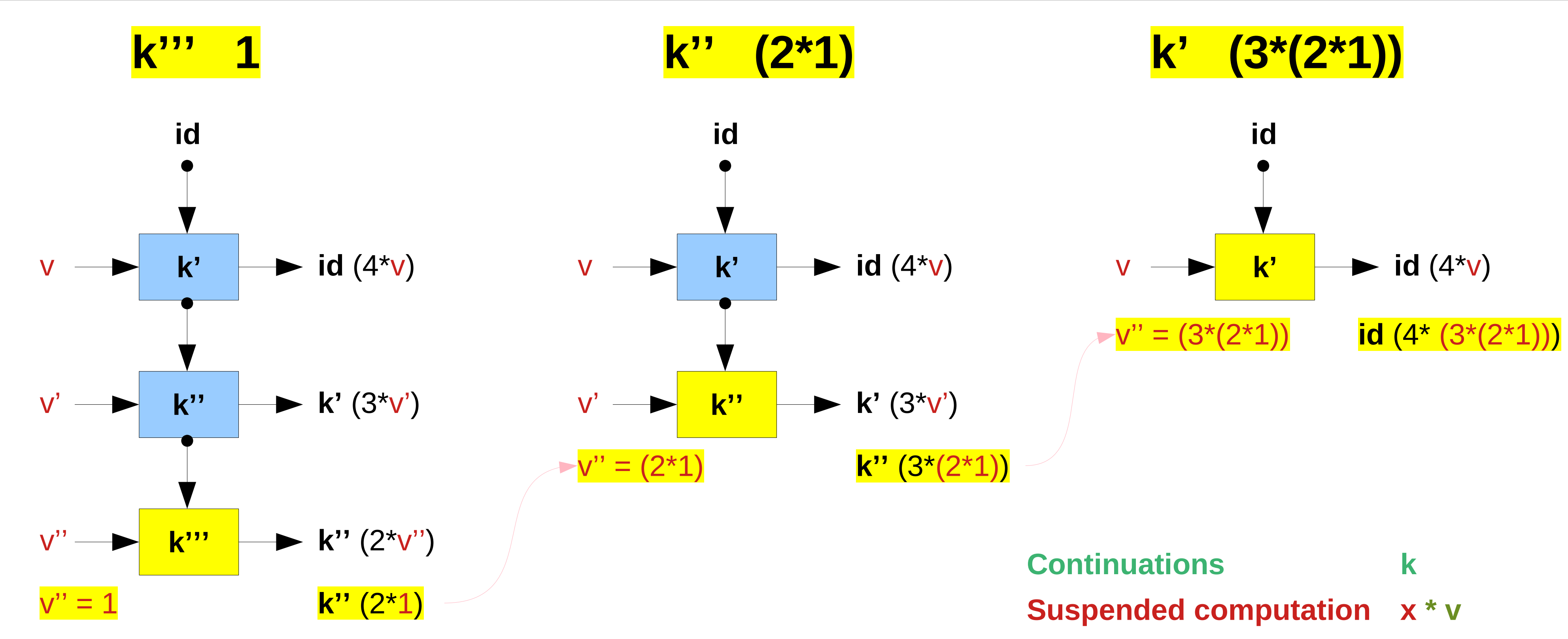
(\v'' -> (\v' -> (\v -> id (4*v)) (3*v')) (2*v'')) 1

~~v'' = 1~~
~~v' = 2~~
~~v' = 3~~

v'' = 1
v' = (2*1)
v = (3*(2*1))

https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

Evaluating continuations



k’’’ 1

k’’ (2*1)

k’ (3*(2*1))

id

v → **k’** → **id** (4*v)

v’ → **k’’** → **k’** (3*v’)

v’’ → **k’’’** → **k’’** (2*v’’)

v’’ = 1

k’’ (2*1)

id

v → **k’** → **id** (4*v)

v’ → **k’’** → **k’** (3*v’)

v’’ = (2*1)

k’’ (3*(2*1))

id

v → **k’** → **id** (4*v)

v’’ = (3*(2*1))

id (4* (3*(2*1)))

Continuations **k**

Suspended computation **x * v**

https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

CPS Ex 1

```
fact_cps x k =
  if x <= 1 then k 1 else fact_cps (x - 1) (\v -> k (x * v))
```

suspended computation — continuations

the **control context** is made explicit

in the **continuation** argument to **fact_cps**. **(\v -> k (x * v)**

never calling to **fact_cps** that is the argument

to some other **computation** like **x * fact (x - 1)**

Instead, each step remembers what to do with the result

as a **first-class function**. **(\v -> k (x * v))**

At the **bottom** of the **recursion**,

these **continuations** are **evaluated**.

https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

CPS Ex 1

- **fact_cps** 4 **id**

- **fact_cps** 3 (\v **->** **id** (4 * v))

  for an input v, compute **id** (4 * v)

- **fact_cps** 2 (\v' **->** (\v -> **id** (4 * v)) (3 * v'))

  for an input v', compute (\v -> **id** (4 * v)) (3 * v')

  = **id** (4 * (3 * v'))

- **fact_cps** 1 (\v'' **->** (\v' -> (\v -> **id** (4 * v)) (3 * v')) (2 * v''))

  for an input v'', compute (\v' -> (\v -> **id** (4 * v)) (3 * v')) (2 * v'')

  = (\v -> **id** (4 * v)) (3 * (2 * v''))

  = **id** (4 * (3 * (2 * v'')))

each step remembers
what to do with the result
as a **first-class function**.

each step associates
with an anonymous function
(lambda expression)
the result

https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

First class functions

functions as **arguments** to other functions,
returning them as the **values** from other functions,
and **assigning** them to **variables** or storing them in data structures.

in the context of "functions as first-class citizens"

1 Higher-order functions: passing functions as arguments
2 Anonymous and nested functions
3 Non-local variables and closures
4 Higher-order functions: returning functions as results
5 Assigning functions to variables
6 Equality of functions

https://en.wikipedia.org/wiki/First-class_function

Higher-order functions

In mathematics and computer science, a **higher-order function** is a function that does at least one of the following:

- takes one or more functions as arguments (i.e. procedural parameters),
- returns a function as its result.

All other functions are **first-order functions**.

In mathematics higher-order functions are also termed **operators** or **functionals**.

https://en.wikipedia.org/wiki/Higher-order_function

CPS (Continuation Passing Style)

An elementary way to take advantage of **continuations**
is to modify our functions so that
they return **suspended computations**
rather than ordinary values.   -- without evaluation

create ***suspended computations*** *and*
pass ***continuations***
rather than return ordinary values

Suspending a computation :
each step remembers
what to do with the result

Get back to the suspended computation :
At the ***bottom*** *of the* ***recursion****,*
these ***continuations*** *are* ***evaluated****.*

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

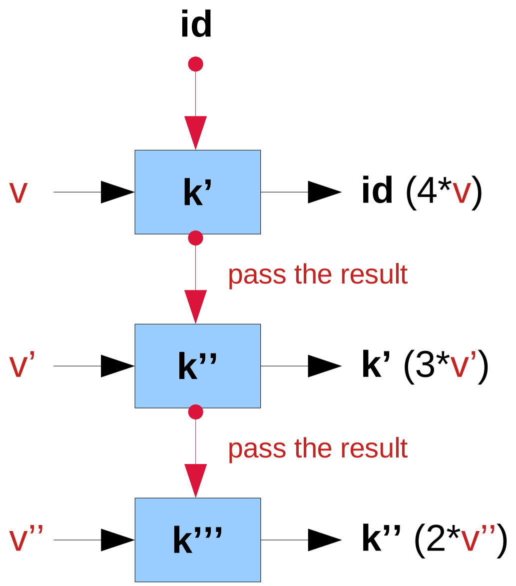
Passing the result

a **function** written in **continuation passing style**

No function call is allowed to **return** to its **caller**, ever.

Instead, it must always pass its **result** directly

to an explicit **continuation**.



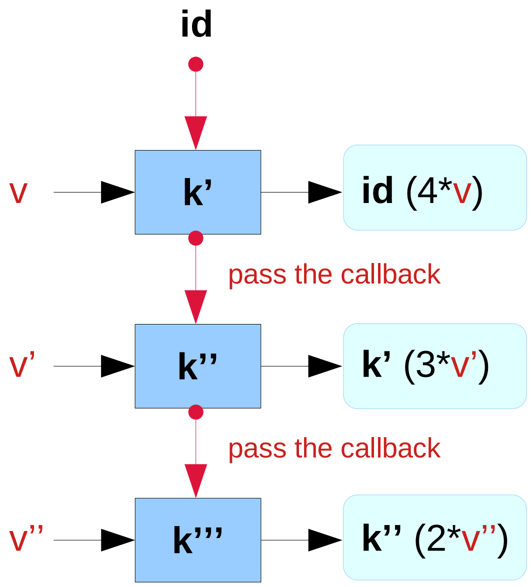
fact_cps 4 **k**	**k** = **id**
fact_cps 3 **k’**	**k’** = (\v **->** **k** (4*v))
fact_cps 2 **k’’**	**k’’** = (\v’ **->** **k’** (3*v’))
fact_cps 1 **k’’’**	**k’’’** = (\v’’ **->** **k’’** (2*v’’))

https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

A callback and its return value

every **function** takes an **extra argument** (a **callback**)
and passes its “**return value**” this **callback**.

every **function** takes an **extra argument** (a **callback**)
and its “**return value**” is the application of this **callback**.



https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

Invoking the current continuation callback

When a function is ready to "return", -- evaluate

it invokes the "**current continuation**" **callback**

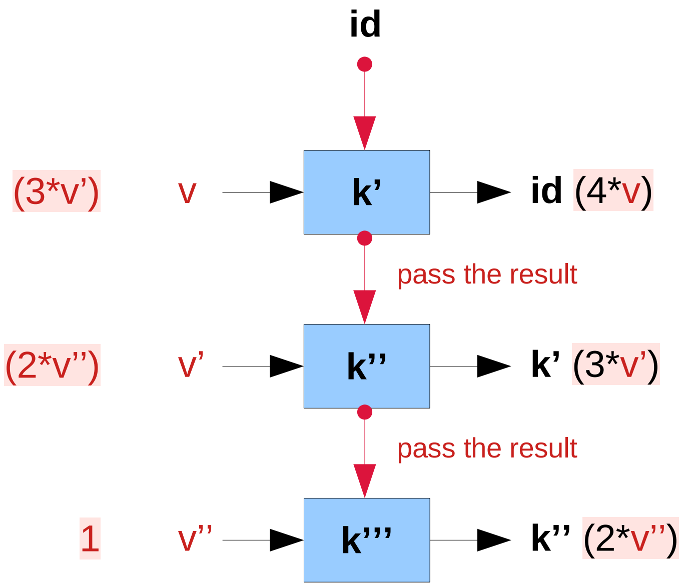
(provided by its caller) on the “return value”

- **return**

k’’’ 1
➡ **k’’** (2*1)

k’’ (2*1)
➡ **k’** (3*(2*1))

k’ (3*(2*1))
➡ **k** (4*(3*(2*1)))
➡ **id** (4*(3*(2*1)))

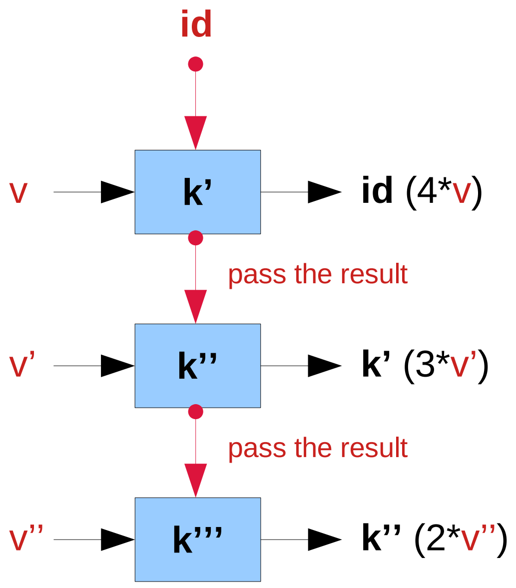


https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

Callers provide the continuation

When calling functions written in **CPS-style**,
callers must also provide the "**continuation**", i.e.
a **function** that says what to do
with the result of the **function call**.

fact_cps 4 **id**



https://www.seas.upenn.edu/~cis552/13fa/lectures/FunCont.html

Control flow

Continuations and **suspended computations**

make it possible

- to explicitly manipulate the **control flow** of a program
- to dramatically alter the **control flow** of a program.

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

Returning early from a procedure

- **returning** early from a procedure
  can be implemented with **continuations**.

- **exceptions** and **failure** can also
  be handled with **continuations**
  - pass in a **continuation** for success,
  - another **continuation** for fail,
  - invoke the appropriate **continuation**.

returning early from a procedure
without evaluations

delayed evaluations

suspended computations

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

Suspending a computation

- suspending a computation

  and returning to it at another time,

  and implementing simple forms of **concurrency**

  notably, one Haskell implementation, Hugs,

  uses **continuations** to implement

  cooperative concurrency

Suspending a computation :

each step remembers

what to do with the result

Get back to the suspended computation :

*At the **bottom** of the **recursion**,*

*these **continuations** are **evaluated**.*

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

Improving performance

In some circumstances, **CPS** can be used

to improve performance by eliminating certain

construction-pattern matching sequences

i.e. a **function** returns a **complex structure**

which the caller will at some point deconstruct

though a sufficiently **smart compiler**

should be able to do the elimination

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

Continuation Passing Style

Example II : Pythagoras Equation Computation

Ver 1

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

CPS (Continuation Passing Style)

```
pow2 :: Float -> Float
pow2 a = a ** 2

add :: Float -> Float -> Float
add a b = a + b

pyth :: Float -> Float -> Float
pyth a b = sqrt (add (pow2 a) (pow2 b))
```

https://en.wikipedia.org/wiki/Continuation-passing_style

CPS (Continuation Passing Style)

To transform the traditional function to CPS,

we need to change its signature.

The **function** will get another **argument** of **function type**,

continuations of the type **(Float -> a)**

and its **return type** depends on that **function**:

cont :: Float -> a

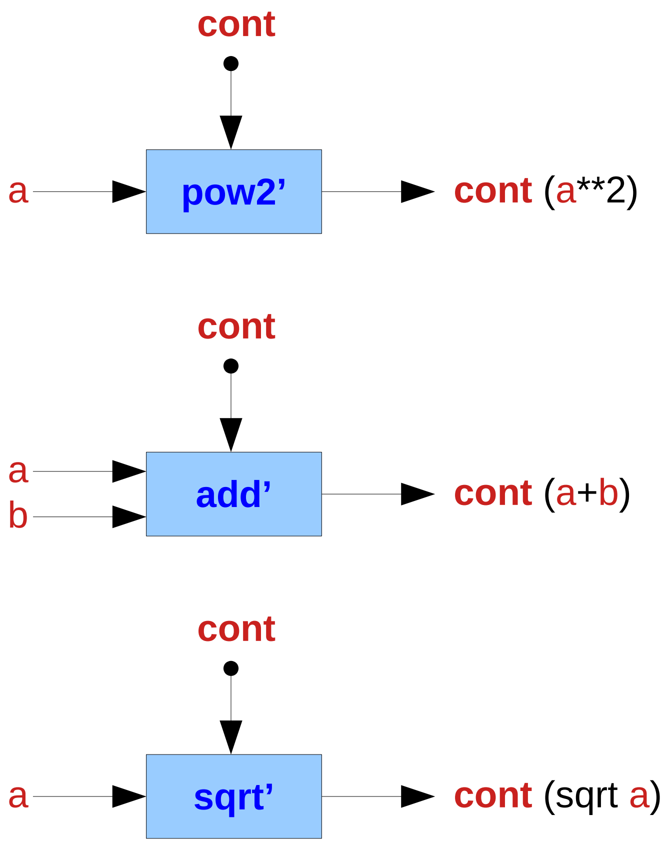
https://en.wikipedia.org/wiki/Continuation-passing_style

CPS (Continuation Passing Style)

```
pow2' :: Float -> (Float -> a) -> a
pow2' a cont = cont (a ** 2)


add' :: Float -> Float -> (Float -> a) -> a
add' a b cont = cont (a + b)


sqrt' :: Float -> ((Float -> a) -> a)
sqrt' a = \cont -> cont (sqrt a)
```



https://en.wikipedia.org/wiki/Continuation-passing_style

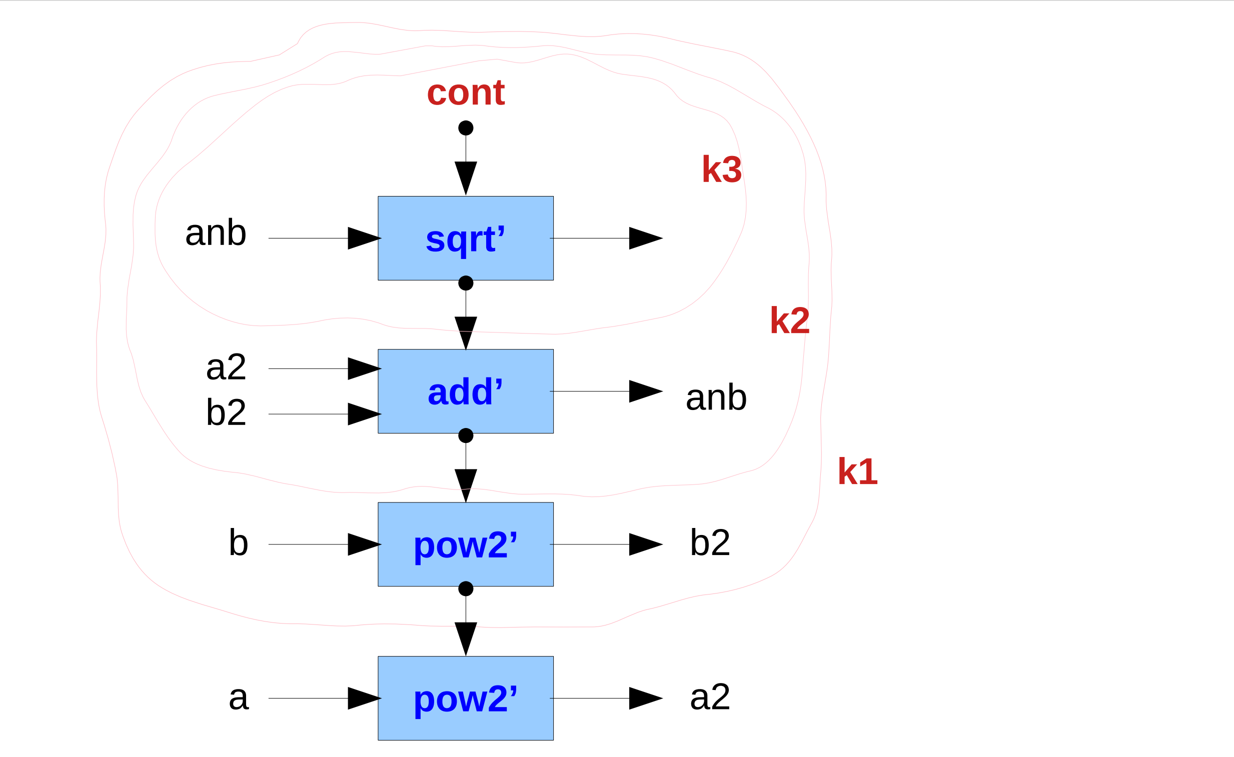
CPS (Continuation Passing Style)

```
-- Types a -> (b -> c) and a -> b -> c are equivalent,
-- so CPS function may be viewed as a higher order function

pyth' :: Float -> Float -> (Float -> a) -> a
pyth' a b cont =
        pow2' a  k1
          (\a2 -> pow2' b   k2
                    (\b2 -> add' a2 b2 k3
                              (\anb -> sqrt' anb cont)))
```

https://en.wikipedia.org/wiki/Continuation-passing_style

CPS (Continuation Passing Style)



```
pyth' a b cont =
pow2' a k1
          pow2' b k2
                    add' a2 b2 k3
                                 sqrt' anb cont
```

k1 = (\a2 -> pow2' b (\b2 -> add' a2 b2 (\anb -> sqrt' anb cont)))

k2 = (\b2 -> add' a2 b2 (\anb -> sqrt' anb cont))

k3 = (\anb -> sqrt' anb cont)

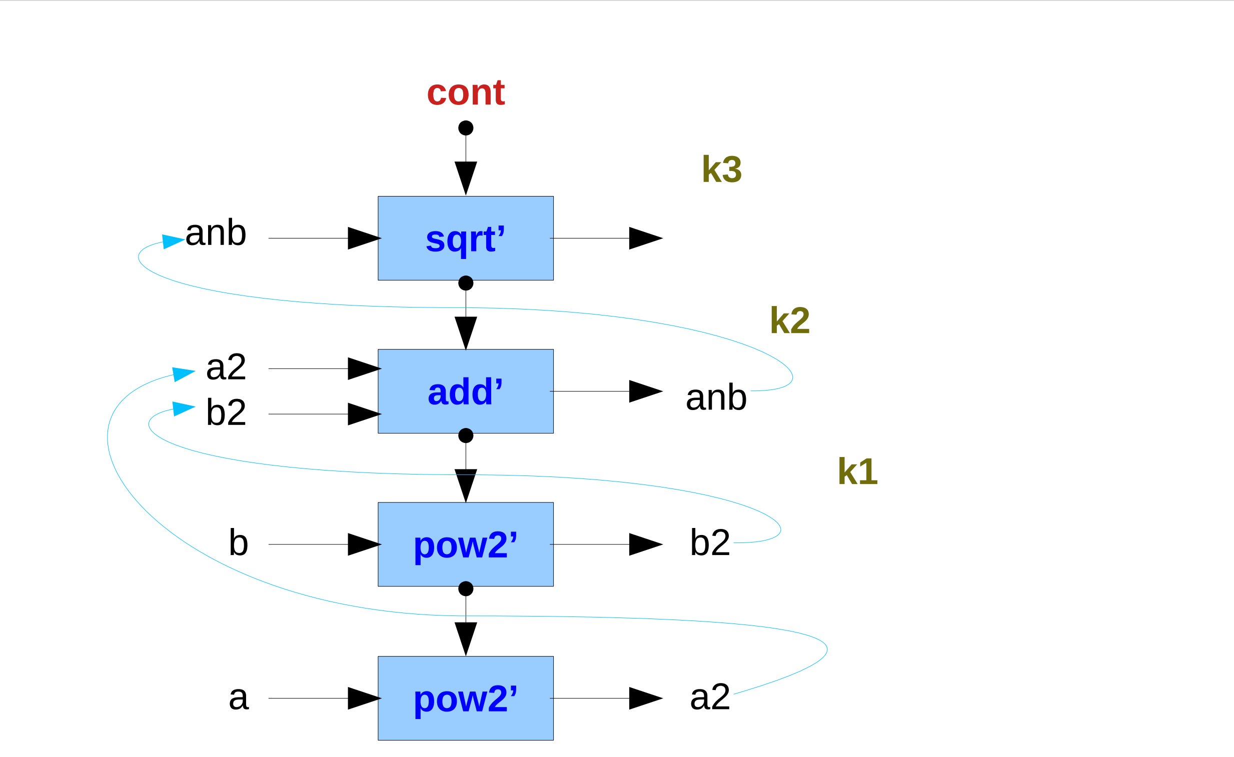
k1 = (\a2 -> pow2' b k2)

k2 = (\b2 -> add' a2 b2 k3)

k3 = (\anb -> sqrt' anb cont)

https://en.wikipedia.org/wiki/Continuation-passing_style

CPS (Continuation Passing Style)



pyth' a b cont =
pow2' a k1
    pow2' b k2
        add' a2 b2 k3
            sqrt' anb cont

k1 = (\a2 -> pow2' b (\b2 -> add' a2 b2 (\anb -> sqrt' anb cont)))

k2 = (\b2 -> add' a2 b2 (\anb -> sqrt' anb cont))

k3 = (\anb -> sqrt' anb cont)

k1 = (\a2 -> pow2' b k2)

k2 = (\b2 -> add' a2 b2 k3)

k3 = (\anb -> sqrt' anb cont)

https://en.wikipedia.org/wiki/Continuation-passing_style

CPS (Continuation Passing Style)

First we calculate the square of a in **pyth'** function

and pass a **lambda function** as a **continuation**

which will accept a **square** of **a** as a first argument.

And so on until we reach the result of our calculations.

To get the result of this function we can pass **id** function

as a final argument which returns the value

that was passed to it unchanged: **pyth' 3 4 id == 5.0**.

https://en.wikipedia.org/wiki/Continuation-passing_style

Continuation Passing Style

Example II : Pythagoras Equation Computation

Ver 2

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

A simple module – without continuation

```
-- We assume some primitives add and square

add :: Int -> Int -> Int
add x y = x + y

square :: Int -> Int
square x = x * x

pythagoras :: Int -> Int -> Int
pythagoras x y = add (square x) (square y)
```

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

A simple module – with continuation

```
-- We assume CPS versions of the add and square primitives,
-- (note: the actual definitions of add_cps and square_cps are not
-- in CPS form, they just have the correct type)

add_cps :: Int -> Int -> ((Int -> r) -> r)
add_cps x y = \k -> k (add x y)

square_cps :: Int -> ((Int -> r) -> r)
square_cps x = \k -> k (square x)
```

Continuations

add_cps x y k = k (add x y)

k :: Int -> r

(add x y) :: Int

k (add x y) :: r

square_cps x k = k (square x)

k :: Int -> r

(square x) :: Int

k (square x) :: r

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

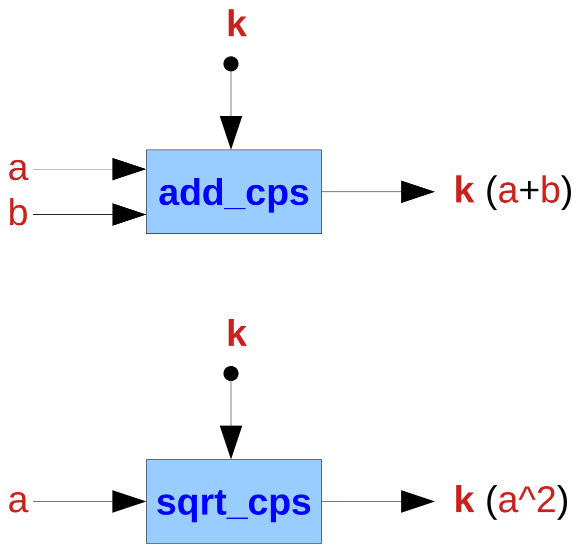
A simple module – with continuation

```
-- We assume CPS versions of the add and square primitives,
-- (note: the actual definitions of add_cps and square_cps are not
-- in CPS form, they just have the correct type)

add_cps :: Int -> Int -> ((Int -> r) -> r)
add_cps x y = \k -> k (add x y)



square_cps :: Int -> ((Int -> r) -> r)
square_cps x = \k -> k (square x)
```



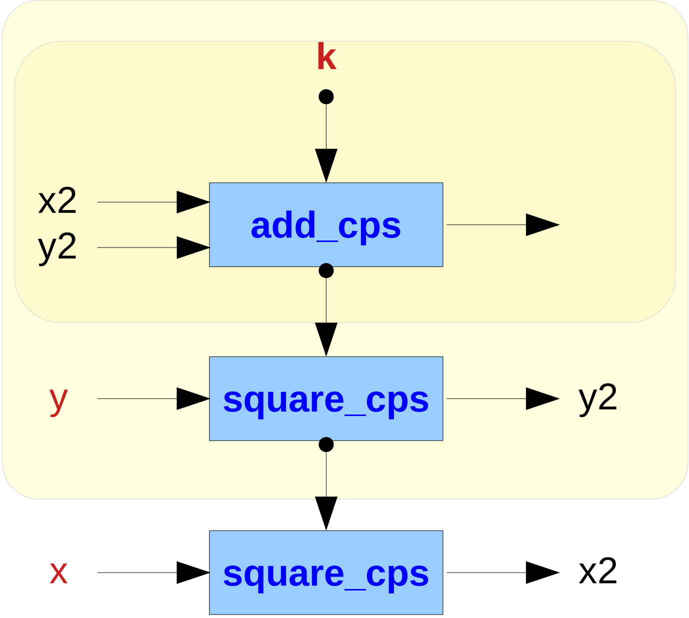
https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

CPS (Continuation Passing Style)

```
pythagoras_cps :: Int -> Int -> ((Int -> r) -> r)
pythagoras_cps x y =
  \k -> square_cps
          x $
          (\x2 ->      square_cps
                           y $
                           (\y2 ->      add_cps
                                            x2
                                            y2 $
                                            k ))
```

(\x2 -> ...) continuation

(\y2 -> ...) continuation



https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

CPS (Continuation Passing Style)

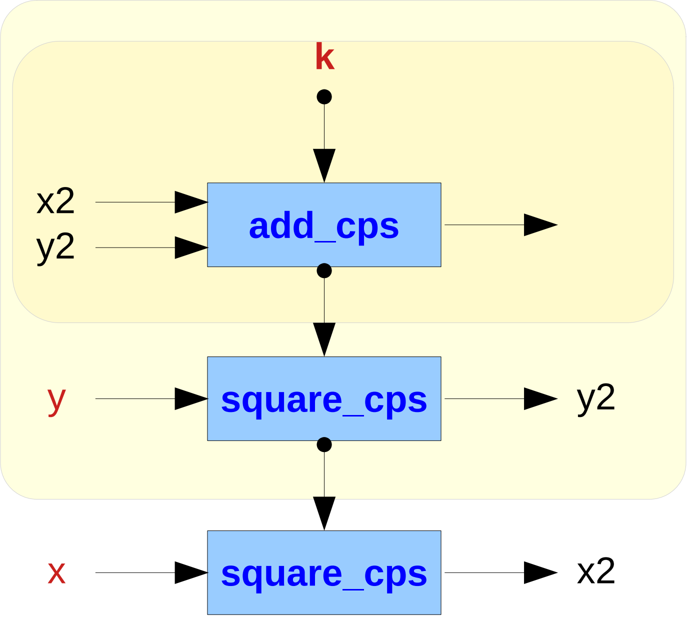
```
pythagoras_cps :: Int -> Int -> ((Int -> r) -> r)
pythagoras_cps x y = \k ->
                    square_cps x $ \x2 ->
                    square_cps y $ \y2 ->
                    add_cps x2 y2 $ k
```

square **x** and throw the result into the (**\x2** -> ...) continuation

square **y** and throw the result into the (**\y2** -> ...) continuation

add **x_squared** and **y_squared** and throw the result

into the top level/program continuation **k**.



We can try it out in GHCi by passing print as the program continuation:

```
*Main> pythagoras_cps 3 4 print
25
```

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

CPS (Continuation Passing Style)

continuations can be used in a similar fashion,

for implementing interesting **control flow** in **monads**.

Note that there usually are alternative techniques

for such use cases,

especially in tandem with **laziness**.

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

CPS (Continuation Passing Style)

The mtl library, which is shipped with GHC,

has the module Control.Monad.Cont.

This module provides the Cont type,

which implements Monad and some other useful functions.

The following snippet shows the pyth' function using **Cont**:

https://en.wikipedia.org/wiki/Continuation-passing_style

CPS (Continuation Passing Style)

```
pow2_m :: Float -> Cont a Float
pow2_m a = return (a ** 2)

pyth_m :: Float -> Float -> Cont a Float
pyth_m a b = do
  a2 <- pow2_m a
  b2 <- pow2_m b
  anb <- cont (add' a2 b2)
  r <- cont (sqrt' anb)
  return r
```

https://en.wikipedia.org/wiki/Continuation-passing_style

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf