# Recursie

Prof. dr. Kris Luyten
Jo Vermeulen

“Geavanceerde Programmeertechnologie”
Academiejaar 2011-2012


http://nl.wikipedia.org/wiki/Droste-effect

# Het venijn zit in de staart

# Het venijn zit in de staart

**Staartrecursie met afhankelijkheid**

```
int fac(int n){
 if(n==0)
  return 1;
 else
  return
     n * fac (n-1);
}
```

**“Vrije” staartrecursie**

```
int fac(int n, int acc){
 if(n==0)
  return 1 * acc;
 else{
  acc = n * acc;
  return
     fac (n-1,acc);
  }
}
```

# Het venijn zit in de staart

```
int fac(int n){
 if(n==0)
   return 1;
 else
   return
      n * fac (n-1);
}
```

```
int fac(int n, int acc){
 if(n==0)
   return 1 * acc;
 else{
   acc = n * acc;
   return
      fac (n-1,acc);
```

1 Recursieve oproep. Fac (n-1) wordt op de stack gepushed

# Het venijn zit in de staart

```
int fac(int n){
 if(n==0)
   return 1;
 else
   return
     n * fac (n-1);
}
```

```
int fac(int n, int acc){
 if(n==0)
   return 1 * acc;
 else{
   acc = n * acc;
   return
      fac (n-1,acc);
   }
}
```

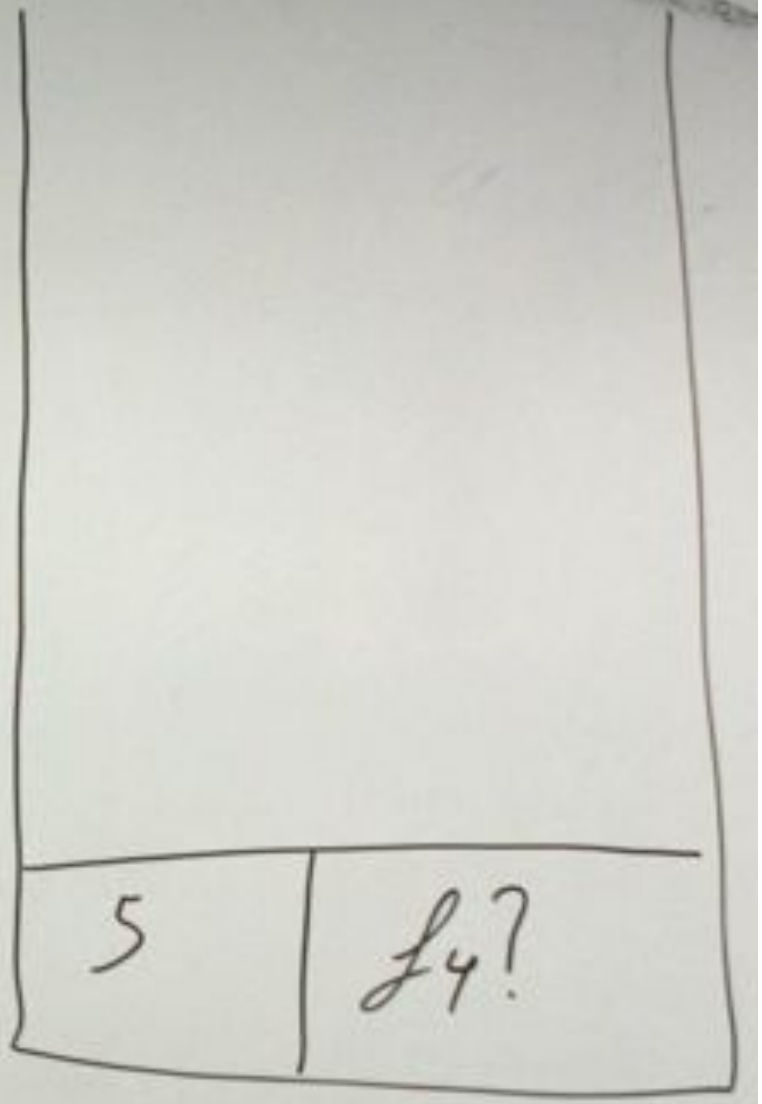
2

Maar de uitvoering van n * moet wachten tot fac terugkeert met resultaat

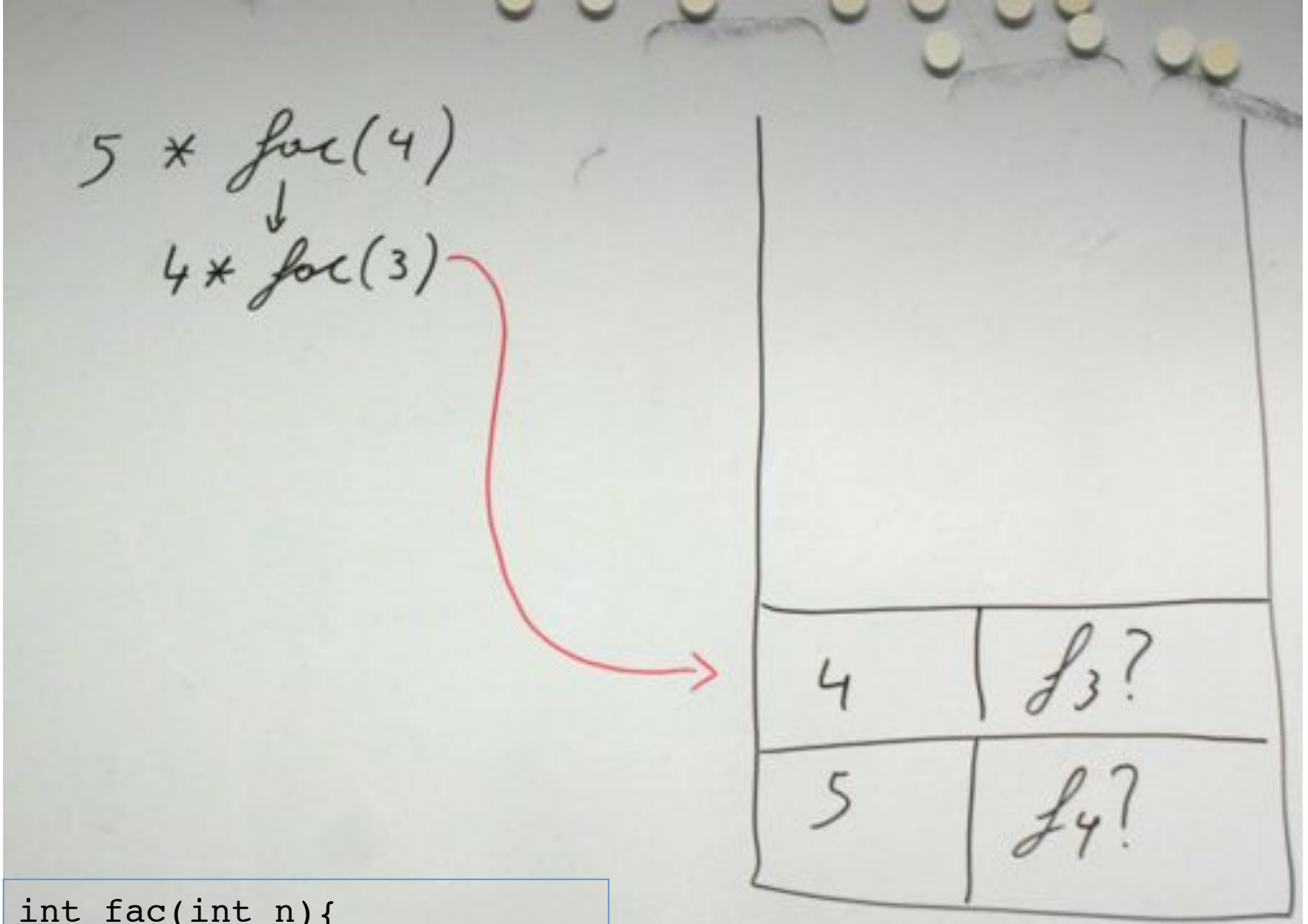
$5 * fac(4)$

$\downarrow$

$4 * fac(3)$




```
int fac(int n){
 if(n==0) return 1;
 else return n * fac (n-1);
}
```




```
int fac(int n){
 if(n==0) return 1;
 else return n * fac (n-1);
}
```

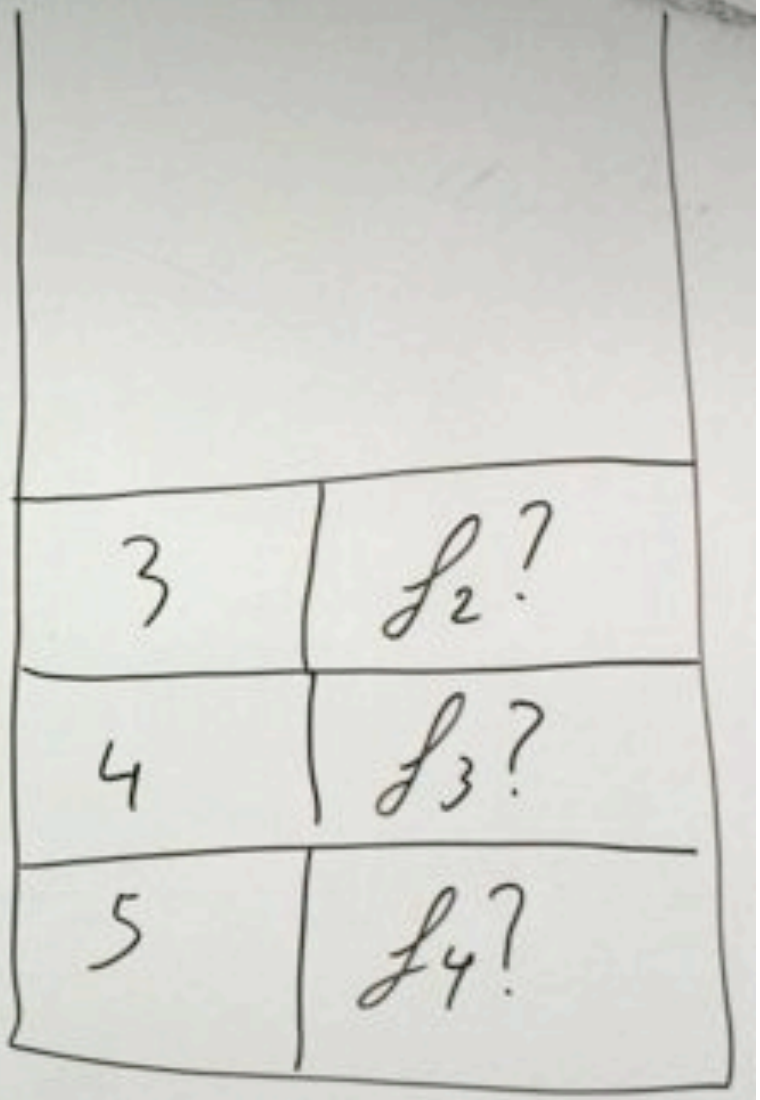
$5 * fac(4)$

$\downarrow$

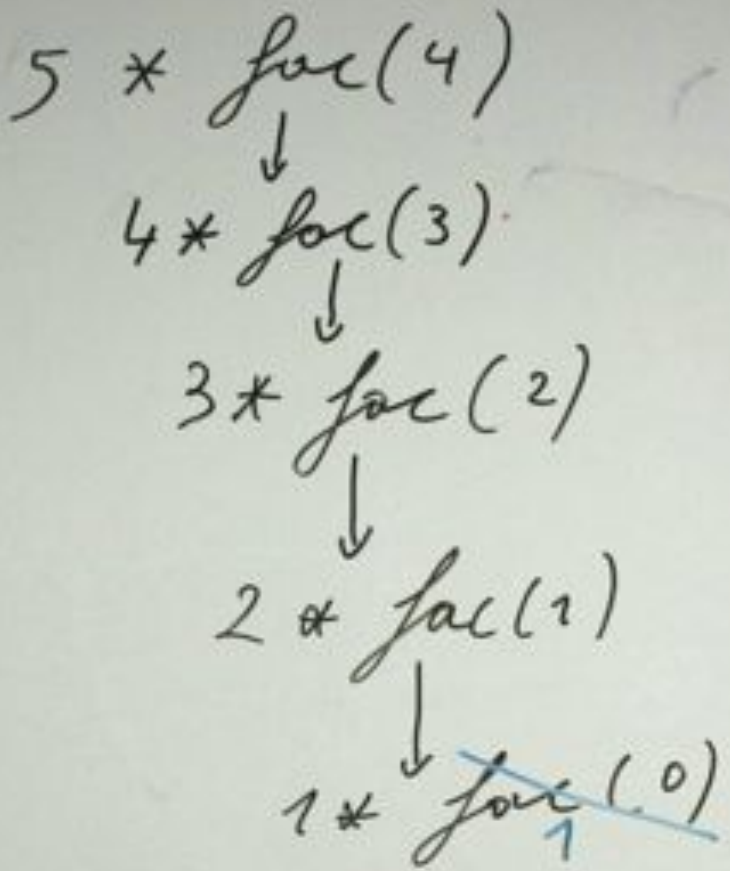
$4 * fac(3)$

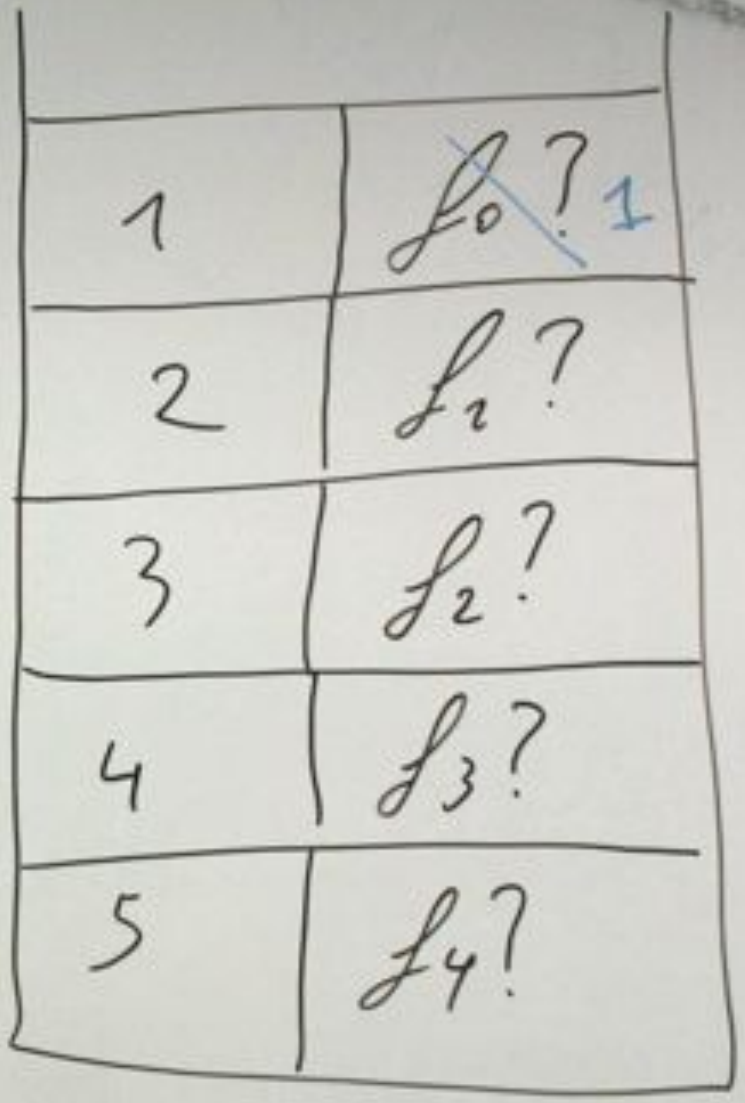
$\downarrow$

$3 * fac(2)$

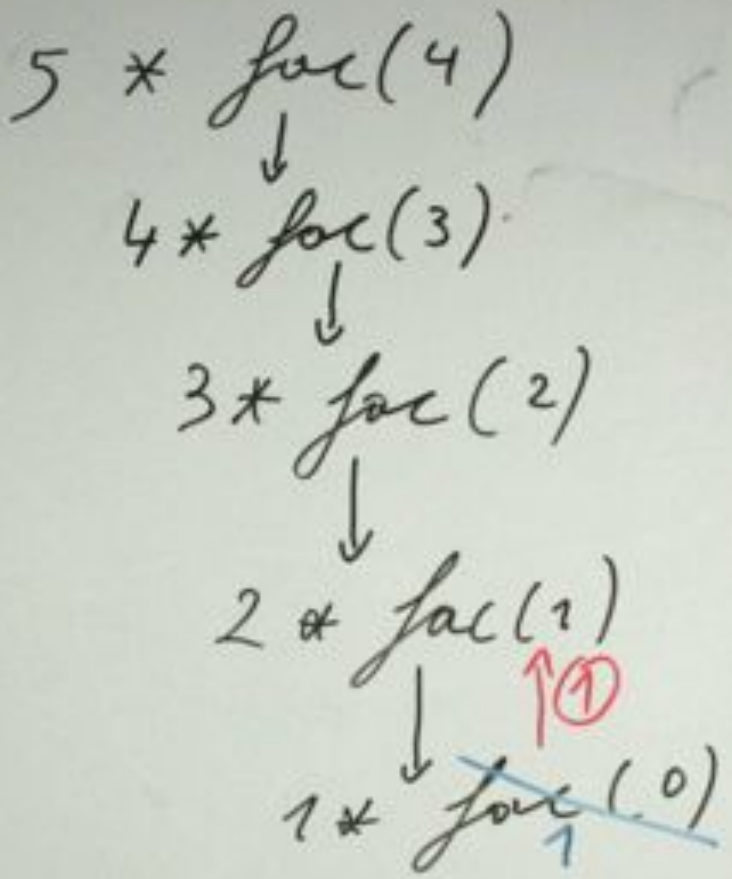


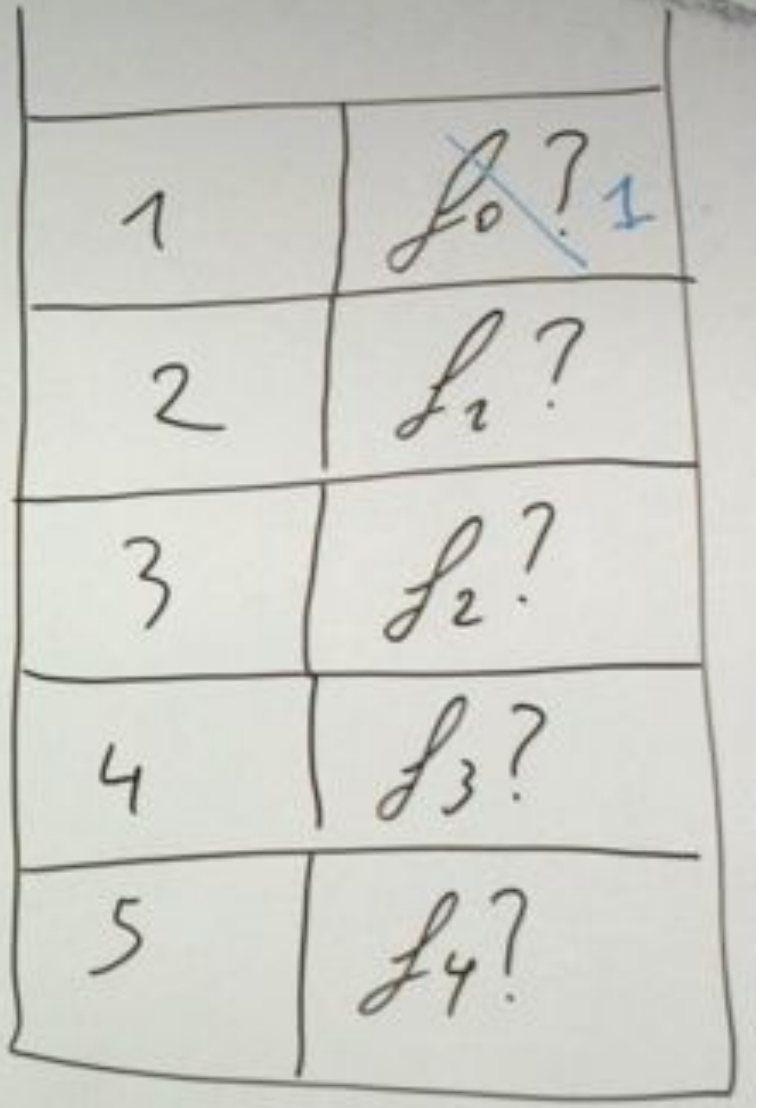

```
int fac(int n){
 if(n==0) return 1;
 else return n * fac (n-1);
}
```



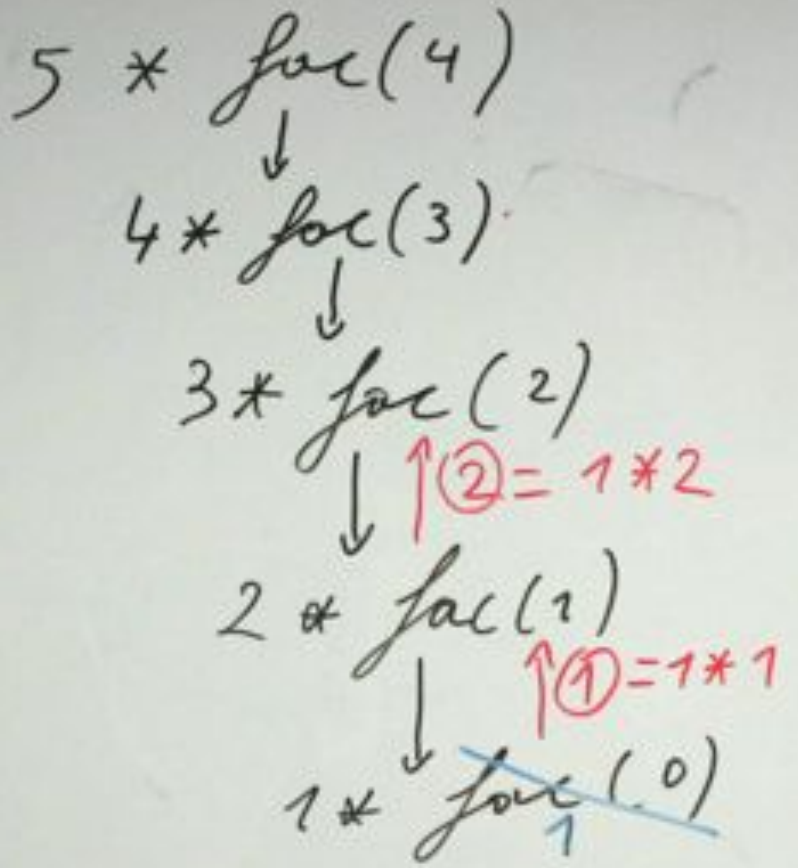



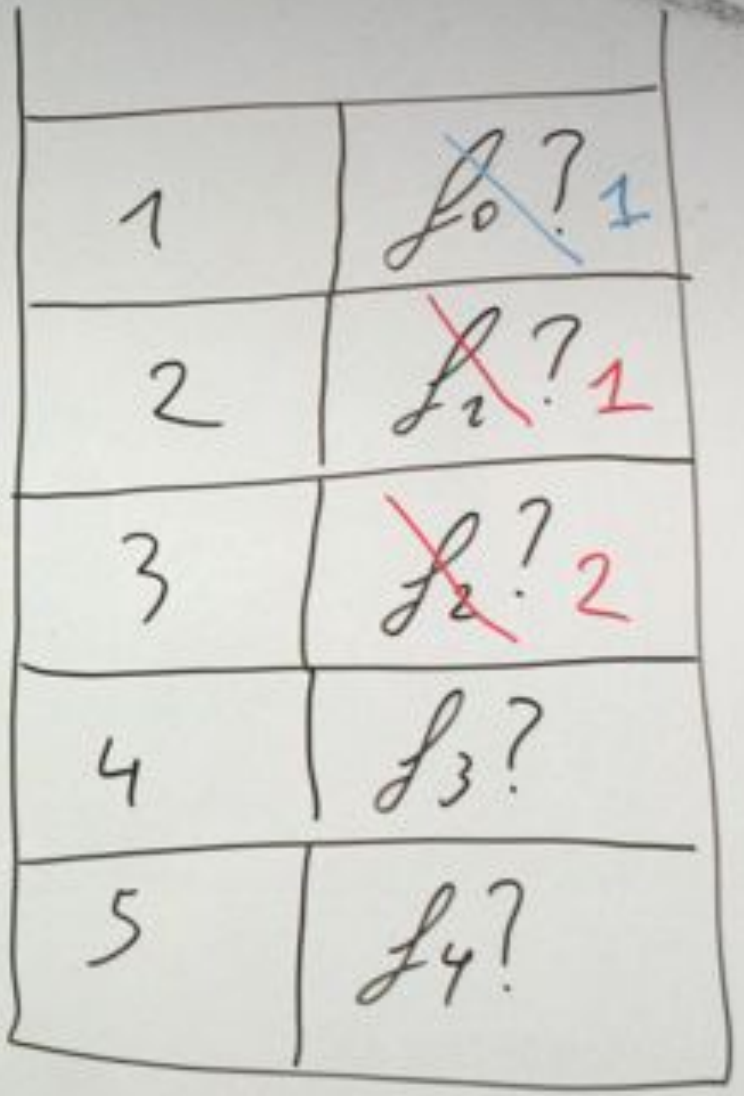

```
int fac(int n){
 if(n==0) return 1;
 else return n * fac (n-1);
}
```



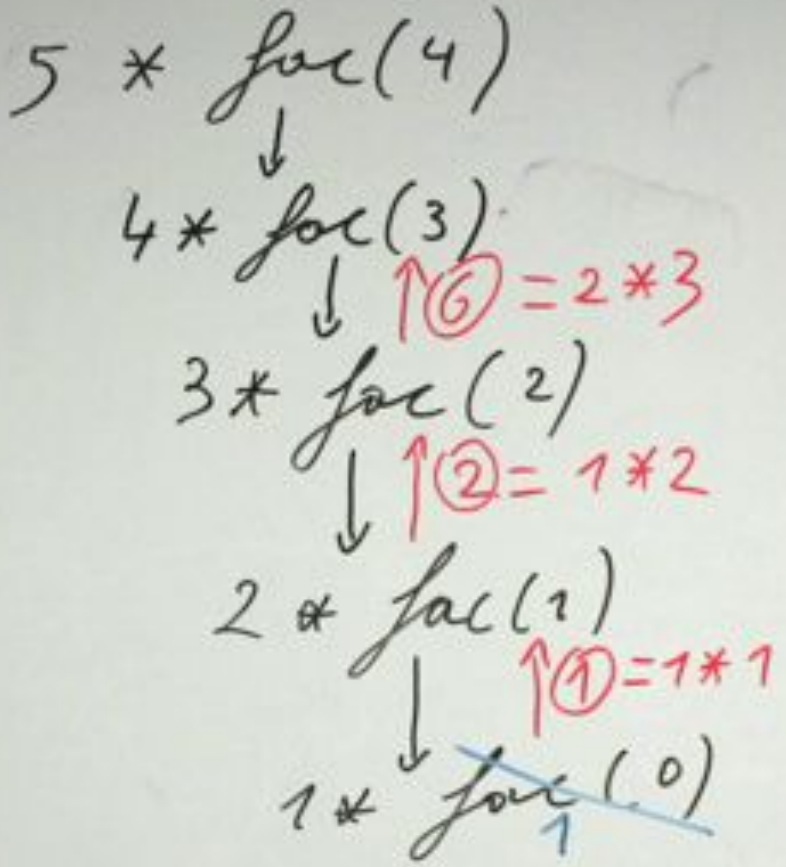



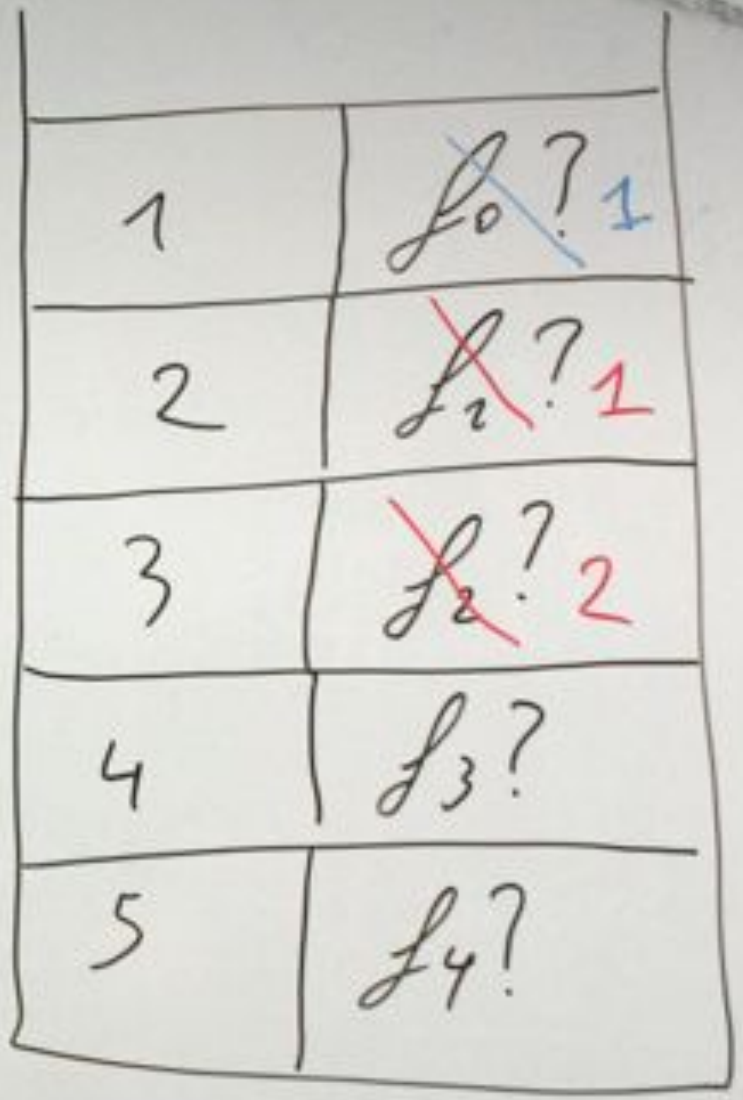

```
int fac(int n){
 if(n==0) return 1;
 else return n * fac (n-1);
}
```



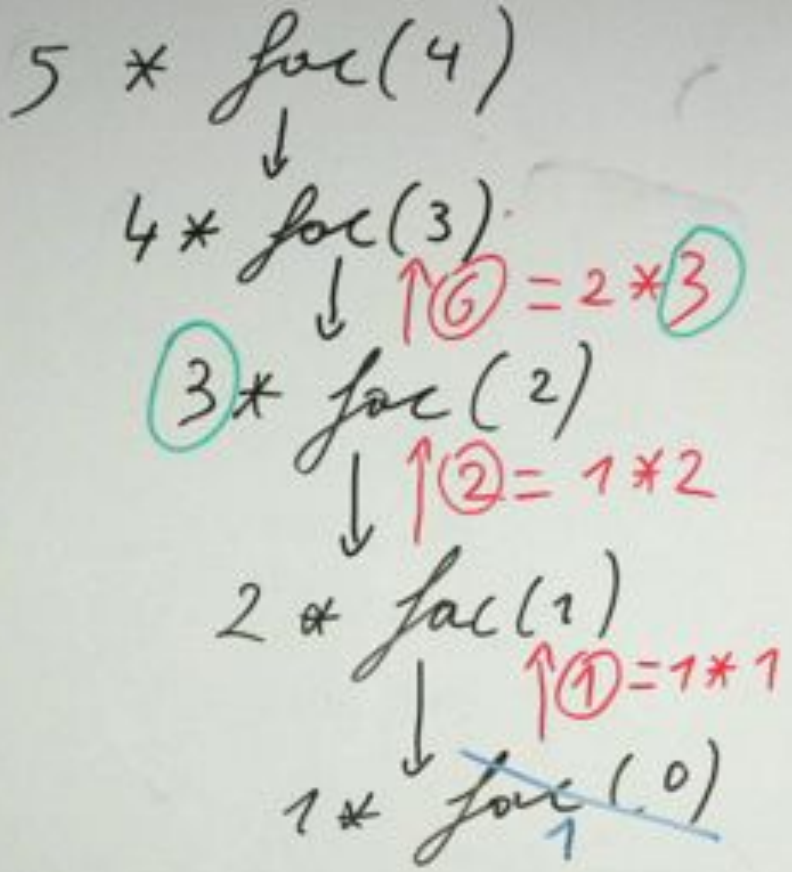



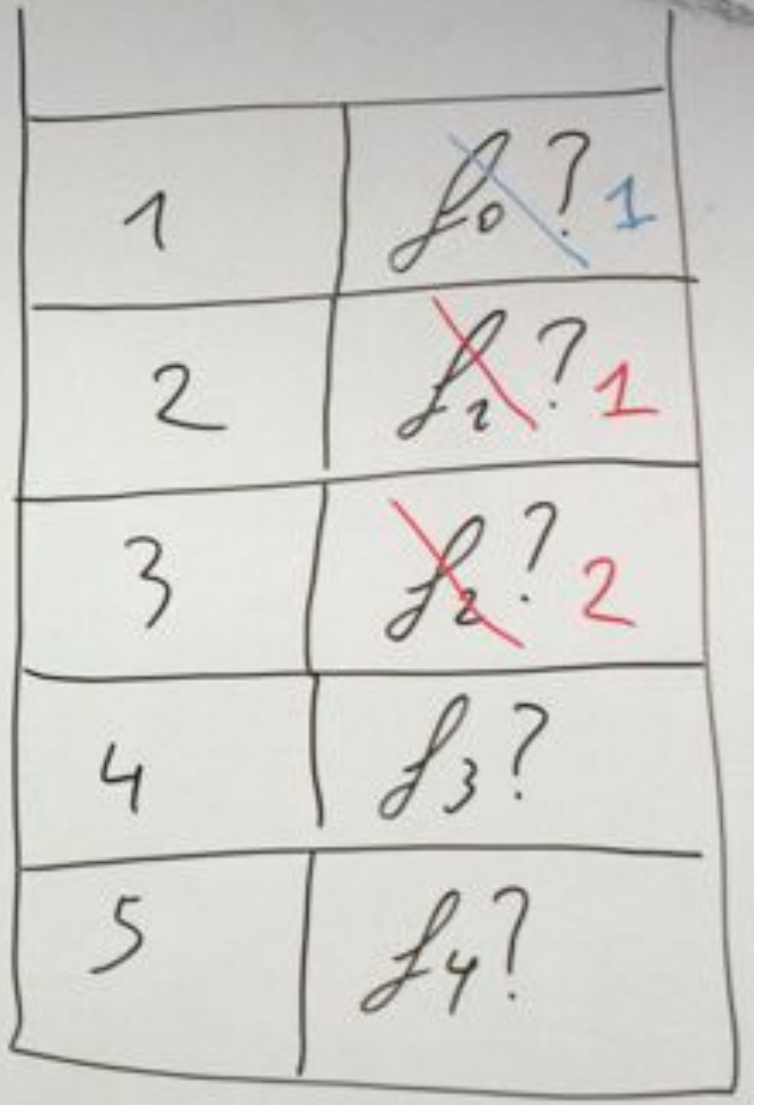

```
int fac(int n){
 if(n==0) return 1;
 else return n * fac (n-1);
}
```



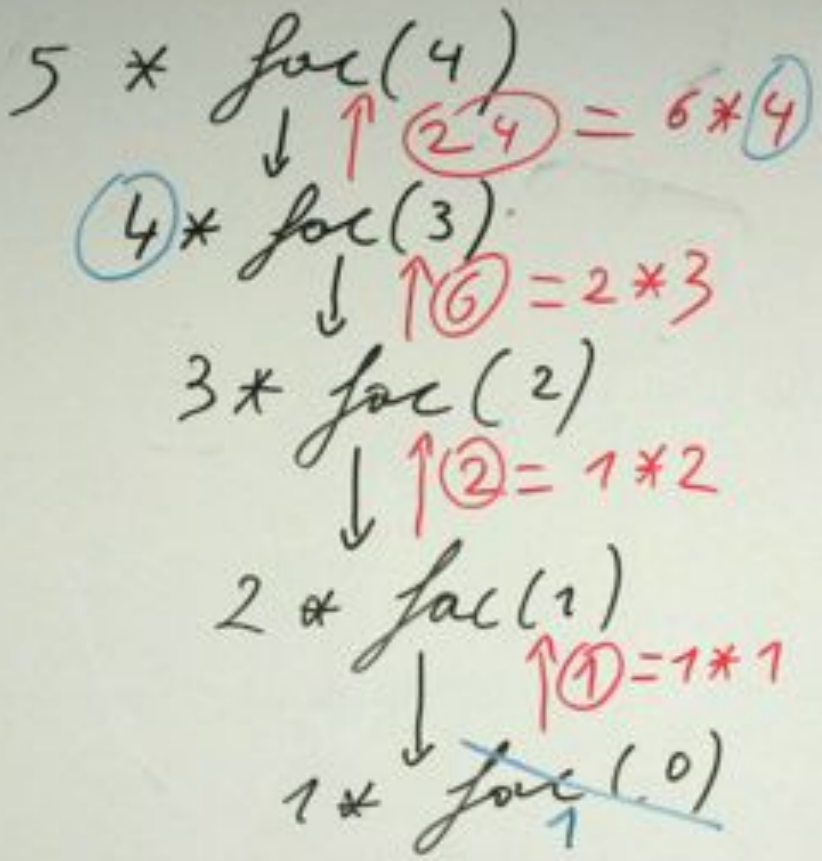




```
int fac(int n){
 if(n==0) return 1;
 else return n * fac (n-1);
}
```



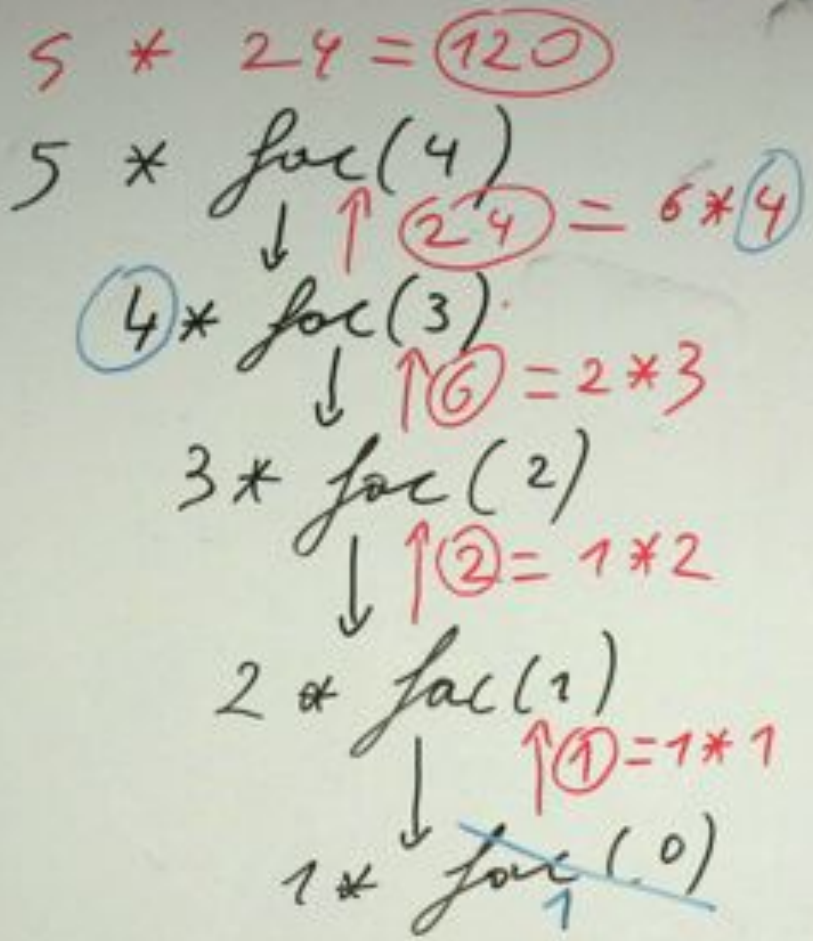



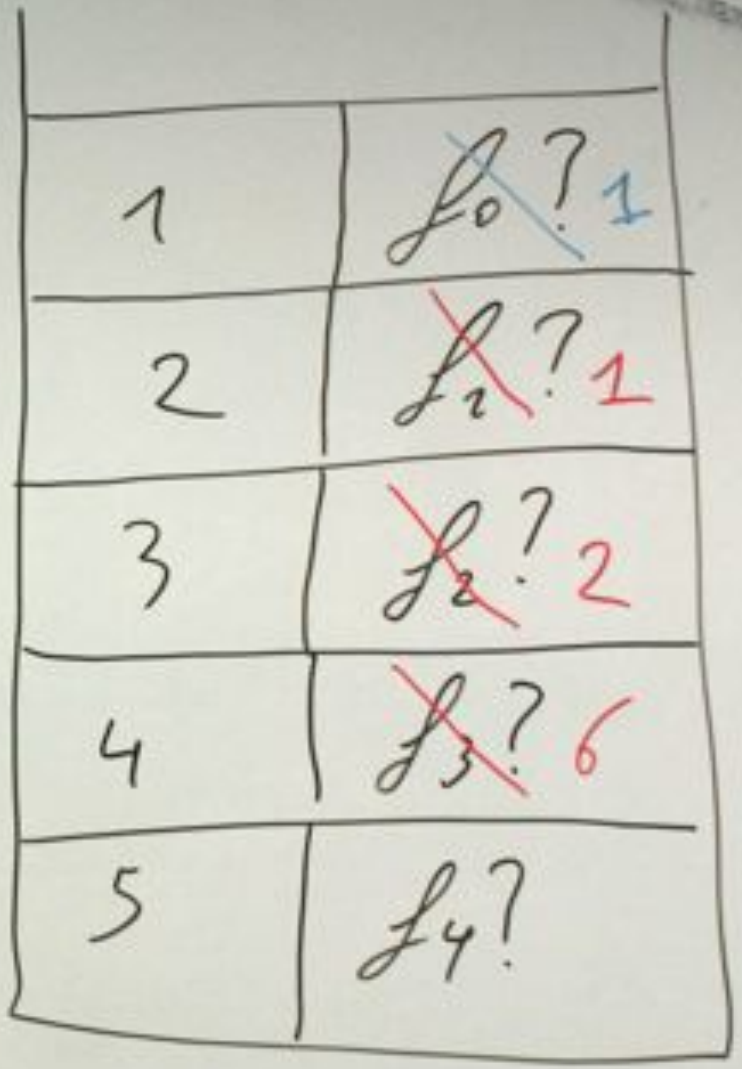

```
int fac(int n){
 if(n==0) return 1;
 else return n * fac (n-1);
}
```




| | |
|---|---|
| 1 | ~~$f_0$~~ ? 1 |
| 2 | ~~$f_1$~~ ? 1 |
| 3 | ~~$f_2$~~ ? 2 |
| 4 | ~~$f_3$~~ ? 6 |
| 5 | $f_4$ ? |

```
int fac(int n){
 if(n==0) return 1;
 else return n * fac (n-1);
}
```




```
int fac(int n){
 if(n==0) return 1;
 else return n * fac (n-1);
}
```

# Tail call optimalisatie

- Bij een tail call:
  - stack gebruik kan (deels) vermeden worden: frame wordt vervangen ipv gepusht op stack
  - snellere uitvoering code door lagere overhead
- Sterk afhankelijk van compiler/interpeter
  - Haskell, ML en Scheme ingebouwde tail call eliminatie
  - Belangrijk voor functionele en logische programmeertalen aangezien deze typisch veel stackspace gebruiken

# Recursie “lichter” maken

Oplossing opbouwen bij indalen recursie, niet bij opstijgen

# Het venijn zit in de staart

**Staartrecursie met afhankelijkheid**

```
int fac(int n){
 if(n==0)
  return 1;
 else
  return
     n * fac (n-1);
}
```

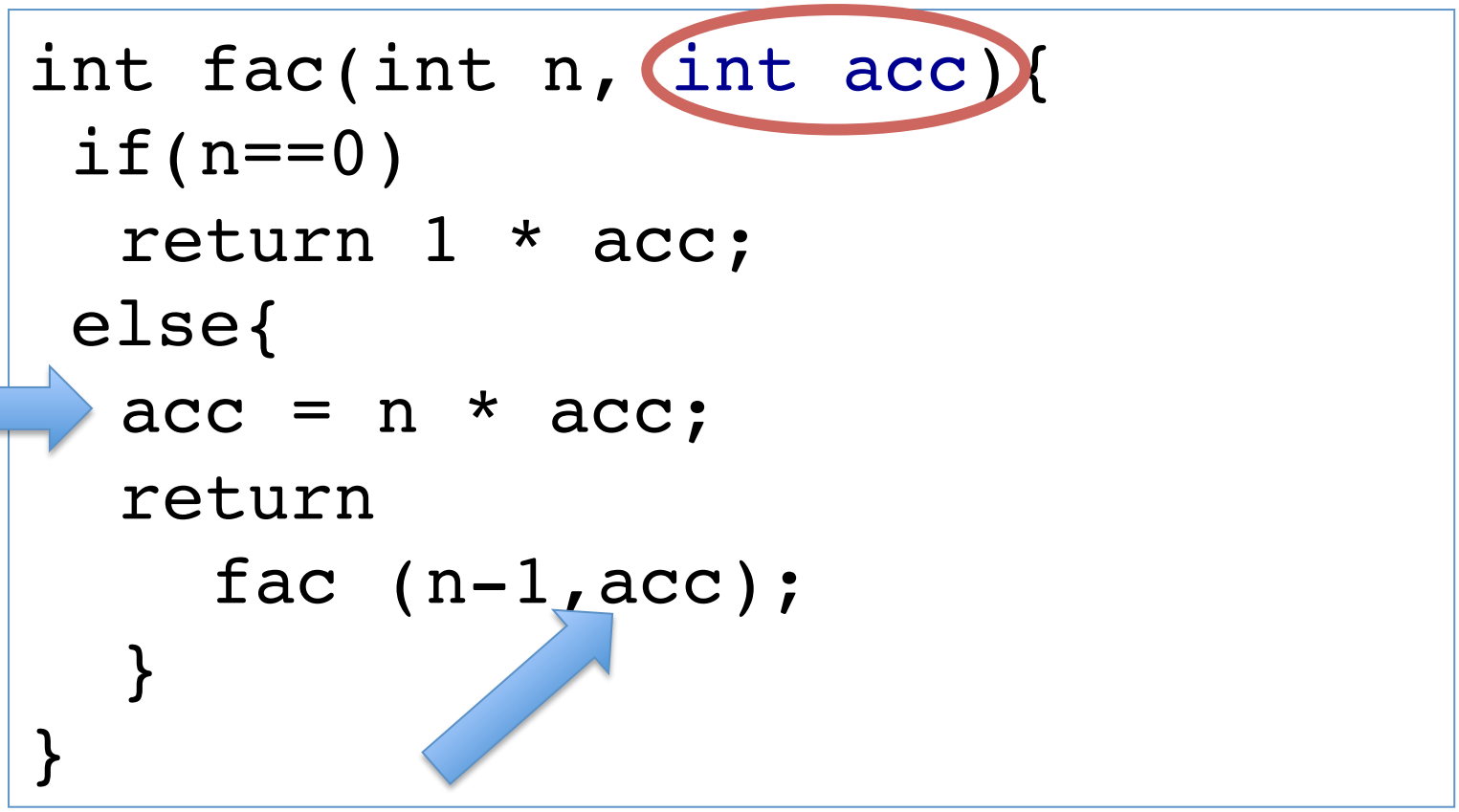
**“Vrije” staartrecursie**

```
int fac(int n, int acc){
 if(n==0)
  return 1 * acc;
 else{
  acc = n * acc;
  return
     fac (n-1,acc);
  }
}
```

Tail call optimalisatie door *accumulerende parameter*

# Accumelerende parameters

Doel: terugkeren in de call stack overbodig maken

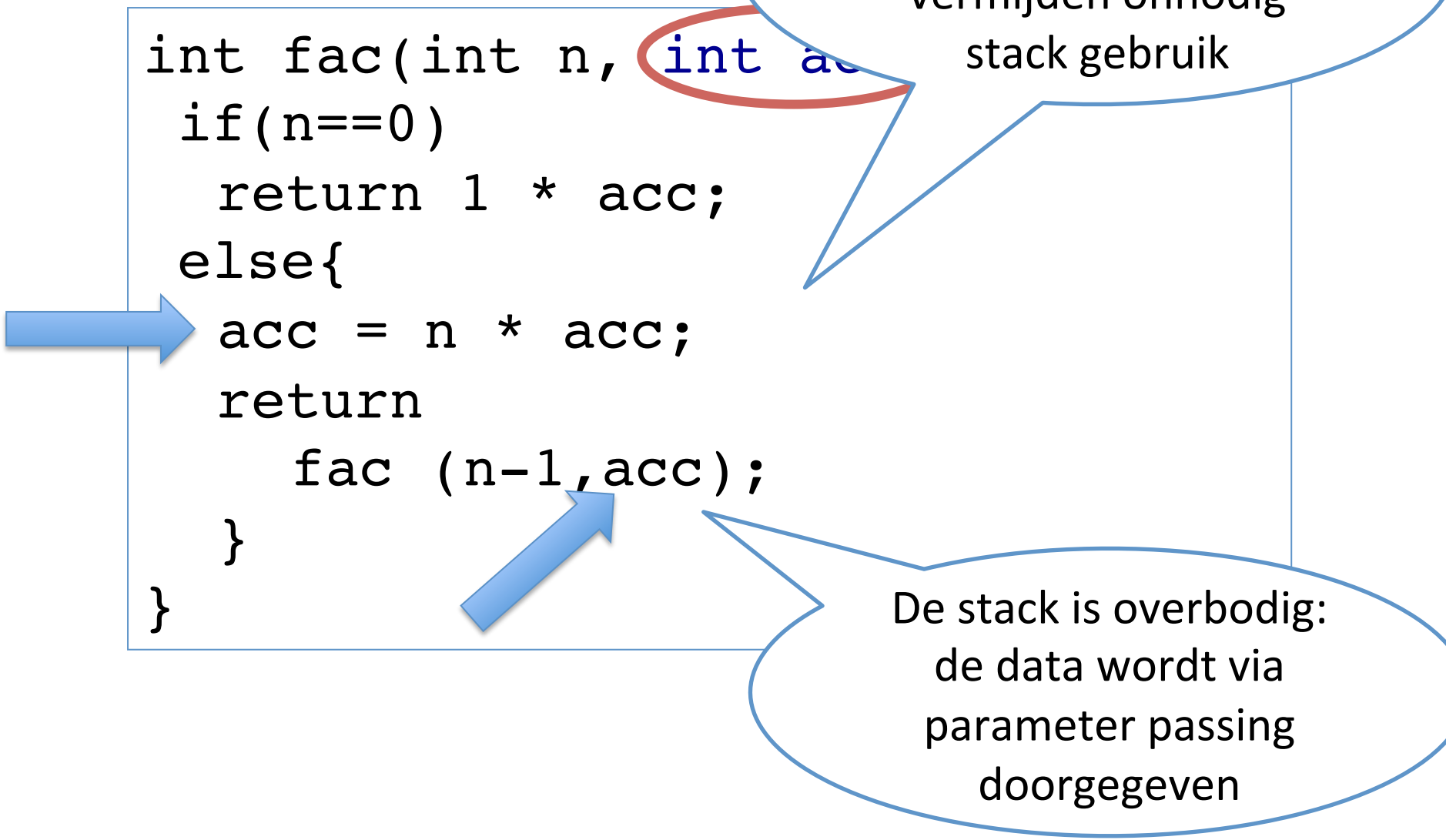

int fac(int n, int acc){
if(n==0)
return 1 * acc;
else{
acc = n * acc;
return
fac (n-1,acc);
}
}

```
int fac(int n, int acc){
 if(n==0)
   return 1 * acc;
 else{
   acc = n * acc;
   return
      fac (n-1,acc);
   }
}
```

Berekening wordt uitgevoerd en meegegeven verder in de recursie

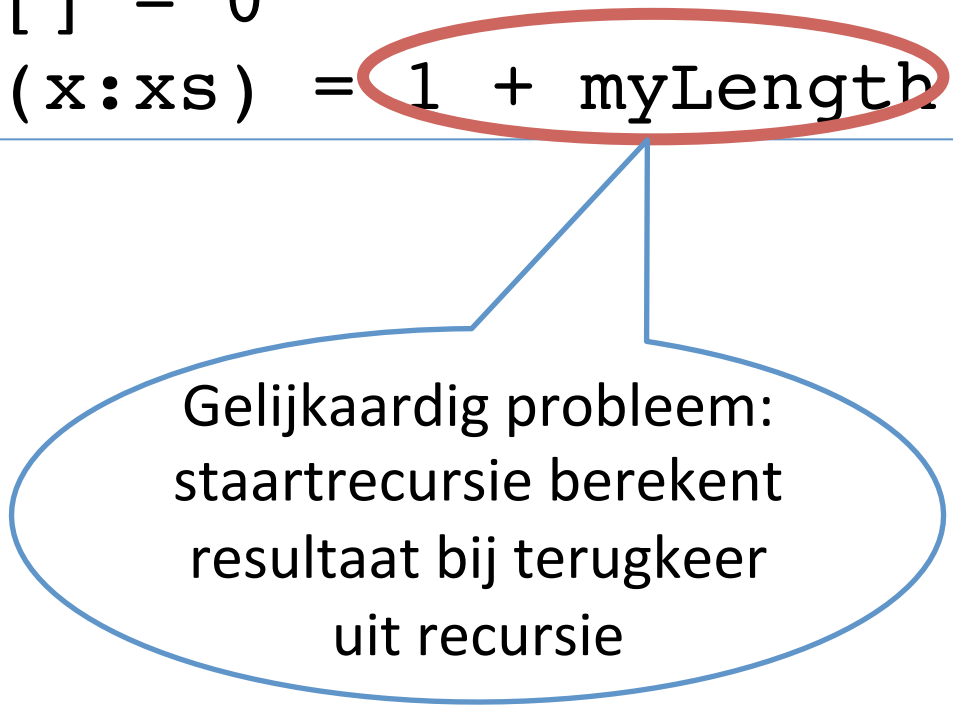
```
int fac(int n, int acc){
 if(n==0)
   return 1 * acc;
 else{
   acc = n * acc;
   return
      fac (n-1,acc);
   }
}
```

De stack is overbodig: de data wordt via parameter passing doorgegeven


int fac(int n, int ac
if(n==0)
return 1 * acc;
else{
acc = n * acc;
return
fac (n-1,acc);
}
}
De meeste moderne compilers zien dit en vermijden onnodig stack gebruik
De stack is overbodig: de data wordt via parameter passing doorgegeven

```haskell
-- Haskell
myLength :: [a] -> Int
myLength [] = 0
myLength (x:xs) = 1 + myLength xs
```

```haskell
-- Haskell
myLength :: [a] -> Int
myLength [] = 0
myLength (x:xs) = 1 + myLength xs
```

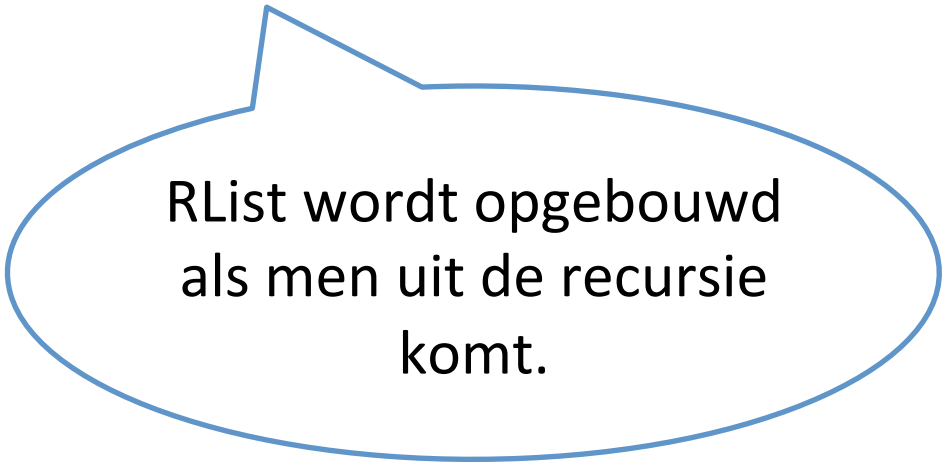


```haskell
-- Haskell
myLength :: [a] -> Int
myLength x = myLength_acc x 0

myLength_acc :: [a] -> Int -> Int
myLength_acc [] t = t;
myLength_acc (x:xs) t =
      myLength_acc xs (t+1)
```

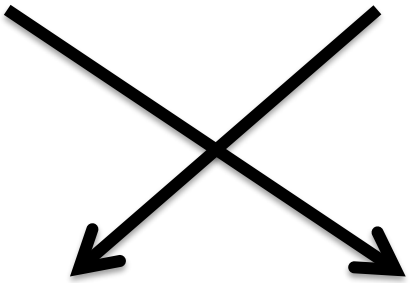
t als parameter om de berekening uit te voeren

```
-- Prolog
keerom([],[]).
keerom([Head|Tail], RList) :-
   keerom(Tail, RTail),
   append(RTail, [Head], RList).
```



```
-- Prolog
append([],X,X).
append([Head|Tail], List, [Head|AppTail]) :-
     append(Tail, List, AppTail).
```

Append biedt in feite een accumulerende parameter aan.
Let op: Acc begint leeg en wordt "gevuld" in de recursie



```
-- Prolog
reverse([],X,X).
reverse([Head|Tail], Acc, List) :-
   reverse(Tail, [Head|Acc], List).

-- "oproep"
reverse(X, Y) :- reverse(X, [], Y).
```

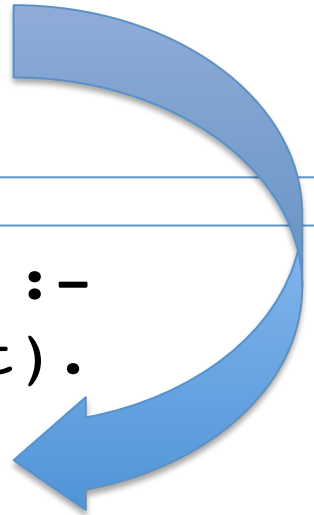
```
?- reverse([1, 2, 3, 4], List).
List = [4, 3, 2, 1].
```

```
?- reverse([1, 2, 3, 4], List).
List = [4, 3, 2, 1].
```

```
reverse([1,2,3,4], Y) :-
   reverse([1,2,3,4], [], Y).
```

```
?- reverse([1, 2, 3, 4], List).
List = [4, 3, 2, 1].
```

```
reverse([1,2,3,4], Y) :-
   reverse([1,2,3,4], [], Y).
```



```
-- reverse([Head|Tail], Acc, List) :-
--    reverse(Tail, [Head|Acc], List).
reverse([1|2,3,4], [], List)
   -> reverse([2,3,4], [1], List)
      -> reverse([3,4], [2,1], List)
         -> reverse([4], [3,2,1], List)
          -> reverse([], [4,3,2,1], List)
```

```
?- reverse([1, 2, 3, 4], List).
List = [4, 3, 2, 1].
```

```
reverse([1,2,3,4], Y) :-
   reverse([1,2,3,4], [], Y).
```

```
-- reverse([Head|Tail], Acc, List) :-
--    reverse(Tail, [Head|Acc], List).
reverse([1|2,3,4], [], List)
   -> reverse([2,3,4], [1], List)
      -> reverse([3,4], [2,1], List)
         -> reverse([4], [3,2,1], List)
          -> reverse([], [4,3,2,1], List)
```

```
-- reverse([],X,X).
           reverse([], [4,3,2,1], [4,3,2,1]).
```