Fifth Chapter

Back on Recursion

- Avoiding Recursion
  - Non-Recursive Form of Tail Recursion
  - Transformation to Tail Recursion
  - Generic Algorithm Using a Stack
- Back-tracking
- Conclusion on Recursion

Why do you want to avoid recursion

What gets done on Function Calls

1. Create a function frame on the stack
2. Push (copy) value of parameters
3. Execute function
4. Pop return value
5. Destruct stack frame

Recursion does not interfere with this schema

- Recursion can thus be less efficient than iterative solutions
- In time: function calling has a price
- In space: the call stack must be stored

Example: gcd of two natural integers

Greatest Common Divisor

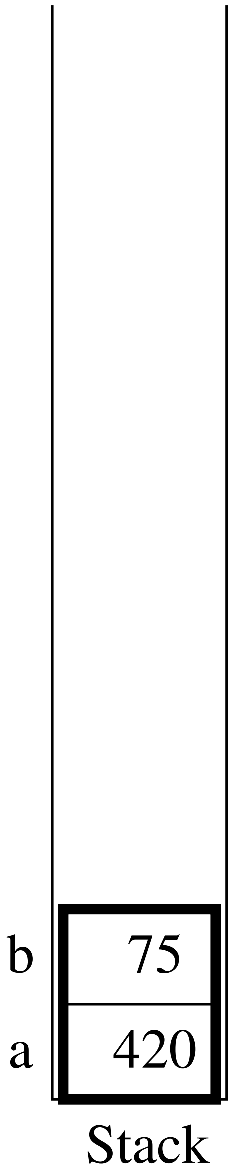
gcd(a, b : Integer) = (r : Integer)

- Precondition: $a \geq b \geq 0$
- Postcondition: $(a \bmod r = 0)$ and $(b \bmod r = 0)$ and $\neg\,(\exists s, (s > r) \wedge (a \bmod s = 0) \wedge (b \bmod s = 0))$

Recursive Definition

$$\boxed{\begin{array}{l} \textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad\quad \textbf{else } \; r \leftarrow gcd(b, a \bmod b) \end{array}}$$

Computation of gcd(420,75)



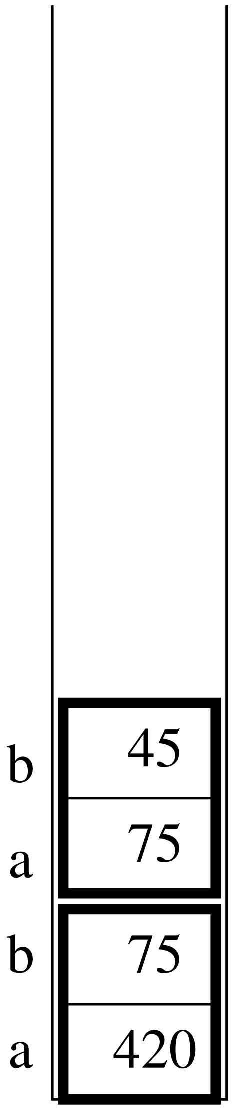
$$\begin{array}{|l|}\hline \textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad \textbf{else } \; r \leftarrow gcd(b, a \bmod b) \\ \hline \end{array}$$

- $gcd(420, 75) =$

Computation of gcd(420,75)

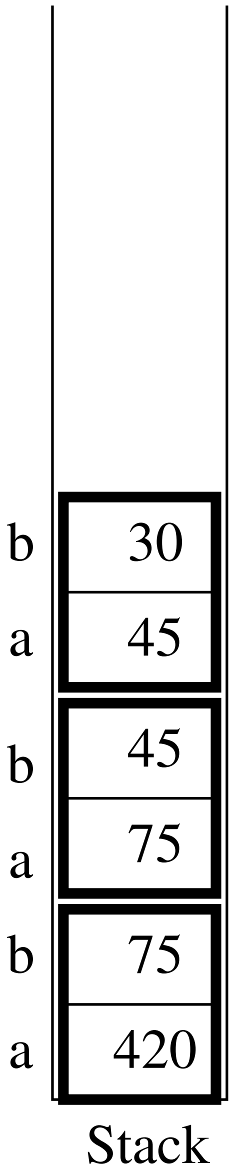
$$\begin{array}{|l|}\hline \textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad\quad \textbf{else } \ r \leftarrow gcd(b, a \bmod b) \\ \hline \end{array}$$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) =$



Stack

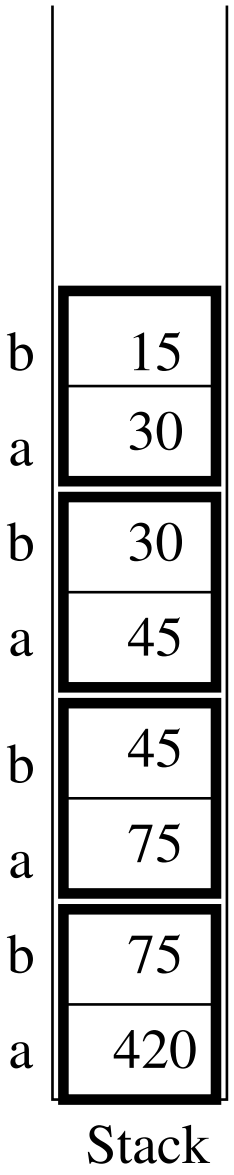
Computation of gcd(420,75)



$$\begin{array}{|l|}\hline \textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad \textbf{else } \; r \leftarrow gcd(b, a \bmod b) \\ \hline \end{array}$$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) =$

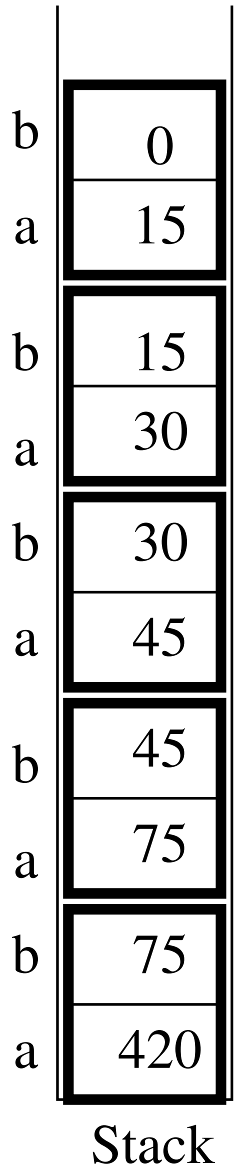
Computation of gcd(420,75)



$$\begin{array}{l} \textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad \textbf{else } \; r \leftarrow gcd(b, a \bmod b) \end{array}$$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) = gcd(30, 15)$
- $gcd(30, 15) =$

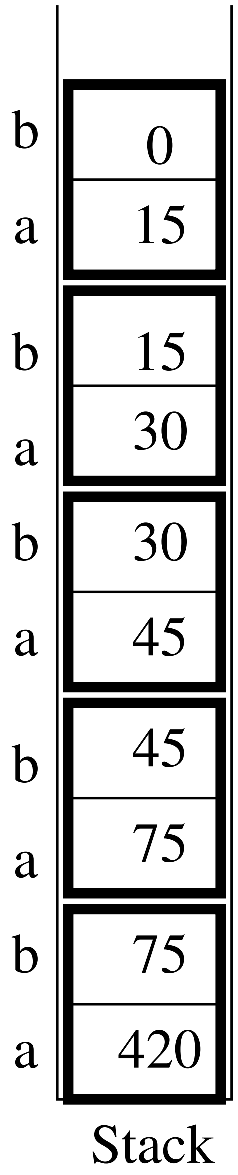
Computation of gcd(420,75)



$$\begin{array}{|l|}\hline \textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad\quad \textbf{else } \ r \leftarrow gcd(b, a \bmod b) \\ \hline \end{array}$$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) = gcd(30, 15)$
- $gcd(30, 15) = gcd(15, 0)$
- $gcd(15, 0) =$

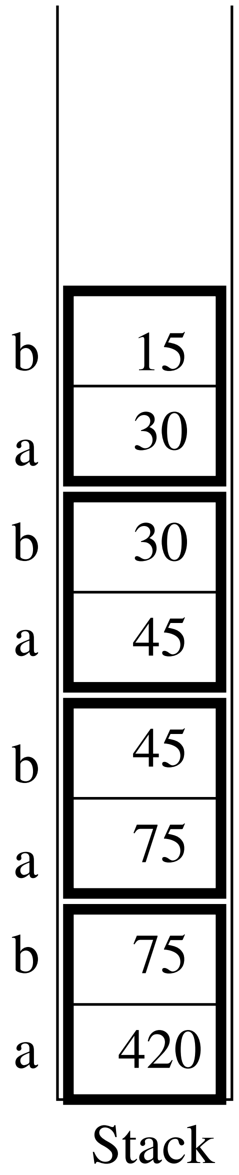
Computation of gcd(420,75)



$$\begin{array}{l} \textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad \textbf{else } \; r \leftarrow gcd(b, a \bmod b) \end{array}$$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) = gcd(30, 15)$
- $gcd(30, 15) = gcd(15, 0)$
- $gcd(15, 0) = 15$
  this is the Base Case

Computation of gcd(420,75)

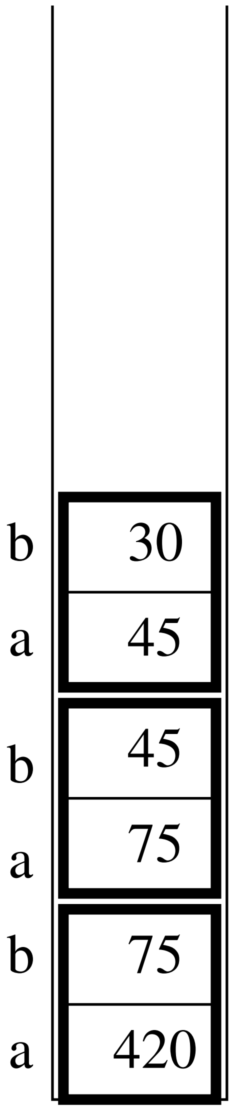


$$
\begin{array}{|l|}\hline
\textbf{if } b = 0 \textbf{ then } r \leftarrow a \\
\qquad\qquad \textbf{else } \; r \leftarrow gcd(b, a \bmod b) \\ \hline
\end{array}
$$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) = gcd(30, 15)$
- $gcd(30, 15) = gcd(15, 0)$
- $gcd(15, 0) = 15$
  this is the Base Case

- Let's pop parameters
- $r \leftarrow r_{int}$ (no other computation: $G(x, y) = y$)

Computation of gcd(420,75)

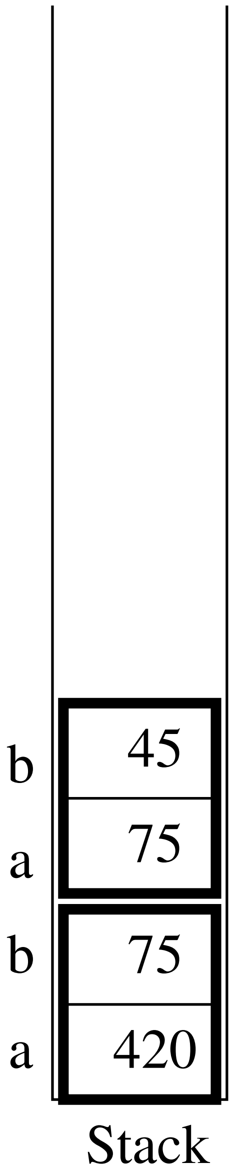


Stack

$$\boxed{\begin{array}{l} \textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad\quad \textbf{else } \ r \leftarrow gcd(b, a \bmod b) \end{array}}$$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) = gcd(30, 15)$
- $gcd(30, 15) = gcd(15, 0) \mathbf{= 15}$
- $gcd(15, 0) = 15$
  this is the Base Case
- Let's pop parameters
- $r \leftarrow r_{int}$ (no other computation: $G(x, y) = y$)

Computation of gcd(420,75)

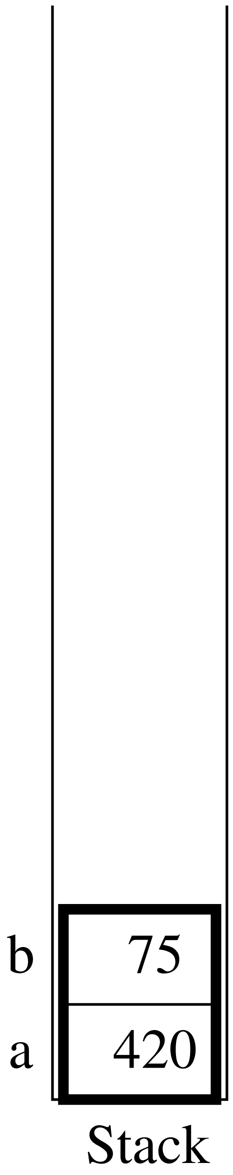


$$\boxed{\begin{array}{l}\textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad \textbf{else } \; r \leftarrow gcd(b, a \bmod b)\end{array}}$$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30)$
- $gcd(45, 30) = gcd(30, 15) \mathbf{= 15}$
- $gcd(30, 15) = gcd(15, 0) \mathbf{= 15}$
- $gcd(15, 0) = 15$
  this is the Base Case

- Let's pop parameters
- $r \leftarrow r_{int}$ (no other computation: $G(x, y) = y$)

Computation of gcd(420,75)



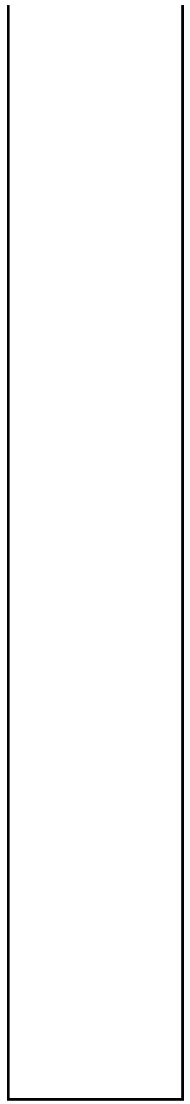
$$\boxed{\begin{array}{l}\textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad \textbf{else } \; r \leftarrow gcd(b, a \bmod b)\end{array}}$$

- $gcd(420, 75) = gcd(75, 45)$
- $gcd(75, 45) = gcd(45, 30) \mathbf{= 15}$
- $gcd(45, 30) = gcd(30, 15) \mathbf{= 15}$
- $gcd(30, 15) = gcd(15, 0) \mathbf{= 15}$
- $gcd(15, 0) = 15$
  this is the Base Case

- Let's pop parameters
- $r \leftarrow r_{int}$ (no other computation: $G(x, y) = y$)

Computation of gcd(420,75)

$$\boxed{\begin{array}{l} \textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad \textbf{else } \; r \leftarrow gcd(b, a \bmod b) \end{array}}$$



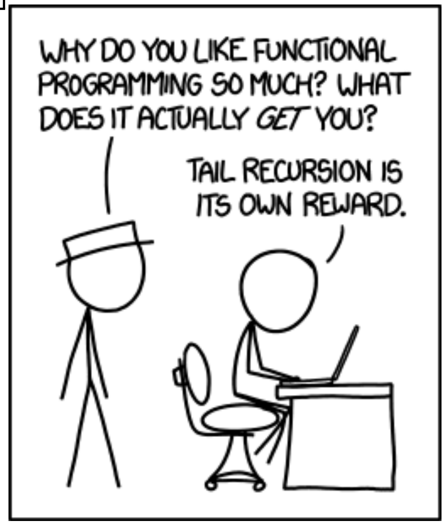
Stack

- $gcd(420, 75) = gcd(75, 45) \mathbf{= 15}$
- $gcd(75, 45) = gcd(45, 30) \mathbf{= 15}$
- $gcd(45, 30) = gcd(30, 15) \mathbf{= 15}$
- $gcd(30, 15) = gcd(15, 0) \mathbf{= 15}$
- $gcd(15, 0) = 15$
  this is the Base Case
- Let's pop parameters
- $r \leftarrow r_{int}$ (no other computation: $G(x, y) = y$)
- The result of initial call is known as early as from Base Case
  This is known as **Tail Recursion**

Computation of gcd(420,75)

$$\begin{array}{l} \textbf{if } b = 0 \textbf{ then } r \leftarrow a \\ \qquad\qquad \textbf{else } \ r \leftarrow gcd(b, a \bmod b) \end{array}$$

- $gcd(420, 75) = gcd(75, 45) \mathbf{= 15}$
- $gcd(75, 45) = gcd(45, 30) \mathbf{= 15}$
- $gcd(45, 30) = gcd(30, 15) \mathbf{= 15}$
- $gcd(30, 15) = gcd(15, 0) \mathbf{= 15}$
- $gcd(15, 0) = 15$
  this is the Base Case



http://xkcd.com/1270/

- Let's pop parameters
- $r \leftarrow r_{int}$ (no other computation: $G(x, y) = y$)

- The result of initial call is known as early as from Base Case
  This is known as **Tail Recursion**
- Factorial: multiplications during climb up
  $\Rightarrow$ **non-terminal** recursion

Stack

Transformation to Non-Recursive Form

Every recursive function can be changed to a non-recursive form

Several Methods depending on function:

- Tail Recursion: very simple transformation
- Non-Tail Recursion: two methods (only one is generic)

Compilers use these optimization techniques (amongst much others)

Non-Recursive Form of Tail Recursion

Cookbook to change Tail Recursion to Non-Recursive Form

- Generic recursive algorithm

$f(x)$:
  if cond(x) **then** $\textsc{BaseCase}$(x)
  else $\textsc{T}$(x); $r \leftarrow f(x_{int})$

Non-Recursive Form of Tail Recursion

Cookbook to change Tail Recursion to Non-Recursive Form

- Generic recursive algorithm

```
f(x):
  if cond(x) then BaseCase(x)
              else  T(x); r ← f(x_int)
```

- Equivalent iterative algorithm

```
f'(x):
  u ← x
  until cond(u) do
    T(u)
    u ← h(u)
  end
  BaseCase(u)
```

Non-Recursive Form of Tail Recursion

Cookbook to change Tail Recursion to Non-Recursive Form

- Generic recursive algorithm

$f(x)$:
  if cond(x) **then** $\textsc{BaseCase}(x)$
  else $\textsc{T}(x)$; $r \leftarrow f(x_{int})$

Example: get last char of string

last(s):
  if empty(s.tail) **then** $r \leftarrow s.head$
  else $r \leftarrow last(s.tail)$

- Equivalent iterative algorithm

$f'(x)$:
  $u \leftarrow x$
  until cond(u) **do**
    T(u)
    $u \leftarrow h(u)$
  end
  $\textsc{BaseCase}(u)$

Non-Recursive Form of Tail Recursion

Cookbook to change Tail Recursion to Non-Recursive Form

- Generic recursive algorithm

$f(x)$:
  if cond(x) **then** $\textsc{BaseCase}$(x)
  else $\textsc{T}$(x); $r \leftarrow f(x_{int})$

Example: get last char of string

last(s):
  if empty(s.tail) **then** $r \leftarrow s.head$
  else $r \leftarrow last(s.tail)$

- Equivalent iterative algorithm

$f'(x)$:
  $u \leftarrow x$
  until cond(u) **do**
    T(u)
    $u \leftarrow h(u)$
  end
  $\textsc{BaseCase}$(u)

last'(s):
  $l \leftarrow s$
  until $empty(l.tail)$ **do**
    // T(u) does nothing
    l←l.tail
  end
  $r \leftarrow l.head$

Other Examples

nth(s,i): get char number i out of s

nth(s,i):
  if n=0 **then** $r \leftarrow s.head$
          else $r \leftarrow nth(s.tail, i-1)$

Two arguments, still no T(u)

Other Examples

nth(s,i): get char number i out of s

nth(s,i):
  if n=0 **then** $r \leftarrow s.head$
  else $r \leftarrow nth(s.tail, i-1)$

Two arguments, still no T(u)

nth'(s,i):
  $l \leftarrow s$; $k \leftarrow i$
  until k=0 **do**
    l←l.tail; k←k-1
  end
  $r \leftarrow l.head$

Other Examples

nth(s,i): get char number i out of s

```
nth(s,i):
  if n=0 then r ← s.head
          else  r ← nth(s.tail, i − 1)
```

Two arguments, still no T(u)

```
nth'(s,i):
  l ← s; k ← i
  until k=0 do
    l←l.tail; k←k-1
  end
  r ← l.head
```

is_member(s,c): assess whether c is member of s

```
is_member(s,c):
  if empty(s) then r ← FALSE
  if s.head=c then r ← TRUE
              else  r ← is_memb(s.tail)
```

2 base cases, still no T(u)

Other Examples

nth(s,i): get char number *i* out of *s*

```
nth(s,i):
  if n=0 then r ← s.head
         else r ← nth(s.tail, i − 1)
```

Two arguments, still no T(u)

```
nth'(s,i):
  l ← s; k ← i
  until k=0 do
    l←l.tail; k←k-1
  end
  r ← l.head
```

is_member(s,c): assess whether *c* is member of *s*

```
is_member(s,c):
  if empty(s) then r ← FALSE
  if s.head=c then r ← TRUE
              else r ← is_memb(s.tail)
```

2 base cases, still no T(u)

```
is_memb'(s,c):
  l ← s
  until empty(l) OR l.head=c do
    l←l.tail
  end
  if empty(s) then r ← FALSE
  r ← TRUE
```

Last Example

Non-Recursive Form of GCD

$gcd(a, b)$:
  if $b = 0$ **then** $r \leftarrow a$
      else $r \leftarrow gcd(b, a \bmod b)$

$gcd'(a, b)$:
  $u \leftarrow a$; $v \leftarrow b$
  until v=0 **do**
    $temp \leftarrow v$
    $v \leftarrow u \bmod v$
    $u \leftarrow temp$
  end
  $r \leftarrow u$

- This is given by an immediate rewriting
- Computers are good at this kind of game (e.g., in compilers)
- Meta-programming troubling at first sight, but still fully mechanic

Non-Recursive form of Non-Tail functions

How to deal with non-tail functions?

- Previous method don't work because of those computations at recursive climb:
- Where should the ongoing computation be stored (they were stacked)?
  $fact(3) = 3 \times fact(2) = 3 \times 2 \times fact(1) = 3 \times 2 \times 1 = 3 \times 2 = 6$

Non-Recursive form of Non-Tail functions

How to deal with non-tail functions?

- Previous method don't work because of those computations at recursive climb:
- Where should the ongoing computation be stored (they were stacked)?
  $fact(3) = 3 \times fact(2) = 3 \times 2 \times fact(1) = 3 \times 2 \times 1 = 3 \times 2 = 6$

what's done is no more to do

- Computing during descent $\rightsquigarrow$ nothing left at climb $\rightsquigarrow$ Tail Recursion
  $fact(3) = \boxed{3} \times fact(2) = 3 \times 2 \times fact(1) = \boxed{6} \times fact(1) = \boxed{6} \times 1 = \boxed{6} = 6$
- One extra variable is enough for the storage of "ongoing" computation
  - Since these computations are done, store their result not the stack of operations
  - Adding an extra parameter to my recursive function does the trick
  - Prototype change $\rightsquigarrow$ put recursion into a *helper* function with more parameters

Non-Recursive form of Non-Tail functions

How to deal with non-tail functions?

- Previous method don't work because of those computations at recursive climb:
- Where should the ongoing computation be stored (they were stacked)?
  $fact(3) = 3 \times fact(2) = 3 \times 2 \times fact(1) = 3 \times 2 \times 1 = 3 \times 2 = 6$

what's done is no more to do

- Computing during descent $\rightsquigarrow$ nothing left at climb $\rightsquigarrow$ Tail Recursion
  $fact(3) = \boxed{3} \times fact(2) = 3 \times 2 \times fact(1) = \boxed{6} \times fact(1) = \boxed{6} \times 1 = \boxed{6} = 6$
- One extra variable is enough for the storage of "ongoing" computation
  - Since these computations are done, store their result not the stack of operations
  - Adding an extra parameter to my recursive function does the trick
  - Prototype change $\rightsquigarrow$ put recursion into a *helper* function with more parameters

Warning: this does not always work!

- Computations done out of order $\rightsquigarrow$ must be associative and commutative
- This (simple) method does not always work; another one comes afterward

Example: Changing Factorial into Tail Recursion

$\textsc{Fact}(\mathsf{n})$:
  if n $= 0$ **then** $r \leftarrow 1$
  else $r \leftarrow n \times fact(n-1)$

$\lambda(n, acc)$:

Step-by-step approach

0. (it works because the addition is commutative and associative)
1. Create a lambda function doing the recursion, with more parameters
   - Local copy of the parameters carrying the recursion
   - Add as many accumulators as operations done on climb up

Example: Changing Factorial into Tail Recursion

FACT(n):
  if n $= 0$ **then** $r \leftarrow 1$
    else $r \leftarrow n \times fact(n-1)$

FACT'(n): $\lambda(n, 1)$
$\lambda(n, acc)$:

Step-by-step approach

0. (it works because the addition is commutative and associative)
1. Create a lambda function doing the recursion, with more parameters
   - Local copy of the parameters carrying the recursion
   - Add as many accumulators as operations done on climb up
2. Main function simply calls the lambda function
   - Copy of the parameters carrying the recursion
   - Initialize accumulators with base case's value (often identity element)

Example: Changing Factorial into Tail Recursion

$\text{FACT}(n)$:
  if n $= 0$ **then** $r \leftarrow 1$
    else $r \leftarrow n \times fact(n-1)$

$\text{FACT'}(n)$: $\lambda(n, 1)$
$\lambda(n, acc)$:
  if n $= 0$
    else $r \leftarrow \lambda(n-1, acc \times n)$

Step-by-step approach

0. (it works because the addition is commutative and associative)
1. Create a lambda function doing the recursion, with more parameters
   - Local copy of the parameters carrying the recursion
   - Add as many accumulators as operations done on climb up
2. Main function simply calls the lambda function
   - Copy of the parameters carrying the recursion
   - Initialize accumulators with base case's value (often identity element)
3. Body of the lambda function:
   - General treatment: as before, but do intermediate ops into the accumulators

Example: Changing Factorial into Tail Recursion

$\textsc{Fact}(\mathrm{n})$:
  if $\mathrm{n} = 0$ **then** $r \leftarrow 1$
  else $r \leftarrow n \times fact(n-1)$

$\textsc{Fact'}(\mathrm{n})$: $\lambda(n, 1)$
$\lambda(n, acc)$:
  if $\mathrm{n} = 0$ **then** $r \leftarrow acc$
  else $r \leftarrow \lambda(n-1, acc \times n)$

Step-by-step approach

0. (it works because the addition is commutative and associative)
1. Create a lambda function doing the recursion, with more parameters
   - Local copy of the parameters carrying the recursion
   - Add as many accumulators as operations done on climb up
2. Main function simply calls the lambda function
   - Copy of the parameters carrying the recursion
   - Initialize accumulators with base case's value (often identity element)
3. Body of the lambda function:
   - General treatment: as before, but do intermediate ops into the accumulators
   - Base case: get result directly from the accumulators

Let's take another example

Computation of a string's length

```
len(str):
  if empty(s) then 0
              else  1+len(cdr(s))
```

```
λ(str,acc):
```

0. (works because addition is commutative and associative)
1. Create a λ function doing the recursion, adding one accumulator per operation

Let's take another example

Computation of a string's length

len(str):
  if empty(s) **then** 0
  else 1+len(cdr(s))

len'(str) = λ(str,0)

λ(str,acc):

0. (works because addition is commutative and associative)
1. Create a λ function doing the recursion, adding one accumulator per operation
2. Main function: calls the lambda function and initializes the parameters

Let's take another example

Computation of a string's length

```
len(str):
  if empty(s) then 0
              else  1+len(cdr(s))
```

```
len'(str) = λ(str,0)

λ(str,acc):
  if empty(str)
                  else  λ(cdr(str), acc+1)
```

0. (works because addition is commutative and associative)
1. Create a λ function doing the recursion, adding one accumulator per operation
2. Main function: calls the lambda function and initializes the parameters
3. Body of the λ function:
   - General treatment: as before, but do intermediate ops into the accumulator(s)

Let's take another example

Computation of a string's length

```
len(str):
  if empty(s) then 0
              else  1+len(cdr(s))
```

```
len'(str) = λ(str,0)

λ(str,acc):
  if empty(str) then acc
                 else  λ(cdr(str), acc+1)
```

0. (works because addition is commutative and associative)
1. Create a λ function doing the recursion, adding one accumulator per operation
2. Main function: calls the lambda function and initializes the parameters
3. Body of the λ function:
   - General treatment: as before, but do intermediate ops into the accumulator(s)
   - Base case: get result directly from the accumulator(s)

Closing the loop: Non-recursive form of Factorial

$\textsc{Fact}(\mathsf{n})$:
  if $\mathsf{n} = 0$ **then** $r \leftarrow 1$
  else $r \leftarrow n \times fact(n-1)$

$\textsc{Fact'}(\mathsf{n})$: $\lambda(n, 1)$
$\lambda(n, acc)$:
  if $\mathsf{n} = 0$ **then** $r \leftarrow acc$
  else $r \leftarrow \lambda(n-1, acc \times n)$

- This function uses Tail Recursion

$\rightsquigarrow$ We can turn the helper into non-recursion with the method seen before

Closing the loop: Non-recursive form of Factorial

$\textsc{Fact}(\mathsf{n})$:
  if $\mathsf{n} = 0$ **then** $r \leftarrow 1$
  else $r \leftarrow n \times fact(n-1)$

$\textsc{Fact}$'(n): $\lambda(n, 1)$
$\lambda(n, acc)$:
  if $\mathsf{n} = 0$ **then** $r \leftarrow acc$
  else $r \leftarrow \lambda(n-1, acc \times n)$

- This function uses Tail Recursion

$\leadsto$ We can turn the helper into non-recursion with the method seen before

$\lambda'(n, acc)$:
  $td \leftarrow n$; $a \leftarrow acc$
  until $td = 0$ **do**
    $a \leftarrow a \times td$ // beware of the
    $td \leftarrow td - 1$ // updates' order
  end
  return a

Closing the loop: Non-recursive form of Factorial

FACT(n):
  if n $= 0$ **then** $r \leftarrow 1$
          else $r \leftarrow n \times fact(n-1)$

FACT'(n): $\lambda(n, 1)$
$\lambda(n, acc)$:
  if n $= 0$ **then** $r \leftarrow acc$
          else $r \leftarrow \lambda(n-1, acc \times n)$

- This function uses Tail Recursion

$\rightsquigarrow$ We can turn the helper into non-recursion with the method seen before

- Then, we combine everything

$\lambda'(n, acc)$:
  $td \leftarrow n$; $a \leftarrow acc$
  until $td = 0$ **do**
    $a \leftarrow a \times td$ // beware of the
    $td \leftarrow td - 1$ // updates' order
  end
  return a

FACT"(n):
  $td \leftarrow n$; $a \leftarrow 1$
  until $td = 0$ **do**
    $a \leftarrow a \times td$
    $td \leftarrow td - 1$
  end
  return a

- These two transformations are simple, automatic and neat...

Closing the loop: Non-recursive form of Factorial

FACT(n):
  if n = 0 **then** $r \leftarrow 1$
        else $r \leftarrow n \times fact(n-1)$

FACT'(n): $\lambda(n, 1)$
$\lambda(n, acc)$:
  if n = 0 **then** $r \leftarrow acc$
        else $r \leftarrow \lambda(n-1, acc \times n)$

- This function uses Tail Recursion

$\rightsquigarrow$ We can turn the helper into non-recursion with the method seen before

- Then, we combine everything

$\lambda'(n, acc)$:
  $td \leftarrow n$; $a \leftarrow acc$
  until $td = 0$ **do**
    $a \leftarrow a \times td$ // beware of the
    $td \leftarrow td - 1$ // updates' order
  end
  return a

FACT"(n):
  $td \leftarrow n$; $a \leftarrow 1$
  until $td = 0$ **do**
    $a \leftarrow a \times td$
    $td \leftarrow td - 1$
  end
  return a

- These two transformations are simple, automatic and neat...
- ...when applicable!!! ☹ If not, let's get angry and mean!

Generic Algorithm Using a Stack

Idea

- Processors are sequential and execute any recursive function
  $\Rightarrow$ *Always possible to express without recursion*
- Principle: simulating the function stack of processors
  By using a stack explicitly

Generic Algorithm Using a Stack

Idea

- Processors are sequential and execute any recursive function
  $\Rightarrow$ *Always possible to express without recursion*
- Principle: simulating the function stack of processors
  By using a stack explicitly

Example with only one recursive call

$$\boxed{\begin{array}{l} \textbf{if } \mathrm{cond}(\mathrm{x}) \textbf{ then } r \leftarrow g(x) \\ \qquad\qquad\quad \textbf{else } \mathrm{T}(\mathrm{x});\ r \leftarrow G(x, f(x_{int})) \end{array}}$$

Generic Algorithm Using a Stack

Idea

- Processors are sequential and execute any recursive function
  $\Rightarrow$ *Always possible to express without recursion*
- Principle: simulating the function stack of processors
  By using a stack explicitly

Example with only one recursive call

if cond(x) **then** $r \leftarrow g(x)$
else T(x); $r \leftarrow G(x, f(x_{int}))$

$p \leftarrow emptyStack$
$a \leftarrow x$ (* a: locale variable *)
(* pushing on stack (descent) *)
until cond(a) **do**
  push(p, a)
  $a \leftarrow h(a)$
end
$r \leftarrow g(a)$ (* Base Case *)
(* poping from stack (climb up) *)
until stackIsEmpty(p) **do**
  $a \leftarrow top(p)$; pop(p); T(a)
  $r \leftarrow G(a, r)$
end

Generic Algorithm Using a Stack

Idea

- Processors are sequential and execute any recursive function
  $\Rightarrow$ *Always possible to express without recursion*
- Principle: simulating the function stack of processors
  By using a stack explicitly

Example with only one recursive call

if cond(x) **then** $r \leftarrow g(x)$
else T(x); $r \leftarrow G(x, f(x_{int}))$

Remark:
If $h()$ is invertible, no need for a stack:
parameter reconstructed by $h^{-1}()$

Stopping Condition = counting calls

```
p ← emptyStack
a ← x (* a: locale variable *)
(* pushing on stack (descent) *)
until cond(a) do
  push(p, a)
  a ← h(a)
end
r ← g(a) (* Base Case *)
(* poping from stack (climb up) *)
until stackIsEmpty(p) do
  a ← top(p); pop(p); T(a)
  r ← G(a, r)
end
```

Non-Recursive Form of Hanoï Towers (1/2)

```
HANOI(n,a,b,c): (a to b, with c as disposal)
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

One should mimic processor behavior wrt stacking

- H(4,a,b,c) = H(3,a,c,b)+D(a,b)+H(3,c,b,a)

We get:

$$\text{H(4,a,b,c)} = \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{H(3,a,c,b)}} + \text{D(a,b)} + \text{H(3,c,b,a)}$$

Non-Recursive Form of Hanoï Towers (1/2)

```
HANOI(n,a,b,c): (a to b, with c as disposal)
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

One should mimic processor behavior wrt stacking

- H(4,a,b,c) = H(3,a,c,b)+D(a,b)+H(3,c,b,a)
- Compute first unknown term: H(3,a,c,b) = H(2,a,b,c)+D(a,c)+H(2,b,c,a)

We get:

$$\text{H(4,a,b,c)} = \underbrace{\underbrace{\qquad\qquad\qquad}_{\text{H(2,a,b,c)}} + \text{D(a,c)} + \text{H(2,b,c,a)}}_{\text{H(3,a,c,b)}} + \text{D(a,b)} + \text{H(3,c,b,a)}$$

Non-Recursive Form of Hanoï Towers (1/2)

HANOI(n,a,b,c): *(a to b, with c as disposal)*
  if $n > 0$ **then** hanoi(n-1, a, c)
                  move(a, b)
                  hanoi(n-1, c, b)

One should mimic processor behavior wrt stacking

- H(4,a,b,c) = H(3,a,c,b)+D(a,b)+H(3,c,b,a)
- Compute first unknown term: H(3,a,c,b) = H(2,a,b,c)+D(a,c)+H(2,b,c,a)
- Compute first unknown term: H(2,a,b,c) = H(1,a,c,b)+D(a,b)+H(1,c,b,a)

We get:

$$H(4,a,b,c) = \underbrace{\underbrace{\quad + D(a,b) + \quad}_{H(2,a,b,c)} + D(a,c) + H(2,b,c,a)}_{H(3,a,c,b)} + D(a,b) + H(3,c,b,a)$$

Non-Recursive Form of Hanoï Towers (1/2)

```
HANOI(n,a,b,c): (a to b, with c as disposal)
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

One should mimic processor behavior wrt stacking

- H(4,a,b,c) = H(3,a,c,b)+D(a,b)+H(3,c,b,a)
- Compute first unknown term: H(3,a,c,b) = H(2,a,b,c)+D(a,c)+H(2,b,c,a)
- Compute first unknown term: H(2,a,b,c) = H(1,a,c,b)+D(a,b)+H(1,c,b,a)
- Compute first unknown term: H(1,a,c,b) = D(a,c)

We get:

$$\text{H(4,a,b,c)} = \underbrace{\underbrace{\text{D(a,c)+D(a,b)+}\qquad}_{\text{H(2,a,b,c)}}\text{+D(a,c)+H(2,b,c,a)}}_{\text{H(3,a,c,b)}}\text{+D(a,b)+H(3,c,b,a)}$$

Non-Recursive Form of Hanoï Towers (1/2)

```
HANOI(n,a,b,c): (a to b, with c as disposal)
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

One should mimic processor behavior wrt stacking

- H(4,a,b,c) = H(3,a,c,b)+D(a,b)+H(3,c,b,a)
- Compute first unknown term: H(3,a,c,b) = H(2,a,b,c)+D(a,c)+H(2,b,c,a)
- Compute first unknown term: H(2,a,b,c) = H(1,a,c,b)+D(a,b)+H(1,c,b,a)
- Compute first unknown term: H(1,a,c,b) = D(a,c)
- Take on something casted aside: H(1,c,b,a) = D(c,b)

We get:

$$H(4,a,b,c) = \underbrace{\underbrace{D(a,c)+D(a,b)+D(c,b)}_{H(2,a,b,c)}+D(a,c)+H(2,b,c,a)}_{H(3,a,c,b)}+D(a,b)+H(3,c,b,a)$$

Non-Recursive Form of Hanoï Towers (1/2)

HANOI(n,a,b,c): *(a to b, with c as disposal)*
  if $n > 0$ **then** hanoi(n-1, a, c)
                    move(a, b)
                    hanoi(n-1, c, b)

One should mimic processor behavior wrt stacking

- H(4,a,b,c) = H(3,a,c,b)+D(a,b)+H(3,c,b,a)
- Compute first unknown term: H(3,a,c,b) = H(2,a,b,c)+D(a,c)+H(2,b,c,a)
- Compute first unknown term: H(2,a,b,c) = H(1,a,c,b)+D(a,b)+H(1,c,b,a)
- Compute first unknown term: H(1,a,c,b) = D(a,c)
- Take on something casted aside: H(1,c,b,a) = D(c,b)
- and so on until everything casted aside is done (until stack is empty)

We get:

$$\text{H(4,a,b,c)} = \underbrace{\underbrace{\text{D(a,c)+D(a,b)+D(c,b)}}_{\text{H(2,a,b,c)}}\text{+D(a,c)+H(2,b,c,a)}}_{\text{H(3,a,c,b)}}\text{+D(a,b)+H(3,c,b,a)}$$

Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :

```
push (n, A, B, 1) on stack
while (stack non-empty)
   (n, A, B, CallKind) ← pop()
   if (n > 0)
      if (CallKind == 1)
         push (n, A, B, 2) on stack (* Cast something aside for later *)
         push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
         move(A, B)
         push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :

```
push (n, A, B, 1) on stack
while (stack non-empty)
   (n, A, B, CallKind) ← pop()
   if (n > 0)
      if (CallKind == 1)
         push (n, A, B, 2) on stack (* Cast something aside for later *)
         push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
         move(A, B)
         push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

4AB1

Step 1 Step 2 Step 3 Step 4 Step 5 Step 6 Step 8 Step 9 Step 11 Step 13

hanoi(4,a,b)=. . .

Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :

```
push (n, A, B, 1) on stack
while (stack non-empty)
   (n, A, B, CallKind) ← pop()
   if (n > 0)
      if (CallKind == 1)
         push (n, A, B, 2) on stack (* Cast something aside for later *)
         push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
         move(A, B)
         push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

	3AC1								
4AB1	4AB2								
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 8	Step 9	Step 11	Step 13

hanoi(4,a,b)=...

Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :

```
push (n, A, B, 1) on stack
while (stack non-empty)
   (n, A, B, CallKind) ← pop()
   if (n > 0)
      if (CallKind == 1)
         push (n, A, B, 2) on stack (* Cast something aside for later *)
         push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
         move(A, B)
         push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

		2AB1							
	3AC1	3AC2							
4AB1	4AB2	4AB2							
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 8	Step 9	Step 11	Step 13

hanoi(4,a,b)=...

Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :

```
push (n, A, B, 1) on stack
while (stack non-empty)
   (n, A, B, CallKind) ← pop()
   if (n > 0)
      if (CallKind == 1)
         push (n, A, B, 2) on stack (* Cast something aside for later *)
         push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
         move(A, B)
         push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

			1AC1						
		2AB1	2AB2						
	3AC1	3AC2	3AC2						
4AB1	4AB2	4AB2	4AB2						
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 8	Step 9	Step 11	Step 13

hanoi(4,a,b)=. . .

Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :

```
push (n, A, B, 1) on stack
while (stack non-empty)
   (n, A, B, CallKind) ← pop()
   if (n > 0)
      if (CallKind == 1)
         push (n, A, B, 2) on stack (* Cast something aside for later *)
         push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
         move(A, B)
         push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

				0AB1					
			1AC1	1AC2					
		2AB1	2AB2	2AB2					
	3AC1	3AC2	3AC2	3AC2					
4AB1	4AB2	4AB2	4AB2	4AB2					
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 8	Step 9	Step 11	Step 13

hanoi(4,a,b)=. . .

Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :

```
push (n, A, B, 1) on stack
while (stack non-empty)
   (n, A, B, CallKind) ← pop()
   if (n > 0)
      if (CallKind == 1)
         push (n, A, B, 2) on stack (* Cast something aside for later *)
         push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
         move(A, B)
         push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

				0AB1					
			1AC1	1AC2	0BC1				
		2AB1	2AB2	2AB2	2AB2				
	3AC1	3AC2	3AC2	3AC2	3AC2				
4AB1	4AB2	4AB2	4AB2	4AB2	4AB2				
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 8	Step 9	Step 11	Step 13

hanoi(4,a,b)=D(ac)+...

Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :

```
push (n, A, B, 1) on stack
while (stack non-empty)
   (n, A, B, CallKind) ← pop()
   if (n > 0)
      if (CallKind == 1)
         push (n, A, B, 2) on stack (* Cast something aside for later *)
         push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
         move(A, B)
         push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

				0AB1					
			1AC1	1AC2	0BC1				
		2AB1	2AB2	2AB2	2AB2	1CB1			
	3AC1	3AC2	3AC2	3AC2	3AC2	3AC2			
4AB1	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2			
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 8	Step 9	Step 11	Step 13

hanoi(4,a,b)=D(ac)+D(ab)+...

Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :

```
push (n, A, B, 1) on stack
while (stack non-empty)
   (n, A, B, CallKind) ← pop()
   if (n > 0)
      if (CallKind == 1)
         push (n, A, B, 2) on stack (* Cast something aside for later *)
         push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
         move(A, B)
         push (n-1, C, B, 1) on stack
```

HANOI(n,a,b):
if $n > 0$ **then** hanoi(n-1, a, c)
move(a, b)
hanoi(n-1, c, b)

				0AB1					
			1AC1	1AC2	0BC1		0AB1		
		2AB1	2AB2	2AB2	2AB2	1CB1	1CB2		
	3AC1	3AC2	3AC2	3AC2	3AC2	3AC2	3AC2		
4AB1	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2		
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 8	Step 9	Step 11	Step 13

hanoi(4,a,b)=D(ac)+D(ab)+...

Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :

```
push (n, A, B, 1) on stack
while (stack non-empty)
   (n, A, B, CallKind) ← pop()
   if (n > 0)
      if (CallKind == 1)
         push (n, A, B, 2) on stack (* Cast something aside for later *)
         push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
         move(A, B)
         push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

				0AB1					
			1AC1	1AC2	0BC1		0AB1		
		2AB1	2AB2	2AB2	2AB2	1CB1	1CB2	0AB1	
	3AC1	3AC2	3AC2	3AC2	3AC2	3AC2	3AC2	3AC2	
4AB1	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2	
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 8	Step 9	Step 11	Step 13

hanoi(4,a,b)=D(ac)+D(ab)+D(cb)+...

Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :

```
push (n, A, B, 1) on stack
while (stack non-empty)
   (n, A, B, CallKind) ← pop()
   if (n > 0)
      if (CallKind == 1)
         push (n, A, B, 2) on stack (* Cast something aside for later *)
         push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
         move(A, B)
         push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

				0AB1					
			1AC1	1AC2	0BC1		0AB1		
		2AB1	2AB2	2AB2	2AB2	1CB1	1CB2	0AB1	
	3AC1	3AC2	3AC2	3AC2	3AC2	3AC2	3AC2	3AC2	2BC1
4AB1	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 8	Step 9	Step 11	Step 13

hanoi(4,a,b)=D(ac)+D(ab)+D(cb)+D(ac)+...

Non-Recursive Form of Hanoï Towers (2/2)

hanoi_derec(n, A, B) :
```
push (n, A, B, 1) on stack
while (stack non-empty)
   (n, A, B, CallKind) ← pop()
   if (n > 0)
      if (CallKind == 1)
         push (n, A, B, 2) on stack (* Cast something aside for later *)
         push (n-1, A, C, 1) (* Compute first unknown soon *)
      else /* ie, CallKind == 2 */
         move(A, B)
         push (n-1, C, B, 1) on stack
```

```
HANOI(n,a,b):
  if n > 0 then hanoi(n-1, a, c)
                move(a, b)
                hanoi(n-1, c, b)
```

				0AB1					
			1AC1	1AC2	0BC1		0AB1		
		2AB1	2AB2	2AB2	2AB2	1CB1	1CB2	0AB1	
	3AC1	3AC2	3AC2	3AC2	3AC2	3AC2	3AC2	3AC2	2BC1
4AB1	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2	4AB2
Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 8	Step 9	Step 11	Step 13

hanoi(4,a,b)=D(ac)+D(ab)+D(cb)+D(ac)+. . .

Rq: simpler iterative algorithms exist (they are not automatic transformations)

J.C. Fournier. *Pour en finir avec la dérécursivation du problème des tours de Hanoï.* 1990.
http://archive.numdam.org/article/ITA_1990__24_1_17_0.pdf