# INF421, Lecture 6
# Recursion

Leo Liberti

LIX, École Polytechnique, France

# Course

- **Objective**: teach notions AND develop intelligence
- **Evaluation**: TP noté en salle info, Contrôle à la fin. Note: $\max(CC, \frac{3}{4}CC + \frac{1}{4}TP)$
- **Organization**: fri 31/8, 7/9, 14/9, 21/9, 28/9, 5/10, 12/10, 19/10, 26/10, amphi 1030-12 (Arago), TD 1330-1530, 1545-1745 (SI:30-34)
- **Books**:
  1. K. Mehlhorn & P. Sanders, *Algorithms and Data Structures*, Springer, 2008
  2. D. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997
  3. G. Dowek, *Les principes des langages de programmation*, Editions de l'X, 2008
  4. Ph. Baptiste & L. Maranget, *Programmation et Algorithmique*, Ecole Polytechnique (Polycopié), 2006
- **Website**: `www.enseignement.polytechnique.fr/informatique/INF421`
- **Blog**: `inf421.wordpress.com`
- **Contact**: `liberti@lix.polytechnique.fr` (e-mail subject: INF421)

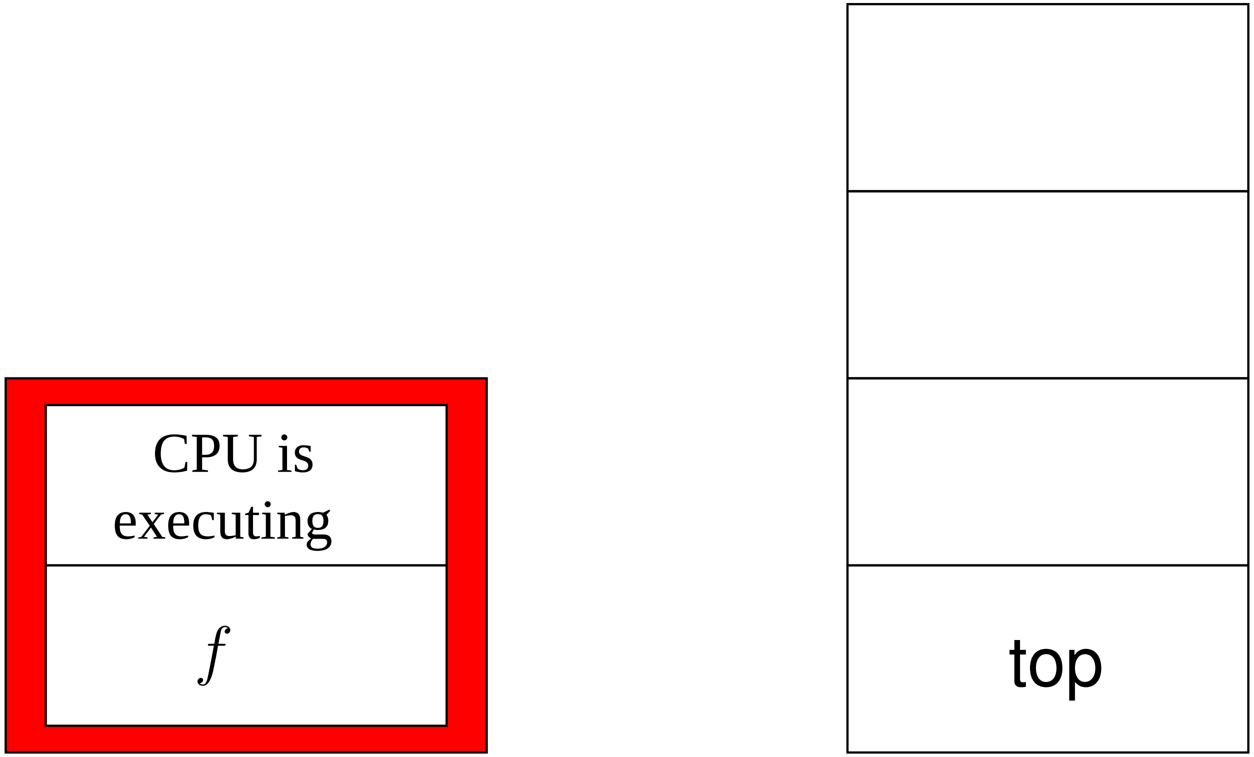
# Lecture summary

- Stacks
- Recursion

# Motivating example

# How functions are called

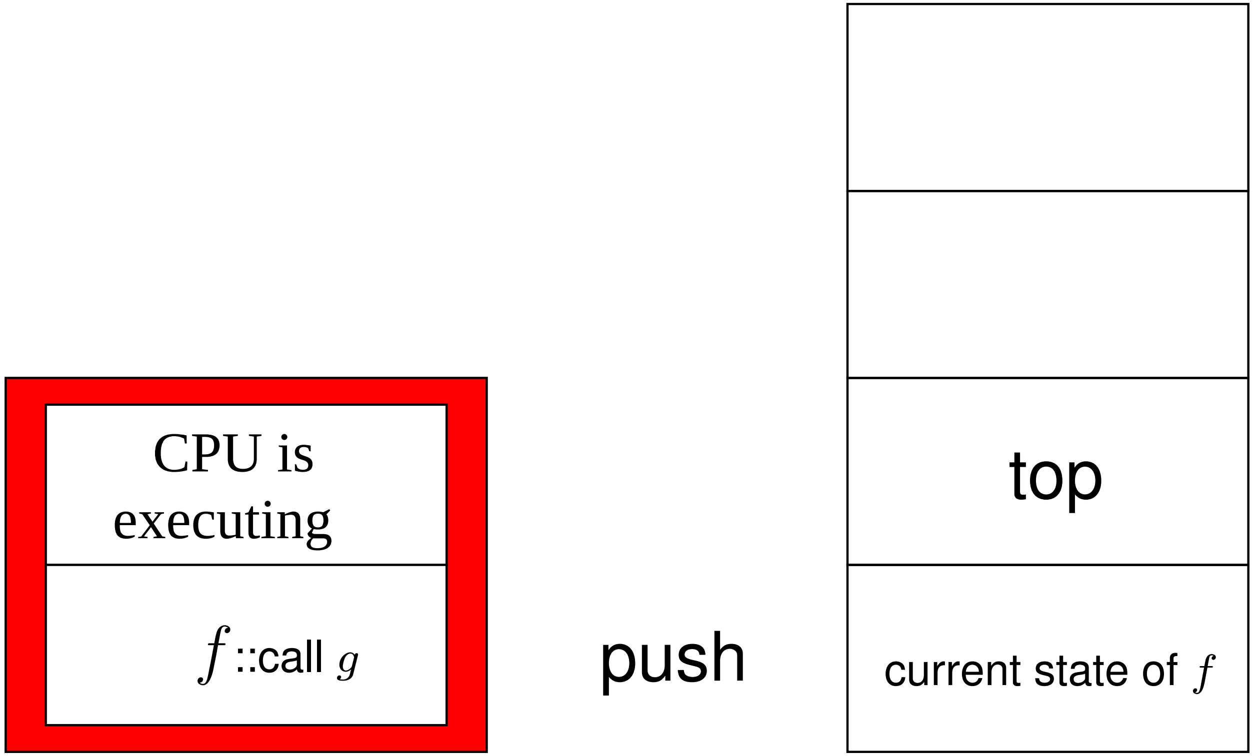
$f$ calls $g$ calls $h$



Memory

# How functions are called

$f$ calls $g$ calls $h$



CPU is executing

$f$::call $g$

push

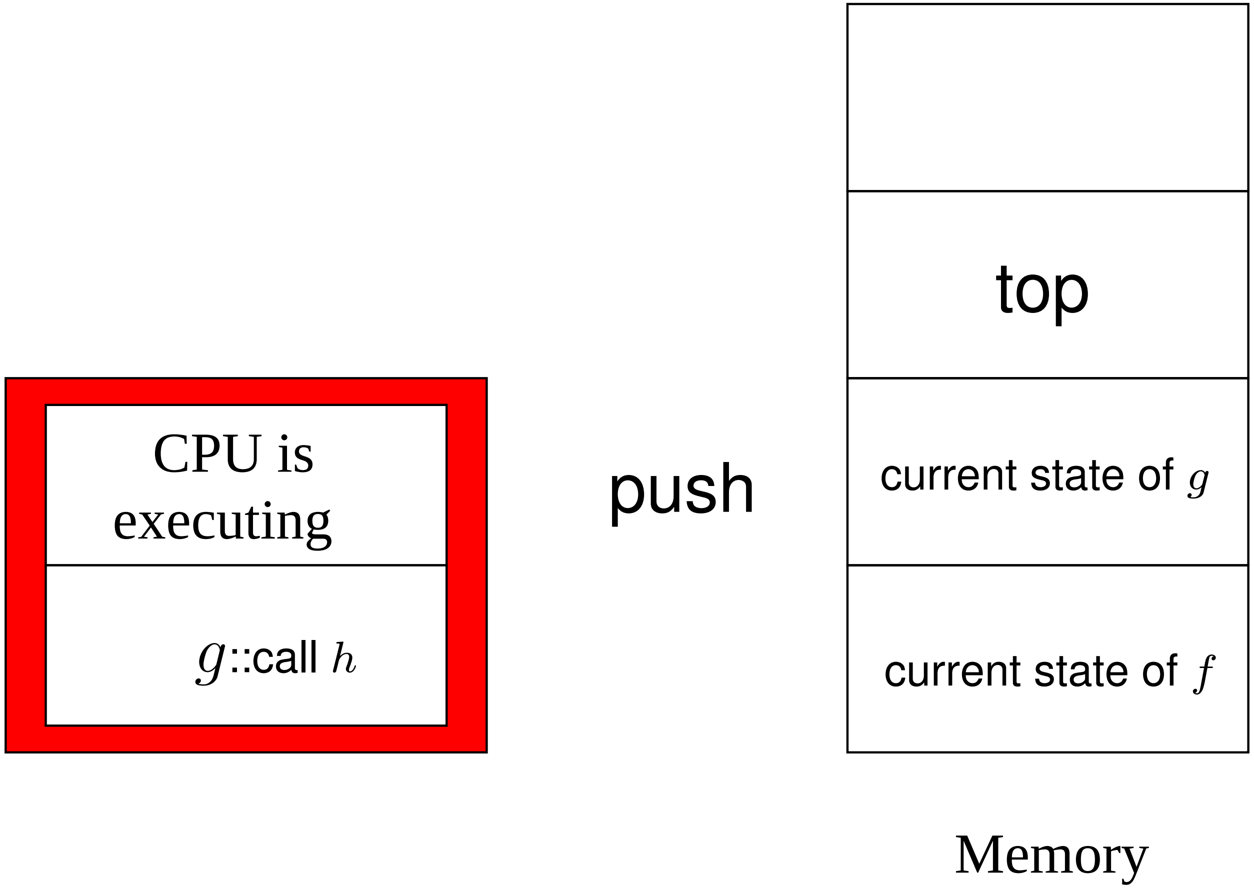
top

current state of $f$

Memory

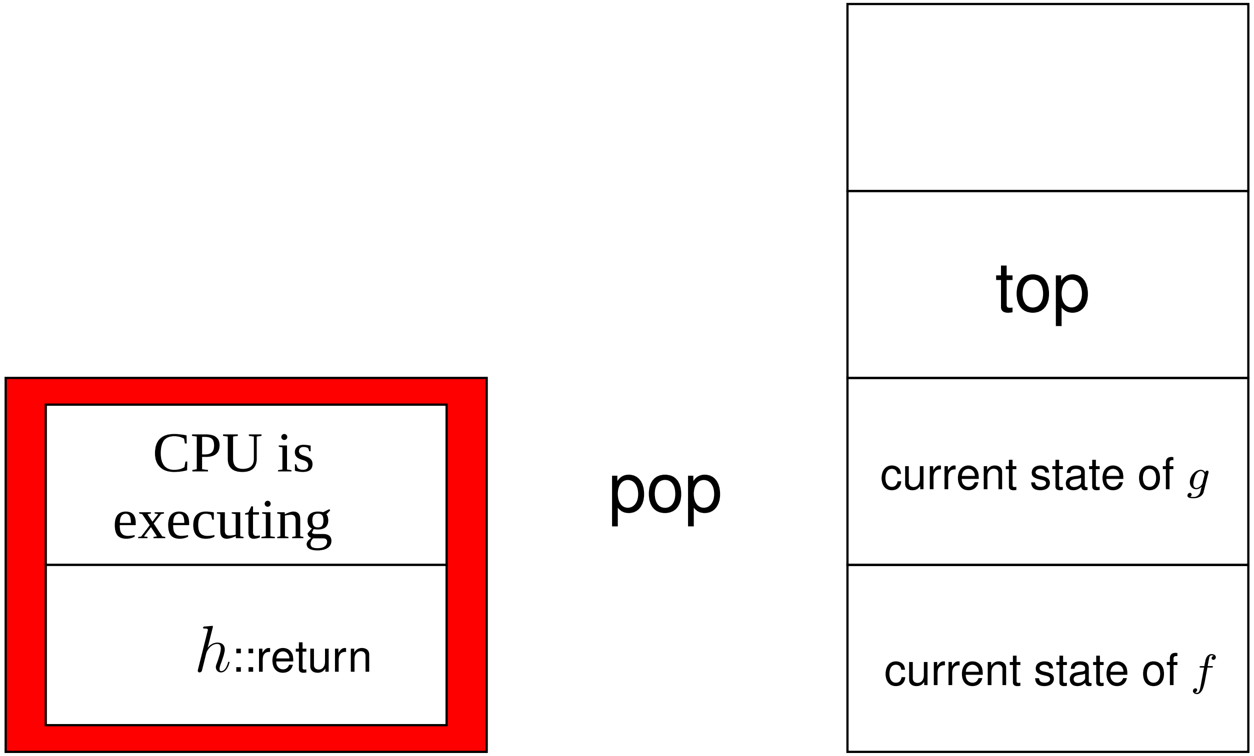
# How functions are called

$f$ calls $g$ calls $h$



# How functions are called

## $f$ calls $g$ calls $h$



CPU is executing

$h$::return

pop

top

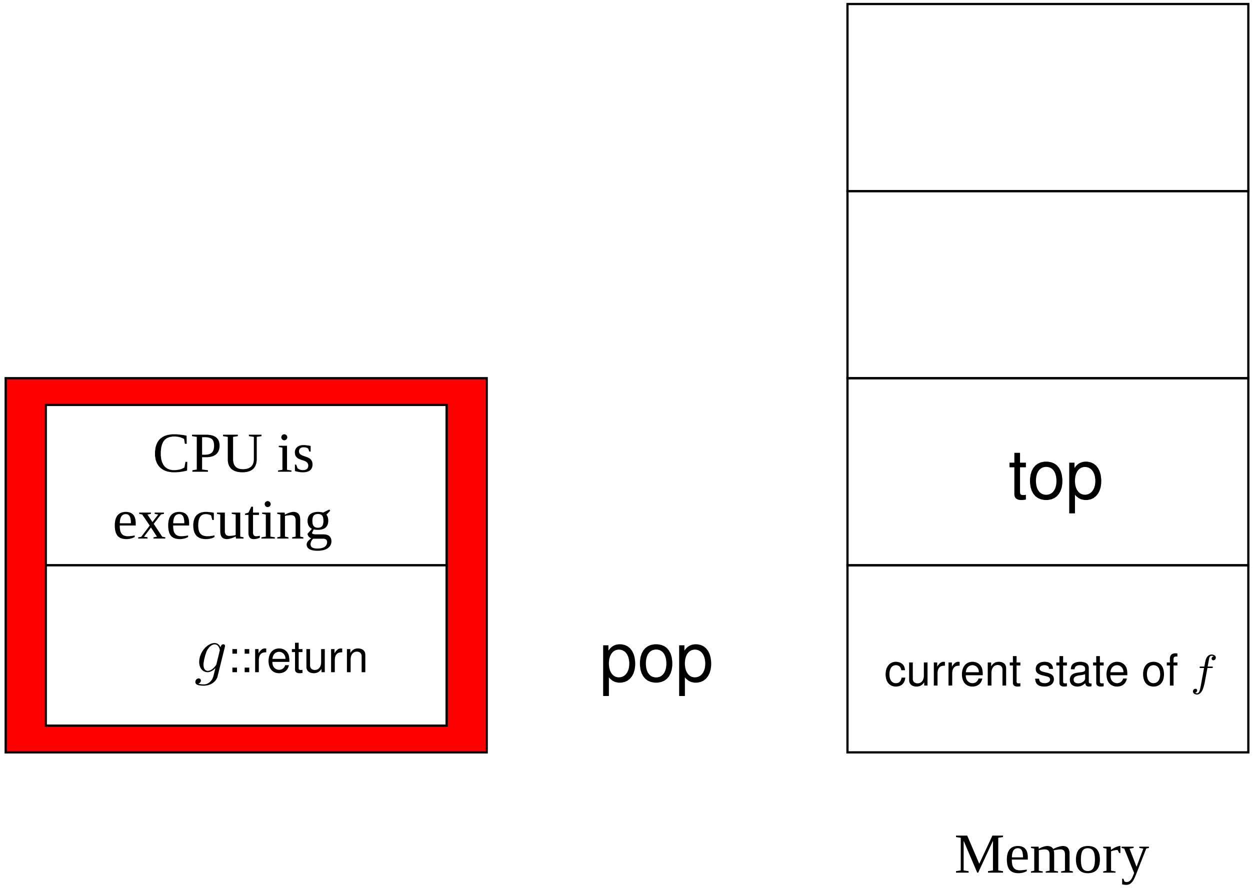
current state of $g$

current state of $f$

Memory

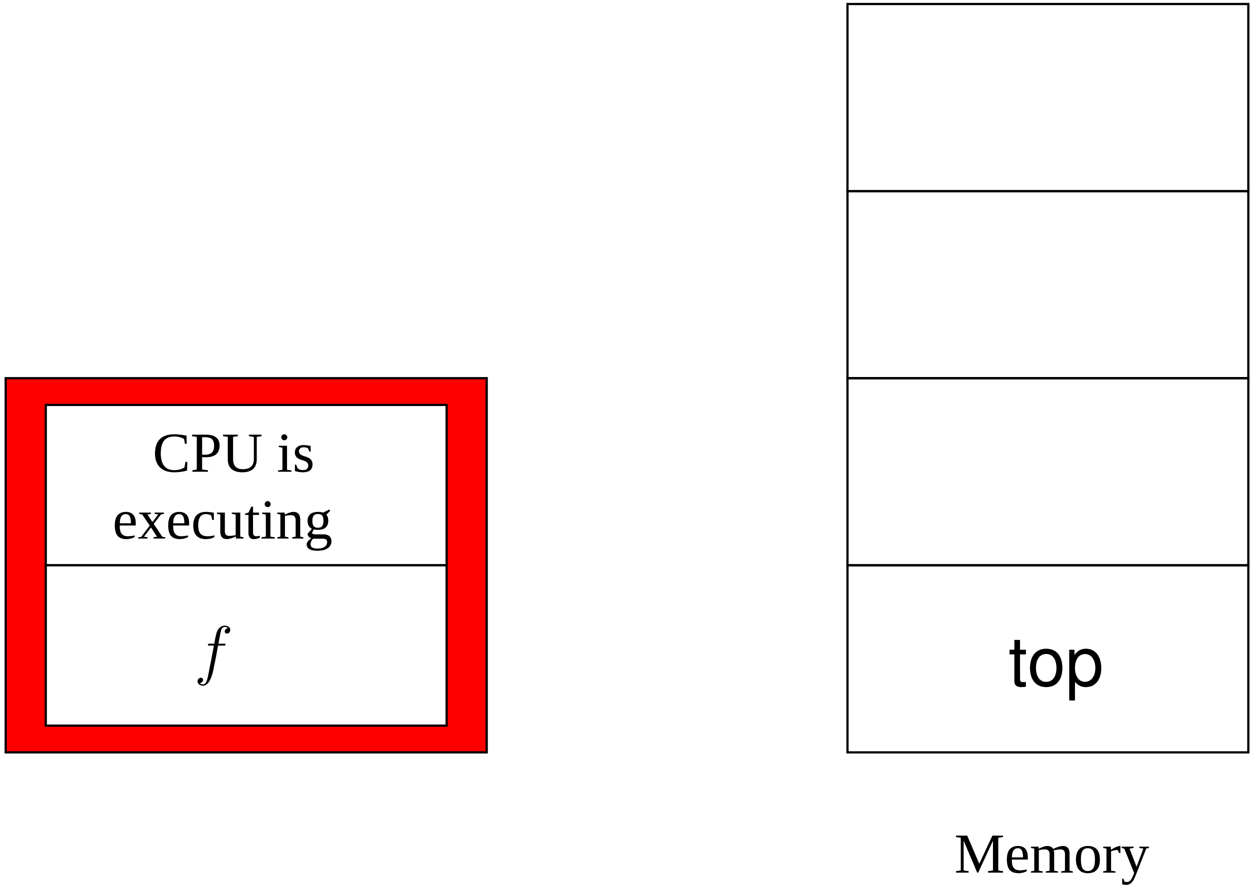
# How functions are called

$f$ calls $g$ calls $h$



# How functions are called

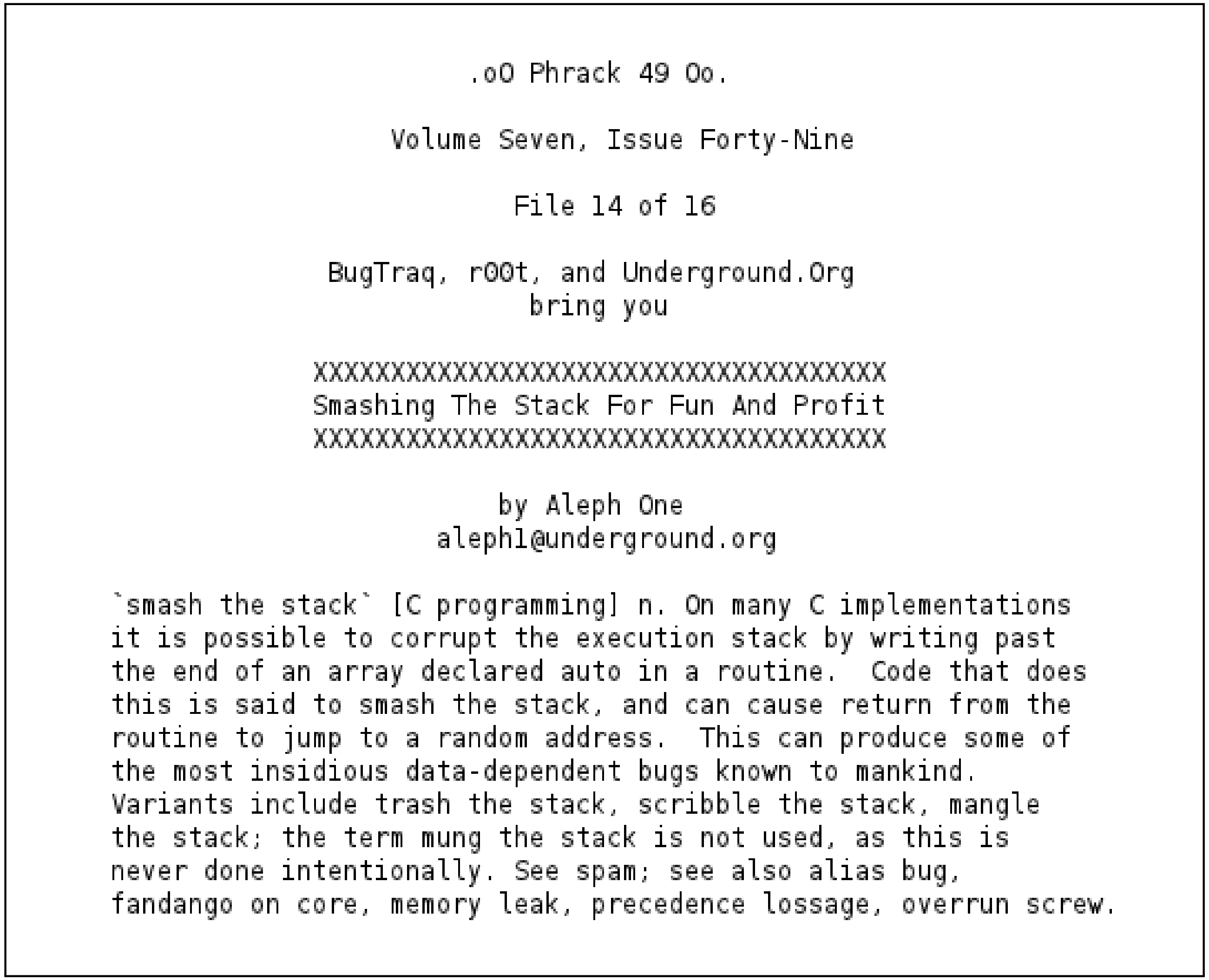
$f$ calls $g$ calls $h$



# Stacks

- Linear data structure
- Accessible from only one end (top)
- Operations:
  - push data on the top of the stack
  - pop data from the top of the stack
  - test whether stack is empty
- Every operation is $O(1)$
- Implement using arrays or lists

# Hack the stack



.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

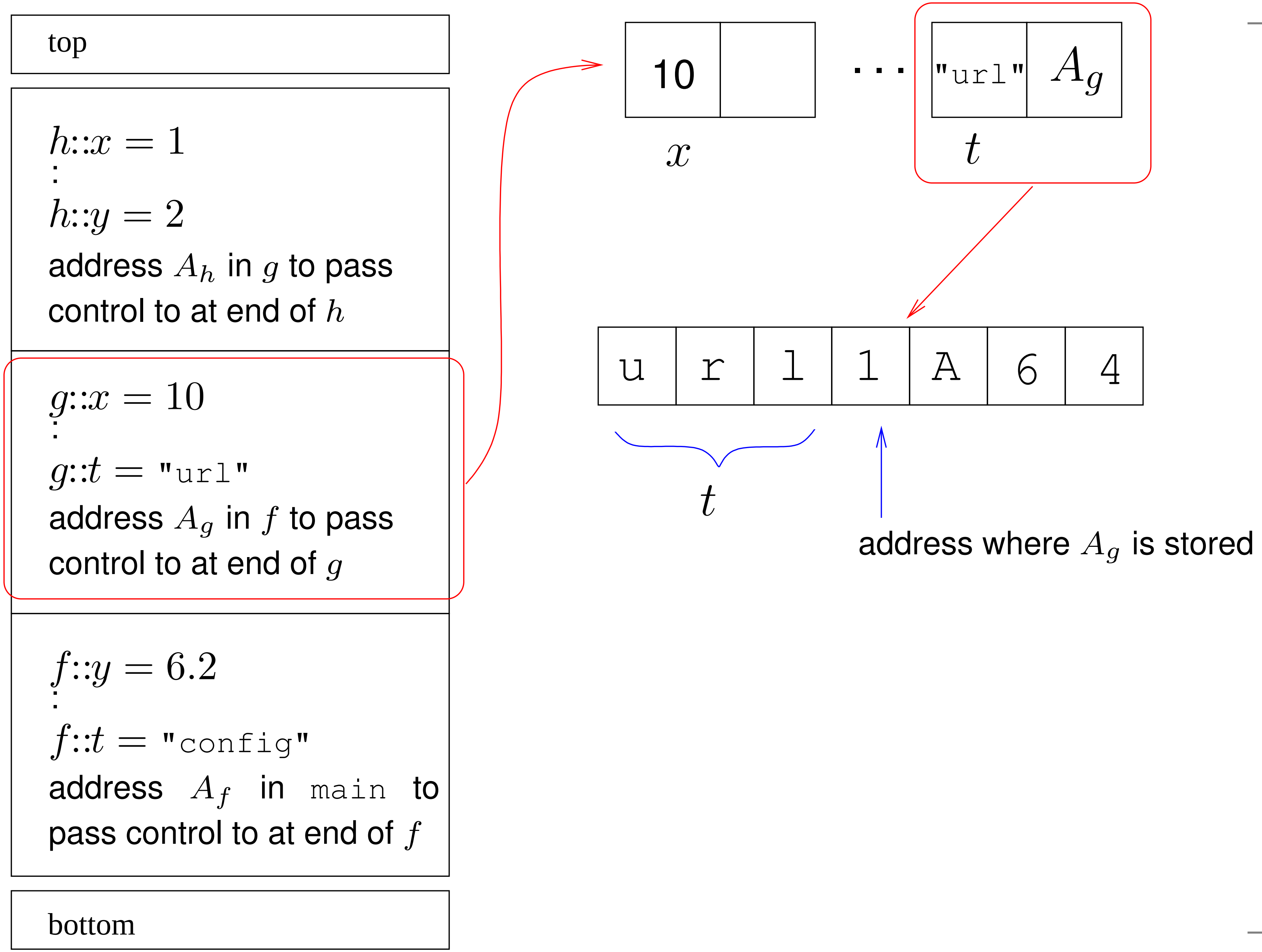
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org

\`smash the stack\` [C programming] n. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung the stack is not used, as this is never done intentionally. See spam; see also alias bug, fandango on core, memory leak, precedence lossage, overrun screw.

Back in 1996, hackers would get into systems by writing disguised code in the execution stack

# How does it work?



top

$h::x = 1$

$\vdots$

$h::y = 2$

address $A_h$ in $g$ to pass control to at end of $h$

$g::x = 10$

$\vdots$

$g::t =$ `"url"`

address $A_g$ in $f$ to pass control to at end of $g$

$f::y = 6.2$

$\vdots$

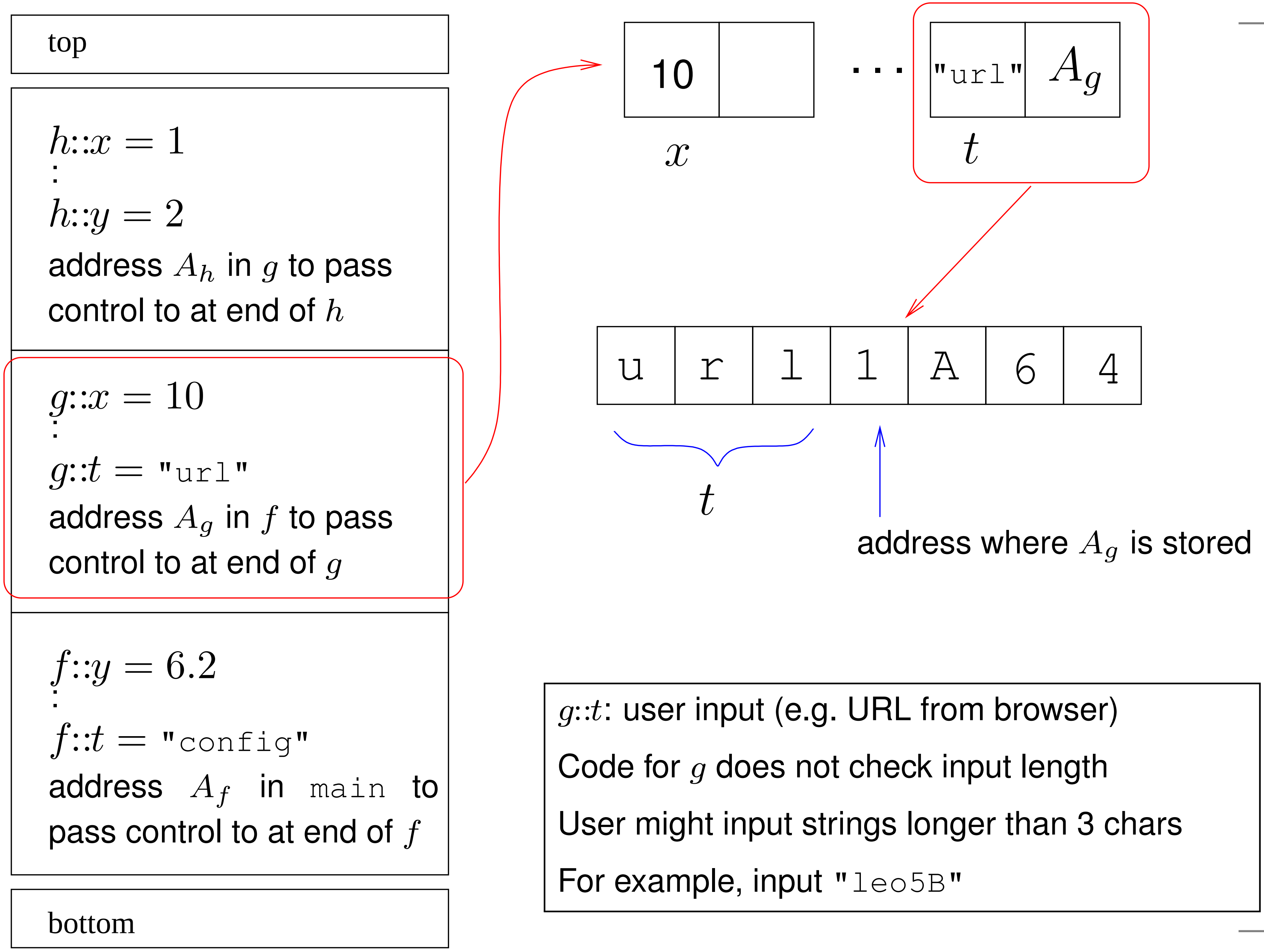
$f::t =$ `"config"`

address $A_f$ in `main` to pass control to at end of $f$

bottom

address where $A_g$ is stored

# How does it work?



top

$h::x = 1$

$\vdots$

$h::y = 2$

address $A_h$ in $g$ to pass control to at end of $h$

$g::x = 10$

$\vdots$

$g::t =$ `"url"`

address $A_g$ in $f$ to pass control to at end of $g$

$f::y = 6.2$

$\vdots$

$f::t =$ `"config"`

address $A_f$ in `main` to pass control to at end of $f$

bottom

| 10 |  |
|---|---|

$x$

$\cdots$

| `"url"` | $A_g$ |
|---|---|

$t$

| u | r | l | 1 | A | 6 | 4 |
|---|---|---|---|---|---|---|

$t$

address where $A_g$ is stored

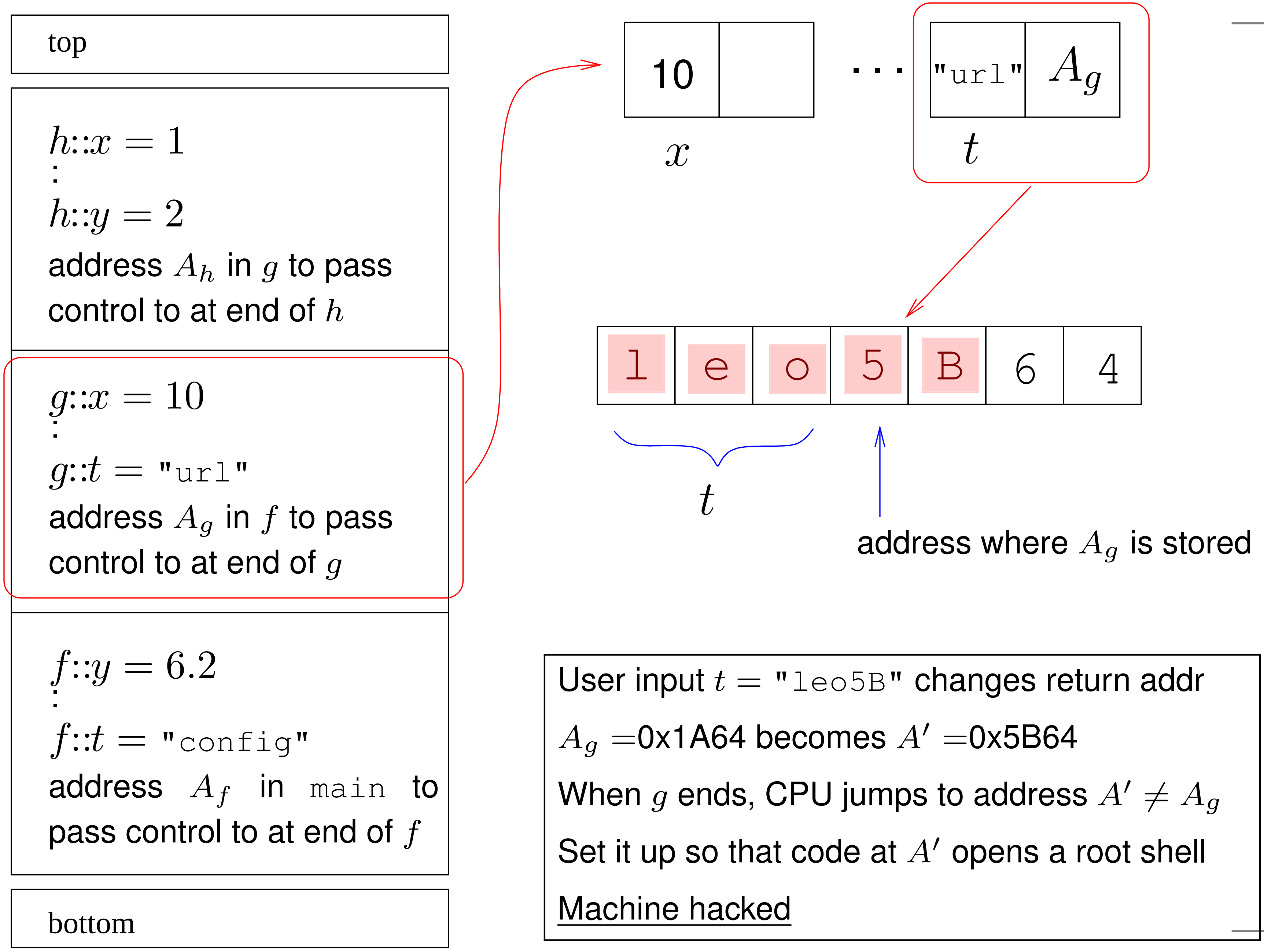
$g::t$: user input (e.g. URL from browser)

Code for $g$ does not check input length

User might input strings longer than 3 chars

For example, input `"leo5B"`

# How does it work?



top

$h::x = 1$

$\vdots$

$h::y = 2$

address $A_h$ in $g$ to pass control to at end of $h$

$g::x = 10$

$\vdots$

$g::t =$ `"url"`

address $A_g$ in $f$ to pass control to at end of $g$

$f::y = 6.2$

$\vdots$

$f::t =$ `"config"`

address $A_f$ in `main` to pass control to at end of $f$

bottom

User input $t =$ `"leo5B"` changes return addr

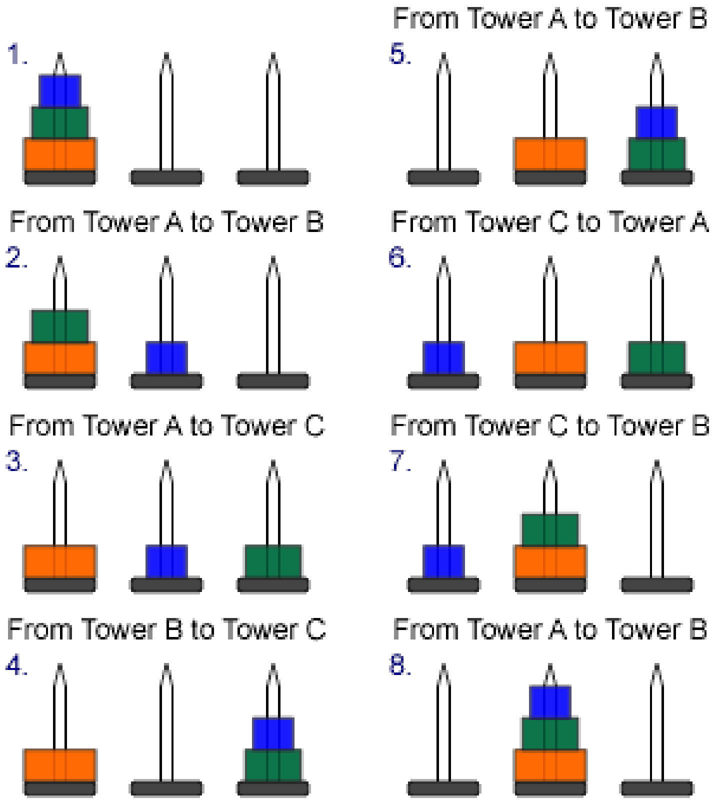
$A_g$ =0x1A64 becomes $A'$ =0x5B64

When $g$ ends, CPU jumps to address $A' \neq A_g$

Set it up so that code at $A'$ opens a root shell

Machine hacked

# The Tower of Hanoi



*Move stack of discs to different pole, one at a time, no larger over smaller*

# Checking brackets

Given a mathematical sentence with two types of brackets "`()`" and "`[]`", write a program that checks whether they have been embedded correctly

$$1+([(x(y-z[\log(n)]/(3-x^2)+\exp(2/[yz]))+1)-2xyz]/2)$$

$$([(([((([(((([1]))))])))])])$$

# Pseudocode

```
input string $s$
for $i \in (1, \ldots, |s|)$ do
  if $s_i =$ '(' or $s_i =$ '[' then
    push ')' or ']' on stack
  else if $s_i =$ ')' or $s_i =$ ']' then
    pop $t$ from stack
    if $t = \emptyset$ (stack is empty) then
      error: (too many closing brackets)
    else if $t \neq s_i$ then
      error: (closing bracket has wrong type)
    end if
  end if
end for
if stack is not empty then
  error: (not enough closing brackets)
end if
```

# Usefulness

Today, stacks are provided by Java/C++ libraries, they are implemented as a subset of operations of lists or vectors. Here are some reasons why you might want to rewrite a stack code

- You’re a student and learning to program

# Usefulness

Today, stacks are provided by Java/C++ libraries, they are implemented as a subset of operations of lists or vectors. Here are some reasons why you might want to rewrite a stack code

- You’re a student and learning to program
- You’re writing an interpreter or a compiler

# Usefulness

Today, stacks are provided by Java/C++ libraries, they are implemented as a subset of operations of lists or vectors. Here are some reasons why you might want to rewrite a stack code

- You're a student and learning to program
- You're writing an interpreter or a compiler
- You're writing an operating system

# Usefulness

Today, stacks are provided by Java/C++ libraries, they are implemented as a subset of operations of lists or vectors. Here are some reasons why you might want to rewrite a stack code

- You’re a student and learning to program
- You’re writing an interpreter or a compiler
- You’re writing an operating system
- You’re writing some graphics code which must execute blighteningly fast and existing libraries are too slow

# Usefulness

Today, stacks are provided by Java/C++ libraries, they are implemented as a subset of operations of lists or vectors. Here are some reasons why you might want to rewrite a stack code

- You’re a student and learning to program
- You’re writing an interpreter or a compiler
- You’re writing an operating system
- You’re writing some graphics code which must execute blighteningly fast and existing libraries are too slow
- You’re a security expert wishing to write an unsmashable stack

# Usefulness

Today, stacks are provided by Java/C++ libraries, they are implemented as a subset of operations of lists or vectors. Here are some reasons why you might want to rewrite a stack code

- You're a student and learning to program
- You're writing an interpreter or a compiler
- You're writing an operating system
- You're writing some graphics code which must execute blighteningly fast and existing libraries are too slow
- You're a security expert wishing to write an unsmashable stack
- You're me trying to teach you stacks

# Recursion

# Compare iteration and recursion

```
while (true) do
  print "hello";
end while
```

```
function $f()$ {
    print "hello";
    $f()$;
}
$f()$;
```

*both programs yield the same infinite loop*

What are the differences?

Why should we bother?

# Difference? Forget assignments

```
input $n$;
$r = 1$
for $(i = 1$ to $n)$ do
  $r = r \times i$
end for
output $r$
```

```
function $f(n)$ {
  if $(n = 0)$ then
    return 1
  end if
  return $n \times f(n-1)$
}
output $f(n)$;
```

- Both programs compute $n!$
- *Iteration*: assignments; *recursion*: no assignments
- **Computation**({tests, assignments, iterations})$=$**Computation**({tests, recursion})

  Function call ⇔ **saving on a stack** *(recursion makes implicit assignments)*

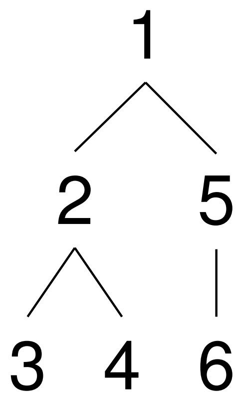
# Termination

- Make sure your recursions **terminate**
- If $f(n)$ is recursive,
  - recurse on smaller integers, e.g. $f(n-1)$ or $f(n/2)$
  - provide “base cases” where you do not recurse, e.g. $f(0)$ or $f(1)$
- Compare with *induction*:

  prove a statement for $n = 0$; prove that if it holds for all $i < n$ then it holds for $n$ too; conclude it holds for all $n$
- *Typical recursive algorithm* $f(n)$:

```
if $n$ is a “base case” then
  compute $f(n)$ directly, do not recurse
else
  recurse on $f(i)$ with some $i < n$
end if
```

# Should we bother? Explore this tree



Try instructing the computer to explore this tree structure in "depth-first order" (i.e. so that it prints $1, 2, 3, 4, 5, 6$)

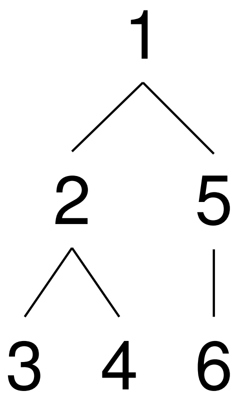
$A_1$: $A_{11} = 2$, $A_{12} = 5$
$A_2$: $A_{21} = 3$, $A_{22} = 4$
$A_3$: $\varnothing$
$A_4$: $\varnothing$
$A_5$: $A_{51} = 6$
$A_6$: $\varnothing$

Encoding: use a jagged array $A$

$A_{ij} =$ label of $j$-th child of node $i$

# The iterative failure

```
int $a = 1$;
print $a$;
for (int $z = 1$ to $|A_a|$) do
  int $b = A_{az}$;
  print $b$;
  for (int $y = 1$ to $|A_b|$) do
    int $c = A_{by}$;
    print $c$;
    ...
  end for
end for
```



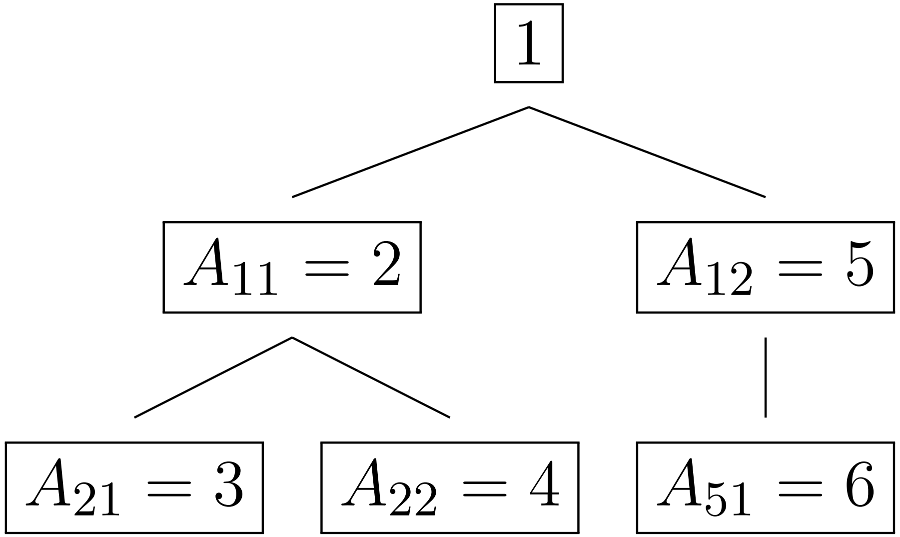
*Must the code change according to the tree structure???*

We want one code which works for **all** trees!

# Rescued by recursion

```
function $f(\texttt{int}\ \ell)$ {
    print $\ell$;
    for $(\texttt{int}\ i = 1$ to $|A_\ell|)$ do
      $f(A_{\ell i})$;
    end for
}
```

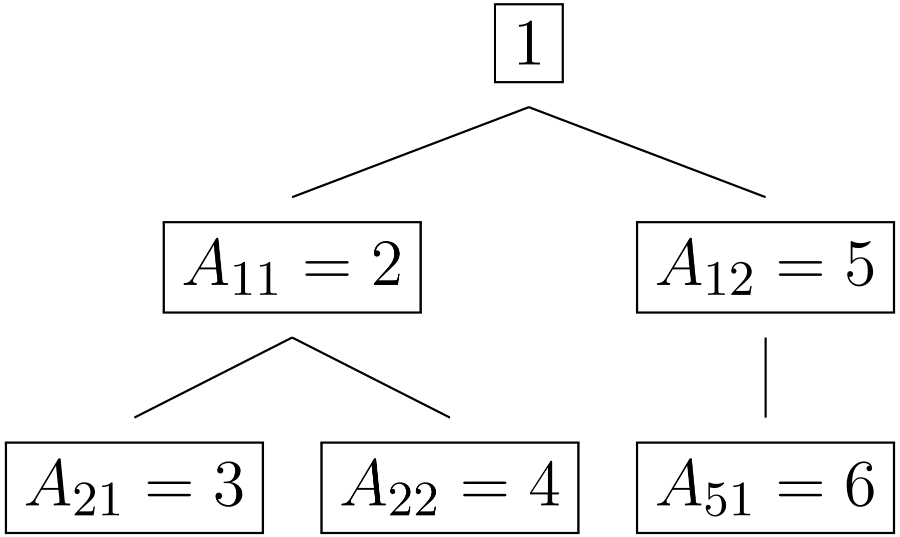
```
main() { $f(1)$; }
```



# Rescued by recursion

```
function $f$(int $\ell$) {
    print $\ell$;
    for (int $i = 1$ to $|A_\ell|$) do
        $f(A_{\ell i})$;
    end for
}
```

```
main() { $f(1)$; }
```



1. $\ell = 1$; print 1
2. $|A_1| = 2$; $i = 1$
3. call $f(A_{11} = 2)$ [push $\ell = 1$]
4. $\ell = 2$; print 2
5. $|A_2| = 2$; $i = 1$
6. call $f(A_{21} = 3)$ [push $\ell = 2$]
7. $\ell = 3$; print 3
8. $A_3 = \varnothing$
9. return [pop $\ell = 2$]
10. $|A_2| = 2$; $i = 2$
11. call $f(A_{22} = 4)$ [push $\ell = 2$]
12. $\ell = 4$; print 4
13. $A_4 = \varnothing$
14. return [pop $\ell = 2$]
15. return [pop $\ell = 1$]
16. $|A_1| = 2$; $i = 2$
17. call $f(A_{12} = 5)$ [push $\ell = 1$]
18. $\ell = 5$; print 5
19. $|A_5| = 1$; $i = 1$
20. call $f(A_{51} = 6)$ [push $\ell = 5$]
21. $\ell = 6$; print 6
22. $A_6 = \varnothing$
23. return [pop $\ell = 5$]
24. return [pop $\ell = 1$]
25. return; end

# Recursion power

- Can recursion can express programs that iterations cannot?
- Same “expressive power”

  *you can write the programs either way*
- Some programs easier to write using recursion

# Applications of recursion

# Listing permutations

- **Given an integer $n > 1$, list all permutations $\{1, \ldots, n\}$**
- Eg. $n = 4$: assume list of permutations of $\{1, 2, 3\}$

$$(1,2,3),(1,3,2),(3,1,2),(3,2,1),(2,3,1),(2,1,3)$$

- Write each four times, write the number 4 in every position:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | **4** | | 3 | 2 | 1 | **4** |
| 1 | 2 | **4** | 3 | | 3 | 2 | **4** | 1 |
| 1 | **4** | 2 | 3 | | 3 | **4** | 2 | 1 |
| **4** | 1 | 2 | 3 | | **4** | 3 | 2 | 1 |
| | | | | | | | | |
| 1 | 3 | 2 | **4** | | 2 | 3 | 1 | **4** |
| 1 | 3 | **4** | 2 | | 2 | 3 | **4** | 1 |
| 1 | **4** | 3 | 2 | | 2 | **4** | 3 | 1 |
| **4** | 1 | 3 | 2 | | **4** | 2 | 3 | 1 |
| | | | | | | | | |
| 3 | 1 | 2 | **4** | | 2 | 1 | 3 | **4** |
| 3 | 1 | **4** | 2 | | 2 | 1 | **4** | 3 |
| 3 | **4** | 1 | 2 | | 2 | **4** | 1 | 3 |
| **4** | 3 | 1 | 2 | | **4** | 2 | 1 | 3 |

# The algorithm

- If you can list permutations for $n-1$, you can do it for $n$
- **Base case**: $n=1$ yields the permutation $(1)$ (no recursion)

```
function permutations($n$) {
if ($n = 1$) then
    $L = \{(1)\}$;
else
    $L' =$ permutations($n-1$);
    $L = \varnothing$;
    for ($\pi = (a_1, \ldots, a_{n-1}) \in L'$) do
        for ($i \in \{1, \ldots, n\}$) do
            $L \leftarrow L \cup \{(a_1, \ldots, a_{i-1}, n, a_i, \ldots, a_{n-1})\}$;
        end for
    end for
end if
return $L$;
}
```

# Implementation details

- $L, L'$ are (mathematical) sets: implementation?
- given perm. $(a_1, \ldots, a_{n-1})$, need to produce perm. $(a_1, \ldots, a_{i-1}, n, a_i, \ldots, a_{n-1})$: implementation?
- **Needed operations**:
  - size of set $L$ (known a priori: $|L| = n!$)
  - scan all elements of set $L'$ in some order (**for** at Step 6)
  - insert list element at arbitrary position at Step 8
  - add an element to $L$
  - $L', L$ must have the same type by Steps 4, 12
- $L', L$ can be arrays or lists
- $(a_1, \ldots, a_{n-1})$ can be a singly-linked (or doubly-linked) list

# Hanoi tower

## Recursive approach

In order to move $k$ discs from stack 1 to stack 3:

1. move topmost $k-1$ discs on stack 1 to stack 2
2. move largest disc on stack 1 to stack 3
3. move $k-1$ discs on stack 2 to stack 3

# Hanoi tower

## Recursive approach

In order to move $k$ discs from stack 1 to stack 3:

1. move topmost $k-1$ discs on stack 1 to stack 2
2. move largest disc on stack 1 to stack 3
3. move $k-1$ discs on stack 2 to stack 3

Reduce the problem to subproblem with $k-1$ discs

**Assumption**: subproblems for $k-1$ at Steps 1 and 3 are the *same type of problem* as for $k$

*The assumption holds because the disc being moved at Step 2 is the largest: a Hanoi tower game "works the same way" if you add largest discs at the bottom of the stacks*

# Hanoi tower

## Recursive approach

In order to move $k$ discs from stack 1 to stack 3:

1. move topmost $k-1$ discs on stack 1 to stack 2
2. move largest disc on stack 1 to stack 3
3. move $k-1$ discs on stack 2 to stack 3

Reduce the problem to subproblem with $k-1$ discs

**Assumption**: subproblems for $k-1$ at Steps 1 and 3 are the *same type of problem* as for $k$

*The assumption holds because the disc being moved at Step 2 is the largest: a Hanoi tower game "works the same way" if you add largest discs at the bottom of the stacks*

*Do you need stacks to implement this algorithm?*

# Recursive functions

# Function class

Aim to define a class $\mathscr{R}$ of recursive functions with special properties

# Initial functions

The following functions are in $\mathscr{R}$

- **zero**: $\forall x \in \mathbb{N} \quad Z(x) = 0$
- **next**: $\forall x \in \mathbb{N} \quad N(x) = x + 1$
- **projection**: $\forall x = (x_1, \ldots, x_n) \in \mathbb{N}^n \quad P_i^n(x) = x_i$

# Replacement schema

- Given:
  - $h_1, \ldots h_m : \mathbb{N}^n \to \mathbb{N}$ in $\mathscr{R}$
  - $g : \mathbb{N}^m \to \mathbb{N}$ in $\mathscr{R}$
  - $x \in \mathbb{N}^n$
- $f(x) = g(h_1(x), \ldots, h_m(x))$ is in $\mathscr{R}$

# Primitive recursion

- Given:
  - $g : \mathbb{N}^n \to \mathbb{N}$ in $\mathscr{R}$
  - $h : \mathbb{N}^{n+2} \to \mathbb{N}$ in $\mathscr{R}$
  - $x \in \mathbb{N}^n$ and $y \in \mathbb{N} \setminus \{0\}$
- The following $f : \mathbb{N}^{n+1} \to \mathbb{N}$ is in $\mathscr{R}$:

$$\begin{aligned} f(x,0) &= g(x) \\ f(x,N(y)) &= h(x,y,f(x,y)) \end{aligned}$$

---

- If $n = 0$, then $f : \mathbb{N} \to \mathbb{N}$ is in $\mathscr{R}$ if $\exists k \in \mathbb{N}$ s.t.:

$$\begin{aligned} f(0) &= k \\ f(N(y)) &= h(y,f(y)) \end{aligned}$$

# $\mu$-operator

- Given:
  - $g : \mathbb{N}^{n+1} \to \mathbb{N}$ s.t. $\forall x \in \mathbb{N}^n \ \exists y \in \mathbb{N} \ (g(x,y) = 0)$
  - a quantifier $\mu$
    s.t. $\mu y \ g(x,y) = \min\{y \in \mathbb{N} \mid g(x,y) = 0\}$
- The function $f(x) = \mu y \ g(x,y)$ is in $\mathscr{R}$

# Examples

- $x + y = +(x, y)$ is in $\mathscr{R}$
  - $+(x, 0) = P_1^1(x)$
  - $+(x, N(y)) = P_3^3(x, y, N(+(x, y)))$
  - $\Rightarrow + \in \mathscr{R}$ by proj., next and primitive recursion
- exchange of variables is in $\mathscr{R}$
  - suppose $g : \mathbb{N}^2 \to \mathbb{N}$ is in $\mathscr{R}$
  - let $f(x, y) = g(y, x)$ for all $x, y \in \mathbb{N}$: is $f \in \mathscr{R}$?
  - we have $x = P_1^2(x, y)$ and $y = P_2^2(x, y)$
  - so, can write $f(x, y) = g(P_2^2(x, y), P_1^2(x, y))$
  - $\Rightarrow f \in \mathscr{R}$ by projection and replacement

# An algorithmic flavour

- Can see these proofs as algorithms
- Extend domains/ranges from $\mathbb{N}$ to arbitrary ordered sets
- **The program** : explicit expression in terms of initial functions and schema

  *(provides description of mechanical procedure)*
- **The tape** : variables with values

  *(recursion stack)*
- Thm.

  A function is recursive iff it is Turing-computable

Recursion is TM-equivalent

# Recursion in logic

- **Axioms**: sentences that are true by definition
- **$\Phi \vdash \psi$**: sentence $\psi$ is a logical consequence of sentences in set $\Phi$
- **Theory**: set $T$ of sentences containing set $A$ of axioms such that for each $\phi \in T$, $A \vdash \phi$
- A theory is **consistent** when it does not contain pairs of contradictory sentences $\phi, \neg\phi$
- A theory is **complete** when every true statement is in the theory
- Let $T$ be a theory that can define $\mathbb{N}$
- **Gödel's sentence**: define $\gamma$ as $T \nvdash \gamma$

# Gödel's incompleteness theorem

Thm.

If $T$ is consistent, then $T$ is incomplete

Proof

- Assume $T$ consistent, aim to show $\exists$ true sentence $\not\in T$
- For all $\phi$, exactly one in $\{\phi, \neg\phi\}$ is true
- $\Rightarrow$ exactly one in $\{\gamma, \neg\gamma\}$ is true
- Is $\gamma \in T$? If so, then $T \vdash \gamma$, which means that $T \vdash (T \not\vdash \gamma)$, i.e. $T \not\vdash \gamma$, i.e. $\gamma \not\in T$ (**contradiction**)
- Is $\neg\gamma \in T$? If so, then $T \vdash \neg\gamma$, i.e. $T \vdash \neg(T \not\vdash \gamma)$, that is $T \vdash (T \vdash \gamma)$, thus $T \vdash \gamma$
- In other words, assuming $T \vdash \neg\gamma$ leads to $T \vdash \gamma$, which implies $T$ is inconsistent (**contradiction**)
- $\Rightarrow$ neither $\gamma$ nor $\neg\gamma$ is in $T$, one of them is true, $T$ is incomplete

# Does this recursion terminate?

- Not immediately evident that the recursive definition $T \nvdash \gamma$ has a “base case”
- In Gödel’s proof sentences and theories are encoded as integers
- Most difficult part of Gödel’s proof: show $\gamma$ can be defined by means of a recursive function

# End of Lecture 3