INF421, Lecture 3
Stacks and recursion

Leo Liberti

LIX, École Polytechnique, France

Course

- **Objective**: to teach you some data structures and associated algorithms
- **Evaluation**: TP noté en salle info le 16 septembre, Contrôle à la fin. Note: $\max(CC, \frac{3}{4}CC + \frac{1}{4}TP)$
- **Organization**: fri 26/8, 2/9, 9/9, 16/9, 23/9, 30/9, 7/10, 14/10, 21/10, amphi 1030-12 (Arago), TD 1330-1530, 1545-1745 (SI31,32,33,34)
- **Books**:
  1. Ph. Baptiste & L. Maranget, *Programmation et Algorithmique*, Ecole Polytechnique (Polycopié), 2006
  2. G. Dowek, *Les principes des langages de programmation*, Editions de l'X, 2008
  3. D. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997
  4. K. Mehlhorn & P. Sanders, *Algorithms and Data Structures*, Springer, 2008
- **Website**: `www.enseignement.polytechnique.fr/informatique/INF421`
- **Contact**: `liberti@lix.polytechnique.fr` (e-mail subject: INF421)

Lecture summary

- Function calls
- Stacks and applications
- Recursion

Minimal knowledge

- A function f can call another function g: every time this happens, the address A_g of the instruction of f just after the instruction `call` g in f is stored in memory; when g ends, control is transfered to A_g
- A stack is a data structure where you can only read (and delete) the last element you added to it
- Recursion is when a function f calls itself. Since essentially it allows repeated execution of the code of f, it is similar to a loop. It is sometimes more convenient to write code using recursion than loops.

Function calls

What is a function call?

A recipe is a program, you are the CPU, your kitchen is the memory

Salad and walnuts recipe

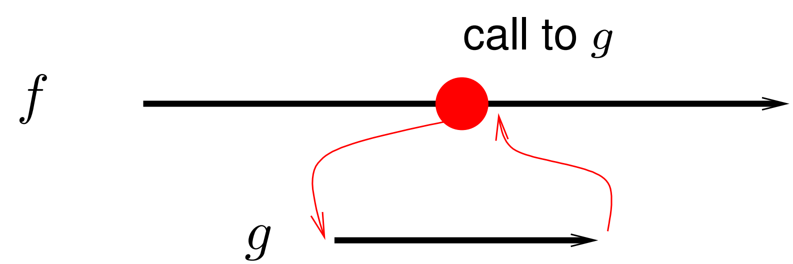
1. add the salad
2. add the walnuts
3. add vinaigrette
4. toss and serve

- Seems simple enough, but when you get to Step 3 you realize that in order to add the vinaigrette you need to *prepare it first!*
- So you leave everything as is, mix oil and vinegar, add salt, then resume the recipe from where you'd left it
- You just called a function

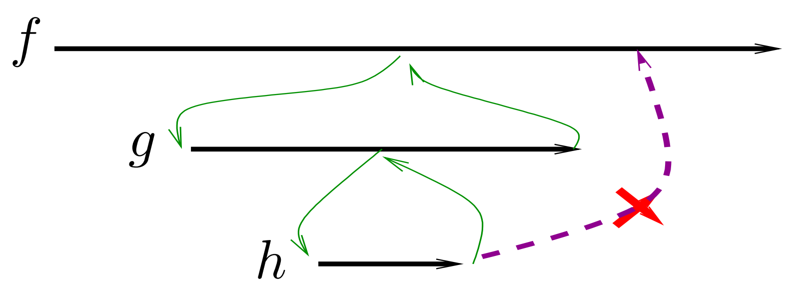
Functions essentials

- A function call is a diversion from the sequential instructions order
  - you need to know where to go next
  - you need to store the current instruction address so you can resume execution once the function terminates

f calls g:



- Assume f calls g and g calls h, and h is currently executing
- In order for f to resume control, g must have terminated first



h cannot pass control to f directly

Saving the state

- Every function defines a "naming scope" (denote an entity x defined within a function f by f::x)
- If f calls g, both may define a local variable x, but f::x and g::x refer to different memory cells
- Before calling g, f must therefore save its *current state*:
  - the name and address of each local variable in f
  - the address of the instruction just after "call g" in f
- When g ends, the current state of f is retrieved, and f resumes
- Need a data structure for saving current states
- As function calls are very common, it must be as simple and efficient as possible

Argument passing

x a variable in f, and g needs to access it:

f calls $g(x)$

- Let variable x name a cell with address A_x and value V_x

Argument passing

x a variable in f, and g needs to access it:

f **calls** $g(x)$

- Let variable x name a cell with address A_x and value V_x
- Passing by reference: $g(A_x)$

  if g changes V_x then the change is visible in f

Argument passing

x a variable in f, and g needs to access it:

f calls $g(x)$

- Let variable x name a cell with address A_x and value V_x
- Passing by reference: $g(A_x)$

  if g changes V_x then the change is visible in f
- Passing by value: $g(V_x)$

  if g changes V_x then the change is **not** visible in f

Argument passing

x a variable in f, and g needs to access it:

f calls $g(x)$

- Let variable x name a cell with address A_x and value V_x
- Passing by reference: $g(A_x)$

  if g changes V_x then the change is visible in f

- Passing by value: $g(V_x)$

  if g changes V_x then the change is **not** visible in f

- This is a *model*, not the actual implementation used by languages

Argument passing

x a variable in f, and g needs to access it:

$\boxed{f \textbf{ calls } g(x)}$

- Let variable x name a cell with address A_x and value V_x
- Passing by reference: $g(A_x)$

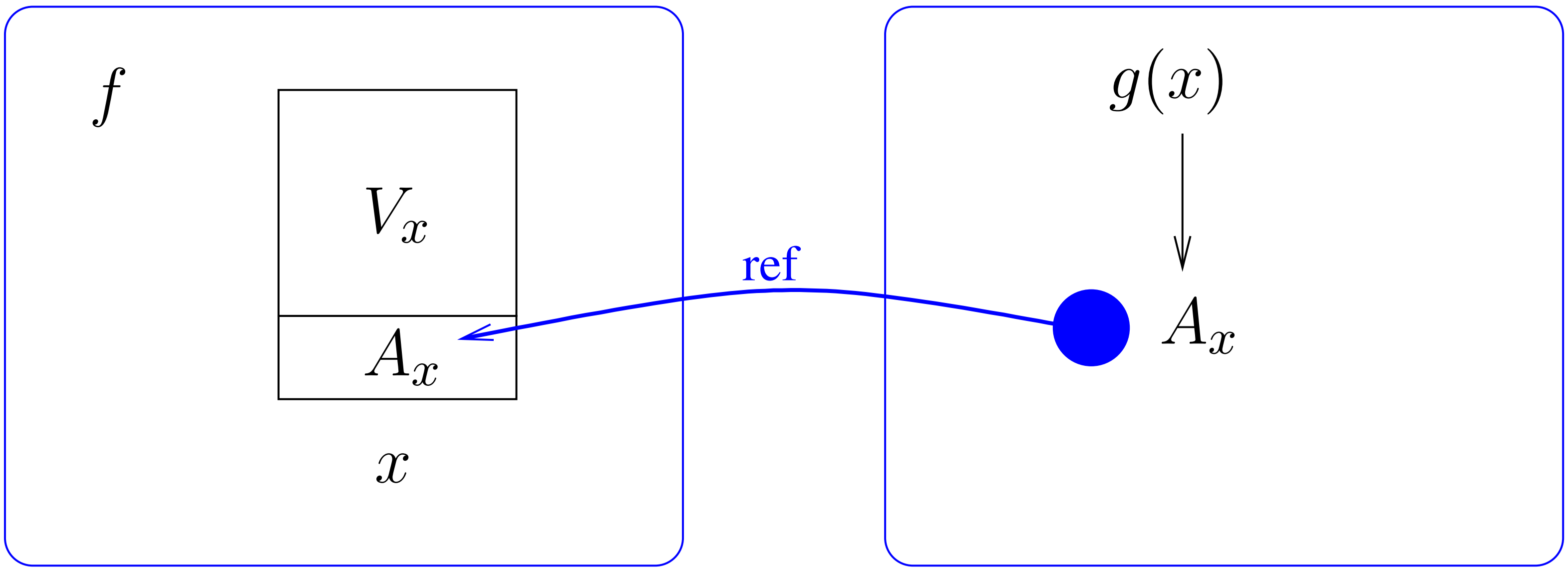
  if g changes V_x then the change is visible in f

- Passing by value: $g(V_x)$

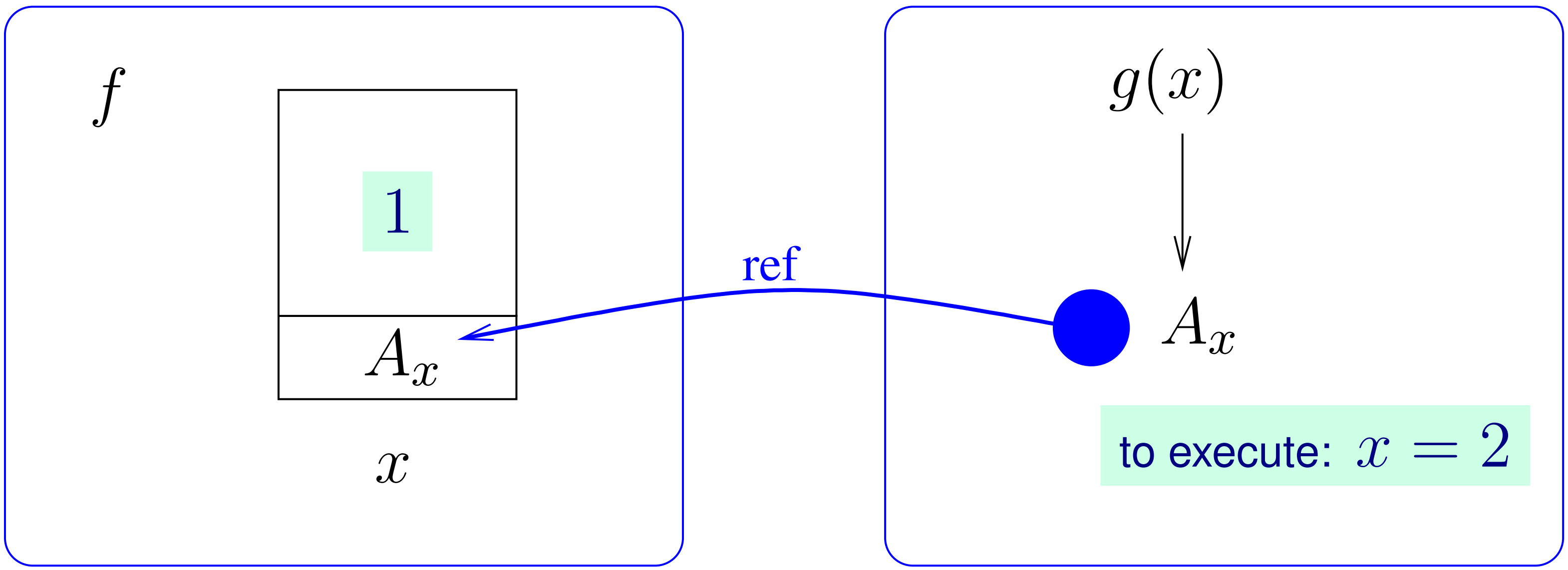
  if g changes V_x then the change is **not** visible in f

- This is a *model*, not the actual implementation used by languages
- In practice, Java behaves as if basic types (`char`, `int`, `long`, `float`, `double`) were passed by value, and composite types by reference

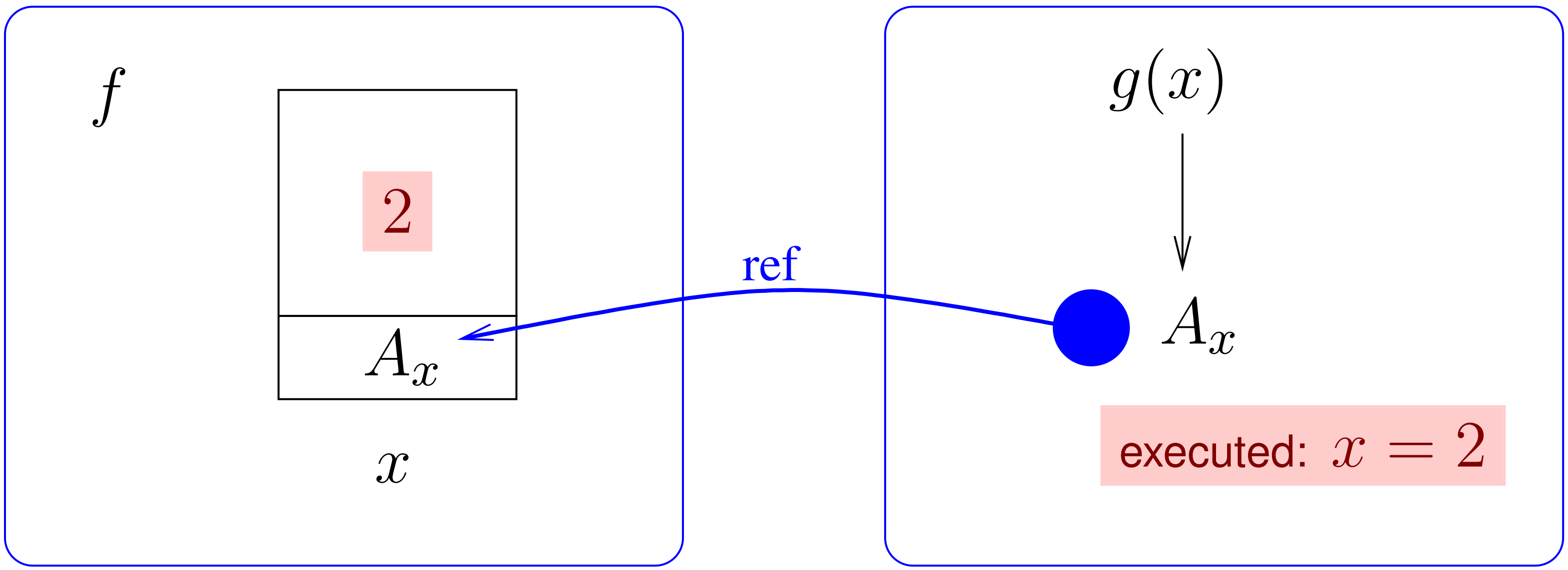
Passing by reference



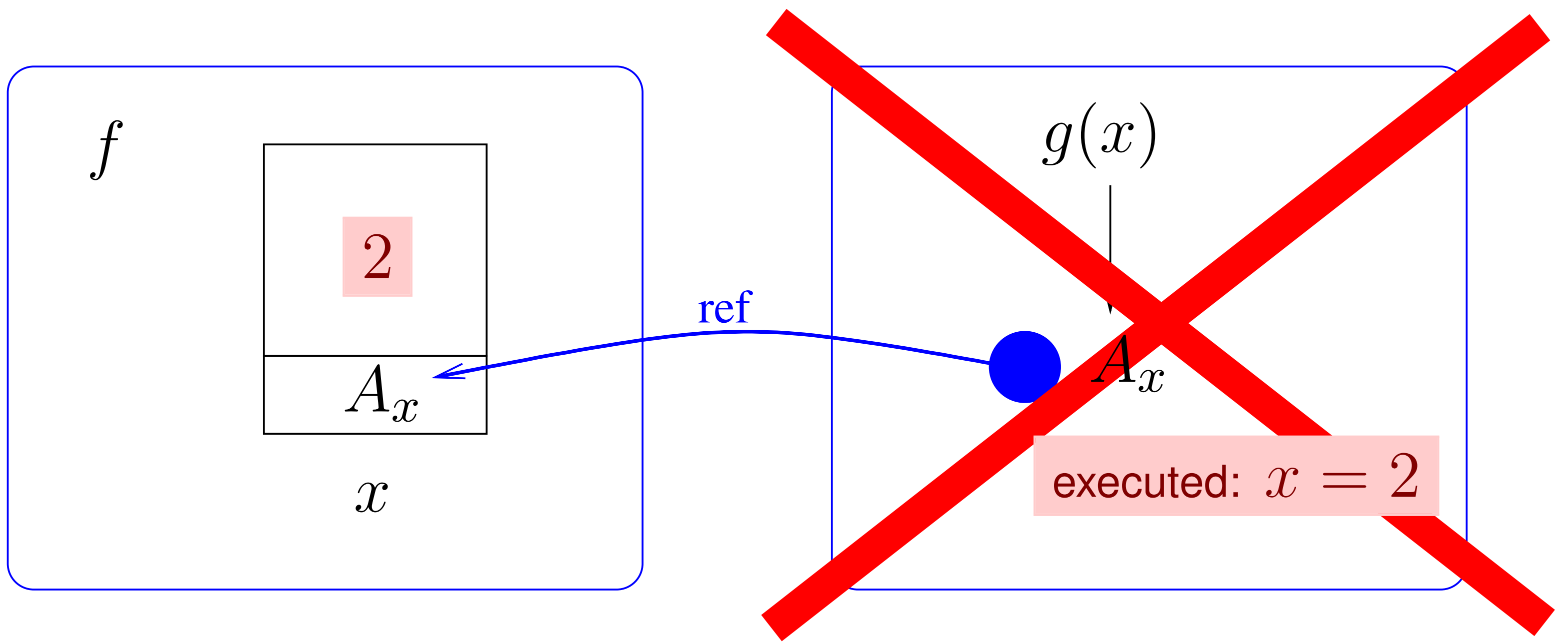
Passing by reference



Passing by reference

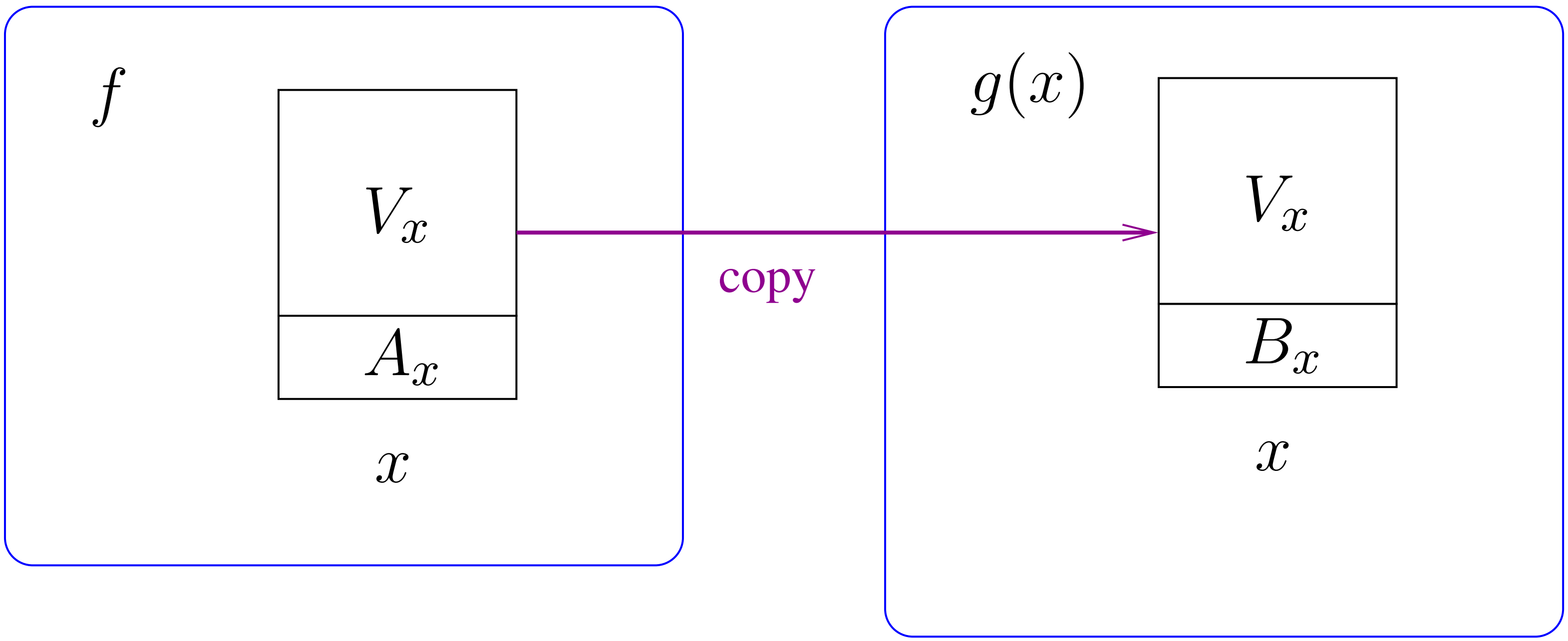


Passing by reference

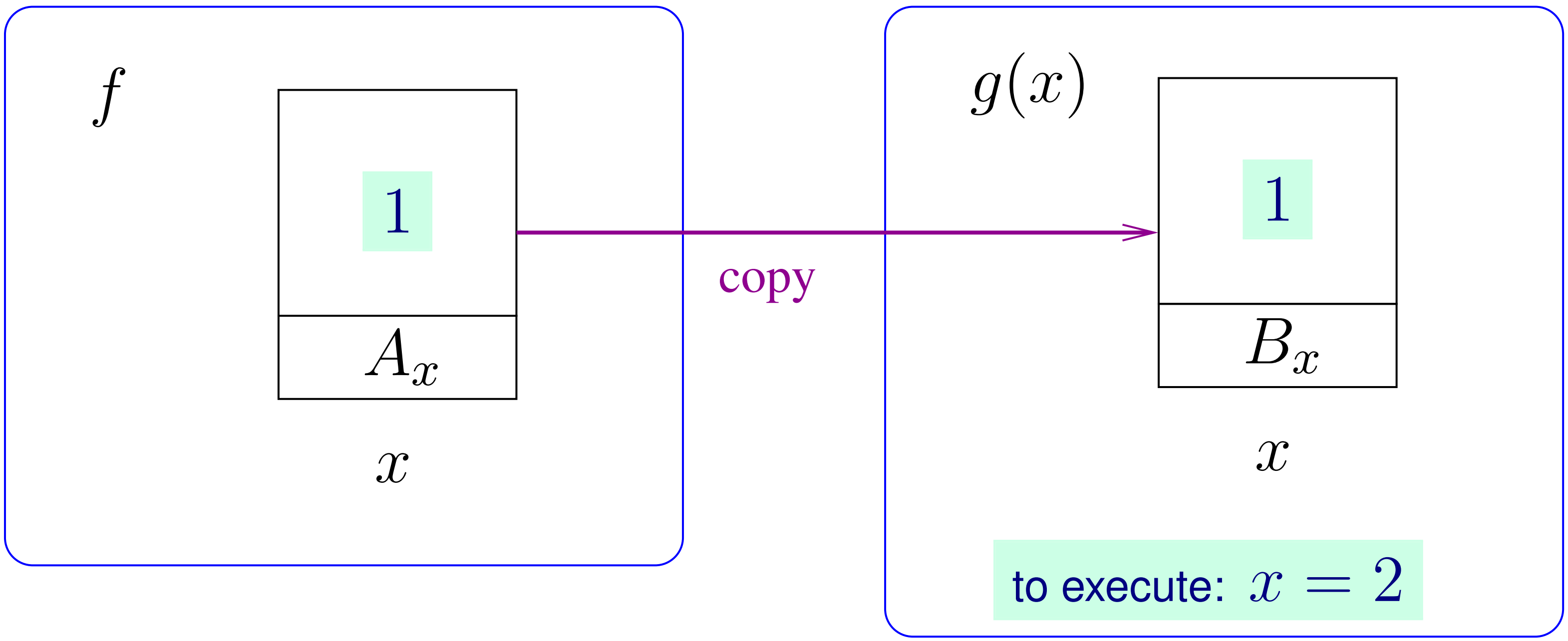


When g terminates, the new value of x is available to f

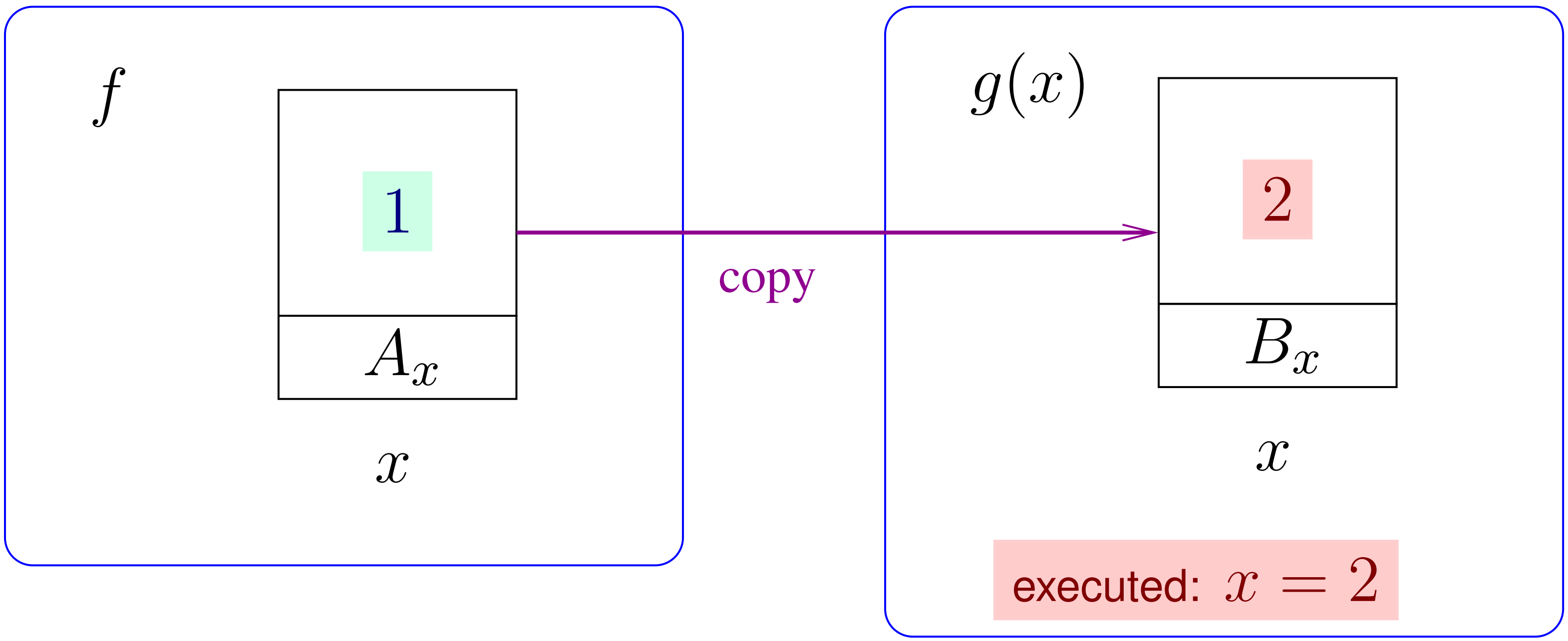
Passing by value



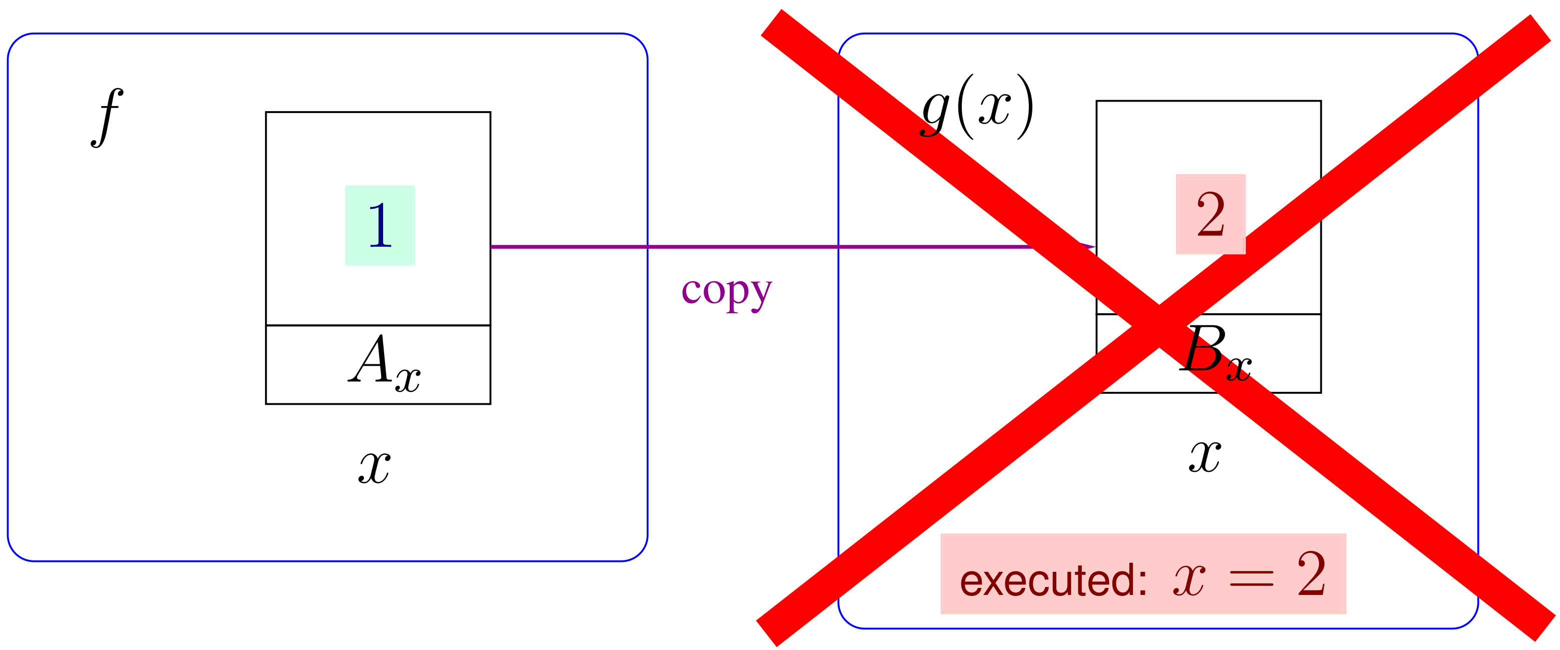
Passing by value



Passing by value



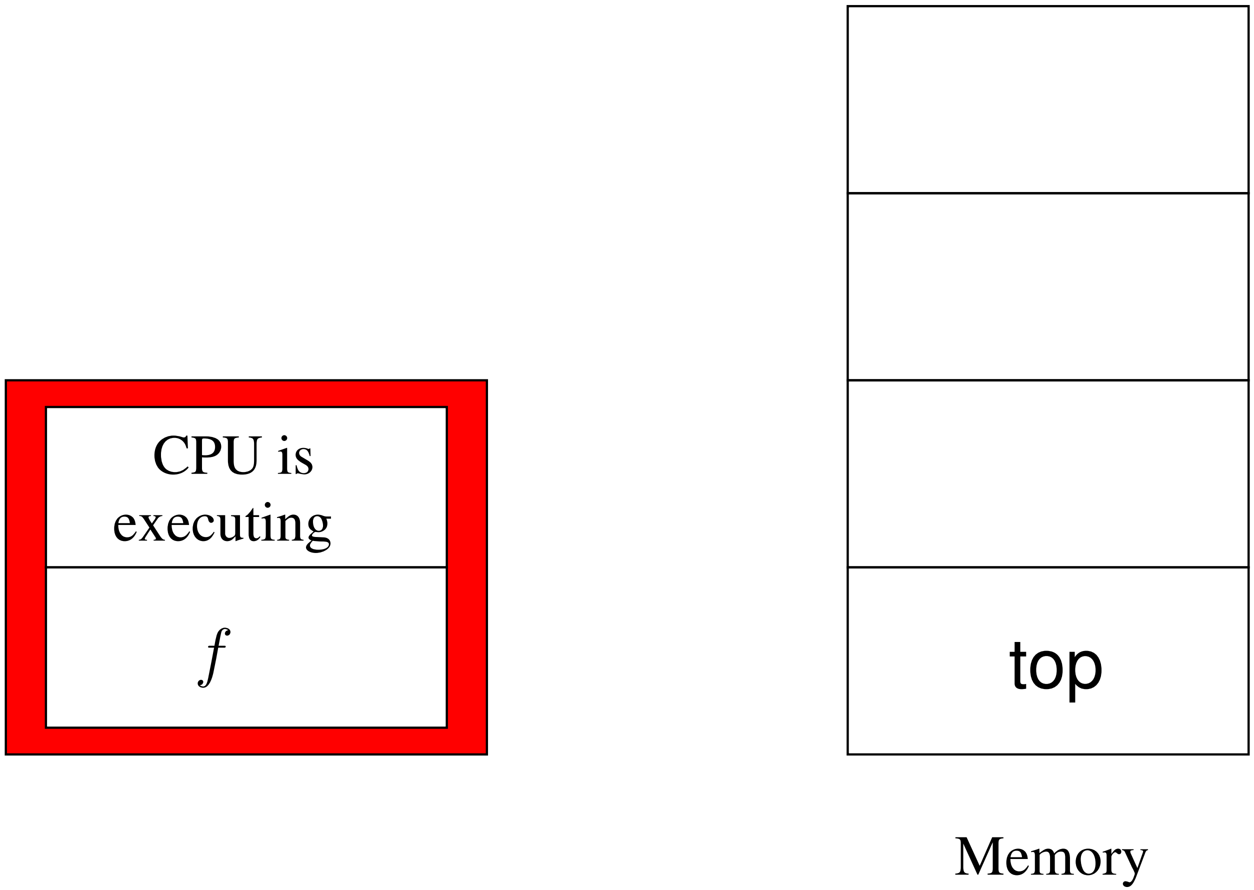
Passing by value



When g terminates, the new value of x is lost

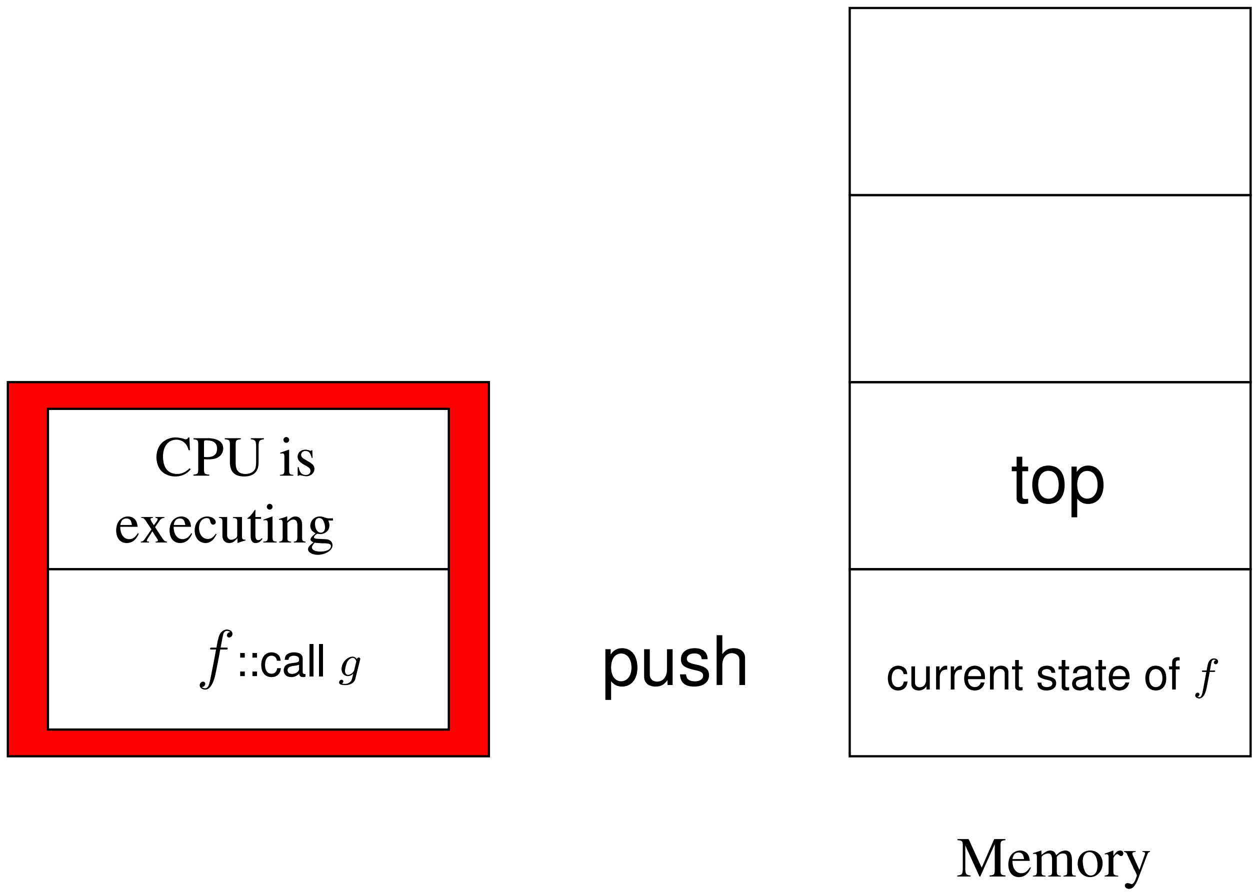
Current states are saved to a stack

f calls g calls h



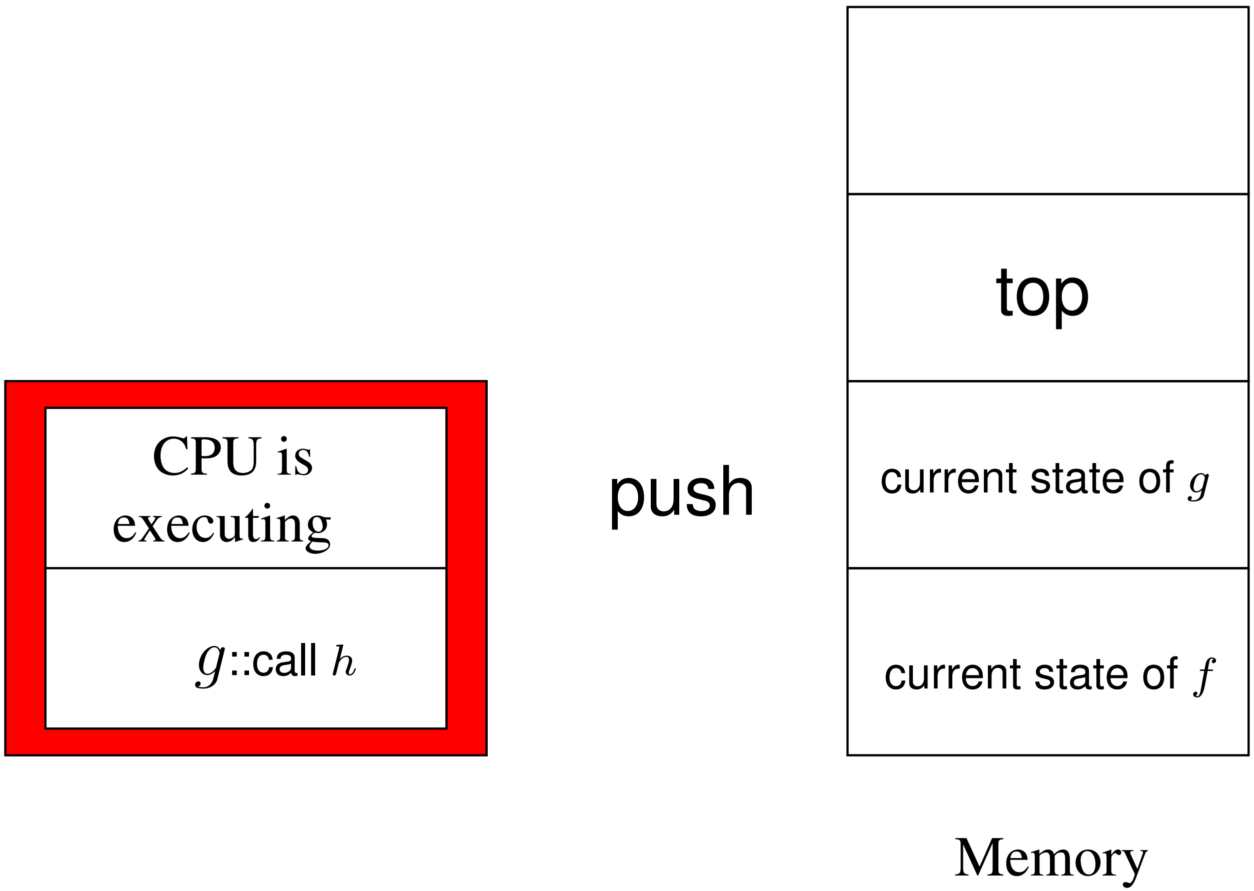
Current states are saved to a stack

f calls g calls h



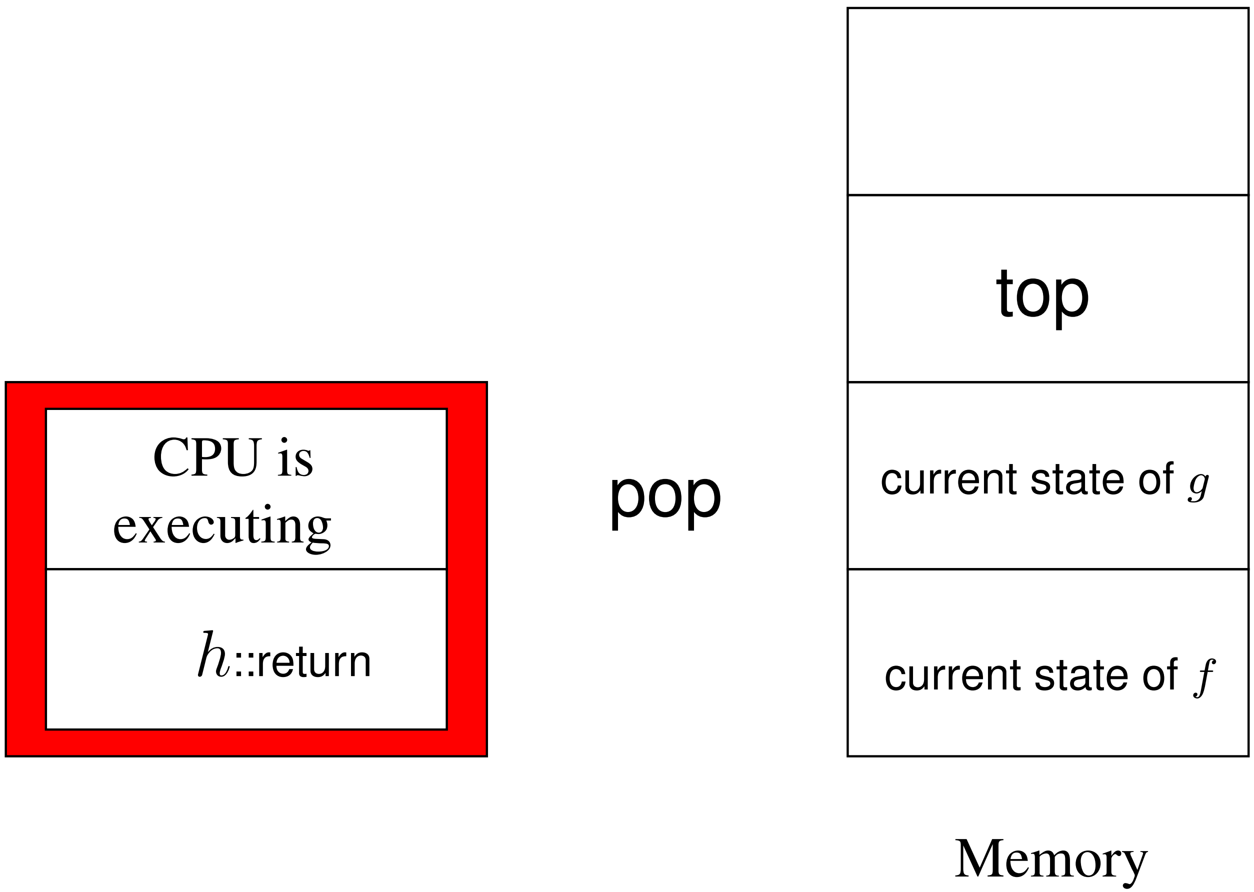
Current states are saved to a stack

f calls g calls h



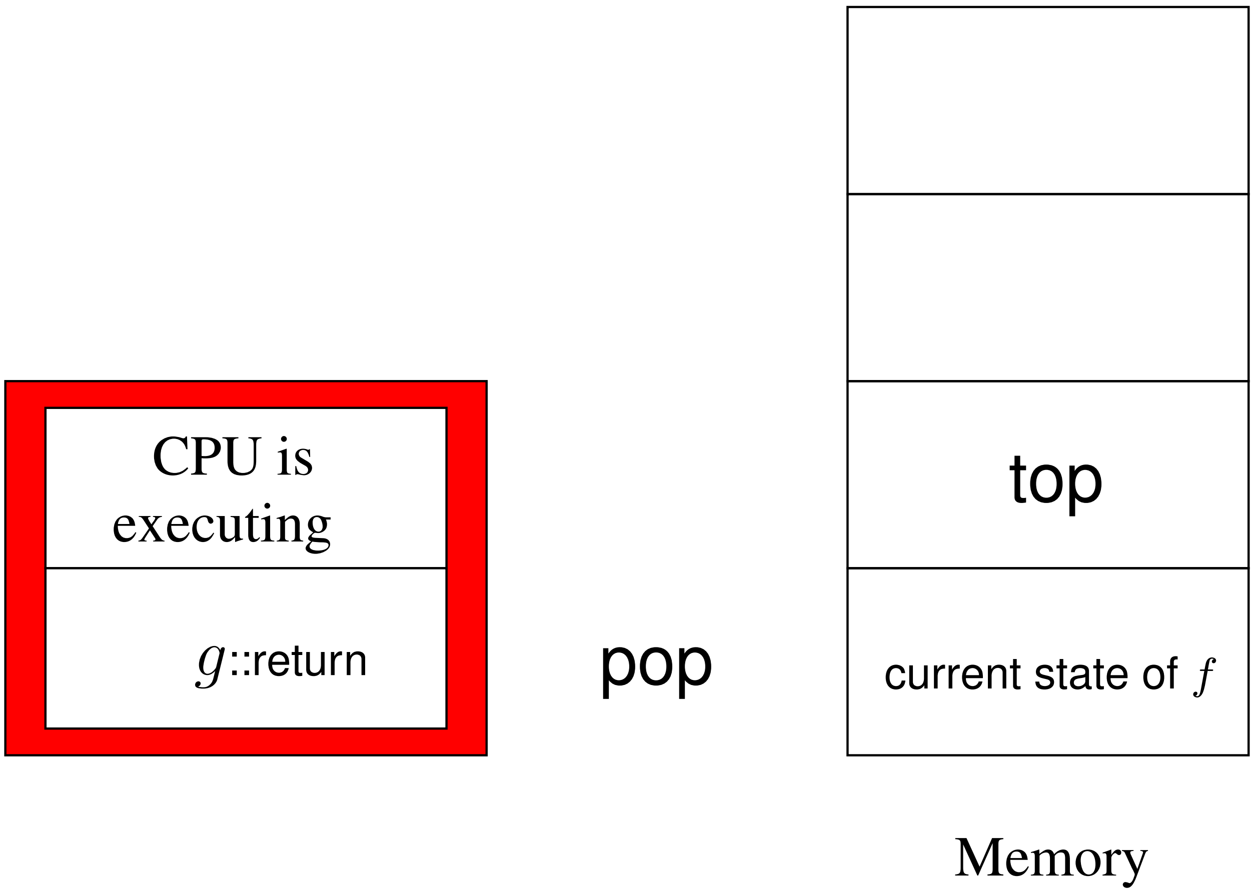
Current states are saved to a stack

f calls g calls h



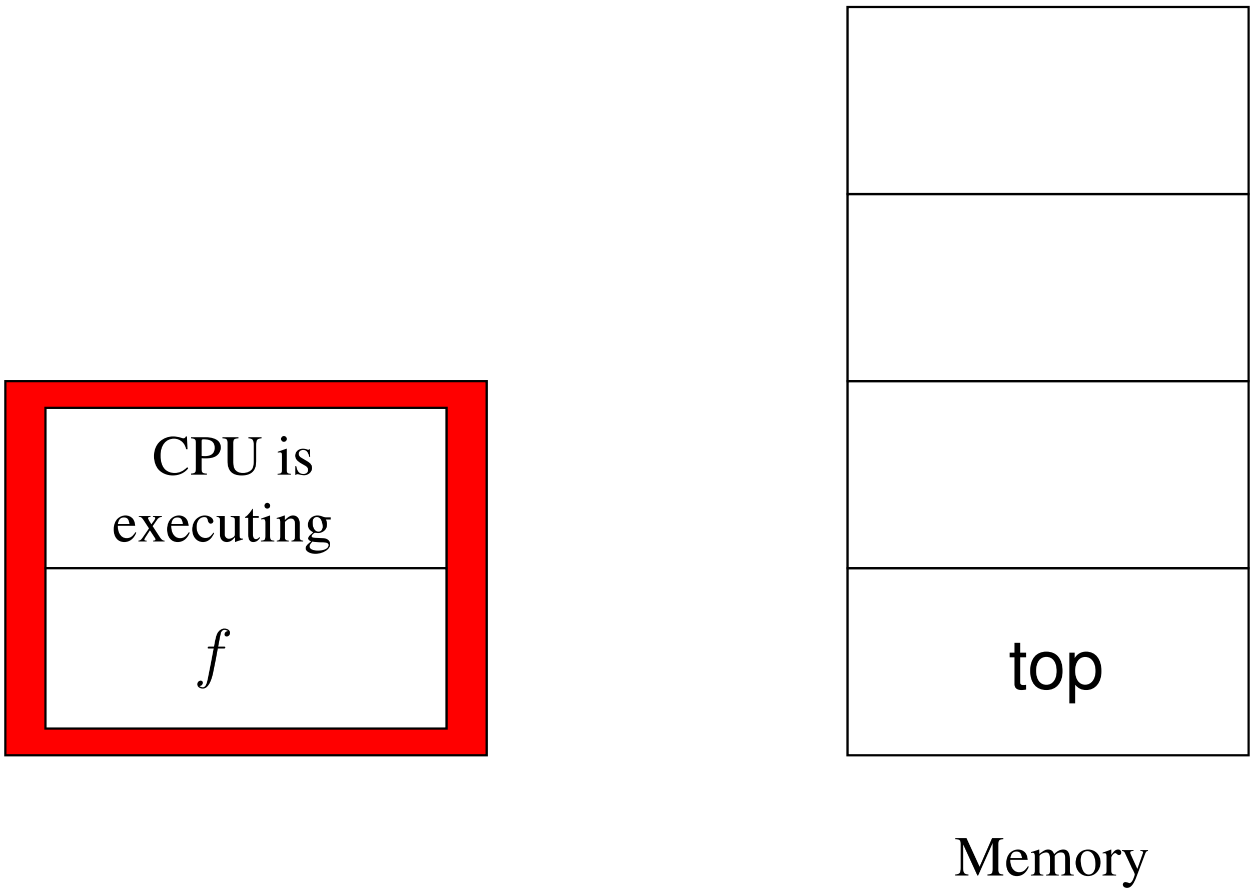
Current states are saved to a stack

f calls g calls h



Current states are saved to a stack

f calls g calls h



Stacks and applications

Stack

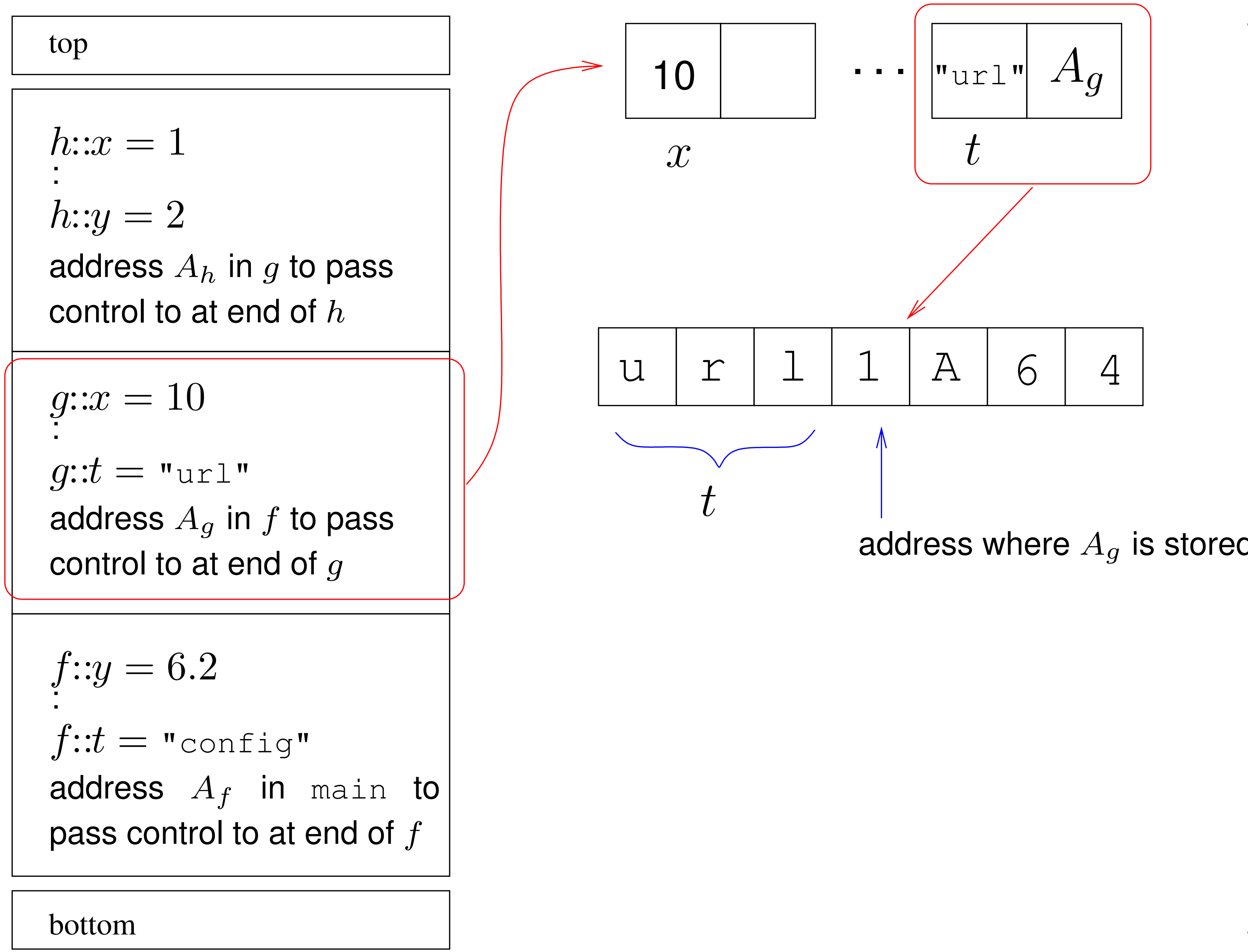
- Linear data structure
- Accessible from only one end (top)
- Operations:
  - add a data node on the top (*push data*)
  - remove a data node from the top (*pop data*)
  - test whether stack is empty
- Every operation must be $O(1)$
- Don't need insertion/removal from the middle: can implement using arrays

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?



How does it work?

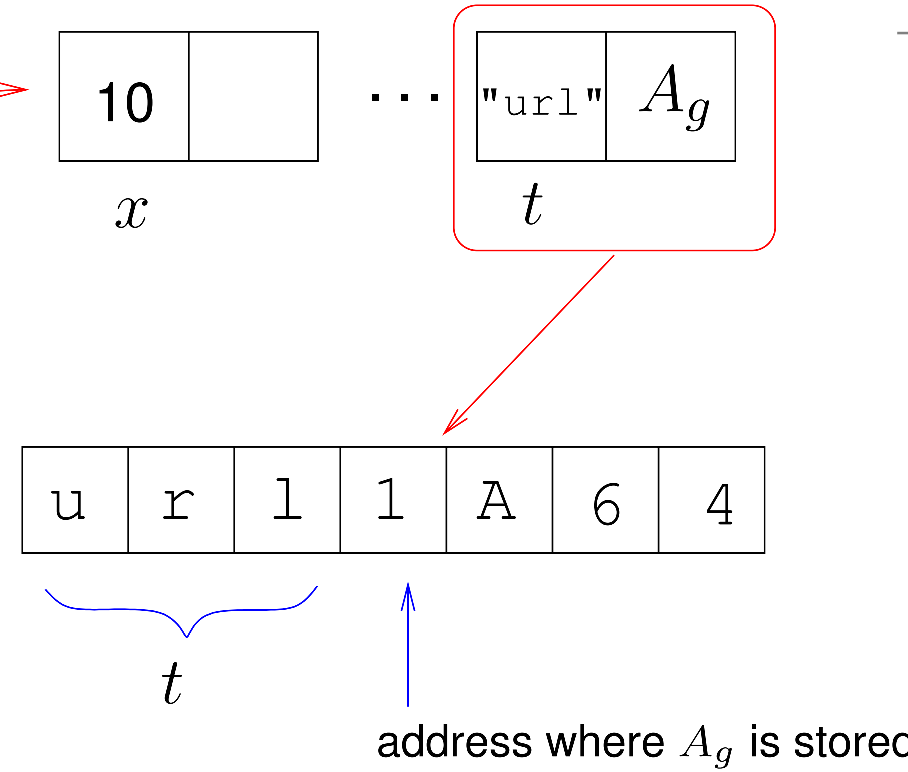
top

$h::x = 1$
⋮
$h::y = 2$
address A_h in g to pass control to at end of h

$g::x = 10$
⋮
$g::t =$ `"url"`
address A_g in f to pass control to at end of g

$f::y = 6.2$
⋮
$f::t =$ `"config"`
address A_f in `main` to pass control to at end of f

bottom



$g::t$: 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

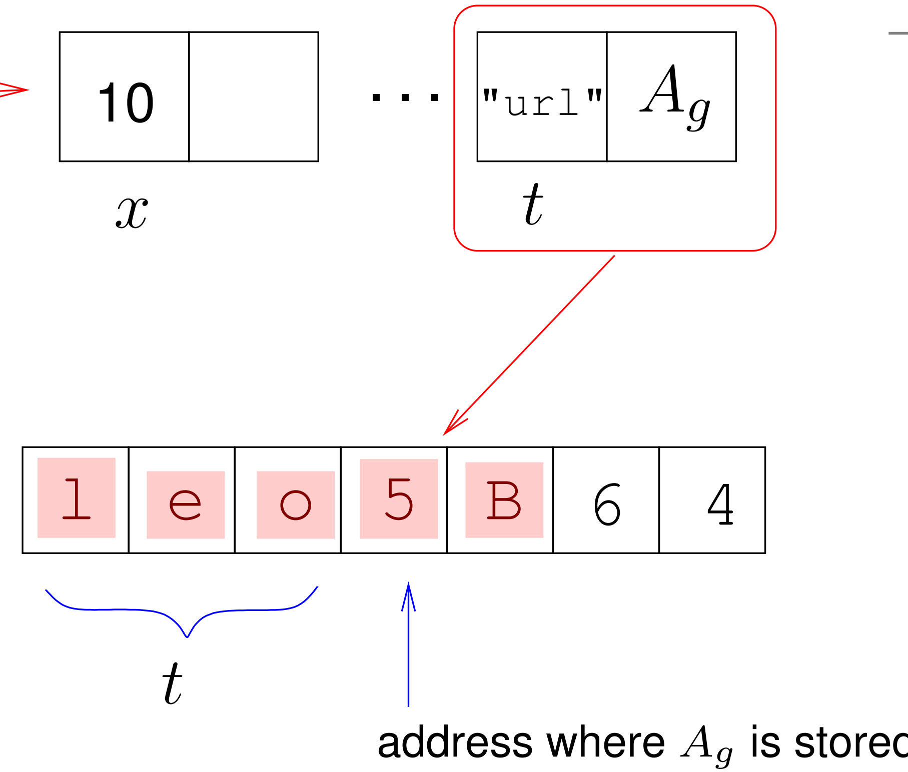
$f::y = 6.2$

$\vdots$

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

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

bottom



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

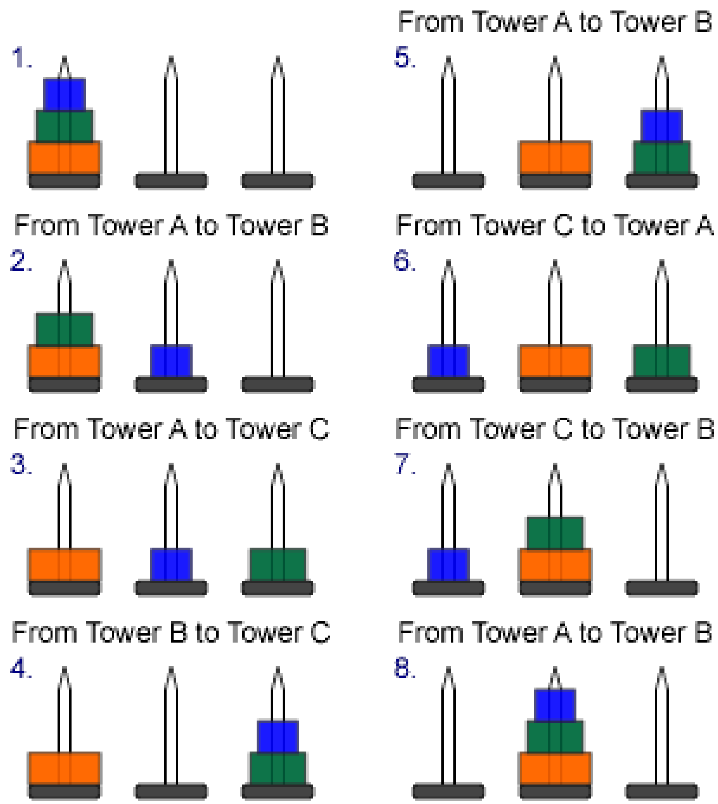
A_g =0x1A64 becomes A' =0x5B64

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

Checking brackets

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

1. s: the input string
2. for each i from 1 to $|s|$:
   (a) if s_i is an open bracket, push the corresponding closing bracket on the stack
   (b) if s_i is a closing bracket, pop a char t from the stack:
   - if the stack is empty, **error**: too many closing brackets
   - if $t \neq s_i$, **error**: closing bracket has wrong type
3. if stack is not empty, **error**: not enough closing brackets

Code for checking brackets

```
input string s; stack T; int i = 0;
while (i ≤ s.length) do
  if (s_i = '(') then
    T.push(')');
  else if (s_i = '[') then
    T.push(']');
  else if (s_i ∈ {')', ']'}) then
    if (T.isEmpty()) then
      error: too many closing brackets;
    else
      t = T.pop();
      if (t ≠ s_i) then
        error: wrong closing bracket type at i;
      end if
    end if
  end if
  i = i + 1;
end while
if (¬T.isEmpty()) 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 × i
end for
output r
```

```
function f(n) {
  if (n = 0) then
    return 1
  end if
  return n × f(n − 1)
}
f(n);
```

- Both programs compute $n!$
- Iterative version has assignments, recursive version does not
- Every computable function can be computed by means of {tests, assignments, iterations} *or* {tests, recursion}
- For language expressivity: "recursion = assignment + iteration"

  Don't forget that calling a function implies **saving the current state on a stack**

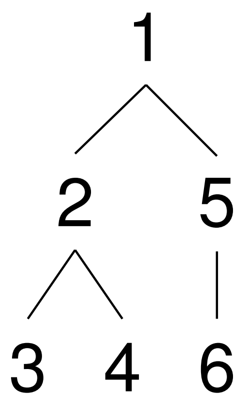
  (in recursion there is an implicit assignment of variable values to the stack memory)

Termination

- Make sure your recursions **terminate**
- For example: 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 $n=0$ and prove that if it holds for all $i<n$ then it holds for n too; conclude it holds for all n
- Typically, a recursive algorithm $f(n)$ is as follows:

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 $1, 2, 3, 4, 5, 6$)

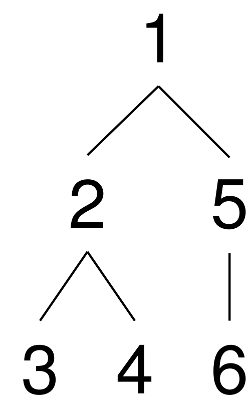
Encoding: use a jagged array A

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: $\varnothing$

$A_{ij} =$ 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



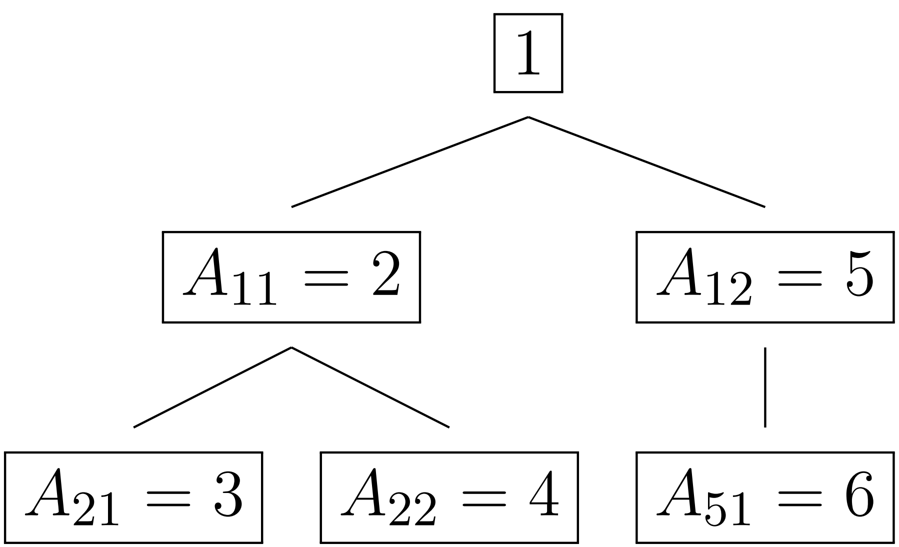
Must the code change according to the tree structure???

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

Rescued by recursion

```
function f(int ℓ) {
    print ℓ;
    for (int i = 1 to |A_ℓ|) do
      f(A_ℓi);
    end for
}
```

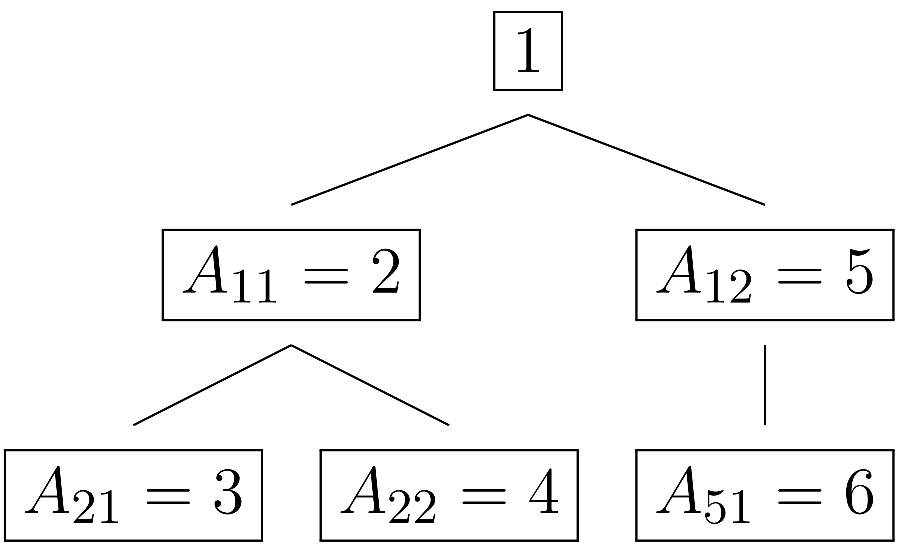
```
main() { f(1); }
```



Rescued by recursion

```
function f(int ℓ) {
    print ℓ;
    for (int i = 1 to |A_ℓ|) do
      f(A_ℓi);
    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

- At first sight, recursion can express programs that iterations cannot!
- As mentioned above, the “expressive power” of recursion and that of iteration are the same

  you can write the programs either way
- However, certain programs are more easily written with iteration, and some other with recursion
- **Warning**: always make sure your recursion terminates!

  There must be some “base cases” which do not recurse

Recursion power

- At first sight, recursion can express programs that iterations cannot!
- As mentioned above, the "expressive power" of recursion and that of iteration are the same

  you can write the programs either way
- However, certain programs are more easily written with iteration, and some other with recursion
- **Warning**: always make sure your recursion terminates!

  There must be some "base cases" which do not recurse

Write a program that lists all permutations of n elements

Listing permutations

- Given an integer $n > 1$, list all permutations $\{1, \dots, n\}$
- Example, $n = 4$
- Suppose you already listed all 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 4 times, and 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 = ∅;
    for ((π_1, ..., π_{n−1}) ∈ L') do
        for (i ∈ {1, ..., n}) do
            L ← L ∪ {(π_1, ..., π_{i−1}, n, π_i, ..., π_{n−1})};
        end for
    end for
end if
return L;
}
```

Implementation details

- L, L' are (mathematical) sets: how do we implement them?
- given list $(\pi_1, \ldots, \pi_{n-1})$, need to produce list $(\pi_1, \ldots, \pi_{i-1}, i, n, \ldots, \pi_{n-1})$: how do we implement these lists?
- **Needed operations**:
  - Size of L known a priori: $|L| = n!$
  - scan all elements of set L' in some order (**for** at Step 6)
  - insert a node at arbitrary position in list $(\pi_1, \ldots, \pi_{n-1})$ at Step 8
  - add an element to set L
  - L', L must have the same type by Steps 4, 12
- L', L can be arrays
- $(\pi_1, \ldots, \pi_{n-1})$ 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?

Appendix

Recursion in logic

- **Axioms**: sentences that are true by definition
- $\Phi \vdash \psi$: sentence ψ is a logical consequence of sentences in set Φ
- **Theory**: set of sentences T containing set of axioms A 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
- Suppose T is a theory that can define the natural numbers
- **Recursive definition**: let γ be defined as $T \nvdash \gamma$

Gödel's theorem

- Show that if T is consistent, then it cannot be complete
- Assume T is consistent, and aim to show that there exists a true sentence which is not in T
- Consider γ: by *tertium non datur*, exactly one sentence in $\{\gamma, \neg\gamma\}$ is true
- Aim to show that neither is in T
- Is $\gamma \in T$? If so, then $T \vdash \gamma$, which means that $T \vdash (T \nvdash \gamma)$, i.e. $T \nvdash \gamma$, i.e. $\gamma \notin T$ (contradiction)
- Is $\neg\gamma \in T$? If so, then $T \vdash \neg\gamma$, i.e. $T \vdash \neg(T \nvdash \gamma)$, 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 that T is inconsistent (contradiction)
- Hence T is incomplete

Does this recursion terminate?

- Not immediately evident that the recursive definition $T \not\vdash \gamma$ has a “base case”
- The most difficult part of Gödel’s proof is to encode all the logic he needed for his argument within positive integers
- In particular, he was able to provide a “finiteness proof” for his recursive definition

End of Lecture 3