

# *Understanding Lifetimes*

Rnic / H.-S. Zheng

*May 25, 2019*

# Outline

Lifetimes Concepts

Phases of the NLL

# Outline

## Lifetimes Concepts

Where are the lifetimes from

What is the lifetimes

## Phases of the NLL

# Lifetimes

```
{  
  let r;                                // -----+-- 'a  
  {                                     // |  
    let x = 5;                          // -+-- 'b |  
    r = &x;                             // | |  
  }                                     // -+ |  
                                     // |  
  println!("r: {}", r);                 // |  
}
```



?



```
{  
    let r;  
  
    {  
        let x = 5;  
  
        r = &x;  
    }  
  
    println!("{}", r);  
}
```

```
{  
  let r;  
  
  {  
    let x = 5;  
  
    r = &x  
    ^^^^^ borrowed value does not live long enough  
  }  
  - `x` dropped here while still borrowed  
  
  println!("{}", r);  
    - borrow later used here  
}
```


*Rust 2018*

```
{  
    let r;  
  
    {  
        let x = 5;  
        r = &x  
        ^^^^^ borrowed value does not live long enough  
    }  
    - `x` dropped here while still borrowed  
  
    println!("{}", r);  
}
```

- borrowed value needs to live until here

*Rust 2015*

```
{  
  let r;  
  
  {  
    let x = 5;  
  
    r = &x;  
  
    println!("{}", r);  
  }  
}
```





```
{  
  let r;  
  
  {  
    let x = 5;  
  
    r = &x;  
  
    println!("{}", r);  
  }  
}
```

*move to*

Rust 2018 

```
{  
  let r;  
  {  
    let x = 5;
```

Rust 2015



```
    r = &x  
    ^^^^^^ borrowed value does not live long enough  
    println!("{}", r);
```

```
  }  
  - `x` dropped here while still borrowed
```

```
}  
- borrowed value needs to live until here
```

# Where are the Lifetimes from?

# Where are the Lifetimes from?

Borrow

# Where are the Lifetimes from?

## Borrow

```
let foo = &bar;
```

# Where are the Lifetimes from?

## Borrow

```
let foo:&'foo T = &'bar bar;
```

# Where are the Lifetimes from?

## Borrow

```
let foo:&'foo T = &'bar bar;
```

# Where are the Lifetimes from?

## Borrow

```
let foo:&'foo T;  
foo = &'bar bar;
```



# Where are the Lifetimes from?

## Borrow

```
let foo:&'foo T;  
  
foo = &'bar bar;
```

A **Borrow** will generate a **reference**,  
and this reference will be tagged with **a Lifetimes**

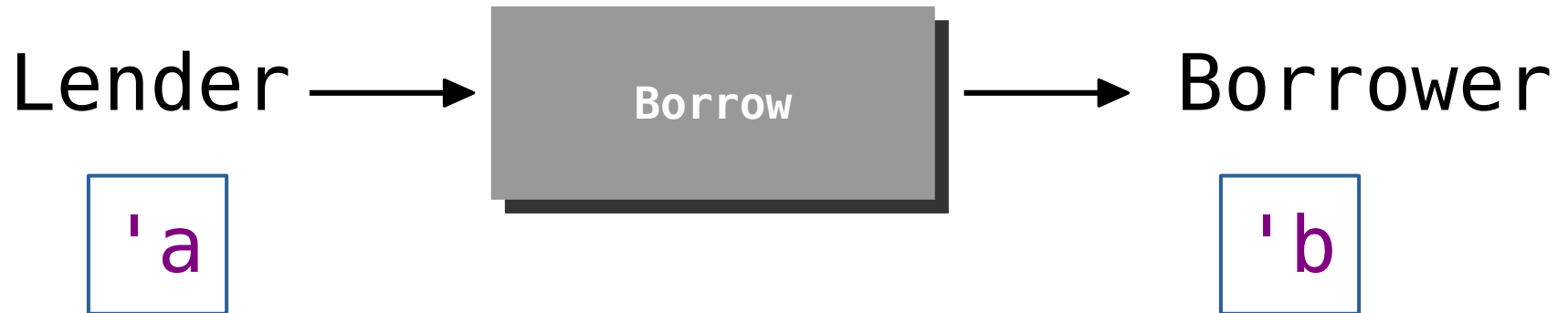
# Subtyping

- **Notation** :  $'a : 'b$
- **Meaning** : Lifetimes  $'a$  is outlives Lifetimes  $'b$

# Subtyping

- **Notation** : 'a : 'b

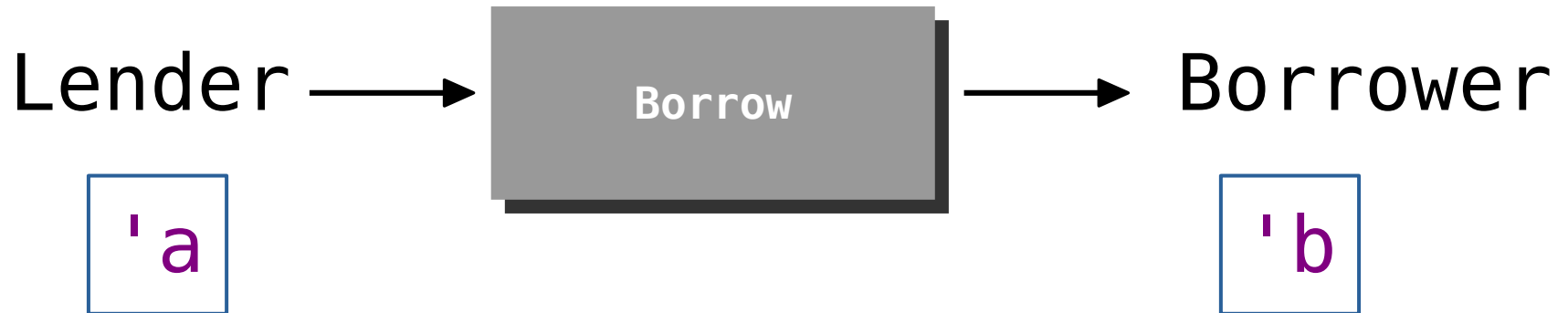
- **Meaning** : Lifetimes 'a is outlives Lifetimes 'b



# Subtyping

- **Notation** :  $'a : 'b$

- **Meaning** : Lifetimes  $'a$  is outlives Lifetimes  $'b$



- **Example** : `let p: &'p T = &'foo foo;`

Then we say:  $'foo : 'p$

# What is Lifetimes ?

**- Definition:**

# What is Lifetimes ?

- **Definition:** set of points on CFG (control flow graph)

# What is Lifetimes ?

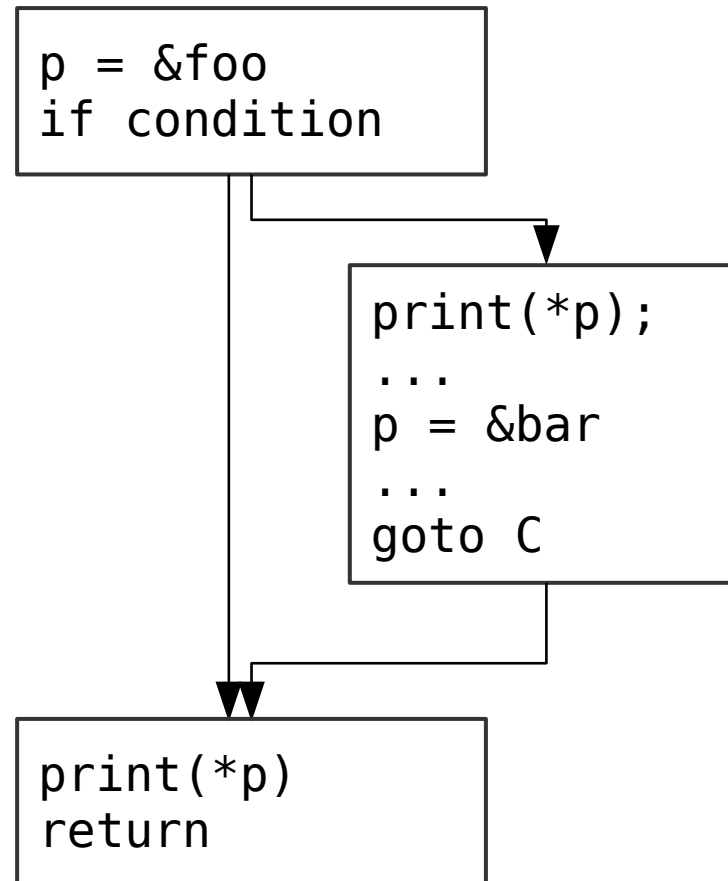
- **Definition:** set of points on CFG (control flow graph)

```
let foo: T = ...;
let bar: T = ...;

let mut p = &foo;

if condition {
    print(*p);
    p = &bar;
}

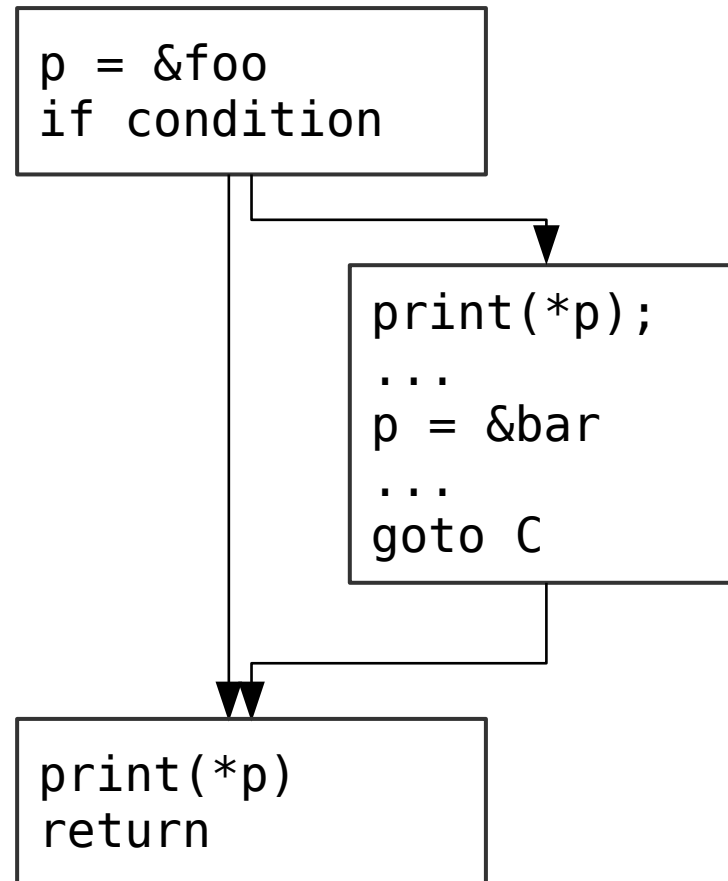
print(*p);
```



# What is Lifetimes ?

- **Definition:** set of points on CFG (control flow graph)

```
let foo: T = ...;  
let bar: T = ...;  
  
let mut p = &'foo foo;  
  
if condition {  
    print(*p);  
    p = &bar;  
}  
  
print(*p);
```





# What is Lifetimes ?

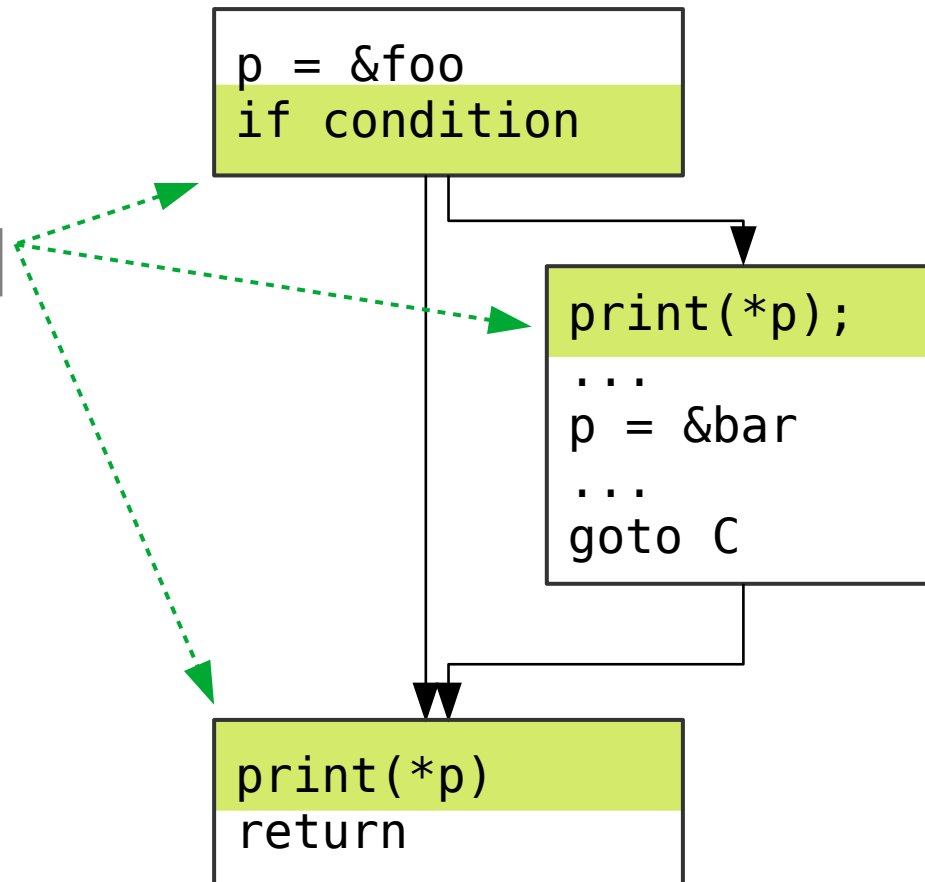
- **Definition:** set of points on CFG (control flow graph)

```
let foo: T = ...;  
let bar: T = ...;
```

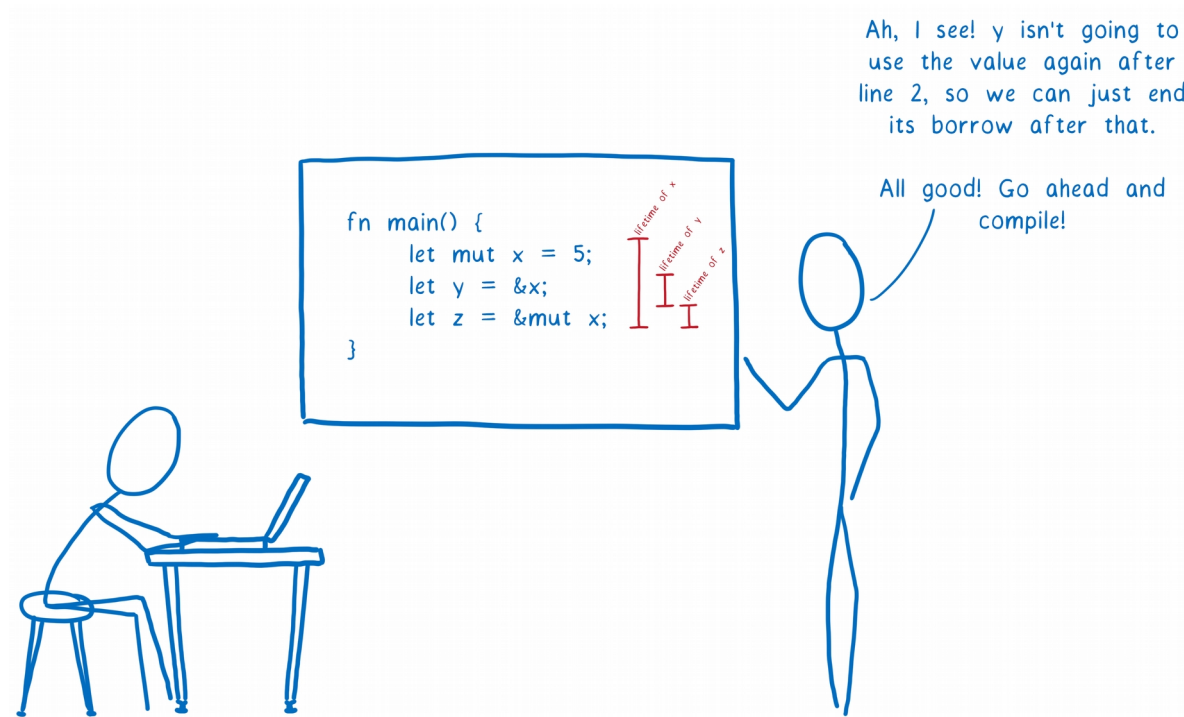
```
let mut p = &'foo foo;
```

```
if condition {  
    print(*p);  
    p = &bar;  
}
```

```
print(*p);
```



# NLL



```
let mut s = "hello".to_string();  
let mut c = || s += " world";  
c();  
println!("{}", s);
```

```
let mut s = "hello".to_string();
```

```
let mut c = || s += ' world';
```

Borrow as **mut**

```
c();
```

```
println!("{}", s);
```

Borrow as **immut**

```

fn process_or_default(map: &mut HashMap<usize, String>,
                      key: usize)
{
    match map.get_mut(&key) {
        Some(value) => {
            process(value);
            return;
        }

        None => {
            map.insert(key, V::default());
        }
    }
}

```

```
fn process_or_default(map: &mut HashMap<usize, String>,
                      key: usize)
{
    match map.get_mut(key) {
        Some(value) => {
            process(value);
            return;
        }
        None => {
            map.insert(key, V::default());
        }
    }
}
```

Borrow as **mut**

Borrow as **mut**

```
fn process_or_default(map: &mut HashMap<usize, String>,
                      key: usize)
{
    match map.get_mut(key) {
        Some(value) => {
            process(value);
            return;
        }
        None => {
            map.insert(key, V::default());
        }
    }
}
```

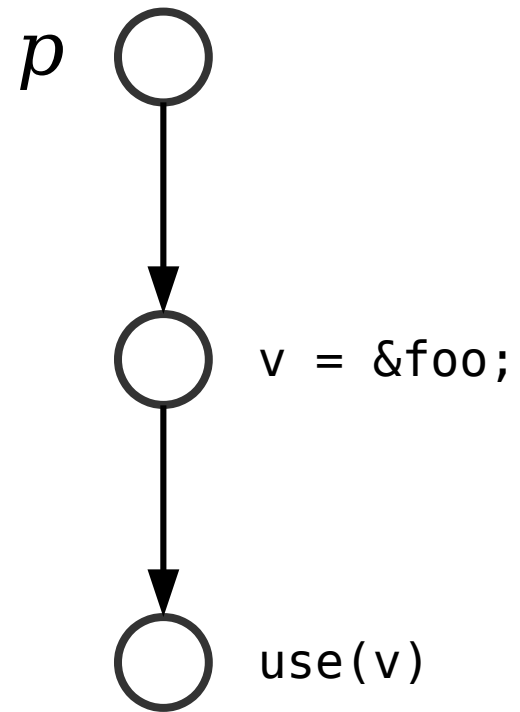
Borrow as **mut**Borrow as **mut**

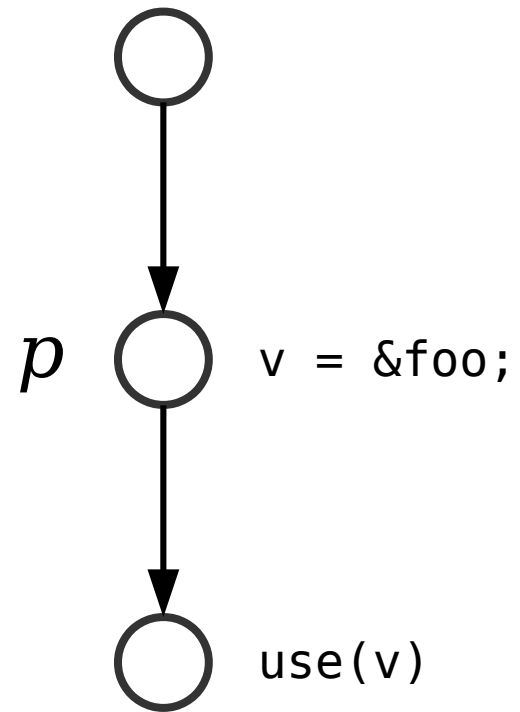
# Liveness

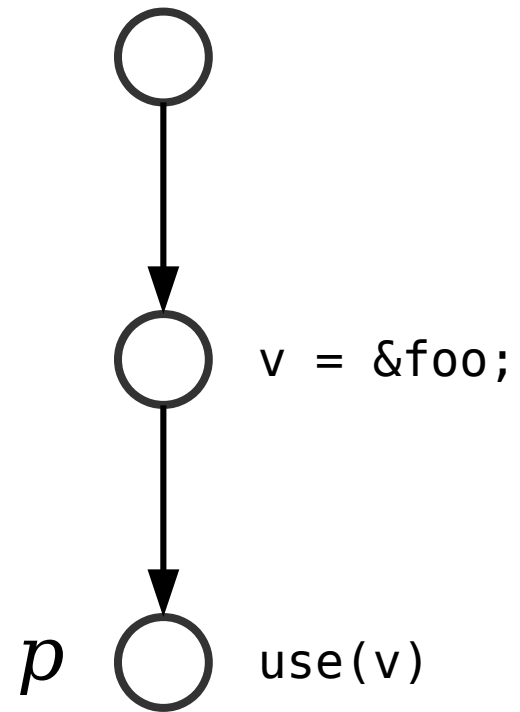
## - **Meaning** :

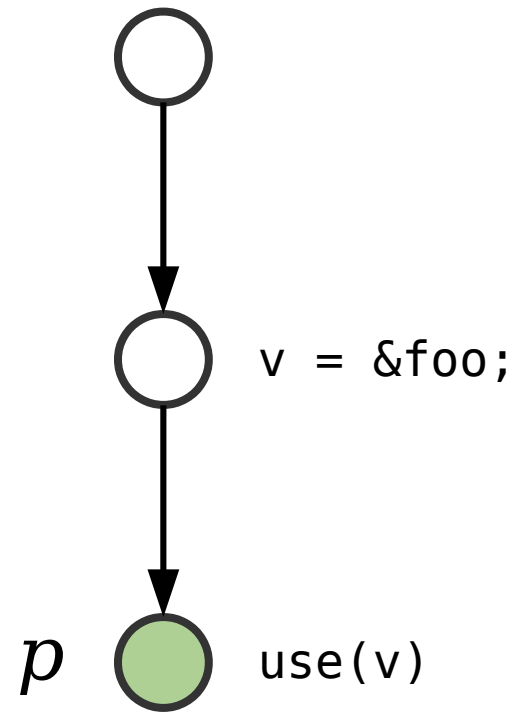
a variable  $v$  is live at point  $p$  if and only if there exists a path in CFG from  $p$  to a use of  $v$  along which  $v$  is not redefined.

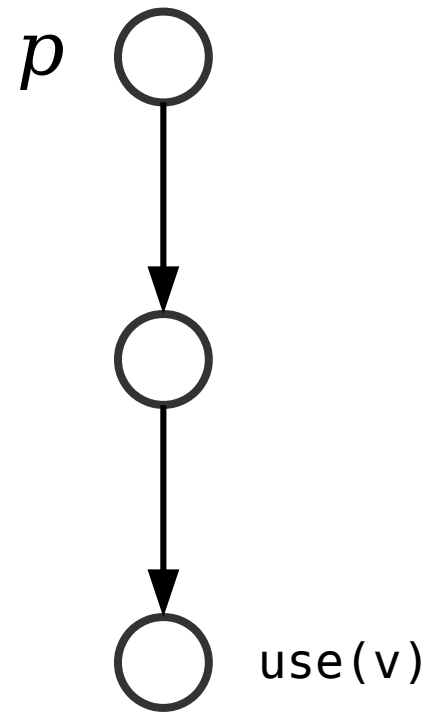


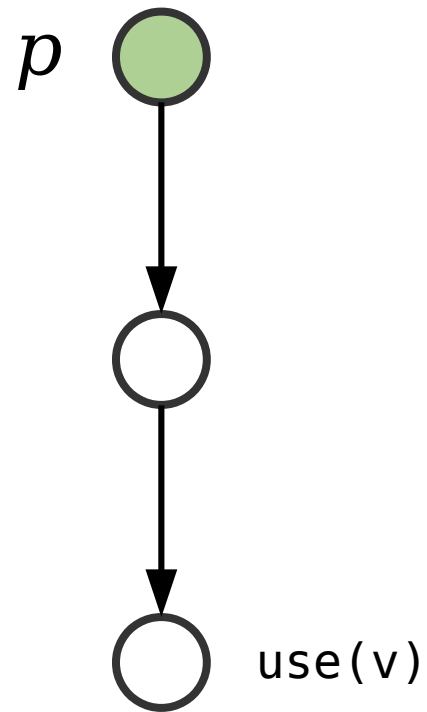


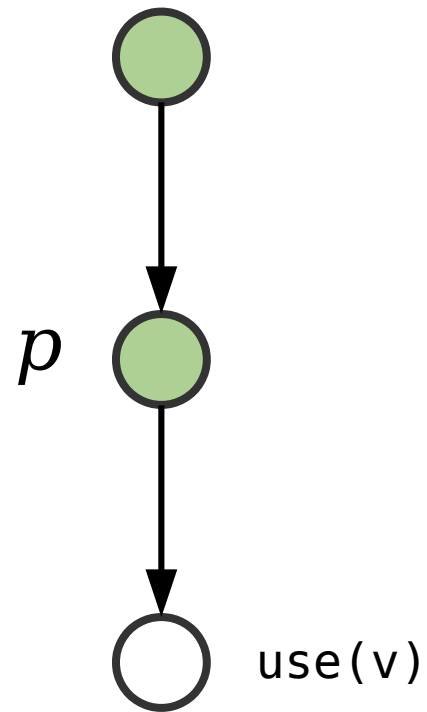


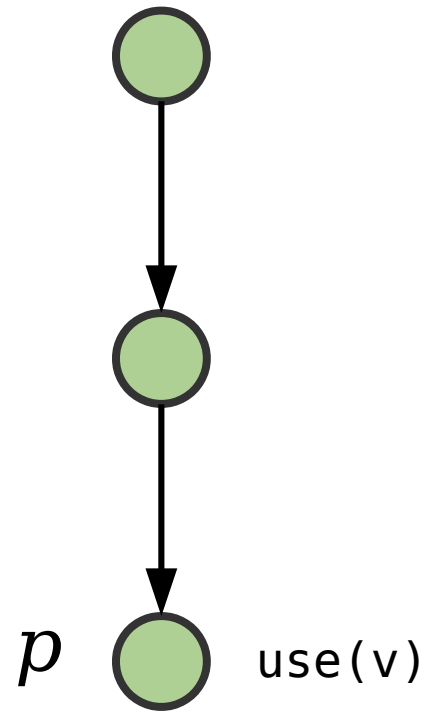










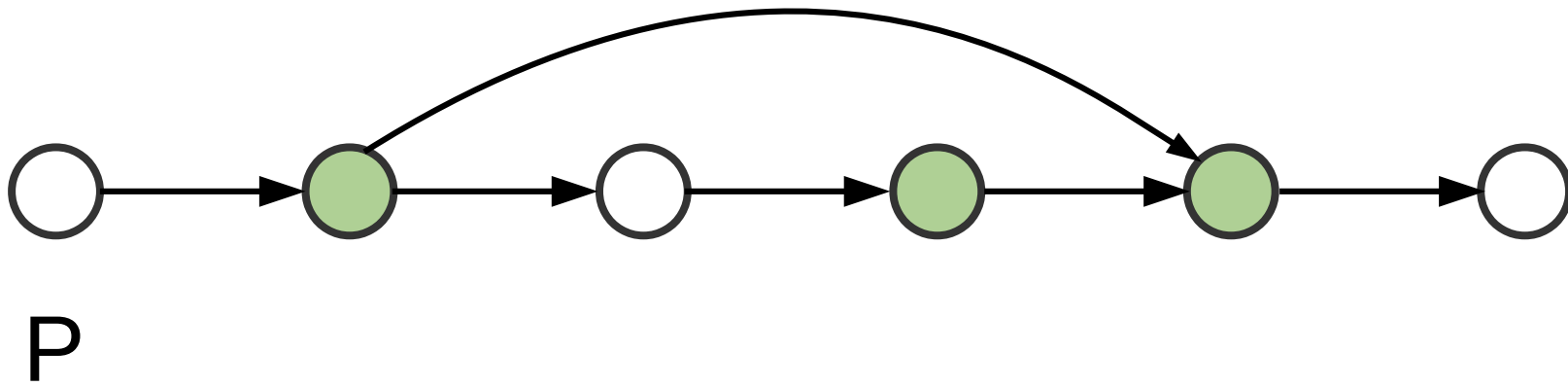
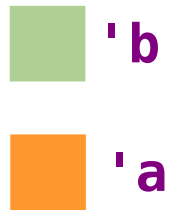




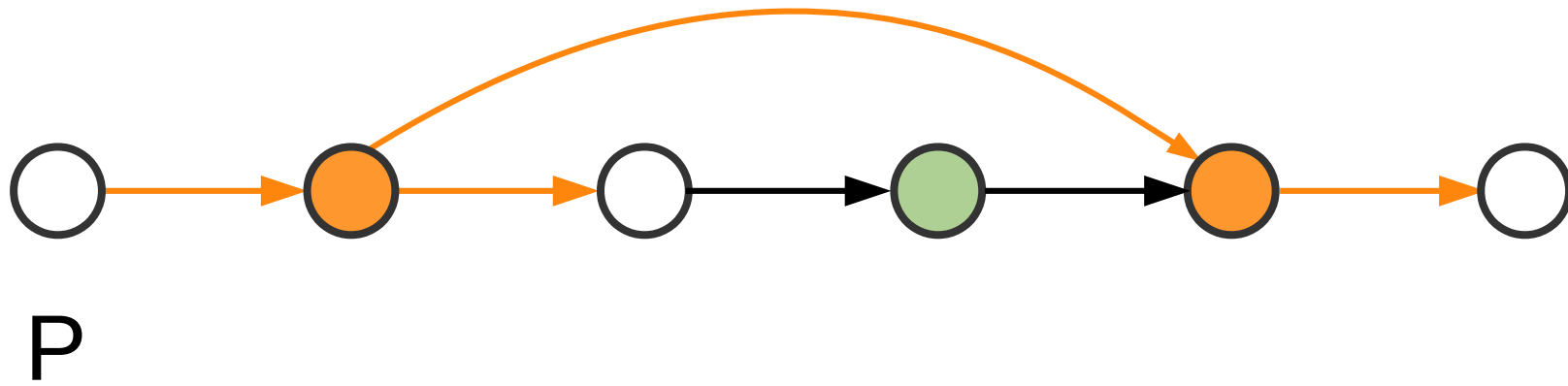
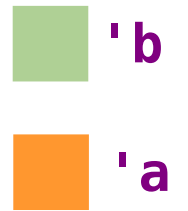
# Location-aware Subtyping

- **Notation** :  $('a : 'b) @ P$
- **Meaning** : Lifetimes  $'a$  must include all points in  $'b$   
that are reachable from location  $P$

('a : 'b) @ P



('a : 'b) @ P



# Phases of NLL

1. Building Liveness and Constraints



2. Solving Constraints



3. Compute Loans in Scope



4. Check action and report Error

# Phases of NLL

1. Building Liveness and Constraints



2. Solving Constraints



3. Compute Loans in Scope



4. Check action and report Error

# Phases of NLL

( 'a : 'b ) @ P

1. Building Liveness and Constraints



2. Solving Constraints



3. Compute Loans in Scope



4. Check action and report Error

# Phases of NLL

1. Building Liveness and Constraints



2. Solving Constraints



3. Compute Loans in Scope



4. Check action and report Error

# Phases of NLL

1. Building Liveness and Constraints



a fixed-point iteration algorithm

2. Solving Constraints



3. Compute Loans in Scope



4. Check action and report Error



# Phases of NLL

1. Building Liveness and Constraints



2. Solving Constraints



3. Compute Loans in Scope



4. Check action and report Error

# Phases of NLL

1. Building Liveness and Constraints



2. Solving Constraints



**Borrow expressions** ←

3. Compute **Loans** in Scope



4. Check action and report Error

# Phases of NLL

1. Building Liveness and Constraints



2. Solving Constraints

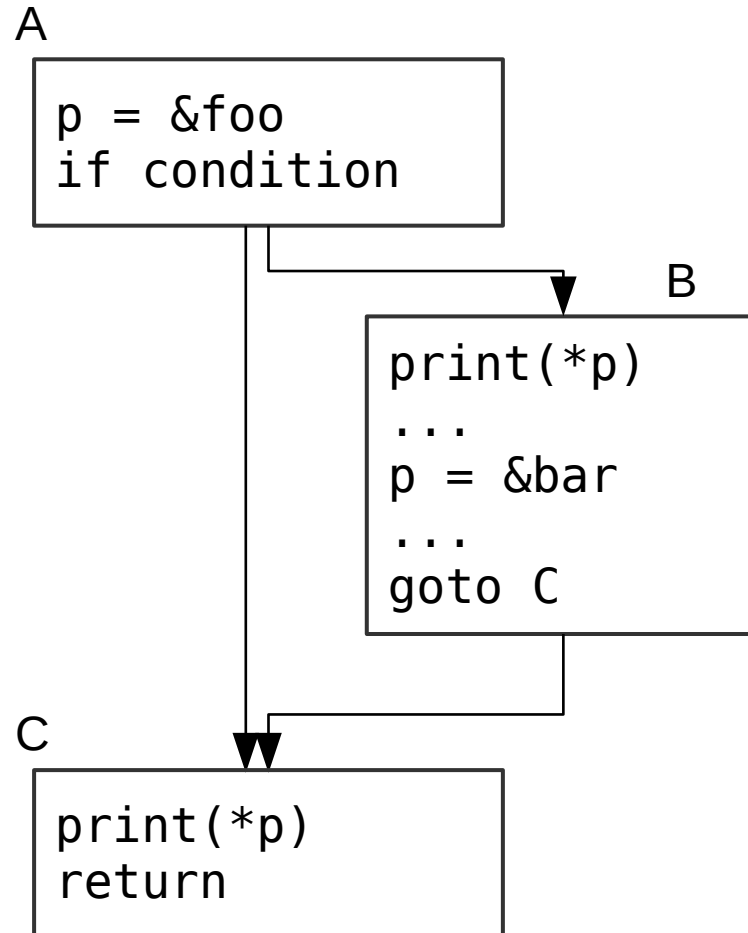


3. Compute Loans in Scope



4. Check action and report Error

# Liveness

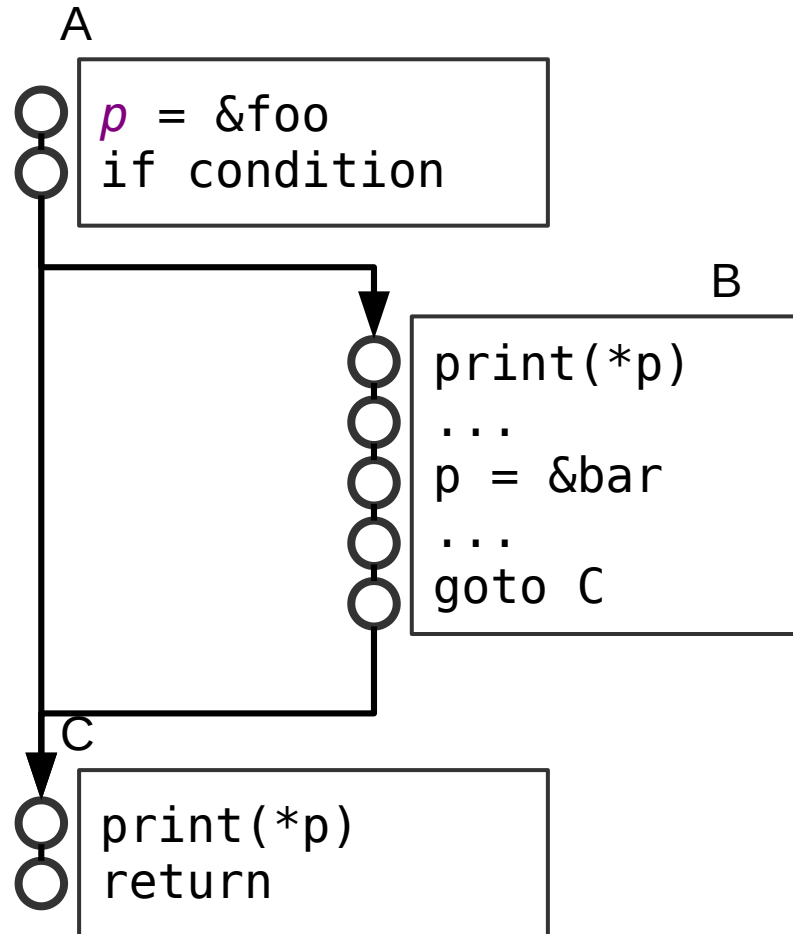


p : { }

foo : { }

bar : { }

# Liveness

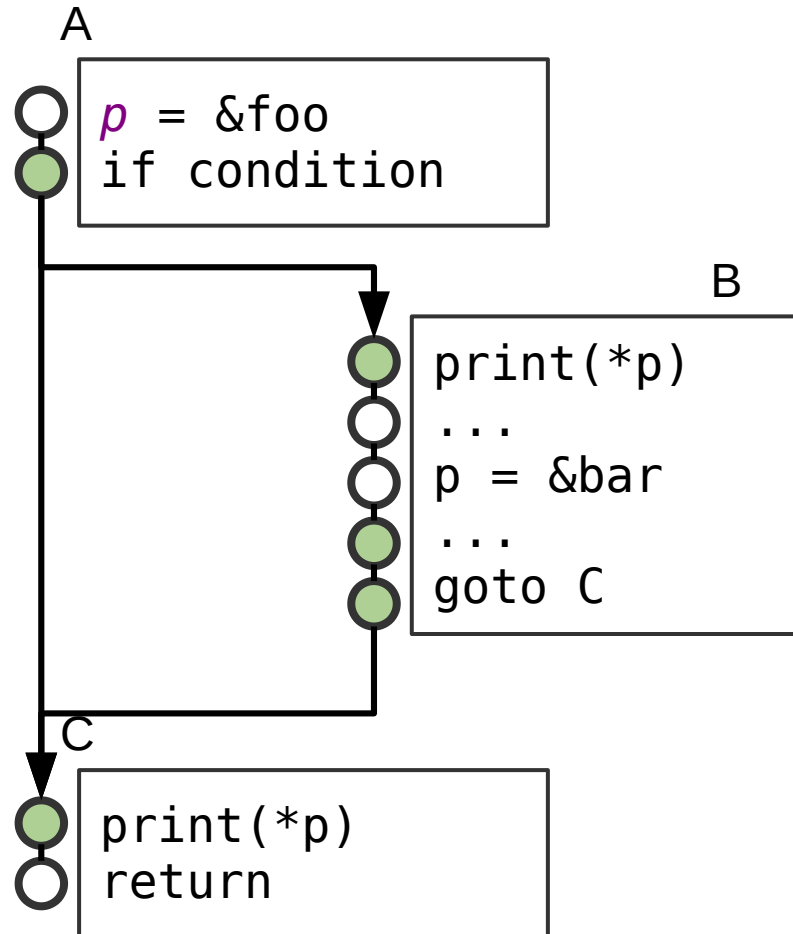


`p : { }`

`foo : { }`

`bar : { }`

# Liveness

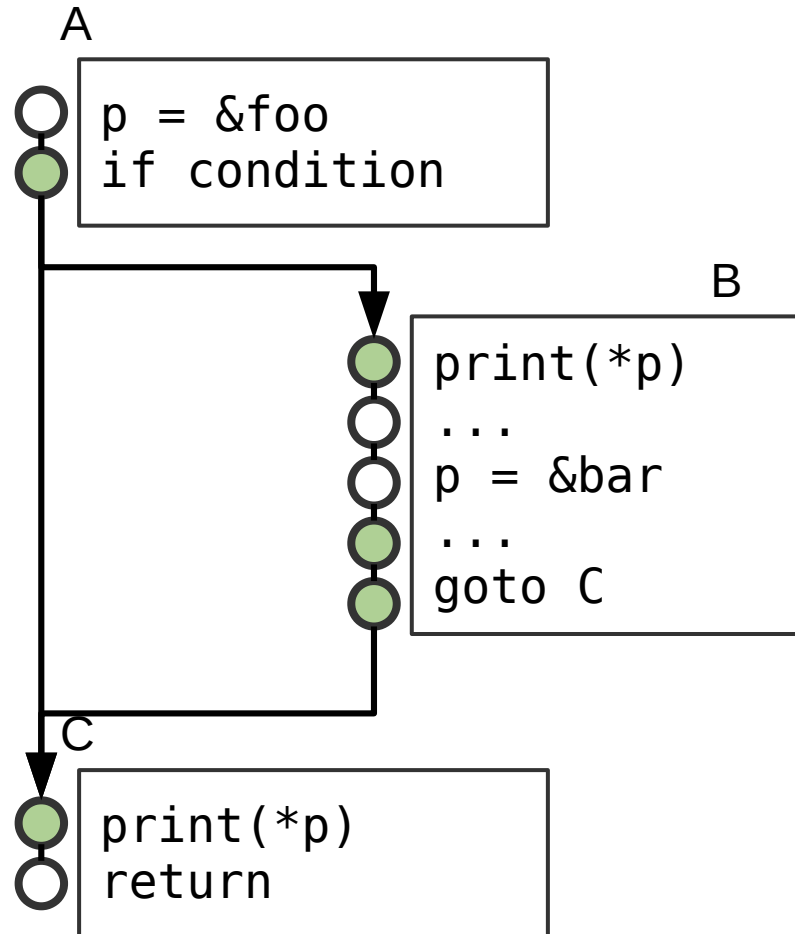


`p : { A/1, B/0, B/3, B/4, C/0 }`

`foo : { }`

`bar : { }`

# Constraints

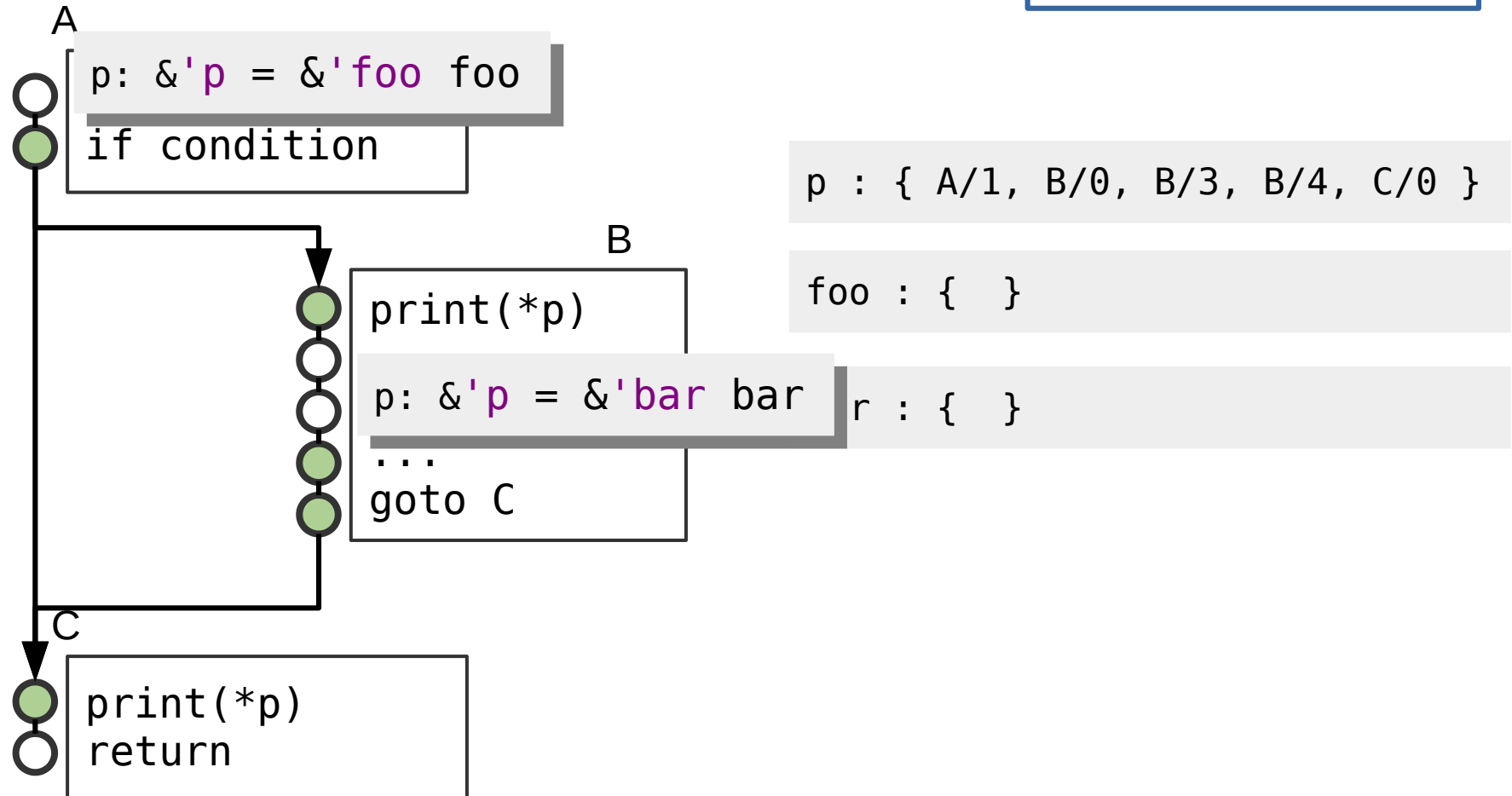


`p : { A/1, B/0, B/3, B/4, C/0 }`

`foo : { }`

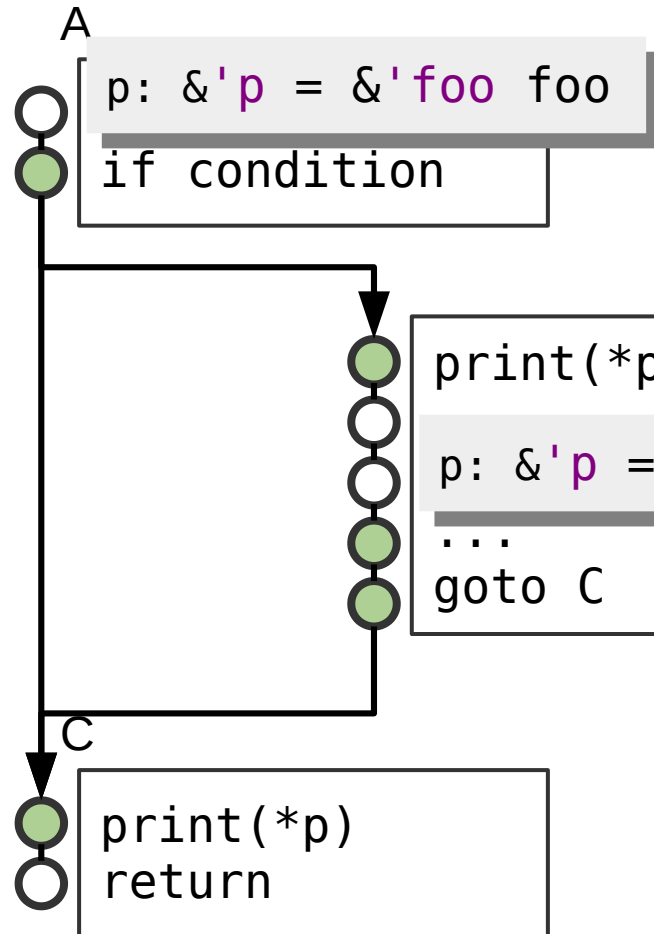
`bar : { }`

# Constraints





# Constraints



`p : { A/1, B/0, B/3, B/4, C/0 }`

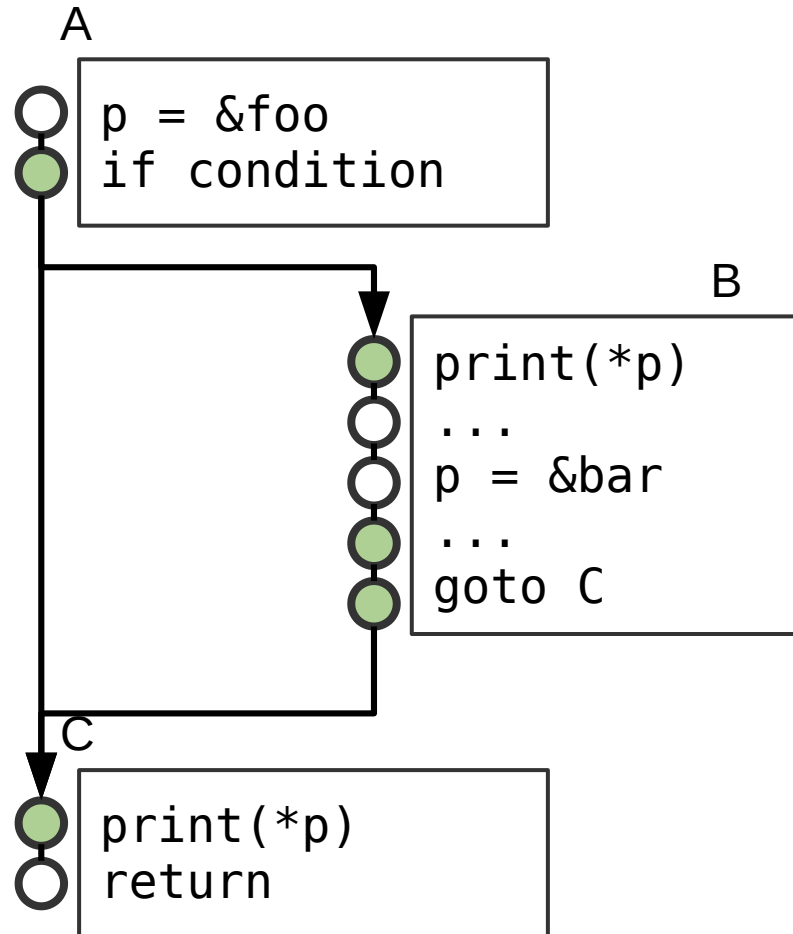
`foo : { }`

`r : { }`

1. `'foo : 'p @ A/0`

2. `'bar : 'p @ B/2`

# Constraints



`p : { A/1, B/0, B/3, B/4, C/0 }`

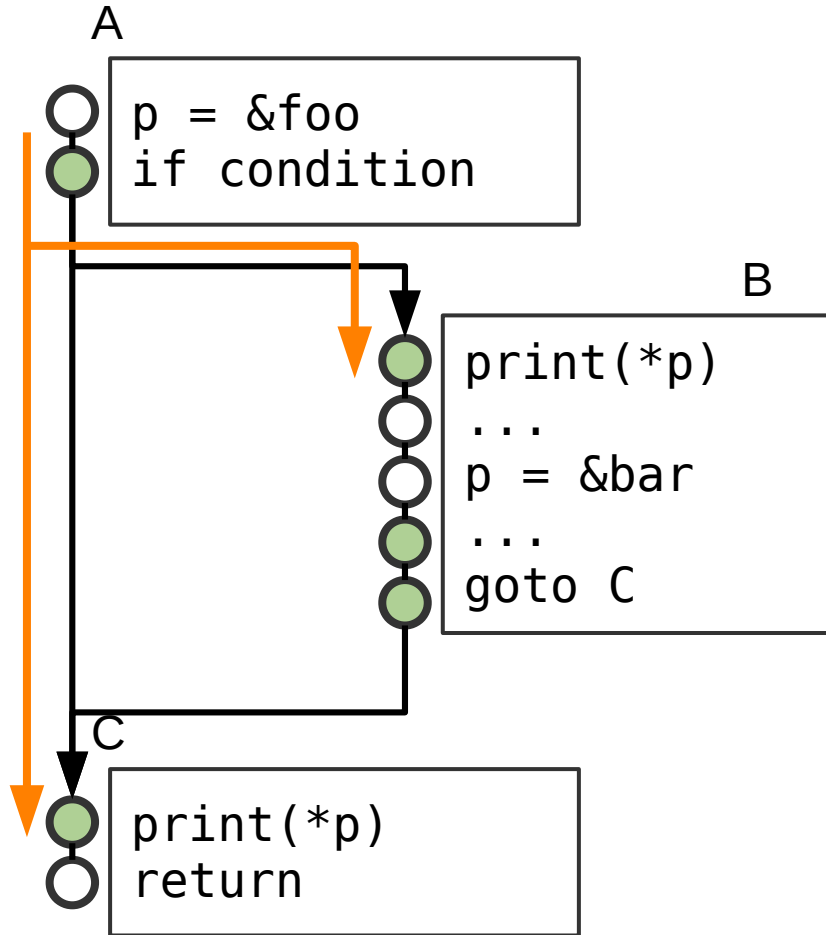
`foo : { }`

`bar : { }`

1. `'foo : 'p @ A/0`

2. `'bar : 'p @ B/2`

# Solving Constraints



`p : { A/1, B/0, B/3, B/4, C/0 }`

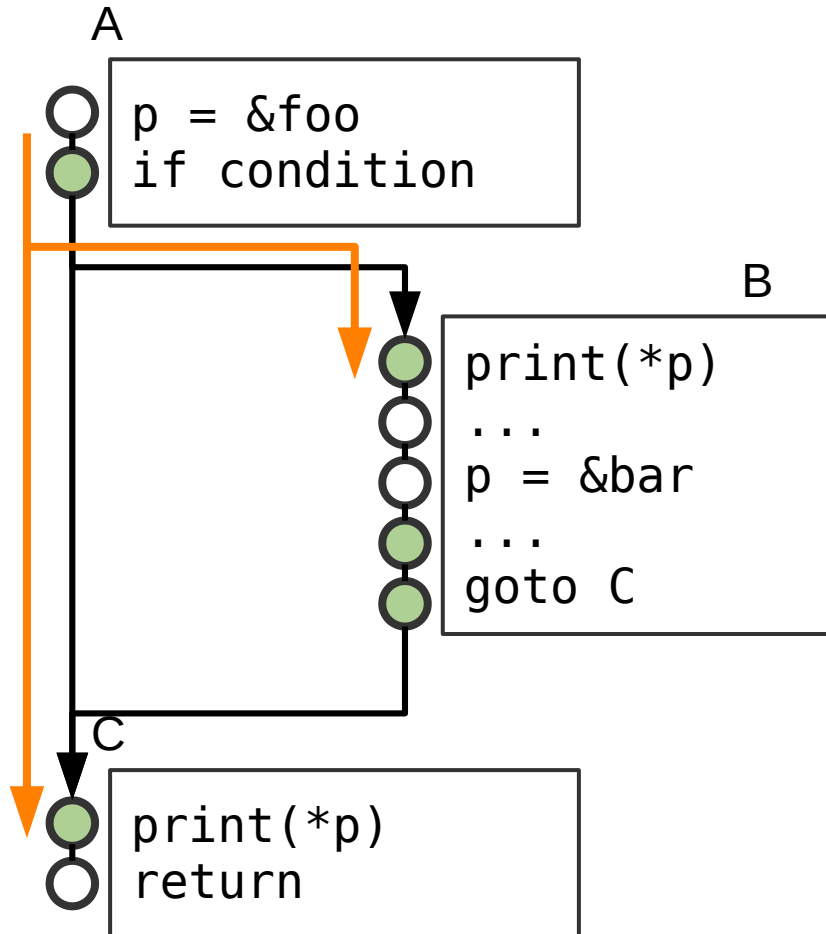
`foo : { }`

`bar : { }`

1. `'foo : 'p @ A/0`

2. `'bar : 'p @ B/2`

# Solving Constraints



`p : { A/1, B/0, B/3, B/4, C/0 }`

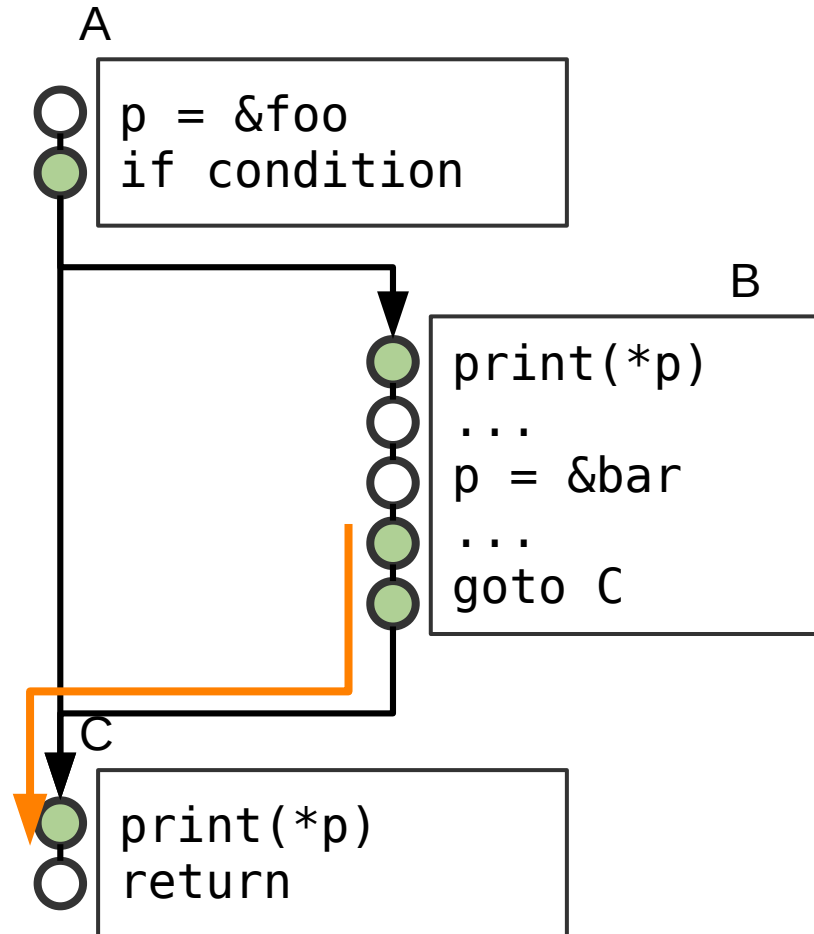
`foo : { A/1, B/0, C/0 }`

`bar : { }`

1. `'foo : 'p @ A/0`

2. `'bar : 'p @ B/2`

# Solving Constraints



`p : { A/1, B/0, B/3, B/4, C/0 }`

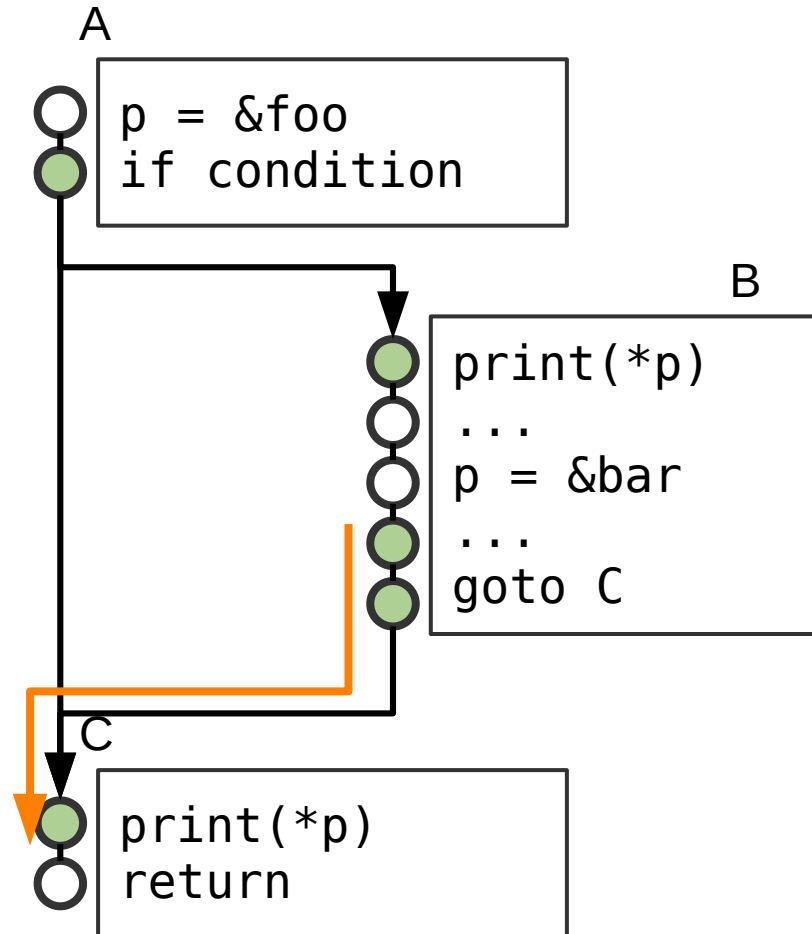
`foo : { A/1, B/0, C/0 }`

`bar : { }`

1. `'foo : 'p @ A/0`

2. `'bar : 'p @ B/2`

# Solving Constraints



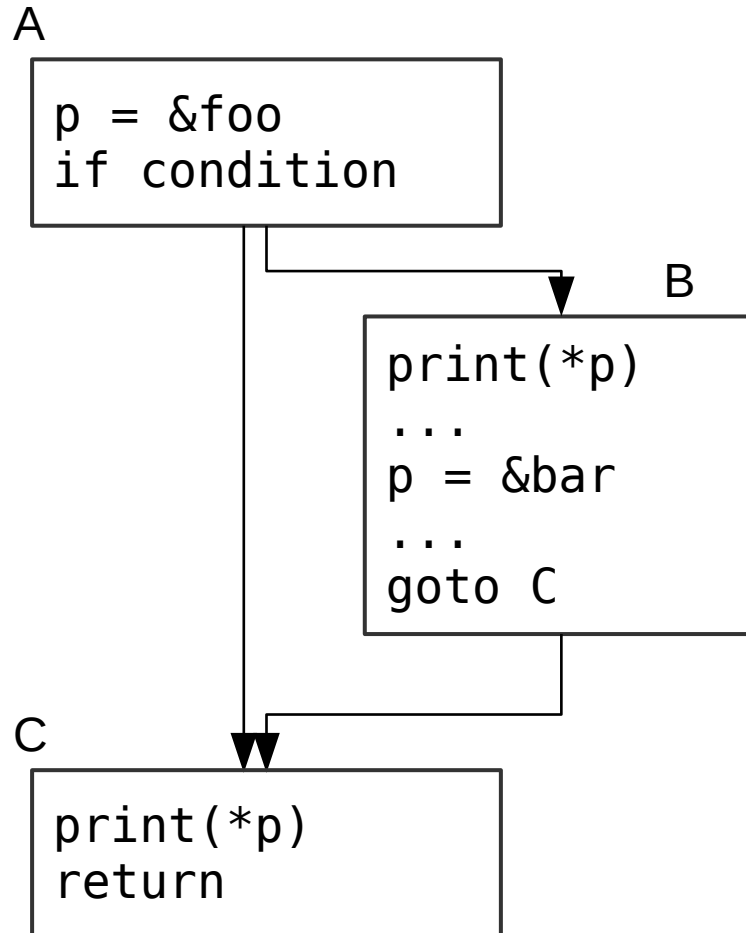
`p : { A/1, B/0, B/3, B/4, C/0 }`

`foo : { A/1, B/0, C/0 }`

`bar : { B/3, B/4, C/0 }`

1. `'foo : 'p @ A/0`

2. `'bar : 'p @ B/2`



## Lifetimes:

p : { A/1, B/0, B/3, B/4, C/0 }

foo : { A/1, B/0, C/0 }

bar : { B/3, B/4, C/0 }

# Compute Loans in Scope

**loans:** a set of borrow expressions



# Compute Loans in Scope

**loans:** a set of borrow expressions

**a Loan:**

```
A/0 p = &'foo foo;
```

# Compute Loans in Scope

**loans:** a set of borrow expressions

**a Loan:**

A/0 p = &'foo foo;

```
Loan L0
{
  point:  A/4,
  path:   foo,
  kind:   shared
  region: 'foo {
    A/1, B/0, C/0
  }
}
```

# Compute Loans in Scope

**loans:** a set of borrow expressions

Borrow checker will compute loans at each point via fixed-point dataflow computation with transfer function:

# Compute Loans in Scope

**loans:** a set of borrow expressions

Borrow checker will compute loans at each point via **fixed-point dataflow computation** with transfer function:

-Example: Dataflow equation of Live variable analysis:

$$Live_{in}(s) = Gen(s) \cup (Live_{out}(s) \cap \overline{Kill(s)})$$

# Compute Loans in Scope

**loans:** a set of borrow expressions

Borrow checker will compute loans at each point via fixed-point dataflow computation with **transfer function**:

## Transfer function:

Kill Li @ P    If     $P \notin \text{Li.region}$

Gen   Li @ P    If    Borrow expression occur @ P

Kill Li @ P    If     $LV \in \text{Li.path}$

```
let list: &mut List<T> = &mut a;
```

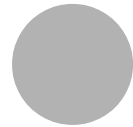
```
let v = &mut (*list).value;
```

```
list = &mut b; // assignment
```

```
use(v)
```

```
use(list)
```

```
let list: &mut List<T> = &mut a;
```



```
let v = &mut (*list).value;
```

```
list = &mut b; // assignment
```



```
use(v)
```



```
use(list)
```



```
let list: &mut List<T> = &mut a;
```



```
let v = &mut (*list).value;
```



```
list = &mut b; // assignment
```

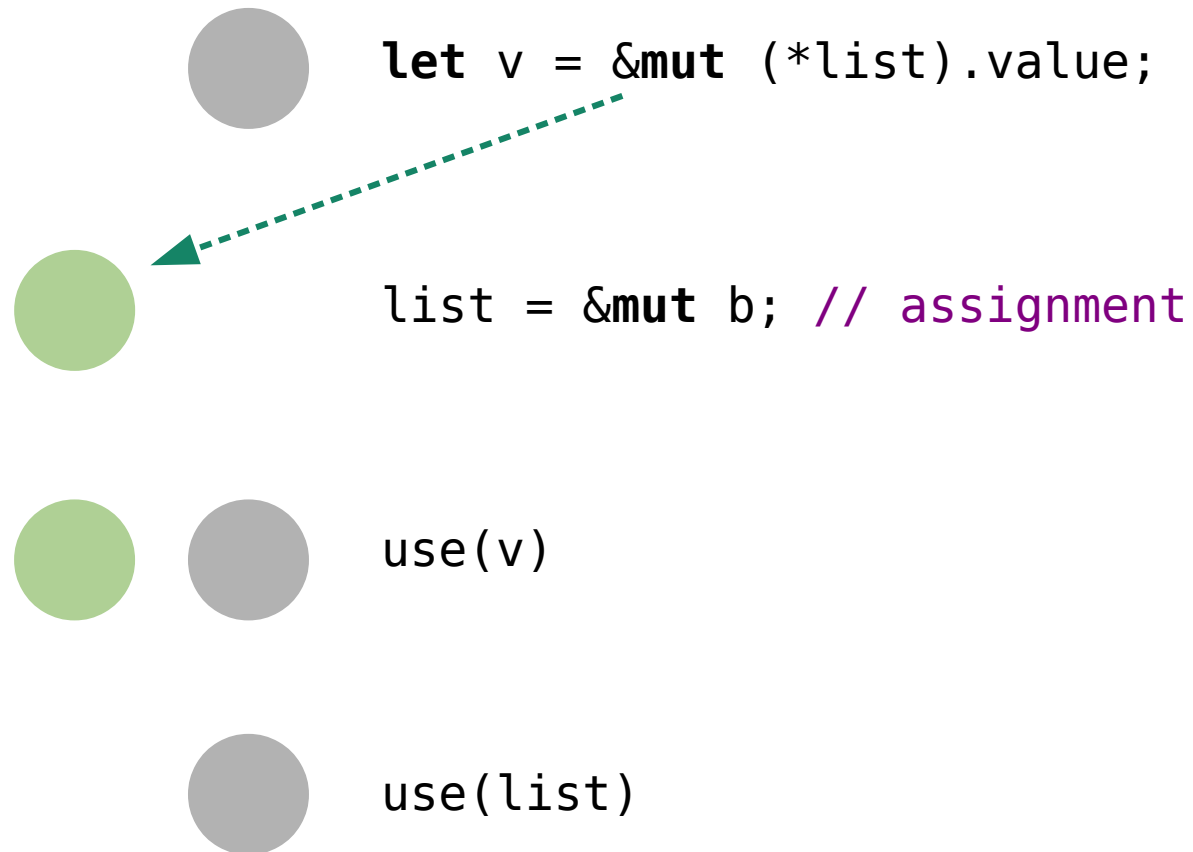


```
use(v)
```



```
use(list)
```

```
let list: &mut List<T> = &mut a;
```



```
let list: &mut List<T> = &mut a;
```



```
let v = &mut (*list).value;
```



```
list = &mut b; // assignment
```

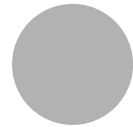


```
use(v)
```



```
use(list)
```

```
let list: &mut List<T> = &mut a;
```



```
let v = &mut (*list).value;
```



```
list = &mut b; // assignment
```



```
use(v)
```



```
use(list)
```

```
let list: &mut List<T> = &mut a;
```



```
let v = &mut (*list).value;
```

gen L1
--------



```
list = &mut b; // assignment
```

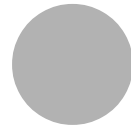


```
use(v)
```



```
use(list)
```

```
let list: &mut List<T> = &mut a;
```



```
let v = &mut (*list).value;
```

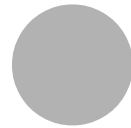
gen	L1
-----	----



```
list = &mut b; // assignment
```

kill	L1
------	----

gen	L2
-----	----



```
use(v)
```



```
use(list)
```

```

Loan L1
{
  point:  A/1,
  path:  {
    (*list).value ,
    *list,
    list
  },
  kind:   shared
  region: 'c {
    A/2, A/3
  }
}

```

```
let list: &mut List<T> = &mut a;
```

```
let v = &mut (*list).value;
```

gen L1

```
list = &mut b; // assignment
```

kill L1

gen L2



use(v)



use(list)

```

Loan L1
{
  point:  A/1,
  path:  {
    (*list).value ,
    *list,
    list
  },
  kind:   shared
  region: 'c {
    A/2, A/3
  }
}

```

```
let list: &mut List<T> = &mut a;
```

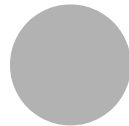
```
let v = &mut (*list).value;
```

gen L1

```
list = &mut b; // assignment
```

kill L1

gen L2

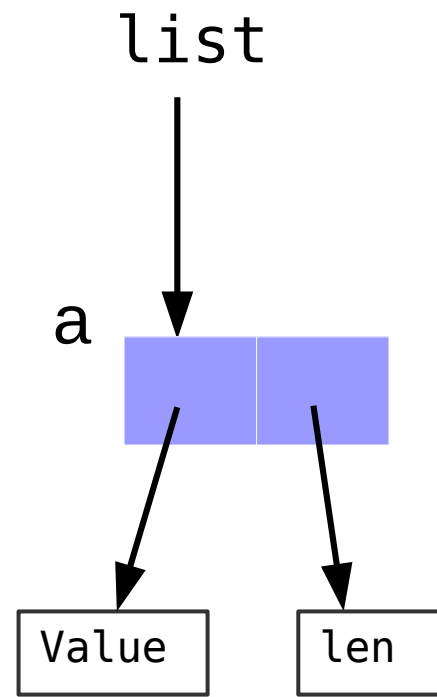


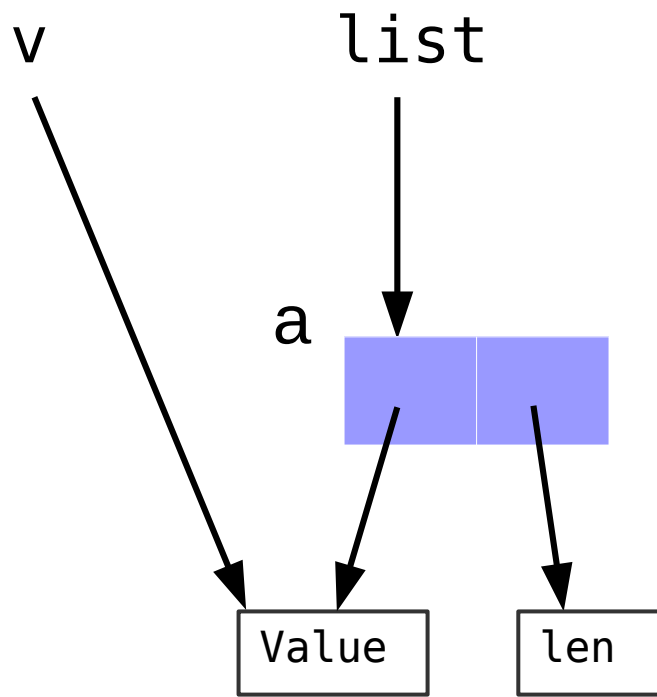
use(v)

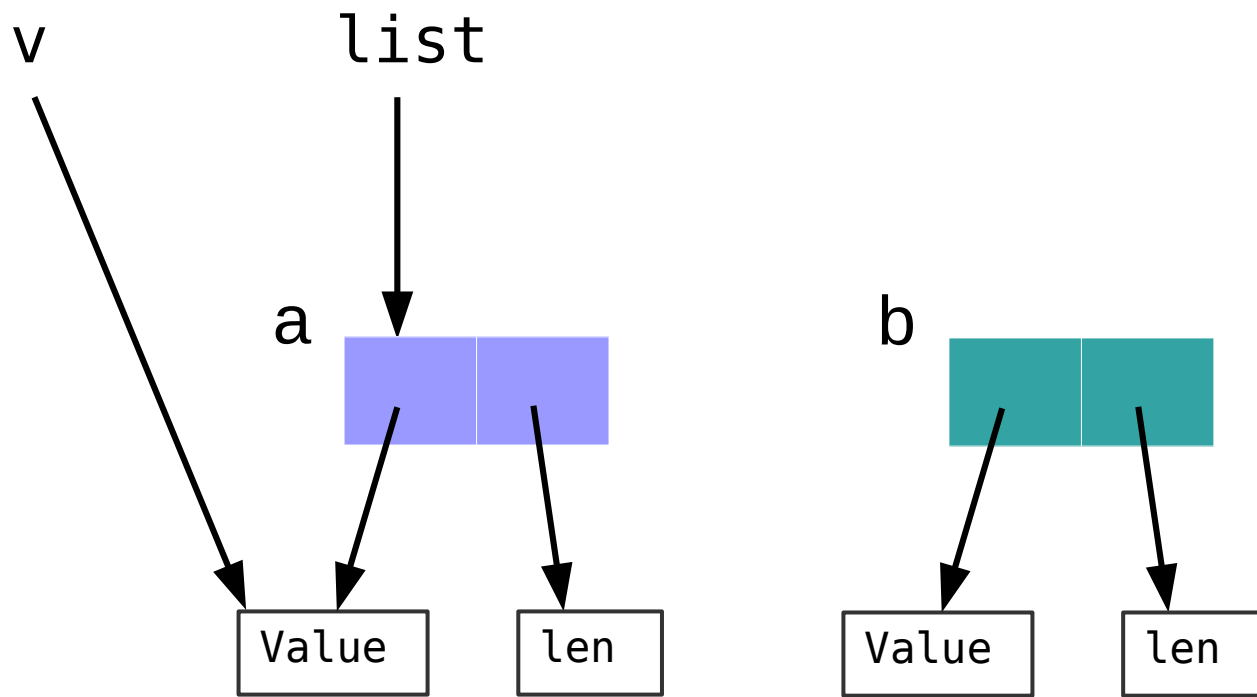


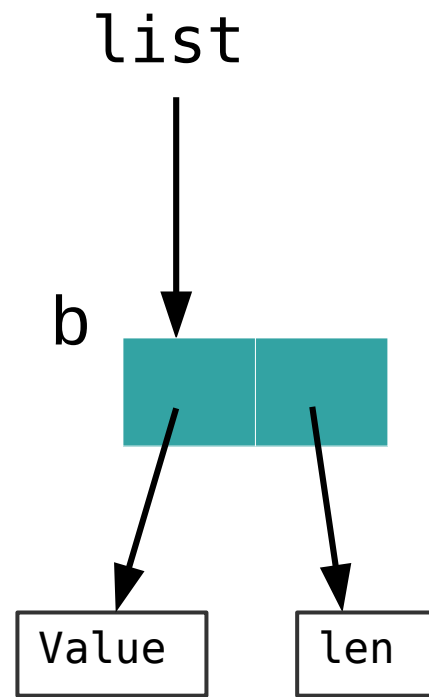
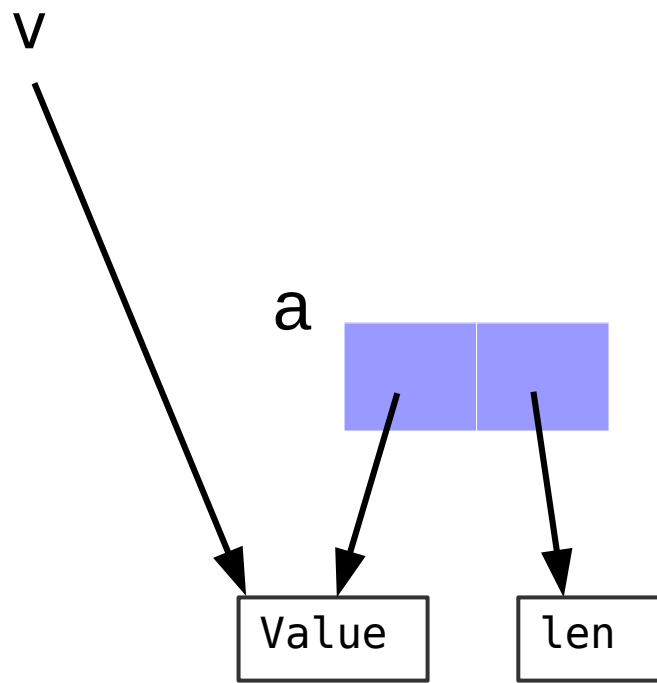
use(list)

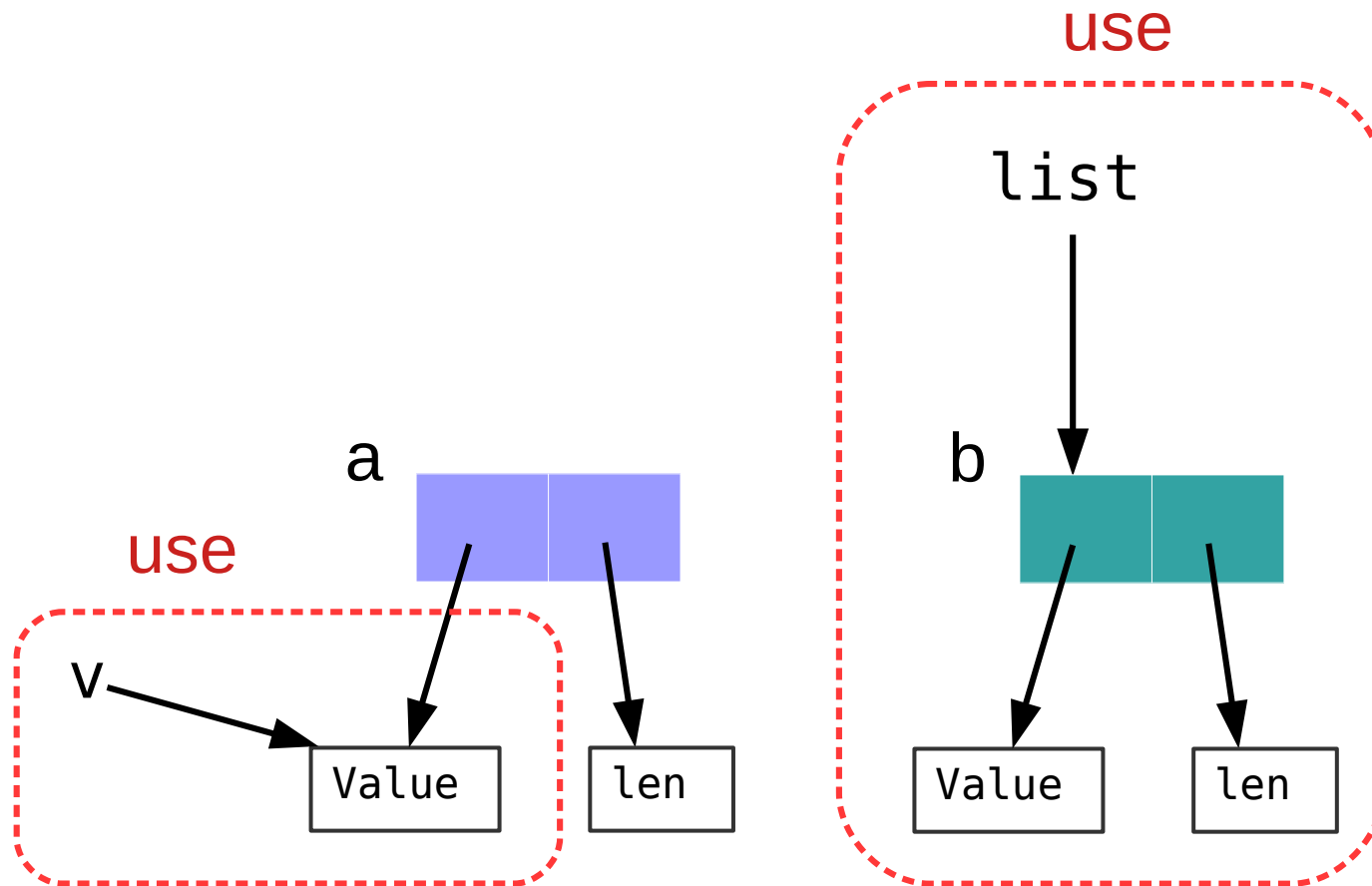












*Another Example*

```

struct Thing;

impl Thing {
    fn maybe_next(&mut self) -> Option<&mut Self> {
        None
    }
}

fn main() {
    let mut temp = &mut Thing;
    loop {
        match temp.maybe_next() {

            Some(v) => { temp = v; }

            None => { }
        }
    }
}

```

```

struct Thing;

impl Thing {
    fn maybe_next(&mut self) -> Option<&mut Self> {
        None
    }
}

fn main() {
    let mut temp = &mut Thing;
    loop {
        match temp.maybe_next() {
            Some(v) => { temp = v; }
            None => { }
        }
    }
}

```

← mutable borrow starts here in previous iteration of loop



```


struct Thing;

impl Thing {
    fn maybe_next(&mut self) -> Option<&mut Self> {
        None
    }
}

fn main() {
    let mut temp = &mut Thing;
    loop {
        match temp.maybe_next()
            Some(v) => { temp = v; }

            None => { }
        }
    }
}

```



```

struct Thing;

impl Thing {
    fn maybe_next(&mut self) -> Option<&mut Self> {
        None
    }
}

fn main() {
    let mut temp = &mut Thing;
    loop {
        match temp.maybe_next()
            Some(v) => { temp = v; }

            None => { }
        }
    }
}

```

gen Loan: L0

```

struct Thing;

impl Thing {
    fn maybe_next(&mut self) -> Option<&mut Self> {
        None
    }
}

fn main() {
    let mut temp = &mut Thing;
    loop {
        match temp.maybe_next() {
            Some(v) => { temp = v; }
            None => { }
        }
    }
}

```

gen Loan: L0

```
struct Thing;
```

```
impl Thing {  
    fn maybe_next(&mut self) -> Option<&mut Self> {  
        None  
    }  
}
```

```
fn main() {  
    let mut temp = &mut Thing;  
    loop {  
        match temp.maybe_next() {  
            Some(v) => { temp = v; }  
            None => { }  
        }  
    }  
}
```

path: temp

gen Loan: L0

```
struct Thing;
```

```
impl Thing {  
    fn maybe_next(&mut self) -> Option<&mut Self> {  
        None  
    }  
}
```

```
fn main() {  
    let mut temp = &mut Thing;  
    loop {  
        match temp.maybe_next() {  
            Some(v) => { temp = v; }  
            None => { }  
        }  
    }  
}
```

path: temp

gen Loan: L0

kill L0

```
struct Thing;
```

```
impl Thing {  
    fn maybe_next(&mut self) -> Option<&mut Self> {  
        None  
    }  
}
```

```
fn main() {  
    let mut temp = &mut Thing;  
    loop {  
        match temp.maybe_next() {  
            Some(v) => { temp = v; }  
            None => { }  
        }  
    }  
}
```

path: temp

gen Loan: L0

kill L0

But L0 not be killed in None arm

```

struct Thing;

impl Thing {
    fn maybe_next(&mut self) -> Option<&mut Self> {
        None
    }
}

fn main() {
    let mut temp = &mut Thing;
    loop {
        match temp.maybe_next() {

            Some(v) => { temp = v; }

            None => { break; }

        }
    }
}

```

*Future : Polonius*



```

struct Thing;


impl Thing {
    fn maybe_next(&mut self) -> Option<&mut Self> {
        None
    }
}

fn main() {
    let mut temp = &mut Thing;
    loop {
        match temp.maybe_next() {

            Some(v) => { temp = v; }

            None => { }
        }
    }
}

```

 require relation is dropped

```
struct Thing;
```

```
impl Thing {  
    fn maybe_next(&mut self) -> Option<&mut Self> {  
        None  
    }  
}
```

```
fn main() {  
    let mut temp = &mut Thing;  
    loop {  
        match temp.maybe_next() {  
            Some(v) => { temp = v; }  
            None => { }  
        }  
    }  
}
```

$$\frac{R_1 \text{ live at } P \quad R_1 \text{ require } L \text{ at } P}{L \text{ live at } P}$$

require relation is dropped

```
struct Thing;
```

```
impl Thing {  
    fn maybe_next(&mut self) -> Option<&mut Self> {  
        None  
    }  
}
```

```
fn main() {  
    let mut temp = &mut Thing;  
    loop {  
        match temp.maybe_next() {  
            Some(v) => { temp = v; }  
            None => { }  
        }  
    }  
}
```

$$\frac{R_1 \text{ live at } P \quad R_1 \text{ require } L \text{ at } P}{L \text{ live at } P}$$



So, L will not live at None arm

*Leetcode example N0026*

# Leetcode - N0026

Remove duplicated from sorted array

**Input :**

0, 0, 1, 1, 1, 2, 2, 3, 3, 4
------------------------------

# Leetcode - N0026

Remove duplicated from sorted array

**Input :**

0, 0, 1, 1, 1, 2, 2, 3, 3, 4

**Output :**

0, 1, 2, 3, 4, 2, 2, 3, 3, 4

**Return :** 5

```
int removeDuplicates(vector<int>& nums) {  
    if (nums.empty()) return 0;  
    auto it = nums.begin();  
    int cache = *it;  
    for (auto x : nums) {  
        if (x != cache) {  
            cache = *(++it) = x;  
        }  
    }  
    return it - nums.begin() + 1;  
}
```

```
int removeDuplicates(vector<int>& nums) {
    if (nums.empty()) return 0;
    auto it = nums.begin();
    int cache = *it;
    for (auto x : nums) {
        if (x != cache) {
            cache = *(++it) = x;
        }
    }
    return it - nums.begin() + 1;
}
```

Borrow as mut



```
int removeDuplicates(vector<int>& nums) {
    if (nums.empty()) return 0;
    auto it = nums.begin();
    int cache = *it;
    for (auto x : nums) {
        if (x != cache) {
            cache = *(++it) = x;
        }
    }
    return it - nums.begin() + 1;
}
```

Borrow as **mut**

Borrow as **immut**

```
pub fn remove_duplicates(nums: &mut Vec<i32>) -> i32 {  
    if nums.is_empty() {  
        return 0;  
    }  
  
    let mut it = nums.iter_mut();  
  
    let mut cache = *(it.next().unwrap());  
  
    let mut count = 1;  
  
    for &x in nums.iter() {  
        if x != cache {  
            ...  
        }  
    }  
}
```

```
pub fn remove_duplicates(nums: &mut Vec<i32>) -> i32 {  
    if nums.is_empty() {  
        return 0;  
    }  
  
    let mut it = nums.iter_mut()  
  
    let mut cache = *(it.next().unwrap());  
  
    let mut count = 1;  
  
    for &x in nums.iter() {  
        if x != cache {  
            ...  
        }  
    }  
}
```

```
pub fn remove_duplicates(nums: &mut Vec<i32>) -> i32 {  
    if nums.is_empty() {  
        return 0;  
    }  
  
    let mut it = nums.iter_mut()  
  
    let mut cache = *(it.next().unwrap());  
  
    let mut count = 1;  
  
    for &x in nums.iter() {  
        if x != cache {  
            ...  
        }  
    }  
}
```

```
pub fn remove_duplicates(nums: &mut Vec<i32>) -> i32 {  
    if nums.is_empty() {  
        return 0;  
    }  
  
    let mut it = nums.iter_mut()  
  
    let mut cache = *(it.next().unwrap());  
  
    let mut count = 1;  
  
    for &x in nums.iter() {  
        if x != cache {
```



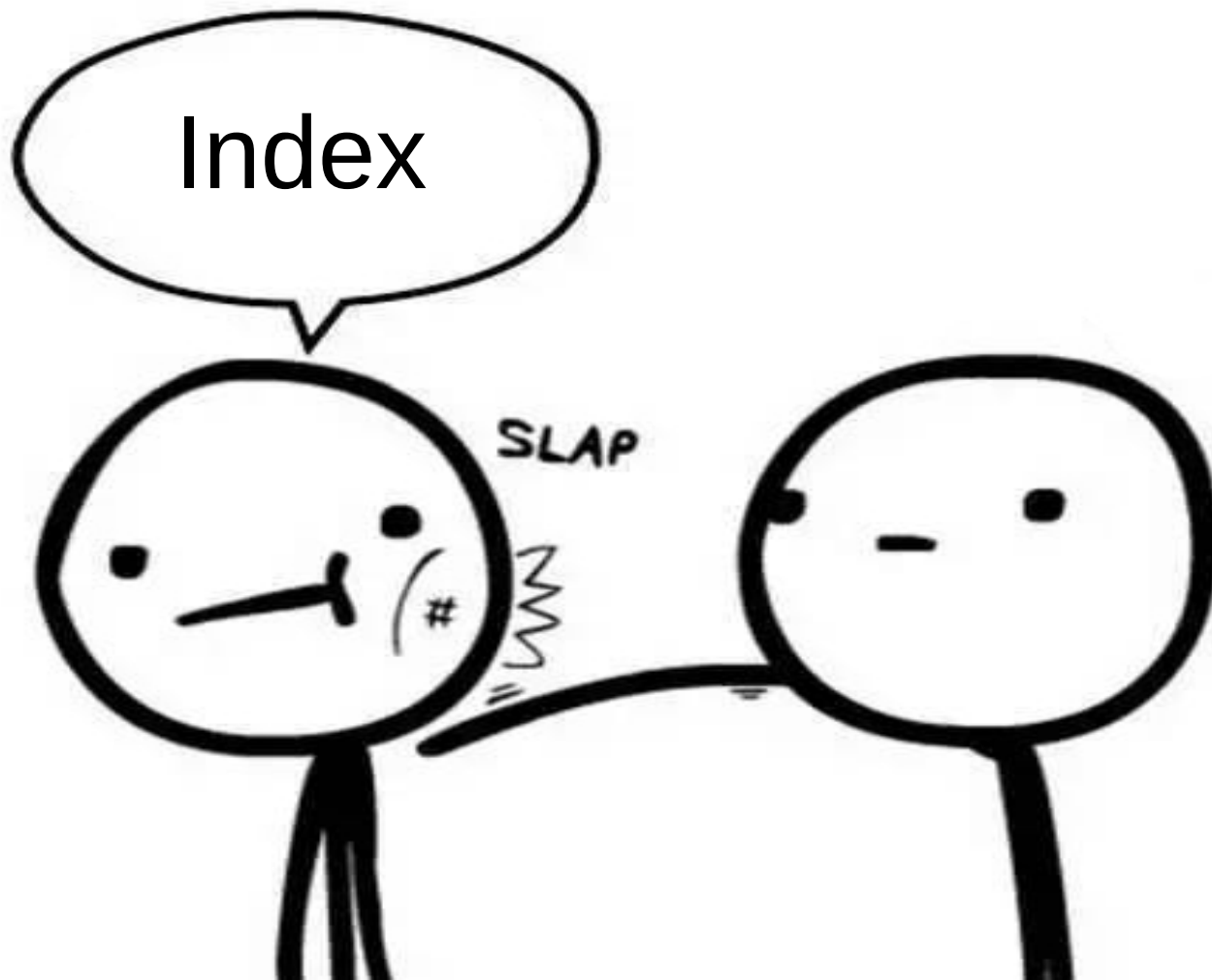
use **x** and **it** at the same time

```
pub fn remove_duplicates(nums: &mut Vec<i32>) -> i32 {  
    if nums.is_empty() {  
        return 0;  
    }  
  
    let mut count = 1;  
  
    let mut cache = nums[0];  
  
    for i in 1..nums.len() {  
        if nums[i] != cache {  
            count+=1;  
  
            nums[count] = nums[i];  
  
            cache = nums[i];  
        }  
    }  
    count  
}
```

# Rust

```
pub fn remove_duplicates(nums: &mut Vec<i32>) -> i32 {  
    if nums.is_empty() {  
        return 0;  
    }  
  
    let mut count = 1;  
  
    let mut cache = nums[0];  
  
    for i in 1..nums.len() {  
        if nums[i] != cache {  
            count+=1;  
            nums[count] = nums[i];  
            cache = nums[i];  
        }  
    }  
    count  
}
```









```
if nums.is_empty() { return 0; }

let mut p = nums.as_mut_ptr();

let mut cache = unsafe { *p };

let mut count = 1;

for &x in nums.iter() {
    if x != cache {
        unsafe {
            p = p.offset(1);
            *p = x;
            cache = *p;
        }

        count += 1;
    }
}

return count;
```

```
if nums.is_empty() { return 0; }

let mut p = nums.as_mut_ptr();

let mut cache = unsafe { *p };

let mut count = 1;

for &x in nums.iter() {
    if x != cache {
        unsafe {
            p = p.offset(1);
            *p = x;
            cache = *p;
        }
        count += 1;
    }
}

return count;
```

```
if nums.is_empty() { return 0; }
```

```
let mut p = nums.as_mut_ptr();
```

```
let mut cache = unsafe { *p };
```

```
let mut count = 1;
```

```
for &x in nums.iter() {
```

```
    if x != cache {
```

```
        unsafe {
```

```
            p = p.offset(1);
```

```
            *p = x;
```

```
            cache = *p;
```

```
        }
```

```
        count += 1;
```

```
    }
```

```
}
```

```
return ((p as usize - nums.as_ptr() as usize) >> 2) as i32 + 1
```

```
fn steal<'a, 'b, T>(x: &'a mut T) -> &'b mut T {  
    let _tmp = x as *mut T;  
    unsafe { std::mem::transmute:::<*mut T, &mut T>(_tmp) }  
}
```

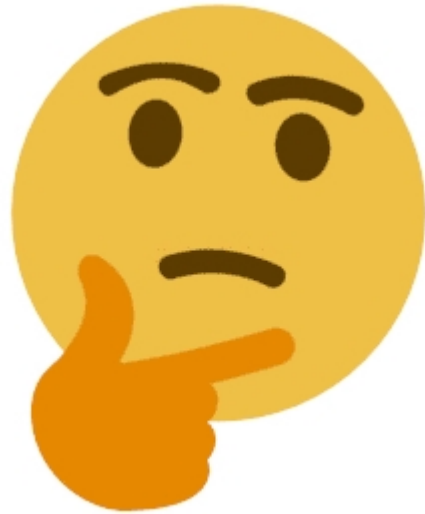
```
fn steal<'a, 'b, T>(x: &'a mut T) -> &'b mut T {  
    let _tmp = x as *mut T;  
    unsafe { std::mem::transmute:::<*mut T, &mut T>(_tmp) }  
}
```

$x: \&'a \text{ mut } T \longrightarrow \_tmp: *mut T \longrightarrow \_0: \&'b \text{ mut } T$

```
fn steal<'a, 'b, T>(x: &'a mut T) -> &'b mut T {  
    let _tmp = x as *mut T;  
    unsafe { std::mem::transmute:::<*mut T, &mut T>(x) }  
}
```

```
fn steal<'a, 'b, T>(x: &'a mut T) -> &'b mut T {  
    unsafe { std::mem::transmute:::<*mut T, &mut T>(x) }  
}
```





```
fn steal<'a, 'b, T>(x: &'a mut T) -> &'b mut T {  
    unsafe { std::mem::transmute:::<&mut T, &mut T>(x) }  
}
```

```
if nums.is_empty() { return 0; }

let mut it = steal(nums).iter_mut();

let mut cache = *(it.next().unwrap());
let mut count = 1;

for &x in nums.iter() {
    if x != cache {
        *(it.next().unwrap()) = x;

        cache = x;

        count += 1;
    }
}

return count;
```

```
if nums.is_empty() { return 0; }

let mut it = steal(nums).iter_mut();

let mut cache = *(it.next().unwrap());

nums.iter().for_each(|&e| if e != cache {
    *(it.next().unwrap()) = e;
    cache = e;
});

return (nums.len() - it.len()) as i32;
```

```
if nums.is_empty() { return 0; }

let mut it = steal(nums).iter_mut();

let mut cache = *(it.next().unwrap());

nums.iter().for_each(|&e| if e != cache {
    *(it.next().unwrap()) = e;
    cache = e;
});

return (nums.len() - it.len()) as i32;
```

*And so once again, the day is saved  
thanks to... The Unsafe*

*That's All*

QA



`github.com/rniczh`