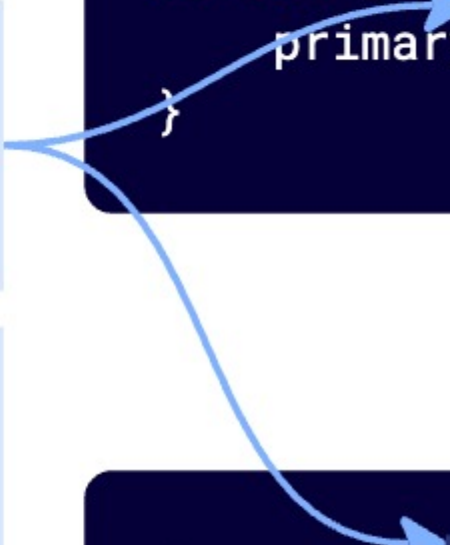


Lifetime annotations

Used with functions, structs, enums, and more

```
struct Account {  
    balance: i32,  
}  
  
struct Bank<'a> {  
    primary_account: &'a Account,  
}
```



```
fn longest<'a>(str_a: &'a str, str_b: &'a str) -> &'a str {  
    if str_a.len() >= str_b.len() {  
        str_a  
    } else {  
        str_b  
    }  
}
```

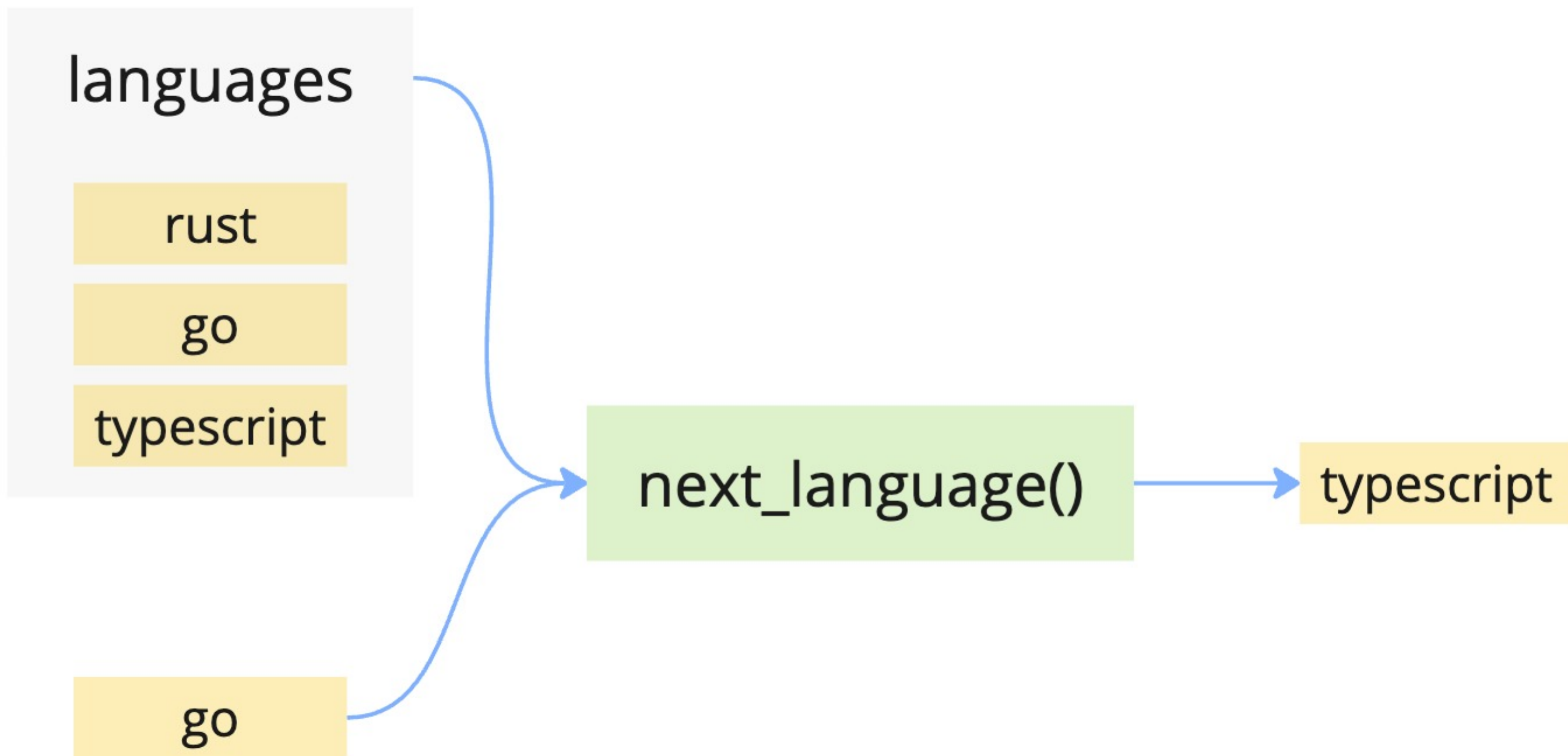
Lifetime annotations

Help the compiler make sure refs won't outlive the value they refer to

Hardest part:

This will seem like something the compiler should do on its own

```
struct Account {  
    balance: i32,  
}  
  
struct Bank<'a> {  
    primary_account: &'a Account,  
}  
  
fn make_bank() -> Bank {  
    let account = Account { balance: 10 };  
    let bank = Bank { primary_account: &account };  
  
    bank  
} // 'account' goes out of scope
```



```
fn next_language(languages: &[String], current: &str) -> &str {
```

```
graph TD; A["fn next_language(languages: &[String], current: &str) -> &str {"] --> B["If you're returning a ref, Rust gets concerned that you might be violating a rule of borrowing"]; B --> C["There are multiple ways we can implement 'next_language()'"]; B --> D["There are multiple ways we can call 'next_language()'"]; C --> E["Some of these combinations will work, others wont"]; D --> E;
```

If you're returning a ref, Rust gets concerned that you might be violating a rule of borrowing

There are multiple ways we can **implement** 'next_language()'


There are multiple ways we can **call** 'next_language()'

Some of these combinations will work, others wont

Implementation A

Returned reference is related to the **first** argument


```
fn next_lang(langs: &[String], current: &str) -> &str {  
    langs.last().unwrap()  
}
```



Implementation B

Returned reference is related to the **second** argument

```
fn next_lang(langs: &[amp;String], current: &str) -> &str {  
    current  
}
```



```
fn next_lang(langs: &[String], current: &str) -> &str {  
    langs.last().unwrap()  
}  
  
fn get_next(langs: &[String]) -> &str {  
    let find_this = "go";  
    next_lang(langs, find_this)  
}  
  
fn main() {  
    let list = vec![/* list */];  
  
    let result = get_next(&langs);  
}
```

**Implementation A
with Invocation A**

Invocation A
'languages' lives long enough

Another way we could implement 'next_language()'

```
fn next_lang(langs: &[String], current: &str) -> &str {
    langs.last().unwrap()
}

fn get_next(langs: &[String]) -> &str {
    let find_this = "go";
    next_lang(langs, find_this)
}

fn main() {
    let list = vec![/* list */];

    let result = get_next(&langs);
}
```

Vec<String>

rust

rust

go


typescript

Ref

'list' binding	
'langs' argument	
'find_this' binding	
'langs' argument	
'current' argument	

Another way we could implement 'next_language'

```
fn next_lang(langs: &[amp;String], current: &str) -> &str {  
    langs.last().unwrap()  
}
```



Returned reference is clearly related to the first argument

This would work

```
fn next_lang(langs: &[String], current: &str) -> &str {
    langs.last().unwrap()
}

fn main() {
    let langs = vec![/* list */];
    let result;

    {
        let find_this = "go";
        result = next_lang(&langs, find_this);
    }

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

Vec<String>		rust	Ref
rust	go	typescript	
'langs' binding			
'find_this' binding			
'langs' arg			
'current' arg			
'result' binding			

This would break!

```
fn next_lang(langs: &[String], current: &str) -> &str {
    langs.last().unwrap()
}

fn main() {
    let langs = vec![/* list */];
    let result;

    {
        let find_this = "go";
        result = next_lang(&langs, find_this);
    }

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

Vec<String>

rust

Ref

rust


go

typescript

'langs' binding	
'find_this' binding	
'langs' arg	
'current' arg	
'result' binding	

Another way we could implement 'next_language'

```
fn next_lang(langs: &[amp;String], current: &str) -> &str {  
    current  
}
```



Returned reference is clearly related to the
second argument

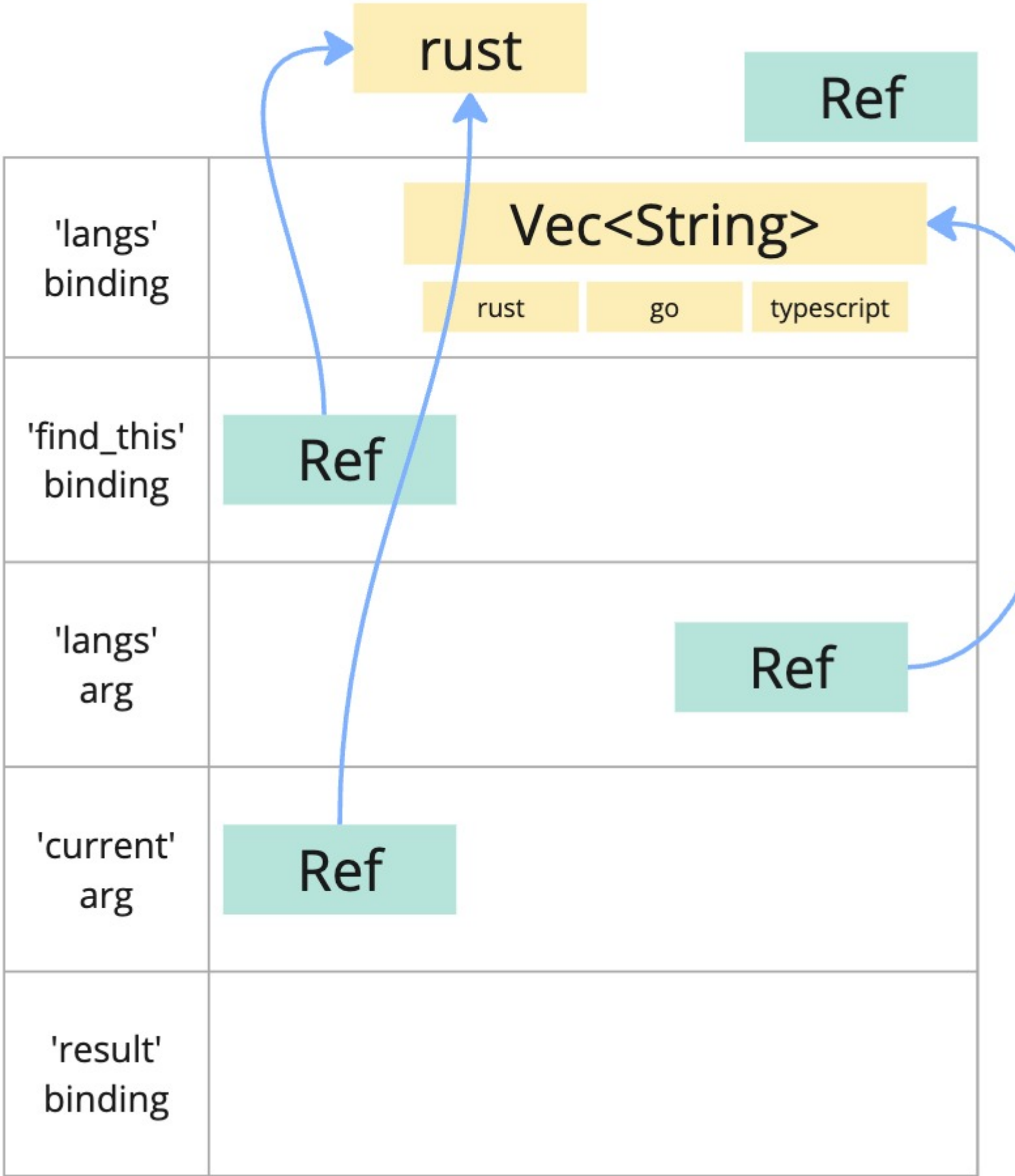
This would work

```
fn next_lang(langs: &[String], current: &str) -> &str {
    current
}

fn main() {
    let find_this = "go";
    let result;

    {
        let langs = vec![/* list */];
        result = next_lang(&langs, find_this);
    }


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



Two ways we could implement 'next_language()'


*Returned ref is
related to the
'langs' arg*

```
fn next_lang(langs: &[String], current: &str) -> &str {  
    langs.last().unwrap()  
}
```



*Returned ref is
related to the
'current' arg*

```
fn next_lang(langs: &[String], current: &str) -> &str {  
    current  
}
```



Two ways we could **call** 'next_language()'

'current' goes out of scope early

```
fn main() {  
    let langs = vec![/* list */];  
    let result;  
  
    {  
        let current = "go";  
        result = next_lang(&langs, current);  
    } // current is about to go out scope  
    // if anything is referring to current  
    // this code will not work!  
  
    println!("{}", result);  
}
```

Only works if **'result'** refers to **'langs'**


'langs' goes out of scope early

```
fn main() {  
    let current = "go";  
    let result;  
  
    {  
        let langs = vec![/* list */];  
        result = next_lang(&langs, current);  
    } // langs is going to go out of scope  
    // value will be dropped  
    // if anything refers to langs, this wont  
    // work  
  
    println!("{}", result);  
}
```

Only works if **'result'** refers to **'current'**


Returned ref related to 'langs'

```
fn next_lang(langs: &[String], current: &str) -> &str {  
    langs.last().unwrap()  
}
```



Returned ref related to 'current'

```
fn next_lang(langs: &[String], current: &str) -> &str {  
    current  
}
```



Works if **'result'** refers to **'langs'**

```
fn main() {  
    let langs = vec![/* list */];  
    let result;  
  
    {  
        let current = "go";  
        result = next_lang(&langs, current);  
    } // current goes out of scope!  
  
    println!("{}", result);  
}
```

Works if **'result'** refers to **'current'**


```
fn main() {  
    let current = "go";  
    let result;  
  
    {  
        let langs = vec![/* list */];  
        result = next_lang(&langs, current);  
    } // langs goes out scope!  
  
    println!("{}", result);  
}
```

At this point, its clear that...

The returned ref can refer to the first or second arg

We need to know if the ref is associated with the first or second arg - affects how we can call the fn

The returned ref *must* be tied to 'languages' or 'current'

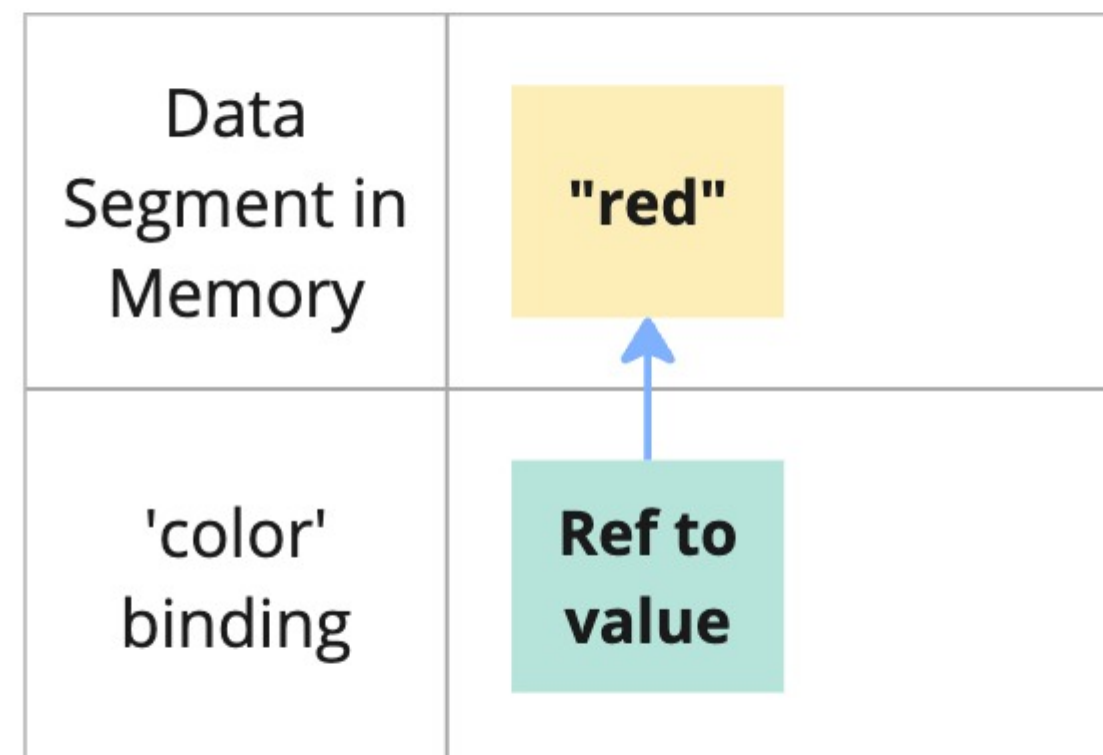



```
fn next_language(languages: &[String], current: &str) -> &str {  
    let mut found = false;  
  
    for lang in languages {  
        if found {  
            return lang;  
        }  
  
        if lang == current {  
            found = true;  
        }  
    }  
  
    languages.last().unwrap()  
}
```

*Technically the returned
ref could point at a
string constant as well*

```
fn next_language(languages: &[String], current:
    &str) -> &str {
    let color = "red";

    color
}
```






```
fn next_lang(langs: &[String], current: &str) -> &str {  
    /* implementation... */  
}
```

```
fn main() {  
    let current = "go";  
    let result;  
    let langs = vec![/* list */];  
  
    {  
        result = next_lang(&langs, current);  
    } // langs goes out of scope, value is  
        dropped. But oops! we still have  
        result which refers to langs  
  
    println!("{}", result);  
}
```

This won't work!

**'langs' goes out of scope
while a ref to one of its
elements still exists**

```
fn next_lang(langs: &[String], current: &str) -> &str {  
    /* implementation... */  
}
```




```
fn main() {  
    let langs = vec![/* list */];  
    let result;  
  
    {  
        let current = "go";  
        result = next_lang(&langs, current);  
    } // current goes out of scope  
  
    println!("{}", result);  
}
```

This won't work!

**'current' goes out of scope
while a ref to it still exists**

The returned ref must refer to either the first or second arg

Might break depending on which arg the ref refers to + how we call the function



```
fn next_language(languages: &[String], current: &str) -> &str {  
    let mut found = false;  
  
    for lang in languages {  
        if found {  
            return lang;  
        }  
  
        if lang == current {  
            found = true;  
        }  
    }  
  
    languages.last().unwrap()  
}
```

Surprising!

Rust *intentionally* doesn't look at your function body to figure out if the ref is tied to the first or second arg!

Your code


```
fn main() {  
    let langs = vec![/* list */];  
    let result;  
  
    {  
        let current = "go";  
        result = next_lang(&langs, current);  
    } // current goes out of scope  
  
    println!("{}", result);  
}
```

Fn from a lib you're using.
This is the only documentation you see



```
fn next_language(languages: &[String], current: &str) -> &str
```

How that function is actually implemented



```
fn next_language(languages: &[String], current: &str) -> &str {  
    let mut found = false;  
  
    for lang in languages {  
        if found {  
            return lang;  
        }  
  
        if lang == current {  
            found = true;  
        }  
    }  
  
    current  
}
```

Rust ***intentionally*** doesn't look at your function body to figure out if the ref is tied to the first or second arg!



Rust wants a function signature to make it clear whether the returned ref relies on the first or second arg (or both)


```

fn last_language(languages: &[String]) -> &str {
    // languages.last().unwrap()
}

fn main() {
    let result;

    {
        let languages = vec![/* strings */];
        result = last_language(&languages);
    } // 'languages' goes out of scope

    println!("Result: {}", result);
}

```

'languages' binding in main	<div>Vector</div> <div>rustgotypescript</div>
'languages' arg	
'result' binding	Ref to value

Argument is a ref

Function returns a ref



```
fn last_language(languages: &[String]) -> &str
```

**Rust assumes the returned ref points at data
owned by the argument ref**

Rust makes assumptions about what the return ref is pointing at in two scenarios

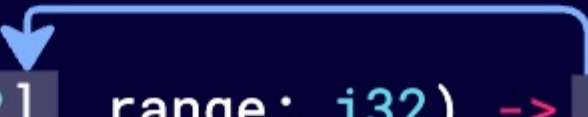
There are more explicit rules for this, these two are the most common

```
fn last_language(languages: &[String]) -> &str
```



Function that takes one ref + any number of values + returns a ref

```
fn generate(set: &[i32], range: i32) -> &str
```




Function that takes one ref + any number of values + returns a ref

```
fn leave(message: &Message, text: String) -> &str
```



Function that takes one ref + any number of values + returns a ref

```
struct Bank {  
    name: String,  
}  
  
impl Bank {  
    fn get_name(&self, default_name: &str) -> &str {  
        &self.name  
    }  
}
```



Method that takes &self and any number of other refs + returns a ref.
Rust assumes the returned ref will point at &self

Rusts assumptions incorrect?
Don't fall into those two scenarios?



Use lifetime annotations

**Lifetime annotations clarify the relationship
between input refs and return refs**

```
let languages = vec![  
  String::from("rust"),  
  String::from("go"),  
  String::from("typescript"),  
];
```

Name	Description	Args	Return
last_language()	Returns the last element in the vector	&[String]	&str
next_language()	Finds a given language and returns the next one	&[String], &str	&str
longest()	Returns the longer of two languages	&str, &str	&str

Argument is a
ref

Argument is a
ref

Returns a ref

```
fn next_lang(langs: &[String], current: &str) -> &str {  
    /* implementation */  
}
```

Rust assumes the returned ref points at data owned by 'langs' or 'current', but which one?

We tell Rust how these references are related by using lifetime annotations