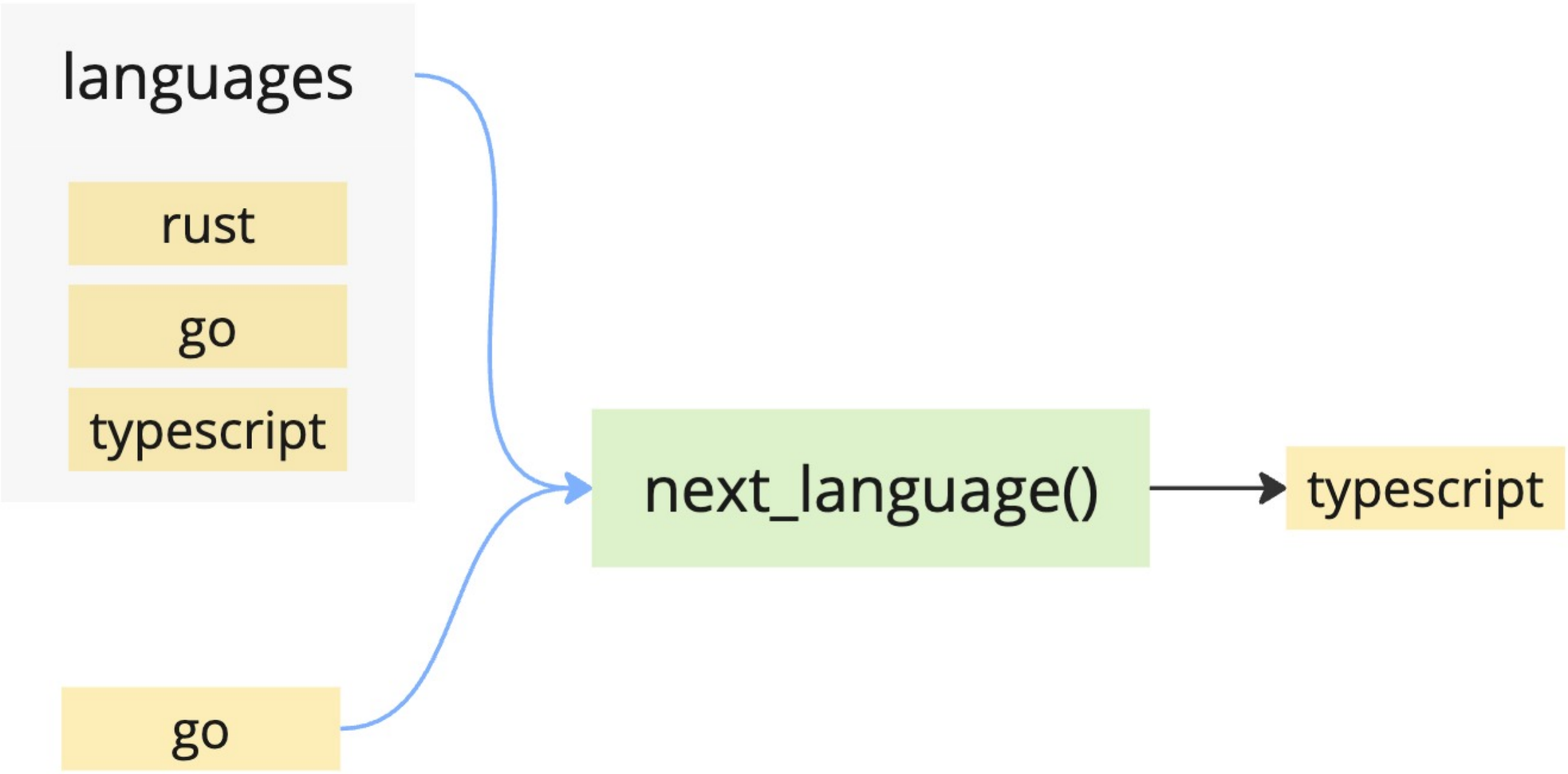

languages
rust
go
typescript
go
next_language()
typescript

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

    let result = next_language(&languages, "go");

    println!("{}", result);
} // languages goes out of scope, value is dropped
```

Lifetime of 'languages'

A lifetime is how long a binding can be used

1	Every value is 'owned' by a single variable, argument, struct, vector, etc at a time	**Ownership**
2	Reassigning the value to a variable, passing it to a function, putting it into a vector, etc, *moves* the value. The old owner can't be used to access the value anymore!	
3	You can create many read-only references to a value that exist at the same time. These refs can all exist at the same time	**Borrowing**
4	You can't move a value while a ref to the value exists	
5	You can make a writeable (mutable) reference to a value *only if* there are no read-only references currently in use. One mutable ref to a value can exist at a time	
6	You can't mutate a value through the owner when any ref (mutable or immutable) to the value exists	
7	Some types of values are *copied* instead of moved (numbers, bools, chars, arrays/tuples with copyable elements)	
8	When an owner goes out of scope, the value owned by it is *dropped* (cleaned up in memory)	**Lifetimes**
9	There can't be references to a value when its owner goes out of scope	
10	References to a value can't outlive the value they refer to	
11	**These rules will dramatically change how you write code (compared to other languages)**	
12	**When in doubt, remember that Rust wants to minimize unexpected updates to data**	

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

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

    let result = next_language(&languages, "go");

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

data segment in memory	go
'languages' binding in main	**Vector** rust go typescript
'current' argument	**Ref to value**
'languages' arg	**Ref to value**
'result' binding	**Ref to value**

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

fn main() {
    let result;

    {
        let languages = vec![
            String::from("rust"),
            String::from("go"),
            String::from("typescript"),
        ];
        result = next_language(&languages, "go");
    } // languages goes out of scope, value is
        dropped!

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

data segment in memory	
'languages' binding in main	**Vector** rust | go | typescript
'current' argument	**Ref to value**
'languages' arg	**Ref to value**
'result' binding	**Ref to value**

Function that takes in two refs and returns a ref

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

If you have a function that
takes in two or more refs
and
returns a ref
Rust will make a **huge assumption**

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

Rust assumes that the return ref will point at data referred to by one of the arguments

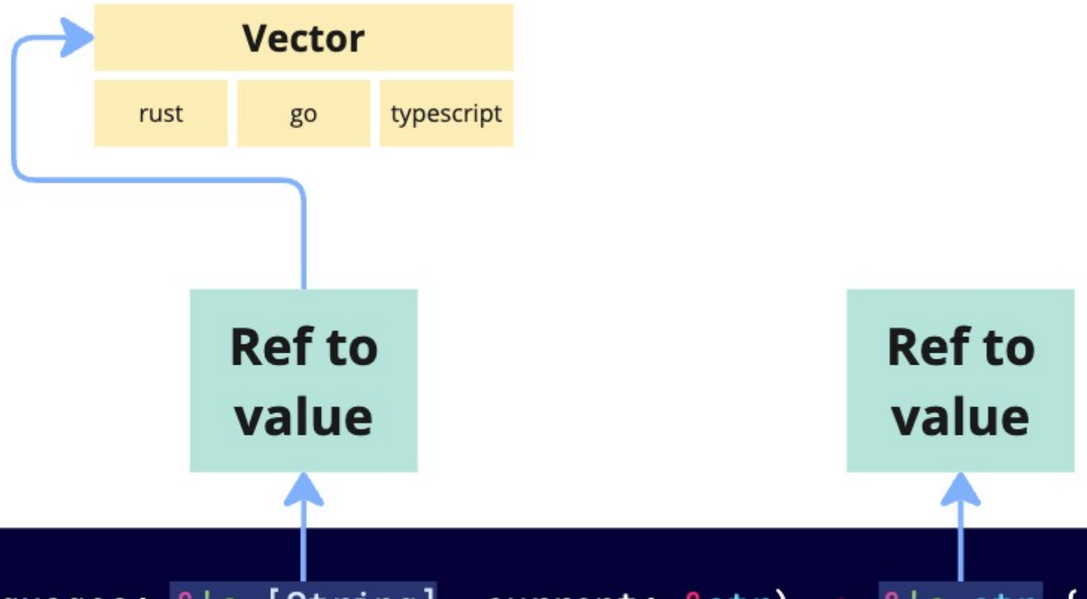
There is a type of ref called 'a'

This first ref is of type 'a'

This returned ref is also of type 'a'

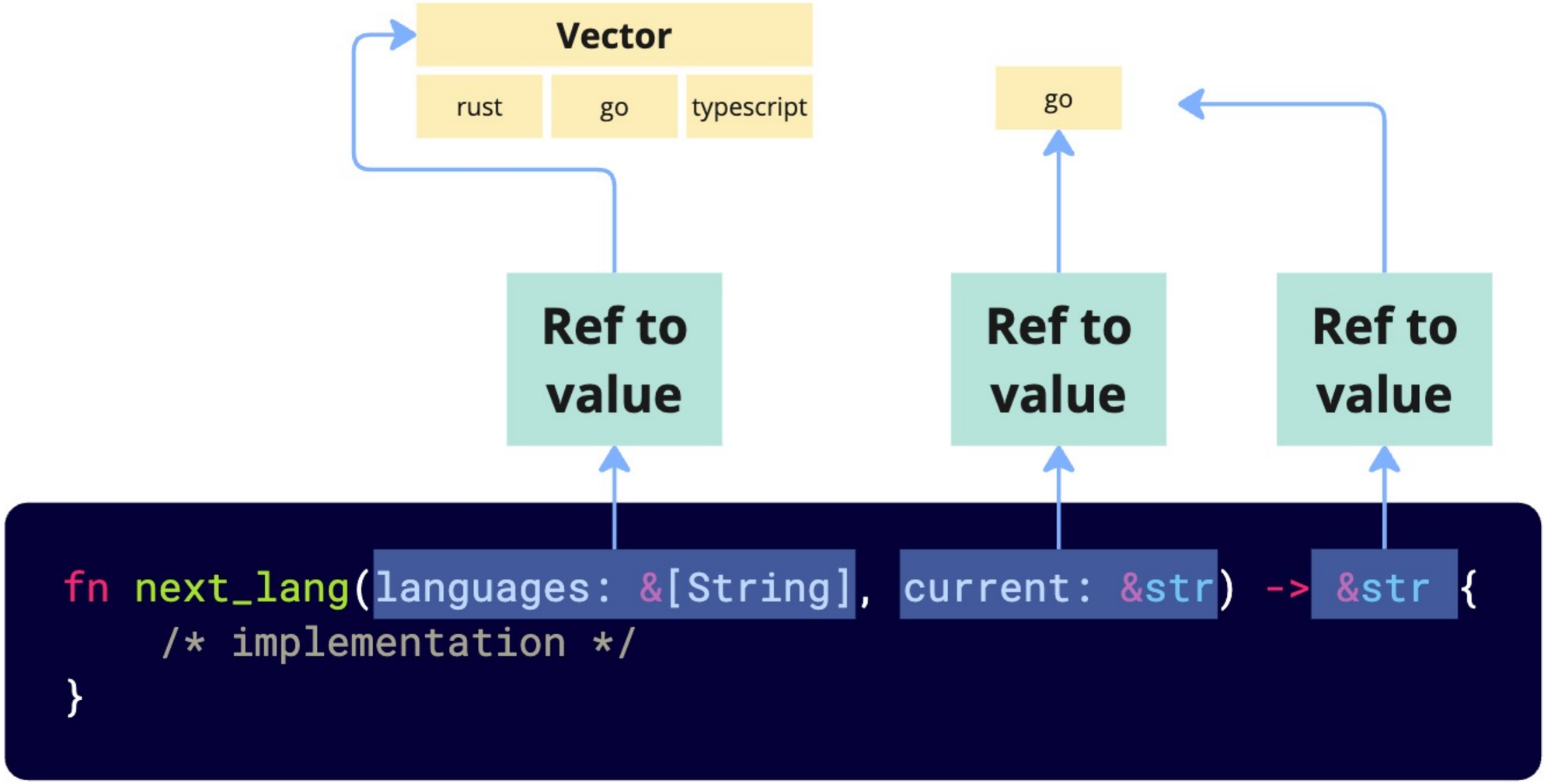
```rust
fn next_lang<'a>(languages: &'a [String], current: &str) -> &'a str {
    let mut found = false;

    for lang in languages {
        if found {
            return lang;
        }

        if lang == current {
            found = true;
        }
    }

    languages.last().unwrap()
}
```

To clarify which ref the return ref is pointing at, we have to add lifetime annotations

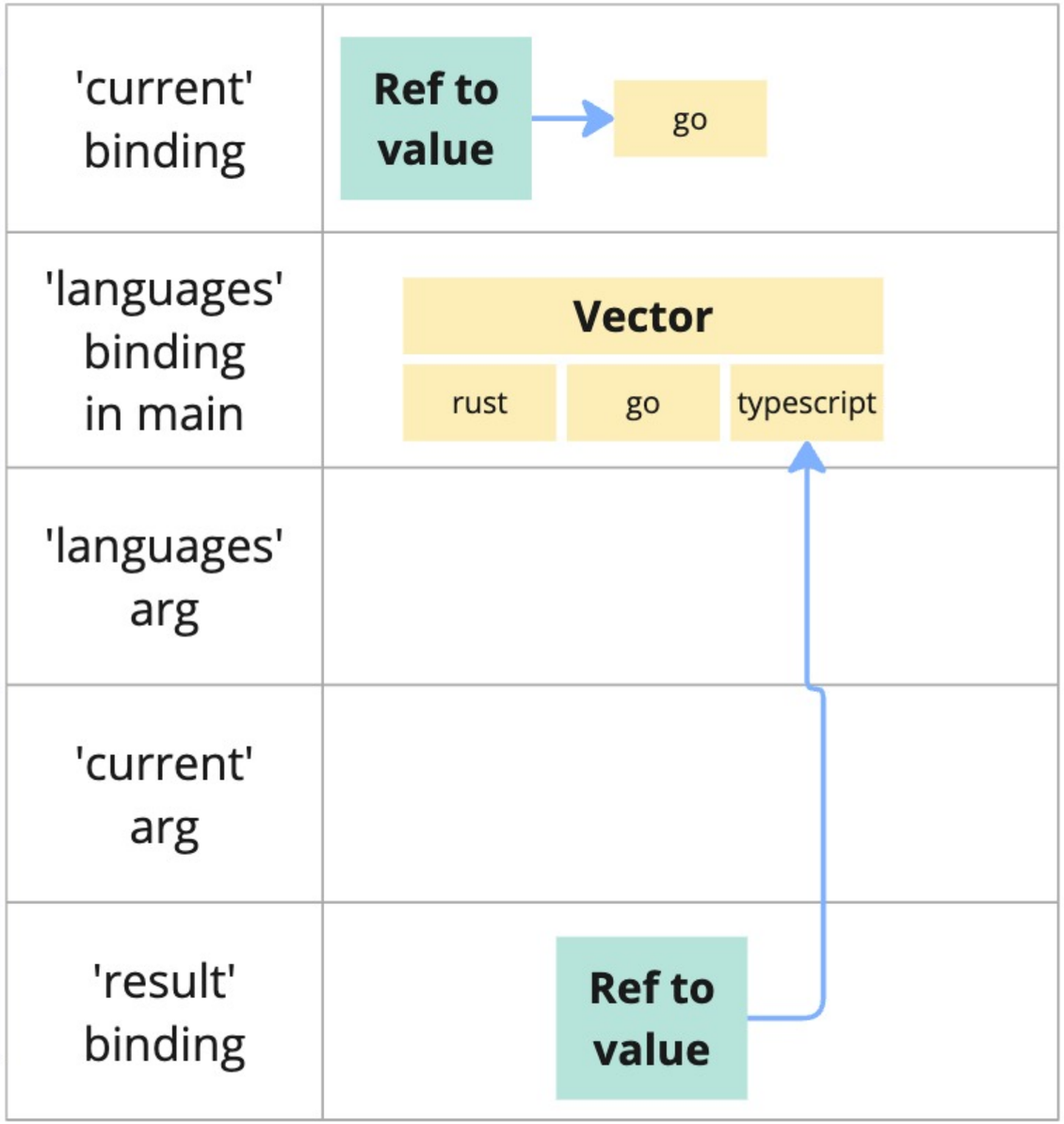



```rust
fn next_lang<'a>(languages: &'a [String], current: &str) -> &'a str {
    let mut found = false;

    for lang in languages {
        if found {
            return lang;
        }

        if lang == current {
            found = true;
        }
    }

    languages.last().unwrap()
}
```

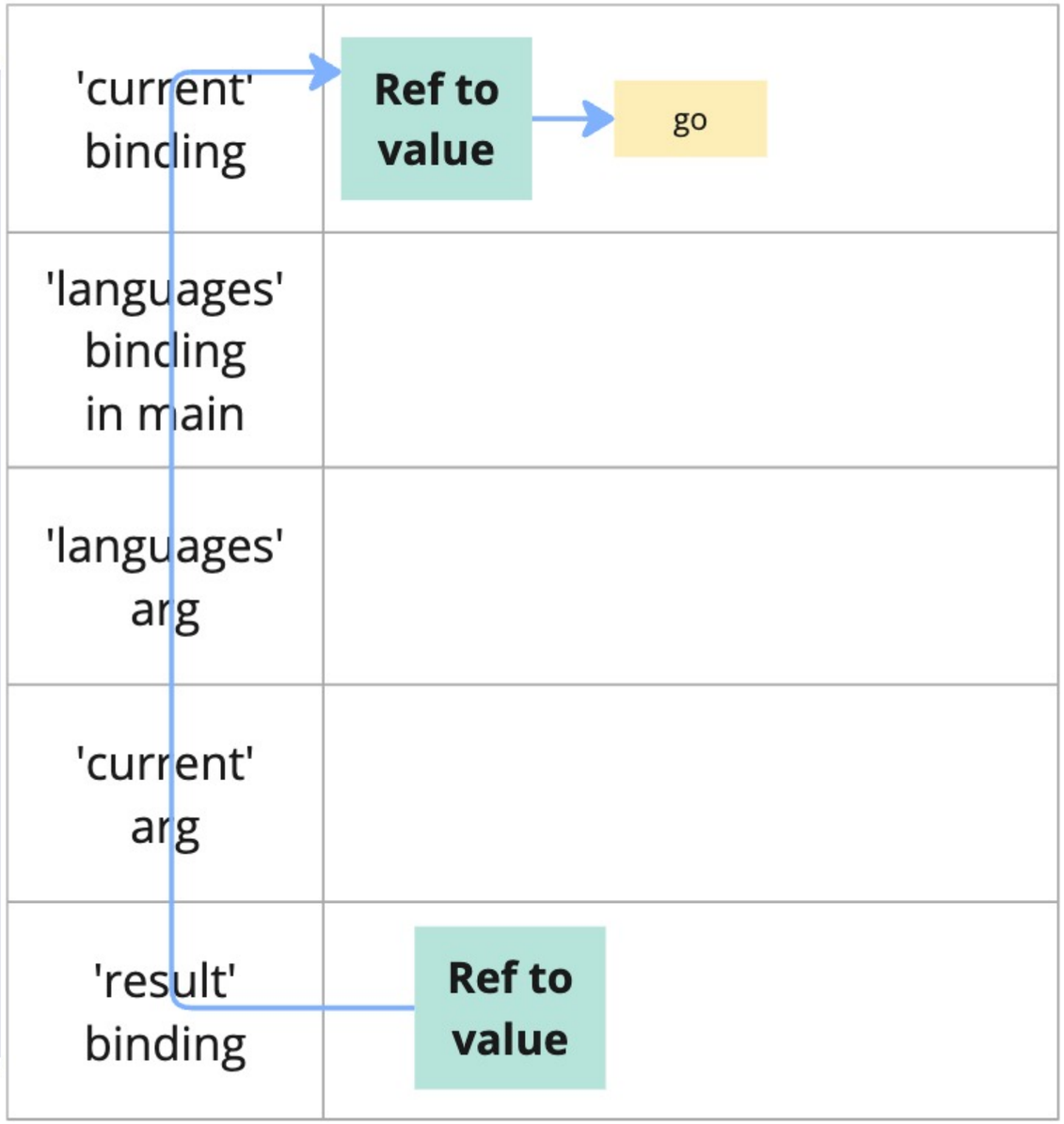
The lifetime annotation makes it clear that the returned ref is pointing at data tied to the first arg


Vector
rust
go
typescript
go
Ref to value
Ref to value
Ref to value
fn next_lang(languages: &[String], current: &str) -> &str {
/* implementation */
}

```rust
fn next_lang(languages: &[String], current: &str) -> &str {
    // Returning a ref to data tied to the FIRST arg
    languages.last().unwrap()
}

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

    {
        let languages = vec![
            String::from("rust"),
            String::from("go"),
            String::from("typescript"),
        ];
        result = next_language(&languages, current);
    } // langugaes goes out of scope!

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

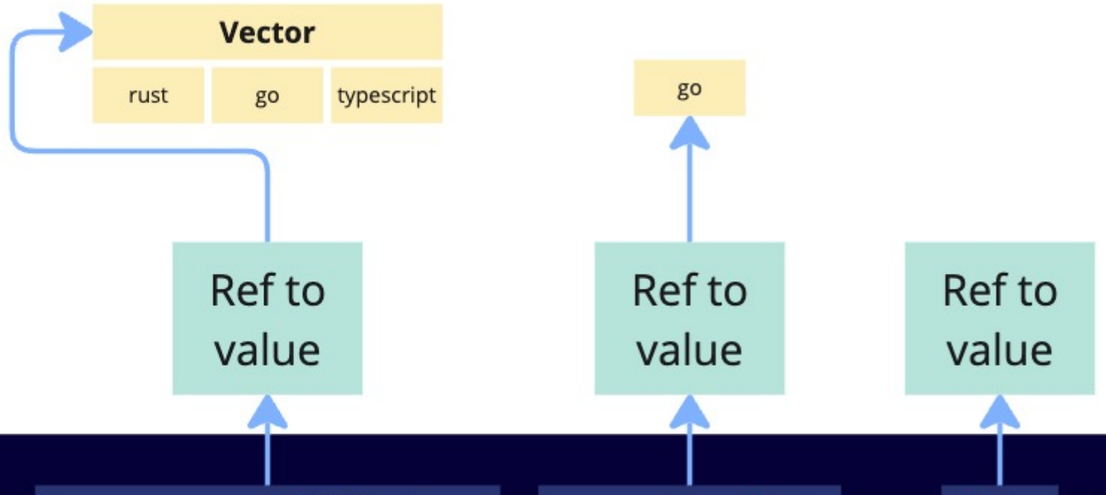


```rust
fn next_lang(languages: &[String], current: &str) -> &str {
    // Returning a ref to data tied to the SECOND arg
    current
}

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

    {
        let languages = vec![
            String::from("rust"),
            String::from("go"),
            String::from("typescript"),
        ];
        result = next_language(&languages, current);
    } // languages goes out of scope!!

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



Super Common Questions

1	Why does it matter whether the return ref points at the first or second arg?
2	Why doesn't Rust analyze the function body to figure out if the returned ref points at the first or second arg?




```rust
fn next_lang(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()
}
```

Rust will not analyze the body of your function to figure out whether the return ref is pointing at the first or second arg

You're using a library that implements a 'split' function

The function signature makes it clear that the returned ref will be tied to the first arg

```
fn split<'a>(s: &'a str, pattern: &str) -> &'a str
```

```rust
fn split<'a>(s: &'a str, pattern: &str) -> &'a str
```

Works!

```rust
fn main() {
    let sentence = "hi how are you";
    let result;

    {
        let pattern = " ";
        result = split(sentence, pattern);
    } // pattern goes out of scope

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

This won't work

```rust
fn main() {
    let pattern = " ";
    let result;

    {
        let sentence = "hi how are you";
        result = split(sentence, pattern);
    } // sentence goes out of scope

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

If we relied on the Rust to figure out the lifetimes, **we wouldn't know if the returned ref uses the first or second arg**

```rust
fn split(s: &str, pattern: &str) -> &str
```

```rust
fn split(s: &str, pattern: &str) -> &str
```

Don't know which will work!

```rust
fn main() {
    let sentence = "hi how are you";
    let result;

    {
        let pattern = " ";
        result = split(sentence, pattern);
    } // pattern goes out of scope

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

```rust
fn main() {
    let pattern = " ";
    let result;

    {
        let sentence = "hi how are you";
        result = split(sentence, pattern);
    } // sentence goes out of scope

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

```rust
fn split(s: &str, pattern: &str) -> &str {
    if random_number() > 0.5 {
        s
    } else {
        pattern
    }
}
```

Rust assumes the returned ref is tied to the only arg

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

You *could* add in lifetime annotations

```rust
fn last_language<'a>(languages: &'a [String]) -> &'a str {
    languages.last().unwrap()
}
```

We have to think about annotations anytime your function receives a ref and returns a ref

You can omit annotations in two scenarios.

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

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

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

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

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

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

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

```rust
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**

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

Omitting lifetime annotations is referred to as **elision**

"I **removed** the lifetime annotations"	→	"I **elided** the lifetime annotations"
"We can **remove** the annotations"	→	"We can **elide** the annotations"
"Think about **removal** of the annotations"	→	"Think about **elision** of the annotations"

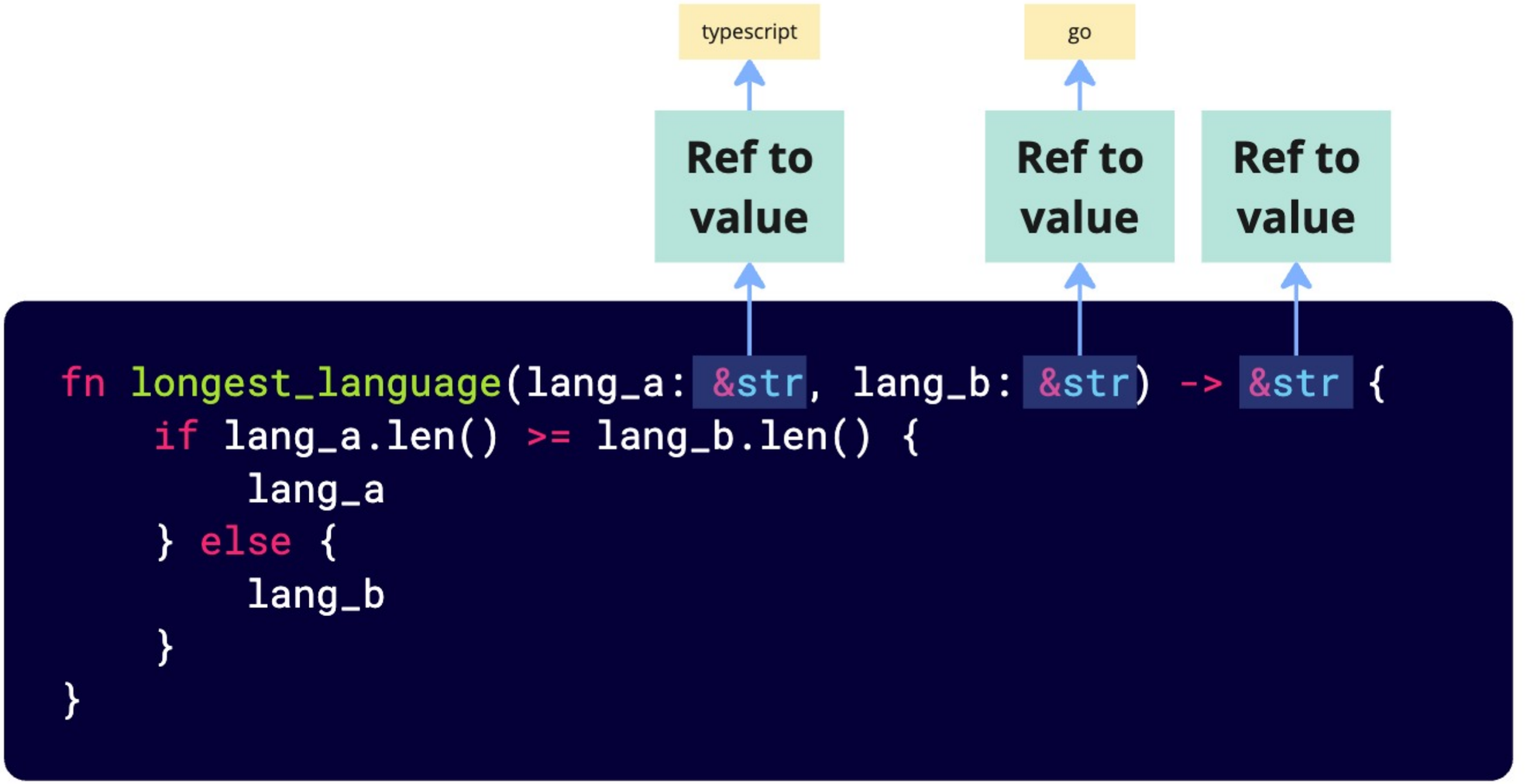
Prounouncing

co-llide	→	***ee-***lide
co-llided	→	***ee***-lided
co-llision	→	***ee-***lision

```rust
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_language()	Returns the longer of two languages	&str, &str	&str


typescript
go
longest()
typescript


typescript
go
Ref to value
Ref to value
Ref to value
fn longest_language(lang_a: &str, lang_b: &str) -> &str {
if lang_a.len() >= lang_b.len() {
lang_a
} else {
lang_b
}
}

There is a type of ref called 'a'

These are both refs of type 'a'

This ref will point at one of the 'a' refs

```rust
fn longest_language<'a>(lang_a: &'a str, lang_b: &'a str) -> &'a str {
    if lang_a.len() >= lang_b.len() {
        lang_a
    } else {
        lang_b
    }
}
```

Channel	
name	&str
messages	Vec<Message>
get_name() -> &str	
get_matching_message(content: &str) -> &str	

Message	
content	String