

Advanced Features

Rnic / H.-S. Zheng

April 20, 2019

Outline

Unsafe

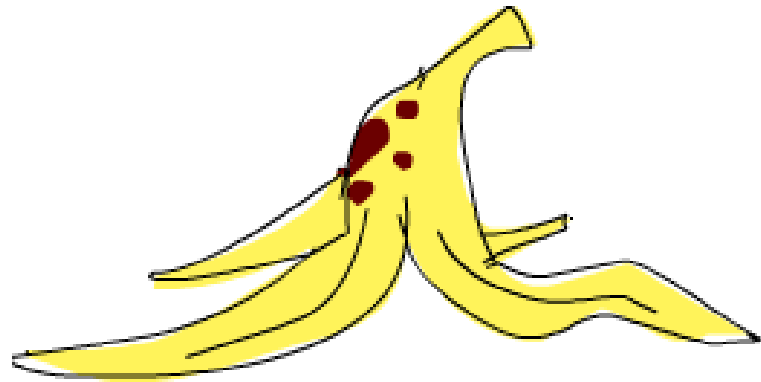
Macros

Lifetimes

Traits

Fn and Closure

Unsafe



What is **unsafe** operation in Rust ?

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait

What is **unsafe** operation in Rust ?

- **Dereference a raw pointer**
- Call an unsafe function or method
- Access or modify a mutable static
- Implement an unsafe trait

```
let ptr;  
{  
    let x = 5i32;  
    ptr = &x as *const i32;  
}  
  
unsafe {  
    println!("{:?}", *ptr);  
}
```

What is **unsafe** operation in Rust ?

- **Dereference a raw pointer**
- Call an unsafe function or method
- Access or modify a mutable static
- Implement an unsafe trait

```
let ptr;  
{  
    let x = 5i32;  
    ptr = &x as *const i32;  
}  
  
unsafe {  
    println!("{}", *ptr);  
}
```

What is **unsafe** operation in Rust ?

- **Dereference a raw pointer**
- Call an unsafe function or method
- Access or modify a mutable static
- Implement an unsafe trait



```
let ptr;  
{  
    let x = 5i32;  
    ptr = &x as *const i32;  
}  
  
unsafe {  
    println!("{:?}", *ptr);  
}
```

What is **unsafe** operation in Rust ?


- Dereference a raw pointer
- **Call an unsafe function or method**
- Access or modify a mutable static variable
- Implement an unsafe trait

```
unsafe fn unsafe_func() { }  
  
fn main() {  
    unsafe {  
        unsafe_func();  
    }  
}
```


What is **unsafe** operation in Rust ?

- Dereference a raw pointer
- **Call an unsafe function or method**
- Access or modify a mutable static variable
- Implement an unsafe trait

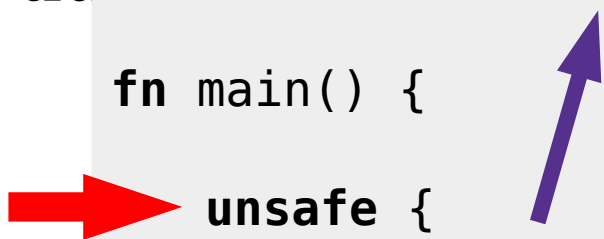
```
unsafe fn unsafe_func() { }  
  
fn main() {  
    unsafe {  
        unsafe_func();  
    }  
}
```



What is **unsafe** operation in Rust ?

- Dereference a raw pointer
- **Call an unsafe function or method**
- Access or modify a mutable static variable
- Implement an unsafe trait

```
unsafe fn unsafe_func() { }  
  
fn main() {  
    unsafe {  
        unsafe_func();  
    }  
}
```



What is **unsafe** operation in Rust ?

- Dereference a raw pointer
- Call an unsafe function or method
- **Access or modify a mutable static variable**
- Implement

```
static mut COUNTER: u32 = 0;
fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}
fn main() {
    add_to_count(3);
    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

What is **unsafe** operation in Rust ?

- Dereference a raw pointer
- Call an unsafe function or method
- **Access or modify a mutable static variable**
- Implement

```
static mut COUNTER: u32 = 0;
fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}
fn main() {
    add_to_count(3);
    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

What is **unsafe** operation in Rust ?

- Dereference a raw pointer
- Call an unsafe function or method
- **Access or modify a mutable static variable**
- Implement

```
static mut COUNTER: u32 = 0;
fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}
fn main() {
    add_to_count(3);
    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}
```

What is **unsafe** operation in Rust ?

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- **Implement an unsafe trait**

```
unsafe trait Foo {  
    // methods go here  
}  
unsafe impl Foo for i32 {  
    // method implementations go here  
}
```

What is **unsafe** operation in Rust ?

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- **Implement an unsafe trait**

```
unsafe trait Foo {  
    // methods go here  
}  
  
unsafe impl Foo for i32 {  
    // method implementations go here  
}
```

What is **unsafe** operation in Rust ?

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- **Implement an unsafe trait**



```
unsafe trait Foo {  
    // methods go here  
}  
unsafe impl Foo for i32 {  
    // method implementations go here  
}
```


What is **unsafe** operation in Rust ?

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- **Implement an unsafe trait**

```
struct MyBox(*mut u8);  
  
unsafe impl Send for MyBox {}  
  
unsafe impl Sync for MyBox {}
```

e.g. impl Send, Sync

What is **unsafe** operation in Rust ?

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- **Implement an unsafe trait**

e.g. `impl Send, Sync`

- **Raw pointer:** isn't **Send**, **Sync**
- **Unsafe Cell:** isn't **Sync** (and therefore `Cell` and `RefCell` aren't)
- **Rc:** isn't **Send**, **Sync**

Creating a Safe Abstraction over Unsafe Code

Wrapping unsafe code in a safe function is a common abstraction.

Example.

```
let mut v = vec![1, 2, 3, 4, 5, 6];  
  
let r = &mut v[..];  
  
let (a, b) = r.split_at_mut(3);  
  
assert_eq!(a, &mut [1, 2, 3]);  
  
assert_eq!(b, &mut [4, 5, 6]);
```

Creating a Safe Abstraction over Unsafe Code

Wrapping unsafe code in a safe function is a common abstraction.

Example.

```
let mut v = vec![1, 2, 3, 4, 5, 6];
```

```
let r = &mut v[..];
```

```
let (a, b) = r.split at mut(3);
```

```
assert_eq!(a, &mut [1, 2, 3]);
```

```
assert_eq!(b, &mut [4, 5, 6]);
```

*We want to implement a
split_at_mut function*

```
fn split_at_mut(slice: &mut [i32], mid: usize)
-> (&mut [i32], &mut [i32])
{
    let len = slice.len();

    assert!(mid <= len);

    (&mut slice[..mid],
     &mut slice[mid..])
}
```

```
fn split_at_mut(slice: &mut [i32], mid: usize)
-> (&mut [i32], &mut [i32])
{
    let len = slice.len();

    assert!(mid <= len);

    (&mut slice[..mid],
     &mut slice[mid..])
}
```

 **two Mutable Borrow occurs at the same time**

Slice: &[i32]

as_ptr()

len()

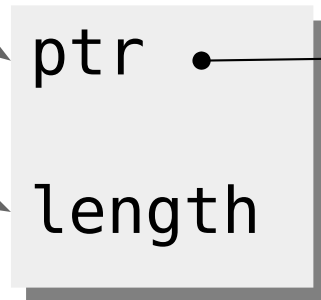
ptr

length

Slice: &[i32]

as_ptr()

len()



Input

from_raw_parts

Output

other_length

newSlice: &[i32]


```

use std::slice;

fn split_at_mut(slice: &mut [i32], mid: usize)
-> (&mut [i32], &mut [i32])
{
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (slice::from_raw_parts_mut(ptr, mid),
         slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}

```

```
use std::slice;

fn split_at_mut(slice: &mut [i32], mid: usize)
-> (&mut [i32], &mut [i32])
{
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (slice::from_raw_parts_mut(ptr, mid),
         slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

```
use std::slice;
```

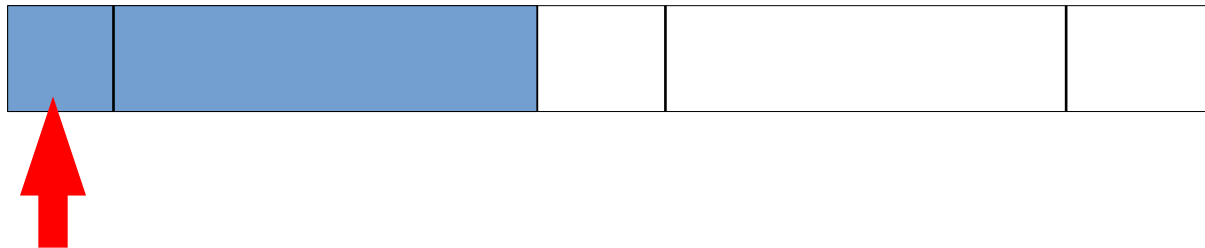
```
fn split_at_mut(slice: &mut [i32], mid: usize)  
-> (&mut [i32], &mut [i32])
```

```
{  
    let len = slice.len();  
    let ptr = slice.as_mut_ptr();
```

```
    assert!(mid <= len);
```

```
    unsafe {  
        (slice::from_raw_parts_mut(ptr, mid),  
         slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))  
    }
```

```
}
```



```
use std::slice;
```

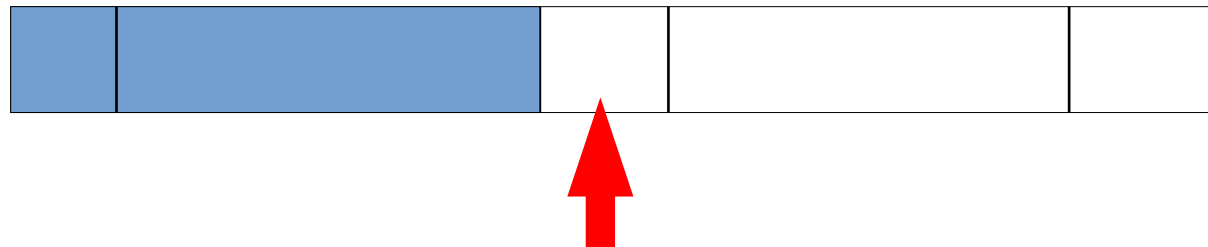
```
fn split_at_mut(slice: &mut [i32], mid: usize)  
-> (&mut [i32], &mut [i32])
```

```
{  
    let len = slice.len();  
    let ptr = slice.as_mut_ptr();
```

```
    assert!(mid <= len);
```

```
    unsafe {  
        (slice::from_raw_parts_mut(ptr, mid),  
         slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))  
    }
```

```
}
```



```
use std::slice;
```

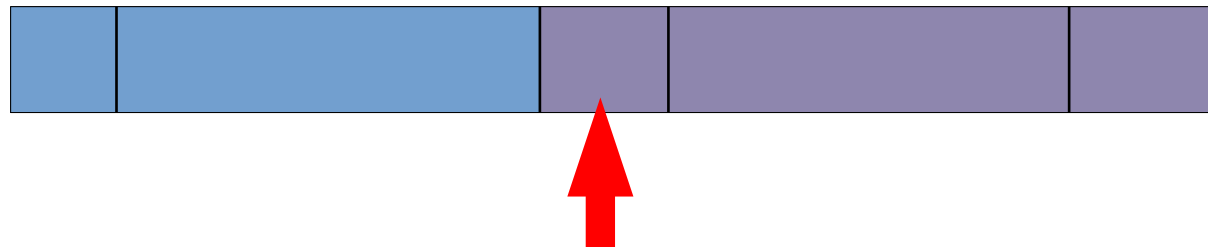
```
fn split_at_mut(slice: &mut [i32], mid: usize)  
-> (&mut [i32], &mut [i32])
```

```
{  
    let len = slice.len();  
    let ptr = slice.as_mut_ptr();
```

```
    assert!(mid <= len);
```

```
    unsafe {  
        (slice::from_raw_parts_mut(ptr, mid),  
         slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))  
    }
```

```
}
```



Convert raw pointer to reference

```
use std::mem::transmute;

fn main() {
    let r;
    {
        let x = 5;

        let p = &x as *const _;

        r = unsafe { transmute::<*const i32, &i32>(p) };
    }
    println!("{}", r);
}
```

Convert raw pointer to reference

```
use std::mem::transmute;
```

```
fn main() {  
    let r;  
    {  
        let x = 5;  
  
        let p = &x as *const _;  
  
        r = unsafe { transmute::<*const i32, &i32>(p) };  
    }  
    println!("{}", r);  
}
```

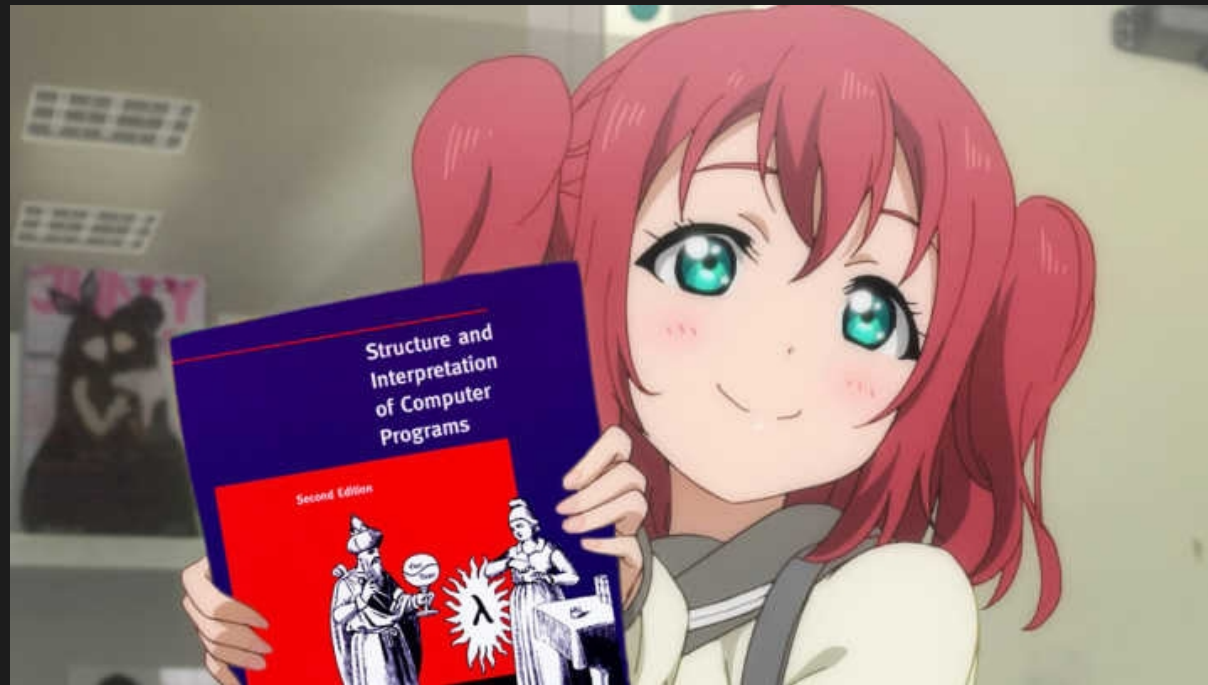
Convert raw pointer to reference

```
use std::mem::transmute;
```

```
fn main() {  
    let r;  
    {  
        let x = 5;  
  
        let p = &x as *const _;  
  
        r = unsafe { transmute::<*const i32, &i32>(p) };  
    }  
    println!("{}", r);  
}
```

Then you can get a dangling pointer ~

來自異世界的魔法：Macros



Macros

- 起源：Lisp
- 特色：Syntax extension (語法擴展)
- 目的：把 AST 轉換成另一個 AST ~
- Rust 的 Macros 分為兩類：
 1. Declarative Macro
 2. Procedural Macro

Macros

- 起源：Lisp
- 特色：Syntax extension (語法擴展)
- 目的：把 AST 轉換成另一個 AST ~
- Rust 的 Macros 分為兩類：
 1. Declarative Macro
 2. Procedural Macro

Q. Why it's hard to read or understand ?

A. You're writing Rust code that writes Rust code

Example

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {

        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Example

When match the token tree(tt) at left-hand side then expand the token tree at right-hand side

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Example

When match the token tree(tt) at left-hand side then expand the token tree at right-hand side

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Declarative Macro

```
( $lhs: tt ) => ( $rhs: tt ); +
```

Declarative Macro

`($lhs: tt) => ($rhs: tt); +`



`$(...) sep rep`

Declarative Macro

`($lhs: tt) => ($rhs: tt); +`

`$(...) sep rep`

Pattern 放裡面 ~

`'+' or '*'`
至少 1 個 至少 0 個

分隔用的符號，可以是 *Empty*

Example

Usage.

```
let x = vec![1,2,3];
```


```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Example

Usage.

```
let x = vec![1, 2, 3];
```

```
#[macro_export]
macro_rules! vec {
  ( $( $x:expr ),* ) => {
    {
      let mut temp_vec = Vec::new();
      $(
        temp_vec.push($x);
      )*
      temp_vec
    }
  };
}
```




```
temp_vec.push(1);
```

Example

Usage.

```
#[macro_export]
macro_rules! vec {
  ( $( $x:expr ),* ) => {
    {
      let mut temp_vec = Vec::new();
      $(
        temp_vec.push($x);
      )*
      temp_vec
    }
  };
}
```



```
let x = vec![1,2,3];
```


```
temp_vec.push(1);
temp_vec.push(2);
```

Example

Usage.

```
let x = vec![1,2,3];
```

```
#[macro_export]
macro_rules! vec {
  ( $( $x:expr ),* ) => {
    {
      let mut temp_vec = Vec::new();
      $(
        temp_vec.push($x);
      )*
      temp_vec
    }
  };
}
```



```
temp_vec.push(1);
temp_vec.push(2);
temp_vec.push(3);
```

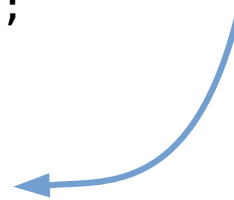
但位子 *SEP* 是空的，所以結束匹配

Example

Usage.

```
let x = vec![1,2,3];
```

```
let x = {  
  let mut temp_vec = Vec::new();  
  temp_vec.push(1);  
  temp_vec.push(2);  
  temp_vec.push(3);  
  temp_vec  
};
```



Problem

1. `vec! [1, 2, 3,]` is **not allowed**, how to modify it to accept this situation?
2. If you want to count the number of elements and initialize with **`with_capacity`** method, then how to modify it?

Solution 1

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* $($,$)* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```


Solution 2

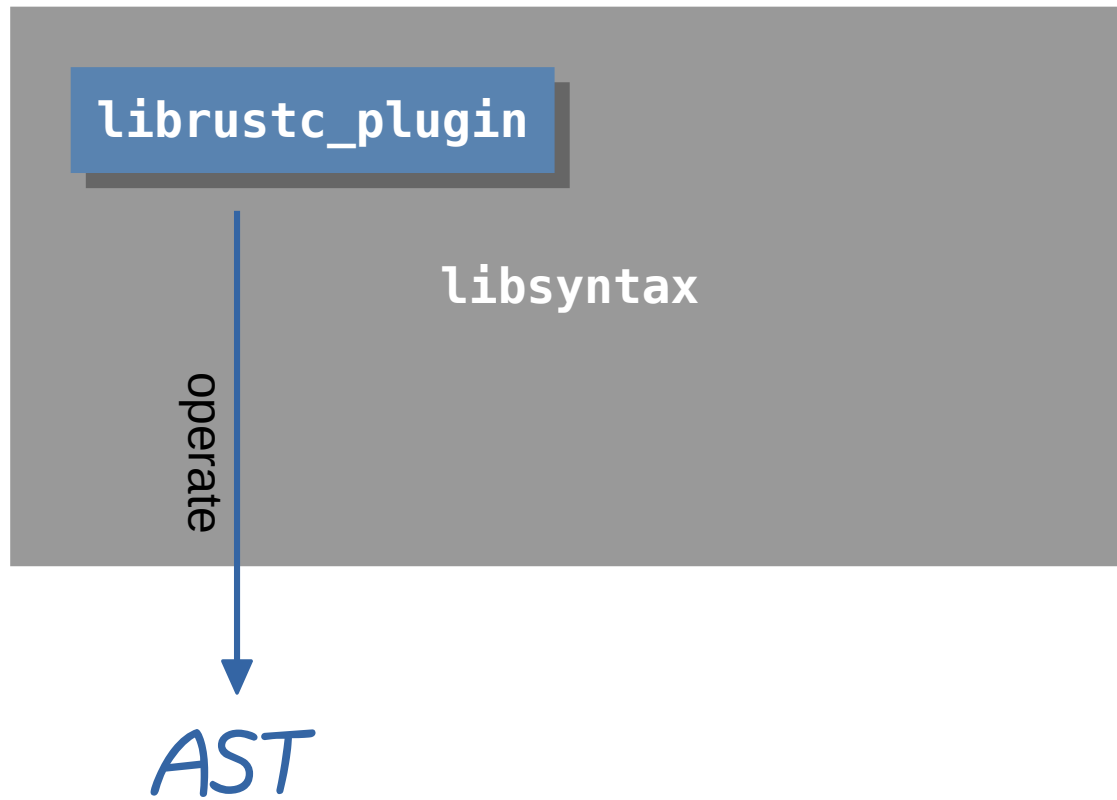
```
#[macro_export]
macro_rules! vec {
    (@unit $($x:tt)*) => (());

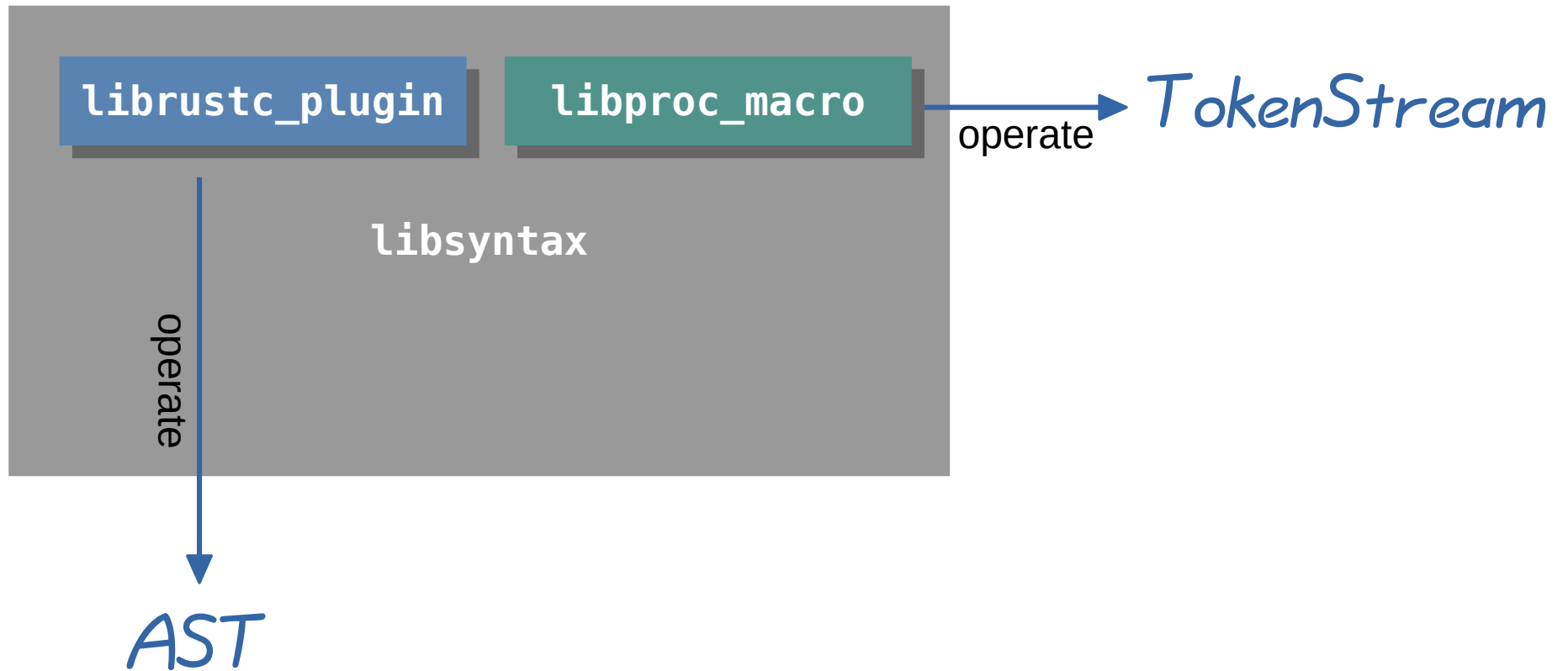
    (@count $($key:expr),*) =>
        (<[()]>::len(&$(vec!(@unit $key)),*]));

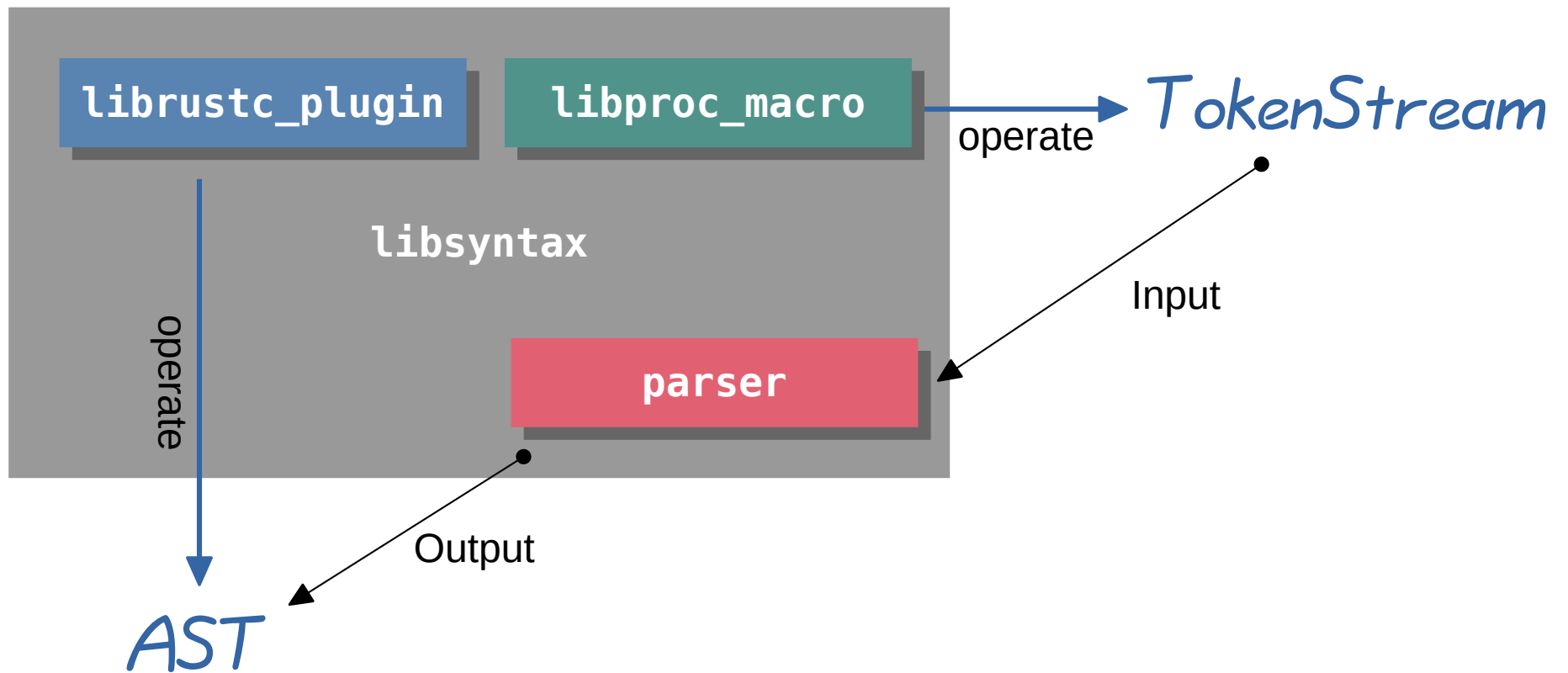
    ($($x:expr),* $(,)* ) => {
        {
            let _cap = vec!(@count $($x),*);
            let mut temp_vec = Vec::with_capacity(_cap);
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

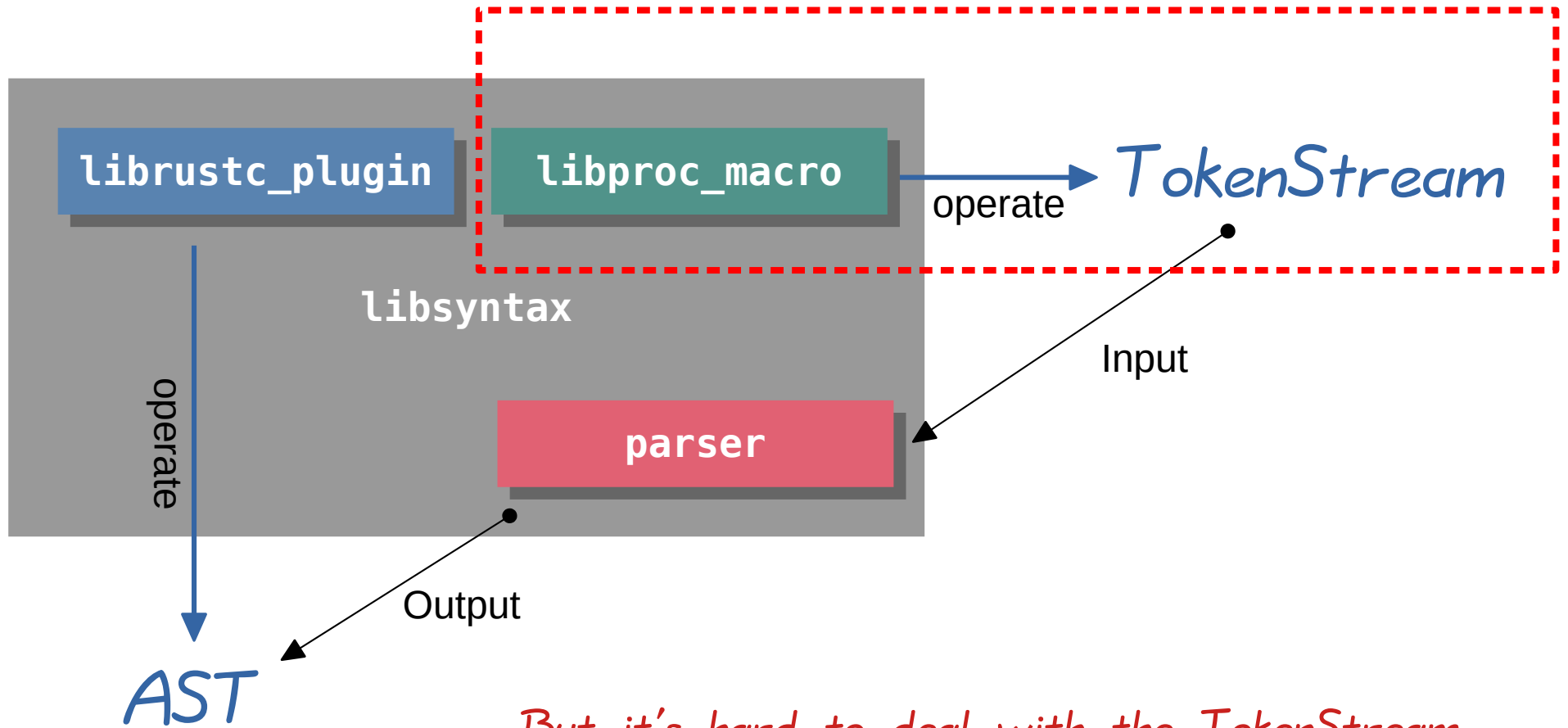
Procedural Macro

Goal : *Syntax extension !*



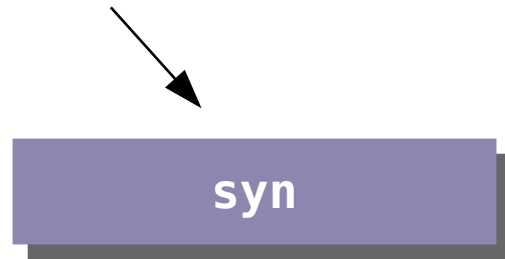




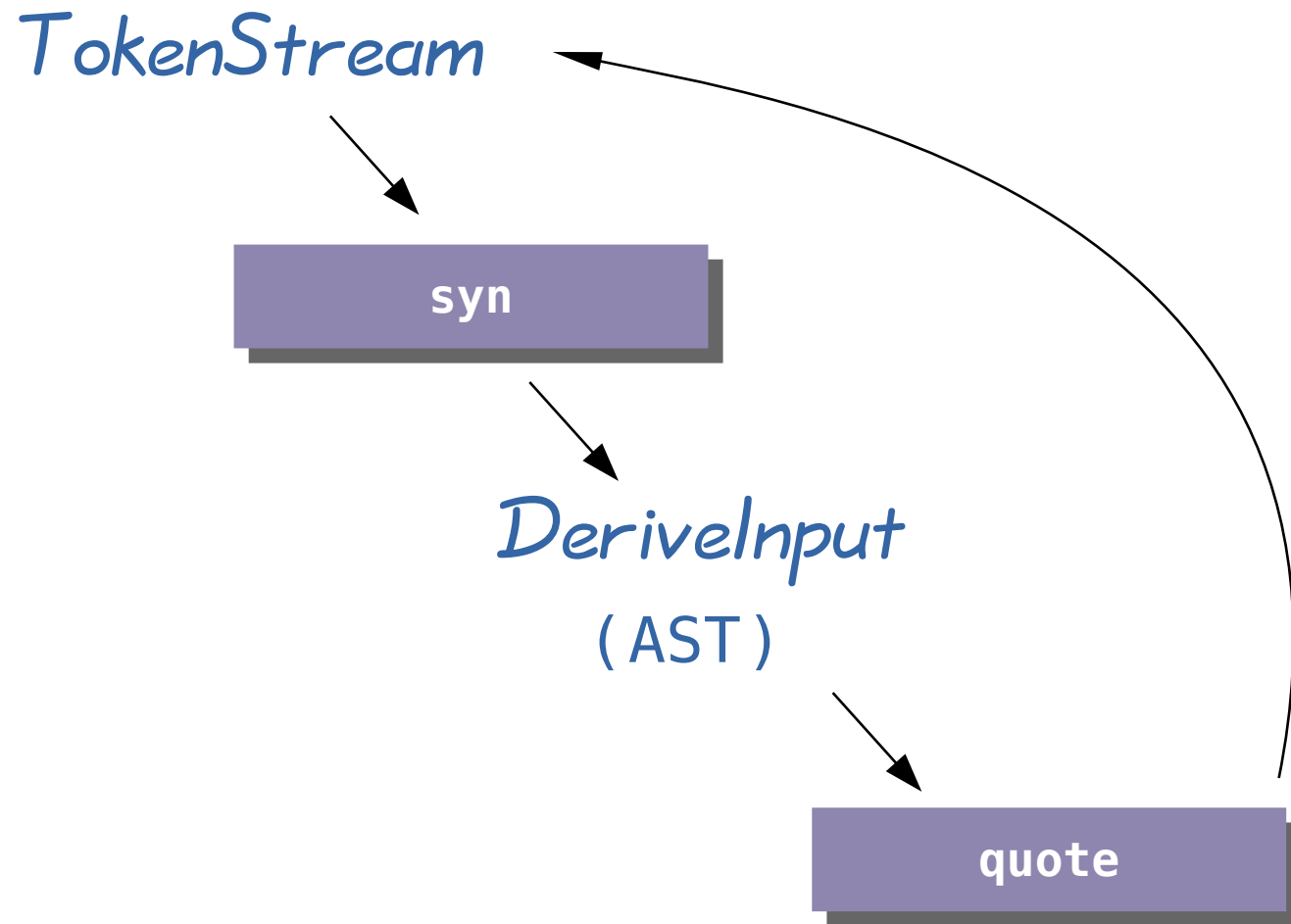


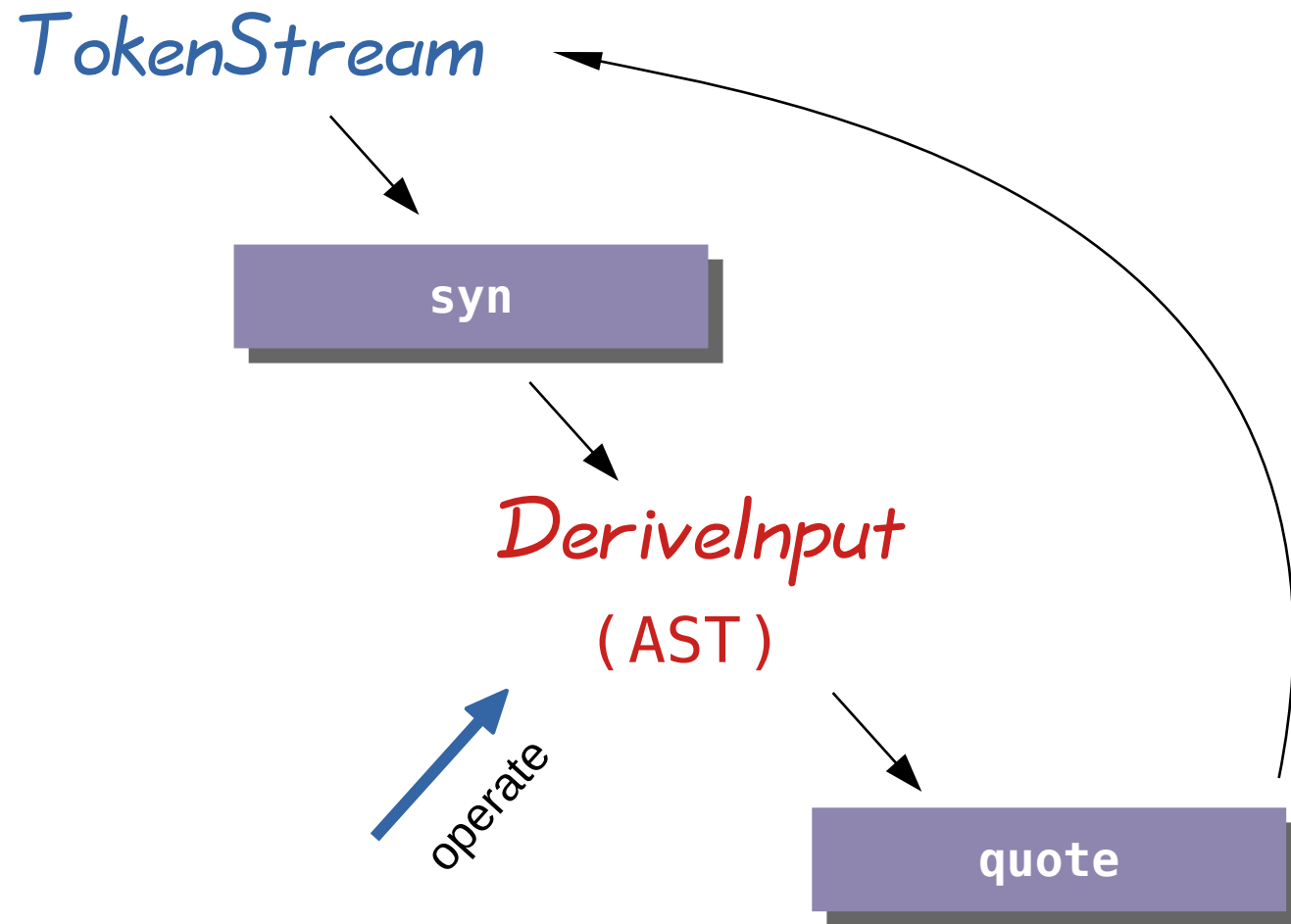
But it's hard to deal with the `TokenStream` directly

TokenStream



DeriveInput
(AST)





Example

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

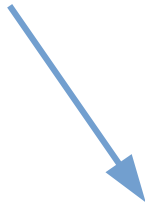
fn main() {
    Pancakes::hello_macro();
}
```

Example

```
use hello_macro::HelloMacro;  
use hello_macro_derive::HelloMacro;
```

```
#[derive(HelloMacro)]  
struct Pancakes;
```

```
fn main() {  
    Pancakes::hello_macro();  
}
```



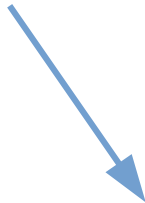
“Hello, Macro! My name is Pancakes”

Example

```
use hello_macro::HelloMacro;  
use hello_macro_derive::HelloMacro;
```

```
#[derive(HelloMacro)]  
struct Pancakes;
```

```
fn main() {  
    Pancakes::hello_macro();  
}
```




“Hello, Macro! My name is Pancakes”

Example

```
use hello_macro::HelloMacro;  
use hello_macro_derive::HelloMacro;
```

```
#[derive(HelloMacro)]  
struct Pancakes;  
  
fn main() {  
    Pancakes::hello_macro();  
}
```

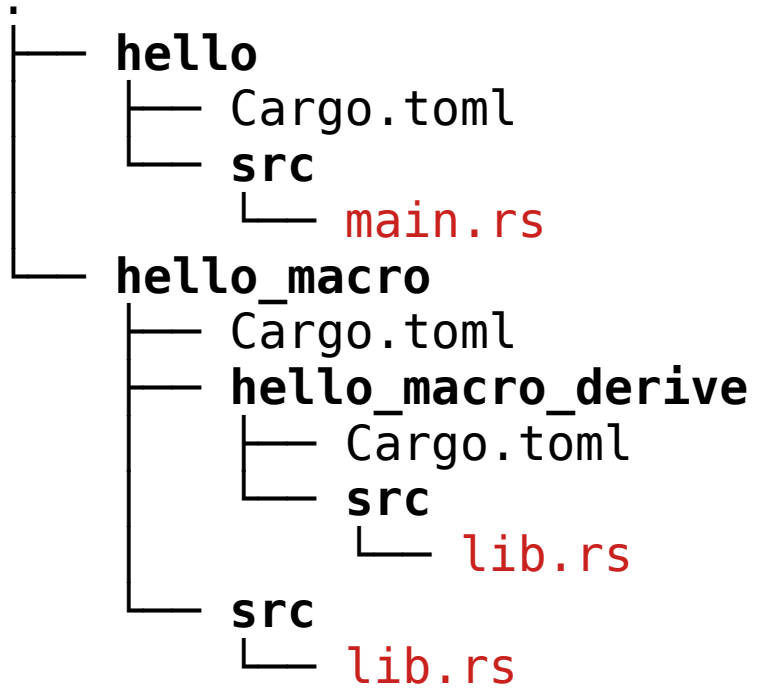


```
impl HelloMacro for Pancakes {  
    fn hello_macro() {  
        println!("Hello, Macro! My name is {}",  
            stringify!(Pancakes));  
    }  
}
```

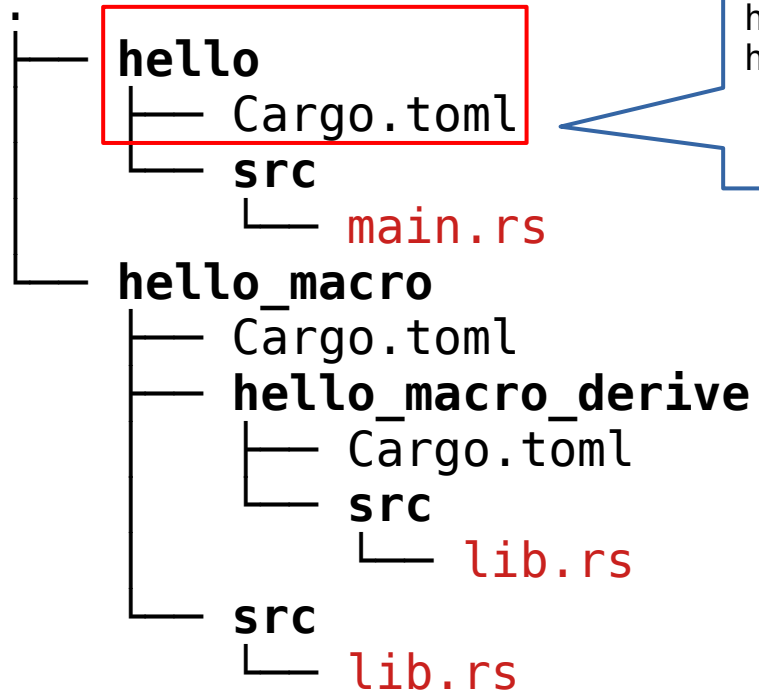


“Hello, Macro! My name is Pancakes”

Example



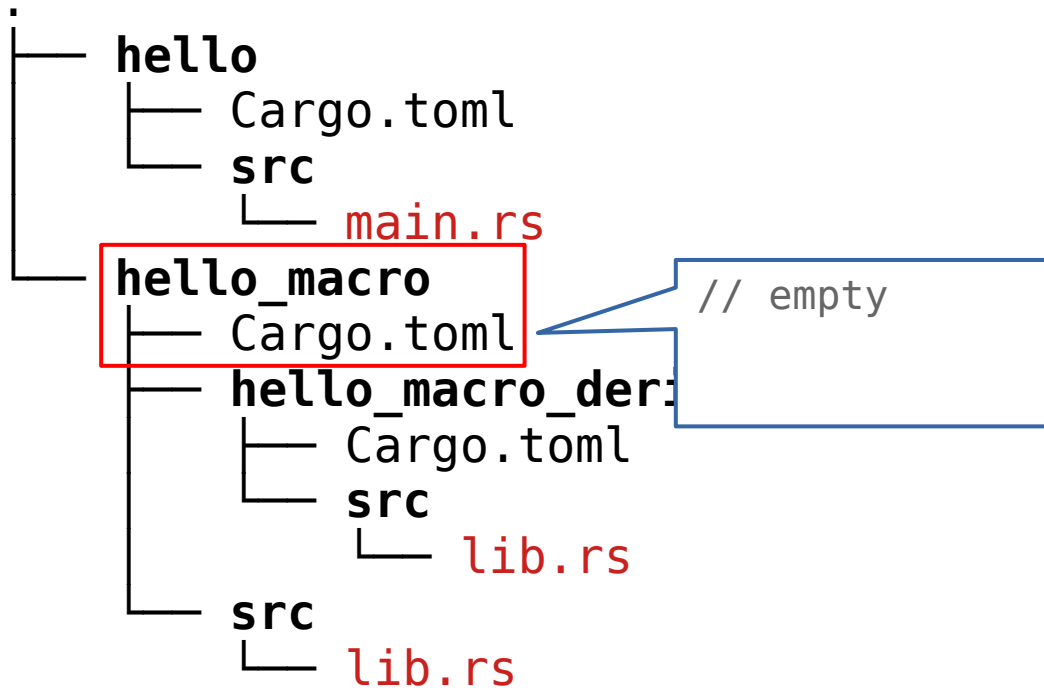
Example



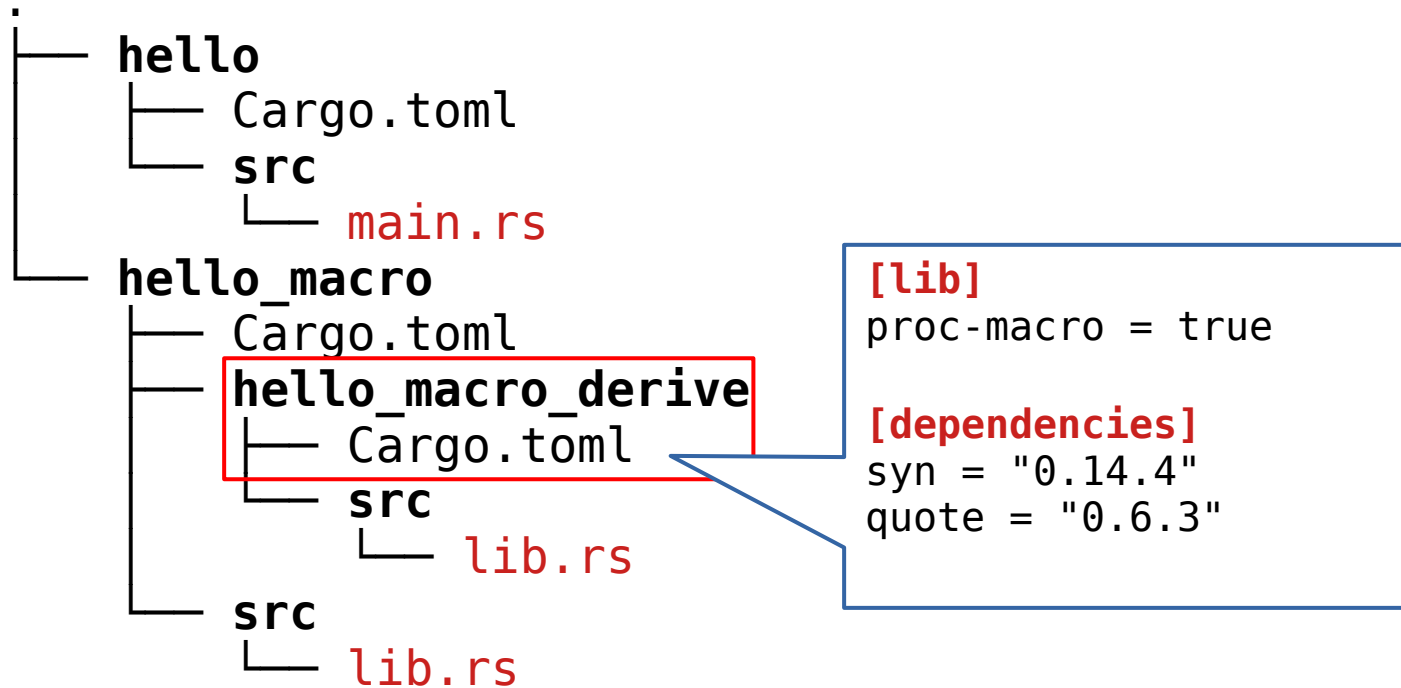
[dependencies]

```
hello_macro = { path = "../hello_macro" }
hello_macro_derive =
  { path = "../hello_macro/hello_macro_derive" }
```

Example



Example




```
pub trait HelloMacro {  
    fn hello_macro();  
}
```

[hello_macro/src/lib.rs](#)

```

extern crate proc_macro;

use crate::proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a representation of Rust code as a syntax tree
    // that we can manipulate
    let ast = syn::parse(input).unwrap();

    // Build the trait implementation
    impl_hello_macro(&ast)
}

fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}", stringify!(#name));
            }
        }
    };
    gen.into()
}

```

[hello_macro_derive/src/lib.rs](#)

```
extern crate proc_macro;
```

```
use crate::proc_macro::TokenStream;  
use quote::quote;  
use syn;
```

*It's corresponding to the
#[dervie(HellMacro)] in hello/src/main.rs*

```
#[proc_macro_derive(HelloMacro)]
```

```
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {  
    // Construct a representation of Rust code as a syntax tree  
    // that we can manipulate  
    let ast = syn::parse(input).unwrap();  
  
    // Build the trait implementation  
    impl_hello_macro(&ast)  
}
```

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {  
    let name = &ast.ident;  
    let gen = quote! {  
        impl HelloMacro for #name {  
            fn hello_macro() {  
                println!("Hello, Macro! My name is {}", stringify!(#name));  
            }  
        }  
    };  
    gen.into()  
}
```

[hello_macro_derive/src/lib.rs](#)

```

extern crate proc_macro;

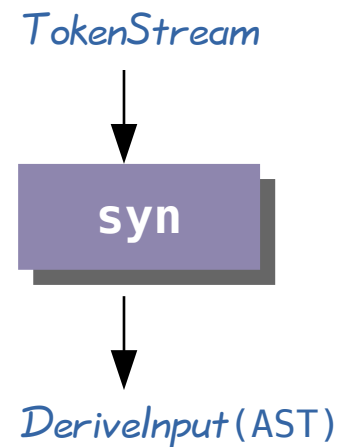
use crate::proc_macro::TokenStream;
use quote::quote;
use syn;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a representation of Rust code as a syntax tree
    // that we can manipulate
    let ast = syn::parse(input).unwrap();

    // Build the trait implementation
    impl_hello_macro(&ast)
}

fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}", stringify!(#name));
            }
        }
    };
    gen.into()
}

```



[hello_macro_derive/src/lib.rs](#)

```
extern crate proc_macro;
```

```
use crate::proc_macro::TokenStream;  
use quote::quote;  
use syn;
```

```
#[proc_macro_derive(HelloMacro)]
```

```
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {  
    // Construct a representation of Rust code as a syntax tree  
    // that we can manipulate  
    let ast = syn::parse(input).unwrap();
```

```
    // Build the trait implementation  
    impl_hello_macro(&ast)  
}
```

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {  
    let name = &ast.ident;  
    let gen = quote! {  
        impl HelloMacro for #name {  
            fn hello_macro() {  
                println!("Hello, Macro! My name is {}", stringify!(#name));  
            }  
        }  
    };  
    gen.into()  
}
```

TokenStream

syn

DeriveInput (AST)

Do Something \ (⊛°▽°) /

[hello_macro_derive/src/lib.rs](#)

```
extern crate proc_macro;
```

```
use crate::proc_macro::TokenStream;  
use quote::quote;  
use syn;
```

```
#[proc_macro_derive(HelloMacro)]
```

```
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {  
    // Construct a representation of Rust code as a syntax tree  
    // that we can manipulate  
    let ast = syn::parse(input).unwrap();
```

```
    // Build the trait implementation  
    impl_hello_macro(&ast)  
}
```

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {  
    let name = &ast.ident; Pancakes  
    let gen = quote! {  
        impl HelloMacro for            {  
            fn hello_macro() {  
                println!("Hello, Macro! My name is {}", stringify!(#name));  
            }  
        }  
    };  
    gen.into()  
}
```

TokenStream

syn

DeriveInput(AST)

Do Something \ (°▽°) /

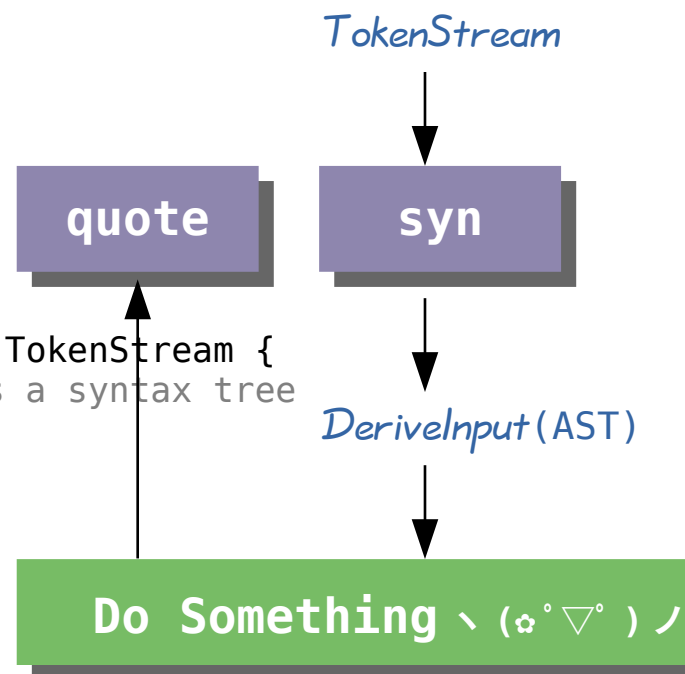
[hello_macro_derive/src/lib.rs](#)

```
extern crate proc_macro;
```

```
use crate::proc_macro::TokenStream;  
use quote::quote;  
use syn;
```

```
#[proc_macro_derive(HelloMacro)]  
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {  
    // Construct a representation of Rust code as a syntax tree  
    // that we can manipulate  
    let ast = syn::parse(input).unwrap();  
  
    // Build the trait implementation  
    impl_hello_macro(&ast)  
}
```

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {  
    let name = &ast.ident;  
    let gen = quote! {  
        impl HelloMacro for #name {  
            fn hello_macro() {  
                println!("Hello, Macro! My name is {}", stringify!(#name));  
            }  
        }  
    };  
    gen.into()  
}
```



[hello_macro_derive/src/lib.rs](#)

```
extern crate proc_macro;
```

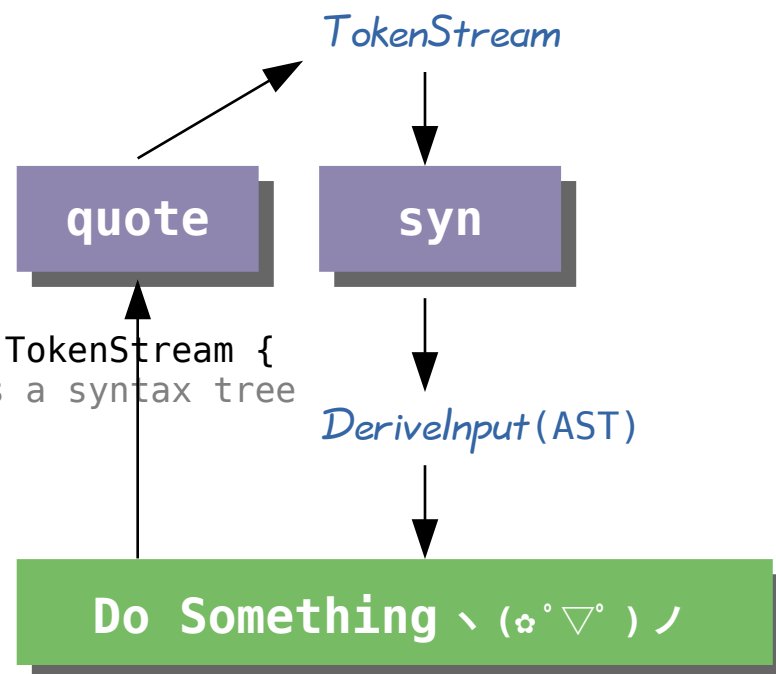
```
use crate::proc_macro::TokenStream;  
use quote::quote;  
use syn;
```

```
#[proc_macro_derive(HelloMacro)]
```

```
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {  
    // Construct a representation of Rust code as a syntax tree  
    // that we can manipulate  
    let ast = syn::parse(input).unwrap();
```

```
    // Build the trait implementation  
    impl_hello_macro(&ast)  
}
```

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {  
    let name = &ast.ident;  
    let gen = quote! {  
        impl HelloMacro for #name {  
            fn hello_macro() {  
                println!("Hello, Macro! My name is {}", stringify!(#name));  
            }  
        }  
    };  
    gen.into()  
}
```



[hello_macro_derive/src/lib.rs](#)

Attribute-like Macro

```
#[route(GET, "/")]  
fn index() { ... }
```

```
#[proc_macro_attribute]  
pub fn route(attr: TokenStream,  
              item: TokenStream) -> TokenStream  
{
```

Function-like Macro

Attribute-like Macro

```
#[route(GET, "/")]  
fn index() { ... }
```

```
#[proc_macro_attribute]  
pub fn route(attr: TokenStream,  
              item: TokenStream) -> TokenStream  
{
```

Function-like Macro

Attribute-like Macro

```
#[route(GET, "/")]  
fn index() { ... }
```

```
#[proc_macro_attribute]  
pub fn route(attr: TokenStream,  
             item: TokenStream) -> TokenStream  
{
```

Function-like Macro

Attribute-like Macro

```
#[route(GET, "/")]  
fn index() { ... }
```

```
#[proc_macro_attribute]  
pub fn route(attr: TokenStream,  
             item: TokenStream) -> TokenStream  
{
```

Function-like Macro

Attribute-like Macro

```
#[route(GET, "/")]  
fn index() { ... }
```

```
#[proc_macro_attribute]  
pub fn route(attr: TokenStream,  
              item: TokenStream) -> TokenStream  
{
```

Function-like Macro

```
let sql = sql!(  
  SELECT * FROM posts WHERE id=1  
);
```

```
#[proc_macro]  
pub fn sql(input: TokenStream) -> TokenStream  
{  
  ...
```

Attribute-like Macro

```
#[route(GET, "/")]  
fn index() { ... }
```

```
#[proc_macro_attribute]  
pub fn route(attr: TokenStream,  
              item: TokenStream) -> TokenStream  
{
```

Function-like Macro

```
let sql = sql!(  
  SELECT * FROM posts WHERE id=1  
);
```

```
#[proc_macro]  
pub fn sql(input: TokenStream) -> TokenStream  
{  
  ...
```

Lifetimes 我們很好 ~



```
struct Context(&str);

struct Parser {
    context: &Context,
}

impl Parser {
    fn parse(&self) -> Result<(), &str> {
        Err(&self.context.0[1..])
    }
}
```



```
struct Context(&str);

struct Parser {
    context: &Context,
}

impl Parser {
    fn parse(&self) -> Result<(), &str> {
        Err(&self.context.0[1..])
    }
}
```

Can not compile ! Missing lifetime specifier

```
struct Context<'a>(&'a str);

struct Parser<'a> {
    context: &'a Context<'a>,
}

impl<'a> Parser<'a> {
    fn parse<'b>(&'b self) -> Result<(), &'b str> {
        Err(&self.context.0[1..])
    }
}
```

```

struct Context<'a>(&'a str);

struct Parser<'a> {
    context: &'a Context<'a>,
}

impl<'a> Parser<'a> {
    fn parse<'b>(&'b self) -> Result<(), &'b str> {
        Err(&self.context.0[1..])
    }
}

```

And we added this function :

```

fn parse_context(context: Context) -> Result<(), &str> {
    Parser { context: &context }.parse()
}

```

```

struct Context<'a>(&'a str);

struct Parser<'a> {
    context: &'a Context<'a>,
}

impl<'a> Parser<'a> {
    fn parse<'b>(&'b self) -> Result<(), &'b str> {
        Err(&self.context.0[1..])
    }
}

```

And we added this function :

```

fn parse_context(context: Context) -> Result<(), &str> {
    Parser { context: &context }.parse()
}

```

Can not compile ! Returns a value referencing data owned by the current function

```
struct Context<'a>(&'a str);
```

```
struct Parser<'a> {  
    context: &'a Context<'a>,  
}
```

```
impl<'a> Parser<'a> {  
    fn parse<'b>(&'b self) -> Result<(), &'b str> {  
        Err(&self.context.0[1..])  
    }  
}
```

Transfer the context's ownership into function



```
fn parse_context(context: Context) -> Result<(), &str> {  
    Parser { context: &context }.parse()  
}
```

```
struct Context<'a>(&'a str);
```

```
struct Parser<'a> {  
    context: &'a Context<'a>,  
}
```

```
impl<'a> Parser<'a> {  
    fn parse<'b>(&'b self) -> Result<(), &'b str> {  
        Err(&self.context.0[1..])  
    }  
}
```

```
fn parse_context(context: Context) -> Result<(), &str> {  
    Parser { context: &context }.parse()  
}
```



Create a Parser, and borrow the context

```
struct Context<'a>(&'a str);
```

```
struct Parser<'a> {  
    context: &'a Context<'a>,  
}
```

```
impl<'a> Parser<'a> {  
    fn parse<'b>(&'b self) -> Result<(), &'b str> {  
        Err(&self.context.0[1..])  
    }  
}
```

```
fn parse_context(context: Context) -> Result<(), &str> {  
    Parser { context: &context }.parse()  
}
```



Create a Parser, and borrow the context

```
struct Context<'a>(&'a str);
```

```
struct Parser<'a> {  
    context: &'a Context<'a>,  
}
```



*We just need to let str and Context
have different lifetimes*

```
impl<'a> Parser<'a> {  
    fn parse<'b>(&'b self) -> Result<(), &'b str> {  
        Err(&self.context.0[1..])  
    }  
}
```

```
fn parse_context(context: Context) -> Result<(), &str> {  
    Parser { context: &context }.parse()  
}
```



```
struct Context<'a>(&'a str);
```

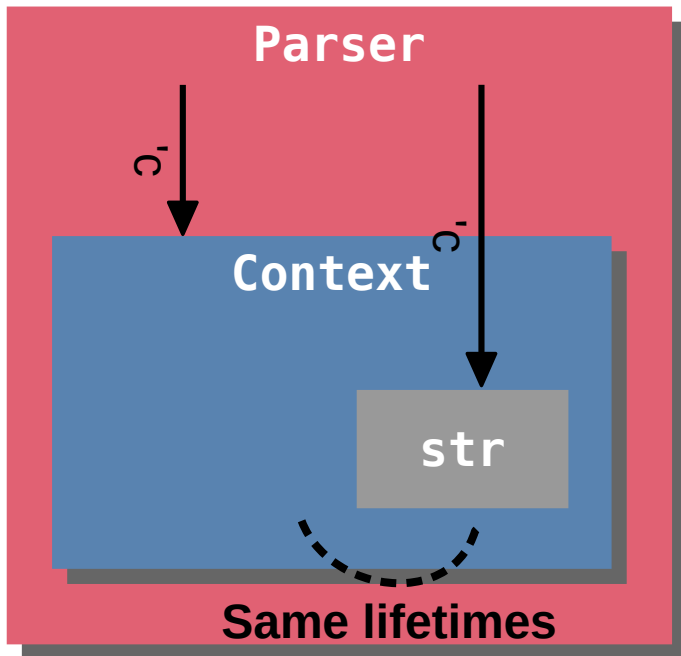
```
struct Parser<'c, 's> {  
    context: &'c Context<'s>,  
}
```

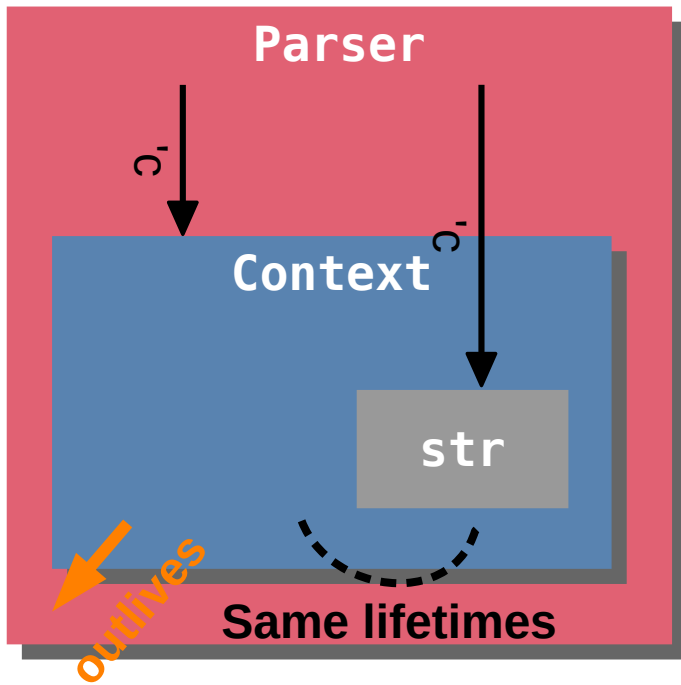


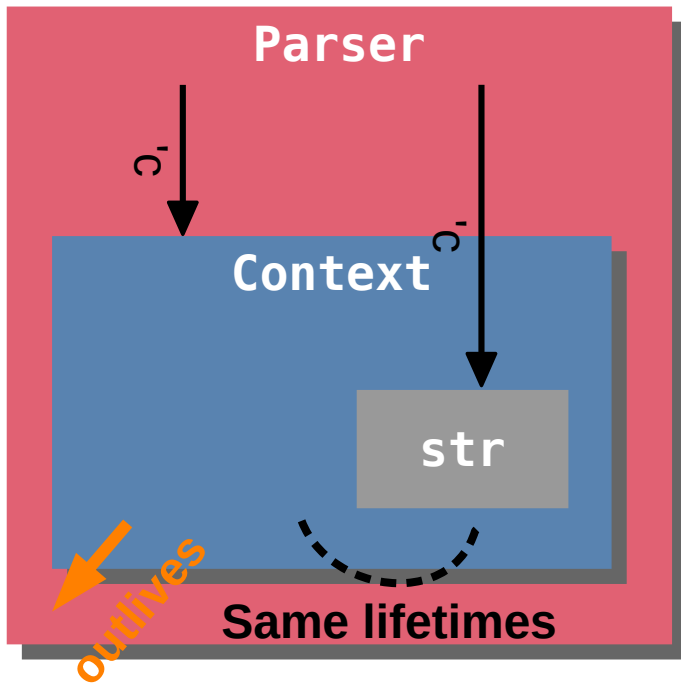
*We just need to let &str and Context
has different lifetimes*

```
impl<'c, 's> Parser<'c, 's> {  
    fn parse<'b>(&'b self) -> Result<(), &'s str> {  
        Err(&self.context.0[1..])  
    }  
}
```

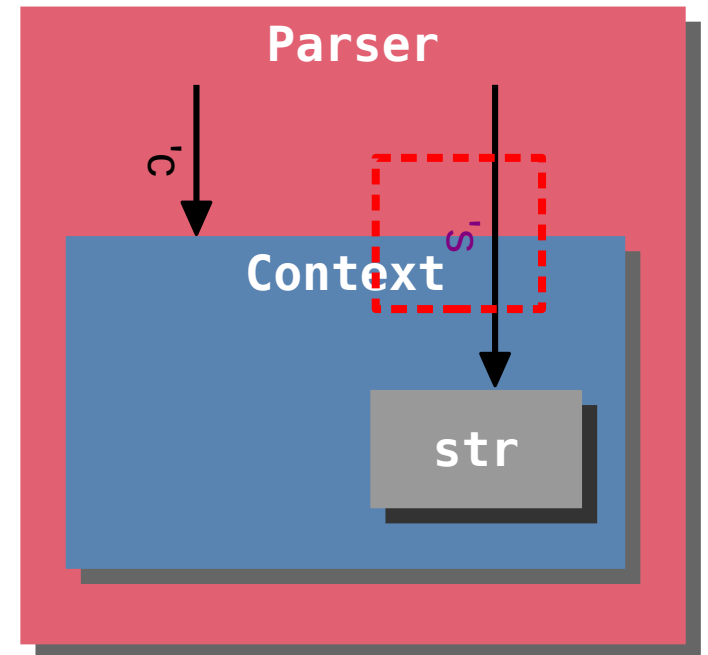
```
fn parse_context(context: Context) -> Result<(), &str> {  
    Parser { context: &context }.parse()  
}
```

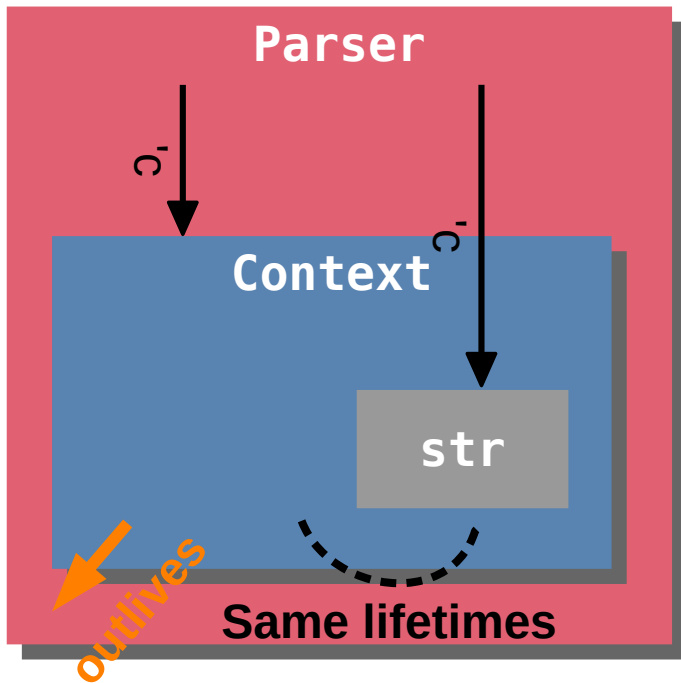




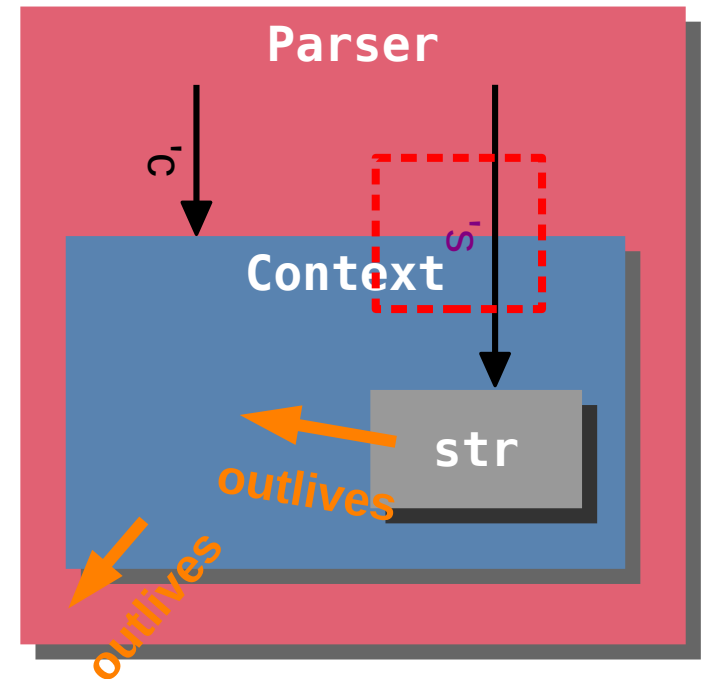


Specify the different lifetimes





Specify the different lifetimes



Problem 1 . Combine the Unsafe and Lifetimes

```
struct Cell<T>{value: T}

impl<T> Cell<T> {
    fn set(&self, v: T) {
        use std::ptr;
        let p = &self.value as *const _ as *mut _;

        unsafe {
            ptr::write(p, v);
        }
    }
}

fn evil(c: &Cell<i32>) {
    let drop_value = 42;

    c.set(&drop_value);
}
```

Problem 2. Combine the Unsafe and Lifetimes

```
struct Cell<T>{value: T}

impl<T> Cell<T> {
    fn set(&self, v: T) {
        use std::ptr;
        let p = &self.value as *const _ as *mut _;

        unsafe {
            ptr::write(p, v);
        }
    }
}

fn evil(c: &Cell<i32>) {
```

Why this is accepted by the Compiler ?

```
    let drop_value = 42;
    c.set(&drop_value);
}
```

drop_value dropped here

Another Problem

```
Use std::cell::Cell;
```

```
fn eat<'a>(x: &'a Cell::<&'a i32>) {}
```

```
fn main() {  
    let x = Cell::new(&5);  
    {  
        eat(&x);  
    }  
    x;  
}
```


Another Problem

Use `std::cell::Cell`;

```
fn eat<'a>(x: &'a Cell::<&'a i32>) {}
```

```
fn main() {  
    let x = Cell::new(&5);  
    {  
        eat(&x);  
    }    -- borrow of x occurs here  
    x;  
    ^  
    |  
    move out of x occurs here  
    borrow later used here
```

error[E0505]: cannot move out of x because it is borrowed

Hint

Subtyping and Variance

Reference.

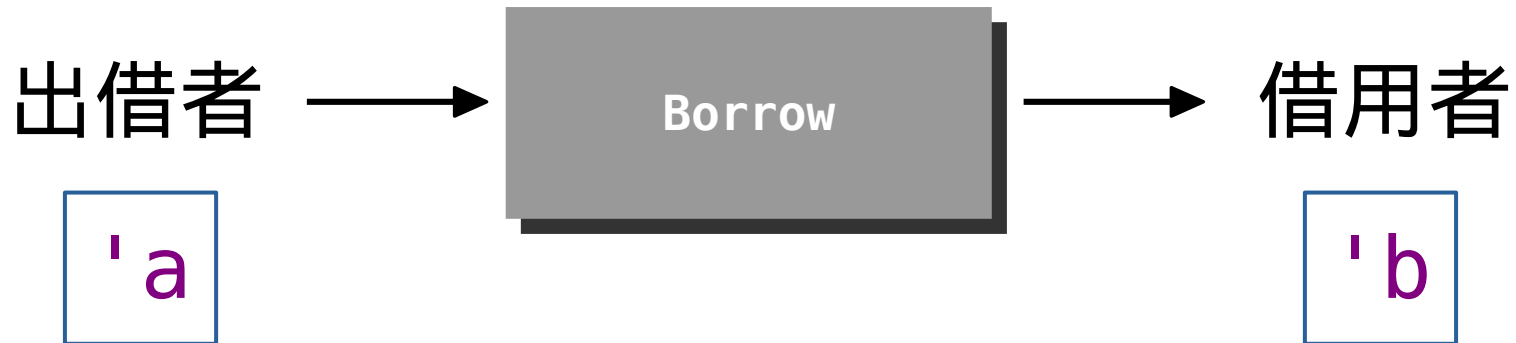
<https://doc.rust-lang.org/nomicon/subtyping.html>

Subtyping

- 寫法 : 'a : 'b
- 意思 : Lifetimes 'a is outlives Lifetimes 'b

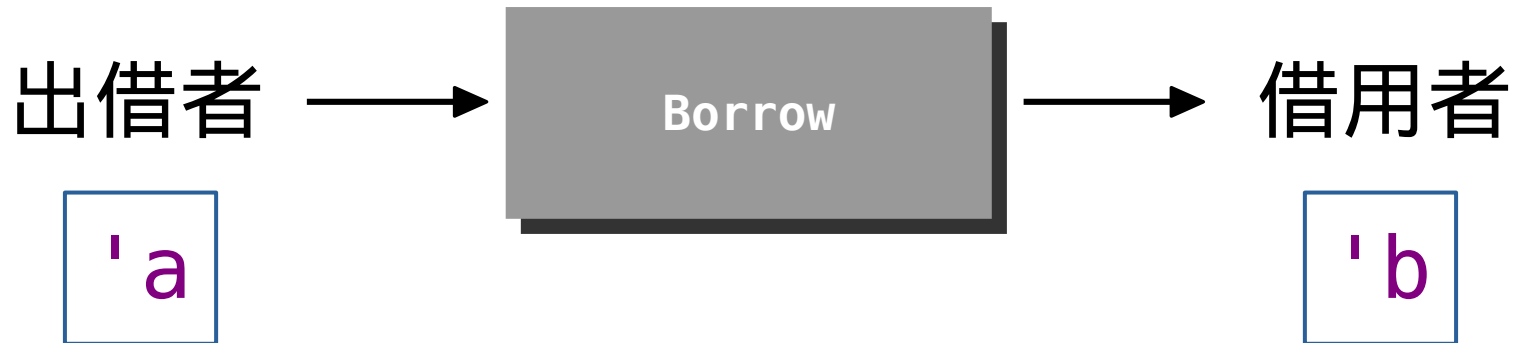
Subtyping

- 寫法： `'a : 'b`
- 意思： Lifetimes `'a` is outlives Lifetimes `'b`



Subtyping

- 寫法 : `'a : 'b`
- 意思 : Lifetimes `'a` is outlives Lifetimes `'b`



- 例子 : `let p: &'p T = &'foo foo;`

Then we say: `'foo : 'p`

```
struct Ref<'a, T: 'a>(&'a T);
```

```
struct Ref<'a, T: 'a>(&'a T);
```

T can be any type, if it contains any reference then it must live at least as long as 'a

Or we say Lifetimes of &T must outlives lifetimes 'a

```
struct Ref<'a, T: 'a>(&'a T);
```

```
let x = 1;
```

```
let r = Ref(&x);
```

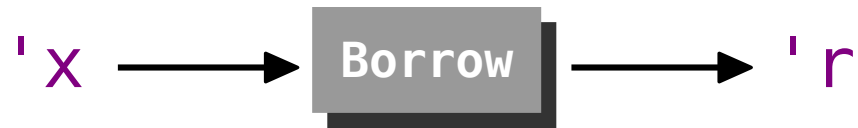
```
}
```



```
struct Ref<'a, T: 'a>(&'a T);
```

```
let x = 1;
```

```
let r: Ref(&'r T) = Ref(&'x x);
```

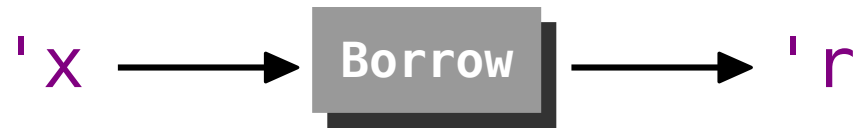


```
}
```

```
struct Ref<'a, T: 'a>(&'a T);
```

```
let x = 1;
```

```
let r: Ref(&'r T) = Ref(&'x x);
```



```
} ← Lifetimes of x end here
```

*The x is outlives r
So the Compiler accept this code*

```
struct Ref<'a, T: 'a>(&'a T);
```



*Actually, you don't need to specify
this today*

Types and Traits



Associated Type

- 目的：

1. Readable
2. Eliminate a bunch of Type annotation

- 用法：

```
trait Add<RHS=Self> {  
    type Output;  
    fn add(self, rhs: RHS) -> Self::Output;  
}
```

```
trait Add<RHS, Output> {  
    fn add(self, rhs: RHS) -> Output;  
}
```

```
impl Add<u32, u32> for u32 {  
    fn add(self, rhs: u32) -> u32 {  
        self + rhs  
    }  
}
```

```
let x: u32 = 1.add(2);
```

```
trait Add<RHS, Output> {  
    fn add(self, rhs: RHS) -> Output;  
}
```

```
impl Add<u32, u32> for u32 {  
    fn add(self, rhs: u32) -> u32 {  
        self + rhs  
    }  
}
```

```
let x: u32 = 1.add(2);
```



It's annoying

```
trait Add<RHS=Self> {  
    type Output;  
    fn add(self, rhs: RHS) -> Self::Output;  
}  
  
impl Add for u32 {  
    type Output = u32;  
    fn add(self, rhs: u32) -> u32 {  
        self + rhs  
    }  
}  
  
let x: u32 = 1.add(2);
```



```
trait Add<RHS=Self> {  
    type Output;  
    fn add(self, rhs: RHS) -> Self::Output;  
}  
  
impl Add for u32 {  
    type Output = u32;  
    fn add(self, rhs: u32) -> u32 {  
        self + rhs  
    }  
}  
  
let x: u32 = 1.add(2);
```

```
struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {

    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

```
let a = Millimeters(200);
```

```
let b = Meters(1);
```

```
let sum = a + b;
```

Millimeters *Meters*

UFCS

- 時機：通常發生在 Impl 多個 trait 時，method 名字衝突時，
Calling Methods with the Same Name

- 目的：可讀性更佳

- 方法：

```
<Type as Trait>::function(receiver_if_method,  
                             next_arg, ... );
```

```
trait Pilot { fn fly(&self); }

trait Wizard { fn fly(&self); }

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}
```

```
trait Pilot { fn fly(&self); }
```

```
trait Wizard { fn fly(&self); }
```

```
struct Human;
```

```
impl Pilot for Human {
```

```
    fn fly(&self) {
```

```
        println!("This is your captain speaking.");
```

```
    }
```

```
}
```

```
impl Wizard for Human {
```

```
    fn fly(&self) {
```

```
        println!("Up!");
```

```
    }
```

```
}
```

```
impl Human {
```

```
    fn fly(&self) {
```

```
        println!("*waving arms furiously*");
```

```
    }
```

```
}
```

飛行員和魔法少女都會飛

```
trait Pilot { fn fly(&self); }
```

```
trait Wizard { fn fly(&self); }
```

```
struct Human;
```

```
impl Pilot for Human {  
    fn fly(&self) {  
        println!("This is your captain speaking.");  
    }  
}
```

```
impl Wizard for Human {  
    fn fly(&self) {  
        println!("Up!");  
    }  
}
```

```
impl Human {  
    fn fly(&self) {  
        println!("*waving arms furiously*");  
    }  
}
```

人可以是飛行員或是魔法少女



或什麼也不是

```
let a = Human;  
a.fly();
```

1.

```
Pilot::fly(&a);  
Wizard::fly(&a);
```

2.

```
<Human as Pilot>::fly(&a);  
<Human as Wizard>::fly(&a);
```

```
let a = Human;  
a.fly();
```

1.

```
Pilot::fly(&a);  
Wizard::fly(&a);
```

2.

```
<Human as Pilot>::fly(&a);  
<Human as Wizard>::fly(&a);
```



Better readability

Function and Closures



Fn and fn

```
let list_of_numbers = vec![1, 2, 3];  
  
let list_of_strings: Vec<String> = list_of_numbers  
    .iter()  
    .map(|i| i.to_string())  
    .collect();
```

Fn and fn

```
let list_of_numbers = vec![1, 2, 3];
```

```
let list_of_strings: Vec<String> = list_of_numbers  
    .iter()  
    .map(|i| i.to_string())  
    .collect();
```

or  ToString::to_string

Because fn also implement Fn, FnMut, FnOnce trait

Fn and fn

```
let list_of_numbers = vec![1, 2, 3];  
  
let list_of_strings: Vec<String> = list_of_numbers  
    .iter()  
    .map(|i| i.to_string())  
    .collect();
```



```
["1", "2", "3"]
```

```
fn returns_closure() -> Fn(i32) -> i32 {  
    |x| x + 1  
}
```

```
fn returns_closure() -> Fn(i32) -> i32 {  
    |x| x + 1  
}
```



Because Closure is a Trait !

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {  
    Box::new(|x| x + 1)  
}
```

```
let c = returns_closure();  
println!("{}", c(1));
```

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {  
    Box::new(|x| x + 1)  
}
```

```
let c = returns_closure();  
println!("{}", c(1));
```



2


```
fn returns_closure() -> impl Fn(i32) -> i32 {  
    |x| x + 1  
}
```

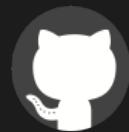
```
let c = returns_closure();  
println!("{}", c(1));
```



Actually, you can just use impl Trait

That's All

QA



`github.com/rniczh`