RUSTikales Rust for advanced coders

Plan for today

1. Recap
2. Weak<T>
3. Declarative Macros

1. Recap

1. Recap

- Ownership and Borrow Checker are very powerful
  - Guarantee memory safety at compile time

1. Recap

- Ownership and Borrow Checker are very powerful
  - Guarantee memory safety at compile time
- However, guarantees come with a price

→ If only one program is invalid and still compiles successfully, everything falls apart

→ Compiler must be strict and reject potentially valid code sometimes

1. Recap

- Ownership and Borrow Checker are very powerful
  - Guarantee memory safety at compile time
- However, guarantees come with a price
  → Compiler must be strict and reject potentially valid code sometimes
- If we know our code is valid, and it won't compile, we can use Smart Pointers to fix that
  - Instead of performing all checks at compile time, we do them at runtime
  - Our duty to make sure everything is correct, else our program panics at runtime

1. Recap

- Ownership and Borrow Checker are very powerful
  - Guarantee memory safety at compile time
- However, guarantees come with a price
  → Compiler must be strict and reject potentially valid code sometimes
- If we know our code is valid, and it won't compile, we can use Smart Pointers to fix that
- Rc<T> allows Shared Ownership
  - The underlying data T is reference counted and can be shared between many owners

1. Recap

- Ownership and Borrow Checker are very powerful
  - Guarantee memory safety at compile time
- However, guarantees come with a price
  → Compiler must be strict and reject potentially valid code sometimes
- If we know our code is valid, and it won't compile, we can use Smart Pointers to fix that
- Rc<T> allows Shared Ownership
- RefCell<T> uses Interior Mutability to check borrows at runtime
  - Using .[try_]borrow() and .[try_]borrow_mut() we can simulate &T and &mut T respectively

1. Recap

- Ownership and Borrow Checker are very powerful
  - Guarantee memory safety at compile time
- However, guarantees come with a price
  → Compiler must be strict and reject potentially valid code sometimes
- If we know our code is valid, and it won't compile, we can use Smart Pointers to fix that
- Rc<T> allows Shared Ownership
- RefCell<T> uses Interior Mutability to check borrows at runtime
  - Using .[try_]borrow() and .[try_]borrow_mut() we can simulate &T and &mut T respectively
  - Interior Mutability
    - We only use &RefCell<T> in our whole codebase to pass the Borrow Checker at compile time
    - Internally, we can still modify the underlying data T

1. Recap

- Ownership and Borrow Checker are very powerful
  - Guarantee memory safety at compile time
- However, guarantees come with a price
  → Compiler must be strict and reject potentially valid code sometimes
- If we know our code is valid, and it won't compile, we can use Smart Pointers to fix that
- Rc<T> allows Shared Ownership
- RefCell<T> uses Interior Mutability to check borrows at runtime
- Combinations of Smart Pointers often have desired effects
  - Rc<RefCell<T>> → Share multiple mutable references to the underlying data

```rust
use std::rc::Rc;
fn func(rc: Rc<u64>, depth: u8) {
    println!("Function: {}", Rc::strong_count(&rc));
    if depth < 3 {
        func(Rc::clone(self: &rc), depth: depth + 1);
    }
}
▶ Run | Debug
fn main() {
    let data: Rc<u64> = Rc::new(5u64);
    {
        let rc: Rc<u64> = Rc::clone(self: &data);
        println!("Block: {}", Rc::strong_count(&rc));
    }
    func(Rc::clone(self: &data), depth: 0);
    println!("After: {}", Rc::strong_count(&data));
}
```

```rust
use std::rc::Rc;
fn func(rc: Rc<u64>, depth: u8) {
    println!("Function: {}", Rc::strong_count(&rc));
    if depth < 3 {
        func(Rc::clone(self: &rc), depth: depth + 1);
    }
}
▶ Run | Debug
fn main() {
    let data: Rc<u64> = Rc::new(5u64);
    {
        let rc: Rc<u64> = Rc::clone(self: &data);
        println!("Block: {}", Rc::strong_count(&rc));
    }
    func(Rc::clone(self: &data), depth: 0);
    println!("After: {}", Rc::strong_count(&data));
}
```

Create Rc<T>

```rust
use std::rc::Rc;
fn func(rc: Rc<u64>, depth: u8) {
    println!("Function: {}", Rc::strong_count(&rc));
    if depth < 3 {
        func(Rc::clone(self: &rc), depth: depth + 1);
    }
}
▶ Run | Debug
fn main() {
    let data: Rc<u64> = Rc::new(5u64);
    {
        let rc: Rc<u64> = Rc::clone(self: &data);
        println!("Block: {}", Rc::strong_count(&rc));
    }
    func(Rc::clone(self: &data), depth: 0);
    println!("After: {}", Rc::strong_count(&data));
}
```

Share data at many points in our code

1. Recap

```rust
use std::rc::Rc;
fn func(rc: Rc<u64>, depth: u8) {
    println!("Function: {}", Rc::strong_count(&rc));
    if depth < 3 {
        func(Rc::clone(self: &rc), depth: depth + 1);
    }
}
▶ Run | Debug
fn main() {
    let data: Rc<u64> = Rc::new(5u64);
    {
        let rc: Rc<u64> = Rc::clone(self: &data);
        println!("Block: {}", Rc::strong_count(&rc));
    }
    func(Rc::clone(self: &data), depth: 0);
    println!("After: {}", Rc::strong_count(&data));
}
```

```
Block: 2
Function: 2
Function: 3
Function: 4
Function: 5
After: 1
```

```rust
use std::cell::RefCell;
▶ Run | Debug
fn main() {
    let data: RefCell<u64> = RefCell::new(42u64);
    {
        let brw: Ref<u64> = data.borrow();
        println!("Before: {}", brw);
    }
    {
        let mut brw_mut: RefMut<u64> = data.borrow_mut();
        *brw_mut = 100;
        println!("Changed!");
    }
    {
        let brw: Ref<u64> = data.borrow();
        println!("After: {}", brw);
    }
}
```

```rust
use std::cell::RefCell;
▶ Run | Debug
fn main() {
    let data: RefCell<u64> = RefCell::new(42u64);
    {
        let brw: Ref<u64> = data.borrow();
        println!("Before: {}", brw);
    }
    {
        let mut brw_mut: RefMut<u64> = data.borrow_mut();
        *brw_mut = 100;
        println!("Changed!");
    }
    {
        let brw: Ref<u64> = data.borrow();
        println!("After: {}", brw);
    }
}
```

Create RefCell<T>

```rust
use std::cell::RefCell;
▶ Run | Debug
fn main() {
    let data: RefCell<u64> = RefCell::new(42u64);
    {
        let brw: Ref<u64> = data.borrow();
        println!("Before: {}", brw);
    }
    {
        let mut brw_mut: RefMut<u64> = data.borrow_mut();
        *brw_mut = 100;
        println!("Changed!");
    }
    {
        let brw: Ref<u64> = data.borrow();
        println!("After: {}", brw);
    }
}
```

Borrows are valid until the value is dropped

→ No violations here, everything is good

1. Recap

```rust
use std::cell::RefCell;
▶ Run | Debug
fn main() {
    let data: RefCell<u64> = RefCell::new(42u64);
    {
        let brw: Ref<u64> = data.borrow();
        println!("Before: {}", brw);
    }
    {
        let mut brw_mut: RefMut<u64> = data.borrow_mut();
        *brw_mut = 100;
        println!("Changed!");
    }
    {
        let brw: Ref<u64> = data.borrow();
        println!("After: {}", brw);
    }
}
```

```
Before: 42
Changed!
After: 100
```

```rust
fn change(rc: Rc<RefCell<u64>>, val: u64) {
    let mut brw_mut: RefMut<u64> = rc.borrow_mut();
    *brw_mut = val;
}
fn print(rc: Rc<RefCell<u64>>) {
    let brw: Ref<u64> = rc.borrow();
    println!("Current: {}", brw);
}
fn reset(rc: Rc<RefCell<u64>>) {
    change(Rc::clone(self: &rc), val: 0);
    print(Rc::clone(self: &rc));
}
▶ Run | Debug
fn main() {
    let data: Rc<RefCell<u64>> = Rc::new(RefCell::new(5u64));
    print(Rc::clone(self: &data));
    change(Rc::clone(self: &data), val: 42);
    print(Rc::clone(self: &data));
    reset(Rc::clone(self: &data));
}
```

```
Current: 5
Current: 42
Current: 0
```

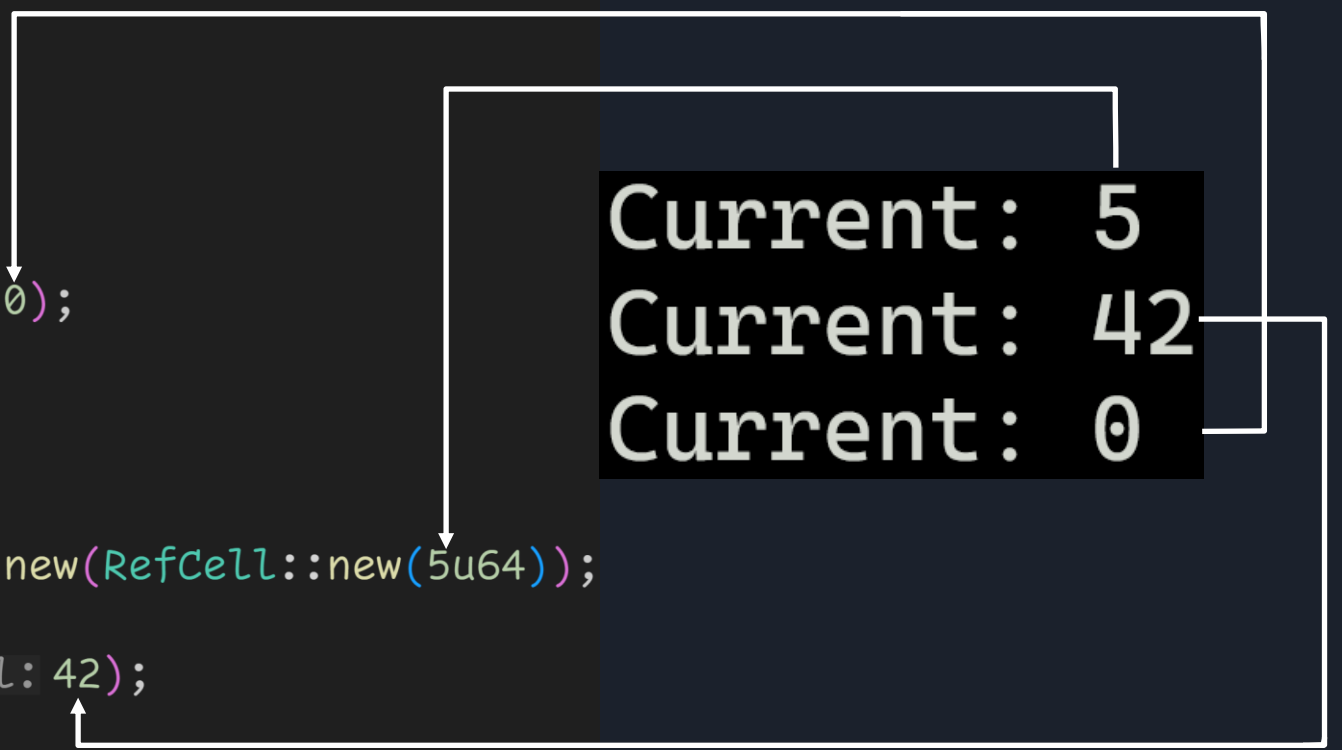
```rust
fn change(rc: Rc<RefCell<u64>>, val: u64) {
    let mut brw_mut: RefMut<u64> = rc.borrow_mut();
    *brw_mut = val;
}
fn print(rc: Rc<RefCell<u64>>) {
    let brw: Ref<u64> = rc.borrow();
    println!("Current: {}", brw);
}
fn reset(rc: Rc<RefCell<u64>>) {
    change(Rc::clone(self: &rc), val: 0);
    print(Rc::clone(self: &rc));
}
▶ Run | Debug
fn main() {
    let data: Rc<RefCell<u64>> = Rc::new(RefCell::new(5u64));
    print(Rc::clone(self: &data));
    change(Rc::clone(self: &data), val: 42);
    print(Rc::clone(self: &data));
    reset(Rc::clone(self: &data));
}
```

Use wherever we want

```
Current: 5
Current: 42
Current: 0
```

```rust
fn change(rc: Rc<RefCell<u64>>, val: u64) {
    let mut brw_mut: RefMut<u64> = rc.borrow_mut();
    *brw_mut = val;
}
fn print(rc: Rc<RefCell<u64>>) {
    let brw: Ref<u64> = rc.borrow();
    println!("Current: {}", brw);
}
fn reset(rc: Rc<RefCell<u64>>) {
    change(Rc::clone(self: &rc), val: 0);
    print(Rc::clone(self: &rc));
}
▶ Run | Debug
fn main() {
    let data: Rc<RefCell<u64>> = Rc::new(RefCell::new(5u64));
    print(Rc::clone(self: &data));
    change(Rc::clone(self: &data), val: 42);
    print(Rc::clone(self: &data));
    reset(Rc::clone(self: &data));
}
```

Change wherever we want



2. Weak<T>

2. Weak<T>

- `Rc<T>` can easily cause reference cycles, which means we never drop values → Memory leaks

2. Weak<T>

```rust
impl<T: Display> Drop for Data<T> {
    fn drop(&mut self) {
        println!("Dropped {}", self.value);
    }
}
type Wrap<T> = Rc<RefCell<Data<T>>>;
```

2 implementations

```rust
struct Data<T: Display> {
    value: T,
    next: Option<Wrap<T>>,
}
```

▶ Run | Debug

```rust
fn main() {
    let d1: Data<i32> = Data::new(5);
}
```

2. Weak<T>

```rust
impl<T: Display> Drop for Data<T> {
    fn drop(&mut self) {
        println!("Dropped {}", self.value);
    }
}
type Wrap<T> = Rc<RefCell<Data<T>>>;
2 implementations
struct Data<T: Display> {
    value: T,
    next: Option<Wrap<T>>,
}
▶ Run | Debug
fn main() {
    let d1: Data<i32> = Data::new(5);
}
```

d1	
value	5
next	None

2. Weak<T>

```rust
impl<T: Display> Drop for Data<T> {
    fn drop(&mut self) {
        println!("Dropped {}", self.value);
    }
}
type Wrap<T> = Rc<RefCell<Data<T>>>;
2 implementations
struct Data<T: Display> {
    value: T,
    next: Option<Wrap<T>>,
}
▶ Run | Debug
fn main() {
    let d1: Data<i32> = Data::new(5);
}
```

d1	
value	5
next	None

```
    Running
Dropped 5

C:\Users\pfh
```

2. Weak<T>

```rust
type Wrap<T> = Rc<RefCell<Data<T>>>;
2 implementations
struct Data<T: Display> {
    value: T,
    next: Option<Wrap<T>>,
}
▶ Run | Debug
fn main() {
    let d1: Data<i32> = Data::new(5);
    let rc: Wrap<i32> = Rc::new(RefCell::new(d1));
    rc.borrow_mut().next = Some(Rc::clone(self: &rc));
}
```

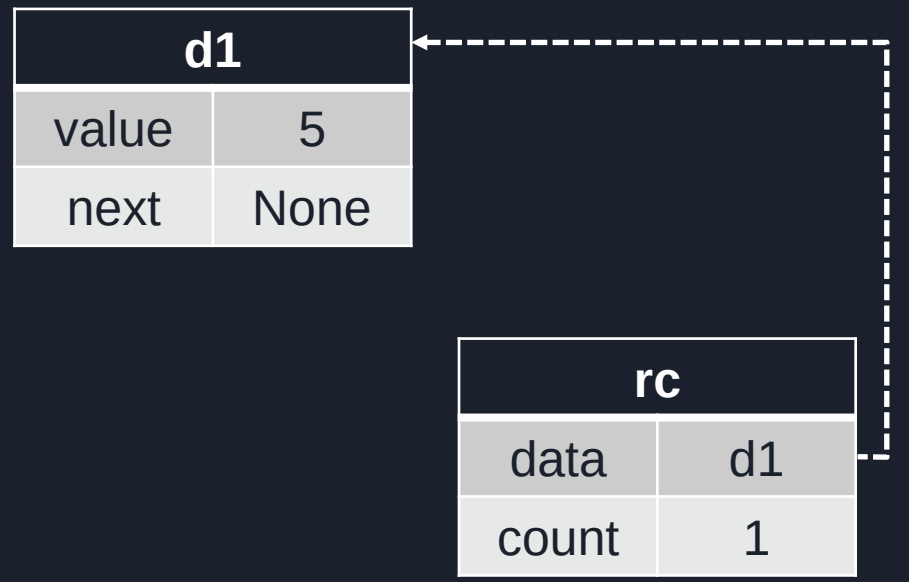
2. Weak<T>

```rust
type Wrap<T> = Rc<RefCell<Data<T>>>;
2 implementations
struct Data<T: Display> {
    value: T,
    next: Option<Wrap<T>>,
}
▶ Run | Debug
fn main() {
    let d1: Data<i32> = Data::new(5);
    let rc: Wrap<i32> = Rc::new(RefCell::new(d1));
    rc.borrow_mut().next = Some(Rc::clone(self: &rc));
}
```

d1	
value	5
next	None

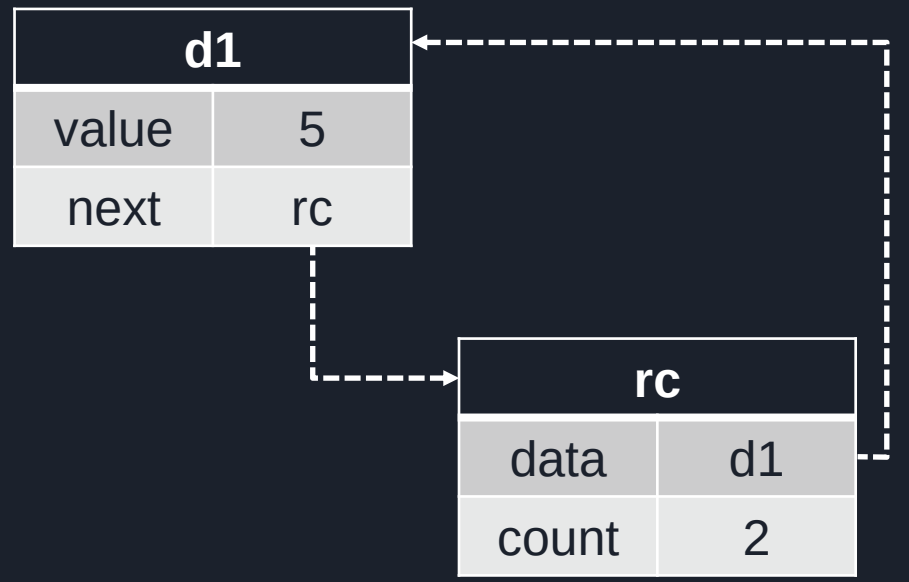
2. Weak<T>

```rust
type Wrap<T> = Rc<RefCell<Data<T>>>;
2 implementations
struct Data<T: Display> {
    value: T,
    next: Option<Wrap<T>>,
}
▶ Run | Debug
fn main() {
    let d1: Data<i32> = Data::new(5);
    let rc: Wrap<i32> = Rc::new(RefCell::new(d1));
    rc.borrow_mut().next = Some(Rc::clone(self: &rc));
}
```



2. Weak<T>

```rust
type Wrap<T> = Rc<RefCell<Data<T>>>;
2 implementations
struct Data<T: Display> {
    value: T,
    next: Option<Wrap<T>>,
}
▶ Run | Debug
fn main() {
    let d1: Data<i32> = Data::new(5);
    let rc: Wrap<i32> = Rc::new(RefCell::new(d1));
    rc.borrow_mut().next = Some(Rc::clone(self: &rc));
}
```

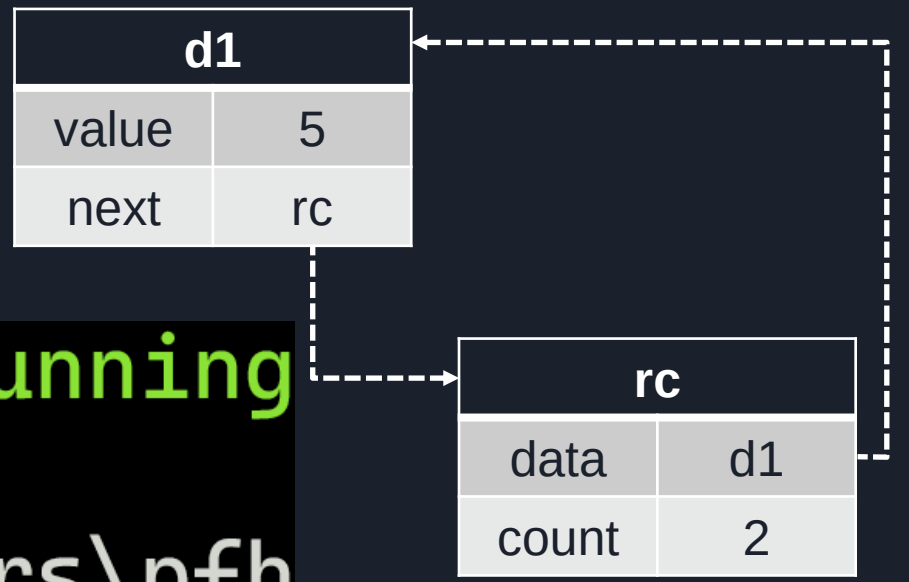


2. Weak<T>

```rust
type Wrap<T> = Rc<RefCell<Data<T>>>;
2 implementations
struct Data<T: Display> {
    value: T,
    next: Option<Wrap<T>>,
}
▶ Run | Debug
fn main() {
    let d1: Data<i32> = Data::new(5);
    let rc: Wrap<i32> = Rc::new(RefCell::new(d1));
    rc.borrow_mut().next = Some(Rc::clone(self: &rc));
}
```

```
Running

C:\Users\pfh
```

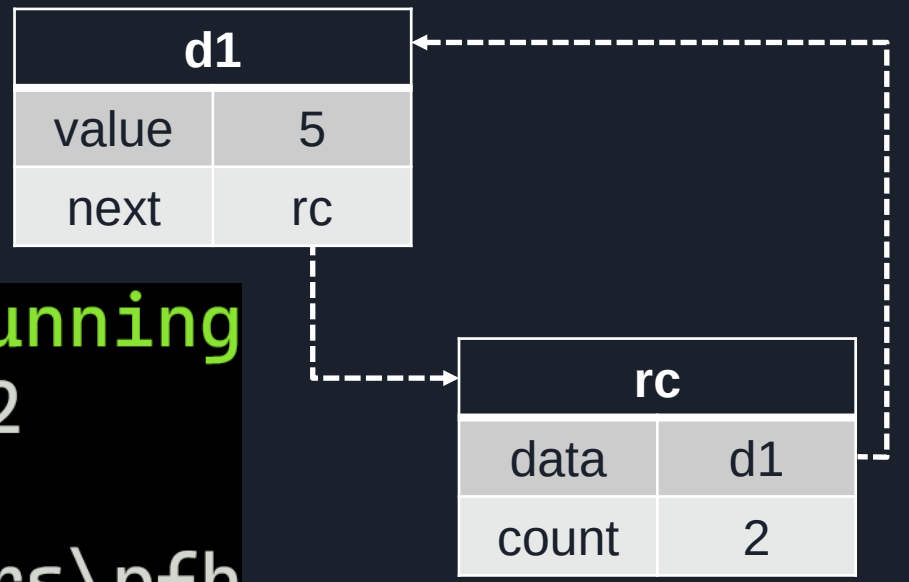


2. Weak<T>

```rust
type Wrap<T> = Rc<RefCell<Data<T>>>;
2 implementations
struct Data<T: Display> {
    value: T,
    next: Option<Wrap<T>>,
}
▶ Run | Debug
fn main() {
    let d1: Data<i32> = Data::new(5);
    let rc: Wrap<i32> = Rc::new(RefCell::new(d1));
    rc.borrow_mut().next = Some(Rc::clone(self: &rc));
}   println!("count={}", Rc::strong_count(&rc));
```

```
Running
count=2

C:\Users\pfh
```



2. Weak<T>

- Rc<T> can easily cause reference cycles, which means we never drop values → Memory leaks
- For that reason, Rc<T> has a downgraded version of itself: Weak<T>

2. Weak<T>

- Rc<T> can easily cause reference cycles, which means we never drop values → Memory leaks
- For that reason, Rc<T> has a downgraded version of itself: Weak<T>
- In reality, Rc<T> keeps track of two reference counts
  - strong_count → How many Rc<T> are there?
  - weak_count → How many Weak<T> are there?

2. Weak<T>

- Rc<T> can easily cause reference cycles, which means we never drop values → Memory leaks
- For that reason, Rc<T> has a downgraded version of itself: Weak<T>
- In reality, Rc<T> keeps track of two reference counts
  - strong_count → How many Rc<T> are there?
  - weak_count → How many Weak<T> are there?
- Rc<T> drops the value once the strong_count is 0, weak_count doesn't matter

2. Weak<T>

- Rc<T> can easily cause reference cycles, which means we never drop values → Memory leaks
- For that reason, Rc<T> has a downgraded version of itself: Weak<T>
- In reality, Rc<T> keeps track of two reference counts
  - strong_count → How many Rc<T> are there?
  - weak_count → How many Weak<T> are there?
- Rc<T> drops the value once the strong_count is 0, weak_count doesn't matter
- Weak means not-binding, we can have Weak<T> references to dropped values
  - Because of that, we can't just access the data of Weak<T>
  - But we can try and upgrade Weak<T> to Rc<T> at any time

2. Weak<T>

```rust
fn main() {
    let rc: Rc<i32> = Rc::new(5);
    let wc: Weak<i32> = Rc::downgrade(this: &rc);
    drop(rc);
    if let Some(data: Rc<i32>) = Weak::upgrade(self: &wc) {
        println!("Data has not been dropped: {}", data);
    } else {
        println!("Data was dropped :(");
    }
}
```

2. Weak<T>

```rust
fn main() {
    let rc: Rc<i32> = Rc::new(5);
    let wc: Weak<i32> = Rc::downgrade(this: &rc);
    drop(rc);
    if let Some(data: Rc<i32>) = Weak::upgrade(self: &wc) {
        println!("Data has not been dropped: {}", data);
    } else {
        println!("Data was dropped :(");
    }
}
```

Turn Rc<T> into Weak<T>

2. Weak<T>

```rust
fn main() {
    let rc: Rc<i32> = Rc::new(5);
    let wc: Weak<i32> = Rc::downgrade(this: &rc);
    drop(rc);
    if let Some(data: Rc<i32>) = Weak::upgrade(self: &wc) {
        println!("Data has not been dropped: {}", data);
    } else {
        println!("Data was dropped :(");
    }
}
```

Try to upgrade → Fails if underlying data was dropped → Returns Option<Rc<T>>

2. Weak<T>

```rust
fn main() {
    let rc: Rc<i32> = Rc::new(5);
    let wc: Weak<i32> = Rc::downgrade(this: &rc);
    drop(rc);
    if let Some(data: Rc<i32>) = Weak::upgrade(self: &wc) {
        println!("Data has not been dropped: {}", data);
    } else {
        println!("Data was dropped :(");
    }
}
```

strong_count decreased to 0 here → Underlying data dropped

```
    Running `targe
Data was dropped :(
```

2. Weak<T>

```
type Wrap<T> = Weak<RefCell<Data<T>>>;
2 implementations
struct Data<T: Display> {
    value: T,
    next: Option<Wrap<T>>,
}
▶ Run | Debug
fn main() {
    let d1: Data<i32> = Data::new(5);
    let rc: Rc<RefCell<Data<i32>>> = Rc::new(RefCell::new(d1));
    rc.borrow_mut().next = Some(Rc::downgrade(this: &rc));
    println!("count={}", Rc::strong_count(&rc));
}
```

2. Weak<T>

```
type Wrap<T> = Weak<RefCell<Data<T>>>;
2 implementations
struct Data<T: Display> {
    value: T,
    next: Option<Wrap<T>>,
}
▶ Run | Debug
fn main() {
    let d1: Data<i32> = Data::new(5);
    let rc: Rc<RefCell<Data<i32>>> = Rc::new(RefCell::new(d1));
    rc.borrow_mut().next = Some(Rc::downgrade(this: &rc));
    println!("count={}", Rc::strong_count(&rc));
}
```

Small modifications, breaking strong reference cycles

2. Weak<T>

```rust
type Wrap<T> = Weak<RefCell<Data<T>>>;
2 implementations
struct Data<T: Display> {
    value: T,
    next: Option<Wrap<T>>,
}
▶ Run | Debug
fn main() {
    let d1: Data<i32> = Data::new(5);
    let rc: Rc<RefCell<Data<i32>>> = Rc::new(RefCell::new(d1));
    rc.borrow_mut().next = Some(Rc::downgrade(this: &rc));
    println!("count={}", Rc::strong_count(&rc));
}
```

```
    Running
count=1
Dropped 5

C:\Users\pfh
```

2. Weak<T>

```
struct Tree<T: Display> {
    data: T,
    parent: Option<Weak<RefCell<Tree<T>>>>,
    left: Option<Rc<RefCell<Tree<T>>>>,
    right: Option<Rc<RefCell<Tree<T>>>>,
}
```

2. Weak<T>

```
struct Tree<T: Display> {
    data: T,
    parent: Option<Weak<RefCell<Tree<T>>>>,
    left: Option<Rc<RefCell<Tree<T>>>>,
    right: Option<Rc<RefCell<Tree<T>>>>,
}
```

Idea:

→ As long as the node lives, the children (left and right) should also exist

→ For some cases we would also like to access the parent (e.g. for traversal)

→ If managed properly, parent will always point to an undropped node

2. Weak<T>

```rust
fn set_left<T: Display>(parent: Rc<RefCell<Tree<T>>>, left: Rc<RefCell<Tree<T>>>) {
    parent.borrow_mut().left = Some(Rc::clone(self: &left));
    left.borrow_mut().parent = Some(Rc::downgrade(this: &parent));
}
▶ Run | Debug
fn main() {
    let t1: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 5)));
    let t2: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 10)));
    let t3: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 8)));
    set_left(parent: Rc::clone(self: &t1), left: Rc::clone(self: &t2));
    set_left(parent: Rc::clone(self: &t2), left: Rc::clone(self: &t3));
    println!("t1.strong={}", Rc::strong_count(&t1));
    println!("t2.strong={}", Rc::strong_count(&t2));
    println!("t3.strong={}", Rc::strong_count(&t3));
}
```

2. Weak<T>

```rust
fn set_left<T: Display>(parent: Rc<RefCell<Tree<T>>>, left: Rc<RefCell<Tree<T>>>) {
    parent.borrow_mut().left = Some(Rc::clone(self: &left));
    left.borrow_mut().parent = Some(Rc::downgrade(this: &parent));
}
```

Management of references: Child node (left) should also know who the parent is

```rust
▶ Run | Debug
fn main() {
    let t1: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 5)));
    let t2: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 10)));
    let t3: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 8)));
    set_left(parent: Rc::clone(self: &t1), left: Rc::clone(self: &t2));
    set_left(parent: Rc::clone(self: &t2), left: Rc::clone(self: &t3));
    println!("t1.strong={}", Rc::strong_count(&t1));
    println!("t2.strong={}", Rc::strong_count(&t2));
    println!("t3.strong={}", Rc::strong_count(&t3));
}
```

2. Weak<T>

```rust
fn set_left<T: Display>(parent: Rc<RefCell<Tree<T>>>, left: Rc<RefCell<Tree<T>>>) {
    parent.borrow_mut().left = Some(Rc::clone(self: &left));
    left.borrow_mut().parent = Some(Rc::downgrade(this: &parent));
}
▶ Run | Debug
fn main() {
    let t1: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 5)));
    let t2: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 10)));
    let t3: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 8)));
    set_left(parent: Rc::clone(self: &t1), left: Rc::clone(self: &t2));
    set_left(parent: Rc::clone(self: &t2), left: Rc::clone(self: &t3));
    println!("t1.strong={}", Rc::strong_count(&t1));
    println!("t2.strong={}", Rc::strong_count(&t2));
    println!("t3.strong={}", Rc::strong_count(&t3));
}
```

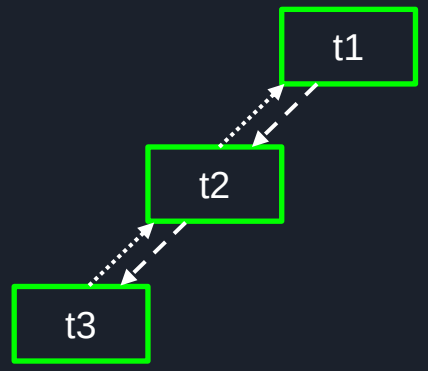
Create some nodes

2. Weak<T>

```rust
fn set_left<T: Display>(parent: Rc<RefCell<Tree<T>>>, left: Rc<RefCell<Tree<T>>>) {
    parent.borrow_mut().left = Some(Rc::clone(self: &left));
    left.borrow_mut().parent = Some(Rc::downgrade(this: &parent));
}
▶ Run | Debug
fn main() {
    let t1: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 5)));
    let t2: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 10)));
    let t3: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 8)));
    set_left(parent: Rc::clone(self: &t1), left: Rc::clone(self: &t2));
    set_left(parent: Rc::clone(self: &t2), left: Rc::clone(self: &t3));
    println!("t1.strong={}", Rc::strong_count(&t1));
    println!("t2.strong={}", Rc::strong_count(&t2));
    println!("t3.strong={}", Rc::strong_count(&t3));
}
```

Set children

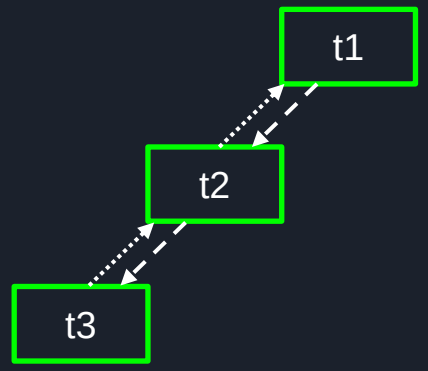
2. Weak<T>



Heap		
t1	data	5
	strong	1
	weak	1
t2	data	10
	strong	2
	weak	1
t3	data	8
	strong	2
	weak	0

```rust
fn set_left<T: Display>(parent: Rc<RefCell<Tree<T>>>, left: Rc<RefCell<Tree<T>>>) {
    parent.borrow_mut().left = Some(Rc::clone(self: &left));
    left.borrow_mut().parent = Some(Rc::downgrade(this: &parent));
}
▶ Run | Debug
fn main() {
    let t1: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 5)));
    let t2: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 10)));
    let t3: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 8)));
    set_left(parent: Rc::clone(self: &t1), left: Rc::clone(self: &t2));
    set_left(parent: Rc::clone(self: &t2), left: Rc::clone(self: &t3));
    println!("t1.strong={}", Rc::strong_count(&t1));
    println!("t2.strong={}", Rc::strong_count(&t2));
    println!("t3.strong={}", Rc::strong_count(&t3));
}
```

2. Weak<T>



Heap		
t1	data	5
	strong	1
	weak	1
t2	data	10
	strong	2
	weak	1
t3	data	8
	strong	2
	weak	0

```
t1.strong=1
t2.strong=2
t3.strong=2
Dropped 5
Dropped 10
Dropped 8
```

```rust
fn set_left<T: Display>(parent: Rc<RefCell<Tree<T>>>, left: Rc<RefCell<Tree<T>>>) {
    parent.borrow_mut().left = Some(Rc::clone(self: &left));
    left.borrow_mut().parent = Some(Rc::downgrade(this: &parent));
}
▶ Run | Debug
fn main() {
    let t1: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 5)));
    let t2: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 10)));
    let t3: Rc<RefCell<Tree<i32>>> = Rc::new(RefCell::new(Tree::new(data: 8)));
    set_left(parent: Rc::clone(self: &t1), left: Rc::clone(self: &t2));
    set_left(parent: Rc::clone(self: &t2), left: Rc::clone(self: &t3));
    println!("t1.strong={}", Rc::strong_count(&t1));
    println!("t2.strong={}", Rc::strong_count(&t2));
    println!("t3.strong={}", Rc::strong_count(&t3));
}
```

3. Declarative Macros

3. Declarative Macros

- Macros are a syntax extension of the language, and serve many purposes

3. Declarative Macros

- Macros are a syntax extension of the language, and serve many purposes
  - Generate boilerplate code (Derive-like macros)

```
#[derive(Debug, PartialEq, Hash,
    Eq, PartialOrd, Ord, Clone,
    Copy)]
8 implementations
struct Data {
    d: u32,
}
```

3. Declarative Macros

- Macros are a syntax extension of the language, and serve many purposes
  - Generate boilerplate code (Derive-like macros)

```
#[derive(Debug, PartialEq, Hash,
    Eq, PartialOrd, Ord, Clone,
    Copy)]
8 implementations
struct Data {
    d: u32,
}
```

Emits
impl <Trait> for Data {
  /* functions */
}
for every of those traits

3. Declarative Macros

- Macros are a syntax extension of the language, and serve many purposes
  - Generate boilerplate code (Derive-like macros)
  - Specify attributes for the compiler, like special warnings and optimizations

3. Declarative Macros

```rust
#![allow(unused)]
#![forbid(redundant_semicolons)]
#[cfg(target_os="windows")]
▶ Run | Debug
fn main() {
    let a: i32 = 1 + 2;;
}
#[cfg(target_os="linux")]
fn main() {
    // No addition
}
```

3. Declarative Macros

```rust
#![allow(unused)]
#![forbid(redundant_semicolons)]
#[cfg(target_os="windows")]
▶ Run | Debug
fn main() {
    let a: i32 = 1 + 2;;
}
#[cfg(target_os="linux")]
fn main() {
    // No addition
}
```

Attributes for the linter:
→ Dont warn that a is unused
→ Throw an error for that semicolon

3. Declarative Macros

```rust
#![allow(unused)]
#![forbid(redundant_semicolons)]
#[cfg(target_os="windows")]
▶ Run | Debug
fn main() {
    let a: i32 = 1 + 2;;
}
#[cfg(target_os="linux")]
fn main() {
    // No addition
}
```

Attributes for the linter:
→ Dont warn that a is unused
→ Throw an error for that semicolon

Conditionally compile code
→ Here we define different entrypoints based on if we‘re on Windows or Linux

3. Declarative Macros

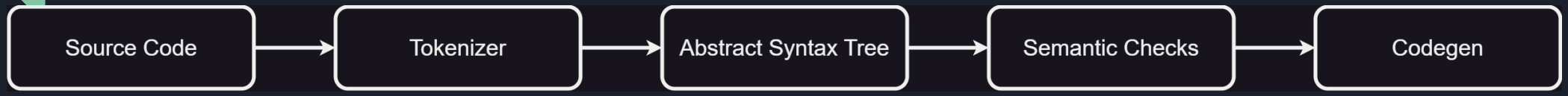
- Macros are a syntax extension of the language, and serve many purposes
  - Generate boilerplate code (Derive-like macros)
  - Specify attributes for the compiler, like special warnings and optimizations
  - Defining functions that take a variadic amount of arguments

```
fn print<T>(arg: T) {}
fn print<T>(arg1: T, arg2: T) {}
```

3. Declarative Macros

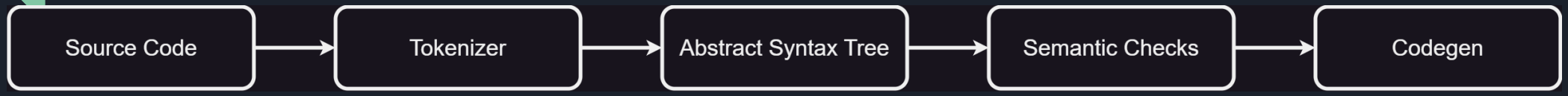
- Macros are a syntax extension of the language, and serve many purposes
  - Generate boilerplate code (Derive-like macros)
  - Specify attributes for the compiler, like special warnings and optimizations
  - Defining functions that take a variadic amount of arguments
- To understand how macros work, we first need to take a small crash course of compilers

3. Declarative Macros




```
2  fn main() {
3      let a: i32 = 1 + 2;
4  }
```

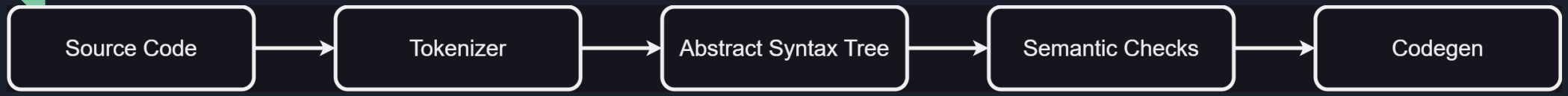
3. Declarative Macros




```
2  fn main() {
3      let a: i32 = 1 + 2;
4  }
```

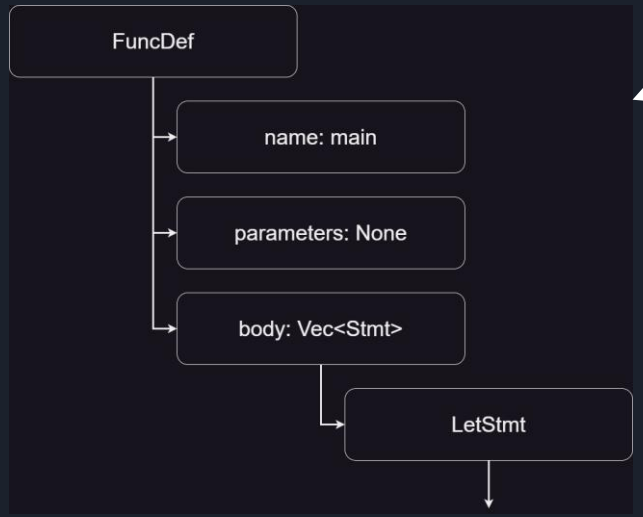
```
FnKeyword [loc: src/main.rs:2:1]
Ident [name: "main", loc: src/main.rs:2:4]
OpenParen [loc: src/main.rs:2:8]
...
```

3. Declarative Macros

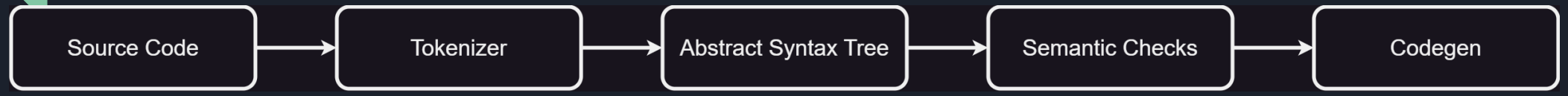



```
2  fn main() {
3      let a: i32 = 1 + 2;
4  }
```

```
FnKeyword [loc: src/main.rs:2:1]
Ident [name: "main", loc: src/main.rs:2:4]
OpenParen [loc: src/main.rs:2:8]
...
```

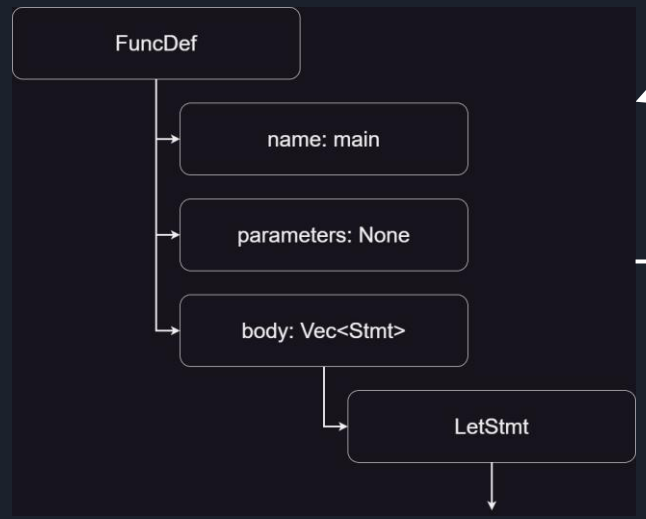


3. Declarative Macros



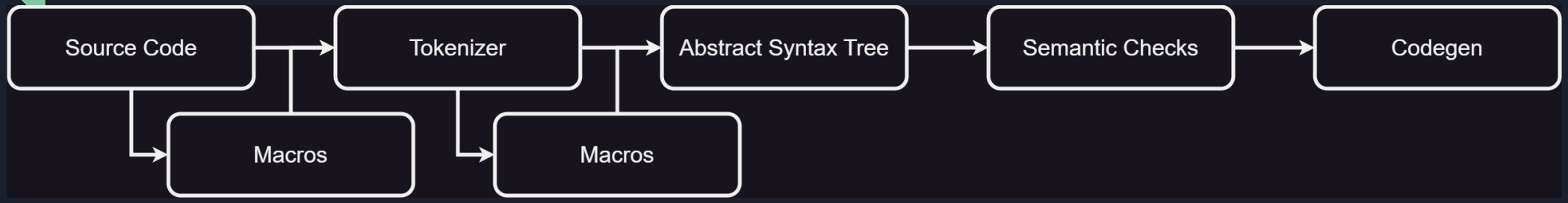

```
2  fn main() {
3      let a: i32 = 1 + 2;
4  }
```

```
FnKeyword [loc: src/main.rs:2:1]
Ident [name: "main", loc: src/main.rs:2:4]
OpenParen [loc: src/main.rs:2:8]
...
```



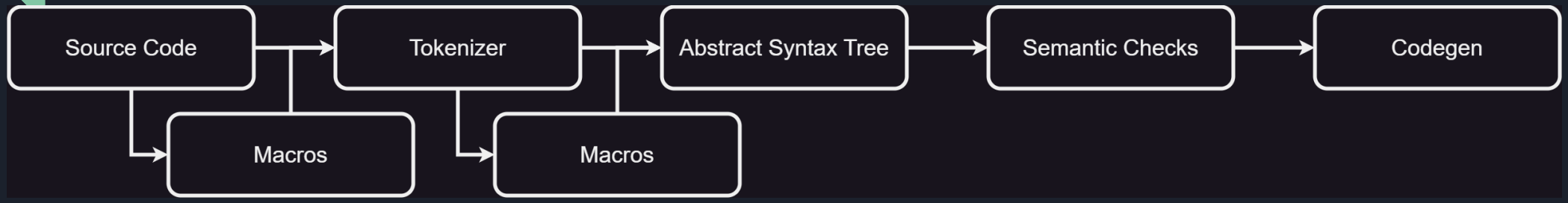

```
_ZN4main4main17h10044c8640edeaecE:
.seh_proc _ZN4main4main17h10044c8640edeaecE
    subq    $40, %rsp
    .seh_stackalloc 40
    .seh_endprologue
    movl    $1, %eax
    addl    $2, %eax
    seto    %al
    testb   $1, %al
    jne .LBB8_2
    addq    $40, %rsp
    retq
```

3. Declarative Macros




```rust
fn main() {
    println!("Hi, {}", "how are you?");
}
```

3. Declarative Macros




```
fn main() {
    println!("Hi, {}", "how are you?");
}
```

println! gets expanded into a sequence of tokens and ultimately an AST node

3. Declarative Macros

- Declarative macros are their own sub-language, with their own rules and syntax
  - Procedural macros are plain Rust, but they'll be covered later

3. Declarative Macros

- Declarative macros are their own sub-language, with their own rules and syntax
- Declarative macros are declared using the keyword macro_rules

3. Declarative Macros

```
macro_rules! showcase {
    () => {
        println!("Passed nothing!");
    };
    ($e:expr) => {
        println!("Passed expr `{}`", stringify!($e));
    };
}

▶ Run | Debug
fn main() {
    showcase!(5 + 10);
    showcase!();
}
```

3. Declarative Macros

```
macro_rules! showcase {
    () => {
        println!("Passed nothing!");
    };
    ($e:expr) => {
        println!("Passed expr `{}`", stringify!($e));
    };
}

▶ Run | Debug
fn main() {
    showcase!(5 + 10);
    showcase!();
}
```

Name of the macro

3. Declarative Macros

```
macro_rules! showcase {
    () => {
        println!("Passed nothing!");
    };
    ($e:expr) => {
        println!("Passed expr `{}`", stringify!($e));
    };
}

▶ Run | Debug
fn main() {
    showcase!(5 + 10);
    showcase!();
}
```

A sequence of cases to match against

→ When you invoke the macro, cases are checked from top to bottom and the first match is chosen

3. Declarative Macros

```rust
macro_rules! showcase {
    () => {
        println!("Passed nothing!");
    };
    ($e:expr) => {
        println!("Passed expr `{}`", stringify!($e));
    };
}

▶ Run | Debug
fn main() {
    showcase!(5 + 10);
    showcase!();
}
```

3. Declarative Macros

```
macro_rules! showcase {
    () => {
        println!("Passed nothing!");
    };
    ($e:expr) => {
        println!("Passed expr `{}`", stringify!($e));
    };
}
```

If the case matches, we insert the code in the block after

```
▶ Run | Debug
fn main() {
    showcase!(5 + 10);
    showcase!();
}
```

3. Declarative Macros

```
macro_rules! showcase {
    () => {
        println!("Passed nothing!");
    };
    ($e:expr) => {
        println!("Passed expr `{}`", stringify!($e));
    };
}

▶ Run | Debug
fn main() {
    showcase!(5 + 10);
    showcase!();
}
```

Matched nodes are assigned to special identifiers (marked with $)
so we can use them in the macro
→ Example expands to stringify!(5 + 10)

3. Declarative Macros

```
macro_rules! showcase {
    () => {
        println!("Passed nothing!");
    };
    ($e:expr) => {
        println!("Passed expr `{}`", stringify!($e));
    };
}

▶ Run | Debug
fn main() {
    showcase!(5 + 10);
    showcase!();
}
```

Macros allow recursion, those are also expanded

3. Declarative Macros

```
macro_rules! showcase {
    () => {
        println!("Passed nothing!");
    };
    ($e:expr) => {
        println!("Passed expr `{}`", stringify!($e));
    };
}

▶ Run | Debug
fn main() {
    println!("Passed expr `{}`", "5 + 10");
    println!("Passed nothing!");
}
```

Our macro would expand to this (ignoring println!)

→ cargo install cargo-expand to see what the raw code looks like

→ Adds cargo expand to the list of possible subcommands

```
macro_rules! rules {
    ($e:expr) => { "Expression."; };
    ($i:ident) => { "Identifier."; };
    ($t:ty) => { "Type."; };
    ($($val:expr),*) => {
        "Comma-separated sequence of expressions.";
        "The sequence can be empty.";
    };
    ($($id:ident)+) => {
        "Sequence of identifiers, with no separators.";
        "There must be at least one identifier. (+)";
    };
    ($($($t:ty),*);+) => {
        "Semicolon-separated sequence of comma-separated sequence of types.";
        "Sequence of types can be empty. (*)";
        "Sequence of sequences must have at least one element. (+)";
    };
    () => {}; // None
}
```

You can match against many different cases in different complexities

```
macro_rules! rules {
    ($e:expr) => { "Expression."; };
    ($i:ident) => { "Identifier."; };
    ($t:ty) => { "Type."; };
    ($($val:expr),*) => {
        "Comma-separated sequence of expressions.";
        "The sequence can be empty.";
    };
    ($($id:ident)+) => {
        "Sequence of identifiers, with no separators.";
        "There must be at least one identifier. (+)";
    };
    ($($($t:ty),*);+) => {
        "Semicolon-separated sequence of comma-separated sequence of types.";
        "Sequence of types can be empty. (*)";
        "Sequence of sequences must have at least one element. (+)";
    };
    () => {}; // None
}
```

You can match against variadic amounts of arguments

```
macro_rules! rules {
    ($e:expr) => { "Expression."; };
    ($i:ident) => { "Identifier."; };
    ($t:ty) => { "Type."; };
    ($($val:expr),*) => {
        "Comma-separated sequence of expressions.";
        "The sequence can be empty.";
    };
    ($($id:ident)+) => {
        "Sequence of identifiers, with no separators.";
        "There must be at least one identifier. (+)";
    };
    ($($($t:ty),*);+) => {
        "Semicolon-separated sequence of comma-separated sequence of types.";
        "Sequence of types can be empty. (*)";
        "Sequence of sequences must have at least one element. (+)";
    };
    () => {}; // None
}
```

rules!(5 + 10, „Hello“) would match this case and bind both 5 + 10 and „Hello“ to $val

```
macro_rules! rules {
    ($e:expr) => { "Expression."; };
    ($i:ident) => { "Identifier."; };
    ($t:ty) => { "Type."; };
    ($($val:expr),*) => {
        "Comma-separated sequence of expressions.";
        "The sequence can be empty.";
    };
    ($($id:ident)+) => {
        "Sequence of identifiers, with no separators.";
        "There must be at least one identifier. (+)";
    };
    ($($($t:ty),*);+) => {
        "Semicolon-separated sequence of comma-separated sequence of types.";
        "Sequence of types can be empty. (*)";
        "Sequence of sequences must have at least one element. (+)";
    };
    () => {}; // None
}
```

rules!(i32, u8; &str, String) would match this case

3. Declarative Macros

```rust
macro_rules! rules {
    ($e:expr) => { "Expression."; };
    ($i:ident) => { "Identifier."; };
    ($t:ty) => { "Type."; };
    ($($val:expr),*) => {
        "Comma-separated sequence of expressions.";
        "The sequence can be empty.";
    };
    ($($id:ident)+) => {
        "Sequence of identifiers, with no separators.";
        "There must be at least one identifier. (+)";
    };
    ($($($t:ty),*);+) => {
        "Semicolon-separated sequence of comma-separated sequence of types.";
        "Sequence of types can be empty. (*)";
        "Sequence of sequences must have at least one element. (+)";
    };
    () => {}; // None
}
```

```rust
fn main() {
    rules!(5 + 10);
    rules!(a);
    rules!(5, 4, 3);
    rules!(a b);
    rules!(i32, u8, &str);
    rules!(i32, u8, &str; String, u8);
}
```

3. Declarative Macros

3/3 Which cases match?

```rust
fn main() {
    rules!(5 + 10);
    rules!(a);
    rules!(5, 4, 3);
    rules!(a b);
    rules!(i32, u8, &str);
    rules!(i32, u8, &str; String, u8);
}
```

```rust
macro_rules! rules {
    ($e:expr) => { "Expression."; };
    ($i:ident) => { "Identifier."; };
    ($t:ty) => { "Type."; };
    ($($val:expr),*) => {
        "Comma-separated sequence of expressions.";
        "The sequence can be empty.";
    };
    ($($id:ident)+) => {
        "Sequence of identifiers, with no separators.";
        "There must be at least one identifier. (+)";
    };
    ($($($t:ty),*);+) => {
        "Semicolon-separated sequence of comma-separated sequence of types.";
        "Sequence of types can be empty. (*)";
        "Sequence of sequences must have at least one element. (+)";
    };
    () => {}; // None
}
```

3. Declarative Macros

3/3 Which cases match?

```rust
fn main() {
    rules!(5 + 10);
    rules!(a);
    rules!(5, 4, 3);
    rules!(a b);
    rules!(i32, u8, &str);
    rules!(i32, u8, &str; String, u8);
}
```

```rust
macro_rules! rules {
    ($e:expr) => { "Expression."; };
    ($i:ident) => { "Identifier."; };
    ($t:ty) => { "Type."; };
    ($($val:expr),*) => {
        "Comma-separated sequence of expressions.";
        "The sequence can be empty.";
    };
    ($($id:ident)+) => {
        "Sequence of identifiers, with no separators.";
        "There must be at least one identifier. (+)";
    };
    ($($($t:ty),*);+) => {
        "Semicolon-separated sequence of comma-separated sequence of types.";
        "Sequence of types can be empty. (*)";
        "Sequence of sequences must have at least one element. (+)";
    };
    () => {}; // None
}
```

3. Declarative Macros

a is also a valid expression
→ Cases are evaluated top to bottom

3/3 Which cases match?

```rust
fn main() {
    rules!(5 + 10);
    rules!(a);
    rules!(5, 4, 3);
    rules!(a b);
    rules!(i32, u8, &str);
    rules!(i32, u8, &str; String, u8);
}
```

```rust
macro_rules! rules {
    ($e:expr) => { "Expression."; };
    ($i:ident) => { "Identifier."; };
    ($t:ty) => { "Type."; };
    ($($val:expr),*) => {
        "Comma-separated sequence of expressions.";
        "The sequence can be empty.";
    };
    ($($id:ident)+) => {
        "Sequence of identifiers, with no separators.";
        "There must be at least one identifier. (+)";
    };
    ($($($t:ty),*);+) => {
        "Semicolon-separated sequence of comma-separated sequence of types.";
        "Sequence of types can be empty. (*)";
        "Sequence of sequences must have at least one element. (+)";
    };
    () => {}; // None
}
```

3. Declarative Macros

```
macro_rules! rules {
    ($e:expr) => { "Expression."; };
    ($i:ident) => { "Identifier."; };
    ($t:ty) => { "Type."; };
    ($($val:expr),*) => {
        "Comma-separated sequence of expressions.";
        "The sequence can be empty.";
    };
    ($($id:ident)+) => {
        "Sequence of identifiers, with no separators.";
        "There must be at least one identifier. (+)";
    };
    ($($($t:ty),*);+) => {
        "Semicolon-separated sequence of comma-separated sequence of types.";
        "Sequence of types can be empty. (*)";
        "Sequence of sequences must have at least one element. (+)";
    };
    () => {}; // None
}
```

3/3 Which cases match?

```
fn main() {
    rules!(5 + 10);
    rules!(a);
    rules!(5, 4, 3);
    rules!(a b);
    rules!(i32, u8, &str);
    rules!(i32, u8, &str; String, u8);
}
```

3. Declarative Macros

3/3 Which cases match?

```
fn main() {
    rules!(5 + 10);
    rules!(a);
    rules!(5, 4, 3);
    rules!(a b);
    rules!(i32, u8, &str);
    rules!(i32, u8, &str; String, u8);
}
```

```
macro_rules! rules {
    ($e:expr) => { "Expression."; };
    ($i:ident) => { "Identifier."; };
    ($t:ty) => { "Type."; };
    ($($val:expr),*) => {
        "Comma-separated sequence of expressions.";
        "The sequence can be empty.";
    };
    ($($id:ident)+) => {
        "Sequence of identifiers, with no separators.";
        "There must be at least one identifier. (+)";
    };
    ($($($t:ty),*);+) => {
        "Semicolon-separated sequence of comma-separated sequence of types.";
        "Sequence of types can be empty. (*)";
        "Sequence of sequences must have at least one element. (+)";
    };
    () => {}; // None
}
```

3. Declarative Macros

→ Missing ; for the last rule
→ Technically valid identifiers
→ valid expressions

```
macro_rules! rules {
    ($e:expr) => { "Expression."; };
    ($i:ident) => { "Identifier."; };
    ($t:ty) => { "Type."; };
    ($($val:expr),*) => {
        "Comma-separated sequence of expressions.";
        "The sequence can be empty.";
    };
    ($($id:ident)+) => {
        "Sequence of identifiers, with no separators.";
        "There must be at least one identifier. (+)";
    };
    ($($($t:ty),*);+) => {
        "Semicolon-separated sequence of comma-separated sequence of types.";
        "Sequence of types can be empty. (*)";
        "Sequence of sequences must have at least one element. (+)";
    };
    () => {}; // None
}
```

3/3 Which cases match?

```
fn main() {
    rules!(5 + 10);
    rules!(a);
    rules!(5, 4, 3);
    rules!(a b);
    rules!(i32, u8, &str);
    rules!(i32, u8, &str; String, u8);
}
```

3. Declarative Macros

→ Missing ; for the last rule
→ Technically valid identifiers
→ valid expressions

```
macro_rules! rules {
    ($e:expr) => { "Expression."; };
    ($i:ident) => { "Identifier."; };
    ($t:ty) => { "Type."; };
    ($($val:expr),*) => {
        "Comma-separated sequence of expressions.";
    }
}

let i32: u8 = 5;
let u8: i16 = 12;
let str: u32 = 1;
println!("{} {} {}", i32, u8, &str);

fn main() {
    rules!(a b);
    rules!(i32, u8, &str);
    rules!(i32, u8, &str; String, u8);
}
```

3. Declarative Macros

```
macro_rules! rules {
    ($e:expr) => { "Expression."; };
    ($i:ident) => { "Identifier."; };
    ($t:ty) => { "Type."; };
    ($($val:expr),*) => {
        "Comma-separated sequence of expressions.";
        "The sequence can be empty.";
    };
    ($($id:ident)+) => {
        "Sequence of identifiers, with no separators.";
        "There must be at least one identifier. (+)";
    };
    ($($($t:ty),*);+) => {
        "Semicolon-separated sequence of comma-separated sequence of types.";
        "Sequence of types can be empty. (*)";
        "Sequence of sequences must have at least one element. (+)";
    };
    () => {}; // None
}
```

3/3 Which cases match?

```
fn main() {
    rules!(5 + 10);
    rules!(a);
    rules!(5, 4, 3);
    rules!(a b);
    rules!(i32, u8, &str);
    rules!(i32, u8, &str; String, u8);
}
```

3. Declarative Macros

```
macro_rules! decl {
    ($i:ident $e:expr) => {
        let $i = $e;
    };
}
▶ Run | Debug
fn main() {
    let b: i32 = 5;
    decl!(a b*b);
    println!("{a}");
}
```

Using macros we can now write simple code generators

3. Declarative Macros

```
macro_rules! decl {
    ($i:ident $e:expr) => {
        let $i = $e;
    };
}
▶ Run | Debug
fn main() {
    let b: i32 = 5;
    let a = b*b;
    println!("{a}");
}
```

Macro call expands to this code

We can now use a here

3. Declarative Macros

```
macro_rules! decl {
    ($i:ident $e:expr) => {
        let $i = b;
    };
}

▶ Run | Debug
fn main() {
    let b: i32 = 5;
    decl!(a b*b);
    println!("{a}");
}
```

However, macros are *partially hygienic*:
→ They are doing more than just inserting code
→ We can not use b because the scopes differ

3. Declarative Macros

```
macro_rules! decl {
    ($i:ident $e:expr) => {
        let $i = b;
    };
}
▶ Run | Debug
fn main() {
    let b: i32 = 5;
    decl!(a b*b);
    println!("{a}");
}
```

However, macros are *partially hygienic*:
→ They are doing more than just inserting code
→ We can not use b because the scopes differ

Even though the generated code is perfectly fine!

```
fn main() {
    let b = 5;
    let a = b;
    {
        ::std:
    };
}
```

3. Declarative Macros

```
macro_rules! decl {
    ($i:ident $e:expr) => {
        let $i = b;
    };
}

▶ Run | Debug
fn main() {
    let b: i32 = 5;
    decl!(a b*b);
    println!("{a}");
}
```

However, macros are *partially hygienic*:
→ They are doing more than just inserting code
→ We can not use b because the scopes differ

Even though the generated code is perfectly fine!
→ Imagine using decl!() where there is no b

```
fn main() {
    let b = 5;
    let a = b;
    {
        ::std:
    };
}
```

```
fn other() {
    decl!(a 0);
}
```

3. Declarative Macros

```
fn main() {
    calc!(5 + 4);
}
macro_rules! calc {
    ($e:expr) => {
        println!("{} = {}", stringify!($e), $e);
    };
}
```

Macros must be defined before they are used

```rust
struct Function {
    params: Vec<Variable>,
    body: Block,
}
```

0 implementations

```rust
struct Method {
    params: Vec<Variable>,
    body: Block,
}
fn method_type_check(method: &Method) -> TCResult {
    for p: &Variable in &method.params { param_type_check(param: &p)?; }
    block_type_check(block: &method.body)?;
    // Other method related checks
    Ok(())
}
fn func_type_check(func: &Function) -> TCResult {
    for p: &Variable in &func.params { param_type_check(param: &p)?; }
    block_type_check(block: &func.body)?;
    // Other function related checks
    Ok(())
}
```

```rust
struct Function {
    params: Vec<Variable>,
    body: Block,
}
```

0 implementations

```rust
struct Method {
    params: Vec<Variable>,
    body: Block,
}
fn method_type_check(method: &Method) -> TCResult {
    for p: &Variable in &method.params { param_type_check(param: &p)?; }
    block_type_check(block: &method.body)?;
    // Other method related checks
    Ok(())
}
fn func_type_check(func: &Function) -> TCResult {
    for p: &Variable in &func.params { param_type_check(param: &p)?; }
    block_type_check(block: &func.body)?;
    // Other function related checks
    Ok(())
}
```

Duplicated code that can not be factored out into a single function
→ How can a function take both a Method and a Function?
→ No function overload in Rust

```
macro_rules! shared_func {
    ($f:ident) => {
        for p in &$f.params { param_type_check(&p)?; }
        block_type_check(&$f.body)?;
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method);
    // Other method related checks
    Ok(())
}
fn func_type_check(func: &Function) -> TCResult {
    shared_func!(func);
    // Other function related checks
    Ok(())
}
```

```rust
macro_rules! shared_func {
    ($f:ident) => {
        for p in &$f.params { param_type_check(&p)?; }
        block_type_check(&$f.body)?;
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method);
    // Other method related checks
    Ok(())
}
fn func_type_check(func: &Function) -> TCResult {
    shared_func!(func);
    // Other function related checks
    Ok(())
}
```

Shared behavior can be factored into a macro, which does not care about any types
→ Further checks will then decide if method.params and func.params is a valid thing

```rust
macro_rules! shared_func {
    ($f:ident) => {
        for p in &$f.params { param_type_check(&p)?; }
        block_type_check(&$f.body)?;
    };
}
```

However, this macro is not hygienic either, and needs refinement (even if it works just fine)

3/3

What could cause problems?

```rust
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method);
    // Other method related checks
    Ok(())
}
fn func_type_check(func: &Function) -> TCResult {
    shared_func!(func);
    // Other function related checks
    Ok(())
}
```

```
macro_rules! shared_func {
    ($f:ident) => {
        for p in &$f.params { param_type_check(&p)?; }
        block_type_check(&$f.body)?;
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method);
    // Other method related checks
    Ok(())
}
fn func_type_check(func: &Function) -> TCResult {
    shared_func!(func);
    // Other function related checks
    Ok(())
}
```

Early exit

Early exit

3/3

What could cause problems?

It‘s not clear to the user that this macro could exit the function
→ It modifies control flow without being visible

```
macro_rules! shared_func {
    ($f:ident) => {
        for p in &$f.params { param_type_check(&p)?; }
        block_type_check(&$f.body)?;
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method)?;
    // Other method related checks
    Ok(())
}
fn func_type_check(func: &Function) -> TCResult {
    shared_func!(func)?;
    // Other function related checks
    Ok(())
}
```

It would be nicer if we could denote this behavior here

```
macro_rules! shared_func {
    ($f:ident) => {
        let mut res = Ok(());
        for p in &$f.params {
            res = param_type_check(&p);
            if res.is_err() { break; }
        }
        if res.is_err() { res }
        else { block_type_check(&$f.body) }
    };
}
```

Idea: Instead of returning early, we just skip everything once we fail

```
macro_rules! shared_func {
    ($f:ident) => {
        let mut res = Ok(());
        for p in &$f.params {
            res = param_type_check(&p);
            if res.is_err() { break; }
        }
        if res.is_err() { res }
        else { block_type_check(&$f.body) }
    };
}
```

Idea: Instead of returning early, we just skip everything once we fail

Last statement is an expression, so it should return like a normal function?

```rust
macro_rules! shared_func {
    ($f:ident) => {
        let mut res = Ok(());
        for p in &$f.params {
            res = param_type_check(&p);
            if res.is_err() { break; }
        }
        if res.is_err() { res }
        else { block_type_check(&$f.body) }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method);
    // Other method related checks
    Ok(())
}
```

```rust
macro_rules! shared_func {
    ($f:ident) => {
        let mut res = Ok(());
        for p in &$f.params {
            res = param_type_check(&p);
            if res.is_err() { break; }
        }
        if res.is_err() { res }
        else { block_type_check(&$f.body) }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method);
    // Other method related checks
    Ok(())
}
```

```rust
fn method_type_check(method: &Method) -> TCResult {
    let mut res = Ok(());
    for p in &method.params {
        res = param_type_check(&p);
        if res.is_err() {
            break;
        }
    }
    if res.is_err() { res } else { block_type_check(&method.body) };
    Ok(())
}
```

```
macro_rules! shared_func {
    ($f:ident) => {
        let mut res = Ok(());
        for p in &$f.params {
            res = param_type_check(&p);
            if res.is_err() { break; }
        }
        if res.is_err() { res }
        else { block_type_check(&$f.body) }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method);
    // Other method related checks
    Ok(())
}
```

Missing the ?

```
fn method_type_check(method: &Method) -> TCResult {
    let mut res = Ok(());
    for p in &method.params {
        res = param_type_check(&p);
        if res.is_err() {
            break;
        }
    }
    if res.is_err() { res } else { block_type_check(&method.body) };
    Ok(())
}
```

```rust
macro_rules! shared_func {
    ($f:ident) => {
        let mut res = Ok(());
        for p in &$f.params {
            res = param_type_check(&p);
            if res.is_err() { break; }
        }
        if res.is_err() { res }
        else { block_type_check(&$f.body) }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method)?;
    // Other method related checks
    Ok(())
}
```

```rust
macro_rules! shared_func {
    ($f:ident) => {
        let mut res = Ok(());
        for p in &$f.params {
            res = param_type_check(&p);
            if res.is_err() { break; }
        }
        if res.is_err() { res }
        else { block_type_check(&$f.body) }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method)?;
    // Other method related checks
    Ok(())
}
```

```rust
fn method_type_check(method: &Method) -> TCResult {
    (let mut res = Ok(()))?;
    Ok(())
}
```

```rust
macro_rules! shared_func {
    ($f:ident) => {
        let mut res = Ok(());
        for p in &$f.params {
            res = param_type_check(&p);
            if res.is_err() { break; }
        }
        if res.is_err() { res }
        else { block_type_check(&$f.body) }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method)?;
    // Other method related checks
    Ok(())
}
```

```rust
fn method_type_check(method: &Method) -> TCResult {
    (let mut res = Ok(()))?;
    Ok(())
}
```

→ Macro parsing and expansion is built into the parser at specific locations (part of the grammar)

```rust
macro_rules! shared_func {
    ($f:ident) => {
        let mut res = Ok(());
        for p in &$f.params {
            res = param_type_check(&p);
            if res.is_err() { break; }
        }
        if res.is_err() { res }
        else { block_type_check(&$f.body) }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method)?;
    // Other method related checks
    Ok(())
}
```

```rust
fn method_type_check(method: &Method) -> TCResult {
    (let mut res = Ok(()))?;
    Ok(())
}
```

→ Macro parsing and expansion is built into the parser at specific locations (part of the grammar)

→ Depending on context macros are evaluated differently

```rust
macro_rules! shared_func {
    ($f:ident) => {
        let mut res = Ok(());
        for p in &$f.params {
            res = param_type_check(&p);
            if res.is_err() { break; }
        }
        if res.is_err() { res }
        else { block_type_check(&$f.body) }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method)?;
    // Other method related checks
    Ok(())
}
```

```rust
fn method_type_check(method: &Method) -> TCResult {
    (let mut res = Ok(()))?;
    Ok(())
}
```

→ Macro parsing and expansion is built into the parser at specific locations (part of the grammar)

→ Depending on context macros are evaluated differently

→ The ? turns the macro call from a statement into an expression

→ Macros don‘t insert code, they expand to AST nodes

```
macro_rules! shared_func {
    ($f:ident) => {
        let mut res = Ok(());
        for p in &$f.params {
            res = param_type_check(&p);
            if res.is_err() { break; }
        }
        if res.is_err() { res }
        else { block_type_check(&$f.body) }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method)?;
    // Other method related checks
    Ok(())
}
```

```
fn method_type_check(method: &Method) -> TCResult {
    (let mut res = Ok(()))?;
    Ok(())
}
```

→ The ? turns the macro call from a statement into an expression

Which is actually what we want, except that we want the *whole* macro to be the expression

```rust
macro_rules! shared_func {
    ($f:ident) => {
        let mut res = Ok(());
        for p in &$f.params {
            res = param_type_check(&p);
            if res.is_err() { break; }
        }
        if res.is_err() { res }
        else { block_type_check(&$f.body) }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method)?;
    // Other method related checks
    Ok(())
}
```

```rust
fn method_type_check(method: &Method) -> TCResult {
    (let mut res = Ok(()))?;
    Ok(())
}
```

→ The ? turns the macro call from a statement into an expression

Which is actually what we want, except that we want the *whole* macro to be the expression
→ We need to tell Rust that those statements form a single expression

```rust
macro_rules! shared_func {
    ($f:ident) => {
        let mut res = Ok(());
        for p in &$f.params {
            res = param_type_check(&p);
            if res.is_err() { break; }
        }
        if res.is_err() { res }
        else { block_type_check(&$f.body) }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method)?;
    // Other method related checks
    Ok(())
}
```

```rust
fn method_type_check(method: &Method) -> TCResult {
    (let mut res = Ok(()))?;
    Ok(())
}
```

→ The ? turns the macro call from a statement into an expression

Which is actually what we want, except that we want the *whole* macro to be the expression
→ We need to tell Rust that those statements form a single expression
→ Single expressions of that form are blocks → We need to add curly brackets

```rust
macro_rules! shared_func {
    ($f:ident) => {
        {
            let mut res = Ok(());
            for p in &$f.params {
                res = param_type_check(&p);
                if res.is_err() { break; }
            }
            if res.is_err() { res }
            else { block_type_check(&$f.body) }
        }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method)?;
    // Other method related checks
    Ok(())
}
```

```rust
macro_rules! shared_func {
    ($f:ident) => {
        {
            let mut res = Ok(());
            for p in &$f.params {
                res = param_type_check(&p);
                if res.is_err() { break; }
            }
            if res.is_err() { res }
            else { block_type_check(&$f.body) }
        }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method)?;
    // Other method related checks
    Ok(())
}
```

Macro expands into a single block

```rust
macro_rules! shared_func {
    ($f:ident) => {
        {
            let mut res = Ok(());
            for p in &$f.params {
                res = param_type_check(&p);
                if res.is_err() { break; }
            }
            if res.is_err() { res }
            else { block_type_check(&$f.body) }
        }
    };
}
fn method_type_check(method: &Method) -> TCResult {
    shared_func!(method)?;
    // Other method related checks
    Ok(())
}
```

Macro expands into a single block

```rust
fn method_type_check(method: &Method) -> TCResult {
    {
        let mut res = Ok(());
        for p in &method.params {
            res = param_type_check(&p);
            if res.is_err() {
                break;
            }
        }
        if res.is_err() { res } else { block_type_check(&method.body) }
    }?;
    Ok(())
}
```

4. Next time

- tt-munchers
- Procedural Macros