

13.

Makros

Warum Makros?

- Programmieren → Abstrahieren
 - Doppelten Code vermeiden
 - Alles Wichtige zu einem Anliegen an einem Ort
 - *Black Box* erleichtert abstraktes Denken
- Bisher: Semantische Abstraktionen
- Makros: Syntaktische Abstraktionen
 - Vorteil: Mehr Möglichkeiten
 - Nachteile:
 - Schwieriger zu verstehen (mehr Unbekanntes)
 - Schlechtere Compilerfehler (Compiler hat weniger semantische Informationen)

Beispiel: `try!()`

- Problem: Doppelter Code für **Err**-Early-Return
- Abstraktion nicht mit Funktionen möglich
- Lösung: Makro `try!()`



Makros sind letzter Ausweg!

Bekannte Möglichkeiten der Abstraktion bevorzugen!

Übersicht

Compiler Plugins

- *Unstable*

Lint/Compiler-Pass

Syntax Extension

- Sehr mächtig, aber kompliziert

„attribute-like“

- Nutzung: `#[foo]`

„function-like“

- Nutzung: `foo!()`

Macro by Example

- Mit `macro_rules!` definiert
- Recht eingeschränkt
- *Stable*



Procedural Macro

„custom derive“

- Nutzung: `#[derive(Foo)]`

Macro by Example

- Bildet *Syntax-Tree* auf *Syntax-Tree* ab
- *Keine* reine Textersetzung (wie in C)
 - Vermeidet typische Probleme von C-Makros
- Definition mit „**macro_rules!**“
- Besteht aus Reihe von Regeln
 - Ähnlich wie **match**-Arme
 - Matching auf *Syntax-Trees*
- „Aufruf“ heißt „Expansion“
 - Syntax-Tree wird umgewandelt

```
macro_rules! do_nothing {  
    () => {}  
}  
  
// use it the macro:  
do_nothing!();  
// yeah, nothing happened! \o/
```

Syntax (vereinfacht)

```
macro_rules! <name> {  
    ( <matcher_a> ) => { <body_a> };  
    ( <matcher_b> ) => { <body_b> };  
}
```

Erstes Beispiel

```
macro_rules! greet {
  ("Peter") => {
    println!("wazup!");
  };
  ("Sabine") => {
    println!("My pleasure, Miss");
  };
}

greet!("Peter"); // prints: wazup!
```

- Compiler sucht passende Regel zur Compilierzeit
- *Expansion*: Ersetzt **greet!(...)** durch **println!(...)**
- Compiler vergleicht nur Token!
 - „String Literal“, „Identifizier“, ...

```
let name = "Peter";
greet!(name);
```




```
error: no rules expected the token `name`
  --> <anon>:15:12
     |
15  |     greet!(name);
     |             ^^^^
```

Fast alle Token sind erlaubt

```
macro_rules! foo {  
    (Peter isst 3% Kartoffeln) => { 1 };  
    (Anna mag Peter) => { 2 };  
    (fn <{}: && <) => { 3 };  
}  
  
foo!(Peter isst 3% Kartoffeln); // works: 1  
foo!(fn <{}: && <); // works: 2  
foo!(Anna ist faul); // error
```

- Matcher kann beliebige Token enthalten
- Klammern müssen ausbalanciert sein
 - Auch beim „Aufruf“



```
error: no rules expected the token `ist`  
--> <anon>:17:15  
17 |         foo!(Anna ist faul);  
    |                   ^^^
```

Metavariablen

```
macro_rules! greet {
  (formal: $name:expr) => {
    println!("Hello, {}", $name);
  };
  (informal: $name:expr) => {
    println!("Sup, {}", $name);
  };
}

greet!(formal: "Claudia");
greet!(informal: 27 * (21 << 1));
```

- Binden Syntaxbaum einer bestimmten *syntaktischen Form* an sich
- *Fragment Specifier* nach „:“ geben syntaktische Form an
 - **expr** erlaubt jede mögliche *Expression*
- Haben nichts mit Typsystem zu tun!

```
// error: (): Display not satisfied
greet!(formal: ());
```

Fehlermeldungen für Macro-Fehler teilweise schwierig zu verstehen!

Beispiel

```
macro_rules! impl_default {
  ($type_name:ident, $ctor:ident) => {
    impl Default for $type_name {
      fn default() -> Self {
        Self::$ctor()
      }
    }
  }
}
```

Makro nicht perfekt!
(Point<T>?)

```
impl_default!(Point, origin);
impl_default!(Fibonacci, new);
```

Macros wie dieses zur
Trait-impl üblich!

Situation:

- **Default Trait:** Sinnvolle Standardinstanz
- Typen mit Konstrukturfunktionen, einige davon ohne Argumente
 - **Point** mit **origin()**
 - **Fibonacci** mit **new()**

Aufgabe/Idee:

- Makro zur Abstraktion
 - **default()** mit schon existierender Konstrukturfunktion implementieren

Normale Implementation

```
impl Default for Point {
  fn default() -> Self {
    Self::origin()
  }
}
```

[Komplettes Beispiel \(Link\)](#)

Textersetzung?

```
macro_rules! twice {  
    ($x:expr) => (2 * $x);  
}
```

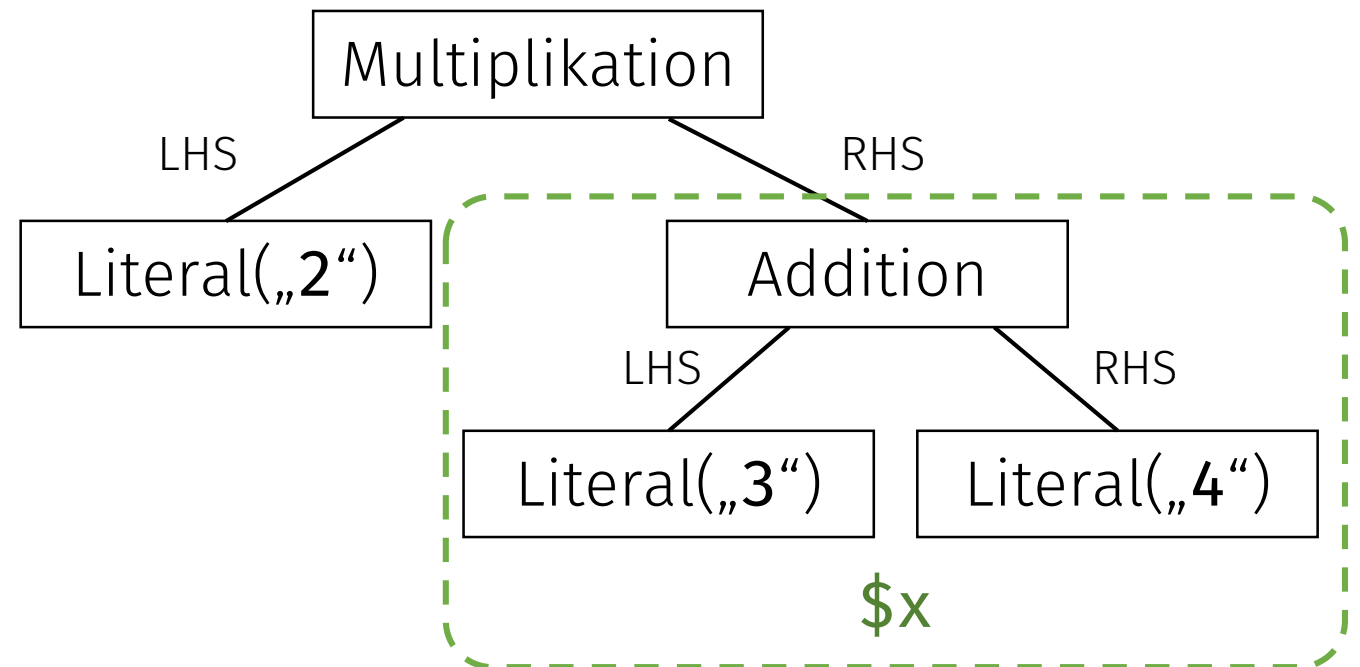
```
// 14 or 10? → 14  
println!("{}", twice!(3 + 4));
```

C Makros

```
#define TWICE(x) 2 * x
```

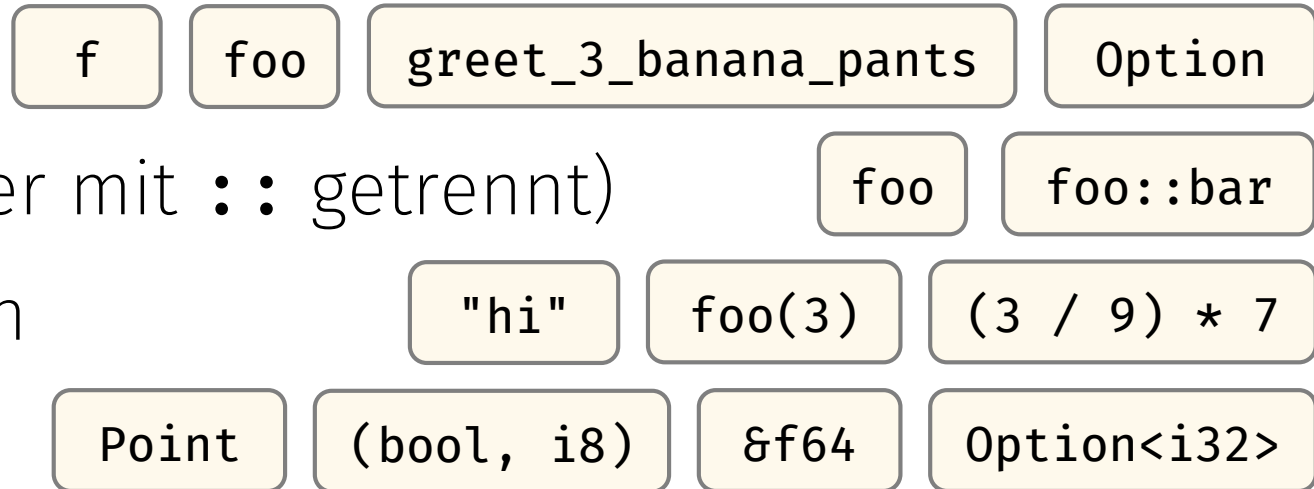
```
// 14 or 10? → 10  
printf("%d\n", TWICE(3 + 4));
```

- *In C*: reine Textersetzung
 - Führt manchmal zu unerwarteten Ergebnissen
 - Idiom zur Vermeidung: $2 * (x)$
- *In Rust*: Ersetzung von Syntaxbäumen



Syntaktische Formen

- **ident**: Identifier
- **path**: Pfade (Identifier mit `::` getrennt)
- **expr**: Eine Expression
- **ty**: Typ
- **stmt**, **block**, **item**, [u.v.m.](#)
- Sonderregeln für Token nach Metavariable!



Wichtig:

Beschreibt nur Syntax! Keine Semantik!

- Pfad muss nicht existieren
- Typ muss nicht definiert sein
- ...

Vorsicht mit Pfaden!

- „Aufruf“ setzt nur resultierenden Syntaxtree ein!
 - Pfade immer aus Sicht des aufrufenden Moduls
- ***Daher***: Absolute Pfade in Definition öffentlicher Makros!

- Für öffentliche Makros: spezielle Variable **\$crate**
 - Verweist auf Crate der Makrodefinition

Originale Definition von try!()

```
#[macro_export]
macro_rules! try {
    ($expr:expr) => (match $expr {
        $crate::result::Result::Ok(val) => val,
        $crate::result::Result::Err(err) => {
            return $crate::result::Result::Err($crate::convert::From::from(err))
        }
    })
}
```

Nicht öffentlichen Makros müssen oft nicht so explizit sein!

Wiederholungen

```
macro_rules! vec {
  ( $( $x:expr ),* ) => {
    { // make this a block-expr
      let mut v = Vec::new();
      $( v.push($x); )*
      v
    }
  };
}
```

```
let a = vec!(1, 2, 3);
// We can use (), [] and {} to delimit
// invocation ([] is idiomatic here!)
let b = vec![76, 7, 8];
```

- *Ziel*: Einfach Vektor mit Inhalt erstellen

Manuell

```
let mut v = Vec::new();
v.push(6);
v.push(7);
v.push(8);
```

- **`$($x:expr),*`**
 - 0 oder mehr Wiederholungen einer Expression (+ statt * → 1 oder mehr)
 - Getrennt mit „`,`“
 - `$($x:expr ,)*` erzwingt trailing „`,`“

Aufruf-Syntax und Hygiene

- Aufruf mit `()`, `[]` und `{}` erlaubt
 - `[]` für Array-artige Dinge
 - Sonstige, einzeilige Aufrufe: meist `()`
- Makrodefinition nutzt Variable `v`
 - Führt das zu Namensproblemen?
- Jede Variable gehört zu einem „*Expansion Context*“
 - Keine Probleme mit Namensüberschattung
 - Äußere Variablen nicht per Makro nutzbar (außer Name wird übergeben)

[Kapitel „Macro Hygiene“
im Rust Buch](#)

Mehr Beispiele

```
macro_rules! sum_all {
    ($($e:expr)*) => { $($e +)* 0 };
}
```

```
sum_all!();           // 0
sum_all!(1 2 3);     // 6
sum_all!(1 2 3 * 7); // 24
```

```
macro_rules! const_f64s {
    ( $( $name:ident = $value:expr ; )* ) => {
        $( const $name : f64 = $value ; )*
    }
}

const_f64s! {
    PI = 3.1415926;
    SQRT_2 = 1.4142135;
}
```

```
($type_name:ident < $( $gen_param:ident ),+ > , $constructor:ident) => {
    impl< $( $gen_param ),+ > Default for $type_name< $( $gen_param ),+ > {
        fn default() -> Self {
            Self::$constructor()
        }
    }
};
```

Default-Beispiel von Folie 8!

[Komplettes Beispiel \(Link\)](#)

Rekursion

```
macro_rules! to_xml {
  () => { "" };
  (: $e:expr) => { $e };
  ($tag:ident { $( $inner:tt )* } $( $tail:tt )* ) => {
    format!(
      "<{}>{}</{}>{}",
      stringify!($tag),
      to_xml!( $( $inner )* ),
      to_xml!( $( $tail )* ),
    )
  };
}
```

[Playground](#)

```
let s = to_xml! {
  html {
    head {
      title { : "hello" }
    }
    body {
      : "This is my beautiful website!"
    }
  }
};

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


Makros und das Modulsystem

- Makros werden früh beim Compilieren expandiert
 - Makros haben keinen Pfad, sind nicht public/private
 - Extraregeln für Makros

```
// Annotate to use macros from module/crate  
#[macro_use]  
mod foo {  
    macro_rules! bar { ... }  
}
```

```
// Explicitly export if other  
// other crates want to use this  
// macro  
#[macro_export]  
macro_rules! bar { ... }
```

- Makros müssen vor der Nutzung definiert werden!
 - Wie Funktionen in C: *vorher im Quellcode!*

Weitere Hinweise

- Makros debuggen ([Mehr Informationen](#))
 - `rustc -Z unstable-options --pretty expanded foo.rs`
 - Unstable Hilfsmakros wie `trace_macros!()` und `log_syntax!()`
- Wichtige Makros in **std** ([in Dokumentation](#))
 - Interessant: `include`, `include_str` und `include_bytes`
 - Außerdem: `concat`, `stringify`

[Komplettes Makro-Kapitel im Rust Buch](#)

Compiler Plugins

- Als *Syntax Extensions* oder interner *Pass*
- Extrem mächtig
 - Arbiträren Code zur Kompilierzeit mit Zugriff auf Quellcode ausführen
- Leider bisher alles *unstable* (benötigt *nightly* Compiler)
 - API-Design sehr schwierig

Beispiele folgen...

Regex

```
static re: Regex = regex!(r"[a-z]+@[a-z]\.de");
```

- Makro kompiliert Regex zur Kompilierzeit
 - Aus Gründen leider gerade langsamer als Standardmethode
 - Lieber **Regex::new(...)** und **lazy_static!()**
- Nebenbei: Rust Regex-Engine ist extrem schnell!
 - Plus kompletten Unicode-Support (eher selten)

serde (Serialisierung)

```
#[derive(Serialize, Deserialize, Debug)]
struct Point { x: i32, y: i32 }

let p = Point { x: 1, y: 2 };
let s = serde_json::to_string(&p).unwrap();

// Prints: {"x":1,"y":2}
println!("{}", s);

let out: Point =
    serde_json::from_str(&s).unwrap();
assert_eq!(out, p);
```

- *Custom derive* zur automatischen Implementierung von Traits
- Einfach (De-)Serialisierung in viele Formate möglich

[GitHub/serde-rs/serde](https://github.com/serde-rs/serde)

Diesel (ORM und QueryBuilder)

- Typen einfach in Datenbank speichern und abfragen

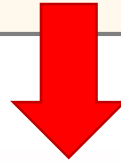
```
#[derive(Queryable, Insertable)]  
struct Point { x: i32, y: i32 }
```

- **infer_schema!("env:DATABASE_URL");**
 - Verbindet sich zur Kompilierzeit mit Datenbank
 - Fragt Tabellen der Datenbank ab
 - Generiert lokalen Code zur Repräsentation der Tabellen
- *Außerdem:* ORM mit QueryBuilder für optimale SQL Queries

Website: diesel.rs

SQL Validator

```
let bad_query = sql!(  
    "SELECT * FORM users WEHRE name = $1"  
);
```



error[E0308]: error: Invalid syntax at position 10: syntax error at or near "FORM"

--> <anon>:4:13

```
4 | "SELECT * FORM users WEHRE name = $1"  
  | ^~~~~~
```

Rust to Shader

```
const VERTEX: &'static str = glassful! {  
    #![version="110"]  
  
    #[attribute] static position: vec2 = UNINIT;  
    #[varying]   static color:     vec3 = UNINIT;  
  
    fn main() {  
        gl_Position = vec4(position, 0.0, 1.0);  
        color = vec3(0.5*(position + vec2(1.0, 1.0)), 0.0);  
    }  
};
```


Rocket (WebFramework)

```
#[get("/hello/<name>/<age>")]  
fn hello(name: &str, age: u8) -> String {  
    format!("Hello, {} year old named {}!", age, name)  
}
```

- Informationen über Handler in Attribute
 - HTTP-Method, Form-Parameter, ...
- Super modern, nutzt viele mächtige Rust Features

Website: rocket.rs

Macros 1.1

- Custom Derives werden teilweise stabilisiert
- Wird auf jeden Fall dann funktionieren:
 - Serde
 - Teile von Diesel
- In Rust 1.15 (nächste Version)