Advanced Rust (2026): Declarative Macros

Lecture 3

Lukáš Hozda

Spring 2026

MFF CUNI

Macro Model

Why Macros Exist

- Generics abstract over types.
- Functions abstract over values.
- Macros abstract over syntax.

Expansion Pipeline

- Parser builds token trees.
- `macro_rules!` patterns match token trees.
- Expansion happens before type checking.
- Expansion errors and type errors are different classes.

Minimal Macro

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

let v = twice!(21);
println!("{v}");
```

Fragment Specifiers

- `expr`, `ident`, `ty`, `pat`, `item`, `tt` are common.
- Choose strictest fragment that fits.
- Narrow matching gives better diagnostics.

```
macro_rules! make_getter {
    ($name:ident, $field:ident, $ty:ty) => {
        fn $name(&self) -> &$ty { &self.$field }
    };
}
```

Repetition

- `$( ... ),*` for comma-separated lists.
- `+` requires at least one element.
- Optional separators can be handled explicitly.

```
macro_rules! sum {
    ($first:expr $(, $rest:expr)*) => {{
        let mut acc = $first;
        $(acc += $rest;)*
        acc
    }};
}
```

Hygiene

- Macro-local identifiers are hygienic.
- Call-site identifiers are still usable through captures.
- Avoid accidental name capture surprises.

```
macro_rules! with_tmp {
    ($body:expr) => {{
        let tmp = 5;
        $body(tmp)
    }};
}
```

`tt` Muncher Pattern

- Recursive token consumption.
- Useful for mini DSL parsing.
- Keep recursion depth controlled.

Error-Friendly Macros

- Provide fallback arm with `compile_error!`.
- Prefer domain words in diagnostics.

```
macro_rules! cmd {
    (add $x:expr, $y:expr) => { $x + $y };
    (mul $x:expr, $y:expr) => { $x * $y };
    ($($rest:tt)*) => {
        compile_error!("cmd!: expected `add <expr>, <expr>` or `mul <expr>,
<expr>`");
    };
}
```

Macro-Generated Tests

- Macros are useful in tests to reduce repetition.
- Keep generated test names readable.

```
macro_rules! case {
    ($name:ident, $input:expr, $expected:expr) => {
        #[test]
        fn $name() {
            assert_eq!(solve($input), $expected);
        }
    };
}
```

API Stability Concerns

- Macro syntax is public API.
- Changing accepted token patterns can be a breaking change.
- Document macro grammar as seriously as function signatures.

`macro_rules!` vs Proc Macros

- `macro_rules!`: fast, stable, local syntax transforms.
- Proc macros: AST-aware, heavier tooling and debugging burden.
- In this course we prioritize robust `macro_rules!` patterns.

Practical Rules

- Keep macro surface small.
- Prefer function + macro wrapper when possible.
- Test expansion behavior with representative call sites.