Your own DSL in Rust

Experiences developing µcad

Micha Winkelmann

2026/11/03 @ RustLab 2026, Bologna, IT.



What is a DSL?

Let’s talk about *fundamental stuff*.

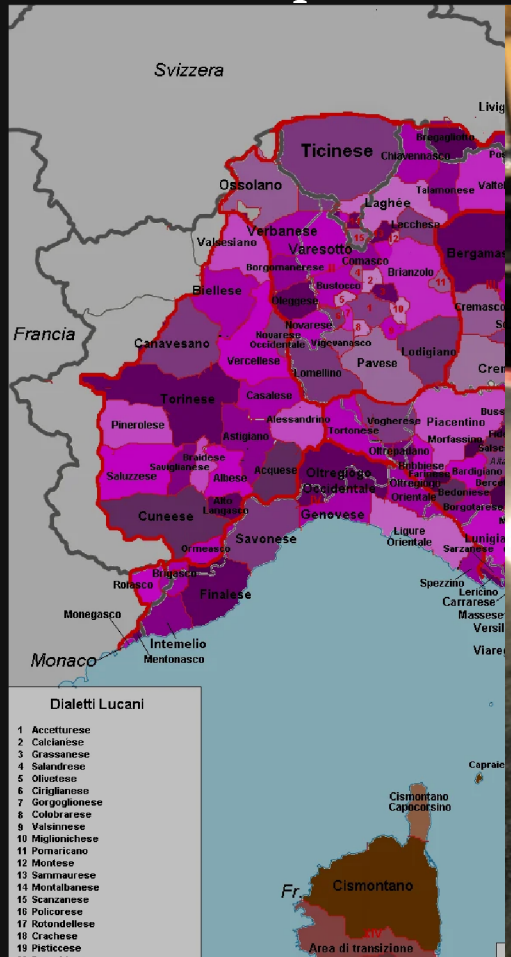
Let’s talk about **coffee** ☕

Let's talk about **Italy** 🇮🇹

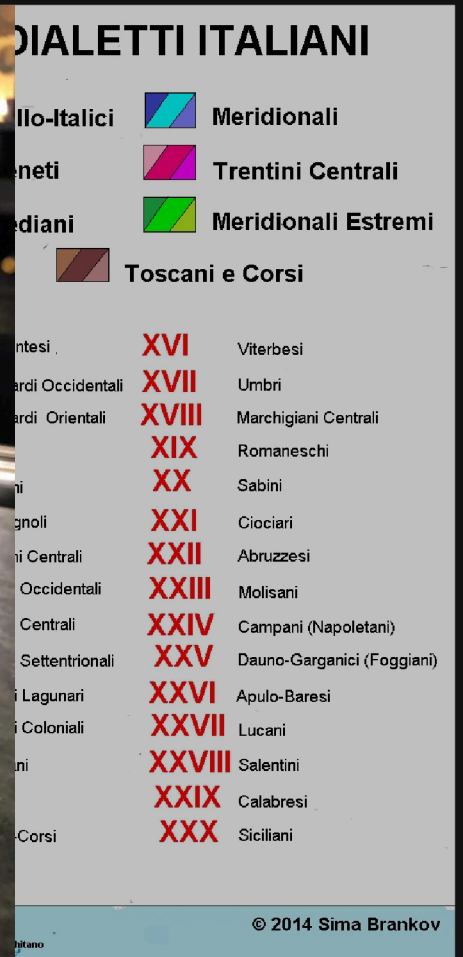


Italian dialects

Order a Capo in B in Trieste.







Order a Capo in B in Sicily.



Let’s inject the token `Capo` into a *Sicilian Compiler*.

Here is what’s gonna happen:

```rust
enum SicilianCompilerError {
    /// Expected a valid Sicilian primitive: `Espresso`, `GranitaConPanna`.
    #[error("`Capo` has not been declared")]
    UnknownIdentifier,

    /// A glass of foamed milk is not considered as coffee.
    #[error("Type `Capo` does not implement trait `Coffee`.")]
    TypeMismatch,

    /// The barista thinks you are addressing them, not ordering coffee.
    #[error("`Capo` means 'boss' in Sicilian language.")]
    KeywordError,
}
```

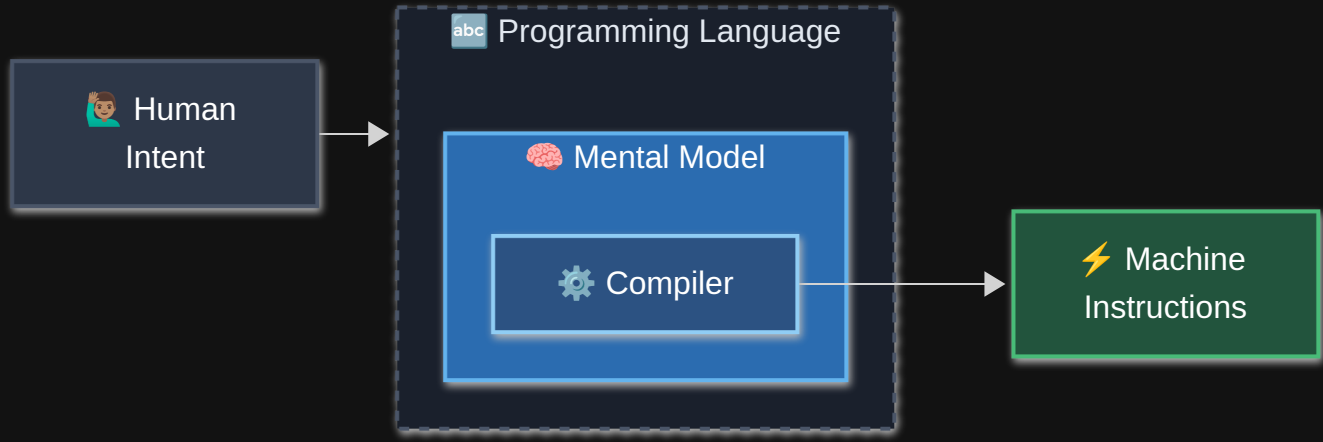
Human languages are *highly ambiguous* and *context dependent*.

Human languages:
ambiguous, informal, highly context-dependent.

Programming languages:
deterministic, formal, strict context.

Programming languages are *ergonomic notations*
for humans to **command computers**.

"OK Computer, please do *exactly* what's in my head."


🙋🏽 Human Intent
🔤 Programming Language
🧠 Mental Model
⚙️ Compiler
⚡ Machine Instructions

What are domain specific languages (DSLs)?

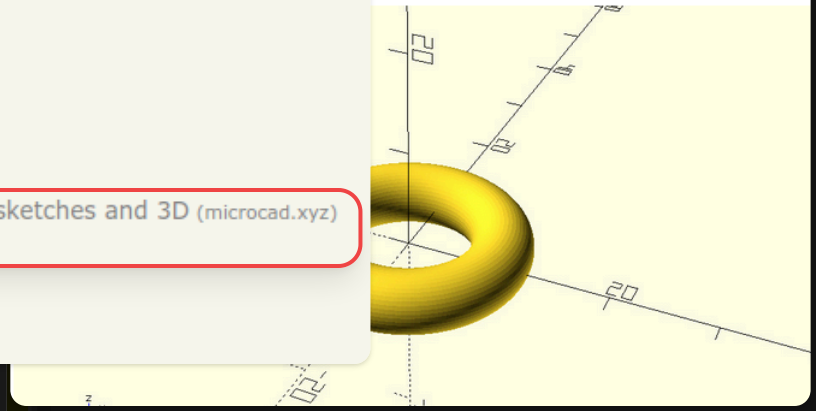
Live demo.

Coming from OpenSCAD

extruded_donut.scad

```
$fa = 1;
$fs = 0.4;
wheel_radius = 12;
```

```
- tyre_diameter/2, 0])
er);
```

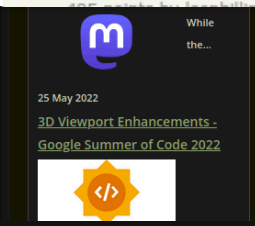


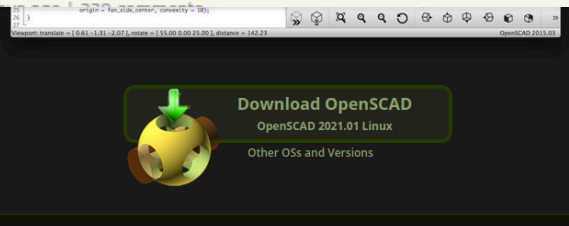
Hacker News new | past | comments | ask | show | jobs | submit | 2025-11-24

Stories from November 24, 2025 (UTC)

Go back a day, month, or year. Go forward a day.

1. ▲ **Shai-Hulud Returns: Over 300 NPM Packages Infected** (helixguard.ai)
   956 points by mrdosija 1 day ago | 727 comments
2. ▲ **Pebble Watch software is now open source** (ericmigi.com)
   1026 points by Larrikin 17 hours ago | 192 comments
3. ▲ **Claude Opus 4.5** (anthropic.com)
   979 points by adocomplete 17 hours ago | 445 comments
4. ▲ **µcad: New open source programming language that can generate 2D sketches and 3D** (microcad.xyz)
   383 points by todsacerdoti 1 day ago | 124 comments
5. ▲ **Chrome Jpegxl Issue Reopened** (chromium.org)
   258 points by markdog12 1 day ago | 117 comments
6. ▲ **Disney Lost Roger Rabbit** (pluralistic.net)





```
use std::ops::*;
use std::geo2d::Circle;

op polar_array(n: Integer, r: Length) {
    @input.translate(x = r)
        .rotate([1..n] / n * 360°) // Builder pattern
}

sketch RustGear(r = 50mm) { // `sketch` Keyword
    r_i = r * 80%; // Assignments

    rim = Circle(r) - Circle(r = r_i); // Binary operations

    tooth = Circle(r * 2%).polar_array(3, r * 10%).hull();
    teeth = tooth.polar_array(32, r);

    hub = Circle(r * 4%).polar_array(4, r * 10%).hull();
    hubs = hub.polar_array(5, r_i);

    holes = Circle(r * 6%).polar_array(5, r_i);

    rim | teeth | hubs - holes;
}
```

Why develop a DSL at all?
We have general purpose languages like Rust!
You can even develop your own **DSLs** using *proc-macros* in Rust!

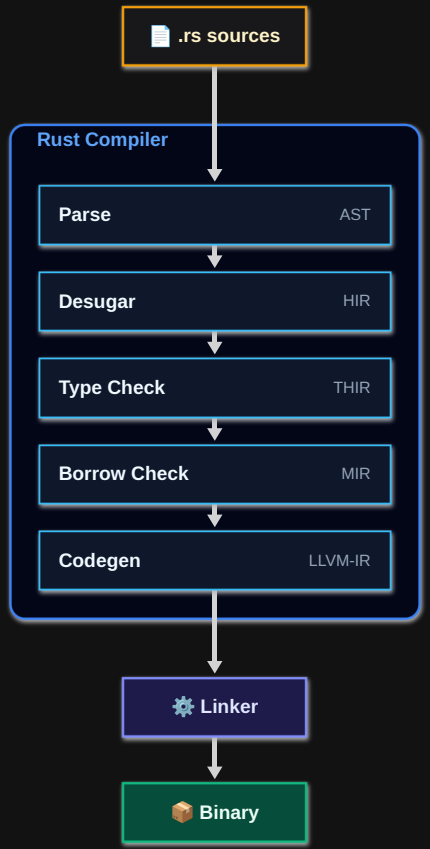
The Rust Software Engineer’s Mindset

"I don't care how hard the borrow checker fights me,
I can't afford any memory bugs at runtime."

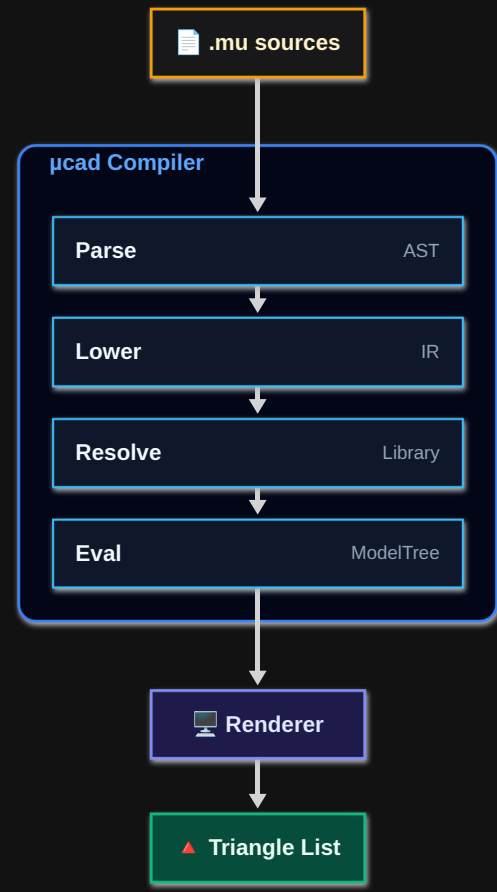
The CAD engineer’s Mindset

"I don't care what a compiler is,
I just want to design the part that's in my head."

Rust Toolchain



µcad Toolchain



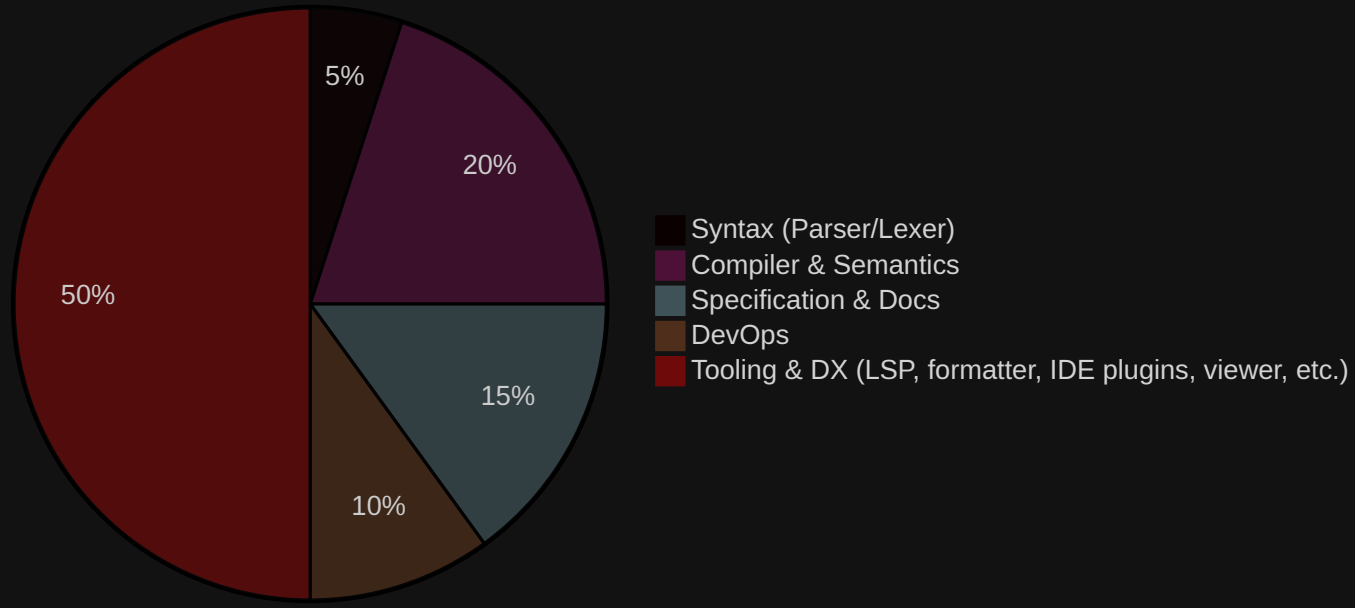
Reality check.

Main issues while developing:

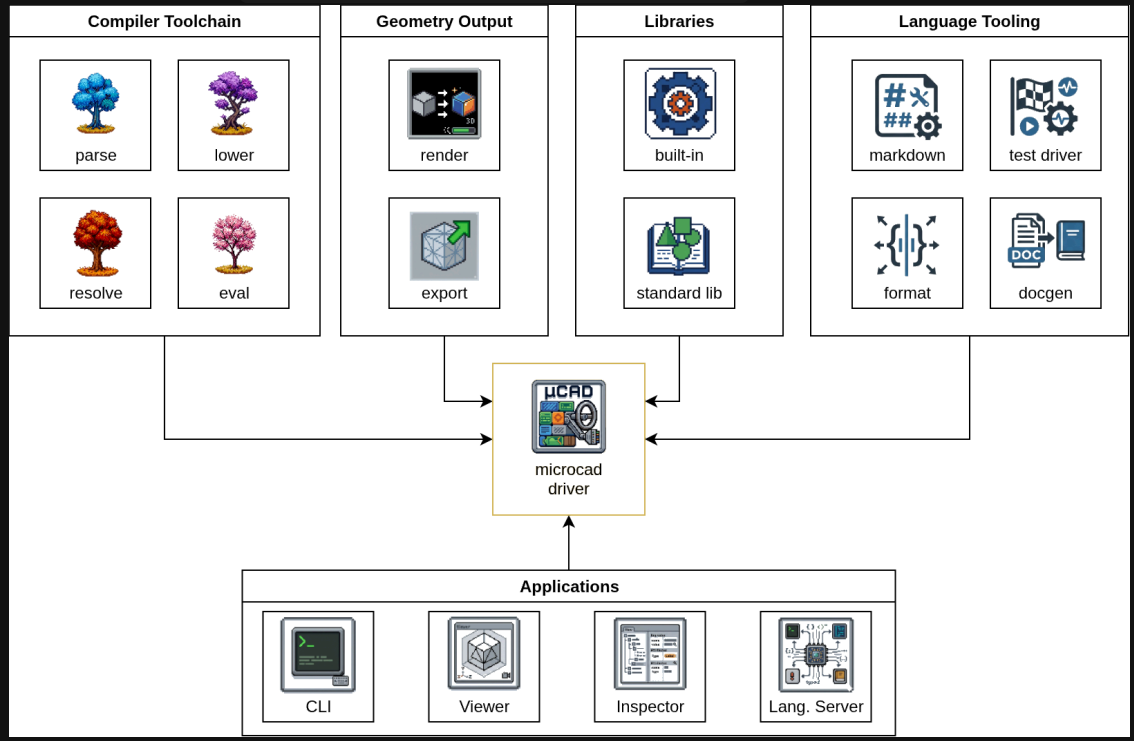
1. Underestimated DX and tooling.
2. Feature creep.
3. Compiler cable salad.

Underestimated DX and tooling.

The Development Costs of a DSL (own estimation)



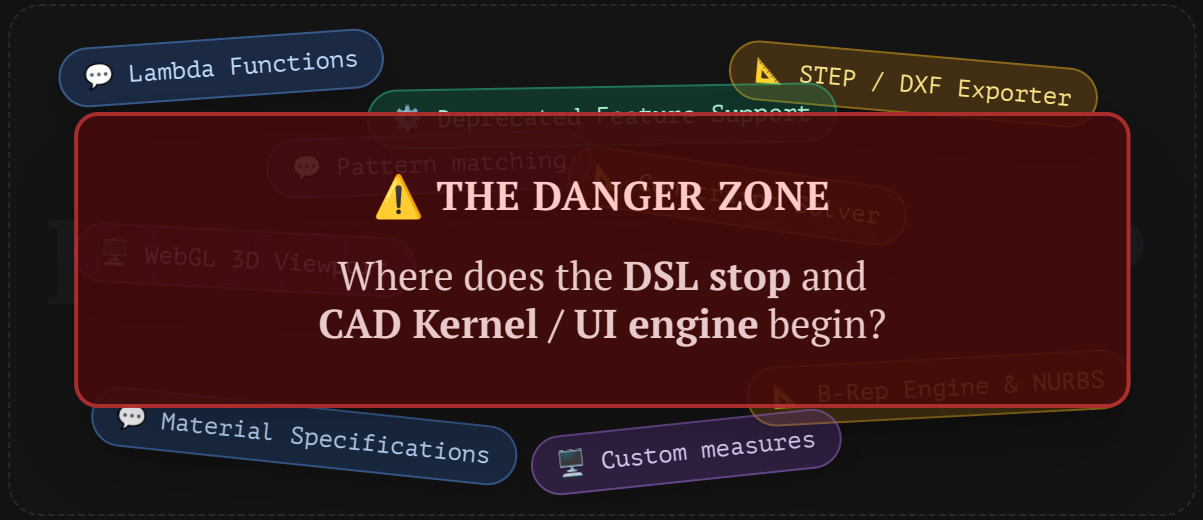
`microcad-driver` crate



Feature creep.

Unclear product definition and specification.

Where does µcad's domain "end"?



Insights:

🚫 No capacities and skills for developing a **CAD kernel**.

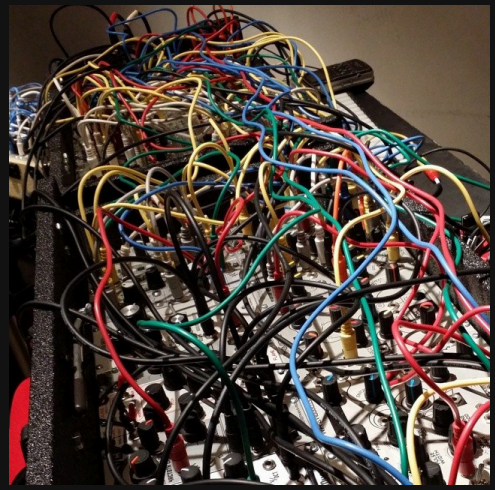
💖 Feedback on the **syntax & language design** have been overwhelming.

⚙️ The **compiler and its API** will be base for all tooling that comes on top.

Decision:

🚀 Focus on the **compiler and its API**.

Compiler cable salad.



Problems:

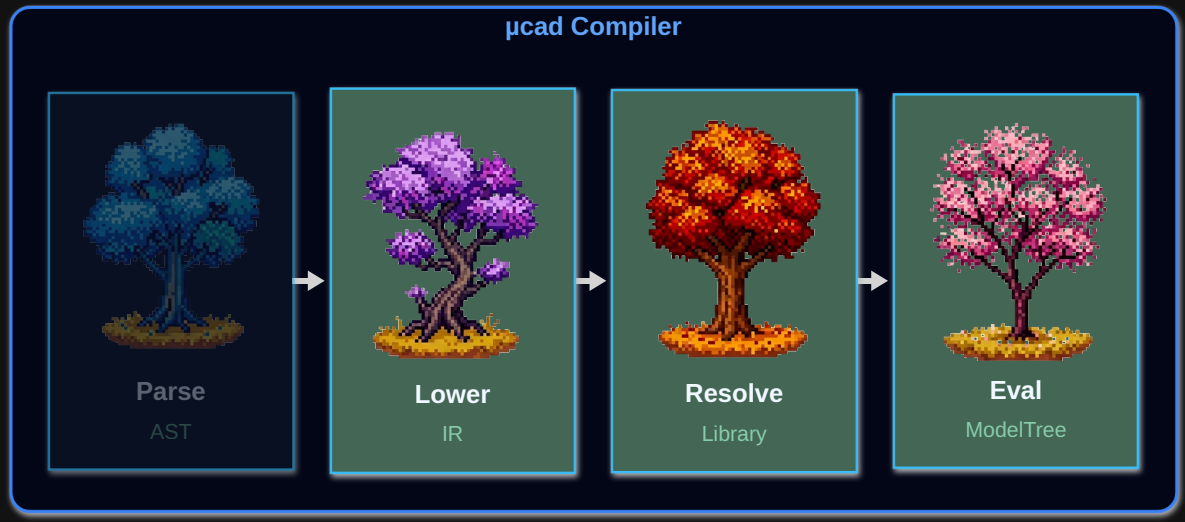
- Big monolith aka **big ball of mud**.
- Hard to fix bugs without breaking anything.
- No clearly defined **APIs** and **tests** between each stage.
- Where to start refactoring?

Let’s do it as a self-experiment.

Let’s take the risk and do a **rewrite**.

Considerations:

- **No new features during rewrite!**
- Let's focus only on Lower/Resolve/Eval:



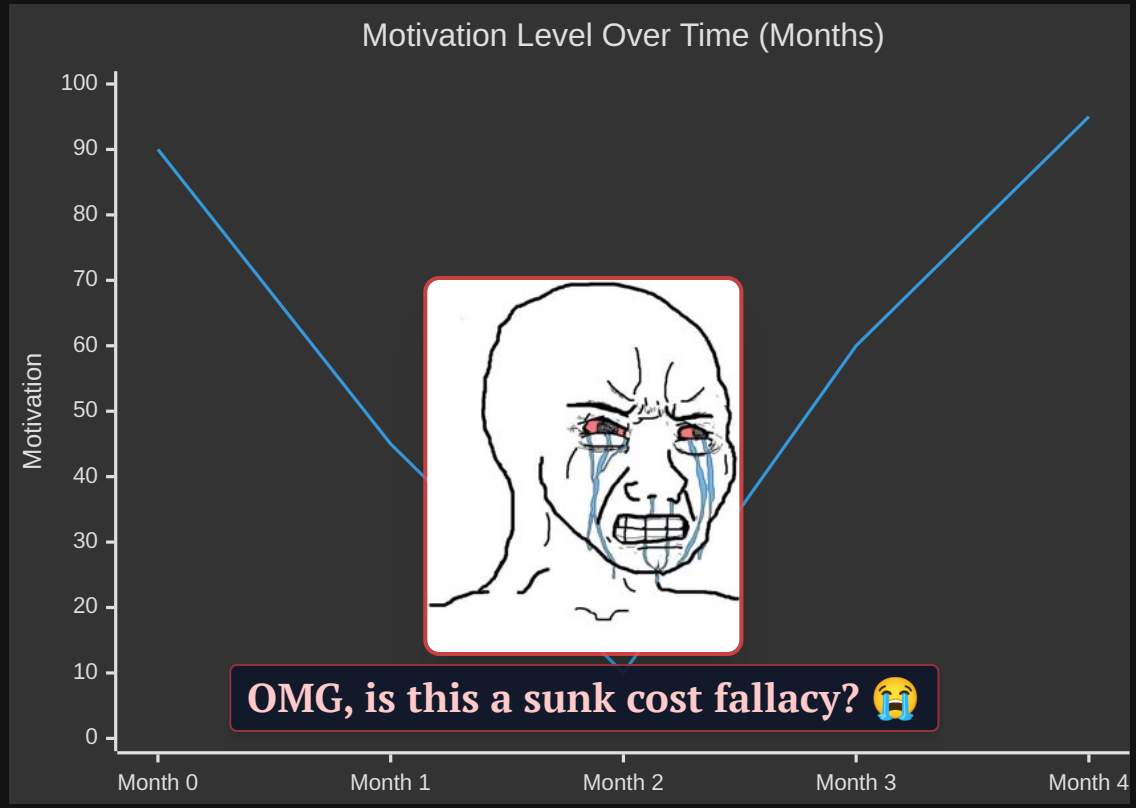
Constraints:

- Keep each stage in separate crates.
- Define `ModelTree` as compiler output and as input for renderer.

*"The **second system** is the most dangerous system a (hu)man ever designs…*
The general tendency is to over-architect it,
having learned all the things that were left out of the first one."

– Fred Brooks, **The Mythical Man-Month** (1975).

The Emotional Curve of the Rewrite 📈





1. Parse

```
pub fn parse(source: Source) -> CompilationResult<Ast> { ... }
```

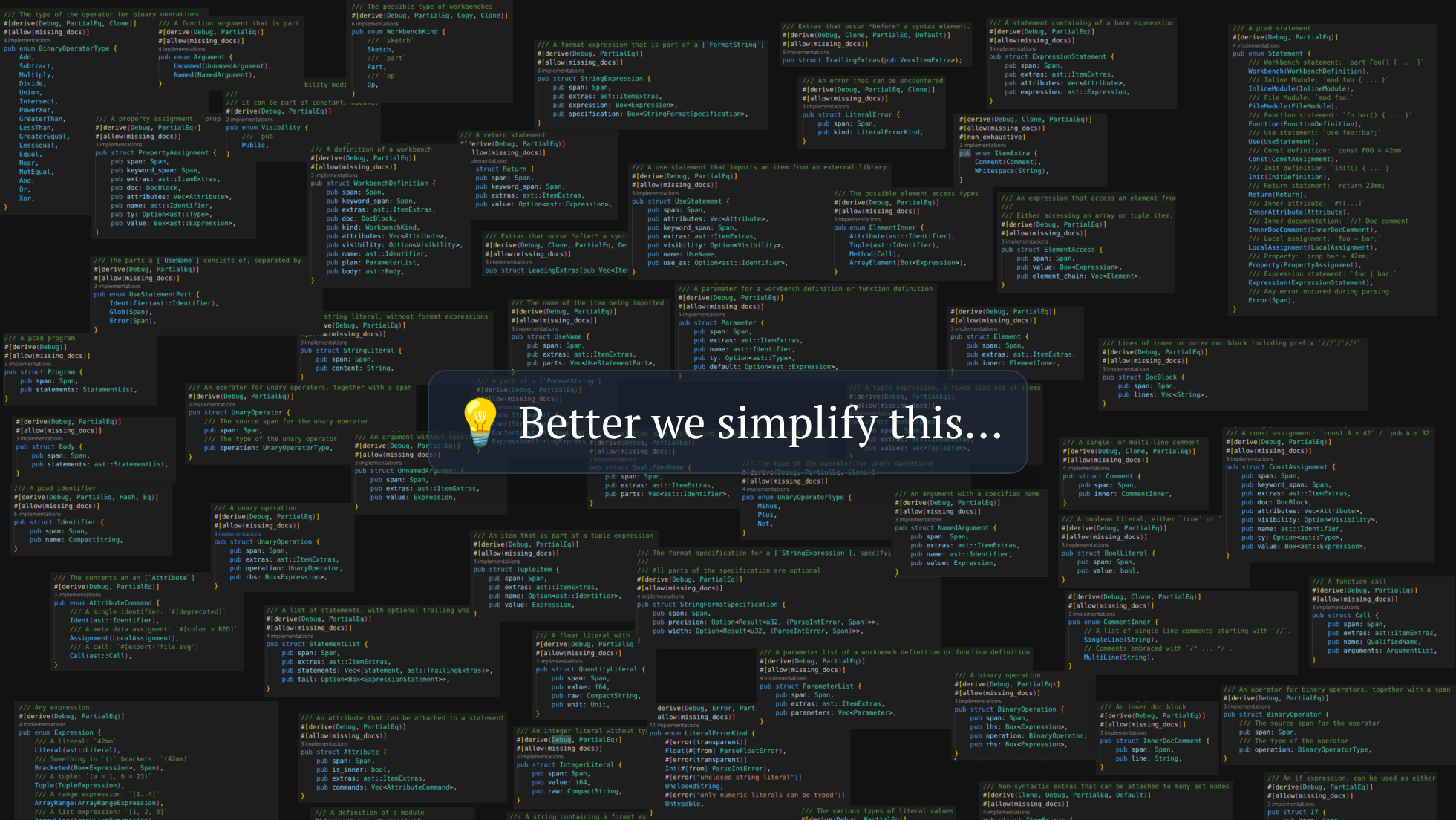
Let's consider this µcad expression:

```
2 + 3
```

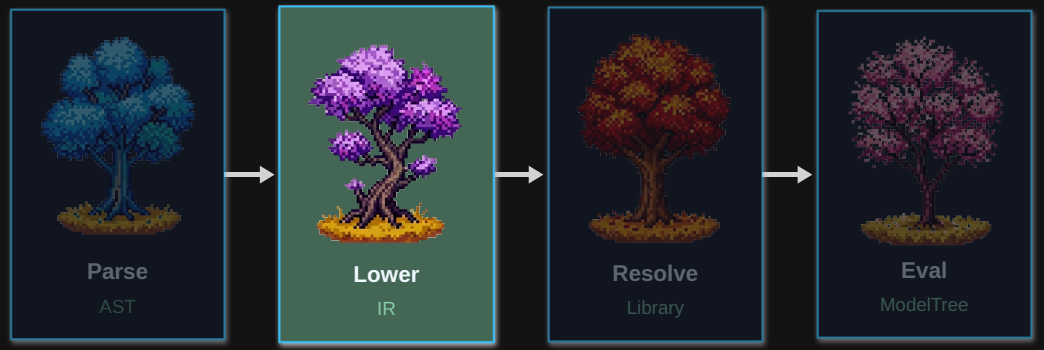
This will be parsed to this AST struct:

```
pub struct BinaryOperation {
    src_ref: SrcRef,
    lhs: Box<Expression> // 2
    op: BinaryOp, // +
    rhs: Box<Expression> // 3
}
```

This has to be done for all syntax elements…


Better we simplify this...

2. Lower



💡 *more simplification* at compile time, *less complexity* at run time.

```
// AST -> IR
pub fn lower(ast: &Ast) -> CompilationResult<Ir> { ... }
```

- `desugar` : Simplify syntax.
- `fold` : Call built-in functions at compile time.

```
trait Desugar<AstNode>: Sized { ... }

impl Desugar<ast::Expression>
    for ir::FunctionExpression {

    fn desugar(
        node: &ast::Expression,
        context: &mut LowerContext)
    -> LowerResult<ir::FunctionExpression> {
        // ...
    }
}
```

Let's desugar `ast::Expression`.

```
pub enum Expression {
    Literal(ast::Literal), // `42mm`
    Bracketed(Box<Expression>, Span), // (10mm)
    Tuple(TupleExpression), // (a, b)
    Range(RangeExpression), // [1..3]
    List(ListExpression), // [1, 2, 3]cm
    String(FormatString), // "String"
    SymbolPath(SymbolPath), // std::math::PI
    Marker(ast::Identifier), // @input
    UnaryOperation(UnaryOperation), // -1
    BinaryOperation(BinaryOperation), // 1 + 2
    Body(ast::Body), // { ... }
    Call(Call), // print("...")
    ElementAccess(ElementAccess), // circle.property
    If(If), // if { ... } else { ... }
    Error(Span), // Any error
}
```

```
// Desugared:
__mu::core::mul(
  __mu::core::list(1, 2, 3),
  10mm
)

// Original:
[1, 2, 3]cm
```

n = Number of `enum` variants.

🔴 AST: `n = 15`

```
pub enum FunctionExpression {
    Value(ir::ConstantValue), // 42mm
    Path(ir::Path), // std::math::PI
    Scope(ir::Scope), // { ... }
    If(ir::FunctionIf), // if { ... } else { ... }
    Call(ir::FunctionCall), // print()
    Invalid, // Error
}
```

🟢 IR: `n = 6`

🚀 Runtime complexity saved!

Folding expressions.

🔍 Now, let consider `list(1, 2, 3)`.

⚡ All function arguments `1, 2, 3` are constants!

💡 This means we can call this function at **compile-time**!

⚙️ We need to have a Rust impl of `list`.

🦀 Rust implementation (real code):

```rust
/// Construct a list.
#[builtin_fn(core::list(*) -> Any)] // Proc-macro to auto-register built-in fn.
pub fn list(
    args: Arguments, // 1, 2, 3
    _ctx: &mut BuiltinEvalContext,
) -> BuiltinResult<Value> {
    // Evaluates directly into an intermediate representation (IR) value
    Ok(Value::from(List::from(args)))
}
```

📦 `Value::List(vec![1,2,3])` will be stored in the IR.

🚀 We have optimized away the function call.

3. Resolve:

💡 *resolve* all symbols from all sources into a `Library`.

```
pub fn resolve(context: &mut ResolveContext) -> CompilationResult<Library> { ... }
```

- A `Library` is a symbol tree with a `mu.toml` manifest file.
- `resolve` crate provides convenient µcad Library API.
- Solid foundation for a potential µcad package manager.

4. Eval

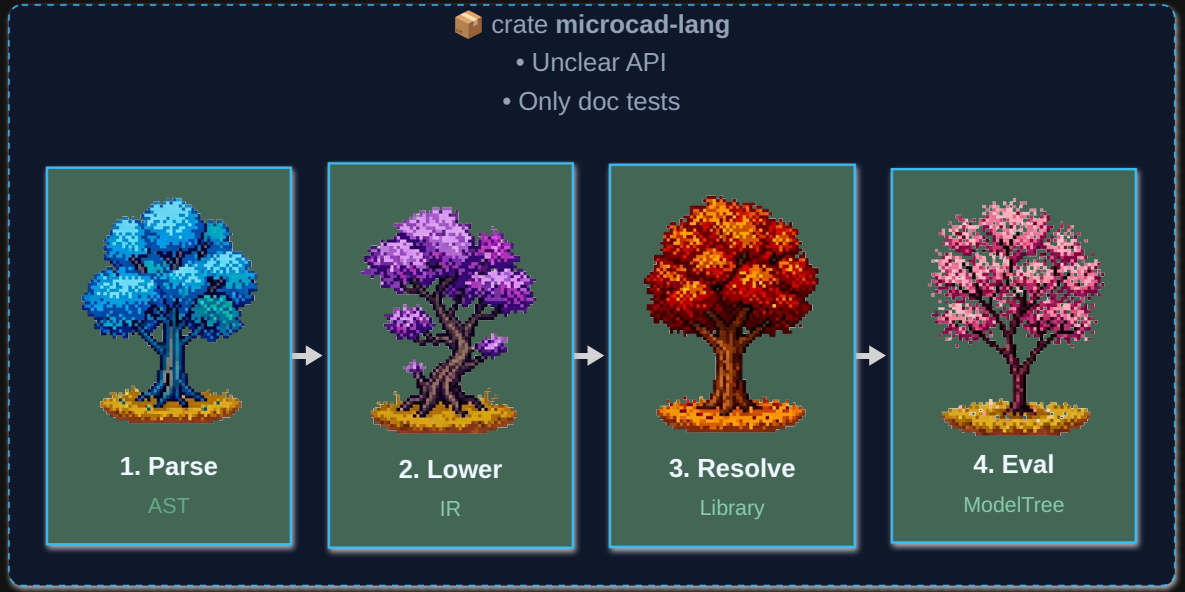


💡 Evaluation produces `ModelTree` from library symbols.

```
let library = Library::load("rust_gear.mu")?;
let model: ModelTree = library.eval(&mut ctx)?;
```

- A `ModelTree` is like a DOM for CAD kernels.
- This means we could potentially integrate it in FreeCAD or Blender etc.
- We still have the µcad minimal CSG kernel.

µcad compiler pipeline (before)



µcad compiler pipeline (after)

Rewrite evaluation.

Bad news:

⛔ No new features.

💥 LSP and IDE tooling still rudimentary.

Good news:

😋 Most cable salad eaten up: More solid APIs for each stage.

🛡️ Increased test coverage and specification.

🚀 Solid foundation for language tooling and evolution.

🧘 Personally: Gathered deep insights about building compilers.

Eating cable salad can be *healthy*.

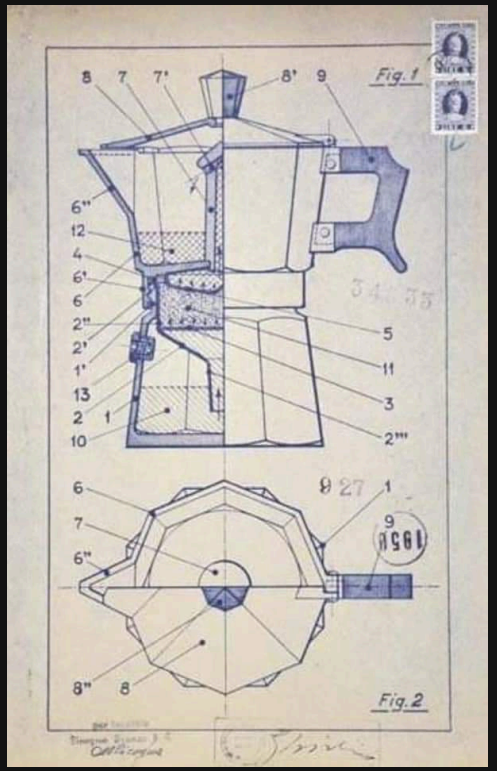
But it’s hard to swallow it up **all at once**.

µcad Roadmap 2027

- 🛠️ **DX:** Build tools on top of compiler API and write comprehensive documentation.
- 🌐 **Web-Viewer:** Bring interactive 3D model previews to the browser.
- 🔌 **Bridge Ecosystems:** Build a FreeCAD plugin.

Now, let’s talk again about coffee.





The original Bialetti design is from 1932 and has never changed since.

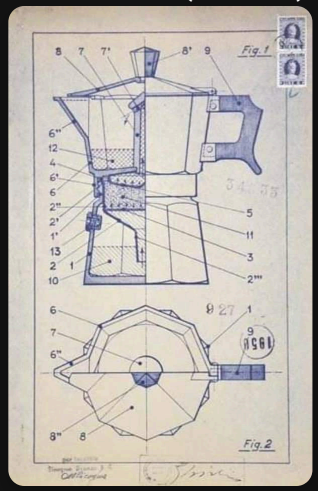
Where are we now?

AI Coffee Machine (2026)



- Touchscreens, sensors, Wi-Fi
- High maintenance friction
- Fails when network drops
- Mental Model: ???

Bialetti (1932)





- 3 mechanical components
- Zero software dependencies
- Perfectly executes its domain
- Mental Model: If it boils, it's ready.

4 Rules for DSL design

1. A DSL must **NOT** try to be a General-Purpose Language.
2. Keep compiler stages strictly isolated.
3. Be aware of language tooling.
4. Don’t build a hyper-engineered, cloud-connected espresso machine when all your domain needs is coffee 

💪 Special thanks to contributors:

`@hryamzik @terts @icewind @hdhia @Evrey`

🌐 Project Website: https://microcad.xyz

⚙️ Code repository: https://codeberg.org/microcad/microcad

📑 Slides: https://rust.services/talks/microcad_rustlab2026.pdf



Thanks!