# Macro

Yihao Sun Utah State University, Sept 2026

# What’s missing

- We learnt the logic meaning of a query , but….



# Hello World!

println!(“Hello world”);

Why there is a “!” ?

println!(“Hello {}”, 41); 

println!(“Hello {} {}”, 41); 

# First Peek

| | |
|---|---|
| stringify!(1) | “1” |
| stringify!(hello) | “hello” |
| stringify!(“hello”) | “\”hello\”” |
| stringify!(1 + 1) | “1 + 1” |

# Type of Macro

Type?

stringify!(1 + 1)




Any code?

# Macro is a vintage idea to program

1958

Fortran

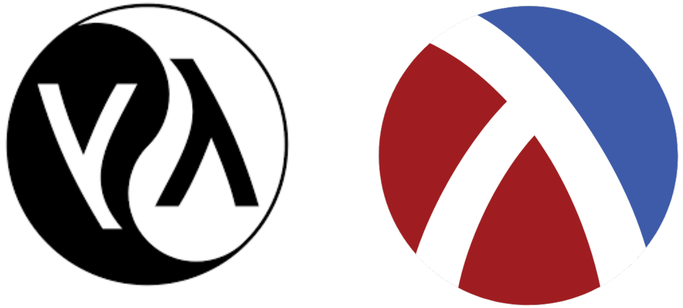


1963

LISP




Now





# “Code is Data”

```
(+ 2 (* 3 4))
; 14

(define code '(+ 2 (* 3 4)))
(car code)
; '+
```

```
(define stx #'(+ 2 (* 3 4)))

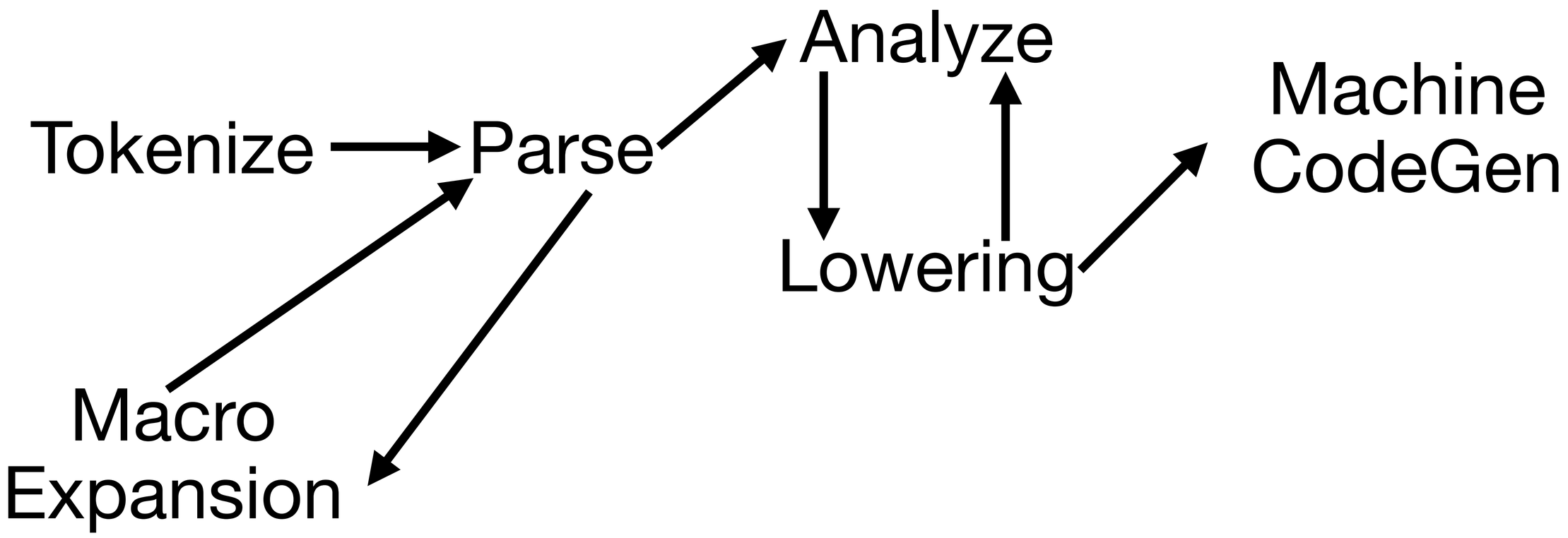
(syntax->datum stx)

; '(+ 2 (* 3 4))

(length (syntax->list stx))

; 3
```

# When Macro is Computed?



# Match and Replace

```
swap!(red, green);


macro_rules! swap {
    ($a:ident, $b:ident) => {{
        let tmp = $a;
        $a = $b;
        $b = tmp;
    }};
}
```

# Try Another Pattern

```
rotate!(red, green, blue);


macro_rules! rotate {
    ($a:ident, $b:ident) => {
        swap!($a, $b)
    };
    ($a:ident, $b:ident, $c:ident) => {{
        swap!($a, $b);
        swap!($b, $c);
    }};
}
```

# Recursive Patterns

```
rotate!(a, b, c, d);


macro_rules! rotate {
    ($a:ident, $b:ident) => {{
        swap!($a, $b);
    }};
    ($a:ident, $b:ident, $($rest:ident),+) => {{
        swap!($a, $b);
        rotate!($b, $($rest),+);
    }};
}
```

# Recursive Expansion

```
rotate!(a, b, c, d);


swap!(a, b);
rotate!(b, c, d);
    ↓
swap!(a, b);
swap!(b, c);
rotate!(c, d);
    ↓
swap!(a, b);
swap!(b, c);
swap!(c, d);
```

# Rotate with Templates

```
// C++: class templates
template<class... T> struct Rotate;
template<class A, class B>
struct Rotate<A, B> {
  static void run(A& a, B& b) {
    std::swap(a, b);
  }
};
template<class A, class B, class... R>
struct Rotate<A, B, R...> {
  static void run(A& a, B& b, R&... r) {
    std::swap(a, b);
    Rotate<B, R...>::run(b, r...);
  }
};
template<class... T> void rotate(T&... x) {
  Rotate<T...>::run(x...);
}
rotate(a, b, c, d);
```

```
// Rust: syntax patterns
macro_rules! rotate {
  ($a:ident, $b:ident) => {{
    swap!($a, $b);
  }};
  ($a:ident, $b:ident,
   $($rest:ident),+) => {{
    swap!($a, $b);
    rotate!($b, $($rest),+);
  }};
}
rotate!(a, b, c, d);
```

# Redflag: Hygiene

```
let (mut tmp, mut other) = (5, 6);


swap!(tmp, other);
// Conceptual expansion
{
    let macro_tmp = tmp;
    tmp = other;
    other = macro_tmp;
}
assert_eq!((tmp, other), (6, 5));
```

# Comp Zoo

| | Macro Must safely Bind | At least its AST safe | Write String is also Macro |
|---|---|---|---|
| Macro Must Name Macro |  |  |  |
| Code Gen is Macro | (Meta) OCaml | Kotlin |  |
| As long as Its compile time… | Haskell |  | ZIG |

# What Code Is Emitted?

```
add_code!(2, 3)


macro_rules! add_code {
    ($a:literal, $b:literal) => { $a + $b };
}
// Expansion
 2 + 3
const FIVE: i32 = add_code!(2, 3);
// FIVE == 5
```

# Code as a Value

Compute following math at compile time:

```
add(2, multiply(3, 4))
```

# Rust: Types for Code

```
add(2, 3)


enum IntegerExpr {
    Literal { value: i64 },
    Call(IntegerCall),
}

struct IntegerCall {
    function: String,
    arguments: Vec<IntegerExpr>,
}
```

# An Expression Value

```
add(2, 3)


IntegerExpr::Call(IntegerCall {
    function: "add".into(),
    arguments: vec![
        IntegerExpr::Literal { value: 2 },
        IntegerExpr::Literal { value: 3 },
    ],
})
```

# Syntax Has Types

```
2       add       "add"

let number: LitInt = parse_quote!(2);

let name: Ident = parse_quote!(add);

let text: LitStr = parse_quote!("add");
```

# Choose a Variant

```
2         add(2, 3)


#[derive(Parse, ToTokens)]
enum IntegerExpr {
    #[parse(peek = LitInt)]
    Literal { value: LitInt },
    Call(IntegerCall),
}
```

# Parse a Call

```
add(2, multiply(3, 4))


#[derive(Parse, ToTokens)]
struct IntegerCall {
    function: Ident,
    #[syn(parenthesized)]
    paren_token: token::Paren,
    #[syn(in = paren_token)]
    #[parse(Punctuated::parse_terminated)]
    arguments: Punctuated<IntegerExpr, Token![,]>,
}
```

# Code → Value → Code

```
add(2, multiply(3, 4))

let input = quote! { add(2, multiply(3, 4)) };

let expression: IntegerExpr =
    syn::parse2(input)?;

let output = quote! { #expression };

// output: add(2, multiply(3, 4))
```

# Give the Code a Meaning

```
add(2, multiply(3, 4))


fn evaluate(expr: &IntegerExpr)
    -> syn::Result<i64>
{
    match expr {
        IntegerExpr::Literal { value } =>
            value.base10_parse(),
        IntegerExpr::Call(call) =>
            evaluate_call(call),
    }
}
```

# Compute with Ordinary Rust

```
// inside evaluate_call(call: &IntegerCall)

let name = call.function.to_string();
let args = &call.arguments;
match (name.as_str(), args.len()) {
    ("add", 2) =>
        Ok(evaluate(&args[0])? + evaluate(&args[1])?),
    ("multiply", 2) =>
        Ok(evaluate(&args[0])? * evaluate(&args[1])?),
    _ => Err(syn::Error::new_spanned(
        call, "expected add/multiply with 2 arguments"
    )),
}
```

# Compute the Replacement

```
eval_integer!(add(2, multiply(3, 4)))


let expression: IntegerExpr =
    syn::parse2(input)?;
let value = evaluate(&expression)?;
// value == 14
Ok(quote! { #value })
// Emits: 14i64
```

# Back to Relations

```
relation road(src: City, dst: City);

let road: HashSet<(City, City)> =
    HashSet::new();
```

# Use the Fields

```
relation road(src: City, dst: City);


let decl: RelationDecl = syn::parse2(input)?;

let name = &decl.name;
let types = decl.columns.iter().map(|c| &c.ty);

quote! {
    let #name:
        ::std::collections::HashSet<(#(#types,)*)> =
        ::std::collections::HashSet::new();
}
```