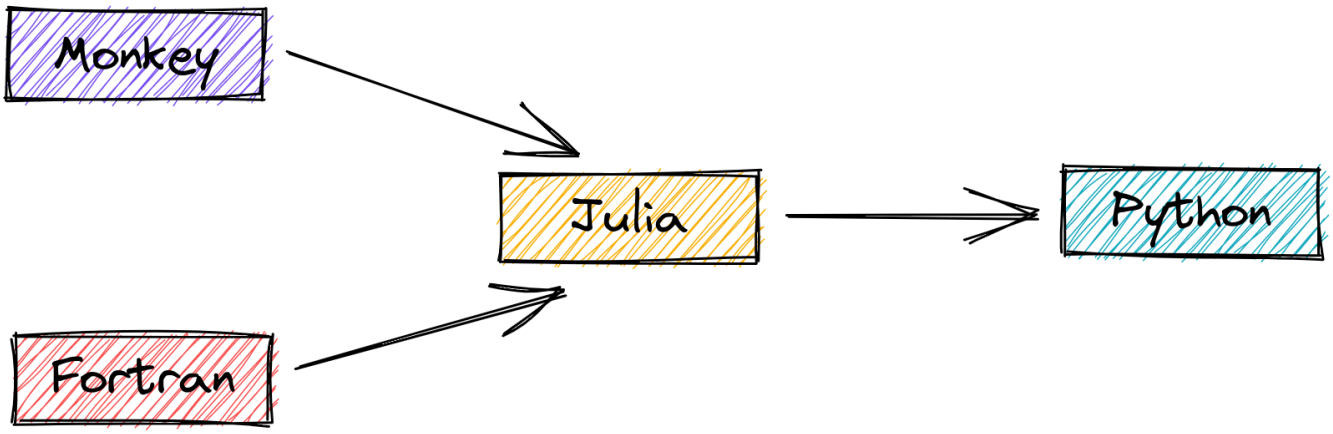
# Introduction

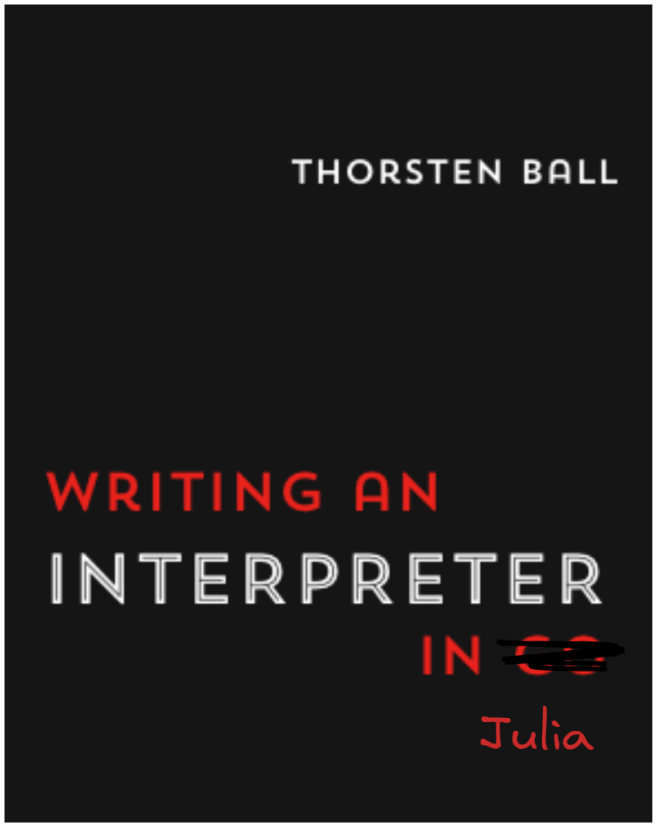



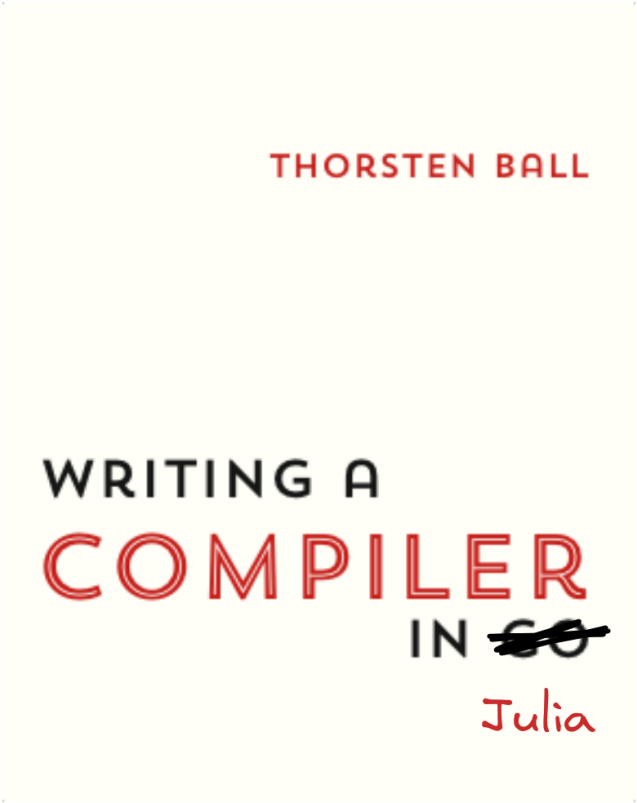

报告人：吴自华 @lucifer1004

Present

```
begin
    using Colors    , ColorVectorSpace    , ImageShow    , FileIO    , ImageIO
    using PlutoUI
    using HypertextLiteral    : @htl, @htl_str
    using MonkeyLang
    using Jl2Py
    using Taichi
    html"<button onclick=present()>Present</button>"
end
```

# MonkeyLang.jl

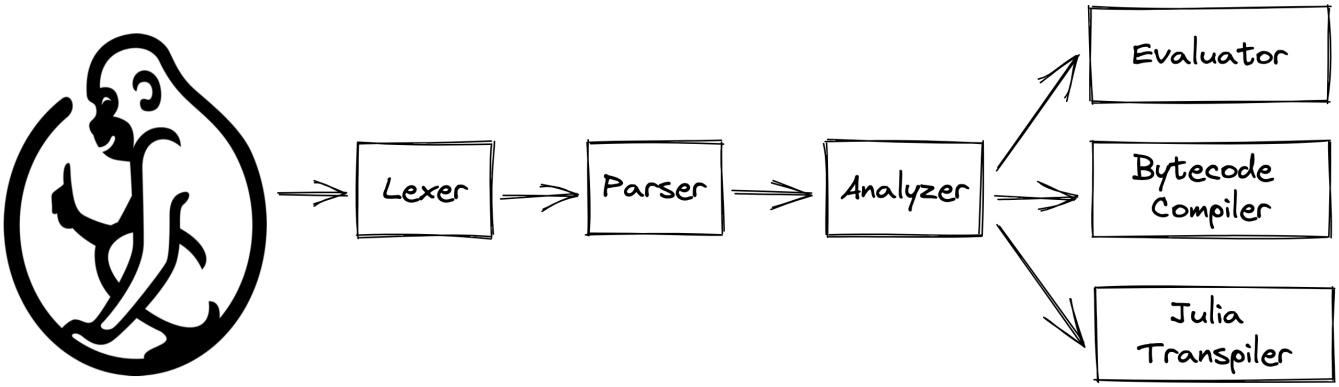



An example Monkey program:

```
let fibonacci = fn(x) {
  if (x == 0) {
    0
  } else {
    if (x == 1) {
      return 1;
    } else {
      fibonacci(x - 1) + fibonacci(x - 2);
    }
  }
};
```

# Project Structure




# Lexer

Convert string of code into tokens.

```
mutable struct Lexer
    input::String
    next::Union{Tuple{Char, Int}, Nothing}
end
```

We need to manage the state ourselves when using Julia's iterators.

```
lexer =  Lexer("fn (x) { return x + x; }", ('f', 2))
```

```
lexer = MonkeyLang.Lexer("fn (x) { return x + x; }")
```

# Parser

Parse the tokens into statements and expressions.

```
mutable struct Parser
    l::Lexer
    errors::Vector{ErrorObj}
    cur_token::Token
    peek_token::Token
    prefix_parse_functions::Dict{TokenType, Function}
    infix_parse_functions::Dict{TokenType, Function}
end
```

```
Parser(Lexer("fn (x) { return (x + x); }", ('x', 6)), [], Token(FUNCTION::TokenType = 23,
```

```
let
    lexer = MonkeyLang.Lexer("fn (x) { return (x + x); }")
    parser = MonkeyLang.Parser(lexer)
end
```

```
fn(x) { return (x + x); }
```

```
let
    lexer = MonkeyLang.Lexer("fn (x) { return (x + x); }")
    parser = MonkeyLang.Parser(lexer)
    MonkeyLang.parse!(parser)
end
```

# Analyzer

Before evaluation, we need to do some syntax-checking, e.g., forbidding redeclaration of variables.

```
ERROR: a is already defined
```

```
monkey_eval"""
let a = 2;
let a = 3;
"""
```

```
ERROR: a is already defined
```

```
3
```

```
monkey_eval"""
let a = 2;
a = 3;
"""
```

# Evaluator

Evaluate the statements and expressions directly. Variables are kept in nested environments.

```
struct Environment
    store::Dict{String, Object}
    outer::Union{Environment, Nothing}
    input::IO
    output::IO

    Environment(; input = stdin, output = stdout) = new(Dict(), nothing, input, outp
ut)
    Environment(outer::Environment) = new(Dict(), outer, outer.input, outer.output)
end
```

For example, the following function deals with integer arithmetic and comparisons.

```
function evaluate_infix_expression(operator::String, left::IntegerObj, right::Intege
rObj)
    if operator == "+"
        return IntegerObj(left.value + right.value)
    elseif operator == "-"
        return IntegerObj(left.value - right.value)
    elseif operator == "*"
        return IntegerObj(left.value * right.value)
    elseif operator == "/"
        if right.value == 0
            return ErrorObj("divide error: division by zero")
        end
        return IntegerObj(left.value ÷ right.value)
    elseif operator == "<"
        return left.value < right.value ? _TRUE : _FALSE
    elseif operator == ">"
        return left.value > right.value ? _TRUE : _FALSE
    elseif operator == "=="
        return left.value == right.value ? _TRUE : _FALSE
    elseif operator == "!="
        return left.value != right.value ? _TRUE : _FALSE
    else
        return ErrorObj("unknown operator: " * type_of(left) * " " * operator * " "
*
                        type_of(right))
    end
end
```

# Bonus: Macros (the Lost Chapter)

`MonkeyLang.jl` supports `quote` and `unquote` based macros.

```
let unless = macro(condition, consequence, alternative) {
    quote(if (!(unquote(condition))) {
        unquote(consequence);
    } else {
        unquote(alternative);
    });
};

unless(10 > 5, puts("not greater"), puts("greater"));
```

```
null
```

```
greater
```

# Bytecode Compiler

The code is compiled and then evaluated in a Bytecode VM.

```
mutable struct Frame
    cl::ClosureObj
    ip::Int
    base_ptr::Int

    Frame(cl::ClosureObj, base_ptr::Int) = new(cl, 0, base_ptr)
end

mutable struct VM
    constants::Vector{Object}
    stack::Vector{Object}
    sp::Int64
    globals::Vector{Object}
    frames::Vector{Frame}
    input::IO
    output::IO

    function VM(bc::ByteCode, globals::Vector{Object} = Object[]; input = stdin,
                output = stdout)
        begin
            main_fn = CompiledFunctionObj(bc.instructions, 0, 0, false)
            main_closure = ClosureObj(main_fn, [])
            main_frame = Frame(main_closure, 0)
            frames = [main_frame]
            new(bc.constants, [], 1, globals, frames, input, output)
        end
    end
end
```

# Julia Transpiler

Transpile Monkey statements and expressions to Julia `Expr`s.

```
# ...

transpile(bs::MonkeyLang.BlockStatement)::Expr = quote
    $(map(transpile, bs.statements)...)
end

transpile(::MonkeyLang.BreakStatement)::Expr = Expr(:break)

transpile(::MonkeyLang.ContinueStatement)::Expr = Expr(:continue)

transpile(es::MonkeyLang.ExpressionStatement) = transpile(es.expression)

transpile(ls::MonkeyLang.LetStatement)::Expr = begin
    value = transpile(ls.value)

    if isa(value, Expr) && value.head == :function
        parameters = value.args[1].args
        body = value.args[2]
        Expr(:function, Expr(:call, Symbol(ls.name.value), parameters...), body)
    else
        Expr(:(=), Symbol(ls.name.value), value)
    end
end

# ...
```

```
quote
    #= /home/ubuntu/.julia/packages/MonkeyLang/oGKHW/src/transpilers/julia/julia_transpi
    __IS_TRUTHY(a) = begin
            #= /home/ubuntu/.julia/packages/MonkeyLang/oGKHW/src/transpilers/julia/julia
            a != false && a != nothing
        end
    #= /home/ubuntu/.julia/packages/MonkeyLang/oGKHW/src/transpilers/julia/julia_transpi
    __IS_FALSY(a) = begin
            #= /home/ubuntu/.julia/packages/MonkeyLang/oGKHW/src/transpilers/julia/julia
            !(__IS_TRUTHY(a))
        end
    #= /home/ubuntu/.julia/packages/MonkeyLang/oGKHW/src/transpilers/julia/julia_transpi
    __WRAPPED_GETINDEX(v::Vector, id::Int) = begin
            #= /home/ubuntu/.julia/packages/MonkeyLang/oGKHW/src/transpilers/julia/julia
            if 0 <= id < length(v)
                v[id + 1]
            else
                nothing
            end
        end
    #= /home/ubuntu/.julia/packages/MonkeyLang/oGKHW/src/transpilers/julia/julia_transpi
    __WRAPPED_GETINDEX(d::Dict, key) = begin
            #= /home/ubuntu/.julia/packages/MonkeyLang/oGKHW/src/transpilers/julia/julia
            get(d, key, nothing)
        end
    #= /home/ubuntu/.julia/packages/MonkeyLang/oGKHW/src/transpilers/julia/julia_transpi
    __WRAPPED_STRING(a) = begin
            #= /home/ubuntu/.julia/packages/MonkeyLang/oGKHW/src/transpilers/julia/julia
            Base.string(a)
        end
    #= /home/ubuntu/.julia/packages/MonkeyLang/oGKHW/src/transpilers/julia/julia_transpi
    __WRAPPED_STRING(::Nothing) = begin
```

```
Text(repr(MonkeyLang.Transpilers.JuliaTranspiler.transpile(
    """
    let f = fn(x) {
        return x * x;
    }

    puts(f(12));
    """
)))
```

# Julia's `@x_str` macros

4

```
monkey_eval"""
let f = fn(x) {
    puts("Hello" + " " + "Monkey");
    return x * 2;
}

f(2)
"""
```

```
Hello Monkey
```

4

```
monkey_vm"""
let f = fn(x) {
    puts("Hello" + " " + "Monkey");
    return x * 2;
}

f(2)
"""
```

```
Hello Monkey
```

4

```
monkey_julia"""
let f = fn(x) {
    puts("Hello" + " " + "Monkey");
    return x * 2;
}

f(2)
"""
```

```
Hello Monkey
```

Main.var"workspace#3".Tmp

```
module Tmp
    using MonkeyLang

    println(monkey_julia"""
    let f = fn(x) {
        puts("Hello" + " " + "Monkey");
        return x * 2;
    }"""(2))
end
```

```
Hello Monkey
4
```

# Interpolation in `@x_str` macros

```
macro monkey_eval_str(code::String)
    quote
        evaluate($(esc(Meta.parse("\"$(escape_string(code))\""))))
    end
end
```

4

```
let
    name = "Monkey"

    monkey_eval"""
    let f = fn(x) {
        puts("Hello" + " " + "$name");
        return x * 2;
    }

    f(2)
    """
end
```

```
Hello Monkey
```

# REPL

Use `PackageCompiler.jl` to compile an executable.

# Julia has a friendly community

Discourse Thread

A big issue: VM was slower than the interpreter. How did this happen?

It turned out to be the misuse of `Ref`, which introduced type instability.

Compare the following two:

```
struct A{T}
    r::Ref{T}
end
```

```
struct A2{T}
    r::Base.RefValue{T}
end
```

In the first one, `Ref{T}` is not concrete!

# Future improvements

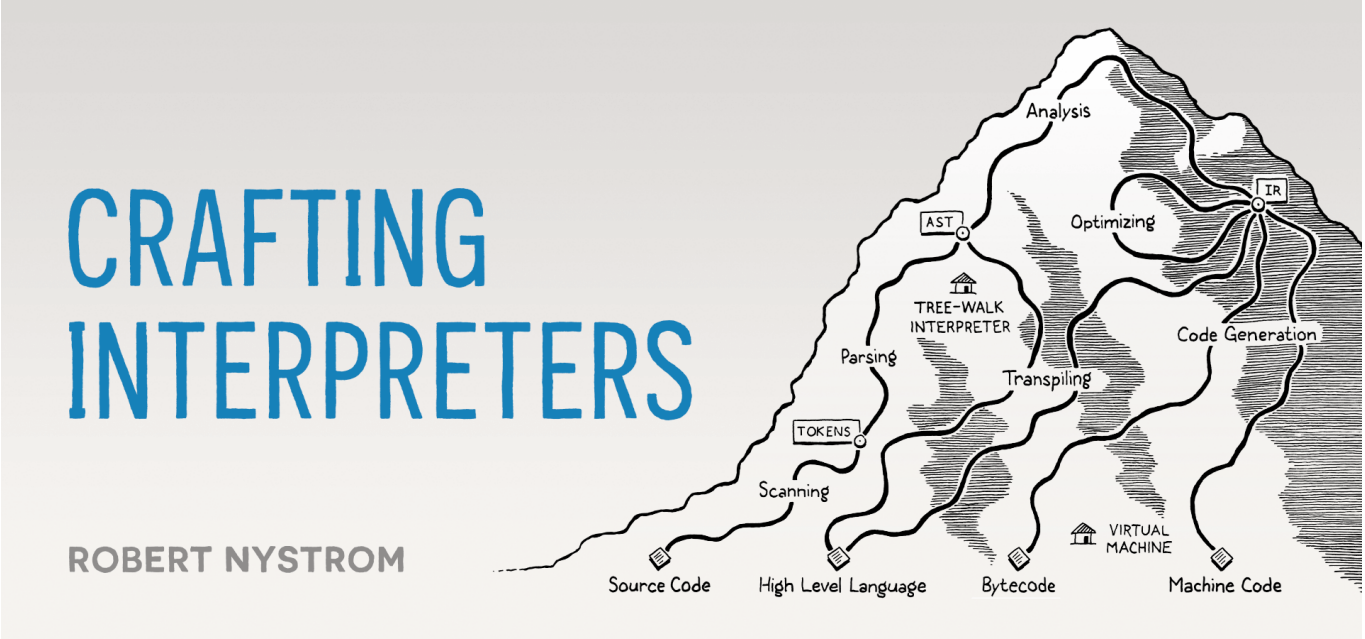
- Type system
- LLVM backend
- ...

```
md"""
## Future improvements

- Type system
- LLVM backend
- ...
"""
```

# Also recommend

Crafting Interpreters




```
md"""
## Also recommend

[Crafting Interpreters](https://craftinginterpreters.com/)

![](https://craftinginterpreters.com/image/header.png)
"""
```

# Jl2Py.jl & Taichi.jl

## Jl2Py.jl

### Why did I make this?


We have LeetCode.jl, but since Leetcode does not support Julia yet, the code cannot run on Leetcode.

```
def f(x: int, y: int, /) -> int:
    return x * y
```

```
Text(jl2py("""
function f(x::Int32, y::Int32)::Int32
    x * y
end
"""))
```

```
def fibonacci(n, /):
    return 1 if n <= 2 else fibonacci(n - 1) + fibonacci(n - 2)
```

```
Text(jl2py("""
function fibonacci(n)
    return n <= 2 ? 1 : fibonacci(n - 1) + fibonacci(n - 2)
end
"""))
```

# Let's try a real-world Leetcode problem

```
from typing import *


def haskey(container, key) -> bool:
    return key in container


def isnothing(x) -> bool:
    return x is None


def iszero(x) -> bool:
    return x == 0


def divrem(x, y) -> Tuple:
    return x // y, x % y


def sort_inplace(x: list, rev: bool = False):
    x.sort(reverse=rev)
    return x


def push_inplace(x: Union[list, set, dict], *y):
    if isinstance(x, list):
        for yi in y:
            x.append(yi)
    elif isinstance(x, set):
        for yi in y:
            x.add(yi)
    elif isinstance(x, dict):
        for yi in y:
```

```
Text(jl2py("""
function two_sum(nums::Vector{Int}, target::Int)::Union{Nothing,Tuple{Int,Int}}
    seen = Dict{Int,Int}()
    for (i, n) in enumerate(nums)
        m = target - n
        if haskey(seen, m)
            return seen[m], i
        else
            seen[n] = i
        end
    end
end
""", apply_polyfill=true))
```

# Refactor with `MLStyle.jl`

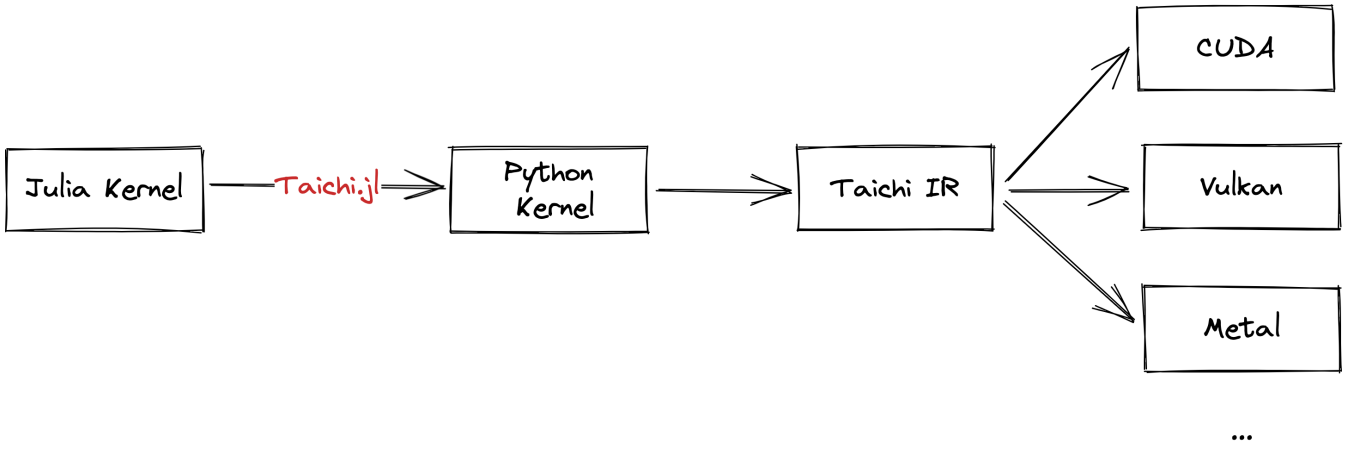
```
function __jl2py(jl_expr::Expr; topofblock::Bool=false, isflatten::Bool=false, iscom
prehension::Bool=false)
    @match jl_expr begin
        Expr(:block, args...) ||
        Expr(:toplevel, args...) => PyList([__jl2py(expr; topofblock=true)
                                            for expr in args if !isa(expr, LineNumbe
rNode)])
        Expr(:function, _...) => __parse_function(jl_expr)
        Expr(:(&&), _...) ||
        Expr(:(||), _...) => __boolop(jl_expr, OP_DICT[jl_expr.head])
        Expr(:tuple, args...) => AST.Tuple(__jl2py(jl_expr.args))
        Expr(:..., arg) => AST.Starred(__jl2py(arg))
        Expr(:., arg1, arg2) => AST.Attribute(__jl2py(arg1), __jl2py(arg2))
        Expr(:->, _...) => __parse_lambda(jl_expr)
        Expr(:if, _...) || Expr(:elseif, _...) => __parse_if(jl_expr)
        Expr(:while, test, body) => AST.While(__jl2py(test), __jl2py(body), nothing)
        Expr(:for, Expr(_, target, iter), body) => AST.fix_missing_locations(AST.For
(__jl2py(target), __jl2py(iter),

__jl2py(body), nothing, nothing))
        Expr(:continue) => AST.Continue()
        Expr(:break) => AST.Break()
        Expr(:return, arg) => AST.Return(__jl2py(arg))
        Expr(:ref, value, slices...) => __parse_ref(value, slices)
        Expr(:comprehension, arg) => AST.ListComp(__jl2py(arg; iscomprehension=tru
e)...)
        Expr(:generator, arg1, args...) => __parse_generator(arg1, args; isflatten,
iscomprehension)
        Expr(:filter, filter, args...) => __parse_filter(filter, args)
        Expr(:flatten, arg) => __jl2py(arg; isflatten=true, iscomprehension=iscompre
hension)
        Expr(:comparison, _...) => __compareop_from_comparison(jl_expr)
        Expr(:call, arg1, args...) => __parse_call(jl_expr, arg1, args; topofblock)
        Expr(:(=), args...) => __parse_assign(args)
        Expr(:vect, args...) => AST.List(__jl2py(args))
        Expr(op, target, value) => AST.fix_missing_locations(AST.AugAssign(__jl2py(t
arget),
                                                                           OP_DICT[o
p](),
                                                                           __jl2py(v
alue)))
        _ => begin
            @warn("Pattern unmatched")
            return AST.Constant(nothing)
        end
    end
end
```

# Taichi.jl

Taichi Lang

## Project Structure



Python None

```
let
    ti.init(; arch=ti.gpu)
    n = 640
    pixels = ti.Vector.field(3; dtype=pytype(1.0), shape=(n * 2, n))

    paint = @ti_kernel (t::Float64) -> for (i, j) in pixels
        c = ti.Vector([-0.8, ti.cos(t) * 0.2])
        z = ti.Vector([i / n - 1, j / n - 0.5]) * 2
        rgb = ti.Vector([0, 1, 1])
        iterations = 0
        while z.norm() < 20 && iterations < 50
            z = ti.Vector([z[0]^2 - z[1]^2, z[0] * z[1] * 2]) + c
            iterations += 1
            pixels[i, j] = (1 - iterations * 0.02) * rgb
        end
    end

    gui = ti.GUI("Julia Set"; res=(n * 2, n))
    i = 0
    flag = 0
    for _ in 1:200
        if flag == 0
            i -= 1
            if i * 0.02 <= 0.2
                flag = 1
            end
        else
            i += 1
            if i * 0.02 > (π * 1.2)
                flag = 0
            end
        end

        paint(i * 0.02)
        gui.set_image(pixels)
        gui.show()
    end

    gui.close()
end
```

# Under the hood

```
macro taichify(func, decorator)
    func_expr = :($func)
    py_func_name = "compiled_julia_func_$(COUNTER[])"
    if func_expr.head == :-> || (func_expr.args[1].head ∉ [:call, :(::)])
        func_expr.head = :function
        if func_expr.args[1].head == :tuple
            __assign_to_kw!(func_expr.args[1].args)
            func_expr.args[1] = Expr(:call, Symbol(py_func_name), func_expr.args[1].
args...)
        elseif func_expr.args[1].head != :(::) || isa(func_expr.args[1].args[1], Sym
bol)
            func_expr.args[1] = Expr(:call, Symbol(py_func_name), func_expr.args[1])
        else
            __assign_to_kw!(func_expr.args[1].args[1].args)
            func_expr.args[1].args[1] = Expr(:call, Symbol(py_func_name), func_expr.
args[1].args[1].args...)
        end
    end
    py_func = jl2py(func_expr)
    py_func.args.args, py_func.args.posonlyargs = py_func.args.posonlyargs, py_func.
args.args
    py_func.name = py_func_name
    tmp_file_name = "__tmp__$(COUNTER[]).py"
    COUNTER[] += 1

    quote
        py_str = "$($decorator)\n" * pyconvert(String, unparse($py_func)) * "\n"
        write($tmp_file_name, py_str)
        code = pycompile(py_str; filename=$tmp_file_name, mode="single")
        namespace = pydict(["ti" => ti, "np" => np, map(x -> string(x.first) => x.se
cond, collect(Base.@locals))...])
        pyexec(code, namespace)
        namespace.get($py_func_name)
    end
end
```

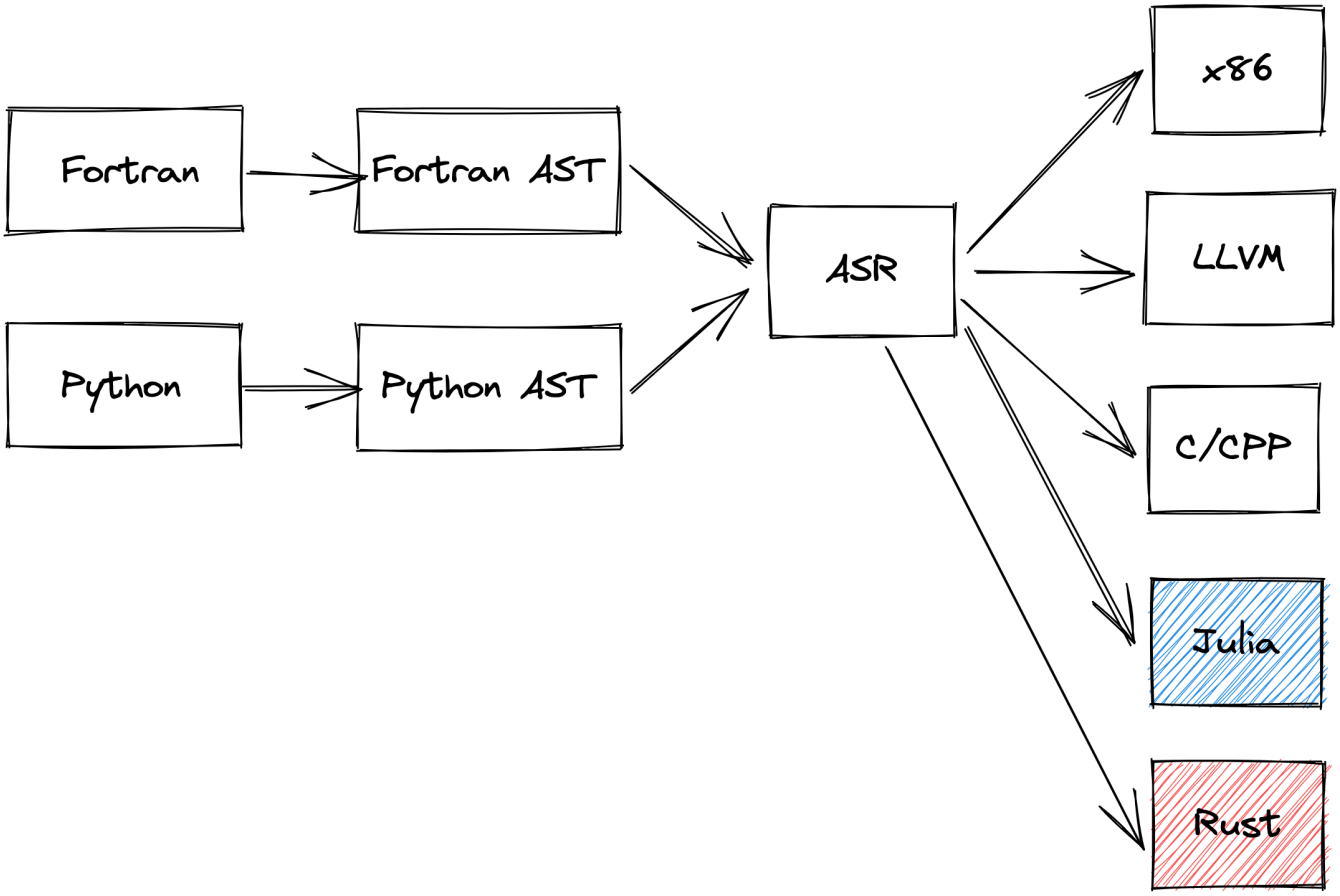
# Future improvements

- Transpile Julia to Taichi IR directly
- Taichi AOT
- ...

# Julia backend for LCompilers

- LFortran
- LPython

# Project Structure




Write ASR to Julia code string.

Tried `libjulia` at first but failed since `lfortran` depends on different version of LLVM.

# Example

Fortran source code:

```
program test_a
   integer :: bb(10, 10, 10) = 2
   print *, "hello"
   call d(bb)
   print *, bb(1, 1, 1)
contains

   subroutine d(arr)
      integer, intent(inout) :: arr(10, 10, 10)
      arr(1, 1, 1) = 2
   end subroutine

   subroutine a()
      integer :: i
      call b(i)
      print *, i
   end subroutine

   subroutine b(out)
      integer, intent(out) :: out
      call c(out)
      out = out * 2
   end subroutine

   subroutine c(in1)
      integer, intent(in) :: in1
      print *, in1
   end subroutine
end program
```

AST:

```
(TranslationUnit [(Program test_a (TriviaNode [] [(EndOfLine) (EndOfLine)]) [] []
[(Declaration (AttrType TypeInteger [] () None) [] [(bb [(1 10 DimensionExpr) (1 10
DimensionExpr) (1 10 DimensionExpr)] [] 2 Equal ())] ())] [(Print 0 () [(String "hel
lo")] ()) (SubroutineCall 0 d [] [(() bb ())] [] ()) (Print 0 () [(FuncCallOrArray b
b [] [(() 1 ()) (() 1 ()) (() 1 ())] [] [])] ())] [(Subroutine d [(arr)] [] () (Triv
iaNode [] [(EndOfLine) (EndOfLine)]) [] [] [] [(Declaration (AttrType TypeInteger []
() None) [(AttrIntent InOut)] [(arr [(1 10 DimensionExpr) (1 10 DimensionExpr) (1 10
DimensionExpr)] [] () None ())] ())] [(= 0 (FuncCallOrArray arr [] [(() 1 ()) (() 1
()) (() 1 ())] [] []) 2 ())] []) (Subroutine a [] [] () (TriviaNode [] [(EndOfLine)
(EndOfLine)]) [] [] [] [(Declaration (AttrType TypeInteger [] () None) [] [(i [] []
() None ())] ())] [(SubroutineCall 0 b [] [(() i ())] [] ()) (Print 0 () [i] ())]
[]) (Subroutine b [(out)] [] () (TriviaNode [] [(EndOfLine) (EndOfLine)]) [] [] []
[(Declaration (AttrType TypeInteger [] () None) [(AttrIntent Out)] [(out [] [] () No
ne ())] ())] [(SubroutineCall 0 c [] [(() out ())] [] ()) (= 0 out (* out 2) ())]
[]) (Subroutine c [(in1)] [] () () [] [] [] [(Declaration (AttrType TypeInteger []
() None) [(AttrIntent In)] [(in1 [] [] () None ())] ())] [(Print 0 () [in1] ())]
[])]))
```

ASR:

```
(TranslationUnit (SymbolTable 1 {test_a: (Program (SymbolTable 2 {a: (Function (Symb
olTable 4 {i: (Variable 4 i Local () () Default (Integer 4 []) Source Public Require
d .false.)}) a [b] [] [(SubroutineCall 2 b () [((Var 4 i))] ()) (Print () [(Var 4
i)] () ())] () Source Public Implementation () .false. .false. .false. .false. .fals
e. [] [] .false.), b: (Function (SymbolTable 5 {out: (Variable 5 out Out () () Defau
lt (Integer 4 []) Source Public Required .false.)}) b [c] [(Var 5 out)] [(Subroutine
Call 2 c () [((Var 5 out))] ()) (= (Var 5 out) (IntegerBinOp (Var 5 out) Mul (Intege
rConstant 2 (Integer 4 [])) (Integer 4 []) ()) ())] () Source Public Implementation
() .false. .false. .false. .false. .false. [] [] .false.), bb: (Variable 2 bb Local
(IntegerConstant 2 (Integer 4 [])) () Save (Integer 4 [((IntegerConstant 1 (Integer
4 [])) (IntegerConstant 10 (Integer 4 []))) ((IntegerConstant 1 (Integer 4 [])) (Int
egerConstant 10 (Integer 4 []))) ((IntegerConstant 1 (Integer 4 [])) (IntegerConstan
t 10 (Integer 4 [])))]) Source Public Required .false.), c: (Function (SymbolTable 6
{in1: (Variable 6 in1 In () () Default (Integer 4 []) Source Public Required .fals
e.)}) c [] [(Var 6 in1)] [(Print () [(Var 6 in1)] () ())] () Source Public Implement
ation () .false. .false. .false. .false. .false. [] [] .false.), d: (Function (Symbo
lTable 3 {arr: (Variable 3 arr InOut () () Default (Integer 4 [((IntegerConstant 1
(Integer 4 [])) (IntegerConstant 10 (Integer 4 []))) ((IntegerConstant 1 (Integer 4
[])) (IntegerConstant 10 (Integer 4 []))) ((IntegerConstant 1 (Integer 4 [])) (Integ
erConstant 10 (Integer 4 [])))]) Source Public Required .false.)}) d [] [(Var 3 ar
r)] [(= (ArrayItem (Var 3 arr) [(() (IntegerConstant 1 (Integer 4 [])) ()) (() (Inte
gerConstant 1 (Integer 4 [])) ()) (() (IntegerConstant 1 (Integer 4 [])) ())] (Integ
er 4 []) ColMajor ()) (IntegerConstant 2 (Integer 4 [])) ())] () Source Public Imple
mentation () .false. .false. .false. .false. .false. [] [] .false.)}) test_a [] [(Pr
int () [(StringConstant "hello" (Character 1 5 () []))] () ()) (SubroutineCall 2 d
() [((Var 2 bb))] ()) (Print () [(ArrayItem (Var 2 bb) [(() (IntegerConstant 1 (Inte
ger 4 [])) ()) (() (IntegerConstant 1 (Integer 4 [])) ()) (() (IntegerConstant 1 (In
teger 4 [])) ())] (Integer 4 []) ColMajor ())] () ())])}) [])
```

Julia:

```
function a()
    local i::Int32 = 0
    __i_ref__= Ref(i)
    b(__i_ref__)
    i = __i_ref__[]
    println(i)
end

function b(out::Base.RefValue{Int32})
    c(out[])
    out[] = out[] * 2
end

function c(in1::Int32)
    println(in1)
end

function d(arr::Array{Int32, 3})
    arr[1, 1, 1] = 2
end

function main()
    local bb::Array{Int32, 3} = fill(2, 10, 10, 10)
    println("hello")
    d(bb)
    println(bb[1, 1, 1])
end

main()
```

# Other projects this year

- StringAlgorithms.jl
  - Manacher, KMP, Suffix automaton, Palindromic automaton
- Hyper.jl
- TaichiMakie.jl
- STMMRunner.jl
- RandomParticles.jl
- PolarizedBRF.jl
- MieScattering.jl
- Contributions to Yggdrasil
  - MSTM, Packmol, VDT, XRootD, web3go, DDSCAT, libRadTran, re2c, LFortran, DynamO, GstPluginsBase, P11Kit, ...
- ...

# Thank you for watching!