

# Introduction to compilers and interpreters

# Logistics

- Course website: <https://bobzhang.github.io/courses/>
- Discussion forum: <https://taolun.moonbitlang.com/>
- Target audience:
  - People who are interested in language design and implementations
  - No PL theory pre-requisites
- Example code language: **MoonBit**
  - Homebrew
  - Compiles to WASM
  - Great runtime performance
  - Extremely fast compiler
  - Easy to install on major platforms including Windows

# We are hiring

- Shenzhen
- Hiring: Language Toolchain, Developer Tools, GC, Editor, IDE

# Introduction

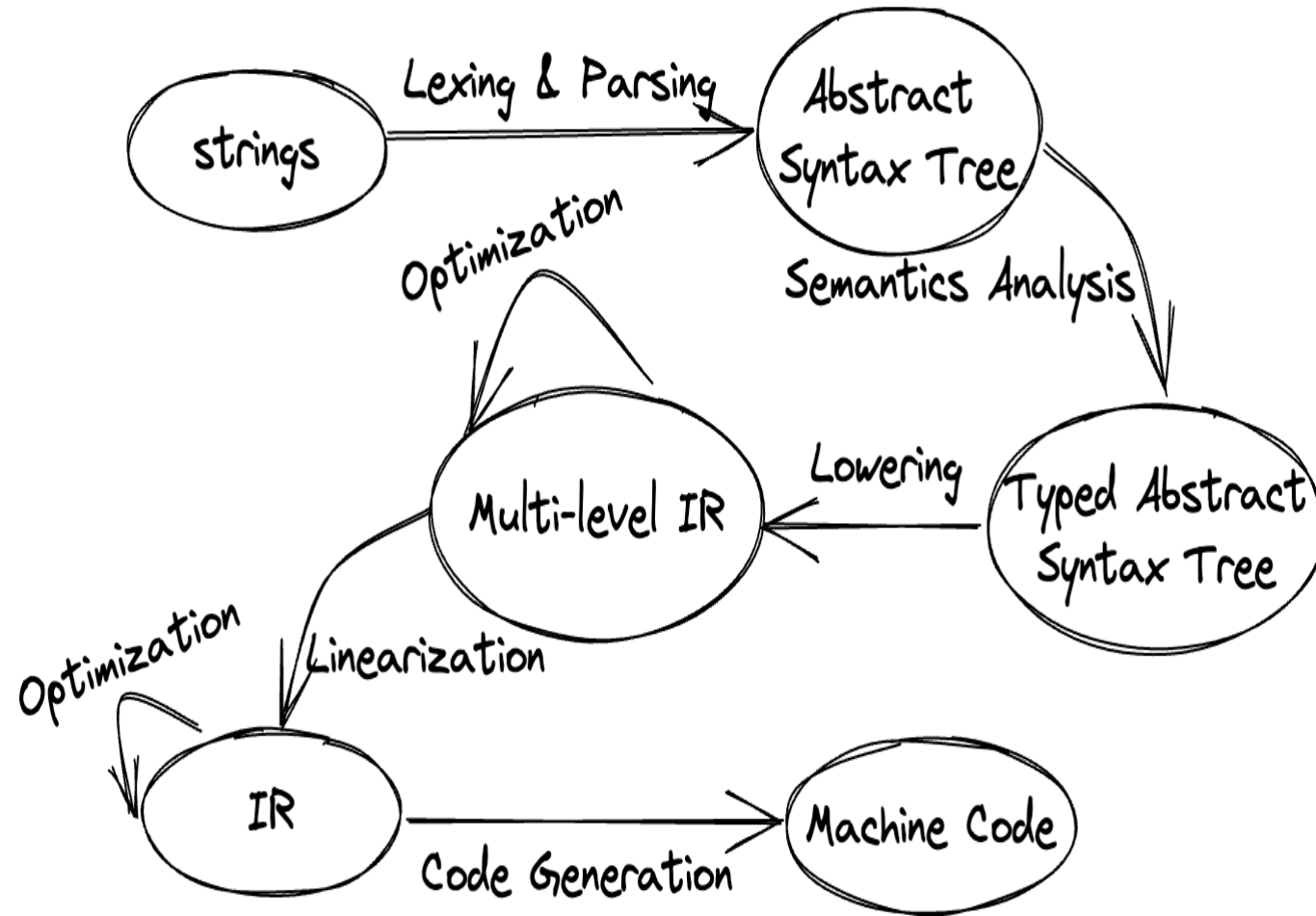
## Why study compiler&interpreters ?

- It is fun
- Understand your tools you use everyday
- Understand the cost of abstraction
  - Hidden allocation when declaring local functions
  - Why memory leak happens
- Make your own DSLs for profit
- Develop a good taste

# Course Overview

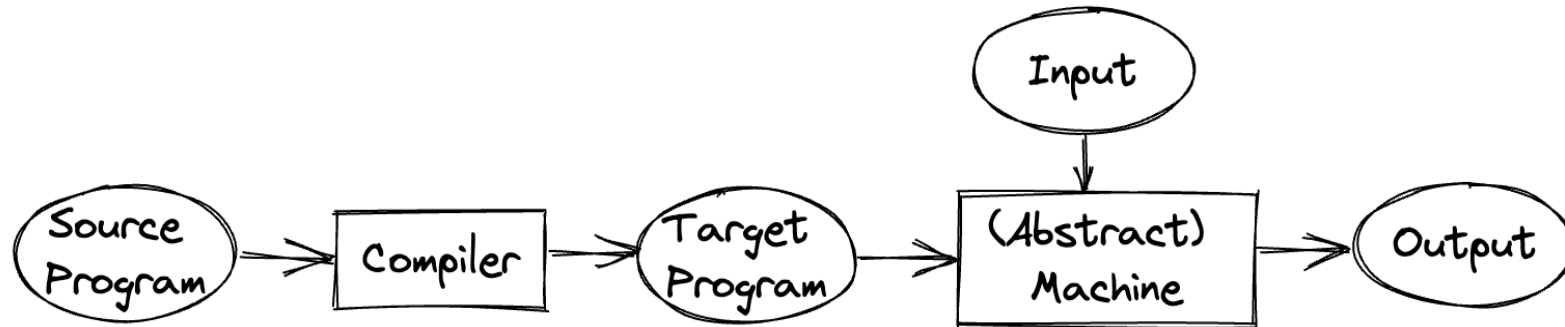
Lec	Topic	Lec	Topic
0	Introduction	6	Stack machine and compilation
1	ReScript crash course	7	WebAssembly
2	$\lambda$ Calculus	8	Garbage Collection and Memory Management
3	Names, Binders, De Bruijn index	9	Type checking
4	Closure Calculus	10	Type Inference and Unification
5	Pattern Matching	11&12	Formal Verification, Guest Lectures

# Compilation Phases



# Compilers, Interpreters

- Compilation and interpretation in two stages

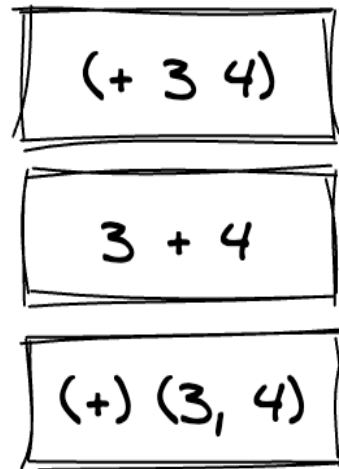


- The native compiler has a CPU interpreter
- Interpretation can be done in high level IRs (Python etc)

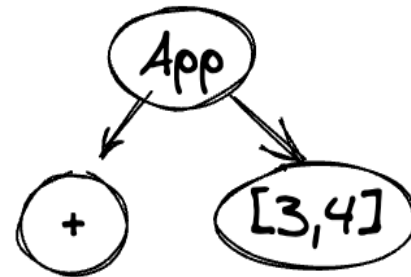
# Lexing & Parsing

- From strings to an abstract syntax tree
- Usually split into two phases: tokenization and parsing
- Lots of tool support, e.g.
  - Lex, Yacc, Bison, Menhir, Antlr, TreeSitter, parsing combinators, etc.

Concrete Syntax



Abstract Syntax





# Semantic Analysis

- Build the symbol table, resolve variables, modules
- Type checking & inference
  - Check that operations are given values of the right types
  - Infer types when annotation is missing
  - Typeclass/Implicits resolving
  - check other safety/security problems
    - Lifetime analysis
- Type soundness: no runtime type error when type checks

# Language specific lowering, optimizations

- Class/Module/objects/typeclass desugaring
- Pattern match desugaring
- Closure conversion
- Language specific optimizations
- IR relatively rich, MLIR, Direct style, ANF, CPS etc

# Linearization & optimizations

- Language & platform agnostics
- Optimizations
  - Constant folding, propagation, CSE, partial evaluation etc
  - Loop invariant code motion
  - Tail call eliminations
  - Intra-procedural, inter-procedural optimization
- IR simplified: three address code, LLVM IR etc

# Platform specific code generation

- Instruction selection
- Register allocation
- Instruction scheduling and machine-specific optimization
- Most influential in numeric computations, DSA

# Abstract Syntax vs. Concrete Syntax

- Modern language design: no semantic analysis during parsing
  - Counter example: C++ parsing is hard, error message is cryptic
- Many-to-one relation from concrete syntax to abstract syntax
- Start from abstract syntax for this course
  - Tutorials later for parsing in ReScript

- Tiny Language 0

## Concrete syntax

```
expr :  INT    // 1
      |  expr "+" expr // 1 + 2 , (1+2) + 3
      |  expr "*" expr // 1 * 2
      |  "(" expr ")"
```

## Abstract Syntax

```
enum Expr {
  Cst(Int)
  Add(Expr, Expr)
  Mul(Expr, Expr)
}
```

```
class Expr {...} class Cst extends Expr {...}
class Add extends Expr {...} class Mul extends Expr{...}
```

# Interpreter

```
enum Expr {  
  Cst(Int)           // i  
  Add(Expr, Expr)    // a + b  
  Mul(Expr, Expr)    // a * b  
}
```

```
fn eval(e : Expr) -> Int {  
  match e {  
    Cst(i) => i  
    Add(a, b) => eval(a) + eval(b)  
    Mul(a, b) => eval(a) * eval(b)  
  }  
}
```

# Formalization

## Semantics

The evaluation result is a value, which is an integer for our expression language

terms :  $e ::= \text{Cst}(i) \mid \text{Add}(e_1, e_2) \mid \text{Mul}(e_1, e_2)$

values :  $v ::= i \in \text{Int}$

The evaluation rules:

$$\frac{}{\text{Cst}(i) \Downarrow i} \text{E-const} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\text{Add}(e_1, e_2) \Downarrow (v_1 + v_2)} \text{E-add} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\text{Mul}(e_1, e_2) \Downarrow (v_1 * v_2)} \text{E-mul}$$



# Inference rules

- The evaluation relation  $e \Downarrow v$  means expression  $e$  evaluates to value  $v$ , for example
  - $\text{Cst}(42) \Downarrow 42$
  - $\text{Add}(\text{Cst}(3), \text{Cst}(4)) \Downarrow 7$
- Inference rules provide a concise way of specifying language properties, analyses, etc
  - If the **premises** are true, then the **conclusion** is true
  - An **axiom** is a rule with no premises
  - Inference rules can be **instantiated** by replacing **metavariables**  $(e, e_1, e_2, x, i, \dots)$  with expressions, program variables, integers

# Proof Tree

- Instantiated rules can be combined into proof trees
- $e \Downarrow v$  holds if and only if there is a finite proof tree constructed from correctly instantiated rules, and leaves of the tree are axioms

# What is the problem of our interpreter?

```
Add(a, b) => eval(a) + eval(b)
```

# Lowering to a stack machine and interpret

```
enum Instr {  
  Cst(Int)  
  Add  
  Mul  
} // non-recursive  
type Instrs @immut/list.T[Instr]  
type Operand Int  
type Stack @immut/list.T[Operand]
```

```
fn eval(instrs : Instrs, stk : Stack) -> Int {  
  match (instrs.0, stk.0) {  
    (Cons(Cst(i), rest), _) => eval(rest, Cons(i, stk.0))  
    (Cons(Add, rest), Cons(Operand(a), Cons(Operand(b), stk))) =>  
      eval(rest, Cons(a + b, stk))  
    (Cons(Mul, rest), Cons(Operand(a), Cons(Operand(b), stk))) =>  
      eval(rest, Cons(a * b, stk))  
    (Nil, Cons(Operand(a), _)) => a  
    _ => abort("Matched none")  
  }  
}
```

## Semantics

The machine has two components:

- a code pointer  $c$  giving the next instruction to execute
- a stack  $s$  holding intermediate results

Notation for stack: top of stack is on the left

$$\begin{array}{ll} s \rightarrow v :: s & (\text{push } v \text{ on } s) \\ v :: s \rightarrow s & (\text{pop } v \text{ off } s) \end{array}$$

# Transition of Stack Machine

Code and stack:

$$\begin{array}{ll} \text{code :} & c ::= \epsilon \mid i ; c \\ \text{stack :} & s ::= \epsilon \mid v :: s \end{array}$$

Transition of the machine:

$$\begin{array}{ll} (\text{Cst}(i); c, s) \rightarrow (c, i :: s) & (\text{I-Cst}) \\ (\text{Add}; c, n_2 :: n_1 :: s) \rightarrow (c, (n_1 + n_2) :: s) & (\text{I-Add}) \\ (\text{Mul}; c, n_2 :: n_1 :: s) \rightarrow (c, (n_1 \times n_2) :: s) & (\text{I-Mul}) \end{array}$$

The execution of a sequence of instructions terminates when the code pointer reaches the end and returns the value on the top of the stack

$$\frac{(c, \epsilon) \rightarrow^* (\epsilon, v :: \epsilon)}{c \downarrow v}$$

# Formalization

The compilation corresponds to the following mathematical formalization.

$$\begin{aligned}\llbracket \text{Cst}(i) \rrbracket &= \text{Cst}(i) \\ \llbracket \text{Add}(e_1, e_2) \rrbracket &= \llbracket e_1 \rrbracket ; \llbracket e_2 \rrbracket ; \text{Add} \\ \llbracket \text{Mul}(e_1, e_2) \rrbracket &= \llbracket e_1 \rrbracket ; \llbracket e_2 \rrbracket ; \text{Mul}\end{aligned}$$

- $\llbracket \cdot \cdot \cdot \rrbracket$  is a commonly used notation for compilation
- Invariant: stack balanced property
- Proof by induction (machine checked proof using Coq)

# Compilation

- The evaluation expr language implicitly uses the stack of the host language
- The stack machine manipulates the stack explicitly

## Correctness of Compilation

A correct implementation of the compiler preserves the semantics in the following sense

$$e \Downarrow v \iff \llbracket e \rrbracket \downarrow v$$



# Homework0

Implement the compilation algorithm in ReScript

# Tiny Language 1

## Abstract Syntax: add names

```
enum Expr {  
    ...  
    Var(String)  
    Let(String, Expr, Expr)  
}
```

# Interpreter

## Semantics with Environment

```
type Env @immut/list.T[(String, Int)]
fn eval(expr : Expr, env : Env) -> Int {
  match (expr, env) {
    (Cst(i), _) => i
    (Add(a, b), _) => eval(a, env) + eval(b, env)
    (Mul(a, b), _) => eval(a, env) * eval(b, env)
    (Var(x), Env(env)) => assoc(x, env).unwrap()
    (Let(x, e1, e2), Env(env)) => eval(e2, Cons((x, eval(e1, env)), env))
  }
}
```

# Formalization

terms :  $e ::= \text{Cst}(i) \mid \text{Add}(e_1, e_2) \mid \text{Mul}(e_1, e_2) \mid \text{Var}(i) \mid \text{Let}(x, e_1, e_2)$   
 envs :  $\Gamma ::= \epsilon \mid (x, v) :: \Gamma$

Notations for the environment:

variable access:  $\Gamma[x]$       variable update:  $\Gamma[x := v]$

The evaluation rules:

$$\begin{array}{c}
 \frac{}{\Gamma \vdash \text{Cst}(i) \Downarrow i} \text{E-const} \qquad \frac{\Gamma \vdash e_1 \Downarrow v_1 \quad \Gamma \vdash e_2 \Downarrow v_2}{\Gamma \vdash \text{Add}(e_1, e_2) \Downarrow (v_1 + v_2)} \text{E-add} \qquad \frac{\Gamma \vdash e_1 \Downarrow v_1 \quad \Gamma \vdash e_2 \Downarrow v_2}{\Gamma \vdash \text{Mul}(e_1, e_2) \Downarrow (v_1 * v_2)} \text{E-mul} \\
 \\
 \frac{\Gamma[x] = v}{\Gamma \vdash \text{Var}(x) \Downarrow v} \text{E-var} \qquad \frac{\Gamma \vdash e_1 \Downarrow v_1 \quad \Gamma[x := v_1] \vdash e_2 \Downarrow v}{\Gamma \vdash \text{Let}(x, e_1, e_2) \Downarrow v} \text{E-let}
 \end{array}$$

# What's the problem in our evaluator

- Where is the redundant work and can be resolved in compile time?
- The length of variable name affect our runtime performance!!

# Tiny Language 2

The position of a variable in the list is its binding depth (index)

```
enum ExprNameless {  
    ...  
    Var(Int)  
    Let(Expr, Expr)  
}
```

# Semantics

## Evaluation function

```
type Env @immut/list.T[Int]
fn eval(e : ExprNameless, env : Env) -> Int {
  match e {
    Cst(i) => i
    Add(a, b) => eval(a, env) + eval(b, env)
    Mul(a, b) => eval(a, env) * eval(b, env)
    Var(n) => env.0.nth(n).unwrap()
    Let(e1, e2) => eval(e2, Cons(eval(e1, env), env.0))
  }
}
```

# Semantics

Terms and values are the same.

Environments become sequence of values  $v_1 :: v_2 :: \dots :: \epsilon$ , accessed by position  $s[n]$

$$\text{envs} : \quad s ::= \epsilon \mid v :: s$$

Evaluation rules:

$$\begin{array}{c}
 \frac{}{s \vdash \mathbf{Cst}(i) \Downarrow i} \text{E-const} \qquad \frac{s \vdash e_1 \Downarrow v_1 \quad s \vdash e_2 \Downarrow v_2}{s \vdash \mathbf{Add}(e_1, e_2) \Downarrow (v_1 + v_2)} \text{E-add} \qquad \frac{s \vdash e_1 \Downarrow v_1 \quad s \vdash e_2 \Downarrow v_2}{s \vdash \mathbf{Mul}(e_1, e_2) \Downarrow (v_1 * v_2)} \text{E-mul} \\
 \\
 \frac{s[i] = v}{s \vdash \mathbf{Var}(i) \Downarrow v} \text{E-var} \qquad \frac{s \vdash e_1 \Downarrow v_1 \quad v_1 :: s \vdash e_2 \Downarrow v}{s \vdash \mathbf{Let}(x, e_1, e_2) \Downarrow v} \text{E-let}
 \end{array}$$



# Explanation

- The evaluation environment  $\Gamma$  for `expr` contains both names and values
- The evaluation environment  $s$  for `Nameless. expr` only contains the values, indexes resolved at compile time

## Lowering `expr` to `Nameless. expr`

```
type Cenv @immut/list.T[String]
fn comp(e : Expr, cenv : Cenv) -> ExprNameless {
  match e {
    Cst(i) => Cst(i)
    Add(a, b) => Add(comp(a, cenv), comp(b, cenv))
    Mul(a, b) => Mul(comp(a, cenv), comp(b, cenv))
    Var(x) => Var(index(cenv.0, x).unwrap())
    Let(x, e1, e2) => Let(comp(e1, cenv), comp(e2, Cons(x, cenv.0)))
  }
}
```

# Compile Nameless. expr

```
enum Instr {  
  ...  
  Var(Int)  
  Pop  
  Swap  
}
```

Semantics of the new instructions

$$(\text{Var}(i); c, s) \rightarrow (c, s[i] :: s) \quad (\text{I-Var})$$

$$(\text{Pop}; c, n :: s) \rightarrow (c, s) \quad (\text{I-Pop})$$

$$(\text{Swap}; c, n_1 :: n_2 :: s) \rightarrow (c, n_2 :: n_1 :: s) \quad (\text{I-Swap})$$

where  $s[i]$  reads the  $i$ -th value from the top of the stack

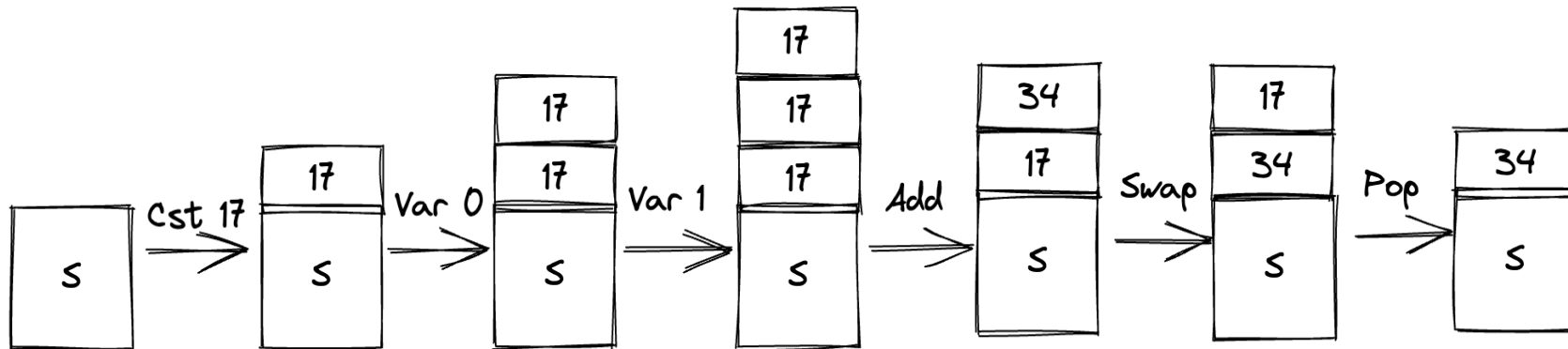
# Stack Machine with Variables

The program:  $\text{Let}(x, \text{CstI}(17), \text{Add}(\text{Var}(x), \text{Var}(x)))$

is compiled to instructions:

$[\text{Cst}(17); \text{Var}(0); \text{Var}(1); \text{Add}; \text{Swap}; \text{Pop}]$

The execution on the stack:



# More examples

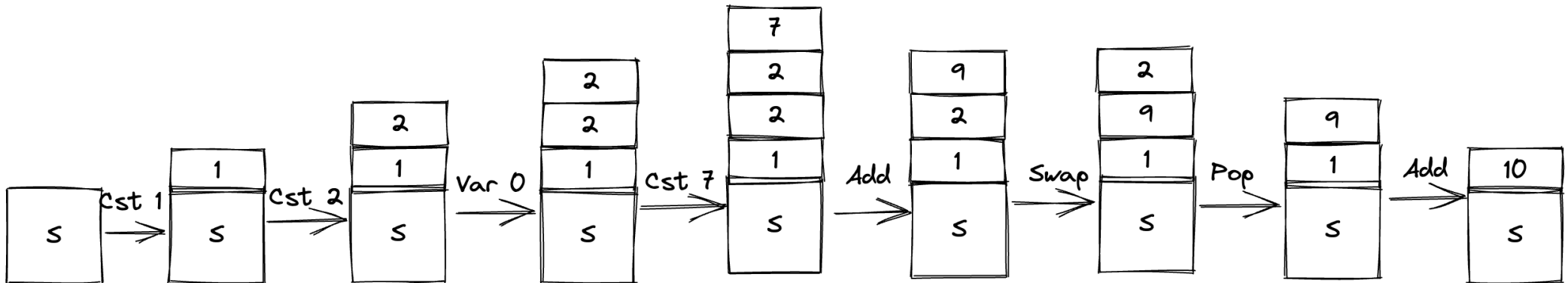
Consider the following program

```
let x = 2
1 + (x + 7)
```

is compiled to instructions

[Cst(1); Cst(2); Var(0); Cst(7); Add; Swap; Pop; Add]

The execution on the stack:

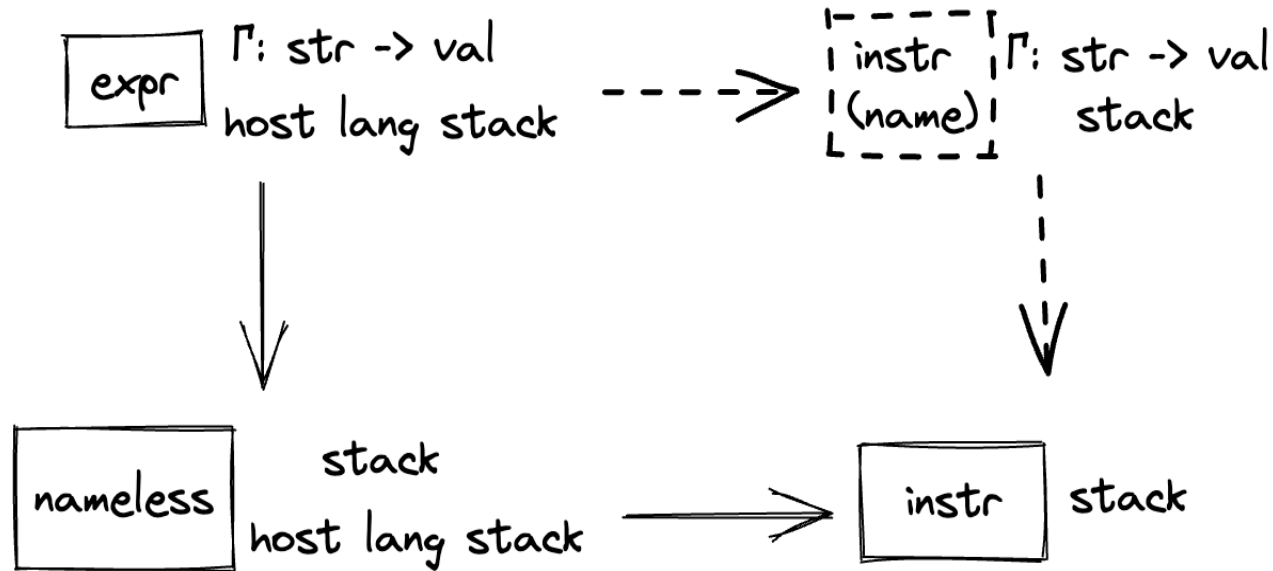


# Summary 1

What have we achieved through compilation? Compare the runtime environment

- Evaluating `expr`
  - a symbolic environment  $\Gamma$  for local variables
  - (implicit) stack of the host language for temporaries
- Evaluating `Nameless. expr`
  - a stack for local variables
  - (implicit) stack of the host language for temporaries
- For stack machine instructions, we have
  - a stack for both local variables and temporaries

# Summary 2



## Homework 1

- Write an interpreter for the stack machine with variables
- Write a compiler to translate `Nameless.expr` to stack machine instructions
- Implement the dashed part (one language + two compilers)