Introduction to compilers and interpreters

Logistics

- Course website: https://moonbitlang.github.io/minimoonbit-public/

- 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

MoonBit Programming Challenge

- Compiler optimization (programming language design)

- Game Development

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 to language design and implementation	5	Stack machine and compilation
1	MoonBit crash course	6	IR designs (ANF, CPS)
2	Parsing	7	Closure Calculus
3	Semantics analysis and type inferences	8	Register allocation & Garbage collection
4	Bidrectional type checking		

Compilation Phases
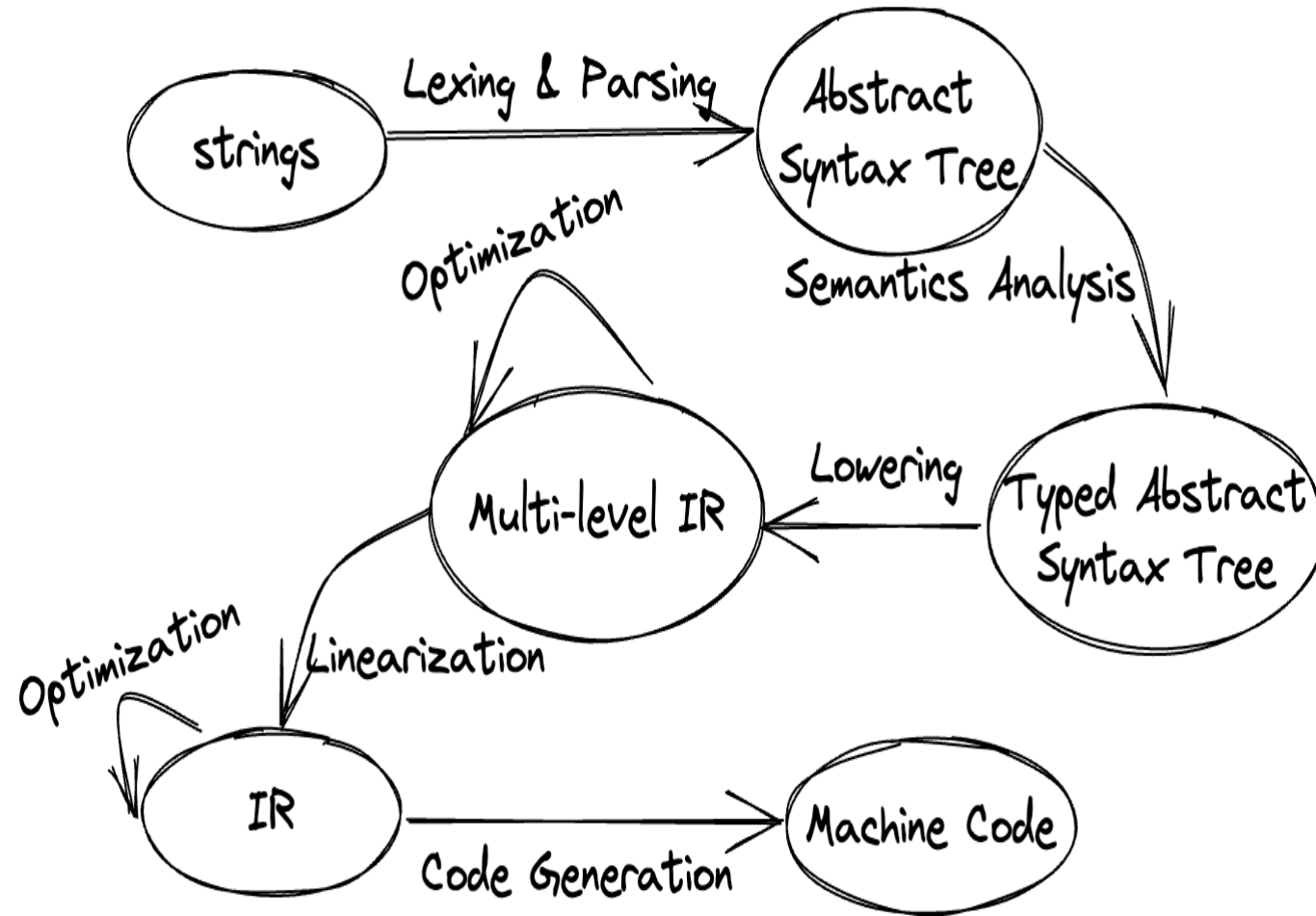
Compilers, Interpreters

- Compilation and interpretation in two stages
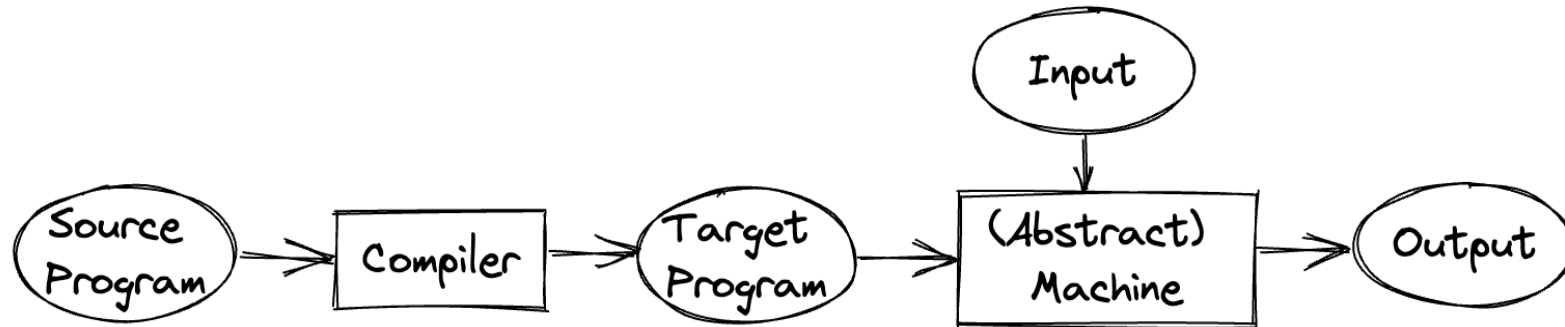
- The native compiler has a CPU interpreter

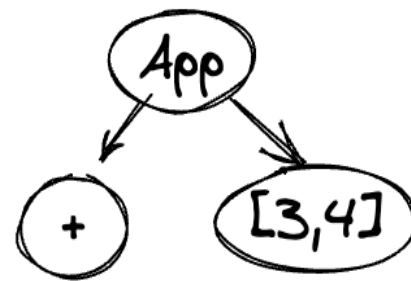- Interpretation can be done in high level IRs (Python etc)

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

(+ 3 4)

3 + 4

(+) (3, 4)

Abstract Syntax

App
+
[3,4]
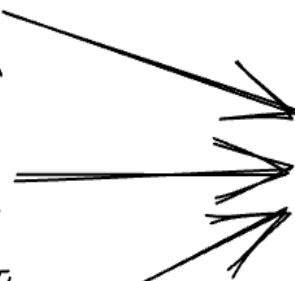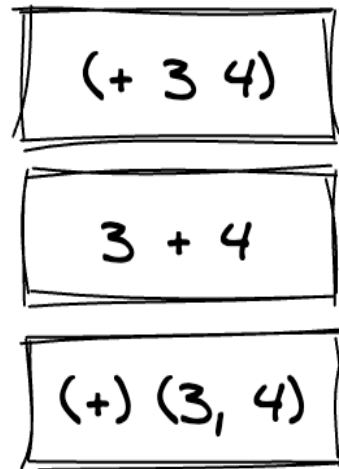
Semantic Analysis

- 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

- Opimizations

  - Constant folding, propogation, CSE, parital 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

- Instuction selection

- Register allocation

- Instruction scheduling and machine-specific optimization

- Most influential in numeric compuations, 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

$$\text{terms}: \quad e ::= \mathsf{Cst}(i) \mid \mathsf{Add}(e_1, e_2) \mid \mathsf{Mul}(e_1, e_2)$$
$$\text{values}: \quad v ::= i \in \mathsf{Int}$$

The evaluation rules:

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

Inference rules

- The evaluation relation $e \Downarrow v$ means expression e evaluates to value v, for example
  - $\mathsf{Cst}(42) \Downarrow 42$
  - $\mathsf{Add}(\mathsf{Cst}(3), \mathsf{Cst}(4)) \Downarrow 7$
- 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** $(e, e_1, e_2, x, i, \cdots)$ 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

$$s \to v :: s \qquad (\text{push } v \text{ on } s)$$
$$v :: s \to s \qquad (\text{pop } v \text{ off } s)$$

Transition of Stack Machine

Code and stack:

$$\text{code}: \qquad c ::= \epsilon \mid i \; ; \; c$$
$$\text{stack}: \qquad s ::= \epsilon \mid v :: s$$

Transition of the machine:

$$(\text{Cst}(i); c, s) \rightarrow (c, i :: s) \qquad\qquad (\text{I-Cst})$$
$$(\text{Add}; c, n_2 :: n_1 :: s) \rightarrow (c, (n_1 + n_2) :: s) \qquad (\text{I-Add})$$
$$(\text{Mul}; c, n_2 :: n_1 :: s) \rightarrow (c, (n_1 \times n_2) :: s) \qquad (\text{I-Mul})$$

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.

$$\llbracket \mathbf{Cst}(i) \rrbracket = \mathrm{Cst}(i)$$
$$\llbracket \mathbf{Add}(\mathsf{e}_1, \mathsf{e}_2) \rrbracket = \llbracket e_1 \rrbracket \,;\, \llbracket e_2 \rrbracket \,;\, \mathrm{Add}$$
$$\llbracket \mathbf{Mul}(\mathsf{e}_1, \mathsf{e}_2) \rrbracket = \llbracket e_1 \rrbracket \,;\, \llbracket e_2 \rrbracket \,;\, \mathrm{Mul}$$

- $\llbracket \cdots \rrbracket$ 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

$$e \Downarrow v \iff [\![e]\!] \downarrow v$$

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 : $e ::= \mathsf{Cst}(i) \mid \mathsf{Add}(e_1, e_2) \mid \mathsf{Mul}(e_1, e_2) \mid \mathsf{Var}(i) \mid \mathsf{Let}(x, e_1, e_2)$

envs : $\Gamma ::= \epsilon \mid (x, v) :: \Gamma$

Notations for the environment:

$$\text{variable access: } \Gamma[x] \qquad \text{variable update: } \Gamma[x := v]$$

The evaluation rules:

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

$$\frac{\Gamma[x] = v}{\Gamma \vdash \mathsf{Var}(x) \Downarrow v}\text{E-var} \qquad \frac{\Gamma \vdash e_1 \Downarrow v_1 \qquad \Gamma[x := v_1] \vdash e_2 \Downarrow v}{\Gamma \vdash \mathsf{Let}(x, e_1, e_2) \Downarrow v}\text{E-let}$$

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 $v_1 :: v_2 :: \cdots :: \epsilon$, accessed by position $s[n]$

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

Evaluation rules:

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

$$\frac{s[i] = v}{s \vdash \mathsf{Var}(i) \Downarrow v}\text{E-var} \qquad \frac{s \vdash e_1 \Downarrow v_1 \qquad v_1 :: s \vdash e_2 \Downarrow v}{s \vdash \mathsf{Let}(x, e_1, e_2) \Downarrow v}\text{E-let}$$

Explanation

- The evaluation environment Γ for **expr** contains both names and values
- The evaluation environment s for $\mathrm{Nameless.}$ **expr** only contains the values, indexes resolved at compile time

Lowering expr to $\mathrm{Nameless.}$ **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

$$(\text{Var}(i); c, s) \rightarrow (c, s[i] :: s) \qquad \text{(I-Var)}$$
$$(\text{Pop}; c, n :: s) \rightarrow (c, s) \qquad \text{(I-Pop)}$$
$$(\text{Swap}; c, n_1 :: n_2 :: s) \rightarrow (c, n_2 :: n_1 :: s) \qquad \text{(I-Swap)}$$

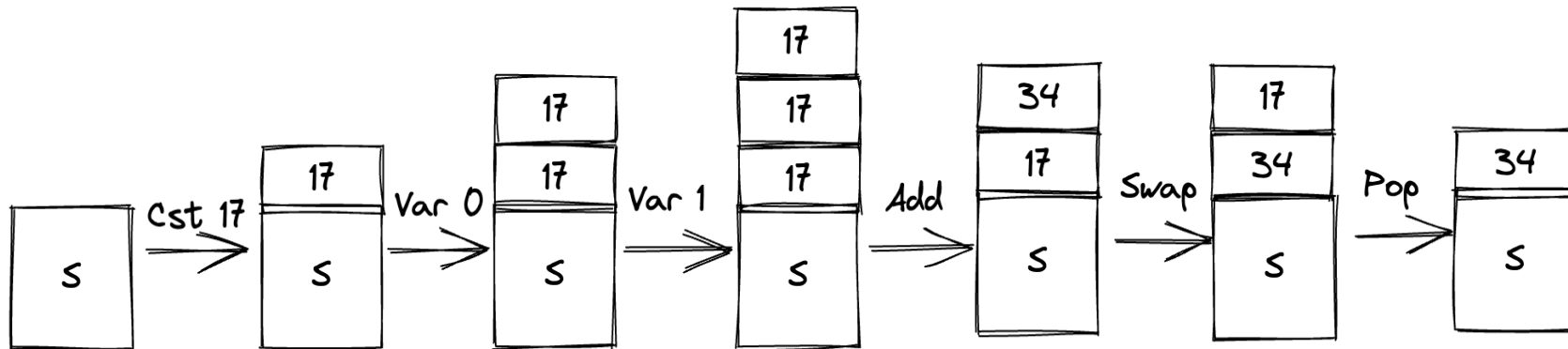where $s[i]$ reads the i-th value from the top of the stack

Stack Machine with Variables

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

is compiled to instructions:

$$[\text{Cst}(17); \text{Var}(0); \textcolor{red}{\text{Var}(1)}; \text{Add}; \text{Swap}; \text{Pop}]$$
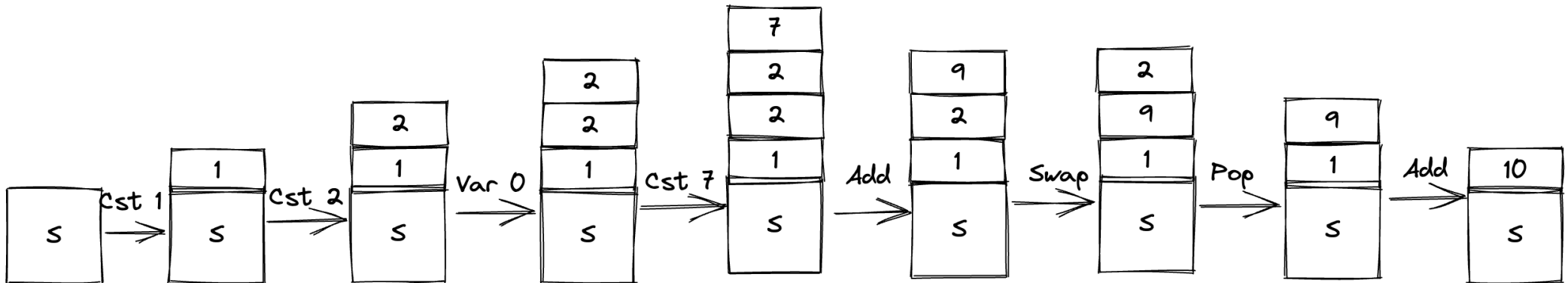
The execution on the stack:

Consider the following program

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

is compiled to instructions

$$[\text{Cst}(1); \text{Cst}(2); \text{Var}(0); \text{Cst}(7); \text{Add}; \text{Swap}; \text{Pop}; \text{Add}]$$

The execution on the stack:

Summary 1

What have we achieved through compilation? Compare the runtime environment

- Evaluating expr
  - a symbolic environment Γ for local variables
  - (implicit) stack of the host language for temperaries
- Evaluating $\mathrm{Nameless.\,expr}$
  - a stack for local variables
  - (implicit) stack of the host language for temperaries
- For stack machine instructions, we have
  - a stack for both local variables and temperaries

Summary 2
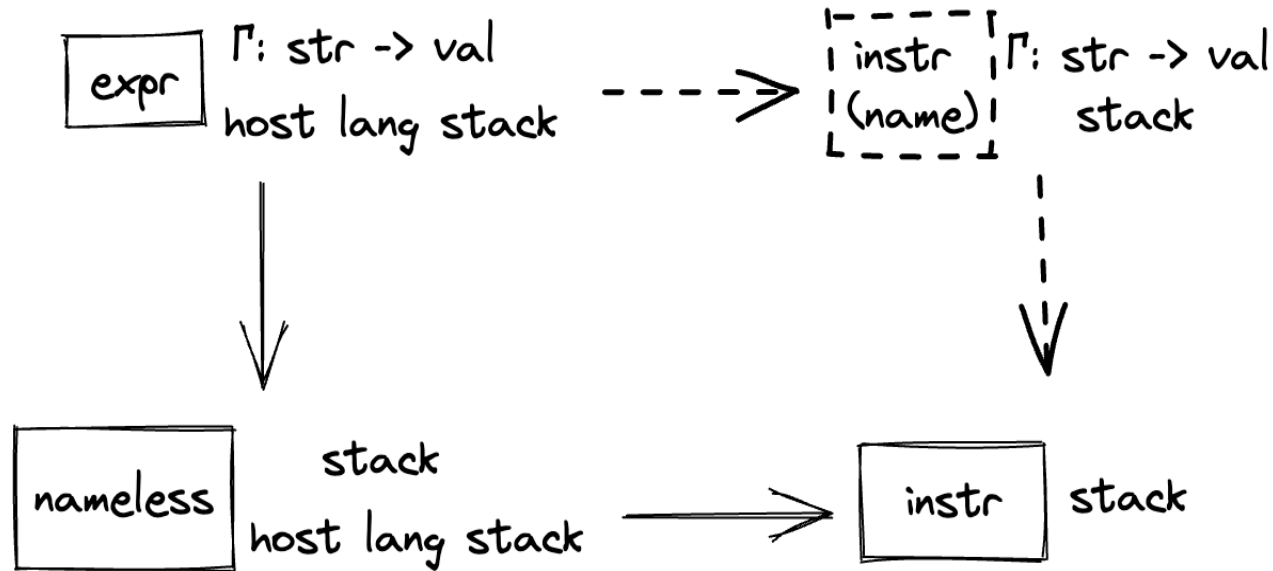
Homework 1

- 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)