idea
基础软件中心

Introduction to compilers and interpreters

Hongbo Zhang

idea
基础软件中心

Logistics

- Course website: https://bobzhang.github.io/courses/
- Discussion forum: https://bbs.csdn.net/forums/raelidea
- Target audience:
  - People who are interested in language design and implementations
  - No PL theory pre-requisites
- Example code language: ReScript
  - Homebrew
  - ReScript is a dialect of ML: Why ML are good for writing compilers
  - Easy to install on major platforms including Windows

Introduction

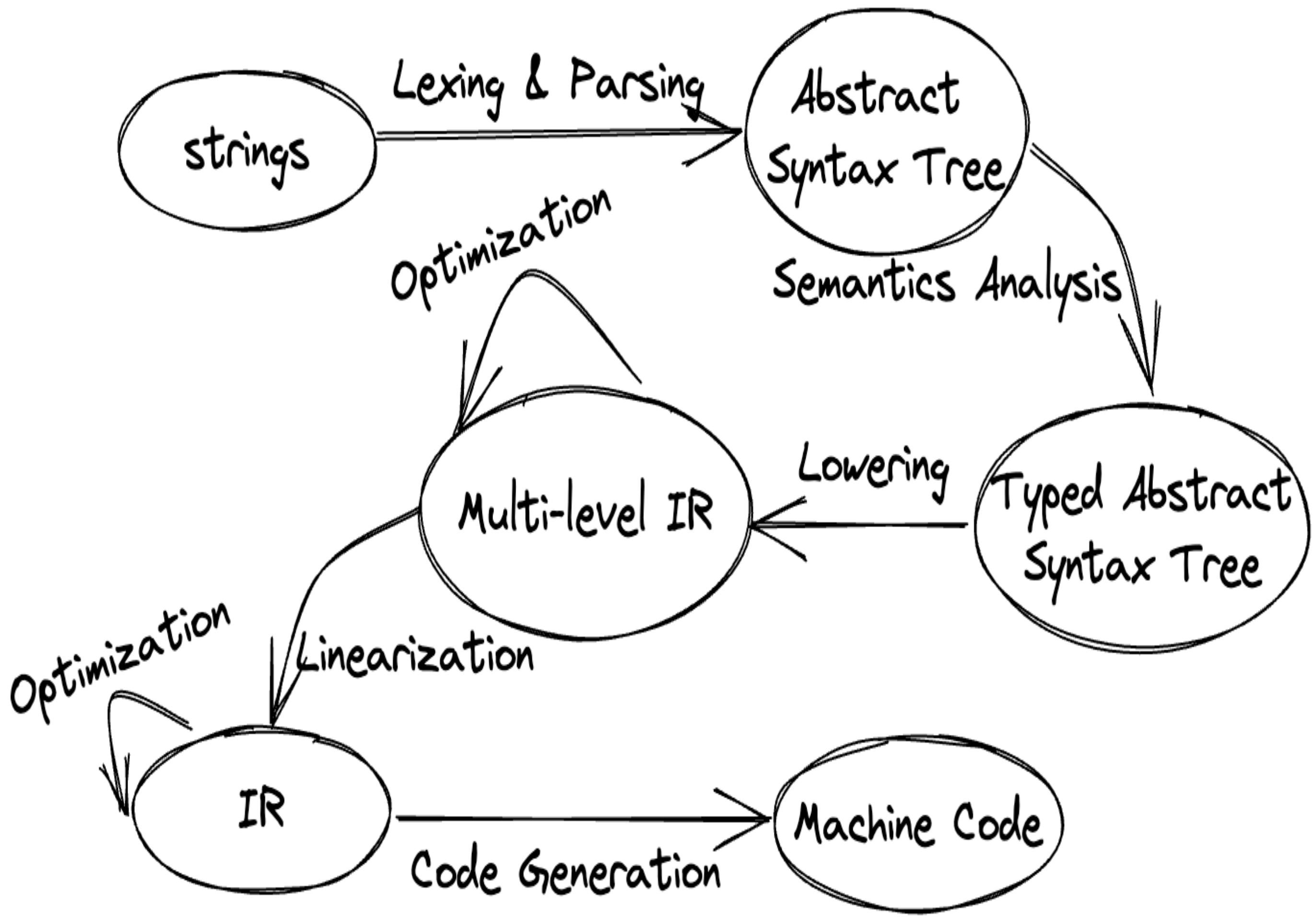
Why study compiler&interpreters ?

- 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

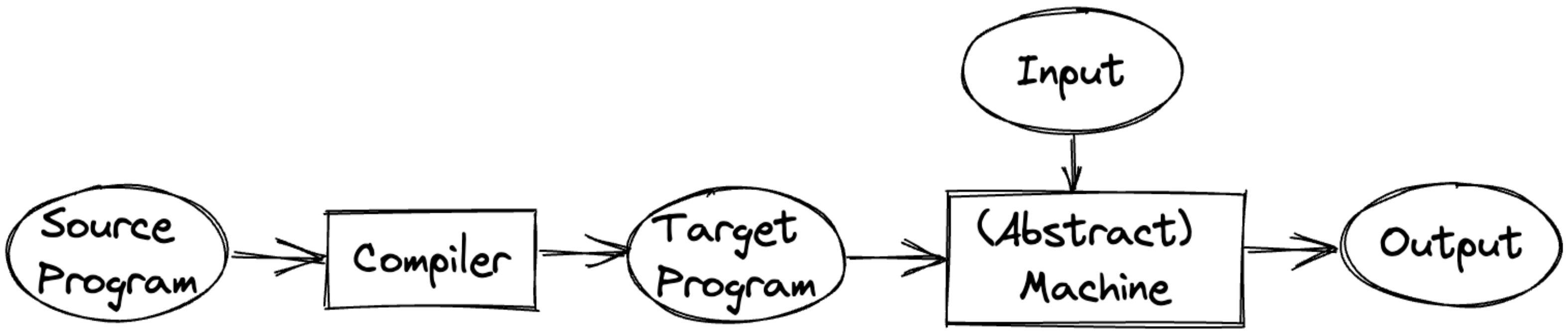
Lec	Topic	Lec	Topic
0	Introduction	6	Stack machine and compilation
1	ReScript crash course	7	WebAssembly
2	λ Calculus	8	Garbage Collection and Memory Management
3	Names, Binders, De Bruijn index	9	Type checking
4	Closure Calculus	10	Type Inference and Unification
5	Pattern Matching	11&12	Formal Verification,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

Compilation Phases



Compilers, Interpreters

- Compilation and interpretation in two stages

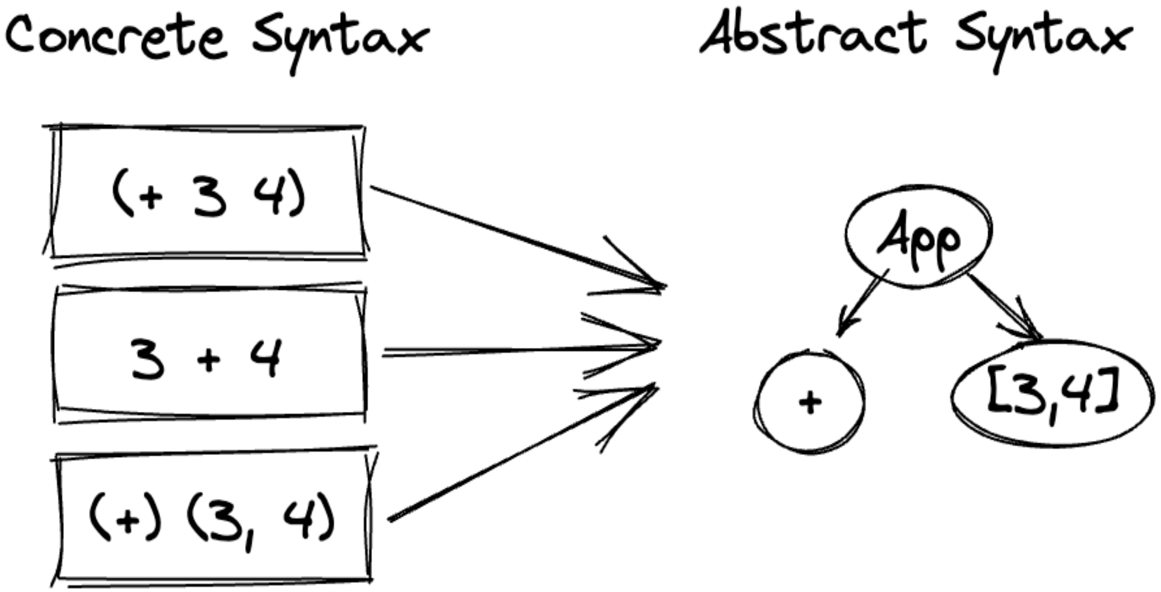


- The native compiler has a CPU interpreter
- Interpretation can be done in high level IRs (Python etc)

Lexing & Parsing

idea 基础软件中心

- 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.



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

```
type rec expr =
  | Cst (int) // i
  | Add (expr,expr) // a + b
  | Mul (expr,expr) // a * b
```

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

idea
基础软件中心

Interpreter

```
type rec expr =
  | Cst (int) // i
  | Add (expr,expr) // a + b
  | Mul (expr,expr) // a * b
```

```
let rec eval = (expr : expr) => {
  switch expr {
  | 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

$$\begin{aligned} \textsf{terms :} \quad & e ::= \textsf{Cst}(i) \mid \textsf{Add}(e_1, e_2) \mid \textsf{Mul}(e_1, e_2) \\ \textsf{values :} \quad & v ::= i \in \textsf{Int} \end{aligned}$$

The evaluation rules:

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

idea 基础软件中心

Inference rules

- The evaluation relation $e \Downarrow v$ means expression e evaluates to value v, for example
  - $\textsf{Cst}(42) \Downarrow 42$
  - $\textsf{Add}(\textsf{Cst}(3), \textsf{Cst}(4)) \Downarrow 7$
- 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

基础软件中心

What is the problem of our interpreter?

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

Lowering to a stack machine and interpret

idea
基础软件中心

```
type instr =  Cst (int) | Add | Mul // no recursive
type instrs = list <instr>
type operand = int
type stack = list <operand>
```

```
let rec eval = (instrs : instrs ,stk : stack) => {
  switch (instrs,stk) {
  | (list{ Cst (i), ... rest},_) =>
        eval(rest, list{i,...stk})
  | (list{Add, ... rest}, list{a,b,...stk}) =>
        eval(rest, list{a+b, ...stk})
  | (list{Mul, ... rest}, list{a,b,...stk}) =>
        eval(rest, list{a*b, ...stk})
  | (list{}, list{a,..._stk}) => a
  | _ => assert false
  }
}
```

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

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

Transition of Stack Machine

idea 基础软件中心

Code and stack:

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

Transition of the machine:

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

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.

$$
\begin{aligned}
[\![\mathsf{Cst}(i)]\!] &= \mathrm{Cst}(i) \\
[\![\mathsf{Add}(\mathsf{e}_1, \mathsf{e}_2)]\!] &= [\![e_1]\!] \ ; [\![e_2]\!] \ ; \mathrm{Add} \\
[\![\mathsf{Mul}(\mathsf{e}_1, \mathsf{e}_2)]\!] &= [\![e_1]\!] \ ; [\![e_2]\!] \ ; \mathrm{Mul}
\end{aligned}
$$

- $[\![\cdots]\!]$ 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

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

Homework0

Implement the compilation algorithm in ReScript

Tiny Language 1

Abstract Syntax: add names

```
type rec expr =
  ...
  | Var (string)
  | Let (string , expr , expr)
```

idea
基础软件中心

Interpreter

Semantics with Environment

```
type env = list<(string, int)>

let rec eval = (expr, env) => {
  switch expr {
  | Cst (i) => i
  | Add(a,b) => eval (a, env) + eval (b, env)
  | Mul(a,b) => eval (a, env) * eval (b, env)
  | Var(x) => assoc (x, env)
  | Let(x,e1,e2) => eval(e2, list{(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)

```
module Nameless = {
  type rec expr =
    ...
    | Var (int)
    | Let (expr, expr)
}
```

idea
基础软件中心

Semantics

Evaluation function

```
type env = list<int>

let rec eval = (expr : Nameless.expr, env) => {
  switch expr {
    | Cst(i) => i
    | Add(a,b) => eval (a, env) + eval (b, env)
    | Mul(a,b) => eval (a, env) * eval (b, env)
    | Var(n) => List.nth (env, n)
    | Let(e1,e2) => eval(e2, list{eval(e1,env), ...env})
  }
}
```

Semantics

Terms and values are the same.

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

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

Evaluation rules:

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

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

idea
基础软件中心

Explanation

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

idea
基础软件中心

Lowering expr to Nameless. expr

```
type cenv = list<string>

let rec comp = (expr : expr , cenv : cenv): Nameless.expr => {
    switch expr {
        | 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, x))
        | Let(x,e1,e2) => Let(comp(e1, cenv), comp(e2, list{x,...cenv}))
    }
}
```

Compile Nameless. expr

```
type instr =  ... | Var (int) | Pop | Swap
```

Semantics of the new instructions

$$
\begin{aligned}
(\mathrm{Var}(i); c, s) &\to (c, s[i] :: s) && \text{(I-Var)} \\
(\mathrm{Pop}; c, n :: s) &\to (c, s) && \text{(I-Pop)} \\
(\mathrm{Swap}; c, n_1 :: n_2 :: s) &\to (c, n_2 :: n_1 :: s) && \text{(I-Swap)}
\end{aligned}
$$

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

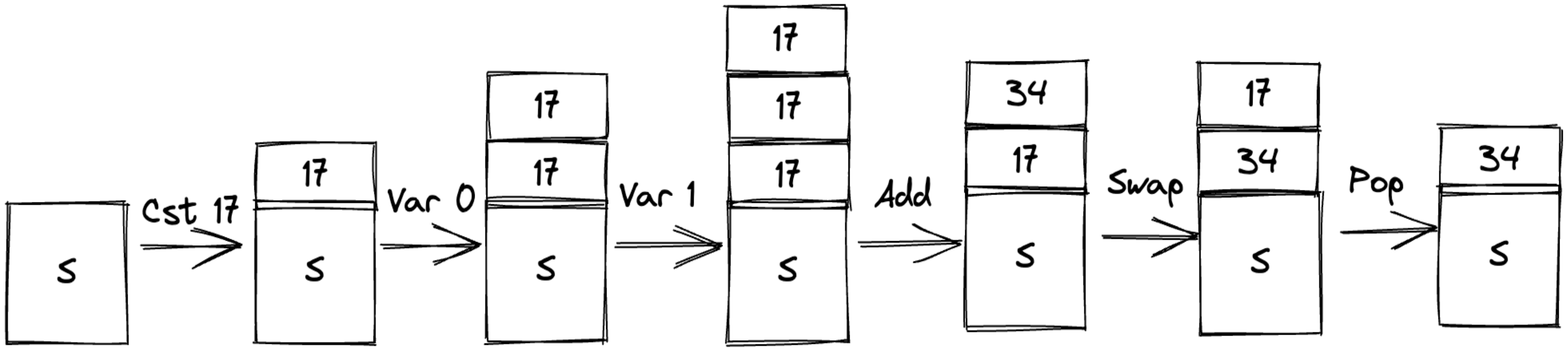
Stack Machine with Variables

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

is compiled to instructions:

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

The execution on the stack:



More examples

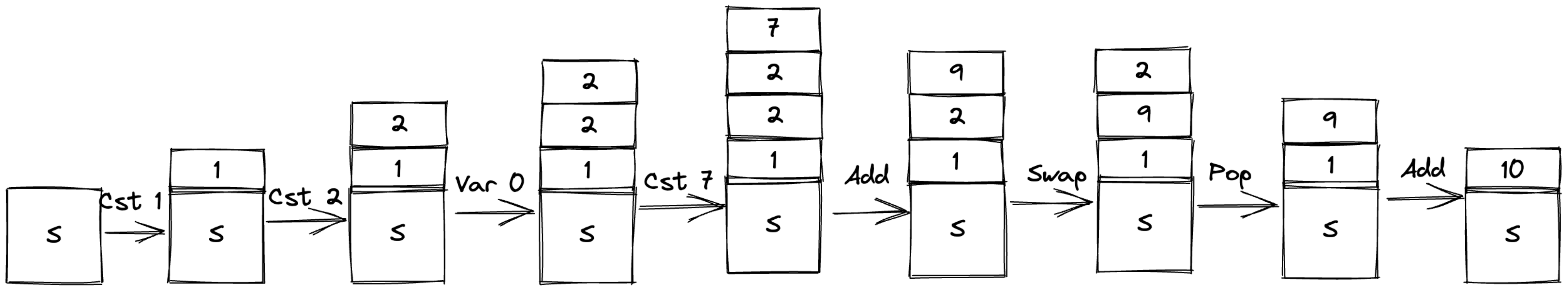
Consider the following program

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

is compiled to instructions

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

The execution on the stack:

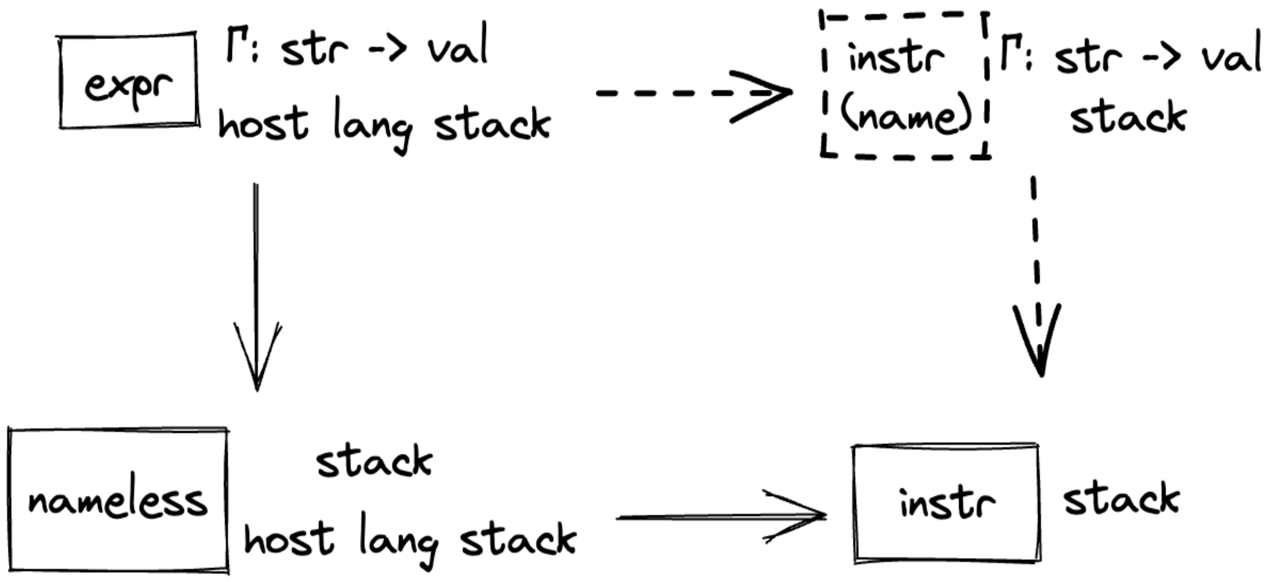


Summary 1

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 $\mathsf{Nameless}$. 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)