基础软件理论与实践公开课 (MoonBit挑战赛辅助教材)

张宏波

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
  - 基于ReScript理论与实践改编，重用了部分内容，新增了一部分高阶内容
- Example code: MoonBit
  - Compiles to WASM/JS
  - Great runtime performance
  - Good IDE support and fit for compiler construction
  - No installation required

基础软件中心

MoonBit Programming Challenge

- Language design and implementation (mini-moonbit in MoonBit)
- Game Development (Wasm4)

Why study compiler&interpreters ?

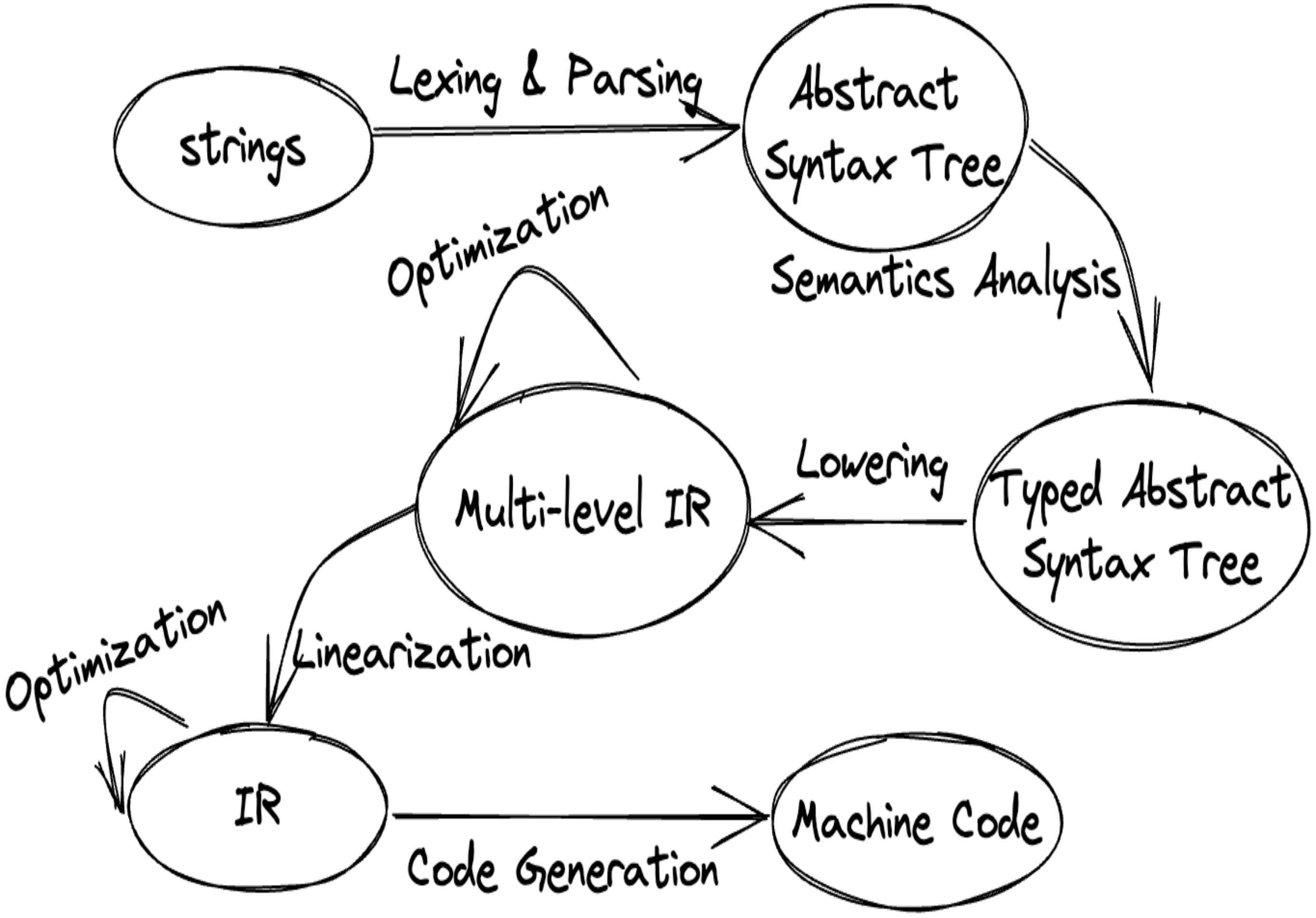
- It is fun
- Understand your tools you use everyday
- Understand the cost of abstraction
  - Hidden allocation when declaring local functions
  - Why memory leak happens
  - Good system programmers need write a toy C compiler
- Make your own DSLs for profit
- Develop a good taste

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	Bidirectional type checking		

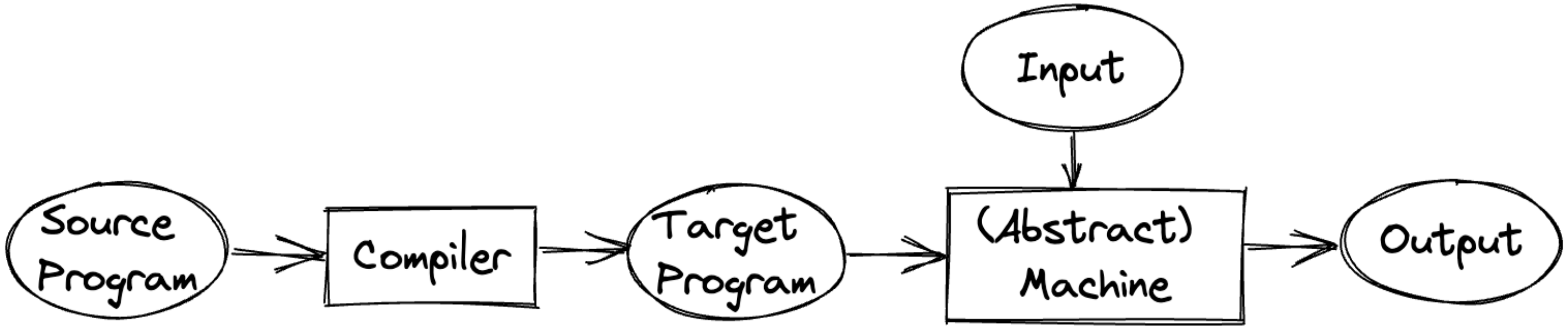
Compilation Phases

idea
基础软件中心



Compilers, Interpreters

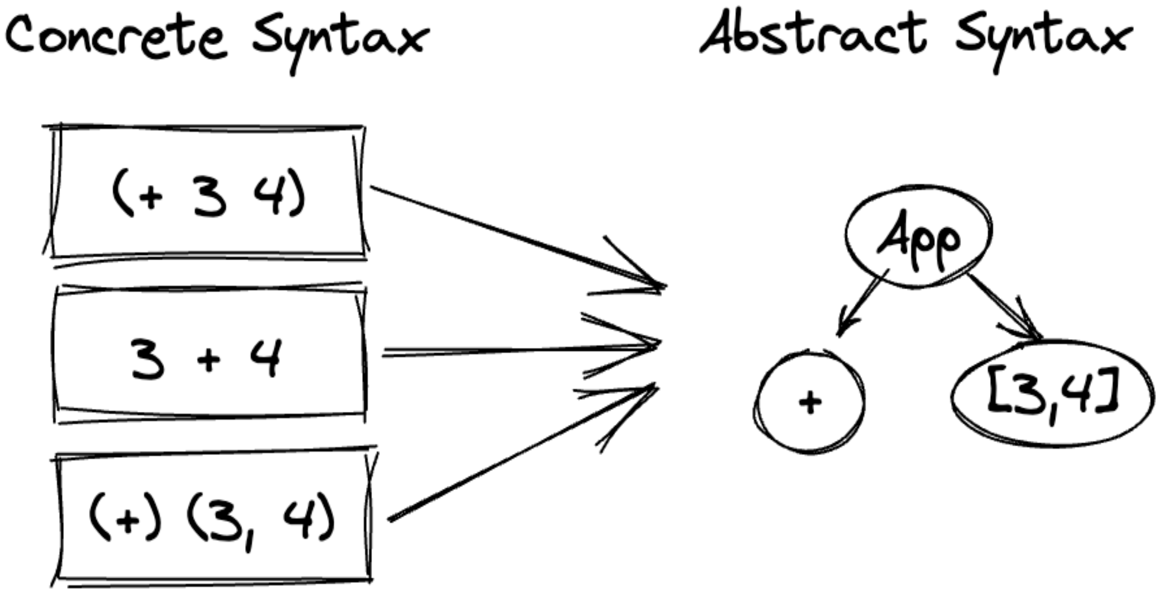
- Compilation and interpretation in two stages



- 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.



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
- Reuse code with IDE tooling

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

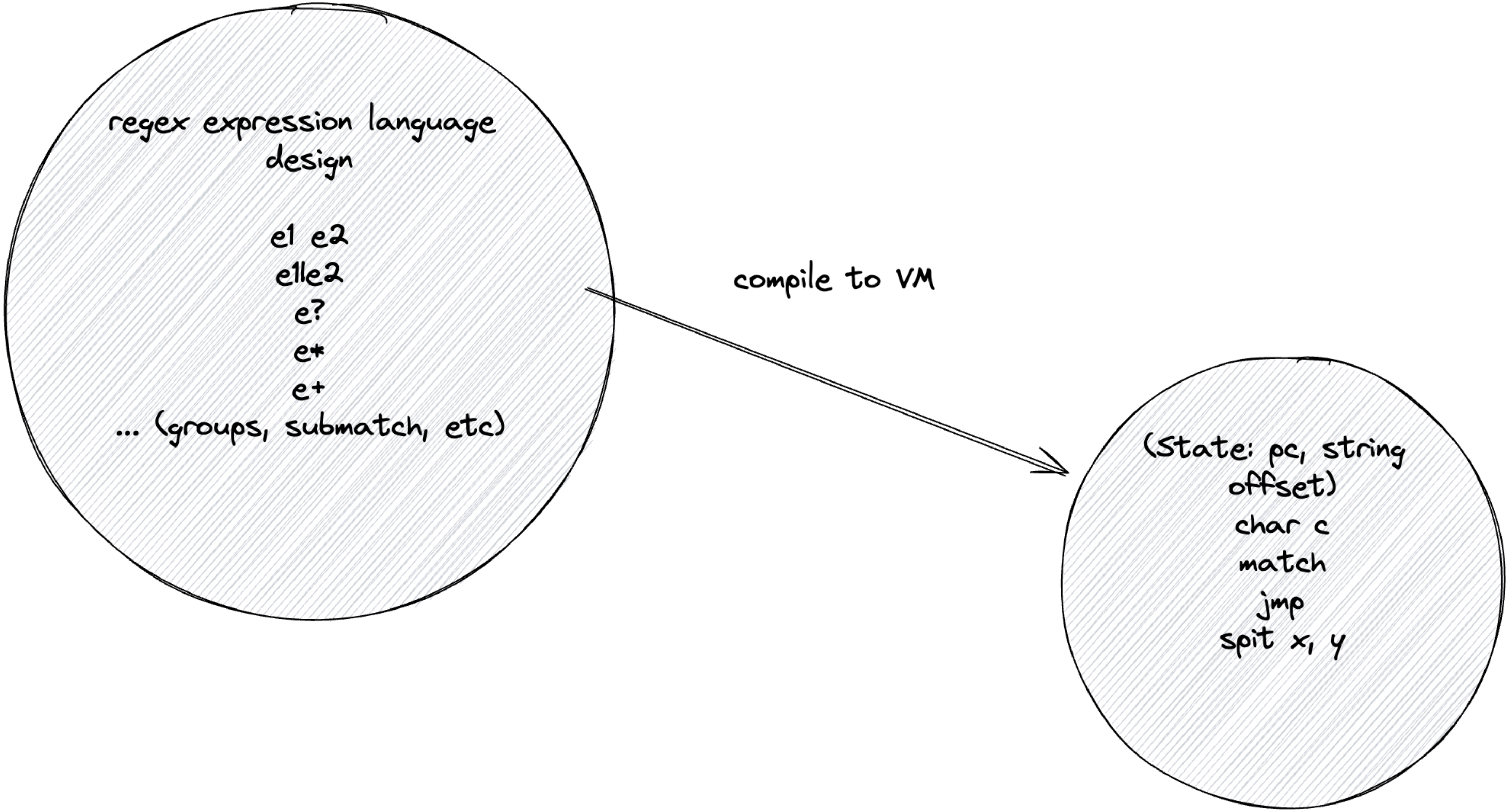
- Language & platform agnostics
- Opimizations
  - Constant folding, propogation, CSE, parital evaluation etc
  - Loop invariant code motion
  - Tail call eliminations
  - Intra-procedural, inter-procedural optimization
- IR simplified: three address code, LLVM IR etc

基础软件中心

Platform specific code generation

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

The smallest practical example: regular language



Regular language compiler

idea
基础软件中心

a		`char a`
e_1e_2		*codes for* e_1
		codes for e_2
$e_1 \mid e_2$		`split L1, L2`
	`L1:`	*codes for* e_1
		`jmp L3`
	`L2:`	*codes for* e_2
	`L3:`	
$e?$		`split L1, L2`
	`L1:`	*codes for* e
	`L2:`	
$e*$	`L1:`	`split L2, L3`
	`L2:`	*codes for* e
		`jmp L1`
	`L3:`	
$e+$	`L1:`	*codes for* e
		`split L1, L3`
	`L3:`	

idea
基础软件中心

Regular language VM

- Interpreter (backtracking)
- Optimized interpreter (backtracking with memoization)
- Linearized interpretation
- Compiler (CPU interpreted)

基础软件中心

Homework: finish regex compiler and regex VM

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

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

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

idea 基础软件中心

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

```
fn loop_eval(instrs : Instrs, stk : Stack) -> Int {
  loop instrs, stk {
    Cons(Cst(i), rest), stk => continue rest, Cons(i, stk)
    Cons(Add, rest), Cons(a, Cons(b, stk)) => continue rest, Cons(a + b, stk)
    Cons(Mul, rest), Cons(a, Cons(b, stk)) => continue rest, Cons(a * b, stk)
    Nil, Cons(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

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

Transition of Stack Machine

idea
基础软件中心

Code and stack:

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

Transition of the machine:

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

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

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

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

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

基础软件中心

Homework

Implement the compilation algorithm

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

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

$$\text{envs}: \quad \Gamma ::= \epsilon \mid (x, v) :: \Gamma$$

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))
  }
}
```

idea
基础软件中心

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:

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

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

Explanation

idea
基础软件中心

- 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)))
  }
}
```

idea
基础软件中心

Next: add new instructions to our VM to support the new langauge features

Compile Nameless. expr

```
enum 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

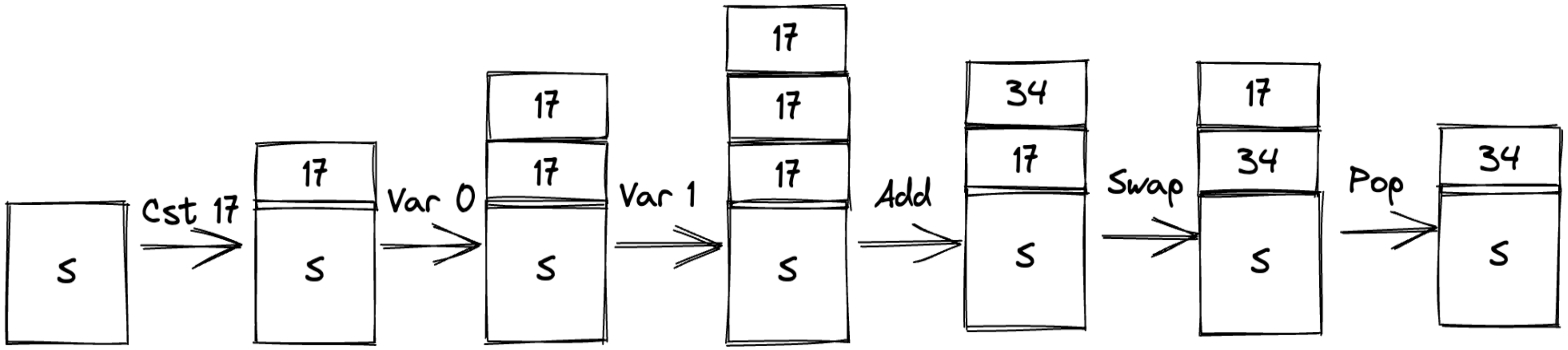
Stack Machine with Variables

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

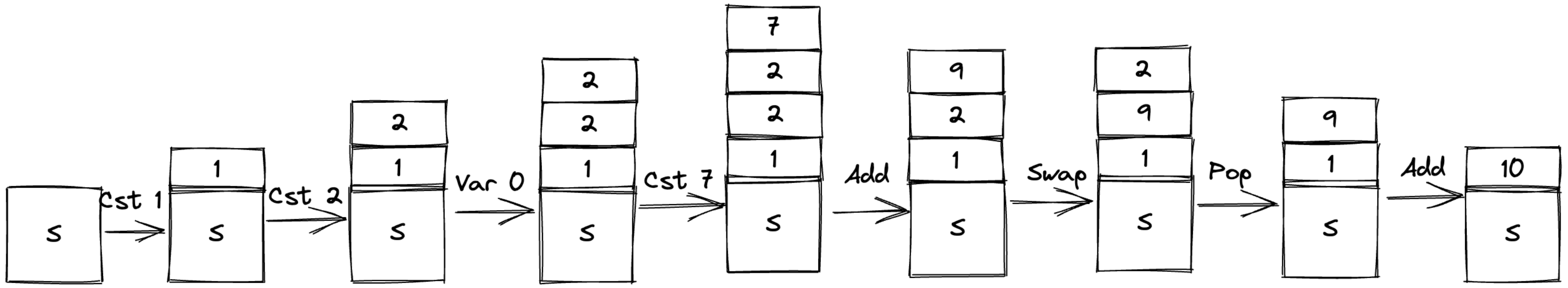
Consider the following program

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

is compiled to instructions

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

The execution on the stack:

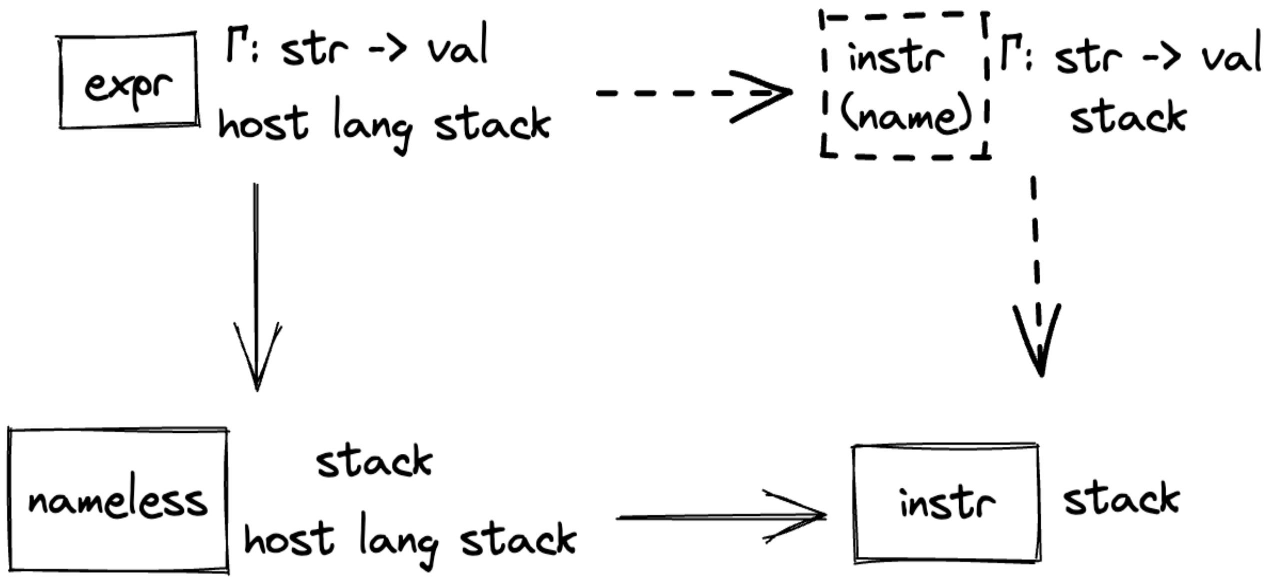


Summary 1

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 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

- 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)