

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

MoonBit Programming Challenge

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

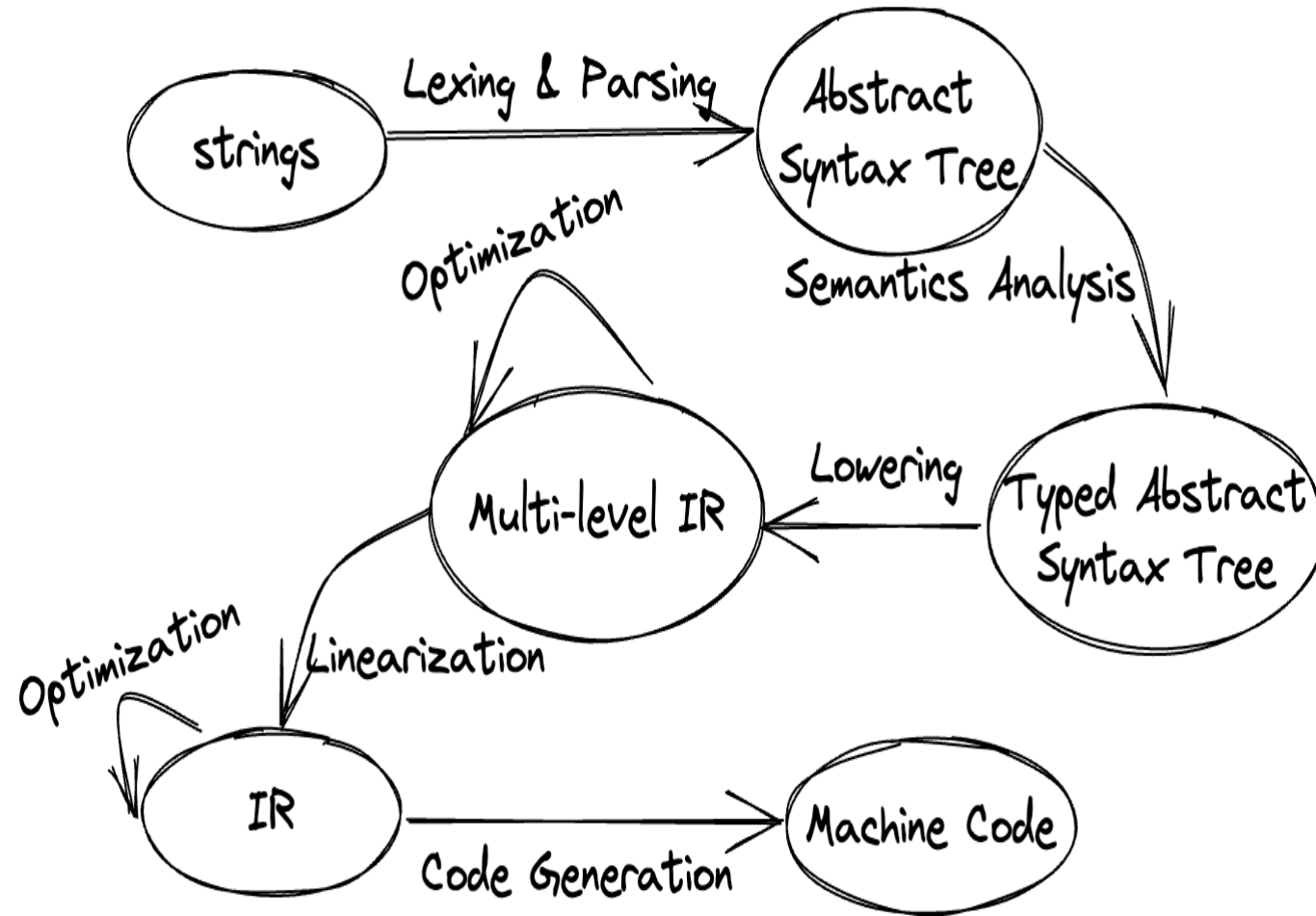
Why study compiler&interpreters ?

- 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

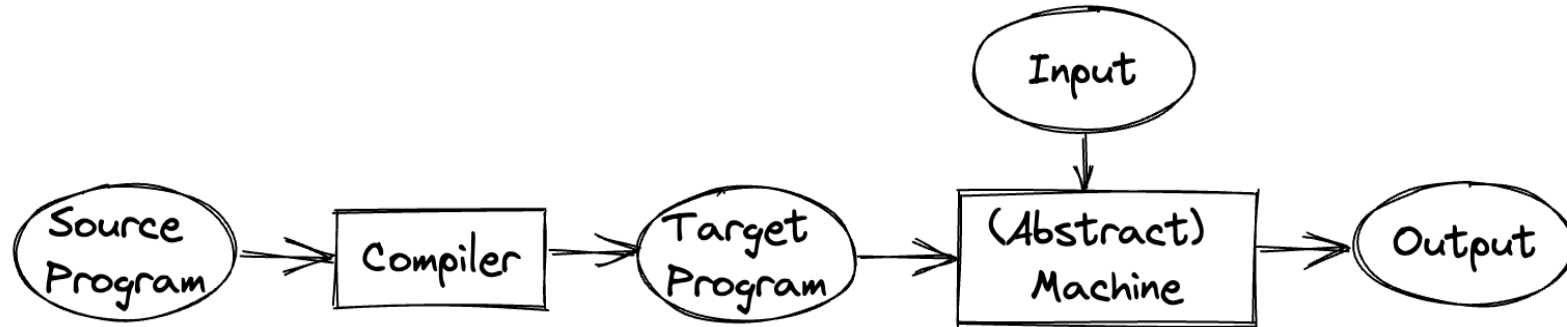
Lec	Topic	Lec	Topic
0	Introduction to language design and implementation	5	IR designs (ANF, CPS, KNF)
1	MoonBit crash course	6	Closure Calculus
2	Parsing	7	Register allocation
3	Semantics analysis and type inferences	8	Garbage collection
4	Bidirectional type checking		

Compilation Phases



Compilers, Interpreters

- Compilation and interpretation in two stages

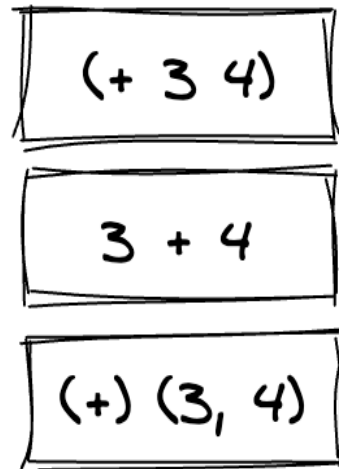


- The native compiler has a CPU interpreter
- Interpretation can be done in high level IRs (Python etc)

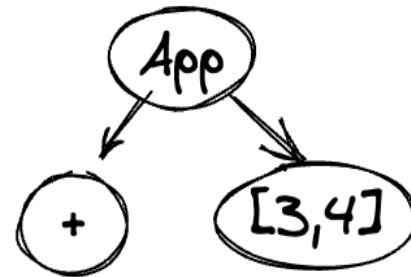
Lexing & Parsing

- 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



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
- 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, KNF, CPS etc

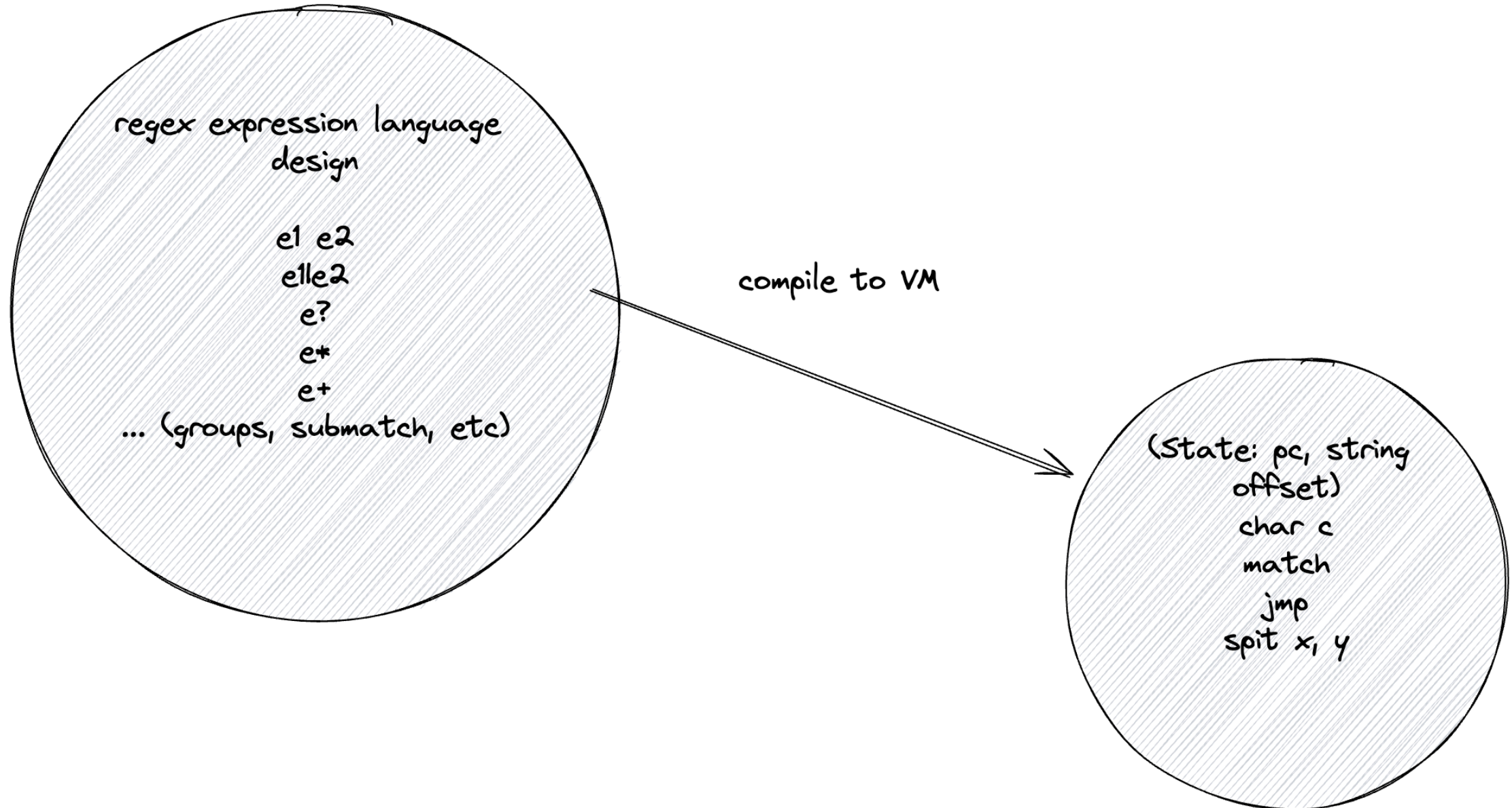
Linearization & optimizations

- 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

The smallest practical example: regular language



Regular language compiler

a	char a
e_1e_2	<i>codes for e_1</i> <i>codes for e_2</i>
$e_1 e_2$	split L1, L2 L1: <i>codes for e_1</i> jmp L3 L2: <i>codes for e_2</i> L3:
$e?$	split L1, L2 L1: <i>codes for e</i> L2:
e^*	L1: split L2, L3 L2: <i>codes for e</i> jmp L1 L3:
e^+	L1: <i>codes for e</i> split L1, L3 L3:

Regular language VM

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

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)  
  }  
}
```

What is the problem of our interpreter?

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

Compile/Lowering to a stack machine

```
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

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

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

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

Homework

Implement the compilation algorithm in MoonBit

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))
  }
}
```

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))
  }
}
```

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)))
  }
}
```


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

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

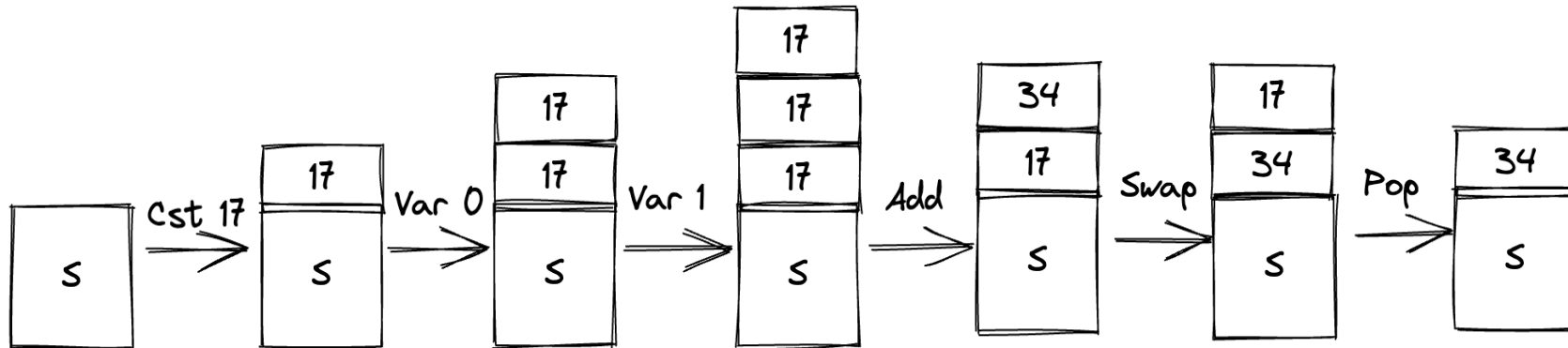
Stack Machine with Variables

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

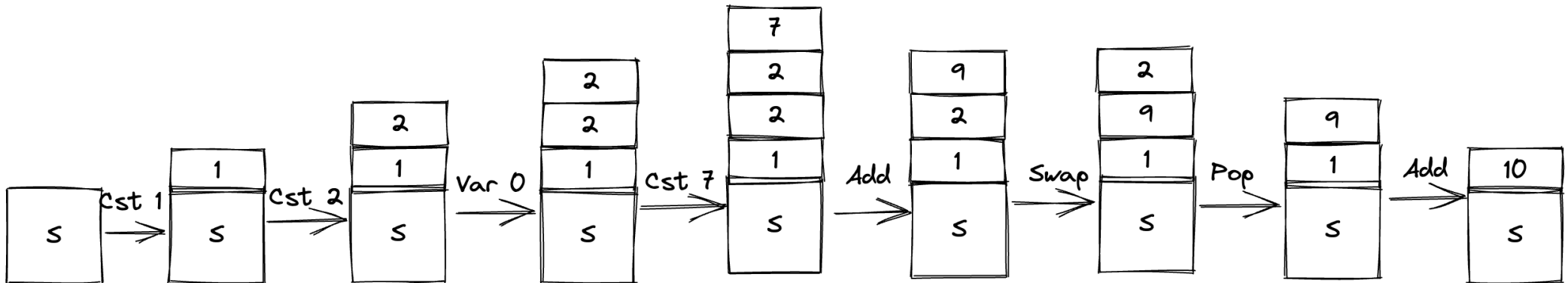
Consider the following program

```
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:



Homework

- Write an interpreter for the stack machine with variables
- Write a compiler to translate **Nameless. expr** to stack machine instructions