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)

idea
基础软件中心

Why study compiler&interpreters ?

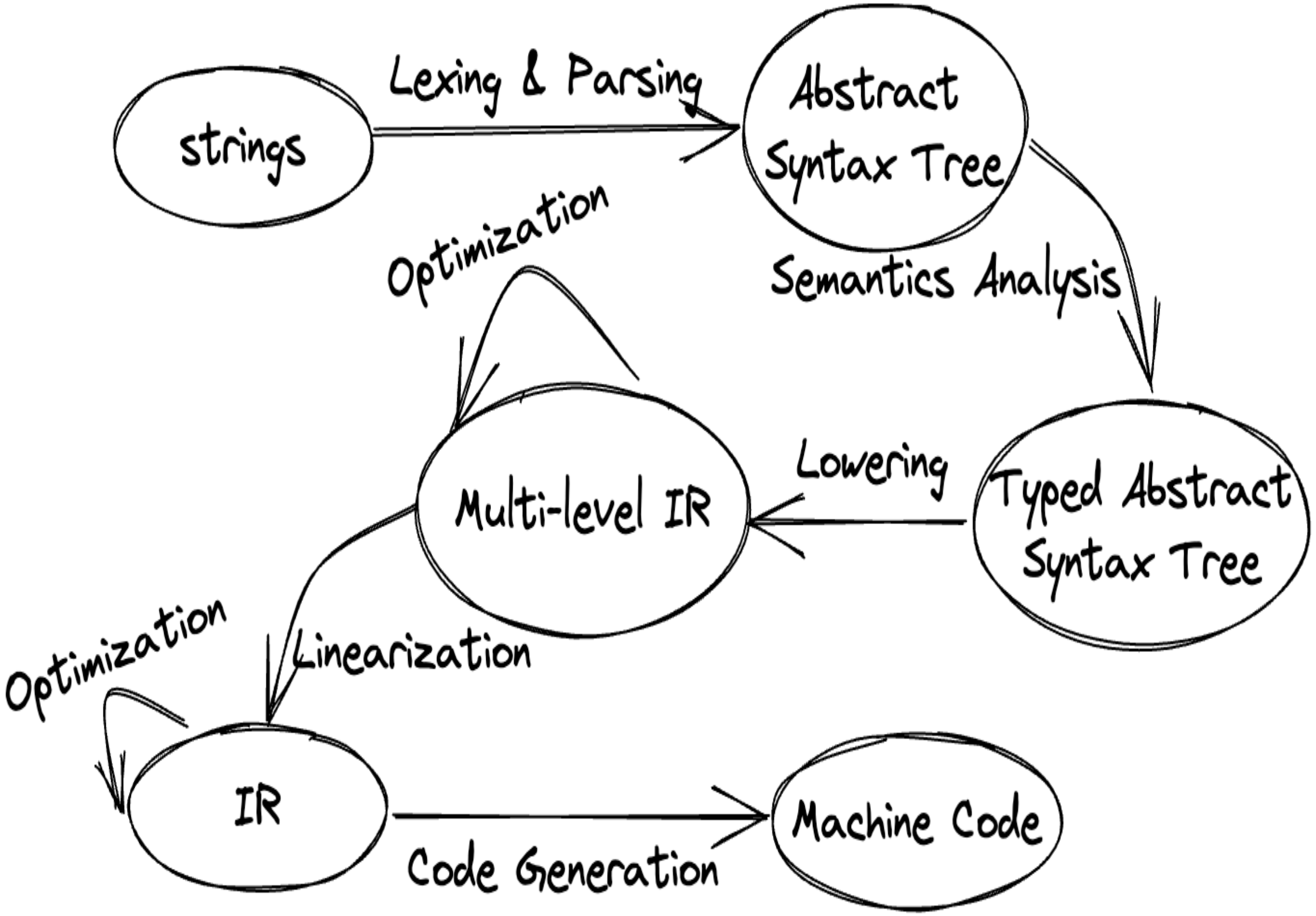
- It is fun
- Understand your tools you use everyday
- Understand the cost of abstraction
  - Hidden allocation when declaring local functions
  - Why memory leak happens
  - Good system programmers need write a toy C compiler
- Make your own DSLs for profit
- Develop a good taste

idea
基础软件中心

Course Overview

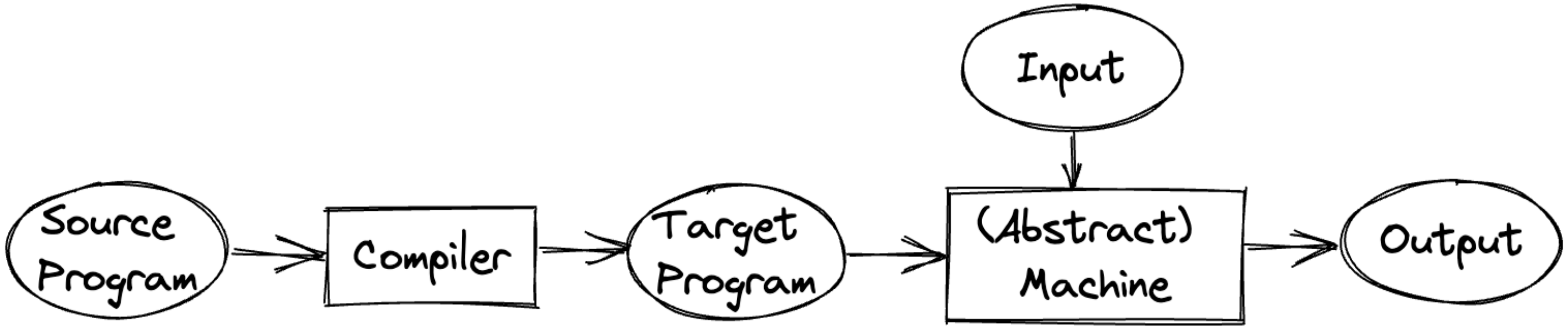
Lec	Topic	Lec	Topic
0	Introduction to language design and implementation	5	IR designs (ANF, CPS, KNF)
1	MoonBit crash course	6	Closure Calculus
2	Parsing	7	Register allocation
3	Semantics analysis and type inferences	8	Garbage collection
4	Bidirectional type checking		

Compilation Phases



Compilers, Interpreters

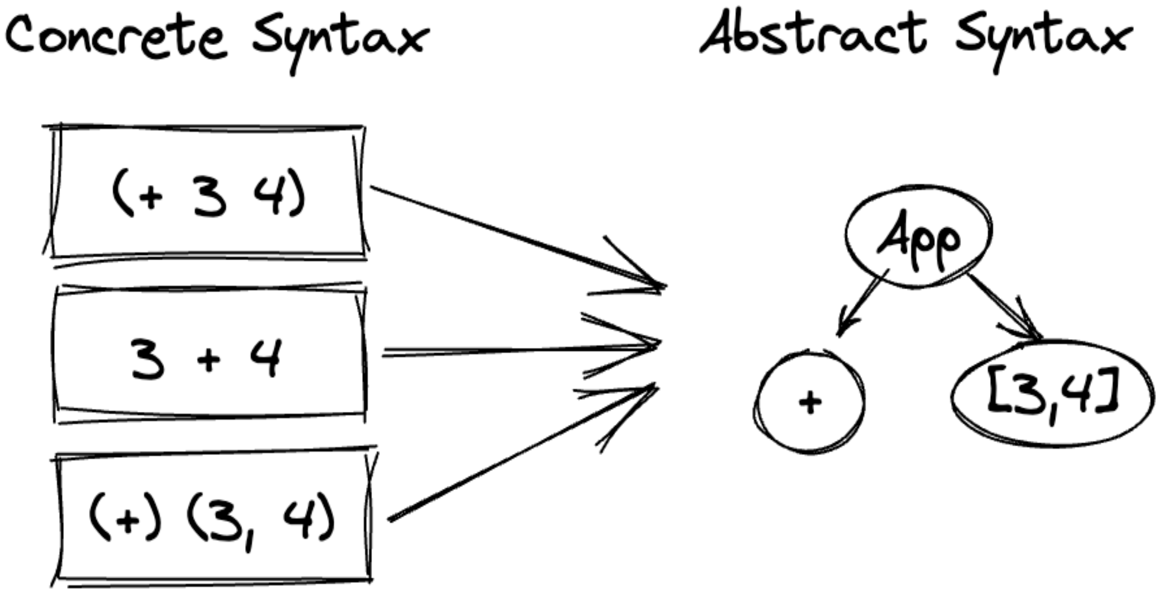
- Compilation and interpretation in two stages



- 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.



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

idea
基础软件中心

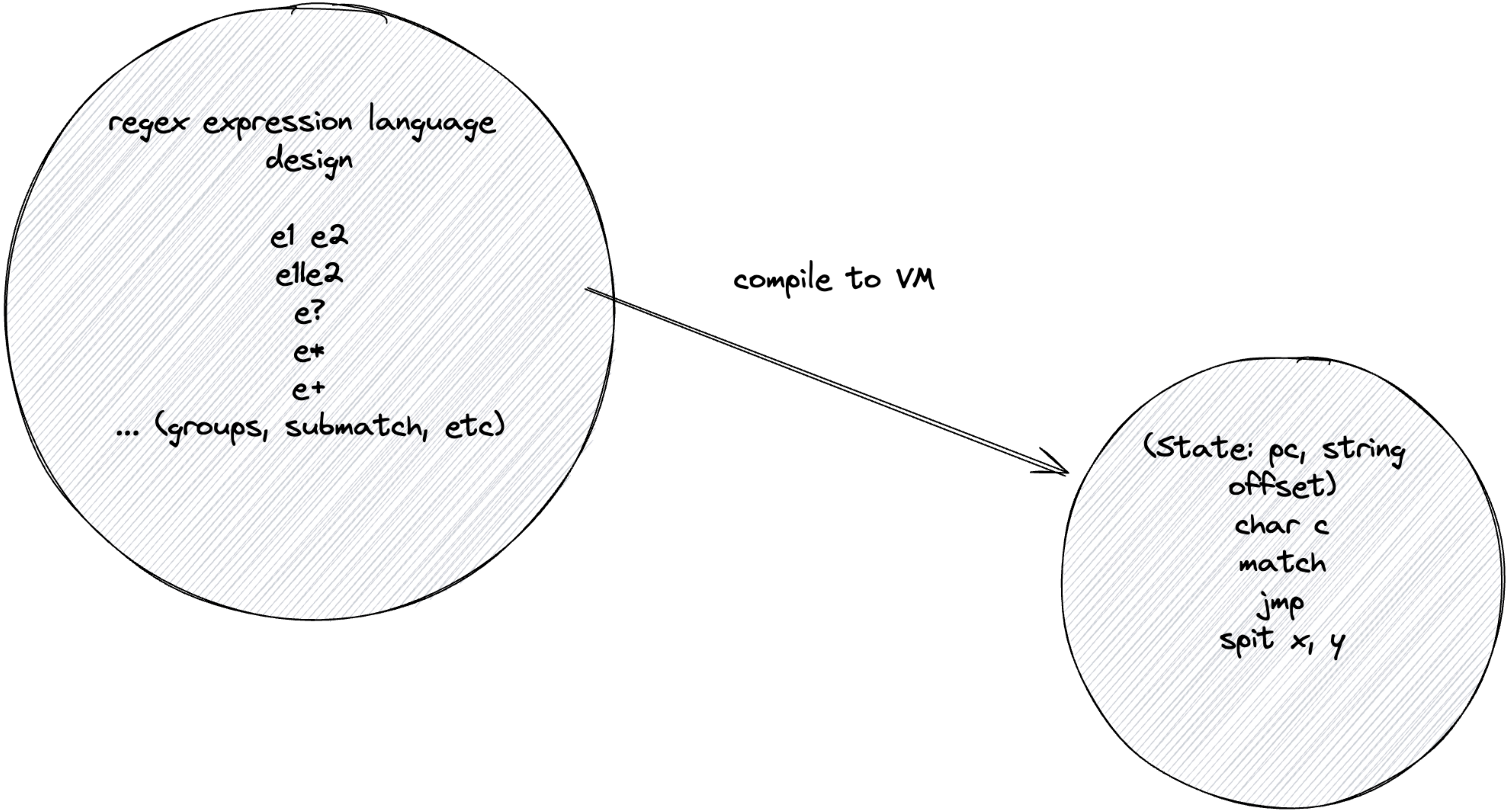
Linearization & optimizations

- 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

The smallest practical example: regular language



Pipelines

```
Run test | Debug test | Update test
test {
  let regex : Json = ["seq",["plus", "a"],["plus", "b" ]]
  let r: Regex = parse!(regex)
  let p: Program = compile(r)
  let i: MProgram = assemble(p)
  inspect!(i, content=
    #|0 : MChar('a')
    #|1 : MSplit(0, 2)
    #|2 : MChar('b')
    #|3 : MSplit(2, 4)
    #|4 : MMatch
    #|
  )
  let state: State = { .. State::default(), data: "ab" }
  let result: Bool = interpret(state, assemble(p))
  inspect!(result, content=
    "true"
  )
}
```

- cst -> ast -> ir0 (asm) -> ir1(machinecode) ---interpreter---> result

Pipelines (cont.)

- cst -> ast (parser)

```
fn parse(cst : Json) -> Regex! {
  match cst {
    String(s) => {
      // TODO: reduce the Empty usage
      let mut acc = Empty
      for i in s {
        acc = Seq(acc, Char(i))
      }
      acc
    }
    ["or", a, b] => Alt(parse!(a), parse!(b))
    ["seq", a, b] => Seq(parse!(a), parse!(b))
    ["star", a] => Star(parse!(a))
    ["plus", a] => Plus(parse!(a))
    ["opt", a] => Opt(parse!(a))
    _ => fail!("invalid cst \{cst}")
  }
}
```

pipelines (cont.)

- ast -> ir0 (compile)

```
fn compile_aux(regex : Regex) -> Array[Instr] {
  match regex {
    Empty => []
    Seq(r1, r2) => [..compile_aux(r1), ..compile_aux(r2)]
    Alt(r1, r2) => {
      let l1 = gen_label()
      let l2 = gen_label()
      let l3 = gen_label()
      [
        Split(l1, l2),
        Label(l1),
        ..compile_aux(r1),
        Jmp(l3),
        Label(l2),
        ..compile_aux(r2),
        Label(l3),
      ]
    }
    ...
```

Regular language compiler

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:`	

pipelines (cont.)

基础软件中心

- ir0 -> ir1 (link)

```
fn assemble(p : Program) -> MProgram {
  let labels = collect_labels(p)
  let instrs = []
  for instr in p.0 {
    match instr {
      Char(c) => instrs.push(MChar(c))
      Match => instrs.push(MMatch)
      Jmp(l) => instrs.push(MJmp(labels[l].unwrap()))
      Split(l1, l2) =>
        instrs.push(MSplit(labels[l1].unwrap(), labels[l2].unwrap()))
      Label(l) => ()
    }
  }
  instrs
}
```

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

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

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

Transition of Stack Machine

Code and stack:

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

Transition of the machine:

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

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

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

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)

idea
基础软件中心

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

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

Homework

Implement the compilation algorithm in MoonBit

idea
基础软件中心

Tiny Language 1

Abstract Syntax: add names

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

idea
基础软件中心

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)))
  }
}
```

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

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

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

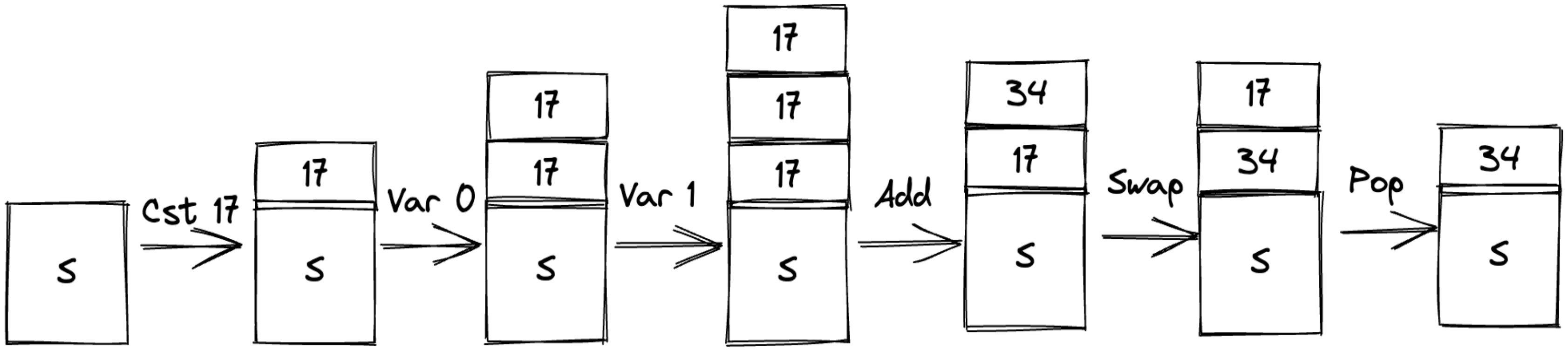
Stack Machine with Variables

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

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:



More examples

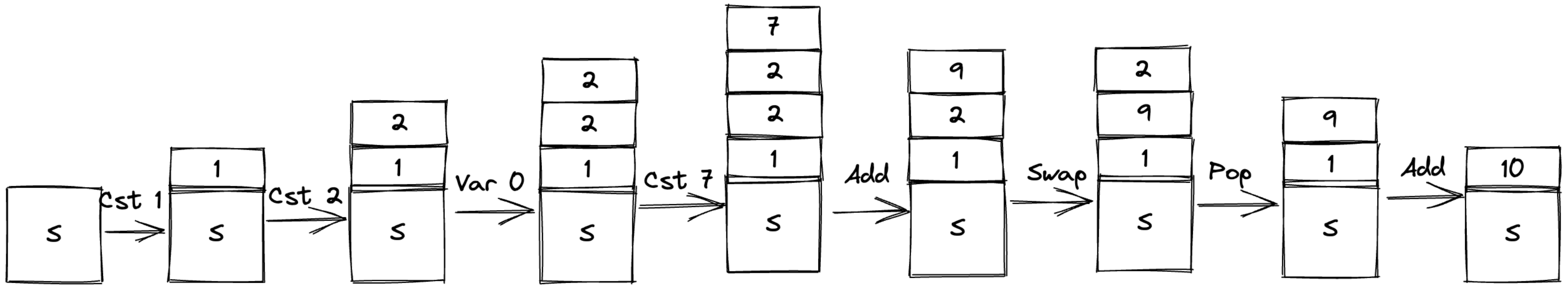
Consider the following program

```
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:



Homework

- Write an interpreter for the stack machine with variables
- Write a compiler to translate $\mathsf{Nameless.expr}$ to stack machine instructions