编译器和解释器导论

基础软件理论与实践公开课

张宏波

- 课程: https://bobzhang.github.io/courses/
- 讨论: https://bbs.csdn.net/forums/raelidea
- 目标听众:
    - 对程序语言设计和编译感兴趣的人
    - 无需程序语言理论基础
- 示例代码: ReScript
- Homebrew
- ReScript 是 ML编程语言的一门方言:
    - 为什么ML适宜制作编译器。
    - 在包含windows的大多数平台上都便捷运行

招募

- 工作地点：深圳
  - 程序语言工具链
  - 开发者工具
  - 垃圾回收
  - 代码编辑器
  - IDE等

导言

为什么要学习编译器&解释器理论

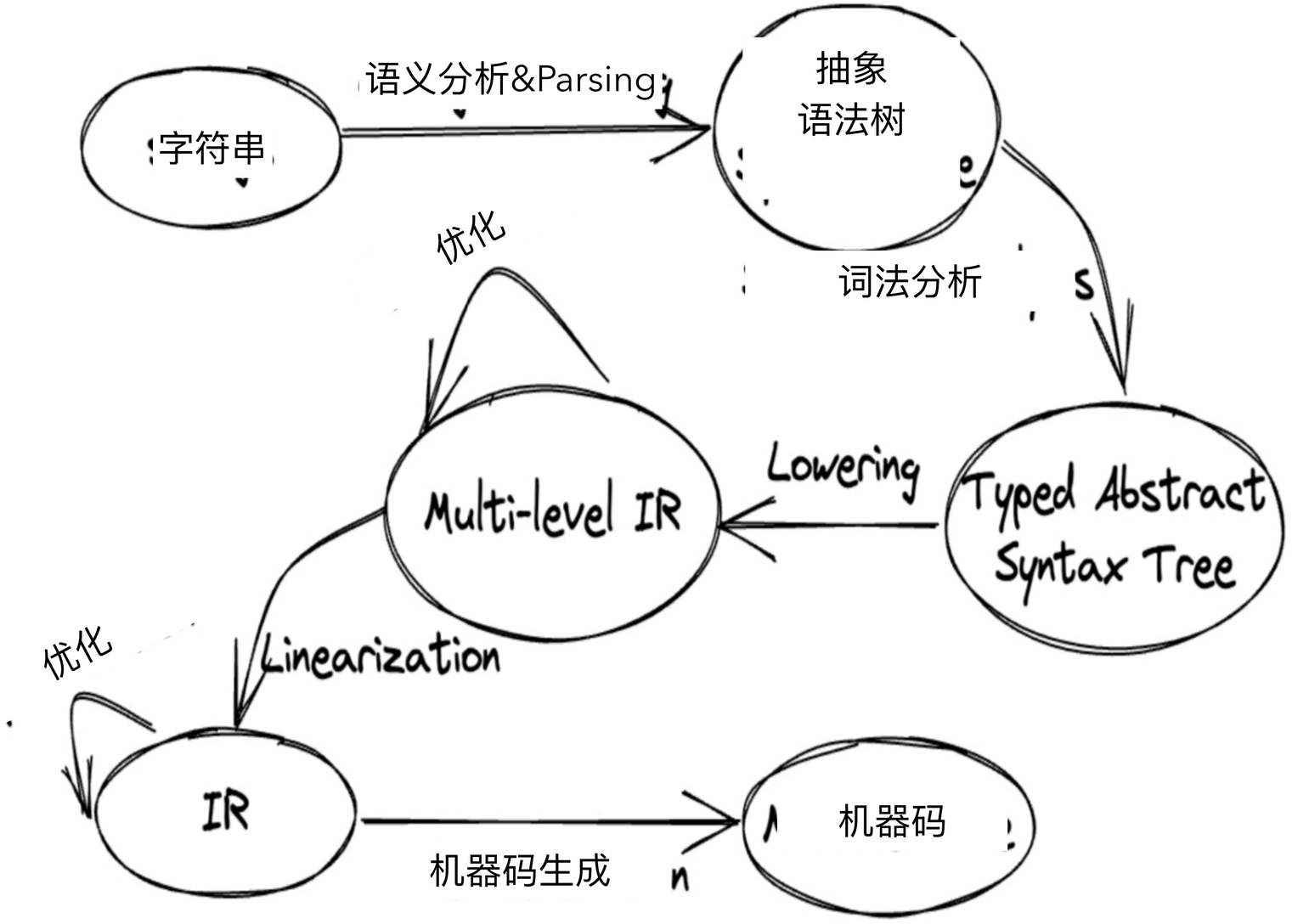
- 有趣
- 理解你每天使用的工具链
- 理解抽象开销:
    - 声明局部函数时的隐藏分配.
    - 为什么发生内存泄漏
- 制作你自己的DSL
- 建立良好的品味

课程一览

0	导言	6	堆栈机器与编译
1	ReScript快速入门	7	WebAssembly
2	邱奇演算	8	垃圾回收与内存管理
3	变量名，绑定，De Bruijn索引*	9	类型检查
4	闭包变换	10	类型推断与unification
5	模式匹配	11&12	形式验证，客座演讲

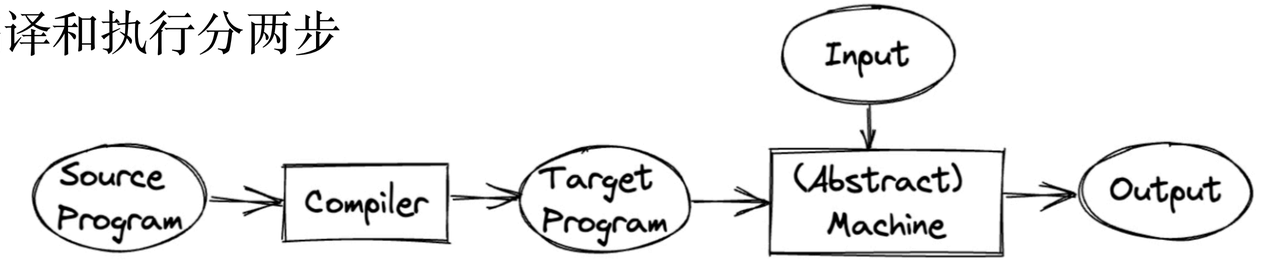
- *演算时无需命名绑定变量，比如Y combinator，S combinator

编译步程

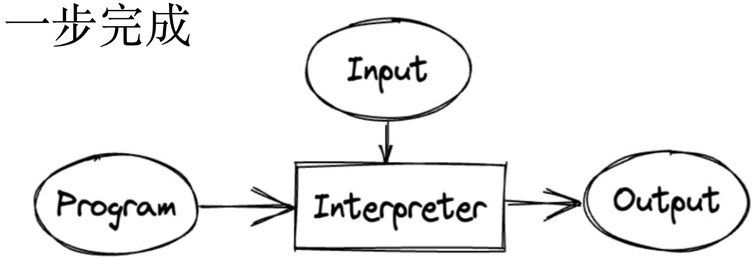

字符串
语义分析&Parsing
抽象
语法树
词法分析
优化
Multi-level IR
Lowering
Typed Abstract
Syntax Tree
优化
Linearization
IR
机器码
机器码生成

编译，转译与解释

编译：编译和执行分两步



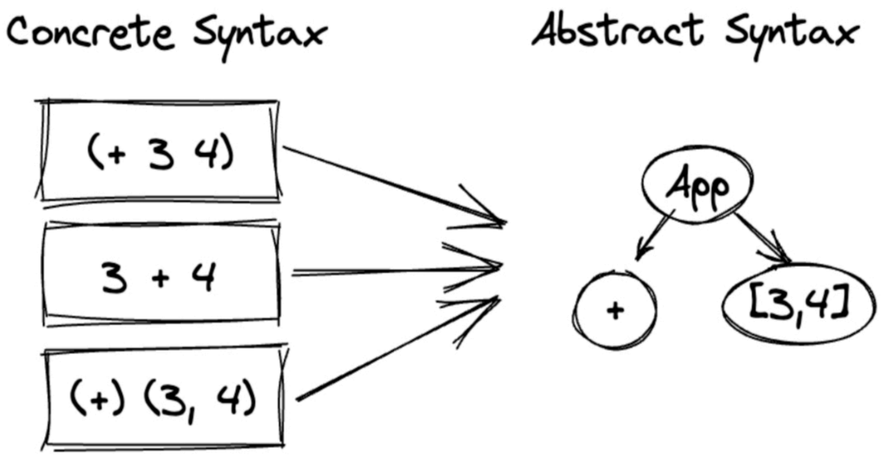
解释：一步完成



转译：从一种程序语言转译为近似抽象层的另一种

词法分析与Parsing

- 从字符串到抽象语法树
- 一般分为两个步骤
  - tokenization
  - parsing
- 众多工具：Lex，Yacc，Menhir，Antlr，TreeSitter，Parsing conbinators 等



语意分析

- 建立符号表，解决变量，库
- 类型检查&推断
  - 检查程序用正确的变量类型执行
  - 类型信息缺失时推理正确类型
  - 类型类&隐含解决
  - 检查安全及其他问题
    - 生命周期检查
- Type soundness：无运行时类型错误

语言特有转换与优化

- 类 / 模块 / 对象 / 类型类脱糖
- 模式匹配脱糖
- 闭包变换
- 专对语言特有的优化
- IR relatively rich，MLIR

线性化&优化

- Laguage & Platform agnostics
- 优化
  - 常量折叠，propogation，CSE，partial evluation
  - 循环不变量转移
  - 尾调用消除
  - 过程内，过程内优化
- IR 简化：three address code，LLVM IR 等

专有机器码生成

- 指令选择
- 寄存器分配
- 指令调度专有机器码优化
- 数学近似计算，DSA

抽象语法 vs 具体语法

- 现代语言设计：parsing阶段不进行词法分析
  - 反面教材：C++ parsing很难，错误信息难以理解

- 从具体语法到抽象语法可以多对一关系
- 这门课从抽象语法开始
  - 之后用ReScript演示parsing

小程序语言 0

抽象语法

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

解释器

```
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})
  }
}
```

形式化

语义

求值结果是一个值，在我们的程序语言里是一个整型

$$\begin{aligned}&\text{terms}: && e ::= \mathsf{Cst}(i) \mid \mathsf{Add}(e_1, e_2) \mid \mathsf{Mul}(e_1, e_2)\\ &\text{values}: && v ::= i \in \mathsf{Int32}\end{aligned}$$

求值规则

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

推理规则

- 求值关系 e $\Downarrow$ v的意思是 释义 e到v比如
  - Cst(42) $\Downarrow$ 42
  - Add(Cst(3), Cst(4)) $\Downarrow$ 7

- 推理规则确保一种简洁的方式来体现，分析特定语言：
  - 如果前提为真，那么结论为真
  - 公理 没有前提
  - 推断规则可以用元变量替换来实例化
    - (e,e1,e2,x,i ...) 含有语句，程序变量，整型。

证明树

- 实例化的规则可以构造证明树
- $e \Downarrow v$ 仅存在于有一个由实例化规则正确构造有限证明树，它的叶子节点是公理。

我们的解释器有什么问题?

转译到堆栈机器后解释

```
type instr =  Cst (int) | Add | Mul
```

```
let rec eval = (instrs,stk) => {
  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})
  | _ => assert false
  }
}
```

语义

- 这个机器有两部分
  - 指针代码 c 指向下一个执行段
  - 堆栈区 s 储存中间结果
- 堆栈符号：堆栈顶位于左侧

$$s \longrightarrow v :: s$$

将 v 压入 s

$$v :: s \longrightarrow s$$

将 v 从 s 弹出

栈机器的过渡

代码和堆栈：

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

Transition of the machine:

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

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

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

形式化

- 汇编对应下面的数学形式

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

- 方括号通常用于表示 汇编 的符号
- Invariant: stack balanced property
- 验证：使用coq

编译

- 对 expr 语言求值使用宿主程序的堆栈
- 堆栈机器明确操纵堆栈

编译的正确性

- 正确的编译的语义满足以下等式

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

小程序语言1

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

解释器

环境语义

```
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})
  }
}
```

形式化

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

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

Notations for the environment:

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

The evaluation rules:

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

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

我们的求值有什么问题?

- 在编译时有什么冗余工作可以消除?
- 变量名长度影响我们的运行时效率

小程序语言2

在List里变量的位置深度：

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

语义

求值程序

```
type env = list<int>

let rec eval = (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})
  }
}
```

语义

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:

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

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

解释

- expr的求值环境 Γ 包含名字和值。
- Nameless.expr 的求值环境只包含编译期索引的值

转译 expr 到 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}))
    }
}
```

编译 Nameless.expr

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

- 新的语义组织:

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

- s[i] 读取堆栈顶端的第i个值

带变量的堆栈机器

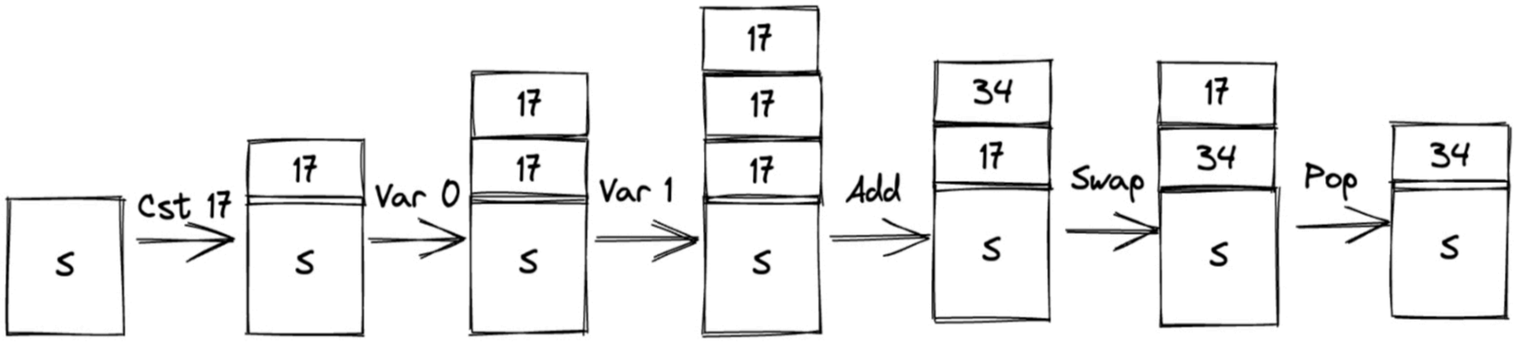
如下程序

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

编译为以下操作

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

堆栈的执行



问题

- 很确定的是我们需要Var(n)来索引堆栈理的变量
- 但是为什么我们需要Swap和pop操作?

更多例子

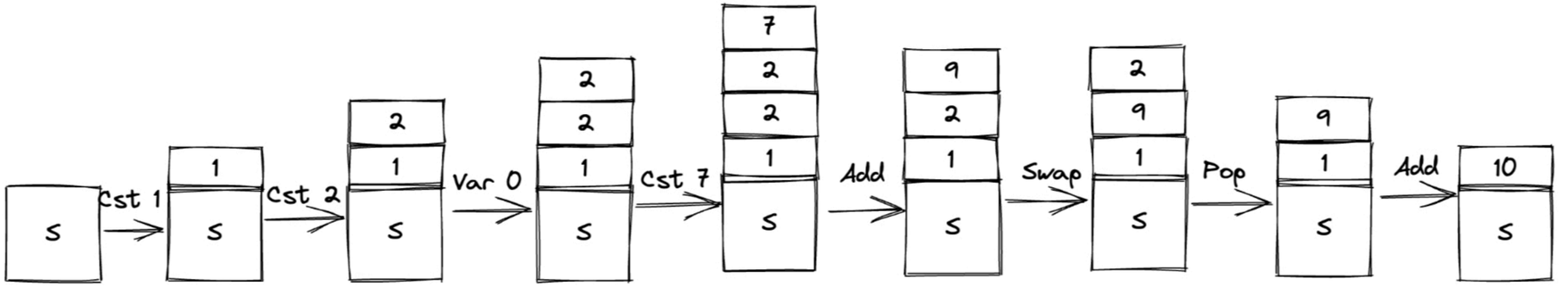
如下程序

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

编译为以下操作指令

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

堆栈的执行



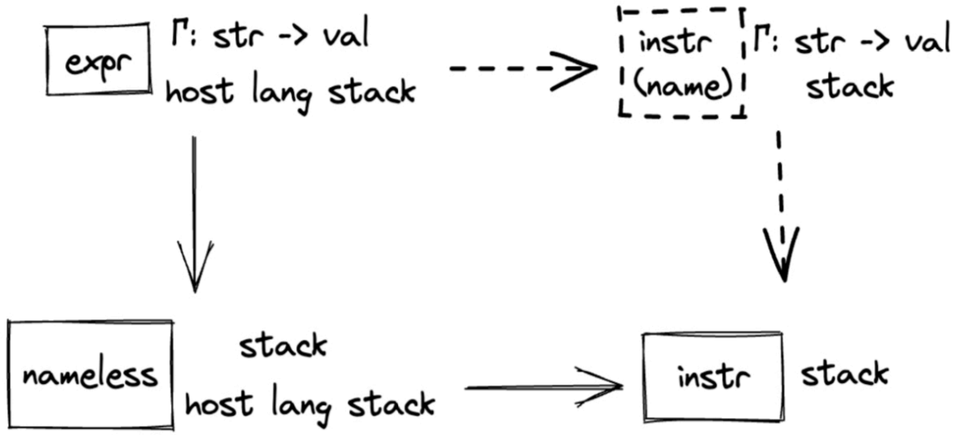
总结

- 从编译过程我们得到了什么？比较运行时环境
    - 求值 expr
        - 本地变量的符号环境
        - 临时的宿主语言（隐式）堆栈

Γ

- 求值Nameless.expr
        - 本地变量的堆栈
        - 临时的宿主语言（隐式）堆栈
- 堆栈机器的构成，我们有
        - 本地和临时变量的堆栈

总结



作业

- 用含变量的堆栈机器写一个解释器
- 写一个编译器将Nameless.expr编译为堆栈机器操作
- 实现虚线的部分（1个语言+2个编译器）