Stack Machines

Concepts of Programming Languages
Lecture 25

CAS CS 320

Outline

- Finish our demo implementation of HM^{-}
- Discuss **stack-based languages** and **stack machines**
- Demo an implementation of compiling arithmetic expressions

Practice Problem

```
fun x -> fun y -> fun z -> x z (y z)
```

Determine the principle type of the above expression

$\forall a. \forall b. \forall c. (a \to b \to c) \to (a \to b) \to a \to c$

Solution `fun x -> fun y -> fun z -> x z (y z)`

$\cdot \vdash \lambda x.\ \lambda y.\ \lambda z.\ x\ z\ (y\ z) : \alpha \to \beta \to \gamma \to \eta \dashv C$

Γ

$\{x : \alpha,\ y : \beta,\ z : \gamma\} \vdash (x\ z)\ (y\ z) : \eta \dashv \boxed{\delta \doteq \varepsilon \to \eta,\ \alpha \doteq \gamma \to \delta,\ \beta \doteq \gamma \to \varepsilon}$ C

$\Gamma \vdash x\ z : \delta \dashv \alpha \doteq \gamma \to \delta$

$\Gamma \vdash x : \alpha \dashv \emptyset$

$\Gamma \vdash z : \gamma \dashv \emptyset$

$\Gamma \vdash y\ z : \varepsilon \dashv \beta \doteq \gamma \to \varepsilon$

$\Gamma \vdash y : \beta \dashv \emptyset$

$\Gamma \vdash z : \gamma \dashv \emptyset$

$\cancel{\delta \doteq \varepsilon \to \eta}$ $v \doteq t$

$\cancel{\alpha \doteq \gamma \to \cancel{\delta}\ \varepsilon \to \eta}$ $v \doteq t$

$\cancel{\beta \doteq \gamma \to \varepsilon}$ $v \doteq t$

$S = \{\delta \mapsto \varepsilon \to \eta,\ \boxed{\alpha} \mapsto \gamma \to \varepsilon \to \eta,\ \boxed{\beta} \mapsto \gamma \to \varepsilon\}$

$S_T = S(\boxed{\alpha} \to \boxed{\beta} \to \gamma \to \eta)$

$(\gamma \to \varepsilon \to \eta) \to (\gamma \to \varepsilon) \to \gamma \to \eta$

$\boxed{\forall a. \forall b. \forall c. (a \to b \to c) \to (a \to b) \to a \to c}$

Recap

Recall: Principle Types

$$\Gamma \vdash e : \tau \dashv \mathscr{C}$$

The constraints $\mathscr{C}$ defined a *unification problem*. Given a most general unifier $\mathscr{S}$ we can get the "actual" type of e:

$$\text{principle}(\tau, \mathscr{C}) = \forall \alpha_1 \ldots \forall \alpha_k . \mathscr{S}\tau \ \texttt{ where } \ \text{FV}(\mathscr{S}\tau) = \{\alpha_1, \ldots, \alpha_k\}$$

i.e, the **principle type** of e (note: it may not exist). Every type we *could* give e is a *specialization* of $\forall \alpha_1, \ldots, \alpha_k . \mathscr{S}\tau$

Recall: HM$^-$ (Typing Variables)

$$\frac{(x : \forall \alpha_1 . \forall \alpha_2 \ldots \forall \alpha_k . \tau) \in \Gamma \qquad \beta_1, \ldots, \beta_k \text{ are fresh}}{\Gamma \vdash x : [\beta_1/\alpha_1] \ldots [\beta_k/\alpha_k]\tau \dashv \varnothing} \text{(var)}$$

If x is declared in Γ, then x can be given the type τ *with all free variables replaced by **fresh variables***

This is where the polymorphism magic happens

Fresh variables can be unified with anything

Recall: Putting everything together

input: program P (sequence of top-level let-expressions)

$\Gamma \leftarrow \varnothing$

FOR EACH top-level let-expression $\text{let } x = e$ in P:

1. *Constraint-based inference:* Determine τ and $\mathscr{C}$ such that $\Gamma \vdash e : \tau \dashv \mathscr{C}$ is derivable
2. *Unification:* Solve $\mathscr{C}$ to get a most general unifier $\mathscr{S}$ (**TYPE ERROR** if this fails)
3. *Generalization:* Quantify over the free variables in $\mathscr{S}\tau$ to get the principle type $\forall \alpha_1 \ldots \forall \alpha_k . \mathscr{S}\tau$ of e
4. Add $(x : \forall \alpha_1 \ldots \forall \alpha_k . \mathscr{S}\tau)$ to Γ

demo

(finishing up type inference)

Stack Machines

High-Level

High-Level

A **stack–oriented language** is a PL which directly manipulates a stack of values (or multiple stacks)

High-Level

A **stack–oriented language** is a PL which directly manipulates a stack of values (or multiple stacks)

<u>There are roughly 2 categories:</u>

High-Level

A **stack-oriented language** is a PL which directly manipulates a stack of values (or multiple stacks)

<u>There are roughly 2 categories:</u>

- "usable" stack-oriented languages, e.g., Forth

High-Level

A **stack-oriented language** is a PL which directly manipulates a stack of values (or multiple stacks)

There are roughly 2 categories:

- "usable" stack-oriented languages, e.g., Forth
- instruction sets for virtual stack machines, e.g., JVM, CPython, Lua (not any more), OCaml bytecode interpreter

High-Level

A **stack-oriented language** is a PL which directly manipulates a stack of values (or multiple stacks)

<u>There are roughly 2 categories:</u>

» "usable" stack-oriented languages, e.g., Forth

» instruction sets for virtual stack machines, e.g., JVM, CPython, Lua (not any more), OCaml bytecode interpreter *(these aren't exactly programming languages)*

Abstract Virtual Machines

Abstract Virtual Machines

A virtual (stack) machine is a computational abstraction, like a Turing machine (but usually **easier to implement**)

Abstract Virtual Machines

A virtual (stack) machine is a computational abstraction, like a Turing machine (but usually **easier to implement**)

Virtual machines are typically implemented as **bytecode interpreters**, where "programs" are streams of bytes and a command is represented as a byte (plus possibly some extra data)

Benefits of Stack Machines

Benefits of Stack Machines

Simplicity: Stacks aren't too complicated

Benefits of Stack Machines

Simplicity: Stacks aren't too complicated

Portability: Any OS should be able to handle a stream of bytes, so the machine dependent part of our programming language can be simplified

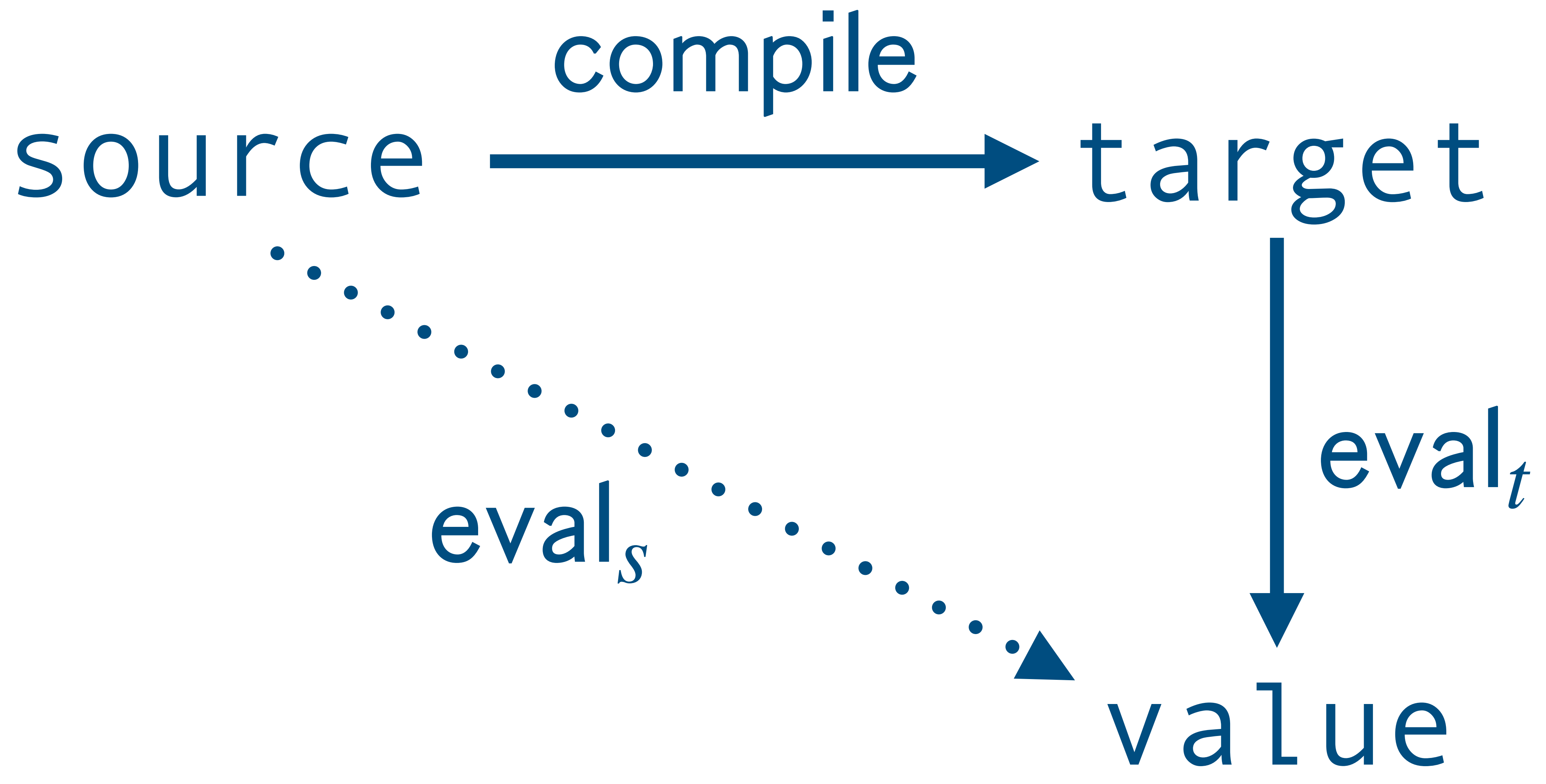
Benefits of Stack Machines

Simplicity: Stacks aren't too complicated

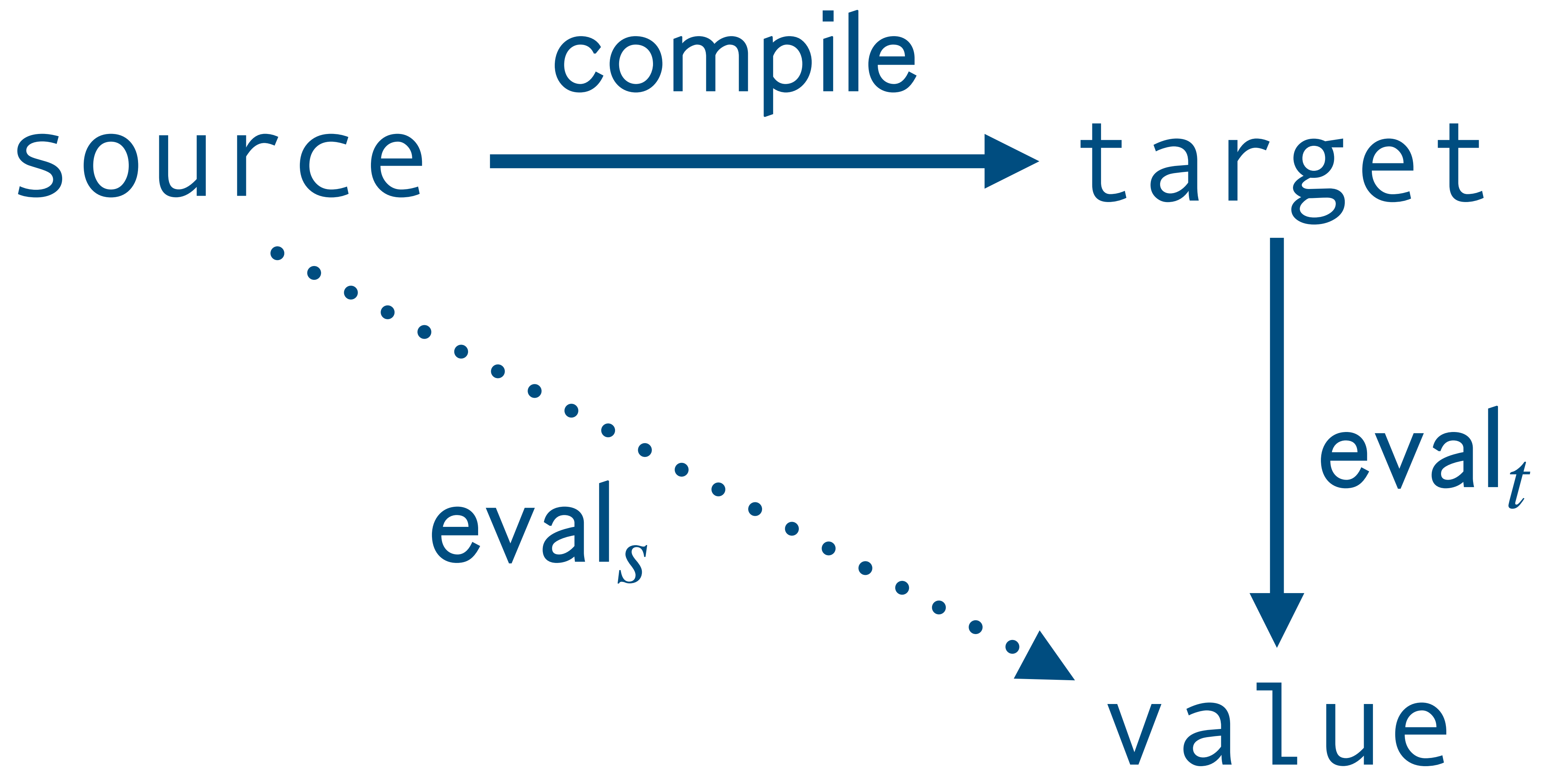
Portability: Any OS should be able to handle a stream of bytes, so the machine dependent part of our programming language can be simplified

Efficiency (sort of): They can be implemented in low-level languages, and so will generally be faster than the interpreters we build in this course (though not as fast as natively compiled code)

Looking Ahead: Compilation

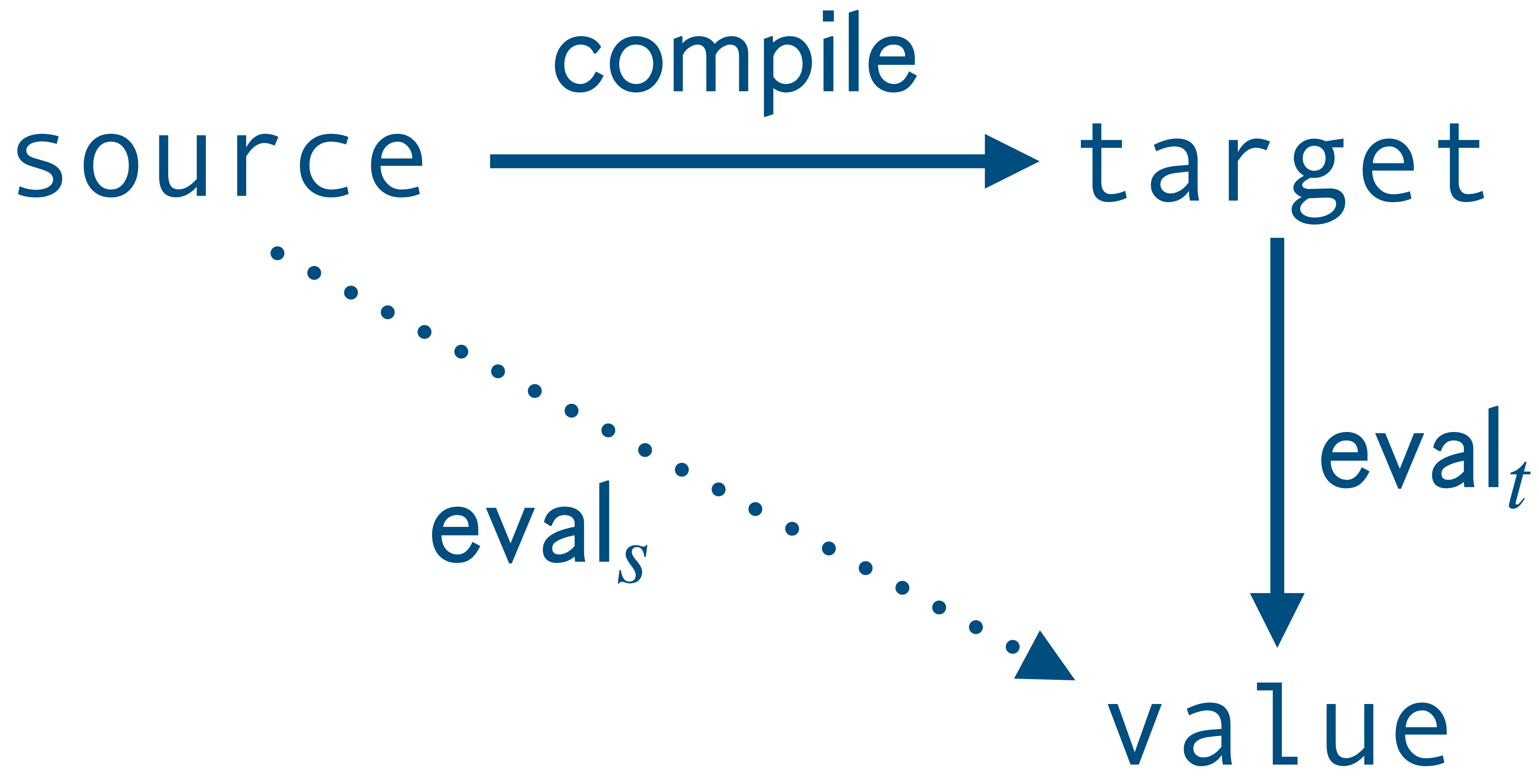


Looking Ahead: Compilation



Compilation is the process of translating a program in one language to another, maintaining semantic behavior

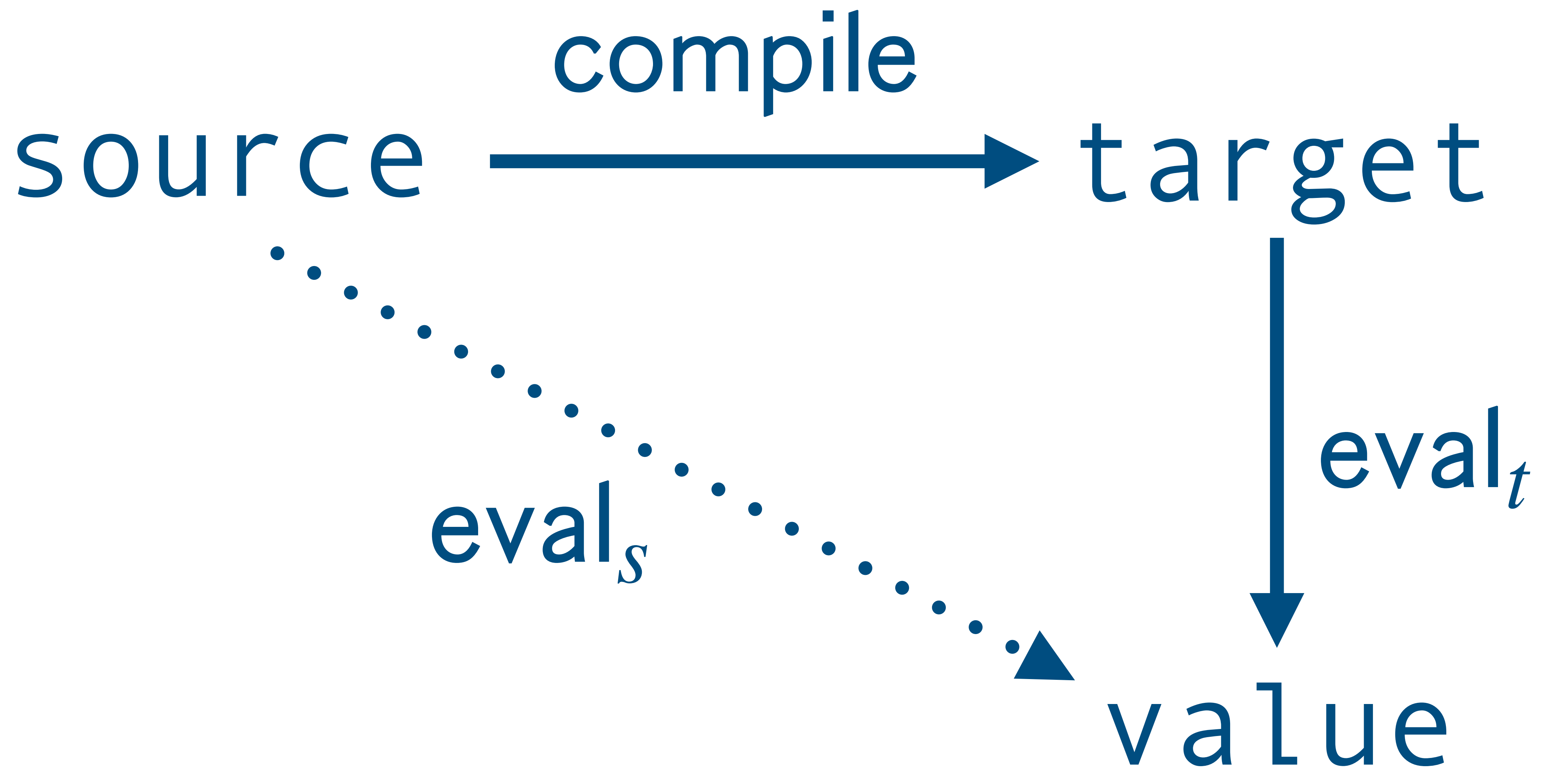
Looking Ahead: Compilation



Compilation is the process of translating a program in one language to another, maintaining semantic behavior

Compilation can be a part of interpretation as well, like with **bytecode interpretation** (this is what OCaml does)

Looking Ahead: Compilation



Compilation is the process of translating a program in one language to another, maintaining semantic behavior

Compilation can be a part of interpretation as well, like with **bytecode interpretation** (this is what OCaml does)

The simple case for today: *every arithmetic expression can be represented as an equivalent expression in reverse polish notation*

Stack-Based Arithmetic

Stack-Based Arithmetic (Syntax)

repetition

<prog> ::= {<com>}

<com> ::= ADD | SUB | MUL | DIV | PUSH <num>

<num> ::= $\mathbb{Z}$

PUSH 2 PUSH 3 ADD

Stack-Based Arithmetic (Semantics)

e.g.

$\langle 2 :: 3 :: \emptyset, \text{ADD SUB} \rangle$

$$\langle \mathcal{S}, P \rangle$$

A **value** is an integer ($\mathbb{Z}$)

A **configuration** is made up of a stack ($\mathcal{S}$) of values and a program (P) given by **<prog>**

Stack-Based Arithmetic (Semantics)

$\langle 2 :: 3 :: \emptyset, \text{ADD } P \rangle \longrightarrow \langle 5 :: \emptyset, P \rangle \longrightarrow \cdots$

$$\frac{}{\langle m :: n :: \mathcal{S}, \texttt{ADD}\ P \rangle \longrightarrow \langle \underline{(m+n) :: \mathcal{S}}, P \rangle}\,(\texttt{add})$$

$$\frac{}{\langle m :: n :: \mathcal{S}, \texttt{SUB}\ P \rangle \longrightarrow \langle (m-n) :: \mathcal{S}, P \rangle}\,(\texttt{sub})$$

$$\frac{}{\langle m :: n :: \mathcal{S}, \texttt{MUL}\ P \rangle \longrightarrow \langle (m \times n) :: \mathcal{S}, P \rangle}\,(\texttt{mul})$$

$$\frac{n \neq 0}{\langle m :: n :: \mathcal{S}, \texttt{DIV}\ P \rangle \longrightarrow \langle (m/n) :: \mathcal{S}, P \rangle}\,(\texttt{div})$$

$$\frac{}{\langle \mathcal{S}, \texttt{PUSH}\ n\ P \rangle \longrightarrow \langle n :: \mathcal{S}, P \rangle}\,(\cancel{\texttt{div}})$$

push

$$\frac{}{\langle m :: \emptyset, \text{ADD } P \rangle \longrightarrow \text{ERROR}}$$

(stack underflow)

could have error config

Example (Evaluation)

$\langle \emptyset,$ **PUSH 2 PUSH 3 SUB PUSH 4 MUL**$\rangle \to$

$\langle 2 :: \emptyset,$ PUSH 3 SUB PUSH 4 MUL$\rangle \to$

$\langle 3 :: 2 :: \emptyset,$ S P4 M$\rangle \to$

$\langle 1 :: \emptyset,$ P 4 M$\rangle \to$

$\langle 4 :: 1 :: \emptyset,$ M$\rangle \to \langle \boxed{4} :: \emptyset, \varepsilon \rangle$ ✓

demo

(stack machine)

Compiling Arithmetic Expressions

$$
\begin{array}{ccc}
\texttt{n} & \Longrightarrow & \texttt{PUSH n} \\
e_1 \;\texttt{+}\; e_2 & \Longrightarrow & \mathscr{C}(e_2)\;\; \mathscr{C}(e_1)\;\; \texttt{ADD} \\
e_1 \;\texttt{-}\; e_2 & \Longrightarrow & \mathscr{C}(e_2)\;\; \mathscr{C}(e_1)\;\; \texttt{SUB} \\
e_1 \;\texttt{*}\; e_2 & \Longrightarrow & \mathscr{C}(e_2)\;\; \mathscr{C}(e_1)\;\; \texttt{MUL} \\
e_1 \;\texttt{/}\; e_2 & \Longrightarrow & \mathscr{C}(e_2)\;\; \mathscr{C}(e_1)\;\; \texttt{DIV}
\end{array}
$$

We need a procedure $\mathscr{C}$ for converting an arithmetic expression into a stack program. *Note the order!*

Example (Compilation)

4 * (2 - 3)

C(2 - 3) ⟶ C(3) ⟶ PUSH 3
C(2) ⟶ PUSH 2
SUB

C(4) ⟶ PUSH 4

MUL

PUSH 3
PUSH 2
SUB
PUSH 4
MUL

demo

(compiling arithmetic expressions)

Variables

Variables (Syntax)

$$
\begin{aligned}
\text{<prog>} &::= \{\text{<com>}\} \\
\text{<com>} &::= \texttt{ADD} \mid \texttt{SUB} \mid \texttt{MUL} \mid \texttt{DIV} \mid \texttt{PUSH}\ \text{<num>} \\
&\mid \quad \texttt{ASSIGN}\ \text{<var>} \mid \texttt{LOOKUP}\ \text{<var>} \\
\text{<num>} &::= \mathbb{Z} \\
\text{<var>} &::= \mathbb{I}
\end{aligned}
$$

Variables (Semantics)

$$\langle \mathscr{S}, \mathscr{E}, P \rangle$$

A **value** is an integer ($\mathbb{Z}$)

A **configuration** is made up of a stack $\mathscr{S}$ of values, an environment $\mathscr{E}$ (mapping of identifiers to values), and a program P given by **<prog>**

Variables (Semantics)

$$\frac{}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \texttt{ADD}\ P\rangle \longrightarrow \langle (m+n) :: \mathcal{S}, \mathcal{E}, P\rangle}\texttt{(add)}$$

$$\frac{}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \texttt{SUB}\ P\rangle \longrightarrow \langle (m-n) :: \mathcal{S}, \mathcal{E}, P\rangle}\texttt{(sub)}$$

$$\frac{}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \texttt{MUL}\ P\rangle \longrightarrow \langle (m\times n) :: \mathcal{S}, \mathcal{E}, P\rangle}\texttt{(mul)}$$

$$\frac{n \neq 0}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \texttt{DIV}\ P\rangle \longrightarrow \langle (m/n) :: \mathcal{S}, \mathcal{E}, P\rangle}\texttt{(div)}$$

$$\frac{}{\langle \mathcal{S}, \mathcal{E}, \texttt{PUSH}\ n\ P\rangle \longrightarrow \langle n :: \mathcal{S}, \mathcal{E}, P\rangle}\texttt{(div)}$$

$$\frac{}{\langle n :: \mathcal{S}, \mathcal{E}, \texttt{ASSIGN}\ x\ P\rangle \longrightarrow \langle \mathcal{S}, \mathcal{E}[x \mapsto n], P\rangle}\texttt{(asn)}$$

$$\frac{}{\langle n :: \mathcal{S}, \mathcal{E}, \texttt{LOOKUP}\ x\ P\rangle \longrightarrow \langle \mathcal{E}(x) :: \mathcal{S}, \mathcal{E}, P\rangle}\texttt{(lkp)}$$

Variables (Semantics)

basically the same

$$\frac{}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \texttt{ADD}\ P\rangle \longrightarrow \langle (m+n) :: \mathcal{S}, \mathcal{E}, P\rangle}\ \texttt{(add)}$$

$$\frac{}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \texttt{SUB}\ P\rangle \longrightarrow \langle (m-n) :: \mathcal{S}, \mathcal{E}, P\rangle}\ \texttt{(sub)}$$

$$\frac{}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \texttt{MUL}\ P\rangle \longrightarrow \langle (m\times n) :: \mathcal{S}, \mathcal{E}, P\rangle}\ \texttt{(mul)}$$

$$\frac{n \neq 0}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \texttt{DIV}\ P\rangle \longrightarrow \langle (m/n) :: \mathcal{S}, \mathcal{E}, P\rangle}\ \texttt{(div)}$$

$$\frac{}{\langle \mathcal{S}, \mathcal{E}, \texttt{PUSH}\ n\ P\rangle \longrightarrow \langle n :: \mathcal{S}, \mathcal{E}, P\rangle}\ \texttt{(div)}$$

$$\frac{}{\langle n :: \mathcal{S}, \mathcal{E}, \texttt{ASSIGN}\ x\ P\rangle \longrightarrow \langle \mathcal{S}, \mathcal{E}[x \mapsto n], P\rangle}\ \texttt{(asn)}$$

$$\frac{}{\langle n :: \mathcal{S}, \mathcal{E}, \texttt{LOOKUP}\ x\ P\rangle \longrightarrow \langle \mathcal{E}(x) :: \mathcal{S}, \mathcal{E}, P\rangle}\ \texttt{(lkp)}$$

Variables (Semantics)

basically the same

$$\frac{}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \texttt{ADD}\ P\rangle \longrightarrow \langle (m+n) :: \mathcal{S}, \mathcal{E}, P\rangle}\ \texttt{(add)}$$

$$\frac{}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \texttt{SUB}\ P\rangle \longrightarrow \langle (m-n) :: \mathcal{S}, \mathcal{E}, P\rangle}\ \texttt{(sub)}$$

$$\frac{}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \texttt{MUL}\ P\rangle \longrightarrow \langle (m\times n) :: \mathcal{S}, \mathcal{E}, P\rangle}\ \texttt{(mul)}$$

$$\frac{n \neq 0}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \texttt{DIV}\ P\rangle \longrightarrow \langle (m/n) :: \mathcal{S}, \mathcal{E}, P\rangle}\ \texttt{(div)}$$

$$\frac{}{\langle \mathcal{S}, \mathcal{E}, \texttt{PUSH}\ n\ P\rangle \longrightarrow \langle n :: \mathcal{S}, \mathcal{E}, P\rangle}\ \texttt{(div)}$$

new rules

$$\frac{}{\langle n :: \mathcal{S}, \mathcal{E}, \texttt{ASSIGN}\ x\ P\rangle \longrightarrow \langle \mathcal{S}, \mathcal{E}[x \mapsto n], P\rangle}\ \texttt{(asn)}$$

$$\frac{}{\langle n :: \mathcal{S}, \mathcal{E}, \texttt{LOOKUP}\ x\ P\rangle \longrightarrow \langle \mathcal{E}(x) :: \mathcal{S}, \mathcal{E}, P\rangle}\ \texttt{(lkp)}$$

Example (Evaluation)

```
PUSH 2 ASSIGN x PUSH 3 ASSIGN y
LOOKUP x LOOKUP y ADD
```

Compiling Let-Expressions (Attempt)

$$\texttt{x} \Longrightarrow \texttt{LOOKUP}\ x$$

$$\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Longrightarrow \mathscr{C}(e_1)\ \texttt{ASSIGN}\ x\ \mathscr{C}(e_2)$$

Compiling Let-Expressions (Attempt)

$$\texttt{x} \Longrightarrow \texttt{LOOKUP}\ x$$

$$\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Longrightarrow \mathscr{C}(e_1)\ \texttt{ASSIGN}\ x\ \mathscr{C}(e_2)$$

Except this isn't quite right

Example

```
let y = 1 in
let x = let y = 2 in y in
y
```

Scoping

```
let y = 1 in
let x = let y = 2 in y in
y
```

The language we’ve just described is only good for compiling from languages with **dynamic scoping**

Next time. We'll add *closures* so that we can deal with lexical scoping (and functions)

Summary

Compilation is the process of translating a program in a source language into a program in a target language which preserves the semantics

Targeting a **virtual machine** can make the implementation of a language more portable and less complex

We'll need **closures** to deal with lexical scoping correctly