Bytecode Interpreters

Concepts of Programming Languages
Lecture 26

CAS CS 320

Outline

» Course Evaluations!

» Discuss **stack-based languages** and **stack machines**

» Look briefly at how to compile **variables** and **functions**

» Finish up the dang course

Course evaluations!

Recap

Recall: Abstract/Virtual Machines

Recall: Abstract/Virtual Machines

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

Recall: Abstract/Virtual Machines

A **virtual 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)

Recall: Benefits of Stack Machines

Recall: Benefits of Stack Machines

Simplicity: Stacks aren't too complicated

Recall: 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

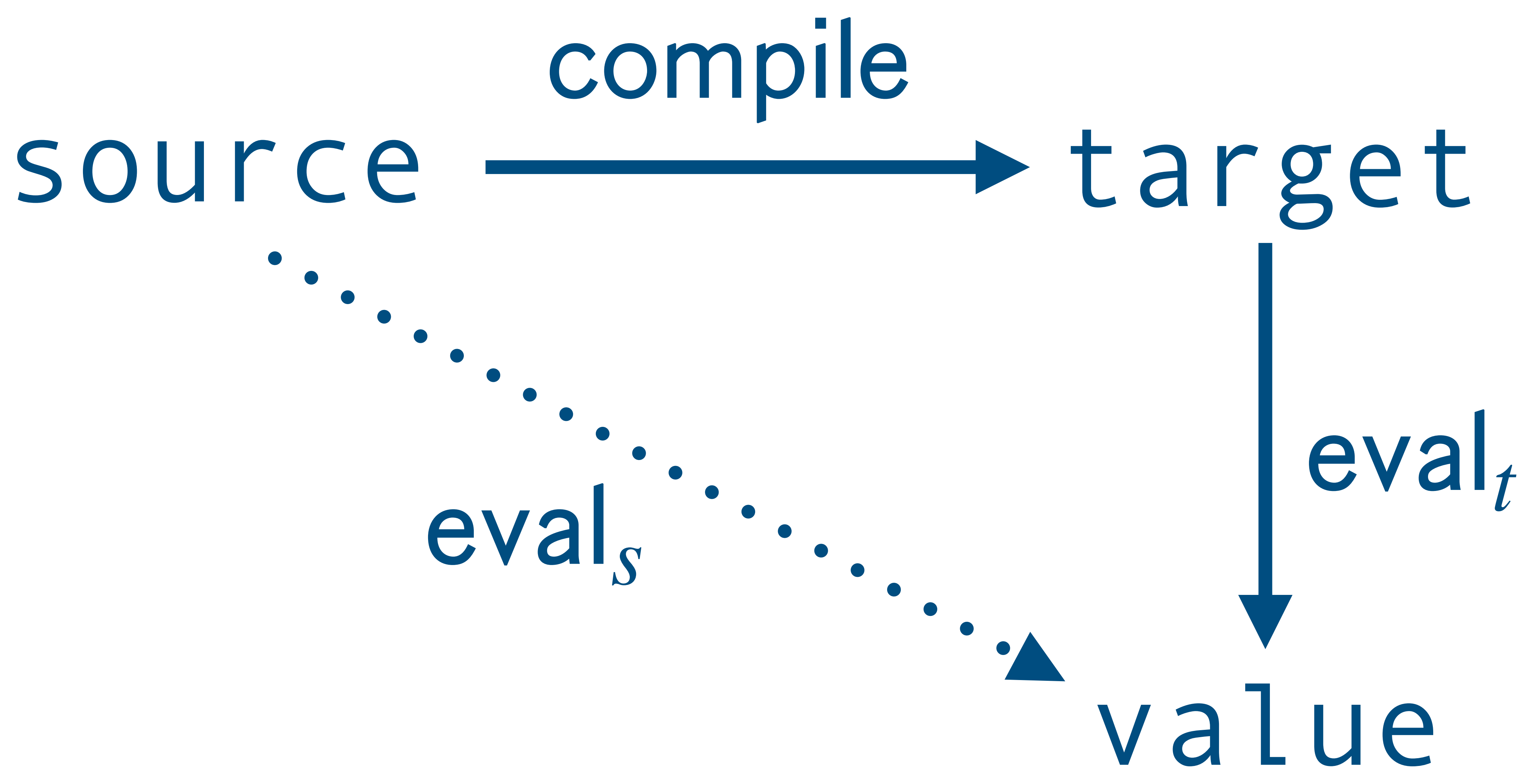
Recall: Benefits of Stack Machines

Simplicity: Stacks aren't too complicated

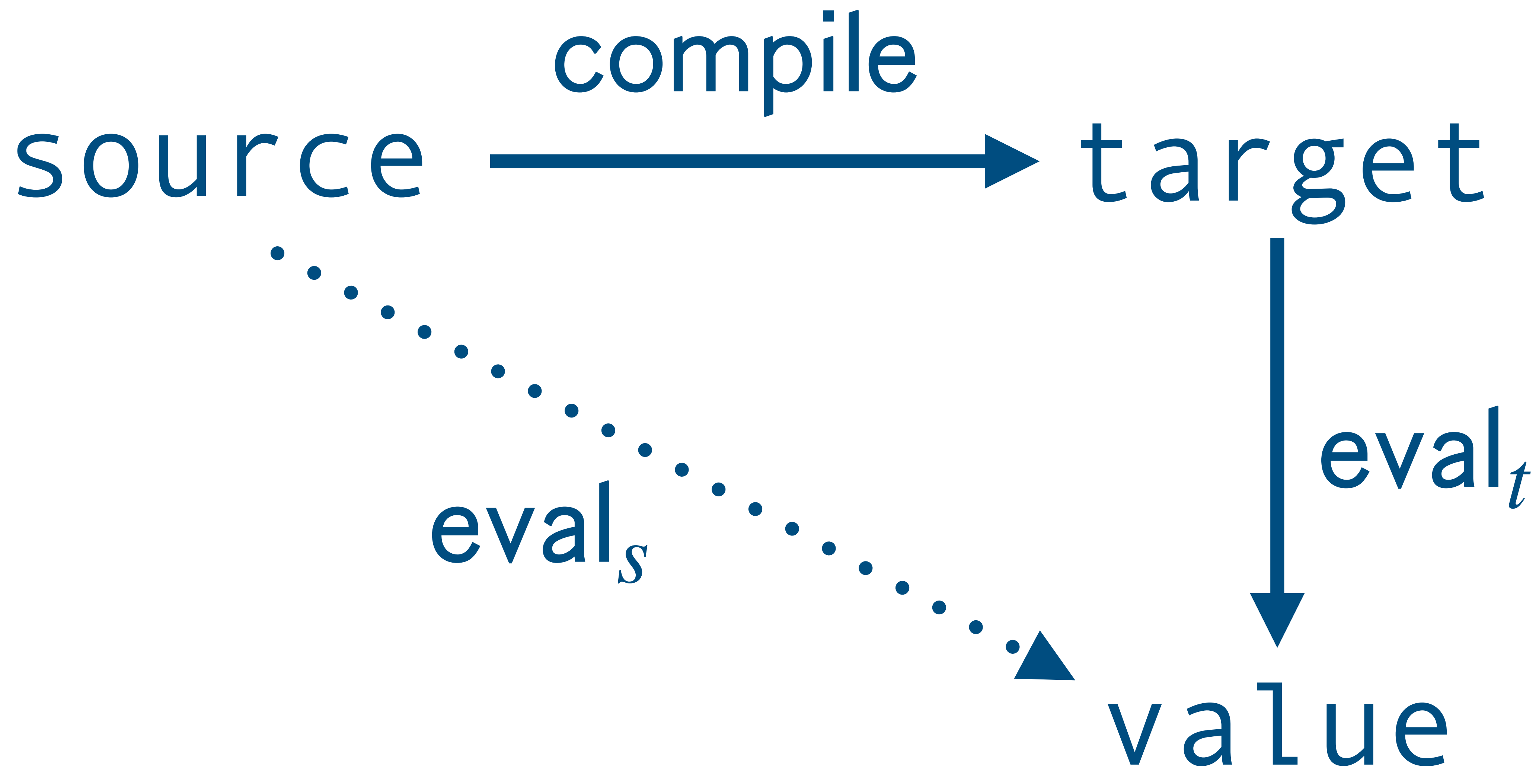
Portability: Any OS should be able to handle a stream of bytes, so the machine dependent part of our programming language can be simplified

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)

Recall: Compilation

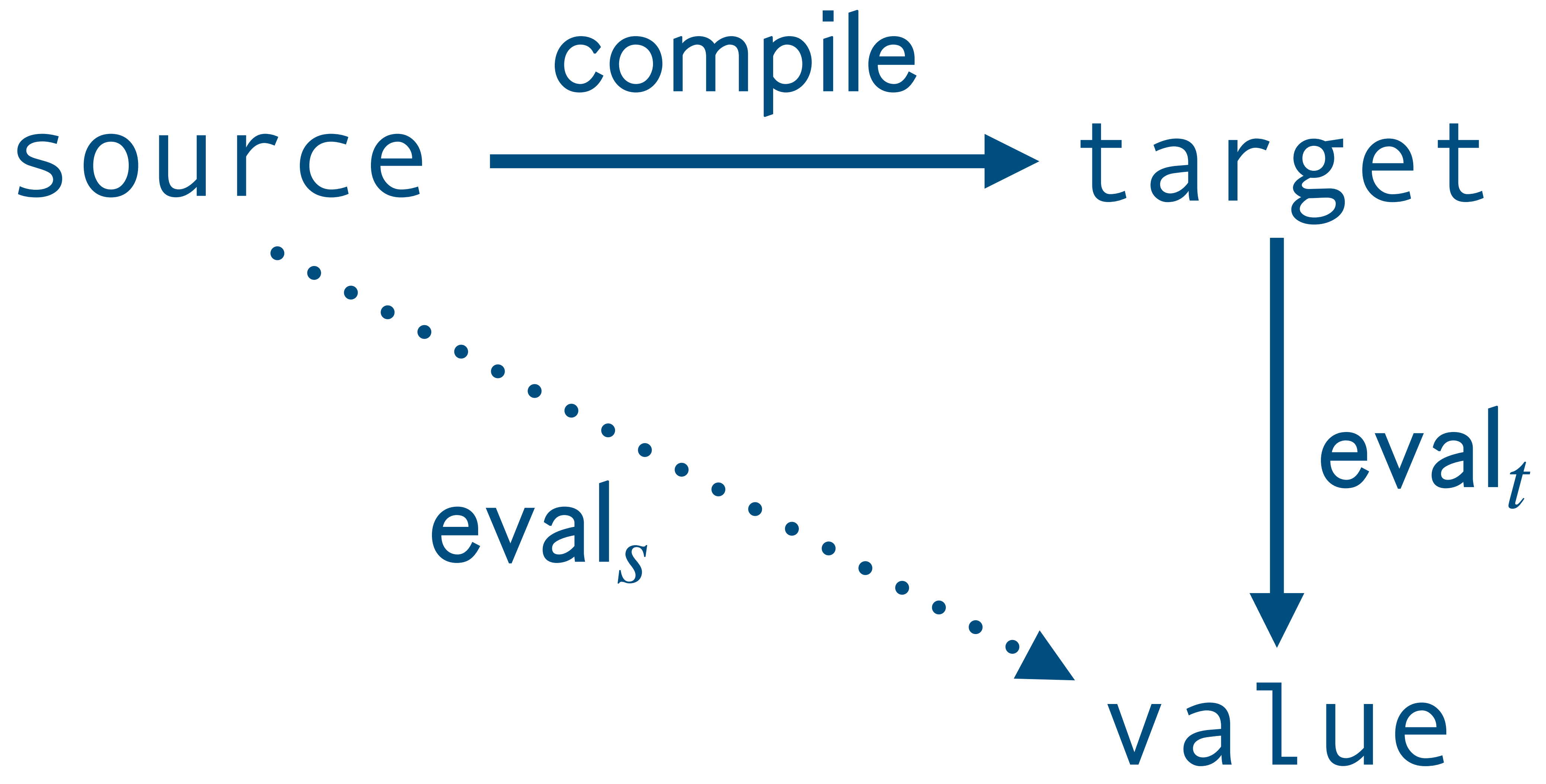


Recall: Compilation



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

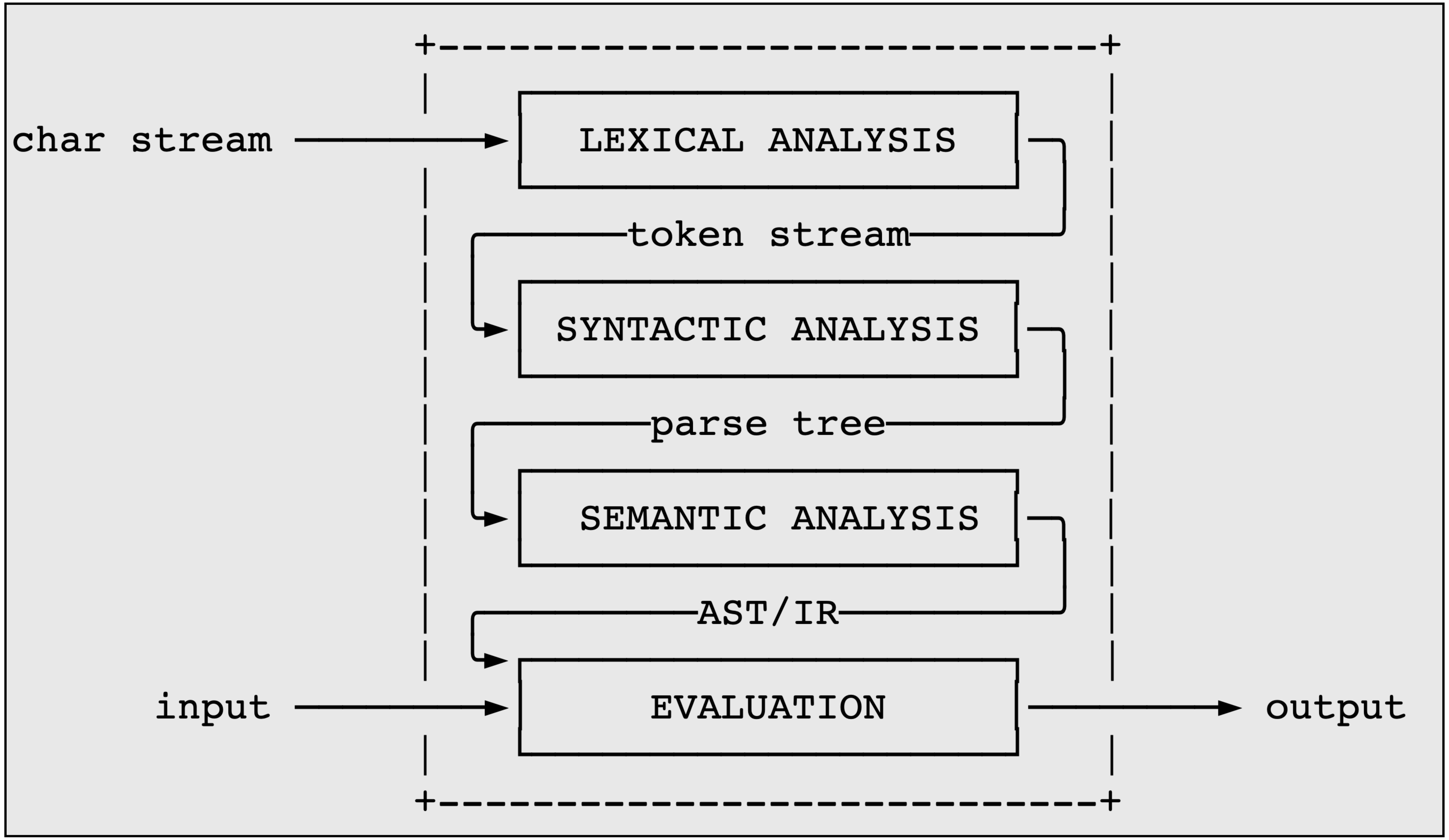
Recall: Compilation



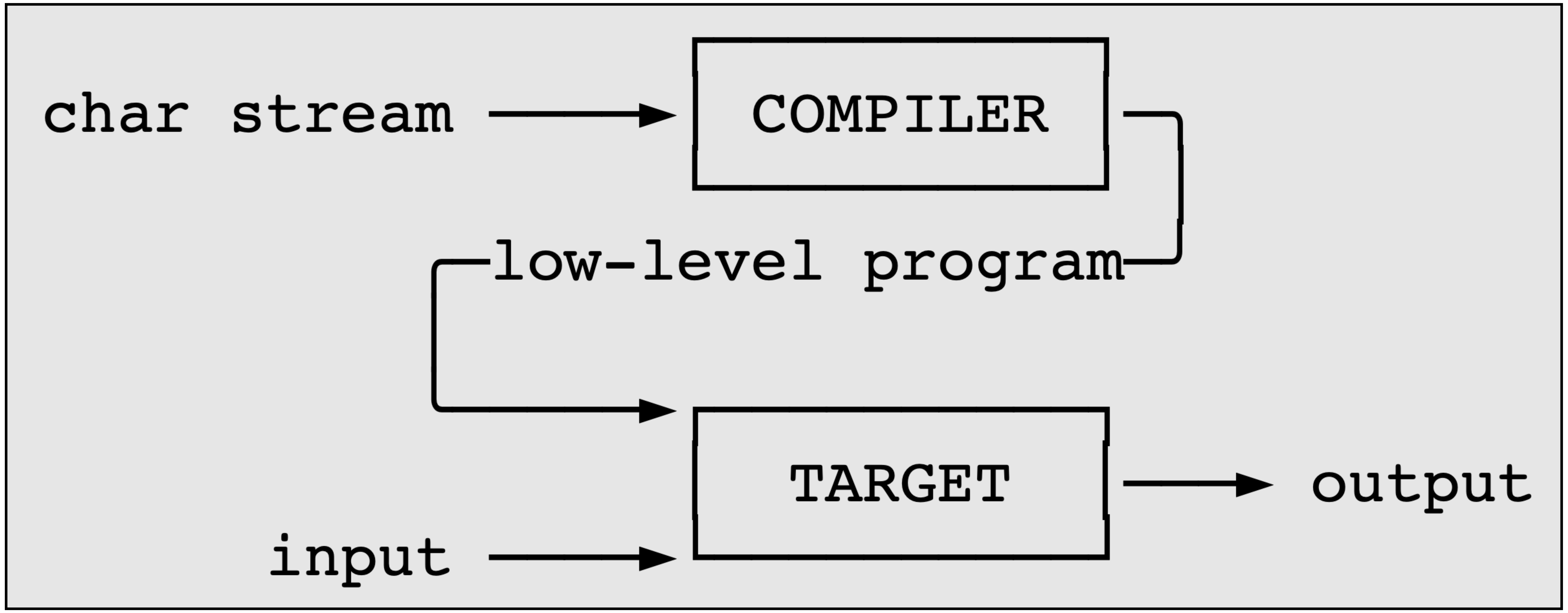
Compilation is the process of translating a program in one language to another, maintaining semantic behavior

Compilation can be a part of interpretation as well, like with **bytecode interpretation**

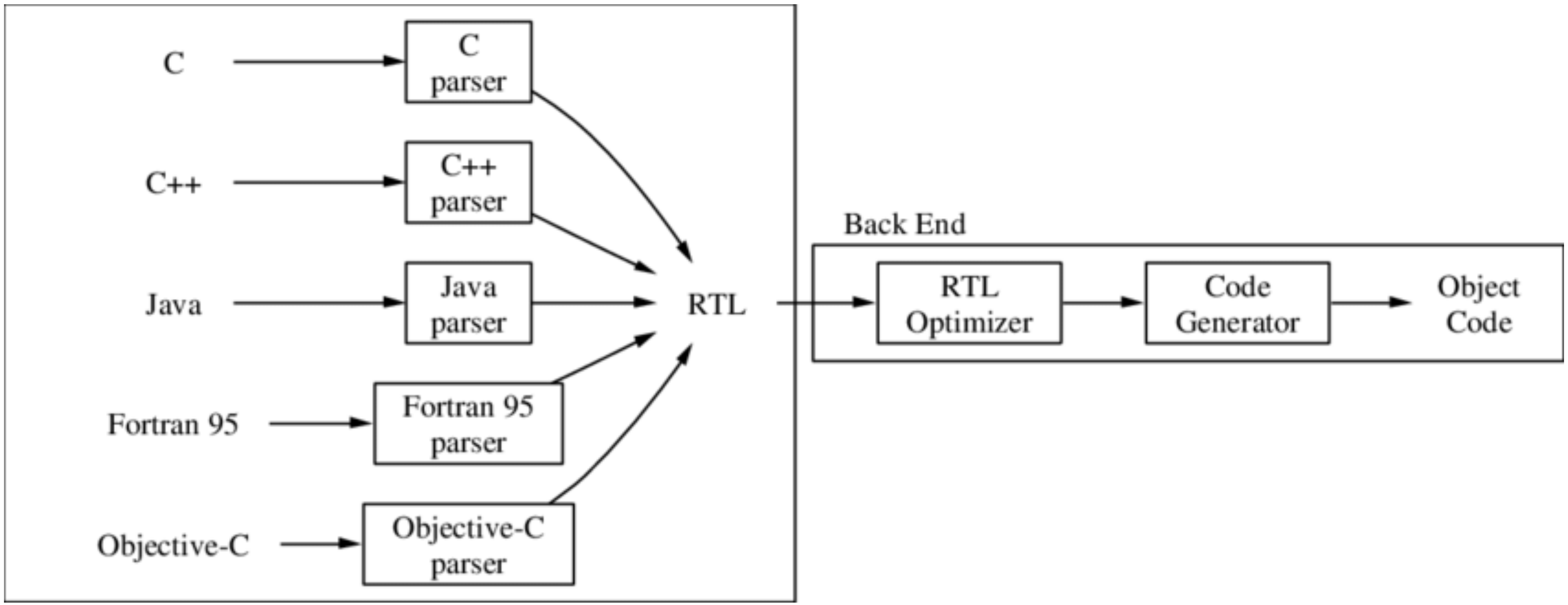
Recall: The Interpretation Pipeline



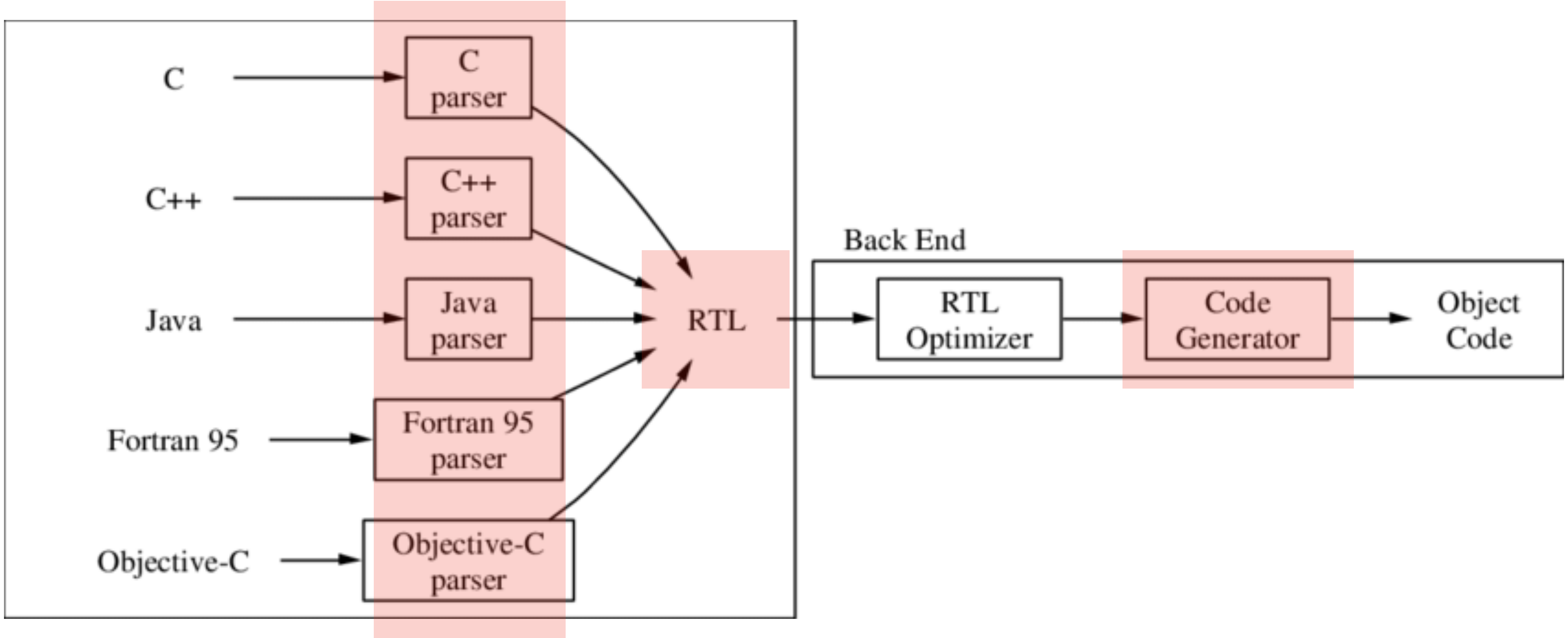
The Compiler Pipeline



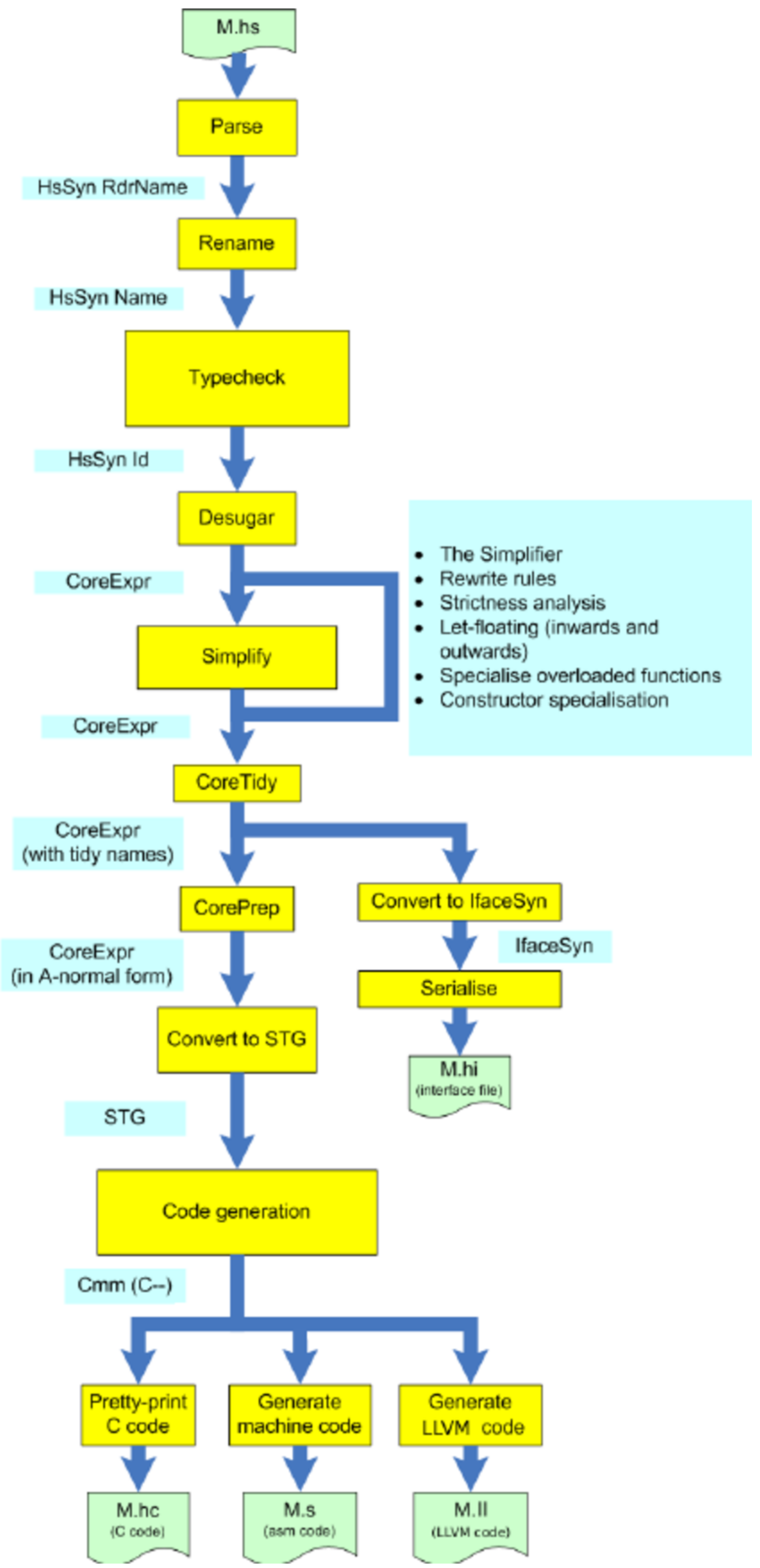
Example Pipelines: gcc



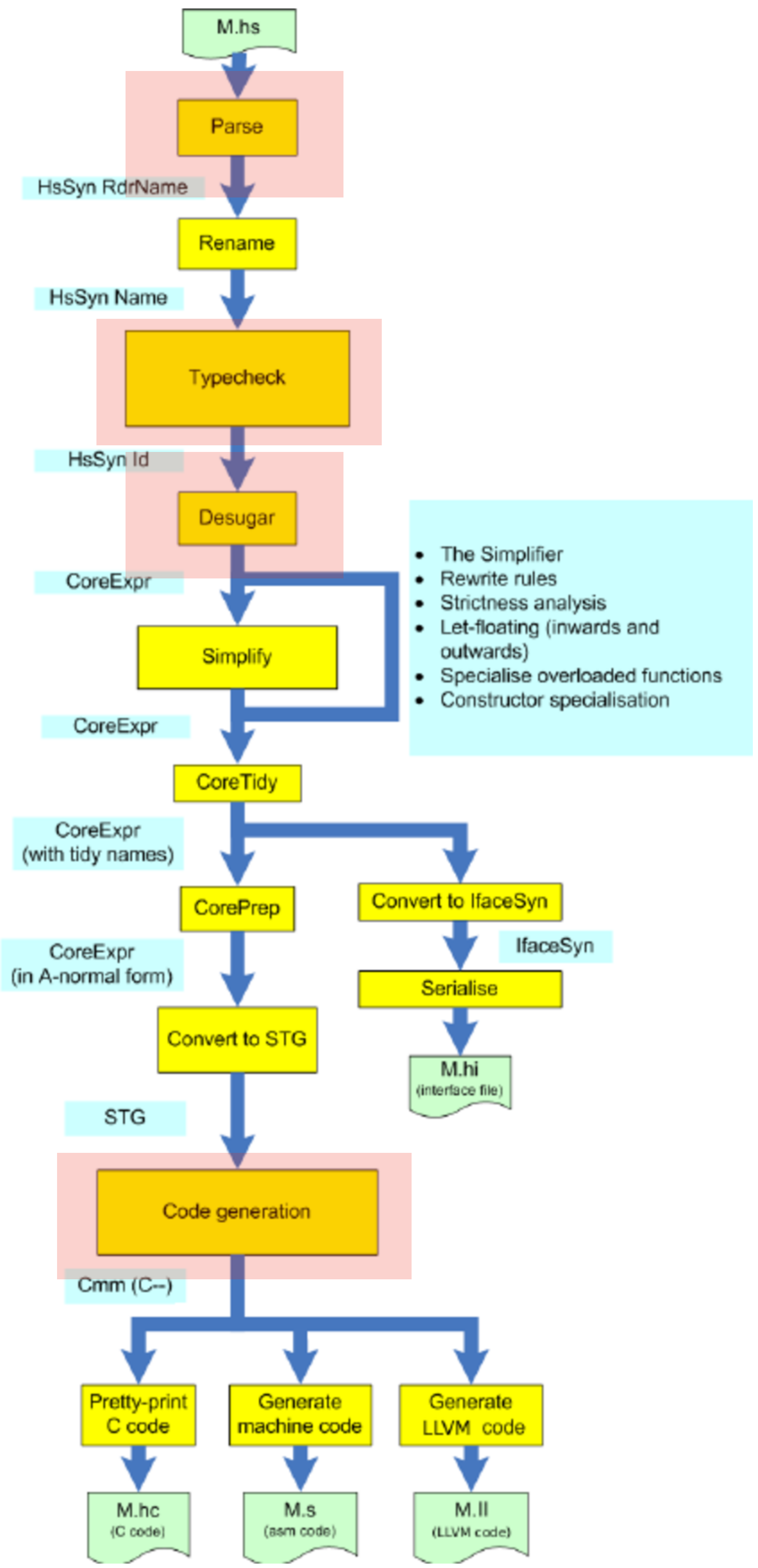
Example Pipelines: gcc



Example Pipelines: GHC (Haskell)



Example Pipelines: GHC (Haskell)



Example Pipelines: CompCert (C)

Example Pipelines: CompCert (C)

Stack-Based Arithmetic

Stack-Based Arithmetic (Syntax)

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

Stack-Based Arithmetic (Semantics)

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

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

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

Stack-Based Arithmetic (Semantics)

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

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

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

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

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

Example (Evaluation)

`PUSH 2 PUSH 3 SUB PUSH 4 MUL`

demo

(stack machine)

Compiling Arithmetic Expressions

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

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

Example (Compilation)

```
4 * (2 - 3)
```

demo

(compiling arithmetic expressions)

Variables

Variables (Syntax)

<prog> ::= {<com>}

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

| ASSIGN <var> | LOOKUP <var>

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

<var> ::= $\mathbb{I}$

Variables (Semantics)

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

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

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

Variables (Semantics)

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

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

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

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

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

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

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

Variables (Semantics)

basically the same

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

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

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

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

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

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

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

Variables (Semantics)

basically the same

$$\frac{}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \mathtt{ADD}\ P\rangle \longrightarrow \langle (m+n) :: \mathcal{S}, \mathcal{E}, P\rangle}\ (\mathtt{add}) \qquad \frac{}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \mathtt{SUB}\ P\rangle \longrightarrow \langle (m-n) :: \mathcal{S}, \mathcal{E}, P\rangle}\ (\mathtt{sub})$$

$$\frac{}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \mathtt{MUL}\ P\rangle \longrightarrow \langle (m\times n) :: \mathcal{S}, \mathcal{E}, P\rangle}\ (\mathtt{mul}) \qquad \frac{n \neq 0}{\langle m :: n :: \mathcal{S}, \mathcal{E}, \mathtt{DIV}\ P\rangle \longrightarrow \langle (m/n) :: \mathcal{S}, \mathcal{E}, P\rangle}\ (\mathtt{div})$$

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

new rules

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

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)

$$\texttt{x} \implies \texttt{LOOKUP}\ x$$

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

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
```

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**

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**

We can use closures to deal with lexical scoping (and functions)!

Functions

The Rough Picture

```
let k = fun x -> fun y -> x in
let a = k 2 in
a 3
```

Compilation is just a big sequence of transformations

The Rough Picture

```
(fun k ->
  let a = k 2 in
  a 3)
(fun x -> fun y -> x)
```

We can simulate let expressions with functions!
(we did this in lab)

The Rough Picture

```
(fun k ->
  (fun a -> a 3)
  (k 2))
(fun x -> fun y -> x)
```

and again...

The Rough Picture

```
[(fun k ->
   (fun a -> a 3)
   (k 2))
 (fun x -> fun y -> x)]
```

that was just to make the next part more convenient...
think of [expr] as as compile(expr)

The Rough Picture

```
[(fun x -> fun y -> x)]
[(fun k -> (fun a -> a 3) (k 2))]
CALL
```

We introduce as CALL command to call functions
Note the order, function/argument will go on a stack!

The Rough Picture

```
FUN ? X
  [fun y -> x]
  RETURN
[(fun k -> (fun a -> a 3) (k 2))]
CALL
```

We introduce a FUN command to define functions and a RETURN command to return from functions

The Rough Picture

```
FUN ? X
  FUN ? Y
    [x]
    RETURN
  RETURN
[(fun k -> (fun a -> a 3) (k 2))]
CALL
```

and again...

The Rough Picture

```
FUN 4 X
  FUN 2 Y
    LOOKUP X
    RETURN
  RETURN
[(fun k -> (fun a -> a 3) (k 2))]
CALL
```

The familiar LOOKUP command...

And functions let us know how many commands they have

The Rough Picture

```
FUN 4 X
  FUN 2 Y
    LOOKUP X
    RETURN
  RETURN
FUN ? K
  [(fun a -> a 3) (k 2)]
  RETURN
CALL
```

and we can keep going...

The Rough Picture

```
FUN 4 X
  FUN 2 Y
    LOOKUP X
    RETURN
  RETURN
FUN ? K
  [k 2]
  [fun a -> a 3]
  CALL
  RETURN
CALL
```

The Rough Picture

```
FUN 4 X
  FUN 2 Y
    LOOKUP X
    RETURN
  RETURN
FUN ? K
  [2]
  [k]
  CALL
  [fun a -> a 3]
  CALL
  RETURN
CALL
```

The Rough Picture

```
FUN 4 X
  FUN 2 Y
    LOOKUP X
    RETURN
  RETURN
FUN ? K
  PUSH 2
  [k]
  CALL
  [fun a -> a 3]
  CALL
  RETURN
CALL
```

The Rough Picture

```
FUN 4 X
  FUN 2 Y
    LOOKUP X
    RETURN
  RETURN
FUN ? K
  PUSH 2
  LOOKUP K
  CALL
  [fun a -> a 3]
  CALL
  RETURN
CALL
```

The Rough Picture

```
FUN 4 X
  FUN 2 Y
    LOOKUP X
    RETURN
  RETURN
FUN ? K
  PUSH 2
  LOOKUP K
  CALL
  FUN ? A
    [a 3]
    RETURN
  CALL
  RETURN
CALL
```

The Rough Picture

```
FUN 4 X
  FUN 2 Y
    LOOKUP X
    RETURN
  RETURN
FUN ? K
  PUSH 2
  LOOKUP K
  CALL
  FUN ? A
    [3]
    [a]
    CALL
    RETURN
  CALL
  RETURN
CALL
```

The Rough Picture

```
FUN 4 X
  FUN 2 Y
    LOOKUP X
    RETURN
  RETURN
FUN ? K
  PUSH 2
  LOOKUP K
  CALL
  FUN ? A
    PUSH 3
    [a]
    CALL
    RETURN
  CALL
  RETURN
CALL
```

The Rough Picture

```
FUN 4 X
  FUN 2 Y
    LOOKUP X
    RETURN
  RETURN
FUN 10 K
  PUSH 2
  LOOKUP K
  CALL
  FUN 4 A
    PUSH 3
    LOOKUP A
    CALL
    RETURN
  CALL
  RETURN
CALL
```

The Rough Picture

```
let k = fun x -> fun y -> x in
let a = k 2 in
a 3
```

⟹

```
FUN 4 X
  FUN 2 Y
    LOOKUP X
    RETURN
  RETURN
FUN 10 K
  PUSH 2
  LOOKUP K
  CALL
  FUN 4 A
    PUSH 3
    LOOKUP A
    CALL
    RETURN
  CALL
  RETURN
CALL
```

Compilation is just a big sequence of transformations

The Rough Picture

```
let k = fun x -> fun y -> x in
let a = k 2 in
a 3
```

$\Longrightarrow$

```
7 4
7 2
10 1
9
9
7 10
0 2
10 0
8
7 4
0 3
10 0
8
9
8
9
8
```

Byte-code interpretation additionally maps each command to a byte value

Syntax

```
<prog> ::= { <com> }
<com>  ::= PUSH <int>
         | ADD | SUB | MUL | DIV
         | FUN <ident> <int> | CALL | RETURN
         | LOOKUP
```

Semantics (Configurations)

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

A **value** is an integer ($\mathbb{Z}$) or a closure ($\mathbb{C}$) of the form $(\mathscr{E}, x, P)$

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>**

Semantics (Functions)

$$\frac{}{\langle \mathcal{S}, \mathscr{E}, \mathsf{FUN}\ x\ n\ P \rangle \longrightarrow \langle (\mathscr{E}, x, P[1..n]) :: \mathcal{S}, \mathscr{E}'[x \mapsto v], P[n+1..] \rangle}\ \texttt{(fun)}$$

Function definitions carry a **parameter name** and an **offset**, which we use to construct the closure

Semantics (Continuation Passing)

$$\frac{}{\langle (\mathscr{E}', x, P') :: v :: \mathcal{S}, \mathscr{E}, \texttt{CALL}\ P \rangle \longrightarrow \langle (\mathscr{E}, \_, P) :: \mathcal{S}, \mathscr{E}'[x \mapsto v], P' \rangle}\ (\texttt{call})$$

$$\frac{}{\langle v :: (\mathscr{E}', \_, P') :: \mathcal{S}, \mathscr{E}, \texttt{RETURN}\ P \rangle \longrightarrow \langle v :: \mathcal{S}, \mathscr{E}', P' \rangle}\ (\texttt{ret})$$

`One challenge:` `when we call a function, where to we "return" to?`

`Answer:` `We put the information on the stack itself in a closure!`

Example

Stack:

Env:

```
FUN 4 X
  FUN 2 Y
    LOOKUP X
    RETURN
  RETURN
FUN 10 K
  PUSH 2
  LOOKUP K
  CALL
  FUN 4 A
    PUSH 3
    LOOKUP A
    CALL
    RETURN
  CALL
  RETURN
CALL
```

Example

Stack:

```
Ø
X
FUN 2 Y
  LOOKUP X
  RETURN
RETURN
```

Env:

```
FUN 10 K
  PUSH 2
  LOOKUP K
  CALL
  FUN 4 A
    PUSH 3
    LOOKUP A
    CALL
    RETURN
  CALL
  RETURN
CALL
```

Example

Stack:

```
Ø
X
FUN 2 Y
  LOOKUP X
  RETURN
RETURN
```

```
Ø
K
PUSH 2
LOOKUP K
CALL
FUN 4 A
  PUSH 3
  LOOKUP A
  CALL
  RETURN
CALL
RETURN
```

Env:

CALL

Example

Stack:

Ø
—
ϵ

Env:

K ↦

Ø
X

FUN 2 Y
  LOOKUP X
  RETURN
RETURN

```
PUSH 2
LOOKUP K
CALL
FUN 4 A
  PUSH 3
  LOOKUP A
  CALL
  RETURN
CALL
RETURN
```

Example

Stack:

Ø – ϵ
2

Env:

K $\mapsto$

```
Ø
X
FUN 2 Y
  LOOKUP X
  RETURN
RETURN
```

```
LOOKUP K
CALL
FUN 4 A
  PUSH 3
  LOOKUP A
  CALL
  RETURN
CALL
RETURN
```

Example

<u>Stack:</u>

$\boxed{\begin{matrix}\emptyset \\ \_ \\ \epsilon\end{matrix}}$

$\boxed{2}$

```
Ø
X
FUN 2 Y
  LOOKUP X
  RETURN
RETURN
```

<u>Env:</u>

K $\mapsto$

```
Ø
X
FUN 2 Y
  LOOKUP X
  RETURN
RETURN
```

```
CALL
FUN 4 A
  PUSH 3
  LOOKUP A
  CALL
  RETURN
CALL
RETURN
```

Example

Stack:

$\boxed{\begin{array}{l} \varnothing \\ \_ \\ \epsilon \end{array}}$

```
K ↦ {...}
_
FUN 4 A
  PUSH 3
  LOOKUP A
  CALL
  RETURN
CALL
RETURN
```

Env:

$X \mapsto \boxed{2}$

```
FUN 2 Y
  LOOKUP X
  RETURN
RETURN
```

Example

Stack:

∅
_
ϵ

```
K ↦ {...}
_
FUN 4 A
  PUSH 3
  LOOKUP A
  CALL
  RETURN
CALL
RETURN
```

```
X ↦ 2
Y
LOOKUP X
RETURN
```

Env:

X ↦ $\boxed{2}$

RETURN

Example

Stack:

$\varnothing$
_
ϵ

$X \mapsto 2$
Y
LOOKUP X
RETURN

Env:

K $\mapsto$

```
∅
X
FUN 2 Y
  LOOKUP X
  RETURN
RETURN
```

```
FUN 4 A
  PUSH 3
  LOOKUP A
  CALL
  RETURN
CALL
RETURN
```

Example

Stack:

$\emptyset$
_
ϵ

X ↦ 2
Y
LOOKUP X
RETURN

K ↦ {...}
A
PUSH 3
LOOKUP A
CALL
RETURN

Env:

K ↦

$\emptyset$
X
FUN 2 Y
LOOKUP X
RETURN
RETURN

```
CALL
RETURN
```

Example

Stack:

```
∅
_
ε
```

```
K ↦ {...}
_
RETURN
```

Env:

```
K ↦ ∅
    X
    FUN 2 Y
      LOOKUP X
      RETURN
    RETURN
```

```
A ↦ X ↦ 2
    Y
    LOOKUP X
    RETURN
```

```
PUSH 3
LOOKUP A
CALL
RETURN
```

Example

Stack:

Ø
—
ϵ

K ↦ {...}
—
RETURN

3

Env:

K ↦

```
Ø
X
FUN 2 Y
  LOOKUP X
  RETURN
RETURN
```

A ↦

```
X ↦ 2
Y
LOOKUP X
RETURN
```

```
LOOKUP A
CALL
RETURN
```

Example

Stack:

∅
—
ϵ

K ↦ {...}
—
RETURN

3

X ↦ 2
Y
LOOKUP X
RETURN

Env:

K ↦

∅
X
FUN 2 Y
  LOOKUP X
  RETURN
RETURN

A ↦

X ↦ 2
Y
LOOKUP X
RETURN

CALL
RETURN

Example

Stack:

∅
—
ϵ

K ↦ {...}
—
RETURN

K ↦ {...}
A ↦ {...}
—
RETURN

Env:

X ↦ $\boxed{2}$

Y ↦ $\boxed{3}$

```
LOOKUP X
RETURN
```

Example

Stack:

$\boxed{\begin{array}{l} \varnothing \\ \_ \\ \epsilon \end{array}}$

$\boxed{\begin{array}{l} \mathtt{K} \mapsto \{...\} \\ \_ \\ \mathtt{RETURN} \end{array}}$

$\boxed{\begin{array}{l} \mathtt{K} \mapsto \{...\} \\ \mathtt{A} \mapsto \{...\} \\ \_ \\ \mathtt{RETURN} \end{array}}$

$\boxed{2}$

Env:

$X \mapsto \boxed{2}$

$Y \mapsto \boxed{3}$

RETURN

Example

Stack:

∅
_
ϵ

K ↦ {...}
_
RETURN

2

Env:

K ↦

```
∅
X
FUN 2 Y
  LOOKUP X
  RETURN
RETURN
```

A ↦

```
X ↦ 2
Y
LOOKUP X
RETURN
```

RETURN

Example

Stack:

∅
_
ϵ

2

Env:

K $\mapsto$

∅
X
FUN 2 Y
LOOKUP X
RETURN
RETURN

RETURN

Example

Stack:

$\boxed{2}$

Env:

ϵ

demo

(show-and-tell)

What's next?

More OCaml:

» Modules, functional data structures, mutability

» GADTS, effects, parallelism

» applications in ML, linear algebra, scientific computing

More PL:

» Come learn Rust (and linear types) with me next semester!

» Learn Haskell, Elm, Scala

More Math/Type Theory:

» Go learn about session types with Professor Das next semester!

» Category theory (functors, monads, comonads), Logic, Type theory

More Computers:

» Compilers, Linkers, LLVM

» Formal verification

» embedded systems programming, tensor program compilation

fin

(thanks everyone)