Collapsing Heterogenous Towers of Evaluators

Working Group on Functional Programming

Michael Buch, **Nada Amin**, Tiark Rompf

Collapsing Towers of Interpreters

NADA AMIN, University of Cambridge, UK
TIARK ROMPF, Purdue University, USA

Given a tower of interpreters, i.e., a sequence of multiple interpreters interpreting one another as input programs, we aim to collapse this tower into a compiler that removes all interpretive overhead and runs in a single pass. In the real world, a use case might be Python code executed by an x86 runtime, on a CPU emulated in a JavaScript VM, running on an ARM CPU. Collapsing such a tower can not only exponentially improve runtime performance, but also enable the use of base-language tools for interpreted programs, e.g., for analysis and verification. In this paper, we lay the foundations in an idealized but realistic setting.

We present a multi-level lambda calculus that features *staging constructs* and *stage polymorphism*: based on runtime parameters, an evaluator either executes source code (thereby acting as an interpreter) or generates code (thereby acting as a compiler). We identify stage polymorphism, a programming model from the domain of high-performance program generators, as the key mechanism to make such interpreters compose in a collapsible way.

We present Pink, a meta-circular Lisp-like evaluator on top of this calculus, and demonstrate that we can collapse arbitrarily many levels of self-interpretation, including levels with semantic modifications. We discuss several examples: compiling regular expressions through an interpreter to base code, building program transformers from modified interpreters, and others. We develop these ideas further to include reflection and reification, culminating in Purple, a reflective language inspired by Brown, Blond, and Black, which realizes a conceptually infinite tower, where every aspect of the semantics can change dynamically. Addressing an open challenge, we show how user programs can be compiled and recompiled under user-modified semantics.

CCS Concepts: • **Software and its engineering → Compilers**; *Interpreters*; *General programming languages*;

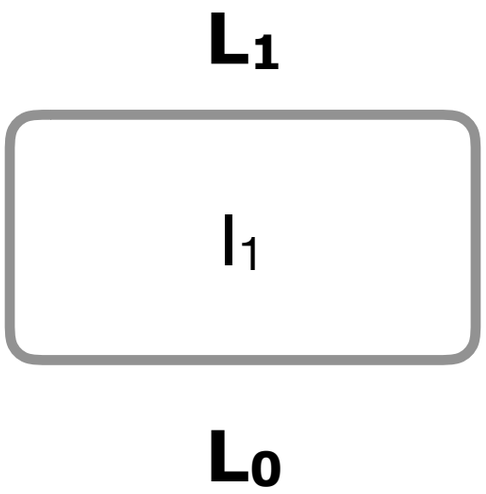
Additional Key Words and Phrases: interpreter, compiler, staging, reflection, Scala, Lisp

ACM Reference Format:
Nada Amin and Tiark Rompf. 2018. Collapsing Towers of Interpreters. *Proc. ACM Program. Lang.* 2, POPL, Article 52 (January 2018), 33 pages. https://doi.org/10.1145/3158140

Overview

- *Collapsing Towers of Interpreters* (POPL 2018)
  focuses on reflective towers of meta-circular interpreters.

- What about heterogeneity?


L1
l1
L0

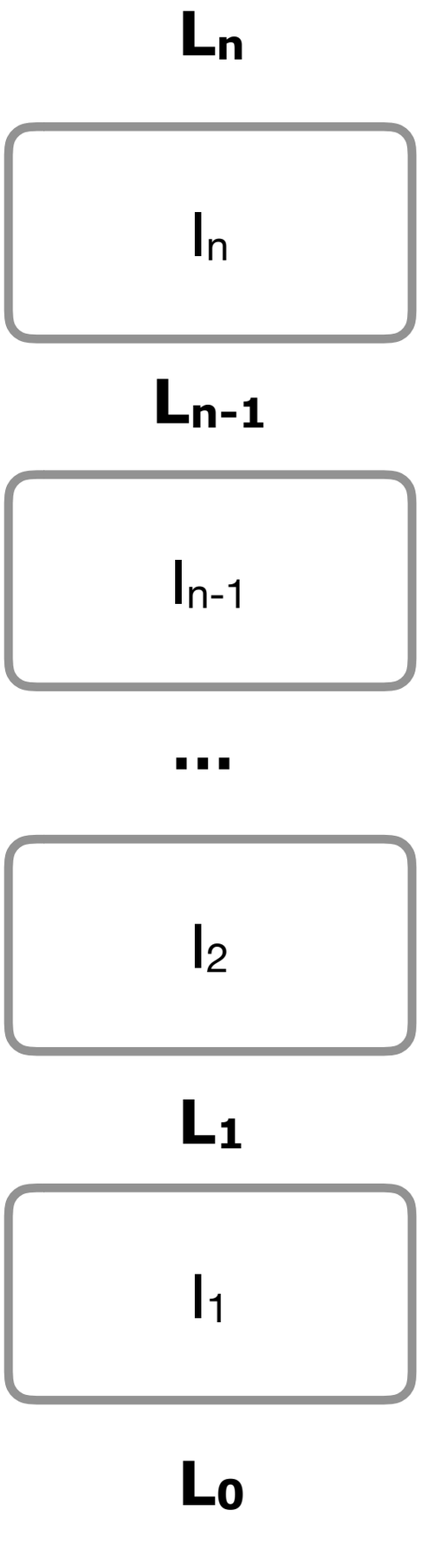
L_2

l_2

L_1

l_1

L_0

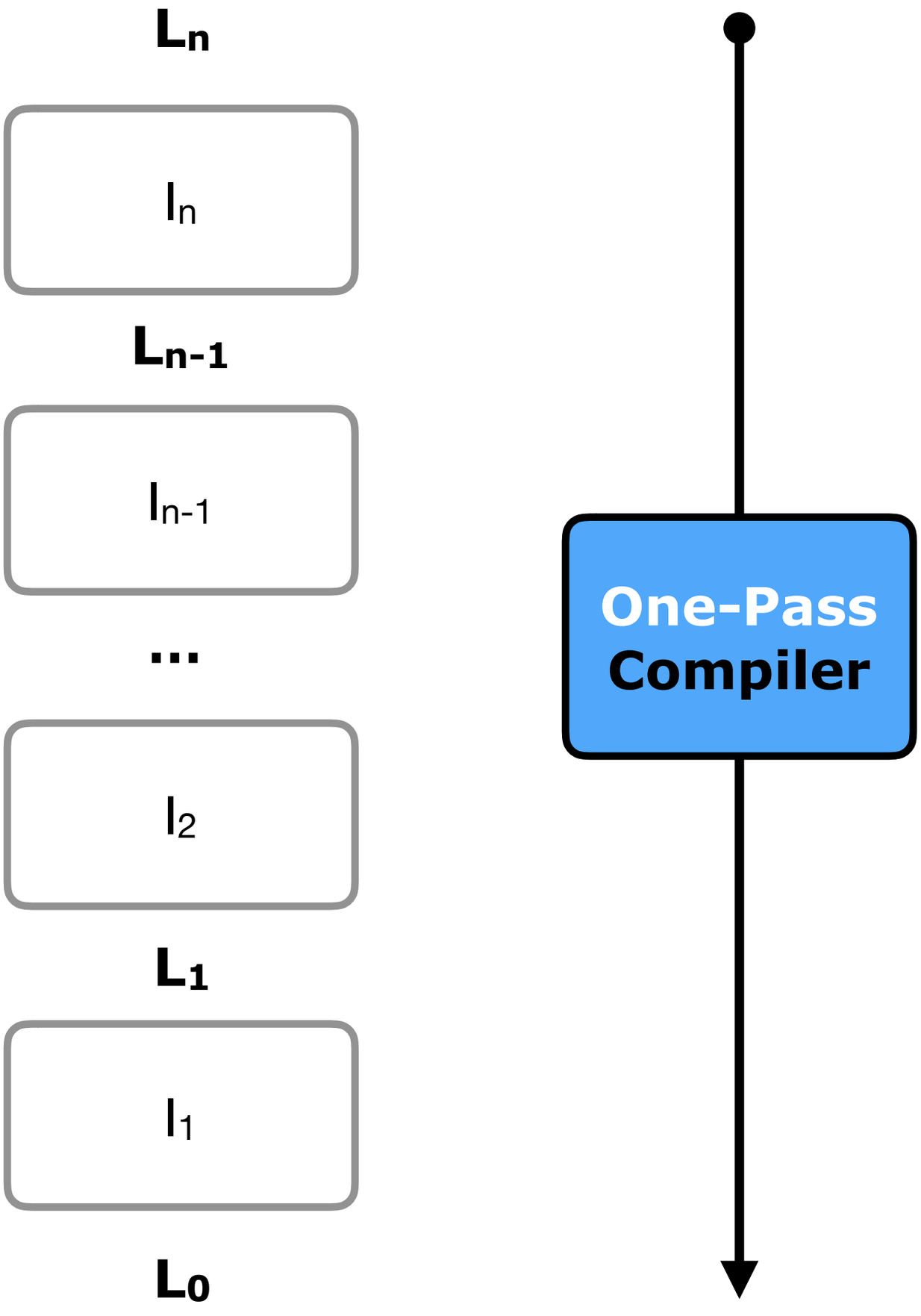


Languages

$L_0, \ldots, L_n$

Interpreters

I_{i+1} for L_{i+1}, written in L_i



Languages

$L_0, \ldots, L_n$

Interpreters

I_{i+1} for L_{i+1}, written in L_i

Python

I_n

Bytecode

I_{n-1}

x86 runtime

I_2

JavaScript VM

I_1

ARM CPU

Python

I_n

Bytecode

I_{n-1}

x86 runtime

I_2

JavaScript VM

I_1

ARM CPU

One-Pass Compiler

Python

I_n

Bytecode

I_{n-1}

x86 runtime

I_2

JavaScript VM

I_1

ARM CPU

One-Pass Compiler

L_n

l_n

L_{n-1}

l_{n-1}

...

l_2

L_1

l_1

L_0

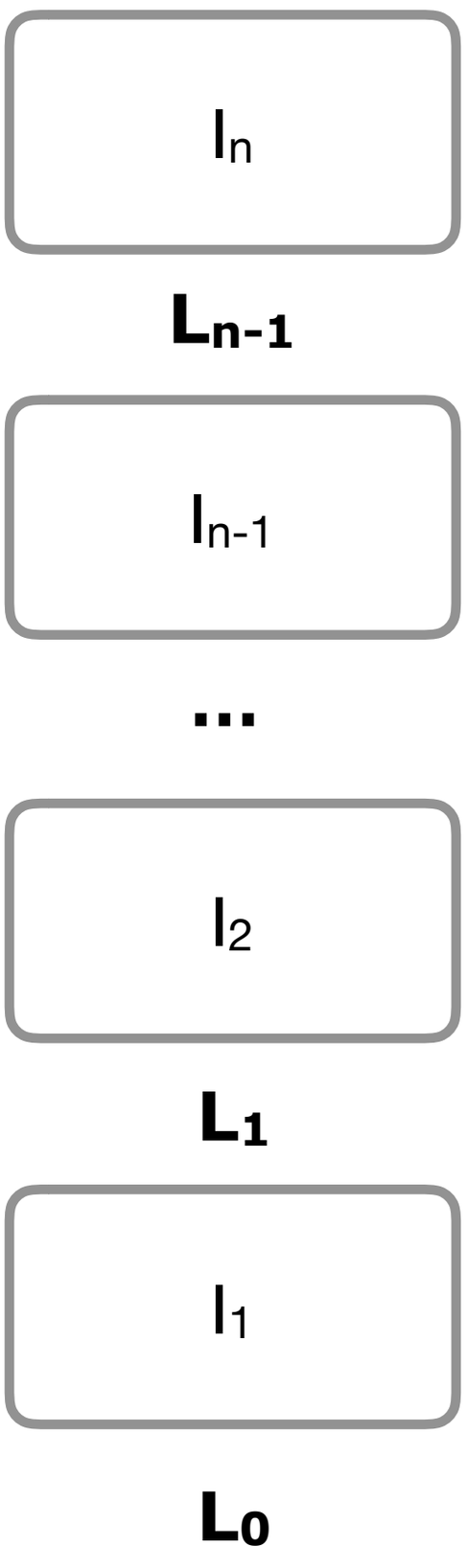
base L_0 = variant of **λ-calculus**

~ conceptually **infinite**

~ **reflective**

can be inspected and modified
at runtime

base L_0 = variant of **λ-calculus**



~ **heterogenous**:
different semantics
and representations
at each level

base L_0 = variant of **λ-calculus**

Program

Python

Python → C

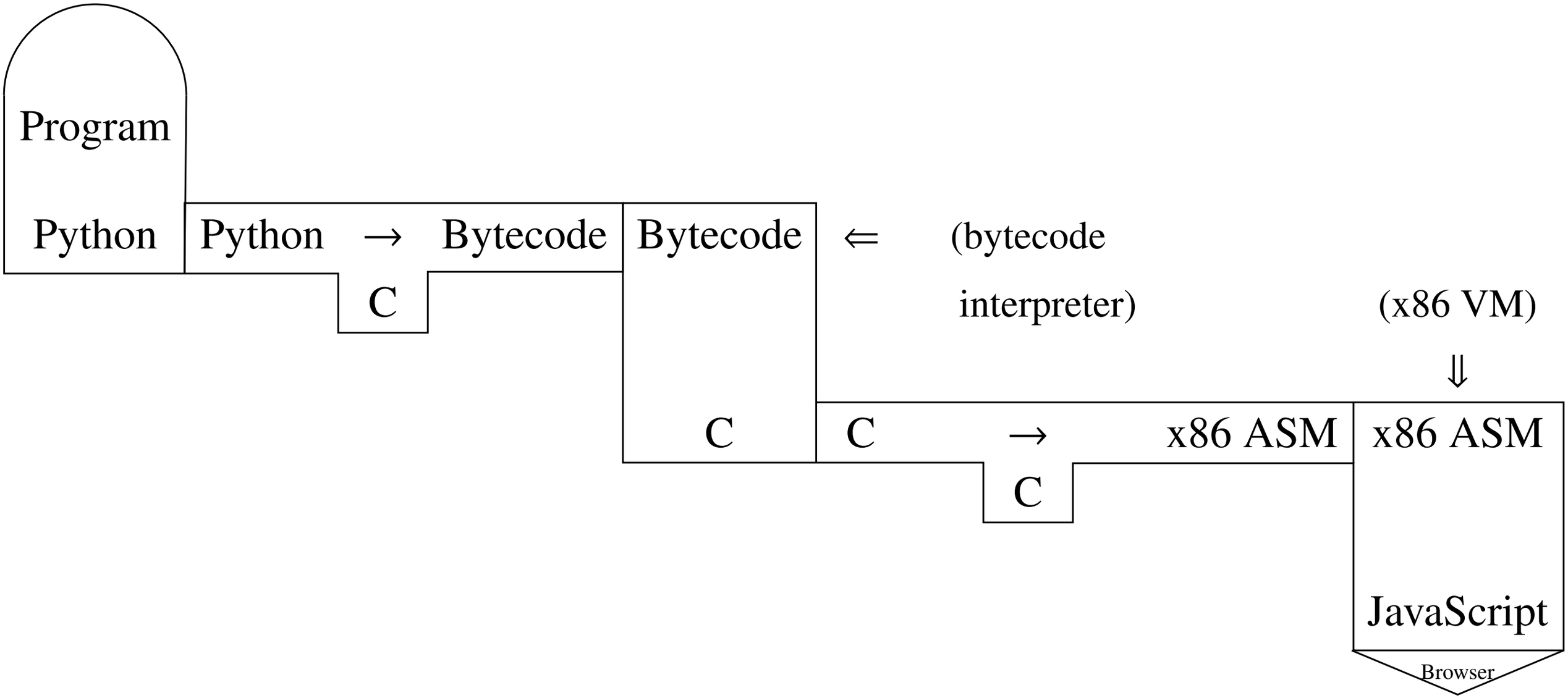
Python → x86 ASM

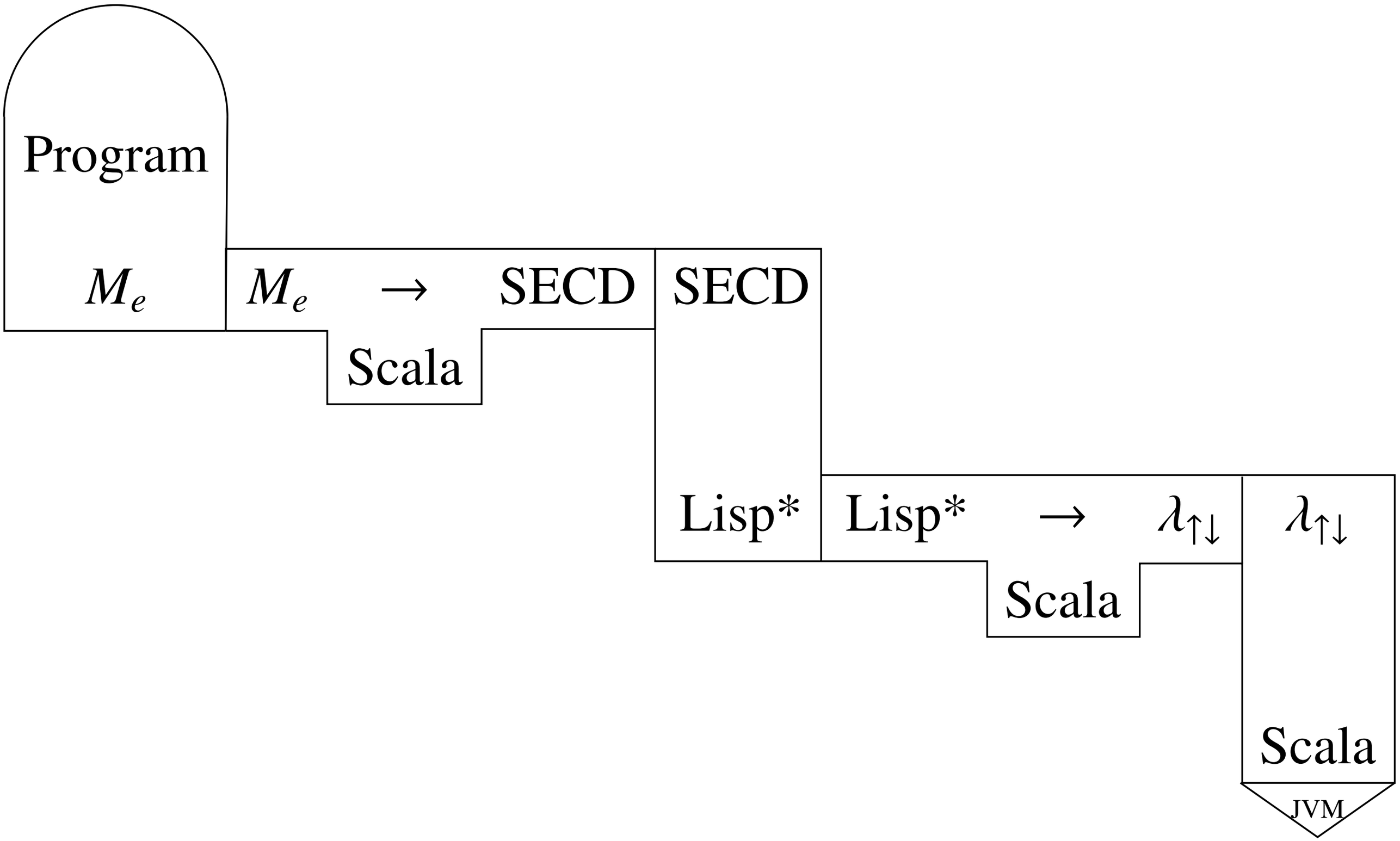
x86 ASM

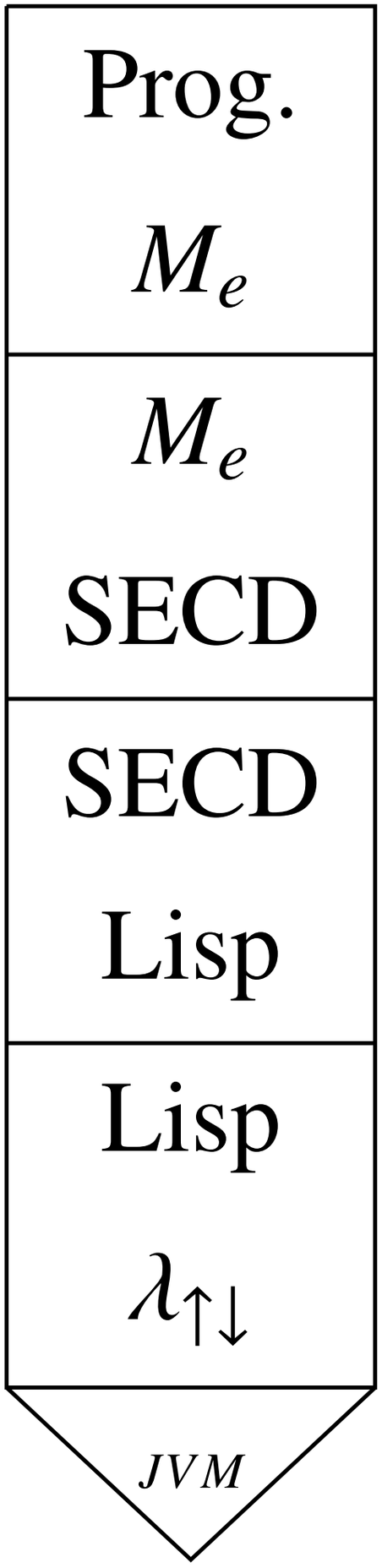
x86 Emulator

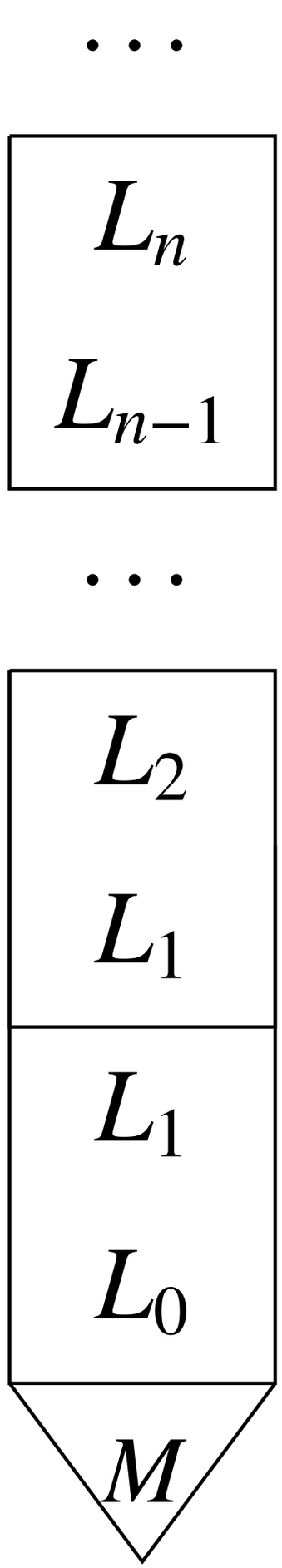
JavaScript

Browser

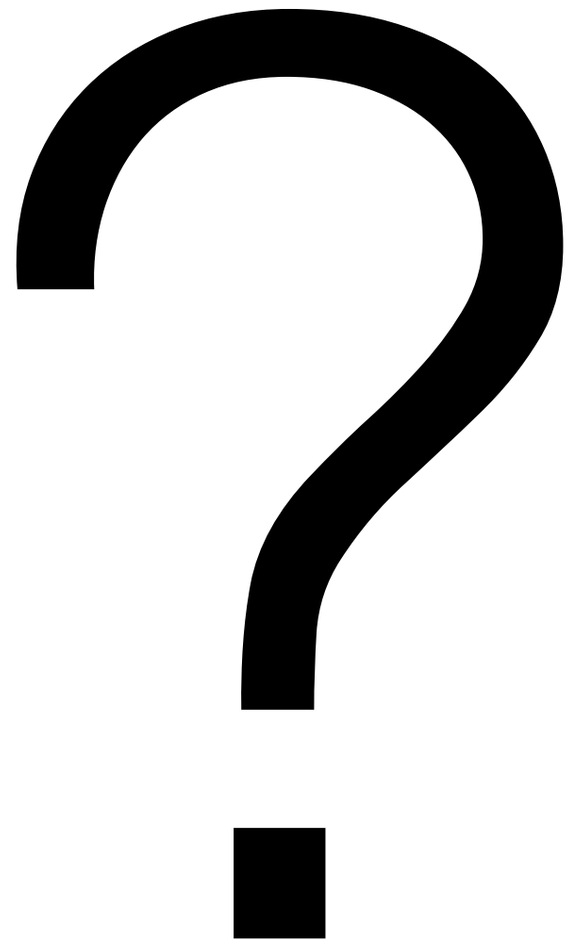

Program
Python
Python → Bytecode
C
Bytecode
C
⇐
(bytecode
interpreter)
(x86 VM)
⇓
C → x86 ASM
C
x86 ASM
JavaScript
Browser


Program
M_e
M_e → SECD
Scala
SECD
Lisp*
Lisp* → $\lambda_{\uparrow\downarrow}$
Scala
$\lambda_{\uparrow\downarrow}$
Scala
JVM

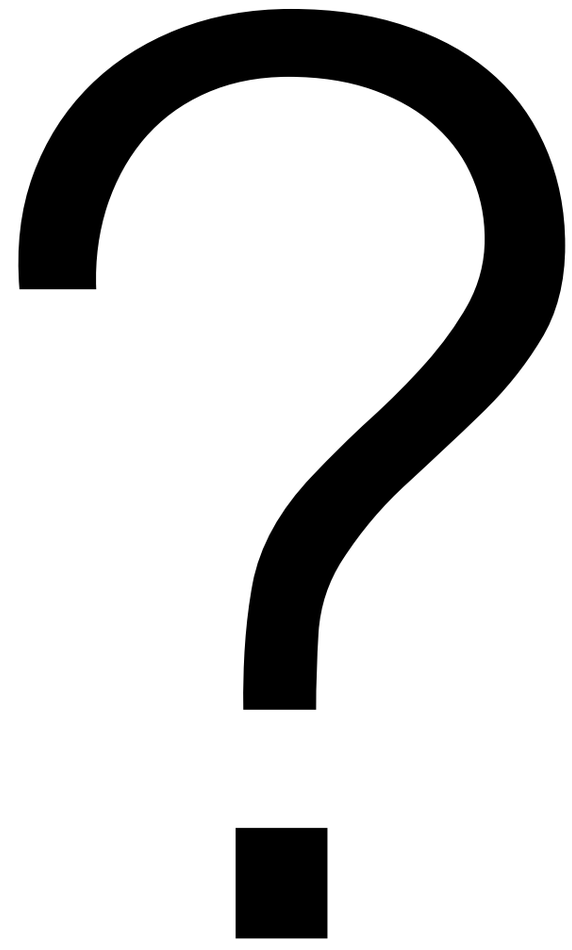

Prog.
M_e
M_e
SECD
SECD
Lisp
Lisp
$\lambda_{\uparrow\downarrow}$
JVM


L_n
L_{n-1}
L_2
L_1
L_1
L_0
M

Solving the Challenge



Solving the Challenge



Collapsing Towers of Interpreters,
Amin & Rompf POPL ’18

1971

Yoshihiko Futamura

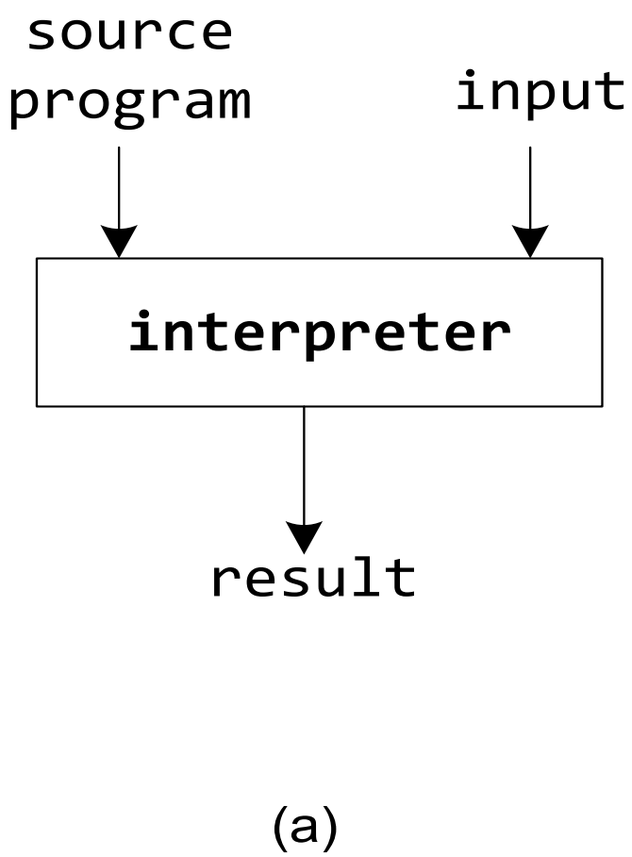
Partial Evaluation of Computation Process and its Application to Compiler Generation

Yoshihiko Futamura,

Central Research Laboratory, Hitachi, Ltd.

Kokubunji, Tokyo, Japan.

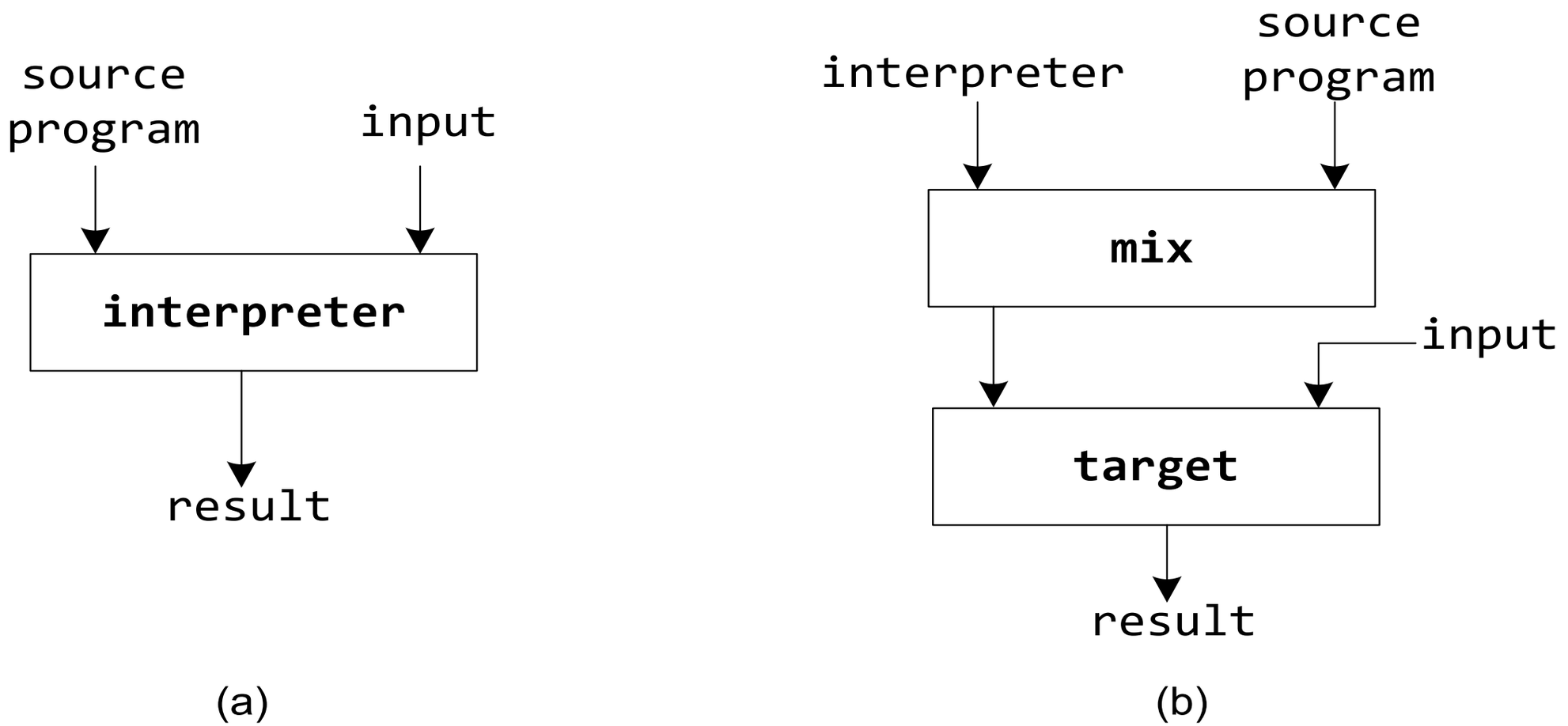
The 1st Futamura Projection



(a)

image credit: Ruby Tahboub (Purdue)

The 1st Futamura Projection

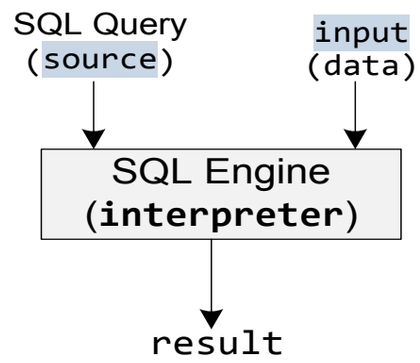


Specializing an **interpreter** with respect to a program produces a **compiled** version of that program.

image credit: Ruby Tahboub (Purdue)

Practical Realization

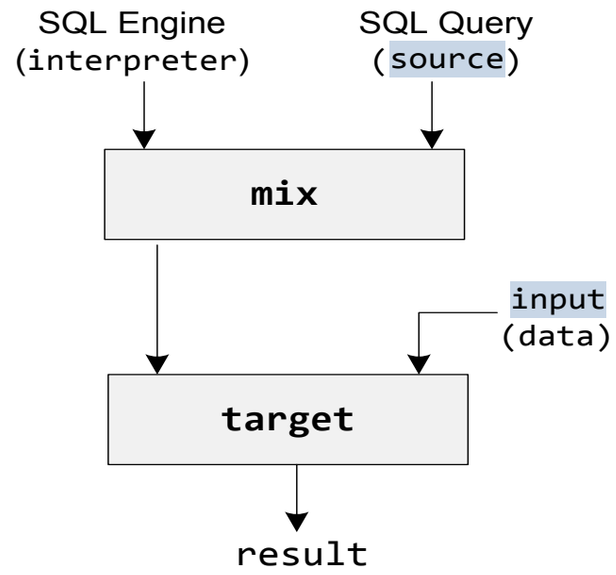
```
result = interpreter(source, input)
```



(a)

Query Interpreter

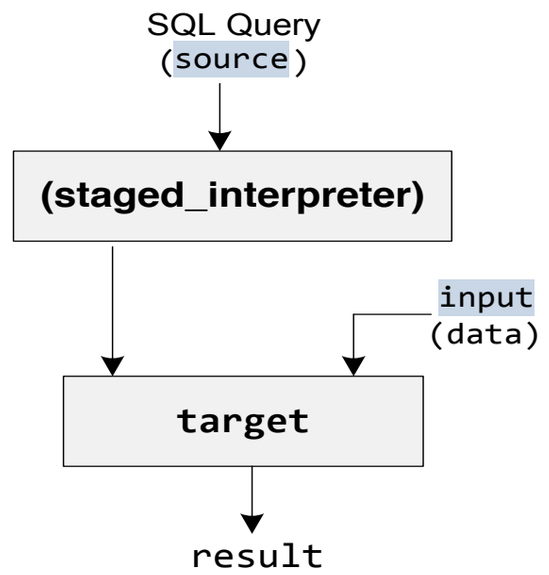
```
target = mix(interpreter, source)
result = target(input)
```



(b)

The first Futamura Projection

```
target = staged_interpreter(source)
result = target(input)
```



(c)

The first Futamura Projection
realization through specialization

Automatic partial evaluation is a hard problem,
especially binding-time analysis (BTA)

Solution: start with a binding-time annotated (staged) program,
in a multi-level language.

image credit: Ruby Tahboub (Purdue)

A staged interpreter is a compiler

Staging

- Multi-level language
  `n | x | e @ᵇ e | λᵇx.e | …`

- Quasiquotation in Lisp / MetaML
  `n | x | e e | λx.e | <e> | ~e | run e`

- Lightweight Modular Staging (LMS) in Scala
  driven by types: `T` vs `Rep[T]`

L_n

I_n

L_{n-1}

I_{n-1}

...

I_2

L_1

I_1

L_0

L_n

Staged I_n

L_{n-1}

I_{n-1}

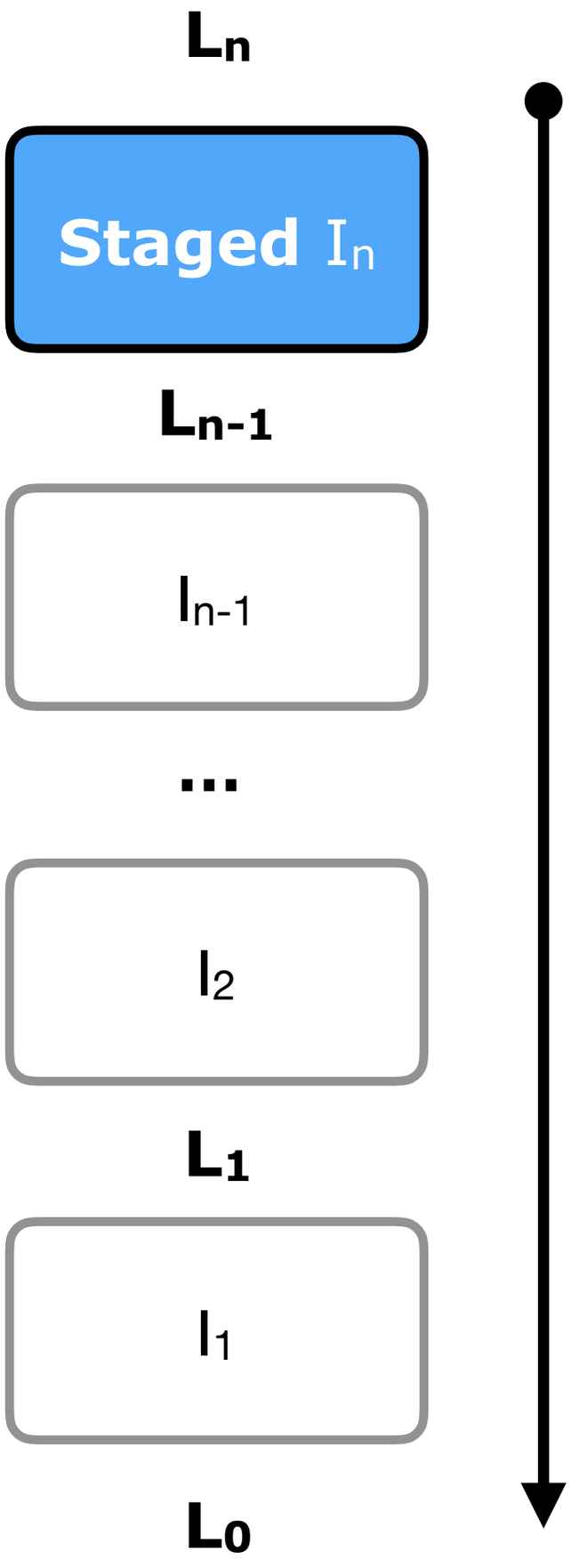
...

I_2

L_1

I_1

L_0



base $L_0 = \lambda\uparrow\downarrow$

multi-level λ-calculus

Stage Polymorphism

user-controlled staging, see
Staging for Generic Programming in Space and Time,
Ofenbeck et al. GPCE ’17

λ↑↓

- Multi-level λ-calculus
- `Lift` operator
- `Let`-insertion
- Stage polymorphism
- Akin to manual *online* partial evaluation

Definitional Interpreter in Scala (or Scheme)

// Multi-stage evaluation

```
def evalms(env: Env, e: Exp): Val = e match {
  case Lit(n)       => Cst(n)
  case Var(n)       => env(n)
  case Cons(e1,e2)  => Tup(evalms(env,e1),evalms(env,e2))
  case Lam(e)       => Clo(env,e)
  case Let(e1,e2)   => val v1 = evalms(env,e1); evalms(env:+v1,e2)
  case App(e1,e2)   => (evalms(env,e1), evalms(env,e2)) match {
    case (Clo(env3,e3), v2)   => evalms(env3:+Clo(env3,e3):+v2,e3)
    case (Code(s1), Code(s2)) => reflectc(App(s1,s2)) }
  case If(c,a,b)    => evalms(env,c) match {
    case Cst(n)       => if (n != 0) evalms(env,a) else evalms(env,b)
    case Code(c1)     => reflectc(If(c1, reifyc(evalms(env,a)), reifyc(evalms(env,b)))) }
case IsNum(e1)    => evalms(env,e1) match {
  case Code(s1)     => reflectc(IsNum(s1))
  case Cst(n)       => Cst(1)
  case v            => Cst(0) }
case Plus(e1,e2)  => (evalms(env,e1), evalms(env,e2)) match {
  case (Cst(n1), Cst(n2))  => Cst(n1+n2)
  case (Code(s1),Code(s2)) => reflectc(Plus(s1,s2)) }
…
case Lift(e)      => liftc(evalms(env,e))
case Run(b,e)     => evalms(env,b) match {
  case Code(b1)     => reflectc(Run(b1, reifyc(evalms(env,e))))
  case _            => evalmsg(env, reifyc({ stFresh = env.length; evalms(env, e) })) }
```

// Multi-stage evaluation

```
def evalms(env: Env, e: Exp): Val = e match {
  case Lit(n)       => Cst(n)
  case Var(n)       => env(n)
  case Cons(e1,e2)  => Tup(evalms(env,e1),evalms(env,e2))
  case Lam(e)       => Clo(env,e)
  case Let(e1,e2)   => val v1 = evalms(env,e1); evalms(env:+v1,e2)
  case App(e1,e2)   => (evalms(env,e1), evalms(env,e2)) match {
    case (Clo(env3,e3), v2)   => evalms(env3:+Clo(env3,e3):+v2,e3)
    case (Code(s1), Code(s2)) => reflectc(App(s1,s2)) }
  case If(c,a,b)    => evalms(env,c) match {
    case Cst(n)       => if (n != 0) evalms(env,a) else evalms(env,b)
    case Code(c1)     => reflectc(If(c1, reifyc(evalms(env,a)), reifyc(evalms(env,b)))) }
  case IsNum(e1)    => evalms(env,e1) match {
    case Code(s1)     => reflectc(IsNum(s1))
    case Cst(n)       => Cst(1)
    case v            => Cst(0) }
  case Plus(e1,e2)  => (evalms(env,e1), evalms(env,e2)) match {
    case (Cst(n1), Cst(n2))  => Cst(n1+n2)
    case (Code(s1),Code(s2)) => reflectc(Plus(s1,s2)) }
  ...
  case Lift(e)      => liftc(evalms(env,e))
  case Run(b,e)     => evalms(env,b) match {
    case Code(b1)     => reflectc(Run(b1, reifyc(evalms(env,e))))
    case _            => evalmsg(env, reifyc({ stFresh = env.length; evalms(env, e) })) }
```

// Multi-stage evaluation

```
def evalms(env: Env, e: Exp): Val = e match {
  case Lit(n)       => Cst(n)
  case Var(n)       => env(n)
  case Cons(e1,e2)  => Tup(evalms(env,e1),evalms(env,e2))
  case Lam(e)       => Clo(env,e)
  case Let(e1,e2)   => val v1 = evalms(env,e1); evalms(env:+v1,e2)
  case App(e1,e2)   => (evalms(env,e1), evalms(env,e2)) match {
    case (Clo(env3,e3), v2)   => evalms(env3:+Clo(env3,e3):+v2,e3)
    case (Code(s1), Code(s2)) => reflectc(App(s1,s2)) }
  case If(c,a,b)    => evalms(env,c) match {
    case Cst(n)       => if (n != 0) evalms(env,a) else evalms(env,b)
    case Code(c1)     => reflectc(If(c1, reifyc(evalms(env,a)), reifyc(evalms(env,b)))) }
  case IsNum(e1)    => evalms(env,e1) match {
    case Code(s1)     => reflectc(IsNum(s1))
    case Cst(n)       => Cst(1)
    case v            => Cst(0) }
  case Plus(e1,e2)  => (evalms(env,e1), evalms(env,e2)) match {
    case (Cst(n1), Cst(n2))  => Cst(n1+n2)
    case (Code(s1),Code(s2)) => reflectc(Plus(s1,s2)) }
  …
  case Lift(e)      => liftc(evalms(env,e))
  case Run(b,e)     => evalms(env,b) match {
    case Code(b1)     => reflectc(Run(b1, reifyc(evalms(env,e))))
    case _            => evalmsg(env, reifyc({ stFresh = env.length; evalms(env, e) })) }
```

// Multi-stage evaluation

```
def evalms(env: Env, e: Exp): Val = e match {
  case Lit(n)       => Cst(n)
  case Var(n)       => env(n)
  case Cons(e1,e2)  => Tup(evalms(env,e1),evalms(env,e2))
  case Lam(e)       => Clo(env,e)
  case Let(e1,e2)   => val v1 = evalms(env,e1); evalms(env:+v1,e2)
  case App(e1,e2)   => (evalms(env,e1), evalms(env,e2)) match {
    case (Clo(env3,e3), v2)   => evalms(env3:+Clo(env3,e3):+v2,e3)
    case (Code(s1), Code(s2)) => reflectc(App(s1,s2)) }
  case If(c,a,b)    => evalms(env,c) match {
    case Cst(n)       => if (n != 0) evalms(env,a) else evalms(env,b)
    case Code(c1)     => reflectc(If(c1, reifyc(evalms(env,a)), reifyc(evalms(env,b)))) }
  case IsNum(e1)    => evalms(env,e1) match {
    case Code(s1)     => reflectc(IsNum(s1))
    case Cst(n)       => Cst(1)
    case v            => Cst(0) }
  case Plus(e1,e2)  => (evalms(env,e1), evalms(env,e2)) match {
    case (Cst(n1), Cst(n2))  => Cst(n1+n2)
    case (Code(s1),Code(s2)) => reflectc(Plus(s1,s2)) }
  ...
  case Lift(e)      => liftc(evalms(env,e))
  case Run(b,e)     => evalms(env,b) match {
    case Code(b1)     => reflectc(Run(b1, reifyc(evalms(env,e))))
    case _            => evalmsg(env, reifyc({ stFresh = env.length; evalms(env, e) })) }
```

Lift

```
def lift(v: Val): Exp = v match {
  case Cst(n)          => Lit(n)
  case Tup(a,b)        => val (Code(u),Code(v))=(a,b);
    reflect(Cons(u,v))
  case Clo(env2,e2) => reflect(Lam(reifyc(evalms(
    env2:+Code(fresh()):+Code(fresh()),e2))))
  case Code(e)         => reflect(Lift(e)) }

def liftc(v: Val) = Code(lift(v))
```

Lift

```
def lift(v: Val): Exp = v match {
  case Cst(n)          => Lit(n)
  case Tup(a,b)        => val (Code(u),Code(v))=(a,b);
    reflect(Cons(u,v))
  case Clo(env2,e2) => reflect(Lam(reifyc(evalms(
    env2:+Code(fresh()):+Code(fresh()),e2))))
  case Code(e)         => reflect(Lift(e)) }

def liftc(v: Val) = Code(lift(v))
```

Lift

```
def lift(v: Val): Exp = v match {
  case Cst(n)          => Lit(n)
  case Tup(a,b)        => val (Code(u),Code(v))=(a,b);
    reflect(Cons(u,v))
  case Clo(env2,e2) => reflect(Lam(reifyc(evalms(
    env2:+Code(fresh()):+Code(fresh()),e2))))
  case Code(e)         => reflect(Lift(e)) }

def liftc(v: Val) = Code(lift(v))
```

Lift

```
def lift(v: Val): Exp = v match {
  case Cst(n)          => Lit(n)
  case Tup(a,b)        => val (Code(u),Code(v))=(a,b);
    reflect(Cons(u,v))
  case Clo(env2,e2) => reflect(Lam(reifyc(evalms(
    env2:+Code(fresh()):+Code(fresh()),e2))))
  case Code(e)         => reflect(Lift(e)) }

def liftc(v: Val) = Code(lift(v))
```

λ↑↓

- Multi-level λ-calculus
- `Lift` operator
- `Let`-insertion
- Stage polymorphism

Pink:

Stage-Polymorphic Meta-Circular Evaluator

```
;; Stage-Polymorphic Meta-Circular Evaluator for Pink

(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env
 (if (num?                exp)  (maybe-lift exp)
 (if (sym?                exp)  (env exp)
 (if (sym?           (car exp))
  (if (eq?  '+       (car exp))  (+   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '-       (car exp))  (-   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '*       (car exp))  (*   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'eq?     (car exp))  (eq? ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'if      (car exp))  (if  ((eval (cadr exp)) env) ((eval (caddr exp)) env)
                                                               ((eval (cadddr exp)) env))
  (if (eq?  'lambda (car exp))  (maybe-lift (lambda f x ((eval (cadddr exp))
   (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))
 (if (eq?  'let     (car exp))  (let x ((eval (caddr exp)) env) ((eval (cadddr exp))
  (lambda _ y (if (eq?  y (cadr exp)) x (env y)))))
 (if (eq?  'lift    (car exp))  (lift ((eval (cadr exp)) env))
 (if (eq?  'run     (car exp))  (run  ((eval (cadr exp)) env) ((eval (caddr exp))env))
 (if (eq?  'num?    (car exp))  (num? ((eval (cadr exp)) env))
 (if (eq?  'sym?    (car exp))  (sym? ((eval (cadr exp)) env))
 (if (eq?  'car     (car exp))  (car  ((eval (cadr exp)) env))
 (if (eq?  'cdr     (car exp))  (cdr  ((eval (cadr exp)) env))
 (if (eq?  'cons    (car exp))  (maybe-lift (cons ((eval (cadr exp)) env)
                                                  ((eval (caddr exp)) env)))
 (if (eq?  'quote   (car exp))  (maybe-lift (cadr exp))
 ((env (car exp)) ((eval (cadr exp)) env))))))))))))))))))))))))))))))))
(((eval (car exp)) env) ((eval (cadr exp)) env)))))))))
```

```
;; Stage-Polymorphic Meta-Circular Evaluator for Pink

(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env
 (if (num?                exp)  (maybe-lift exp)
 (if (sym?                exp)  (env exp)
 (if (sym?           (car exp))
  (if (eq?  '+       (car exp))  (+   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '-       (car exp))  (-   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '*       (car exp))  (*   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'eq?     (car exp))  (eq? ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'if      (car exp))  (if  ((eval (cadr exp)) env) ((eval (caddr exp)) env)
                                                               ((eval (cadddr exp)) env))
  (if (eq?  'lambda (car exp))  (maybe-lift (lambda f x ((eval (cadddr exp))
   (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))
  (if (eq?  'let     (car exp))  (let x ((eval (caddr exp)) env) ((eval (cadddr exp))
   (lambda _ y (if (eq?  y (cadr exp)) x (env y)))))
  (if (eq?  'lift    (car exp))  (lift ((eval (cadr exp)) env))
  (if (eq?  'run     (car exp))  (run  ((eval (cadr exp)) env) ((eval (caddr exp))env))
  (if (eq?  'num?    (car exp))  (num? ((eval (cadr exp)) env))
  (if (eq?  'sym?    (car exp))  (sym? ((eval (cadr exp)) env))
  (if (eq?  'car     (car exp))  (car  ((eval (cadr exp)) env))
  (if (eq?  'cdr     (car exp))  (cdr  ((eval (cadr exp)) env))
  (if (eq?  'cons    (car exp))  (maybe-lift (cons ((eval (cadr exp)) env)
                                                   ((eval (caddr exp)) env)))
  (if (eq?  'quote   (car exp))  (maybe-lift (cadr exp))
  ((env (car exp)) ((eval (cadr exp)) env))))))))))))))))
(((eval (car exp)) env) ((eval (cadr exp)) env)))))))))
```

```
;; Stage-Polymorphic Meta-Circular Evaluator for Pink

(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env
 (if (num?                exp)  (maybe-lift exp)
 (if (sym?                exp)  (env exp)
 (if (sym?          (car exp))
  (if (eq?  '+      (car exp))  (+   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '-      (car exp))  (-   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '*      (car exp))  (*   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'eq?    (car exp))  (eq? ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'if     (car exp))  (if  ((eval (cadr exp)) env) ((eval (caddr exp)) env)
                                                              ((eval (cadddr exp)) env))
  (if (eq?  'lambda (car exp))  (maybe-lift (lambda f x ((eval (cadddr exp))
   (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))
  (if (eq?  'let    (car exp))  (let x ((eval (caddr exp)) env) ((eval (cadddr exp))
   (lambda _ y (if (eq?  y (cadr exp)) x (env y)))))
  (if (eq?  'lift   (car exp))  (lift ((eval (cadr exp)) env))
  (if (eq?  'run    (car exp))  (run  ((eval (cadr exp)) env) ((eval (caddr exp))env))
  (if (eq?  'num?   (car exp))  (num? ((eval (cadr exp)) env))
  (if (eq?  'sym?   (car exp))  (sym? ((eval (cadr exp)) env))
  (if (eq?  'car    (car exp))  (car  ((eval (cadr exp)) env))
  (if (eq?  'cdr    (car exp))  (cdr  ((eval (cadr exp)) env))
  (if (eq?  'cons   (car exp))  (maybe-lift (cons ((eval (cadr exp)) env)
                                                  ((eval (caddr exp)) env)))
  (if (eq?  'quote  (car exp))  (maybe-lift (cadr exp))
  ((env (car exp)) ((eval (cadr exp)) env)))))))))))))))))
 (((eval (car exp)) env) ((eval (cadr exp)) env)))))))))
```

```
;; Stage-Polymorphic Meta-Circular Evaluator for Pink

(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env
 (if (num?                exp)  (maybe-lift exp)
 (if (sym?                exp)  (env exp)
 (if (sym?           (car exp))
  (if (eq?  '+       (car exp))  (+   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '-       (car exp))  (-   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '*       (car exp))  (*   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'eq?     (car exp))  (eq? ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'if      (car exp))  (if  ((eval (cadr exp)) env) ((eval (caddr exp)) env)
                                                               ((eval (cadddr exp)) env))
  (if (eq?  'lambda (car exp))  (maybe-lift (lambda f x ((eval (cadddr exp))
   (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))
 (if (eq?  'let     (car exp))  (let x ((eval (caddr exp)) env) ((eval (cadddr exp))
  (lambda _ y (if (eq?  y (cadr exp)) x (env y)))))
 (if (eq?  'lift    (car exp))  (lift ((eval (cadr exp)) env))
 (if (eq?  'run     (car exp))  (run  ((eval (cadr exp)) env) ((eval (caddr exp))env))
 (if (eq?  'num?    (car exp))  (num? ((eval (cadr exp)) env))
 (if (eq?  'sym?    (car exp))  (sym? ((eval (cadr exp)) env))
 (if (eq?  'car     (car exp))  (car  ((eval (cadr exp)) env))
 (if (eq?  'cdr     (car exp))  (cdr  ((eval (cadr exp)) env))
 (if (eq?  'cons    (car exp))  (maybe-lift (cons ((eval (cadr exp)) env)
                                                  ((eval (caddr exp)) env)))
 (if (eq?  'quote   (car exp))  (maybe-lift (cadr exp))
 ((env (car exp)) ((eval (cadr exp)) env))))))))))))))))
(((eval (car exp)) env) ((eval (cadr exp)) env)))))))))
```

```
;; Stage-Polymorphic Meta-Circular Evaluator for Pink

(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env
 (if (num?                exp)  (maybe-lift exp)
 (if (sym?                exp)  (env exp)
 (if (sym?           (car exp))
  (if (eq?  '+       (car exp))  (+   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '-       (car exp))  (-   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '*       (car exp))  (*   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'eq?     (car exp))  (eq? ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'if      (car exp))  (if  ((eval (cadr exp)) env) ((eval (caddr exp)) env)
                                                              ((eval (cadddr exp)) env))
  (if (eq?  'lambda (car exp))  (maybe-lift (lambda f x ((eval (cadddr exp))
   (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))
 (if (eq?  'let     (car exp))  (let x ((eval (caddr exp)) env) ((eval (cadddr exp))
  (lambda _ y (if (eq?  y (cadr exp)) x (env y)))))
 (if (eq?  'lift    (car exp))  (lift ((eval (cadr exp)) env))
 (if (eq?  'run     (car exp))  (run  ((eval (cadr exp)) env) ((eval (caddr exp))env))
 (if (eq?  'num?    (car exp))  (num? ((eval (cadr exp)) env))
 (if (eq?  'sym?    (car exp))  (sym? ((eval (cadr exp)) env))
 (if (eq?  'car     (car exp))  (car  ((eval (cadr exp)) env))
 (if (eq?  'cdr     (car exp))  (cdr  ((eval (cadr exp)) env))
 (if (eq?  'cons    (car exp))  (maybe-lift (cons ((eval (cadr exp)) env)
                                                  ((eval (caddr exp)) env)))
 (if (eq?  'quote   (car exp))  (maybe-lift (cadr exp))
 ((env (car exp)) ((eval (cadr exp)) env))))))))))))))))))))))))))))))))
(((eval (car exp)) env) ((eval (cadr exp)) env)))))))))
```

```
;; Stage-Polymorphic Meta-Circular Evaluator for Pink

(lambda _ maybe-lift (lambda _ eval (lambda _ exp (lambda _ env
 (if (num?                exp)  (maybe-lift exp)
 (if (sym?                exp)  (env exp)
 (if (sym?           (car exp))
  (if (eq?  '+       (car exp))  (+   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '-       (car exp))  (-   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  '*       (car exp))  (*   ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'eq?     (car exp))  (eq? ((eval (cadr exp)) env) ((eval (caddr exp)) env))
  (if (eq?  'if      (car exp))  (if  ((eval (cadr exp)) env) ((eval (caddr exp)) env)
                                                               ((eval (cadddr exp)) env))
  (if (eq?  'lambda (car exp))  (maybe-lift (lambda f x ((eval (cadddr exp))
   (lambda _ y (if (eq? y (cadr exp)) f (if (eq? y (caddr exp)) x (env y)))))))
  (if (eq?  'let     (car exp))  (let x ((eval (caddr exp)) env) ((eval (cadddr exp))
   (lambda _ y (if (eq?  y (cadr exp)) x (env y)))))
  (if (eq?  'lift    (car exp))  (lift ((eval (cadr exp)) env))
  (if (eq?  'run     (car exp))  (run  ((eval (cadr exp)) env) ((eval (caddr exp))env))
  (if (eq?  'num?    (car exp))  (num? ((eval (cadr exp)) env))
  (if (eq?  'sym?    (car exp))  (sym? ((eval (cadr exp)) env))
  (if (eq?  'car     (car exp))  (car  ((eval (cadr exp)) env))
  (if (eq?  'cdr     (car exp))  (cdr  ((eval (cadr exp)) env))
  (if (eq?  'cons    (car exp))  (maybe-lift (cons ((eval (cadr exp)) env)
                                                   ((eval (caddr exp)) env)))
  (if (eq?  'quote   (car exp))  (maybe-lift (cadr exp))
  ((env (car exp)) ((eval (cadr exp)) env))))))))))))))))
(((eval (car exp)) env) ((eval (cadr exp)) env)))))))))
```

Pink Interpretation

```
(define eval ((lambda ev e
  (((eval-poly (lambda _ e e)) ev) e))
   #nil)))

(define fac-src (quote (lambda f n
  (if (eq? n 0) 1 (* n (f (- n 1)))))))

> ((eval fac-src) 4) ;=> 24
```

Double & Triple Pink Interpretation

```
> ((eval fac-src) 4)
```

```
> (((eval eval-src) fac-src) 4)
```

```
> ((((eval eval-src) eval-src) fac-src) 4)
```

```
;=>24
```

Pink Compilation

```
(define evalc ((lambda ev e
  (((eval-poly (lambda _ e (lift e)))
    ev) e)) #nil)))

(define fac-src (quote (lambda f n
  (if (eq? n 0) 1 (* n (f (- n 1))))))))

> (evalc fac-src) ;=> <code of fac>
```

Pink Compilation

```
> (evalc fac-src) ;; =>

(lambda f0 x1
  (let x2 (eq? x1 0)
  (if x2 1
    (let x3 (- x1 1)
    (let x4 (f0 x3)
    (* x1 x4))))))
```

Pink Collapsing

```
> (evalc fac-src)
> ((eval evalc-src) fac-src)
> ((eval eval-src) evalc-src) fac-src)
;=> <code of fac>
```

Pink Collapsing

```
> ((eval eval-src) evalc-src) fac-src)
 ;; =>

(lambda f0 x1
  (let x2 (eq? x1 0)
  (if x2 1
    (let x3 (- x1 1)
    (let x4 (f0 x3)
    (* x1 x4))))))
```

L_n

staged l_n

l_{n-1}

...

l_2

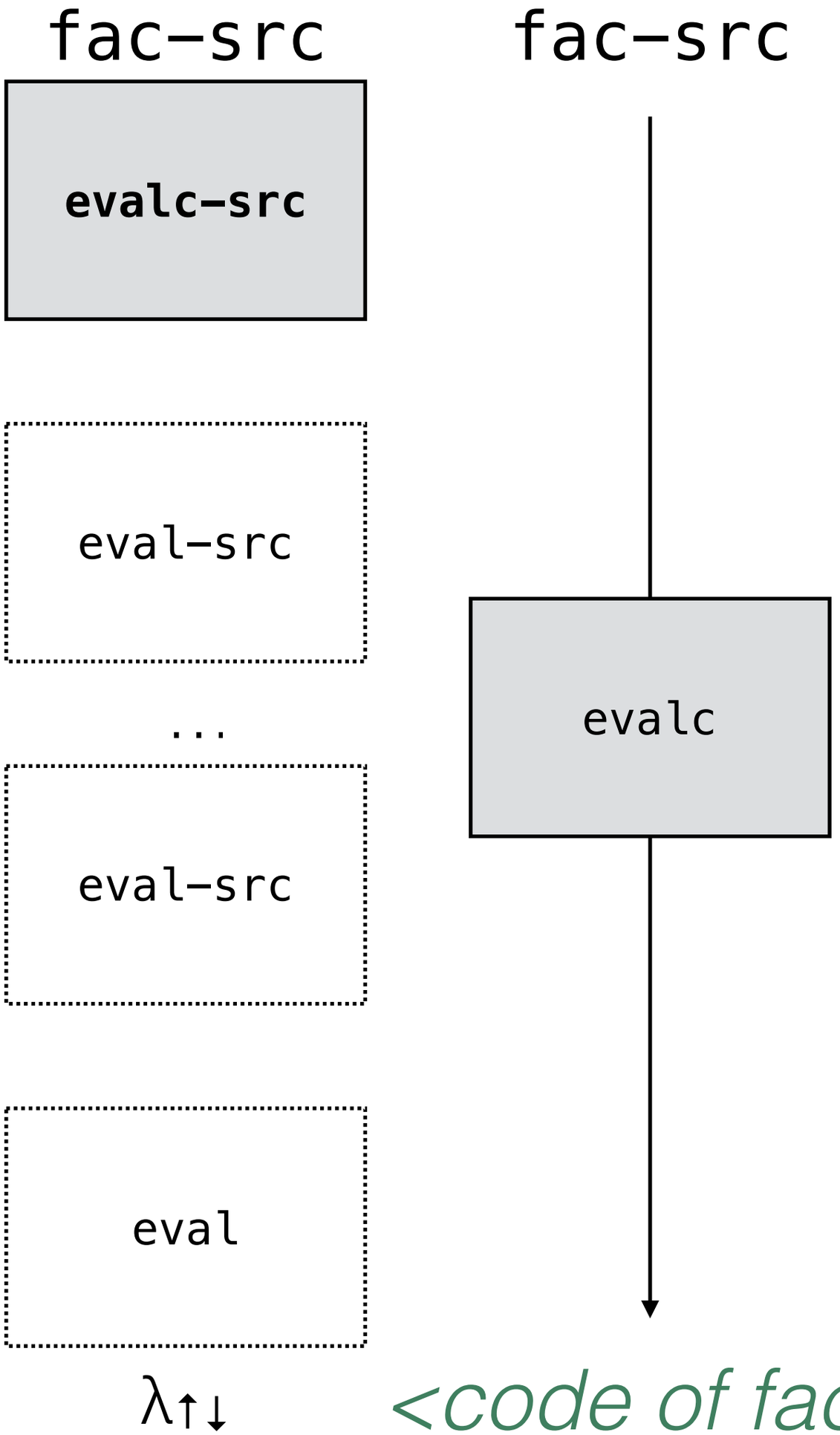
l_1

L_0

L_n

One-Pass Compiler

base L_0 = $\lambda_{\uparrow\downarrow}$ = **multi-level** λ-calculus




```
> (((eval eval-src) … eval-src)
    evalc-src) fac-src)

;=> <code of fac>
```

fac-src

modified
evalc-src

eval-src

...

eval-src

eval

λ↑↓

fac-src

modified
evalc
= transformer

<*modified* code of fac>

```
> (((eval eval-src) … eval-src)

   modified-evalc-src) fac-src)

;=> <modified code of fac>
```

Pink Transformers

```
> (evalc fac-src) ;; =>
(lambda f0 x1
  (let x2 (eq? x1 0)
  (if x2 1
    (let x3 (- x1 1)
    (let x4 (f0 x3)
    (* x1 x4))))))
```

```
> (trace-n-evalc fac-src) ;; =>
(lambda f0 x1
  (let x2 (log 0 x1)
  (let x3 (eq? x2 0)
  (if x3 1
    (let x4 (log 0 x1)
    (let x5 (log 0 x1)
    (let x6 (- x5 1)
    (let x7 (f0 x6)
    (* x4 x7)))))))))
```

```
> (cps-evalc fac-src) ;; =>
(lambda f0 x1 (lambda f2 x3
  (let x4 (eq? x1 0)
  (if x4 (x3 1)
    (let x5 (- x1 1)
    (let x6 (f0 x5)
    (let x7 (lambda f7 x8
      (let x9 (* x1 x8) (x3 x9)))
    (x6 x7)))))))))
```

Heterogeneity

Python

I_n

bytecode

I_{n-1}

x86 runtime

I_2

JavaScript VM

I_1

ARM CPU

Program

Python

Python → C

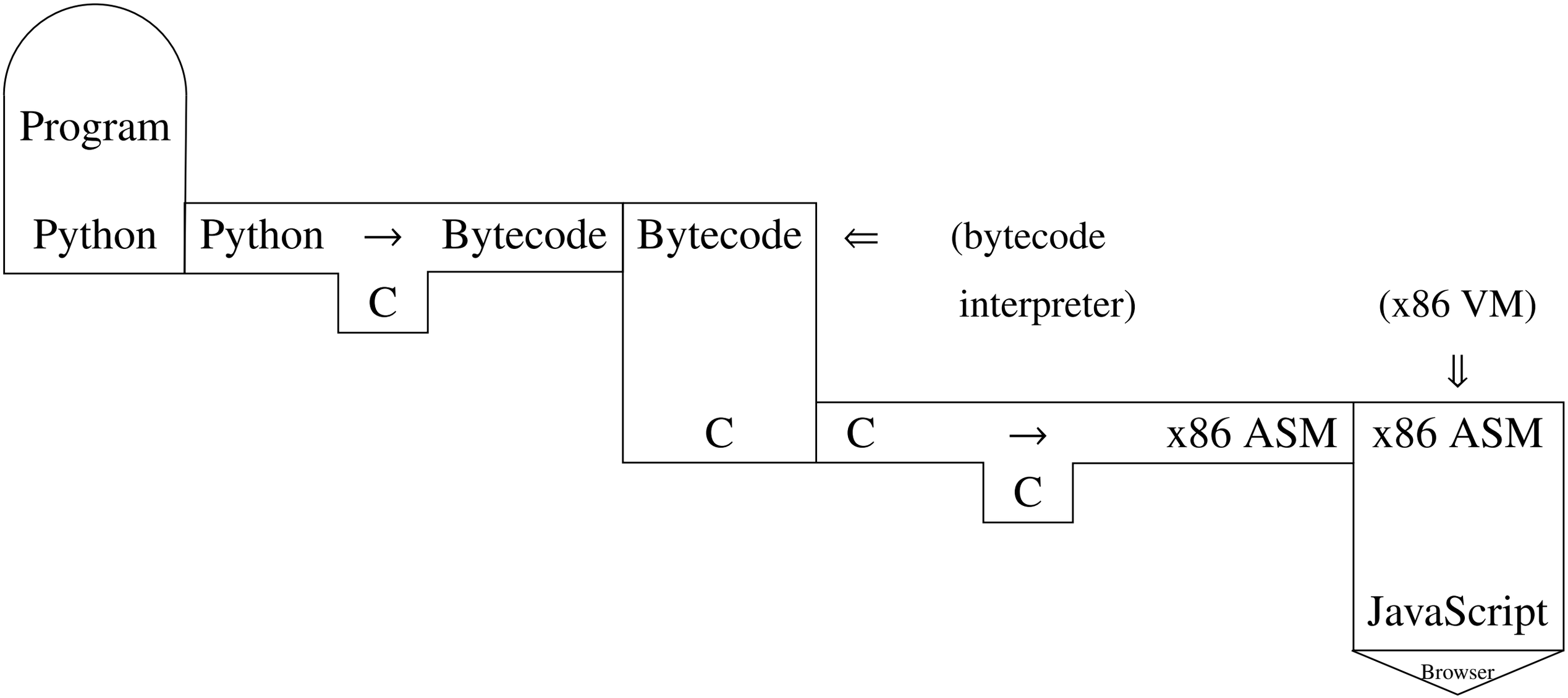
Python → x86 ASM

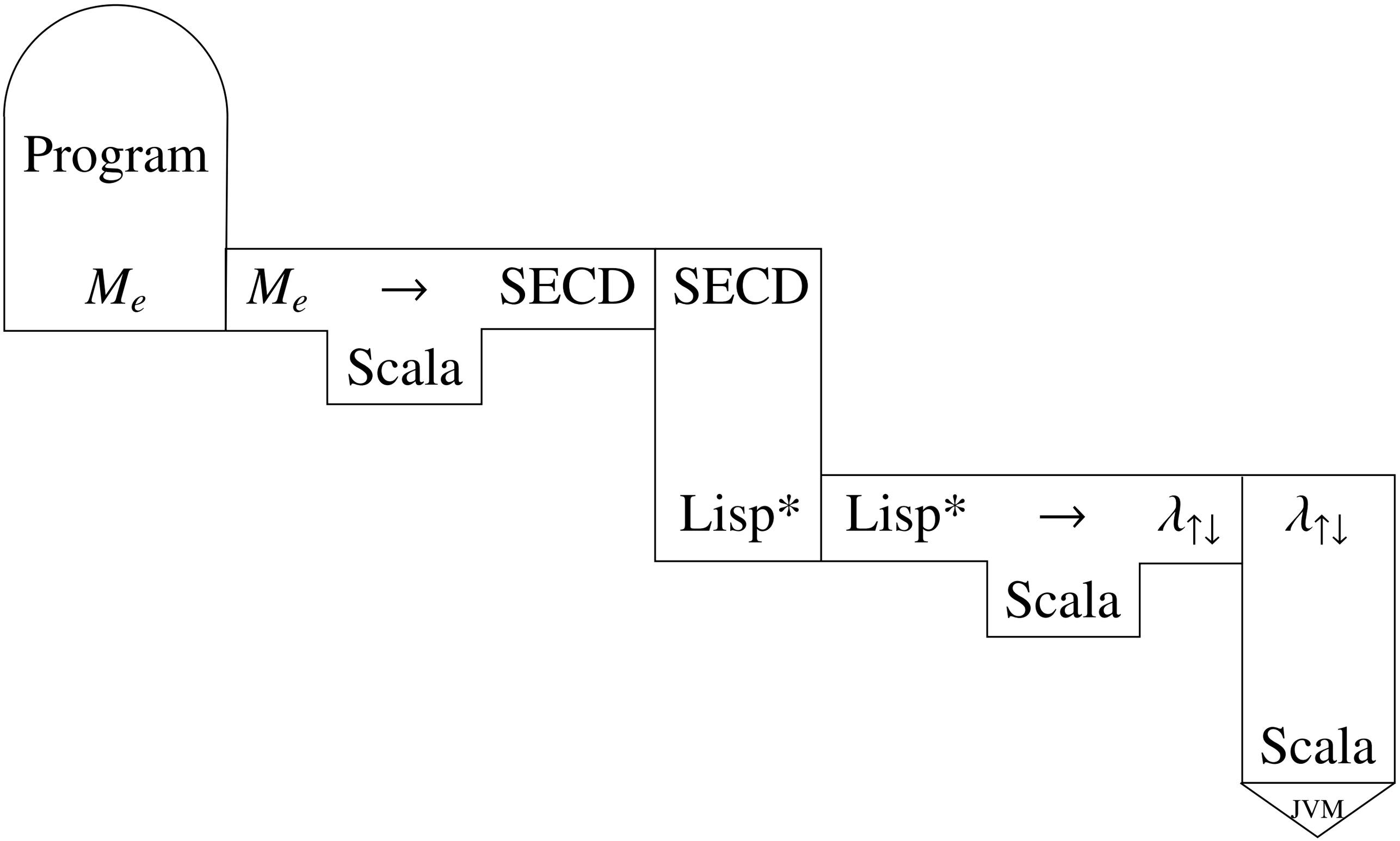
x86 ASM

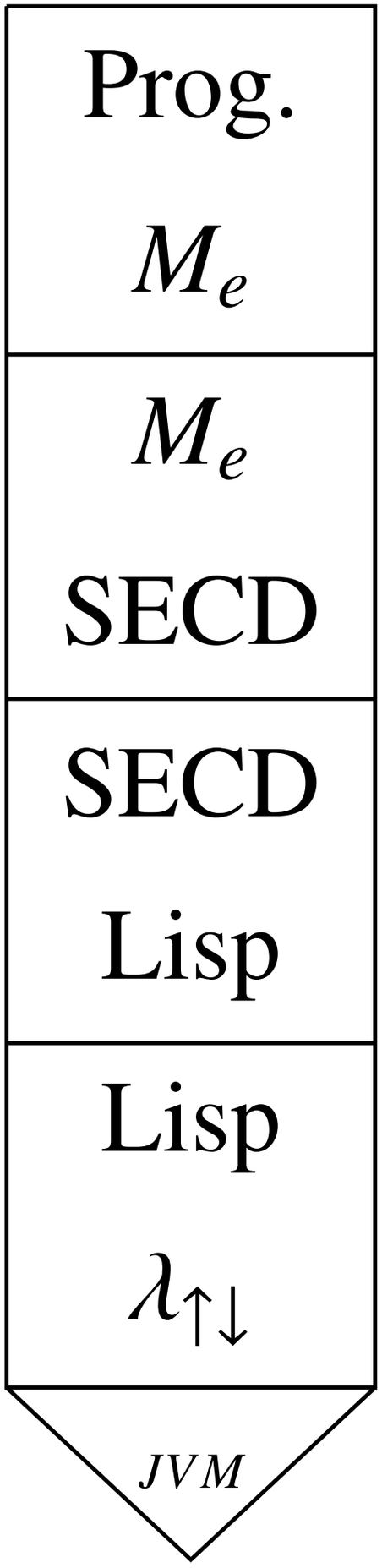
x86 Emulator

JavaScript

Browser


Program
Python
Python → Bytecode
C
Bytecode
C
⇐
(bytecode
interpreter)
(x86 VM)
⇓
C → x86 ASM
C
x86 ASM
JavaScript
Browser


Program
M_e
M_e → SECD
Scala
SECD
Lisp*
Lisp* → $\lambda_{\uparrow\downarrow}$
Scala
$\lambda_{\uparrow\downarrow}$
Scala
JVM


Prog.
M_e
M_e
SECD
SECD
Lisp
Lisp
$\lambda_{\uparrow\downarrow}$
JVM

Prog.

M_e

M_e

SECD

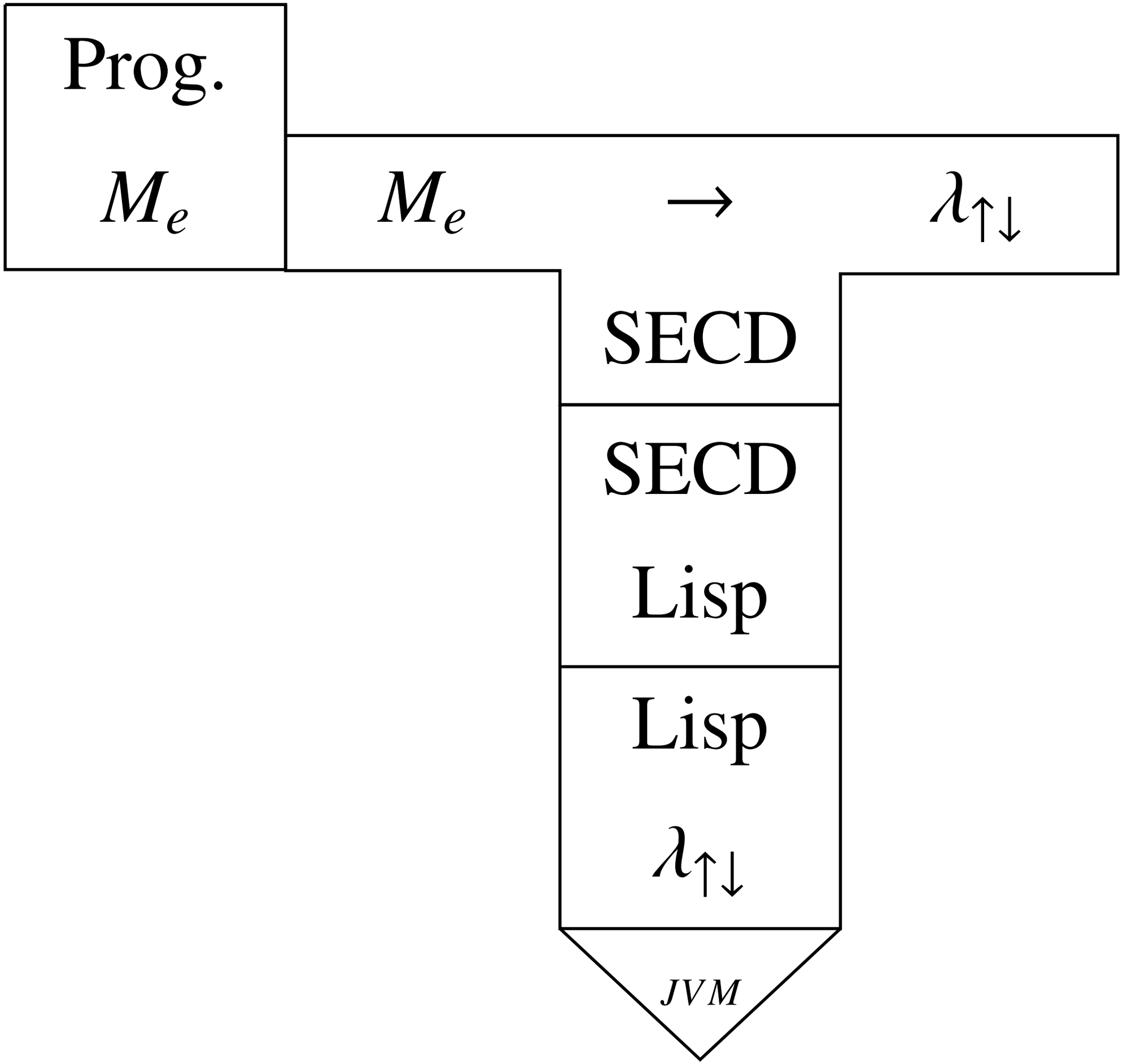
SECD $\rightarrow$ $\lambda_{\uparrow\downarrow}$

Lisp

Lisp

$\lambda_{\uparrow\downarrow}$

JVM


Prog.
M_e
M_e
M_e
$\rightarrow$
$\lambda_{\uparrow\downarrow}$
SECD
SECD
Lisp
Lisp
$\lambda_{\uparrow\downarrow}$
JVM

SECD machine

SECD Machine Examples

- `LDC 3 WRITEC`
- `LDC 1 LDC 2 ADD WRITEC`
- `LDC 2 SEL (LDC 3 JOIN) (LDC 0 JOIN) WRITEC`
- `NIL LDC 2 CONS LDC 1 CONS LDF (LDC 2 LDC 1 ADD RTN) AP WRITEC`
- `NIL LDC 2 CONS LDC 1 CONS LDF (LD (1 2) LD (1 1) ADD RTN) AP WRITEC`

RAP

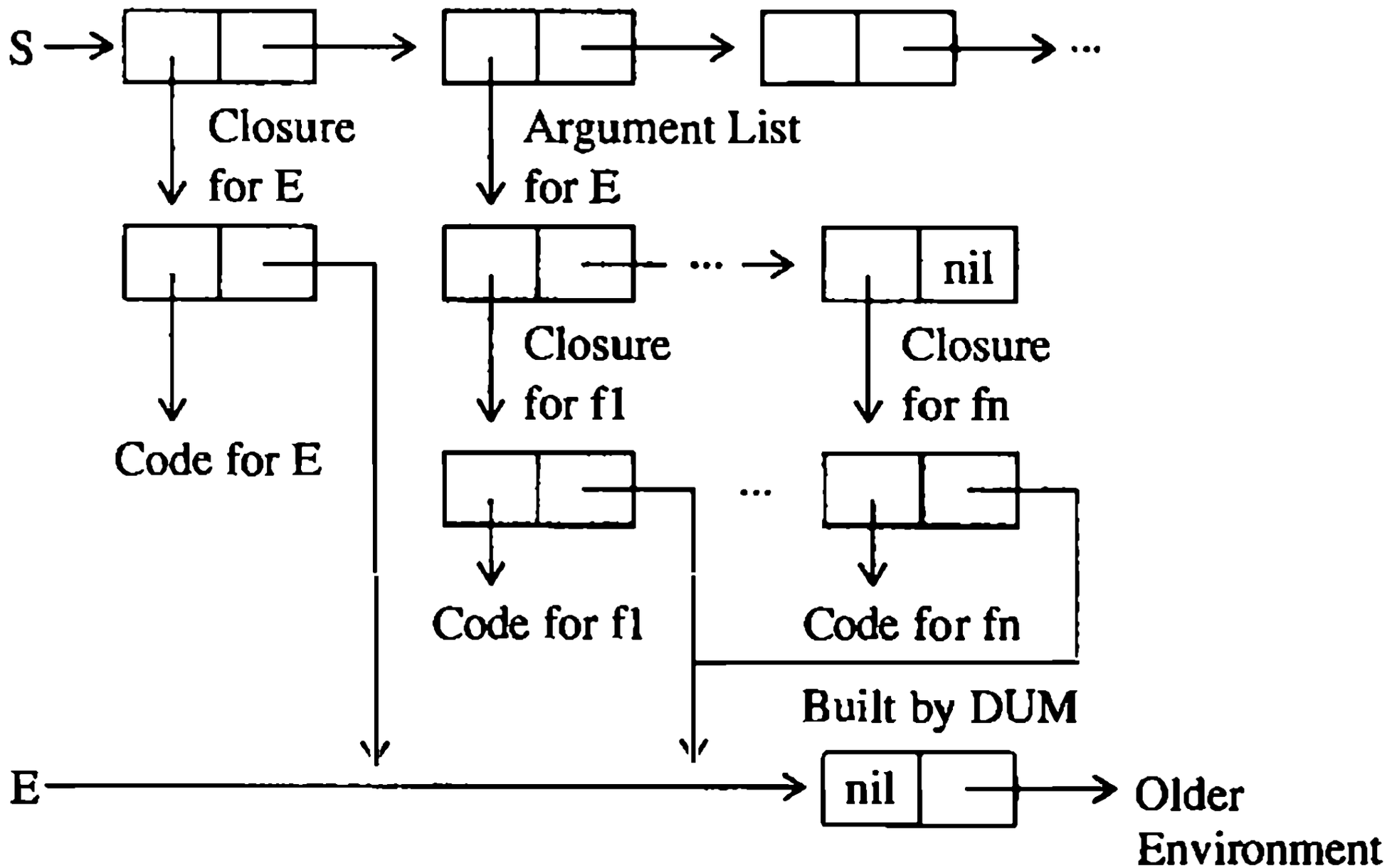
Assume: letrec f1 = A1 and ... and fn = An in E

= (λ f1 ... fn | E) A1 ... An

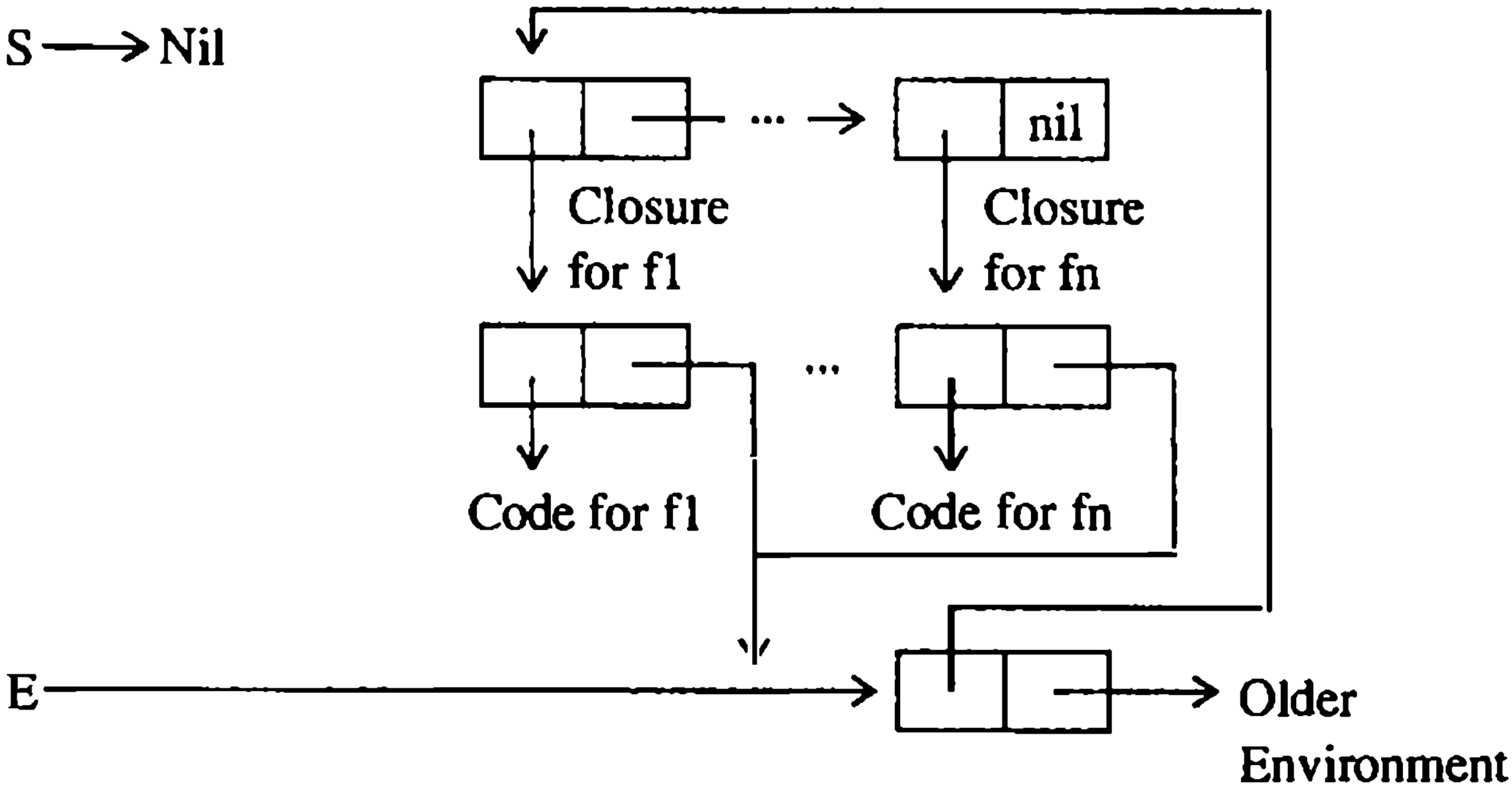
Code = (DUM NIL LDF (..code for An... RTN) CONS

LDF (..code for A1..RTN) CONS

LDF (..code for E..RTN) RAP)



(*a*) Before RAP.



(*b*) After RAP.

RAP Example

```
DUM NIL LDF

(LDC 0 LD (1 1) EQ SEL

(LDC 1 JOIN)

(LD (1 1) NIL LDC 1 LD (1 1) SUB CONS LD (2 1) AP MPY
JOIN)

RTN)

CONS LDF

(NIL LDC 6 CONS LD (1 1) AP RTN) RAP WRITEC
```

SECD Machine Impl.

```
(lambda machine s (lambda _ e (lambda _ c (lambda _ d

(if (eq? 'NIL (car c)) ((((machine (cons '() s)) e) (cdr
c)) d)

(if (eq? 'LDC (car c)) ((((machine (cons (cadr c) s)) e)
(cddr c)) d)

(if (eq? 'SEL (car c)) (if (car s)

  ((((machine (cdr s)) e) (cadr c))  (cons (cdddr c) d))

  ((((machine (cdr s)) e) (caddr c)) (cons (cdddr c) d)))

(if (eq? 'JOIN (car c)) ((((machine s) e) (car d)) (cdr
d))
```

RAP case

```
(if (eq? 'RAP (car c))

  (let f (car (cdr (car s))))

  (let ep (cdr (cdr (car s))))

  (let v (cadr s)

  ((((machine '()) (set-car! ep v)) f) (cons
(cddr s) (cons (cdr e) (cons (cdr c) d))))))))
```

Lift

SECD Lift

```
(lambda _ x (if (and (pair? x) (eq? '__clo (car x)))

   (let memo (cons (lift '()) '())

   (let _ (set-car! funs (cons (cons (cdr x) memo)
                               (car funs)))

(lift (lambda fun args

  (let _ (set-car! memo fun)

  ((((machine '()) (cons args (cdr (cdr x))))
                   (car (cdr x))) 'ret))))))

(lift x)))
```

LDF case

```
(if (eq? 'LDF (car c))

  ((((machine (cons (cons '__clo (cons (cadr
c) e)) s)) e) (cddr c)) d)
```

AP case

```
(if (eq? 'AP (car c))

 (let v (cadr s)

 (let r (if (code? (car s)) (cons '() (cons (car s) '())))
    ((assq (cdr (car s))) (car funs)))

 (if (eq? r '())

 (let f (car (cdr (car s))) (let ep (cdr (cdr (car s)))

 ((((machine '()) (cons v ep)) f) (cons (cddr s) (cons e
(cons (cdr c) d))))))

 (let fun (cadr r) ((((machine (cons (fun ((deeplift-if-
code fun) v)) (cddr s))) e) (cdr c)) d)))))
```

RTN case

```
(if (eq? 'RTN (car c))
    (if (eq? d 'ret) (car s)

    ((((machine (cons (car s) (car d))) (cadr
d)) (caddr d)) (cdddr d)))
```

Collapsing Towers of Interpreters: Recipe

- Make base of tower stage-polymorphic.
- Stage the user-most interpreter.
- For **heterogeneity**: adapt lift at each level to change representation of values to lower representation.

Python

M_e

bytecode

SECD

x86 runtime

Lisp*

JavaScript VM

λ↑↓

VM

ARM CPU