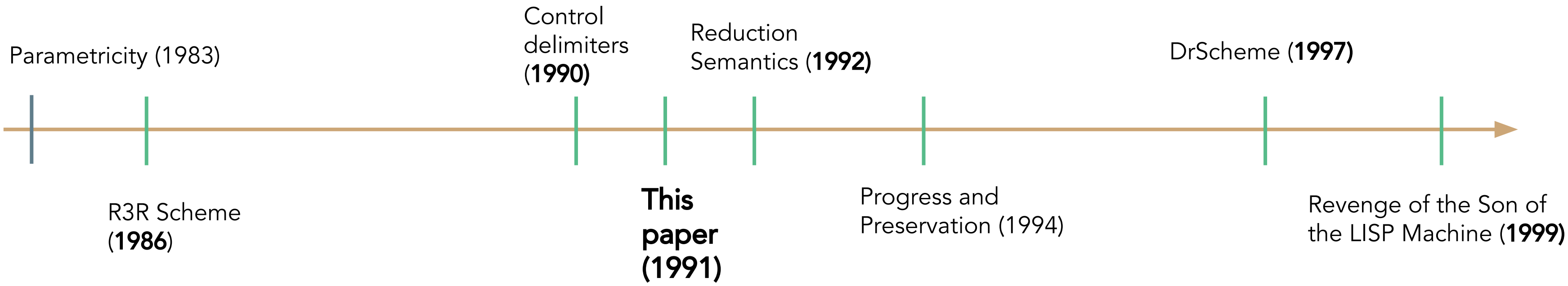
On the Expressive Power

of Programming Languages

Historical Context



Parametricity (1983)

R3R Scheme (**1986**)

Control delimiters (**1990**)

This paper (1991)

Reduction Semantics (**1992**)

Progress and Preservation (1994)

DrScheme (**1997**)

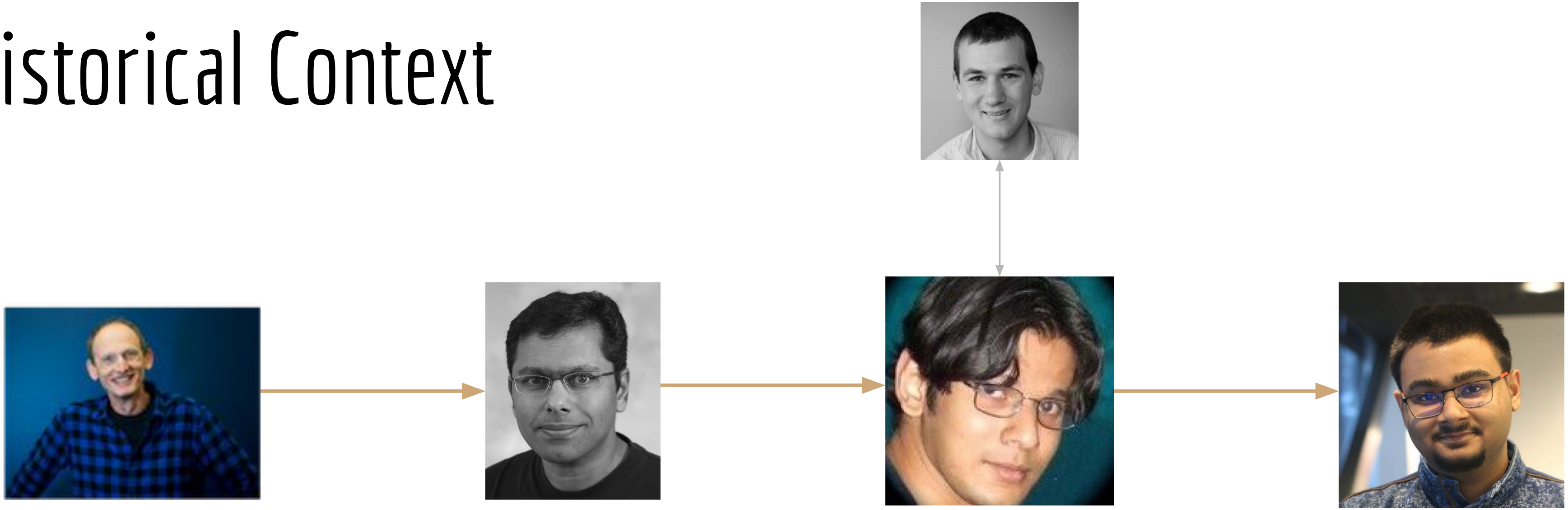
Revenge of the Son of the LISP Machine (**1999**)

Historical Context

1991: Writes this paper

1994: Shriram pivots from CompBio after reading it

Historical Context



1991: Writes this paper

1994: Shriram pivots from CompBio after reading it

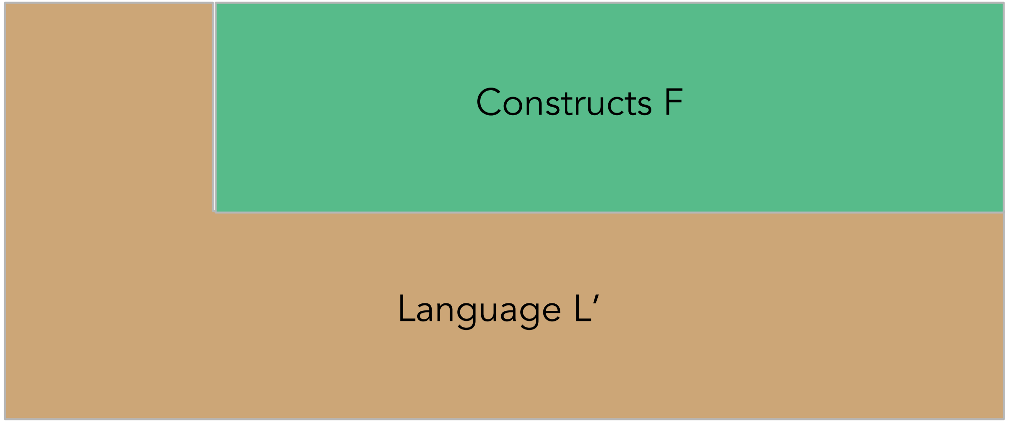
2010: Essence of JavaScript

2019: nothing of note.

Expressivity

Language L’

Expressivity

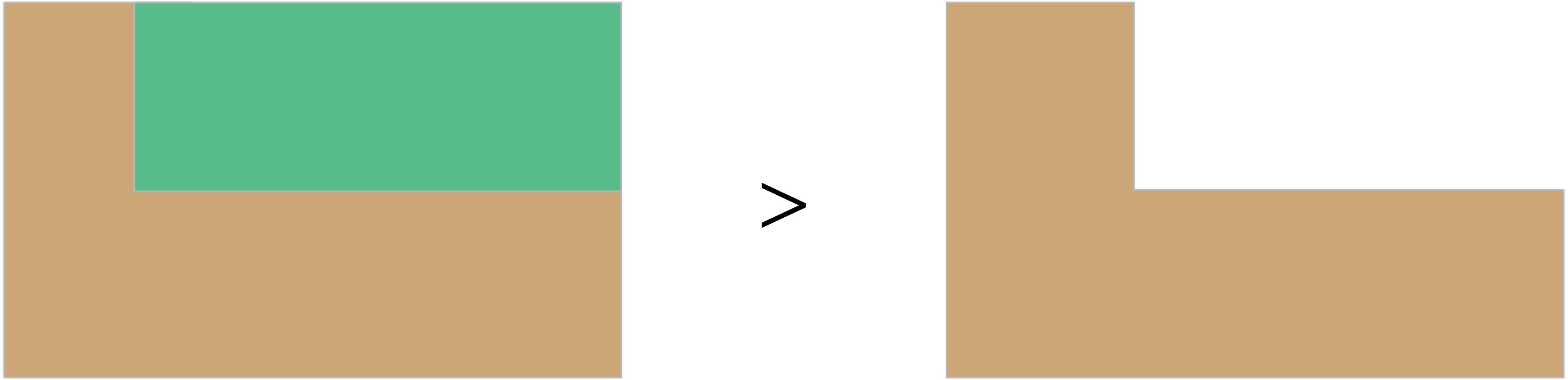


Constructs F

Language L’

Language L

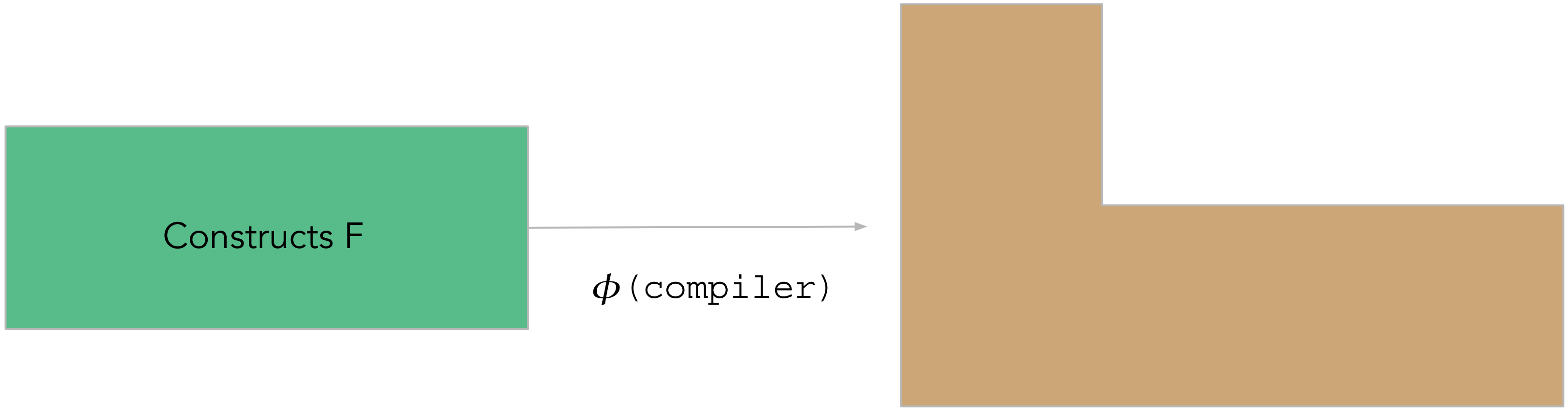
Expressivity



Language L

Language L’

Expressivity



Constructs F

ϕ `(compiler)`

Language L’

Expressivity

```
Let x = init in body
```

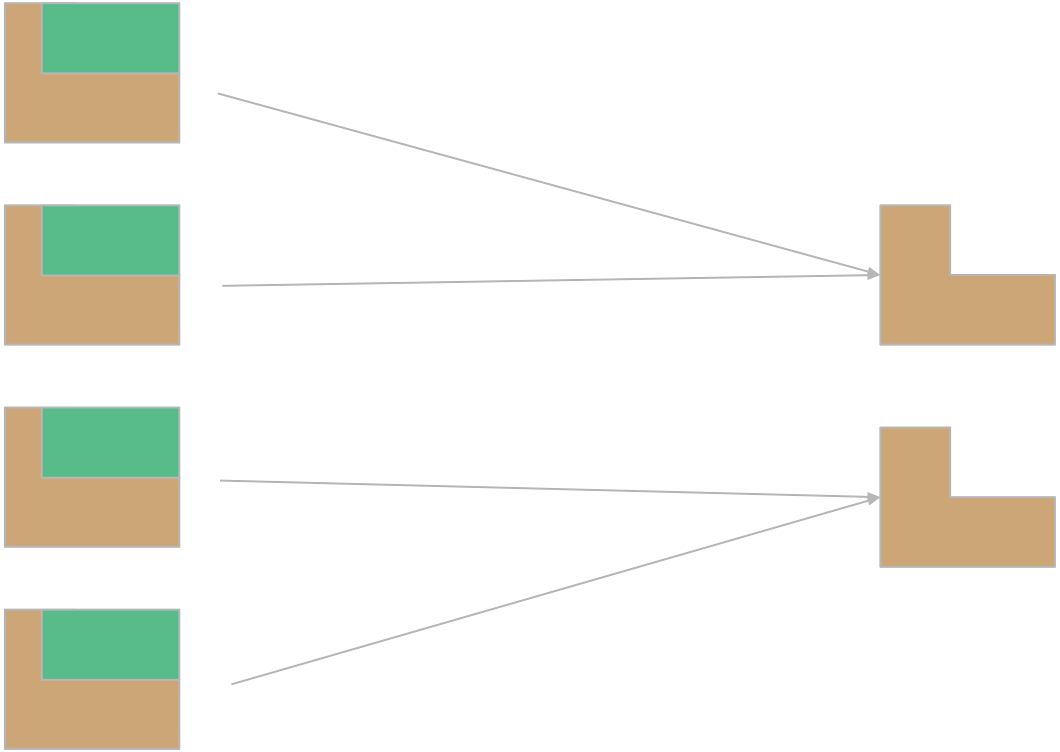
```
(fun x -> body) init
```

```
Let x = ref 0 in x++
```

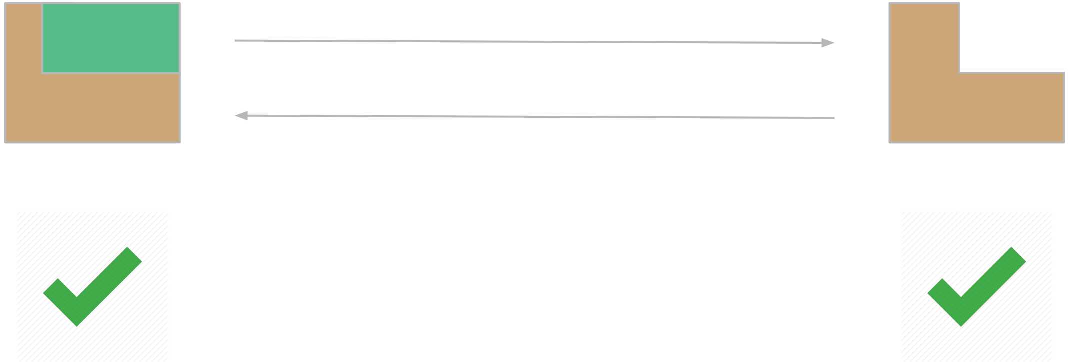
```
Let x = makeBox () in
x.setRef(x.getRef + 1)
```

Eliminable Constructs

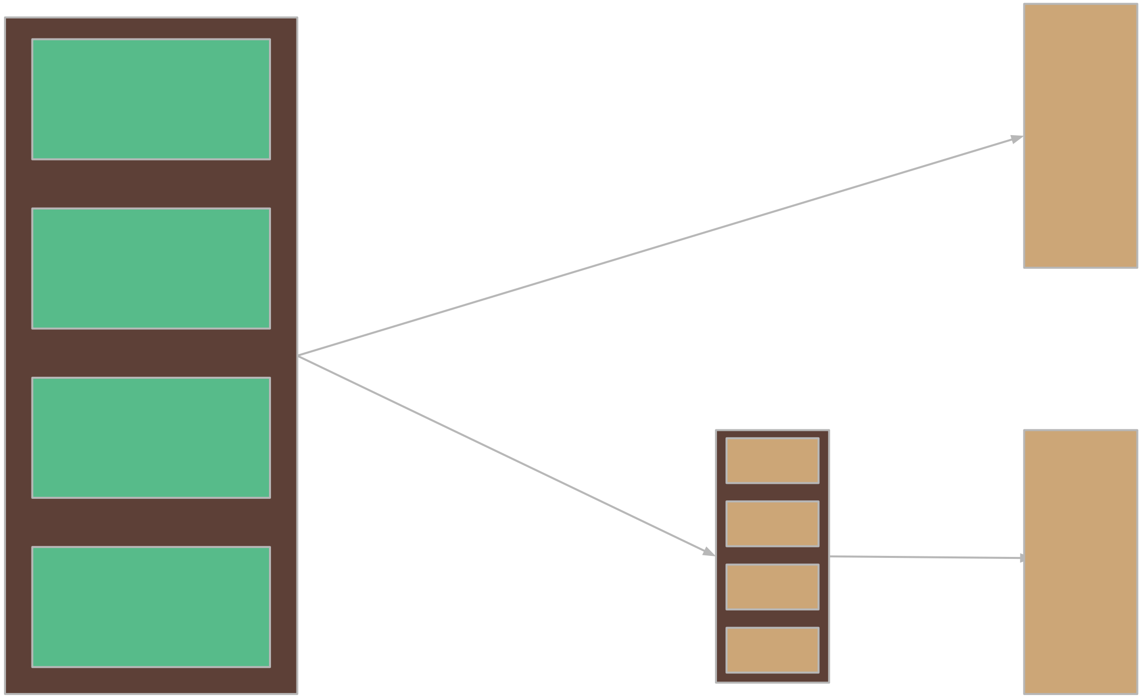
E1 $\varphi(e)$ is an $\mathcal{L}'$-program for all $\mathcal{L}$-programs e;



E3 $eval_{\mathcal{L}}(e)$ holds if and only if $eval_{\mathcal{L}'}(\varphi(e))$ holds for all $\mathcal{L}$-programs e.



E2 $\varphi(\mathbb{F}(e_1, \ldots, e_a)) = \mathbb{F}(\varphi(e_1), \ldots, \varphi(e_a))$ for all facilities $\mathbb{F}$ of $\mathcal{L}'$, i.e., φ is homomorphic in all constructs of $\mathcal{L}'$; and



Eliminable: Example

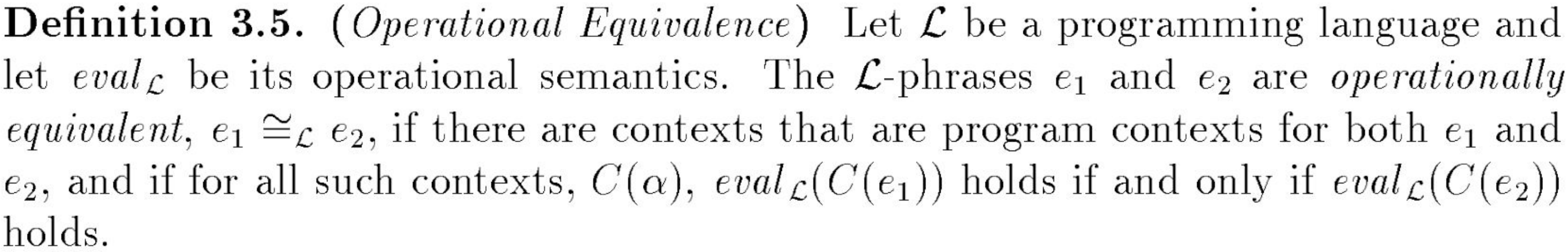
```
Let x = init in body
```

```
φ(Let x = init in body) => (fun x -> φ(body)) φ(init)
```

```
(fun x -> body) init
```

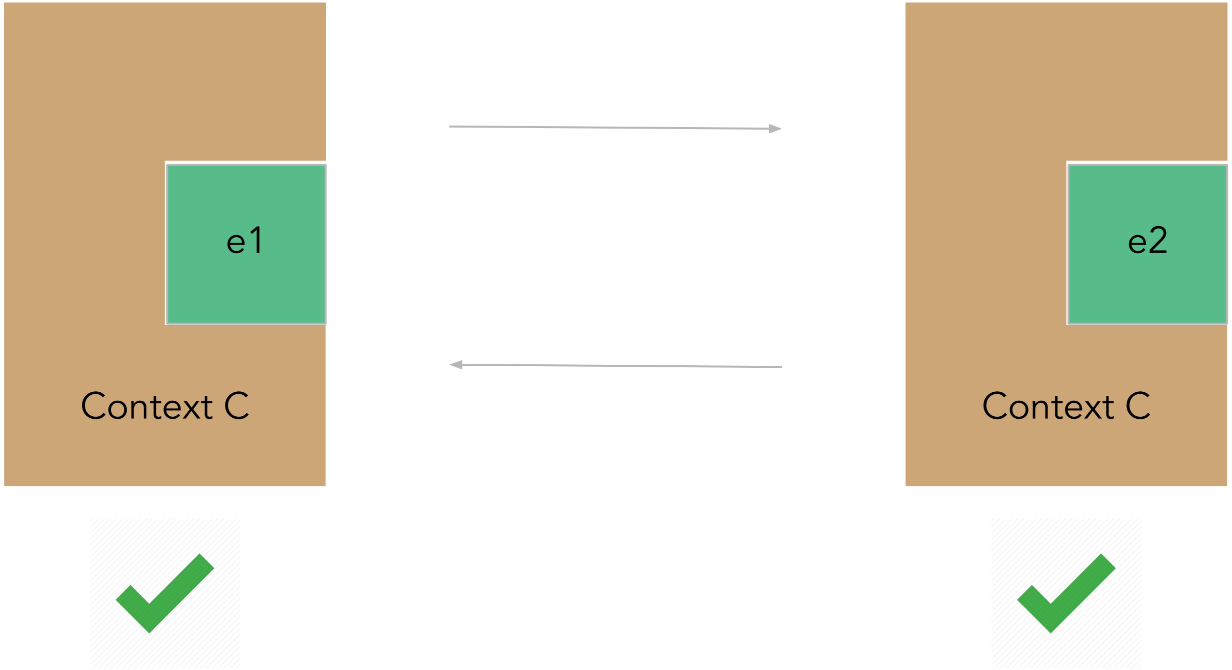
Contextual Equivalence

Or Observational Equivalence

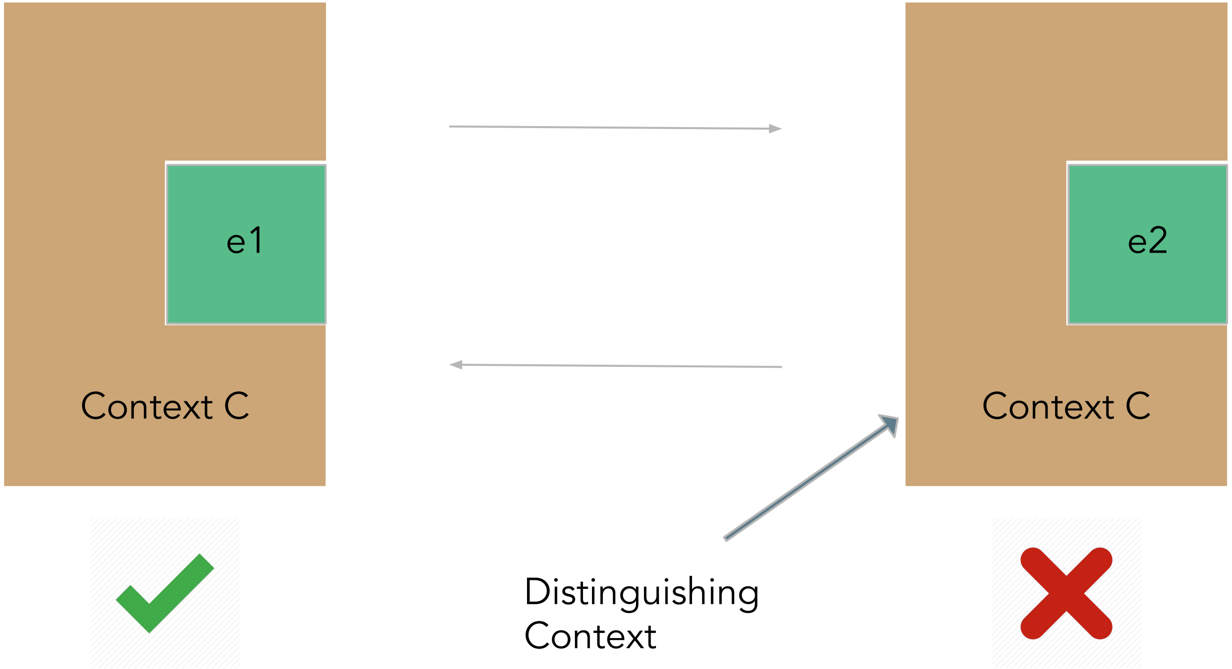


Definition 3.5. (*Operational Equivalence*) Let $\mathcal{L}$ be a programming language and let $eval_{\mathcal{L}}$ be its operational semantics. The $\mathcal{L}$-phrases e_1 and e_2 are *operationally equivalent*, $e_1 \cong_{\mathcal{L}} e_2$, if there are contexts that are program contexts for both e_1 and e_2, and if for all such contexts, $C(\alpha)$, $eval_{\mathcal{L}}(C(e_1))$ holds if and only if $eval_{\mathcal{L}}(C(e_2))$ holds.

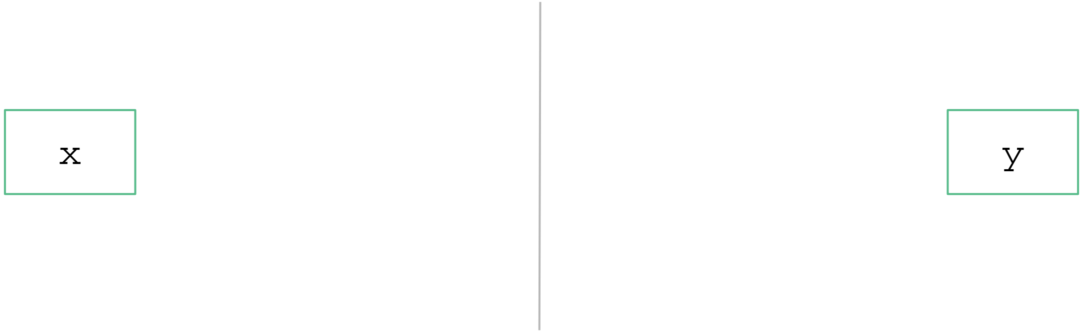
Definition 3.5. (*Operational Equivalence*) Let $\mathcal{L}$ be a programming language and let $eval_{\mathcal{L}}$ be its operational semantics. The $\mathcal{L}$-phrases e_1 and e_2 are *operationally equivalent*, $e_1 \cong_{\mathcal{L}} e_2$, if there are contexts that are program contexts for both e_1 and e_2, and if for all such contexts, $C(\alpha)$, $eval_{\mathcal{L}}(C(e_1))$ holds if and only if $eval_{\mathcal{L}}(C(e_2))$ holds.



Definition 3.5. (*Operational Equivalence*) Let $\mathcal{L}$ be a programming language and let $eval_{\mathcal{L}}$ be its operational semantics. The $\mathcal{L}$-phrases e_1 and e_2 are *operationally equivalent*, $e_1 \cong_{\mathcal{L}} e_2$, if there are contexts that are program contexts for both e_1 and e_2, and if for all such contexts, $C(\alpha)$, $eval_{\mathcal{L}}(C(e_1))$ holds if and only if $eval_{\mathcal{L}}(C(e_2))$ holds.



Distinguishing Context



```
C(a) = (fun x, y -> a) 1 (throw 1)
```

Contextual Equivalence: Example

```
(fun x, y -> x) (fun x -> x) (throw 1)
```

```
1
```

```
(fun x, y -> y) (fun x -> x) (throw 1)
```

```
(throw 1)
```

```
⊥
```

Contextual Equivalence: Example

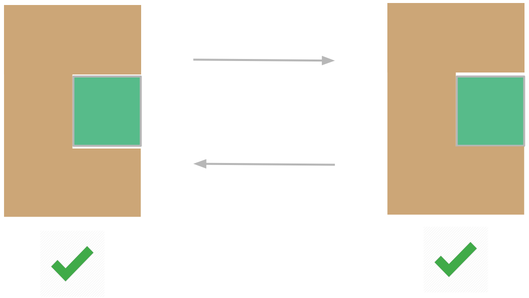
Expressivity

Constructs F is expressible in Language L if

→

ϕ satisfies **E1, E2, E3**

F is eliminable

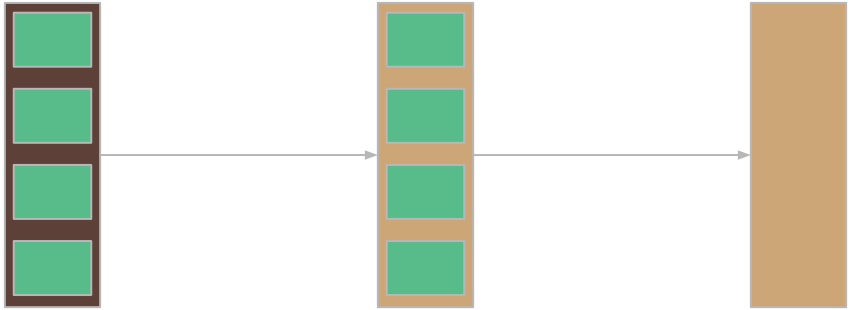


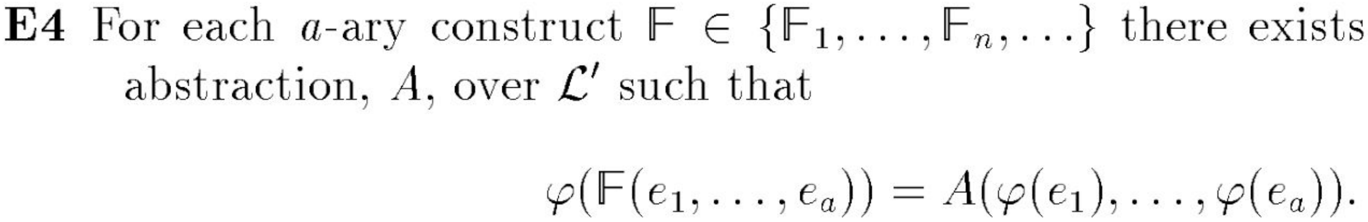
There is no *distinguishing context* for F and ϕ(F).

Macro-expressivity

E4 For each a-ary construct $\mathbb{F} \in \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ there exists an a-ary syntactic abstraction, A, over $\mathcal{L}'$ such that

$$\varphi(\mathbb{F}(e_1, \ldots, e_a)) = A(\varphi(e_1), \ldots, \varphi(e_a)).$$





E4 For each a-ary construct $\mathbb{F} \in \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ there exists an a-ary syntactic abstraction, A, over $\mathcal{L}'$ such that

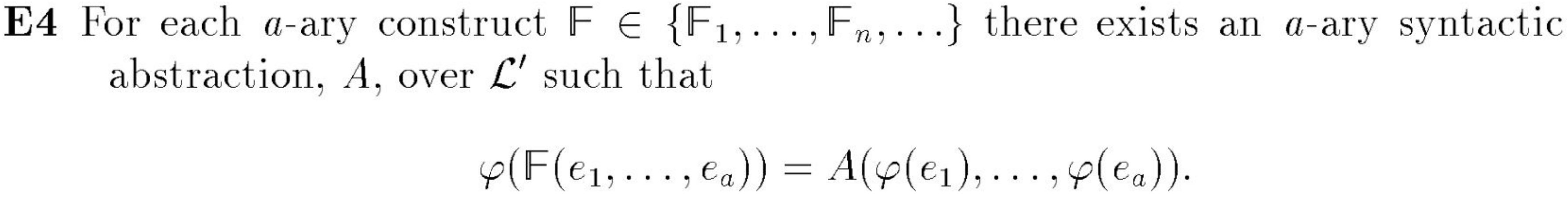
$$\varphi(\mathbb{F}(e_1, \ldots, e_a)) = A(\varphi(e_1), \ldots, \varphi(e_a)).$$

```
For (init, test, update, body)
```

```
While (test, body)
```

Macro expressivity: Example

Macro expressivity: Example



E4 For each a-ary construct $\mathbb{F} \in \{\mathbb{F}_1, \ldots, \mathbb{F}_n, \ldots\}$ there exists an a-ary syntactic abstraction, A, over $\mathcal{L}'$ such that

$$\varphi(\mathbb{F}(e_1, \ldots, e_a)) = A(\varphi(e_1), \ldots, \varphi(e_a)).$$

```
For (init, test, update, body)
```

```
For (init, test, upd, body) => init in While (ϕ(test), ϕ(body) ++ ϕ(update))
```

```
While (test, body)
```

Polymorphic **let**

$$\frac{A \vdash e : \tau; A \vdash b[x/e] : \tau'}{A \vdash \textbf{let}\ x = e\ \textbf{in}\ b : \tau'}$$

Call-by-value STLC

Expressive but Macro-inexpressive

Polymorphic **let**

$$\boxed{\frac{A \vdash e : \tau; A \vdash b[x/e] : \tau'}{A \vdash \mathbf{let}\ x = e\ \mathbf{in}\ b : \tau'}}$$

```
Let (x, e, b) => (fun x -> subst(x, φ(e), φ(b))) φ(e)
```

Call-by-value STLC

Expressive!

Expressive but Macro-inexpressive

AST function, not a *syntactic abstraction!*

```
Let (x, e, b) => (fun x -> subst(x, φ(e), φ(b))) φ(e)
```

Recursive macros are not a problem! Macro-based **subst** implementation will generate scoped macros. **subst** is truly performing a *compile-time computation.*

Expressive but Macro-inexpressive

What do we get?

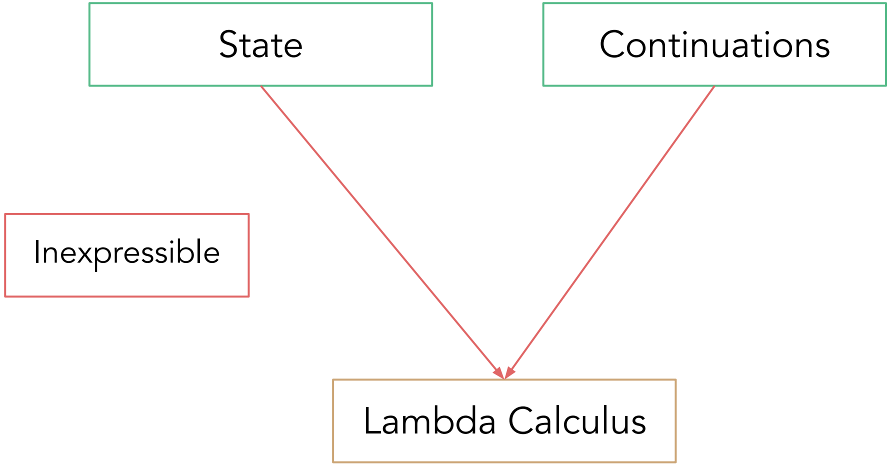
Eliminability

(Macro) expressivity

What do we get?

Eliminability

(Macro) expressivity



State

Continuations

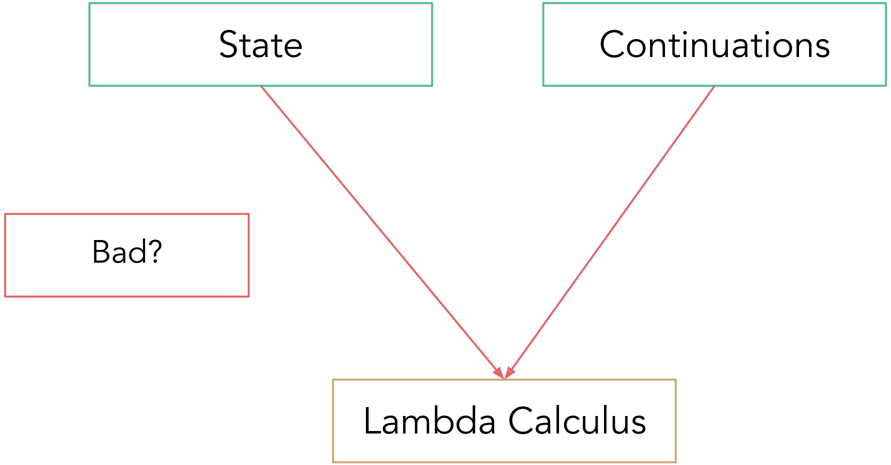
Inexpressible

Lambda Calculus

What do we get?

Eliminability

(Macro) expressivity



State

Continuations

Bad?

Lambda Calculus

Case Study

Essence of JavaScript*

```
let x = {
  a: 10,
  b: 20,
}

> { a: 10, b: 20 }
```

```
with (x) {
  a + b + 10
}

> 30
```

* The Essence of JavaScript (2010); Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi

Essence of JavaScript*

```
let x = {
  a: 10,
  b: 20,
}

> { a: 10, b: 20 }
```

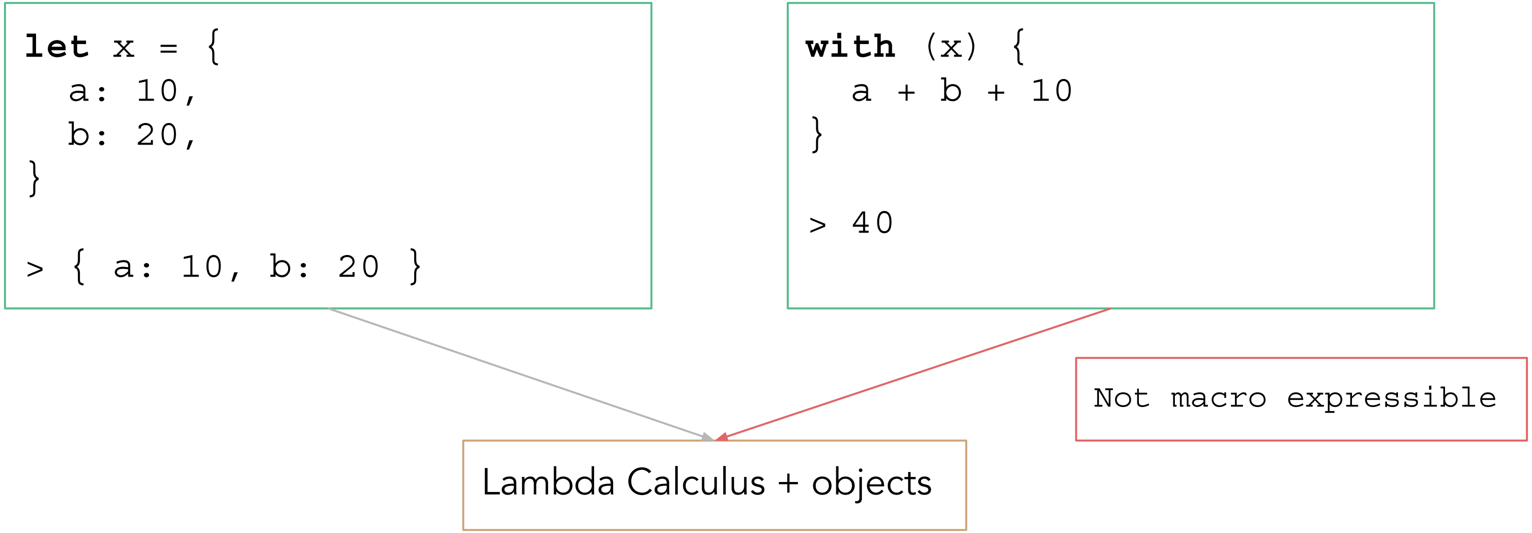
```
with (x) {
  a + b + 10
}

> 30
```

Lambda Calculus + objects

* The Essence of JavaScript (2010); Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi

Essence of JavaScript*



```
let x = {
  a: 10,
  b: 20,
}

> { a: 10, b: 20 }
```

```
with (x) {
  a + b + 10
}

> 40
```

Not macro expressible

Lambda Calculus + objects

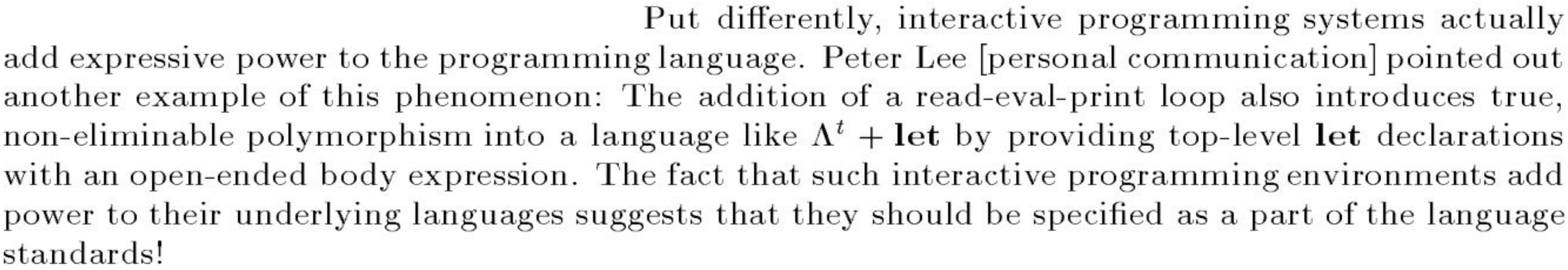
* The Essence of JavaScript (2010); Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi

Thanks!

```
(call/cc
 (lambda (return)
   (while (true)
     (return "Power Overwhelming!"))))
```

Discussion points

- Expressivity as a language design principle vs type directed language design.
- Why is this not the prevailing way of designing languages?
- Programming languages: isolated mathematical formalisms or complete ecosystems?



Put differently, interactive programming systems actually add expressive power to the programming language. Peter Lee [personal communication] pointed out another example of this phenomenon: The addition of a read-eval-print loop also introduces true, non-eliminable polymorphism into a language like $\Lambda^t + \textbf{let}$ by providing top-level **let** declarations with an open-ended body expression. The fact that such interactive programming environments add power to their underlying languages suggests that they should be specified as a part of the language standards!