Functional Programming is Easy, and Good for You

Matthias Felleisen (PLT)
Northeastern University

I am not a salesman.

Functional Programming

Functional Programming **Functional Programming Languages**

Functional Programming != Functional Programming Languages

Theorem

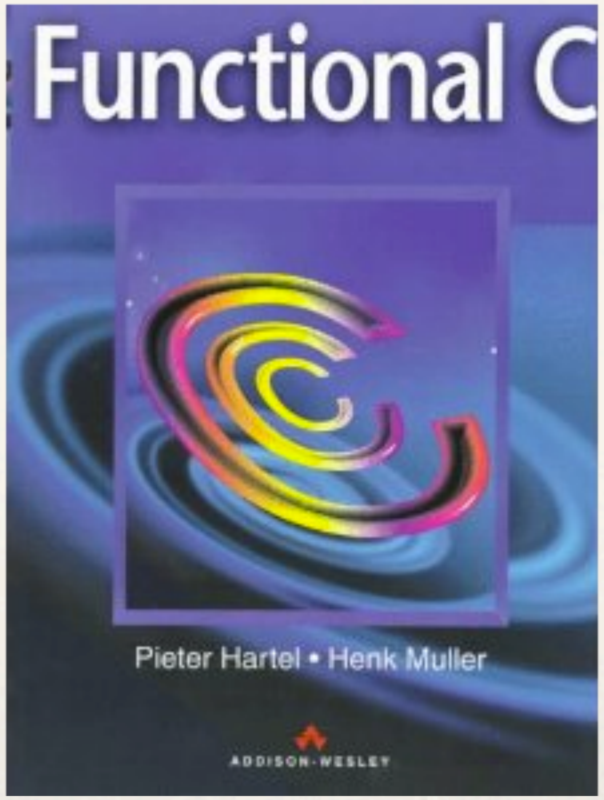
Functional Programming != Functional Programming Languages

Proof:

Theorem

Functional Programming **!=** Functional Programming Languages

Proof:



Functional Programming

Functional Programming Languages

pure
Clean

Functional Programming

Functional Programming Languages

mostly
OCaml

pure
Clean

strict
all others

Functional Programming

Functional Programming Languages

lazy
Haskell

mostly
OCaml

pure
Clean

untyped
Scheme

strict
all others

Functional Programming

Functional Programming Languages

lazy
Haskell

typed
SML

mostly
OCaml

pure
Clean

higher-order
all others

untyped
Scheme

strict
all others

Functional Programming

Functional Programming Languages

lazy
Haskell

typed
SML

first-order
ACL2

mostly
OCaml

pure
Clean

higher-order
all others

untyped
Scheme

standard VM
F#, Scala

strict
all others

Functional Programming

Functional Programming Languages

lazy
Haskell

special VM
Racket

typed
SML

first-order
ACL2

mostly
OCaml

Functional Programming

Functional Programming Languages

pure
Clean

higher-order
all others

untyped
Scheme

strict
all others

parallel
Clojure

standard VM
F#, Scala

lazy
Haskell

special VM
Racket

distributed
Erlang

typed
SML

first-order
ACL2

mostly
OCaml

If all of this is **functional programming (languages)**,
isn’t it all **overwhelming and difficult?**

If all of this is **functional programming (languages)**,
isn't it all **overwhelming and difficult?**

Not at all. And I am here to explain { what / why / +/- }

What is Functional Programming?
What is a Functional Programming Language?

Pop Quiz

Pop Quiz: Who said this?

Though [it] came from many motivations, ... one was **to find a more flexible version of assignment, and then to try to eliminate it altogether.**

Favor immutability.

Use **value objects** when possible.

Answer: The OO Experts

Though [it] came from many motivations, ... one was **to find a more flexible version of assignment, and then to try to eliminate it altogether.**

Alan Kay,
History of Smalltalk (1993)

Favor immutability.

Joshua Bloch,
Effective Java (2001)

Use **value objects** when possible.

Kent Beck,
Test Driven Development (2001)

One Definition of Functional Programming

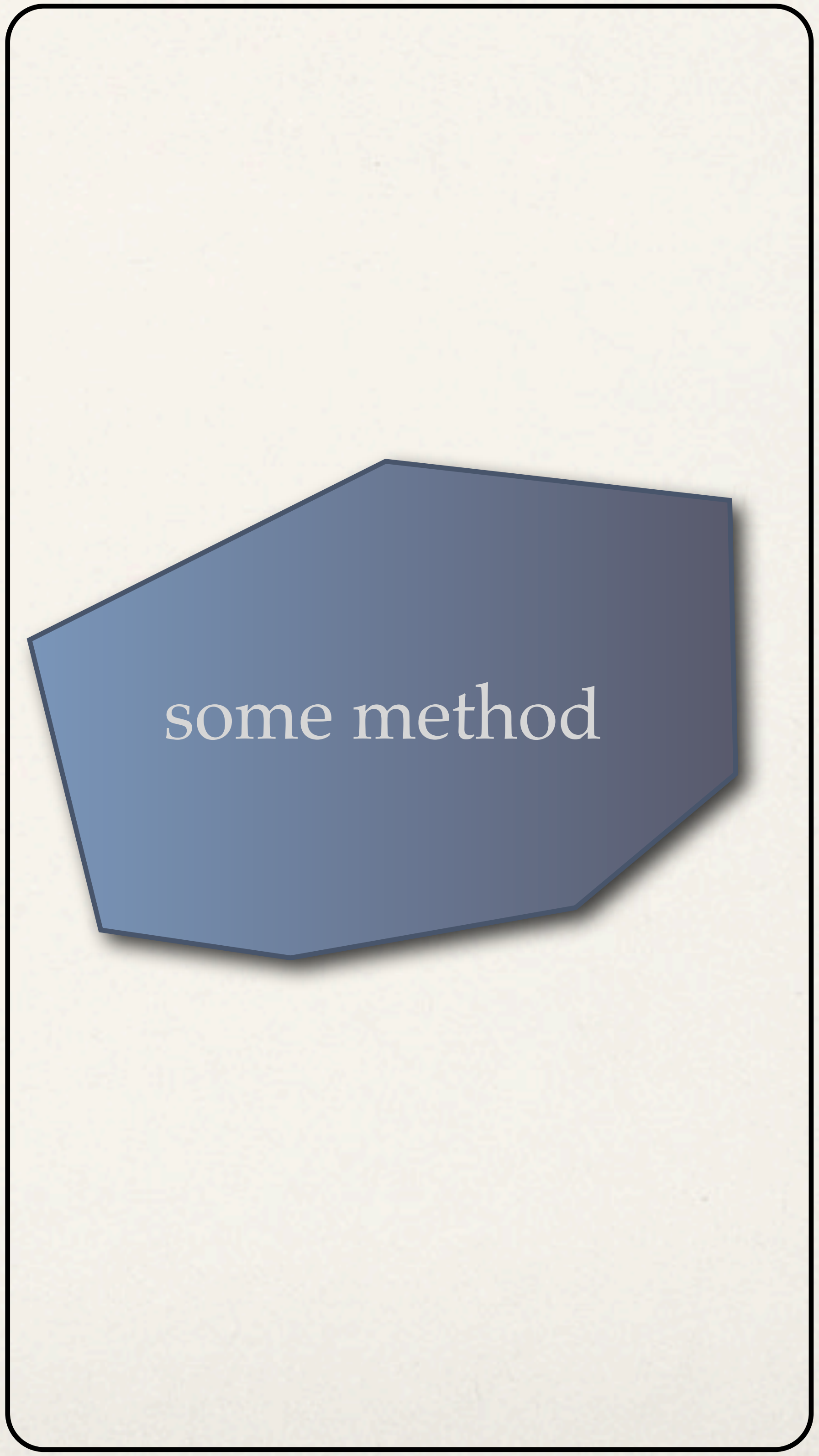
So one definition of functional programming is

no (few) assignment statements
no (few) mutable objects.

One Definition of Functional Programming

So one definition of functional programming is

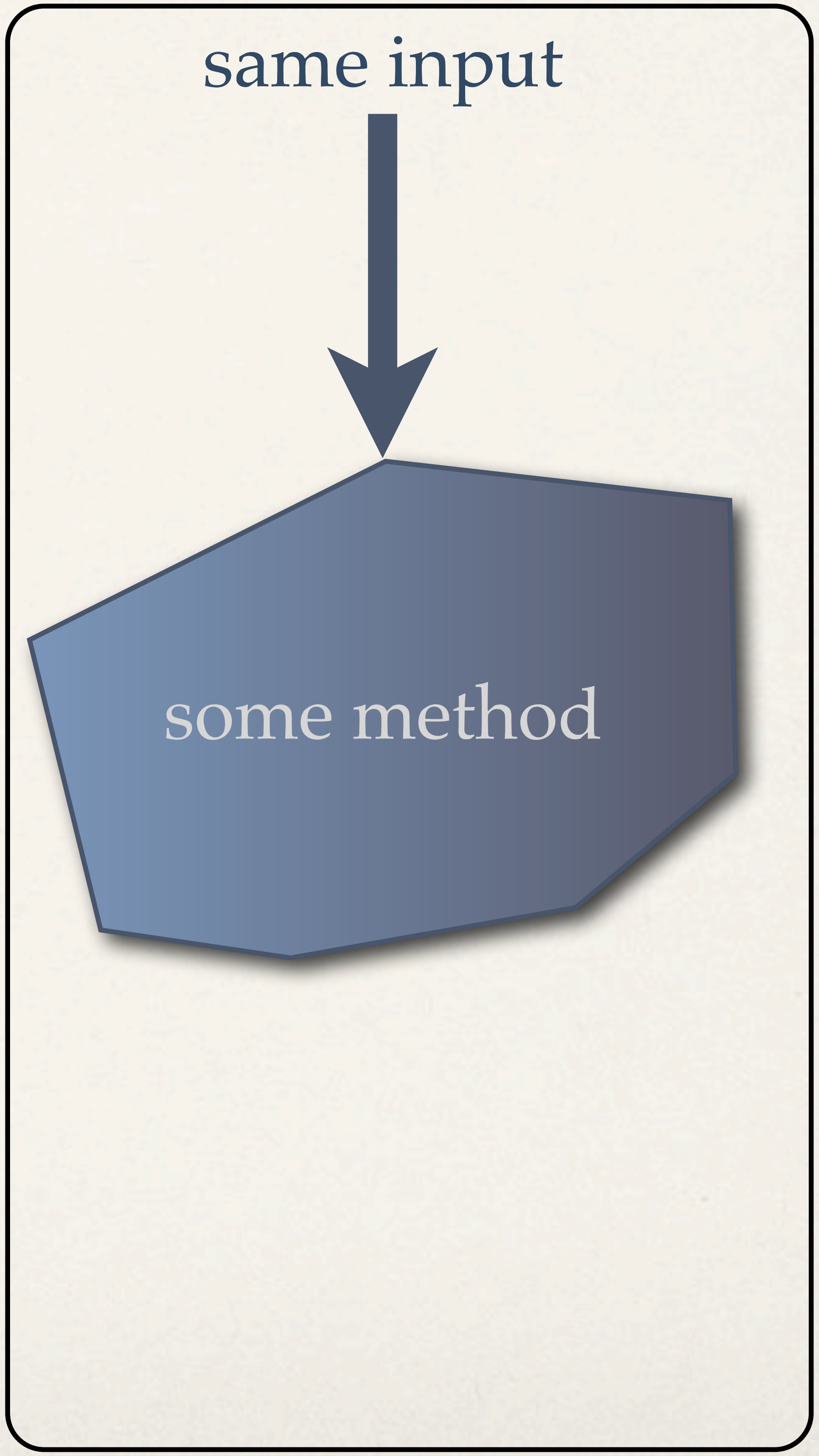
no (few) assignment statements
no (few) mutable objects.



One Definition of Functional Programming

So one definition of functional programming is

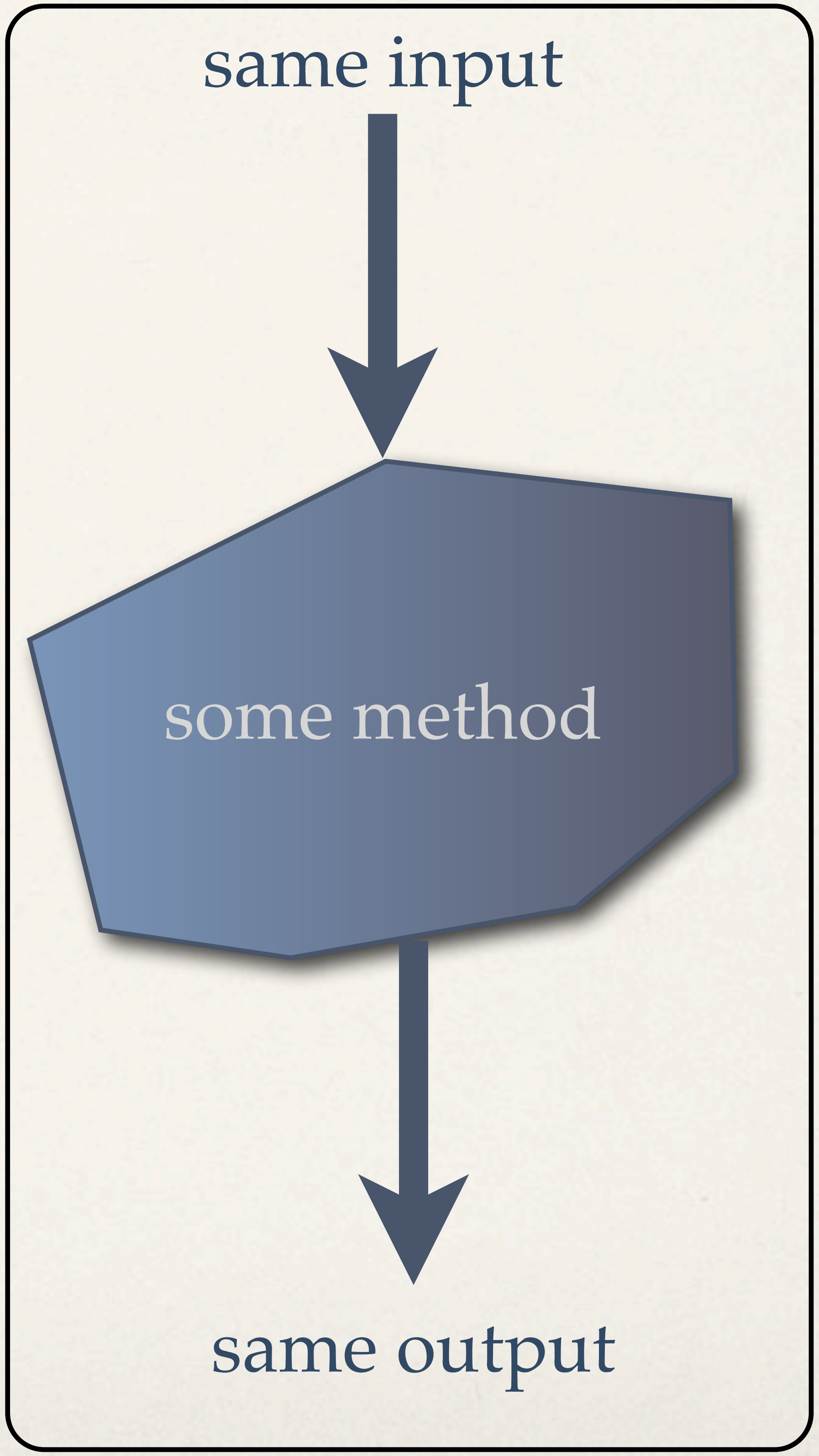
no (few) assignment statements
no (few) mutable objects.



One Definition of Functional Programming

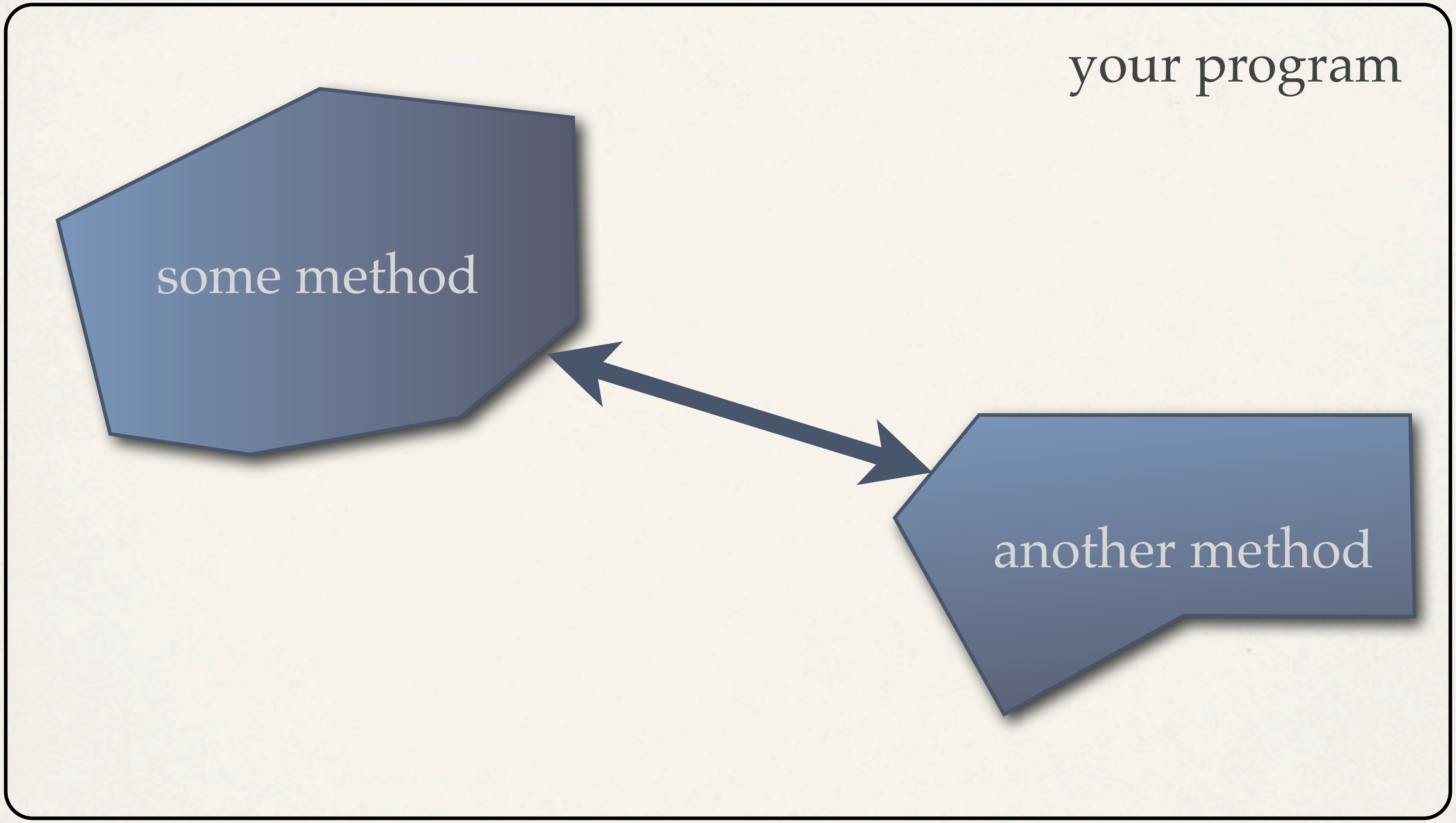
So one definition of functional programming is

no (few) assignment statements
no (few) mutable objects.

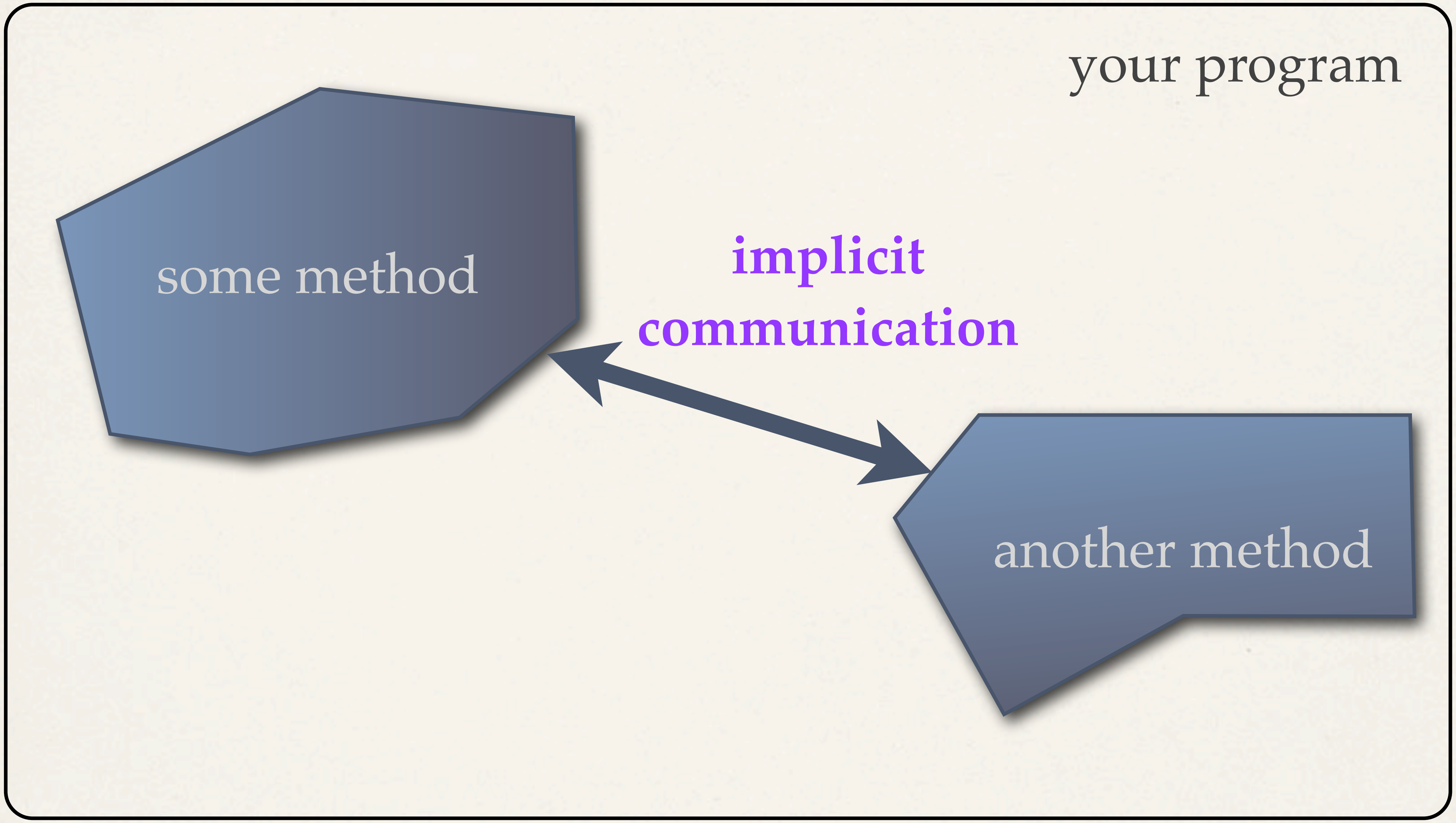


(most of the time)

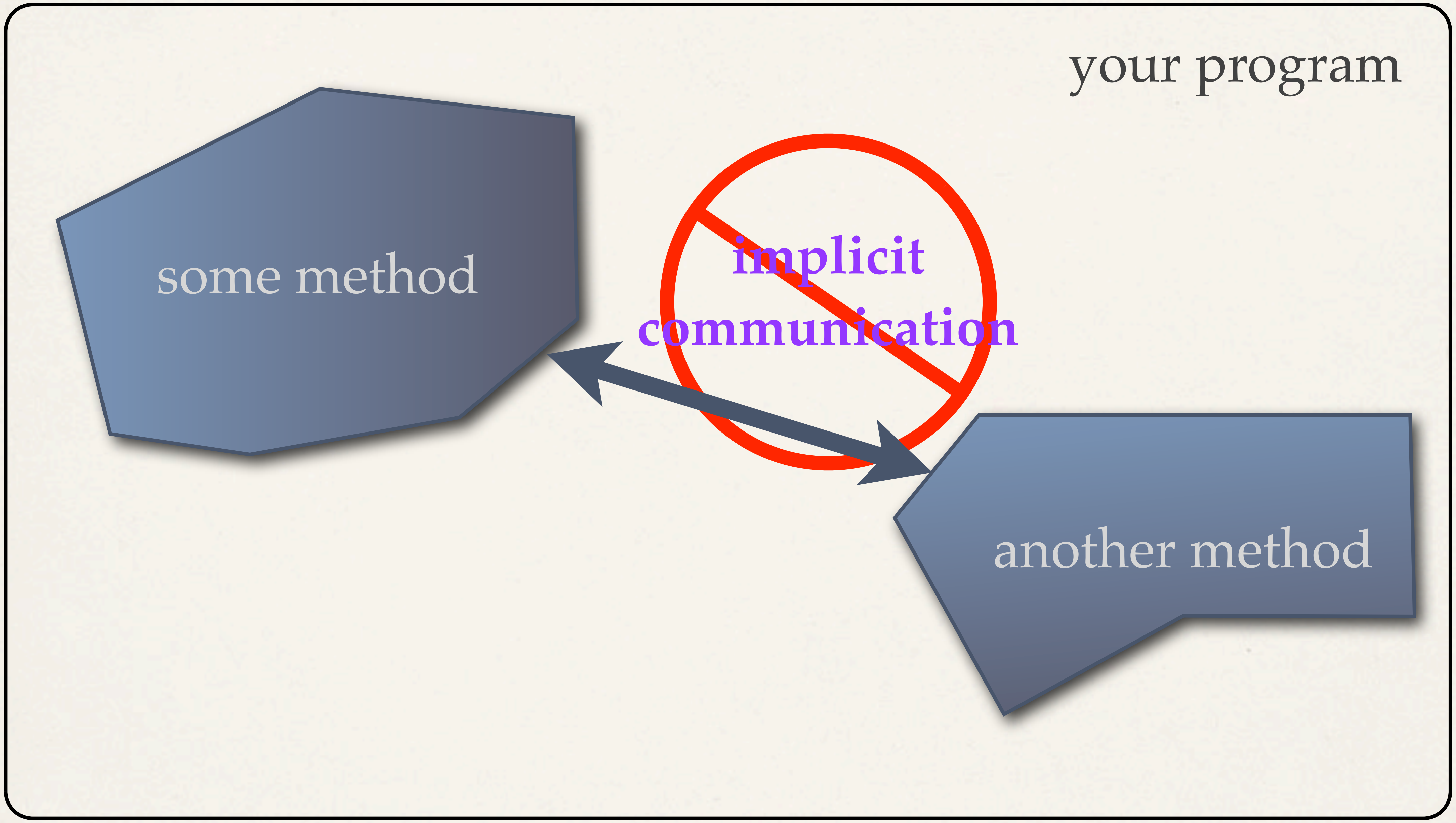
Another Definition



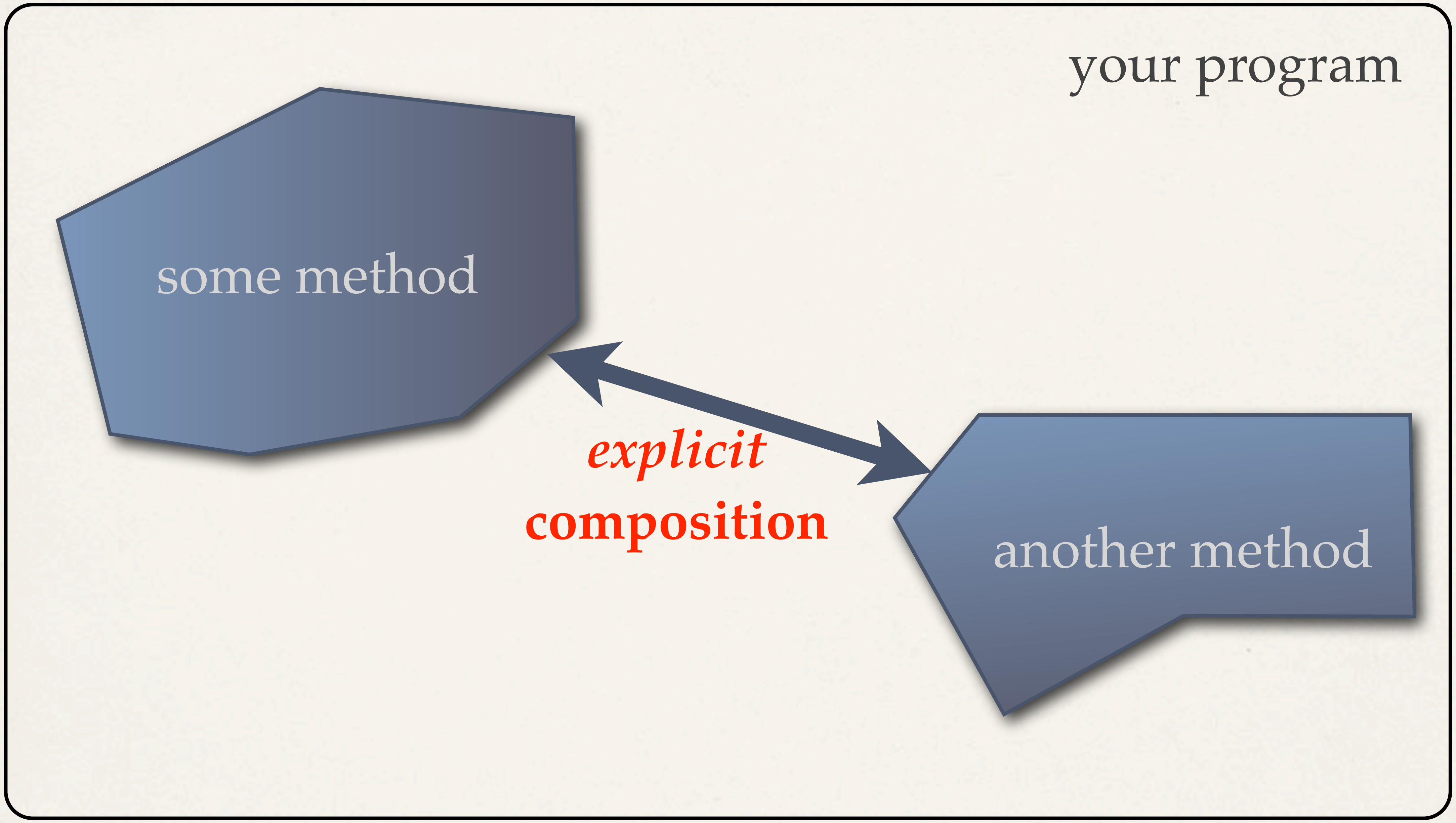
Another Definition



Another Definition



Another Definition

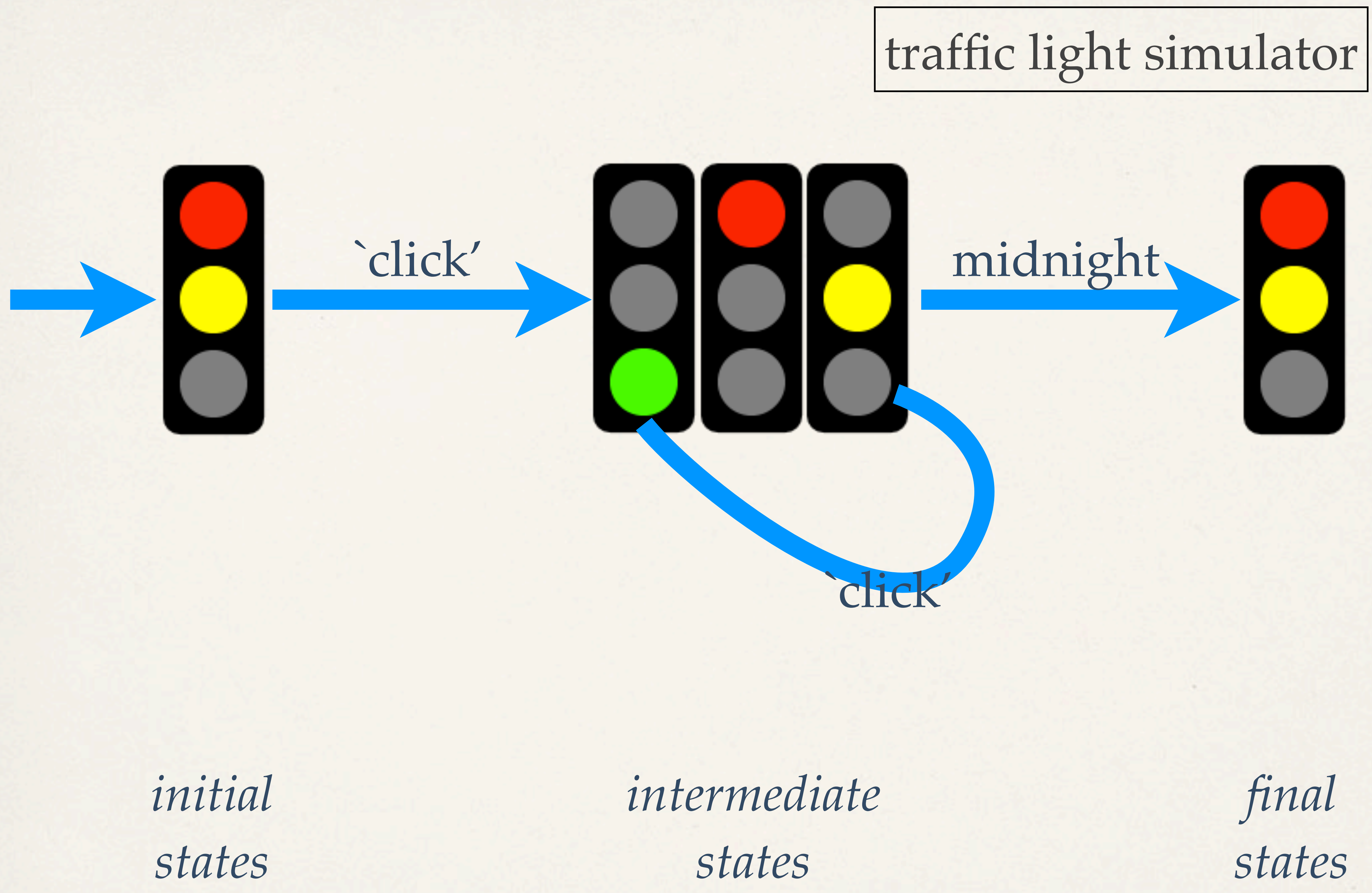


What does this mean *concretely*?

According to either definition, you can program functionally in any programming language.

According to either definition, you can program functionally in any programming language.

A **functional language** ensures that you don't accidentally cheat.


traffic light simulator
`click’
midnight
`click’
initial
states
intermediate
states
final
states

imperative OOPL

```
initial:  setToRed
onTick:   setTime
onClick:  nextColor
stopWhen: atMidnight,renderWarning
toDraw:   renderTrafficLight
```

```
type State = Color x Time
State current = ...
```

```
void setToRed() { ... }
void nextColor() { ... }
void renderTrafficLight() { ... }
void setTime() { ... }
boolean atMidnight() { ... }
void renderWarning() { ... }
```

imperative OOPL

```
initial:  setToRed
onTick:   setTime
onClick:  nextColor
stopWhen:atMidnight,renderWarning
toDraw:   renderTrafficLight
```

```
type State = Color x Time
State current = ...
```

```
void setToRed() { ... }
void nextColor() { ... }
void renderTrafficLight() { ... }
void setTime() { ... }
boolean atMidnight() { ... }
void renderWarning() { ... }
```

functional

```
initial:  setToRed
onTick:   setTime
onClick:  nextColor
stopWhen: atMidnight,renderWarning
toDraw:   renderTrafficLight
```

```
type State =
  Initial U Intermediate U Final
```

```
Initial setToRed()
State   nextColor(State current)
Image   renderLight(State current)
State   setTime(State current)
boolean atMidnight(State current) : Final
Image   renderWarning(Final current)
```

functional

```
initial:   setToRed
onTick:    setTime
onClick:   nextColor
stopWhen:  atMidnight,renderWarning
toDraw:    renderTrafficLight
```

```
type State =
  Initial U Intermediate U Final
```

```
Initial setToRed()
State   nextColor(State current)
Image   renderLight(State current)
State   setTime(State current)
boolean atMidnight(State current) : Final
Image   renderWarning(Final current)
```

explicit state transformations allow local reasoning

functional

```
initial:   setToRed
onTick:    setTime
onClick:   nextColor
stopWhen:  atMidnight,renderWarning
toDraw:    renderTrafficLight
```

```
type State =
  Initial U Intermediate U Final
```

```
Initial setToRed()
State   nextColor(State current)
Image   renderLight(State current)
State   setTime(State current)
boolean atMidnight(State current) : Final
Image   renderWarning(Final current)
```

explicit state
transformations
allow local reasoning

Imperative Programming

```
setToRed();
renderLight();
nextColor();
nextColor();
setTime();
renderLight();
nextColor();

if atMidnight()
 renderWarning()
else
  renderLight();
```

Imperative Programming

```
setToRed();
renderLight();
nextColor();
nextColor();
setTime();
renderLight();
nextColor();

if atMidnight()
 renderWarning()
else
  renderLight();
```

Functional Programming

```
State s1 = setToRed()
Image i1 = renderLight(s1)
State s2 = nextColor(s1)
State s3 = nextColor(s2)
State s4 = setTime(s3)
Image i4 = renderLight(s4)
State s5 = nextColor(s4)

Image i5 =
 atMidnight(s5) ?
   renderWarning(s5),
   renderLight(s5)
```

It all looks easy.

It all looks easy.

So what's the catch?

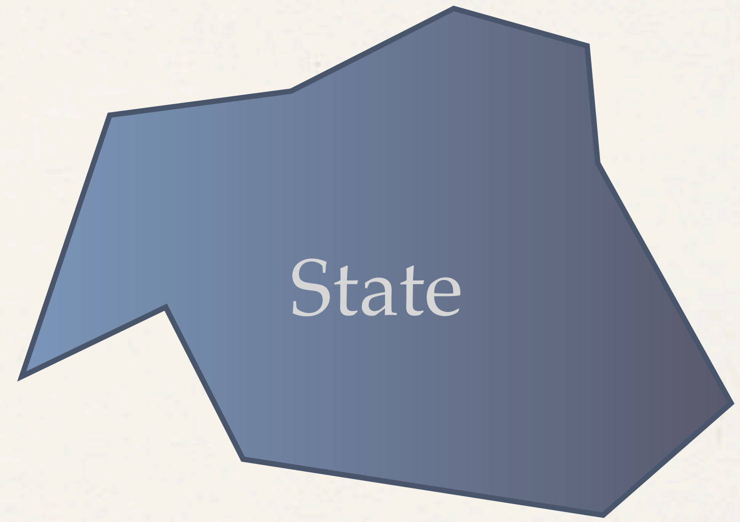
Imagine a state that uses a *record*, which contains *vector* in each slot, and each record contains *maps* that map *names* to *lists* of immutable data. And imagine that you want to equip the monster with a dagger.

```
type State   =
  { monsters : Vector<Monster>,
    fighter  : Status,
    turns    : Natural }
type Monster =
  Map<String,List<Weapon>>
type Weapon  = ...
```

Imagine a state that uses a *record*, which contains *vector* in each slot, and each record contains *maps* that map *names* to *lists* of immutable data. And imagine that you want to equip the monster with a dagger.

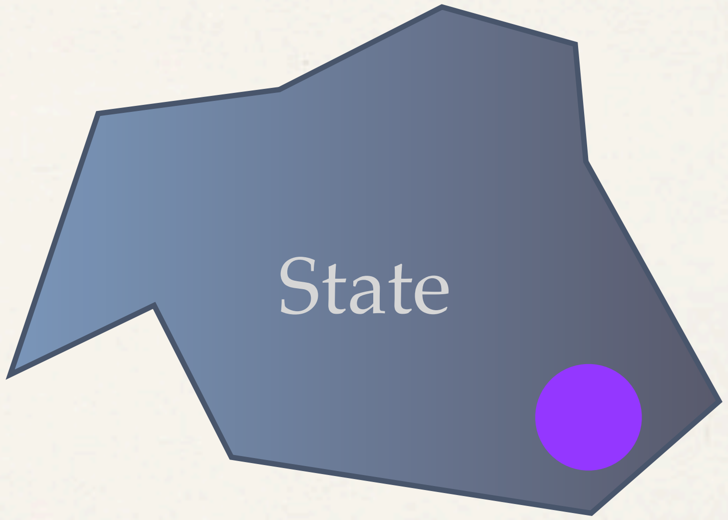
```
type State   =
  { monsters : Vector<Monster>,
    fighter  : Status,
    turns    : Natural }
type Monster =
  Map<String,List<Weapon>>
type Weapon  = ...
```

Imagine a state that uses a *record*, which contains *vector* in each slot, and each record contains *maps* that map *names* to *lists* of immutable data. And imagine that you want to equip the monster with a dagger.



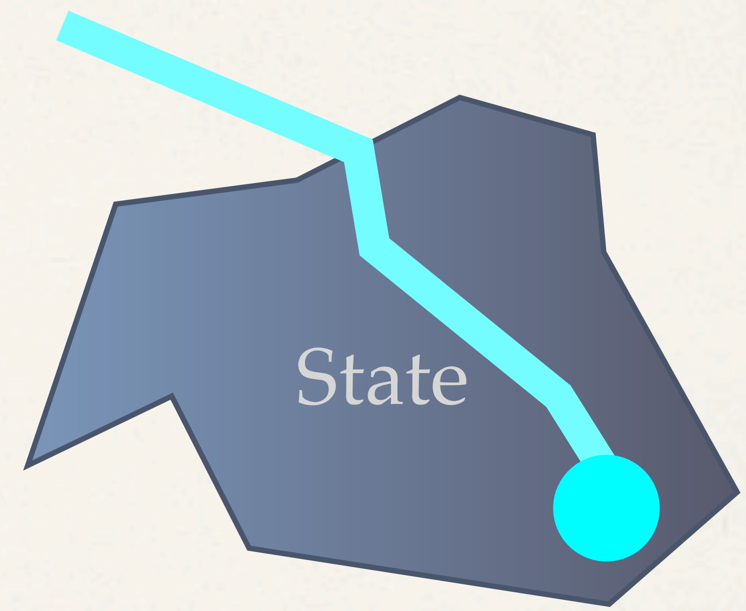

```
type State   =
  { monsters : Vector<Monster>,
    fighter  : Status,
    turns    : Natural }
type Monster =
  Map<String,List<Weapon>>
type Weapon  = ...
```

Imagine a state that uses a *record*, which contains *vector* in each slot, and each record contains *maps* that map *names* to *lists* of immutable data. And imagine that you want to equip the monster with a dagger.




```
type State   =
  { monsters : Vector<Monster>,
    fighter  : Status,
    turns    : Natural }
type Monster =
  Map<String,List<Weapon>>
type Weapon  = ...
```

Imagine a state that uses a *record*, which contains *vector* in each slot, and each record contains *maps* that map *names* to *lists* of immutable data. And imagine that you want to equip the monster with a dagger.




```
type State   =
  { monsters : Vector<Monster>,
    fighter  : Status,
    turns    : Natural }
type Monster =
  Map<String,Li <Weapon>>
type Weapon  = ...
```

```
type State   =
  { monsters : Vector<Monster>,
    fighter  : Status,
    turns    : Natural }
type Monster =
  Map<String,List<Weapon>>
type Weapon  = ...
```

Imperative Programming

```
state.monsters[i][“orc”].addList(“dagger”);
```

```
type State    =
  { monsters : Vector<Monster>,
    fighter  : Status,
    turns    : Natural }
type Monster =
  Map<String,List<Weapon>>
type Weapon  = ...
```

Functional Programming

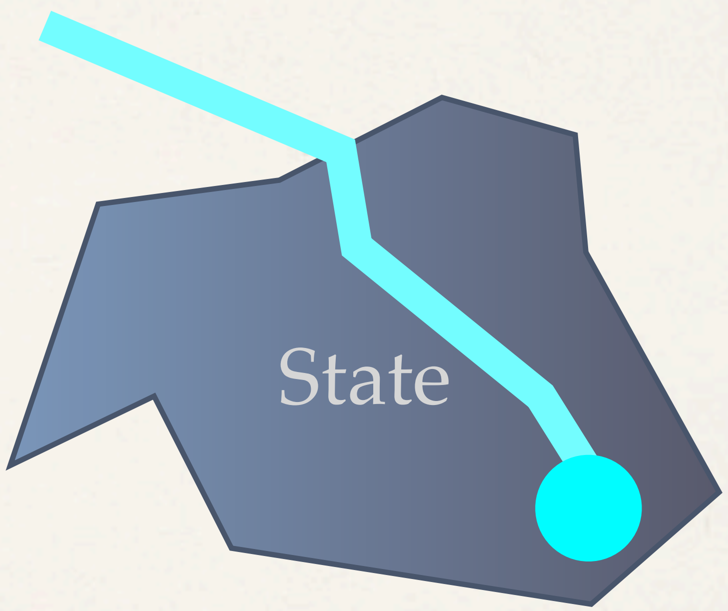
```
Context x List<Weapon> <c,w> = unzip(state);
List<Weapon> new_list = addList(“dagger”);
zip(c,new_list);
```

```
type State    =
  { monsters : Vector<Monster>,
    fighter  : Status,
    turns    : Natural }
type Monster =
  Map<String,L t<Weapon>>
type Weapon  = ...
```

complex operations

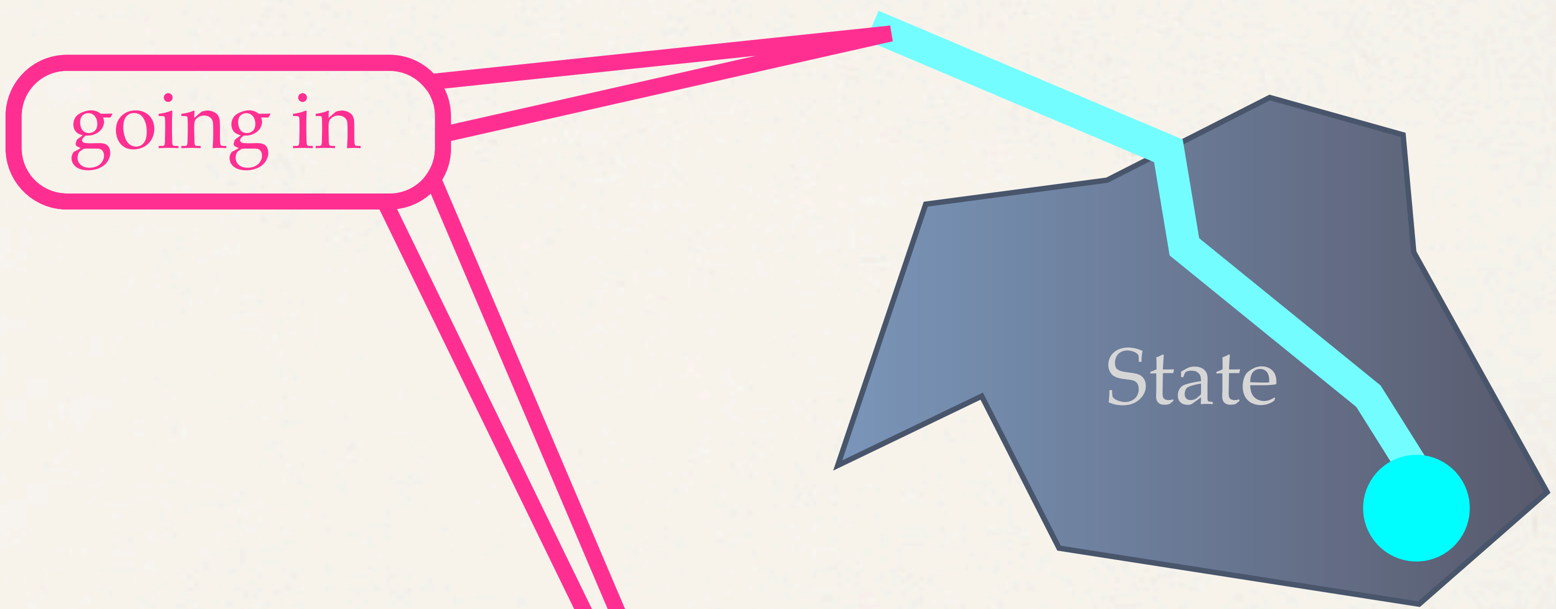
Functional Programming

```
Context x List<Weapon> <c,w> = unzip(state);
List<Weapon> new_list = addList("dagger");
zip(c,new_list);
```



Functional Programming

```
Context x List<Weapon> <c,w> = unzip(state);
List<Weapon> new_list = addList(“dagger”);
zip(c,new_list);
```



Functional Programming

```
Context x List<Weapon> <c,w> = unzip(state);
List<Weapon> new_list = addList(“dagger”);
zip(c,new_list);
```

Functional Programming

```
Context x List<Weapon> <c,w> = unzip(state);
List<Weapon> new_list = addList(“dagger”);
zip(c,new_list);
```

a problem of
expressiveness
("notational" overhead)

a problem of
algorithmics
("slow" performance)

a problem of
expressiveness
("notational" overhead)

a problem of
algorithmics
("slow" performance)

solution 1: zip/unzip &
functional data structures

a problem of expressiveness ("notational" overhead)

a problem of algorithmics ("slow" performance)

solution 1: zip/unzip & functional data structures

solution 2: **monads** and other fancy constructs a problem of expressiveness ("notational" overhead)

a problem of algorithmics ("slow" performance)

solution 1: zip/unzip & functional data structures

solution 2: **monads** and other fancy constructs

solution 3: "bite the bullet" -- allow mutation in FP and FPLs

a problem of expressiveness

solution 1: **functional data structures** do not truly eliminate notational overhead

a problem of expressiveness

solution 1: **functional data structures** do not truly eliminate notational overhead

solution 2: **monads** gets close. The remaining type overhead is arguably an **advantage**. It helps tame side effects.

a problem of expressiveness

solution 1: **functional data structures** do not truly eliminate notational overhead

solution 2: **monads** gets close. The remaining type overhead is arguably an **advantage**. It helps tame side effects.

solution 3: **mutation** in FP and FPLs eliminates the problem as much as desired. **Danger**: it opens the flood gate for careless programmers.

a problem of algorithmics: **theory**

functional data structures: we have no proof that **functional data structures** are as efficient as imperative programming.

a problem of algorithmics: **theory**

functional data structures: we have no proof that **functional data structures** are as efficient as imperative programming.

monads: they are implemented imperatively. Period.

a problem of algorithmics: **theory**

functional data structures: we have no proof that **functional data structures** are as efficient as imperative programming.

monads: they are implemented imperatively. Period.

assignments in FPLs: they eliminates the problem as much as desired. **Danger: it tempts programmers to use mutation too much.**

a problem of algorithmics: **in practice**

mix and match: people tend to combine monads or mutation with functional data structures.

a problem of algorithmics: **in practice**

mix and match: people tend to combine monads or mutation with functional data structures.

measuring end-to-end performance: efficiency is **in practice indistinguishable** from imperative programming.

a problem of algorithmics: **in practice**

mix and match: people tend to combine monads or mutation with functional data structures.

measuring end-to-end performance: efficiency is **in practice indistinguishable** from imperative programming.

catch: it takes experience to reach this point.

About Myself

I am not neutral.

I am not a purist.

About Myself

I am not neutral.

I am not a purist.

research: objects, assignment statements, design patterns, web servlets, continuations, modules, functional I/O, etc.

programming: mostly functional, but OO and imperative as neede

About Myself

I am not neutral.

I am not a purist.

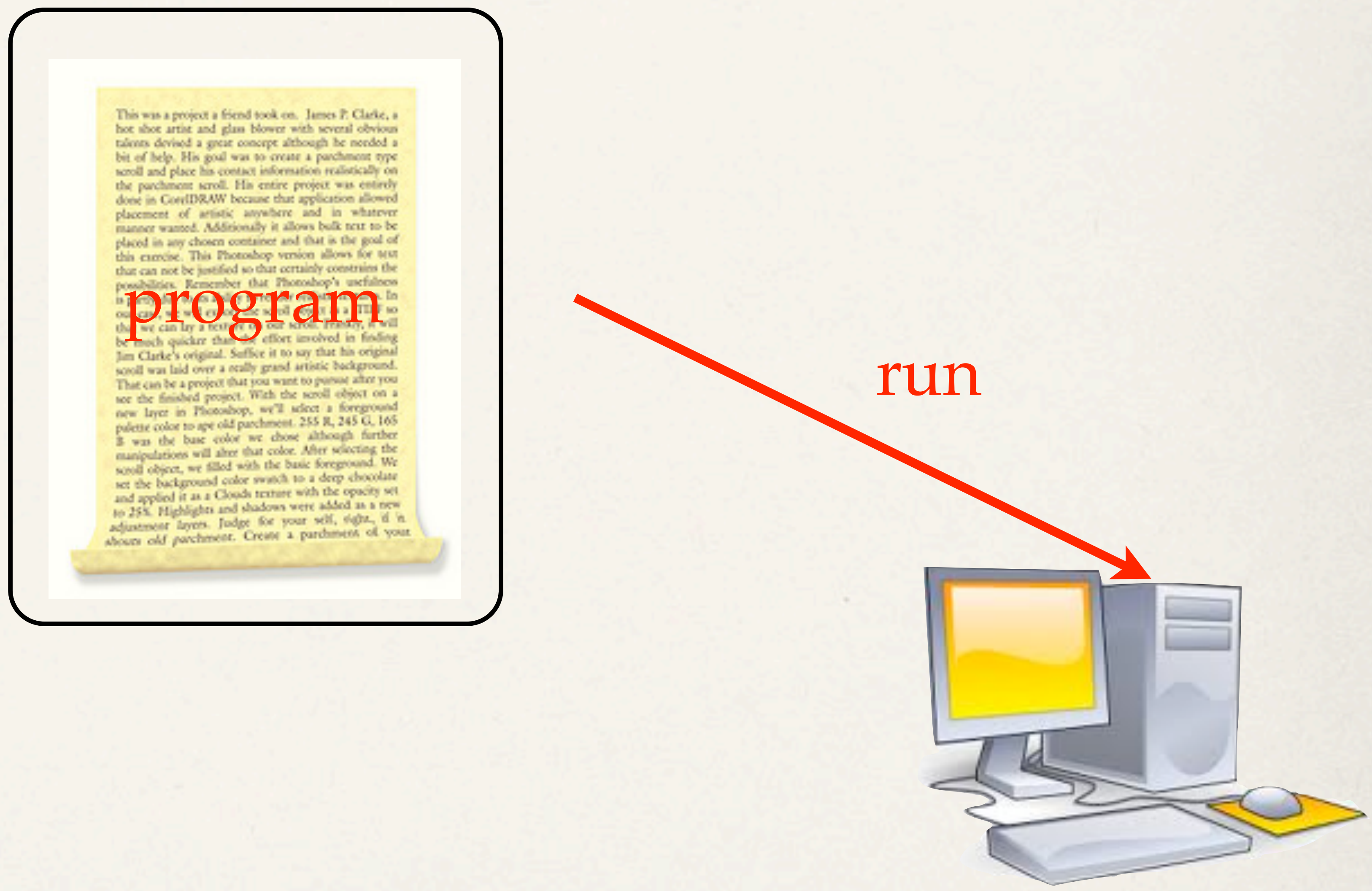
research: objects, assignment statements, design patterns, web servlets, continuations, modules, functional I/O, etc.

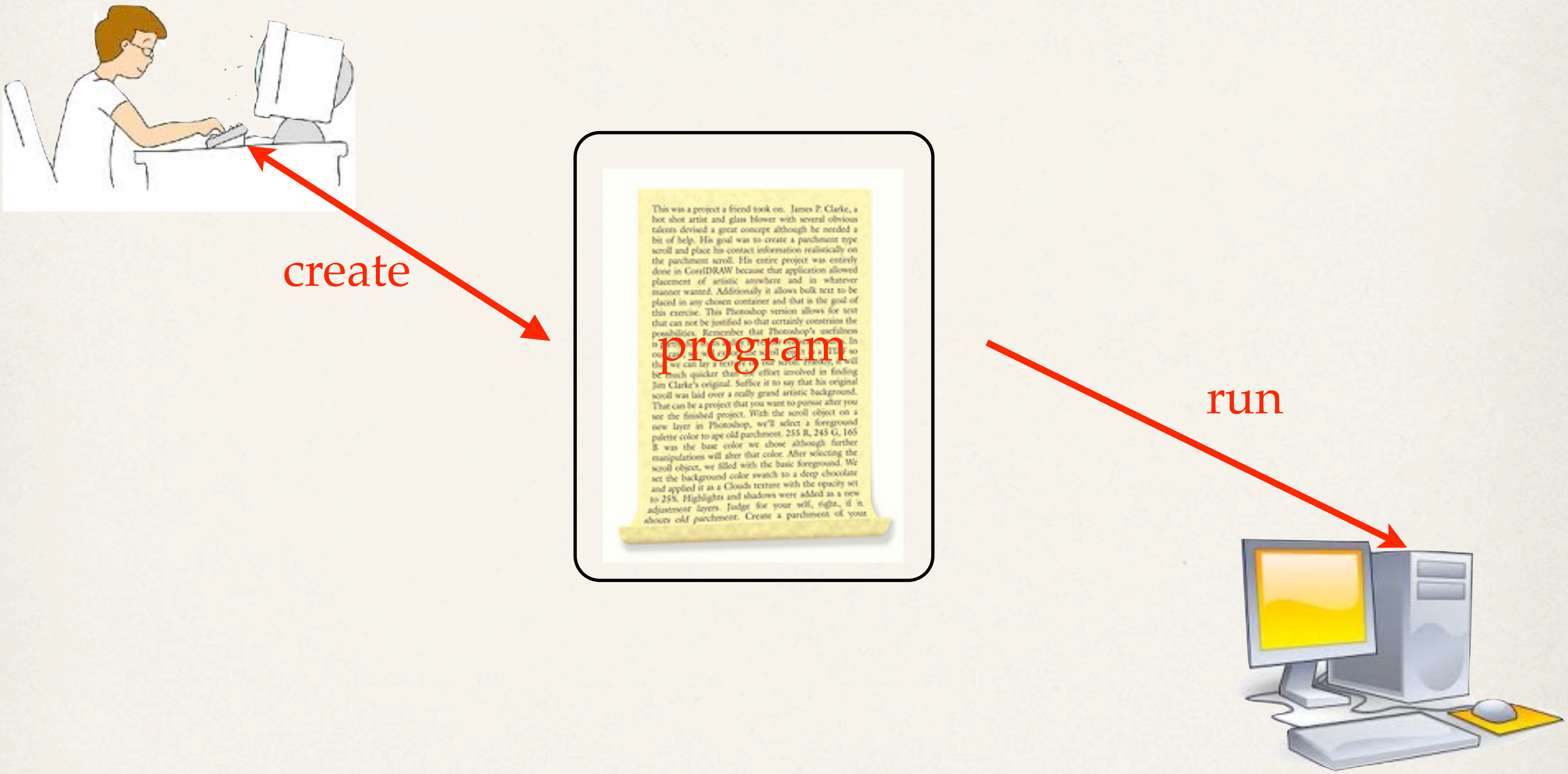
programming: mostly functional, but OO and imperative as neede

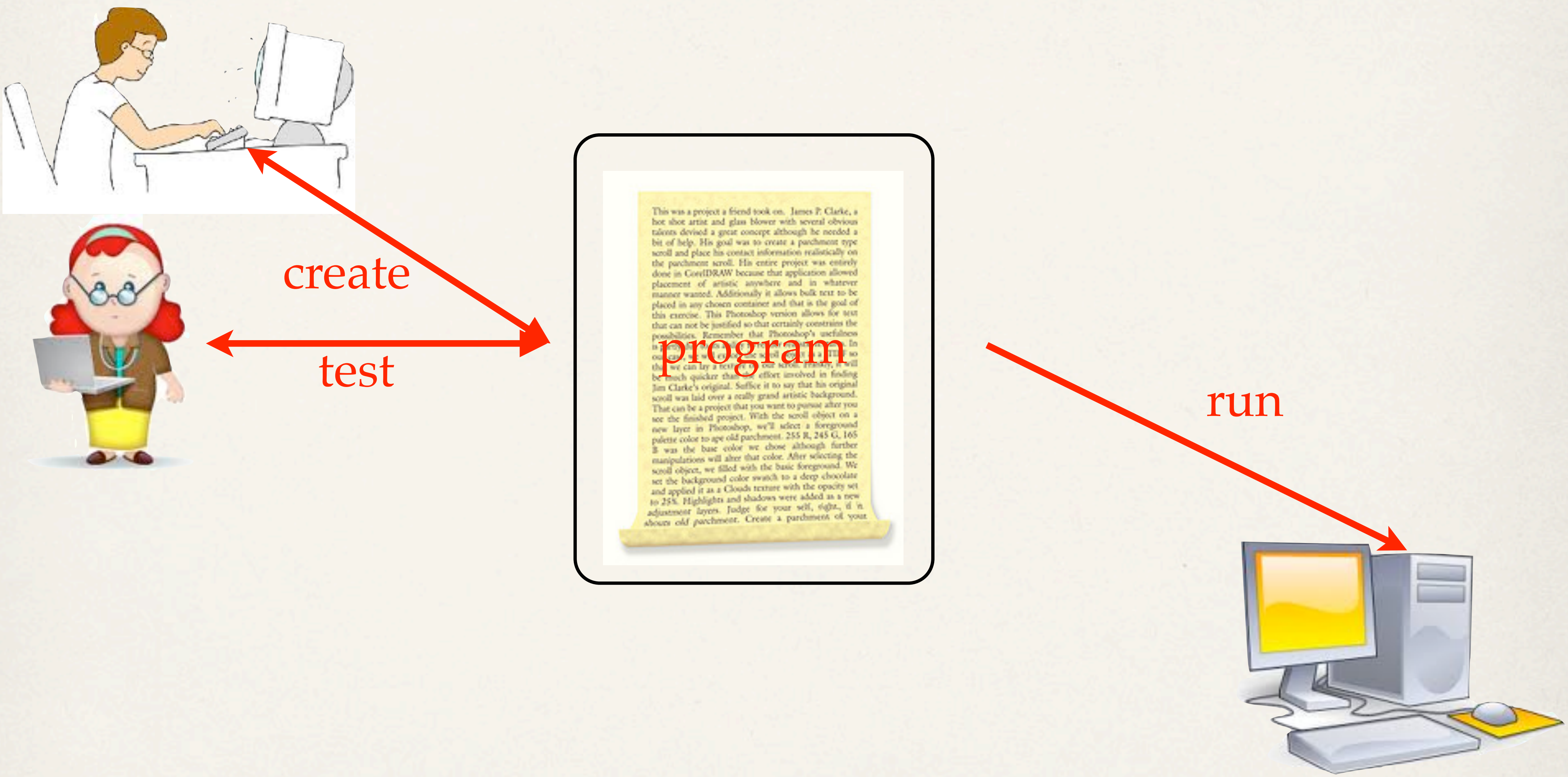
teaching: start with functional programming in **purely functional, strict** languages for 10,000s of students, starting in 7th grade all the way to M.S.

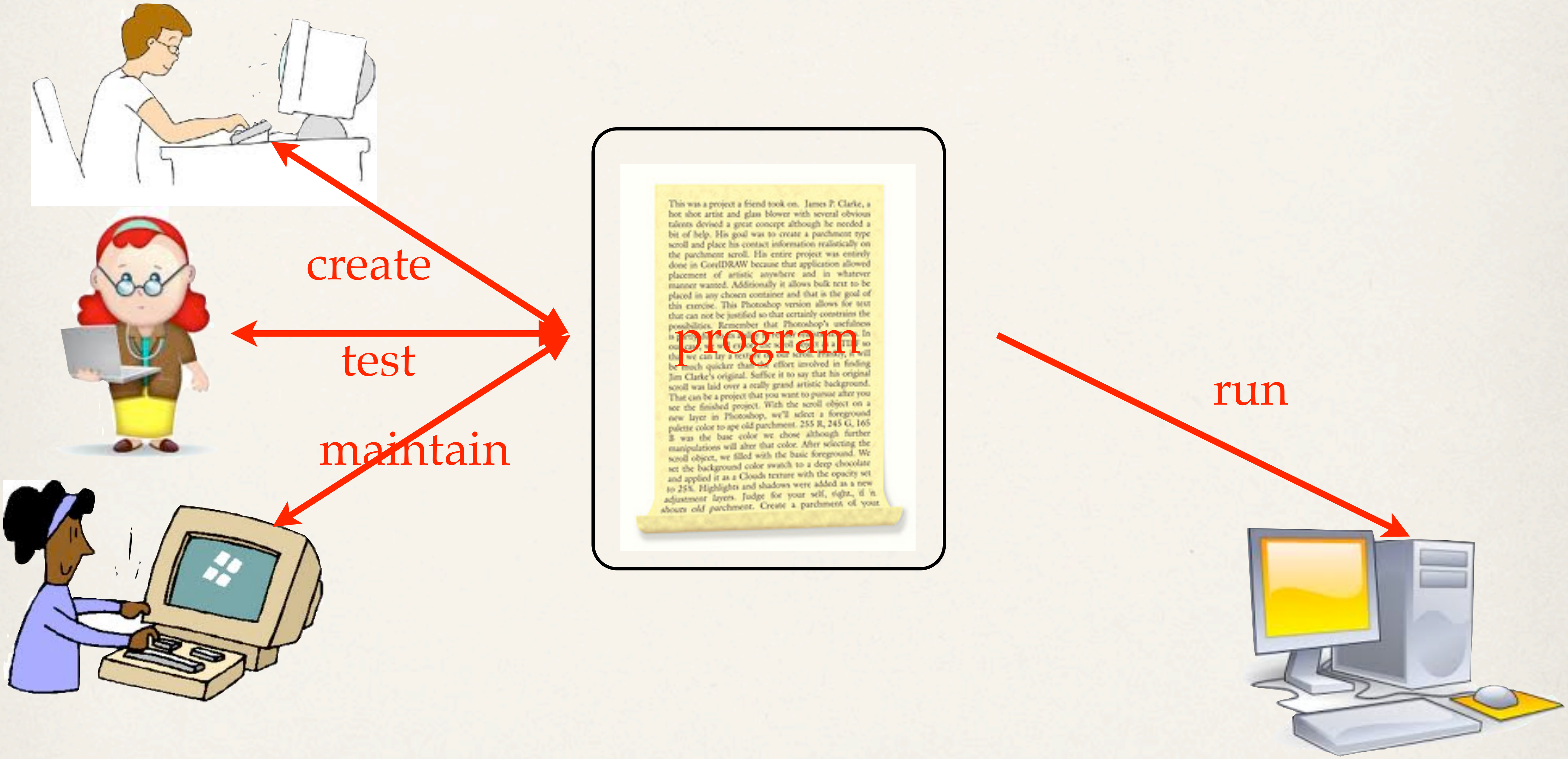
Why Functional Programming?
Why a Functional Programming Language?


program

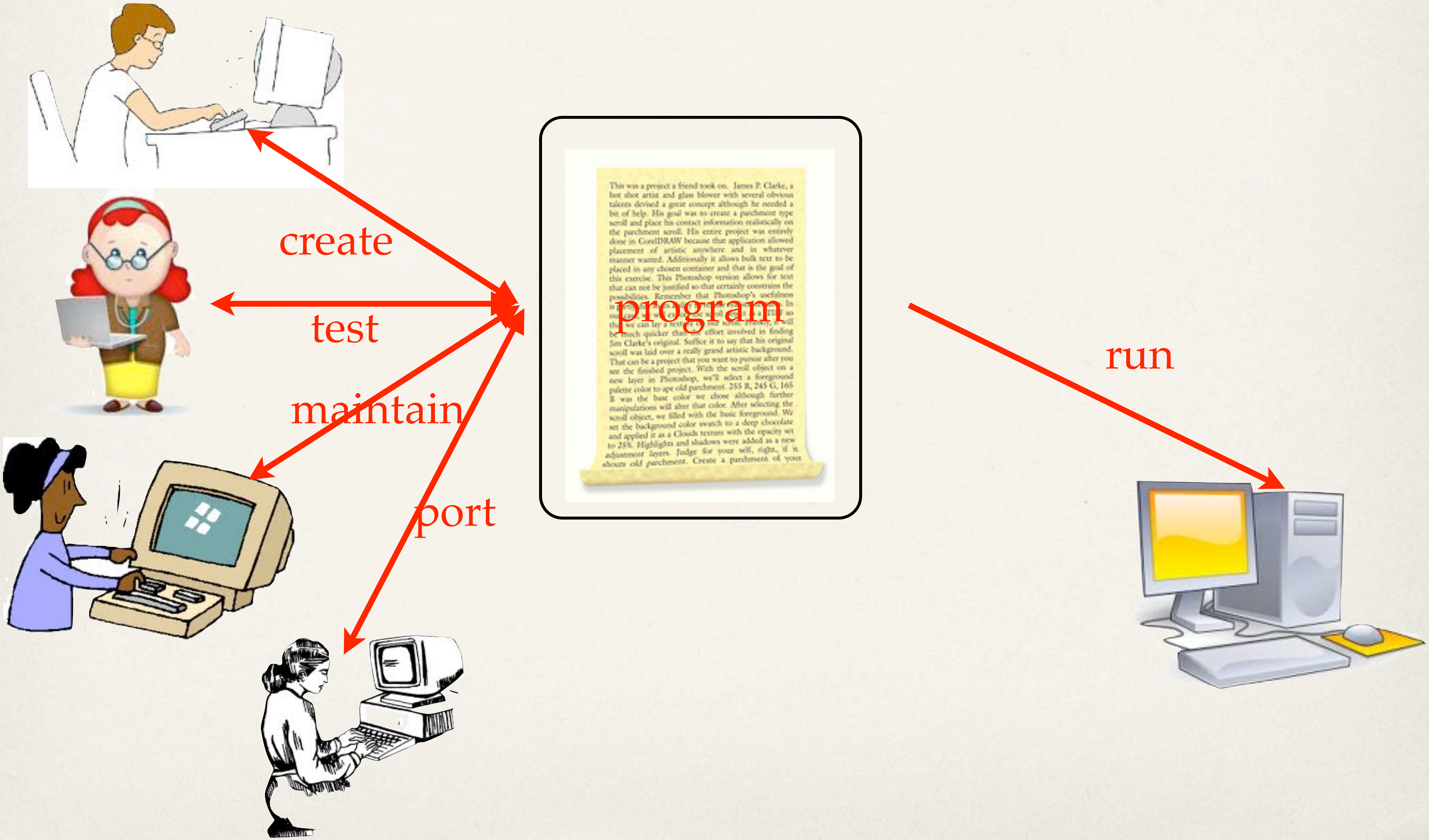

program
run


create
program
run

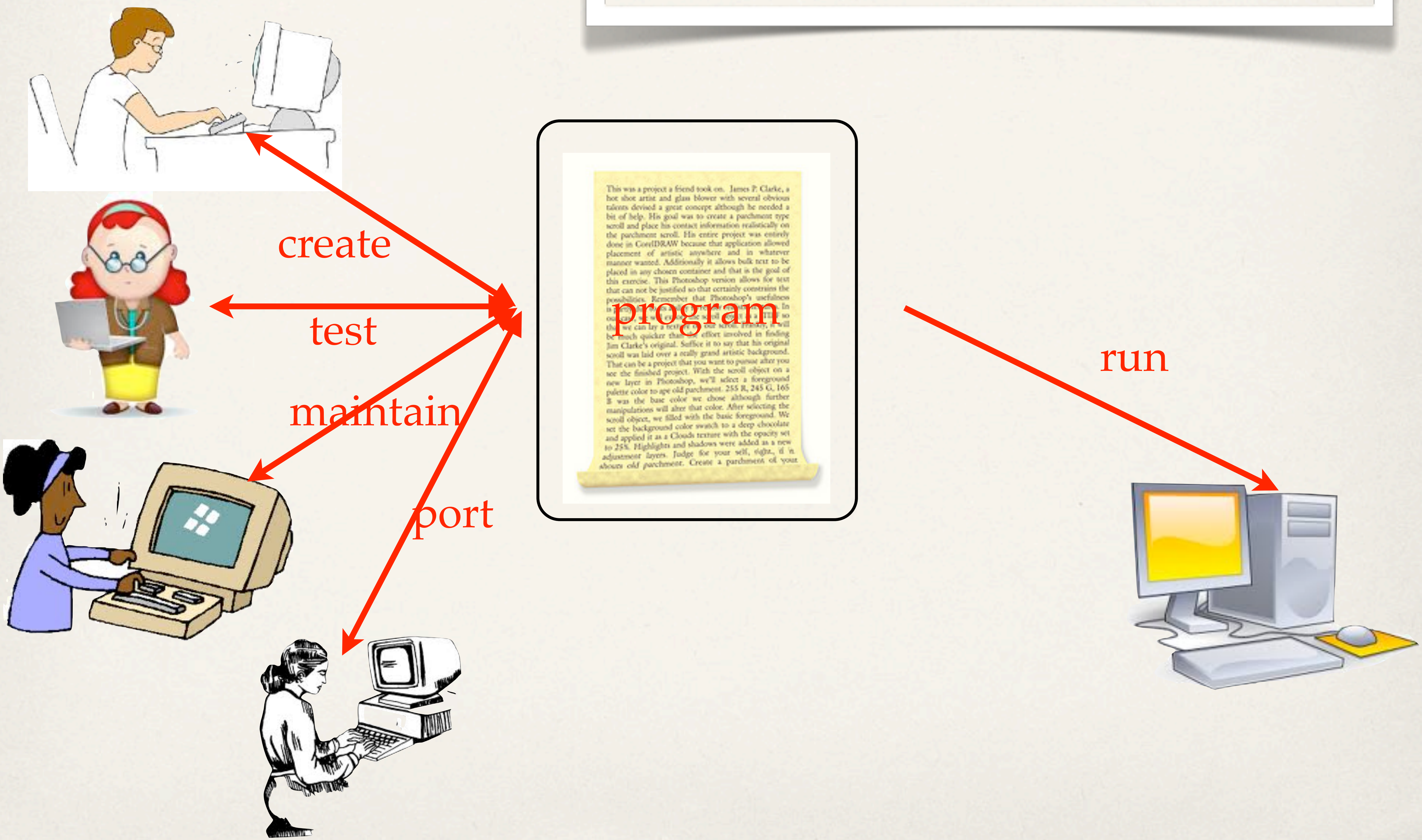

create
test
program
run



create
test
maintain
program
run


create
test
maintain
port
program
run

Programs must be written for people to read, and only incidentally for machines to execute. *from*: Abelson & Sussman, *SICP*



The cost of software is a function of the cost of programmer communication.

The cost of software is a function of the cost of programmer communication.

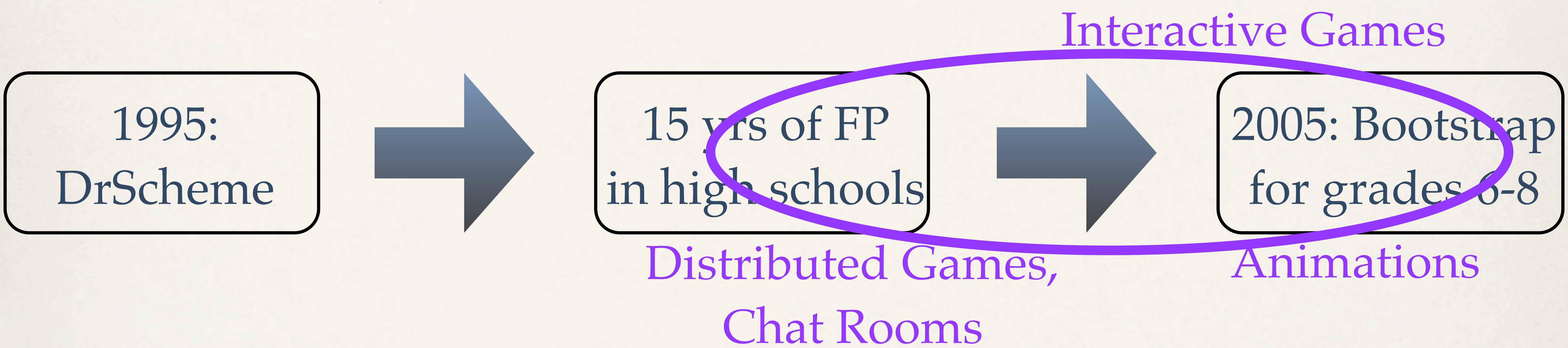
Functional programming and better functional programming languages greatly reduce the cost of communication and thus the cost of software.

There are many sides to the cost story:
human, training, technical.

1995:
DrScheme

15 yrs of FP
in high schools

1995: DrScheme → 15 yrs of FP in high schools → 2005: Bootstrap for grades 6-8


1995: DrScheme
15 yrs of FP in high schools
2005: Bootstrap for grades 6-8
Interactive Games
Distributed Games, Chat Rooms
Animations

If these students can do FP, *it is easy*

1995: DrScheme

15 yrs of FP in high schools

2005: Bootstrap for grades 6-8

Interactive Games

Distributed Games, Chat Rooms

Animations

If these students can do FP, *it is easy*

Interactive Games

1995: DrScheme → 15 yrs of FP in high schools → 2005: Bootstrap for grades 6-8

Distributed Games, Chat Rooms

Animations

Why did these students improve so much in math? Why do they *pass the state test?*

Northeastern University:
5 year programs, including *three* 6-month supervised co-op positions in industry

2001:
conventional first-year introduction to OO (Java) programming and discrete math

Northeastern University:
5 year programs, including *three* 6-month supervised co-op positions in industry

2001:
conventional first-year introduction to OO (Java) programming and discrete math

↓

only 1/3 of the students get co-op positions that involve programming

Northeastern University:
5 year programs, including *three* 6-month supervised co-op positions in industry

2001:
conventional first-year introduction to OO (Java) programming and discrete math

2006:
functional-then-OO first-year introduction to programming and discrete math

only 1/3 of the students get co-op positions that involve programming

Northeastern University:
5 year programs, including *three* 6-month supervised co-op positions in industry

2001:
conventional first-year introduction to OO (Java) programming and discrete math

2006:
functional-then-OO first-year introduction to programming and discrete math

only 1/3 of the students get co-op positions that involve programming

over 2/3 of the students get co-op positions that involve programming

Northeastern University:
5 year programs, including *three* 6-month supervised co-op positions in industry

Graduate dean: "Industry asks, why can't your MS students program as well as your undergraduates?"

programming and discrete math

and discrete math

only 1/3 of the students get co-op positions that involve programming

over 2/3 of the students get co-op positions that involve programming

Northeastern University:
2 year MS programs (one co-op)
now comes with a 4-month introduction
to Functional Program Design
called “Bootcamp”

Teaching FP has a highly beneficial effect on programmers

even if they don't end up programming that way.

Teaching FP has a highly beneficial effect on programmers
even if they don't end up programming that way.

Time to look at some technical points.

From mathematical models to programs

Mathematics

$$f(x,y,z) = \ldots \int f(g(x), h(y), i(z,x))\, dx \ldots$$

From mathematical models to programs

Mathematics

$$f(x,y,z) = \;\ldots \int f(g(x), h(y), i(z,x))\, dx \ldots$$

Program

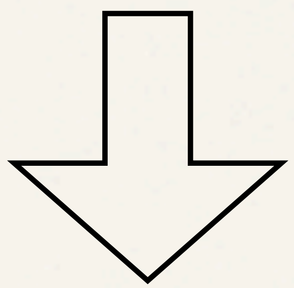
```
f(x,y,z) =
    ...integrate(f(g(x), h(y), i(z,x)))...
```

From mathematical models to programs

Mathematics

$f(x,y,z) =$
$\quad \ldots \int \; f(g(x), h(y), i(z,x)) \, dx \ldots$

Yes, they basically look the same and it is easy to convince yourself that they mean the same.



Program

```
f(x,y,z) =
    ...integrate(f(g(x), h(y), i(z,x)))...
```

From algebraic types to functions

```
type Contract =
    zero
  | scale of Contract * Double
  | and   of Contract * Contract
  | until of Contract * Observation
  | ...
```

From algebraic types to functions

```
type Contract =
    zero
  | scale of Contract * Double
  | and   of Contract * Contract
  | until of Contract * Observation
  | ...
```

algebraic types translate directly into a function outline

```
fun Number value(Contract c, Model m) =
 case c
   zero             -> ...
 | scale(base,fac) -> . value(base,m) .
 | and(c1,c2)       -> . value(c1,m) ...
                       . value(c2,m) ...
 | until(base,obs) -> . value(base,m) ...
 | ...
```

From algebraic types to functions

```
type Contract =
   zero
```

Imagine all the OO design patterns you need in Java.

```
 | until of Contract * Observation
 | ...
```

algebraic types translate directly into a function outline

```
fun Number value(Contract c, Model m) =
 case c
   zero              -> ...
 | scale(base,fac) -> . value(base,m) .
 | and(c1,c2)       -> . value(c1,m) ...
                       . value(c2,m) ...
 | until(base,obs) -> . value(base,m) ...
 | ...
```

From function signatures to understanding

Imperative

```
type State = Color x Time
```

```
void setToRed() { ... }
void nextColor() { ... }
void renderTrafficLight() { ... }
void setTime() { ... }
boolean atMidnight() { ... }
void renderWarning() { ... }
```

From function signatures to understanding

Imperative

```
type State = Color x Time
```

```
void setToRed() { ... }
void nextColor() { ... }
void renderTrafficLight() { ... }
void setTime() { ... }
boolean atMidnight() { ... }
void renderWarning() { ... }
```

Functional

```
type State = Initial U Intermediate U Final
```

```
Initial setToRed()
State   nextColor(State current)
Image   renderLight(State current)
State   setTime(State current)
boolean atMidnight(State current) : Final
Image   renderWarning(Final current)
```

From function signatures to understanding

Imperative

```
type State = Color x Time
```

```
void setToRed() { ... }
void nextColor() { ... }
void renderTrafficLight() { ... }
void setTime() { ... }
boolean atMidnight() { ... }
void renderWarning() { ... }
```

Functional

```
type State = Initial U Intermediate U Final
```

```
Initial   setTo
State     nextColor(State current)
Image     renderLight(State current)
State     setTime(State current)
boolean   atMidnight(State current) : Final
Image     renderWarning(Final current)
```

Type signatures convey a lot of information.

From function signatures to understanding

Imperative

```
type State = Color x Time
```

```
void setToRed() { ... }
void nextColor() { ... }
void renderTrafficLight() { ...
void setTime() { ... }
boolean atMidnight() { ... }
void renderWarning() { ... }
```

VOID conveys nothing.

Functional

```
type State = Initial U Intermediate U Final
```

```
Initial setTo
State   nextColor(State current)
Image   renderLight(State current)
State   setTime(State current)
boolean atMidnight(State current) : Final
Image   renderWarning(Final current)
```

Type signatures convey a lot of information.

From function signatures to understanding

Imperative

```
type State = Color x Time
```

```
void setToRed() { ... }
void nextColor() { ... }
void renderTrafficLight() { ...
void setTime() { ... }
boolean atMidnight() { ... }
```

VOID conveys nothing.

Subtract $10 for every VOID return type in your programmers code.

Functional

```
type State = Initial U Intermediate U Final
```

```
Initial setTo...
State   nextColor(State current)
Image   renderLight(State current)
State   setTime(State current)
boolean atMidnight(State current) : Final
Image   renderWarning(Final current)
```

Type signatures convey a lot of information.

Imperative

```
test case {
    setUpForSetTime();
    setTime();
    testCurrentState(expectedState);
    testFrameConditions();
    tearDownSetTime()
}
```

From functions to testing

Imperative

```
test case {
    setUpForSetTime();
    setTime();
    testCurrentState(expectedState);
    testFrameConditions();
    tearDownSetTime()
}
```

Functional

```
test case {
 compare(setTime(someState),expectedState);
}
```

From functions to testing

Imperative

```
test case {
    setUpForSetTime();
    setTime();
    testCurrentState(expectedState);
    testFrameConditions();
    tearDownSetTime()
}
```

Functional

```
test case {
  compare(setTime(someState), expectedState);
}
```

state is transferred explicitly
and can be understood in isolation

From functions to testing

Imperative

```
test case {
    setUpForSetTime();
    setTime();
    testCurrentState(expectedState);
```

Tests in the *functional world* become "one liners". And that works for compositions, too.

Functional

```
test case {
  compare(setTime(someState), expectedState);
}
```

state is transferred explicitly
and can be understood in isolation

Function Composition in Action

```
search_good_solution(
   criteria,
   generate_all_solutions(model, state0));
```

```
search_winning_move(
    improve_likelihood(current_state),
    generate_all_moves(model, state0));
```

```
search_winning_move(
    improve_likelihood(current_state),
    generate_all_moves(model, state0));
```

all?

Yes, on demand.
Lazy data structures enable
a powerful, yet simple
compositional style

```
search_winning_move(
    improve_likelihood(current_state),
    generate_all_moves(model, state0));
```

all?


```
search_winning_move(
    improve_likelihood(current_state),
    generate_all_moves(model, state0));
```


```
search_winning_move(
    improve_likelihood(current_state),
    generate_all_moves(model, state0));
```

evaluate *some* nodes, not all

Imperative OOP can express this idea, but only in extremely ugly ways, too ugly for this slide.


```
search_winning_move(
    improve_likelihood(current_state),
    generate_all_moves(model, state0));
```

evaluate *some* nodes, not all

Function composition is pervasive,
even in the strict world.

Financial Contracts as Functional Compostion

Combinator DSL

```
type Contract ... Observation ... Currency
```

```
fun Contract zero() ...
fun Contract one(Currency c) ...
fun Contract when(Obs t, Contract c) ...
fun Contract scale(Double s, Contract c)...
fun Observation at(Date d) : Obs ...
```

Financial Contracts as Functional Compostion

Combinator DSL

```
type Contract ... Observation ... Currency
```

```
fun Contract zero() ...
fun Contract one(Currency c) ...
fun Contract when(Obs t, Contract c) ...
fun Contract scale(Double s, Contract c)...
fun Observation at(Date d) : Obs ...
```

```
fun zero_coupon_discount_bond(t,x,k) =
  when (at t) (scale (konst x) (one k))
```

Financial Contracts as Functional Compostion

Combinator DSL

```
type Contract ... Observation ... Currency
```

```
fun Contract zero() ...
fun Contract one(Currency c) ...
fun Contract when(Obs t, Contract c) ...
fun Contract scale(Double s, Contract c)...
fun Observation at(Date d) : Obs ...
```

One Contract

```
fun zero_coupon_discount_bond(t,x,k) =
  when (at t) (scale (konst x) (one k))
```

Financial Contracts as Functional Compostion

Combinator DSL

```
type Contract ... Observation ... Currency
```

```
fun Contract zero() ...
fun Contract one(Currency c) ...
fun Contract when(Obs t, Contract c) ...
fun Contract scale(Double s, Contract c)...
fun Observation at(Date d) : Obs ...
```

Simon Peyton Jones

One Contract

```
fun zero_coupon_discount_bond(t,x,k) =
  when (at t) (scale (konst x) (one k))
```

Simple functions represent basic ideas.

Simple functions represent basic ideas.

Combinator functions combine ideas.

Simple functions represent basic ideas.

Combinator functions combine ideas.

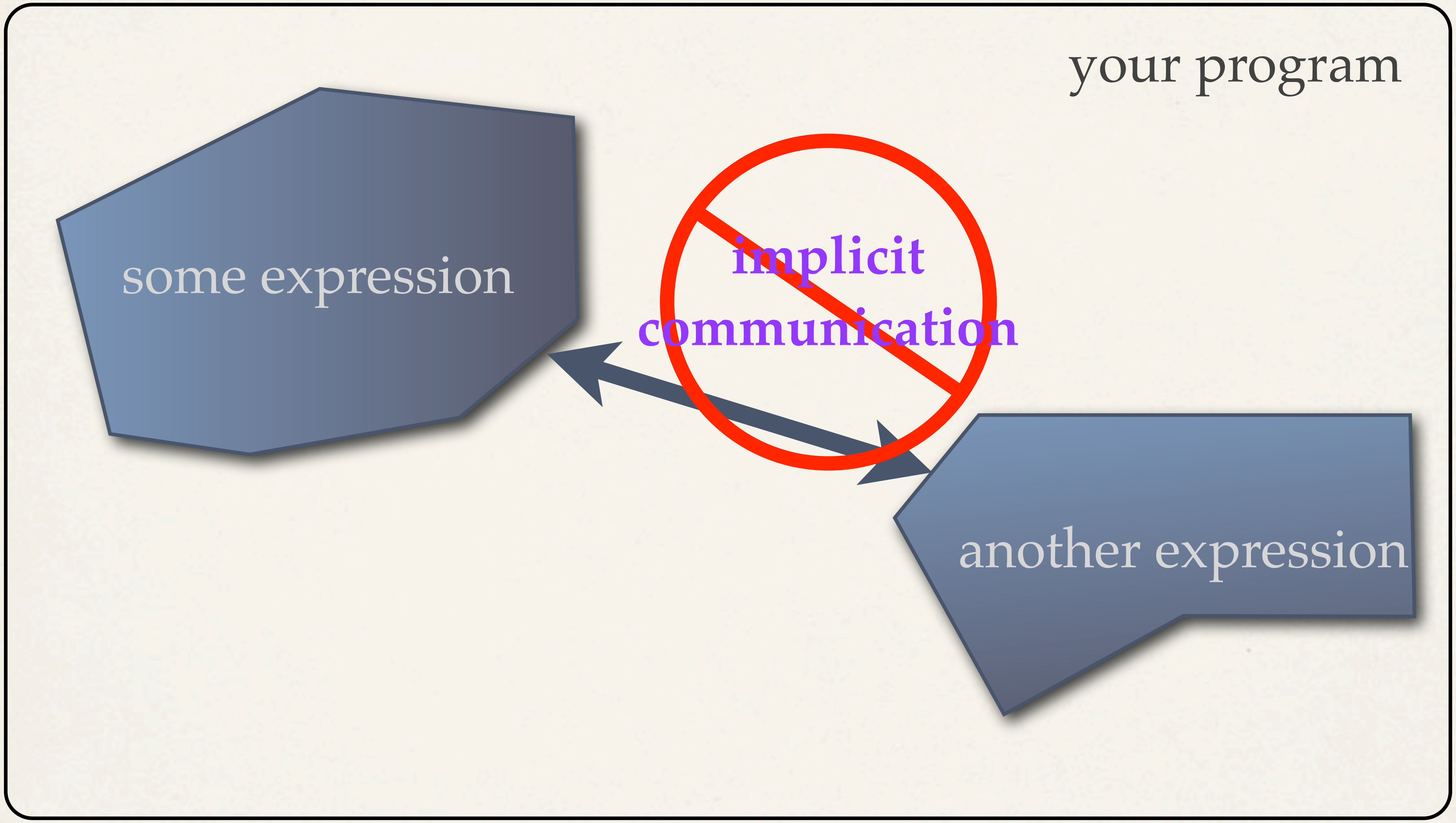
With function composition programmers create and communicate programs in combinator DSLs.

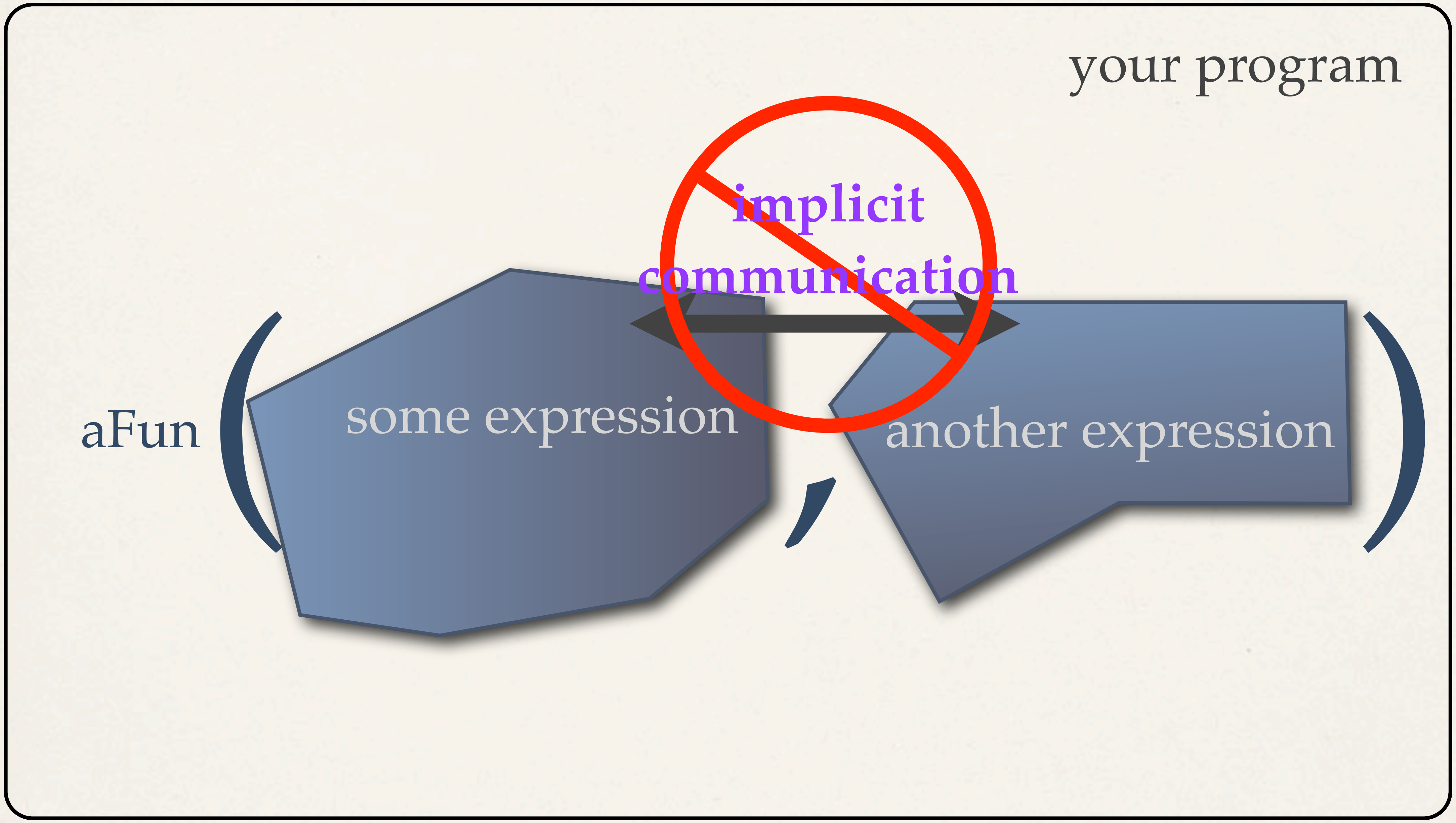
Functional programming languages in the LISP tradition use a "template" approach to DSLs in addition to combinators (Scheme, Clojure, Racket, Template Haskell).

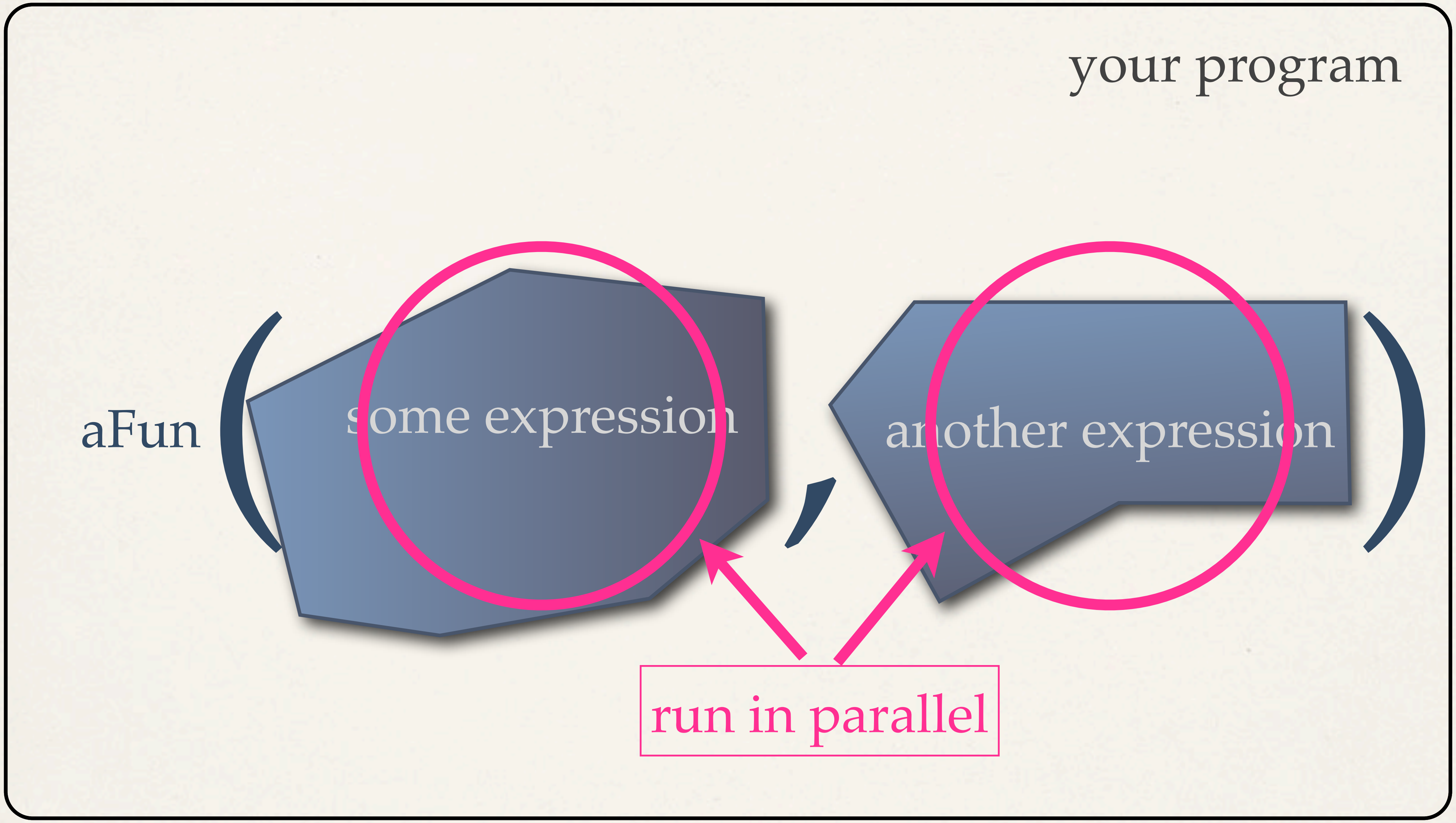
The last part of the functional story: parallelism.

Compilers think, too.

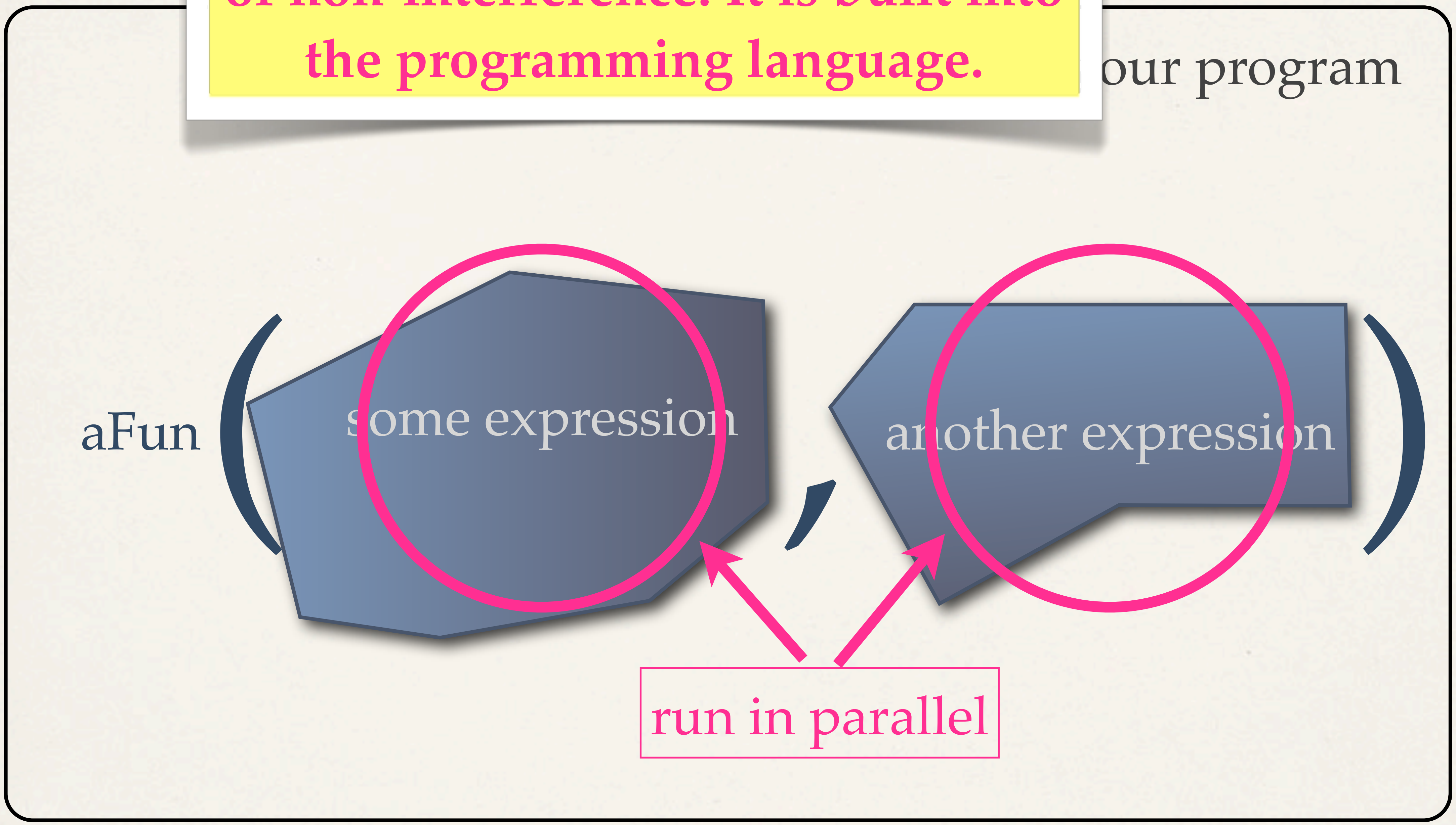
Remember the Definition




your program
implicit
communication
aFun
some expression
another expression


your program
aFun
some expression
another expression
run in parallel

In the purely functional world, the compiler does not need proof of non-interference. It is built into the programming language.



Implicit parallelism is free
in functional programming languages.

Sadly, this story is naive and unrealistic,
and yet it contains the key to a parallel future.

Sadly, this story is naive and unrealistic, and yet it contains the key to a parallel future.

In the imperative world mutation creates **too few opportunities** for automatic **parallel** execution.

In the functional world a **lack of dependencies** means **too many opportunities** for automatic **parallel** execution.

Sadly, this story is naive and unrealistic,
and yet it contains the key to a parallel future.

In the imperative world mutation creates **too few opportunities** for automatic **parallel** execution.

In the functional world a **lack of dependencies** means **too many opportunities** for automatic **parallel** execution.

The imperative world will see **explicit parallel programming** and the big battle against **race condition** bugs.

Sadly, this story is naive and unrealistic, and yet it contains the key to a parallel future.

In the imperative world mutation creates **too few opportunities** for automatic **parallel** execution.

In the functional world a **lack of dependencies** means **too many opportunities** for automatic **parallel** execution.

The imperative world will see **explicit parallel programming** and the big battle against **race condition** bugs.

The functional world will provide **explicit parallel programming** with fewer race conditions.

25 years of research on parallelism for FORTRAN calls for **mostly functional intermediate compiler representations** (PDGs, SSAs).

Explicit parallelism is easy in functional programming languages.

25 years of research on parallelism for FORTRAN calls for **mostly functional intermediate compiler representations** (PDGs, SSAs).

Functional programming languages make the dependencies explicit and thus facilitate the compiler's reasoning task.

Explicit parallelism is easy in functional programming languages.

So what is my favorite functional language?

What is my favorite
functional programming language?

The Racket language

- pattern matching et al.
- classes
- cross-platform GUIs
- extensive libraries
- rich web programming

The Racket language

- pattern matching et al.
- classes
- cross-platform GUIs
- extensive libraries
- rich web programming

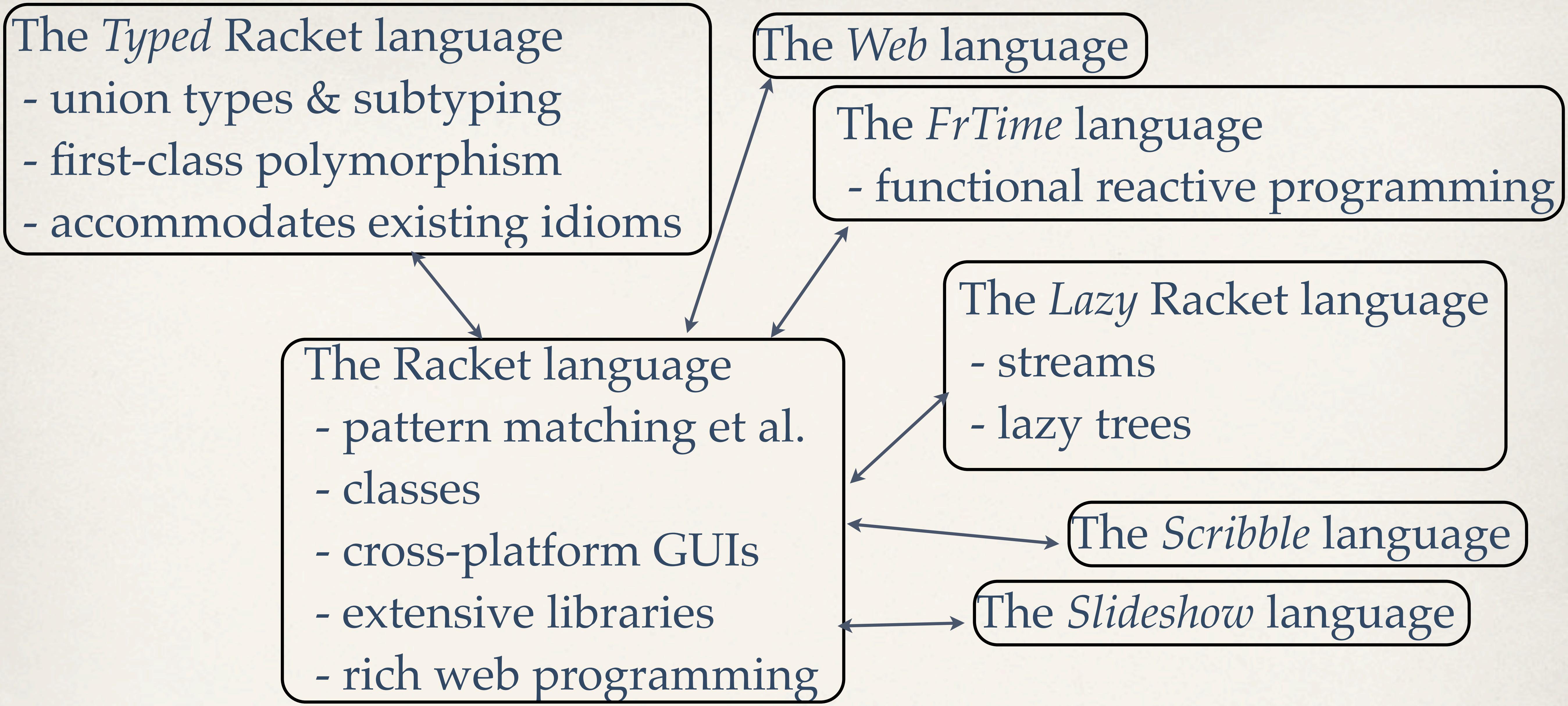
The *Lazy* Racket language

- streams
- lazy trees

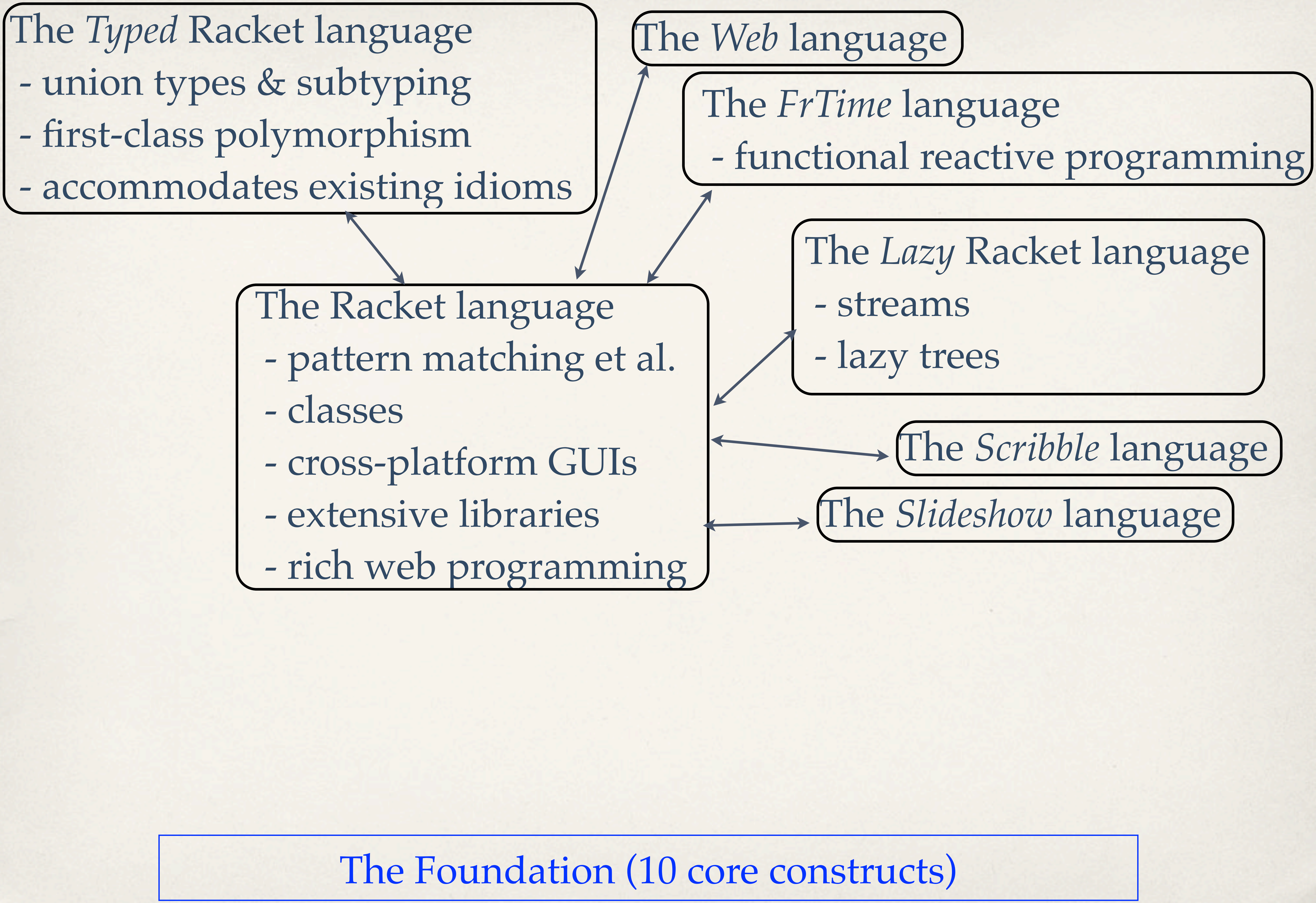
The *Typed* Racket language
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms

The Racket language
- pattern matching et al.
- classes
- cross-platform GUIs
- extensive libraries
- rich web programming

The *Lazy* Racket language
- streams
- lazy trees



The *Typed* Racket language
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms
The *Web* language
The *FrTime* language
- functional reactive programming
The *Lazy* Racket language
- streams
- lazy trees
The Racket language
- pattern matching et al.
- classes
- cross-platform GUIs
- extensive libraries
- rich web programming
The *Scribble* language
The *Slideshow* language


The *Typed* Racket language
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms
The *Web* language
The *FrTime* language
- functional reactive programming
The *Lazy* Racket language
- streams
- lazy trees
The Racket language
- pattern matching et al.
- classes
- cross-platform GUIs
- extensive libraries
- rich web programming
The *Scribble* language
The *Slideshow* language
The Foundation (10 core constructs)

The *Typed* Racket language
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms

The *Web* language

The *FrTime* language
- functional reactive programming

The *Lazy* Racket language
- streams
- lazy trees

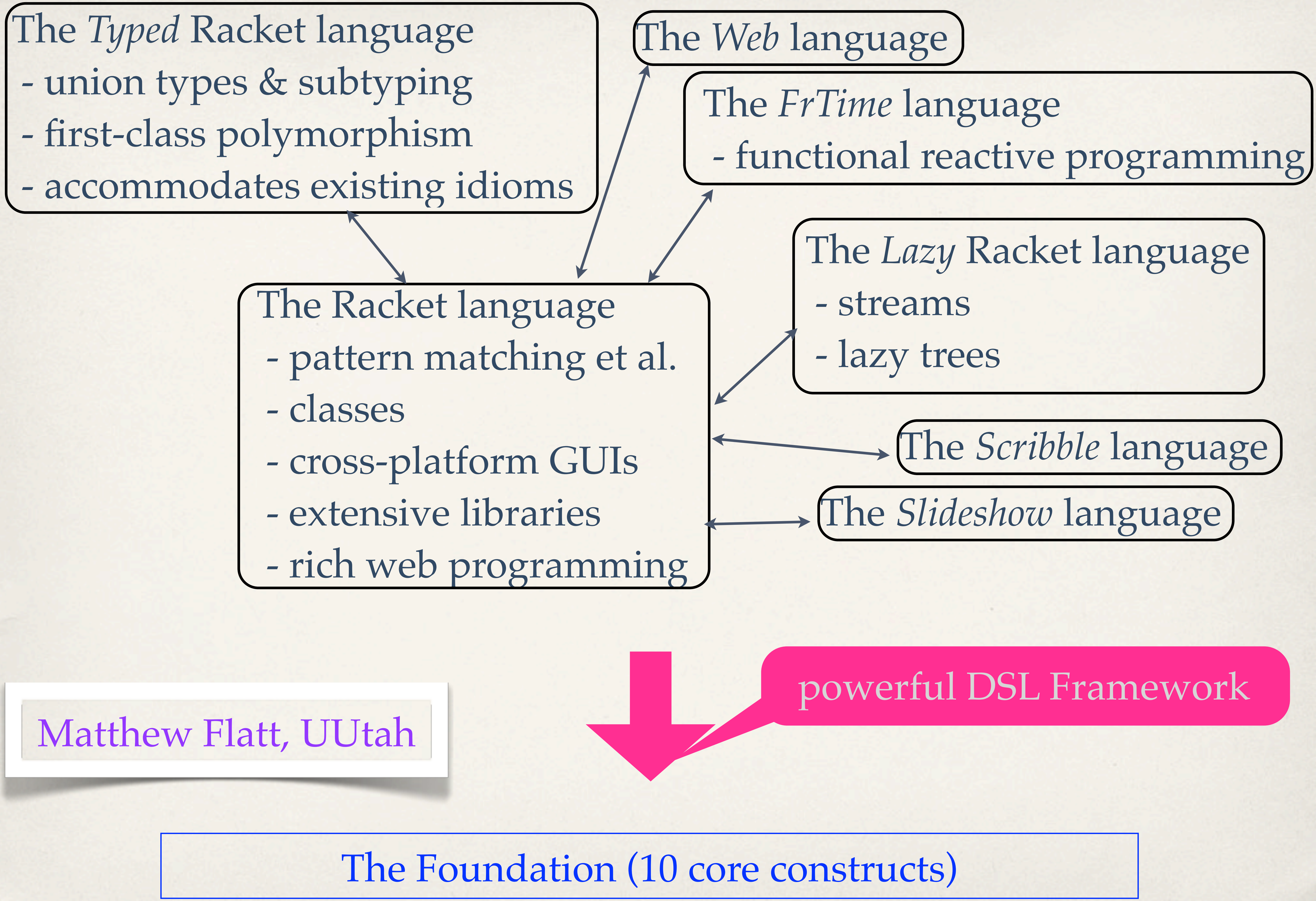
The Racket language
- pattern matching et al.
- classes
- cross-platform GUIs
- extensive libraries
- rich web programming

The *Scribble* language

The *Slideshow* language

powerful DSL Framework

The Foundation (10 core constructs)


The *Typed* Racket language
- union types & subtyping
- first-class polymorphism
- accommodates existing idioms
The *Web* language
The *FrTime* language
- functional reactive programming
The *Lazy* Racket language
- streams
- lazy trees
The Racket language
- pattern matching et al.
- classes
- cross-platform GUIs
- extensive libraries
- rich web programming
The *Scribble* language
The *Slideshow* language
powerful DSL Framework
Matthew Flatt, UUtah
The Foundation (10 core constructs)

Summary

Functional programming is about clear, concise **communication** between **programmers**.

Functional programming **languages** keep you **honest** about being **functional**.

A good transition needs training, but training pays off.

Thank You

Thank You

Though Smalltalk came from many motivations, ... one was **to find a more flexible version of assignment, and then to try to eliminate it altogether.**
Alan Kay,
History of Smalltalk (1993)

Favor immutability.
Joshua Bloch,
Effective Java (2001)

Use **value objects** when possible.
Kent Beck,
Test Driven Development (2001)