Resugaring: Lifting Evaluation Sequences through Syntactic Sugar

Justin Pombrio,
Shriram Krishnamurthi
Brown University

Syntactic Sugar

Syntactic Sugar

```
x + 2
```


```
x.__add__(2)
```

Syntactic Sugar

```
x + 2
```


```
x.__add__(2)
```

```
[x*x | x <- lst]
```


```
map (\x -> x*x) lst
```

Syntactic Sugar

```
x + 2
```


```
x.__add__(2)
```

```
[x*x | x <- lst]
```


```
map (\x -> x*x) lst
```

```
x or y
```


```
let t = x in
  if t then t else y
```

Surface language
(what you write)

Syntactic Sugar

```
x + 2
```


```
x.__add__(2)
```

```
[x*x | x <- lst]
```


```
map (\x -> x*x) lst
```

```
x or y
```


```
let t = x in
  if t then t else y
```

Syntactic Sugar

Surface language (what you write)

Core language (what runs)

```
x + 2
```

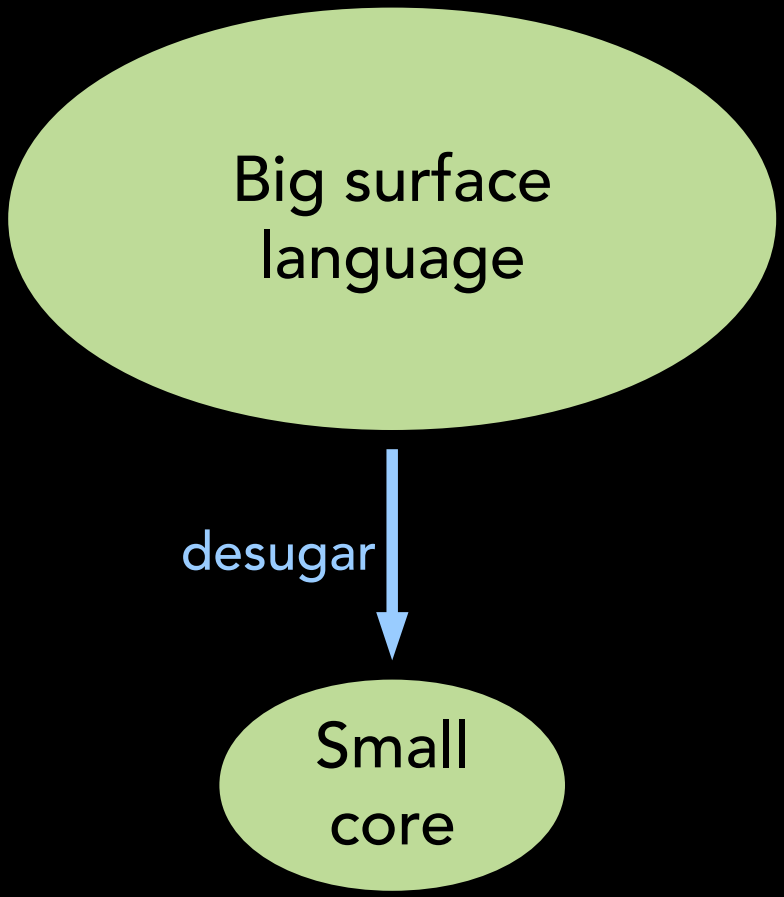

```
x.__add__(2)
```

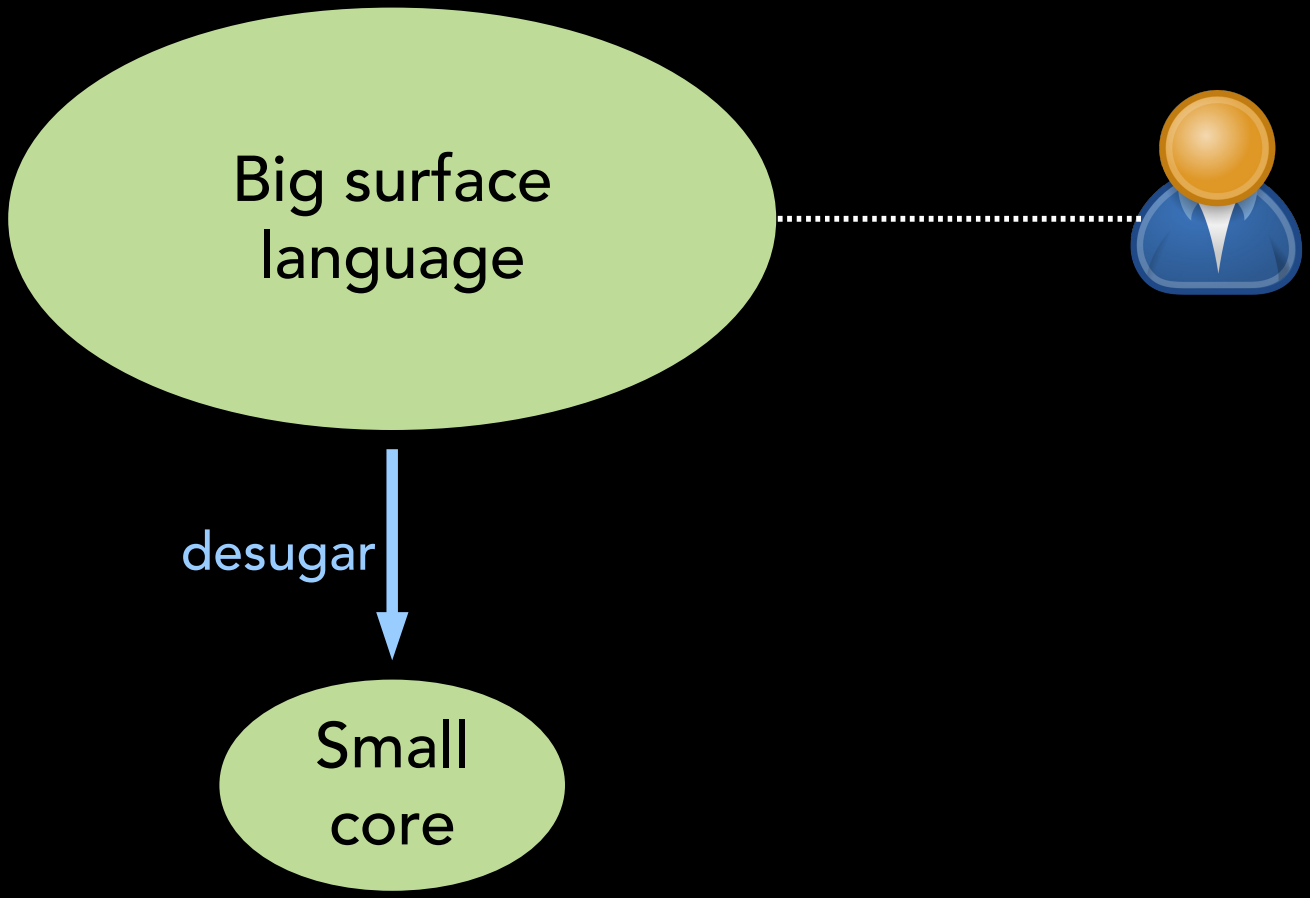
```
[x*x | x <- lst]
```

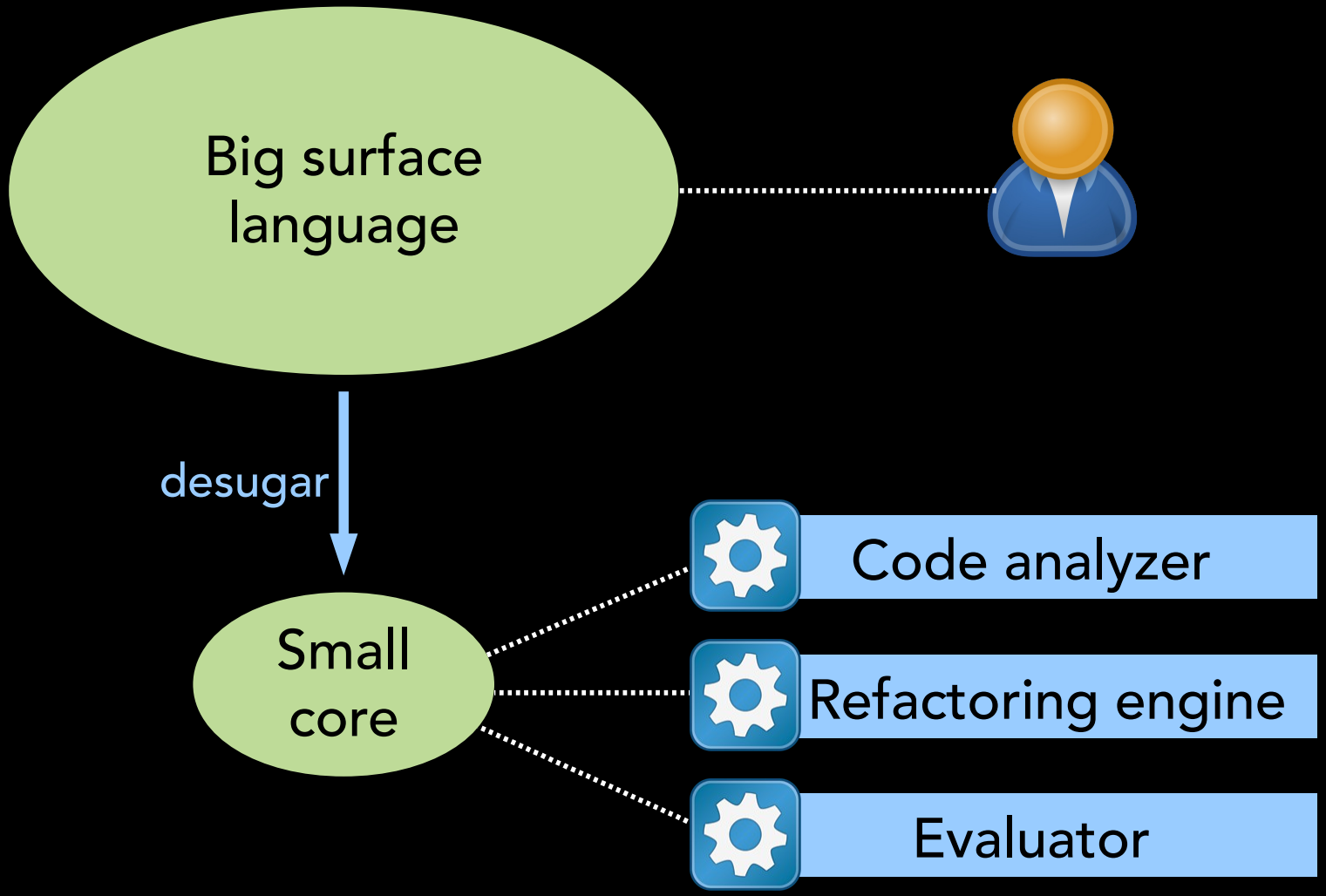

```
map (\x -> x*x) lst
```

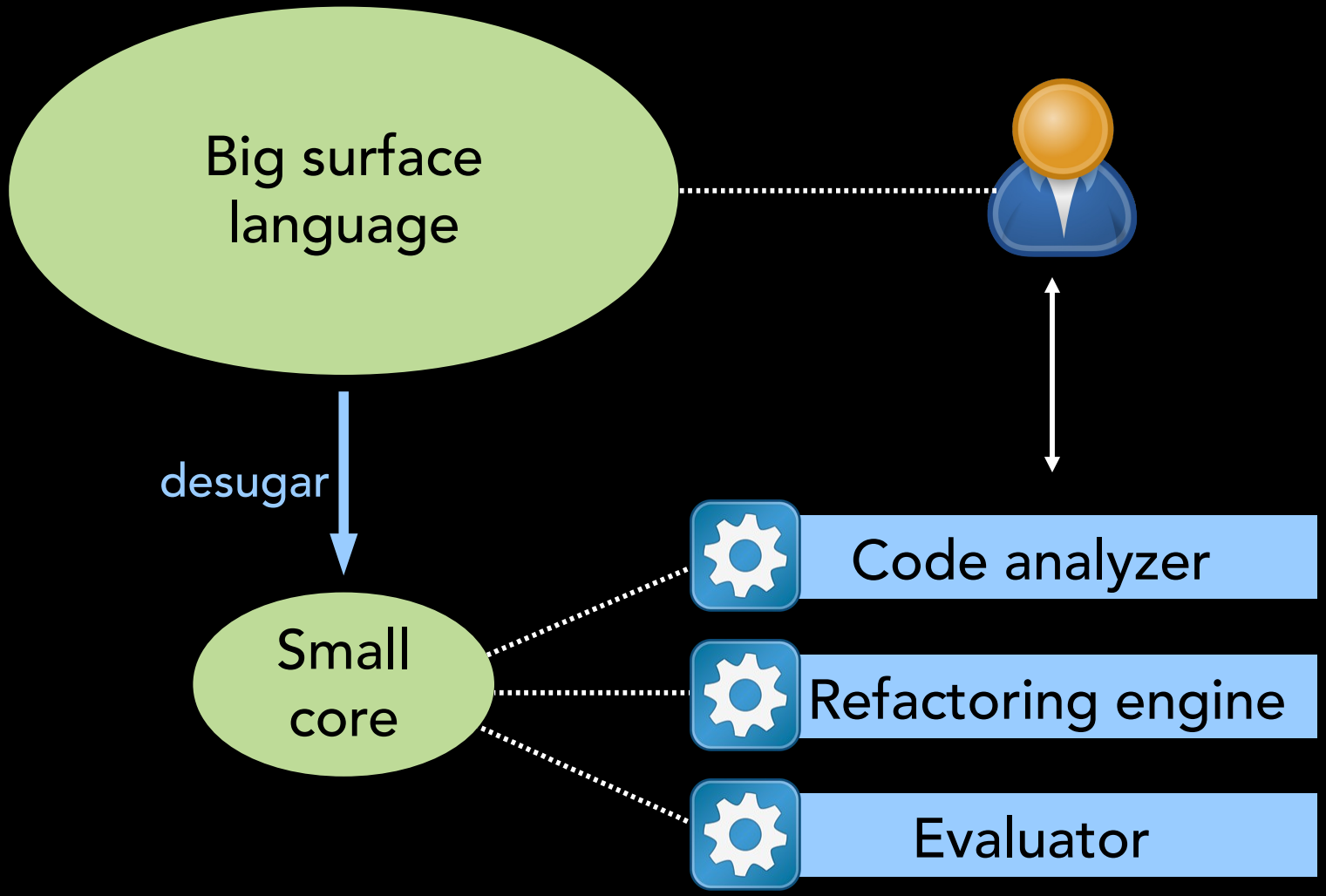
```
x or y
```

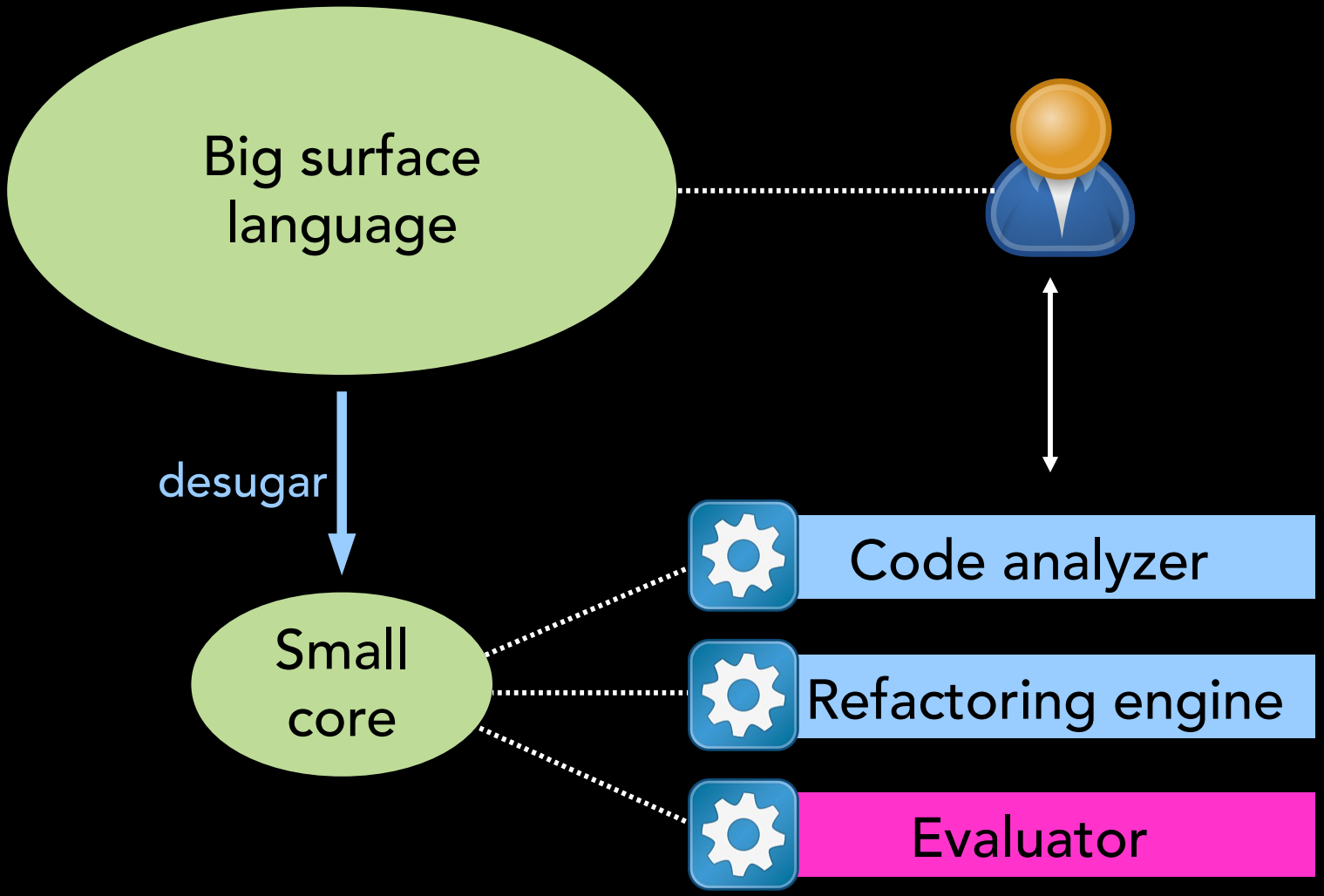

```
let t = x in
  if t then t else y
```


Big surface
language
desugar
Small
core


Big surface
language
desugar
Small
core


Big surface language
desugar
Small core
Code analyzer
Refactoring engine
Evaluator


Big surface language
desugar
Small core
Code analyzer
Refactoring engine
Evaluator


Big surface language
desugar
Small core
Code analyzer
Refactoring engine
Evaluator

Surface

```
not(true) or true
```

Core

Surface

```
not(true) or true
```

Core

```
let t = not(true) in
  if t then t else true
```

Surface

```
not(true) or true
```

desugar

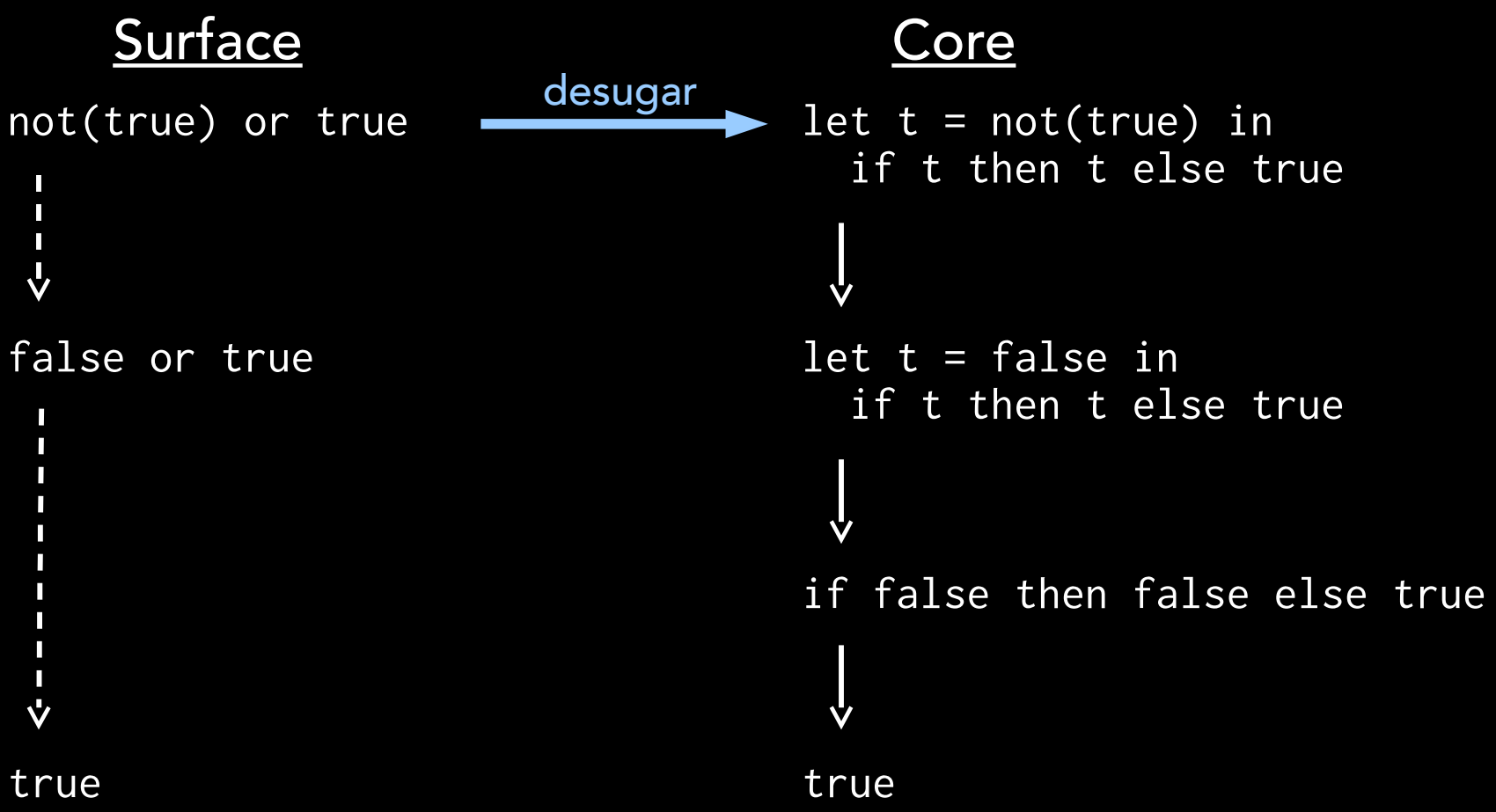
Core

```
let t = not(true) in
  if t then t else true
```

```
let t = false in
  if t then t else true
```

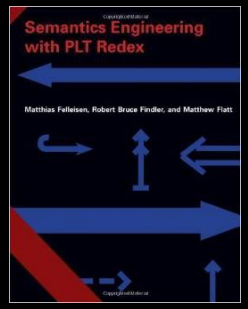
```
if false then false else true
```

```
true
```

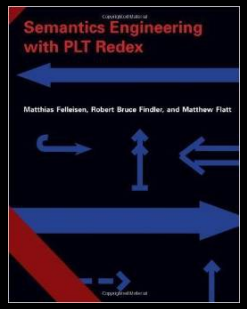

Surface
Core
desugar
not(true) or true
false or true
true
let t = not(true) in
if t then t else true
let t = false in
if t then t else true
if false then false else true
true

Confection

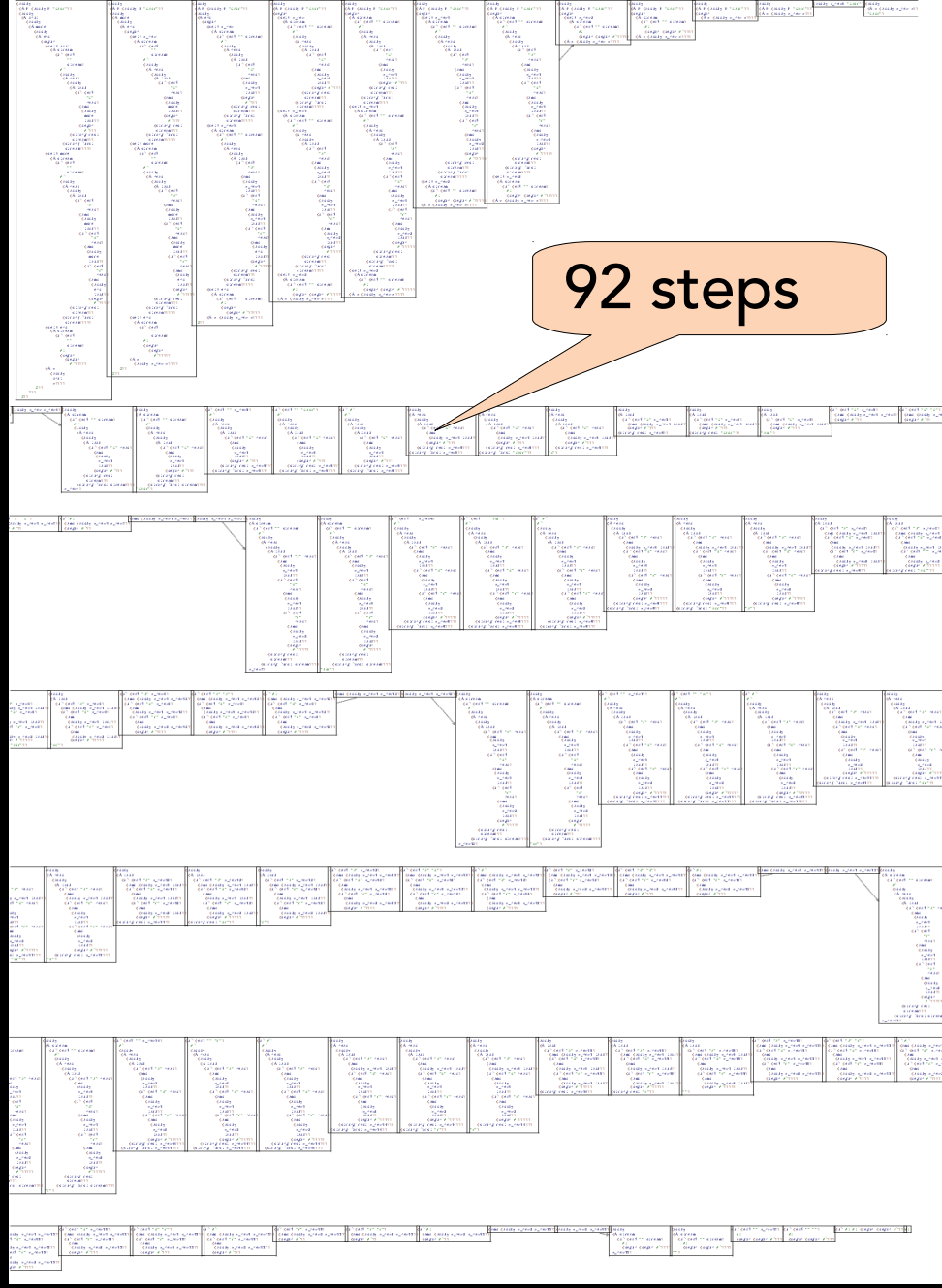
core eval seq → surface eval seq

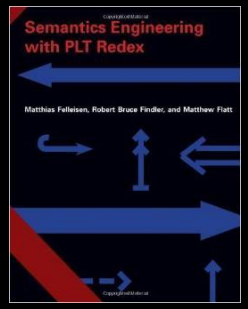



```
(Let
 M
 (Automaton
  init
  (init : ("c" -> more))
  (more
   :
   ("a" -> more)
   ("d" -> more)
   ("r" -> end))
  (end : "accept"))
 (apply M "cadr"))
```




```
(Let
 M
 (Automaton
  init
  (init : ("c" -> more))
  (more
   :
   ("a" -> more)
   ("d" -> more)
   ("r" -> end))
  (end : "accept"))
 (apply M "cadr"))
```






```
(Let
 M
 (Automaton
  init
  (init : ("c" -> more))
  (more
   :
   ("a" -> more)
   ("d" -> more)
   ("r" -> end))
  (end : "accept"))
 (apply M "cadr"))
```

7 steps

(apply M "cadr")

(apply init "cadr")

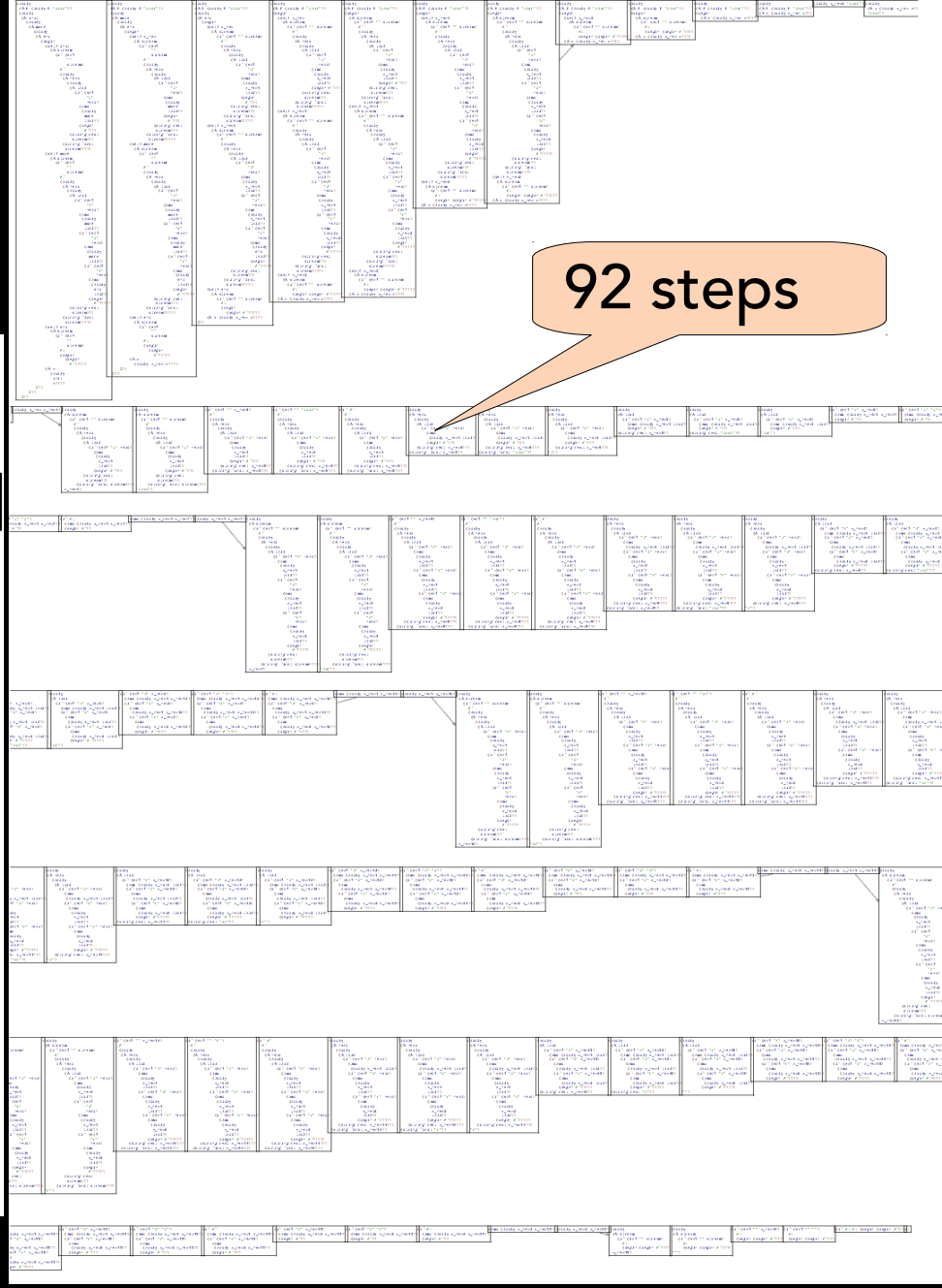
(apply more "adr")

(apply more "dr")

(apply more "r")

(apply end "")

#t



Surface

```
not(true) or true
```

Core

Surface	Core

```
not(true) or true
```

```
false or true
```

```
true
```

Surface

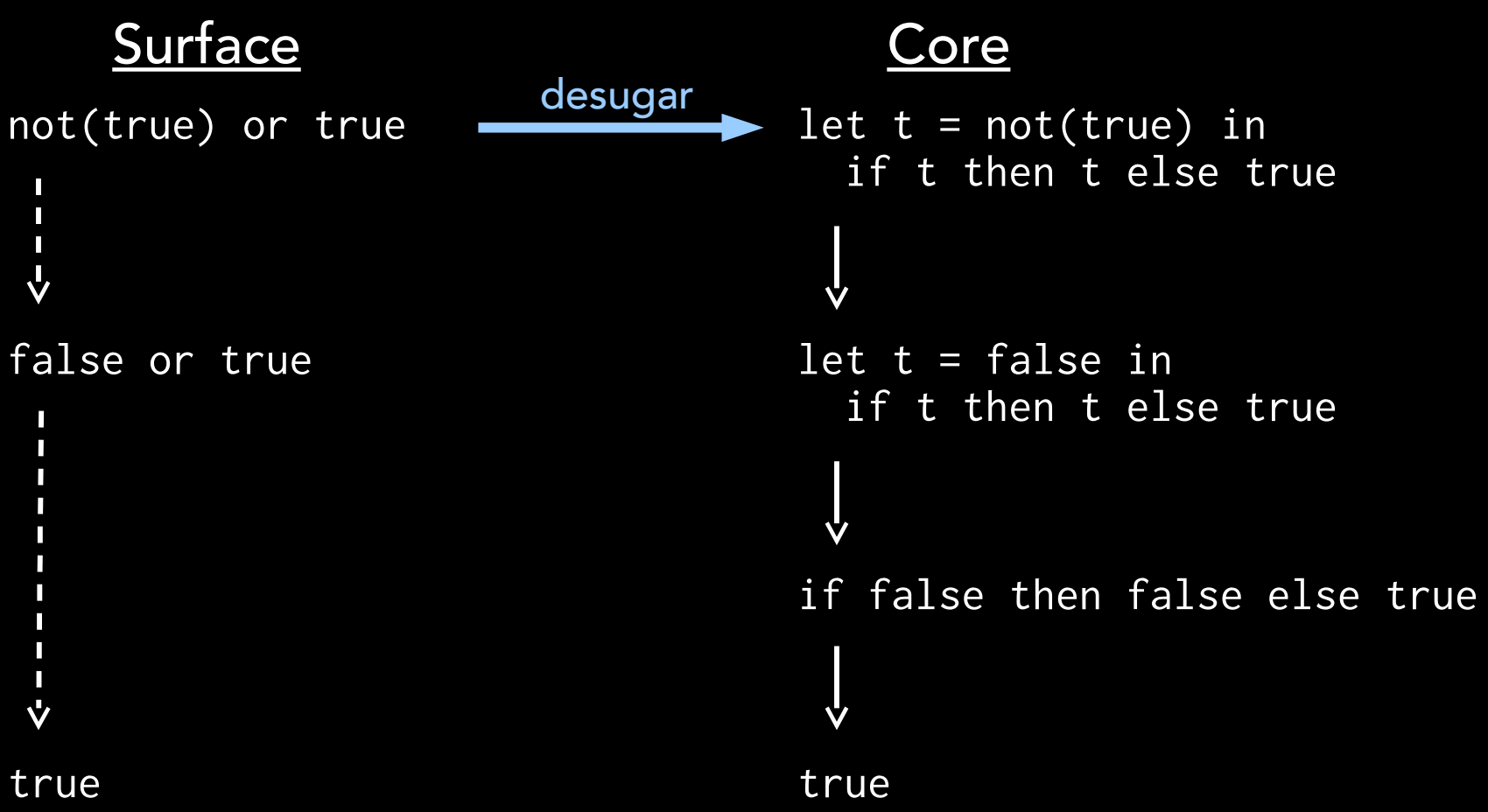
```
not(true) or true
```

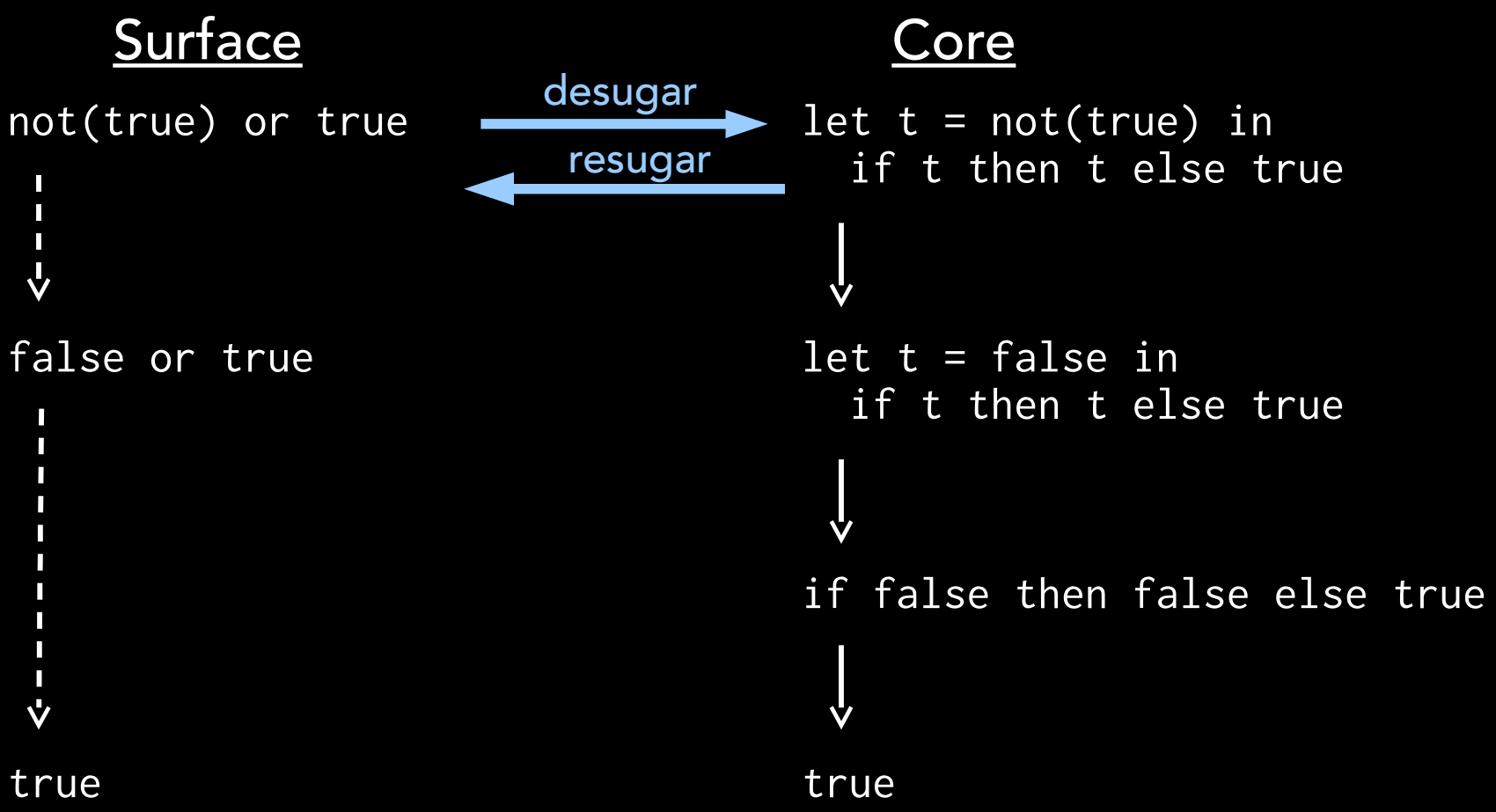
```
false or true
```

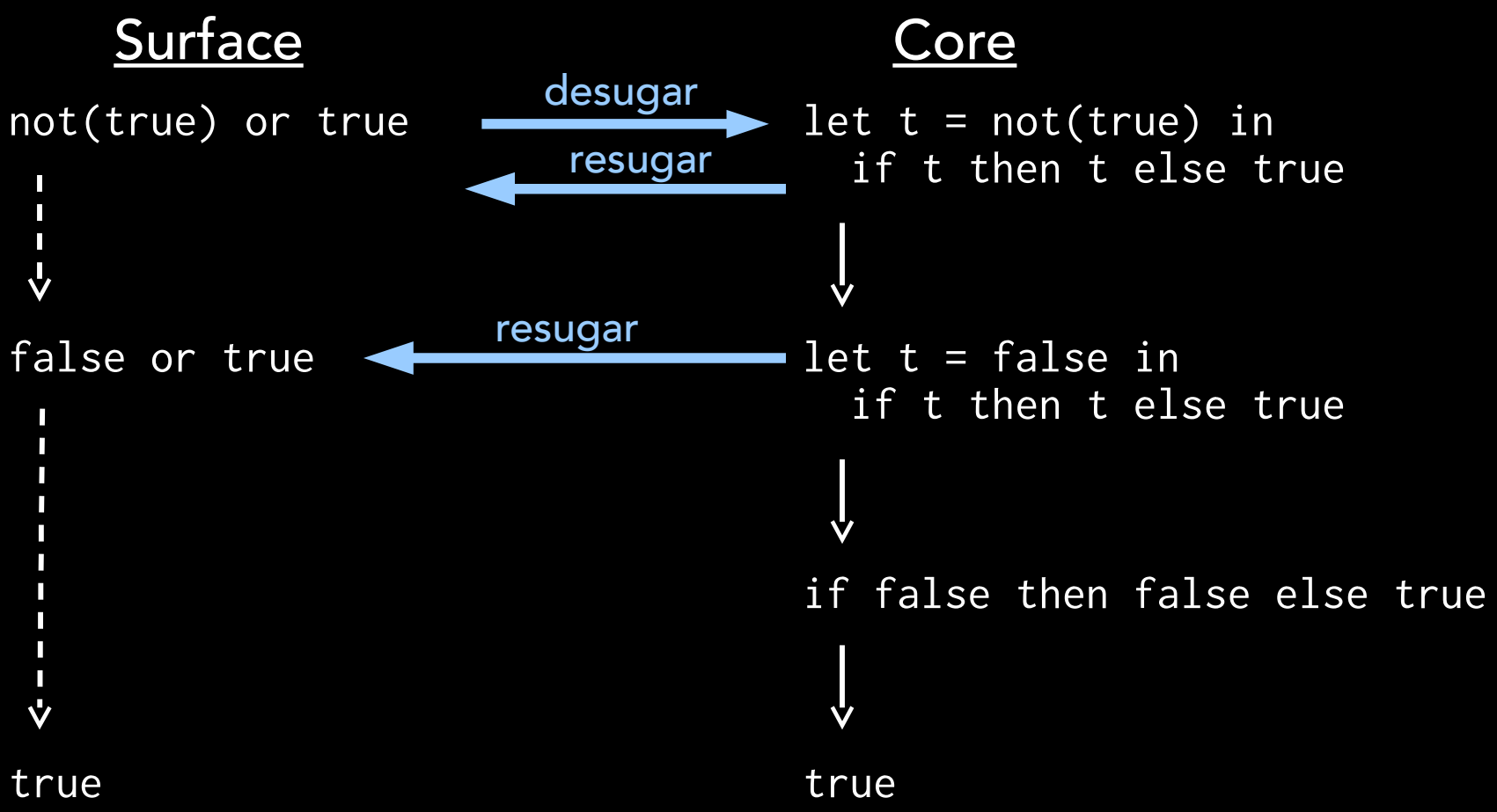
```
true
```

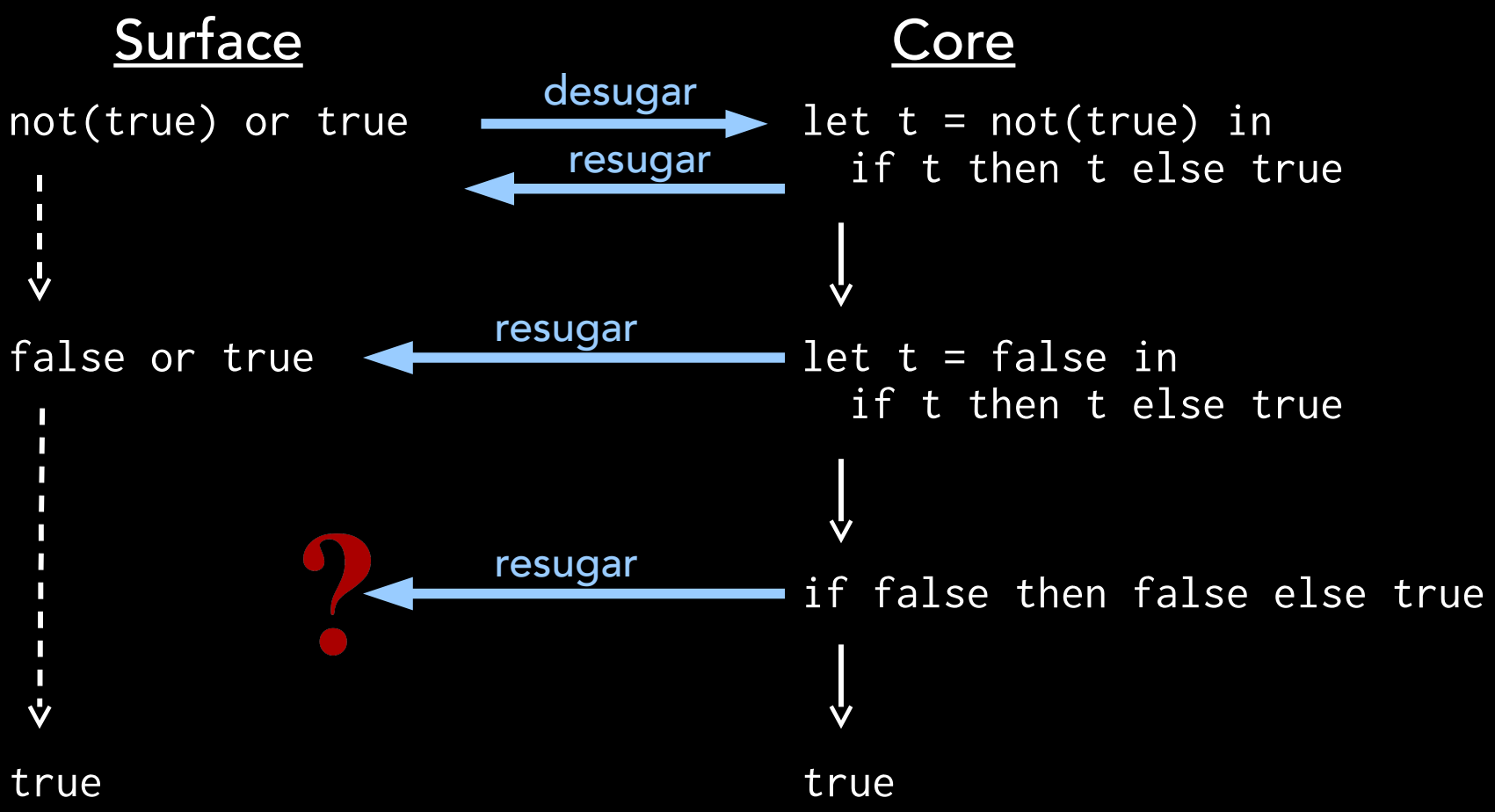
Core

```
let t = not(true) in
  if t then t else true
```

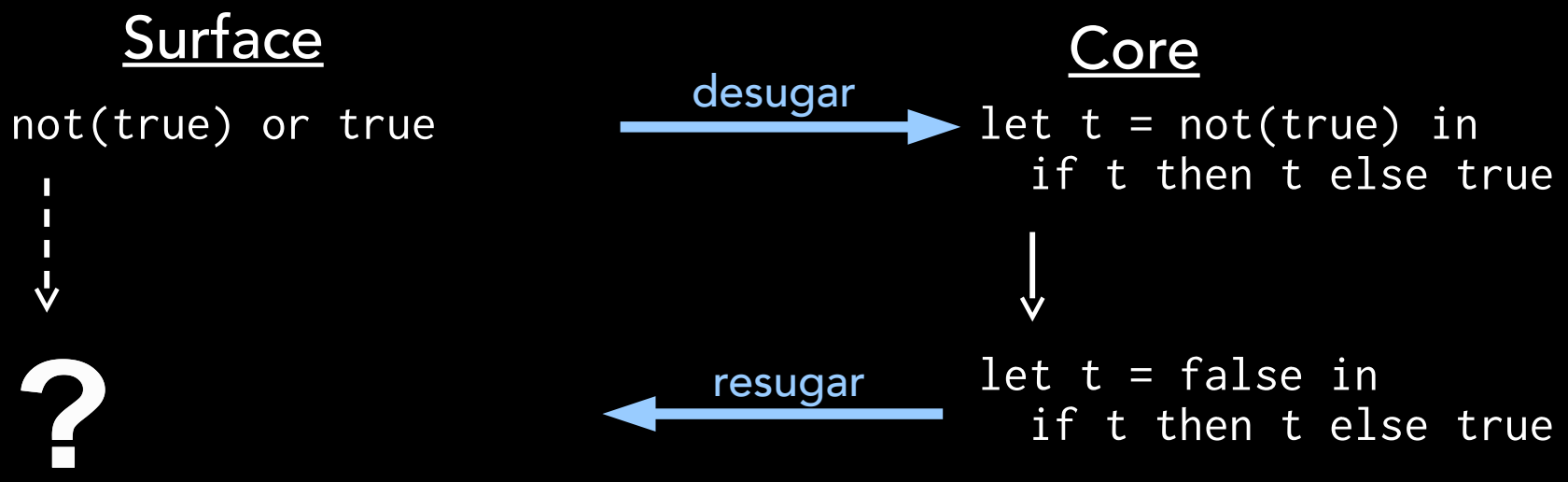

Surface
Core
desugar
not(true) or true
let t = not(true) in
if t then t else true
false or true
let t = false in
if t then t else true
if false then false else true
true
true

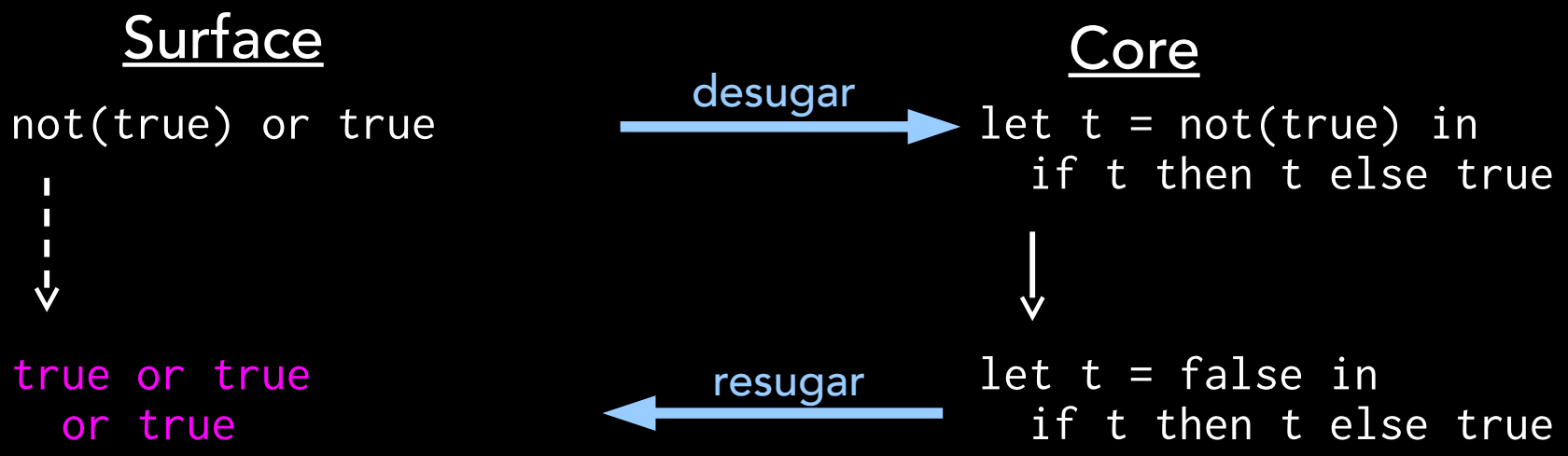

Surface
Core
desugar
resugar
not(true) or true
let t = not(true) in
if t then t else true
false or true
let t = false in
if t then t else true
if false then false else true
true
true

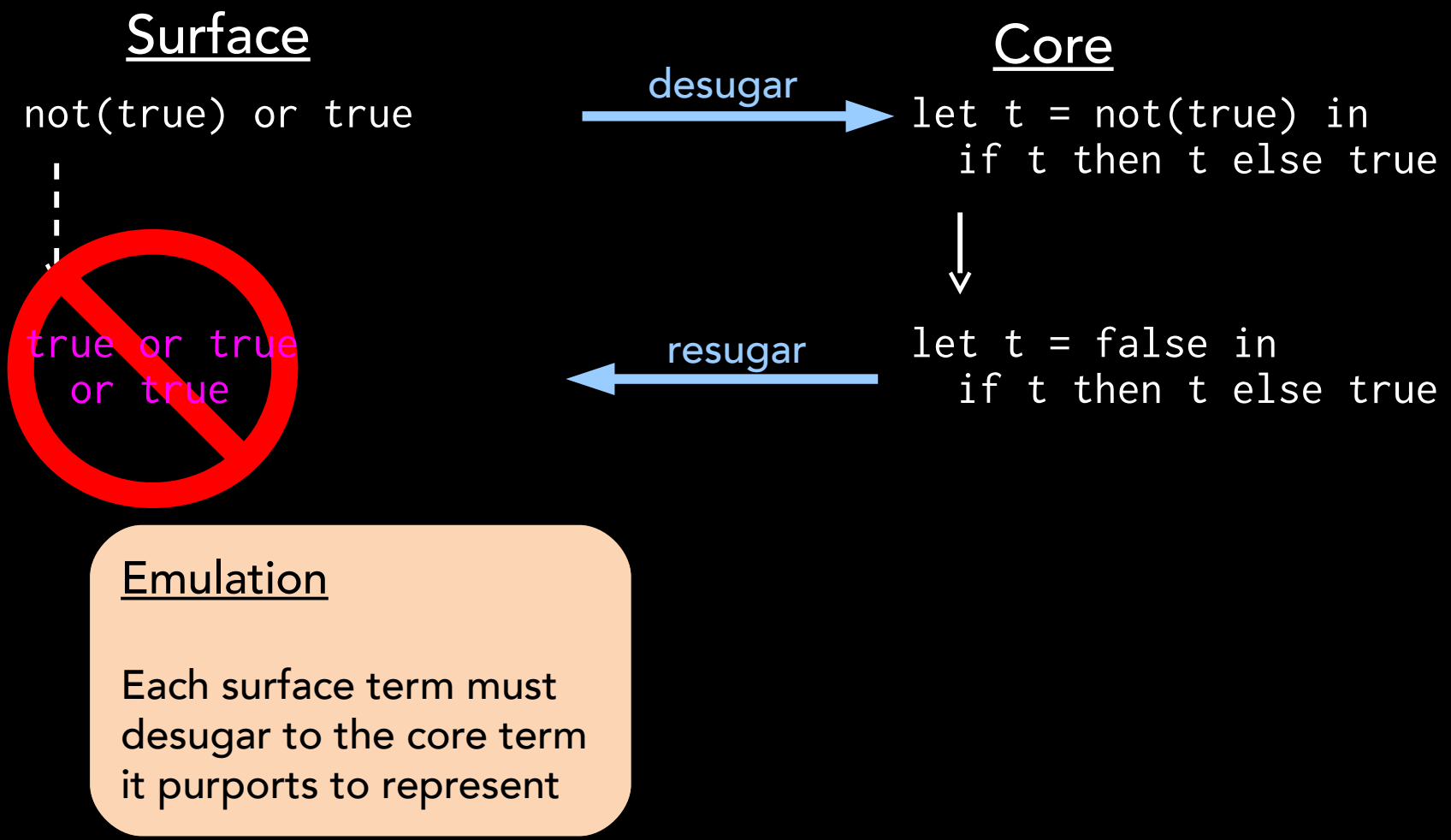

Surface
Core
desugar
resugar
not(true) or true
let t = not(true) in
if t then t else true
resugar
false or true
let t = false in
if t then t else true
if false then false else true
true
true


Surface
Core
desugar
resugar
not(true) or true
let t = not(true) in
if t then t else true
resugar
false or true
let t = false in
if t then t else true
resugar
?
if false then false else true
true
true

THREE KEY PROPERTIES OF RESUGARING

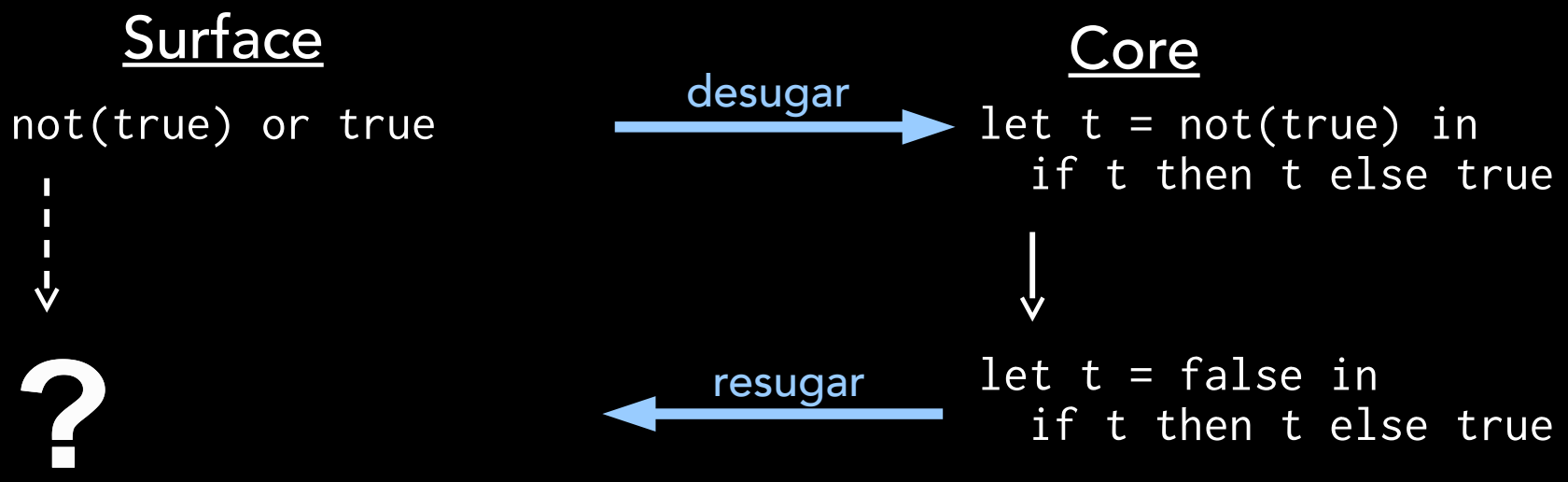

Surface
Core
desugar
not(true) or true
let t = not(true) in
if t then t else true
?
resugar
let t = false in
if t then t else true

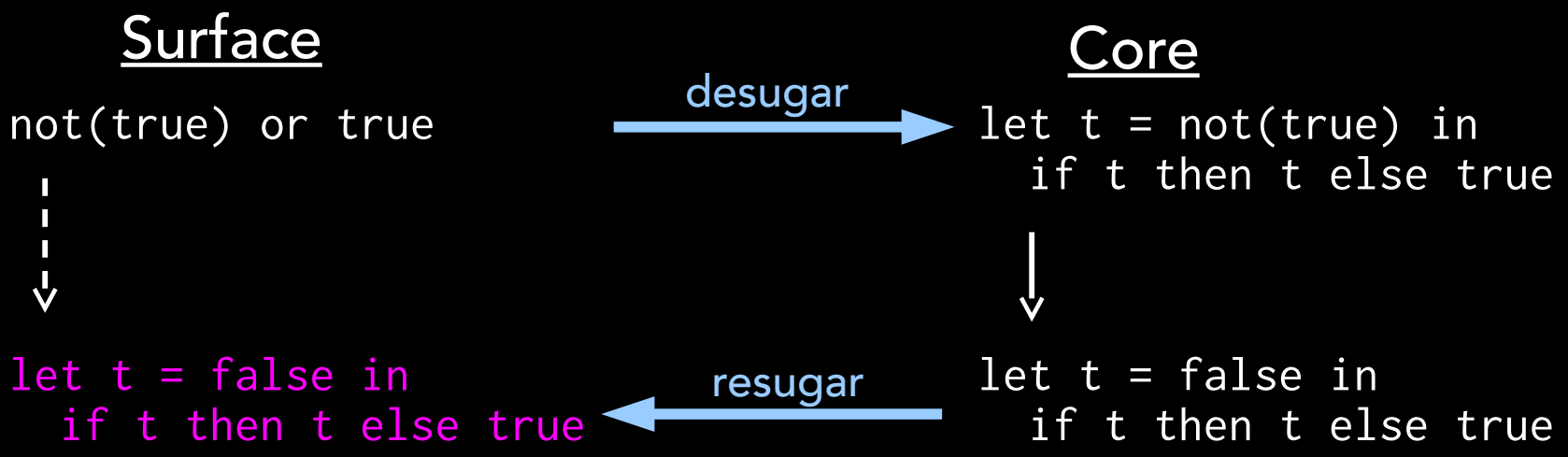

Surface
Core
desugar
not(true) or true
let t = not(true) in
if t then t else true
true or true
or true
resugar
let t = false in
if t then t else true

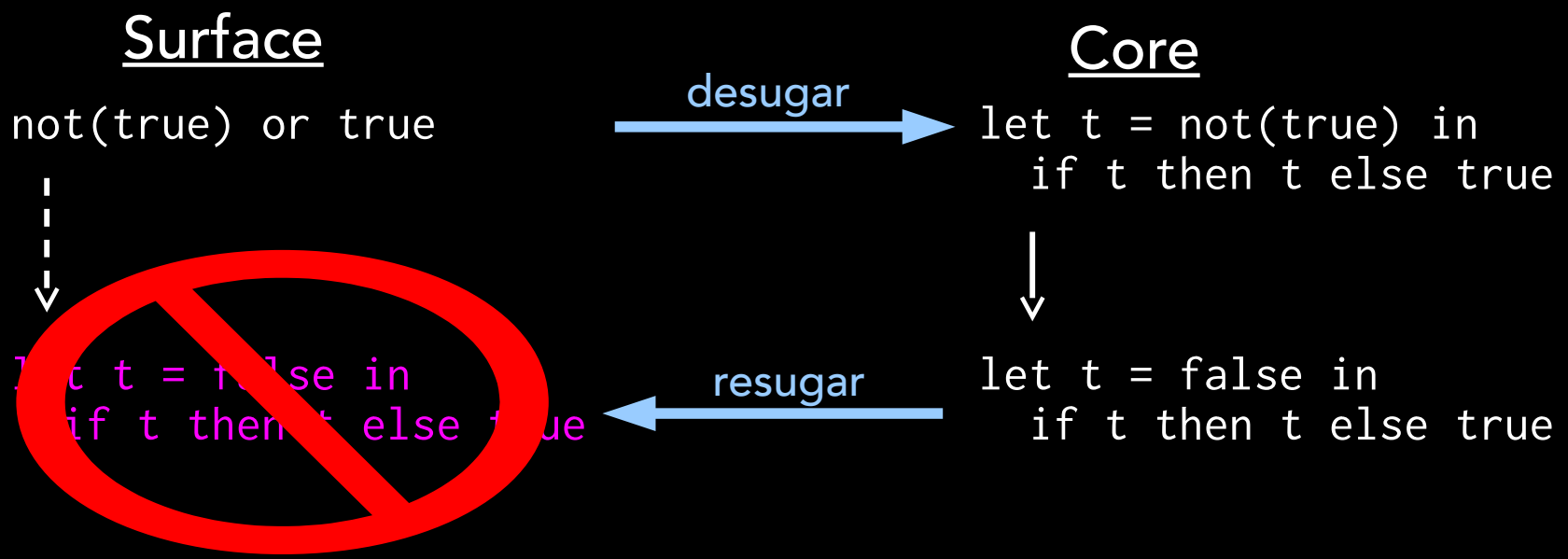


Emulation

Each surface term must desugar to the core term it purports to represent

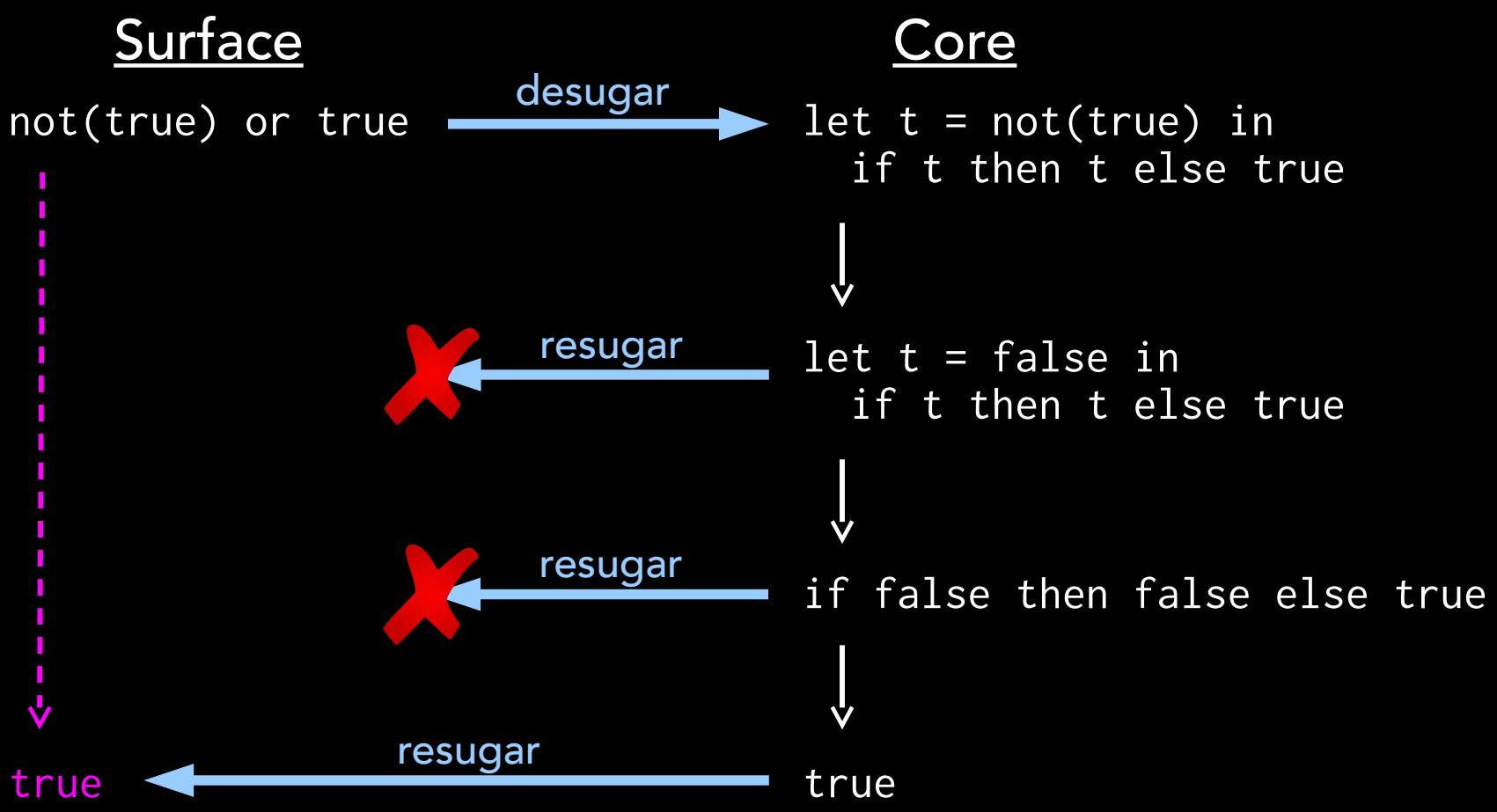

Surface
Core
desugar
not(true) or true
let t = not(true) in
if t then t else true
?
resugar
let t = false in
if t then t else true

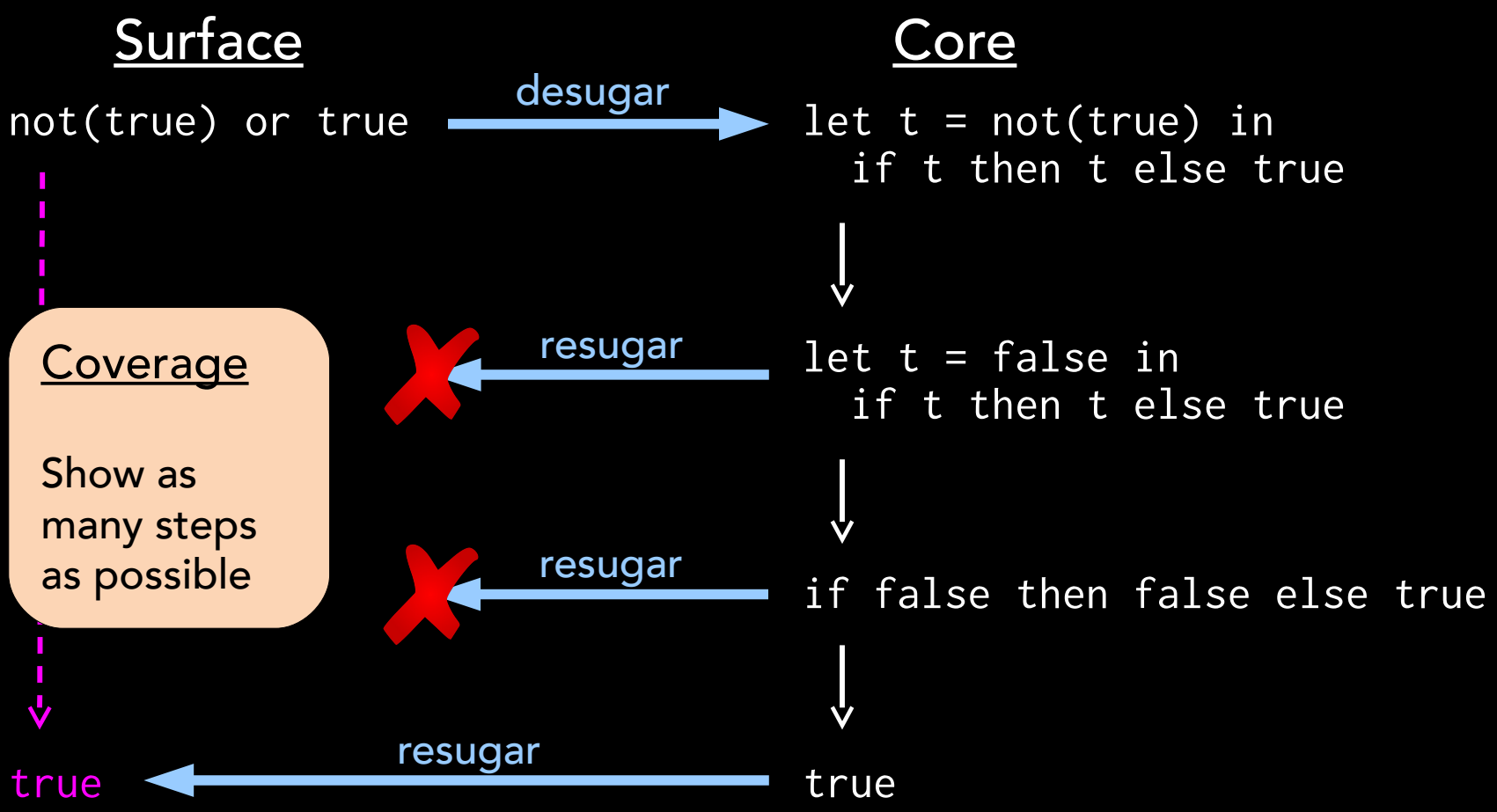

Surface
Core
desugar
not(true) or true
let t = not(true) in
if t then t else true
let t = false in
if t then t else true
resugar
let t = false in
if t then t else true



Abstraction

Show things in terms of a sugar precisely when the programmer used that sugar.


Surface
Core
not(true) or true
desugar
let t = not(true) in
if t then t else true
resugar
let t = false in
if t then t else true
resugar
if false then false else true
resugar
true
true

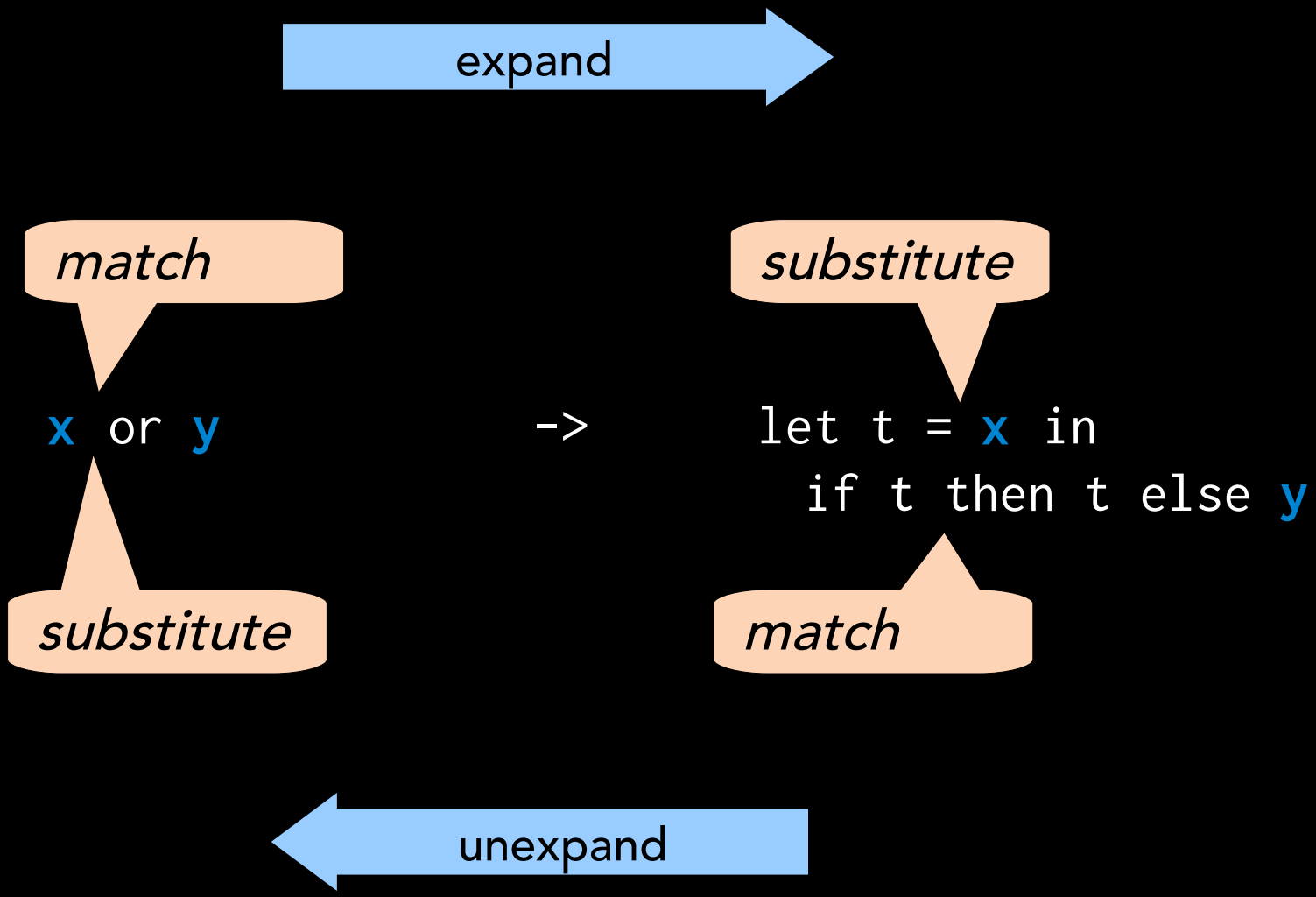

Surface
Core
not(true) or true
desugar
let t = not(true) in
if t then t else true
Coverage
Show as
many steps
as possible
resugar
let t = false in
if t then t else true
resugar
if false then false else true
resugar
true
true

```
x or y          ->      let t = x in
                          if t then t else y
```

match

substitute

```
x or y          ->      let t = x in
                          if t then t else y
```


expand
match
substitute
x or y
->
let t = x in
if t then t else y
substitute
match
unexpand

A Little Theory

$$\begin{aligned} \text{expand} &: \text{Surf Term} \to \text{Core Term} \\ \text{unexpand} &: \text{Core Term} \times \text{Surf Term} \to \text{Surf Term} \end{aligned}$$

A Little Theory

expand : Surf Term → Core Term

unexpand : Core Term × Surf Term → Surf Term

It's a lens!

A Little Theory

expand : Surf Term → Core Term

unexpand : Core Term × Surf Term → Surf Term

It's a lens!

unexpand (expand T) T = T — *GetPut*

expand (unexpand T′ T) = T′ — *PutGet*

A Little Theory

expand : Surf Term → Core Term

unexpand : Core Term × Surf Term → Surf Term

It's a lens!

unexpand (expand T) T = T — *GetPut*

expand (unexpand T′ T) = T′ — *PutGet*

Well-formedness criteria on rules ensure these laws.

Key Properties

- [ ] Emulation
- [ ] Abstraction
- [ ] Coverage

Key Properties

- [x] Emulation (Lens Laws)
- [ ] Abstraction
- [ ] Coverage

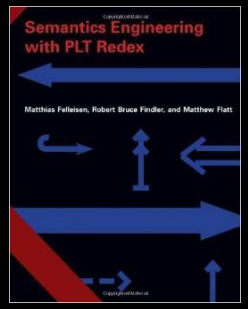
Key Properties

- [x] Emulation (Lens Laws)
- [x] Abstraction (Tagging – see paper)
- [ ] Coverage

Key Properties

- [x] Emulation (Lens Laws)
- [x] Abstraction (Tagging – see paper)
- [x] Coverage (Empirical)

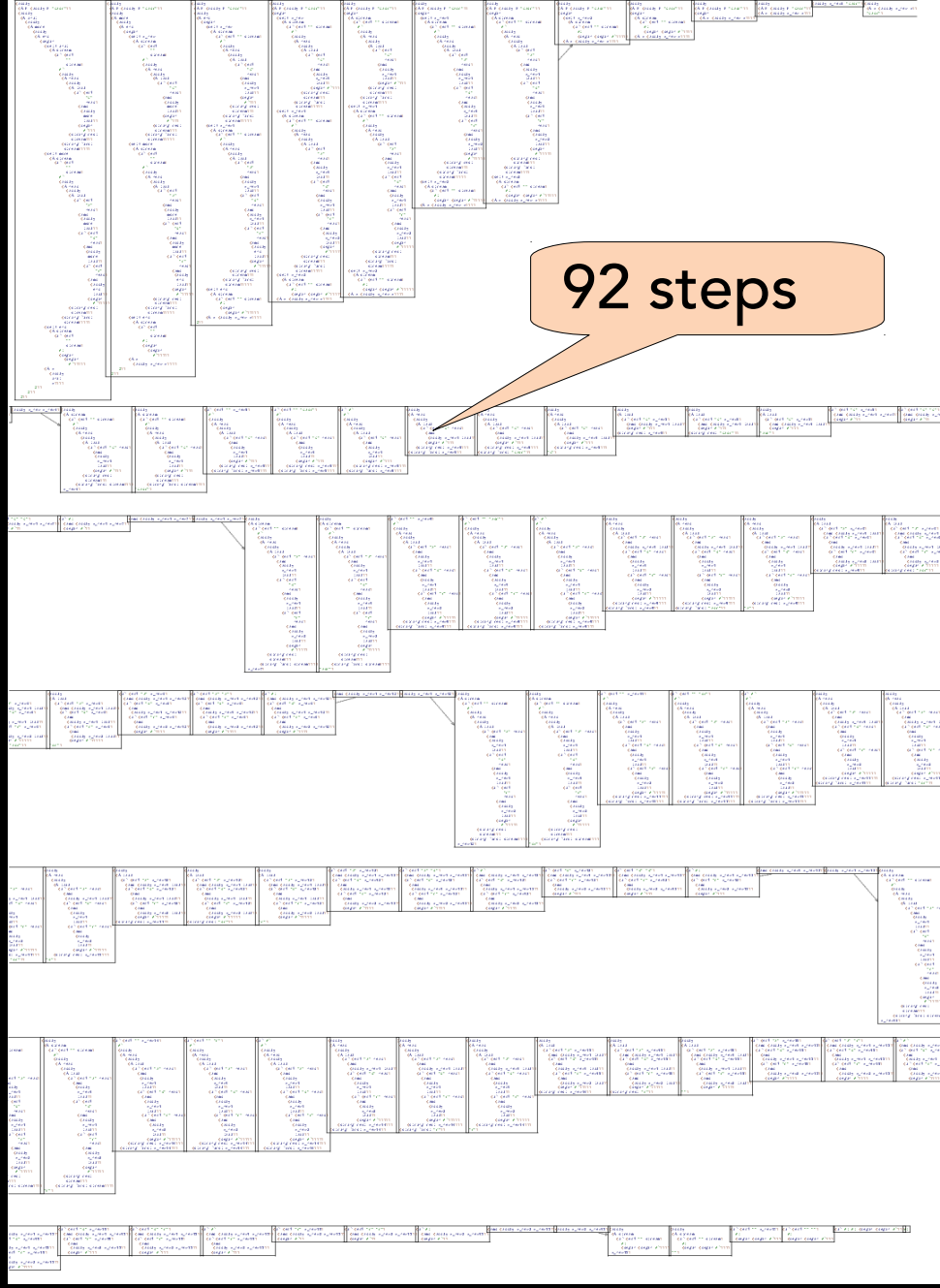
A WORKING SYSTEM




```
(Let
 M
 (Automaton
  init
  (init : ("c" -> more))
  (more
   :
   ("a" -> more)
   ("d" -> more)
   ("r" -> end))
  (end : "accept"))
 (apply M "cadr"))
```

```
(apply M "cadr")
(apply init "cadr")
(apply more "adr")
(apply more "dr")
(apply more "r")
(apply end "")
#t
```

7 steps



Surface

```
(or (and #t #f) #f)
```

Core

Surface

```
(or (and #t #f) #f)
```

desugar

Core

```
((lambda (t) (begin
   (if t t ((lambda () #f)))))
 (if #t ((lambda () #f)) #f))
```

Surface

```
(or (and #t #f) #f)
```

desugar

Core

```
((lambda (t) (begin
   (if t t ((lambda () #f)))))
 (if #t ((lambda () #f)) #f))
```

⇓

```
((lambda (t) (begin
   (if t t ((lambda () #f)))))
 ((lambda () #f)))
```

⇓

```
((lambda (t) (begin
   (if t t ((lambda () #f))))) #f)
```

⇓

```
(if #f t ((lambda () #f)))
```

⇓

```
((lambda () #f))
```

⇓

```
#f
```

Surface

```
(or (and #t #f) #f)
 ⇣
(or #f #f)
 ⇣
(or #f)
 ⇣
#f
```

desugar →

Core

```
((lambda (t) (begin
   (if t t ((lambda () #f)))))
 (if #t ((lambda () #f)) #f))
 ⇣
((lambda (t) (begin
   (if t t ((lambda () #f)))))
 ((lambda () #f)))
 ⇣
((lambda (t) (begin
   (if t t ((lambda () #f))))) #f)
 ⇣
(if #f t ((lambda () #f)))
 ⇣
((lambda () #f))
 ⇣
#f
```

Surface

Core

```
my-list = [2]
cases(List) my-list:
  | empty() => print("empty")
  | link(something, _) =>
    print("not empty")
end
```

Surface

```
my-list = [2]
cases(List) my-list:
  | empty() => print("empty")
  | link(something, _) =>
    print("not empty")
end
```

desugar

Core

```
my-list = list.["link"](2, list.["empty"])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({
    "empty" : fun(): print("empty") end,
    "link" : fun(something, _):
      print("not empty") end
  },
  fun(): raise("cases: no cases matched") end)
end
```

Surface

```
my-list = [2]
cases(List) my-list:
  | empty() => print("empty")
  | link(something, _) =>
    print("not empty")
end
```

desugar

Core

```
my-list = list.["link"](2, list.["empty"])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({
    "empty" : fun(): print("empty") end,
    "link" : fun(something, _):
      print("not empty") end
  },
  fun(): raise("cases: no cases matched") end)
end
```

```
my-list = obj.["link"](2, list.["empty"])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty") end,
"link" : fun(something, _):   print("not empty") end}, fun():
raise("cases: no cases matched") end)
end
```

```
my-list = obj.["link"](2, list.["empty"])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty") end,
"link" : fun(something, _):   print("not empty") end}, fun():
raise("cases: no cases matched") end)
end
```

```
my-list = <func>(2, list.["empty"])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty") end,
"link" : fun(something, _):   print("not empty") end}, fun():
raise("cases: no cases matched") end)
end
```

```
my-list = <func>(2, obj.["empty"])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty") end,
"link" : fun(something, _):   print("not empty") end}, fun():
raise("cases: no cases matched") end)
end
```

```
my-list = <func>(2, obj.["empty"])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty") end,
"link" : fun(something, _):   print("not empty") end}, fun():
raise("cases: no cases matched") end)
end
```

```
my-list = <func>(2, [])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty")
end, "link" : fun(something, _):   print("not empty") end},
fun():   raise("cases: no cases matched") end)
end
```

```
my-list = [2]
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty")
end, "link" : fun(something, _):   print("not empty") end},
fun():   raise("cases: no cases matched") end)
end
```

```
tempMODRIOUJ :: List = [2]
tempMODRIOUJ.["_match"]({"empty" : fun(): print("empty")
end, "link" : fun(something, _): print("not empty") end},
fun(): raise("cases: no cases matched") end)
```

```
[2].["_match"]({"empty" : fun(): print("empty") end,
"link" : fun(something, _): print("not empty") end}, fun():
raise("cases: no cases matched") end)
```

```
[2].["_match"]({"empty" : fun(): print("empty") end,
"link" : fun(something, _): print("not empty") end}, fun():
raise("cases: no cases matched") end)
```

```
<func>({"empty" : fun(): end, "link" : fun(something, _):
print("not empty") end}, fun(): raise("cases: no cases
matched") end)
```

```
<func>({"empty" : fun(): end, "link" : fun(): end}, fun():
raise("cases: no cases matched") end)
```

```
<func>(obj, fun(): raise("cases: no cases matched") end)
```

```
<func>(obj, fun(): end)
```

```
<func>("not empty")
```

```
"not empty"
```

Surface

```
my-list = [2]
cases(List) my-list:
  | empty() => print("empty")
  | link(something, _) =>
    print("not empty")
end
```

↓

```
my-list = [2]
cases(List) my-list:
  | empty() => print("empty")
  | link(something, _) =>
    print("not empty")
end
```

↓

```
cases(List) [2]:
  | empty() => print("empty")
  | link(something, _) =>
    print("not empty")
end
```

↓

```
<func>("not empty")
```

↓

```
"not empty"
```

desugar

Core

```
my-list = list.["link"](2, list.["empty"])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({
    "empty" : fun(): print("empty") end,
    "link" : fun(something, _):
      print("not empty") end
  },
  fun(): raise("cases: no cases matched") end)
end
```

```
my-list = obj.["link"](2, list.["empty"])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty") end,
"link" : fun(something, _):   print("not empty") end}, fun():
raise("cases: no cases matched") end)
end
```

```
my-list = obj.["link"](2, list.["empty"])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty") end,
"link" : fun(something, _):   print("not empty") end}, fun():
raise("cases: no cases matched") end)
end
```

```
my-list = <func>(2, list.["empty"])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty") end,
"link" : fun(something, _):   print("not empty") end}, fun():
raise("cases: no cases matched") end)
end
```

```
my-list = <func>(2, obj.["empty"])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty") end,
"link" : fun(something, _):   print("not empty") end}, fun():
raise("cases: no cases matched") end)
end
```

```
my-list = <func>(2, obj.["empty"])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty") end,
"link" : fun(something, _):   print("not empty") end}, fun():
raise("cases: no cases matched") end)
end
```

```
my-list = <func>(2, [])
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty")
end, "link" : fun(something, _):   print("not empty") end},
fun():   raise("cases: no cases matched") end)
end
```

```
my-list = [2]
block:
  tempMODRIOUJ :: List = my-list
  tempMODRIOUJ.["_match"]({"empty" : fun():   print("empty")
end, "link" : fun(something, _):   print("not empty") end},
fun():   raise("cases: no cases matched") end)
end
```

```
tempMODRIOUJ :: List = [2]
tempMODRIOUJ.["_match"]({"empty" : fun(): print("empty")
end, "link" : fun(something, _): print("not empty") end},
fun(): raise("cases: no cases matched") end)
```

```
[2].["_match"]({"empty" : fun(): print("empty") end,
"link" : fun(something, _): print("not empty") end}, fun():
raise("cases: no cases matched") end)
```

```
[2].["_match"]({"empty" : fun(): print("empty") end,
"link" : fun(something, _): print("not empty") end}, fun():
raise("cases: no cases matched") end)
```

```
<func>({"empty" : fun(): end, "link" : fun(something, _):
print("not empty") end}, fun(): raise("cases: no cases
matched") end)
```

```
<func>({"empty" : fun(): end, "link" : fun(): end}, fun():
raise("cases: no cases matched") end)
```

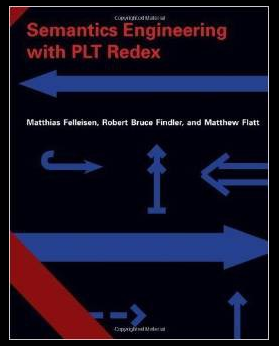
```
<func>(obj, fun(): raise("cases: no cases matched") end)
```

```
<func>(obj, fun(): end)
```

```
<func>("not empty")
```

```
"not empty"
```

1. Redex semantics engineering tool



2. Racket (racket.org)

3. Pyret (pyret.org)

Resugaring

Fork us on GitHub
tinyurl.com/resugar

The *Confection* tool

- Declarative sugar specification
- Language agnostic
- Guarantees Emulation and Abstraction

Current/future work

- Hygiene!
- Better error messages through resugaring

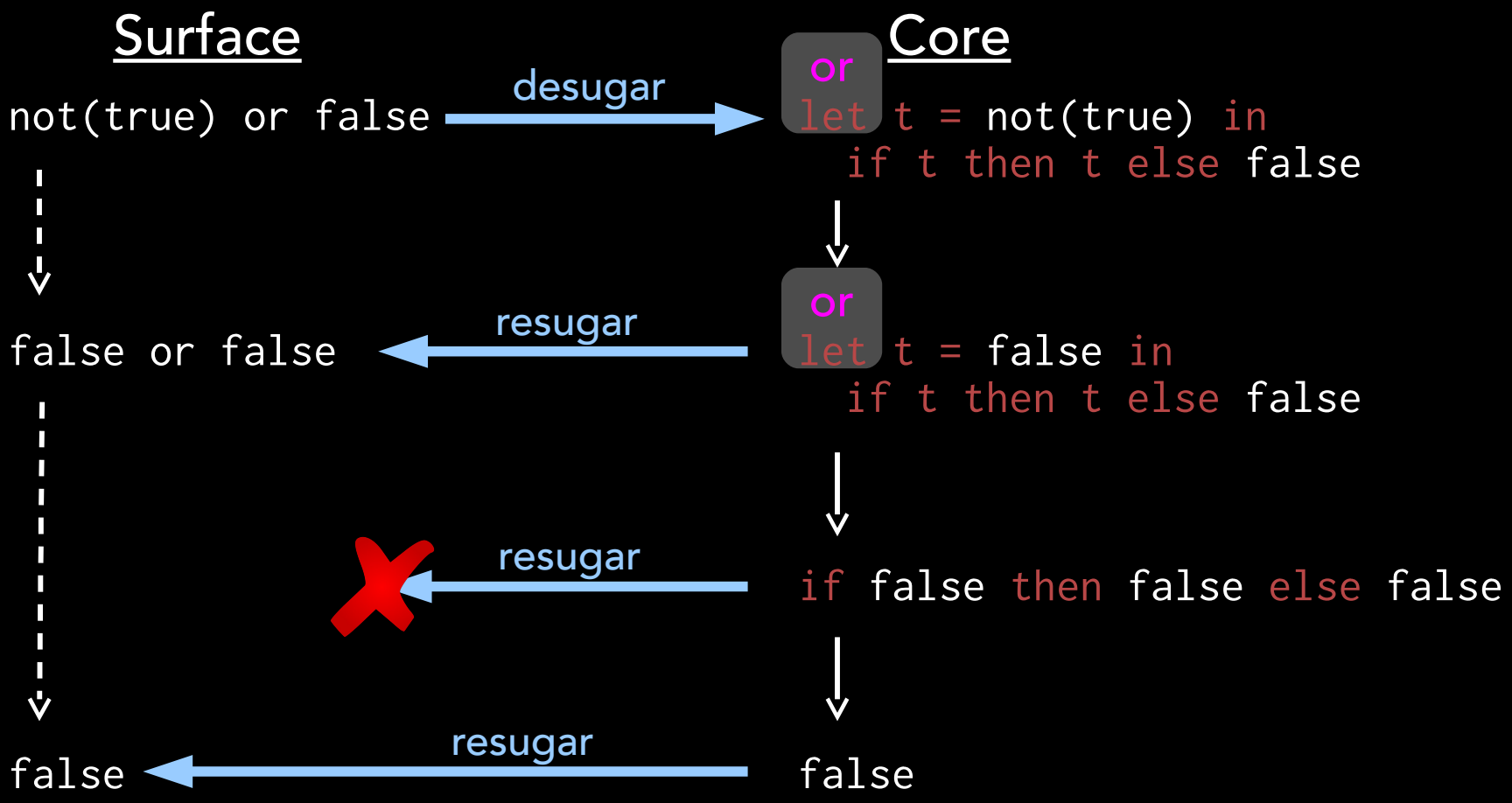
BACKUP SLIDES

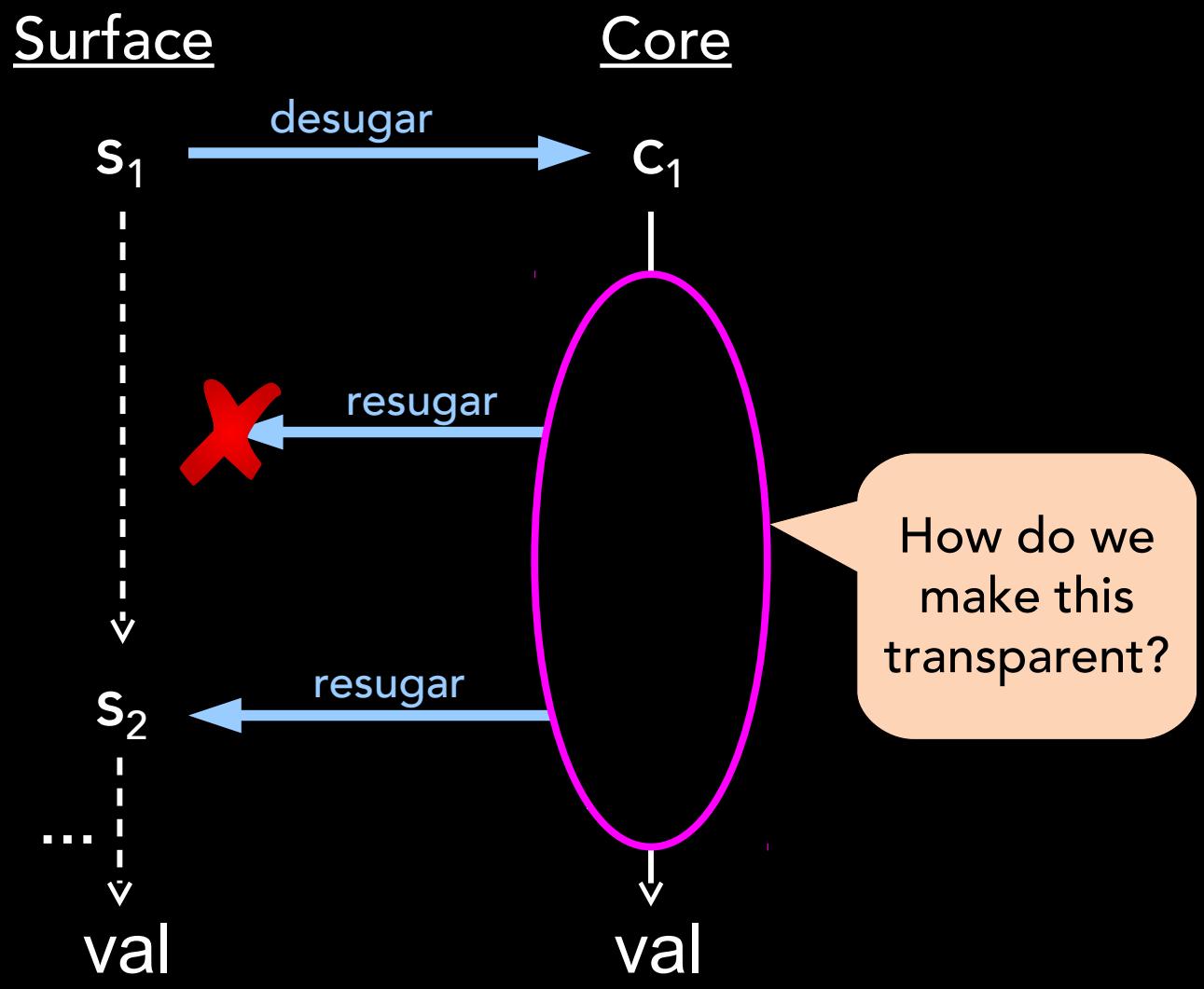
How should this resugar?

```
let t = not(true) in
  if t then t else true
```

Looks like the desugaring of an ‘or’.

But maybe the user wrote it?


Surface
Core
not(true) or false
desugar
or
let t = not(true) in
if t then t else false
false or false
resugar
or
let t = false in
if t then t else false
resugar
if false then false else false
false
resugar
false


Surface
Core
desugar
s_1
c_1
resugar
How do we make this transparent?
resugar
s_2
...
val
val

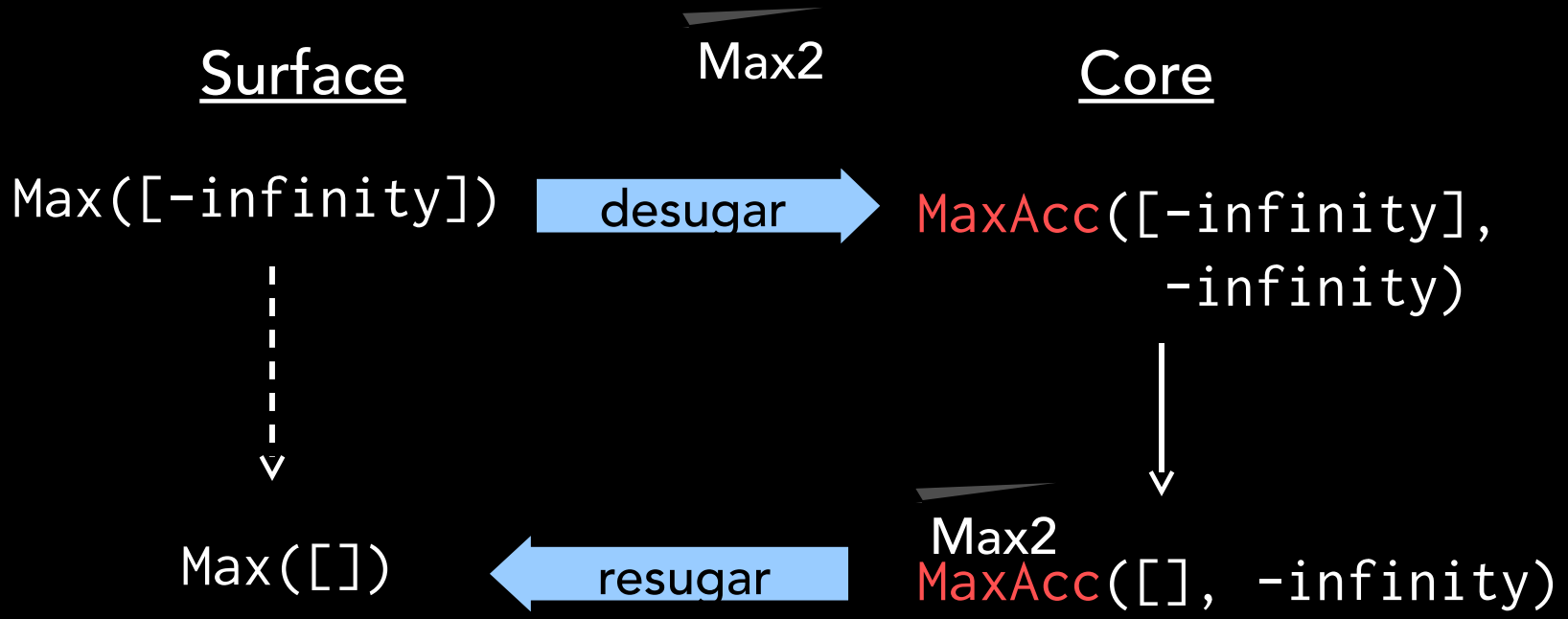
Reconstruct the (core) program as source at each evaluation step (along with its tags).

Either instrument the compiler
or do a pre-compilation step

CPS/ANF makes this easier;
or just statefully keep track of the stack

```
Max([]) -> Raise("empty list");

Max(xs) -> MaxAcc(xs, -infinity);
```



Related Work

- J. Hennessy. Symbolic debugging of optimized code. Transactions on Programming Languages and Systems, 4(3), 1982.
- J. Clements, M. Flatt, and M. Felleisen. Modeling an algebraic stepper. In European Symposium on Programming Languages and Systems, 2001.
- R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In International Conference on Functional Programming, 2012
- A. V. Deursen, P. Klint, and F. Tip. Origin tracking. Journal of Symbolic Computation, 15(5–6), 1993.